Méthodes algorithmiques et programme modulaire pour la résolution des jeux combinatoires

Application au Sprouts, au Cram, et au Dots-and-boxes

Thèse

par

Julien Lemoine et Simon Viennot

présentée pour obtenir le grade de :

Docteur

Spécialité : **Informatique**

sous la direction de Jean-Paul Delahaye

et Tristan Cazenave

soutenue le 8 novembre 2011

Université Lille-I
Laboratoire d'Informatique Fondamentale de Lille
École Doctorale des Sciences Pour l'Ingénieur – Université Lille Nord de France

Table des matières

Chapitre 1

Présentation de la thèse

Notre thèse est dédiée à l'étude des jeux combinatoires. Notre objectif est de déterminer comment gagner à certains de ces jeux, et plus précisément, comment *être sûr* de gagner.

1.1 Contexte

1.1.1 Théorie des jeux combinatoires

La théorie des jeux combinatoires à laquelle nous faisons référence ici est celle décrite principalement par Berlekamp, Conway et Guy dans le livre fondateur *Winning Ways for your mathematical plays*, publié en 1982, et republié en 2001 [6]. On trouvera un historique du développement de ce domaine dans l'article de 2009 par Richard J. Nowakowski [32], dont nous redonnons ci-dessous les principales étapes.

Le point de départ de la théorie vient des jeux impartiaux, où les coups disponibles à partir de toute position sont les mêmes pour les deux joueurs. Le premier jeu impartial considéré est le jeu de Nim, résolu dès 1902 par C. L. Bouton [7]. Sprague et Grundy ont ensuite découvert indépendamment en 1935 [42] et 1939 [22] la classification des jeux impartiaux grâce au jeu de Nim, aujourd'hui référencée sous le nom de *théorème de Sprague-Grundy*. Le jeu de Nim est vraiment remarquable : ses règles sont parmi les plus simples et les plus élégantes qu'il soit possible d'imaginer, mais il n'en joue pas moins un rôle central dans la théorie des jeux impartiaux.

La théorie s'est ensuite grandement enrichie avec l'étude des jeux partisans dans les années 1960. La rencontre de Berlekamp, Conway et Guy aboutira en 1982 à la publication de *Winning Ways*, qui reste encore une référence incontournable sur le sujet. Les concepts décrits dans Winning Ways sont très nombreux, avec en particulier les notions de coup réversible, d'option dominée, de valeur d'un jeu et de somme de jeux.

Le développement théorique depuis la publication de Winning Ways est moins facile à retracer. Richard J. Nowakowski publie régulièrement des sélections d'articles dans une série de livres intitulés *Games of No Chance*, avec de nombreuses études spécifiques de jeux et des généralisations théoriques. Parmi les travaux les plus intéressants par rapport au contenu de cette thèse, on peut citer une avancée théorique sur les jeux impartiaux en version misère par Thane Plambeck en 2005 [33] avec l'introduction du concept de *quotient misère*.

1.1.2 Calculs des jeux combinatoires

Le problème du calcul des stratégies gagnantes des jeux combinatoires est rattaché historiquement à un autre domaine, celui de l'*intelligence artificielle*. Ce domaine a fait l'objet d'un nombre très élevé de recherches et de publications, en particulier à partir des années

90, lorsque la capacité de calcul des ordinateurs a atteint un niveau suffisant pour résoudre des problèmes intéressants.

Les calculs de stratégies gagnantes peuvent être subdivisés en deux branches principales : d'une part, les calculs de stratégie partielle, dont le but est de jouer aussi bien que possible contre un joueur humain, et d'autre part, les calculs de stratégie complète, dont le but est de déterminer la stratégie théorique exacte permettant d'assurer la victoire de l'un des joueurs. Ces deux domaines utilisent certaines méthodes communes (en particulier les algorithmes de recherche au sein de l'arbre de jeu), mais posent des problèmes différents. Dans cette thèse, nous nous intéresserons uniquement aux calculs de stratégie complète.

Calculs partiels

Dans le cas de calculs partiels, un élément essentiel est la rapidité du calcul, puisque celui-ci est réalisé au cours d'une partie réelle avec un joueur humain, mais l'exactitude peut par contre être sacrifiée. Même si le programme contient des erreurs, cela n'a pas de conséquence plus grave que de jouer un mauvais coup. Il est également possible d'éliminer des zones entières de l'arbre de jeu, si des heuristiques permettent de prédire leur résultat avec une bonne probabilité. On pourra consulter l'article de 2000 de Jonathan Schaeffer [35], qui donne un état des lieux à l'aube du nouveau millénaire, et reste pour l'essentiel d'actualité.

Le jeu d'échecs a été l'un des plus étudiés en terme de calculs partiels, permettant à l'ordinateur d'atteindre le niveau des meilleurs humains dès 1987 (ChipTest, ancêtre de Deep Blue), avant de battre de justesse le champion du monde Garry Kasparov en 1997 (Deep Blue, super-ordinateur fabriqué par IBM). L'amélioration constante des programmes d'échecs permet maintenant d'atteindre le même niveau de jeu avec une puissance de calcul bien plus faible (un téléphone portable au lieu d'un super-ordinateur...).

Le niveau des meilleurs humains est désormais atteint pour la plupart des jeux. On peut citer le jeu d'Othello (1997, programme Logistello contre le champion du monde Takeshi Murakami) ou les dames (depuis environ 2005, mais sans match d'exhibition). Le jeu de Go est l'un des jeux les plus résistants aux calculs de stratégie partielle. Des progrès récents, en utilisant des algorithmes probabilistes de type Monte-Carlo, ont cependant permis de se rapprocher du niveau professionnel sur les plateaux de petite taille (9×9 cases, programme Mogo), et du niveau de fort amateur sur les plateaux de taille normale (19×19 cases). Sur cette taille de plateau, les meilleurs programmes (MogoTW, Zen) sont encore environ 5 à 6 pierres plus faibles que les professionnels en 2011.

Calculs complets

Dans le cas de calculs complets, la stratégie gagnante est avant tout un résultat mathématique, qui est soumis au même problème d'exactitude que tout autre résultat mathématique. Une erreur dans le programme est donc susceptible de rendre tout ou partie des résultats faux. Par ailleurs, il est moins facile d'éliminer certaines zones de l'arbre de jeu. Même si un coup semble mauvais, encore faut-il le *prouver*. En échange, les calculs complets n'ont aucune contrainte de temps, et il est possible de réaliser le calcul sur plusieurs années, si nécessaire. Les calculs de stratégie gagnante complète réalisés ces vingt dernières années sont très nombreux. Nous en présentons ci-dessous seulement quelques-uns qui nous semblent représentatifs.

Le connect-four est l'un des premiers jeux grand public dont la stratégie gagnante complète a été calculée, en 1988, par James D. Allen et indépendamment par Victor Allis [2]. Les calculs ont ensuite été régulièrement étendus par John Tromp, qui a calculé une solution pour le plateau de taille 9×6 en 2005. Le jeu du moulin (Nine Men's Morris) a été résolu par Ralph Gasser en 1993 [21]. De nombreux calculs ont été faits sur le Domineering, Dennis Breuker, Jos Uiterwijk et Jaap van den Herik obtenant une solution pour le plateau 8×8

en 2000 [8], puis 9×9. Nathan Bullock est parvenu à atteindre ensuite le plateau 10×10 en 2002 [10].

Le calcul de stratégie gagnante complète le plus long et le plus difficile réalisé jusqu'ici est celui des dames anglaises (jeu de *Checkers*, ou *English draughts* en anglais). Jonathan Schaeffer a commencé le développement du programme Chinook en 1989, atteignant un niveau à peu près égal à celui du champion du monde dès 1990. Les calculs réalisés à ce moment-là étaient seulement des calculs partiels, mais Schaeffer et Lake ont conjecturé dans un article de 1996 [16] qu'une extension pour obtenir une solution complète serait sans doute possible à l'avenir. En 2007, après des calculs étalés au total sur une quinzaine d'années et des centaines d'ordinateurs, Schaeffer est finalement parvenu à obtenir une stratégie complète [36].

1.1.3 Positionnement de la thèse

Le point central de la théorie des jeux combinatoires réside dans la notion de sommes de jeux, et cherche notamment à calculer le résultat d'une somme en fonction du résultat de ses composantes. Ce domaine se rattache plutôt aux mathématiques. Les calculs de jeux combinatoires (que ce soit d'une stratégie gagnante partielle ou complète) sont par contre plutôt focalisés sur les méthodes de parcours de l'arbre de jeu, et se rattachent nettement à l'informatique. Bien que ces deux domaines de recherche partagent un objet d'étude commun (les jeux combinatoires), ils ont évolué jusqu'ici de façon relativement indépendante.

Une des raisons tient en partie à la différence entre les jeux étudiés. La théorie des jeux combinatoires a tendance à s'intéresser à des jeux sur lesquels une riche théorie mathématique est possible (jeu de Nim, Dots-and-boxes, Domineering), alors que les calculs de jeux combinatoires ont été appliqués en premier lieu à des jeux très joués par les humains (échecs, Othello, dames), dans lesquels la notion théorique centrale de somme de jeux n'apparaît pas.

Dans cette thèse, nous avons cherché à calculer les stratégies gagnantes complètes de jeux combinatoires (Sprouts, Cram, Dots-and-boxes), ce qui rattache notre travail en premier lieu au domaine du calcul des jeux combinatoires. Cependant, alors que ces calculs pourraient être réalisés uniquement avec le concept d'issue gagnante ou perdante, nous avons introduit dans les algorithmes des concepts plus complexes, issus de la théorie des jeux combinatoires (en particulier le *nimber*). Cela nous a permis d'accélérer les calculs notablement. De ce point de vue, cette thèse peut être considérée à la frontière entre les deux domaines exposés ci-dessus.

1.2 Historique de la thèse

Premier programme

Nous nous sommes intéressés au jeu de Sprouts pour la première fois lorsque nous étions étudiants en 1999. Le jeu avait un certain succès parmi les étudiants, sous le nom de *taupe du Pérou*, avec des parties géantes à la craie au tableau. Nous avons découvert quelques années plus tard que ce jeu était surtout connu sous le nom de *Sprouts*, ce qui nous a naturellement amené à l'article de 1991 d'Applegate, Jacobson et Sleator, la référence sur le sujet [4].

Nous avons commencé à programmer sérieusement le Sprouts en 2005, une fois nos études finies. L'annonce en 2006 de nouveaux résultats par Josh Purinton [34] nous a motivés à approfondir notre travail et à le mettre en forme. Nous avons ainsi publié sur internet [1] en 2007 nos résultats, et le code source de notre programme, accompagnés d'un article expliquant les éléments essentiels de nos calculs [27].

1. `http://sprouts.tuxfamily.org/`

L'originalité de ce premier travail était théorique d'une part, avec le mélange des concepts de nimber et d'issue dans les calculs, mais aussi pratique, avec la possibilité d'interactions humaines en cours de calcul. Cela nous a permis d'atteindre 32 points de départ, le record précédent de 2006 étant de 14 points seulement.

Début de la thèse

Les résultats obtenus en 2007 ont ensuite attiré l'attention de Jean-Paul Delahaye en 2008, dans le cadre de sa préparation d'un article sur le jeu de Sprouts pour la revue *Pour la Science* [13]. L'échange de mails a débouché sur l'idée d'approfondir le travail déjà effectué. C'est ainsi que nous avons commencé une thèse, à partir de septembre 2008, sous sa direction.

Nous avons commencé par étudier le Sprouts sur les surfaces compactes, une généralisation assez naturelle du jeu sur le plan. Nous avions déjà réfléchi aux aspects théoriques avant de débuter la thèse, et il ne restait donc plus qu'à programmer, ce qui nous a occupés d'août à décembre 2008, avec la publication d'un article sur arXiv [28].

L'intérêt principal de cette généralisation aux surfaces compactes est indirect : en cherchant à formaliser certaines propriétés, cela nous a amenés au concept *d'arbre canonique*, concept qui s'est ensuite répandu petit à petit dans l'ensemble de notre travail.

Le Sprouts en version misère

À partir de janvier 2009 principalement, nous avons étudié l'autre variante naturelle du Sprouts, qui consiste cette fois à inverser la convention de victoire, plutôt qu'à changer les propriétés topologiques du terrain de jeu. Cette variante avait déjà été étudiée par Applegate, Jacobson et Sleator en 1991, puis par Josh Purinton et Roman Khorkov de 2006 à 2008. Les résultats de Purinton et Khorkov, jusqu'à 16 points de départ, se sont d'ailleurs révélés difficiles à battre.

Nous ne disposons pas en version misère d'une méthode aussi efficace que les nimbers de la version normale pour tenir compte des découpages en positions indépendantes. Nous avons cependant réussi à atteindre 20 points de départ, en utilisant de façon originale le concept d'*arbre canonique réduit* pour accélérer les calculs. Nous avons publié un article en août 2009 sur arXiv [29].

L'étude de la version misère du Sprouts nous a également permis de comprendre plus profondément certaines propriétés théoriques des calculs en version normale, à base de nimbers. Nous avons publié un article sur ce sujet sur arXiv en 2010 [31].

Le jeu de Cram

De juin 2009 à août 2010, la majeure partie de notre travail s'est focalisée sur la généralisation des algorithmes à d'autres jeux que le Sprouts. L'idée était de réutiliser les algorithmes et la base de code existants. Le candidat idéal qui s'est présenté (après quelques fausses pistes) est le jeu de Cram. Il s'agit d'un jeu impartial, dont certaines positions sont découpables en positions indépendantes. Ces deux propriétés permettent d'appliquer à ce jeu exactement les mêmes algorithmes que ceux développés initialement pour le Sprouts.

Malheureusement, s'il est simple sur le papier d'utiliser un même algorithme pour deux jeux différents, cela l'est beaucoup moins dans un programme informatique. Notre base de code début 2009 résultait de plusieurs années de travail entièrement consacrées au jeu de Sprouts, et rien n'avait été conçu dans un but plus général. Il a donc fallu un long travail de démêlage du code, pour séparer la partie spécifique au Sprouts de celle qui pouvait être réutilisée.

La suppression des dépendances au Sprouts dans la partie appelée à devenir commune au Cram (les bases de données, les algorithmes récursifs de calculs, le suivi, etc.) s'est révélée l'une des étapes les plus difficiles, à tel point que certaines fonctions centrales du programme

ont finalement été réécrites entièrement. C'est ce processus de généralisation du programme qui nous a pris le plus de temps, et qui a rendu le Cram difficile à programmer. Le jeu de Cram en lui-même — tout du moins une programmation classique sous la forme d'un tableau — est en effet relativement aisé à implémenter, par rapport au jeu de Sprouts.

Les premiers records de Cram, dépassant les résultats précédents de 2009 par Martin Schneider [39], ont finalement été obtenus à partir de mai 2010.

Le Proof-number search

L'idée d'appliquer des algorithmes de parcours de type Proof-number search remonte au début de la thèse, en 2008, lorsqu'elle nous fut suggérée par Tristan Cazenave. Il a cependant fallu attendre septembre 2010 pour que notre programme soit capable de réaliser concrètement ce type de calculs. Cela vient en partie du fait que que nous nous sommes concentrés d'abord sur les sujets présentés précédemment, mais il y a également une raison intrinsèque.

Au contraire d'un algorithme de type depth-first, l'algorithme de parcours Proof-number search demande de maintenir en mémoire une bonne partie des nœuds développés dans l'arbre de recherche. Cela implique des méthodes de programmation totalement différentes de l'algorithme de parcours alpha-bêta, que nous utilisions jusqu'alors. Nous avons commencé à préparer l'implémentation du PN-search dès 2009, lors de la généralisation des algorithmes au jeu de Cram, mais les difficultés techniques de programmation nous ont finalement demandé plus d'un an et demi de travail.

Le PN-search a donné de bons résultats sur le Sprouts, meilleurs que le simple alpha-bêta, et presque aussi bons que l'alpha-bêta avec interactions manuelles. L'ajout d'interactions manuelles au PN-search lui-même a ensuite permis d'améliorer tous nos records de Sprouts à partir de décembre 2010.

Le Dots-and-boxes

Le Dots-and-boxes est le dernier sujet que nous avons abordé dans le cadre de cette thèse, à partir d'avril 2011. Ce jeu est très différent du Sprouts et du Cram étudiés jusqu'ici. Non seulement il n'est pas impartial, mais il fait même intervenir potentiellement la notion de partie nulle. Plusieurs propriétés relient cependant le Dots-and-boxes au reste de la thèse : le jeu est presque impartial, il permet des découpages du plateau en composantes séparées (bien que non indépendantes), et il est possible d'utiliser une représentation informatique très proche de celle du Cram.

La théorie des calculs du Dots-and-boxes est cependant très différente. Plusieurs approches sont envisageables, et nous avons choisi de baser nos calculs sur les notions de *score* et de *contrat*. Combiné avec une théorie détaillée de certaines équivalences de positions, cela nous a permis d'obtenir de premiers résultats dès juin 2011.

Nous confirmons tous les résultats obtenus par David Wilson en 2002 [45], et les complétons avec le score de plusieurs nouvelles positions de départ.

1.3 Méthodes de travail

Travail en commun

Notre travail est particulier, dans la mesure où son intégralité a été menée à deux sur une durée de plusieurs années, ce qui n'est pas si fréquent dans le monde de la recherche[2]. C'est le cas aussi bien pour le développement du programme, que pour les réflexions théoriques, ou même la rédaction des articles et des mémoires.

2. Malgré certains exemples célèbres, comme Appel et Haken, lors de leurs travaux sur le théorème des 4 couleurs.

La séparation des chapitres entre les deux mémoires n'a donc pas été facile, mais elle reflète finalement en partie des préférences différentes : plutôt théoriques et algorithmiques d'un côté, avec par exemple les chapitres sur l'utilisation du nimber, la représentation des positions de Sprouts et son extension aux surfaces compactes ; plutôt orientées vers la programmation pratique de l'autre, avec par exemple les chapitres sur le suivi et l'architecture du programme.

Cette différence d'approche est sans doute l'un des éléments les plus utiles du travail collaboratif. Si l'une des approches bloque sur un problème, l'autre apporte souvent une perspective différente, à même de résoudre ou de contourner les difficultés. L'inconvénient du travail collaboratif est par contre de nécessiter un temps assez important de communication, et parfois de négociation, certaines idées devant nécessairement être privilégiées par rapport à d'autres.

Théorie et pratique

De manière plus générale, il y a une opposition et une complémentarité entre la théorie (algorithmique, combinatoire) et l'implémentation pratique des calculs.

L'approche théorique privilégie par exemple instinctivement la solution parfaite, mais les impératifs de programmation en termes de temps de calcul, espace mémoire, ou temps de programmation, nécessitent souvent des concessions. On peut citer la canonisation des représentations en chaînes des positions du Sprouts (voir section 9.4). La méthode cherchant à déterminer la canonisation exacte est vouée à l'échec car d'une complexité trop grande. À la place, il vaut mieux avoir recours à une *pseudo-canonisation*, imparfaite, mais suffisamment rapide pour permettre un calcul effectif.

Inversement, il est facile de perdre de vue des considérations théoriques essentielles lors du travail de programmation. Notamment, certaines optimisations de code peuvent se révéler dérisoires lorsque l'on découvre un nouvel algorithme nettement plus performant. La célèbre citation de Donald Knuth en 1974 est toujours d'actualité : « L'optimisation prématurée est la source de tous les maux en programmation ».

Le juste équilibre entre la théorie et l'implémentation est difficile à trouver, notamment en terme de timing. Si la programmation est entreprise trop tôt, elle est vouée à n'être qu'un brouillon sans intérêt rapidement balayé par des considérations théoriques. Entreprise trop tard, et la théorie n'avance plus, comme si celle-ci attendait d'avoir sous les yeux un objet à détruire avant de se développer.

Ce n'est pas un hasard si nos premiers résultats sur le Sprouts reposent principalement sur deux idées, relevant chacune d'une approche différente : le nimber, concept théorique qui a permis de diminuer de plusieurs ordres de grandeur la difficulté du calcul, et le zappage, concept expérimental qui a permis de débloquer à la volée des calculs trop longs.

Outils de travail

Le travail en commun nécessite des outils adaptés, en particulier pour deux personnes habitant respectivement au Japon et en France. Les deux éléments essentiels qui nous ont permis de travailler dans de bonnes conditions sont un wiki et un repository.

Le wiki (le programme utilisé est Mediawiki, le moteur de wikipedia) permet la rédaction aisée ainsi que la structuration des idées qui émergent. En comparaison, la communication par mail n'a été que très peu utilisée.

Le repository (Subversion) permet de sauvegarder sur internet des fichiers, avec création d'un historique, ce qui a prouvé son utilité tant dans la conception du programme que la rédaction des articles. Ces deux outils sont classiques lors du développement collaboratif de logiciel sur internet.

Tous les outils que nous avons utilisés, que ce soit pour la communication, le développement du logiciel ou la rédaction des articles, sont des logiciels libres. En accord avec leur

philosophie, et dans un souci de transparence, nous avons décidé non seulement de diffuser le code source de notre programme, mais aussi les bases de données issues de nos calculs.

1.4 Plan de la thèse

Théorie des jeux combinatoires

Nous avons regroupé dans ce premier chapitre l'ensemble des notions générales qui nous semblent communes à tous les autres chapitres, afin d'harmoniser la terminologie employée.

Notre travail se situe à la frontière entre la théorie mathématique des jeux combinatoires, et la théorie informatique du calcul des stratégies gagnantes des jeux, deux domaines qui ont évolué jusqu'ici de façon relativement indépendante, avec une terminologie parfois différente. Un exemple parmi d'autres : la théorie des jeux combinatoires parle de l'ensemble des *options* disponibles pour un joueur, là où dans les calculs de stratégies gagnantes, on parle plus fréquemment de l'ensemble des *coups* disponibles pour un joueur.

Nous avons donc essayé de redonner une définition des principaux concepts classiques, en particulier celui de *jeu combinatoire impartial*, d'*arbre de jeu*, d'*issue* gagnante ou perdante, de *stratégie gagnante*.

Par ailleurs, la théorie des jeux combinatoires est relativement récente, ce qui fait que certaines notions, même élémentaires, ne sont pas encore clairement établies en tant que telles, ou bien ne possèdent pas une définition consensuelle. Nous définissons en particulier les notions d'*arbre solution*, d'*arbre canonique*, et de *jeu découpable*.

Jeux impartiaux en version normale

Ce chapitre présente les algorithmes que nous avons appliqués au jeu de Sprouts et au jeu de Cram en version normale. Le point commun de ces deux jeux est d'être impartiaux et découpables, c'est-à-dire que certaines positions peuvent être découpées en composantes indépendantes. Les algorithmes que nous avons utilisés sont relativement simples, et nous les avons présentés dès 2007, dans notre premier article sur le jeu de Sprouts [27]. Ces algorithmes utilisent le concept théorique de *nimber*, qui découle du *théorème de Sprague-Grundy*, et qui est particulièrement important dans l'étude des jeux impartiaux en version normale.

Ces algorithmes sont une composante essentielle des records obtenus sur le Sprouts et sur le Cram. Ils permettent un gain exponentiel par rapport à des méthodes plus basiques qui n'exploiteraient pas la notion de découpage.

En 2009, nous avons ensuite compris plusieurs propriétés théoriques intéressantes de ces algorithmes. D'une façon informelle, la méthode de calcul avec les nimbers est *inévitable*, dans le sens où la solution d'un calcul sans les nimbers contient la solution d'un calcul avec la théorie des nimbers. La formalisation nous a conduit au concept d'*arbre solution nimber*, qui généralise le concept d'*arbre solution*.

Jeux impartiaux en version misère

Ce chapitre détaille les algorithmes que nous avons utilisés pour effectuer des calculs de jeux impartiaux en version misère, c'est-à-dire lorsque la convention de victoire est inversée. Comme dans la version normale, nous essayons d'exploiter au mieux les découpages de certaines positions en composantes indépendantes.

La théorie de la version misère fait intervenir les concepts de *coup réductible* et d'*arbre canonique réduit*. Ces concepts sont classiques depuis le livre de Conway *On Numbers And Games* [12], mais l'idée de les utiliser pour accélérer certains types de calculs en version misère est nouvelle.

Notre méthode de calcul de l'issue en version misère est constituée de deux étapes. Nous commençons dans une phase préliminaire par calculer les arbres canoniques réduits de nombreuses composantes de petite taille. Puis, dans l'étape principale du calcul, nous remplaçons systématiquement les petites composantes par leur arbre canonique réduit dans les sommes de positions. Cette méthode a été appliquée aussi bien au Sprouts qu'au Cram en version misère.

Suivi des calculs

Nous abordons dans ce chapitre les mécanismes de suivi des calculs et d'interaction manuelle que nous avons programmés. Bien que le suivi des calculs à travers une interface graphique soit une idée en apparence toute simple, elle nous a permis de découvrir de nombreuses propriétés des différents jeux que nous avons étudiés.

Nous avons amélioré petit à petit notre interface pour disposer du maximum d'informations pertinentes sur l'arbre de recherche. Cela nous a permis de découvrir des faiblesses dans les choix de parcours faits par les algorithmes, et nous avons alors donné la possibilité à l'utilisateur humain d'en détecter une partie et de les corriger en temps réel.

Nous avons appliqué cette méthode aussi bien à l'algorithme alpha-bêta, qu'à l'algorithme PN-search, qui nécessitent chacun une méthode de suivi et une méthode d'interaction différentes à cause des différences fondamentales entre ces algorithmes.

Algorithmes de parcours

Nous détaillons dans ce chapitre les algorithmes de parcours utilisés dans notre programme, à savoir l'alpha-bêta et le PN-search. La présentation que nous donnons du PN-search est un peu particulière dans la mesure où elle est adaptée aux jeux impartiaux. Les principales améliorations apportées à ces algorithmes résultent de la possibilité d'intervenir directement sur leur déroulement pendant le calcul.

Par ailleurs, nous livrons quelques réflexions théoriques, que nous n'avons cependant pas pu valider par le calcul, faute de les avoir implémentées. Elles concernent l'application de coefficients aux proof et disproof numbers du PN-search, afin de focaliser le calcul sur certains nœuds de l'arbre de recherche, ainsi que l'adaptation du PN-search aux algorithmes basés sur le nimber (algorithmes présentés dans le chapitre 3).

Vérification

Les algorithmes de vérification ont été développés suite à l'introduction des mécanismes d'interaction en temps réel dans notre programme. Ces mécanismes, difficiles à sécuriser, nous ont amenés a douter de la validité de nos calculs.

La vérification est un calcul qui se mène a posteriori, et qui consiste à utiliser les résultats précédemment obtenus pour guider le nouveau calcul, à travers une boucle dont la programmation est si simple qu'elle ne peut contenir d'erreur. Son utilisation permet de ramener les risques d'erreurs liées au parcours des arbres de recherche à une probabilité infime.

De plus, nous avons développé une technique originale de *colmatage*, qui consiste à reboucher les trous de tables de transpositions incomplètes, dans lesquelles seules quelques positions manquent.

Dans un deuxième temps, nous avons constaté que l'algorithme de vérification permettait d'obtenir des arbres solutions de taille réduite, ce qui était initialement involontaire, mais s'est montré fort utile, tant pour obtenir des preuves compactes que pour améliorer les capacités de calcul de notre programme. Enfin, l'utilisation lors du calcul de vérification d'un parcours *aléatoire* de l'arbre de recherche nous a permis d'engendrer des arbres solutions encore plus petits.

Architecture du programme

L'architecture de notre logiciel a évolué au cours du temps, vers une modularisation croissante. Au départ, le programme n'était destiné qu'à mener des calculs de Sprouts en version normale. Puis, petit à petit, d'autres fonctionnalités sont apparues : la version misère, la généralisation à d'autres jeux...

Nous décrivons dans ce chapitre les principaux éléments constituant le programme, notamment la boucle principale de calcul, les tables de transpositions, la sérialisation des données, et l'interface graphique.

Un aspect essentiel de l'architecture est la possibilité d'ajouter des modules de l'un des trois types suivants : jeux, algorithmes de calcul, et algorithmes de parcours. Si le nouveau module est défini conformément à certaines règles, il devient alors immédiatement fonctionnel dans notre programme, et l'on peut lui appliquer tous les autres outils à notre disposition, comme le suivi, les interactions en temps réel, ou l'affichage des tables de transpositions.

Représentation des positions de Sprouts

Le jeu de Sprouts est difficile à représenter informatiquement à cause de l'aspect topologique du jeu. Nous décrivons dans ce chapitre comment il est possible de décrire les positions de Sprouts par des chaînes de caractères. Notre représentation est un approfondissement de celle de l'article décrivant le premier programme capable de mener des calculs de Sprouts [4]. C'est un des principaux éléments qui nous ont permis d'obtenir des records sur ce jeu.

Une première représentation est obtenue en analysant les positions avec les concepts de *sommet*, de *frontière* et de *région*. Nous simplifions ensuite ces chaînes de caractères, lors d'un processus de *réduction*, qui consiste à éliminer certains éléments (sommets ou régions) inutilisables, et à distinguer certaines catégories de sommets (sommets génériques, distinction des sommets internes à une région seulement ou non).

Nous détaillons ensuite la *canonisation* des chaînes de caractères, dont le but est d'associer une unique chaîne à chaque position, et la *pseudo-canonisation*, qui permet de réaliser une canonisation imparfaite, mais plus rapide. Nous utilisons enfin nos calculs pour discuter de la complexité spatiale du jeu de Sprouts à n points.

Le jeu de Sprouts

Nous présentons dans ce chapitre un historique des calculs de Sprouts que nous avons menés, en version normale et en version misère. Ce chapitre ne traite que la version la plus classique du Sprouts, sur une surface plane.

Nous comparons tout d'abord les calculs en version normale avec et sans la théorie du nimber, puis montrons l'effet de différentes heuristiques de l'ordre d'exploration des options. Nous montrons ensuite comment les calculs ont pu être améliorés avec le suivi, le zappage, puis l'algorithme Proof-number search. Nous récapitulons enfin les meilleurs résultats obtenus jusqu'ici après utilisation des algorithmes de vérification.

Ce chapitre a surtout pour but de montrer comment la recherche de l'obtention de records sur le jeu de Sprouts a servi de moteur à l'élaboration de nombreuses idées, qui sont chacune détaillées dans des chapitres séparés.

Le Sprouts sur les surfaces compactes

Ce chapitre détaille la théorie nécessaire pour étendre les calculs du jeu de Sprouts classique, sur le plan, à des surfaces compactes quelconques. Le jeu sur les surfaces compactes est la généralisation la plus naturelle du jeu classique, ce qui justifie que nous nous soyons intéressés à son étude. Notre programme est le premier à être capable de mener des calculs

sur les surfaces, ce qui n'est pas une mince affaire, tant le code nécessaire pour le simple calcul des options d'une position est compliqué.

Après un rappel des propriétés essentielles des surfaces compactes et de leur classification, nous montrons tout d'abord quels sont les coups possibles lorsque le jeu se déroule sur une surface compacte, que celle-ci soit orientable ou non-orientable.

Nous décrivons ensuite comment modifier la représentation des positions de Sprouts pour tenir compte des spécificités des surfaces, puis comment nous pouvons calculer les différents types de coups possibles à partir de cette représentation.

Nous établissons enfin le *théorème du genre limite*, qui montre que pour un nombre fixé de points de départ, les propriétés essentielles de la position finissent par se stabiliser quand le genre de la surface augmente. Cette propriété est confirmée par les résultats obtenus, que ce soit en version normale, ou en version misère.

Le jeu de Cram

Le jeu de Cram est un jeu impartial découpable, auquel nous avons pu appliquer les mêmes algorithmes de calcul que pour le jeu de Sprouts. Nous décrivons dans ce chapitre les aspects spécifiques au Cram.

Tout d'abord, nous détaillons la représentation des positions sous la forme de tableau ou de chaînes de caractères. Puis nous montrons l'importance de la *canonisation*, qui cherche à identifier autant que possible les positions de Cram équivalentes. Nous tenons compte des symétries, des cases que l'on ne peut plus utiliser, et des découpages en positions indépendantes. Nous présentons une méthode intéressante pour évaluer la qualité de cette canonisation avec les *arbres canoniques*.

Nous détaillons ensuite nos différentes heuristiques d'ordre des options, avant de terminer par un récapitulatif des différents résultats obtenus, en version normale et en version misère. Dans les deux versions du jeu, nous avons pu dépasser les résultats précédents de 2009 par Martin Schneider [39].

Le jeu de Dots-and-boxes

Le jeu de Dots-and-boxes, un jeu partisan, est nettement différent du Sprouts et du Cram, qui eux, sont impartiaux. En particulier, pour décrire le résultat d'une position de Dots-and-boxes, on peut utiliser une notion de *score*, qui est plus précise que l'issue. Nous introduisons ainsi le concept de *contrat*, qui permet d'effectuer des calculs de score comme s'il s'agissait de calculs d'issue.

L'obligation de rejouer lorsqu'un joueur capture un jeton nécessite de distinguer les notions de *coup* et de *tour de jeu*. Nous développons aussi une théorie de la capture des jetons, en coloriant les jetons en blanc ou noir suivant qu'ils sont capturables ou non durant le tour de jeu. Le *théorème des jetons blancs* exprime une propriété importante limitant fortement les possibilités optimales du joueur concernant le nombre de jetons blancs qu'il doit capturer.

Nous montrons ensuite comment détecter certaines équivalences de positions grâce à des déformations, puis comment une utilisation adéquate des tables de transpositions permet d'optimiser le rapport entre le temps de calcul et la quantité de mémoire utilisée.

Les résultats obtenus sont légèrement meilleurs que les précédents records, obtenus par David Wilson [45] par une méthode totalement différente de résolution forte, sans pour autant les dépasser nettement. Une analyse détaillée des résultats obtenus jusqu'ici nous permet enfin d'estimer la difficulté des calculs futurs.

Chapitre 2

Théorie des jeux combinatoires

Dans ce chapitre, nous présentons des notions générales de la théorie des jeux combinatoires, utiles à la compréhension des autres chapitres. La plupart de ces notions sont classiques, mais du fait de la relative jeunesse de cette théorie au regard de l'histoire des mathématiques, certains éléments de vocabulaire (notamment « arbre canonique » et « jeu découpable ») sont propres à notre travail, bien qu'ils traduisent des notions élémentaires.

2.1 Jeux combinatoires

2.1.1 Qu'est-ce qu'un jeu combinatoire ?

Nous nous intéressons dans cette thèse aux *jeux combinatoires*. Deux joueurs s'affrontent, ils jouent alternativement jusqu'à ce qu'il ne soit plus possible de jouer. On détermine alors le vainqueur (ou l'on déclare le match nul) selon une règle qui dépend du jeu considéré.

Les jeux combinatoires sont à *information complète* : pour choisir son coup, chaque joueur dispose de toutes les informations concernant le jeu pour prendre sa décision. Ceci exclut par exemple le jeu de la bataille navale, où le plateau de l'adversaire est caché.

Il n'y a pas d'intervention du hasard : on ne lance pas de dé comme au Yahtzee, on ne tire pas de cartes comme au Poker.

Voici quelques jeux combinatoires parmi les plus célèbres :

* les échecs.
* le jeu de Go.
* les Dames.
* le Tic-tac-toe (souvent appelé « Morpion »).
* le Connect-Four (« Puissance 4 »).

2.1.2 Jeux partisans et impartiaux

Au sein des jeux combinatoires, nous pouvons distinguer deux grandes catégories. Tout d'abord, les *jeux partisans*, où les coups que l'on peut jouer à partir d'une position donnée diffèrent suivant le joueur dont c'est le tour. C'est le cas des échecs, où le premier joueur ne peut déplacer que les pièces blanches, et le second joueur, que les pièces noires. Si au contraire, les deux joueurs peuvent jouer les mêmes coups à partir d'une position donnée, on parle de *jeu impartial*. Le jeu de Nim décrit dans le paragraphe 2.2.1 est l'exemple le plus classique de jeu impartial.

Les jeux impartiaux et les jeux partisans sont de nature fondamentalement différente. Dans les jeux partisans, un joueur peut accumuler de l'avance. Aux Dames, un joueur qui a encore 15 pions et qui est opposé à un joueur qui n'a plus que 5 pions dispose, sauf certains

cas pathologiques, d'une nette avance. Au jeu de Go, l'avance accumulée par le gagnant se matérialise lors du décompte des points en fin de partie. Il est ainsi assez naturel, pour un joueur humain, de développer des heuristiques pour évaluer les positions des jeux partisans.

Dans les jeux impartiaux, par contre, c'est uniquement la parité du nombre de coups qui détermine le gagnant. Plus précisément, dans un jeu impartial en *version normale*, le joueur qui joue le dernier coup gagne. À l'inverse, en *version misère*, le joueur qui joue le dernier coup perd. Le fait que la victoire se joue forcément à peu de choses — un seul coup — rend les jeux impartiaux plus difficiles à appréhender. Généralement, le joueur est incapable de prédire l'issue de la partie, jusqu'au moment où il l'a totalement analysée. Les heuristiques sont plus difficiles à imaginer.

Fait a priori surprenant, malgré leur définition très similaire, les jeux impartiaux en version misère sont généralement beaucoup plus difficiles à étudier que les jeux en version normale. Ce point est développé en particulier dans le chapitre 4.

Les jeux impartiaux ont un intérêt mathématique particulier. Historiquement, c'est un jeu impartial, le jeu de Nim, qui a été le premier jeu combinatoire à disposer d'une résolution exacte et complète. Ce jeu a ensuite débouché sur une classification des jeux impartiaux en version normale (théorème de Sprague-Grundy), classification qui a été historiquement le point de départ de l'étude des jeux combinatoires, qu'ils soient impartiaux ou partisans.

La présentation usuelle, mise en œuvre tant dans [6] que [38], consiste à définir les jeux impartiaux comme des cas particuliers de jeux partisans (le cas où, quelle que soit la position, les coups jouables par les deux joueurs sont identiques). Puisque nous étudions surtout des jeux impartiaux dans le cadre de cette thèse, nous avons au contraire choisi de présenter dans ce premier chapitre des définitions spécifiques aux jeux impartiaux.

2.2 Jeux étudiés dans cette thèse

Nous présentons ici les jeux qui ont été plus particulièrement étudiés dans le cadre de cette thèse. Si le jeu de Nim est depuis longtemps déjà complètement résolu, il apparaît néanmoins dans ce document du fait de son intérêt sur le plan théorique. Les autres jeux ont tous fait l'objet d'une étude particulière de notre part, débouchant sur des résultats nouveaux.

2.2.1 Jeu de Nim

Le jeu de Nim se joue avec des colonnes d'objets, par exemple des allumettes [1]. Un coup consiste à enlever un certain nombre d'allumettes dans une seule colonne. Ainsi, les mêmes coups sont jouables quel que soit le joueur dont c'est le tour, et le jeu de Nim est impartial. Lorsque le jeu se joue en version normale, le joueur qui enlève la dernière allumette gagne (car l'autre joueur ne peut alors plus jouer).

On notera $\square_n$ la colonne de Nim à n allumettes et l'on utilisera l'opérateur $+$ pour indiquer les différentes colonnes. Ainsi, la position $\square_7 + \square_5 + \square_4 + \square_2$ est la position composée de 4 colonnes contenant respectivement 7, 5, 4 et 2 allumettes. Le joueur dont c'est le tour pourrait par exemple choisir d'enlever 3 allumettes dans la deuxième colonne, ce qui conduirait à la position $\square_7 + \square_2 + \square_4 + \square_2$. Ou alors, il pourrait enlever toutes les allumettes de la troisième colonne, et la nouvelle position serait $\square_7 + \square_5 + \square_0 + \square_2$.

La résolution du jeu de Nim a été décrite pour la première fois par Bouton, en 1902 [7]. Ce jeu tient un rôle fondamental dans la théorie des jeux impartiaux. Non seulement il s'agit du premier jeu impartial complètement résolu, mais il permet également de classifier les jeux impartiaux en version normale.

1. C'est ainsi que le jeu, en version misère, apparaît dans le film d'Alain Resnais *L'année dernière à Marienbad* (1961).

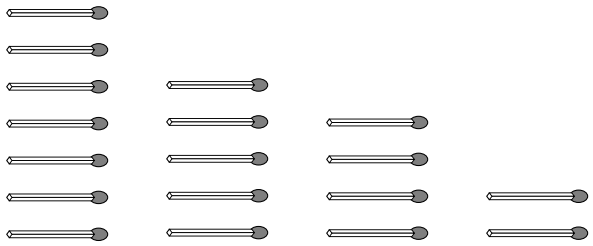


FIGURE 2.1 – Position $7 + 5 + 4 + 2$ du jeu de Nim.

2.2.2 Jeu de Sprouts

Le jeu de *Sprouts* est un jeu impartial, créé récemment (1967), et qui jouit d'une certaine notoriété dans le milieu scientifique. Le premier article présentant ce jeu est dû à Martin Gardner [19]. Outre cet article, on peut également trouver une présentation de ce jeu dans *Winning Ways* [6]. Le jeu se joue sur une feuille de papier, il débute avec un certain nombre de points tracés sur la feuille. À chaque coup, le joueur dont c'est le tour doit relier un point à un autre (éventuellement à lui-même) avec une ligne, puis rajouter un point sur cette ligne. Deux conditions doivent être respectées : les lignes ne doivent pas se croiser, et d'un même point ne peuvent partir plus de 3 lignes.

FIGURE 2.2 – Exemple de partie de Sprouts, en commençant avec 2 points (le deuxième joueur gagne).

Le jeu de Sprouts est le point de départ de cette thèse. C'est en cherchant à calculer les stratégies gagnantes du Sprouts que nous nous sommes intéressés à la théorie plus générale des jeux impartiaux, en version normale, puis en version misère. Le Sprouts occupe donc une place importante dans cette thèse. Les chapitres sur la théorie des jeux impartiaux en version normale (chapitre 3) et en version misère (chapitre 4) s'appliquent directement au Sprouts. Les chapitres 9, 10 et 11 décrivent respectivement la méthode de représentation des positions de Sprouts, les calculs réalisés sur la version classique du jeu de Sprouts, lorsqu'il se déroule sur le plan, et l'étude d'une généralisation du jeu lorsqu'il se déroule sur d'autres surfaces que le plan.

2.2.3 Jeu de Cram

Le jeu de Cram est un jeu impartial qui se joue sur un quadrillage avec des règles extrêmement simples : les joueurs posent alternativement un domino sur deux cases vides adjacentes, jusqu'à ce que l'un d'entre eux ne puisse plus jouer. On trouve par exemple une présentation de ce jeu dans le volume 3 de *Winning Ways* [6].

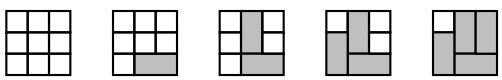


FIGURE 2.3 – Exemple de partie de Cram sur une grille 3×3 (le deuxième joueur gagne).

On verra dans la section 2.6 que le Cram partage avec le Sprouts une propriété essentielle de découpage en positions indépendantes, ce qui nous a motivé à étudier ce jeu. Le chapitre 12 est consacré au jeu de Cram.

2.2.4 Dots-and-boxes

Le Dots-and-boxes est un jeu partisan qui se joue sur un quadrillage. À chaque coup, un joueur ajoute une arête. S'il complète un carré, il le marque de son initiale, puis joue un autre coup. Il passe son tour dès qu'il ne peut plus compléter de carré. À la fin, le joueur qui a remporté le plus de carrés gagne.

La première publication relative au Dots-and-boxes, due à Édouard Lucas, date de 1882. Par la suite, ce jeu a été étudié en profondeur, en particulier par Elwyn Berlekamp [5], un des trois co-rédacteurs de *Winning Ways* [6].

Le Dots-and-boxes est presque un jeu impartial, puisque étant donnée une position, les deux joueurs peuvent jouer les mêmes coups ; la seule différence avec un jeu impartial est que l'initiale marquée sur chaque carré dépend du joueur qui a joué le coup.

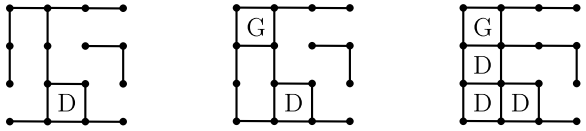


FIGURE 2.4 – Deux coups dans une partie de Dots-and-boxes.

Autre différence notable avec le Sprouts ou le Cram, le Dots-and-boxes peut faire intervenir la notion de partie nulle si les deux joueurs obtiennent un même nombre de carrés à la fin de la partie, ce qui pourrait modifier en profondeur certains algorithmes. Nous verrons cependant dans le chapitre 13 qu'il est en fait possible de mener des calculs sur le Dots-and-boxes sans avoir besoin d'introduire la notion de partie nulle.

2.3 Notion de jeu

2.3.1 Définition formelle des jeux

Les jeux impartiaux peuvent être définis récursivement, comme par exemple dans Schleicher et Stoll [38], définition 7.2 p. 27.

Définition 1.

- *si $\mathscr{G}$ est un ensemble de jeux impartiaux, alors $\mathscr{G}$ est un jeu impartial.*
- *Il n'existe pas de suite infinie $\mathscr{G}^0$, $\mathscr{G}^1$, $\mathscr{G}^2$, ... où $\mathscr{G}^{i+1} \in \mathscr{G}^i$ pour tout $i \in \mathbb{N}$ (condition de terminaison).*

En particulier, l'ensemble vide $\emptyset$ est un jeu impartial, appelé *jeu terminal*. Les éléments de l'ensemble $\mathscr{G}$ sont appelés les *options* de $\mathscr{G}$, et seront notés $\mathscr{G} = \{\mathscr{G}_1, \mathscr{G}_2, \mathscr{G}_3, ...\}$. Aucune restriction n'est faite a priori sur l'ensemble des options, qui peut donc être infini, dénombrable ou non.

Contrairement à l'usage habituel dans la théorie des jeux combinatoires, nous autoriserons la même option à apparaître plusieurs fois dans la définition des jeux impartiaux. L'ensemble des options ne sera donc plus un ensemble au sens strict, mais plutôt un multi-ensemble, sans que cela ne pose de problème particulier. Par exemple, on s'autorisera à considérer le jeu $\mathscr{G} = \{\mathscr{G}_1, \mathscr{G}_2\}$, même si $\mathscr{G}_1 = \mathscr{G}_2$. L'explication de cette convention est donnée à la section 2.5.

2.3.2 Déroulement du jeu

Deux joueurs peuvent jouer à un jeu $\mathscr{G}$ de la façon suivante :

- Ils choisissent qui joue en premier.

* La position courante est $\mathscr{G}$ en début de jeu.
* Le joueur dont c'est le tour choisit une option de la position courante, et cette option devient la position courante.
* Les joueurs jouent à tour de rôle jusqu'à ce que l'un d'entre eux ne puisse plus jouer.

Les *positions* d'un jeu $\mathscr{G}$ donné sont $\mathscr{G}$ (la *position de départ*), et toutes les positions des différentes options de $\mathscr{G}$. C'est-à-dire que les positions d'un jeu représentent tous les états atteignables au cours d'une partie quelconque, et la *position courante* représente l'état du jeu à un moment de la partie.

Les options correspondent donc aux coups disponibles pour le joueur dont c'est le tour. Une option est une position, donc un état, tandis qu'un *coup* est la transition entre deux états. Une *partie* est une suite de positions, dont chacune est une option de la précédente. La condition de terminaison de la définition des jeux impartiaux assure que la partie se termine en un nombre fini de coups.

Enfin, pour définir quel joueur est le gagnant, nous avons vu que deux conventions sont possibles : soit le joueur qui ne peut plus jouer est le perdant (version normale), soit celui-ci est le gagnant (version misère).

2.3.3 Jeux courts

On appelle *jeu court* (« *short game* », défini dans *On Number And Games* [12] p. 97) un jeu dont l'ensemble des positions est fini. Dans l'ensemble de cette thèse, nous nous restreindrons aux jeux courts.

Certains résultats ne nécessitent pas cette restriction, mais pour des raisons évidentes de terminaison, les algorithmes de calcul ne sont en général applicables qu'aux jeux courts : il paraît difficile de stocker ou d'étudier une infinité de positions avec un ordinateur. Heureusement, en pratique, les gens raisonnables jouent surtout à des jeux courts, et les algorithmes décrits dans cette thèse s'appliquent à de nombreux jeux classiques. En particulier, le Cram ou le Dots-and-boxes sont des jeux courts. Par contre, le jeu de Nim avec une infinité d'allumettes ([12] p. 124) n'est pas un jeu court.

Le jeu de Sprouts est un cas particulier. A priori, il ne s'agit pas stricto sensu d'un jeu court — étant donnée une position avec deux points, il existe une infinité de façons de tracer une ligne entre ces deux points. Cependant, si l'on identifie les positions identiques à déformation près, qui conduisent exactement aux mêmes parties, le Sprouts redevient un jeu court, c'est pourquoi il est possible de l'étudier informatiquement.

Dans toute la suite de ce chapitre, nous utiliserons simplement le terme de *jeu* pour désigner un *jeu impartial court*.

2.3.4 Induction de Conway

La définition des jeux impartiaux étant récursive, la plupart des preuves les concernant sont des preuves par induction. Pour harmoniser la rédaction de ces preuves, nous utiliserons *l'induction de Conway* telle que présentée par Schleicher et Stoll [38], théorème 2.3 p. 3. Adaptée au cas particulier des jeux impartiaux, celle-ci peut s'énoncer :

Théorème 1. *Soit P une proposition définie sur les jeux impartiaux qui est vraie pour un jeu $\mathscr{G} = \{\mathscr{G}_1, \mathscr{G}_2, \mathscr{G}_3, ...\}$ dès que P est vraie pour tout $\mathscr{G}_i$. Alors, P est vraie pour tout jeu $\mathscr{G}$.*

Démonstration. Supposons qu'il existe un jeu $\mathscr{G}^0$ pour laquelle P est fausse. Alors, il existe une option $\mathscr{G}^1$ de $\mathscr{G}^0$ telle que P est fausse pour $\mathscr{G}^1$. En réappliquant cet argument, on peut construire une suite de jeux $\mathscr{G}^0$, $\mathscr{G}^1$, $\mathscr{G}^2$, ... dont chacun est une option du précédent. Cette suite infinie est en contradiction avec la condition de terminaison de la définition des jeux impartiaux. □

Pour prouver qu'une certaine proposition P est vraie pour l'ensemble des jeux, il sera donc suffisant de montrer qu'elle est vraie pour un jeu $\mathscr{G}$ dès lors qu'elle est vraie pour toutes les options de $\mathscr{G}$. En particulier, P doit être vraie pour le jeu terminal {}, car ce jeu n'a pas d'option, donc toute proposition est vraie pour l'ensemble de ses options.

Dans les démonstrations utilisant l'induction de Conway, nous supposerons que P est vraie pour toutes les options de $\mathscr{G}$ (ce que nous appellerons *l'hypothèse d'induction*), et nous montrerons simplement que P est vraie pour $\mathscr{G}$.

2.4 Stratégies gagnantes

2.4.1 Issue d'une position

On définit récursivement l'*issue* d'une position (*outcome* en anglais) comme *perdante* (ou l'on dira simplement que la position est perdante) si aucune de ses options n'est d'issue perdante. Une position qui n'est pas d'issue perdante sera dite d'*issue gagnante*, ou simplement *gagnante*. Toute position d'un jeu donné est donc soit perdante, soit gagnante.

L'ensemble vide est d'issue perdante en version normale, et d'issue gagnante en version misère.

Le théorème suivant justifie les termes de « position perdante » et de « position gagnante ».

Théorème 2. *À partir d'une position gagnante, le joueur dont c'est le tour dispose d'une stratégie lui assurant la victoire, et réciproquement, à partir d'une position perdante, quel que soit le coup choisi par le joueur dont c'est le tour, c'est son adversaire qui dispose d'une stratégie assurant la victoire.*

Démonstration. Par induction de Conway :

* Par définition, une position gagnante possède au moins une option perdante. La stratégie assurant la victoire consiste alors simplement à choisir cette option perdante, puisque cela place l'adversaire dans une situation où, par hypothèse d'induction, il ne possède aucun coup lui assurant la victoire.
* Inversement, étant donné une position perdante, toutes les options disponibles sont des positions gagnantes. Par hypothèse d'induction, l'adversaire possède une stratégie gagnante pour chacune de ces positions. Quelle que soit l'option choisie par le joueur dont c'est le tour, il ne peut donc pas éviter la victoire de son adversaire.

□

2.4.2 Arbre de jeu

Définition 2. *L'*arbre de jeu *d'un jeu $\mathscr{G}$ est l'arbre dont les nœuds sont les positions de $\mathscr{G}$, et où deux positions $\mathscr{P}_1$ et $\mathscr{P}_2$ sont reliées par une arête si $\mathscr{P}_2$ est une option de $\mathscr{P}_1$.*

La figure 2.5 présente en guise d'exemple l'arbre de jeu d'une position de Cram, obtenu en identifiant les positions égales à symétrie(s) près, et en supprimant les cases isolées.

L'arbre de jeu constitue une représentation graphique de l'ensemble des positions atteignables à partir de la position de départ, la *racine* de l'arbre. Nous avons établi une correspondance entre les termes de théorie des jeux combinatoires et le vocabulaire de théorie des graphes associé aux arbres dans la table 2.1.

Il est possible de déterminer récursivement l'issue d'une position à partir de son arbre de jeu : en version normale, les *feuilles* sont perdantes, puis, étant donné un nœud interne de l'arbre, si ce nœud a une option perdante, alors il est gagnant, sinon, il est perdant.

Sur la figure 2.5, nous avons indiqué l'issue des positions rencontrées si l'on joue en version normale. Nous avons utilisé les lettres W et L pour les positions gagnantes (Win en anglais) et perdantes (Loss en anglais).

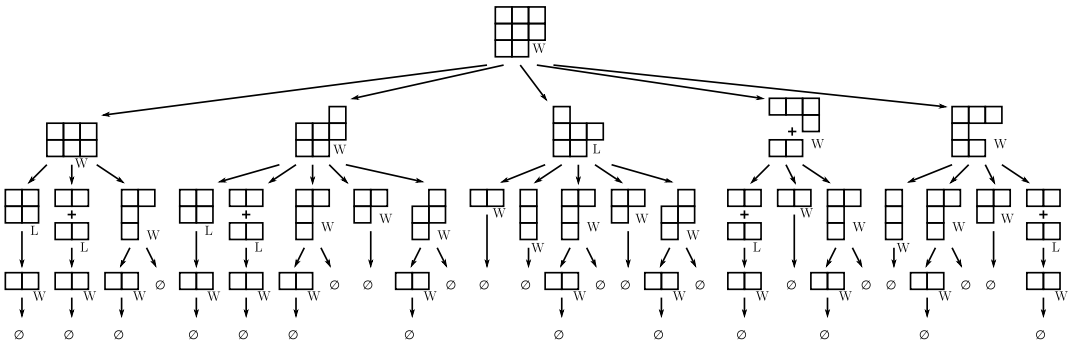


FIGURE 2.5 – Arbre de jeu d'une position de Cram.

Jeu	Arbre de jeu
Position	Sommet ou nœud
Option	Fils
Coup	Arête
Position de départ	Racine
Position terminale	Sommet terminal ou feuille

TABLE 2.1 – Termes de théorie des jeux combinatoires et de théorie des graphes.

2.4.3 Arbre solution

La définition de l'issue d'une position donnée dans le paragraphe 2.4.1 montre qu'il suffit de trouver une unique option perdante pour démontrer qu'un nœud est gagnant. Cela implique qu'il est en fait possible de déterminer l'issue de la racine d'un arbre de jeu sans connaître les issues de tous ses descendants.

Sur la figure 2.6, nous n'avons gardé qu'un ensemble de nœuds de l'arbre de la figure 2.5 qui suffit à démontrer que la racine est perdante. Il y a trois nœuds gagnants pour lesquels nous n'avons pas eu besoin de calculer toutes les options.

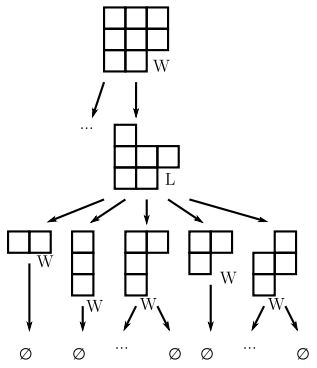


FIGURE 2.6 – Arbre solution d'une position de Cram.

Un tel ensemble de nœuds sera appelé *arbre solution* de la racine. Il fournit de facto une stratégie gagnante pour le joueur qui est en position de force.

Les arbres solutions sont définis formellement par induction :

Définition 3. $*$ *$\mathscr{S} = \varnothing$ est un arbre solution de $\mathscr{G} = \varnothing$.*

* *Si $\mathcal{G}_1$ est une option perdante de $\mathcal{G}$ et $\mathcal{S}_1$ un arbre solution de $\mathcal{G}_1$, alors $\mathcal{S} = \{\mathcal{S}_1\}$ est un arbre solution de $\mathcal{G}$.*
* *Si toutes les options $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, ...$ d'un jeu $\mathcal{G}$ sont gagnantes, et si pour tout i, $\mathcal{S}_i$ est un arbre solution de l'option $\mathcal{G}_i$, alors $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3...\}$ est un arbre solution de $\mathcal{G}$.*

Cette définition formelle consiste simplement à élaguer l'arbre de jeu de ses positions inutiles en ne conservant qu'une seule option perdante chaque fois que c'est possible. On remarquera que si une option gagnante possède plusieurs options perdantes, on a le choix de l'option perdante retenue dans l'arbre solution. L'arbre solution n'est donc pas unique.

Il est immédiat par induction qu'un arbre solution permet au joueur en situation de force de jouer une stratégie gagnante, au sens du théorème 2.

Les arbres solutions sont beaucoup plus petits que les arbres de jeu auxquels ils sont associés. Les figures 2.5 et 2.6 montrent qu'un arbre de jeu à 66 nœuds peut avoir un arbre solution à 12 nœuds, mais l'écart peut être bien plus significatif. Le jeu de Sprouts, spectaculaire de par le déséquilibre de ses arbres de jeu (certaines parties de ces arbres sont bien plus complexes que d'autres), permet d'obtenir des arbres solutions limités à quelques milliers de nœuds, pour des arbres de jeu ayant plus de 10^{20} nœuds.

Remarquons également que les arbres solutions comportent généralement beaucoup moins de positions perdantes que de positions gagnantes. Ceci s'explique facilement, étant donnée la dissymétrie entre les positions perdantes, pour lesquelles il est nécessaire de montrer que toutes les options sont gagnantes, et les positions gagnantes, pour lesquelles il suffit de trouver une option perdante. Cette remarque s'avèrera utile pour économiser de la mémoire, comme nous le montrerons au paragraphe 2.7.5.

Les arbres solutions sont au centre du chapitre 7 sur la vérification des calculs.

2.4.4 Preuves par méthode

Un arbre solution permet donc de disposer d'une stratégie gagnante tout en stockant moins d'information que l'arbre de jeu complet. Il est parfois possible de faire encore mieux, quand la stratégie gagnante peut être décrite par un simple algorithme. On parle alors de *preuve par méthode* (« solve the problem by knowledge » [9]), par opposition aux *preuves par recherche* (« by search ») qui cherchent à déterminer des arbres solutions.

Une preuve par méthode est en fait un moyen de compresser l'information d'un arbre solution particulier, car à partir de cette preuve, on peut fabriquer un arbre solution.

Un exemple simple de preuve par méthode est la *stratégie de symétrie*. Étant donné un jeu impartial en version normale, si une position est composée de deux composantes indépendantes et identiques, alors le deuxième joueur peut gagner en copiant systématiquement les coups de son adversaire.

En guise d'exemple, expliquons comment le premier joueur peut gagner la position de Cram de taille 2×7. Il commence par jouer un coup qui scinde la position de départ en 2 composantes identiques, puis il applique la *stratégie de symétrie* en jouant le symétrique du coup joué par son adversaire. La figure 2.7 présente le début du développement de l'arbre solution engendré par cette stratégie.

La résolution du jeu de Nim par Bouton [7] que nous expliciterons dans le chapitre 3 est un autre exemple de preuve par méthode. En général, les résolutions de jeux obtenues par des moyens informatiques sont des combinaisons de preuves par méthode et par recherche : des preuves par méthode permettent de simplifier les arbres de jeu, de sorte que les preuves par recherche s'effectuent plus rapidement.

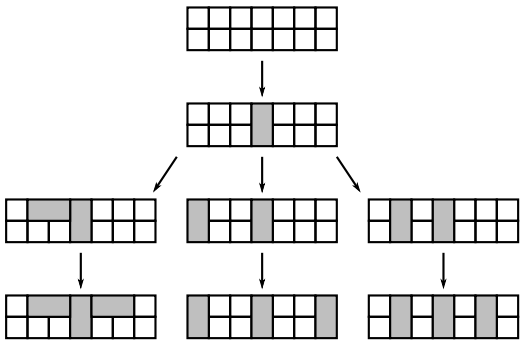


FIGURE 2.7 – Stratégie de symétrie.

2.5 Arbres canoniques

2.5.1 Canonisation

Lorsque l'on dispose d'une relation d'équivalence sur l'ensemble des positions, cela permet de ranger ces positions par classes d'équivalence. Parmi toutes les positions d'une même classe d'équivalence, nous choisissons une unique position, que nous appelons le *représentant canonique* de cette classe d'équivalence.

Cette technique présente un intérêt dès lors que les positions appartenant à une même classe d'équivalence conduisent à jouer des parties similaires. Durant l'exécution du programme, on remplace alors chaque position par le représentant canonique de sa classe d'équivalence, ce qui permet de réduire le nombre de nœuds de l'arbre de jeu, et donc de faciliter son étude. Ce procédé est appelé *canonisation*, et l'on dit que la position qui a été remplacée a été *canonisée*.

Des considérations de symétrie ou de déformation [2] permettent généralement d'établir de telles relations d'équivalence. Par exemple, les positions de Cram de la figure 2.8 sont équivalentes.

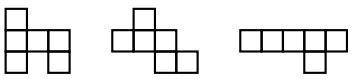


FIGURE 2.8 – Positions de Cram équivalentes.

Parfois, le choix du représentant canonique est naturel, et parfois, aucun élément de la classe ne se détache. Dans ce cas, nous choisissons généralement comme représentant canonique le plus petit pour un ordre arbitraire, le plus simple à calculer possible, comme l'ordre lexicographique pour les chaînes de caractères.

Dans l'étude de la plupart des jeux, le travail sur la canonisation est crucial pour que l'étude informatique du jeu soit performante. En particulier, une bonne partie du chapitre 9 est consacrée à la canonisation des positions du Sprouts.

2.5.2 Arbres canoniques

Lorsque deux branches d'un arbre de jeu sont parfaitement identiques, cela signifie que le joueur peut effectuer un choix parmi deux coups qui mènent exactement à la même situation. Effectuer l'un ou l'autre choix n'influence pas le déroulement de la partie, et les branches

2. Ce sont les raisons les plus fréquemment rencontrées, mais cette liste n'est pas exhaustive.

redondantes d'un arbre de jeu sont donc inutiles. On appelle *arbre canonique* l'arbre de jeu dans lequel on a éliminé toutes les branches redondantes.

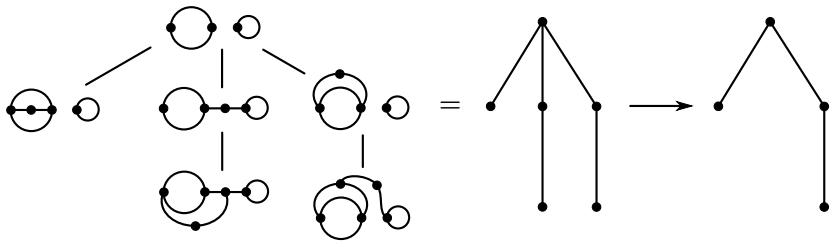


FIGURE 2.9 – Arbre de jeu d'une position de Sprouts, et arbre canonique associé.

La figure 2.9 présente à gauche l'arbre de jeu d'une position de Sprouts. On remarque qu'à partir de la position de départ, les deux coups de droite conduisent à deux parties qui se déroulent de la même façon. On fusionne donc ces deux branches quand on représente l'arbre canonique à droite.

Définition 4. *Pour calculer récursivement l'arbre canonique associé à un arbre de jeu,*
- ∗ *on calcule l'arbre canonique de chacun des fils de la racine.*
- ∗ *on supprime les doublons parmi les fils canonisés.*

La figure 2.10 montre la canonisation de l'arbre de jeu d'une position de Sprouts, celle qui s'obtient à partir de la position de départ à 2 points en reliant un point à lui-même (la position de gauche sur la figure 2.11).

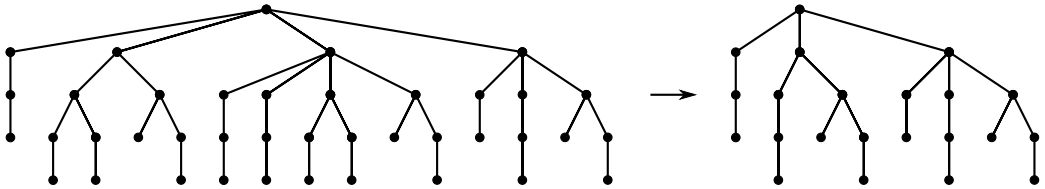


FIGURE 2.10 – Canonisation d'un arbre de jeu.

Cette notion d'arbre canonique permet de conserver la quantité d'information nécessaire et suffisante pour décrire une position : si deux positions ont le même arbre canonique, alors toute partie jouée sur l'arbre de jeu non canonique est équivalente à une certaine partie jouée sur l'arbre canonique et inversement.

L'objectif d'une canonisation, telle qu'expliquée au paragraphe précédent, est de simplifier l'arbre de jeu au maximum pour le rapprocher de l'arbre canonique. Plus l'arbre de jeu obtenu après canonisation est petit et proche de l'arbre canonique, et plus la canonisation est performante.

L'implémentation des arbres canoniques sera détaillée dans le chapitre 4 sur la théorie des jeux impartiaux en version misère. On verra que ce concept peut être amélioré pour faciliter certains calculs en version misère.

2.5.3 Lien avec la définition formelle des jeux

La terme d'*arbre canonique* n'est pas un terme standard de la théorie des jeux combinatoires. Cependant, il désigne une notion tout à fait naturelle, puisqu'il correspond à la notion de *jeu* au sens de Conway (dans *Winning Ways* [6] ou *ONAG* [12]) : une même option ne peut apparaître qu'une et une seule fois.

Cette notion de jeu ne modélise pas parfaitement les jeux réels. En effet, deux positions peuvent sembler très différentes d'après les règles du jeu, et être pourtant *équivalentes*, c'est-à-dire correspondre au même jeu (et donc avoir le même arbre canonique). La figure 2.11 montre ainsi deux positions de Sprouts, qui bien que n'ayant a priori rien à voir l'une avec l'autre, sont équivalentes. Elles ont le même arbre canonique, celui de la figure 2.10.

FIGURE 2.11 – Deux positions de Sprouts avec le même arbre canonique.

Pour modéliser au mieux cet aspect des jeux réels, il nous semble donc naturel qu'un jeu soit défini formellement comme un ensemble qui peut contenir plusieurs options équivalentes (définition 1). C'est-à-dire que si un jeu défini au sens de Conway correspond à la notion d'arbre canonique, un jeu défini comme dans la définition 1 correspond plutôt à la notion d'arbre de jeu.

2.6 Jeux découpables

2.6.1 Somme de jeux

Définition 5. *Soient deux jeux impartiaux* $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, ...\}$ *et* $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3, ...\}$. *On définit récursivement le jeu somme* $\mathcal{G}+\mathcal{H}$ *par* $\mathcal{G}+\mathcal{H} = \{\mathcal{G}_1+\mathcal{H}, \mathcal{G}_2+\mathcal{H}, \mathcal{G}_3+\mathcal{H}, ..., \mathcal{G}+\mathcal{H}_1, \mathcal{G}+\mathcal{H}_2, \mathcal{G}+\mathcal{H}_3, ...\}$.

$\mathcal{G}$ *et* $\mathcal{H}$ *seront appelées les* composantes *de la somme* $\mathcal{G}+\mathcal{H}$.

Dans le cas de l'ensemble vide, qui ne possède aucune option, on a donc $\mathcal{G}+\emptyset = \mathcal{G}$ et $\emptyset+\mathcal{H} = \mathcal{H}$.

Le jeu somme $\mathcal{G}+\mathcal{H}$ se comporte en fait comme si les deux jeux $\mathcal{G}$ et $\mathcal{H}$ étaient placés côte à côte, et le joueur dont c'est le tour peut jouer un coup soit dans $\mathcal{G}$, soit dans $\mathcal{H}$, laissant l'autre jeu intact.

Cette définition s'étend sans difficulté à la somme d'un nombre quelconque de jeux, et la somme possède alors les propriétés habituelles d'associativité et de commutativité.

Cette notion de somme pourrait sembler artificielle puisqu'en dehors des mathématiciens, deux joueurs n'ont aucune raison particulière de jouer soudainement à une somme de deux jeux. Pourtant, cette notion apparaît spontanément à l'intérieur même de certains jeux, ce qui justifie son introduction (voir les figures 2.12 et 2.13).

2.6.2 Positions découpables

Définition 6. *Étant donné un jeu* $\mathcal{G}$ *donné, on dit qu'une position* $\mathcal{P}$ *du jeu* $\mathcal{G}$ *est* découpable *si celle-ci peut s'écrire comme une somme de jeux* $\mathcal{P} = \mathcal{G}_A + \mathcal{G}_B + ...$

Les jeux $\mathcal{G}_A, \mathcal{G}_B, ...$ sont souvent désignés comme étant des *positions indépendantes*. Voici la justification de cette dénomination.

Proposition 1. *Les composantes* $\mathcal{G}_A$, $\mathcal{G}_B$, ... *d'une position* découpable *sont des positions du jeu* $\mathcal{G}$.

Démonstration. Il est possible pour les joueurs de ne pas jouer dans le jeu $\mathcal{G}_A$ jusqu'à ce que tous les autres jeux constituant la somme soient ramenés à l'ensemble vide. Cela prouve que $\mathcal{G}_A$ est atteignable depuis $\mathcal{G}$ et donc que $\mathcal{G}_A$ est une position du jeu $\mathcal{G}$. Un argument similaire est possible pour chacune des composantes de la somme. □

Définition 7. *On dira donc qu'une position découpable $\mathscr{P}$ est une* somme de positions $\mathscr{P}_A$, $\mathscr{P}_B$, ..., *et les composantes de la somme seront dites* positions indépendantes.

Ce découpage en positions indépendantes se produit dès lors qu'une position $\mathscr{P}$ peut s'écrire comme une réunion de positions $\mathscr{P}_A$, $\mathscr{P}_B$, ... et qu'à chaque tour, tout coup joué n'influence qu'une et une seule de ces positions.

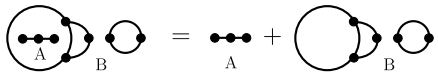


FIGURE 2.12 – Position de Sprouts découpable.

Par exemple, la position de Sprouts de la figure 2.12 peut être vue comme la somme de deux positions indépendantes : on ne peut plus jouer avec les deux points à l'interface des régions A et B, si bien que lors de chaque coup ultérieur, le joueur dont c'est le tour devra choisir de jouer, ou bien dans la région A, ou bien dans la région B.

Le découpage en positions indépendantes se produit également dans le cas du Cram. La figure 2.13 montre un exemple de partie de Cram jouée sur un quadrillage de taille 3×5 : après deux coups, la position obtenue se décompose en une somme de deux positions indépendantes.

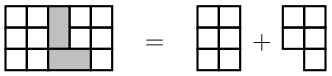


FIGURE 2.13 – Position de Cram découpable.

Définition 8. *Nous appelons* jeux découpables *les jeux dans lesquels interviennent des positions découpables.*

Comme le montrent les figures 2.12 et 2.13, le Sprouts et le Cram sont des jeux découpables. Le jeu de Nim ou les jeux octaux sont également des jeux impartiaux découpables. Par contre, le jeu de Nim restreint à une seule colonne ou le jeu de Chomp sont bien des jeux impartiaux, mais ne sont pas découpables.

L'un des objectifs de cette thèse est de décrire des méthodes efficaces pour résoudre les jeux découpables, comme le Sprouts et le Cram. Les chapitres 3 et 4 s'attacheront donc à montrer comment exploiter efficacement les découpages pour accélérer les calculs, en version normale, puis en version misère.

Bien que les notions de positions et de jeux découpables soient fondamentales dans la théorie des jeux impartiaux, nous attirons l'attention du lecteur sur le fait que le terme « découpable » choisi dans le cadre de cette thèse ne semble pas exister de façon standard dans la littérature, ni en français, ni en anglais.

2.6.3 Indistinguabilité

Deux positions sont dites *indistinguables* si, dans toute somme de positions indépendantes où l'une d'elle intervient, on peut la remplacer par l'autre sans changer l'issue de la somme. On peut alors remplacer la position la plus compliquée par la plus simple dans chaque somme la comportant, ce qui permet d'accélérer les calculs.

Par exemple, dans la figure 2.14, les deux positions à gauche sont indistinguables, ce qui implique que la somme de positions du milieu a la même issue que la somme de positions de droite. Si notre objectif est de calculer l'issue de la somme du milieu, alors nous avons simplifié ce problème, puisque l'étude de la somme de droite sera plus rapide.

Plus formellement, nous pouvons donner la définition suivante de l'indistinguabilité :

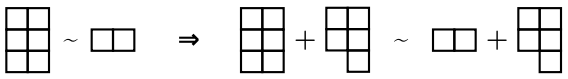


FIGURE 2.14 – Simplification d'une somme de positions de Cram en utilisant l'indistinguabilité.

Définition 9. *Soit $\mathscr{E}$ un ensemble de positions de jeux impartiaux.*

Deux positions $\mathscr{P}_1$ et $\mathscr{P}_2$ seront dites indistinguables dans l'ensemble $\mathscr{E}$, et nous noterons $\mathscr{P}_1 \sim^+_{\mathscr{E}} \mathscr{P}_2$ (respectivement $\mathscr{P}_1 \sim^-_{\mathscr{E}} \mathscr{P}_2$), si quelle que soit la position $\mathscr{P} \in \mathscr{E}$, les sommes de positions $\mathscr{P}_1 + \mathscr{P}$ et $\mathscr{P}_2 + \mathscr{P}$ ont la même issue en version normale (respectivement en version misère).

Dans cette définition, les positions $\mathscr{P}_1$ et $\mathscr{P}_2$ peuvent appartenir à n'importe quel jeu impartial.

Il est important de préciser si nous travaillons en version normale ou en version misère. Si nous prenons pour $\mathscr{E}$ l'ensemble des positions du jeu de Sprouts, et pour $\mathscr{P}_1$ et $\mathscr{P}_2$ les positions de départ du Sprouts à respectivement 1 et 2 points, alors $\mathscr{P}_1 \sim^+_{\mathscr{E}} \mathscr{P}_2$: nous verrons que ces deux positions ont le même nimber, $\mathbb{0}$, ce qui assure leur indistinguabilité en version normale. Par contre, $\mathscr{P}_1 \nsim^-_{\mathscr{E}} \mathscr{P}_2$. En effet, on peut prendre la position terminale comme position $\mathscr{P}$ pour les distinguer, puisqu'en version misère $\mathscr{P}_1$ est gagnante, et $\mathscr{P}_2$ est perdante.

L'indistinguabilité étant une relation d'équivalence, on peut déterminer des classes d'indistinguabilité. En prenant pour $\mathscr{E}$ l'ensemble des jeux impartiaux, en version normale, le théorème de Sprague-Grundy établit que toute position d'un jeu impartial est indistinguable d'une certaine colonne du jeu de Nim, appelée son *nimber*. L'utilisation des nimbers pour accélérer les calculs sera l'objet du chapitre 3.

En version misère, les classes d'indistinguabilité sont bien plus nombreuses et compliquées. C'est le concept d'*arbre canonique réduit*, décrit dans le chapitre 4, qui modélise ces classes. Cette difficulté supplémentaire illustre le fait que les jeux en version misère sont plus compliqués à étudier.

Si l'on limite l'ensemble $\mathscr{E}$ à un jeu particulier (par exemple, l'ensemble des positions du Sprouts), il est possible que les classes d'indistinguabilité soient moins nombreuses que les arbres canoniques réduits. Mais il est rarement facile de déterminer précisément ces classes. Au moins, le concept d'arbre canonique réduit est assuré de fonctionner sur tous les jeux impartiaux en version misère.

2.7 Calculs informatiques

2.7.1 Résolution des jeux

Le principal objectif de cette thèse est de *résoudre* des jeux, c'est-à-dire de déterminer si la position de départ est gagnante ou perdante, et d'expliciter la stratégie du joueur en position de force. Notre but sera donc en général de calculer un arbre solution pour les positions de départ de ces jeux.

On trouve parfois dans la littérature le terme de *résolution faible* (« weak solution ») pour désigner le calcul d'un arbre solution, par opposition aux termes de résolution *ultra-faible* (« ultra-weak solution ») et *forte* (« strong solution »). Une résolution ultra-faible consiste seulement à déterminer l'issue gagnante ou perdante de la racine, sans forcément expliciter un arbre solution, et une solution forte consiste cette fois à déterminer l'issue gagnante ou perdante de l'ensemble des positions du jeu.

Si l'on reprend les exemples des paragraphes précédent, la figure 2.5 constitue une solution forte pour la racine et la figure 2.6 une solution faible. Une solution ultra-faible consisterait à prouver par un moyen simple et indirect que la racine est gagnante.

Un exemple classique de résolution ultra-faible est le suivant : aux échecs, Kasparov nous laisse choisir Blanc ou Noir. Or, la position de départ classique aux échecs est soit gagnante pour les blancs, soit gagnante pour les noirs, soit nulle. Dans les deux premiers cas, nous nous trouvons dans une position gagnante, et dans le troisième cas, nous nous trouvons dans une position nulle. Ainsi, la position dans laquelle nous nous trouvons est de résolution triviale, c'est une position nulle ou gagnante. Mais, comme il ne s'agit que d'une résolution ultra-faible, il y a fort à parier qu'au final nous perdions notre partie contre ce bon Garry...

On remarquera que le calcul d'une solution forte se ramène en général à un simple calcul récursif et direct de la totalité de l'arbre de jeu, ce qui nécessite des techniques algorithmiques spécifiques, axées sur la compression de données, plutôt que sur l'exploration des arbres de jeu. Le principal facteur limitant est la taille de l'arbre de jeu. Une solution forte n'est praticable que pour des jeux dont la combinatoire est relativement limitée, la limite étant surtout fixée par les outils informatiques existants.

Inversement, une solution ultra-faible est généralement obtenue par un argument non constructif, qui permet d'affirmer que la victoire appartient à tel joueur, mais sans indiquer de quelle façon celui-ci peut gagner. Il s'agit souvent d'un argument immédiat, du type *vol de stratégie* (« strategy-stealing argument »). Une solution ultra-faible n'est d'aucun intérêt pratique pour le joueur, et n'est possible que pour certains jeux particuliers.

Le cas le plus intéressant est en fait celui des jeux qui sont trop complexes pour être résolus fortement, mais qui restent tout de même accessibles à une résolution faible. Il faut alors développer des algorithmes « intelligents », capables de trouver un arbre solution de taille raisonnable au sein d'un arbre de jeu parfois de taille extrêmement importante.

La plupart des calculs réalisés dans cette thèse sont des résolutions faibles. Nous verrons cependant que les calculs d'arbres canoniques ou d'arbres canoniques réduits sont équivalents à une résolution forte. Nous avons également pu observer la possibilité d'utiliser des techniques de résolution ultra-faible dans les jeux impartiaux, et nous avons même implémenté de telles techniques pour accélérer les calculs de Dots-and-boxes.

2.7.2 Complexité spatiale d'un jeu

La *complexité spatiale*[3] d'un jeu est le nombre de positions différentes de ce jeu. C'est un indicateur usuel de la difficulté des jeux : a priori, plus la complexité spatiale est importante, plus la résolution du jeu est difficile à obtenir.

Voici les complexités spatiales de quelques jeux classiques.

Tic-Tac-Toe	10^3
Connect Four	10^{13}
Dames anglaises	10^{20}
Échecs	10^{47}
Go	10^{171}

TABLE 2.2 – Complexité spatiale de certains jeux.

On pourra consulter à ce sujet l'article fondateur de Claude Shannon sur le jeu d'échecs [40], dans lequel il livrait dès 1950 une première estimation du nombre qui porte désormais son nom, ainsi que le célèbre article de Jonathan Schaeffer de 2007 concernant les dames anglaises

3. Le terme de *complexité spatiale* n'est pas standard. Il correspond à l'anglais *state-space complexity*, et n'a aucun lien avec le terme de *complexité en espace* (anglais *space complexity*) utilisé dans la théorie de la complexité lors de l'étude de la mémoire utilisée par un algorithme.

[36], où il décrit comment il a pu obtenir une résolution faible du jeu. La complexité spatiale du Connect Four dans sa version standard (grille de taille 6×7) a été déterminée de façon exacte par Peter Kissmann en 2008 [26] et indépendamment par John Tromp [43]. Une borne supérieure pour le jeu de Go a été calculée en 2009 par John Tromp et Gunnar Farnebäck et ils conjecturent que la valeur exacte sera atteignable dans les dix prochaines années [44].

Cependant, la complexité spatiale est loin d'être le seul élément permettant de justifier de la difficulté d'un jeu. Nous avons résolu le jeu de Sprouts à 53 points, jeu dont la complexité spatiale dépasse celle des échecs... mais nous serions évidemment bien incapables d'obtenir le même résultat sur les échecs. La notion de complexité spatiale ne recouvre pas toutes les simplifications théoriques, compressions de données, ou autres idées que l'on peut avoir pour accélérer la résolution des jeux.

2.7.3 Arbres de recherche

Notre objectif principal, étant donné un jeu, est donc de déterminer un arbre solution pour ce jeu. Pour cela, nous développons partiellement un arbre de jeu – partiellement, car sinon, trop de mémoire serait occupée par le développement de l'arbre complet. Puis, à partir des positions terminales dont nous connaissons l'issue, nous remontons l'information jusqu'à obtenir l'issue de la racine. Cet arbre de jeu partiellement développé, qui évolue au cours du calcul, s'appelle *arbre de recherche*.

Pour développer l'arbre de recherche, on utilise un *algorithme de parcours*. Le plus élémentaire consiste à parcourir l'arbre en profondeur (« depth-first » en anglais, ou alpha-bêta). La figure 2.15 illustre cet algorithme classique.

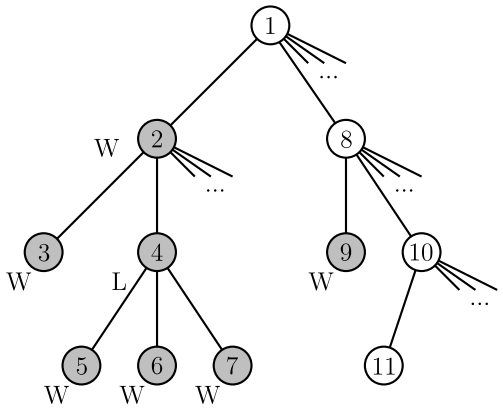


FIGURE 2.15 – Parcours d'un arbre en profondeur.

Les nœuds sont numérotés dans l'ordre de leur étude. La force de l'algorithme de parcours en profondeur est qu'il ne nécessite que peu de mémoire. À un instant donné, il lui suffit de stocker en mémoire la *branche de calcul* (les nœuds en blanc sur la figure 2.15), et il finira par déterminer l'issue de la racine. Remarquons que cet algorithme fait des calculs inutiles. Par exemple, le calcul du nœud numéroté « 3 » est inutile, puisque le nœud « 4 » est perdant.

On peut améliorer les performances de l'algorithme de parcours en développant des heuristiques pour trier les nœuds, de sorte à éviter d'étudier des nœuds inutiles, et à étudier en priorité les nœuds perdants dont le sous-arbre est le plus simple possible, lorsqu'il est possible de choisir entre plusieurs nœuds perdants.

Ces améliorations se révèlent parfois insuffisantes. Il est alors nécessaire de se tourner vers des parcours de type « meilleur d'abord » (« best-first » en anglais). En particulier, nous utiliserons l'algorithme *Proof-number search* (PN-search), développé initialement par L. Victor Allis [1].

2.7.4 Transpositions

Une *transposition* intervient dès lors qu'une position peut être atteinte par des suites de coups différentes. Il en résulte que dans les arbres de jeu, on peut identifier les nœuds correspondants à une même position. Ce faisant, on obtient des graphes qui ne sont plus des arbres, mais on continuera à parler d'*arbres* de jeu ou de recherche, par abus de langage.

La figure 2.16 montre un exemple de transposition, que l'on a isolée à l'intérieur de l'arbre de jeu de la position de départ 3×5 du Cram. Les transpositions de ce type sont très fréquentes dans les jeux combinatoires, ce sont celles qui se produisent quand un joueur joue un coup A, que l'autre joueur joue un coup B, et que les coups A et B se produisent dans des endroits différents des plateaux de jeu. Ainsi, l'ordre dans lequel on a joué les coups A et B n'a pas d'importance, on retrouve la même position après avoir joué ces deux coups. Des transpositions de ce type interviennent tout à la fois dans le Sprouts, le Cram ou le Dots-and-boxes.

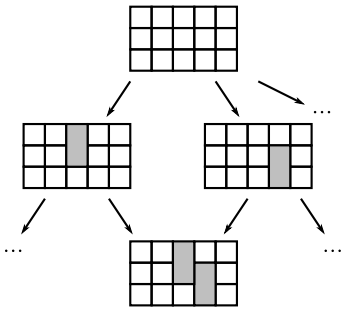


FIGURE 2.16 – Transposition dans un arbre de jeu.

Ce ne sont cependant pas les seules transpositions possibles. Certaines transpositions, comme sur la figure 2.17, interviennent suite à l'identification de positions équivalentes. Ici, on a identifié les positions identiques après suppression des carrés isolés, dans lesquels on ne peut plus jouer. Le coup consistant à jouer au centre de la pièce de 4 carrés, et qui produit 2 carrés isolés, engendre une position équivalente aux deux coups qui recouvrent complètement la pièce.

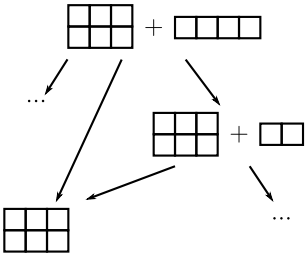


FIGURE 2.17 – Transposition suite à l'identification de positions équivalentes.

Une transposition comme celle de cette figure ne peut intervenir, a priori, que dans un jeu impartial. Dans un jeu partisan, il y a alternance entre les étages de l'arbre de jeu : la racine (étage 0), et tous les nœuds placés aux étages pairs correspondent aux positions pour lesquelles c'est le tour du premier joueur, et les nœuds placés aux étages impairs, à celles pour lesquelles c'est le tour du deuxième joueur. Les transpositions ne peuvent donc intervenir qu'entre des positions dont les étages ont la même parité. Nous verrons toutefois dans le chapitre 13 qu'il est possible d'identifier des positions dont les étages sont de parité différente même pour des jeux partisans.

L'utilité principale des transpositions est d'éviter de refaire de nombreuses fois les mêmes calculs. Une fois l'issue d'une position calculée dans une partie de l'arbre de recherche, on peut en profiter dans toutes les autres parties de l'arbre où cette position intervient. Le gain en temps de calcul est tel que pour tous les calculs un tant soit peu complexes que nous avons menés, il aurait été inenvisageable d'espérer obtenir le résultat sans tenir compte des transpositions.

Le prix à payer pour ce gain en temps de calcul, c'est le stockage d'une *table de transpositions*, dans laquelle nous stockons les issues des positions précédemment calculées. Nous décrirons dans ce mémoire plusieurs techniques visant à empêcher ces tables de saturer la mémoire. Ce n'est pas le seul inconvénient des transpositions, nous pouvons notamment citer la perte du caractère arborescent de l'arbre de recherche, qui pose des problèmes avec l'algorithme de parcours PN-search.

2.7.5 Espace et temps de calcul

Les transpositions sont une des multiples illustrations d'un problème classique en informatique, celui de la transformation de l'espace en temps de calcul. Le titre du mémoire de Dennis Breuker, « Memory versus Search in Games » [9], retranscrit ce problème de manière éloquente.

La transformation est possible dans les deux sens. Dans le cas des transpositions, une perte de mémoire permet un gain beaucoup plus important en temps, ce qui permet de justifier leur utilisation. Mais nous avons fréquemment dû lutter contre la saturation de la mémoire, du fait de la taille importante de ces tables.

Une méthode classique que nous avons presque systématiquement utilisée pour contrer ce problème consiste à ne stocker que les positions perdantes. Nous avons vu au paragraphe 2.4.3 que les positions perdantes sont nettement moins nombreuses que les positions gagnantes. Ainsi, on peut nettement diminuer la taille des tables de transpositions (typiquement, d'un facteur 5 ou 10 dans les jeux que nous avons programmés), pour le prix d'un temps de calcul plus important. En effet, lorsque l'on rencontre à nouveau une position que l'on savait gagnante, il faut d'abord calculer ses options, puis se rendre compte qu'une de ces options est dans la table de transpositions, avant de constater que la position est gagnante.

Cette méthode est donc un exemple de transformation dans l'autre sens, on utilise moins d'espace mémoire mais plus de temps de calcul.

Chapitre 3

Jeux impartiaux en version normale

3.1 Introduction

Le chapitre 2 a posé les bases du problème qui nous intéresse. Nous cherchons à résoudre des jeux impartiaux par le calcul informatique, et en particulier les jeux découpables. Pour tout jeu impartial, il existe deux versions de jeu, suivant la convention de victoire choisie : version *normale* si le joueur qui ne peut plus jouer a perdu, et version *misère* si le joueur qui ne peut plus jouer a gagné. En règle générale, l'étude des jeux en version misère est nettement plus compliquée que celle des jeux en version normale. Dans ce chapitre, ce sont ces derniers que nous allons étudier.

Nous reviendrons tout d'abord sur le concept de sommes de positions, présenté brièvement dans le chapitre précédent. Il débouche sur le résultat fondamental de Sprague et Grundy qui permet de classifier tous les jeux impartiaux en version normale en se basant sur le jeu de Nim.

Dans les sections suivantes, qui forment le point original de ce chapitre, nous aborderons alors le problème du calcul de l'issue d'une somme de positions, ainsi que celui du calcul du nimber d'une position. La notion d'arbre solution, et son extension, la notion d'*arbre solution nimber* nous permettront d'obtenir plusieurs théorèmes, dont le *théorème d'inévitabilité* qui apporte une réponse au problème du calcul de l'issue d'une somme de positions. Ces théorèmes débouchent directement sur des algorithmes, algorithmes qui ont contribué à la résolution de deux jeux impartiaux, le Sprouts, et le Cram.

3.2 Sommes de positions

La notion de *somme de positions* ayant déjà été présentée à la section 2.6, dans cette section, nous expliquons des techniques élémentaires qui permettent d'utiliser les sommes de positions pour accélérer les calculs. Le but de la suite de ce chapitre sera alors d'exploiter au mieux ces sommes de positions à l'aide de méthodes plus élaborées.

3.2.1 Stratégie de symétrie

Nous présentons en premier lieu un résultat simple concernant l'issue des sommes de positions, à savoir que la somme d'une position avec elle-même est toujours d'issue perdante.

Théorème 3 (stratégie de symétrie)**.** *Quelle que soit la position $\mathscr{P}$, la somme $\mathscr{P}+\mathscr{P}$ est d'issue perdante.*

Démonstration. Par induction de Conway : le joueur dont c'est le tour choisit une option, qui est forcément du type $\mathscr{P}+\mathscr{P}^i$ (la somme étant commutative, $\mathscr{P}^i+\mathscr{P}$ correspond à la même position). Mais son adversaire peut alors choisir l'option $\mathscr{P}^i+\mathscr{P}^i$ qui est perdante par hypothèse d'induction. Toutes les options de $\mathscr{P}+\mathscr{P}$ sont donc gagnantes, ce qui prouve que cette somme est perdante. □

La stratégie décrite dans cette preuve est appelée stratégie de symétrie, car le joueur qui joue en second se contente de copier dans l'autre composante chaque coup de son adversaire, s'assurant ainsi de ne jamais jouer en dernier.

3.2.2 Issue d'une somme de positions

Le résultat du paragraphe précédent ne s'applique que rarement en pratique. Le théorème qui suit est plus utile, il énonce que dans certains cas, on peut déterminer l'issue de la somme avec la seule connaissance des issues de chacune des composantes.

Théorème 4. *La somme de deux positions d'issue perdante est d'issue perdante. La somme d'une position d'issue perdante et d'une position d'issue gagnante est d'issue gagnante.*

Démonstration. Par induction de Conway.

Tout coup joué à partir d'une somme $\mathscr{P}$ de deux positions perdantes conduit à une somme d'une position perdante et d'une position gagnante (la position gagnante étant celle dans laquelle le coup a été joué). Par hypothèse d'induction, toutes ces sommes sont des positions gagnantes et donc $\mathscr{P}$ est perdante.

Inversement, à partir d'une somme $\mathscr{P}$ d'une position perdante et d'une position gagnante, on peut jouer un coup dans la position gagnante pour que l'adversaire soit confronté à une somme de deux positions perdantes. Par hypothèse d'induction, cette somme est perdante et donc $\mathscr{P}$ est gagnante. □

Ce résultat a notamment été utilisé dans la programmation du jeu de Sprouts par Applegate, Jacobson et Sleator [4]. On notera cependant qu'il n'indique rien sur l'issue de la somme lorsque les deux composantes sont gagnantes. Pour déterminer l'issue de la somme même dans ce cas, il faut faire appel à la notion de *nimber*, qui fait l'objet de la section 3.3.

3.3 Nimber

3.3.1 Le jeu de Nim

Nous avons déjà présenté le jeu de Nim dans le chapitre précédent (§2.2.1). Il se joue avec des colonnes d'allumettes, et chaque coup consiste à enlever un certain nombre d'allumettes dans une seule colonne. Rappelons que la colonne de Nim à n allumettes est notée $\mathbb{n}$. Par exemple, $\mathbb{7}+\mathbb{5}+\mathbb{4}+\mathbb{2}$ est la position composée de 4 colonnes, de 7, 5, 4 et 2 allumettes respectivement.

Comme à chaque coup, on ne peut jouer que dans une et une seule colonne, les colonnes sont indépendantes les unes des autres. Le jeu de Nim est donc un *jeu découpable* au sens de la définition 8. Une position du jeu de Nim est la somme de ses colonnes, qui forment chacune une position indépendante.

La résolution du jeu de Nim a été décrite pour la première fois par Charles L. Bouton, en 1902 [7]. Cette méthode utilise l'opérateur $\oplus$ (*ou exclusif bit à bit*), que nous appellerons la *Nim-addition*. Pour calculer la Nim-addition de deux nombres entiers, on met chaque nombre sous forme binaire, puis on ajoute les bits correspondants avec l'addition de $\mathbb{Z}/2\mathbb{Z}$ (0+0=0,

0+1=1 et 1+1=0). Par exemple, $9 \oplus 12$ s'écrit en binaire $1001 \oplus 1100$, ce qui donne 0101 en binaire, donc $5 : 9 \oplus 12 = 5$.

La méthode de Bouton peut maintenant s'exprimer simplement :

Théorème 5 (Bouton)**.** *Une position du jeu de Nim est perdante si et seulement si la Nim-addition de toutes ses colonnes vaut 0.*

En reprenant l'exemple précédent, la position $\mathbb{7}+\mathbb{5}+\mathbb{4}+\mathbb{2}$ est gagnante, car $7 \oplus 5 \oplus 4 \oplus 2 = 4$. Les coups gagnants sont ceux qui conduisent à une position perdante : il faut donc jouer vers $\mathbb{3}+\mathbb{5}+\mathbb{4}+\mathbb{2}$ (car $3 \oplus 5 \oplus 4 \oplus 2 = 0$), vers $\mathbb{7}+\mathbb{1}+\mathbb{4}+\mathbb{2}$ ou vers $\mathbb{7}+\mathbb{5}+\mathbb{0}+\mathbb{2}$.

3.3.2 Indistinguabilité

La notion d'indistinguabilité a été présentée dans un cadre général au paragraphe 2.6.3. Dans ce chapitre-ci, nous dirons que deux positions $\mathscr{A}$ et $\mathscr{B}$ sont *indistinguables*, et nous noterons $\mathscr{A} \sim \mathscr{B}$, si quelle que soit la position $\mathscr{P}$, les sommes $\mathscr{A}+\mathscr{P}$ et $\mathscr{B}+\mathscr{P}$ ont la même issue. Dans cette définition, les positions $\mathscr{A}$, $\mathscr{B}$ ou $\mathscr{P}$ peuvent appartenir à n'importe quel jeu combinatoire impartial.

Rappelons que l'intérêt de l'indistinguabilité est de permettre de remplacer, dans une somme de positions, des positions compliquées par des positions plus simples, de façon à accélérer les calculs. Un premier résultat va nous permettre d'identifier des positions indistinguables :

Proposition 2. *$\mathscr{A}$ et $\mathscr{B}$ sont indistinguables si et seulement si $\mathscr{A}+\mathscr{B}$ est d'issue perdante.*

Démonstration. Si $\mathscr{A}$ et $\mathscr{B}$ sont indistinguables, alors quelle que soit la position $\mathscr{P}$, les sommes $\mathscr{A}+\mathscr{P}$ et $\mathscr{B}+\mathscr{P}$ ont la même issue. En particulier, $\mathscr{A}+\mathscr{A}$ et $\mathscr{B}+\mathscr{A}$ ont la même issue. Or, $\mathscr{A}+\mathscr{A}$ est d'issue perdante par la stratégie de symétrie du théorème 3, et donc $\mathscr{A}+\mathscr{B}$ est également perdante.

Inversement, si $\mathscr{A}+\mathscr{B}$ est d'issue perdante, alors, d'après le théorème 4, $\mathscr{A}+\mathscr{P}$ a la même issue que $(\mathscr{A}+\mathscr{P})+(\mathscr{A}+\mathscr{B})$, c'est-à-dire la même issue que $(\mathscr{B}+\mathscr{P})+(\mathscr{A}+\mathscr{A})$. Or, d'après le théorème 3, $\mathscr{A}+\mathscr{A}$ est d'issue perdante, et donc avec une nouvelle application du théorème 4, $(\mathscr{B}+\mathscr{P})+(\mathscr{A}+\mathscr{A})$ a la même issue que $\mathscr{B}+\mathscr{P}$. On a donc bien montré que pour toute position $\mathscr{P}$, les sommes $\mathscr{A}+\mathscr{P}$ et $\mathscr{B}+\mathscr{P}$ ont la même issue. □

Ce résultat va maintenant nous servir à démontrer le résultat fondamental des jeux impartiaux en version normale, qui décrit les classes d'équivalence pour la relation d'indistinguabilité.

3.3.3 Nimber

Le principal résultat de la théorie des jeux impartiaux est le théorème de Sprague-Grundy, qui fut découvert indépendamment par Roland Sprague en 1935 [42] et Patrick Grundy en 1939 [22]. La définition suivante est nécessaire à l'énoncé du théorème.

Définition 10. *On définit le* mex *(minimum exclu) d'un ensemble de nombres entiers comme le plus petit entier naturel n'appartenant pas à cet ensemble.*

Par exemple, $\mathrm{mex}(1,4) = 0$, $\mathrm{mex}(0,1,2,5) = 3$.

Théorème 6 (Sprague-Grundy)**.** *Toute position d'un jeu combinatoire impartial court est indistinguable d'une certaine colonne de Nim, appelée son* nimber*, qui vaut le* mex *des nimbers de ses options.*

Signalons que le nimber est également appelé *nombre* ou *fonction de Sprague-Grundy*.

Démonstration. Nous allons prouver ce résultat par induction de Conway.

Soit $\mathscr{P}$ une position d'un jeu combinatoire impartial. Par hypothèse d'induction, on suppose que chaque option $\mathscr{P}^i$ de $\mathscr{P}$ est indistinguable d'une certaine colonne de Nim, notée $\mathbb{p}_i$. Il nous faut montrer que $\mathscr{P}$ est indistinguable de $\mathbb{p} = \text{mex}(\mathbb{p}_i)$, c'est-à-dire que $\mathscr{P} + \mathbb{p}$ est perdante d'après la proposition 2.

Les options de $\mathscr{P} + \mathbb{p}$ appartiennent à l'une des trois catégories suivantes, et l'on va montrer qu'elles sont toutes gagnantes, c'est-à-dire qu'elles possèdent chacune une option perdante :

* $\mathscr{P} + \mathbb{q}$, avec $\mathbb{q} < \mathbb{p}$. Or, par définition du mex, il existe nécessairement une option $\mathscr{P}^i$ indistinguable de $\mathbb{q}$, c'est-à-dire telle que $\mathscr{P}^i + \mathbb{q}$ est perdante.
* $\mathscr{P}^i + \mathbb{p}$ où $\mathscr{P}^i$ est indistinguable d'un certain $\mathbb{p}_i < \mathbb{p}$. Comme $\mathbb{p}_i$ est inférieur à $\mathbb{p}$, c'est une option de $\mathbb{p}$, et donc $\mathscr{P}^i + \mathbb{p}_i$ est une option perdante de $\mathscr{P}^i + \mathbb{p}$.
* $\mathscr{P}^i + \mathbb{p}$ où $\mathscr{P}^i$ est indistinguable d'un certain $\mathbb{p}_i > \mathbb{p}$. Or, par hypothèse d'induction, $\mathbb{p}_i$ est le mex des options de $\mathscr{P}^i$, et donc $\mathscr{P}^i$ admet une option $\mathscr{P}^{ij}$ de nimber $\mathbb{p}$. $\mathscr{P}^{ij} + \mathbb{p}$ est donc une option perdante de $\mathscr{P}^i + \mathbb{p}$.

□

Ce théorème et sa démonstration appellent quelques commentaires.

Nous nous sommes restreints dans notre présentation du théorème aux jeux courts, et les nimbers sont donc simplement des colonnes de Nim avec un nombre fini d'allumettes. Mais ce théorème peut se généraliser aux jeux combinatoires impartiaux quelconques, auquel cas il faut étendre la définition des nimbers à un ordinal quelconque [1].

Dans la démonstration, nous avons utilisé la caractérisation suivante qui se déduit de la proposition 2 :

Proposition 3. *Une position $\mathscr{P}$ est de nimber $\mathbb{k}$ (i.e. $\mathscr{P} \sim \mathbb{k}$) si et seulement si $\mathscr{P} + \mathbb{k}$ est d'issue perdante.*

Cette caractérisation s'avère très utile en pratique, et dans le cas particulier où $\mathbb{k} = \mathbb{0}$, on obtient :

Proposition 4. *Une position est perdante si et seulement si son nimber est $\mathbb{0}$.*

On notera enfin que la définition du mex implique la propriété suivante :

Proposition 5. *Si l'on peut jouer n coups à partir d'une position, alors son nimber est au plus $\mathbb{n}$.*

3.3.4 Détermination du nimber

En pratique, si l'on connaît l'arbre de jeu d'une position, il est facile de calculer récursivement le nimber de cette position. En effet, le nimber d'une position terminale est $\mathbb{0}$, puis le fait que le nimber d'une position soit égal au mex des nimbers de ses options permet de remonter les valeurs.

Sur la figure 3.1, nous avons appliqué ce principe à une position de Cram.

3.3.5 Nimber d'une somme

Le nimber trouve son intérêt lorsque l'on rencontre une somme de positions indépendantes. On peut en effet calculer le nimber de la somme si l'on connaît le nimber de chacune des composantes, avec la Nim-addition, puis en déduire son issue avec la proposition 4 : la somme sera perdante si son nimber est $\mathbb{0}$, et gagnante si son nimber est $\geq \mathbb{1}$. Cette méthode

1. Voir par exemple *ONAG* [12], p.124, ou Schleicher et Stoll [38], théorème 7.5.

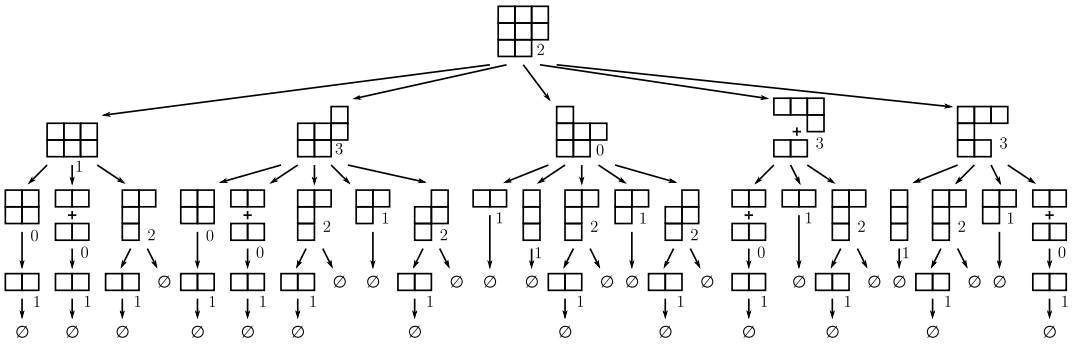


FIGURE 3.1 – Nimbers de l'arbre de jeu d'une position de Cram.

est plus efficace que le théorème 4, car elle permet de trouver l'issue de la somme même lorsque ses deux composantes sont gagnantes.

En guise d'exemple, nous reprenons sur la figure 3.2 deux sommes de positions que nous avions présentées avec les figures 2.12 et 2.13.

FIGURE 3.2 – Deux sommes de positions.

La première composante de la somme de gauche est de nimber $\mathbb{1}$, et l'autre de nimber $\mathbb{0}$, donc la somme de gauche est de nimber $\mathbb{1}$ (car $1 \oplus 0 = 1$). On en déduit que la somme est gagnante, résultat que l'on aurait pu déduire aussi du théorème 4.

Mais dans la somme de droite, chaque composante est de nimber 2, donc la somme est de nimber $\mathbb{0}$ (car $2 \oplus 2 = 0$). On en déduit que la somme de droite est perdante, cas que le théorème 4 ne savait pas traiter.

Le résultat suivant explique comment déterminer l'issue de la somme à partir des nimbers sans qu'il soit nécessaire d'avoir recours à la Nim-addition. En effet, en remarquant que $m \oplus n = 0 \Leftrightarrow m = n$, on obtient :

Proposition 6. *Soit $\mathscr{P}_1$ et $\mathscr{P}_2$ deux positions.*
- *$\mathscr{P}_1 + \mathscr{P}_2$ est perdante $\Leftrightarrow$ les nimbers de $\mathscr{P}_1$ et $\mathscr{P}_2$ sont égaux.*
- *$\mathscr{P}_1 + \mathscr{P}_2$ est gagnante $\Leftrightarrow$ les nimbers de $\mathscr{P}_1$ et $\mathscr{P}_2$ sont différents.*

3.4 Arbre solution nimber

3.4.1 Position du problème

Étant donnée une position $\mathscr{P}$ d'un jeu impartial donné, il se pose désormais principalement deux questions :
- Quelle est l'issue de $\mathscr{P}$?
- Quel est le nimber de $\mathscr{P}$?

La première question se pose évidemment pour le joueur qui veut disposer d'une stratégie gagnante à partir de cette position. La deuxième question trouve son utilité lorsque la position $\mathscr{P}$ intervient dans une somme de positions, la connaissance de son nimber étant utile à la détermination de l'issue de la somme via la proposition 6.

Dans la suite de ce chapitre, nous allons donc décrire des algorithmes qui permettent de répondre à ces deux principales questions concernant une position donnée en version normale,

à savoir calculer l'issue et/ou le nimber de cette position. Mais avant cela, il est nécessaire de préciser ce que nous entendons par « calculer ».

Dans les jeux combinatoires, il est parfois possible de connaître l'issue d'une position grâce à une preuve non constructive. On parle alors de résolution ultra-faible (§2.7.1). La figure 3.3 fournit un exemple simple d'une résolution de ce type.

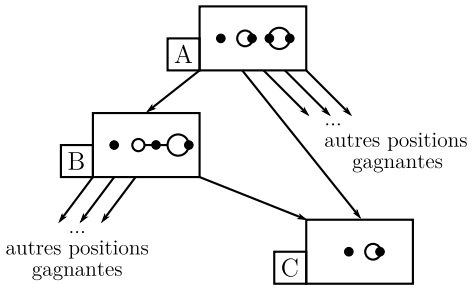


FIGURE 3.3 – Preuve non constructive de l'issue d'une position.

Dans cet exemple, issu du jeu de Sprouts, on ne connaît pas l'état des positions B et C, mais on sait tout de même que la position A est gagnante. En effet, ou bien la position C est perdante, et donc la position A est gagnante ; ou bien la position C est gagnante, et donc la position B est perdante, et la position A est gagnante également. Un joueur confronté à cette situation saurait qu'il dispose d'une stratégie gagnante, mais il ne saurait pas choisir entre l'option B et l'option C.

Une telle résolution n'est pas pleinement satisfaisante : il serait frustrant de déterminer l'issue ou le nimber d'une position sans pouvoir en déduire une stratégie gagnante. Ainsi, les résolutions que nous mènerons seront des résolutions faibles. L'outil adéquat pour les décrire est l'*arbre solution* (§2.4.3).

Par ailleurs, pour déterminer les arbres solutions, nous effectuerons des *preuves par recherche* (§2.4.4), où l'on explore les arbres de jeu, notion qui s'oppose aux *preuves par méthode*, où l'on détermine une méthode assurant un joueur de la victoire (comme dans la résolution du jeu de Nim).

Les preuves par méthodes sont plus efficaces que les preuves par recherche. Elles réalisent en quelque sorte la compression des données de certains arbres solutions, puisqu'il est possible de déduire un arbre solution d'une preuve par méthode. Mais il est souvent difficile de résoudre un jeu exclusivement par méthode. En pratique, la résolution d'un jeu est souvent obtenue par une combinaison de preuves par méthode et par recherche.

En résumé, les résultats que nous démontrons plus loin (en particulier, le théorème d'inévitabilité) sont à considérer dans le cadre de preuves par recherche exclusivement, débouchant sur une résolution faible. Notre objectif sera de simplifier les arbres solutions que nous obtenons de façon à accélérer le calcul.

3.4.2 Arbre solution nimber

Le fait que la définition du mex utilise toutes les options d'une position a parfois fait croire, à tort, que le calcul du nimber d'une position nécessite de développer tout son arbre de jeu, comme nous l'avons fait au paragraphe 3.3.4 (lire par exemple [4], p. 17). Si ce raisonnement était correct, l'utilisation du nimber serait trop coûteuse en temps de calcul lorsque l'on cherche seulement à déterminer l'issue d'une somme de positions, hormis pour les positions presque terminales.

Une simple observation suffit pour se rendre compte de l'erreur de jugement : si le nimber de la position est $\mathbb{0}$, il suffit de montrer qu'elle est perdante pour obtenir son nimber. Or,

le principe d'un arbre solution est justement de ne pas développer tout l'arbre de jeu de la position pour démontrer qu'elle est perdante.

Nous allons généraliser le concept d'arbre solution aux nimbers, et définir la notion d'*arbre solution nimber* d'une position, qui sera un arbre dont la donnée est suffisante pour calculer le nimber de la position. À cette fin, nous commençons par établir deux règles de calcul qui se déduisent immédiatement de la définition du mex.

Règle 1. *La position* $\mathscr{P} \sim \mathbb{n} \Leftrightarrow \mathscr{P}$ *admet une option* $\sim \mathbb{0}$*, une option* $\sim \mathbb{1}$*, ..., une option* $\sim \mathbb{n} - \mathbb{1}$*, et toutes les autres options sont* $\nsim \mathbb{n}$*.*

Règle 2. *La position* $\mathscr{P} \nsim \mathbb{n} \Leftrightarrow \mathscr{P} \sim \mathbb{0}$*, ou* $\mathscr{P} \sim \mathbb{1}$*, ..., ou* $\mathscr{P} \sim \mathbb{n} - \mathbb{1}$*, ou* $\mathscr{P}$ *admet une option* $\sim \mathbb{n}$*.*

L'énoncé de la règle 2 est l'énoncé dual de celui de la règle 1. On remarquera aussi qu'en appliquant ces définitions avec $n = 0$, on retombe sur les règles classiques de calcul de l'issue.

En utilisant ces règles, nous avons établi la figure 3.4. Il s'agit d'un sous-arbre de la figure 3.1 qui suffit pour déterminer le nimber de la racine. On notera en particulier que pour trois des fils de la racine, on montre simplement que le nimber est différent de 2. Pour le démontrer, il nous a suffi de calculer qu'une de leurs options était de nimber 2, et il n'a pas été nécessaire d'étudier les autres options. Partout ailleurs, l'arbre est élagué des branches qui sont inutiles pour démontrer que la racine est de nimber 2.

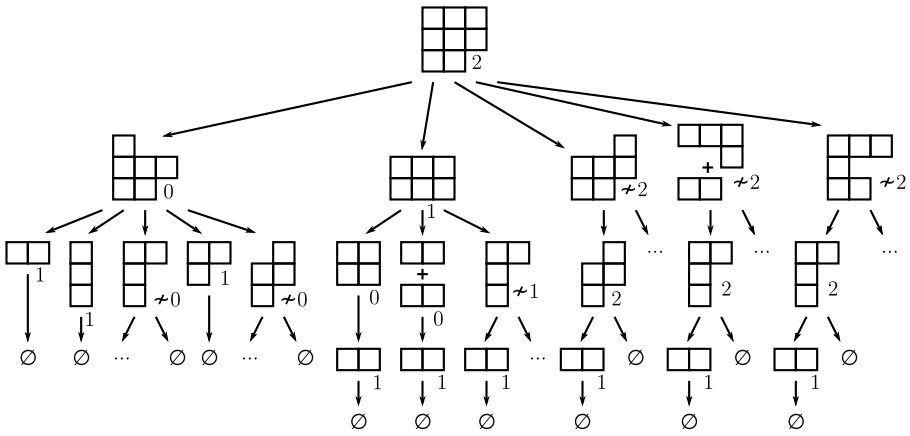


FIGURE 3.4 – Arbre de jeu partiel permettant de calculer le nimber de la racine.

Cet arbre est en fait un exemple d'arbre solution nimber. Nous allons maintenant les définir formellement. Ils sont de deux types : un arbre solution nimber de $\mathscr{P} \sim \mathbb{n}$ démontre que le nimber de $\mathscr{P}$ est $\mathbb{n}$, et un arbre solution nimber de $\mathscr{P} \nsim \mathbb{n}$ démontre que le nimber de $\mathscr{P}$ est différent de $\mathbb{n}$. Là encore, la définition est inductive :

Définition 11.
- $\mathscr{S} = \varnothing$ *est un arbre solution nimber de* $\varnothing \sim \mathbb{0}$*.*
- *Si* $\mathscr{S}$ *est un arbre solution nimber de* $\mathscr{P} \sim \mathbb{n}$*, alors* $\mathscr{S}$ *est un arbre solution nimber de* $\mathscr{P} \nsim \mathbb{p}$ *avec* $p > n$*.*
- *Si* $\mathscr{S}_1$ *est un arbre solution nimber de* $\mathscr{P}^1 \sim \mathbb{n}$*, où* $\mathscr{P}^1$ *est une option de* $\mathscr{P}$*, alors* $\mathscr{S} = \{\mathscr{S}_1\}$ *est un arbre solution nimber de* $\mathscr{P} \nsim \mathbb{n}$*.*
- *Si* $\mathscr{P} = \{\mathscr{P}^0, \mathscr{P}^1, ..., \mathscr{P}^{n-1}, ...\}$*, et si* $\mathscr{S}_0$*,* $\mathscr{S}_1$*, ... sont des arbres solutions nimbers de* $\mathscr{P}^0 \sim \mathbb{0}$*,* $\mathscr{P}^1 \sim \mathbb{1}$*, ...,* $\mathscr{P}^{n-1} \sim \mathbb{n} - \mathbb{1}$*, et de* $\mathscr{P}^i \nsim \mathbb{n}$ *pour* $i \geq n$*, alors* $\mathscr{S} = \{\mathscr{S}_0, \mathscr{S}_1, ..., \mathscr{S}_{n-1}, ...\}$ *est un arbre solution nimber de* $\mathscr{P} \sim \mathbb{n}$*.*

Remarquons qu'un arbre solution est un cas particulier d'arbre solution nimber, qui correspond au cas où $n = 0$. En effet, un arbre solution d'une position $\mathscr{P}$ perdante est également

un arbre solution nimber de $\mathscr{P} \sim \mathbb{0}$, et un arbre solution d'une position $\mathscr{P}$ gagnante est également un arbre solution nimber de $\mathscr{P} \nsim \mathbb{0}$.

Malgré son apparente complexité, cette définition est une simple conséquence des règles 1 et 2, et l'on a les propriétés suivantes :

Proposition 7. $*$ *$\mathscr{P} \sim \mathbb{n} \Leftrightarrow$ il existe un arbre solution nimber de $\mathscr{P} \sim \mathbb{n}$.*
$*$ *$\mathscr{P} \nsim \mathbb{n} \Leftrightarrow$ il existe un arbre solution nimber de $\mathscr{P} \nsim \mathbb{n}$*

3.4.3 Théorème pas à pas

Soit $\mathscr{P}$ une position dont nous souhaitons calculer le nimber. A priori, nous ne savons pas quelle est la valeur de ce nimber. Quelle valeur essayer en premier ? Le théorème suivant résout ce problème, et montre qu'en fait, on peut simplement essayer les différentes valeurs possibles du nimber dans l'ordre croissant.

Théorème 7 (pas à pas)**.** *Un arbre solution nimber de $\mathscr{P} \sim \mathbb{n}$ contient un arbre solution nimber de $\mathscr{P} \nsim \mathbb{0}$, un arbre solution nimber de $\mathscr{P} \nsim \mathbb{1}$, ..., et un arbre solution nimber de $\mathscr{P} \nsim \mathbb{n} - \mathbb{1}$.*

Démonstration. La preuve se déduit immédiatement de la définition 11. Supposons que l'on a calculé un arbre solution nimber $\mathscr{S}$ de $\mathscr{P} \sim \mathbb{n}$. $\mathscr{S}$ est nécessairement de la forme $\mathscr{S} = \{\mathscr{S}_0, \mathscr{S}_1 ... \mathscr{S}_{n-1}, ... \mathscr{S}_i ...\}$ avec $\mathscr{S}_i$ un arbre solution nimber de $\mathscr{P}^i \sim \mathbb{i}$ si $i < n$, et de $\mathscr{P}^i \nsim \mathbb{n}$ si $i \geq n$. Pour chaque $i < n$, on déduit alors $\mathscr{S}'_i = \{\mathscr{S}_i\}$, un arbre solution nimber de $\mathscr{P} \nsim \mathbb{i}$. □

L'intérêt pratique de ce théorème est important, dans la mesure où il implique que pour calculer le nimber de $\mathscr{P}$, on peut simplement déterminer dans l'ordre si $\mathscr{P} \sim \mathbb{0}$, puis si $\mathscr{P} \sim \mathbb{1}$, ... jusqu'à ce que l'on trouve le nimber, sans que cela n'introduise d'information inutile dans l'arbre solution nimber obtenu.

L'algorithme 5 décrit plus loin explicite la manière dont nous avons implémenté cette idée dans notre programme.

3.5 Calcul de l'issue d'une somme

On suppose dans cette section que l'on cherche à calculer l'issue d'une position d'un jeu impartial découpable. Le cas intéressant est celui des positions découpables en sommes de positions indépendantes, et il se pose la question de savoir quelle est la meilleure façon de calculer l'issue de cette somme.

3.5.1 Calcul élémentaire de l'issue d'une somme

L'algorithme 1 présente la méthode la plus simple pour calculer l'issue d'une somme de positions $\mathscr{P}_1 + \mathscr{P}_2$. Elle consiste à ne pas tenir compte du découpage, et à simplement considérer l'ensemble des options de la somme complète $\mathscr{P}_1 + \mathscr{P}_2$.

La figure 3.5 est un exemple d'utilisation de cette méthode. Toutes les sommes de positions de cet arbre solution — en particulier la position de départ — sont du type W+W, et donc le théorème 4 n'est d'aucun secours pour faciliter le calcul de leurs issues.

3.5.2 Autres méthodes élémentaires

L'algorithme 2 présente une amélioration possible grâce au théorème 4, qui indique qu'une position d'issue perdante n'influence pas l'issue de la somme.

Algorithme 1

1: **Pour chaque** option de $\mathscr{P}_1 + \mathscr{P}_2$ **faire**
2: calculer l'issue de l'option
3: **Si** l'option est perdante **alors**
4: renvoyer gagnant
5: **fin Si**
6: **fin Pour**
7: renvoyer perdant (car toutes les options sont gagnantes)

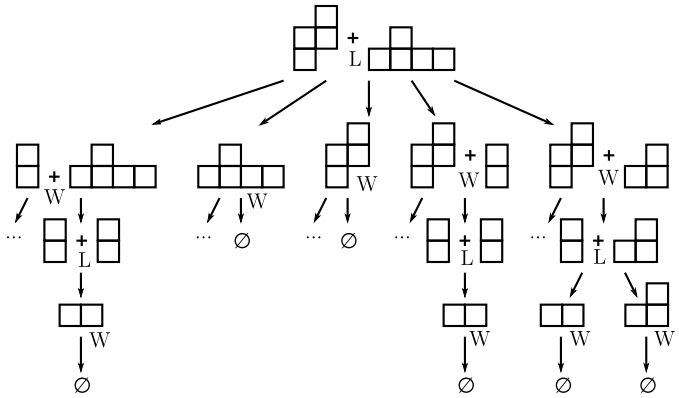


FIGURE 3.5 – Arbre solution d'une somme de positions indépendantes.

Un exemple d'utilisation de cette deuxième méthode pour le Sprouts peut être trouvé dans [4]. La faiblesse de cette méthode est par contre qu'une somme de deux positions gagnantes peut être soit gagnante, soit perdante selon les cas. Le calcul séparé des issues ne permet pas de conclure dans ce cas, et l'on est ramené au calcul de l'issue $\mathscr{P}_1 + \mathscr{P}_2$ avec l'algorithme 1.

Une troisième méthode, présentée dans l'algorithme 3, permet de résoudre le point faible des méthodes précédentes. Il s'agit bien sûr d'appliquer le théorème de Sprague-Grundy, en calculant les nimbers de $\mathscr{P}_1$ et de $\mathscr{P}_2$, pour en déduire le nimber de $\mathscr{P}_1 + \mathscr{P}_2$, et donc l'issue de $\mathscr{P}_1 + \mathscr{P}_2$.

Cette troisième méthode a l'avantage d'exploiter systématiquement le découpage en positions indépendantes, mais elle utilise des calculs de nimbers, au lieu de calculs d'issue, et l'on peut alors se demander si le calcul des nimbers des positions n'est pas plus coûteux que le calcul direct de l'issue de la somme. Le théorème d'inévitabilité va se charger d'apporter une réponse à ce problème.

Algorithme 2

1: calculer l'issue de $\mathscr{P}_1$
2: calculer l'issue de $\mathscr{P}_2$
3: **Si** $\mathscr{P}_1$ est perdante **alors**
4: renvoyer l'issue de $\mathscr{P}_2$
5: **fin Si**
6: **Si** $\mathscr{P}_2$ est perdante **alors**
7: renvoyer l'issue de $\mathscr{P}_1$
8: **fin Si**
9: calculer et renvoyer l'issue de $\mathscr{P}_1 + \mathscr{P}_2$ avec l'algorithme 1

Algorithme 3

1: calculer le nimber de $\mathscr{P}_1$
2: calculer le nimber de $\mathscr{P}_2$
3: en déduire le nimber de $\mathscr{P}_1 + \mathscr{P}_2$ (Nim-addition des nimbers)
4: **Si** le nimber de $\mathscr{P}_1 + \mathscr{P}_2$ est nul **alors**
5: renvoyer perdant
6: **sinon**
7: renvoyer gagnant
8: **fin Si**

3.5.3 Théorème d'inévitabilité des nimbers

Théorème 8 (inévitabilité des nimbers)**.** *De tout arbre solution pour une somme de positions $\mathscr{P}_1 + \mathscr{P}_2$, on peut extraire un arbre solution nimber pour $\mathscr{P}_1$ et $\mathscr{P}_2$.*

Si la somme est perdante, alors c'est un arbre solution nimber de $\mathscr{P}_1 \sim \mathbb{n}$ et de $\mathscr{P}_2 \sim \mathbb{n}$ pour un certain nimber $\mathbb{n}$.

Si la somme est gagnante, alors c'est un arbre solution nimber de $\mathscr{P}_1 \sim \mathbb{n}$ et de $\mathscr{P}_2 \nsim \mathbb{n}$, ou de $\mathscr{P}_1 \nsim \mathbb{n}$ et de $\mathscr{P}_2 \sim \mathbb{n}$, pour un certain nimber $\mathbb{n}$.

Démonstration. La démonstration fonctionne par induction, et explique la signification du mot « extraire » de l'énoncé.

* Cas terminal : si $\mathscr{P}_1 = \varnothing$ et $\mathscr{P}_2 = \varnothing$, alors $\mathscr{S} = \varnothing$ est un arbre solution pour l'issue perdante de $\mathscr{P}_1 + \mathscr{P}_2$, et aussi un arbre solution nimber de $\mathscr{P}_1 \sim \mathbb{0}$ et de $\mathscr{P}_2 \sim \mathbb{0}$.

* Induction :

– Cas 1 : supposons que l'on dispose d'un arbre solution pour la somme gagnante $\mathscr{P}_1+\mathscr{P}_2$. On dispose donc d'un arbre solution pour une option perdante de $\mathscr{P}_1 + \mathscr{P}_2$, par exemple de la forme $\mathscr{P}_1 + \mathscr{P}_2^i$.

Par hypothèse d'induction, on peut en extraire un arbre solution nimber $\mathscr{S}_1$ de $\mathscr{P}_1 \sim \mathbb{n}$, et un arbre solution nimber $\mathscr{S}_2$ de $\mathscr{P}_2^i \sim \mathbb{n}$ pour un certain nimber $\mathbb{n}$. $\{\mathscr{S}_2\}$ est donc un arbre solution nimber de $\mathscr{P}_2 \nsim \mathbb{n}$.

– Cas 2 : supposons que l'on dispose d'un arbre solution pour la somme perdante $\mathscr{P}_1+\mathscr{P}_2$. Il contient donc un arbre solution pour chaque option gagnante de $\mathscr{P}_1 + \mathscr{P}_2$. Ces options sont soit de la forme $\mathscr{P}_1^i + \mathscr{P}_2$, soit de la forme $\mathscr{P}_1 + \mathscr{P}_2^j$.

Appelons $\mathbb{n}$ le nimber de $\mathscr{P}_1$ et de $\mathscr{P}_2$. Nous allons montrer que l'on peut extraire de l'arbre solution de $\mathscr{P}_1 + \mathscr{P}_2$ un arbre solution nimber de $\mathscr{P}_1 \sim \mathbb{n}$, la preuve pour l'arbre solution nimber de $\mathscr{P}_2 \sim \mathbb{n}$ étant symétrique.

Si l'une des options $\mathscr{P}_1 + \mathscr{P}_2^j$ fournit un arbre solution nimber de $\mathscr{P}_1 \sim \mathbb{n}$ (et donc de $\mathscr{P}_2^j \nsim \mathbb{n}$), il n'y a rien à faire. Sinon, chaque option $\mathscr{P}_1+\mathscr{P}_2^j$ fournit un arbre solution nimber de $\mathscr{P}_2^j \sim \mathbb{n}_j$ et de $\mathscr{P}_1 \nsim \mathbb{n}_j$, avec $\mathbb{n}_j \neq \mathbb{n}$.

De plus, comme chaque option $\mathscr{P}_2^j$ de $\mathscr{P}_2$ est représentée, et que $\mathscr{P}_2$ est de nimber $\mathbb{n}$, on dispose donc d'un arbre solution nimber de $\mathscr{P}_1 \nsim \mathbb{n}_j$ pour tout $\mathbb{n}_j < \mathbb{n}$. Chacun de ces arbres est un arbre solution nimber de $\mathscr{P}_1^i \sim \mathbb{n}_j$, où $\mathscr{P}_1^i$ est une certaine option de $\mathscr{P}_1$.

Regardons maintenant les options de la somme $\mathscr{P}_1 + \mathscr{P}_2$ de la forme $\mathscr{P}_1^i + \mathscr{P}_2$. Chacune de ces options, soit fournit un arbre solution nimber de $\mathscr{P}_2 \sim \mathbb{n}$, et donc de $\mathscr{P}_1^i \nsim \mathbb{n}$, soit fournit un arbre solution nimber de $\mathscr{P}_2 \nsim \mathbb{k}$ avec $\mathbb{k} \neq \mathbb{n}$, et donc également un arbre solution nimber de $\mathscr{P}_1^i \sim \mathbb{k}$. Or un tel arbre contient un arbre solution nimber de $\mathscr{P}_1^i \nsim \mathbb{n}$.

En résumé, on dispose d'un arbre solution nimber de $\mathscr{P}_1^i \sim \mathbb{n}_j$ pour tout $\mathbb{n}_j < \mathbb{n}$, et d'un arbre solution nimber de $\mathscr{P}_1^i \nsim \mathbb{n}$ pour toute option $\mathscr{P}_1^i$ de $\mathscr{P}_1$: d'après la définition 11, on en déduit un arbre solution nimber de $\mathscr{P}_1 \sim \mathbb{n}$. □

Ce résultat est surprenant : il exprime que tout algorithme de calcul de l'issue d'une somme par une résolution faible fournit suffisamment d'informations pour déterminer le

nimber d'une des deux positions, et si le nimber de l'autre est égal ou différent (suivant que la somme est perdante ou gagnante). C'est le cas bien sûr de l'algorithme 1, mais plus généralement de tout algorithme qui permet de déterminer un arbre solution pour l'issue de la somme.

Il est donc illusoire de se passer des nimbers en supposant que leur calcul sera trop coûteux, c'est pourquoi nous parlons d'*inévitabilité*. Au contraire, il est toujours plus rentable de calculer l'un des deux nimbers.

Par exemple, plutôt que de développer tout l'arbre de la figure 3.5, il est plus efficace de calculer séparément les nimbers de chacune des positions indépendantes comme dans la figure ci-après. Chacune de ces deux positions est de nimber 2, et donc la somme est de nimber $\mathbb{0}$ (car $2 \oplus 2 = 0$), donc est perdante.

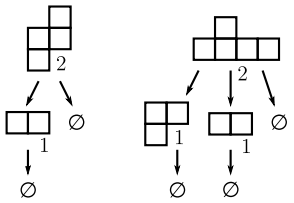


FIGURE 3.6 – Calcul des nimbers des composantes de la somme.

3.5.4 Évaluation des algorithmes

Il est désormais clair que l'algorithme 1 est loin d'être optimal s'il s'agit de déterminer l'issue d'une somme. Mais les algorithmes 2 et 3 ne sont pas non plus parfaits.

L'algorithme 2 est optimal si l'une des composantes est perdante, mais il effectue encore plus de calculs inutiles que l'algorithme 1 si les deux composantes sont gagnantes. En effet, si les deux composantes sont gagnantes, on a certes besoin de calculer le nimber (supérieur ou égal à $\mathbb{1}$) de l'une des composantes et donc également son issue, mais pour l'autre composante, il suffit de montrer que son nimber est différent de cette valeur, ce qui ne nécessite pas forcément de calculer son issue.

L'algorithme 3, qui consiste à calculer le nimber de *chaque* composante, effectue également des calculs inutiles dans certains cas : seul le nimber de l'une des composantes est nécessaire. Pour l'autre composante, il est suffisant de calculer si le nimber est identique ou non.

3.5.5 Calcul d'une somme de 3 positions ou plus

Lorsque nous serons confrontés à une somme de deux positions, en application du théorème d'inévitabilité, nous calculerons le nimber d'une des deux composantes, puis nous chercherons à déterminer si l'autre composante est de nimber identique (dans ce cas la somme sera perdante), ou différent (la somme sera gagnante).

Mais en pratique, comme on le verra dans la section 3.6, les algorithmes de calcul font souvent intervenir des sommes de la forme $\mathscr{P}_1 + \mathscr{P}_2 + \mathbb{n}$, où $\mathbb{n}$ est un certain nimber, lorsque plusieurs découpages ont eu lieu successivement.

Le résultat suivant explique comment prendre en compte ce cas dans nos algorithmes.

Proposition 8. *Pour calculer l'issue d'une somme de positions $\mathscr{P}_1 + \mathscr{P}_2 + \mathbb{n}$, il suffit de calculer le nimber $\mathbb{p}$ de l'une des composantes $\mathscr{P}_1$ ou $\mathscr{P}_2$, et si le nimber de l'autre composante est égal à $\mathbb{p} + \mathbb{n}$ (il sera égal si la somme est perdante, et différent si la somme est gagnante).*

Rappelons que la somme $\mathbb{p} + \mathbb{n}$ doit s'effectuer avec la *Nim-addition*.

Démonstration. Si le nimber de l'une des composantes $\mathscr{P}_1$ ou $\mathscr{P}_2$ est $\mathbb{p}$ et celui de l'autre est $\mathbb{p}+\mathbb{n}$, alors le nimber de la somme $\mathscr{P}_1+\mathscr{P}_2+\mathbb{n}$ est $\mathbb{p}+(\mathbb{p}+\mathbb{n})+\mathbb{n}=\mathbb{0}$, donc la somme est perdante.

Si le nimber de l'autre composante est différent de $\mathbb{p}+\mathbb{n}$, alors le nimber de la somme $\mathscr{P}_1+\mathscr{P}_2+\mathbb{n}$ est différent de $\mathbb{0}$, donc la somme est gagnante. □

On peut étendre ce résultat à un nombre quelconque de composantes :

Proposition 9. *Pour calculer l'issue d'une somme de positions $\mathscr{P}_1+\mathscr{P}_2+...+\mathscr{P}_n+\mathbb{n}$, il suffit de calculer les nimbers $\mathbb{p}_1$, $\mathbb{p}_2$, ..., $\mathbb{p}_{n-1}$ de $(n-1)$ composantes, puis de calculer si le nimber de la composante restante est égal à $\mathbb{p}_1+\mathbb{p}_2+...+\mathbb{p}_{n-1}+\mathbb{n}$ (il sera égal si la somme est perdante, et différent si la somme est gagnante).*

Démonstration. Comme pour la proposition 8, la preuve repose sur l'associativité de la Nim-addition, et sur le fait que pour tout nimber $\mathbb{p}$, $\mathbb{p}+\mathbb{p}=\mathbb{0}$. □

Ce cas général avec un nombre quelconque de composantes est assez rare dans les calculs de Sprouts ou de Cram que nous avons effectués. Il se produit lorsqu'une position est découpable d'un coup en trois composantes ou plus. Pour le Sprouts, en un coup, une position se découpe au maximum en 3 positions indépendantes. Pour le Cram, le maximum est de 4 positions indépendantes. La figure 3.7 présente des exemples de coups qui engendrent ces nombres maximaux de composantes indépendantes.

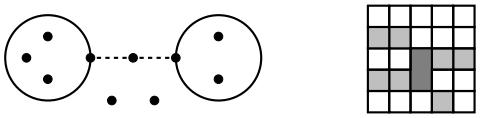


FIGURE 3.7 – Découpage de positions de Sprouts et de Cram.

3.6 Algorithmes de calcul

Les résultats des sections précédentes se traduisent directement en algorithmes de calcul pour les jeux impartiaux en version normale. On se place dans le cadre du calcul, soit de l'issue, soit du nimber, d'une position d'un jeu impartial découpable. Les algorithmes sont applicables également aux jeux non découpables, mais leur intérêt est moindre dans ce cas.

3.6.1 Algorithme de calcul de $\mathscr{P}\sim\mathbb{n}$

Le but de cet algorithme va être de déterminer si le nimber de la position $\mathscr{P}$ est égal ou différent de $\mathbb{n}$. La proposition 3 a montré que le calcul de $\mathscr{P}\sim\mathbb{n}$ était équivalent à celui de $\mathscr{P}+\mathbb{n}$ est d'issue perdante, et $\mathscr{P}\nsim\mathbb{n}$ à celui de $\mathscr{P}+\mathbb{n}$ est d'issue gagnante. Pour savoir si $\mathscr{P}\sim\mathbb{n}$ ou $\mathscr{P}\nsim\mathbb{n}$, on calcule donc simplement l'issue de $\mathscr{P}+\mathbb{n}$.

Informatiquement, on peut représenter $\mathscr{P}+\mathbb{n}$ par un *couple* $(\mathscr{P}, \mathbb{n})$. Dans ce cas, on appelle $\mathscr{P}$ la *partie position* du couple, et $\mathbb{n}$ la *partie nimber*. Les options d'un tel couple $(\mathscr{P}, \mathbb{n})$ sont de deux sortes :

* celles de la partie position, du type $(\mathscr{P}^i, \mathbb{n})$, où $\mathscr{P}^i$ est une option de $\mathscr{P}$.
* celles de la partie nimber, du type $(\mathscr{P}, \mathbb{i})$ avec $\mathbb{i}<\mathbb{n}$.

Le point-clef dans le cas d'un jeu découpable est de tester si $\mathscr{P}$ est découpable ou non avant de calculer $(\mathscr{P}, \mathbb{n})$ récursivement. Si $\mathscr{P}$ est découpable, on utilise l'algorithme 6 présenté plus loin.

On remarquera que si $\mathbb{n}=\mathbb{0}$ et si $\mathscr{P}$ n'est pas découpable, cet algorithme est simplement l'algorithme classique de calcul de l'issue de $\mathscr{P}$.

Algorithme 4 Calcul récursif de l'issue de $(\mathscr{P}, \mathbb{n})$

1: **Si** $\mathscr{P}$ est découpable sous la forme $\mathscr{P}_1 + ... + \mathscr{P}_p$ **alors**
2: calculer l'issue de $(\mathscr{P}_1 + .. + \mathscr{P}_p, \mathbb{n})$ avec l'algorithme 6
3: **sinon**
4: **Pour chaque** option $(\mathscr{P}^i, \mathbb{n})$ de la partie position et $(\mathscr{P}, \mathbb{i})$ de la partie nimber **faire**
5: calculer l'issue de l'option, et si celle-ci est perdante, renvoyer gagnant
6: **fin Pour**
7: renvoyer perdant (car toutes les options sont gagnantes)
8: **fin Si**

3.6.2 Algorithme pas à pas de calcul du nimber

Le théorème pas à pas (théorème 7) a montré que l'on pouvait calculer le nimber d'une position en essayant successivement les nimbers par ordre croissant, ce qui se traduit par l'algorithme suivant :

Algorithme 5 Calcul du nimber de $\mathscr{P}$

1: $\mathbb{n} \leftarrow \mathbb{0}$
2: **Tant que** le calcul de l'issue de $(\mathscr{P}, \mathbb{n})$ avec l'algorithme 4 renvoie gagnant **faire**
3: $\mathbb{n} \leftarrow \mathbb{n} + \mathbb{1}$
4: **fin Tant que**
5: renvoyer la valeur finale de $\mathbb{n}$

La valeur renvoyée est bien le nimber puisque la sortie de la boucle se produit lorsque $\mathscr{P} + \mathbb{n}$ est trouvé perdant.

3.6.3 Algorithme de calcul de l'issue d'une somme

La proposition 9 nous fournit l'algorithme pour calculer l'issue d'une somme $\mathscr{P}_1 + \mathscr{P}_2 + ... + \mathscr{P}_p + \mathbb{n}$. On calcule d'abord les nimbers $\mathbb{n}_1, ..., \mathbb{n}_{p-1}$ de $(p-1)$ composantes, puis on calcule si le nimber de la dernière composante est identique ou non à $\mathbb{n}_1 + ... + \mathbb{n}_{p-1} + \mathbb{n}$, c'est-à-dire que l'on calcule l'issue de $\mathscr{P}_p + (\mathbb{n}_1 + ... + \mathbb{n}_{p-1} + \mathbb{n})$.

Algorithme 6 Calcul de l'issue de $\mathscr{P}_1 + \mathscr{P}_2 + ... + \mathscr{P}_p + \mathbb{n}$

1: **Pour** i de 0 à $p-1$ **faire**
2: calculer $\mathbb{n}_i$, le nimber de $\mathscr{P}_i$ avec l'algorithme 5
3: **fin Pour**
4: calculer $\mathbb{n}' = \mathbb{n}_1 + ... + \mathbb{n}_{p-1} + \mathbb{n}$ avec la Nim-addition
5: calculer l'issue de $(\mathscr{P}_p, \mathbb{n}')$ avec l'algorithme 4 et renvoyer la valeur obtenue

3.6.4 Application pratique

Ces algorithmes peuvent être utilisés :

* pour calculer l'issue d'une position $\mathscr{P}$ d'un jeu découpable, en appliquant l'algorithme 4 au couple $(\mathscr{P}, \mathbb{0})$.
* pour calculer le nimber d'une position $\mathscr{P}$ d'un jeu, découpable ou non, en appliquant l'algorithme 5 à $\mathscr{P}$.

Par contre, ces algorithmes n'apportent rien dans le cas du calcul de l'issue d'une position d'un jeu non découpable, puisqu'ils se ramènent alors tout simplement au calcul de l'issue avec la méthode classique de l'algorithme 1.

On remarquera que même dans le cas du calcul du nimber d'une position avec l'algorithme 5, c'est toujours à l'intérieur de l'algorithme 4 que l'on est confronté à des positions découpables, ce qui amène à calculer l'issue des sommes de positions indépendantes, et non pas leur nimber.

Le théorème d'inévitabilité implique que l'utilisation de ces algorithmes relatifs au nimber est au moins aussi efficace que les algorithmes 1 et 2. En pratique, l'utilisation du nimber, tout du moins dans le cadre du Sprouts et du Cram, se révèle incomparablement plus rapide.

Ainsi, l'utilisation de l'algorithme 2 dans [4] a conduit les auteurs de l'article à calculer l'issue de la position de Sprouts à 11 points en 100 000 positions environ. En utilisant le nimber, quelques centaines de positions nous suffisent à mener le même calcul. Même si une partie des améliorations ont d'autres causes (comme une meilleure représentation des positions), la majeure partie de l'écart est due à l'utilisation du nimber.

En effet, tandis que nous pouvons nous contenter, avec les algorithmes 4, 5 et 6, de ne stocker qu'une seule occurrence de chaque position indépendante dans la table de transpositions, les algorithmes 1 et 2 conduisent à très vite saturer la table de transpositions avec des sommes de positions indépendantes.

Par exemple, les positions de Sprouts représentées[2] par les chaînes de caractères `1` ; `22` ; `2A|2A` et `AB|AB` sont des petites positions, chacune de nimber $\mathbb{1}$, qui interviennent très fréquemment dans les calculs de Sprouts. Alors que nos algorithmes de calcul n'ajouteront que 4 entrées relatives à ces positions dans la table de transpositions, les algorithmes 1 et 2 ajouteront toutes les façons de faire une somme de deux ou même plus de ces positions (`1+22` ; `1+2A|2A` ; `1+1+AB|AB`...), sans compter les sommes faisant intervenir d'autres positions que ces quatre (comme `0*4+22`).

3.6.5 Ordre de calcul des options

Ordre dans l'algorithme 4

L'ordre dans lequel les calculs sont réalisés a en général une influence très forte sur le temps d'exécution. Dans l'algorithme 1, il est bien connu que le point-clef, lorsque l'on cherche à démontrer qu'une position est gagnante, est de trouver le plus rapidement possible une option perdante. Idéalement, si l'option perdante est toujours la première à être explorée, cela équivaut à réduire de moitié la profondeur de l'arbre de jeu. Dans le cas des positions perdantes, par contre, l'ordre de parcours des options n'a pas d'importance, puisque de toute façon, il est nécessaire de les calculer toutes.

Ce problème classique apparaît dans l'algorithme 4 de calcul de l'issue de $(\mathscr{P}, \mathbb{n})$, avec la même solution : idéalement, il faut explorer en premier l'option perdante parmi les options de $(\mathscr{P}, \mathbb{n})$, qui sont les options de la partie position $(\mathscr{P}^i, \mathbb{n})$, et les options de la partie nimber $(\mathscr{P}, \mathbb{i})$, avec $\mathbb{i} < \mathbb{n}$.

On peut noter que si le calcul de $(\mathscr{P}, \mathbb{n})$ est réalisé en tant que l'un des pas de l'algorithme pas à pas, alors les options $(\mathscr{P}, \mathbb{i})$ avec $\mathbb{i} < \mathbb{n}$ ont déjà toutes été démontrées gagnantes, et que le problème ne concerne donc que l'ordre de calcul des options $(\mathscr{P}^i, \mathbb{n})$. Mais dans le cas général, il faut considérer l'ensemble des options, aussi bien du type $(\mathscr{P}^i, \mathbb{n})$ que $(\mathscr{P}, \mathbb{i})$ avec $\mathbb{i} < \mathbb{n}$. Il n'y a pas de règle simple pour savoir quelle est l'option qui a le plus de chances d'être perdante, car tous les cas sont possibles, comme le montrent les exemples ci-dessous.

Exemples

Premier exemple : une position $\mathscr{P}$ a deux options, $\mathscr{P}^1$ et $\mathscr{P}^2$, de nimbers respectifs $\mathbb{0}$ et $\mathbb{5}$, et l'on cherche à calculer l'issue de $\mathscr{P} + \mathbb{2} = \{\mathscr{P}^1 + \mathbb{2}, \mathscr{P}^2 + \mathbb{2}, \mathscr{P} + \mathbb{0}, \mathscr{P} + \mathbb{1}\}$. Dans ce cas, $\mathscr{P}$ est de nimber $\mathbb{1}$, l'issue de $\mathscr{P} + \mathbb{2}$ est gagnante, et la seule option perdante est $\mathscr{P} + \mathbb{1}$.

2. Le `1` et le `2` utilisés dans la représentation du Sprouts ne doivent pas être confondus avec les nimbers $\mathbb{1}$ et $\mathbb{2}$, avec lesquels ils n'ont aucun rapport. Leur signification est expliquée dans le chapitre 9.

Deuxième exemple : une position $\mathscr{P}$ a trois options, $\mathscr{P}^1$, $\mathscr{P}^2$ et $\mathscr{P}^3$, de nimbers respectifs $\mathbb{0}$, $\mathbb{1}$ et $\mathbb{2}$, et l'on cherche là encore à calculer l'issue de $\mathscr{P}+\mathbb{2} = \{\mathscr{P}^1+\mathbb{2}, \mathscr{P}^2+\mathbb{2}, \mathscr{P}^3+\mathbb{2}, \mathscr{P}+\mathbb{0}, \mathscr{P}+\mathbb{1}\}$. Cette fois, $\mathscr{P}$ est de nimber $\mathbb{3}$, l'issue de $\mathscr{P}+\mathbb{2}$ est gagnante, et la seule option perdante est $\mathscr{P}^3+\mathbb{2}$.

Ces deux exemples montrent qu'il est possible que l'option que l'on doive calculer en premier soit une option de la partie position ($\mathscr{P}^i$, $\mathbb{n}$), ou de la partie nimber ($\mathscr{P}$, $\mathbb{i}$), avec $\mathbb{i} < \mathbb{n}$.

Le deuxième exemple montre également que, contrairement à ce qui se passe quand on cherche à calculer un nimber avec l'algorithme 5, lorsque l'on cherche à calculer l'issue de $\mathscr{P}+\mathbb{2}$, calculer d'abord l'issue de $\mathscr{P}+\mathbb{0}$ ou de $\mathscr{P}+\mathbb{1}$ peut s'avérer contre-productif. En effet, si l'on calcule d'abord l'issue de $\mathscr{P}+\mathbb{0}$, cette issue est gagnante, et sa seule option perdante est $\mathscr{P}^1+\mathbb{0}$, or le calcul de cette option est inutile pour prouver que $\mathscr{P}+\mathbb{2}$ est gagnante.

Ordre dans l'algorithme 6

Dans l'algorithme 6, on calcule le nimber de toutes les composantes, sauf une certaine composante $\mathscr{P}_p$, que l'on calcule en dernier, et pour laquelle on calcule seulement l'issue de ($\mathscr{P}_p$, $\mathbb{n}$), où $\mathbb{n}$ est la Nim-addition des nimbers des autres composantes. Là encore, il est nécessaire de faire un choix parmi les composantes $\mathscr{P}_i$, pour déterminer celle que l'on calculera en dernier.

Dans notre implémentation, nous choisissons généralement pour $\mathscr{P}_p$ la composante qui semble la plus difficile, le critère de difficulté choisi étant spécifique au jeu étudié. Ainsi, plutôt que de calculer son nimber, il suffira de calculer si celui-ci est égal ou différent de $\mathbb{n}$. Les règles implémentées sur la difficulté des composantes sont approximatives, mais elles fonctionnent assez bien si les composantes ont des degrés de difficulté nettement différents.

On peut remarquer que ce problème de l'ordre de calcul des composantes n'influence le temps de calcul que si la somme est gagnante, car le calcul de l'issue de ($\mathscr{P}_p$, $\mathbb{n}$) est alors plus facile que celui du nimber de $\mathscr{P}_p$. Dans le cas d'une somme perdante, le calcul de l'issue de ($\mathscr{P}_p$, $\mathbb{n}$) est équivalent au calcul du nimber de $\mathscr{P}_p$, et l'ordre de calcul n'a donc pas d'importance.

3.6.6 Table de transpositions

Comme dans la plupart des calculs classiques, il est important pour le temps d'exécution d'éviter de faire plusieurs fois les mêmes calculs. Une méthode simple pour cela est d'implémenter une *table de transpositions* qui stocke le résultat des calculs déjà effectués. Les informations à stocker pour une position donnée sont soit le nimber de la position, soit une liste de nimbers dont on sait qu'ils ne sont pas ceux de la position. Alternativement, on peut stocker des couples ($\mathscr{P}$, $\mathbb{n}$), en précisant s'il s'agit de couples perdants ou gagnants.

Dans notre implémentation, nous utilisons généralement une table de transpositions qui stocke uniquement les couples ($\mathscr{P}$, $\mathbb{n}$) perdants, c'est-à-dire les positions dont on connaît le nimber. Cela permet de diminuer la quantité d'information stockée, au détriment d'une perte de temps lors des calculs. En effet, si l'on rencontre à nouveau un couple ($\mathscr{P}$, $\mathbb{n}$) gagnant lors d'un calcul, il faut calculer ses options et les rechercher dans la table jusqu'à ce que l'on retrouve l'option perdante, avant de pouvoir conclure que ce couple est gagnant.

3.6.7 Nœuds multiples

Nous avons implémenté la notion de couple en associant un nœud différent de l'arbre de recherche à chaque couple rencontré. De cette manière, ces nouveaux algorithmes basés sur le nimber sont suffisamment similaires à l'algorithme classique de calcul des issues (algorithme

1) pour permettre le recours aux algorithmes usuels de parcours des arbres de recherche (de type depth-first, ou best-first, comme le PN-search), moyennant quelques adaptations.

Il y a cependant un prix à payer pour cet avantage. Le nœud $(\mathscr{P}, \mathbb{n})$ de l'arbre de recherche admet pour fils tous les nœuds de la forme $(\mathscr{P}, \mathbb{i})$, avec $\mathbb{i} < \mathbb{n}$ (si l'on joue un coup dans la partie nimber). Chacun de ces nœuds admet lui-même des nœuds de la forme $(\mathscr{P}, \mathbb{j})$, avec $\mathbb{j} < \mathbb{i}$ (hormis le nœud $(\mathscr{P}, \mathbb{0})$).

Au total, l'arbre de recherche est susceptible de contenir $n+1$ nœuds admettant $\mathscr{P}$ comme partie position, voire 2^n si l'algorithme de parcours n'implémente pas les transpositions. Et de même, l'algorithme 4 est susceptible de calculer $n+1$, voire 2^n fois les options de $\mathscr{P}$.

Il est possible d'éviter ces calculs inutiles, en utilisant un algorithme plus complexe, qui n'associe qu'un unique nœud à chaque position. L'implémentation concrète est cependant difficile, si bien que notre implémentation actuelle correspond pour l'instant à celle de l'algorithme 4.

Le surcoût, dans le cas d'un algorithme depth-first, est juste un surcoût de temps de calcul dû aux calculs multiples des options d'une même position, puisque de toute façon le nombre de nœuds stockés dans l'arbre de recherche est à chaque instant négligeable. Dans le cas d'un algorithme comme le PN-search qui, lui, conserve un arbre de recherche conséquent en mémoire, le surcoût est en temps de calcul comme en mémoire.

Dans les calculs que nous avons menés, ce surcoût n'est a priori pas si important, dans la mesure où la plupart des nimbers rencontrés sont petits. Quoi qu'il en soit, l'amélioration des algorithmes en tenant compte de ces remarques constitue une piste de recherche intéressante.

3.7 Résultats obtenus

3.7.1 Jeu de Sprouts

Nous avons appliqué au jeu de Sprouts les algorithmes décrits dans ce chapitre, ce qui nous a permis de résoudre le jeu jusqu'à 32 points de départ en 2007, puis jusqu'à 44 points de départ en 2011, ainsi que quelques valeurs isolées jusqu'à 53 points de départ. Chacune des issues calculées vérifie la « conjecture du Sprouts » émise dans [4] : la position de départ à p points est perdante si et seulement si $p \equiv 0$, 1 ou 2 mod 6.

Il est intéressant de noter qu'avant l'introduction des algorithmes présentés dans cette thèse, la plus grande valeur connue était le cas $p = 11$, avec plus de 100 000 positions perdantes stockées en fin de calcul [4], alors que nous sommes désormais capables de calculer la même position avec seulement 113 couples perdants. Les algorithmes utilisant le nimber sont particulièrement efficaces dans le cas du Sprouts parce que les découpages sont très nombreux et interviennent rapidement (il peut suffire de 2 coups à partir d'une position de départ pour qu'un découpage intervienne).

Des détails sur les optimisations liées au Sprouts sont donnés dans le chapitre 9, qui concerne la représentation des positions par des chaînes de caractères, et le chapitre 10, qui revient plus généralement sur les différentes étapes qui nous ont menés à la résolution de ce jeu.

3.7.2 Jeu de Cram

Nous avons appliqué les mêmes algorithmes au jeu de Cram, et nous présentons ici les résultats obtenus en version normale.

Il existe une stratégie de symétrie sur les plateaux de taille pair×pair, qui sont perdants (et donc de nimber $\mathbb{0}$), et sur les plateaux de taille pair×impair, qui sont gagnants. Mais cette stratégie ne fonctionne pas sur les plateaux de taille impair×impair, et n'indique rien non plus sur le nimber des plateaux de taille pair×impair (en dehors du fait que ce nimber est non nul).

Dans les tableaux ci-dessous, nous avons indiqué entre parenthèses les résultats qui ne nécessitent pas de calcul grâce à la stratégie de symétrie. Par ailleurs, nous avons indiqué par un « − » les plateaux $n \times m$ avec $n > m$, car la valeur est identique par symétrie à celle du plateau $m \times n$.

À notre connaissance, les meilleurs résultats connus précédemment étaient ceux de Martin Schneider en 2009 [39] que nous avons indiqués par un astérisque dans les tableaux.

3	4	5	6	7	8	9	10
*0	*1	*1	*4	*1	*3	*1	2

11	12	13	14	15	16	17	18
0	1	2	3	1	4	0	1

TABLE 3.1 – Résultats obtenus sur les plateaux de taille $3 \times n$.

	4	5	6	7	8	9
4	(0)	*2	(0)	3	(0)	1
5	−	*0	2	*1	1	W
6	−	−	(0)	> 3	(0)	(W)
7	−	−	−	W	(W)	

TABLE 3.2 – Résultats obtenus sur les plateaux de taille $n \times m$, avec $n \geq 4$ et $m \geq 4$.

Les résultats nouveaux sur les plateaux de taille supérieure à 4 sont donc les nimbers des plateaux 4×7, 4×9, 5×6, et 5×8, et l'issue gagnante des plateaux 5×9 et 7×7. Par ailleurs, l'algorithme 5 a permis de montrer que le nimber du plateau 6×7 est strictement supérieur à 3, mais sans parvenir pour l'instant à calculer la valeur exacte.

Dans le cas du jeu de Cram, nous avons constaté expérimentalement que la moindre augmentation de la taille du plateau crée une forte augmentation de difficulté du calcul. Cela est dû d'une part à l'augmentation exponentielle du nombre de positions possibles avec le nombre de lignes ou de colonnes, et aussi au fait que le découpage en positions indépendantes intervient d'autant plus tard que le plateau est grand.

Les optimisations spécifiques au Cram sont l'objet du chapitre 12.

3.8 Conclusion

Depuis la découverte du théorème de Sprague-Grundy, les nimbers ont été utilisés avec succès pour analyser les jeux impartiaux, en particulier les nombreuses variantes du jeu de Nim, comme par exemple les jeux octaux. Cependant, dans le cas de jeux très complexes à analyser, comme le Sprouts ou le Cram, les nimbers étaient jusqu'ici considérés comme trop coûteux en temps de calcul, et n'étaient pas ou peu utilisés dans les calculs d'issues.

Le théorème présenté dans ce chapitre montre que l'utilisation du nimber dans les jeux impartiaux découpables est en fait inévitable, même dans ce dernier cas, dans la mesure où le calcul classique d'une somme de positions sans utiliser le nimber est systématiquement moins efficace. Les algorithmes qui en découlent ont été appliqués avec succès au Sprouts et au Cram, deux jeux impartiaux découpables, et le nimber est un concept sous-jacent indispensable pour obtenir les records présentés.

Chapitre 4

Jeux impartiaux en version misère

4.1 Introduction

Deux conventions possibles de victoire sont envisageables dans le cas des jeux impartiaux. Dans la version dite *normale*, celui qui ne peut plus jouer a perdu. Cette convention peut être considérée comme naturelle d'un point de vue psychologique : la liberté de mouvement est largement préférable à son impossibilité, aussi bien dans la plupart des jeux que dans la vie réelle.

La convention de jeu inverse, dite *misère* consiste au contraire à déclarer vainqueur celui qui ne peut plus jouer. *Winning Ways* [6] présente cette convention de façon amusante, en imaginant le cas d'un joueur qui souhaite s'assurer de perdre à un jeu en version normale, pour faire plaisir à un adversaire plus faible par exemple.

4.1.1 Algorithme élémentaire commun

La seule différence entre la version normale et la version misère réside dans la convention de victoire des positions terminales, si bien que l'algorithme 7 décrit ci-dessous permet de calculer l'issue d'une position $\mathscr{P}$ dans les deux versions de jeu. La différence entre les versions normale et misère est prise en compte aux lignes 2 et 3 de l'algorithme.

Algorithme 7 Calcul récursif de l'issue de $\mathscr{P}$

```
1:  Si 𝒫 est vide alors
2:      en version normale : renvoyer perdant
3:      en version misère : renvoyer gagnant
4:  sinon
5:      calculer la liste des options de 𝒫
6:      Pour chaque option 𝒫_i faire
7:          calculer l'issue de 𝒫_i, et si celle-ci est perdante, renvoyer gagnant
8:      fin Pour
9:      renvoyer perdant (car toutes les options sont gagnantes)
10: fin Si
```

Cet algorithme élémentaire montre que dans le cas d'un jeu sans propriétés théoriques supplémentaires, la version misère n'est pas plus difficile que la version normale. La difficulté est même tout à fait similaire.

4.1.2 Difficultés de la version misère

L'algorithme 7 n'est cependant pas optimal dès lors que les jeux possèdent des propriétés théoriques permettant d'écrire des algorithmes plus complexes. Le cas qui nous intéresse dans le cadre de cette thèse est bien sûr celui des jeux découpables, présentés dans la section 2.6 du chapitre d'introduction.

Les jeux découpables sont les jeux combinatoires, comme le Sprouts ou le Cram, dont certaines positions sont découpables en composantes indépendantes. On cherche alors à exploiter ces découpages pour réaliser, dans le cas idéal, des calculs indépendants sur chacune des composantes.

Dans le cas de la version normale, nous avons détaillé dans le chapitre 3 comment le concept de nimber permettait d'effectuer des calculs indépendants sur les composantes d'une somme. Nous allons voir dans ce chapitre que, de façon surprenante, l'étude des sommes en version misère est nettement plus difficile qu'en version normale.

Nous illustrons dans les deux paragraphes qui suivent la difficulté de la version misère en montrant que certaines stratégies simples, qui fonctionnent sur les sommes de positions en version normale, sont mises en défaut en version misère.

Inefficacité de la stratégie de symétrie

Dans le cas des jeux impartiaux en version normale, une stratégie de symétrie permet de montrer que pour tout jeu $\mathscr{G}$, la somme $\mathscr{G} + \mathscr{G}$ est d'issue perdante. En effet, le second joueur peut s'assurer de jouer le dernier coup de la partie simplement en copiant tout coup du premier joueur dans l'autre composante.

Cette stratégie ne peut pas fonctionner en version misère. Si le deuxième joueur l'applique, il s'assure bien de jouer le dernier coup, mais en version misère, cela signifie uniquement qu'il a perdu, ce qui ne l'intéresse évidemment pas !

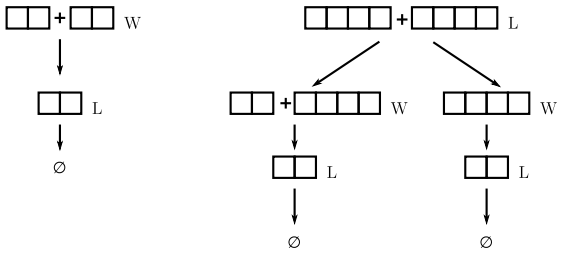


FIGURE 4.1 – Arbres solutions de positions de Cram.

La figure 4.1 montre qu'en fait une somme $\mathscr{G} + \mathscr{G}$ peut être gagnante ou perdante en version misère. L'arbre de gauche est un arbre solution prouvant que la position de Cram somme de deux plateaux de taille 1×2 est gagnante, et l'arbre de droite prouve que la somme de deux plateaux de taille 1×4 est perdante.

Inefficacité de la simplification des composantes perdantes

Un autre résultat immédiat de la théorie des jeux impartiaux en version normale est la possibilité de supprimer les composantes perdantes dans les sommes. Si $\mathscr{G}$ est d'issue perdante, alors $\mathscr{G}+\mathscr{H}$ a la même issue que $\mathscr{H}$. Ce résultat s'obtient en observant que lorsqu'un joueur joue dans $\mathscr{G}$, son adversaire lui répond dans $\mathscr{G}$ de façon à retrouver une position perdante. Le point-clef réside dans la condition d'arrêt : si un joueur s'entête à jouer dans $\mathscr{G}$ (parce qu'il ne veut pas jouer dans $\mathscr{H}$), son adversaire va finir par jouer le dernier coup dans cette composante, et le joueur qui ne voulait pas jouer dans $\mathscr{H}$ y sera forcé quand même.

On voit qu'en version normale, c'est cette propriété de conservation de la parité du nombre de coups qui est à l'origine de toutes les simplifications : les joueurs ne peuvent pas utiliser les composantes perdantes pour modifier celui qui a le trait, ce qui rend les composantes perdantes sans influence sur l'issue d'une somme.

En version misère, il n'y a pas de conservation de la parité, ce qui empêche la simplification des composantes perdantes. Considérons une somme $\mathscr{G}+\mathscr{H}$ en version misère avec $\mathscr{G}$ perdante. Le fait que $\mathscr{G}$ soit perdante en version misère signifie que celui qui commence à jouer dans cette composante peut s'assurer d'y jouer le dernier coup. Contrairement à la version normale, le joueur qui a le trait peut donc utiliser $\mathscr{G}$ pour « passer son tour » (inverser la parité). Cela peut lui permettre de ne pas jouer en premier dans $\mathscr{H}$, et donc $\mathscr{G}+\mathscr{H}$ est un jeu essentiellement différent de $\mathscr{H}$.

En fait, il n'y a aucune règle simple dans le cas de la version misère. Par exemple, en version normale, la somme de deux positions perdantes est perdante à cause de la propriété que nous venons de décrire, alors qu'en version misère, elle peut être gagnante ou perdante.

4.1.3 Présentation de notre travail

La notion qui, en version misère, sera amenée à jouer le rôle que joue le nimber en version normale — remplacer les différentes composantes d'une somme de positions indépendantes par l'objet le plus simple possible — est la notion d'*arbre canonique réduit*.

Nous présentons dans ce chapitre la théorie relative à cette notion, des détails sur son implémentation, et les résultats qu'elle a permis d'obtenir sur les jeux de Sprouts et de Cram.

4.2 Arbres canoniques réduits

Dans cette section, nous allons rappeler quelques résultats usuels de l'étude des jeux en version misère, sans chercher systématiquement à fournir les démonstrations. Un bon point de départ bibliographique pour cette théorie est *On Numbers And Games* [12], le livre de Conway (chapitre 12, p. 136–152), ou bien le fameux *Winning Ways* [6] de Berlekamp, Conway et Guy (chapitre 13, p. 413–452).

4.2.1 Indistinguabilité

Nous avons déjà présenté la notion d'indistinguabilité dans le chapitre sur la théorie des jeux combinatoires, au paragraphe 2.6.3.

Suivant [33], nous dirons que deux jeux $\mathscr{G}$ et $\mathscr{H}$ sont *indistinguables*, et l'on notera $\mathscr{G} \sim \mathscr{H}$, si quel que soit le jeu $\mathscr{T}$, les sommes $\mathscr{G}+\mathscr{T}$ et $\mathscr{H}+\mathscr{T}$ ont la même issue. Cette notion a un intérêt pratique : si l'on sait que deux positions d'un jeu donné sont indistinguables, on peut remplacer la plus compliquée par la plus simple à chaque fois qu'on la rencontre, ce qui permet d'accélérer le calcul.

Cette notion dépend du jeu considéré, mais aussi de la version du jeu : deux positions peuvent être indistinguables en version normale, mais pas en version misère. Par exemple, pour le Sprouts, les positions de départ à 1 et 2 points S_1 et S_2 sont indistinguables dans la version normale (on pourra noter $S_1 \sim_+ S_2$). Mais elles ne le sont pas dans la version misère ($S_1 \not\sim_- S_2$) : il suffit de prendre pour $\mathscr{T}$ la position vide pour les distinguer, car en misère, S_1 est gagnante et S_2 est perdante.

4.2.2 Indistinguabilité en version normale

L'indistinguabilité est une relation d'équivalence, et les classes d'équivalence correspondantes sont appelées *classes d'indistinguabilité*. En version normale, il s'avère que les classes

d'indistinguabilité sont particulièrement simples et peu nombreuses, comme l'affirme le théorème de Sprague-Grundy :

Théorème 9. *(de Sprague-Grundy) Étant donné un jeu combinatoire impartial, toute position de ce jeu est indistinguable d'une certaine colonne du jeu de Nim, appelée nimber de la position.*

Dans le cadre du Sprouts, on peut observer par exemple : $S_2 \sim_+ \mathbb{0}$, ou encore : `ABCD|AB|CD` $\sim_+ \mathbb{3}$, c'est-à-dire que le nimber de S_2 est $\mathbb{0}$ et celui de `ABCD|AB|CD` est $\mathbb{3}$ (voir le chapitre 9 pour la notation des positions de Sprouts).

Pour les jeux en version misère, malheureusement, ce théorème ne fonctionne pas. Les classes d'indistinguabilité sont bien plus nombreuses, et John Conway démontre dans [12] que ce qui joue le rôle du nimber dans le cas des jeux misère est le concept, décrit ci-après, d'*arbre canonique réduit.*

4.2.3 Arbres canoniques

Nous avons déjà défini dans le chapitre 2 les notions d'arbre de jeu (§2.4.2) et d'arbre canonique (§2.5). Rappelons que l'arbre canonique s'obtient à partir de l'arbre de jeu en supprimant l'ensemble des branches redondantes, car celles-ci n'influencent pas les possibilités de jeu des joueurs.

La figure 4.2 ci-dessous reprend l'exemple de la figure 2.5 et montre la canonisation de l'arbre de jeu de la position de Sprouts `0.AB|AB` (cette position s'obtient à partir de S_2, en reliant un point à lui-même), obtenu avec notre programme.

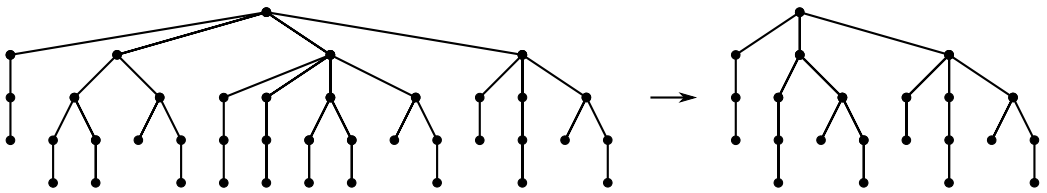


FIGURE 4.2 – Canonisation d'un arbre de jeu

Cette notion d'arbre canonique permet de conserver la quantité d'information nécessaire et suffisante pour décrire une position : si deux positions ont le même arbre canonique, alors elles sont indistinguables, que ce soit en version normale ou misère.

Cependant, l'élagage des branches redondantes n'est pas le seul possible. Il est en fait possible de supprimer d'autres branches de l'arbre tout en conservant la propriété essentielle que l'arbre de jeu résultant est indistinguable de l'arbre de jeu initial.

4.2.4 Coups réversibles

Coups réversibles

Définition 12. *Soit $\mathscr{G} = \{\mathscr{G}_1, \mathscr{G}_2, \mathscr{G}_3, ...\}$ un arbre canonique non vide.*

On dit qu'un arbre canonique $\mathscr{H}$ s'obtient à partir de $\mathscr{G}$ en ajoutant des coups réversibles lorsque $\mathscr{H} = \{\mathscr{G}_1, \mathscr{G}_2, \mathscr{G}_3, ..., \mathscr{R}_1, \mathscr{R}_2, \mathscr{R}_3, ...\}$ et que chaque arbre $\mathscr{R}_j$ a un fils qui est $\mathscr{G}$ (les coups $\mathscr{R}_j$ sont dits réversibles*).*

$\mathscr{G}$ et $\mathscr{H}$ ont la même issue : si $\mathscr{G}$ est gagnant, c'est qu'un des $\mathscr{G}_i$ est perdant. $\mathscr{H}$ ayant ce $\mathscr{G}_i$ comme fils, il est donc également gagnant. Inversement, si $\mathscr{G}$ est perdant, c'est que chaque $\mathscr{G}_i$ est gagnant. De plus, chaque $\mathscr{R}_i$ est gagnant, car chacun a $\mathscr{G}$ comme fils. Donc $\mathscr{H}$ est également perdant.

Mais surtout, $\mathscr{G}$ et $\mathscr{H}$ sont indistinguables en version misère. En effet, si l'un des joueurs dispose d'une stratégie gagnante pour la somme $\mathscr{G}+\mathscr{T}$, alors, pour jouer $\mathscr{H}+\mathscr{T}$, il lui suffit de jouer les mêmes coups, hormis dans le cas suivant :

* si à un moment donné, l'autre joueur joue un coup du type $\mathscr{R}_j$, alors il faut répondre en jouant $\mathscr{G}$ pour cette composante du jeu (d'où le nom de *coups réversibles*).

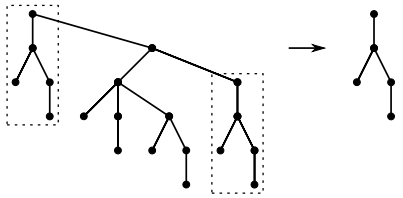


FIGURE 4.3 – Exemple de coup réversible

La figure 4.3 présente un exemple de coup réversible : $\mathscr{H}$ est l'arbre à gauche de la flèche, $\mathscr{G}$ celui à droite de la flèche. Il existe un coup réversible $\mathscr{R}_1$, à partir duquel on peut jouer deux coups, l'un des deux étant $\mathscr{G}$.

Cas particulier de l'arbre vide

Si $\mathscr{G}$ est l'arbre vide, la définition est similaire, mais il faut rajouter une clause pour pouvoir affirmer que $\mathscr{G}$ et $\mathscr{H}$ sont indistinguables en version misère : il faut que $\mathscr{H}$ soit gagnant dans la version misère.

En effet, lorsque les joueurs jouent $\mathscr{H}+\mathscr{T}$ comme décrit ci-dessus, un cas particulier supplémentaire se présente quand $\mathscr{T}$ est terminé avant d'avoir joué le moindre coup dans $\mathscr{H}$. Si ce cas se présente lorsque $\mathscr{G}$ est non vide, la fin de partie repose sur l'explication donnée ci-dessus du fait que $\mathscr{G}$ et $\mathscr{H}$ ont la même issue.

Mais si $\mathscr{G}$ est vide, le jeu sur $\mathscr{G}+\mathscr{T}$ est supposé être terminé, et le joueur dont c'est le tour devrait avoir gagné. Or, il se retrouve obligé de jouer l'un des coups réversibles $\mathscr{R}_j$ et donc, pour lui assurer de pouvoir gagner dans ce cas particulier, il faut que $\mathscr{H}$ soit gagnant dans la version misère. La clause impose en fait que là encore, $\mathscr{G}$ et $\mathscr{H}$ aient la même issue.

4.2.5 Arbres canoniques réduits

Réducteurs

Nous avons vu que si $\mathscr{H}$ s'obtient à partir de $\mathscr{G}$ en ajoutant des coups réversibles, alors $\mathscr{G}$ et $\mathscr{H}$ sont indistinguables en version misère, mais ce qui nous intéresse est plutôt la démarche inverse : partant d'un arbre $\mathscr{H}$, on cherche à le simplifier en le ramenant à un certain arbre $\mathscr{G}\subset\mathscr{H}$, obtenu en ôtant des coups réversibles. Un tel arbre $\mathscr{G}$ sera appelé *réducteur* de $\mathscr{H}$.

Remarquons tout d'abord que $\mathscr{G}$ n'est pas forcément unique, ce qui pose a priori un problème de choix lorsque plusieurs réducteurs différents sont possibles. Cependant, les coups réversibles $\mathscr{R}_1,\mathscr{R}_2,\mathscr{R}_3,...$ doivent contenir $\mathscr{G}$, et donc avoir une hauteur au moins égale à celle de $\mathscr{G}$ plus 1. On en déduit immédiatement que :

* Un réducteur est toujours de la forme : {fils de $\mathscr{H}$ d'une hauteur inférieure à une certaine valeur}.
* Deux réducteurs sont toujours comparables pour l'inclusion. S'il existe deux réducteurs différents, alors le plus petit est inclus dans le plus grand, et même, c'est un réducteur du plus grand.
* Il existe un (unique) plus petit réducteur.

Arbres canoniques réduits

On peut alors définir *l'arbre canonique réduit* à partir de l'arbre canonique, en supprimant toutes les possibilités de coups réversibles qu'il contient. On détermine cet arbre canonique réduit récursivement :

* calcul de l'arbre canonique réduit de chacun des fils.
* suppression des doublons parmi les fils canonisés et réduits.
* réduction de l'arbre obtenu en utilisant le plus petit réducteur possible.

4.2.6 Indistinguabilité en version misère

L'intérêt de cette notion est que si deux positions ont le même arbre canonique réduit, alors elles sont indistinguables dans la version misère. Il paraît alors logique de s'intéresser à la réciproque. Cette réciproque est valide pour un jeu combinatoire en version normale : on sait que si deux positions n'ont pas le même nimber, alors elles sont distinguables. En effet, si $\mathscr{P}_1$ et $\mathscr{P}_2$ n'ont pas le même nimber, alors $\mathscr{P}_1$ les distingue, car $\mathscr{P}_1 + \mathscr{P}_1$ est perdante, et $\mathscr{P}_2 + \mathscr{P}_1$ est gagnante.

Pour le cas d'un jeu misère, un résultat démontré dans [12] (p. 149) semble à première vue régler la question : étant donné deux arbres canoniques réduits $\mathscr{G}$ et $\mathscr{H}$ différents, il existe un arbre canonique réduit $\mathscr{T}$ tel que $\mathscr{G} + \mathscr{T}$ et $\mathscr{H} + \mathscr{T}$ n'aient pas la même issue en version misère, et donc $\mathscr{T}$ permet de distinguer $\mathscr{G}$ et $\mathscr{H}$.

On pourrait donc croire qu'en version misère, les classes d'indistinguabilité correspondent exactement aux arbres canoniques réduits. Cependant, lorsque l'on se place dans le cadre d'un jeu particulier, il est possible que des positions qui n'ont pas le même arbre canonique réduit soient tout de même indistinguables. Nous allons donner un exemple.

On considère le jeu de Nim, restreint aux colonnes de taille ≤ 2. En utilisant le fait que $\mathbb{1} + \mathbb{1} \sim_- \mathbb{0}$, on obtient que les classes d'indistinguabilité sont du type $n \times 2$ ou $n \times 2 + \mathbb{1}$ ($n \geq 0$). Les coups que l'on peut jouer à partir de ces positions sont faciles à décrire :

* à partir de $n \times 2$ ($n \geq 1$), on peut jouer $(n-1) \times 2 + \mathbb{1}$ ou $(n-1) \times 2$.
* à partir de $n \times 2 + \mathbb{1}$ ($n \geq 1$), on peut jouer $n \times 2$, $(n-1) \times 2 + \mathbb{1}$ ou $(n-1) \times 2$.

Ceci nous permet de déterminer récursivement que les seules positions perdantes sont $\mathbb{1}$, et $2n \times 2$ ($n \geq 1$), puis que les seules classes d'indistinguabilité sont $\mathbb{0}$; $\mathbb{1}$; 2 ; $2 + \mathbb{1}$; $2 + 2$; $2 + 2 + \mathbb{1}$. En effet, dès qu'une position comporte au moins 3 fois 2, enlever une paire de 2 ne change pas l'issue de cette position.

Ainsi, même si les arbres canoniques de 2 et $2 + 2 + 2$ sont différents, comme aucun arbre canonique apparaissant dans le cadre ce jeu ne permet de les distinguer, ils sont indistinguables [1]. En fait, ce phénomène peut se produire dès lors que tous les arbres canoniques réduits n'apparaissent pas dans le déroulement du jeu. Les nouvelles classes d'indistinguabilité, plus grandes et donc moins nombreuses, sont à la base des travaux de Thane Plambeck (voir par exemple [33]) sur les *quotients misère*.

Malheureusement, cette théorie semble difficilement applicable dans le cas du Sprouts ou du Cram, où une grande variété d'arbres canoniques réduits apparaît. Nous nous sommes donc restreints à l'étude des arbres canoniques réduits dans nos travaux, même si la détermination de classes d'indistinguabilité moins nombreuses reste toutefois envisageable.

Pour en revenir à l'exemple, 2 et $2 + 2 + 2$ sont par contre distinguables dans le cadre du Sprouts : la position `ABCD|ABEF|CDFE`, dont l'arbre canonique réduit est $\{\mathbb{1}; \{2\}\}$, permet de les distinguer, car $2 + \{\mathbb{1}; \{2\}\}$ est gagnante, et $2 + 2 + 2 + \{\mathbb{1}; \{2\}\}$ est perdante.

1. Cet exemple est expliqué plus en détail dans [41].

4.2.7 Dénombrement

Il est intéressant de dénombrer les arbres canoniques et les arbres canoniques réduits, ce qui permet de se faire une première idée de l'intérêt pratique de ces notions.

Pour commencer, on peut dénombrer le nombre exact d'arbres canoniques d'une certaine hauteur, comme sur la figure 4.4, qui montre les 16 arbres canoniques de hauteur ≤ 3.

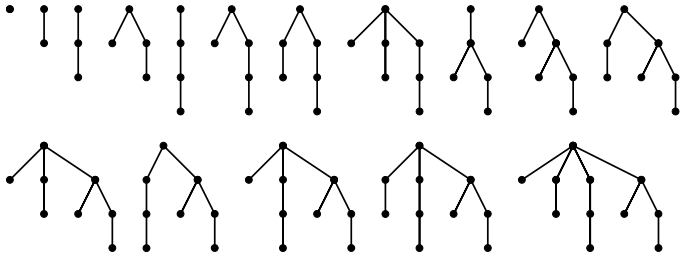


FIGURE 4.4 – Arbres canoniques de hauteur ≤ 3

Proposition 10. *Il y a* $2^{(2^{(\ldots^{(2^0)})})}$ *(avec* $h+1$ *fois le nombre 2) arbres canoniques de hauteur* $\leq h$.

Cette relation se démontre simplement par récurrence. Si l'on note $\mathscr{C}_h$ l'ensemble des arbres canoniques de hauteur $\leq h$ et c_h son cardinal, le dénombrement ci-dessus des arbres canoniques peut se réécrire $c_{h+1} = 2^{c_h}$. Cette relation découle directement du fait qu'un arbre canonique de hauteur $\leq h+1$ se définit comme l'ensemble des arbres canoniques de ses enfants, qui sont de hauteur $\leq h$. L'ensemble $\mathscr{C}_{h+1}$ des arbres canoniques de hauteur $\leq h+1$ est donc en bijection avec $\mathscr{P}(\mathscr{C}_h)$, l'ensemble des parties de $\mathscr{C}_h$.

Le nombre d'arbres canoniques de hauteur $\leq n$ vaut donc, pour n croissant : 1 ; 2 ; 4 ; 16 ; 65 536 ; $2^{65\,536}$... À comparer avec le nombre de nimbers correspondant à des arbres de hauteur $\leq n$: 1 ; 2 ; 3 ; 4 ; 5 ; 6...

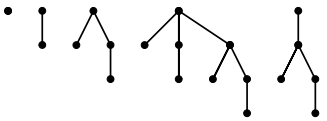


FIGURE 4.5 – Arbres canoniques réduits de hauteur ≤ 3 : $\mathbb{0}$; $\mathbb{1}$; 2 ; $\mathbb{3}$ et $\{2\}$

Quand on regarde le nombre d'arbres canoniques réduits, on observe une croissance malheureusement plus proche du premier cas : 1 ;2 ;3 ;5 ;22 ;4 171 780... (voir [23]). En fait, il y a beaucoup de simplifications pour les petites valeurs, mais très vite, il n'y en a plus qu'un nombre négligeable et l'on retrouve une croissance du type $c_{n+1} = 2^{c_n}$. Par exemple, $2^{22} = 4\,194\,304$ est très proche de 4 171 780 (et le terme suivant vaut plus de 99,99% de $2^{4\,171\,780}$, la formule exacte étant disponible dans [12] p. 140).

4.2.8 Colonnes de Nim

Les arbres canoniques réduits correspondant aux positions de Sprouts, utilisés dans la version misère, sont donc en général bien plus nombreux et compliqués que les nimbers de la version normale. Néanmoins, les positions presque terminales ont souvent un arbre canonique réduit très simple, à savoir celui d'une colonne de Nim. Nous allons expliquer quelles sont les propriétés à l'origine de cette situation.

Définition 13. *Le* mex *(minimum exclu) d'un ensemble de nombre entiers naturels est défini comme le plus petit nombre entier naturel n'appartenant pas à cet ensemble.*

Par exemple, $\mathrm{mex}(0;1;4;3;1;7) = 2$.

Les arbres canoniques correspondants à des colonnes de Nim ne peuvent pas se simplifier avec des coups réversibles. Par contre, une position dont les fils sont de tels arbres est fréquemment simplifiable (ce résultat, ainsi que le suivant, sont démontrés dans [12] p. 139) :

Théorème 10. *Une position dont tous les fils sont des colonnes de Nim est elle-même indistinguable, en version misère, d'une colonne de Nim, à moins que chacun des fils comporte deux allumettes ou plus. Pour trouver la colonne dont elle est indistinguable, on calcule le* mex *des fils.*

Par exemple, $\mathscr{P}_1 = \{\mathbb{0};\mathbb{1};\mathbb{3};\mathbb{5}\}$ (une position dont les fils sont des colonnes de Nim à 0 ;1 ;3 ;5 allumettes) est indistinguable d'une colonne de Nim à 2 allumettes : $\mathscr{P}_1 = \{\mathbb{0};\mathbb{1};\mathbb{3};\mathbb{5}\} \sim_- \mathbb{2}$.

Par contre, $\mathscr{P}_2 = \{\mathbb{2};\mathbb{3}\}$ (une position dont les fils sont des colonnes de Nim à 2 et 3 allumettes) ne peut pas se simplifier.

Par ailleurs, les colonnes de Nim possèdent une deuxième propriété intéressante, qui exprime qu'une position somme de colonnes de Nim est elle aussi parfois simplifiable :

Théorème 11. *Si au moins un des deux nombres m, n est égal à 0 ou à 1, alors :* $\mathbb{m}+\mathbb{n} = \mathbb{q}$, *où $q = m \oplus n$.*

« $\oplus$ » décrit la *Nim-addition*, c'est-à-dire que l'on effectue le *ou exclusif bit à bit* des deux nombres. Par exemple, $\mathbb{3} + \mathbb{1} \sim \mathbb{2}$, ou $\mathbb{4} + \mathbb{1} \sim \mathbb{5}$.

Là encore, il y a un cas problématique : lorsque m et n sont ≥ 2. Dans ce cas, $\mathbb{m}+\mathbb{n}$ n'est pas indistinguable d'une colonne de Nim. Par exemple, $\mathbb{2} + \mathbb{2} = \{\mathbb{2} + \mathbb{0}; \mathbb{2} + \mathbb{1}\} = \{\mathbb{2};\mathbb{3}\}$, et l'on a déjà vu ci-dessus que cette position ne pouvait pas se simplifier.

Citons en exemple le calcul de l'arbre de jeu de la position de Sprouts S_3. Il y a 55 arbres canoniques différents dans les positions issues de cette position de départ. Dans le lot, seules 2 d'entre elles ne sont pas réductibles à une colonne de Nim :

* S_2 =`0*2`, qui a deux fils, tous les deux indistinguables de 2. Son arbre canonique réduit est donc $\{2\}$.
* `0*2.AB|AB`, qui est « contaminée » par S_2 quand on remonte dans l'arbre : son arbre canonique réduit est $\{\mathbb{1};\{\mathbb{2}\}\}$.

Les 53 autres positions se ramènent à des colonnes de Nim, ce qui montre l'importance de cette notion dans l'analyse des petites positions du Sprouts misère.

4.2.9 Rétablissement grâce aux coups réversibles

Nous avons vu dans le paragraphe précédent que la position `0*2.AB|AB` était contaminée par l'un de ses fils. Mais il arrive parfois qu'en remontant l'arbre, certaines positions ne le soient pas, c'est-à-dire qu'elles sont indistinguables de colonnes de Nim, même si ce n'est pas le cas de certaines positions de leur descendance.

C'est le cas par exemple de S_3 : elle a deux fils dont l'arbre canonique réduit est $\mathbb{0}$, et un autre fils qui est `0*2.AB|AB`. Elle est donc indistinguable de $\{\mathbb{0};\{\mathbb{1};\{\mathbb{2}\}\}\}$. Or ceci est réductible à $\mathbb{1}$, car s'obtient à partir de $\mathbb{1}$ en rajoutant le coup réversible $\{\mathbb{1};\{\mathbb{2}\}\}$. Donc S_3 est indistinguable d'une colonne de Nim, alors même que ce n'était pas le cas de l'un de ses fils.

Cette propriété de rétablissement grâce aux coups réversibles augmente le nombre de colonnes de Nim dans les positions presque terminales. Elle n'est bien sûr pas limitée aux colonnes de Nim, et peut se produire avec des arbres canoniques plus compliqués. C'est le cas par exemple de l'arbre de la figure 4.3, qui correspond à la position de Sprouts `0*2.A|1A`.

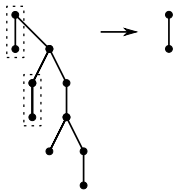


FIGURE 4.6 – Arbre canonique réduit de la position de départ à 3 points

4.2.10 Factorisation par $\mathbb{1}$

Certains arbres canoniques réduits peuvent s'écrire sous la forme : $\mathscr{G} + \mathbb{1}$. L'intérêt est que lorsque l'on considère une somme de tels arbres, on peut réduire la taille des arbres considérés grâce à la propriété $\mathbb{1} + \mathbb{1} \sim \mathbb{0}$.

Par exemple, en utilisant que $\mathbb{3} \sim \mathbb{2} + \mathbb{1}$ et que $\{\mathbb{3}; \{\mathbb{2}\}\} \sim \mathbb{1} + \{\mathbb{2}\}$, on obtient : $\mathbb{3} + \{\mathbb{3}; \{\mathbb{2}\}\} \sim \mathbb{2} + \mathbb{1} + \mathbb{1} + \{\mathbb{2}\} \sim \mathbb{2} + \{\mathbb{2}\}$, et l'on est passé de la somme de deux arbres de hauteur 3 et 4, à la somme de deux arbres de hauteur 2 et 3.

D'autres simplifications conduisant à une diminution de la taille des arbres existent. Conway observe par exemple que $\{\mathbb{0}; \{\mathbb{2}\}; \{\mathbb{3}; \{\mathbb{2}\}\}\} + \mathbb{2} \sim \{\mathbb{2}\}$ dans [12] p. 151. Mais ces simplifications semblent trop rares pour être utiles, et nous détaillerons dans le paragraphe 4.4.8 pourquoi, dans notre programme, nous avons uniquement implémenté la simplification : $\mathbb{1} + \mathbb{1} \sim \mathbb{0}$.

4.3 Calcul des arbres canoniques réduits

4.3.1 Représentation et stockage

Représentation en chaîne

La façon intuitive de représenter les ACR[2] est de définir récursivement des chaînes de caractères, chaque ACR étant représenté par la liste de ses fils. Par exemple, la position de départ à 4 points serait représentée par la chaîne $\{\mathbb{3}; \{\mathbb{1}; \mathbb{2}; \{\mathbb{3}; \{\mathbb{2}\}\}\}\}$[3].

Mais cette représentation en chaîne devient rapidement inutilisable quand les ACR deviennent grands : si le même ACR apparaît en plusieurs endroits d'un ACR plus gros, son information est dupliquée autant de fois qu'il apparaît, alors qu'il est évident qu'il suffirait de stocker une seule fois cette information. Ce défaut est encore plus important dès lors que l'on stocke de multiples ACR dans la base de données.

Représentation par liens

Pour contourner le problème de la redondance des représentations en chaînes, nous avons implémenté une représentation par liens, en affectant à chaque ACR un identifiant unique (un nombre). Un ACR reste représenté par la liste de ses fils, mais cette fois-ci, nous ne stockons que la liste de leurs identifiants, au lieu de la liste de leurs représentations complètes. Cela permet de ne stocker qu'une seule fois un ACR donné dans la base et ensuite d'y faire référence plusieurs fois à travers son identifiant.

Cependant, lors des calculs utilisant les ACR, nous avons fréquemment besoin de certaines informations à propos d'un ACR donné : la hauteur de l'arbre, ou son issue (gagnante ou perdante). Il est bien sûr possible de les calculer récursivement, mais cela est coûteux en

2. Dans la suite de ce chapitre, *arbre canonique réduit* sera abrégé en *ACR*.

3. c'est une forme compacte. Avec une représentation en chaîne même pour les colonnes de Nim, il faudrait remplacer $\mathbb{0}$ par {}, $\mathbb{1}$ par {{}}, $\mathbb{2}$ par {{};{{}}} et $\mathbb{3}$ par {{};{{}};{{};{{}}}}

temps de calcul car la manipulation de la représentation par liens nécessite des recherches d'identifiants dans une base de données.

Nous avons donc choisi une représentation par liens qui contient les principales informations dont nous avons besoin à propos d'un ACR. L'identifiant d'un ACR est composé de trois paramètres :

* la hauteur de l'ACR.
* un numéro unique qui permet de différencier les ACR de même hauteur.
* le caractère « `W` » ou « `L` » selon l'issue de l'ACR (en version misère).

Par convention, le numéro vaut `0` si l'ACR est une colonne de Nim. Sinon, on numérote `1` le premier ACR d'une hauteur donnée rencontré, `2` le deuxième...

Le caractère qui décrit l'issue de l'ACR sert lors de la phase de réduction, car un arbre n'est réductible à $\mathbb{0}$ que s'il est gagnant en version misère (cf paragraphe 4.3.2).

Nous allons maintenant donner un exemple avec l'ACR de la figure 4.7, qui est celui de la position de Sprouts `1ABC|BCDE|ADE`. La représentation usuelle de cet ACR est $\{\mathbb{0};2;\{\mathbb{3}\};\{\mathbb{1};\mathbb{3};\{2\}\}\}$.

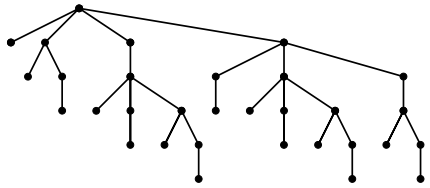


FIGURE 4.7 – Arbre canonique réduit de hauteur 5

Dans le tableau 4.1, pour chacun des différents sous-arbres de cet ACR, nous donnons son identifiant, ainsi que la liste des identifiants de ses enfants.

ACR	identifiant	liste des enfants
$\mathbb{0}$	`0-0-W`	-
$\mathbb{1}$	`1-0-L`	`0-0-W`
2	`2-0-W`	`0-0-W 1-0-L`
$\mathbb{3}$	`3-0-W`	`0-0-W 1-0-L 2-0-W`
$\{2\}$	`3-1-L`	`2-0-W`
$\{\mathbb{3}\}$	`4-1-L`	`3-0-W`
$\{\mathbb{1};\mathbb{3};\{2\}\}$	`4-2-W`	`1-0-L 3-0-W 3-1-L`
$\{\mathbb{0};2;\{\mathbb{3}\};\{\mathbb{1};\mathbb{3};\{2\}\}\}$	`5-1-W`	`0-0-W 2-0-W 4-1-L 4-2-W`

TABLE 4.1 – Exemple d'identifiants représentant des ACR.

Les ACR rencontrés lors des calculs sont stockés dans une base de données semblable aux deux dernières colonnes du tableau, sous la forme de couples (ACR ; liste des enfants de l'ACR).

Dépendance vis-à-vis de l'ordre des calculs

La représentation par liens présente toutefois un inconvénient : le numéro de l'identifiant ne dépend que de l'ordre dans lequel les ACR ont été rencontrés, si bien que d'un calcul à l'autre, un même ACR peut avoir différents identifiants. Il faut donc faire attention à la non-compatibilité des bases de données engendrées.

Nous verrons au paragraphe 4.4.6 que dans nos calculs, nous avons produit une base de données d'ACR une fois pour toutes, de façon à éviter ce problème.

4.3.2 Calcul de l'arbre canonique réduit d'une position

La définition du paragraphe 4.2.5 s'adapte immédiatement pour fournir un algorithme récursif de calcul de l'ACR d'une position :

* calcul de l'ACR de chacun des fils de la position.
* suppression des doublons parmi ces ACR.
* réduction de l'arbre obtenu (si possible), en utilisant le plus petit réducteur possible.

Les deux premières étapes ne posent aucune difficulté, mais la programmation de la réduction mérite d'être détaillée.

Réduction dans le cas de colonnes de Nim

La première réduction qu'il est utile de traiter est celle correspondant au théorème 10 : si tous les ACR des fils de la position sont des colonnes de Nim, et si l'un au moins est $\mathbb{0}$ ou $\mathbb{1}$, alors, l'ACR de la position est lui-même une colonne de Nim que l'on peut déterminer avec la règle du mex.

Réduction à $\mathbb{0}$

Ensuite, il faut tester si l'ACR de la position est $\mathbb{0}$. Cette réduction est particulière, car nous avons vu dans le paragraphe 4.2.4 qu'elle n'est possible que si la position est gagnante en version misère.

Il faut donc commencer par vérifier si l'un des fils est perdant en misère, ce qui est immédiat, car l'issue est stockée dans l'identifiant de l'ACR (sans cela, il faudrait déterminer l'issue de la position en menant un calcul sur tout l'arbre, ce qui serait bien plus coûteux en temps de calcul).

Puis, on regarde si chacun des ACR des fils est un coup réversible, c'est-à-dire s'il a $\mathbb{0}$ comme fils.

Autres réducteurs

Si aucune des réductions précédentes n'a fonctionné, il faut voir s'il est possible de trouver un autre réducteur. Imaginons donc que nous sommes en train de calculer l'ACR d'une position, et qu'après avoir supprimé les doublons parmi les ACR de ses fils, nous ayons obtenu l'ensemble : $\{\mathscr{A}_1; \mathscr{A}_2; \mathscr{A}_3 ...\}$. On note h_i La hauteur de $\mathscr{A}_i$. On trie les ACR $\mathscr{A}_i$ de sorte que la suite (h_i) soit croissante. Alors le résultat du paragraphe 4.2.5 implique qu'il suffit de tester les réducteurs de la forme $\{\mathscr{A}_1; \mathscr{A}_2; ...; \mathscr{A}_i\}$, où $h_{i+1} \geq h_i + 2$.

En reprenant les notations de l'exemple du paragraphe 4.3.1, imaginons que nous cherchons à réduire : {`0-0-W 2-0-W 4-1-L 4-2-W 5-0-W 7-0-W`}. Nous devons donc tester uniquement les réducteurs potentiels suivants :

* {`0-0-W`}
* {`0-0-W 2-0-W`}
* {`0-0-W 2-0-W 4-1-L 4-2-W 5-0-W`}

Pour être sûr de trouver le plus petit réducteur possible, notre algorithme examine en priorité les petits réducteurs potentiels.

{`0-0-W`}=`1-0-L` n'est pas un réducteur convenable, car `1-0-L` est certes un fils de `2-0-W`, mais pas de `4-1-L`.

Ensuite, notre algorithme regarde le réducteur potentiel {`0-0-W 2-0-W`}. Il commence par chercher dans la base de données l'identifiant de l'ACR qui a ces deux fils, mais il ne le trouve pas. Et pour cause : {`0-0-W 2-0-W`} se réduit à `1-0-L` (voir paragraphe 4.3.2). Comme le plus petit réducteur possible doit être un ACR, et que cet ACR doit être le fils d'au moins un des fils de la position, la nature récursive de l'algorithme implique que cet ACR a déjà été stocké dans la base de données. Donc, quand notre algorithme ne trouve pas

un réducteur potentiel dans la base de données, c'est que ce réducteur potentiel est lui-même réductible, et donc il n'est pas nécessaire de l'essayer.

Il reste juste à tester {0-0-W 2-0-W 4-1-L 4-2-W 5-0-W} comme réducteur potentiel. Or, il ne peut pas convenir, car les seuls fils de 7-0-W sont des colonnes de Nim. Donc, {0-0-W 2-0-W 4-1-L 4-2-W 5-0-W 7-0-W} ne peut pas être réduit. C'est un nouvel ACR, et notre programme lui fournit alors un identifiant de la forme 8-n-W (comme le fils 4-1-L est perdant en misère, alors il est gagnant en misère).

4.3.3 Factorisation par $\mathbb{1}$

Nous avons vu dans le paragraphe 4.2.10 qu'il pouvait être utile de repérer quels ACR pouvaient s'écrire comme somme d'un autre ACR et de la colonne de Nim $\mathbb{1}$. Nous allons détailler ici l'implémentation de cette factorisation, l'étape de factorisation devant se dérouler juste après les différentes réductions dans l'algorithme récursif.

Représentation d'une position factorisable par $\mathbb{1}$

Si deux ACR $\mathscr{G}$ et $\mathscr{H}$ vérifient $\mathscr{G} \sim \mathscr{H} + \mathbb{1}$, alors leurs hauteurs diffèrent de 1. Dans la notation choisie, on ne changera pas l'identifiant du plus petit des deux, mais par contre, celui du plus grand s'écrira en fonction de celui du plus petit.

Explicitons cette nouvelle notation. On sait que $\mathbb{3} \sim \mathbb{2} + \mathbb{1}$ (ou, symétriquement, que $\mathbb{2} \sim \mathbb{3} + \mathbb{1}$). Le plus petit arbre des deux est $\mathbb{2}$. Donc $\mathbb{2}$ conserve son identifiant : 2-0-W, tandis que $\mathbb{3}$ aura désormais l'identifiant : 2-0+1-W.

On a vu également que $\{\mathbb{3}; \{\mathbb{2}\}\} \sim \mathbb{1} + \{\mathbb{2}\}$. Comme l'identifiant de $\{\mathbb{2}\}$ est : 3-1-L, celui de $\{\mathbb{3}; \{\mathbb{2}\}\}$ sera : 3-1+1-W (l'issue W est celle de 3-1+1, pas celle de 3-1).

Détection d'une position factorisable par $\mathbb{1}$

Reprenons l'exemple et les notations du paragraphe 4.3.1. En utilisant le nouvel identifiant décrit au paragraphe précédent, on a :

4-2-W={0-0+1-L 2-0+1-W 3-1-L}

Maintenant, calculons les fils de 4-2-W+$\mathbb{1}$. Pour obtenir un fils, soit on joue un coup dans 4-2-W, soit dans $\mathbb{1}$. On a donc :

4-2-W+$\mathbb{1}$={0-0-W 2-0-W 3-1+1-W 4-2-W}

On peut observer sur cet exemple un résultat plus général : si $\mathscr{G}$ est un ACR qui ne se factorise pas par $\mathbb{1}$, alors $\mathscr{G} + \mathbb{1} = \{(\text{fils de } \mathscr{G})+\mathbb{1}\,;\, \mathscr{G}\}$. Donc dans notre algorithme, pour repérer que l'ensemble des ACR des fils d'une position correspond à une position factorisable par $\mathbb{1}$, il suffit de vérifier que cet ensemble est bien de cette forme. En particulier, $\mathscr{G}$ doit être l'unique ACR de hauteur maximale qui ne se factorise pas par $\mathbb{1}$.

4.4 Algorithme de calcul utilisant les ACR

La théorie des arbres canoniques réduits décrite jusqu'ici souffre d'un défaut majeur, comparé à la version normale : elle ne permet pas de déduire immédiatement l'issue d'une somme de composantes à partir d'informations indépendantes sur les composantes. Il ne va donc pas être possible de séparer le calcul de la somme en calcul sur les composantes. Heureusement, cela ne rend pas la théorie des arbres canoniques inutiles pour autant.

4.4.1 Composantes des sommes

L'idée générale est de commencer par remarquer que dans la plupart des jeux (c'est le cas aussi bien du Sprouts que du Cram), certaines petites positions réapparaissent fréquemment. Elles peuvent réapparaître en tant que positions à part entière, auquel cas il s'agit d'une transposition classique, comme expliqué dans le paragraphe 2.7.4 du chapitre d'introduction. Mais elles peuvent apparaître plus généralement en tant que composante dans des positions de type somme.

Imaginons par exemple le cas d'une position découpée en deux composantes $\mathscr{P}_1 + \mathscr{D}$ et d'une autre position somme $\mathscr{P}_2 + \mathscr{D}$. La composante $\mathscr{D}$ est commune aux deux positions, tandis que $\mathscr{P}_1$ et $\mathscr{P}_2$ sont différentes. On ne peut pas séparer le calcul de la somme en calcul sur les composantes, comme en version normale, mais on peut tout de même essayer de profiter du fait que $\mathscr{D}$ est une composante commune.

Par exemple, on pourrait précalculer les options de $\mathscr{D}$ une fois pour toutes, et les stocker dans une table. Cela consommerait de la mémoire supplémentaire, et accélèrerait en échange le temps de calcul puisque l'on n'aurait besoin de calculer les options de $\mathscr{D}$ qu'une seule fois. On peut bien sûr aller plus loin, et préparer à l'avance le calcul des options de $\mathscr{D}$, et même tout l'arbre de jeu de $\mathscr{D}$.

C'est ici qu'intervient une idée intéressante : quitte à préparer le calcul de tout l'arbre de jeu de $\mathscr{D}$, autant en profiter aussi pour supprimer les branches redondantes. Cela reviendrait tout simplement à remplacer $\mathscr{D}$ par son arbre canonique. Et en poussant cette idée jusqu'au bout, on arrive finalement à la notion qui nous intéresse : quitte à remplacer $\mathscr{D}$ par son arbre canonique, autant calculer l'arbre canonique réduit, qui permet notamment de supprimer les coups réductibles en plus des coups redondants.

4.4.2 Simplification des positions avec les ACR

Voici maintenant comment il est possible d'utiliser les ACR pour accélérer les calculs. Lors de l'exécution de l'algorithme 7, on remplace certaines composantes des sommes par leurs ACR. La difficulté principale est que l'on ne peut plus se contenter de manipuler des positions d'un jeu donné, il faut manipuler des objets plus complexes, qui sont des sommes hybrides de positions du jeu et d'ACR. Ainsi, un nœud de l'arbre de recherche sera composé de trois parties :

* la partie position (de Sprouts ou de Cram), qui comporte une ou plusieurs positions indépendantes, trop grandes pour que l'on puisse calculer leur ACR.
* la partie $\mathbb{0}/\mathbb{1}$, qui vaut $\mathbb{0}$ ou $\mathbb{1}$ suivant la parité du nombre de $\mathbb{1}$.
* la partie ACR, qui contient une liste d'ACR.

4.4.3 Exemple sur une position de Sprouts

Nous allons détailler la nature de ces nœuds sur un exemple. On considère la position de Sprouts :

`0*8 + 22 + 2ab2ba + 0*2.A|2A`

Cette position comporte 4 positions indépendantes, plus ou moins complexes :

* `0*8` est trop grande pour que l'on puisse calculer son ACR.
* `22`$\sim \mathbb{1}$.
* `2ab2ba`$\sim \mathbb{3} \sim \mathbb{1} + \mathbb{2} \sim \mathbb{1}$+`2-0-W`.
* `0*2.A|2A`$\sim \mathbb{1}$+`3-1-L`.

Les identifiants des ACR sont ceux de la table 4.1 utilisée dans un exemple précédent.

Finalement, en utilisant que $\mathbb{1} + \mathbb{1} + \mathbb{1} \sim \mathbb{1}$, on obtient que cette position a la même issue que `0*8`+$\mathbb{1}$+{`2-0-W`+`3-1-L`}.

4.4.4 Calcul des enfants d'un nœud

Le calcul des enfants d'un nœud se fait en prenant en compte les 3 types de coups possibles. Chaque coup s'effectue dans une des 3 parties du nœud, en ne changeant pas les deux autres :

* un coup dans la partie position.
* un coup dans la partie $\mathbb{0}/\mathbb{1}$, qui consiste à remplacer $\mathbb{1}$ par $\mathbb{0}$.
* un coup dans la partie ACR, qui consiste à remplacer un des ACR par un de ses fils.

4.4.5 Intérêt des ACR

Le remplacement des positions indépendantes par leur ACR quand il est connu a plusieurs avantages. Quel que soit le jeu considéré, beaucoup de positions presque terminales ont le même ACR, donc les remplacer par leur ACR permet de simplifier l'arbre de jeu, en diminuant le nombre de nœuds stockés et explorés (et donc, cela diminue la RAM consommée et améliore le temps de calcul).

Pour les positions de taille plus grande, il est rare qu'elles soient identiques, même après le calcul de l'ACR. Le gain est alors essentiellement celui décrit dans le paragraphe 4.4.1 : on gagne du temps chaque fois que la position apparaît dans une somme, en évitant d'avoir à recalculer la liste de ses options. La base de données des couples (ACR ; liste des options de l'ACR) procure une mémoire cache qui permet de ne pas réaliser inutilement les mêmes calculs d'options plusieurs fois.

4.4.6 Remplacement d'une position par son ACR

Le calcul de l'ACR d'une position demande l'exploration de tout son arbre de jeu. Ceci limite malheureusement la taille des positions que l'on peut remplacer par leur ACR. La taille de l'arbre de jeu d'une position augmente en effet extrêmement rapidement quand la taille de la position augmente, que ce soit sur le Sprouts, ou sur le Cram. On peut s'en apercevoir dans le tableau 4.2 : nous avons compté le nombre d'arbres canoniques différents qui interviennent dans les arbres de jeu de positions de départ du Sprouts.

points de départ	nombre d'arbres canoniques
2	10
3	55
4	713
5	10 461
6	150 147
7	2 200 629

TABLE 4.2 – Nombre d'arbres canoniques différents dans les arbres de jeu des positions de départ du Sprouts.

Nous avons envisagé deux critères différents pour décider de calculer l'ACR d'une position. Le premier serait de calculer l'ACR de toutes les positions en dessous d'une certaine limite sur la taille de la position. Dans le cas du Sprouts, il pourrait s'agir du nombre de vies. Dans le cas du Cram, on pourrait utiliser le nombre de cases encore vides comme critère. Cette méthode présente l'avantage de s'adapter au calcul en cours : on ne calcule les ACR que pour les positions effectivement rencontrées. Par contre, elle a pour inconvénient de modifier la base des ACR stockés au fur et à mesure du calcul.

Nous avons remarqué également que ce critère dynamique n'est pas bien adapté au cas particulier du Sprouts : deux positions ayant le même nombre de vies peuvent engendrer des arbres de complexités très diverses. Par exemple, dans l'arbre de jeu de la position de départ

à 3 points (donc à 9 vies), on rencontre 55 arbres canoniques différents, alors que dans l'arbre de jeu de la position `1abcde2edcba.2` (qui comporte également 9 vies) on en compte 478.

Nous avons donc retenu un autre critère de remplacement des positions par les ACR : dans une étape de précalcul, nous calculons les ACR d'un certain ensemble de positions bien choisies. Une fois cet ensemble d'ACR calculé, nous n'en calculons plus aucun autre, et nous disposons ainsi d'une base d'ACR fixe pour lancer le calcul principal qui nous intéresse.

4.4.7 Cas du Sprouts

Dans le cas du Sprouts, les positions de départ avec p points, ainsi que les positions de leur descendance, interviennent rapidement dans les calculs de positions avec un nombre de points plus élevé. Nous avons donc calculé l'ACR de la position de départ avec 6 points de départ.

Ensuite, lors de la phase de calcul principale, si une composante s'avère être une position apparaissant dans l'arbre de jeu à 6 points de départ, elle sera remplacée par son ACR. Par exemple, imaginons que depuis la position de Sprouts à 12 points, on joue le coup `0*8.AB|0*3.AB`, puis le coup `0*8 + 0*2.A|0.A`. À ce moment-là, notre algorithme modifiera le nœud, car on peut lire dans la base de données précalculée que `0*2.A|0.A`$\sim$`3-1+1-W`. Par contre, le reste de la position n'apparaît pas dans cette base.

Le nouveau nœud est donc `0*8`+$\mathbb{1}$+`3-1-L`.

4.4.8 Sommes de positions

On pourrait envisager de regrouper les parties $\mathbb{0}/\mathbb{1}$ et la liste d'ACR des nœuds dans un unique ACR, en calculant l'ACR de leur somme. Dans le cas du paragraphe 4.4.3, on remplacerait $\mathbb{1}+\{$`2-0-W+3-1-L`$\}$ par un unique ACR de hauteur 6, `5-1+1-W`. L'intérêt d'une telle méthode est double : outre une écriture simplifiée des nœuds, elle permettrait de détecter les simplifications décrites à la fin du paragraphe 4.2.10.

Cependant, cette méthode n'est pas envisageable, car le calcul de la somme des ACR nécessite le stockage de trop nombreux ACR supplémentaires. Par exemple, dans le cas du Sprouts, après avoir mis en mémoire l'ACR de la position de départ à 6 points, imaginons que nous cherchions à calculer l'issue de la position `0*5 + 0*4`. Les ACR de ces deux positions indépendantes sont connus (et sont de hauteurs respectives 13 et 6).

L'algorithme décrit précédemment remplacerait chacune de ces deux positions par son ACR, et lancerait donc le calcul sur le nœud [4] :

$$\varnothing + \mathbb{0}+\{\texttt{13-7-W+6-208-L}\}$$

L'algorithme de calcul permet alors de trouver l'issue de cette position en stockant seulement 10 positions, alors qu'au contraire, le calcul de l'ACR de `13-7-W+6-208-L` nécessite de stocker plus de 35 000 nouveaux ACR, soit plus que pour le calcul de l'ACR de la position de Sprouts à 6 points de départ.

Lorsque nous menons un calcul misère, nous sommes fréquemment amenés à étudier de telles sommes (et même de plus complexes), et il n'est donc pas envisageable de calculer l'ACR résultant. Enfin, les simplifications espérées du paragraphe 4.2.10 ne compensent pas ce problème : lors des tests que nous avons menés, nous n'en avons observé aucune.

4.5 Résultats

Nous avons appliqué les algorithmes à base d'arbres canoniques réduits aux jeux de Sprouts et de Cram en version misère.

4. Rappelons que les numéros des identifiants dépendent du précalcul des ACR. Les numéros `7` et `208` n'ont donc aucune importance.

4.5.1 Jeu de Sprouts

Nous avons pu calculer les issues gagnantes (W) ou perdantes (L) des positions de Sprouts jusqu'à 20 points de départ, alors que le meilleur résultat connu auparavant était celui de la position de départ à 16 points, par Josh Jordan Purinton et Roman Khorkov [25].

points	1	2	3	4	5	6	7	8	9	10
issue	W	L	L	L	W	W	L	L	L	W

points	11	12	13	14	15	16	17	18	19	20
issue	W	W	L	L	L	W	W	W	L	L

TABLE 4.3 – Issues gagnantes/perdantes calculées.

Les résultats concernant le jeu de Sprouts sont détaillés dans le chapitre 10.

4.5.2 Jeu de Cram

Les meilleurs résultats connus précédemment en version misère étaient ceux de Martin Schneider en 2009 [39], indiqués par un astérisque dans les tableaux.

	4	5	6	7	8	9
4	*L	*L	*L	W	W	W
5	–	*W	W	W		
6	–	–	W			

TABLE 4.4 – Résultats misère obtenus sur les plateaux de taille $n \times m$, avec $n \geq 4$ et $m \geq 4$.

Les principaux résultats nouveaux sont le plateau misère 4×9 (36 cases), 5×7 (35 cases) et 6×6 (36 cases), alors que le meilleur résultat connu auparavant était le plateau misère 5×5 (25 cases).

Ces résultats, ainsi que d'autres obtenus sur les plateaux de taille $3 \times n$, sont détaillés dans le chapitre 12.

4.6 Conclusion

La théorie des arbres canoniques réduits ainsi que l'algorithme de calcul qui en découle pour les jeux impartiaux en version misère se sont avérés particulièrement efficaces dans le cas du Sprouts et du Cram. Même si l'on n'est pas capable de séparer le calcul de la somme en calcul sur les composantes uniquement, le remplacement des petites composantes par leurs arbres canoniques réduits permet de tirer partie des découpages de positions en sommes de composantes indépendantes.

Chapitre 5

Suivi des calculs

Le suivi des calculs est une idée qui est apparue dès nos premiers calculs de Sprouts. Il n'était alors pas possible de calculer l'issue de la position de départ à 12 points en version normale, la première valeur inconnue à l'époque, vu le temps que prenait le calcul sans pour autant réussir à se terminer.

Il était donc nécessaire d'avoir une image relativement précise du degré d'avancement de ce calcul, pour décider si cela valait le coup de le laisser se poursuivre, ou s'il fallait l'arrêter et chercher une amélioration. Le suivi des calculs est donc dès le départ une méthode d'aide à la décision pour le programmeur.

Nous décrivons ci-après, par ordre chronologique, les différentes techniques mises en place pour suivre le déroulement des calculs. Nous verrons comment cela nous a permis de faire évoluer le programme.

5.1 Affichage de la branche de calcul

La première information intéressante à afficher est la branche de l'arbre de recherche en cours de calcul. Lors du parcours d'un arbre en profondeur, la *branche de calcul* est constituée d'une suite de nœuds dont chacun est un fils du précédent. La figure 5.1 montre en blanc les nœuds correspondants à la branche de calcul. Les nœuds en gris avec l'indication « L » ou « W » (*Loss* ou *Win*) sont ceux qui ont déjà été calculés, dans l'ordre indiqué par les numéros. Les nœuds en gris sans numéro sont ceux qui n'ont pas encore été calculés, soit parce qu'il n'y en aura pas besoin (le nœud à gauche), soit parce qu'ils seront calculés plus tard (les nœuds à droite).

Le premier suivi que nous avons implémenté consiste donc à afficher cette branche de calcul (les nœuds 1-8-10-11 de la figure 5.1). Chacun des cadres dessinés sur la figure 5.1 correspond à un niveau de l'interface de suivi. On notera que dans le cas d'un algorithme alpha-bêta classique, les nœuds en mémoire à un instant donné sont exactement ceux encadrés : à chaque étape de calcul, on développe les enfants d'un nœud, on les ordonne avec différentes heuristiques, puis on effectue la même opération récursivement sur les enfants, dans l'ordre obtenu.

Pour chaque niveau, nous affichons la représentation en chaîne de caractères de la position correspondant à ce nœud, le nombre de nœuds de ce niveau et le nombre de nœuds restants à calculer. Le nombre de nœuds restants à calculer sur un niveau donné est une information importante pour se faire une idée de l'avancement des calculs, car s'il ne reste plus que 3 nœuds sur 14 à calculer sur un niveau donné, on peut en général supposer que le calcul est plus avancé que s'il en restait encore 12 sur 14. La table 5.1 montre l'interface de suivi correspondant à la figure 5.1.

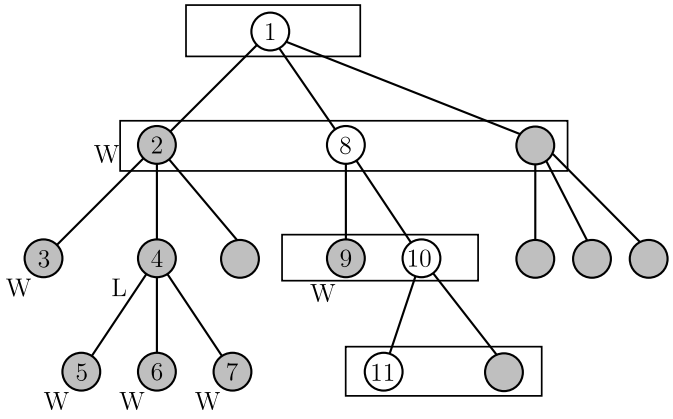


FIGURE 5.1 – Branche de calcul (nœuds en blancs).

profondeur	nombre de nœuds	nœuds encore inconnus	nœud en cours de calcul
1	1	1	nœud 1
2	3	2	nœud 8
3	2	1	nœud 10
4	2	2	nœud 11

TABLE 5.1 – Interface de suivi correspondant à la figure 5.1.

La première version du suivi a été programmée avec la bibliothèque Ncurses [1], qui permet d'afficher du texte dans un terminal. Le programme ne disposait pas encore d'interface graphique à cette époque. Assez rapidement, cependant, le besoin d'une interface graphique s'est fait ressentir, par exemple pour pouvoir choisir facilement les options de calcul avec des boutons. La bibliothèque Ncurses a été abandonnée, pour laisser la place à une interface graphique avec la bibliothèque Qt. La perte de performances liée à l'utilisation de Qt s'est révélée relativement faible, et largement compensée par la richesse des outils disponibles.

5.2 Zappage

5.2.1 Positions bloquantes

Le suivi des calculs a permis dans un premier temps d'améliorer l'ordre des positions de Sprouts, en visualisant plus facilement l'effet de certaines priorités. Dans un deuxième temps, il a surtout révélé qu'il est très difficile d'établir un ordre adapté au Sprouts. La nature impartiale du jeu rend difficile la création d'heuristiques pour donner la priorité aux positions perdantes, et d'autre part, l'arbre de Sprouts est extrêmement déséquilibré. Il y a largement un facteur 100 voire 1000 ou plus de différence de difficulté entre certaines positions d'un même niveau de l'arbre de jeu, parfois sans raison évidente. Quel que soit l'ordre que nous avons essayé, il a fini par donner la priorité, à un moment ou à un autre, à une position nettement plus difficile que les autres, avec un effet catastrophique sur le temps de calcul. Nous appelons *positions bloquantes* ces positions nettement plus difficiles à calculer que les autres.

1. `http://www.gnu.org/software/ncurses/`

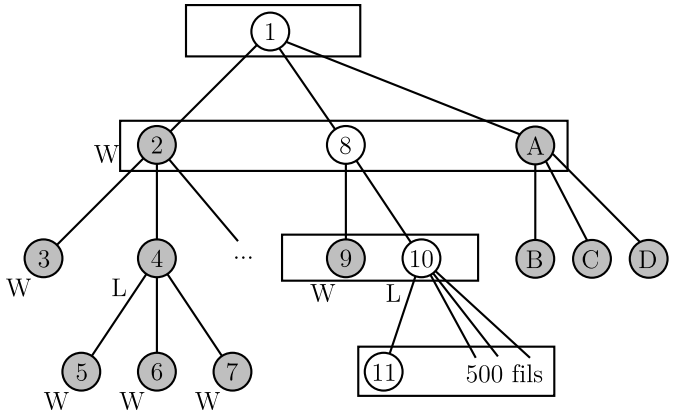


FIGURE 5.2 – Exemple d'arbre avec une position bloquante.

La figure 5.2 illustre comment une position bloquante pourrait apparaître dans l'arbre de recherche de la figure 5.1. Lorsque les fils du nœud 1 ont été calculés, il est tout à fait imaginable que les nœuds aient été ordonnés dans l'ordre 2 ; 8 ; A. Par exemple, il fréquent que le nombre d'enfants soit l'un des critères pour ordonner les nœuds, auquel cas il est logique de donner la priorité au nœud 8 sur le nœud A.

Maintenant, imaginons que le nœud 10 soit perdant, comme indiqué sur la figure. Pour le prouver, il va falloir calculer les 500 autres fils du nœud 10, en plus du nœud 11 en cours de calcul. Le nœud 8 est donc une position bloquante.

5.2.2 Principe du zappage manuel

Le suivi correspondant à la figure 5.2 est le même que celui de la table 5.1 à ceci près que la dernière ligne indiquerait les valeurs 501 et 501, au lieu de 2 et 2 pour le nombre total de nœuds et le nombre de nœuds inconnus du niveau 4. L'intérêt du suivi est de rendre visible à l'utilisateur que la position 8 est bloquante : le calcul va rester longtemps bloqué sur les nœuds 8 et 10, en comparaison du temps passé sur les nœuds précédents.

L'idée du zappage est alors de permettre à l'utilisateur de lancer un ordre de « changement du nœud en cours » sur le niveau 2, pour forcer le calcul à abandonner le nœud 8 et commencer le calcul du nœud A. Ainsi, si le nœud A est perdant et se calcule rapidement, le calcul du nœud 8 ne sera pas nécessaire, et des calculs difficiles seront évités.

5.2.3 Implémentation multi-processus

Le suivi et le zappage ne sont pas faciles à implémenter car ils nécessitent une communication entre le calcul et l'interface utilisateur. Par ailleurs, pour ne pas avoir une interface figée pendant le calcul, il faut rendre la main régulièrement au moteur de rendu graphique de la bibliothèque Qt. La meilleure méthode d'implémentation consiste en fait à exécuter en permanence le calcul et le moteur de rendu graphique de Qt, à travers des processus (*threads* en anglais) différents. La difficulté est alors de communiquer des informations entre les processus.

La figure 5.3 montre les principaux éléments mis en œuvre pour cette communication. Ce que nous désignons par « interface » correspond à la fonction en cours d'exécution dans le thread graphique. La boucle de calcul est la fonction en cours d'exécution dans le thread de calcul. Ces deux processus communiquent à travers l'arbre de recherche et la zone de stockage

des ordres. L'arbre de recherche est stocké sous la forme d'une table qui contient la liste des nœuds en cours de calcul, avec toutes les informations correspondant aux différents nœuds, comme le nombre de fils, le nombre de fils encore inconnus, le fils en cours d'étude, etc. La zone de stockage des ordres ne contient que quelques octets d'informations pour stocker les ordres émis par l'utilisateur.

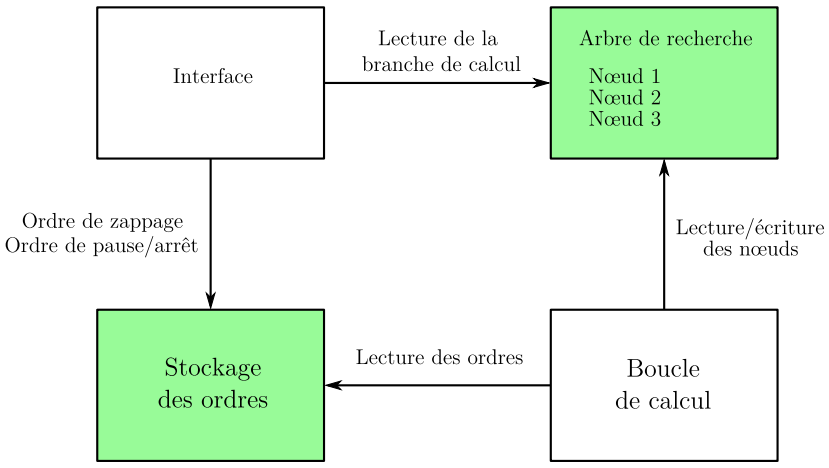


FIGURE 5.3 – Implémentation du suivi et du zappage.

L'interface accède à l'arbre de recherche à intervalles réguliers, généralement de 0,1 s, grâce à un Timer. Le ralentissement des calculs lié à ce suivi en temps réel est relativement mineur, de l'ordre de quelques pourcents. L'influence sur les calculs est détaillée plus loin (§5.5.3). La boucle de calcul accède quant à elle à la zone de stockage des ordres à la fin de chaque calcul d'un nœud. Le temps nécessaire pour cette opération très simple est tout à fait négligeable.

5.2.4 Sécurisation des objets partagés

L'arbre de recherche ainsi que la zone de stockage des ordres sont des objets *thread-safe*, ce qui signifie que plusieurs processus peuvent y accéder simultanément sans risque d'erreur. La programmation d'un objet thread-safe est délicate. En effet, si un processus modifie un objet pendant qu'un autre processus est en train de lire l'état de cet objet, le comportement du programme devient imprévisible. Il faut impérativement forcer les processus à accéder aux objets partagés à tour de rôle, et non pas simultanément.

La bibliothèque Qt fournit un objet spécial pour sécuriser les objets partagés, appelé *mutex*, avec des fonctions lock et unlock. Les fonctions lock et unlock du mutex ont la propriété classique d'être *atomiques*, à savoir qu'elles ne nécessitent qu'une seule exécution du microprocesseur, garantissant ainsi que leur accès est séquentiel. L'appel d'un lock par un processus ne peut pas commencer avant que le lock d'un autre processus soit terminé.

Avant d'accéder à un objet partagé, il faut vérifier que cet objet est disponible avec la fonction mutex.lock(). Si l'objet est disponible, cette fonction se termine immédiatement et bloque l'utilisation de l'objet par d'autres processus. Si l'objet n'est pas disponible, la fonction est mise en attente, jusqu'à ce que l'objet devienne disponible. Après utilisation de l'objet, il ne faut pas oublier de libérer son utilisation en appelant la fonction unlock. Typiquement, le code d'utilisation d'un objet partagé prend la forme de l'algorithme 8.

La simplicité de l'algorithme 8 cache en fait des problèmes redoutables. L'oubli d'un lock ou d'un unlock provoque en général des crashs intempestifs et quasiment impossibles à débugguer. Deux appels successifs à un lock par le même processus sans libération par

Algorithme 8 Utilisation d'un objet partagé $\mathcal{O}$

Appel de mutex.lock() pour l'objet $\mathcal{O}$
Utilisation de $\mathcal{O}$
Appel de mutex.unlock()

unlock provoquent cette fois un blocage permanent (appelé *dead-lock*). Ces deux cas peuvent se résoudre en vérifiant que l'utilisation de l'objet partagé est toujours encadré par le mutex, et en vérifiant que chaque appel à lock est bien suivi d'un appel à unlock. Pour donner un ordre de grandeur, il y a environ une quarantaine de blocs de code protégés par des mutex dans le programme, regroupés dans 2 fichiers principalement.

La vraie difficulté apparaît quand plusieurs objets partagés existent, avec des mutex différents pour chacun d'entre eux. Seule une étude précise de l'ordre d'appel des fonctions permet de s'assurer qu'aucun dead-lock ne peut apparaître. Si cela est possible, le plus simple est de ne jamais bloquer simultanément l'accès de deux objets partagés. Dans notre cas, nous nous sommes donc simplement assurer que toutes les fonctions (peu nombreuses) qui accèdent à la fois à l'arbre de recherche et à la zone de stockage des ordres de l'utilisateur prennent soin de bien libérer un objet avant d'accéder à l'autre.

5.2.5 Déroulement du zappage

Pour accélérer un calcul de Sprouts (ou de tout autre jeu disponible dans notre programme), il suffit à l'utilisateur de surveiller le déroulement du calcul alpha-bêta. Comme l'interface est rafraîchie en temps réel, l'utilisateur ressent intuitivement l'apparition d'un ralentissement des calculs, et il peut alors lancer des ordres de zappage sur les niveaux qui semblent plus lents.

L'idéal est de trouver sur un niveau une position qui semble perdante facilement. L'utilisateur peut adopter plusieurs stratégies :

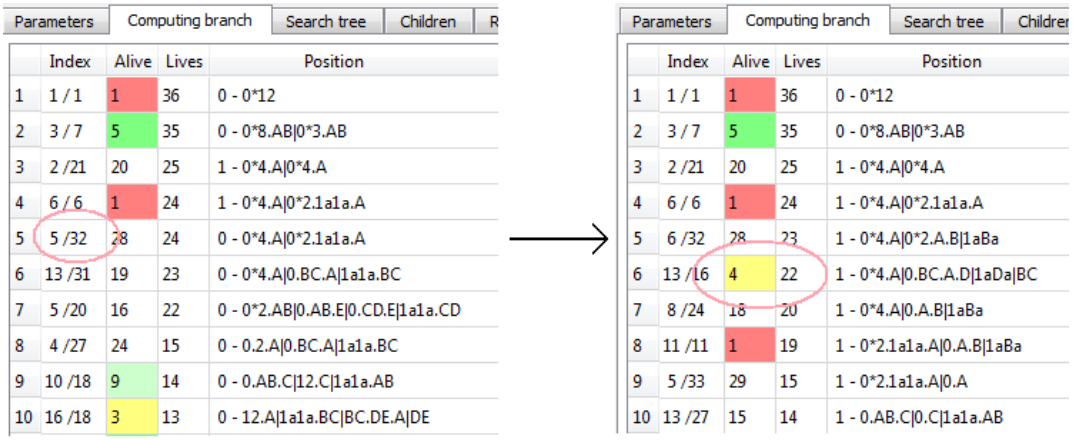
* utiliser des connaissances spécifiques au jeu qui n'ont pas encore été programmées, en se basant sur les chaînes de caractères qui représentent les positions, pour choisir des positions que l'on suppose perdantes et faciles à étudier.
* observer la forme du sous-arbre pour comparer la difficulté relative des différents nœuds d'un même niveau.

Nous avons par la suite amélioré l'interface pour faciliter les choix de l'utilisateur, en particulier en indiquant le statut de chaque niveau avec un code de couleurs. Nous matérialisons par une couleur le fait qu'il reste peu de nœuds inconnus sur un niveau : rouge vif, s'il n'en reste qu'un seul, puis orange, et jaune, pour quelques nœuds seulement, jusqu'au vert clair pour 10 nœuds. Au-delà de 10 nœuds inconnus, le niveau est indiqué simplement en blanc.

La figure 5.4 montre une capture d'écran de l'interface de suivi lors d'un calcul de Sprouts avec 12 points de départ. L'interface reprend essentiellement le contenu de la table 5.1. Les différentes colonnes signifient plus précisément :

* `Index` : numéro du fils en cours de calcul / nombre total de fils du niveau.
* `Alive` : nombre de fils encore inconnus sur le niveau.
* `Lives` : nombre de vies de la position de Sprouts (notion spécifique au Sprouts).
* `Position` : partie nimber du nœud suivi de la représentation en chaîne de la position.

Par exemple, on voit sur la capture de gauche que le fils en cours d'étude du niveau 2 est le 3^{e} sur un total de 7, et qu'il ne reste que 5 fils inconnus. On voit aussi que le niveau 4 est presque terminé, car il est affiché en rouge (dernier fils inconnu). Par contre, le fils en cours d'étude du niveau 5 n'a pas l'air simple : il possède 31 fils, dont 19 sont encore inconnus. L'utilisateur peut donc décider de zapper le fils en cours sur le niveau 5, en cliquant sur la cellule entourée.



Parameters | Computing branch | Search tree | Children | R

	Index	Alive	Lives	Position
1	1 / 1	1	36	0 - 0*12
2	3 / 7	5	35	0 - 0*8.AB|0*3.AB
3	2 /21	20	25	1 - 0*4.A|0*4.A
4	6 / 6	1	24	1 - 0*4.A|0*2.1a1a.A
5	5 /32	28	24	0 - 0*4.A|0*2.1a1a.A
6	13 /31	19	23	0 - 0*4.A|0.BC.A|1a1a.BC
7	5 /20	16	22	0 - 0*2.AB|0.AB.E|0.CD.E|1a1a.CD
8	4 /27	24	15	0 - 0.2.A|0.BC.A|1a1a.BC
9	10 /18	9	14	0 - 0.AB.C|12.C|1a1a.AB
10	16 /18	3	13	0 - 12.A|1a1a.BC|BC.DE.A|DE

Parameters | Computing branch | Search tree | Childre

	Index	Alive	Lives	Position
1	1 / 1	1	36	0 - 0*12
2	3 / 7	5	35	0 - 0*8.AB|0*3.AB
3	2 /21	20	25	1 - 0*4.A|0*4.A
4	6 / 6	1	24	1 - 0*4.A|0*2.1a1a.A
5	6 /32	28	23	1 - 0*4.A|0*2.A.B|1aBa
6	13 /16	4	22	1 - 0*4.A|0.BC.A.D|1aDa|BC
7	8 /24	18	20	1 - 0*4.A|0.A.B|1aBa
8	11 /11	1	19	1 - 0*2.1a1a.A|0.A.B|1aBa
9	5 /33	29	15	1 - 0*2.1a1a.A|0.A
10	13 /27	15	14	1 - 0.AB.C|0.C|1a1a.AB

FIGURE 5.4 – Suivi et zappage dans un calcul de Sprouts à 12 points de départ.

Le résultat de ce zappage apparaît à droite : le fils en cours d'étude sur le niveau 5 est passé du 5^{e} fils au 6^{e}, qui semble bien meilleur. Il n'a en effet lui-même plus que 4 fils inconnus, ce qui est indiqué par la couleur jaune du niveau 6, et il n'en restera bientôt plus que 3 car le niveau 8 est presque terminé.

5.2.6 Alternance des couleurs

L'alternance de couleurs obtenue sur la figure 5.4 est le motif graphique de référence que l'on recherche lorsque l'on effectue des zappages : les niveaux colorés indiquent que presque toutes les positions de ce niveau sont gagnantes, et les niveaux en blanc au-dessus correspondent donc en général à des positions perdantes.

Si deux niveaux consécutifs sont blancs, cela signifie souvent que le calcul est dans une zone où il n'a pas encore beaucoup avancé (c'était le cas des niveaux 5 et 6 à gauche de la figure). Inversement, deux niveaux consécutifs colorés correspondent souvent à une sorte de cas indéterminé : le calcul est bien avancé, mais on ne sait pas trop si l'issue sera gagnante ou perdante.

Quand on zappe, il est donc souvent suffisant de chercher à obtenir une alternance de couleurs, avec si possible des couleurs les plus proches possibles du rouge. Un peu d'entraînement permet d'effectuer des zappages efficaces rien qu'en se basant sur les couleurs, sans réfléchir, et le guidage du calcul s'apparente en quelque sorte à un jeu vidéo.

5.2.7 Avantages et inconvénients

Le zappage a eu un impact bien supérieur à ce que nous imaginions sur l'efficacité des calculs de Sprouts. Le déséquilibre des arbres de jeu du Sprouts rend en fait très efficace le fait d'éviter les positions bloquantes. Des calculs qui mettaient 24 heures avec le meilleur ordre programmé ont pu être ramenés à des durées de l'ordre d'une dizaine de minutes, grâce à un zappage régulier. Le zappage était un élément essentiel à l'obtention des records de Sprouts à partir de 15 points de départ.

Pour le Cram et le Dots-and-boxes, le zappage n'a pas eu un impact aussi important. La raison principale est que le déséquilibre des arbres de jeu est moins important pour ces deux jeux, si bien que les positions bloquantes sont moins flagrantes.

Le zappage n'en est pas moins un outil performant pour analyser et comprendre les arbres de jeu. Lors du développement du Cram, comme du Dots-and-boxes, il a permis de trouver de nombreuses astuces spécifiques à ces jeux pour améliorer l'ordre par défaut.

L'inconvénient principal du zappage est bien sûr le temps humain nécessaire pour surveiller en continu l'interface et effectuer les choix de nœuds à calculer. Lors des premiers records de Sprouts, le temps total passé à suivre et guider les calculs manuellement se compte en dizaines voire en centaines d'heures. Il est bien évident qu'une telle méthode n'est pas complètement satisfaisante.

5.2.8 Automatisation du zappage

L'idée même de zappage manuel semble une technique bien étrange dans un programme qui effectue des calculs a priori automatiques de jeux combinatoires. L'utilisateur humain est extrêmement lent, comparé à l'ordinateur. Si lent, que même si l'utilisateur clique dans l'interface aussi vite qu'il le peut, il s'écoule entre deux ordres largement une demi-seconde, pendant laquelle des milliers de choix par défaut sont effectués par le programme. Les interactions manuelles de l'utilisateur représentent donc une quantité d'information infime comparée à l'immense majorité des choix automatiques faits par le programme entre temps. Et pourtant, dans la très grande majorité des cas, le zappage manuel accélère les calculs.

En fait, ce paradoxe apparent vient de l'extrême sensibilité des calculs aux choix effectués, en particulier ceux en haut de l'arbre. Or, dans le cas d'un algorithme alpha-bêta, les choix effectués ne sont jamais remis en cause. Un très petit nombre de remises en cause dans le haut de l'arbre peut donc avoir un effet nettement perceptible sur le temps de calcul.

Malheureusement, le fait même que les informations données par l'utilisateur soient très peu nombreuses rend difficile l'automatisation du zappage. D'une certaine façon, l'utilisateur corrige les contre-exemples de l'ordre par défaut du programme, et une bonne partie de ces contre-exemples n'ont pas ou peu de points communs les uns avec les autres. On ne peut donc pas espérer automatiser le zappage simplement en améliorant l'ordre par défaut. Il faut plutôt se tourner vers des techniques différentes de parcours de l'arbre de jeu, qui ne soient plus un algorithme alpha-bêta, ou du moins pas seulement.

5.3 Visualisation des arbres solutions

Les méthodes de vérification des calculs décrites dans le chapitre 7 permettent d'obtenir des arbres solutions de faible taille. Il est alors possible de les tracer pour disposer de solutions visuelles compactes des positions de départ considérées. Par exemple, la figure B.1 présentée en annexe B montre un arbre solution prouvant que le Sprouts à 5 points de départ en version normale est gagnant. Cette figure a été tracée à l'aide du logiciel Graphviz [2].

La version de 2007 de notre programme était capable, lors du processus de vérification, de générer des fichiers compatibles avec Graphviz. Le principe est simple : lorsque l'on rencontre une nouvelle position, on crée un nœud qui lui correspond. Puis, lorsque l'on calcule les options d'une position, on crée les arêtes qui relient cette position à ses options.

Pour des arbres solutions plus importants que celui de l'annexe, la représentation graphique à l'aide de Graphviz ne donne qu'un enchevêtrement d'arêtes illisible. Il était alors possible de limiter cette représentation aux premiers étages de l'arbre. La figure 5.5 montre un tel arbre : c'est un arbre solution de la position de Sprouts à 8 points, en version normale. On a représenté par un cercle les positions perdantes, par un triangle les positions gagnantes, et par un carré les sommes de positions indépendantes, dont on déduit l'issue à partir de positions qui apparaissent plus bas dans l'arbre.

Notre programme actuel n'est malheureusement plus capable de produire de tels graphiques. Ces graphiques étaient générés par une version du programme qui ne permettait de faire que des calculs de Sprouts, uniquement sur le plan, et uniquement en version normale. Depuis, de nouveaux jeux sont apparus (le Cram, le Dots-and-boxes), ainsi que de

2. `http://www.graphviz.org/`

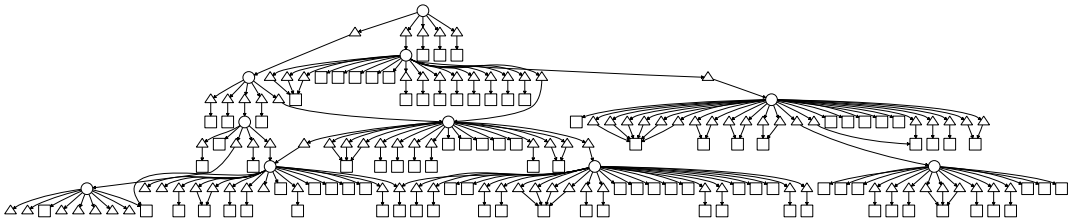


FIGURE 5.5 – Partie supérieure d'un arbre solution.

nouveaux algorithmes (version misère des jeux impartiaux, algorithme de score/contrat pour le Dots-and-boxes). Nous avons fait le choix très tôt d'implémenter tous nos algorithmes autour d'un moteur de calcul unique, de façon à ne pas devoir reprogrammer plusieurs fois certains éléments récurrents (bases de données, interfaces, boucle de calcul dans un processus indépendant, suivi des calculs, etc).

Beaucoup de fonctionnalités ont été perdues au cours de ce processus de généralisation du programme, avant de réapparaître ensuite, sous une forme souvent plus générale et plus efficace. La génération de fichiers compatibles avec Graphviz pour tracer des arbres est une des dernières fonctionnalités qui n'est toujours pas rétablie [3].

Son rétablissement serait envisageable non seulement pour illustrer les arbres solutions, mais aussi pour suivre en temps réel le développement de l'arbre de recherche, ou tout du moins des premiers étages de cet arbre. Pour des raisons de temps de calcul du programme Graphviz, l'actualisation ne serait cependant pas possible à une fréquence trop rapide : on peut imaginer mettre à jour le graphique toutes les 10 secondes, mais pas 10 fois par seconde, la fréquence de mise à jour dépendant évidemment de la taille de la partie de l'arbre de recherche qui serait représentée.

5.4 Algorithmes de type Proof-number search

5.4.1 Particularités de ces algorithmes

Nous avons implémenté des algorithmes de parcours de l'arbre de jeu plus complexes que le simple alpha-bêta, comme par exemple le Proof-number search (abrégé PN-search). Ces algorithmes explorent l'arbre de jeu dans un ordre totalement différent de l'alpha-bêta. L'idée générale est de développer l'arbre en meilleur d'abord (*best-first*), et non pas en profondeur.

L'application du PN-search au jeu de Sprouts nous a été suggérée initialement par Tristan Cazenave. Cela s'est révélé un bon choix, car le PN-search est bien adapté au parcours d'arbres de jeu déséquilibrés. Le PN-search maintient en permanence, pour chaque nœud de l'arbre de recherche, des paramètres p et d qui décrivent grossièrement la difficulté de calcul du nœud. À chaque étape de calcul, c'est la branche qui semble la plus facile qui est choisie, et c'est le nœud terminal de cette branche qui est développé. Au fur et à mesure du développement de l'arbre, les paramètres p et d sont affinés, si bien que le PN-search est capable de remettre ses choix en cause, contrairement à l'algorithme alpha-bêta.

Par plusieurs aspects, le PN-search adopte un comportement proche du zappage manuel :

* la branche de calcul est choisie de façon dynamique, en fonction de toutes les données accumulées au cours du calcul, contrairement à l'algorithme alpha-bêta qui se base sur un ordre statique prédéfini.
* le PN-search est capable de détecter qu'une branche devient bloquante par rapport à ses voisines, et va donner la priorité aux branches les plus faciles.

3. en juillet 2011.

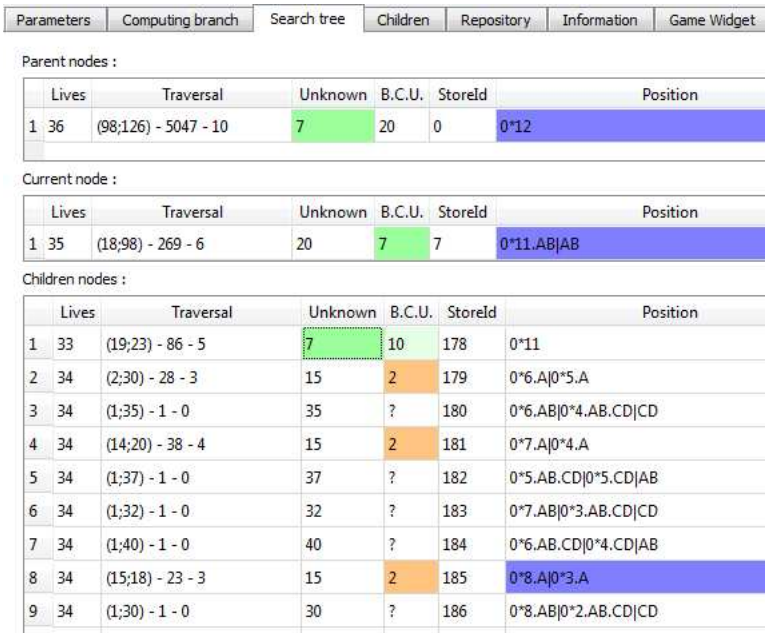
Ces fonctionnalités sont obtenues au prix du maintien en RAM de tout l'arbre développé depuis le début du calcul. La consommation de RAM est donc rapidement un facteur limitant. Par ailleurs, le PN-search se heurte à l'explosion combinatoire quand le facteur d'embranchement est élevé. Si une position possède 200 fils, le PN-search va avoir tendance à explorer ces 200 fils en parallèle, jusqu'à en trouver un nettement plus simple que les autres. L'algorithme se retrouve alors parfois perdu dans un arbre immense.

Le PN-search et les variantes que nous avons essayé d'implémenter sont décrites plus en détail dans le chapitre 6 sur les algorithmes de parcours.

5.4.2 Suivi du PN-search

La particularité du PN-search est de choisir à chaque étape la branche de calcul qui semble la plus simple à calculer à partir des valeurs connues des paramètres p et d. La branche de calcul change donc en permanence, ce qui rend relativement inutile le suivi que nous avons utilisé jusqu'ici pour l'alpha-bêta. Les positions affichées changent sans arrêt. Il n'est pas possible de comprendre et de suivre l'avancement de l'algorithme PN-search uniquement avec la branche de calcul.

Pour suivre le PN-search, nous avons donc développé une autre méthode, qui consiste cette fois à naviguer librement dans la partie de l'arbre de jeu qui a été déjà développée par l'algorithme PN-search. L'interface affiche toutes les informations utiles concernant un nœud donné : au centre de l'interface (current node), les différents éléments du nœud lui-même, dont les paramètres p et d du PN-search, en haut le nœud parent (parent node), et en base la liste des enfants (children nodes). L'utilisateur peut alors cliquer sur un enfant pour s'enfoncer dans l'arbre ou sur un parent pour remonter. Lors du clic sur un nœud parent ou enfant, c'est ce nœud qui devient le nœud central.



	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	36	(98;126) - 5047 - 10	7	20	0	0*12

	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	35	(18;98) - 269 - 6	20	7	7	0*11.AB|AB

	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	33	(19;23) - 86 - 5	7	10	178	0*11
2	34	(2;30) - 28 - 3	15	2	179	0*6.A|0*5.A
3	34	(1;35) - 1 - 0	35	?	180	0*6.AB|0*4.AB.CD|CD
4	34	(14;20) - 38 - 4	15	2	181	0*7.A|0*4.A
5	34	(1;37) - 1 - 0	37	?	182	0*5.AB.CD|0*5.CD|AB
6	34	(1;32) - 1 - 0	32	?	183	0*7.AB|0*3.AB.CD|CD
7	34	(1;40) - 1 - 0	40	?	184	0*6.AB.CD|0*4.CD|AB
8	34	(15;18) - 23 - 3	15	2	185	0*8.A|0*3.A
9	34	(1;30) - 1 - 0	30	?	186	0*8.AB|0*2.AB.CD|CD

FIGURE 5.6 – Suivi du PN-search sur le Sprouts à 12 points de départ.

Comme pour le suivi de la branche de calcul, les informations affichées sont mises à jour à intervalles réguliers, si bien que l'on voit les paramètres du nœud choisi évoluer en temps réel. L'implémentation ne pose pas de difficulté particulière. Il s'agit essentiellement d'une

extension du suivi normal, en utilisant le même objet partagé (l'arbre de recherche, qui contient les nœuds et toutes leurs informations relatives).

La figure 5.6 est une capture d'écran du suivi lors d'un calcul de PN-search sur le Sprouts à 12 points de départ. L'écran est subdivisé de haut en bas en trois zones : au centre, le nœud qui nous intéresse (`0*11.AB|AB`), en haut le parent de ce nœud (qui est la position de départ `0*12`), et en bas, la liste des enfants (coupée au 9e par la capture d'écran). Les colonnes `Lives` et `Position` ont la même signification que dans le suivi de la branche de calcul. Voici le détail des autres colonnes :

* `StoreId` : identifiant unique du nœud dans l'arbre de recherche. Cet identifiant sert de paramètre lorsque l'on clique sur un enfant ou un parent pour descendre ou remonter dans l'arbre.
* `Traversal` : paramètres p et d du nœud, nombre de nœuds développés dans le sous-arbre, et profondeur du sous-arbre.
* `Unknown` : nombre de fils inconnus.
* `B.C.U.` : valeur minimale d'`Unknown` parmi les fils.

Le nœud en cours de calcul est affiché en bleu, et l'on peut constater que l'enfant en cours de calcul (le 8e) est celui avec la plus petite valeur du paramètre d (18).

La colonne `Unknown` indique le nombre d'enfants inconnus du nœud, avec la même convention de couleur que dans le suivi de la branche de calcul. Plus la couleur se rapproche du rouge, et plus le nœud a de chances d'être *perdant*.

La notion de B.C.U., abréviation de « best-child unknown », est une notion que nous avons introduite dans le suivi par analogie avec la colonne `Unknown`, mais dans le but inverse, à savoir disposer d'une méthode graphique pour visualiser qu'un nœud a de fortes chances d'être *gagnant*. Cela se produit par définition lorsque le nœud a un fils qui a de fortes chances d'être perdant. Le fils qui a le plus de chances d'être perdant étant celui qui a lui-même le plus petit nombre de fils inconnus, c'est-à-dire la plus petite valeur d'`Unknown`, on définit donc le B.C.U. comme la valeur minimum d'`Unknown` parmi les fils. Plus la couleur du B.C.U. se rapproche du rouge, et plus le nœud a de chances d'être gagnant.

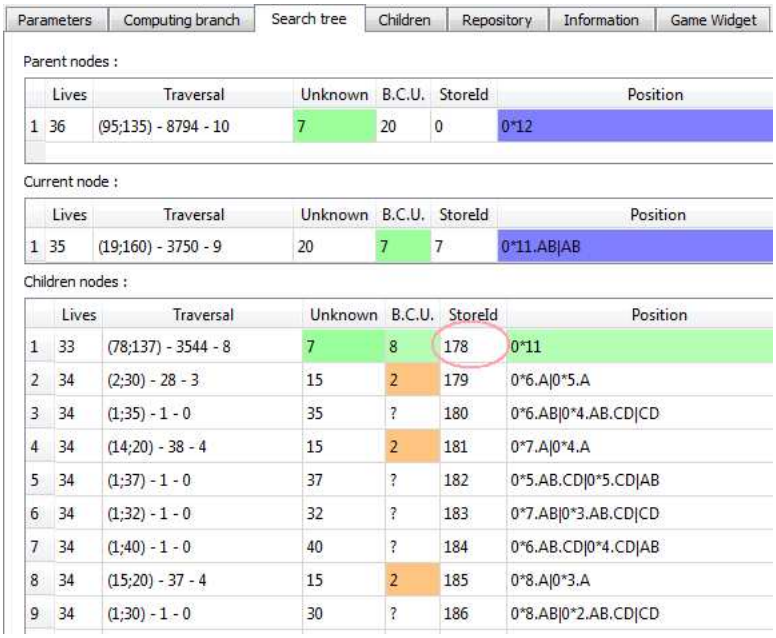
5.4.3 Interaction avec le PN-search

La possibilité de zappage manuel dans l'algorithme alpha-bêta ayant donné de bons résultats, il est naturel de chercher à introduire de la même façon des interactions manuelles dans l'algorithme PN-search. La principale faiblesse que nous avons observée dans le cas de l'algorithme PN-search est une saturation rapide de la mémoire à cause d'un développement excessif en largeur. Nous avons donc ajouté une possibilité d'interaction manuelle qui consiste à bloquer le PN-search sur un nœud donné de l'arbre.

Une fois que le PN-search est bloqué sur un nœud donné, l'algorithme de développement va se poursuivre normalement, mais uniquement en dessous du nœud indiqué, sans remonter jusqu'à la racine de l'arbre principal. Tout se passe comme si l'on avait stoppé le calcul principal, et que l'on avait lancé un calcul secondaire du nœud choisi, avec l'algorithme PN-search.

Là encore, l'implémentation est une simple extension des interactions manuelles dans le suivi normal. Le système de communication avec le calcul est le même, grâce à la zone de stockage des ordres de l'utilisateur.

La figure 5.7 montre un exemple d'interaction peu après la capture d'écran de la figure 5.6. L'utilisateur a cliqué sur la cellule entourée, pour indiquer au programme de bloquer le PN-search sur ce nœud. Le nœud sur lequel le calcul est bloqué est affiché en vert clair dans l'interface. On voit que le nombre de positions du sous-arbre augmente (3 544) alors que celui des autres nœuds du même niveau reste stable. Autre façon de vérifier que le blocage est bien effectif : le nœud en cours de calcul (`0*11`) a une valeur de paramètre $d = 137$. Si le



	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	36	(95;135) - 8794 - 10	7	20	0	0*12

	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	35	(19;160) - 3750 - 9	20	7	7	0*11.AB|AB

	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	33	(78;137) - 3544 - 8	7	8	178	0*11
2	34	(2;30) - 28 - 3	15	2	179	0*6.A|0*5.A
3	34	(1;35) - 1 - 0	35	?	180	0*6.AB|0*4.AB.CD|CD
4	34	(14;20) - 38 - 4	15	2	181	0*7.A|0*4.A
5	34	(1;37) - 1 - 0	37	?	182	0*5.AB.CD|0*5.CD|AB
6	34	(1;32) - 1 - 0	32	?	183	0*7.AB|0*3.AB.CD|CD
7	34	(1;40) - 1 - 0	40	?	184	0*6.AB.CD|0*4.CD|AB
8	34	(15;20) - 37 - 4	15	2	185	0*8.A|0*3.A
9	34	(1;30) - 1 - 0	30	?	186	0*8.AB|0*2.AB.CD|CD

FIGURE 5.7 – Blocage du PN-search sur le sous-arbre du nœud d'identifiant 178.

PN-search n'était pas bloqué, il abandonnerait ce nœud, et reprendrait le calcul du 2e ou du 8e nœud dont la valeur $d = 20$ est plus petite.

5.4.4 Intérêt des interactions

Comme espéré, les possibilités d'interaction dans le PN-search ont eu le même effet bénéfique que celles dans l'alpha-bêta. En empêchant manuellement le PN-search de se développer trop en largeur, nous compensons sa principale faiblesse. Cela a permis d'obtenir de nouveaux records sur le jeu de Sprouts, qui auraient été inatteignables autrement : l'alpha-bêta avec le zappage manuel aurait demandé un temps humain de guidage bien trop long, et le PN-search sans interaction se perd assez vite dans l'immensité des arbres de jeu du Sprouts au delà d'une trentaine de points de départ.

Les interactions ont permis d'identifier des positions qui induisent le PN-search en erreur. Par exemple, dans le cas du Sprouts, les positions du type `0.1aAa|0.1aBa|...` avec peu de vies ont en général un nombre élevé de positions inconnues dans leur sous-arbre. Le PN-search a donc tendance à les éviter, alors qu'elles sont en réalité presque terminales et souvent faciles à résoudre.

Inversement, le blocage est assez souvent contre-productif : si l'on se trompe dans le pronostic, le programme n'est pas capable de se débloquer par lui-même. Il serait donc intéressant de disposer d'une méthode non pas pour bloquer l'algorithme sur un nœud précis, mais plutôt pour favoriser certains nœuds par rapport à d'autres, avec des coefficients. Cela permettrait au calcul de favoriser le nœud pendant une durée limitée. Si le nœud est conforme au pronostic, il va être calculé en priorité par rapport aux nœuds non favorisés. Mais si le pronostic s'avère faux, le poids de ce nœud va augmenter, et au-delà d'un certain seuil, qui dépend du coefficient choisi, le calcul va finir par reprendre son exploration plus haut dans l'arbre.

5.4.5 Recherche de nouveaux algorithmes

Il est à noter que le PN-search avait été implémenté dans le but d'éviter les zappages manuels dans les calculs de Sprouts. Ce but a été partiellement atteint, car le PN-search seul permet de calculer des positions de départ plus complexes que l'alpha-bêta seul, et des positions de départ presque aussi difficiles que celles obtenues avec l'alpha-bêta combiné au zappage manuel. Mais l'introduction d'interactions au sein du PN-search permet de nouveau de nettement améliorer ses performances, et l'on retrouve le problème initial de l'automatisation complète des calculs.

En fait, l'algorithme PN-search, comme l'algorithme alpha-bêta, repose sur des règles simples et systématiques pour trouver un chemin possible vers la solution. En comparaison, l'utilisateur humain est bien plus souple dans les stratégies qu'il met en œuvre, ce qui lui permet d'identifier visuellement les faiblesses liées à ces règles systématiques. Dans le cas de l'alpha-bêta, la principale faiblesse identifiée est la non remise en cause de l'ordre par défaut des nœuds. Dans le cas du PN-search, c'est au contraire la remise en cause excessive des choix, donnant l'impression d'une dilution de l'algorithme dans l'immensité de l'arbre de jeu.

Les interactions de l'utilisateur lors de l'exécution du calcul lui permettent de corriger ces défauts dans la zone en haut de l'arbre. Un petit nombre d'interactions peut avoir des conséquences non négligeables à cause de la structure même d'arbre : plus on est en haut de l'arbre et plus l'impact de chaque choix influence le temps de calcul total. On remarquera que les défauts de l'alpha-bêta sont inverses de ceux du PN-search, et en conséquence les interactions de l'utilisateur aussi. Dans le cas de l'alpha-bêta, l'utilisateur force le programme à abandonner un choix par défaut en zappant. Dans le cas du PN-search, l'utilisateur force au contraire le programme à faire un choix, et à se fixer sur un nœud précis.

Il est raisonnable de supposer que quel que soit le degré de raffinement des algorithme de parcours, le suivi et les interactions manuelles resteront un outil intéressant, aussi bien pour surveiller le déroulement d'un long calcul que pour identifier des faiblesses dans les algorithmes et essayer de les corriger.

5.5 Affichage des plateaux

5.5.1 Position du problème

L'affichage des plateaux est la fonctionnalité la plus récente du suivi, dont le besoin est apparu avec la programmation de jeux de plateaux. Contrairement au Sprouts, la représentation d'un plateau sous forme de chaîne de caractères est parfaitement incompréhensible pour l'œil humain. Par exemple, les positions de Dots-and-boxes et de Cram de la figure 5.8 sont représentées respectivement par les chaînes `CLBLB*LGLGGBGCLBLGLGLQLCLBE` et `0000*0GG0GG00E`. L'affichage tel quel de ces chaînes de caractères dans l'interface ne permet pas de visualiser facilement quelles sont les positions en cours de calcul.

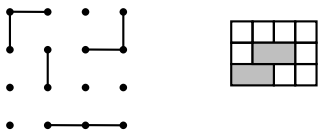


FIGURE 5.8 – Position de Dots-and-boxes et position de Cram.

5.5.2 Affichage en temps réel

En nous appuyant sur la bibliothèque graphique Qt, nous avons donc développé un affichage graphique des plateaux. La bibliothèque Qt contient des fonctions de dessin vectoriel,

si bien que techniquement, ce n'est pas très difficile à réaliser. Essentiellement, il suffit de dessiner un quadrillage à l'écran pour représenter le plateau, puis de remplir les différentes cases avec une couleur différente pour chaque lettre de la chaîne de caractères.

La puissance de la bibliothèque Qt apparaît surtout lorsque l'on combine cette possibilité d'affichage graphique des plateaux avec les fonctionnalités existantes de suivi. Le principe de Qt est de pouvoir fabriquer de nouveaux objets graphiques à partir des briques existantes de la bibliothèque. Nous avons donc développé un objet graphique (techniquement, cela prend la forme d'une classe C++ qui dérive de la classe fondamentale QWidget) permettant d'afficher un plateau de jeu avec les fonctions de dessin vectoriel. Puis nous avons inséré cet objet graphique dans le tableau de suivi du calcul, là où d'ordinaire nous affichons les chaînes de caractères des positions en cours de calcul. Ainsi, nous obtenons un affichage graphique en temps réel des plateaux de jeu en cours de calcul.

La figure 5.9 montre une capture d'écran lors du suivi d'un calcul de Cram du plateau de taille 3×16. Il s'agit d'un calcul en version normale. La position en cours de calcul sur le niveau 4 est constituée de deux composantes indépendantes. L'algorithme en version normale basé sur les nimbers va donc calculer le nimber de la première composante, ce qui est indiqué sur le niveau 5 par la couleur jaune.

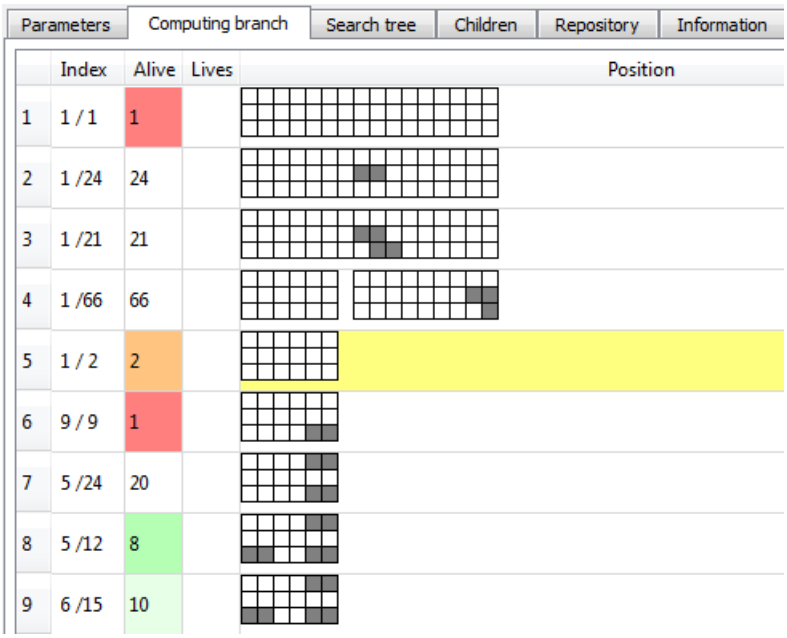


FIGURE 5.9 – Suivi graphique d'un calcul de Cram 3×16.

Nous n'avons pas encore développé de fonctionnalité similaire pour le jeu de Sprouts, d'une part parce que l'intérêt est moindre en terme de suivi/guidage des calculs (voire contre-productif, les chaînes étant généralement plus faciles à comprendre que les positions graphiques), et surtout parce que l'affichage graphique d'une position de Sprouts à partir de sa chaîne de caractères est d'une difficulté autrement plus sérieuse que l'affichage d'un plateau de jeu.

5.5.3 Influence sur le temps de calcul

L'affichage graphique en temps réel est bien sûr une opération nettement plus coûteuse que le simple affichage de la chaîne de caractères. Le suivi avec un affichage graphique, tel

que nous l'avons programmé, peut facilement atteindre 20 à 30 pourcents du temps de calcul, à comparer avec les quelques pourcents du suivi à base de chaînes de caractères. Pour éviter de ralentir les calculs (et aussi pour des raisons de confort visuel), nous avons créé une option dans l'interface permettant de régler l'intervalle de temps entre deux mises à jour du suivi. Un intervalle suffisamment élevé (par exemple 1 s au lieu de 0,1 s) permet de limiter l'impact du suivi sur le temps de calcul.

Il est à noter que sur une machine bi-processeur, le suivi ne ralentit pas du tout les calculs. En effet, comme le programme est multi-processus, avec un processus pour les calculs, et un processus pour l'affichage graphique, les calculs et l'affichage des graphiques s'exécutent chacun sur un processeur indépendant.

Enfin, pour éviter dans tous les cas une utilisation inutile des processeurs, nous avons développé un mode de « mise en veille ». Le suivi des calculs n'est actif que si l'onglet de suivi est visible dans l'interface. Lorsque nous laissons les calculs fonctionner longtemps sans regarder l'interface, il nous suffit de cliquer sur un autre onglet pour masquer et donc désactiver le suivi.

5.5.4 Affichage des tables de transpositions

Nous avons ensuite étendu l'utilisation de l'objet graphique d'affichage des plateaux à la visualisation du contenu des tables de transpositions. Avant l'introduction de cet outil, notre seule méthode de consultation des bases de données calculées était l'affichage direct dans un éditeur de texte. Pour la même raison que le suivi, cette analyse est difficile dans le cas des jeux de plateaux parce que les chaînes de caractères représentant les positions ne sont pas facilement compréhensibles.

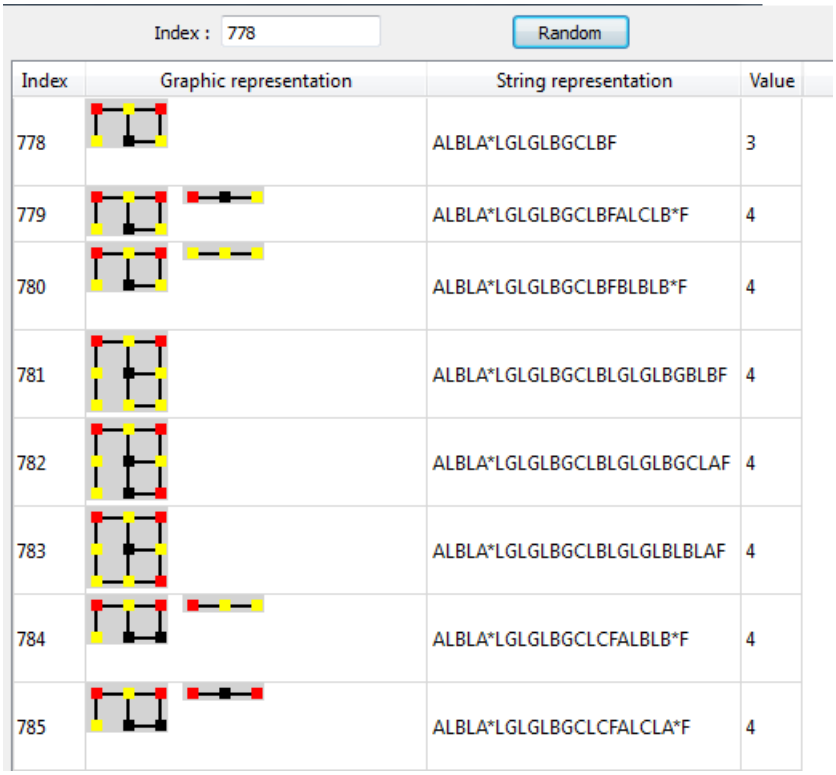


FIGURE 5.10 – Extrait d'une table de transpositions de Dots-and-boxes.

La figure 5.10 montre une capture d'écran d'un extrait de tables de transpositions de Dots-and-boxes. On voit qu'il s'agit des positions 778 à 785 de la table.

L'utilisateur peut indiquer une valeur d'index dans la zone prévue à cet effet pour afficher la partie de la table commençant à cet index, ce qui est utile par exemple pour reconsulter un endroit de la table que l'on avait noté préalablement. L'utilisation des flèches du clavier permet de faire défiler l'affichage de la table.

Enfin, le bouton « Random » permet de se positionner sur un index aléatoire dans la table. Ce bouton est utile lorsque l'on souhaite évaluer rapidement la proportion de tel ou tel type de positions dans la table. On peut par exemple cliquer une vingtaine de fois sur le bouton Random et évaluer le nombre de fois que l'on a rencontré une position du type souhaité pour en déduire une estimation grossière de la proportion de ces positions dans la table.

L'affichage graphique des tables de transpositions a permis de trouver plusieurs idées sur le Dots-and-boxes. Par exemple, cela a révélé la forte proportion de certaines équivalences, que nous avons donc implémentées en priorité. L'idée de déduire le score de positions comportant deux jetons rouges isolés à partir du score de la position sans les jetons isolés vient également d'un examen détaillé des tables de transpositions.

Chapitre 6

Algorithmes de parcours

Nous décrivons dans ce chapitre différents algorithmes de développement des arbres de recherche. L'objectif de ces algorithmes est d'explorer les arbres de jeu, de façon à calculer l'issue de la racine (gagnante ou perdante).

Leur performance peut être jugée sur plusieurs critères. Des critères de rapidité ou d'espace mémoire occupé tout d'abord, car ce sont les points essentiels pour que le calcul se déroule avec succès. Mais on peut également souhaiter minimiser la taille des *arbres solutions* obtenus à la fin du calcul, lorsque l'on utilise des tables de transpositions.

Certains des algorithmes décrits ont été implémentés, mais d'autres n'ont pas dépassé le stade des réflexions théoriques. Il convient d'être prudent concernant leur potentiel, dans la mesure où la validation expérimentale est irremplaçable pour juger de l'efficacité d'un algorithme.

6.1 Algorithme alpha-bêta

6.1.1 Minimax et Négamax

On suppose que l'on étudie un jeu combinatoire, dont le résultat peut être représenté par un nombre, la *valeur* du jeu. Cette valeur doit être d'autant plus grande que le résultat avantage le premier joueur. Par exemple, dans le jeu de Go, à la fin d'une partie, ce nombre serait le nombre de points du premier joueur, moins le nombre de points du deuxième. Pour le jeu d'échecs, on pourrait fixer ce nombre à 1 en cas de victoire des blancs, -1 en cas de victoire des noirs, et 0 en cas de match nul.

Une fois la valeur du jeu fixée pour les positions terminales, on peut remonter dans l'arbre de jeu pour déterminer la valeur des autres positions. Étant donné une position particulière, sa valeur est le résultat d'une partie qui se déroulerait en partant de cette position, si les deux joueurs jouaient parfaitement. En remontant tout l'arbre de jeu, on finit par obtenir la valeur de la racine, donc de la position de départ.

L'algorithme le plus classique pour déterminer la valeur d'une position est l'algorithme récursif *Minimax*, explicité dans l'algorithme 9. L'appel récursif intervient aux lignes 6 et 8. L'algorithme commence par calculer récursivement la valeur du fils $\mathscr{N}_1$, avant de calculer la valeur du fils $\mathscr{N}_2$... et une fois toutes les valeurs connues, on en déduit la valeur de $\mathscr{N}$. Cette méthode conduit à effectuer un parcours en profondeur de l'arbre de jeu : il s'agit d'un algorithme de type *depth-first*.

L'algorithme Minimax prend en compte les intérêts opposés des deux joueurs : le premier joueur a intérêt à jouer vers la position dont la valeur est la plus grande, alors que le deuxième joueur doit au contraire jouer vers la position de valeur la plus petite. Un nœud pour lequel

Algorithme 9 Calcul de la valeur d'un nœud $\mathscr{N}$ avec l'algorithme Minimax.

Si le nœud $\mathscr{N}$ est terminal **alors**
  renvoyer la valeur du nœud $\mathscr{N}$
fin Si
Calculer les fils $\mathscr{N}_i$ de $\mathscr{N}$
Si c'est au tour du premier joueur **alors**
  renvoyer $\max(\text{minimax}(\mathscr{N}_1), ..., \text{minimax}(\mathscr{N}_k))$
sinon
  renvoyer $\min(\text{minimax}(\mathscr{N}_1), ..., \text{minimax}(\mathscr{N}_k))$
fin Si

c'est le tour du premier joueur est ainsi appelé un *nœud Max*, au contraire des *nœuds Min*, lorsque c'est le tour du deuxième joueur.

Cette distinction entre les nœuds Min et Max traduit un schéma de pensée issu de l'étude des jeux partisans : à partir d'une position, les coups disponibles ne sont pas les mêmes pour les deux joueurs. Il est donc nécessaire de tenir compte du tour de jeu pour identifier les positions. Ainsi, dans les arbres de jeu, les étages de hauteur paire (en considérant que la racine est de hauteur 0) sont ceux pour lesquels c'est le tour du premier joueur, et les étages de hauteur impaire sont ceux pour lesquels c'est le tour du deuxième joueur. Il ne peut y avoir de transpositions qu'entre nœuds placés à des étages de même parité.

Cette distinction n'est plus nécessaire dans le cadre des jeux impartiaux, et même, elle s'avère contre-productive. En effet, il est possible d'avoir des transpositions entre des étages de parités différentes, comme la figure 6.1 le montre pour le jeu de Cram. La distinction entre les nœuds Min et Max ne permet pas d'en tenir compte.

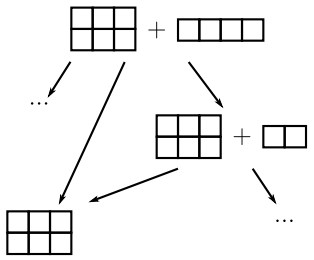


FIGURE 6.1 – Transposition entre étages de parités différentes.

Il est cependant possible de réécrire l'algorithme Minimax sous une forme qui ne différencie pas les nœuds Min et Max, appelée *Négamax*, et qui paraisse plus logique pour l'étude des jeux impartiaux. En ce qui concerne les valeurs des positions, une position perdante aura pour valeur -1, et une position gagnante aura pour valeur 1.

Algorithme 10 Calcul de la valeur d'un nœud $\mathscr{N}$ avec l'algorithme Négamax.

Si le nœud $\mathscr{N}$ est terminal **alors**
  renvoyer la valeur du nœud $\mathscr{N}$
fin Si
Calculer les fils $\mathscr{N}_i$ de $\mathscr{N}$
Renvoyer $\max(-\text{négamax}(\mathscr{N}_1), ..., -\text{négamax}(\mathscr{N}_k))$

C'est cette implémentation que nous avons privilégiée dans notre programme, qui a été initialement développé pour étudier des jeux impartiaux (Sprouts, puis Cram). Elle s'est également avérée utile pour le Dots-and-boxes, qui, bien qu'étant un jeu partisan, peut lui

aussi engendrer des transpositions entre étages de parités différentes, du fait de sa structure presque impartiale (voir la figure 13.10).

Remarquons enfin que même des jeux purement partisans peuvent engendrer de telles transpositions, via un argument de symétrie. C'est le cas du jeu d'échecs : partant de la position de départ, les séquences de coups `1.a4 c5` et `1.c3 a5 2.c4` conduisent à deux positions équivalentes pour les joueurs dont c'est le tour (Blanc dans le premier cas et Noir dans le deuxième).

6.1.2 Élagage alpha-bêta

L'élagage alpha-bêta est une amélioration des algorithmes Minimax et Négamax qui repose sur une remarque : si, pour un nœud Max $\mathscr{N}$, on sait que la valeur du fils $\mathscr{N}_i$ est supérieure ou égale à celle du fils $\mathscr{N}_j$, alors il est inutile de calculer la valeur de $\mathscr{N}_j$ pour obtenir la valeur de $\mathscr{N}$.

Dans le cas des jeux impartiaux, une fois que l'on sait que la valeur d'un fils est -1, c'est-à-dire que ce fils est perdant, on en déduit que négamax$(\mathscr{N}) = 1$, c'est-à-dire que $\mathscr{N}$ est gagnant. Il est donc inutile de calculer les autres fils de $\mathscr{N}$. L'algorithme 11 prend en compte cette idée.

Algorithme 11 Élagage alpha-bêta pour un jeu impartial.

```
Si le nœud N est terminal alors
    renvoyer la valeur du nœud N
fin Si
Calculer les fils N_i de N
Pour chaque fils N_i de N faire
    Si la valeur de N_i est perdant alors
        renvoyer gagnant
    fin Si
fin Pour
Renvoyer perdant
```

L'algorithme alpha-bêta présente l'avantage de nécessiter seulement une petite quantité de mémoire pour fonctionner : à un instant donné, seule la *branche de calcul* est conservée en mémoire[1], c'est-à-dire une liste $\mathscr{N}^0$, $\mathscr{N}^1$, $\mathscr{N}^2$... où $\mathscr{N}^0$ est la racine de l'arbre, et où pour tout i, $\mathscr{N}^{i+1}$ est un certain fils de $\mathscr{N}^i$.

L'adjonction d'une table de transpositions permet souvent d'accélérer le calcul, en échange d'une consommation de mémoire plus importante : on conserve l'issue des nœuds précédemment calculés, ce qui évite d'avoir à les recalculer si on les rencontre ailleurs lors de l'exploration de l'arbre de jeu.

6.1.3 Ordre des options

L'algorithme alpha-bêta montre qu'il n'est nullement nécessaire d'étudier tout l'arbre de jeu d'une position pour déterminer son issue. Sur la plupart des calculs, il permet un gain de plusieurs ordres de grandeur par rapport à l'algorithme Minimax, qui n'est jamais utilisé en pratique.

Le principal levier dont on dispose pour améliorer l'algorithme alpha-bêta est l'ordre dans lequel on étudie les fils des nœuds. Cet ordre influe tant sur la rapidité du calcul que sur la taille de la table de transpositions, et ce, de trois façons. Premièrement, si un nœud $\mathscr{N}$ est gagnant, il vaut mieux étudier en premier un de ses fils perdants (tout fils de $\mathscr{N}$

1. Ainsi que la liste des fils de chaque nœud de la branche de calcul.

calculé gagnant le serait en pure perte). De plus, si $\mathcal{N}$ a plusieurs fils perdants, il vaut mieux étudier en priorité celui dont le sous-arbre est le plus simple. Enfin, si l'on utilise des tables de transpositions, on peut améliorer leur efficacité en dirigeant systématiquement le calcul vers les mêmes zones de l'arbre de jeu, ce qui aura tendance à augmenter le nombre de transpositions rencontrées.

Ainsi, dans chacun des jeux que nous avons étudiés, la détermination de règles générales pour ordonner les fils des nœuds, basées sur les trois points que l'on vient d'énoncer, a souvent été une des premières étapes pour disposer d'algorithmes de parcours performants. L'efficacité de l'algorithme alpha-bêta est maximale lorsque ces règles ne sont jamais prises en défaut. Or, en pratique, c'est rarement le cas : un jeu pour lequel on pourrait prédire quelles sont les positions perdantes serait résolu par méthode. Si l'on ne dispose pas d'une telle résolution, on ne peut être assuré que l'on ordonnera les fils de la meilleure manière.

Des améliorations de l'ordre sont donc souvent possibles. Une première façon de procéder consiste à mener plusieurs calculs basés sur différents ordres, en conservant les modifications qui semblent avoir un effet bénéfique. Cette approche est celle que nous avons adoptée sur le jeu de Dots-and-boxes (voir à ce sujet la section 13.7). En revanche, sur le jeu de Sprouts, quelles que soient les modifications apportées, le calcul semble systématiquement s'engluer sur certaines positions, qualifiées de *bloquantes*, qui correspondent à des zones de l'arbre de jeu dont l'étude est très difficile. Il s'est donc avéré nécessaire de développer une autre approche afin d'éviter autant que possible ces positions bloquantes.

6.1.4 Zappage

Pour faire face aux difficultés induites par les positions bloquantes dans l'étude du Sprouts, nous avons développé une technique intitulée *zappage*, qui consiste à modifier en temps réel les nœuds étudiés par l'algorithme alpha-bêta, afin de le diriger vers d'autres zones de l'arbre de jeu lorsque l'on rencontre une position bloquante.

L'utilisateur dispose d'une interface de suivi qui affiche la liste des nœuds de la branche de calcul (figure 6.2). Un clic sur un de ces nœuds, le *zappage*, conduit à étudier le nœud suivant dans sa fratrie.

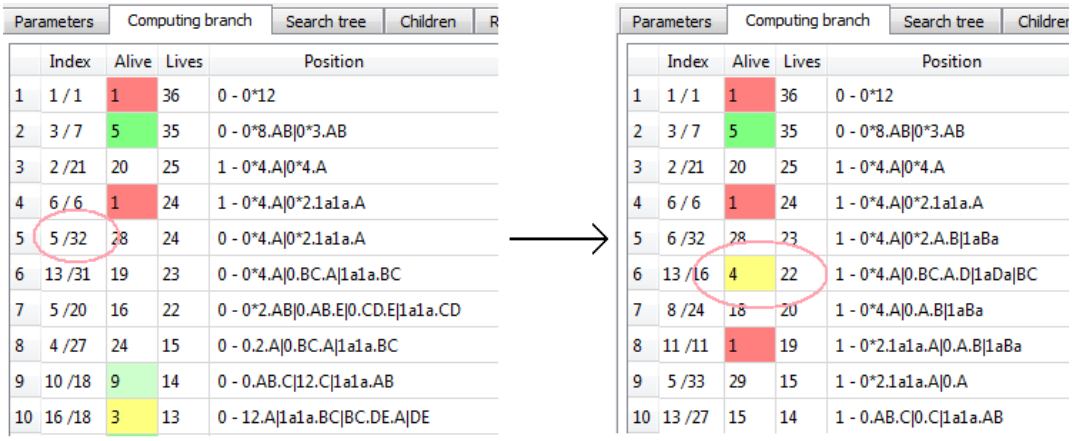


FIGURE 6.2 – Suivi et zappage dans un calcul de Sprouts à 12 points de départ.

Cette méthode constitue ainsi un choix manuel de l'ordre de l'exploration de l'arbre de jeu. L'utilisateur détermine ses choix en fonction des informations que l'interface lui renvoie : dès qu'un nœud semble bloquant, c'est-à-dire que l'algorithme alpha-bêta l'étudie pendant un temps que l'utilisateur considère comme étant exagérément long, il peut décider de le

zapper et de lancer l'étude sur un nœud frère. Il réitère ensuite ce procédé autant qu'il lui semble nécessaire.

Le zappage a donné des résultats probants dans l'étude du Sprouts en version normale, bien meilleurs que n'importe quel ordre que nous avons imaginé. Néanmoins, cette technique est perfectible : sa principale faiblesse est l'utilisateur lui-même, limité par sa condition d'être humain. Ainsi, son intervention dans le parcours de l'arbre est limitée aux étages supérieurs de l'arbre de jeu, l'algorithme alpha-bêta étant trop rapide sur les zones de l'arbre situées en bas. De plus, il peut être sujet à des erreurs d'interprétation, ayant du mal à quantifier ses choix. Et puis, il est évidemment sujet à la fatigue, son temps d'intervention est donc limité.

Or, quand il se sert de l'interface pour zapper, l'utilisateur adopte seulement un petit nombre de comportements différents. Par exemple, si l'étude montre que la position de départ est probablement perdante, il va chercher des positions perdantes aux étages de hauteur paire, et des positions gagnantes aux étages de hauteur impaire. L'automatisation des comportements de l'utilisateur permettrait de remédier aux faiblesses que nous venons d'énoncer. L'algorithme PN-search, que nous étudions dans la section suivante, répond bien à cette problématique.

6.2 PN-search

6.2.1 Principe

Le Proof-number search est un algorithme développé au début des années 1990 par Victor Allis [1]. Au contraire de l'algorithme alpha-bêta, il ne s'agit pas d'un algorithme de type depth-first : le PN-search cherche à étudier en priorité les nœuds dont l'étude semble la plus prometteuse, on parle d'un algorithme de type *best-first*.

Dans la logique de ce que nous avons fait avec l'algorithme alpha-bêta, nous allons donner une présentation du PN-search adaptée aux jeux impartiaux. Ce n'est pas la présentation classique, mais elle nous paraît plus adaptée aux jeux que nous avons étudiés, et c'est là encore celle que nous avons implémentée.

À chaque nœud, nous associons un couple $(p; d)$ de nombres de $[0; \infty]$, le *proof-number* p et le *disproof-number* d.

Le proof-number p traduit une estimation de la difficulté de la tâche consistant à *prouver* le nœud, c'est-à-dire démontrer qu'il est gagnant. Une valeur petite signifie qu'il semble facile de prouver le nœud, et une valeur grande signifie à l'inverse qu'il semble difficile de prouver le nœud. Les extremums sont atteints lorsque l'issue du nœud est connue : $p = 0$ signifie que le nœud est gagnant, et $p = \infty$ signifie que le nœud est perdant.

De même, le disproof-number d traduit une estimation de la difficulté de la tâche consistant à *déprouver* le nœud, c'est-à-dire démontrer qu'il est perdant. $d = 0$ signifie que le nœud est perdant, et $d = \infty$ que le nœud est gagnant.

Les valeurs p et d sont déterminées en partant des feuilles, puis en remontant l'arbre. Chaque feuille est initialisée à une certaine valeur qui sera discutée au paragraphe suivant, quant aux règles de remontée des valeurs, elles sont une conséquence logique des relations entre nœuds gagnants et perdants.

Soit $\mathscr{N}$ un nœud, de valeurs $(p; d)$. Ses fils $\mathscr{N}_1$, ..., $\mathscr{N}_k$ sont de valeurs respectives $(p_1; d_1)$, ..., $(p_k; d_k)$. $\mathscr{N}$ est gagnant si et seulement si un de ses fils est perdant, le coût de la preuve de $\mathscr{N}$ est donc celui de son fils le moins cher à déprouver. On a donc :

Règle 3. $p = \min(d_1, ..., d_k)$

Inversement, $\mathscr{N}$ est perdant si et seulement si tous ses fils sont gagnants. Le coût de la démonstration de la nature perdante de $\mathscr{N}$ est la somme des coûts des preuves de ses fils.

Règle 4. $d = p_1 + ... + p_k$

6.2.2 Initialisation des feuilles

Dans la version classique de l'algorithme PN-search, les feuilles sont initialisées à $(1;1)$. De cette façon, quel que soit le nœud de l'arbre, p représente le nombre minimal de feuilles à calculer pour prouver le nœud, et d, le nombre minimal de feuilles à calculer pour le déprouver.

Une variante fréquemment rencontrée consiste à initialiser les feuilles avec une heuristique qui dénote la difficulté de la démonstration de la position. Ainsi, une position qui semble difficile à calculer sera initialisée avec des valeurs plus grandes qu'une position qui semble facile à calculer ; et une position qui semble perdante sera initialisée avec une valeur de p plus grande que d.

Dans le cas du Sprouts, les quelques essais d'heuristiques menés n'ont pas été concluants. L'initialisation classique semble en fait ne pas si mal fonctionner, dans la mesure où beaucoup de nœuds se font calculer immédiatement par l'intermédiaire d'un découpage.

6.2.3 Développement de l'arbre

Au fur et à mesure que le calcul progresse, l'arbre de recherche évolue. L'algorithme PN-search procède par étapes, et à chaque étape :

* il détermine quelle feuille est la plus intéressante à calculer.
* il la *développe*, ce qui signifie qu'il calcule ses fils, puis qu'il les intègre à l'arbre de recherche.
* il met à jour les valeurs des nœuds de l'arbre en tenant compte des valeurs des nouveaux nœuds.

La feuille que l'algorithme décide de développer est appelée *most-proving node*. Pour la déterminer, partons de la racine. Si cette racine est perdante, alors il faut calculer tous ses fils ; mais si elle est gagnante, il suffit de montrer qu'un de ses fils est perdant. On choisit en priorité celui qui semble le plus simple à étudier, c'est-à-dire celui dont le disproof-number est le plus petit. Le même raisonnement peut s'appliquer au fils choisi, et ainsi de suite jusqu'à obtenir une feuille, qui se trouve être le most-proving node.

En résumé, le most-proving node s'obtient à partir de la racine en choisissant systématiquement le fils qui a le plus petit disproof-number. Il est possible de démontrer que le most-proving node est à la fois dans l'ensemble de feuilles de coût minimal permettant de prouver la racine, et dans l'ensemble de feuilles de coût minimal permettant de déprouver la racine, ce qui justifie de se concentrer sur son étude [37].

Une fois les fils du most-proving node calculés, et leurs valeurs initialisées, on met à jour les valeurs $(p;d)$ des nœuds en remontant l'arbre de recherche. On utilise pour cela les règles 3 et 4. Seuls sont concernés les ancêtres du most-proving node. Si, lors de la remontée, on rencontre un ancêtre pour lequel le développement du most-proving node ne change pas le couple $(p;d)$, alors il est inutile de continuer à remonter l'arbre, et pour déterminer le most-proving node de l'étape suivante, on peut repartir de ce nœud plutôt que de la racine pour gagner un peu de temps de calcul.

6.2.4 Faiblesses

Le PN-search a deux faiblesses principales. La première est sa forte consommation mémoire, le PN-search ayant besoin de stocker beaucoup de nœuds dans l'arbre de recherche pour s'orienter convenablement dans l'arbre de jeu. De nombreuses techniques ont été développées pour remédier à ce problème. On peut citer entre autres :

* La suppression des branches obsolètes. Une fois qu'un nœud a été calculé, on peut supprimer tout son sous-arbre, ce qui libère de la mémoire.

* Delete-least-proving-node. On fixe une limite au nombre de nœuds, et lorsque cette limite est atteinte, on supprime les nœuds qui seront le moins probablement développés dans la suite de la recherche.
* L'algorithme PN^2 initialise chaque feuille en utilisant pendant un temps limité l'algorithme PN-search sous cette feuille. Une fois le délai écoulé, on initialise la feuille à l'aide des valeurs renvoyées par le PN-search, puis on supprime l'arbre de recherche sous la feuille. L'algorithme PN^2 permet au PN de disposer de plus d'informations pour s'orienter, du fait de la meilleure évaluation des feuilles, en échange d'un temps de calcul plus important.

La deuxième faiblesse est que le PN-search est un algorithme qui a été développé pour les arbres, il ne tient pas compte des transpositions. Ceci induit un double problème. Tout d'abord, le PN-search détermine mal le most-proving node, car il ne tient pas compte du fait qu'une feuille peut apparaître en plusieurs endroits de l'arbre, ce qui augmente son importance. Et ensuite, si l'on vient de calculer un nœud dans un endroit de l'arbre, et que ce nœud est une transposition, on ne s'en rend pas compte dans les autres endroits de l'arbre où ce même nœud intervient.

La solution de ce problème est bien expliquée dans [37]. Martin Schijf décrit tout d'abord une solution théorique qui permet de déterminer le most-proving node d'un arbre de recherche comprenant des transpositions, à l'aide de formes normales disjonctives. Cependant, cette solution théorique est d'une complexité bien trop importante pour que son implémentation soit utile. Il décrit donc ensuite un algorithme pratique imparfait, mais utilisable, qui améliore nettement les performances du PN-search par rapport à une version qui ne tiendrait pas compte des transpositions.

6.3 Intervention dans le PN-search

L'algorithme PN-search que nous avons développé dans notre programme est encore assez rudimentaire. Parmi les idées classiques décrites dans le paragraphe 6.2.4, nous n'avons implémenté pour l'instant que la suppression des branches obsolètes. Néanmoins, nous avons programmé ou imaginé certaines méthodes, notamment en faisant usage de l'interface graphique (de la même façon que pour l'algorithme alpha-bêta avec le zappage).

6.3.1 Redémarrage

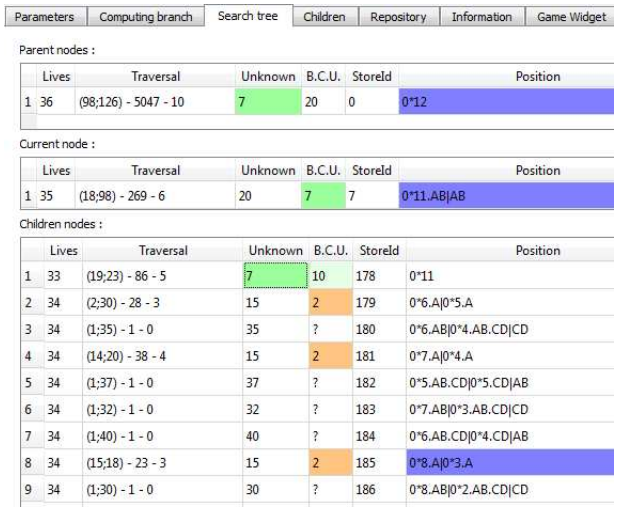
Un calcul typique de Sprouts en utilisant l'algorithme PN-search sature la mémoire en 24 heures sur nos ordinateurs de bureau. Cette saturation est le fait de l'arbre de recherche, bien plus que de la table de transpositions, qui n'utilise qu'une petite fraction de la mémoire disponible.

Ainsi, la première méthode élémentaire que nous avons utilisée pour contrer le problème de la saturation mémoire du PN-search est tout simplement le redémarrage de l'algorithme. Ce faisant, l'arbre de recherche est vidé, mais on dispose tout de même des informations stockées dans les tables de transpositions, que l'on conserve.

6.3.2 Suivi

Comme pour l'algorithme alpha-bêta, nous avons développé une interface nous permettant de suivre le déroulement de l'algorithme PN-search. Mais contrairement au cas de l'alpha-bêta, l'affichage de la branche de calcul n'est pas pertinent, cette branche changeant en permanence, au fur et à mesure que le most-proving node se déplace dans l'arbre de recherche.

La méthode de suivi choisie est présentée dans la figure 6.3. On affiche les informations détaillées d'un nœud : entre autres, ses valeurs $(p; d)$, le nombre de nœuds dans son sous-arbre, et la hauteur de son sous-arbre (colonne « Traversal »). On affiche également le nœud père (il pourrait y avoir plusieurs nœuds pères si l'on implémentait les transpositions), et la liste des nœuds fils.



	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	36	(98;126) - 5047 - 10	7	20	0	0*12

	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	35	(18;98) - 269 - 6	20	7	7	0*11.AB|AB

	Lives	Traversal	Unknown	B.C.U.	StoreId	Position
1	33	(19;23) - 86 - 5	7	10	178	0*11
2	34	(2;30) - 28 - 3	15	2	179	0*6.A|0*5.A
3	34	(1;35) - 1 - 0	35	?	180	0*6.AB|0*4.AB.CD|CD
4	34	(14;20) - 38 - 4	15	2	181	0*7.A|0*4.A
5	34	(1;37) - 1 - 0	37	?	182	0*5.AB.CD|0*5.CD|AB
6	34	(1;32) - 1 - 0	32	?	183	0*7.AB|0*3.AB.CD|CD
7	34	(1;40) - 1 - 0	40	?	184	0*6.AB.CD|0*4.CD|AB
8	34	(15;18) - 23 - 3	15	2	185	0*8.A|0*3.A
9	34	(1;30) - 1 - 0	30	?	186	0*8.AB|0*2.AB.CD|CD

FIGURE 6.3 – Affichage des informations d'un nœud de l'arbre de recherche.

Un clic sur un nœud père ou fils affiche les informations de ce nœud. De cette manière, on peut naviguer dans l'arbre de recherche. Ces opérations peuvent se dérouler soit pendant que le calcul est mis en pause, soit pendant qu'il est en train de se dérouler, et que les diverses informations se mettent à jour sous nos yeux.

6.3.3 Blocage

La principale technique d'interaction que nous avons mise au point est appelée *blocage*. Un clic dans une zone dédiée de l'interface conduit à bloquer le PN-search sur le nœud choisi. Le calcul se poursuit comme si ce nœud était la racine de l'arbre de recherche, jusqu'à soit qu'il soit calculé (que l'issue soit gagnante ou perdante), soit que l'on décide d'interrompre le blocage (auquel cas les valeurs $(p; d)$ des ancêtres du nœud sont mises à jour en tenant compte des nouvelles valeurs du nœud).

Cette méthode permet de focaliser l'étude sur un point précis de l'arbre de recherche que l'on estime important pour la démonstration, sans que l'algorithme ne se disperse. Elle est équivalente à peu de choses près à couper le calcul, puis le redémarrer sur la position choisie, l'interface permettant juste de réaliser cette opération de manière plus aisée.

L'inconvénient principal, c'est que l'algorithme PN-search n'est pas capable de remettre notre choix en question de lui-même. Ainsi, si on laisse le calcul tourner trop longtemps sur un nœud mal choisi, cela peut s'avérer contre-productif (perte de temps de calcul et d'espace mémoire).

6.3.4 Coefficients sur les nœuds internes du PN-search

Pour éviter de privilégier exagérément un seul nœud de l'arbre de recherche, on pourrait introduire un système de favorisation de certains noeuds par rapport à d'autres, plutôt que de blocage. Idéalement, il faut pouvoir favoriser certains noeuds par rapport à d'autres,

tout en gardant l'aspect essentiel du PN-search qui consiste à remettre en permanence en question les choix de parcours. Si l'étude d'un nœud favorisé s'avère en définitive trop difficile, l'algorithme doit pouvoir de lui-même partir explorer d'autres zones de l'arbre.

La solution envisagée se décrit simplement : il suffit de multiplier les valeurs $(p;d)$ du nœud que l'on souhaite favoriser, et juste celui-ci, par un coefficient k, c'est-à-dire que l'on calcule les valeurs $(p;d)$ de ce nœud avec la variante suivante des règles 3 et 4 : $p = k \times \min(d_1, ..., d_k)$ et $d = k \times (p_1 + ... + p_k)$.

Rien d'autre n'est changé dans l'algorithme, en particulier, on continue à déterminer le most-proving node de la même façon. Décrivons le comportement qu'adopte l'algorithme en fonction du coefficient k :

* $k = 0$ correspond au blocage, avec une légère différence : si l'on donne le coefficient au nœud alors qu'il n'appartient pas à la branche de calcul, il faut attendre que le calcul revienne dessus pour que le blocage soit effectif. Ensuite, le blocage perdure jusqu'à ce que le nœud soit calculé.
* $k = 1$ correspond au PN-search classique.
* $0 < k < 1$ favorise d'autant plus le nœud que k est proche de 0.
* $k > 1$, au contraire, lèse ce nœud par rapport aux autres.

Par exemple, dans le cas où l'on initialise les feuilles à $(1;1)$, choisir $k = 0{,}5$ équivaut à considérer que chaque feuille en dessous du nœud est initialisée à $(0{,}5;0{,}5)$, i.e. qu'elle est deux fois plus facile à calculer que les autres.

La problème qui se pose ensuite est la détermination des nœuds pour lesquels on applique un coefficient, et lequel. Là encore, on pourrait le déterminer interactivement via l'interface graphique, ce qui permettrait dans un premier temps de se faire une idée de l'effet de cette technique sur le développement de l'arbre de recherche.

Dans un second temps, il serait possible de faire intervenir une méthode automatique, basée sur des heuristiques. En particulier, nous avons imaginé une méthode basée sur les observations que nous avons faites de l'évolution des couples en cours de calcul. Elle consiste à favoriser d'autant plus un nœud que le quotient $\frac{p}{d}$ est grand, ce qui signifie que le nœud a de grandes chances d'être perdant.

En effet, un nœud de couple $(13;546)$ est un nœud pour lequel il faut calculer au minimum 13 feuilles pour démontrer qu'il est gagnant, et 546 pour démontrer qu'il est perdant. Il est donc probable que ce soit un nœud gagnant, et on a $\frac{p}{d} = \frac{1}{42}$. À l'inverse, si son couple est $(546;13)$, ce nœud a de grandes chances d'être perdant, et $\frac{p}{d} = 42$.

Une fonction du type $k = \mathrm{e}^{-\frac{p}{d}}$ conduirait k à être proche de 1 pour un nœud probablement gagnant, et proche de 0 pour un nœud probablement perdant.

6.4 Adaptation du PN-search aux calculs de nimber

Nous discutons dans cette section de l'implémentation du PN-search dans le cas de calculs de nimbers. Ces calculs, adaptés à l'étude de jeux impartiaux découpables en version normale, ont été présentés dans le chapitre 3. Or l'algorithme PN-search a été développé pour des calculs classiques d'issue, si bien que l'introduction des nimbers pose divers problèmes techniques.

Nous décrivons deux méthodes d'implémentation dans cette section. La première, la plus simple à mettre en œuvre, est celle qui est utilisée dans notre programme. La seconde, plus technique, n'a pas été implémentée, mais contient plusieurs améliorations potentielles.

6.4.1 Méthode 1 : un nœud par couple (position, nimber)

Méthode

Comme expliqué dans le paragraphe 3.6.1, nous pouvons associer un nœud différent à chaque couple ($\mathscr{P}$, $\mathbb{n}$) (où $\mathscr{P}$ est une position, et $\mathbb{n}$ un nimber). Il va nous falloir adapter les algorithmes 4, 5 et 6 du chapitre 3 pour calculer les valeurs $(p;d)$ de chacun de ces nœuds.

L'algorithme 4 ne pose aucun problème, on utilise les règles classiques 3 et 4 en vigueur dans les calculs d'issues. Le cas problématique intervient lorsque l'on rencontre une position découpable [2], dans un nœud du type ($\mathscr{P}_1 + \mathscr{P}_2$, $\mathbb{n}$).

Un tel nœud a deux fils d'un type un peu particulier. Le fils ($\mathscr{P}_1$, $\mathbb{0}$), une fois calculé gagnant, devient ($\mathscr{P}_1$, $\mathbb{1}$), puis ($\mathscr{P}_1$, $\mathbb{2}$)... en vertu de l'algorithme 5. L'incrémentation du nimber continue jusqu'à ce que l'on trouve la valeur $\mathbb{n}_1$ pour laquelle ($\mathscr{P}_1$, $\mathbb{n}_1$) est perdante, à savoir le nimber de $\mathscr{P}_1$. Le même algorithme s'applique au fils ($\mathscr{P}_2$, $\mathbb{0}$). Puis, une fois l'un de ces deux nimbers calculés, par exemple $\mathbb{n}_1$, l'algorithme 6 le fusionne dans le nœud somme ($\mathscr{P}_1 + \mathscr{P}_2$, $\mathbb{n}$) qui devient ($\mathscr{P}_2$, $\mathbb{n} + \mathbb{n}_1$) (la somme des nimbers étant effectuée avec la Nim-addition).

La question se pose alors de savoir comment l'algorithme PN-search peut calculer les valeurs $(p;d)$ du nœud somme ($\mathscr{P}_1 + \mathscr{P}_2$, $\mathbb{n}$). Il n'est malheureusement pas possible de donner des valeurs traduisant à coup sûr la difficulté du calcul du nœud somme. En effet, si $\mathscr{P}_1$ et $\mathscr{P}_2$ sont de nimber $\mathbb{2}$, alors au début du calcul, les deux nœuds fils sont ($\mathscr{P}_1$, $\mathbb{0}$) et ($\mathscr{P}_2$, $\mathbb{0}$), et on ne peut évaluer la difficulté de la démonstration de ($\mathscr{P}_1$, $\mathbb{2}$) et ($\mathscr{P}_2$, $\mathbb{2}$) qui seront pourtant nécessaires pour calculer le nœud somme.

Faute de mieux, nous posons $p = d = \min(p_1;d_1) + \min(p_2;d_2)$, qui sont des minorants des valeurs réelles. En conséquence, le PN-search aura tendance à se diriger trop souvent vers le nœud somme, en sous-estimant la difficulté de ce nœud. Enfin, lors de la recherche du most-proving node, on se dirige vers le fils du nœud somme pour lequel la valeur de $\min(p_i;d_i)$ est minimale.

Inconvénients

Cette méthode présente l'avantage d'être simple à programmer, et de n'avoir que peu de différences avec l'algorithme classique du PN-search. Mais elle présente aussi un certain nombre d'inconvénients.

Outre des valeurs $(p;d)$ peu réalistes pour un nœud somme, un autre problème apparaît lors du calcul de l'issue d'un couple ($\mathscr{P}$, $\mathbb{n}$) avec $\mathbb{n} > \mathbb{0}$. Dans ce cas, l'arbre de recherche contient $n+1$ nœuds admettant $\mathscr{P}$ comme partie position. Pour chacun de ces nœuds, nous devons calculer ses fils ; or le calcul des options des positions de Sprouts est particulièrement coûteux en temps de calcul, il est donc préférable de ne pas l'effectuer plusieurs fois.

De plus, si $\mathbb{n}$ est grand, il y a de nombreuses transpositions entre les nœuds admettant une même partie position. Par exemple, ($\mathscr{P}$, $\mathbb{0}$) est le fils d'à la fois ($\mathscr{P}$, $\mathbb{1}$), ($\mathscr{P}$, $\mathbb{2}$)... Or nous avons vu au paragraphe 6.2.4 que le PN-search a du mal à prendre en compte les transpositions.

La méthode décrite au paragraphe suivant a pour objectif de résoudre ce problème de la multiplication des nœuds et des transpositions, en n'associant qu'un seul nœud à chaque position rencontrée.

2. Nous ne traitons ici que le cas d'une somme de deux positions indépendantes. Le cas de trois termes ou plus n'est pas beaucoup plus compliqué, et en pratique, il n'intervient que très rarement.

6.4.2 Méthode 2 : un nœud par position

(dis)proof-numbers généralisés

Étant donné une position $\mathscr{P}$, nous allons définir les nombres $p_n(\mathscr{P})$ et $d_n(\mathscr{P})$ pour $n \geq 0$. Ces nombres vont permettre d'évaluer la difficulté de la démonstration du fait que la position $\mathscr{P}$ admette ou n'admette pas $\mathbb{n}$ comme nimber. Ainsi, à l'intérieur d'un nœud, nous stockerons à la fois la position $\mathscr{P}$, et les valeurs $p_n(\mathscr{P})$ et $d_n(\mathscr{P})$.

* $p_n(\mathscr{P})$ est le proof-number de $\mathscr{P} + \mathbb{n}$. Il représente le coût de la démonstration de $\mathscr{P} \nsim \mathbb{n}$.
* $d_n(\mathscr{P})$ est le disproof-number de $\mathscr{P} + \mathbb{n}$. Il représente le coût de la démonstration de $\mathscr{P} \sim \mathbb{n}$.

Il pourrait sembler étonnant d'associer « proof » à $\nsim$ et « disproof » à $\sim$. Cependant, dans la définition classique, « proof » est associée à une position gagnante, et « disproof » à une position perdante. Ici, $\mathscr{P} + \mathbb{n}$ est prouvé $\Leftrightarrow$ $\mathscr{P} + \mathbb{n}$ est gagnante $\Leftrightarrow$ $\mathscr{P} \nsim \mathbb{n}$, et $\mathscr{P} + \mathbb{n}$ est déprouvé $\Leftrightarrow$ $\mathscr{P} + \mathbb{n}$ est perdante $\Leftrightarrow$ $\mathscr{P} \sim \mathbb{n}$. Ainsi, notre définition est cohérente avec la définition usuelle.

Si l'on prend $n = 0$, on remarque que $p_0(\mathscr{P})$ et $d_0(\mathscr{P})$ sont les proof et disproof-numbers classiques, car $\mathscr{P} \nsim \mathbb{0} \Leftrightarrow \mathscr{P}$ est gagnante, et $\mathscr{P} \sim \mathbb{0} \Leftrightarrow \mathscr{P}$ est perdante.

Valeurs possibles

Comme avec les nombres classiques, $(p_n(\mathscr{P}); d_n(\mathscr{P})) = (0; \infty)$ signifie que l'on a démontré que $\mathscr{P} \nsim \mathbb{n}$, et $(p_n(\mathscr{P}); d_n(\mathscr{P})) = (\infty; 0)$ signifie que l'on a démontré que $\mathscr{P} \sim \mathbb{n}$.

Il y a une limite naturelle aux valeurs de n pour lesquelles il est utile de calculer $(p_n(\mathscr{P}); d_n(\mathscr{P}))$. Si $\mathscr{P}$ a k fils, on sait par la proposition 5 (du chapitre 3) que son nimber est $\leq \mathbb{k}$, et donc que $(p_n(\mathscr{P}); d_n(\mathscr{P})) = (0; \infty)$ pour $n > k$. Ainsi, l'algorithme peut fonctionner sans qu'il soit nécessaire de stocker une infinité de valeurs dans les nœuds.

Une seule relation lie les p_i et les d_j d'une même position :

Proposition 11. *Si $i \neq j$, alors $p_i \leq d_j$.*

En effet, si $\mathscr{P} \sim \mathbb{j}$, alors $\mathscr{P} \nsim \mathbb{i}$.

Par contre, on peut avoir $p_i > d_i$ (par exemple, si l'on a démontré $\mathscr{P} \sim \mathbb{i}$, alors $p_i(\mathscr{P}) = \infty$ et $d_i(\mathscr{P}) = 0$).

On peut également avoir $i > j$ et $p_i < p_j$ (par exemple si l'on a démontré $\mathscr{P} \sim \mathbb{j}$), ou bien $i > j$ et $d_i < d_j$ (par exemple si l'on a démontré $\mathscr{P} \sim \mathbb{i}$).

Règles de remontée des valeurs

Nous généralisons ici les règles 3 et 4 qui régissent la remontée des valeurs $(p; d)$ depuis les feuilles jusqu'à la racine de l'arbre de recherche. On notera $\vee$ l'opérateur minimum. De la règle 2, on déduit la relation suivante :

Règle 5. *Soit $\mathscr{P}$ une position dont les fils sont $\mathscr{P}_1$, ..., $\mathscr{P}_k$.*
Pour tout $n \geq 0$, on a :

$$p_n(\mathscr{P}) = \left(\bigvee_{i=0}^{n-1} d_i(\mathscr{P}) \right) \vee \left(\bigvee_{j=1}^{k} d_n(\mathscr{P}_j) \right)$$

Cette formule n'engendre aucune difficulté particulière. Il n'en est pas de même pour la suivante, qui découle de la règle 1 :

Règle 6. *Soit $\mathscr{P}$ une position dont les fils sont $\mathscr{P}_1$, ..., $\mathscr{P}_k$.*
Pour tout $n \geq 0$, on a :

$$d_n(\mathscr{P}) = \bigvee_{(x_0;\ldots;x_{n-1})\subset[\![1;k]\!]} \left\{ \left(\sum_{i=0}^{n-1} d_i(\mathscr{P}_{x_i}) \right) + \left(\sum_{j\in[\![1;k]\!]\setminus(x_0;\ldots;x_{n-1})} p_n(\mathscr{P}_j) \right) \right\}$$

$d_n(\mathscr{P})$ se calcule donc en faisant le minimum de $n!C_k^n$ éléments, chaque élément étant une somme de k termes. Néanmoins, on peut écrire cette somme différemment pour simplifier le calcul. Si l'on pose $S = \sum_{i=1}^{k} p_n(\mathscr{P}_i)$, il vient :

$$d_n(\mathscr{P}) = S + \bigvee_{(x_0;\ldots;x_{n-1})\subset[\![1;k]\!]} \left\{ \sum_{i=0}^{n-1} d_i(\mathscr{P}_{x_i}) - p_n(\mathscr{P}_{x_i}) \right\}$$

Calculer d_n se ramène donc au problème suivant : on dispose de n lignes de k nombres, avec $n \leq k$. On doit choisir un nombre par ligne, mais ne pas choisir plus d'un nombre par colonne. Comment choisir ces nombres de façon à minimiser leur somme ? Il s'agit du principal problème combinatoire à régler pour implémenter efficacement cette deuxième méthode.

Cas des sommes de positions

Contrairement au cas de la première méthode, il est possible de calculer les valeurs d'un nœud somme.

Règle 7. *Si $\mathscr{P} = \mathscr{P}_1 + \mathscr{P}_2$:*

$$d_n(\mathscr{P}) = \bigvee_{i=0}^{+\infty} d_i(\mathscr{P}_1) + d_{i\oplus n}(\mathscr{P}_2)$$

$$p_n(\mathscr{P}) = \bigvee_{i=0}^{+\infty} (d_i(\mathscr{P}_1) + p_{i\oplus n}(\mathscr{P}_2)) \vee (p_{i\oplus n}(\mathscr{P}_1) + d_i(\mathscr{P}_2))$$

Remarquons qu'en pratique, on ne calcule le minimum que d'un nombre fini de valeurs pour la raison évoquée précédemment.

Chapitre 7

Vérification

Dans le cadre de cette thèse, nous avons mené de nombreux calculs, nécessitant souvent un temps de calcul important, et trop complexes pour que leur résultat soit vérifié manuellement. Qui plus est, l'écriture de notre programme a nécessité des dizaines de milliers de lignes de codes. On peut dès lors craindre que des erreurs de programmation, ou des erreurs dues à une mauvaise exécution du code suite à une défaillance matérielle, aient conduit certains de nos résultats à être faux. Dans ce chapitre, nous allons décrire les techniques que nous avons mises en œuvre de sorte à nous prémunir du mieux possible de ces erreurs. Cependant, nous allons voir que nos objectifs initiaux ont été étendus, puisque certaines de ces techniques ont comme application la détermination d'arbres solutions de taille réduite.

7.1 Calcul des options

Nous pouvons schématiquement séparer le travail effectué par notre programme, lorsqu'il étudie des jeux combinatoires, en deux parties bien distinctes. La partie « calcul des options » est chargée de calculer les options d'une position, et la partie « recherche » détermine l'issue d'une position en étudiant son arbre de jeu. La plus grosse partie de ce chapitre concerne la partie « recherche », mais dans cette section, nous allons discuter de la validité de la partie « calcul des options ».

En fait, nous pouvons subdiviser cette partie en autant de composantes qu'il y a de jeux étudiés, chaque composante étant chargée de calculer les options d'une position pour un jeu particulier [1].

7.1.1 Erreurs potentielles

Pour chaque jeu étudié, la partie « calcul des options » peut être divisée en deux sous-parties, la première étant chargée de la génération basique des options, et la seconde, de la *canonisation* de ces options (procédé qui permet d'identifier les positions qui engendrent les mêmes parties, voir le paragraphe 2.5.1).

La simple génération des options peut être très simple ou très compliquée suivant le jeu étudié. Pour le Cram ou le Dots-and-boxes, les règles du jeu sont si simples qu'il est quasiment impossible que le code concerné contienne une erreur. À l'inverse, le Sprouts, en particulier sur les surfaces, nécessite des fonctions très complexes, et leur vérification est un problème sérieux.

1. Ces composantes ne sont pas indépendantes, dans la mesure où nous avons programmé les jeux de plateau selon une base commune.

La partie canonisation, elle, est rarement simple. Il faut non seulement produire un code juste, mais ne pas non plus faire d'erreur théorique, c'est-à-dire, éviter d'identifier des positions qui ne soient pas équivalentes. Même sur des jeux de définition simple, la partie relative à la canonisation peut rapidement se complexifier, car le gain dû aux équivalences est souvent suffisamment important pour justifier le recours à des équivalences compliquées. La théorie « Chocolat-Toile-Forêt » présentée en annexe C procure un exemple d'équivalences particulièrement complexes.

7.1.2 Méthodes de vérification

La première méthode de vérification utilisée est la vérification manuelle des tables de transpositions générées par le programme. Par exemple, nous avons vérifié à la main la solution générée par notre programme du fait que la position de Sprouts à 9 points de départ (en version normale) est gagnante. Bien sûr, cela ne prouve en rien la validité du programme, mais c'est une des multiples armes que nous avons pour diminuer la probabilité que nos calculs soient faux.

Une autre méthode, qui fonctionne pour s'assurer de la justesse du programme dans son intégralité, consiste à vérifier la cohérence de nos résultats avec ceux précédemment publiés. Là encore, cette méthode est très imparfaite. Première difficulté, il est rare que les auteurs publient plus que leur résultat. Lorsque nous disposons des bases de données calculées dans leur intégralité, ou au moins d'une partie de ces données (comme c'est le cas avec les données de David Wilson pour le Dots-and-boxes [45]), nous pouvons faire une vérification plus fiable que lorsque nous ne connaissons que l'issue des positions de départ calculées, comme c'est malheureusement le plus souvent le cas. Ce souci de vérification nous a conduit à publier en ligne l'intégralité de nos bases de données, afin de permettre de les confronter aux données d'éventuels futurs autres programmes.

Une autre difficulté réside dans la stabilité des arbres de jeu. Des erreurs mineures n'ont parfois pas un impact suffisant pour changer l'issue des positions de départ. Ainsi, il est possible d'arriver à retrouver le bon résultat, mais par des moyens incorrects. Lors de nos tous premiers calculs concernant le Sprouts, nous trouvions des résultats identiques à [4], en dépit d'erreurs qui furent détectées ultérieurement.

Une méthode idéale de vérification serait le recours à un assistant de preuve. Dans le cas du Sprouts, un tel recours serait loin d'être aisé, quand on voit que les justifications topologiques à l'écriture d'un programme occupent les quelques 350 pages d'un mémoire entièrement dédié au sujet [14], utilisant des résultats, comme le théorème de Jordan, qui sont à la limite des possibilités actuelles des assistants de preuve.

7.2 Parcours de l'arbre de recherche

7.2.1 Erreurs potentielles

On peut distinguer principalement deux types d'erreurs liées aux parcours des arbres de recherche :

* des calculs incomplets.
* des calculs faux.

Un calcul est incomplet si les issues calculées (celle de la racine, et celles des nœuds de son arbre de jeu qui ont été calculées) sont toutes justes, mais qu'une partie de l'arbre de jeu nécessaire pour le calcul n'a pas été étudiée, si bien que la démonstration est incomplète. Par exemple, une position perdante a été considérée comme telle, alors que seules certaines de ses options ont été calculées gagnantes.

Un calcul est faux si un nœud perdant a été calculé gagnant, ou inversement.

Les sources potentielles d'erreurs sont multiples. On peut bien sûr imaginer des erreurs logicielles, liées par exemple à la complexité du code de certains algorithmes, ou à des techniques de programmation comme le « zappage » (modification par l'utilisateur, via l'interface graphique et en temps réel, de la branche de calcul de l'algorithme depth-first) assez périlleuses à mettre en œuvre. Mais on peut également imaginer des erreurs matérielles : corruption de RAM, rayons cosmiques... le développement de techniques de vérification n'est donc pas superflu.

Cependant, les techniques de vérification que nous expliquerons plus loin, et que nous avons systématiquement utilisées, ont permis de détecter avec une certitude quasi-absolue la totalité des erreurs liées au parcours des arbres de recherche. Nous savons ainsi qu'il y a eu de rares erreurs de ce type dans nos calculs, et le plus souvent, du premier type : c'était une position qui manquait.

Toutes les erreurs n'ont pas été expliquées, mais nous pouvons tout de même livrer un exemple d'erreur récurrente. Lorsque nous modifions les algorithmes de canonisation, les anciennes tables de transpositions deviennent obsolète, car le calcul des options d'une même position ne renvoie plus le même résultat. Ainsi, si dans un premier temps, on a prouvé qu'une position était gagnante grâce à une certaine option perdante, et qu'ensuite, la modification de la canonisation change la représentation de cette option, on ne saura plus prouver que la position est gagnante.

L'erreur intervient suite à une modification involontaire de la canonisation : les tables de transpositions précédemment calculées deviennent alors incomplètes. Il nous est ainsi arrivé qu'une modification mineure de la canonisation rende obsolète une table de transpositions de 30 000 positions pour une unique position. Il est à noter que ce type d'erreurs est en fait lié au calcul des options, même si c'est la vérification de la partie « recherche » qui a permis de la détecter.

7.2.2 Cohérence des tables de transpositions

Nous pouvons présenter une première technique élémentaire de vérification, à même de détecter des calculs faux. Il s'agit de la vérification de la cohérence de plusieurs tables de transpositions, obtenues lors de différents calculs. Sur un calcul classique d'issues, on vérifie que les issues des positions communes à plusieurs tables de transpositions sont égales dans toutes ces tables : si une position s'avère perdante dans une table et gagnante dans une autre, c'est qu'il y a une erreur.

Cette vérification de la cohérence peut s'avérer un peu moins binaire sur d'autres types de calculs. Sur un calcul de nimber, le fait que le couple $(\mathscr{P};\mathbb{n})$ soit perdant prouve que le nimber de la position $\mathscr{P}$ est égal à $\mathbb{n}$. Inversement, si le couple $(\mathscr{P};\mathbb{n})$ est gagnant, alors le nimber de $\mathscr{P}$ est différent de $\mathbb{n}$. Une incohérence peut être détectée si, pour une même position, le nimber est par exemple égal à $\mathbb{1}$ dans une table, et égal à $\mathbb{0}$ dans une autre, ou bien différent de $\mathbb{1}$. Par contre, trouver le nimber différent de $\mathbb{0}$ dans une table et différent de $\mathbb{2}$ dans une autre reste cohérent.

Le même type de phénomène apparaît avec le score dans le cadre du Dots-and-boxes : il est possible d'avoir obtenu que le score d'une position est ≤ 5 dans une table de transpositions et ≤ 6 dans une autre, sans qu'il n'y ait de contradiction.

7.3 Calcul de vérification

Nous présentons dans cette section les algorithmes de vérification qui constituent le point essentiel de ce chapitre. La vérification est un second calcul, qui se mène une fois le calcul terminé. Outre le fait qu'elle règle le problème des erreurs liées au parcours des arbres de

recherche, la vérification a également des applications imprévues que nous détaillerons dans les sections suivantes.

7.3.1 Calcul d'un nœud

Nous commençons par présenter avec l'algorithme 12 un algorithme récursif de calcul d'un nœud qui servira de base au calcul de vérification. Le pseudo-code présenté dans l'algorithme 12 existe quasiment tel quel dans notre programme.

Algorithme 12 Calcul du résultat du nœud $\mathscr{N}$.

```
1: Si le résultat de N existe déjà dans la table de transpositions alors
2:    renvoyer ce résultat
3: fin Si
4: Calculer et ordonner la liste des fils de N
5: Pour chaque fils N_i de N faire
6:    Calculer le résultat de N_i (appel récursif de l'algorithme 12)
7:    Si le résultat de N peut se déduire de celui de N_i alors
8:       Stocker le résultat de N dans la table de transpositions
9:       Renvoyer le résultat de N
10:   fin Si
11: fin Pour
```

L'intérêt de cet algorithme est qu'il reste très général. Il ne détaille pas ce qu'est un nœud $\mathscr{N}$, ni ce que signifie le résultat de $\mathscr{N}$. Il ne détaille pas non plus comment on calcule les fils de $\mathscr{N}$, ou comment on déduit le résultat de $\mathscr{N}$ à partir de l'une (ou plusieurs) de ses fils $\mathscr{N}_i$. Une telle généralité permet au code de cet algorithme d'être commun à tous les algorithmes de type depth-first que nous avons implémentés dans notre programme :

- calcul classique d'issue (gagnante/perdante).
- calcul des jeux impartiaux en version normale avec le nimber.
- calcul misère utilisant les arbres canoniques réduits ([29]).
- calcul complet de l'arbre de jeu, par exemple pour les arbres canoniques.
- calcul du score dans le cas du Dots-and-boxes.

Chaque algorithme spécifique réimplémente les différentes notions apparaissant dans l'algorithme 12. Dans le cas d'un calcul classique de l'issue d'une position, un nœud $\mathscr{N}$ est simplement une position $\mathscr{P}$, le résultat du nœud est l'issue de $\mathscr{P}$, la liste des fils du nœud est la liste des options de $\mathscr{P}$, et le résultat du nœud est gagnant dès qu'un fils perdant est calculée.

Dans le cas d'un calcul de score pour le Dots-and-boxes, le résultat du nœud est toujours une issue, mais le nœud $\mathscr{N}$ est cette fois un couple (position, contrat). Les règles de manipulation de ces couples sont détaillées dans le paragraphe 13.3.2 du chapitre sur le Dots-and-boxes.

Dans le cas du calcul d'un arbre canonique, le nœud $\mathscr{N}$ est une position $\mathscr{P}$, le résultat du nœud est l'arbre canonique de la position $\mathscr{P}$, et ne peut pas se déduire d'un fils seulement. Il ne se déduit que de l'ensemble des résultats des fils, la boucle doit donc arriver à son terme pour que l'on obtienne le résultat.

7.3.2 Calcul de vérification

L'algorithme de vérification que nous allons maintenant décrire est une modification mineure de l'algorithme 12. L'algorithme 12 étant commun à tous les types de calculs menés, l'algorithme de vérification sera mécaniquement lui aussi commun à tous ces calculs. Ainsi,

dans notre programme, nous n'avons pas eu besoin d'implémenter une fonction récursive spécifique pour chaque type de calcul.

Le point essentiel des calculs de vérification est qu'il n'est pas nécessaire de mener à nouveau une recherche dans l'arbre de jeu, car la vérification peut se baser sur les résultats déjà obtenus lors du calcul initial pour se guider dans l'arbre de jeu. Par conséquent, il faudra considérer deux tables de transpositions lors d'un calcul de vérification : celle du calcul initial, et celle contenant les résultats déjà vérifiés, qui va se remplir petit à petit. Dans l'algorithme 13, nous désignerons ces tables par *table-I* et *table-V* respectivement.

Algorithme 13 Vérification du résultat du nœud $\mathscr{N}$.

```
1:  Si le résultat de 𝒩 existe dans table-V alors
2:      renvoyer ce résultat
3:  fin Si
4:  Calculer et ordonner la liste des fils de 𝒩
5:  Vérifier les résultats des fils dans table-I
6:  Réordonner les fils en donnant une priorité aux fils adéquats
7:  Pour chaque fils 𝒩_i de 𝒩 faire
8:      Calculer le résultat de 𝒩_i (appel récursif de l'algorithme 13)
9:      Si le résultat de 𝒩 peut se déduire de celui de 𝒩_i alors
10:         Vérifier que le résultat de 𝒩 est cohérent avec table-I
11:         Stocker le résultat de 𝒩 dans table-V
12:         Renvoyer le résultat de 𝒩
13:     fin Si
14: fin Pour
```

La prise en compte des résultats connus intervient suite au calcul des fils, lignes 5–6. On commence par vérifier, à la ligne 5, que les fils dont le résultat est stocké dans la table-I permettent de déduire le résultat de $\mathscr{N}$. Par exemple, dans le cas d'un calcul classique d'issue, il faut soit disposer d'un fils perdant, soit que tous les fils soient gagnants. Dans le cas contraire, on renvoie un message d'erreur.

Ensuite, ligne 6, on donne la priorité à certains fils adéquats. Ces fils adéquats dépendent de l'algorithme en question, mais dans la majeure partie des cas, il s'agit tout simplement des fils perdants. En donnant la priorité aux fils trouvés perdants dans le calcul précédent, l'algorithme de vérification se dirige immédiatement vers les branches correctes de l'arbre de jeu, et évite des calculs inutiles de fils gagnants.

La vérification à proprement parler intervient ligne 10 : si l'on détecte un nœud dont le résultat n'est pas cohérent avec le calcul précédent, on interrompt le calcul, et l'on renvoie un message d'erreur précisant le problème.

7.3.3 Intérêt de la vérification

Au fur et à mesure du développement de notre programme, nous avons implémenté des techniques d'intervention de l'utilisateur en temps réel pour guider le calcul. Ces techniques ont nettement complexifié le code, et ont parfois engendré de l'instabilité dans le fonctionnement du programme (crashs). Il était donc nécessaire de s'assurer de la validité de nos résultats.

La vérification a permis de résoudre ce problème de manière complètement satisfaisante : la programmation de la boucle de vérification décrite dans l'algorithme 13 n'a nécessité que quelques lignes de code, si bien qu'il est certain que ce code extrêmement simple ne comporte aucune erreur. Peu importe l'instabilité du code des algorithmes de parcours ou d'intervention en temps réel, l'utilisation a posteriori de la vérification permet de valider tout calcul, et même de se prémunir d'éventuels problèmes matériels.

Cette fonctionnalité nous a ainsi permis développer des algorithmes de parcours expérimentaux en toute tranquillité. Concernant la validité de nos calculs, seul subsiste le problème de la justesse des fonctions de calcul des options.

7.3.4 Vérification complète

Lors de l'exécution de l'algorithme 13, les fils se font réordonner. Il s'ensuit que tous les nœuds de la table-I ne sont pas traités par l'algorithme. Même si la vérification effectuée suffit à s'assurer de la justesse du résultat concernant le nœud initial, on peut vouloir vérifier les résultats de tous les nœuds de la table-I.

L'algorithme 14 apporte une réponse à ce problème. On lance une vérification relative à la première entrée de la table-I. Si à la suite de cette vérification, toutes les entrées de la table-I n'ont pas été vérifiées, on relance une vérification sur la première entrée qui ne l'ait pas encore été. On recommence jusqu'à ce que l'intégralité de la table-I ait été vérifiée.

Algorithme 14 Vérification complète d'une table de transpositions.

```
1: Pour chaque nœud 𝒩 de table-I faire
2:   Si le résultat de 𝒩 n'est pas dans table-V alors
3:     exécuter l'algorithme 13 sur 𝒩
4:   fin Si
5: fin Pour
```

7.3.5 Colmatage

Si le calcul est incomplet, mais sans être faux, l'algorithme 13 s'en rend compte ligne 5. Il renvoie alors un message d'information, mais ne s'arrête pas pour autant. L'algorithme depth-first classique continue à s'exécuter, jusqu'à ce que le nœud soit calculé.

De plus, si le nœud qui manque se trouve en milieu d'arbre, les nœuds situés plus bas dans l'arbre précédemment stockés dans table-I continuent de guider le calcul. Ainsi, l'algorithme finit par reboucher le trou, avec l'efficacité d'un algorithme depth-first qui dispose des résultats du calcul précédent pour le guider.

En pratique, nous avons utilisé cette technique de colmatage pour deux tâches bien différentes. Premièrement, nous avons vu au paragraphe 7.2.1 que les changements de canonisation ont pour effet de rendre les tables de transpositions déjà calculées obsolètes. Dans un tel cas, on prend l'ancienne table comme table-I, et on lance l'algorithme 13, si bien que les trous sont rebouchés et que l'on retrouve des tables fonctionnelles. Pour un changement de canonisation mineur, les trous sont peu nombreux, et le depth-first, bien qu'étant un algorithme basique, n'a aucun mal à les colmater. Nous avons utilisé cette méthode avec succès plusieurs fois sur le jeu de Sprouts, pour lequel la représentation des positions a subi de nombreuses évolutions.

L'autre utilisation de cette technique intervient dans le cas du Dots-and-boxes. Le paragraphe 13.4.5 décrit une méthode permettant d'obtenir de manière immédiate le résultat de certains nœuds, ce qui permet de mener le calcul plus rapidement et en utilisant moins de mémoire. Cependant, cette méthode est non constructive : on sait quel joueur dispose de la stratégie gagnante, mais on ne connaît pas cette stratégie.

Il est possible de disposer de l'avantage de cette méthode, sans son inconvénient. On mène le premier calcul avec la méthode non constructive, puis on relance le calcul de vérification sans cette méthode. Ainsi, la vérification rencontre certains nœuds dont on ne connaît plus le résultat, qu'elle colmate selon la technique que l'on vient d'expliquer. Là encore, dans les calculs que nous avons menés, le depth-first n'a pas eu de mal à colmater ces nœuds [2].

2. Il est cependant tout à fait envisageable de rencontrer des cas où le colmatage se fasse plus difficilement,

7.4 Arbre solution

7.4.1 Obtention d'un arbre solution

Nous avons expliqué au paragraphe 7.3.4 que le calcul de vérification ne passait pas en revue toutes les entrées de la table de transpositions du premier calcul. Expliquons ce phénomène.

Nous avons tracé un arbre de jeu sur la figure 7.1. Supposons que l'algorithme de parcours en profondeur étudie les nœuds de cet arbre en commençant par la gauche, les nœuds seront donc étudiés dans l'ordre de leur numérotation.

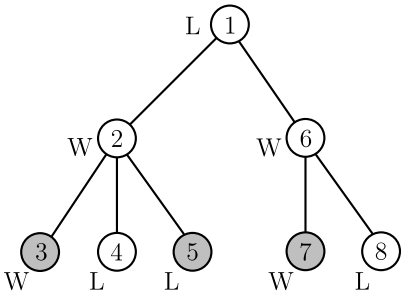


FIGURE 7.1 – Arbre de jeu.

Lors du premier calcul de l'issue de la racine, tous les nœuds sont parcourus, hormis le nœud numéro 5. En effet, l'algorithme détermine l'issue du nœud 3, puis du nœud 4. Comme le nœud 4 est perdant, on en déduit immédiatement que le nœud 2 est gagnant, et il n'est pas nécessaire de connaître l'issue du nœud 5.

Lors du calcul de vérification, on ordonne les nœuds de façon à parcourir les nœuds perdants en premier. Ainsi, les nœuds numéro 3 et 7, qui sont inutiles pour connaître l'issue de la racine, ne seront pas parcourus, et n'apparaîtront pas dans la nouvelle table de transpositions.

La vérification permet ainsi de se débarrasser des nœuds inutiles de la table de transpositions, et fournit des arbres solutions plus compacts. Il est à noter que cette propriété de la vérification est un effet de bord initialement involontaire. Nous l'avons observée lors de nos premiers calculs de vérification, nous demandant même s'il ne s'agissait pas d'une erreur, avant d'expliquer ce comportement. A posteriori, c'est maintenant l'inverse : la plupart des calculs de vérification que nous menons ont pour objectif l'épuration des arbres solutions plutôt que la détection d'éventuelles erreurs.

7.4.2 Applications

Jeu manuel

Cette propriété de la vérification profite en premier lieu au joueur humain. Elle permet d'obtenir des preuves compactes pour des positions de départ suffisamment simples. Dans le cas du Sprouts, notre programme a ainsi permis de déterminer des preuves très compactes pour les positions de départ jusqu'à 11 points. D'une part, ces preuves sont vérifiables en un temps raisonnable par un être humain (moins d'une journée de travail), alors que la preuve manuelle la plus importante publiée jusqu'alors concernait le jeu à 7 points de départ [17]. Et d'autre part, elles contiennent toutes moins de 140 positions perdantes [3], il est donc

du fait de l'efficacité limitée du depth-first pour certains problèmes.

3. Une journée de travail peut être nécessaire pour vérifier 140 positions perdantes, car certaines de ces positions peuvent avoir plusieurs dizaines d'options.

envisageable pour un joueur humain de les apprendre par cœur afin de jouer parfaitement sur papier contre un adversaire.

En guise d'exemple, nous présentons en annexe B un arbre solution de la position de Sprouts à 5 points, déterminé à l'aide de notre programme.

Le recours à l'ordinateur peut paraître superflu pour la détermination de preuves manuelles, et pourtant, il est au contraire essentiel. En effet, certaines preuves fabriquées informatiquement n'auraient probablement pas vu le jour sans programme, car trop difficiles à découvrir, cachées dans la complexité des arbres de jeu. Par contre, une fois les preuves calculées, leur utilisation par des joueurs humains devient possible.

Jeu assisté par ordinateur

Pour des positions de départ plus complexes, il devient difficile pour un joueur humain d'assimiler la totalité de la preuve. Le jeu assisté par ordinateur est alors la solution : si les solutions ont été précalculées, l'ordinateur peut nous fournir les coups de façon à jouer parfaitement les parties, du moment que l'on est en position de force lors du premier coup.

Dans cette optique, la vérification permet de fabriquer des binaires capables de jouer parfaitement les plus petits possible. Par exemple, pour le Sprouts, la solution de l'ensemble des jeux jusqu'à 30 points n'occupe, après compression par un algorithme standard, que 56 Ko.

Aide à la recherche

Par ailleurs, en diminuant la taille des tables de transpositions, la vérification permet de diminuer la consommation de RAM. Cela peut s'avérer utile dans le cas d'un calcul qui serait trop important pour être mené en une seule fois : lorsque la mémoire est proche de la saturation, on utilise un calcul de vérification pour réduire la table de transpositions, puis on reprend le calcul. En pratique, cela nous a été utile pour terminer certains calculs de Dots-and-boxes qui, sans cela, auraient dépassé les capacités mémoire de nos ordinateurs.

Il est également envisageable d'utiliser cette propriété pour le calcul distribué : les différents ordinateurs participant au calcul peuvent ainsi renvoyer des données de taille moins importante à celui chargé de les centraliser.

7.4.3 Performances

Le tableau 7.1 présente les résultats de quelques calculs de vérification. S_8^+ désigne la position de Sprouts à 8 points en version normale, $C_{5\times5}^+$ la position de Cram de taille 5×5 en version normale, et $D_{3\times5-am7}$ la position de Dots-and-boxes américaine de taille 5×5, avec le contrat 7. Le premier calcul a été réalisé avec l'algorithme depth-first, puis on a réalisé une vérification.

On peut observer que le nombre de nœuds supprimés par la vérification n'est jamais négligeable. Il n'est inférieur à 50% que sur les calculs de Sprouts avec un petit nombre de points de départ, sur lesquels l'ordre des fils par défaut fonctionne bien. Remarquons également l'énorme différence de difficulté entre S_{11}^+ et S_{12}^+ : sur le Sprouts, à partir de 12 points de départ, l'ordre par défaut se retrouve systématiquement pris en défaut, et conduit l'algorithme depth-first à explorer des zones de l'arbre beaucoup plus complexes que celles qui conduisent au calcul le plus rapide.

En effet, regardons maintenant le tableau 7.2 qui présente les mêmes résultats, en utilisant cette fois l'algorithme PN-search. Le PN-search permet d'obtenir des tables de transpositions finales plus petites que le depth-first, au prix d'une utilisation mémoire en cours d'algorithme nettement plus importante (pour le stockage de l'arbre de recherche). Le gain sur les tables de S_8^+ à S_{11}^+ n'est pas flagrant, mais il est remarquable sur S_{12}^+ : le calcul de PN-search se termine

Position de départ	Taille de la table avant vérification	Taille de la table après vérification	Pourcentage de nœuds supprimés
S_8^+	468	263	44%
S_9^+	113	81	28%
S_{10}^+	420	284	32%
S_{11}^+	417	279	33%
S_{12}^+	669 321	37 973	94%
$C_{5\times5}^+$	683	164	76%
$C_{5\times7}^+$	76 029	5 968	92%
$D_{3\times5-am7}$	637 271	276 532	57%
$D_{3\times5-am8}$	9 354 626	3 192 441	66%
$D_{4\times4-am8}$	4 519 269	1 932 199	57%
$D_{4\times4-am9}$	7 508 596	1 331 000	82%

TABLE 7.1 – Performance de la vérification sur des calculs réalisés avec l'algorithme depth-first.

en stockant 1 000 fois moins de nœuds que le depth-first dans la table de transpositions, et même 60 fois moins que le depth-first suivi d'une vérification.

Position de départ	Taille de la table avant vérification	Taille de la table après vérification	Pourcentage de nœuds supprimés
S_8^+	359	159	56%
S_9^+	120	73	39%
S_{10}^+	278	215	23%
S_{11}^+	304	144	53%
S_{12}^+	651	372	43%
$C_{5\times5}^+$	252	142	44%
$C_{5\times7}^+$	5 545	1 443	74%

TABLE 7.2 – Performance de la vérification sur des calculs réalisés avec l'algorithme PN-search.

Cela montre que la vérification ne peut pas compenser à elle seule les défauts de l'algorithme de parcours initial. Si l'on cherche à obtenir des tables de transpositions les plus petites possibles, l'algorithme de parcours est l'élément déterminant, la vérification ne venant que dans une seconde étape pour épurer les bases de données obtenues.

Par ailleurs, plus l'algorithme de parcours est performant vis-à-vis de la taille des tables de transpositions finales, et moins la vérification est efficace, car l'algorithme aura tendance à stocker moins de nœuds inutiles. Sur les cas les plus représentatifs des calculs que nous avons mené (S_{12}^+, $C_{5\times5}^+$ et $C_{5\times7}^+$), on constate que la vérification semble, en pourcentage, moins performante après un algorithme de PN-search qu'après un algorithme de depth-first.

Mais même dans le cas d'un algorithme de type PN-search, qui est un des algorithmes qui stocke le moins de nœuds inutiles, la vérification permet tout de même de réduire la taille des tables de transpositions d'un facteur pouvant aller jusqu'à plus de 70%. Cela vient du fait que nos algorithmes de PN-search ne prennent pas en compte les transpositions de façon optimale, mais aussi d'une difficulté intrinsèque des calculs. Les algorithmes de parcours ont en effet des difficultés à calculer les nœuds qui sont des *faux-amis*, c'est-à-dire des nœuds gagnants qui semblent perdants, car ils n'ont en proportion que très peu de fils perdants.

7.5 Suppression des nœuds inutiles

7.5.1 Nœuds inutiles

L'algorithme 13 de vérification ne permet cependant pas de supprimer la totalité des nœuds inutiles de l'arbre. Un problème apparaît dans certains cas de transpositions perdantes. La figure 7.2 montre un exemple d'arbre de jeu où ce phénomène se produit : cet arbre ne diffère de celui de la figure 7.1 que par le nœud 5, qui remplace le nœud 8 à droite de la figure. Ce nœud est donc présent deux fois, c'est une transposition perdante.

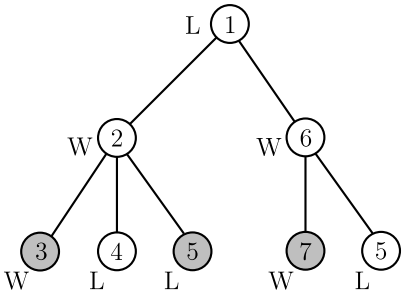


FIGURE 7.2 – Arbre de jeu avec une transposition perdante.

On suppose là encore que l'exploration de l'arbre par l'algorithme depth-first se mène de la gauche vers la droite. Par conséquent, lors du premier calcul, tous les nœuds seront explorés par l'algorithme. En particulier, le nœud 5 ne sera pas rencontré dans la partie gauche de l'arbre, car l'algorithme remonte une fois le nœud 4 calculé perdant, mais il sera calculé dans la partie droite de l'arbre.

La vérification, telle que nous l'avons présentée jusqu'ici, va supprimer les nœuds indiqués en gris. On ne conserve qu'un seul fils perdant pour chaque nœud gagnant, ce qui rend inutile les nœuds 3, 5 (à gauche) et 7. Or, le nœud 5 réapparaît ensuite dans le calcul, si bien qu'il sera présent dans la table de transpositions à la fin de la vérification.

Le problème est le suivant : l'arbre solution obtenu va contenir à la fois le nœud 4 et le nœud 5, alors qu'idéalement le nœud 5 seul pourrait suffire. Le nœud 4 est *inutile* dans notre arbre solution, et le but de cette section sera de supprimer le plus possible de tels nœuds.

7.5.2 Parcours aléatoire

Le problème de la figure 7.2 vient de l'ordre de parcours de l'arbre de jeu lors de la vérification. Si, à gauche, les nœuds 3, 4 et 5 étaient ordonnés 3, 5, 4, ou bien 5, 3, 4, le nœud 5 serait choisi avant le nœud 4, ce qui permettrait d'éliminer correctement le nœud 4 de l'arbre solution.

Il n'y a cependant pas de solution simple pour savoir a priori quel est l'ordre idéal de parcours des nœuds lors de la vérification, d'autant plus que contrairement à l'exemple de la figure, les nœuds 2 et 6 peuvent être très éloignés au sein de l'arbre de jeu. Pour contourner ce problème, nous avons implémenté une solution originale de parcours *aléatoire* de l'arbre solution.

L'algorithme de vérification est inchangé, mais au lieu d'ordonner les nœuds avec l'ordre par défaut du programme, on utilise un ordre aléatoire, avec une fonction *random*. Dans le cas de la figure 7.2, un calcul simple indique que la probabilité d'éliminer le nœud 4 est alors de 75% (il suffit de calculer la probabilité du cas gênant : une chance sur deux de choisir le nœud 2 en premier, suivie d'une chance sur deux de donner la priorité au nœud 4 sur le nœud 5). On peut ensuite augmenter cette probabilité en effectuant plusieurs vérifications

aléatoires d'affilée. Avec 2 vérifications aléatoires consécutives, on passe à 94%, et avec 3 vérifications aléatoires consécutives à plus de 98%.

De façon générale, les séries de vérifications aléatoires permettent de supprimer les nœuds inutiles avec une forte probabilité.

7.5.3 Aléatoire et déterminisme

Sur la figure 7.3, nous avons comparé plusieurs séries de vérifications aléatoires. Nous avons commencé par effectuer un calcul de Sprouts en version normale avec 13 points de départ, qui s'est terminé en 1 446 couples (position, nimber) perdants. Nous avons alors réalisé trois séries de 10 vérifications aléatoires.

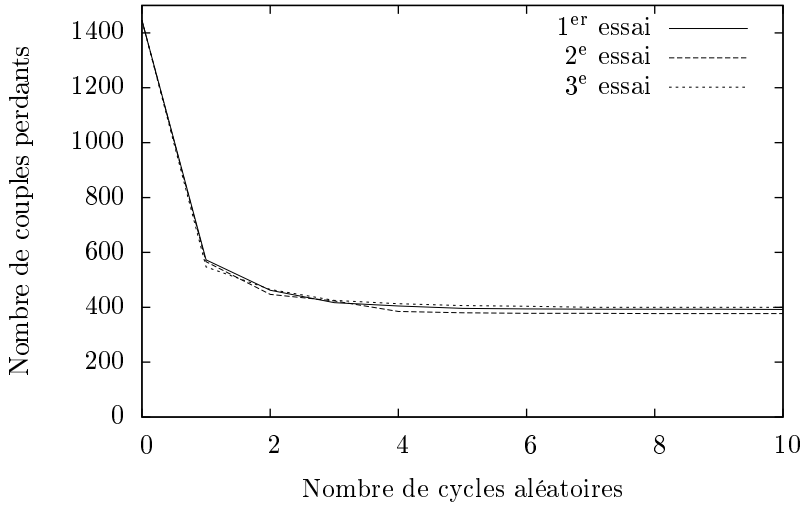


FIGURE 7.3 – Taille de l'arbre solution de S_{13}^{+} en fonction du nombre de vérifications aléatoires.

La figure 7.3 indique le nombre de couples perdants restant dans la table de transpositions en fonction du nombre de vérifications aléatoires réalisées. On remarque tout d'abord que les trois essais mènent à des résultats similaires, et à peu près à la même vitesse. Les graphes des différents essais semblent quasiment superposés à l'échelle de la figure 7.3.

Pourtant, il suffit de zoomer sur la bande de 360 à 480 couples, comme sur la figure 7.4, pour constater que les trois essais sont réellement différents. En particulier, ils semblent converger vers des valeurs légèrement différentes. Ce phénomène peut s'expliquer en imaginant un panier rempli d'objets de diverses formes et tailles. Si l'on demande à différentes personnes de ranger le panier pour le tasser et diminuer son encombrement, on obtiendra des empilements localement différents (l'aspect aléatoire), mais avec une certaine convergence globale.

La figure 7.3 montre que dans le cas étudié, l'essentiel du gain de la vérification aléatoire est obtenu après 3 itérations, et la figure 7.4 montre à une échelle plus fine que la convergence est quasiment atteinte après une dizaine d'itérations. Ces valeurs dépendent du jeu, de la position, et du calcul initial considérés, mais ces ordres de grandeur sont représentatifs.

7.5.4 Performances

Le gain que l'on peut espérer de cette technique est assez modeste par rapport à la vérification classique : on gagne quelques pourcents sur la taille de la table si l'on effectue une vérification aléatoire après la vérification classique, et le gain maximal que l'on peut

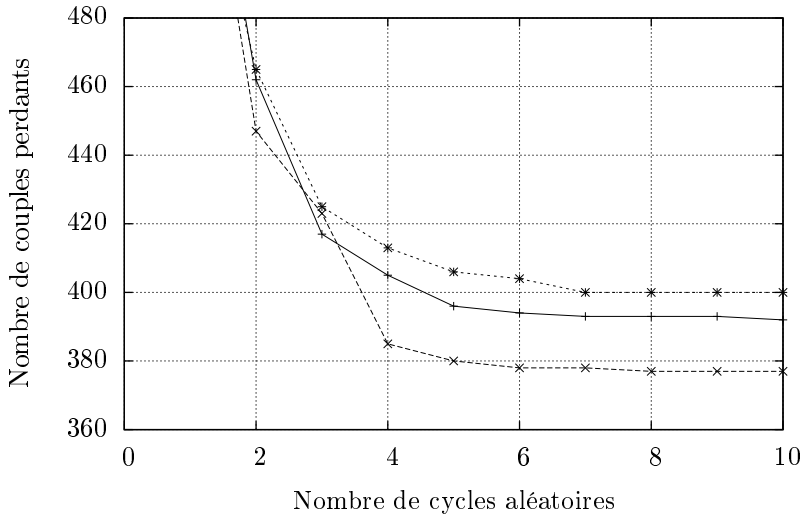


FIGURE 7.4 – Zoom dans la bande 360–480 couples.

espérer avec une série de vérifications est souvent compris entre 10% et 25%. Compte-tenu du petit gain en mémoire et du temps de calcul nécessaire pour mener les vérifications, il ne paraît pas évident que cette technique puisse être d'un grand secours pour aider à terminer un calcul qui consommerait trop de mémoire.

En revanche, dans l'optique d'obtenir des arbres solutions les plus petits possibles pour des démonstrations manuelles, cette technique est tout à fait utile. C'est elle notamment qui nous a permis d'obtenir la démonstration de S_5^+ présentée dans l'annexe B avec seulement 15 positions perdantes.

Par contre, elle ne peut pas garantir que l'arbre solution obtenu soit minimal : une autre série de vérifications aléatoires pourrait converger vers une valeur plus petite, et des arbres solutions plus petits pourraient être obtenus avec un autre calcul initial.

Chapitre 8

Architecture du programme

8.1 Outils de programmation

Nous présentons tout d'abord dans les paragraphes qui suivent les outils qui nous ont permis de développer notre programme de résolution des jeux combinatoires, nommé « Glop ». Nous avons travaillé principalement sur des plateformes Linux, mais les outils de programmation choisis sont tous multi-plateforme, ce qui rend notre programme compilable aussi bien sous Linux que Windows ou Macintosh.

Le code source est disponible sous licence GNU GPL, ce qui autorise d'autres programmeurs à étudier et modifier notre programme comme ils le souhaitent, par exemple s'ils veulent s'appuyer sur notre travail pour obtenir de nouveaux résultats. Nous publions ce code source sur notre site web [1], accompagné de fichiers exécutables immédiatement fonctionnels pour les principaux systèmes d'exploitation.

8.1.1 Langage C++

Nous avons développé notre programme dans le langage C++. Bien que le langage en lui-même ne soit pas très adapté aux programmes mathématiques, il a l'avantage de disposer de nombreuses bibliothèques d'extension, et d'être connus par beaucoup de programmeurs. L'un des inconvénients du C++ est la présence de nombreux concepts qui peuvent rendre le code difficile à comprendre (comme les pointeurs). En contrepartie, le C++ permet de concevoir des programmes avec une architecture modulaire grâce aux concepts de classes et de polymorphisme, comme nous le montrerons dans la section 8.4.

Le code C++ est facilement compilable sur toutes les plateformes, avec par exemple l'incontournable GCC (GNU Compiler Collection). GCC est un logiciel libre (licence GNU GPL), qui est devenu le compilateur de référence pour l'ensemble des logiciels libres.

8.1.2 Bibliothèque STL

La bibliothèque STL est une extension standard du C++. Elle fournit des concepts utiles qui n'existent pas dans le langage C++ de base, à savoir les listes, les vecteurs, les ensembles, et les map (tableaux triés). Ces objets ont l'inconvénient d'être assez consommateurs de mémoire, et ils ne sont donc pas forcément à recommander dans le cas d'une application critique en terme de mémoire. Dans notre cas, nous les utilisons de façon importante, parce qu'ils permettent d'écrire la majeure partie du code dont nous avons besoin dans le cadre de notre programme. Par ailleurs, ils possèdent une syntaxe commune, ce qui permet d'obtenir un code cohérent et homogène.

1. `http://sprouts.tuxfamily.org/`

8.1.3 Bibliothèque Qt

La bibliothèque Qt est une bibliothèque libre (licence GNU GPL), multiplateforme (Linux, Mac, Windows), écrite en C++, qui fournit à peu près toutes les briques logicielles nécessaires pour écrire un programme quelconque. Elle propose en particulier des objets pour construire des applications graphiques, gérer des fichiers XML, accéder à internet, ou écrire des programmes multi-processus.

La bibliothèque Qt est particulièrement bien conçue, ce qui permet de développer facilement la partie graphique du programme. Dans notre cas, nous utilisons aussi activement les capacités multi-processus de Qt, pour afficher l'avancement des calculs en temps réel. Autant que possible, cependant, nous évitons d'utiliser la bibliothèque Qt dans le cœur du programme (la partie qui réalise les calculs), pour que cette portion du code soit réutilisable et compréhensible même par quelqu'un qui ne connaîtrait pas Qt.

8.1.4 Subversion

Subversion est un logiciel libre (licence Apache et BSD) de gestion de versions du code source. Il permet de créer un historique des modifications apportées au code source, et de les sauvegarder sur un *repository*, en général stocké sur un serveur distant, pour que tous les développeurs puissent y accéder à travers Internet. Subversion permet de visualiser les modifications faites par chaque programmeur, ce qui facilite le développement d'un logiciel en équipe.

8.1.5 Doxygen

Doxygen est un logiciel libre (licence GNU GPL), qui permet de créer la documentation d'un logiciel à partir des commentaires du code source, pour peu que ceux-ci soient écrits avec une syntaxe prédéfinie. La documentation produite peut par exemple être sous forme de pages HTML, et publiée ensuite en ligne. L'utilisation d'un outil de documentation du code source permet de maintenir la documentation à jour en permanence.

8.1.6 Dokuwiki

Dokuwiki est un projet libre de type wiki (licence GNU GPL), qui permet de créer un site web similaire à Wikipedia. La mise à jour du site web est donc très simple et ne nécessite aucune connaissance de programmation web. Un historique des modifications de chaque page est créé. Dokuwiki permet de gérer facilement les droits d'accès et de modifications des pages, par exemple pour donner un droit d'écriture uniquement aux membres du projet.

8.1.7 Tuxfamily

Tuxfamily est une plateforme d'hébergement pour les projets libres, et en particulier pour les logiciels libres. Les services proposés contiennent notamment un repository Subversion (hébergement du code source), un serveur PHP pour héberger un site web dynamique (comme Dokuwiki par exemple), une zone de stockage pour proposer des fichiers au téléchargement, et aussi un service mail. Tuxfamily permet donc de regrouper en un seul point d'Internet tous les outils nécessaires au travail en équipe autour d'un projet libre.

8.2 Principaux éléments du programme

8.2.1 Figure d'ensemble

La figure 8.1 décrit l'architecture de notre programme. Nous avons essayé de le modulariser au maximum, de sorte que l'implémentation d'une nouvelle fonctionnalité (par exemple, un nouvel algorithme de parcours) soit systématiquement disponible pour l'ensemble des jeux étudiés.

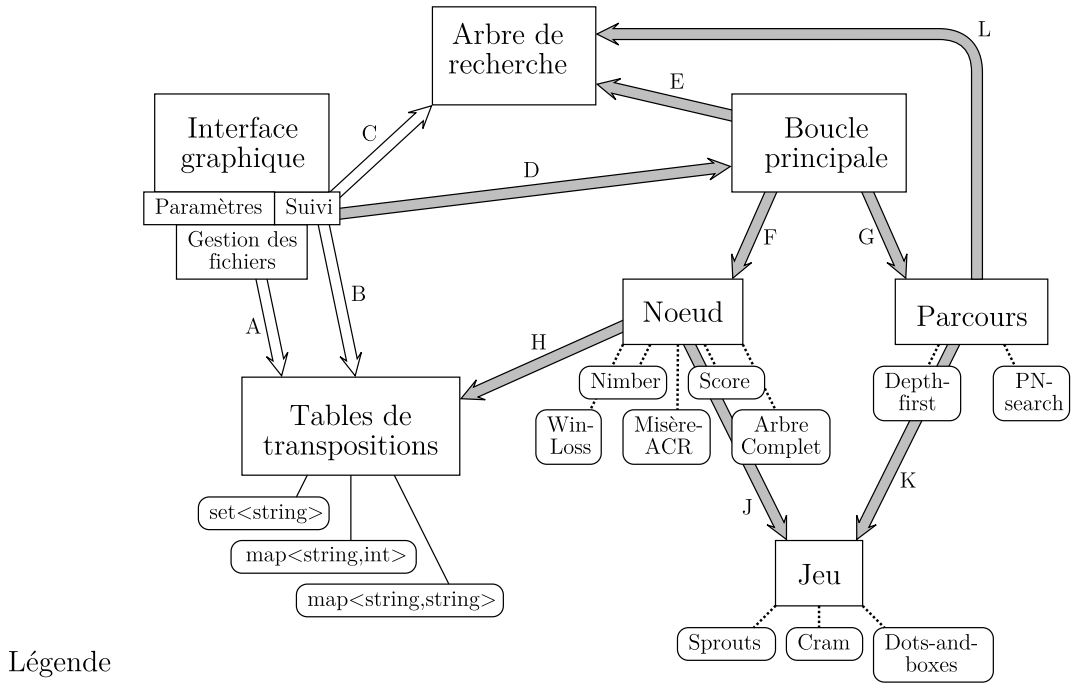


FIGURE 8.1 – Architecture modulaire du programme.

Les différents blocs de la figure 8.1 correspondent à un premier niveau de modularisation, le programme étant divisé en sept grands ensembles conceptuels, à savoir : « Interface graphique », « Boucle principale », « Arbre de recherche », « Tables de transpositions », « Nœud », « Parcours » et « Jeu ».

Les flèches de la figure indiquent les interactions entre les différents blocs, et correspondent à des appels de fonctions. Le bloc à l'origine de la flèche lance l'appel, qui se concrétise par l'exécution d'une certaine fonction au sein du bloc cible. Les flèches ont été étiquetées avec les lettres de A à L [2]. Les flèches blanches correspondent à des interactions passives (qui ne modifient pas le bloc cible), tandis que les flèches grises indiquent par opposition des interactions actives, qui modifient le bloc cible, ou bien qui impliquent des calculs coûteux.

Spécialisation de blocs

Un deuxième niveau de modularisation est présent au sein des blocs « Nœud », « Parcours » et « Jeu ». Lors d'un calcul, chacun de ces blocs prend une forme spécifique, confor-

2. à l'exception de la lettre I, pour des raisons de lisibilité.

mément aux choix de l'utilisateur. Les choix possibles sont indiqués sur la figure dans les bulles.

Le bloc « Jeu » est un objet informatique qui représente une position d'un jeu donné. Le bloc « Nœud » représente un nœud de l'arbre de jeu : dans le cas le plus simple du nœud « WinLoss », il s'agit d'une position de jeu ; dans le cas du nœud « Nimber », il s'agit d'un couple (position, nimber), etc. Les différents nœuds possibles sont détaillés dans la section 8.3. Enfin, le bloc « Parcours » est une extension d'un nœud qui permet de stocker les informations spécifiques au parcours de l'arbre de jeu. Typiquement, il s'agit des coefficients de l'algorithme Proof-Number Search.

Le mécanisme de modularisation est réalisé de telle sorte que, vu de l'extérieur, les spécificités du bloc sont invisibles. Par exemple, lorsque la boucle principale accède à un nœud, elle ne voit qu'un nœud générique, alors qu'il s'agit en réalité d'un nœud spécifique, comme par exemple « Nimber » ou « Score ». De la même façon, les nœuds ne connaissent que l'existence d'un jeu générique, qui prend en réalité suivant le contexte la forme d'un jeu spécifique, par exemple de « Sprouts » ou de « Dots-and-boxes ». Ce mécanisme de modularisation, mis en oeuvre à l'aide de la *dérivation de classe*, est décrit dans la section 8.4.

Boucle principale de calcul

La modularisation des concepts de nœud, de jeu et de parcours, et leur manipulation à travers une forme générique permet d'écrire une seule et unique boucle principale de calcul, indiquée sur la figure par le bloc « Boucle principale ». Nous détaillons dans la section 8.3 comment cette boucle de calcul est capable de manipuler de façon unique tous les types de nœuds possibles, ce qui correspond à la flèche notée F sur la figure 8.1.

Tables de transpositions

Les tables de transpositions sont des bases de données qui stockent les résultats de calcul des différents nœuds calculés. Trois types différents de bases de données ont été implémentées, pour les besoins spécifiques des différents calculs. Les tables de transpositions sont décrites dans la section 8.5.

Les nœuds accèdent aux tables de transpositions (flèche H de la figure 8.1) avant de calculer le résultat, pour vérifier s'il n'est pas déjà connu, et en fin de calcul, pour y ajouter un nouveau résultat. L'accès aux tables de transpositions des conversions entre les objets informatiques calculés et leurs représentations sous forme de chaînes de caractères. Cette technique, appelée *sérialisation*, fait l'objet de la section 8.6.

Par ailleurs, la flèche A de la figure 8.1 correspond à l'accès depuis l'interface pour sauvegarder un table de transpositions donnée dans un fichier.

Interface graphique

Enfin, l'interface graphique du programme est décrite dans la section 8.7. Le suivi en temps réel des calculs est un sujet suffisamment riche pour être traité séparément, et fait l'objet du chapitre 5. Nous décrivons simplement ici les principales interactions avec les autres éléments du programme :

* La flèche B de la figure 8.1 indique l'accès en lecture seule du suivi aux tables de transpositions, pour afficher l'évolution en temps réel de la taille des tables.
* La flèche C correspond à l'accès en lecture seule du suivi à la table contenant l'arbre de recherche. Cela permet d'afficher la branche en cours de calcul ou de naviguer au sein de l'arbre de recherche lors d'un algorithme de type PN-search.
* La flèche D correspond à l'envoi d'ordres de l'interface vers la boucle principale : lancement et arrêt du calcul, mais aussi interactions manuelles pour guider le calcul, comme le zappage lors d'un algorithme de type depth-first.

8.2.2 Taille du programme

La figure 8.1 permet de distinguer quatre grands ensembles de code assez distincts dans le programme.

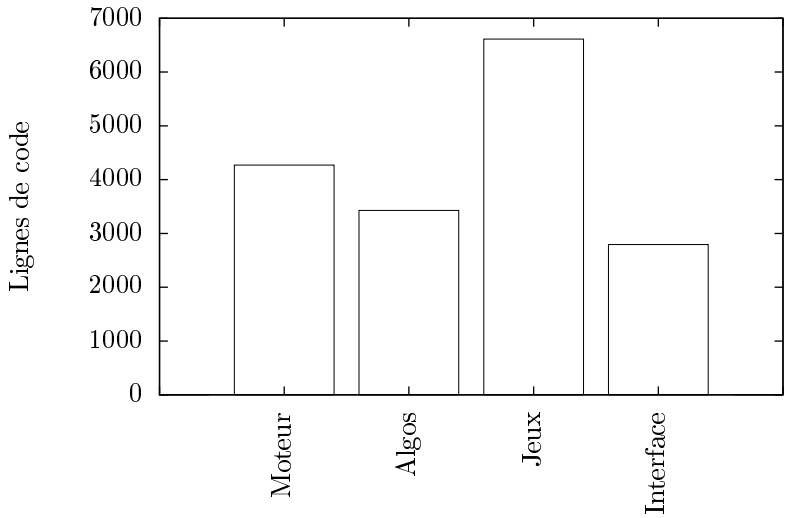


FIGURE 8.2 – Nombre de lignes de code des quatre principaux ensembles du programme.

* Moteur de calcul : nous avons regroupé sous ce terme la boucle principale, la table de stockage de l'arbre de recherche et les tables de transpositions de la figure 8.1, ainsi que la partie non graphique du suivi.
* Algorithmes : ce sont les différents types de calculs, et de parcours de l'arbre. Les types de calculs implémentés sont « nimber », « misère », « winloss » et « arbre complet » pour les jeux impartiaux, et « score » pour le Dots-and-boxes. Les algorithmes de parcours sont « depth-first » et « PN-search ».
* Jeux : implémentation des jeux de Sprouts, Cram et Dots-and-boxes. Il s'agit plus précisément du code concernant la représentation en chaînes, le calcul des options et la canonisation.
* Interface : le code qui permet d'afficher l'interface, de régler les paramètres, de suivre et d'interagir avec le calcul.

Le diagramme en barres 8.2 indique le nombre de lignes de code de ces quatre grands ensembles, en incluant les commentaires (hormis les 20 lignes d'entêtes des fichiers indiquant la licence du programme).

Le diagramme en barres 8.3 indique le nombre de lignes de code pour des modules plus détaillés.

On constate en particulier la difficulté de codage du jeu de Sprouts, ainsi que la difficulté de codage de l'algorithme misère basé sur les arbres canoniques réduits par rapport à l'algorithme à base de nimbers de la version normale.

Le faible nombre apparent de lignes de code pour le jeu de Cram est lié au fait que la majeure partie du code nécessaire au Cram est contenu dans le module Board, classe générique permettant de représenter des jeux de plateaux découpables. Ce module est commun au Dots-and-boxes, et donc le Dots-and-boxes nécessite en fait environ 3200 lignes de code, presque le même nombre de lignes de code que le Sprouts.

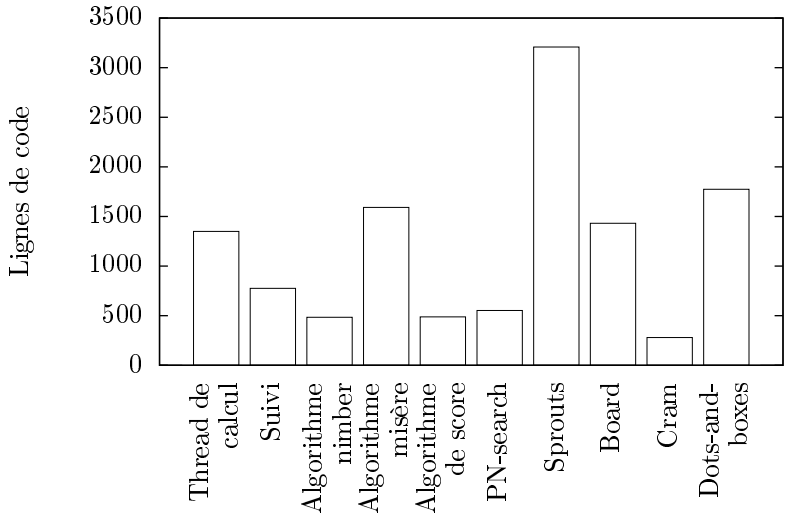


FIGURE 8.3 – Nombre de lignes de code des principaux modules du programme.

8.2.3 Programme multi-processus

Comme expliqué dans le chapitre 5 sur le suivi, le programme est multi-processus pour permettre de rafraîchir l'interface graphique tout en réalisant les calculs en tâche de fond.

Tout d'abord, au lancement du programme, on crée un premier processus (*thread* en anglais), dans lequel s'exécutent les fonctions de l'interface graphique. Puis, lorsque l'utilisateur clique sur le bouton « Start », on crée un deuxième processus, dans lequel s'exécutent cette fois les fonctions du calcul. Ce deuxième processus est détruit lorsque le calcul se termine, éventuellement lors d'une interruption manuelle avec le bouton « Stop ». En cours de calcul, le programme se compose donc de deux processus exécutés simultanément : d'une part le *processus graphique*, qui affiche l'interface du programme, et d'autre part le *processus de calcul*, qui réalise effectivement les calculs.

Si on reprend la figure 8.1, le bloc « Interface graphique » est exécuté dans le processus graphique, alors que les blocs « Boucle principale », « Nœud », « Parcours » et « Jeu » sont exécutés dans le processus de calcul. Les blocs « Arbre de recherche » et « Tables de transpositions » sont quant à eux des objets partagés, qui servent notamment à la communication entre les deux processus.

Techniquement, la création de processus est simple grâce à la classe QThread de la bibliothèque Qt. Par contre, les objets partagés et la communication entre processus sont source de problèmes techniques délicats, que nous décrivons dans le §5.2.4 du chapitre 5.

Sur les processeurs multi-cœurs, désormais standards même dans l'informatique grand public, le processus graphique et le processus de calculs sont exécutés sur des cœurs différents, ce qui rend négligeable le surcoût lié à l'utilisation de la bibliothèque graphique Qt.

8.3 Boucle de calcul principale

8.3.1 Calculs avec ou sans suivi

Il existe en réalité deux versions différentes de la boucle principale suivant que le suivi est actif ou non. Nous désignons par *suivi* le système d'affichage et d'interaction en temps réel avec les calculs. Comme expliqué dans le chapitre 5m c'est un élément important du programme, qui permet d'avoir une vision d'ensemble de l'avancement des calculs, et de modifier directement en cours de calcul les choix d'exploration effectués par le programme,

au prix cependant d'un code informatique complexe. Pour profiter des avantages du suivi, tout en se protégeant des risques d'erreurs liés à sa complexité, nous avons donc donné la possibilité de réaliser les calculs avec ou sans le suivi.

Si l'option « use the minimal recursive structure » est cochée, les calculs sont réalisés avec une boucle principale minimale, sans suivi, basée sur une structure récursive très simple. Par contre, si l'option est décochée (valeur par défaut) les calculs sont réalisés avec le suivi, ce qui implique principalement un stockage des nœuds de calcul dans la table de l'arbre de recherche (voir figure 8.1). C'est cette table de recherche et les mécanismes d'accès en temps réel par deux processus différents (processus graphique et processus de calcul), qui rendent le suivi complexe et difficile à débugguer.

Pour comparaison, la boucle principale sans suivi, présentée dans son intégralité dans le paragraphe suivant, fait 11 lignes de code seulement (commentaires exclus), alors que la boucle principale avec suivi et l'ensemble du code de l'arbre de recherche en font environ 1000 !

La philosophie adoptée pour s'assurer de la validité des calculs consiste à réaliser les calculs avec une structure aussi complexe que nécessaire, même si celle-ci contient potentiellement des bugs liés à sa complexité, puis dans une deuxième étape à vérifier les calculs avec une structure aussi simple que possible. Effectuer a posteriori un calcul de vérification avec les 11 lignes de la boucle principale minimale permet de s'assurer que le résultat des calculs n'est pas entaché d'un bug qui se trouverait dans les 1000 lignes de code de l'arbre de recherche.

8.3.2 Boucle principale sans suivi

Le listing 8.1 montre que lors d'un calcul sans suivi, la boucle principale est simplement une fonction récursive (nommée Game::recursiveLoop). Cette fonction prend en argument un *nœud* de l'arbre de recherche. Notre programme propose plusieurs types de nœuds, suivant le type de calculs que l'on souhaite réaliser, mais tous ces nœuds sont utilisés de la même façon par la boucle principale sous la forme d'un objet générique BaseNode.

Listing 8.1 – Fonction recursiveLoop.

```
void Game::recursiveLoop(BaseNode& node) {
  if(node.nodeExists()) return;

  list<BaseNode> nodeChildren;
  if(node.computeNodeChildren(nodeChildren)) return;

  list<BaseNode>::iterator child;
  for(child=nodeChildren.begin(); child!=nodeChildren.end(); child++) {
    do {
      recursiveLoop(*child);
    } while(!child->computeIsFinished());

    if(node.computeLocalResult(*child)) return;
  }
  node.computeFinalResult(nodeChildren);
}
```

Bien que cette boucle principale semble extrêmement simple après coup, c'est en réalité l'une des fonctions qui a mis le plus de temps à émerger et à se stabiliser vers sa forme définitive. En effet, le programme était initialement bâti autour d'une unique fonction, qui regroupait tous les types de calculs possibles (version normale ou misère, avec vérification...). Les embranchements vers les différents calculs étaient faits par de nombreuses commandes **if**, de sorte que le code était devenu illisible au fur et à mesure de l'ajout de nouvelles options de calcul.

Il a ensuite fallu de nombreux tâtonnements pour séparer les différents types de calculs, puis leur trouver un point commun qui permette de les manipuler à travers une fonction unique (d'une taille raisonnable, cette fois). La séparation des différents types de calculs a abouti à la notion de *nœud*, et la recherche d'un point commun a abouti quant à elle à la boucle principale du listing 8.1.

Les informations que l'on souhaite calculer et la façon d'effectuer le calcul peuvent différer notablement suivant le jeu en question, mais la fonction Game::recursiveLoop montre que dans tous les cas que nous avons traité dans cette thèse, l'algorithme suit un schéma directeur commun. On prend donc en argument un nœud, et l'on commence par vérifier si le résultat de ce nœud n'est pas déjà connu ou non, en appelant la fonction nodeExists(). On calcule ensuite la liste des enfants du nœud avec computeNodeChildren() qui renvoie la liste des nœuds enfants, puis on lance récursivement le calcul sur chaque enfant, avec recursiveLoop(*child). Chaque fois que le calcul sur un enfant est terminé, on essaie d'en déduire le résultat du nœud parent avec computeLocalResult(*child). Enfin, si tous les enfants ont été calculés sans que le résultat de l'un d'entre eux n'ait permis de déduire le résultat du parent, alors on déduit le résultat du parent avec l'ensemble des résultats des enfants, grâce à computeFinalResult(nodeChildren).

Lors de l'appel récursif pour calculer le résultat d'un enfant, on remarquera qu'il y a une boucle do-while, qui relance le calcul sur le même enfant tant que computeIsFinished ne renvoie pas *true* (valeur qui signifie que le calcul de cet enfant est terminé). Cette notion est facultative et n'apparaît que dans certains algorithmes particuliers où le résultat d'un nœud donné ne peut être calculé que par *étapes* successives.

On notera que la boucle principale n'a pas « conscience » de la façon concrète dont on vérifie que le résultat d'un nœud est déjà connu, ni sur la façon de calculer les enfants d'un nœud, ou de déduire le résultat du nœud parent à partir d'un nœud enfant. Nous ne définissons d'ailleurs même pas ce qu'est un nœud ni ce qu'est le résultat d'un nœud. Ces notions dépendent en fait du type de nœud considéré, et chaque nœud doit donc les redéfinir lui-même. Du point de vue de la boucle principale, un nœud est simplement un objet informatique de type BaseNode qui dispose des 5 fonctions nodeExists, computeNodeChildren, computeIsFinished, computeLocalResult et computeFinalResult.

8.3.3 Nœuds disponibles

Nous décrivons dans la table 8.1 les différents nœuds que nous avons implémentés, avec leurs principales caractéristiques du point de vue de la boucle principale. Dans l'ensemble du tableau, $\mathscr{P}$ désigne n'importe quelle position du jeu en cours de calcul.

Nœud	Résultat	Contenu du nœud	Nœuds enfants
WinLoss	Issue	Position $\mathscr{P}$	option($\mathscr{P}$)
Nimber	Issue ou Nimber	($\mathscr{P}$, nimber $\mathbb{n}$)	(option($\mathscr{P}$), $\mathbb{n}$) ou ($\mathscr{P}$, $\mathbb{i}$) avec i<n
Arbre complet	Arbre canonique (réduit ou non)	Position $\mathscr{P}$	option($\mathscr{P}$)
Misère-ACR	Issue	($\mathscr{P}$, ACR_1, ..., ACR_n)	(option($\mathscr{P}$), ACR_1, ..., ACR_n) ou ($\mathscr{P}$, ..., option(ACR_i), ...)
Score	Issue ou Score	($\mathscr{P}$, contrat c)	(option($\mathscr{P}$), boîtes disponibles$-c+1$)

TABLE 8.1 – Nœuds disponibles dans le programme, avec leurs caractéristiques.

Lors d'un calcul d'issue, que ce soit avec les nœuds WinLoss, Nimber, Misère-ACR et Score, les règles de déduction du résultat du nœud en fonction du résultat de ses enfants

sont toujours les mêmes. La règle de déduction locale (fonction computeLocalResult) consiste à affirmer que si un enfant est d'issue perdante, alors le nœud d'issue gagnante, et la règle de déduction globale (computeFinalResult) affirme quant à elle que si tous les enfants sont gagnants, alors le nœud est perdant.

Les règles de déduction du résultat sont particulières dans le cas d'un calcul d'arbre complet. Tout d'abord, on n'implémente aucune règle de déduction locale, ce qui signifie que l'on ne peut jamais déduire le résultat du nœud à partir d'un enfant seulement. La règle de déduction globale, qui consiste par définition à calculer le résultat du nœud à partir de l'ensemble des résultats de ses enfants, se traduit dans ce cas précis par le calcul de l'arbre canonique du nœud à partir de la liste des arbres canoniques des enfants. Dans le cas d'un calcul d'abre canonique *réduit*,on remplace simplement la notion d'arbre canonique par celle d'arbre canonique réduit.

Ces différents exemples montrent que les notions générales manipulées par la boucle principale, à savoir celles de nœud, de résultat, d'enfants d'un nœud, de déduction locale et de déduction globale sont exactement les notions naturelles liées à la nature même des jeux combinatoires. Il est donc probable que la boucle principale de notre programme sera capable de supporter telle quelle ou avec des modifications minimes les développements futurs du programme.

8.3.4 Boucle principale avec suivi

Dans le cas d'un calcul avec suivi, la boucle principale reste essentiellement similaire à celle décrite précedemment. La principale différence réside dans un stockage des nœuds en cours de calcul, ainsi que des relations entre les nœuds parents et les nœuds enfants, dans la table de l'arbre de recherche, auquel le thread graphique peut accéder en temps réel. Le thread graphique affiche alors à l'écran des informations sur l'avancement du calcul et permet des interactions de l'utilisateur directement pendant le calcul. Enfin, le stockage des nœuds dans une table permet également d'effectuer des algorithmes plus complexes au niveau de l'ordre des calculs, par exemple pour mieux tenir compte des transpositions dans l'arbre de jeu, ou bien pour effectuer les calculs dans un ordre qui dépende de l'ensemble des nœuds présents en mémoire.

Exactement comme pour la boucle principale sans suivi, la boucle principale avec suivi ne possède aucun savoir sur ce qu'est réellement un nœud ou le résultat d'un nœud. Elle sait uniquement qu'un nœud possède un certain nombre de fonctions, et définit l'ordre dans lequel ces fonctions sont appelées. Les nœuds doivent posséder plusieurs fonctions supplémentaires par rapport à celles présentées dans le cas des calculs sans suivi. En particulier, un nœud doit posséder une fonction displayStringList () qui renvoie une chaîne de caractères représentant ses informations essentielles. Cette chaîne de caractère sera transmise à l'interface graphique pour l'affichage en temps réel.

8.3.5 Arbre de recherche

Dans le cas d'un calcul avec suivi, les nœuds du calcul sont stockées dans une table (classe NodeStore au niveau du code), intitulée arbre de recherche sur la figure 8.1. Cette table a été prévue pour remplir deux fonctions principales. D'une part, l'accès simultané au tableau par le thread de calcul et par le thread graphique permet de suivre l'avancement des calculs. Et d'autre part, le stockage sous forme d'une base de données permet d'implémenter des algorithmes plus complexes que l'alpha-bêta, comme le PN-search ou des variantes.

Quand un nouveau nœud (objet BaseNode) est ajouté à la table de recherche, on commence par lui attribuer un identifiant numérique unique, noté id dans ce paragraphe. L'arbre de recherche est alors une table qui à un id donné associe un nœud acompagné d'informations

supplémentaires. Un élément complet de la table de recherche est composé essentiellement de la façon suivante :

* un objet BaseNode : le nœud en lui-même, concept central de la boucle principale, et qui correspond à un bloc « Nœud » de la figure 8.1.
* la liste des ids des nœuds enfants
* l'id du nœud parent
* un objet Traversal, qui correspond à un bloc « Parcours » de la figure 8.1, et permet d'enrichir le nœud avec des informations permettant d'influencer l'ordre de parcours de l'arbre de recherche
* des meta-informations qui seront affichées dans l'interface graphique : le nombre de nœuds enfants, le nombre de nœuds enfants encore inconnus, le nœud enfant en cours de calcul, et si le calcul du nœud est terminé ou non

8.4 Modularisation des jeux, nœuds et parcours

8.4.1 Classes C++ et dérivation

Le C++ est un langage qui permet de modulariser facilement les éléments d'un programme grâce à la notion de classe. Quand le programme est bien conçu, une classe correspond à un concept humain adéquat pour l'application considérée. Dans le cas de la programmation des jeux combinatoires, il y a une classe pour représenter le Sprouts (qui se subdivise en fait en plusieurs classes plus précises pour représenter les différents éléments du Sprouts), une pour les jeux de plateaux, une pour représenter les calculs à base de nimber, une autre encore pour la notion d'arbre canonique, etc. De façon peut-être encore plus facile à appréhender, chaque élément de l'interface graphique prend également la forme d'une classe. On a donc une classe pour représenter le bouton associé à une base de données, une pour la table affichant le suivi du calcul en temps réel, etc.

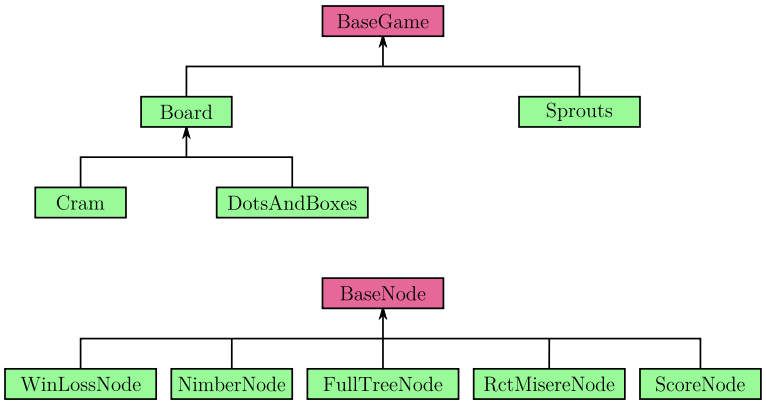


FIGURE 8.4 – Hiérarchie des classes de jeux et de nœuds.

Le C++ donne par ailleurs la possibilité de faire dériver une classe depuis une autre. Le terme de dérivation n'est pas très explicite. Il correspond en fait à l'idée de spécialisation. Par exemple, dans notre cas, il y a une classe Board, qui sert à représenter les jeux de plateaux. Cette classe a besoin d'être *spécialisée*, en lui ajoutant des fonctionnalités, pour s'adapter aux besoins plus précis de chaque jeu. Les classes Cram et DotsAndBoxes, qui représentent bien sûr les jeux correspondants, dérivent donc de la classe Board. Cette technique de dérivation permet de mettre en commun très naturellement le code utilisé à la fois dans le Cram et dans le Dots-and-boxes, à savoir le code qui exprime le fait que ces jeux sont essentiellement des

jeux de plateaux.

La figure 8.4 montre la hiérarchie des classes représentant les jeux et les nœuds. Les nœuds correspondent aux nœuds de l'arbre de jeu, et représentent un certain type de calcul, pour calculer par exemple l'issue ou le nimber d'une position, ou bien le score.

8.4.2 Polymorphisme

Le C++, comme tous les langages dits *objets*, fournit un mécanisme remarquable, appelé *polymorphisme*, qui permet de manipuler une classe dérivée à travers sa classe de base. Nous allons expliquer ce mécanisme sur un exemple concret du programme. Si l'on reprend la figure 8.4, on remarque que tous les jeux dérivent d'une même classe de base, appelée BaseGame. L'une des fonctions de BaseGame est la fonction computeOptionSet, qui consiste à calculer la liste des options accessibles à partir d'une position de jeu donnée. Cette fonction de calcul des options dépend bien sûr du jeu donné, et donc chaque jeu doit en fournir sa propre implémentation spécifique, comme le montre le listing 8.2.

Listing 8.2 – Fonction virtuelle computeOptionSet().

```
//déclaration dans la classe de base
class BaseGame {
  public:
  virtual void computeOptionsSet();
  ...
}

//implémentation dans la classe dérivée Cram
void Cram::computeOptionsSet() {
  //compute the children of each board of the list
  int index;
  for(index = 0; index < (int) boardList.size(); index++) {
      ...
}

//implémentation dans la classe dérivée Sprouts
void Sprouts::computeOptionsSet() {
  //compute the children of each sub-structure
  list<Land>::iterator it;
  for(it=r_str.begin(); it!=r_str.end(); it++) {
      ...
}
```

Cette fonction de calcul des options est utilisée en particulier lors des calculs à base de nimbers, dans la classe NimberNode. Le fond du problème vient du fait que l'on souhaite écrire une fois pour toutes les algorithmes de calcul à base de nimbers, et pouvoir les appliquer soit au Cram, soit au Sprouts, ou même encore à un autre jeu si besoin. Le code de l'algorithme des nimbers doit donc répondre à deux exigences en apparence contradictoires : d'une part, on souhaite qu'il soit indépendant du Cram et du Sprouts, mais dans le même temps, on souhaite qu'il utilise les fonctions spécifiques de chacun de ces jeux pour calculer la liste des options d'une position donnée.

C'est ici que le mécanisme de polymorphisme intervient. Ce mécanisme consiste tout d'abord à faire manipuler des objets de type BaseGame par l'algorithme des nimbers, ce qui résout le premier problème : l'algorithme des nimbers ne connaîtra ainsi que la notion abstraite BaseGame, et restera indépendant des formes plus précises de cette notion, que sont le Cram et le Sprouts. Le cœur du mécanisme de polymorphisme est alors de transmettre l'appel vers les fonctions de BaseGame aux fonctions correspondantes de la bonne classe

dérivée, ce qui résout cette fois le second problème : quand l'algorithme de nimber appellera BaseGame::computeOptionsSet, cet appel sera en fait transmis à Cram::computeOptionsSet ou bien Sprouts :: computeOptionsSet suivant le contexte.

L'utilisation concrète du polymorphisme en C++ est malheureusement assez technique, et nécessite de recourir à la notion de *pointeur*. En effet, le C++ ne permet pas tel quel de considérer un objet de type Cram comme un objet de type BaseGame. Il permet seulement de traiter un pointeur vers un objet de type Cram comme un pointeur vers un objet de type BaseGame. Un pointeur étant simplement une adresse en mémoire, cela signifie en quelque sorte que l'on peut traiter l'habitant à une certaine adresse mémoire comme un habitant de type BaseGame, même s'il s'agit en fait d'un habitant de type Cram.

Listing 8.3 – Utilisation possible du polymorphisme de computeOptionsSet dans les calculs de nimber.

```
//quelque part en début de calcul
BaseGame* game;
if(computeSprouts) {
  game = new Sprouts;  //cas d'un calcul de Sprouts
} else {
  game = new Cram;  //cas d'un calcul de Cram
}

//utilisation dans NimberNode
win_loss NimberNode::compute_children_Nimber(list<BaseNode> &children) {
  game -> initWith(currentPosition);
  game -> computeOptionsSet();
  ...
}
```

Le listing 8.3 montre comment on pourrait utiliser concrètement le mécanisme de polymorphisme dans les calculs de nimber. On crée en début de calcul soit un objet de Sprouts, soit un objet de Cram, et l'on stocke l'adresse dans la variable game, qui est un pointeur de type BaseGame. Les calculs de nimber utilisent ensuite ce pointeur BaseGame pour calculer la liste des options à partir de la position courante (la variable currentPosition). Le polymorphisme intervient lors de l'appel game->computeOptionsSet(). Suivant que l'on a créé initialement un objet de Sprouts ou de Cram, c'est la fonction computeOptionsSet() de la classe correspondante, Sprouts ou Cram, qui sera appelée.

Ce mécanisme de polymorphisme est à la fois une technique basique et avancée du C++. Basique, car elle constitue l'un des fondement du langage, et avancée, car elle demande un effort notable d'abstraction pour comprendre son fonctionnement.

8.4.3 Le cauchemar des pointeurs

Nous avons vu dans le paragraphe précédent que le polymorphisme était un mécanisme puissant du C++, mais qu'il nécessite de manipuler des pointeurs, qui sont en fait les adresses en mémoire des objets. Or, comme le titre de ce paragraphe le suggère, les pointeurs sont très délicats à manipuler. Les problèmes apparaissent dès lors que l'on effectue des copies des variables. Le code 8.4 illustre ce problème.

On crée d'abord une variable nommée s qui est chaîne de caractères, et l'on effectue une copie, nommée sCopy. Il s'agit d'une copie classique, d'objets. sCopy est un nouvel objet, dont la valeur initiale est la même que s, mais s et sCopy ont désormais des vies indépendantes. Si l'on modifie l'un, cela n'a strictement aucune influence sur l'autre.

On effectue ensuite la même opération avec des pointeurs. On crée un objet string en mémoire (avec **new**), et l'on stocke son adresse dans une variable p. On dit que la variable p

pointe vers une chaîne de caractères. On effectue ensuite une copie, nommée pCopy. Il s'agit ici d'une copie de l'adresse de la chaîne de caractères, et donc p et pCopy pointent vers la même chaîne de caractères en mémoire. Toute modification de la chaîne pointée par p se répercute sur pCopy, puisque qu'il s'agit du même objet. Ce comportement contre-intuitif est facilement source d'erreurs.

Listing 8.4 – Copie d'un objet et copie d'un pointeur.

```
string s;
string sCopy = s;

string* p = new string;
string* pCopy = p;
```

Les pointeurs peuvent vite devenir un cauchemar à cause de ce problème de la copie. De façon éventuellement indirecte, c'est en fait la cause la plus fréquente de crash des programmes.

Les pointeurs ont par ailleurs un autre inconvénient majeur : le programmeur ayant créé l'objet lui-même avec l'opérateur **new**, il doit également le supprimer lui-même avec l'opérateur **delete**, sous peine de provoquer une fuite de mémoire.

En pratique, dans l'ensemble du programme, nous évitons au maximum les pointeurs, et limitons leur utilisation uniquement à quelques cas particuliers, à savoir :

* les itérateurs de la bibliothèque STL : ce sont des pointeurs, mais ils ne servent que localement et temporairement pour parcourir une liste ou un ensemble.
* les pointeurs vers des objets graphiques de Qt : la bibliothèque Qt s'occupe elle-même du problème de la copie et de la suppression des objets.

Malgré cette utilisation limitée à des contextes très précis, les pointeurs occupent une bonne place dans la liste des pires bugs que nous ayons rencontrés :

* une mauvaise utilisation d'un itérateur de la STL provoquait des crashs intempestifs, mais suffisamment rares pour nous empêcher de comprendre l'origine du problème immédiatement. Il nous a fallu plusieurs semaines de débuggage pour découvrir finalement qu'un itérateur de la STL était utilisé sur des données qui avaient été supprimées. Comme l'utilisation était faite presque immédiatement après la suppression, le contenu de l'adresse mémoire était le plus souvent encore correct... sauf lorsque pour une raison quelconque cette adresse mémoire avait déjà été réutilisée.
* une utilisation inadéquate d'un pointeur vers un objet graphique de la bibliothèque Qt provoquait une fuite de mémoire.

Dans le cas de notre programme, il y a également un autre obstacle à l'utilisation des pointeurs. Nous utilisons activement les objets de listes et d'ensembles de la bibliothèque STL. Or, au niveau informatique, ces listes et ces ensembles sont des listes et des ensembles d'objets, qui sont fondamentalement incompatibles avec les pointeurs.

Listing 8.5 – Liste de pointeurs.

```
string* p = new string("test");
list<string*> listString;

listString.push_back(p);  //ajout de p à la liste
p->[0] = 'm';      //modification de la première lettre de p
listString.push_back(p);  //nouvel ajout de p à la liste
```

Par exemple, le code 8.5 montre un exemple d'utilisation erronée d'une liste de pointeurs vers des strings (chaînes de caractères). On crée une chaîne de caractères contenant le mot « test », on l'ajoute à la liste, on modifie la première lettre en « m », et l'on ajoute le nouveau mot « mest » à la liste. Ce code ne poserait aucun problème si l'on manipulait directement

des strings, mais comme ici l'on manipule des pointeurs, le résultat est assez inattendu : on a simplement ajouté deux fois la même adresse à la liste, qui pointe vers le même mot. Donc la liste contient deux fois le mot « mest ».

Or, notre programme utilise des listes de positions (« BaseGame »), des listes de nœuds (« BaseNode »), des ensembles de positions, des tableaux triés de positions, etc, avec bien sûr des copies d'objets lors de chaque insertion. À moins de disposer d'un bon stock d'aspirine, il semble donc préférable de ne pas se lancer dans la manipulation des objets fondamentaux du programme à travers des pointeurs.

8.4.4 Polymorphisme avec des objets

Nous avons montré dans les paragraphes précédents que le mécanisme de polymorphisme nécessite l'utilisation de pointeurs, qui ne sont pas compatibles avec une utilisation intensive de la bibliothèque STL. Pour contourner ce problème, nous avons développé un mécanisme original permettant de disposer du polymorphisme sans pour autant avoir besoin de manipuler directement des pointeurs. Le code 8.6 montre le principe de cette technique en reprenant l'exemple de la classe BaseGame déjà discuté en 8.2.

De façon assez surprenante, on utilise une définition récursive, en déclarant au sein de la classe BaseGame un pointeur vers un objet BaseGame, nommé p_baseGame. Puis on définit une implémentation par défaut de la fonction computeOptionsSet, qui consiste à appeler la même fonction sur le pointeur. Le code du Cram (ou du Sprouts) du listing 8.3 est, lui, inchangé.

Listing 8.6 – Principe du polymorphisme d'objets

```
//définition de la classe BaseGame
class BaseGame {
  private:
  BaseGame * p_baseGame;

  public:
  BaseGame();
  virtual void computeOptionsSet();
  ...
}

//constructeur appelé lors de la création d'un objet BaseGame
BaseGame::BaseGame()
  if(computeSprouts) {
    p_baseGame= new Sprouts; //cas d'un calcul de Sprouts
  } else {
    p_baseGame= new Cram;    //cas d'un calcul de Cram
  }
}

//implémentation par défaut : appel de la même fonction sur le pointeur
void BaseGame::computeOptionsSet() {
  p_baseGame->computeOptionsSet();
}

//implémentation (inchangée) dans la classe dérivée Cram
void Cram::computeOptionsSet() {
  //compute the children of each board of the list
  int index;
  for(index = 0; index < (int) boardList.size(); index++) {
      ...
}
```

Le point-clef qui permet de comprendre le fonctionnement de cette classe réside dans la définition du constructeur, qui est appelé lorsque l'on crée un objet de type BaseGame. On

remarque qu'il y a création, soit d'un objet de Sprouts, soit d'un objet de Cram, en fonction du type de calcul, et que l'adresse de cet objet est retenue dans le pointeur p_baseGame.

Le code 8.7 est une réécriture du code 8.3 en utilisant la nouvelle classe BaseGame que l'on vient de définir. On voit que le pointeur externe game a disparu, car il est remplacé par le pointeur interne p_baseGame. Le nouveau code manipule maintenant un objet BaseGame, et non plus un pointeur. C'est cette différence fondamentale qui va nous permettre de profiter du polymorphisme, sans pour autant avoir besoin de saupoudrer le code de tout le programme avec des pointeurs. En enfermant toute la mécanique des pointeurs dans la classe BaseGame, il devient ensuite possible de programmer tout le reste (le Sprouts, le Cram, le calcul des nimbers, etc) sans se préocupper de ces problèmes techniques difficiles.

Listing 8.7 – Utilisation du polymorphisme avec la nouvelle classe.

```
//utilisation dans NimberNode
win_loss NimberNode::compute_children_Nimber(list<BaseNode> &children) {
  BaseGame game;
  game.initWith(currentPosition);
  game.computeOptionsSet();
  ...
}
```

Examinons maintenant en détail ce qui se passe lors de l'exécution du code 8.7, afin de comprendre le fonctionnement de la classe BaseGame. On suppose dans cet exemple qu'il s'agit d'un calcul de Cram (la variable computeSprouts vaut **false**). Le calcul effectue alors les étapes suivantes :

* un objet de type BaseGame, appelé game, est créé.
* la création de l'objet game appelle le constructeur BaseGame::BaseGame().
* cela crée un objet de Sprouts ou de Cram suivant le contexte du calcul — de Cram, dans cet exemple.
* l'adresse de l'objet créé est stockée dans le pointeur interne p_baseGame de l'objet game.
* on initialise l'objet game avec la position en cours de calcul (nous ne détaillons pas cette étape, il s'agit du même enchaînement d'évènements que la fonction computeOptionsSet qui nous sert d'exemple et qui est détaillée ci-dessous).
* on appelle la fonction computeOptionsSet() de l'objet game.
* comme l'objet game est de type BaseGame, la fonction appelée est l'implémentation par défaut BaseGame::computeOptionsSet().
* BaseGame::computeOptionsSet() appelle la même fonction sur le pointeur interne p_baseGame.
* le pointeur interne p_baseGame pointe vers un objet de type Cram, donc le mécanisme de polymorphisme du C++ prend le relais, et provoque l'appel de la fonction Cram::computeOptionsSet() de l'objet pointé, ce qui était le but recherché.

L'enchaînement d'étapes conduit bien à ce qui est attendu, à savoir que l'appel de la fonction game.computeOptionsSet() provoque au bout du compte l'exécution de Cram::computeOptionsSet(). L'intérêt principal de BaseGame est de masquer ce mécanisme, qui est invisible de l'extérieur. Le code de calcul des nimbers du listing 8.7 ne saurait être plus simple.

Il est à noter que nous n'avons trouvé nulle part de référence à cette technique pour effectuer du polymorphisme avec des objets grâce à un pointeur interne vers la classe elle-même.

8.4.5 Destruction, copie et clonage

Le paragraphe précédent a montré comment on pouvait obtenir le polymorphisme tout en continuant de manipuler des objets, en utilisant un système de pointeurs internes à la classe. Cela ne résout cependant pas le problème fondamental de la copie et de la suppression des pointeurs, qui va en effet réapparaître lorsque l'on fait des copies de la classe BaseGame, ou que l'on détruit un objet BaseGame. En effet, si l'on effectue une copie de la classe BaseGame en l'état (par exemple une copie de l'objet game du code 8.7), la valeur du pointeur interne sera copiée telle quelle. On aura deux objets BaseGame différents, dont le pointeur interne pointe vers un même objet de Cram, cas d'erreur typique.

Ce problème commun à toutes les classes qui contiennent des pointeurs est bien connu et documenté. On pourra consulter par exemple l'article de Richard Gillam *The Anatomy of the Assignment Operator* qui décrit sur un ton humoristique comment les problématiques mises en jeu sont suffisament difficiles pour recaler la quasi-totalité des programmeurs C++ lors d'un entretien d'embauche [20].

La solution consiste à implémenter manuellement dans BaseGame l'opérateur d'affectation, le constructeur de copie, et le destructeur. Ces fonctions sont définies automatiquement par le compilateur, et en général, il n'y a pas lieu de s'en préocupper. La définition à la main n'est nécessaire que si la classe contient des pointeurs, car les fonctions par défaut fournies par le compilateur ne tiennent pas compte de l'aspect particulier des pointeurs. Elles se contentent de copier les adresses des objets pointés au lieu des objets eux-mêmes.

Le destructeur, appelé lors de la destruction d'un objet BaseGame, est très simple : il suffit de supprimer l'objet pointé par p_baseGame avec le mot clef spécial **delete**. Cette destruction manuelle correspond à la création manuelle faite dans le constructeur de BaseGame (voir listing 8.6).

Listing 8.8 – Destructeur de la classe BaseGame.

```
BaseGame::~BaseGame() {
  if(p_baseGame!=0) {
    delete p_baseGame;
    p_baseGame = 0;
  }
}
```

Les classes sont susceptibles d'être copiées de deux façons uniquement : soit à travers le signe égal, qui sert d'opérateur d'affectation en C++, soit lors d'une création de classe par copie.

Examinons tout d'abord le cas de la création de classe par copie. Il s'agit d'initialiser un objet BaseGame de façon à ce qu'il soit identique à un autre objet BaseGame passé en paramètre. Dans notre cas, la classe ne contient qu'un pointeur, et il faut simplement faire attention à ne pas copier telle quelle la valeur du pointeur, mais à créer un nouvel objet pointé similaire. La difficulté est que l'objet pointé peut prendre la forme d'un objet de Sprouts ou de Cram suivant le contexte. Le plus simple dans ce cas, est de demander à l'objet de fournir une copie de lui-même, opération appelée *clonage*. On obtient le code 8.9.

Listing 8.9 – Constructeur de copie de la classe BaseGame.

```
BaseGame::BaseGame(const BaseGame& a) {
  if(a.p_baseGame != 0) {
    p_baseGame = a.p_baseGame->clone();
  } else {
    p_baseGame = 0;
  }
}
```

Dans le cas de l'opérateur d'affectation, le problème est exactement le même, à ceci près que l'objet BaseGame existe déjà, et donc que le pointeur interne pointe peut-être déjà vers quelque chose. Il faut alors supprimer cet objet avec l'opérateur **delete**, avant de faire pointer le pointeur vers le clone. On obtient le code 8.10.

Listing 8.10 – Opérateur d'affectation de la classe BaseGame.

```
BaseGame& BaseGame::operator=(const BaseGame& a) {
  if(this != &a) {
    //delete the old p_baseGame
    if(p_baseGame!=0) {
      delete p_baseGame;
      p_baseGame=0;
    }

    if(a.p_baseGame != 0) {
      p_baseGame = a.p_baseGame->clone();
    } else {
      p_baseGame = 0;
    }
  }

  return *this;
}
```

Nous terminons enfin en montrant le code 8.11 de clonage de la classe Cram, qui tient en une seule ligne.

Listing 8.11 – Clonage de la classe Cram.

```
virtual BaseGame* Cram::clone() const {
  return new Cram(*this);
};
```

Nous ne détaillerons pas plus les difficultés techniques de l'implémentation de la classe BaseGame, qui sont toutes résolues avec différentes astuces spécifiques au langage C++. Mentionnons simplement un casse-tête technique pour ceux qui souhaitent y réfléchir : dans le code 8.6, le constructeur de BaseGame crée un objet de Cram. Or, comme le Cram dérive lui-même de BaseGame, la construction d'un objet de Cram provoque l'appel automatique du constructeur de BaseGame, qui va tenter à son tour de construire un objet de Cram, qui va appeler le constructeur de BaseGame, etc. En l'état, il s'agit d'une boucle infinie.

8.4.6 Intérêts et inconvénients

Les avantages de la classe BaseGame sont multiples. Tout d'abord, cette classe préserve le mécanisme de polymorphisme du C++ : tout appel d'une fonction de BaseGame est répercuté grâce au polymorphisme vers l'objet pointé, qui prend la forme du jeu en cours de calcul, Cram, Sprouts, Dots-and-boxes, ou autre. On peut donc programmer des algorithmes généraux en manipulant des objets abstraits BaseGame. Chaque jeu est libre ensuite de réimplémenter spécifiquement les différentes fonctions abstraites de BaseGame, dont computeOptionsSet est un exemple.

Par ailleurs, au niveau informatique BaseGame est un objet. On peut donc manipuler sans le moindre problème des listes ou des ensembles de BaseGame avec la bibliothèque STL, ce qui permet d'écrire facilement des algorithmes complexes. BaseGame se charge de masquer complètement et de sécuriser les problèmes compliqués — et dangereux — liés aux pointeurs.

La complexité interne de BaseGame est invisible, que ce soit dans le code des algorithmes de calcul ou dans celui des jeux.

Il faut avoir conscience qu'à l'inverse, BaseGame n'est pas exempt d'inconvénients. Le pointeur interne est une surcharge de mémoire, et les manipulations liées à ce pointeur sont une surcharge de temps de calcul. Une telle technique ne pourrait pas être utilisée pour des petits objets simples en grand nombre. Dans notre cas, c'est heureusement l'inverse, nous manipulons en général dans les calculs un petit nombre d'objets compliqués.

Tout d'abord, nous ne stockons pas directement des objets BaseGame ou BaseNode dans les bases de données, mais nous les transformons préalablement en chaînes de caractères. Dans la plupart de nos calculs, le nombre d'objets BaseGame ou BaseNode existant simultanément en mémoire est donc relativement limité. Et par ailleurs, les objets dérivant de BaseGame, comme le Sprouts, effectuent des opérations complexes, coûteuses en temps de calcul, qui rendent totalement négligeable la surcharge en temps liée aux transferts des appels à l'intérieur de BaseGame.

8.5 Tables de transpositions

Les résultats intermédiaires d'un calcul sont stockés dans des tables de transpositions, pour permettre d'une part d'accélérer les calculs grâce aux transpositions, et d'autre part de vérifier le résultat d'un calcul rapidement, sans avoir à effectuer de nouveau une recherche de la stratégie gagnante dans l'arbre de jeu. Nous décrivons ici comment nous avons implémenté ce concept de table de transpositions dans des bases de données.

8.5.1 Types de bases de données

Les résultats de calcul que l'on souhaite stocker obéissent à un schéma similaire. On souhaite associer à une position d'un jeu donnée le résultat d'un calcul, avec la possibilité de retrouver cette position rapidement dans la base. Nous avons implémenté trois types de bases de données :

* map <string, **unsigned char**> : tableau trié qui associe un entier entre 0 et 255 à une chaîne de caractères donnée.
* map <string, string> : tableau trié qui associe une chaîne de caractères à une chaîne de caractères donnée.
* set <string> : ensemble trié de chaînes de caractères.

Ces trois types de bases de données suffisent pour la plupart des calculs sur les jeux combinatoires. Ils sont par exemple utilisés respectivement pour :

* les calculs de nimbers : base (position, nimber).
* les calculs d'arbres canoniques : base (position, arbre canonique).
* les calculs misère : ensemble des positions perdantes.

On remarquera que les entrées des bases de données sont des chaînes de caractères. Si l'on veut stocker des objets plus complexes, il faut les convertir préalablement en chaînes de caractères avant d'accéder aux bases. Ce problème se pose par exemple dans les calculs misère à base d'arbres canoniques réduits. Les positions sont dans ce cas des listes de composantes indépendantes, dont certaines sont des arbres canoniques réduits représentés informatiquement par des combinaisons d'entiers et de booléens. Convertir cet objet complexe en chaîne de caractères n'est pas trivial. Ce processus est appelé *sérialisation* et fait l'objet de la section 8.6.

8.5.2 Objet informatique de stockage

Les bases de données sont implémentées concrètement avec les conteneurs map et set de la bibliothèque STL. Cela a l'avantage de la simplicité. Notamment, nous n'avons pas besoin de

fonction de hachage (avec les difficultés associées de collision, par exemple). L'inconvénient principal est une perte d'efficacité notable comparé à une table de hachage. Les recherches et les ajouts dans la table sont nettement plus lents, et chaque entrée de la table provoque un surcoût non négligeable en terme d'utilisation de la mémoire.

Par exemple, une entrée dans une map nécessite typiquement un surcoût de 16 à 32 octets. Cela ne nous a pas posé de problème lors des calculs de Sprouts et de Cram, car les algorithmes de calcul parviennent à réduire les bases à des valeurs inférieures au million de positions. Par contre, sur le Dots-and-boxes, les bases atteignent très vite la dizaine de millions de positions, et la consommation de mémoire devient alors un facteur limitant.

8.5.3 Interface d'accès

Nous avons développé une classe nommée db (pour « database »), qui sert d'interface aux algorithmes de calcul pour accéder aux bases de données. Le principal intérêt d'une telle interface est d'empêcher les algorithmes de calcul d'accéder directement aux bases. Si nécessaire, il serait assez simple de modifier l'implémentation interne des bases de données, sans avoir à modifier quoi que ce soit dans les algorithmes de calcul, du moment que l'interface d'accès est préservée.

Les algorithmes de calcul accèdent aux bases à travers les fonctions db::add() et db::find(), qui permettent respectivement d'ajouter ou de retrouver une entrée dans une base. La fonction db::create() permet aux algorithmes de calcul de créer des bases de données (lors du lancement du programme). Cette fonction de création renvoie un identifiant permettant au calcul de spécifier ensuite à quelle base il souhaite accéder. Nous donnons ci-dessous un exemple d'utilisation dans le cadre de l'algorithme de calcul des jeux impartiaux en version normale avec les nimbers.

Listing 8.12 – Utilisation des bases de données dans l'algorithme nimber.

```
void NimberNode::createDataStorage() {
  dbNimberIndex = db::create(/*stringnimber*/ 0, QString("Nimber") ...);
}

void NimberNode::resultFromAllChildren( ... ) {
  winLossResult = global::Loss;
  db::add(gameString(positionA), nimberA, dbNimberIndex, Parameter::isCheck);
}

void NimberNode::apply_known_nimbers()
  ...
  list<Line> components=g.sumComponents();
  list<Line>::iterator Li;
  for(Li=components.begin(); Li!=components.end(); Li++) {
    found = db::find(gameString(*Li), dbNimberIndex, Parameter::isCheck);
    ...
  }
```

Lors du lancement du programme, la fonction createDataStorage() est appelée et crée une base de données du type (string, **unsigned char**) qui servira à stocker le nimber d'une position de jeu. L'identifiant de la base est stocké dans la variable dbNimberIndex, qui sera utilisée lors de chaque accès aux bases par cet algorithme de calcul. La fonction resultFromAllChildren() montre le code exécuté lorsque tous les enfants d'une position sont gagnants. On sait que la position est perdante, et l'on en déduit donc son nimber. On ajoute ce résultat dans la base de données. Enfin, la fonction apply_known_nimbers() montre le point du programme où l'on réutilise les résultats connus. Avant d'effectuer le calcul des enfants d'une position,

```
[Positions+Score]
A*EALALB*E 2
A*EALB*E 2
A*EALCLB*E 2
A*EBLBLB*E 2
A*EBLC*E 2
ALALALA*E 2
BLB*E 1
BLCLB*E 1
```

TABLE 8.2 – Fichier obtenu lors d'un calcul de Dots-and-boxes.

on commence par chercher dans la base si l'on ne connaît pas déjà le nimber de certaines composantes.

8.5.4 Fichiers de calculs

Les bases de données sont visibles dans l'interface du programme sous la forme d'un bouton cliquable. Le nombre de positions dans la base est affiché sur le bouton et rafraîchi en temps réel au cours du calcul, ce qui permet de suivre l'évolution du nombre de positions stockées. Un clic sur le bouton correspondant à une base donnée donne accès à un menu déroulant, permettant de purger la base, de la sauvegarder dans un fichier, ou bien d'y ajouter une base préalablement sauvegardée.

La sauvegarde de la base se fait très simplement car les données sont déjà sous forme de chaîne de caractères. Le fichier obtenu est donc un simple listing au format texte des positions de la base. Nous donnons en exemple dans la table 8.2 le fichier obtenu lors du calcul de Dots-and-boxes d'un plateau de taille 2×3, avec un contrat de 2.

8.6 Sérialisation

8.6.1 Problème de la conversion entre chaînes et objets

Les objets informatiques manipulés par le programme sont constitués principalement de trois types élémentaires, à savoir les chaînes de caractères, les entiers et les booléens, qui sont ensuite combinés à travers des opérations d'union, de listes ou d'ensembles. Une difficulté classique en programmation apparaît lorsque l'on souhaite stocker les objets informatiques dans des fichiers. En effet, les fichiers ne permettent de manipuler simplement que les chaînes de caractères. Dès que l'on souhaite stocker des objets plus complexes, le processus de stockage dans les fichiers nécessite une conversion préalable de l'objet à stocker vers une chaîne de caractères.

Les conversions entre objets et chaînes de caractères nécessitent souvent de nombreuses lignes de code, avec certains mécanismes qui reviennent de façon répétitive. Par exemple, pour convertir une liste d'éléments ou un ensemble d'éléments en une chaîne de caractères, il suffit d'écrire bout à bout les chaînes représentant chaque élément, en séparant chaque élément avec un symbole adéquat, par exemple un espace ou un tiret. De façon générale, ce processus de conversion d'un objet complexe en une chaîne de caractères est connu sous le terme de *sérialisation*.

Dans le cas de notre programme, les conversions vers des chaînes de caractères apparaissent également dans le contexte des bases de données. En effet, certains objets complexes, comme les listes, ont une empreinte mémoire importante, à cause des pointeurs utilisés pour faire le lien entre un élément de la liste et le suivant. Les listes sont adaptées aux calculs

qui nécessitent de nombreuses insertions ou suppressions d'éléments, mais cela se fait au détriment de la mémoire consommée. Stocker telle quelle une liste dans une base de données ferait donc perdre énormément d'espace mémoire. L'idée est alors de convertir la liste en une chaîne de caractères au moment où l'on souhaite la stocker dans une base de données. Comme notre programme stocke les résultats de calculs dans des tables de transpositions, nous sommes systématiquement confrontés à ce problème de la sérialisation.

Il existe des librairies générales permettant de traiter le problème de la sérialisation, comme la librairie « boost », par exemple. L'avantage est de disposer d'un code solide et testé. L'inconvénient est de devoir ajouter une dépendance du programme vers cette librairie, et d'être obligé de se plier à ses règles spécifiques. Dans notre cas, nous avons choisi de développer notre propre outil de sérialisation, car nos besoins sont limités à quelques objets très précis (principalement les conteneurs de la bibliothèque standard, à savoir les listes, les vecteurs, les ensembles et les map), et nécessitent de pouvoir paramétrer assez finement les chaînes obtenues.

8.6.2 Classe StringConverter

L'outil de sérialisation que nous avons implémenté est une classe C++, qui se nomme StringConverter, et qui supporte deux opérateurs, << pour la conversion d'objets vers des chaînes (écriture vers une chaîne), et >> pour la conversion inverse (lecture depuis une chaîne). Ces opérateurs sont définis dans la classe StringConverter pour la plupart des types courants (lettre isolée, chaîne de caractères, entiers, booléens). Le point-clef de cette classe est la définition des opérations de lecture et d'écriture pour les conteneurs de la bibliothèque STL, à savoir les listes (list), les vecteurs (vector), les ensembles (set), et les multi-ensembles (multiset).

Nous montrons ci-dessous le principe du code permettant de convertir une liste d'éléments de type T en une chaîne de caractères. L'utilisation d'un template C++ permet de traiter des éléments de type T quelconque. Les éléments de la liste sont ajoutés l'un après l'autre à l'objet strConv, à travers la commande strConv << *Ti, avec ajout du caractère de séparation stlSeparator. Un code similaire est utilisé pour les vecteurs ou les ensembles d'éléments, avec plusieurs raffinements permettant par exemple d'ajouter ou non le caractère de séparation après le tout dernier élément.

Listing 8.13 – Template de conversion d'une liste en chaîne de caractères.

```
template<typename T>
StringConverter& operator<<(StringConverter& strConv, const list<T>& x) {
  typename list<T>::const_iterator Ti;
  for(Ti=x.begin(); Ti != x.end(); Ti++) {
    if(Ti!=x.begin()) strConv.internalString += strConv.stlSeparator;
    strConv << *Ti;
  }
  return strConv;
}
```

Ce code fonctionne pour tout objet sur lequel l'opérateur << est défini. Comme la classe StringConverter supporte directement les objets courants, cet opérateur ne doit être défini que pour les objets plus complexes.

Nous ne détaillons pas les techniques de lecture des chaînes, qui sont tout à fait similaires aux techniques d'écriture, avec cette fois l'opérateur >>.

8.6.3 Exemple d'utilisation

La classe StringConverter est par exemple très utile dans le cadre des arbres canoniques réduits (abrégés en *RCT* dans le code, de l'anglais *reduced canonical tree*) qui ont été présentés dans le chapitre 4. Si l'arbre canonique réduit est une colonne de Nim, on le représente simplement par la taille de la colonne de Nim. Dans le cas général, on le représente par la hauteur de l'arbre, suivi d'un identifiant unique, suivi de $+1$ si l'on a pu factoriser l'arbre par la colonne de Nim $\mathbb{1}$, et suivi enfin de « W » ou « L » suivant que la position est gagnante ou perdante, le tout séparé par des tirets. On remarque que le code est direct, grâce à l'objet StringConverter .

Listing 8.14 – Conversion d'un arbre canonique réduit en chaîne de caractères.

```
StringConverter& operator<<(StringConverter& conv, const Rct& x) {
  if(x.identifier==0){
    conv << (int) (x.depth+x.sum_with_1); //the Rct is a Nim-column
  } else {
    conv << (int) x.depth << "-" << x.identifier;
    if(x.sum_with_1) conv << "+1";
    conv << "-" << CvCd("BWL") << x.misere_win_loss;
  }
  return conv;
}
```

Les arbres canoniques réduits sont ensuite utilisés dans les calculs misère. Le code ci-dessous montre la conversion d'une position de calcul misère en une chaîne de caractères. La position complète est constituée d'un ensemble de composantes indépendantes. La représentation en chaîne commence par la composante principale, qui correspond à la ou les composantes trop complexes pour être simplifiées en arbres canoniques réduits, suivie de 0 ou de 1 suivant que la colonne de Nim $\mathbb{1}$ existe ou non dans les composantes, et se termine par un multi-ensemble d'arbres canoniques réduits. La conversion en chaîne du multi-ensemble d'arbres canoniques réduits se fait en une seule ligne de code grâce à StringConverter . Cette conversion utilise le template 8.13, et c'est ce template qui appelle sur chaque élément du multi-ensemble la fonction 8.14 de conversion d'un arbre canonique réduit en chaîne de caractères.

Listing 8.15 – Conversion d'une position de calcul misère en chaîne de caractères.

```
string PosRctToString(const PosRct& PosRctA) {
  StringConverter conv;
  conv << gameString(PosRctA.pos) << string("␣"); // position
  conv << CvCd("B10") << PosRctA.nimCol << string("␣"); //Nim-column 0/1
  conv << CvCd("LF") << PosRctA.RctSet; //multiset of Rct
  return conv.getString();
}
```

8.6.4 Paramétrage du format des chaînes

Nous avons introduit un mécanisme original pour paramétrer le format des chaînes avec l'utilisation de commandes directement lors des opérations de lecture/écriture. Dans les exemples précédents, ces commandes sont appelées avec le mot-clef CvCd et ont la signification suivante :

- BWL indique que les lettres pour représenter les booléens sont W et L (au lieu de T et F par défaut).
- B10 indique que les lettres pour représenter les booléens sont 1 et 0.

* LF indique de ne pas utiliser de caractère séparateur après le dernier élément du multi-ensemble.

D'autres commandes sont disponibles, par exemple pour définir le caractère séparateur entre les éléments d'un conteneur (espace par défaut), ou entre les éléments d'une chaîne (tiret par défaut).

8.6.5 Intérêt et limites

Avant l'implémentation de cet outil de sérialisation StringConverter, les fonctions de conversion liées aux arbres canoniques nécessitaient au total plusieurs centaines de lignes de code, difficiles à débugguer, et rendant très difficile toute modification éventuelle du format des chaînes de caractères. Comme l'ont montré les exemples de code précédents, il ne faut maintenant plus que quelques dizaines de lignes, et une modification éventuelle du format des chaînes ne serait pas difficile, grâce aux possibilités de paramétrage de StringConverter.

La principale limitation actuelle de l'outil StringConverter est le fait qu'il ne sait pas prendre en compte des conteneurs imbriqués les uns dans les autres si ceux-ci utilisent le même caractère séparateur. Il faut donc faire attention à bien utiliser un caractère séparateur différent si l'on utilise des structures imbriquées, comme par exemple une liste d'ensemble.

8.7 Interface graphique

8.7.1 Description de l'interface

Pour effectuer un calcul, l'utilisateur commence par choisir dans l'onglet « Parameters » le jeu qui l'intéresse. Le choix du jeu rend alors disponibles ou indisponibles les onglets situés en haut de l'interface. Ces onglets correspondent au type de *nœud* utilisé lors des calculs, et reflètent le type de résultats que l'on souhaite calculer sur le jeu en question. Les principaux nœuds disponibles sont les suivants :

* WinLoss permet de calculer l'issue (gagnante ou perdante) d'un jeu combinatoire en version normale ou misère, selon l'algorithme de calcul de l'issue le plus standard. Les jeux permettant ce type de calcul sont ceux où il n'y a pas de parties nulles possibles, et où la victoire/défaite est uniquement déterminée par le fait qu'un joueur ne peut plus jouer.
* Nimber permet de calculer le nimber ou l'issue d'un jeu impartial en utilisant efficacement les découpages du jeu si cela est possible.
* FullTree permet de réaliser des calculs sur l'arbre de jeu complet d'une position, en particulier, l'arbre canonique, ou l'arbre canonique réduit utile pour les jeux impartiaux en version misère.
* RctMisere permet de calculer l'issue ou la valeur de Grundy misère d'un jeu impartial en version misère, en utilisant autant que possible les découpages du jeu.
* Score permet de calculer le score d'une position (utile uniquement pour le Dots-and-boxes).

Une fois le jeu et le type de nœud choisis, il suffit de cliquer sur le bouton « Start » pour lancer le calcul. L'avancement du calcul est alors affiché en temps réel dans l'onglet « Computation ». Le bouton « Pause » permet de figer le calcul à son point actuel, ce qui peut être utile par exemple pour effectuer temporairement une autre tâche sur l'ordinateur effectuant les calculs. Enfin, le bouton « Stop » met fin aux calculs. En l'absence d'arrêt forcé par l'utilisateur, le programme poursuit les calculs jusqu'à l'obtention du résultat demandé.

L'onglet « Children » permet d'afficher la liste des enfants d'une position donnée. Pour chaque enfant, des informations sont affichées en fonction du contenu des bases de données. Les informations affichées sont celles correspondant au type de nœud choisi dans l'interface.

Enfin, l'onglet « Repository » permet de sauvegarder/recharger les paramètres d'un calcul. Cette fonctionnalité permet de sauvegarder les paramètres du jeu et du nœud, ainsi que des méta-informations sur le calcul (auteur, commentaires), ce qui permet de s'organiser plus facilement lorsque l'on effectue de nombreux calculs.

8.7.2 Création simple des interfaces

La création d'interfaces graphiques avec Qt peut devenir fastidieuse car il faut plusieurs lignes de code pour chaque élément de l'interface et d'autres lignes de code pour relier cette interface aux variables du programme. Nous avons implémenté de nombreux jeux et de nombreux nœuds différents, et pour chacun d'entre eux, il faut une interface utilisateur permettant de régler les paramètres spécifiques à cet objet. Comme les interfaces des différents jeux et nœuds ont de nombreux points communs, nous avons développé un mécanisme général qui permet de définir très simplement une interface graphique.

L'objet Interface permet de représenter une sorte de grille graphique, dans lequel sont placés des éléments graphiques, la valeur de chacun d'entre eux étant reliée à une variable du programme. L'objet Interface supporte les fonctions suivantes pour ajouter une étiquette (Label), une valeur numérique réglable (SpinBox), un bouton de choix rond (RadioButton), une case cochable (CheckButton), une ligne d'entrée de caractères (LineEdit), ou un bouton représentant une base de données (DataBaseButton) :

```
class Interface {
...
  void addLabel(...);
  void addSpinBox(...);
  void addRadioButton(...);
  void addCheckButton(...);
  void addLineEdit(...);
  void addDataBaseButton(...);
}
```

Les paramètres de ces fonctions correspondent aux valeurs initiales et à la position dans la grille graphique. Quand un objet vient d'être ajouté à l'interface, il est possible de relier sa valeur à une variable du programme avec les fonctions suivantes :

```
class Interface {
...
  void link(bool &boolTarget0, string name);
  void link(int &intTarget0, string name);
  void link(string &stringTarget0, string name);
}
```

Il est bien sûr important de relier les objets à des variables de même type : numérique (**int**), chaîne de caractères (string) ou booléen (**bool**). Le premier argument est celui de la variable à laquelle le dernier objet ajouté dans l'interface va être relié, et le deuxième est une chaîne de caractères qui servira de nom pour la variable dans les fichiers XML. Le lien avec la variable est réalisé de façon interne avec un pointeur, donc il faut impérativement que la variable choisie reste valide pendant toute la durée du programme.

Tous les jeux et tous les nœuds doivent définir une fonction getParamDef() qui renvoie un objet du type Interface . Le code de l'interface graphique se charge alors de créer automatiquement les interfaces graphiques correspondantes aux définitions faites avec l'objet Interface, et met automatiquement à jour les variables du programme avec les valeurs entrées par l'utilisateur dans l'interface graphique. Notre programme est également capable d'exporter la liste des variables dans un fichier XML, et inversement d'initialiser les variables à partir du fichier XML sauvegardé. Le nom des variables dans le deuxième argument des fonctions

link est utilisé comme nom de balise dans les fichiers XML. Pour un objet Interface donné il faut donc choisir des noms différents pour chacune des variables associées à l'interface.

Nous donnons ci-dessous un exemple de définition de l'objet Interface pour le nœud NimberNode, correspondant aux calculs à base de nimbers. L'interface se compose de deux boutons correspondant aux bases de données du calcul normal et de la vérification, d'une case cochable pour choisir de faire un calcul normal ou de vérification, d'une variable numérique pour pouvoir régler la partie nimber initiale si nécessaire, et enfin d'un choix au moyens de boutons radio entre le calcul de l'issue ou le calcul du nimber :

```
Interface NimberNode::getParamDef() {
  Interface result;
  result.name=string("Nimber");
  result.addDataBaseButton(dbNimberIndex, /*pos*/ 0, 0);
  result.addDataBaseButton(dbNimberIndex + 1, /*pos*/ 0, 1);
  result.addCheckButton("Activate check", false, /*pos*/ 0, 2);
  result.link(Parameter::isCheck, string("isCheck"));

  result.addLabel(string("Start Nimber part :"), /*pos*/ 1, 0);
  result.addSpinBox(0, 99, 0, /*pos*/ 1, 1);
  result.link(Parameter::given_nimber, string("startNimberPart"));

  result.addLabel(string("Compute :"), /*pos*/ 2, 0);
  result.addRadioButton("outcome", true, /*pos*/ 2, 1);
  result.link(NimberNode::computeOutcome, string("computeOutcome"));
  result.addRadioButton("nimber", false, /*pos*/ 2, 2);

  return result;
}
```

Ce mécanisme de définition des interfaces permet donc de rajouter très facilement un nouveau paramètre de calcul dans l'interface graphique et dans les fichiers XML, sans avoir besoin de connaître la bibliothèque Qt sous-jacente.

Chapitre 9

Représentation des positions du Sprouts

9.1 Introduction

Le *Sprouts* est un jeu combinatoire impartial, qui fut créé en 1967, par John H. Conway et Michael S. Paterson, à l'université de Cambridge. L'article de Martin Gardner [19], écrit quelques mois après la création du jeu, fournit de bonnes informations tant sur sa genèse que pour démarrer dans l'étude du jeu. Outre cet article, on peut également trouver une présentation de ce jeu dans *Winning Ways* [6].

Le Sprouts ne nécessite qu'une feuille de papier et un crayon pour pouvoir être pratiqué, et ses règles sont particulièrement simples. Le jeu débute avec un certain nombre de points tracés sur la feuille. À chaque coup, le joueur dont c'est le tour doit relier un point à un autre (éventuellement à lui-même) avec une ligne, puis rajouter un point sur cette ligne. Deux conditions doivent être respectées : les lignes ne doivent pas se croiser, et d'un même point ne peuvent partir plus de 3 lignes.

FIGURE 9.1 – Exemple de partie de Sprouts, en commençant avec 2 points (le deuxième joueur gagne en version normale).

Dans la version *normale* du jeu, le résultat est déterminé par la règle suivante : le joueur qui ne peut plus jouer a perdu. En version *misère*, c'est le contraire, le joueur qui ne peut plus jouer a gagné. Il ne peut donc y avoir de match nul. Dans un cas comme dans l'autre, le vainqueur est déterminé par la parité du nombre de coups de la partie.

La terminaison du jeu n'est pas évidente a priori. Comme à chaque coup, on crée un nouveau point, il est nécessaire de trouver un argument qui justifie que ces points ne peuvent être créés indéfiniment. La proposition suivante règle la question.

Proposition 12. *Le nombre de coups d'une partie de Sprouts commençant avec p points est majoré par $3p - 1$.*

Démonstration. On appelle *vie* un coup qui peut potentiellement encore être joué avec un point : un point dispose initialement de 3 vies. Si k lignes ($k \leq 3$) partent de ce point, alors il lui reste $3 - k$ vies. Étant donnée une partie de Sprouts commençant avec p points, il y a donc globalement $3p$ vies au départ. Chaque coup consomme deux vies, mais crée un point

déjà relié à deux lignes, donc avec une vie. Au total, chaque coup diminue ainsi d'exactement 1 le nombre de vies de la position. Après $3p-1$ coups, la partie est terminée, car il ne reste plus qu'une vie dans la position, or on a besoin de deux vies pour pouvoir jouer un coup. □

La partie peut cependant se terminer en moins de $3p-1$ coups, si dans la position terminale, il y a plusieurs points *isolés*, c'est-à-dire des points à une vie, que l'on ne peut relier entre eux car ils sont séparés par des lignes. Par exemple, une partie avec deux points au départ se termine en au plus $3\times 2-1=5$ coups. Mais la partie de la figure 9.1 se termine en 4 coups, car dans la position terminale, il y a deux points isolés. En général, le nombre de coups d'une partie est minoré par $2p$, comme démontré par des considérations élémentaires dans [6] p. 601.

Il semble que dans ses premières années, le jeu de Sprouts ait été l'objet d'un certain engouement, au point de servir de trame à un roman de 1969, *Macroscope* [3]. L'intérêt pour ce jeu peut s'expliquer par la complexité qui émerge de règles du jeu relativement simples. Rien que l'étude du jeu à deux points de départ n'est pas triviale. Mais cette complexité est peut-être aussi la cause de la désaffection pour ce jeu durant les deux décennies suivantes. Ainsi, si dans son article de 1967, Gardner fait état de la résolution manuelle du jeu à 6 points de départ, qui fit l'objet d'une analyse de 49 pages par Denis Mollison, il fallut attendre la première analyse par ordinateur en 1991 par Applegate, Jacobson et Sleator [4] pour que ce record soit battu.

On peut se demander pourquoi le jeu de Sprouts n'a pas été programmé plus tôt. Comme nous allons le voir dans ce chapitre, déterminer une représentation des positions du Sprouts sous une forme permettant à l'ordinateur de calculer les coups est difficile, le jeu sous forme papier-crayon n'étant pas adapté à la programmation. Le premier problème à traiter est le fait que sous cette forme, le jeu de Sprouts n'est pas un *jeu court* (voir à ce sujet le paragraphe 2.3.3).

À partir d'une position, on peut jouer une infinité de parties, et même, une infinité de coups, car il existe une infinité de chemins entre deux points. Mais dès lors que l'on identifie les positions égales à *déformation*[1] près, il n'y a plus qu'un nombre fini de parties possibles, et le Sprouts devient un jeu court.

À titre d'illustration, nous présentons dans la figure 9.2 l'arbre de jeu d'une position de Sprouts qui s'obtient en deux coups à partir de la position de départ à deux points. Dans cet arbre de jeu, nous avons identifié les positions égales à déformation près.

La position étudiée est très simple, elle intervient dans l'analyse du jeu à deux points de départ. Elle n'a que 4 vies, et l'argument de la démonstration de la proposition 12 peut s'adapter pour montrer que l'on ne peut jouer qu'un maximum de 3 coups à partir de cette position. Malgré cette relative simplicité, son arbre de jeu ne contient rien de moins que 25 nœuds. Une page ne suffirait pas pour tracer l'arbre de jeu de la position de départ à deux points.

Il va donc être nécessaire d'identifier des positions *équivalentes* (au sens du paragraphe 2.5.3), c'est-à-dire qui ont le même arbre canonique, pour réduire la taille des arbres de jeu et avoir une chance d'étudier les positions de Sprouts en un temps raisonnable. La première équivalence que nous pouvons constater est celle entre positions symétriques. Cette équivalence intervient entre 4 paires de positions dans la figure 9.2, celles qui sont séparées par le caractère « $\sim$ ».

L'équivalence suivante est moins évidente, mais plus efficace. Elle permet d'identifier les positions marquées par la même lettre. Pour constater cette équivalence, il faut avoir recours à la *projection stéréographique*, qui permet de montrer que jouer au Sprouts sur le plan est équivalent à jouer sur la sphère. Pour être précis, la projection stéréographique permet de ramener toute position tracée sur le plan à une position tracée sur la sphère privée d'un

1. La notion exacte est un *homéomorphisme* du plan sur lui-même, c'est-à-dire une bijection continue, de réciproque continue.

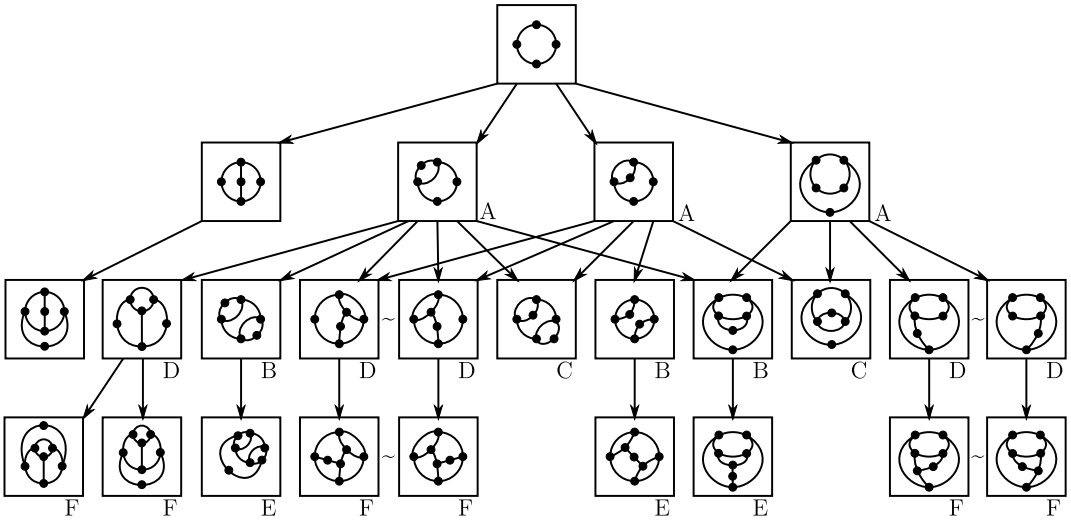


FIGURE 9.2 – Arbre de jeu d'une position de Sprouts.

unique point (le pôle nord sur la figure 9.3). Mais que ce soit sur un plan ou sur une sphère, supprimer un unique point de la surface ne change pas les parties jouables, car il est toujours possible de jouer les mêmes coups, éventuellement en déviant légèrement les lignes pour éviter de toucher le point supprimé. Il est donc équivalent de jouer sur le plan ou sur la sphère, c'est-à-dire que toute position, qu'elle soit tracée sur le plan ou sur la sphère, aura le même arbre canonique.

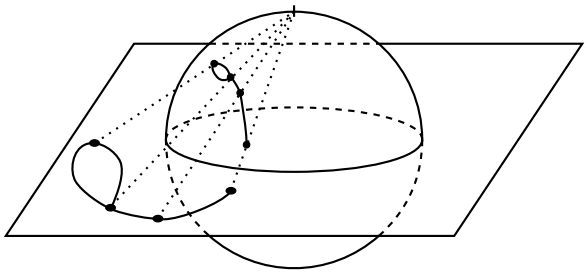


FIGURE 9.3 – Projection stéréographique.

Une fois une position de Sprouts représentée sur la sphère, on peut à nouveau utiliser la projection stéréographique pour la projeter sur le plan. Mais cette position découpe la sphère en plusieurs régions, et suivant la région dans laquelle on place le pôle lors de la projection, on obtient différentes positions du plan. Ces différentes positions du plan sont équivalentes, puisqu'elles dérivent de la même position sur la sphère.

Par exemple, les 3 positions marquées « A » sur la figure 9.2 correspondent à une seule et même position sur la sphère. C'est une position à 3 régions, lesquelles comportent respectivement 3, 4 et 5 sommets sur leur frontière. Lorsque l'on effectue la projection stéréographique avec le pôle dans une région donnée, cette région devient l'*extérieur* (la seule région non bornée) de la position projetée sur le plan. La position « A » de gauche est donc obtenue en mettant le pôle dans la région à 5 points, celle du milieu en mettant le pôle dans la région à 4 points, et celle de droite en mettant le pôle dans la région à 3 points.

Depuis la construction de l'arbre de jeu de la figure 9.2, nous avons présenté deux nouvelles équivalences : la symétrie, et l'équivalence entre intérieur et extérieur. Une fois ces

équivalences prises en compte, on obtient l'arbre de jeu de la figure 9.4, de taille bien plus raisonnable. On peut trouver un exemple un peu plus fourni dans *Winning Ways* [6], p. 599, avec le développement de l'arbre de jeu de la position de départ à 2 points, utilisant ces deux équivalences. Cet arbre contient la bagatelle de 51 positions, ce qui traduit la complexité du jeu de Sprouts.

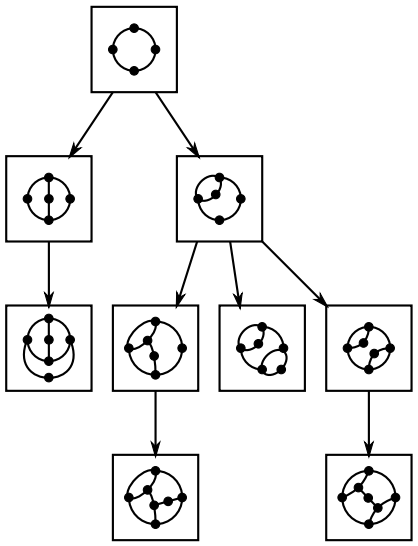


FIGURE 9.4 – Arbre de jeu d'une position de Sprouts.

Le but de ce chapitre sera ainsi de développer une représentation qui soit adaptée à la programmation, mais aussi qui identifie le maximum de positions équivalentes, afin de contrer du mieux possible la complexité spatiale particulièrement élevée du Sprouts.

9.2 Représentation des positions du Sprouts

9.2.1 Historique

Nous avons vu qu'entre l'invention du jeu, en 1967, et la première programmation effective du Sprouts, en 1991, 24 ans se sont écoulés. En effet, de par sa nature topologique (les joueurs tracent des lignes dans le plan), le jeu de Sprouts est particulièrement peu adapté à la programmation.

La première tentative de programmation dont nous avons trouvé la trace remonte à 1987, il s'agit d'un article intitulé *A logic programming model of the game of Sprouts* [11]. Les auteurs de cet article avaient établi les principaux concepts de la représentation en chaîne décrite plus tard dans [4]. Cependant, leur article contenait une erreur concernant l'orientation (sujet abordé au paragraphe 9.4.1). Et bien qu'ayant programmé leur représentation, ils ne l'avaient pas exploitée, ne poussant pas leur travail jusqu'à calculer l'issue de certaines positions.

Nous pouvons ensuite signaler un document de topologie de 350 pages [14], dont le but est de démontrer que le jeu de Sprouts sous sa forme topologique (papier et crayon) est équivalent au jeu sous sa forme informatique (représentation en chaîne).

La première programmation du jeu de Sprouts qui permit d'obtenir des résultats est dûe à Applegate, Jacobson et Sleator [4]. Non seulement ils développèrent une représentation efficace, mais ils surent également la mettre en œuvre de façon à obtenir la résolution faible du jeu normal jusqu'à 11 points de départ, et du jeu misère jusqu'à 9 points de départ. Leur

document ne se limite pas à décrire leur représentation, mais comporte de nombreuses idées sur la programmation du jeu de Sprouts en général.

Notre représentation des positions de Sprouts, sous forme de chaînes de caractères, dérive de celle de [4]. Nous la développons dans la suite de ce chapitre.

9.2.2 Régions et frontières

Une position de Sprouts peut être visualisée comme un graphe $\mathscr{G}$ plongé dans le plan $\mathscr{P}$. On appelle *région* l'union d'une composante connexe de $\mathscr{P} - \mathscr{G}$ et des éléments de $\mathscr{G}$ qui la touchent. Les règles du jeu interdisent de croiser une ligne existante, si bien qu'un coup se déroule nécessairement dans une seule région, et peut éventuellement la diviser en deux. Au cours d'une partie, le nombre de régions ne peut donc qu'augmenter.

Utilisant le vocabulaire classique de la théorie des graphes, les points seront désignés sous le nom de *sommets*, et les lignes sous le nom d'*arêtes*.

À l'intérieur d'une région, on appelle *frontières* les composantes connexes de la partie du graphe $\mathscr{G}$ en contact avec cette région. Un *sommet vierge* (un sommet qui n'est relié à aucune arête), est considéré comme une frontière à part entière.

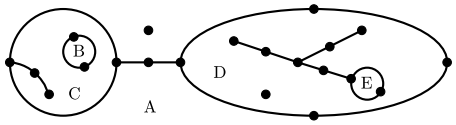


FIGURE 9.5 – Une position de Sprouts obtenue après 10 coups, avec 11 points de départ.

Par exemple, la figure 9.5 contient 5 régions et 9 frontières : il y a 3 frontières dans la région D, 2 frontières dans les régions A et C et 1 frontière dans les régions B et E.

9.2.3 Représentation en chaîne de caractères

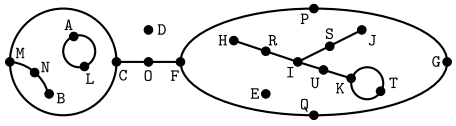
Chaque sommet est noté par une lettre, et le graphe est décrit de la façon suivante :

- Le graphe est représenté par la liste des chaînes des régions le composant. Le caractère séparateur de deux régions dans la liste est : « | »
- Chaque région est représentée par la liste des chaînes des frontières la composant. Le caractère séparateur de deux frontières dans la liste est : « . »
- Chaque frontière est représentée par la liste de ses sommets (sans caractère séparateur, chaque sommet étant identifié par un caractère unique).

Pour une frontière donnée, nous écrivons les sommets dans leur ordre d'apparition lorsque l'on parcourt la frontière, jusqu'à revenir au point de départ. Le sens dans lequel on tourne autour de la frontière a son importance, et sera discuté dans le paragraphe 9.4.1.

Pour le moment, nous adoptons la convention suivante. Étant donnée une région, une unique frontière entoure cette région (hormis pour la région extérieure, qui n'a pas de telle frontière). Nous tournons autour de cette unique frontière dans le sens direct, et nous tournons autour de toutes les autres frontières dans le sens indirect. En donnant des noms aux sommets de la figure 9.5, une chaîne représentant la position est par exemple :

```
AL|AL.BNMCMN|D.COFPGQFOCM|E.HRISJSIUKTKUIR.FQGP|KT
```



La frontière AL apparaît deux fois, parce qu'elle est à l'intérieur de la même région que les 4 autres sommets B;N;M;C, et parce qu'elle entoure en elle-même une région.

Détaillons la façon dont nous obtenons la frontière contenant les sommets B;N;M;C : en partant de B, nous passons sous N, puis M, puis nous suivons le bas de la frontière avant de rencontrer C. Nous continuons ensuite le long du haut de la frontière, et un demi-tour plus tard, nous rencontrons une nouvelle fois M puis N (mais sur le côté opposé) et nous nous arrêtons lorsque nous sommes revenu au point de départ.

En fait, une frontière est topologiquement équivalentes à une liste circulaire de sommets, comme l'illustre la figure 9.6. Nous avons utilisé sur la figure les notations M, M', N, N' pour bien montrer la distinction qui est faite entre les deux côtés d'un même sommet. On remarquera que le choix du premier sommet est arbitraire. On aurait tout aussi bien pu décrire la frontière par la chaîne MCMNBN.

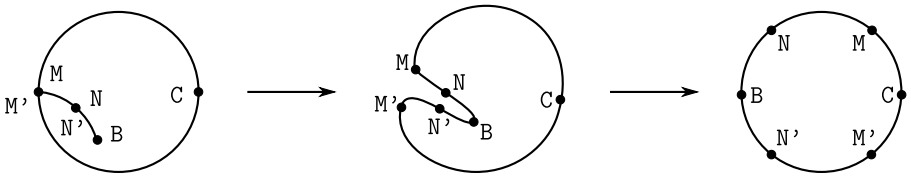


FIGURE 9.6 – Liste circulaire correspondant à une frontière (du point de vue de la région intérieure).

9.2.4 Positions équivalentes et représentation en chaîne

La représentation en chaîne de caractères présente l'avantage d'intégrer les équivalences dues aux déformations, et l'équivalence entre intérieur et extérieur. Par exemple, la chaîne de caractères BDCDBE|BE.A peut représenter les deux positions de la figure 9.7, qui sont équivalentes via la projection stéréographique.

Du fait que la représentation en chaîne intègre les équivalences dues aux déformations, elle profite du statut de jeu court du Sprouts, et ainsi, on obtient des arbres de jeu avec un nombre fini de nœuds, pour peu que l'on s'oblige à utiliser les premières lettres de l'alphabet dans les représentations en chaînes.

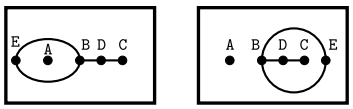


FIGURE 9.7 – Deux positions équivalentes.

L'inconvénient principal de la représentation en chaîne vient malheureusement tempérer l'avantage que nous venons de décrire. Si cette représentation permet effectivement d'identifier des positions équivalentes, elle produit également l'effet contraire à savoir qu'elle engendre des *doublons*. Ces doublons sont dus aux noms que l'on donne aux lettres (quel sommet appelle t-on A, quel autre sommet appelle t-on B,...), et à l'ordre dans lequel on cite les sommets d'une frontière, les frontières d'une région ou les régions d'une position.

Par exemple, la première position de la figure 9.7 peut conduire aux représentations BDCDBE|BE.A ou A.EB|DCDBEB, mais aussi B.AC|EDECAC en nommant les sommets différemment. Au total, il y a 5760 représentations qui conviennent si l'on utilise les lettres de A à E (soit 5! = 120 façons d'affecter une lettre à chaque sommet, multiplié par 48 façons d'ordonner les caractères).

En résumé, une chaîne de caractères unique représente une infinité de positions, puisque l'on peut tracer les arêtes d'une infinité de façons différentes du moment que la topologie est respectée. Inversement, à une position ne correspond qu'un nombre fini de chaînes de caractères, sous réserve de n'utiliser que les premières lettres de l'alphabet. Ce nombre de chaînes, bien que fini, est bien trop important pour ne pas s'en soucier. Rien que le choix du nom des sommets conduit à $n!$ possibilités, s'il y a n sommets.

Pour éviter une explosion combinatoire, il est nécessaire d'identifier les représentations en chaînes équivalentes. L'étape de canonisation décrite dans la section 9.4 est chargée de mener à bien cette identification.

9.2.5 Coups

La représentation en chaîne décrite plus haut permet de définir et de calculer simplement les coups autorisés. Les coups ont toujours lieu à l'intérieur d'une seule et même région, mais ils sont de deux types différents, suivant que l'on relie une frontière à elle-même, ou bien deux frontières différentes.

Coup (dans une frontière)**.** *Un* coup dans une frontière *consiste à relier deux sommets appartenant à une même frontière.*

Soit $\mathtt{x}_1...\mathtt{x}_n$ *une frontière, avec* $n \geq 2$*. Nous supposons que* $\mathtt{x}_i$ *et* $\mathtt{x}_j$ *sont des sommets qui apparaissent deux fois ou moins dans la totalité de la chaîne représentant la position, avec* $1 \leq i < j \leq n$*, ou bien que* $i = j$ *et* $\mathtt{x}_i$ *apparaît une seule fois dans la totalité de la chaîne*[2]*.*

Le coup consiste alors à séparer les autres frontières de la même région en 2 ensembles $\mathtt{F}_1$ *et* $\mathtt{F}_2$*, et la région initiale est divisée en deux nouvelles régions :* $\mathtt{x}_1...\mathtt{x}_i\mathtt{z}\mathtt{x}_j...\mathtt{x}_n.\mathtt{F}_1$ *et* $\mathtt{x}_i...\mathtt{x}_j\mathtt{z}.\mathtt{F}_2$ *où* $\mathtt{z}$ *est le nouveau sommet créé.*

La même définition reste valide si $n = 1$*, mais dans ce cas,* $\mathtt{x}_j...\mathtt{x}_n$ *est une frontière vide.*

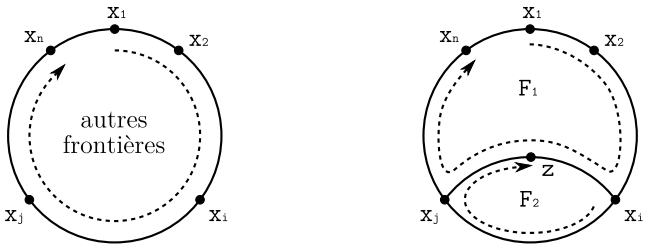


FIGURE 9.8 – Coup dans une frontière.

Coup (dans deux frontières)**.** *Un* coup dans deux frontières *consiste à relier deux sommets appartenant à des frontières différentes.*

Soit $\mathtt{x}_1...\mathtt{x}_m$ *et* $\mathtt{y}_1...\mathtt{y}_n$ *deux frontières différentes à l'intérieur d'une même région, avec* $m \geq 2$ *et* $n \geq 2$*. Nous supposons que* $\mathtt{x}_i$ *et* $\mathtt{y}_j$ *sont des sommets qui apparaissent deux fois ou moins dans la totalité de la chaîne représentant la position, avec* $1 \leq i \leq m$ *et* $1 \leq j \leq n$*.*

Le coup consiste alors à fusionner les deux frontières en $\mathtt{x}_1...\mathtt{x}_i\mathtt{z}\mathtt{y}_j...\mathtt{y}_n\mathtt{y}_1...\mathtt{y}_j\mathtt{z}\mathtt{x}_i...\mathtt{x}_m$ *où* $\mathtt{z}$ *est le nouveau sommet créé.*

La même définition est valide si $m = 1$ *ou si* $n = 1$*, mais dans ces cas-là,* $\mathtt{x}_i...\mathtt{x}_m$ *et* $\mathtt{y}_j...\mathtt{y}_n$ *sont des frontières vides.*

Donnons un exemple de ces deux types de coups dans une partie avec 3 points de départ. La représentation en chaîne initiale est `A.B.C`. Tout d'abord, nous effectuons un coup dans

2. Cette précaution permet de s'assurer de la validité du coup, il ne doit pas créer de sommet de degré supérieur à 3.

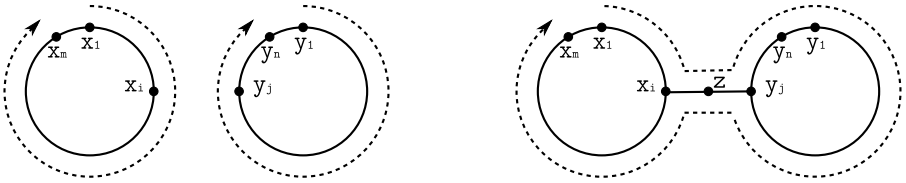


FIGURE 9.9 – Coup dans deux frontières.

deux frontières (`B` et `C`), ce qui nous amène à `A.BDCD`, puis nous effectuons un coup dans une frontière (`BDCD`), ce qui sépare la position en deux régions, et donne la chaîne de caractères `BDCDBE|BE.A`. La figure 9.10 montre les trois positions correspondantes.

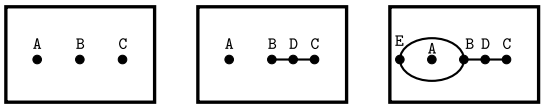


FIGURE 9.10 – Un début de partie à partir de 3 points de départ.

Les coups dans une frontière sont généralement bien plus nombreux que ceux dans deux frontières. En effet, une fois choisis les deux sommets à relier dans une même frontière, il faut ensuite répartir les autres frontières de la région dans laquelle on joue dans les deux régions nouvellement créées. S'il y a n frontières *différentes* à répartir, il y a 2^n façons de les répartir dans ces deux régions.

Ce phénomène est problématique ; par sa faute, le nombre d'options d'une position de Sprouts est souvent très élevé. Dans nos calculs, nous avons été fréquemment confrontés à des positions ayant plusieurs centaines d'options. Il a tout de même été possible d'accomplir ces calculs, grâce aux techniques énumérées ci-après.

* Lorsqu'il y a n frontières *identiques* à répartir dans deux régions, il n'y a pas 2^n, mais $(n+1)$ façons de le faire. Ce problème est abordé au paragraphe 9.5.2.
* Le jeu comporte de nombreuses transpositions. L'implémentation d'une table de transpositions (voir paragraphe 2.7.4) s'est avérée très efficace pour élaguer l'arbre de recherche dans le cadre du Sprouts.
* Les parties de l'arbre de recherche comportant les positions les plus simples (avec le plus petit nombre d'options) sont étudiées en priorité.

9.3 Réduction des chaînes

Les définitions données dans la section 9.2 sont suffisantes pour créer une première version d'un programme de calcul du Sprouts. Ce programme pourrait déterminer l'issue du jeu à p points de départ pour quelques valeurs de p (peut-être 4 ou 5), mais à cause des nombreuses chaînes de caractères équivalentes, la mémoire serait rapidement saturée. Il va donc être nécessaire d'identifier ces chaînes équivalentes, ce dont sera chargé le processus de *canonisation* décrit dans la section 9.4.

Nous allons expliquer dans cette section-ci des simplifications destinées à préparer ce processus de canonisation, et qui constituent l'étape de *réduction* des chaînes. La réduction est ainsi nommée dans la mesure où son exécution engendre des chaînes de caractères plus courtes, et utilisant moins de lettres différentes.

Les simplifications abordées dans cette section sont de deux types. Celles décrites dans les paragraphes 9.3.2 (concernant les sommets génériques `0` et `1`), 9.3.3 et 9.3.4 ne réduisent pas le nombre de sommets de l'arbre de jeu, mais permettent de rendre le processus de

canonisation plus rapide. Accessoirement, elles fournissent des chaînes de caractères plus lisibles, il est ainsi plus facile d'imaginer à quelle position une chaîne correspond.

Les simplifications décrites dans les paragraphes 9.3.1, 9.3.2 (concernant le sommet générique 2) et 9.3.5, elles, réduisent le nombre de sommets de l'arbre de jeu. En ce sens, elles effectuent également un travail de canonisation, puisqu'elle permettent de rapprocher l'arbre de jeu de l'arbre le plus petit possible, l'arbre canonique. Cependant, nous ne les avons pas énoncées dans la section 9.4, dans laquelle nous traitons uniquement le problème des doublons issus de la représentation en chaîne.

9.3.1 Suppression des parties mortes

Dans le tracé d'une position de Sprouts, on peut supprimer les sommets reliés à 3 arêtes. Puisque l'on ne peut plus jouer avec ces sommets, leur suppression n'altère pas le déroulement ultérieur du jeu. Nous dirons que ce sont des *sommets morts*. Dans la figure 9.11, représentant 4 positions équivalentes, la première étape correspond à la suppression de ces sommets.

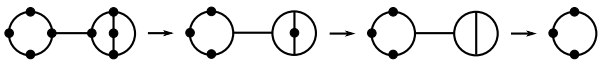


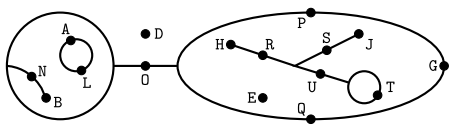
FIGURE 9.11 – Suppression des parties mortes.

La deuxième étape est la suppression d'un *sommet isolé*, devenu inutile, car on ne peut plus jouer avec lui. Enfin, la troisième étape est la suppression des régions mortes (ne comprenant plus aucun sommet), ainsi que des arêtes inutiles (tant que la suppression de ces arêtes ne modifie pas le degré des sommets encore vivants).

Ces simplifications peuvent s'opérer avec les représentations en chaînes de caractères. Tout d'abord, on supprime les sommets morts, que l'on détecte sachant qu'ils apparaissent 3 fois dans la chaîne. Puis nous supprimons les frontières vides (celles dont tous les sommets étaient morts), et enfin nous supprimons les régions mortes (celles avec 0 ou 1 vie, ce qui fait disparaître les sommets isolés).

Dans l'exemple du paragraphe 9.2.3, nous pouvons supprimer les 5 sommets morts `M`, `C`, `F`, `I` et `K`. Il n'y a pas de frontière vide, mais la région `KT` est morte. La nouvelle représentation en chaîne devient :

```
AL|AL.BNN|D.OPGQO|E.HRSJSUTUR.QGP
```



9.3.2 Sommets génériques

Nous pouvons remplacer les sommets vierges (ceux qui apparaissent une seule fois dans la totalité de la chaîne, à l'intérieur d'une frontière contenant un seul sommet) par le sommet générique « `0` ». Nous remplaçons également les sommets qui apparaissent une seule fois, mais qui ne sont pas vierges (c'est-à-dire qu'ils apparaissent dans une frontière avec plusieurs sommets) par le sommet générique « `1` ». Cette utilisation de sommets génériques est possible car il n'y a pas de risque de confusion : ces sommets n'apparaissent qu'une fois dans la chaîne et ne nécessitent pas d'être distingué par une lettre en propre. Le choix des notations est lié au degré des sommets concernés.

L'utilisation de ces sommets génériques permet de limiter le nombre de lettres utilisées, ce qui simplifiera le processus de canonisation expliqué dans la section 9.4.

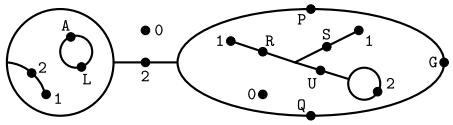
Le première idée nouvelle par rapport à la représentation de [4] est l'introduction du sommet générique « 2 ». L'idée est d'utiliser ce sommet générique pour les sommets de degré 2 (et qui devraient donc apparaître deux fois dans la chaîne), mais pour lesquels aucune confusion n'est possible. Deux cas sont possibles.

Le premier cas est celui des sommets qui apparaissaient deux fois à l'origine, mais qui n'apparaissent plus qu'une seule fois à cause de la suppression d'une région morte, comme `T` dans notre exemple. Comme le sommet n'apparaît plus qu'une seule fois dans la représentation, il est suffisant de retenir son degré, sans lui attribuer une lettre en propre.

Le deuxième cas est celui des sommets qui apparaissent deux fois d'affilée dans une même frontière, comme `N` ou `O` dans notre exemple. L'utilisation des lettres permet normalement de bien faire correspondre les deux côtés d'un sommet de degré 2, mais comme ici les deux lettres se suivent immédiatement, il n'y a pas de risque de confusion, et l'on peut utiliser le sommet générique « 2 ».

La représentation en chaîne de l'exemple devient donc :

```
AL|AL.12|0.2PGQ|0.1RS1SU2UR.QGP
```



L'interprétation graphique de ce sommet générique réside dans la figure 9.12. Le premier cas est illustré par la position de gauche, et le deuxième, par celle du milieu. Le sommet générique de type « 2 » est indiqué par une flèche sur chaque position.

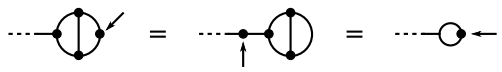


FIGURE 9.12 – Sommet générique 2.

Ces deux sommets ont un rôle équivalent, car dès qu'un coup sera joué avec un de ces sommets, toute la partie de droite de la position qui les contient sera morte, et pourra alors être supprimée. On peut donc représenter ces positions de manière simplifiée comme sur la position de droite.

9.3.3 Pays

Lorsque la position étudiée est découpable (voir la section 2.6), nous la décomposons en une somme de positions indépendantes, appelées *pays*. Le caractère séparateur des pays dans les chaînes de caractères est logiquement « + ».

Il est facile de détecter les pays dans une chaîne de caractères : chaque pays est un ensemble de régions, et deux pays différents doivent n'avoir aucune lettre en commun.

Dans notre exemple, il y a 2 pays, et la chaîne devient :

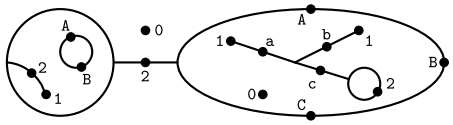
```
AL|AL.12+0.2PGQ|0.1RS1SU2UR.QGP
```

9.3.4 Renommage des lettres

À ce point, les lettres désignent des sommets qui apparaissent deux fois dans la chaîne. Ces sommets sont de deux types différents : ceux qui apparaissent deux fois dans la même frontière, que nous désignerons par des lettres minuscules, et ceux qui apparaissent dans deux régions différentes, que nous désignerons par des lettres majuscules. L'idée de distinguer ces deux types de sommets est nouvelle par rapport à la représentation de [4].

Nous renommons les sommets minuscules à partir de « `a` » pour chaque nouvelle frontière, et nous renommons les sommets majuscules à partir de « `A` » pour chaque nouveau pays, sans pour l'instant nous soucier de quelle lettre nous affectons à chaque sommet.

```
AB|AB.12+0.2ABC|0.1ab1bc2ca.CBA
```



L'intérêt de cette distinction entre les deux types de sommets vient du fait que les sommets minuscules sont forcément internes à une frontière donnée. Cela permet de réutiliser les mêmes lettres minuscules d'une frontière à l'autre, et donc de diminuer le nombre total de lettres utilisées. Ce phénomène ne se produit pas dans la position de l'exemple, car il faut au moins deux frontières utilisant des minuscules pour qu'il soit apparent. En diminuant le nombre de lettres majuscules utilisées, on augmente l'efficacité du processus de canonisation décrit plus loin.

La distinction entre les majuscules et les minuscules pose cependant quelques difficultés lors du calcul des options. Si l'on relie deux frontières qui utilisent chacune des lettres minuscules, un conflit se produit entre les lettres minuscules de la première frontière et celles de la seconde frontière. Autre souci, lorsque l'on relie une frontière à elle-même, il est possible que certaines minuscules de cette frontière deviennent des majuscules.

Pour éviter tout problème, avant de calculer les options d'une position donnée, nous renommons toutes les lettres de la position en majuscules. Puis, lors de la phase de réduction de ces options, nous distinguons à nouveau les minuscules des majuscules. Ces opérations s'avèrent coûteuses en temps de calcul, mais cela est compensé par une efficacité plus élevée de la canonisation. Au final, la distinction entre majuscules et minuscules s'est révélée relativement neutre sur le temps de calcul, et a permis une réduction notable de la taille des bases de données.

Enfin, remarquons[3] que les minuscules pourraient être remplacées par de simples parenthèses. Dans une chaîne, en remplaçant la première occurrence de chaque lettre minuscule par une parenthèse ouvrante, et la deuxième occurrence par une parenthèse fermante, on obtiendrait un parenthésage correct. Nous n'avons cependant pas implémenté cette fonctionnalité, pour des raisons de compatibilité avec le jeu sur des surfaces, raisons qui seront abordées dans le chapitre 11 (§11.4.1).

9.3.5 Équivalences de régions

Quand une région possède 3 vies ou moins, nous obtenons une position équivalente en fusionnant les frontières. Par exemple, la région `A.BC` devient `ABC`, et `2.2` devient `22`. De plus, on peut remplacer toute minuscule qui intervient dans une telle région par le sommet générique `2` (par exemple, la région `aAaB` devient `2AB`).

3. Cette remarque est due à Dan Hoey.

En effet, considérons deux positions identiques en tous points, hormis qu'elles contiennent deux régions équivalentes de ce type. Les coups disponibles à partir de ces deux positions sont les mêmes : il y a les coups à l'extérieur de la région, et les coups à l'intérieur de la région, qui consistent à relier deux des sommets de la région en créant un sommet générique 2.

L'équivalence provient du fait qu'il sera toujours possible de relier toute paire de sommets de la région, quels que soient les coups qui seront joués, du moment qu'ils sont encore de degré ≤ 2.

Le même argument permet de montrer qu'une région du type `A.B.C.D` est équivalente à `AB.CD`. Par contre, la région `AB.C.D` n'est pas équivalente à ces deux régions, car le coup indiqué en pointillés sur la figure 9.13 conduit à séparer les sommets `C` et `D`, ce qui n'est pas possible avec les deux autres régions.

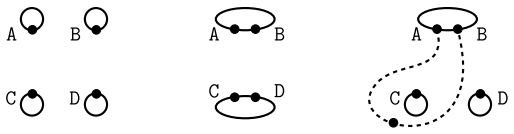


FIGURE 9.13 – Deux régions équivalentes, et une troisième qui diffère.

Cette équivalence des petites régions est une idée nouvelle par rapport à la représentation de [4], et s'avère très utile dans notre programme, en réduisant considérablement le nombre de chaînes à étudier.

9.4 Canonisation des chaînes

Nous savons déjà que plusieurs chaînes peuvent représenter la même position. La position de notre exemple pourrait ainsi être également représentée par :

```
BA2C.0|0.2ca1ab1bc.CBA+AB.21|AB
```

De la même façon que dans [4], nous appellerons *canonisation* dans ce chapitre le choix d'une chaîne particulière parmi toutes les chaînes équivalentes possibles représentant une position. En fusionnant des branches équivalentes dans l'arbre de jeu, la canonisation permet de diminuer efficacement la consommation de mémoire et le temps d'exécution. Par contre, la canonisation elle-même consomme du temps de calcul, et il faut trouver un juste équilibre entre les gains et les pertes liés à son utilisation.

9.4.1 Orientation

Jusqu'ici, une convention nous permettait de déterminer le sens dans lequel il fallait tourner autour des frontières lorsque l'on cherchait à déterminer une représentation en chaîne d'une position de Sprouts. Nous avions *orienté* le plan, et cette orientation s'appliquait à chacune des régions de la position. On obtiendrait une représentation tout aussi correcte en changeant l'orientation du plan, ou, de manière équivalente, si l'on considérait le symétrique de la position.

Or, nous pouvons aller plus loin. Si nous changeons l'orientation d'une unique région, c'est-à-dire, si nous changeons simultanément le sens de toutes les frontières dans sa représentation, nous obtenons une représentation équivalente. Cette remarque nous permet d'identifier un certain nombre de représentations en chaînes, et le processus de canonisation en sera plus efficace.

Il est à noter qu'il est interdit de changer l'orientation d'une frontière séparément. Nous pouvons seulement changer l'orientation d'une région complète, c'est-à-dire l'orientation

de toutes les frontières d'une région donnée. Nous pouvons fournir un contre-exemple : `12A.1B222|2A|2B` représente une position gagnante en version misère (à gauche sur la figure 9.14), tandis que `12A.222B1|2A|2B` représente une position perdante[4] (à droite sur la figure 9.14). Ces deux positions ne sont donc pas équivalentes.

FIGURE 9.14 – Deux positions qui ne sont pas équivalentes.

9.4.2 Canonisation

Nous expliquons maintenant le processus de canonisation. Commençons par définir une équivalence sur l'ensemble des chaînes. Deux chaînes seront équivalentes si elles sont égales modulo :

* le premier sommet choisi dans l'énumération des sommets de la frontière.
* l'ordre des frontières dans la région.
* l'ordre des régions dans le pays.
* l'ordre des pays dans la position.
* l'orientation choisie pour chaque région.
* un renommage quelconque des sommets non génériques (les lettres).

Nous pouvons maintenant définir formellement le terme de *canonisation* : nous dirons que c'est le choix d'une chaîne unique dans chaque classe d'équivalence. Ce choix devrait être aussi simple que possible, et nous avons choisi la chaîne minimale pour l'ordre lexicographique suivant :

`0 < 1 < 2 < a < b < ... < A < B < ... < . < | < +`

Une courte réflexion montre que la chaîne canonisée de l'exemple précédent est :

`0.1ab1bc2ca.ABC|0.2ABC+12.AB|AB`

ce qui correspond à la figure 9.15 (qui est identique à la précédente, hormis un renommage des sommets `A` et `C` à droite).

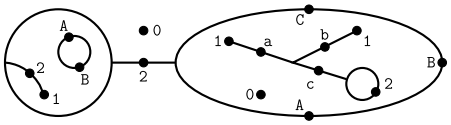


FIGURE 9.15 – Nom des sommets après canonisation.

9.4.3 Pseudo-canonisation

Réaliser une canonisation parfaite est une opération très coûteuse en temps de calcul. Autant ordonner frontières, régions, pays, ou choisir la meilleure orientation de chaque région ne pose pas de problème particulier, autant le renommage des sommets est difficile à mettre en œuvre. En particulier, celui des lettres majuscules, c'est-à-dire des sommets qui appartiennent à deux régions différentes.

4. Des exemples plus évidents existent peut-être, mais celui-ci est le plus simple que nous connaissions, et malheureusement, il ne peut être vérifié que par un calcul informatique, vu sa complexité.

ABCDEF|ABCDEG|FG
ABCDEF|ABCDGF|EG
ABCDEF|ABCGEF|DG
ABCDEF|ABGDEF|CG
ABCDEF|BCDEFG|AG

TABLE 9.1 – Chaînes de caractères représentant une même position.

Si un pays possède k lettres majuscules, une méthode de vraie canonisation consisterait à choisir la chaîne minimale pour l'ordre lexicographique parmi les $k!$ renommages possibles des sommets. Comme on rencontre fréquemment des pays comprenant de multiples lettres majuscules, cet algorithme est inenvisageable du fait de sa complexité factorielle.

Pour cette raison, en pratique, nous nous contentons d'effectuer une *pseudo-canonisation*, c'est-à-dire une canonisation imparfaite : la même position pourra être représentée par des chaînes différentes. Notre pseudo-canonisation peut par exemple engendrer les chaînes `0.AB.CD|0.AB|CD` et `0.AB.CD|0.CD|AB`, bien qu'elles représentent les mêmes positions.

Notre algorithme de pseudo-canonisation se décrit rapidement : on renomme les lettres majuscules à partir de la lettre « `A` » dans leur ordre d'apparition, puis on trie la chaîne. Enfin, on effectue une seconde fois ces deux opérations, car l'expérience montre que cela est plus efficace tant au niveau de la mémoire consommée que du temps de calcul.

Le tri est effectué avec un ordre particulier. Pour distinguer deux chaînes, nous utilisons dans un premier temps l'ordre suivant :

```
0 < 1 < 2 < a = b = ... < A = B = ... < . < | < +
```

Puis, si cet ordre n'a pas permis de départager les deux chaînes, l'ordre lexicographique décrit au paragraphe précédent s'en charge.

Voici l'intérêt de cet algorithme de tri. Les trois représentations en chaînes suivantes désignent la même position : `ABCDE|ABC|DF|EF` ; `ABC|ABDEC|DF|EF` ; `AB|ACDEF|BF|CDE`. Si l'on n'utilise que l'ordre lexicographique pour trier les chaînes, notre pseudo-canonisation engendre ces trois chaînes. Mais si l'on utilise d'abord le deuxième ordre, seule la première représentation est engendrée. Le deuxième ordre permet en effet de commencer par trier les régions seulement en fonction du nombre de lettres qu'elles contiennent.

Avec cet algorithme de pseudo-canonisation, nous ne perdons que quelques pourcents de mémoire par rapport à une vraie canonisation dans les calculs d'issue, tandis que le temps de calcul nécessaire augmente raisonnablement avec la taille des chaînes. L'algorithme de vraie canonisation a lui une complexité si élevée qu'il ne permettrait pas de calculer l'issue des jeux avec p points de départ au-delà de $p \geq 5$ ou 6.

Dans le but d'évaluer les performances de notre pseudo-canonisation, nous avons développé l'arbre de jeu complet pour p points de départ avec $p \leq 7$. La table 9.1 présente des positions extraites de l'arbre de jeu complet pour le jeu à 4 points de départ.

Toutes ces chaînes représentent la même position, et une vraie canonisation n'aurait affiché qu'une seule chaîne au lieu des cinq. Par contre, cette position n'est pas nécessaire pour calculer l'issue du jeu à 4 points de départ. En fait, les performances de la pseudo-canonisation (par rapport à la vraie canonisation) sont meilleures dans un calcul réel d'issue que dans le développement d'arbres complets. Cela est dû au fait que dans un calcul d'issue, on rencontre surtout des chaînes plus faciles à calculer, avec moins de lettres majuscules.

La table 9.2 donne le nombre de chaînes pseudo-canonisées stockées après un développement complet de l'arbre de jeu avec p points de départ.

Cette table pourrait aider à comparer les performances de notre pseudo-canonisation à celles d'autres programmes. Les résultats décrits dans la section suivante vont nous permettre d'évaluer ses performances dans l'absolu.

p	nombre de chaînes
2	18
3	157
4	1796
5	24784
6	393103
7	6849627

TABLE 9.2 – Performance de notre pseudo-canonisation sur les positions de départ du Sprouts.

p	nombre d'arbres canoniques
2	10
3	55
4	713
5	10461
6	150147
7	2200629

TABLE 9.3 – Nombre d'arbres canoniques dans les arbres de jeu des positions de départ.

9.4.4 Complexité spatiale

Étant donnée une position de départ du Sprouts à n points, combien y a t-il de positions différentes dans son arbre de jeu ? Ce problème, celui de la *complexité spatiale* du jeu de Sprouts (voir le paragraphe 2.7.2 qui introduit cette notion) est difficile à définir clairement.

En effet, le jeu de Sprouts n'est pas stricto sensu un jeu court, étant donné qu'il y a une infinité de façons de relier deux points par une ligne. On pourrait ainsi énoncer que la complexité spatiale du jeu de Sprouts est infinie. Cependant, nous avons vu qu'en identifiant les positions égales à déformation près, le Sprouts redevient un jeu court, et l'on pourrait alors déterminer la complexité spatiale d'une position. Par exemple, la complexité spatiale de la position étudiée dans la figure 9.2 serait 25.

Mais est-il normal que dans ces 25 positions, on compte chacune des positions égales à symétrie près ? Ou celles qui sont égales modulo l'équivalence entre intérieur et extérieur ? Il paraît normal de ne pas les compter, car la représentation en chaîne intègre ces équivalences automatiquement. Ainsi, la complexité spatiale de cette position passerait de 25 à 9.

Et que penser des équivalences décrites au paragraphe 9.3.1 (suppression des parties mortes), ou au paragraphe 9.3.2 (utilisation du sommet générique 2) ? Où s'arrêter ?

La notion qui permet de tenir compte d'absolument toutes les équivalences imaginables est celle d'arbre canonique (voir la section 2.5). Deux positions avec le même arbre canonique sont susceptibles d'être identifiées par une équivalence, tandis que deux positions avec deux arbres canoniques différents sont deux positions essentiellement différentes, qu'aucune équivalence ne pourra identifier. Notre programme permet de calculer les arbres canoniques, ce qui nous permet de déterminer le nombre de positions essentiellement différentes dans les arbres de jeu des positions de départ. Ceci est résumé dans la table 9.3.

Quelle que soit la définition choisie en définitive pour la complexité spatiale, il sera ainsi possible de la minorer par le nombre donné dans cette table. Par interpolation, nous pouvons en déduire un minorant crédible de la complexité spatiale pour un nombre n de points de départ, en remarquant que le rapport de deux termes de la colonne de droite se rapproche approximativement de 14,5. On obtient la formule :

$$2{,}2 \times 10^6 \times 14{,}5^{n-7}$$

La figure 9.16 montre la bonne corrélation entre la loi extrapolée et les valeurs exactes

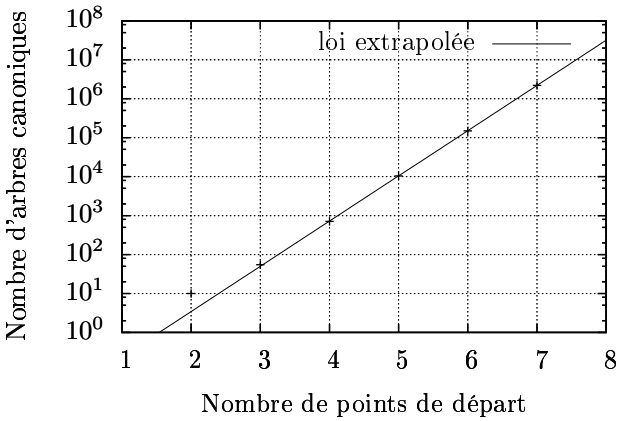


FIGURE 9.16 – Nombre d'arbres canoniques en fonction du nombre de points (échelle logarithmique)

connues jusqu'à 7 points de départ. En supposant que cette loi reste globalement valable pour un nombre plus élevé de points initiaux, on en déduit une complexité spatiale de $5{,}8 \times 10^{59}$ pour $n = 53$, la position de départ la plus grande que nous ayons calculée. Pour $n = 44$ (nous avons calculé toutes les issues des positions à 44 points de départ ou moins), on trouve 2×10^{49}. C'est une complexité spatiale supérieure à celle du jeu d'échecs.

Bien sûr, nous ne serions pas capable de résoudre le jeu d'échecs. Ce résultat montre surtout que la complexité spatiale n'est pas un critère suffisant de difficulté pour les jeux combinatoires. De fait, le Sprouts a certaines différences avec le jeu d'échecs qui rendent sa résolution plus facile, à complexité spatiale identique. Les arbres de jeu du Sprouts sont très déséquilibrés, ce qui permet d'obtenir une résolution rapide en menant les calculs dans les parties les plus simples de ces arbres, l'utilisation du nimber permet de simplifier l'étude des positions découpables, et le Sprouts comporte de plus nombreuses transpositions que le jeu d'échecs. Toutes ces raisons sont détaillées dans le chapitre 10 concernant la résolution du jeu de Sprouts.

Enfin, une comparaison entre les tables 9.2 et 9.3 permet d'évaluer la qualité de notre pseudo-canonisation. Dans le pire des cas, le nombre de chaînes de caractères est à peine trois fois plus élevé que le nombre d'arbres canoniques. Ceci est probablement améliorable, mais dans la mesure où en pratique, les doublons se trouvent dans des parties des arbres de recherche que nous n'avons pas besoin d'étudier, développer une meilleure pseudo-canonisation n'est clairement pas un objectif prioritaire si l'on désire pousser les calculs du Sprouts un peu plus loin.

9.5 Programmation

9.5.1 Généralités

Les notions théoriques que nous venons de présenter, et qui sont nécessaires pour développer une représentation performante des positions du jeu de Sprouts, sont loin d'être évidentes. La programmation de ces notions n'est pas non plus une partie de plaisir. Elle nécessite l'écriture de nombreuses fonctions [5], pas toujours faciles à écrire, et est donc une source de bugs dont il faut se méfier.

Il est probable que ces difficultés soient la raison pour laquelle si peu de programmes capables de mener des calculs sur le jeu de Sprouts ont vu le jour jusqu'ici. En effet, outre

5. De l'ordre de 2 000 lignes de code au total.

le programme d'Applegate, Jacobson et Sleator en 1991 [4] et le nôtre, nous avons juste connaissance d'un autre programme, jamais publié cependant [6].

Dans l'exécution de notre programme, la plus grande partie du temps de calcul est consommée par nos fonctions de calcul des options, du fait de leur complexité. Ceci n'est pas évident a priori ; pour les autres jeux combinatoires, c'est plutôt les accès aux tables de transpositions, du fait de leur grande taille, ou la gestion de l'arbre de recherche, qui s'avèrent les plus coûteux. Il est donc important que cette partie du programme soit efficace. Nous la détaillerons au paragraphe 9.5.3.

9.5.2 Frontières multiples

La représentation que nous avons détaillée jusqu'ici présente un inconvénient. La représentation de la position de départ à 8 points est `0.0.0.0.0.0.0.0`, or cette écriture gaspille de l'espace mémoire, et de plus, elle n'est pas très lisible. Nous utilisons ainsi dans notre programme la notation compacte `0*8`.

Il est à noter que cette notation compacte pourrait être étendue à d'autres frontières que `0`. La position `0.1a1a.1a1a` pourrait ainsi être notée `0.1a1a*2`. En pratique, la quasi-totalité des frontières multiples rencontrées dans nos calculs sont des `0`, et nous nous sommes donc limités au traitement de ce cas.

La prise en compte des frontières multiples est par ailleurs incontournable lors de la génération des coups. En effet, lorsque l'on joue un coup dans une frontière, cela crée deux nouvelles régions et il faut alors répartir de toutes les façons possibles les autres frontières de la région dans les deux régions nouvellement créées.

S'il s'agit d'une région contenant n occurrences de la frontière `0`, une programmation naïve conduirait à 2^n répartitions possibles, alors qu'en tenant compte du fait que ces n frontières sont de même nature, le nombre de répartitions possibles est en réalité de $n+1$ seulement. Cette astuce de programmation est devenue nécessaire à partir d'une quinzaine de points de départ, car le simple calcul des options de la position de départ prenait plusieurs secondes à cause de ce problème (et évidemment, deux fois plus de temps à chaque fois que l'on rajoutait un point de départ).

En général, une région est constituée à la fois de frontières `0` et de frontières quelconques. Pour obtenir l'ensemble des répartitions possibles, il faut considérer les $n+1$ façons de répartir les n frontières `0`, et les 2^k façons de répartir les k frontières quelconques. Le nombre total de répartitions est donc de $(n+1) \times 2^k$.

9.5.3 Graphe d'appel

Nous décrivons ici la liste des fonctions utilisées pour calculer les options d'une position. Ce document n'est pas assez exhaustif ni explicite pour permettre de reproduire exactement nos algorithmes, mais permet de se faire une bonne idée des techniques mises en œuvre.

Les fonctions présentées sur l'arbre de la figure 9.17 sont exécutées dans l'ordre d'un parcours en profondeur de l'arbre, la priorité étant donnée aux fonctions placées en haut. Cet ordre est respecté partout, sauf dans certaines fonctions liées au tri (en bas de la figure), qui peuvent être exécutées plusieurs fois (nous détaillerons lesquelles un peu plus loin).

Le code s'exécute en deux blocs indépendants : on commence par calculer les options non canonisées (bloc du haut), puis on les canonise dans un second temps (bloc du bas). Ces fonctions sont implémentées en 4 classes, matérialisées par les grands cadres rectangulaires : la classe `Frontière`, à droite, dans laquelle on retrouve les fonctions ne se déroulant que dans une frontière, et de même, les classes `Région`, `Pays` et `Position`.

6. Il s'agit de *Aunt Beast*, programmé par Josh Purinton, un des participants au *World Game Of Sprouts Association* (W.G.O.S.A.), principal lieu d'échange autour du jeu de Sprouts sur la toile. `http://www.wgosa.org/`

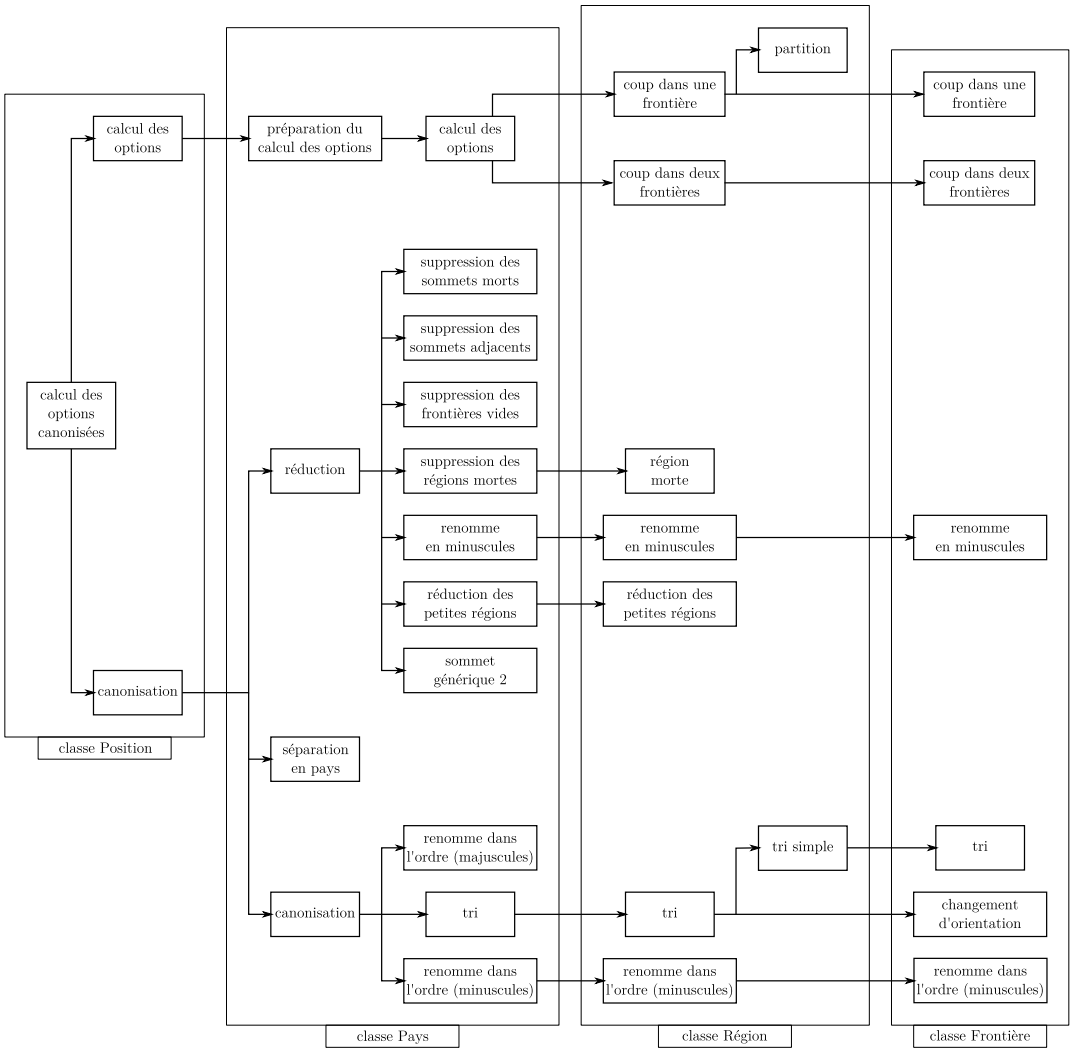


FIGURE 9.17 – Graphe d'appel des fonctions de calcul des options.

Nous décrivons dans la suite ce que chacune des fonctions effectue, dans l'ordre de leur exécution par le programme.

Calcul des options non canonisées

- `Position::calcul_des_options` : calcule les coups que l'on peut jouer à partir d'une position. Un coup ne se déroule que dans un pays, sans modifier les autres.

- `Pays::préparation_du_calcul_des_options` : renomme les sommets minuscules (`a`, `b`, ...) en majuscules (`A`, `B`...) comme expliqué dans le §9.3.4. Par exemple, `0.1a1a.AB|AB` devient `0.1C1C.AB|AB`. Dans la suite, les chaînes de caractères ne contiendront que des sommets génériques et des majuscules jusqu'à l'exécution de `Pays::renomme_en_minuscules`.

- `Pays::calcul_des_options` : calcule les coups jouables dans un pays. Un coup se déroule toujours à l'intérieur d'une seule région [7].

7. Il peut avoir de l'influence dans d'autres régions par l'intermédiaire des sommets situés à la frontière, mais cette influence ne se manifestera que dans les fonctions de canonisation, où l'on repèrera les sommets « morts ».

- `Région::coup_dans_une_frontière` : calcule les coups jouables en reliant un sommet d'une frontière à un autre sommet d'une même frontière. Une nouvelle région est alors créée.

- `Région::partition` : quand on sépare une région en deux en reliant une frontière à elle-même, cette fonction permet de répartir de toutes les façons possibles les autres frontières de la région, avec la technique du §9.5.2.

- `Frontière::coup_dans_une_frontière` : calcule les deux frontières obtenues quand on relie une frontière à elle-même. Il faut faire attention de ne pas relier un sommet de degré 2 (2 ou une lettre) à lui-même.

- `Région::coup_dans_deux_frontières` : calcule les coups jouables en reliant un sommet d'une frontière à un sommet d'une autre frontière. Ces deux frontières fusionnent alors.

- `Frontière::coup_dans_deux_frontières` : calcule les moitiés de frontière que l'on peut obtenir si la frontière étudiée est reliée à une autre frontière.

Canonisation des options

- `Position::canonisation` : commence par réduire la chaîne, puis teste si le pays a été séparé en plusieurs pays suite au coup qui vient d'être joué. Enfin, termine la canonisation de la chaîne.

- `Pays::réduction` : fonctions destinées à simplifier la chaîne de caractères, voir la section 9.3.

- `Pays::suppression_des_sommets_morts` : suppression des sommets de degré 3, détectés facilement, car une même lettre apparaît trois fois dans la chaîne.

- `Pays::suppression_des_sommets_adjacents` : si une même lettre apparaît deux fois de suite dans une frontière, on supprime une des deux occurrences. Par exemple, `1AA` donne `1A`, ou `A11A` donne `11A`.

- `Pays::suppression_des_frontières_vides` : supprime les frontières devenues vides suite à la suppression des sommets de degré 3.

- `Pays::suppression_des_régions_mortes` : supprime les régions mortes.

- `Région::région_morte` : teste si la région est morte. On ne peut plus jouer dans une région si elle ne contient plus que 0 ou 1 vie.

- `Frontière::renomme_en_minuscules` : renomme en minuscules les sommets qui n'appartiennent qu'à une seule frontière (à partir de `a`). Cette fonction s'exécute dans chaque frontière indépendamment des autres, les fonctions correspondantes des classes `Région` et `Pays` ne servent qu'à l'appeler sur chaque frontière.

- `Pays::réduction_des_petites_régions` : simplifications sur les régions à 3 vies ou moins décrites dans le §9.3.5.

- `Pays::sommet_générique_2` : utilisation de la lettre 2 pour remplacer les lettres n'apparaissant qu'une seule fois dans le pays (suite à la suppression des sommets adjacents et des régions mortes).

- `Pays::séparation_en_pays` : teste si le pays est séparable en plusieurs pays, suite au coup joué.

- `Pays::canonisation` : processus de canonisation discuté dans la section 9.4. Si ses trois fonctions filles sont numérotées dans l'ordre 1, 2, 3 en partant du haut, cette fonction appelle dans l'ordre 1 ; 2 ; 3 ; 1 ; 2.

- `Pays::renomme_dans_l'ordre(majuscules)` : les sommets apparaissant dans 2 régions sont renommés (`A`, `B`...) dans l'ordre où on les rencontre.

- `Pays::tri` : trie chaque région avec `Région::tri`, puis trie la liste des régions, en utilisant l'ordre du §9.4.3.

- `Région::tri` : on trie la région avec `Région::tri_simple` puis on change l'orientation de chaque frontière (§9.4.1), et l'on exécute la même fonction. Enfin, on garde la plus petite des deux possibilités pour l'ordre du §9.4.3.

- `Région::tri_simple` : on trie chaque frontière en appelant la fonction `Frontière::tri`, puis on trie la liste des frontières, à chaque fois avec l'ordre du §9.4.3.

- `Frontière::renomme_dans_l'ordre(minuscules)` : les lettres apparaissant deux fois dans une seule frontière sont renommés (`a`, `b`...) dans l'ordre où on les rencontre. Là encore, les fonctions correspondantes des classes `Région` et `Pays` ne servent qu'à appeler cette fonction sur chaque frontière.

9.6 Équivalences sporadiques

Nous nous sommes interrogés sur les améliorations supplémentaires que nous pourrions apporter à notre représentation des positions du Sprouts, pour la rendre plus performante. Ce que nous avons envisagé concerne essentiellement des équivalences qu'il serait possible d'implémenter, en plus de celles décrites dans la section 9.3. Comme souvent, la programmation de ces équivalences serait à double tranchant : le gain se ferait par la réduction de la taille des arbres de recherche et des tables de transpositions qui découlerait de l'identification des positions équivalentes, mais au prix d'une perte de rapidité d'exécution des fonctions de canonisation.

Voici comment nous avons procédé pour découvrir ces équivalences. On commence par calculer les arbres canoniques d'une grande quantité de positions, par exemple, toutes les positions de l'arbre de jeu de la position de départ à 5 points. Ceci fournit environ 24 800 positions dans l'état actuel de notre canonisation, pour environ 10 500 arbres canoniques différents. Ces deux nombres sont différents, c'est donc qu'il y a des doublons, des positions dont la représentation en chaîne est différente, mais qui ont le même arbre canonique. Ensuite, nous observons ces doublons. Certains d'entre eux sont dus à l'imperfection de notre pseudo-canonisation, c'est le cas de ceux présentés dans la table 9.1. Mais certains autres doublons ne sont pas liés à la pseudo-canonisation, et ils sont parfois dus à une équivalence plus générale, qu'il faut alors détecter puis démontrer.

Nous allons rapidement décrire certaines de ces équivalences dans la suite de cette section, avant de discuter de l'intérêt de leur implémentation. Précisons enfin que certaines des équivalences abordées ont été préalablement découvertes par un membre du W.G.O.S.A. et sont répertoriées sur la page suivante : `http://wgosa.org/Equivalences.htm`

9.6.1 Équivalences de frontières

Nous connaissons une unique équivalence de frontières : on peut indistinctement utiliser la frontière `1` ou la frontière `22` dans une position, ce qui signifie par exemple que les positions `0.1a1a.1` et `0.1a1a.22` sont équivalentes.

Attention, ceci n'est valable que si ces frontières sont isolées. Les positions `0*2.1AB|AB` et `0*2.22AB|AB` n'ont ainsi pas le même arbre canonique.

9.6.2 Équivalences de régions

Le paragraphe 9.3.5 énonce une première équivalence de régions qui concerne les régions contenant 3 vies ou moins. Ainsi, on peut remplacer la région `A.B.C` par `A.BC` ou `ABC` dans une position, sans changer son arbre canonique.

Cependant, c'est loin d'être la seule équivalence de régions. En voici quelques autres :

* `A.B.C.D` $\sim$ `AB.CD` (également valable en remplaçant une ou plusieurs lettres par le sommet générique `2`).
* `1A` $\sim$ `22A` $\sim$ `ABC|BD|CD` $\sim$ `2AB|2B`
* `0.B|2AB` $\sim$ `0+1A`
* `22aAa` $\sim$ `1A.2`

Dans l'équivalence `1A` $\sim$ `22A`, la lettre `A` correspond à un sommet situé entre la région et le reste de la position. Un tel sommet n'est pas forcément nommé `A` dans toutes les positions rencontrées. Par exemple, cette équivalence permet de dire que `0.2.A|1B|AB` et `0.2.A|22B|AB` ont le même arbre canonique.

Hormis celle de la première ligne, qui est une généralisation des équivalences du paragraphe 9.3.5, ces équivalences se produisent entre des régions topologiquement très différentes, et sont donc difficiles à justifier par des arguments uniquement topologiques. Nous les qualifions donc de *sporadiques*.

À titre d'illustration, nous allons maintenant démontrer l'équivalence `1A` $\sim$ `2AB|2B`. Les autres équivalences peuvent se démontrer par des arguments similaires.

Démonstration. Sur la figure 9.18, nous pouvons voir l'aspect de l'arbre de jeu d'une position contenant la région `1A`, que nous notons `1A|resteA`. La chaîne `reste`, qui code le reste de la position, est associée à `A`, car une des régions de ce reste contient le sommet `A`.

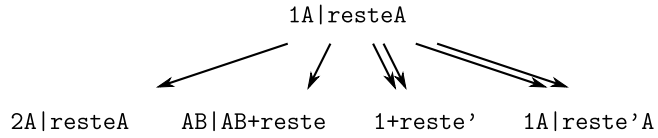


FIGURE 9.18 – Arbre de jeu d'une position contenant la région `1A`.

Les deux options de gauche correspondent à des coups joués dans la région `1A`. La première s'obtient en reliant le sommet `1` à lui-même, et la seconde, en le reliant au sommet `A`. Dans ce second cas, la position se scinde alors en deux pays, dont l'un est le reste de la position. Le sommet `A` est mort, donc il n'apparaît plus accolé au reste.

On peut voir deux flèches dirigées vers la troisième option, car en fait, la chaîne `1+reste'` code toutes les options obtenues en jouant un coup dans le reste qui utilise le sommet `A`. De même, la chaîne `1A|reste'A` code toutes les options obtenues en jouant un coup dans le reste qui n'utilise pas le sommet `A`.

Si, sur ce modèle, on établit maintenant l'arbre de jeu de la même position, en remplaçant la région `1A` par `2AB|2B`, on obtient la figure 9.19.

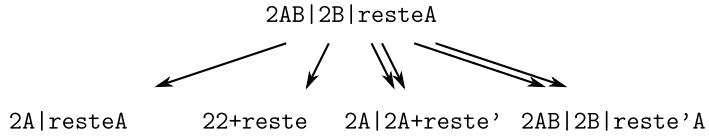


FIGURE 9.19 – Arbre de jeu d'une position contenant la région `2AB|2B`.

Les options `AB|AB+reste` du premier arbre et `22+reste` du second sont équivalentes, car les pays `AB|AB` et `22` ont le même arbre canonique. De même pour les options du type `1+reste'` et `2A|2A+reste'`, car `1` et `2A|2A` ont le même arbre canonique.

Enfin, il ne reste que les options qui correspondent à un coup joué dans le reste, et qui ne fait pas intervenir le sommet `A`. Ces options sont exactement du même type que les positions de départ. Ainsi, on peut régler ce cas en faisant une preuve par récurrence sur le nombre maximal de coups jouables à partir de la position de départ (à savoir, la hauteur de l'arbre canonique de cette position). □

9.6.3 Équivalences plus générales

Ces équivalences (les positions équivalentes ont le même arbre canonique) ne sont pas les seules qu'il est utile de considérer. Par exemple, voici des équivalences de régions qui

fournissent des positions qui n'ont pas le même arbre canonique, mais qui ont le même *arbre canonique réduit*.

* `0.A` $\sim$ `2A`
* `0.AB` $\sim$ `2AB`

Cette notion d'arbre canonique réduit est détaillée dans le chapitre 4, et si deux positions ont le même arbre canonique réduit, alors on sait qu'elles auront même issue tant en version normale que misère. Leur implémentation serait donc tout à fait envisageable, et accélérerait les calculs d'autant plus que certaines régions seraient remplacées par d'autres avec moins de vies, dont l'étude est plus rapide.

9.6.4 Intérêt

Les équivalences les plus importantes ont été présentées dans cette section. Les autres sont sans doute bien plus nombreuses, mais n'apparaissent quasiment pas dans les calculs réels. Il est probable que la programmation des équivalences que nous avons présentées déboucherait sur une amélioration des performances. La détection d'une équivalence ne nécessite en effet qu'un test, ce qui n'est pas coûteux en temps de calcul.

Cependant, le gain que l'on pourrait espérer serait sans doute de l'ordre de quelques pourcents, alors qu'au fil de la programmation du jeu de Sprouts, nous avons plusieurs fois apporté des améliorations qui divisaient le temps de calcul par un facteur qui s'exprime avec des puissances de 10. Il est donc peu probable à notre avis qu'une franche amélioration des résultats découle de la programmation de ces équivalences.

Chapitre 10

Le jeu de Sprouts

10.1 Introduction

Nous avons présenté le *Sprouts* dans le chapitre 2 qui concerne les jeux combinatoires. Dans ce chapitre, nous allons décrire les méthodes qui nous ont permis d'obtenir les stratégies gagnantes pour des parties de Sprouts avec divers nombres de points de départ, tant en version normale qu'en version misère.

10.1.1 Versions normale et misère

Dans la version *normale* du jeu, le joueur qui ne peut plus jouer a perdu, et dans la version *misère*, il a au contraire gagné. Dans les deux cas, il ne peut y avoir de match nul. De plus, comme le nombre de coups d'une partie commençant avec p points est fini (car majoré par $3p-1$), l'un des deux joueurs dispose forcément d'une stratégie gagnante.

Définition 14. *Dans ce chapitre, on notera S_p^+ le jeu de Sprouts à p points en version normale, et S_p^- en version misère.*

10.1.2 Résolution du jeu

Comme expliqué dans le paragraphe 2.7.1 du chapitre 2, plusieurs niveaux différents de résolution peuvent être considérés pour une position de départ donnée. L'information principale que l'on cherche en général à calculer est l'issue gagnante ou perdante de la position, ainsi que la stratégie concrète permettant au joueur en position de force d'obtenir ce résultat. On parle dans ce cas de *résolution faible*.

Puisque le jeu de Sprouts est impartial, on peut également chercher à calculer, en version normale, le nimber de la position. Le nimber est plus difficile à calculer puisqu'il contient plus d'informations que l'issue, mais il s'agit là aussi d'une résolution faible, si l'on utilise les algorithmes adéquats. Dans les deux cas, le calcul informatique consiste à trouver au sein de l'arbre de jeu un *arbre solution*, qui est beaucoup plus petit que l'arbre de jeu, et qui est suffisant pour déterminer la valeur (issue ou nimber) recherchée.

Il est également possible d'étudier les caractéristiques de la totalité de l'arbre de jeu d'une position de départ donnée. Le terme de *résolution forte* est généralement utilisé pour désigner le fait de calculer l'issue de toutes les positions apparaissant dans l'arbre de jeu. On peut généraliser ce terme au calcul de toute information qui nécessite d'explorer la totalité de l'arbre de jeu. L'information la plus complète possible (mais aussi la plus difficile à calculer en terme de ressources de mémoire) consiste à déterminer la forme exacte de l'arbre de jeu une fois totalement développé et une fois éliminées les branches redondantes : cet arbre est appelé *arbre canonique* de la position.

10.1.3 Historique des calculs de Sprouts

La détermination du joueur possédant une stratégie gagnante est difficile du fait de la complexité du jeu. Les premières analyses manuelles en 1967 n'ont permis de déterminer l'issue du jeu que jusqu'à S_6^+, et ce au prix de nombreuses pages de calcul. La première programmation du Sprouts par Applegate, Jacobson et Sleator [4] en 1991 (abrégé « AJS » dans la suite de ce chapitre), a ensuite permis de calculer l'issue jusqu'à S_{11}^+ en version normale et jusqu'à S_9^- en version misère.

AJS avaient formulé deux conjectures en se basant sur leurs résultats de calculs, une en version normale, et une en version misère.

Conjecture 1. *Le premier joueur a une stratégie gagnante sur S_p^+ si et seulement si p est égal à* 3, 4 *ou* 5 *modulo* 6.

Conjecture 2 (fausse)**.** *Le premier joueur a une stratégie gagnante sur S_p^- si et seulement si p est égal à* 0 *ou* 1 *modulo* 5.

Les progrès suivants en version normale ont été obtenus 15 ans plus tard par Josh Jordan Purinton, avec le calcul des issues jusqu'à S_{14}^+ en mai 2006 [34]. Nous avons ensuite publié nos premiers résultats sur le Sprouts avec les calculs d'issues, mais aussi de nimbers, jusqu'à S_{32}^+ en avril 2007 (ainsi que certaines valeurs éparses jusqu'à S_{47}^+). Puis nous avons réussi à obtenir les issues jusqu'à S_{44}^+ en janvier 2011 (ainsi que certaines valeurs éparses jusqu'à S_{53}^+). Tous les résultats obtenus jusqu'ici confirment la conjecture d'AJS.

Au niveau de la version misère, Josh Purinton et Roman Khorkov ont calculé les issues jusqu'à S_{15}^- en septembre 2007 [24], puis l'issue de S_{16}^- en janvier 2009 [25]. Ils ont calculé en particulier que S_{12}^- est gagnante, invalidant ainsi la conjecture d'AJS en version misère. De notre côté, nous avons ensuite publié l'issue misère de S_{17}^- en août 2009, avant d'obtenir finalement les issues jusqu'à S_{20}^- en mars 2011.

Josh Purinton et Roman Khorkov n'ont pas publié de document décrivant leurs techniques de calculs, mais nous avons pu confirmer tous leurs résultats. Les différents échanges que nous avons pu avoir par mail nous ont également convaincu du sérieux de leur travail.

Sinon, à notre connaissance, seul notre programme est capable de résoudre fortement les positions de départ en calculant leur arbre canonique. Nous avons calculé les arbres canoniques des positions de départ jusqu'à S_6 au début de l'année 2009, et l'arbre canonique de S_7 en janvier 2011.

10.2 Programme élémentaire de calcul

Les éléments nécessaires pour créer un premier programme de calcul du jeu de Sprouts sont en théorie assez limités. Il suffit de disposer d'une représentation des positions, d'une méthode de calcul des options d'une position, et d'un algorithme de calcul (récursif) de l'issue des positions. Nous présentons ces différents éléments ci-dessous.

10.2.1 Représentation des positions

Le Sprouts est un jeu que l'on pourrait qualifier de « graphique » ou de « topologique », car les joueurs tracent successivement des lignes reliant des points sur une feuille de papier. La représentation informatique du Sprouts à base de chaînes de caractères est difficile. Elle demande une analyse détaillée des positions, qui relève à la fois de la topologie et de la théorie des graphes. Le chapitre 9 est entièrement consacré à ce sujet. Nous donnons ici uniquement les principaux éléments de cette représentation.

Les positions de Sprouts sont composées essentiellement de trois éléments topologiques : les sommets, les régions et les frontières. On affecte une lettre pour désigner chacun des

sommets (les points) de la position. Les frontières sont alors représentées par des chaînes de caractères qui listent de façon circulaire les sommets dont elles sont composées. Les régions sont représentées par la liste de leurs frontières. Enfin, la position complète est un ensemble de régions.

```
0.1ab1bc2ca.ABC|0.2ABC+12.AB|AB
```

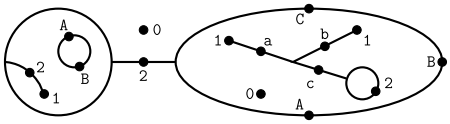


FIGURE 10.1 – Position de Sprouts et représentation en chaîne correspondante.

Ce processus complexe de description d'une position de Sprouts avec une chaîne de caractères peut mener à plusieurs chaînes de caractères différentes suivant les choix arbitraires faits au cours de la description. L'ensemble des opérations pour essayer d'associer une chaîne unique à une position de Sprouts donnée est appelé *canonisation*. Une canonisation parfaite étant extrêmement coûteuse en temps de calcul, nous réalisons en fait une pseudo-canonisation, qui essaye de trouver le meilleur compromis entre le temps de calcul et le nombre de *doublons* (chaînes différentes qui représentent en réalité une même position).

10.2.2 Options d'une position

L'intérêt de la représentation des positions de Sprouts est de permettre le calcul de la liste des options d'une position, uniquement grâce à des manipulations de chaînes. Là encore, ce processus est complexe. AJS ne le décrivent pas en détail dans leur article, expliquant à juste titre qu'il s'agit « d'une opération chirurgicale directe sur les chaînes de caractères, avec traitement des sommets et des régions mortes [...] dont nous omettons les détails sanglants ».

Une des particularités intéressantes du Sprouts est le nombre très variable d'options en fonction des caractéristiques topologiques de la position. Notamment, lorsqu'une région est coupée en deux par une ligne qui relie une frontière à elle-même, le joueur peut tracer la ligne de façon à répartir comme il le souhaite les autres frontières de la région dans les deux nouvelles régions créées. Si la région comporte 8 autres frontières, il y a 256 répartitions possibles ! On notera aussi que le calcul de la liste des options est très lent comparé à d'autres jeux à cause de la complexité des opérations nécessaires.

La complexité de la représentation et du calcul des options des positions de Sprouts rend ce jeu difficile à programmer. À notre connaissance, seuls deux autres programmes en plus du nôtre sont capables de réaliser des calculs de Sprouts :

* le programme d'AJS de 1991.
* le programme de Josh Purinton (programmé surtout de 2006 à 2010), aidé par Roman Khorkov.

Notre représentation des positions est directement basée sur celle d'AJS, mais nous ne savons pas précisément ce qu'il en est de celle utilisée par Purinton.

10.2.3 Algorithme élémentaire de calcul

Armé de la représentation des positions et de l'algorithme de calcul des options, on peut alors calculer facilement l'issue d'une position de Sprouts. Il suffit d'utiliser l'algorithme récursif 15 (alpha-bêta) qui découle directement de la définition même de l'issue, comme décrit dans la section 2.4 du chapitre 2.

Algorithme 15 Calcul récursif de l'issue de $\mathscr{P}$

```
1: calculer la liste des options de P
2: Pour chaque option P_i faire
3:     calculer l'issue de P_i, et si celle-ci est perdante, renvoyer gagnant
4: fin Pour
5: renvoyer perdant (car toutes les options sont gagnantes)
```

Cet algorithme élémentaire est disponibles dans notre programme, en utilisant l'onglet « WinLoss » de l'interface. Il permet de calculer les positions de Sprouts en version normale jusqu'à S_{11}^{+} en stockant environ 1 million de positions perdantes.

10.2.4 Transpositions

De nombreuses positions du jeu de Sprouts réapparaissent en plusieurs endroits de l'arbre de jeu. C'est le cas dès lors qu'un coup est joué dans un endroit de la position, puis un autre coup dans un autre endroit, sans que ces deux coups interfèrent. En inversant l'ordre des deux coups, on obtient ainsi une autre façon d'aboutir à la même position, comme dans la figure 10.2.

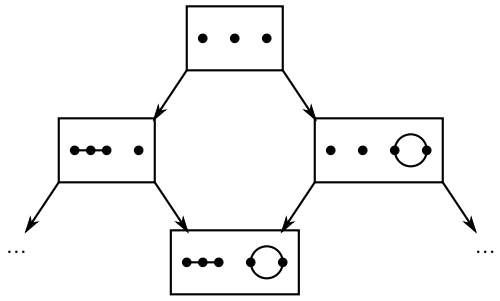


FIGURE 10.2 – Une transposition.

On peut noter sur cette figure la perte du caractère arborescent dès lors que l'on identifie les nœuds identiques.

Le Sprouts recèle de nombreuses transpositions de ce type. D'autres transpositions apparaissent suite aux opérations de canonisation. Il est à noter que le Sprouts étant un jeu impartial, il est même possible d'observer des transpositions entre des nœuds dont la distance à la racine n'a pas la même parité, ce qui n'est pas possible dans les jeux partisans, où les étages pairs correspondent aux positions où c'est le tour d'un joueur, et les étages impairs, aux positions où c'est le tour de l'autre joueur.

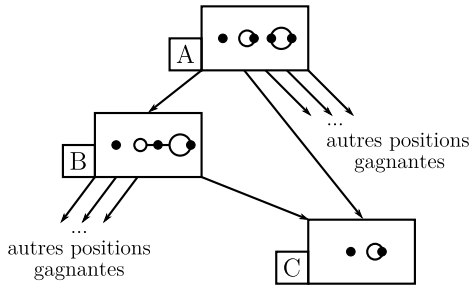


FIGURE 10.3 – Une transposition entre des étages de parités différentes.

Plutôt que de calculer plusieurs fois les mêmes positions, il est intéressant de stocker les résultats des positions déjà calculées dans une table de transpositions, pour pouvoir les réutiliser ensuite.

Une des limitations majeures d'AJS en 1991 était le manque de RAM des ordinateurs de l'époque, ce qui les a amené à stocker uniquement les positions perdantes dans la table de transpositions. Comme les positions gagnantes ne sont pas stockées, un calcul est nécessaire pour retrouver leur caractère gagnant. Heureusement, il suffit d'un seul calcul d'options, car l'un d'entre eux est perdant et est donc stocké dans la table de transpositions.

Cette idée de stockage des positions perdantes seulement permet de consommer moins de RAM, en échange d'un temps de calcul plus long. Nous avons repris cette idée dès le début de nos calculs de Sprouts, car la RAM était également notre facteur limitant initialement. Ce stockage des perdants uniquement a traversé toutes les évolutions du programme, et s'est maintenu jusqu'à maintenant... bien que notre facteur limitant soit désormais le temps de calcul, bien plus que la RAM.

10.3 Nimber

10.3.1 Idée d'utilisation des nimbers

Le Sprouts est un jeu impartial découpable. Lorsqu'une position est découpable en composantes indépendantes, on peut donc utiliser la théorie des jeux combinatoires pour déduire l'issue de la position à partir d'informations obtenues indépendamment sur chacune des composantes. AJS ont mis en œuvre dès 1991 une première idée, qui consiste à supprimer les composantes perdantes, car elles n'influencent pas l'issue de la position totale. Les premières versions (non publiées) de notre programme implémentaient également cette idée, avant de l'abandonner ensuite devant l'efficacité des nimbers.

À la fin de leur article, AJS suggèrent l'utilisation possible du nimber. Ce concept-clef des jeux impartiaux en version normale permet de déterminer le nimber d'une position (et donc son issue) à partir des nimbers des composantes, ce qui semble très prometteur pour séparer totalement le calcul des sommes en calculs indépendants sur chaque composante. Cependant, nous avons longtemps reporté l'implémentation des nimbers à cause d'une fausse idée, à savoir qu'il serait nécessaire de calculer tout le sous-arbre d'une position pour calculer son nimber.

10.3.2 Calcul du nimber d'une position

AJS indiquent en fin d'article que lors d'un calcul de nimber, on ne peut pas profiter de la simplification fondamentale qui consiste à déduire qu'une position est gagnante dès lors qu'une de ses options est perdante. Le nimber est une notion classique de la théorie des jeux impartiaux, mais de façon surprenante, il n'existe quasiment aucune étude sur le problème de la difficulté du calcul du nimber. L'idée que le calcul du nimber nécessite de calculer le nimber de toutes les options vient de la définition trompeuse du nimber d'une position sous la forme du mex (minimum exclu) du nimber des options.

Or, il suffit de réécrire le calcul du nimber sous une forme différente pour comprendre que toutes les options ne sont pas nécessaires. Imaginons que l'on calcule l'issue gagnante/perdante de la somme $\mathscr{P} + \mathbb{n}$, où $\mathscr{P}$ est une position de Sprouts et $\mathbb{n}$ une colonne du jeu de Nim de taille n. La position $\mathscr{P}$ est alors de nimber $\mathbb{n}$ si et seulement si le résultat est perdant. Comme il s'agit d'un calcul d'issue classique, on peut bien sûr déduire en cours de calcul qu'une position est gagnante dès qu'elle possède une option perdante.

Pour trouver le nimber d'une position, il suffit donc de faire des calculs d'issue de $\mathscr{P} + \mathbb{n}$, avec des valeurs croissantes de $\mathbb{n}$, jusqu'à obtenir un résultat perdant. AJS étaient sans doute extrêmement proches de cette idée, car ils décrivent la façon d'implémenter le concept d'une

somme $\mathscr{P} + \mathbb{n}$, sous la forme d'un couple avec une partie position et une partie nimber, et comment on pourrait effectuer des calculs d'issues à partir de là.

10.3.3 Calcul de l'issue d'une somme avec les nimbers

Pour obtenir l'issue d'une position somme de deux composantes $\mathscr{P}_1 + \mathscr{P}_2$, il n'est pas utile de calculer le nimber des deux composantes. Il suffit de calculer le nimber $\mathbb{n}_1$ de l'une d'entre elles seulement avec la méthode décrite ci-dessus, par exemple $\mathscr{P}_1$, puis de calculer l'issue de $\mathscr{P}_2 + \mathbb{n}_1$. L'algorithme 16 montre que l'implémentation complète de la théorie des nimbers tient en quelques lignes seulement. C'est notamment l'implémentation de cet algorithme qui a nous permis d'obtenir nos premiers résultats jusqu'à S_{32}^+ en 2007.

Algorithme 16 Calcul récursif de l'issue de $(\mathscr{P}, \mathbb{n})$

Si $\mathscr{P}$ est découpable sous la forme $\mathscr{P}_1 + \mathscr{P}_2$ **alors**
  $\mathbb{n}_1 \leftarrow 0$
  Tant que le calcul de l'issue de $(\mathscr{P}_1, \mathbb{n}_1)$ renvoie gagnant **faire**
    $\mathbb{n}_1 \leftarrow \mathbb{n}_1 + 1$
  fin Tant que
  renvoyer l'issue de $(\mathscr{P}_2, \mathbb{n} \oplus \mathbb{n}_1)$
sinon
  Pour chaque option $(\mathscr{P}_i, \mathbb{n})$ de la partie position et chaque option $(\mathscr{P}, \mathbb{i})$ de la partie nimber **faire**
    calculer l'issue de l'option, et si celle-ci est perdante, renvoyer gagnant
  fin Pour
  renvoyer perdant (car toutes les options sont gagnantes)
fin Si

10.3.4 Nécessité des calculs de nimber

Ce n'est qu'en 2009 que nous avons compris que l'algorithme 16 était inévitable en un certain sens. Nous expliquons les raisons théoriques qui le justifient dans le chapitre 3 : en fait, même un calcul d'issue gagnante/perdante qui n'utiliserait pas les nimbers calcule autant d'informations que l'algorithme 16.

Concrètement, cela signifie que si AJS publiaient la base de positions perdantes qu'ils ont obtenue en fin de calcul, nous pourrions en déduire les nimbers de certaines des composantes des positions découpables, et construire la base de couples (position, nimber) perdants qu'ils auraient obtenus s'ils avaient utilisés l'algorithme 16. Mieux, cette base serait bien plus petite, comme le montre les comparaisons du paragraphe suivant.

La base de positions obtenue par AJS, ou celle obtenue par notre programme avec l'algorithme 15, contient de manière cachée les nimbers de certaines positions, mais comme cette information n'est pas exploitée comme elle le devrait, elle y est présente en de multiples exemplaires, ce qui fait enfler la taille de la base. C'est d'ailleurs une situation vraiment paradoxale : l'information en apparence « complexe » de nimber est déjà présente dans les « simples » bases de positions perdantes.

10.3.5 Comparaisons des calculs avec et sans nimber

La figure 10.4 compare le nombre de positions perdantes stockées à la fin du calcul d'issue pour un certain nombre de points de départs dans différentes conditions. Les calculs ont été faits en repartant à chaque fois de la base de positions perdantes avec un point de moins, pour permettre une comparaison avec les calculs de 1991 d'AJS. Nous avons reporté sur le

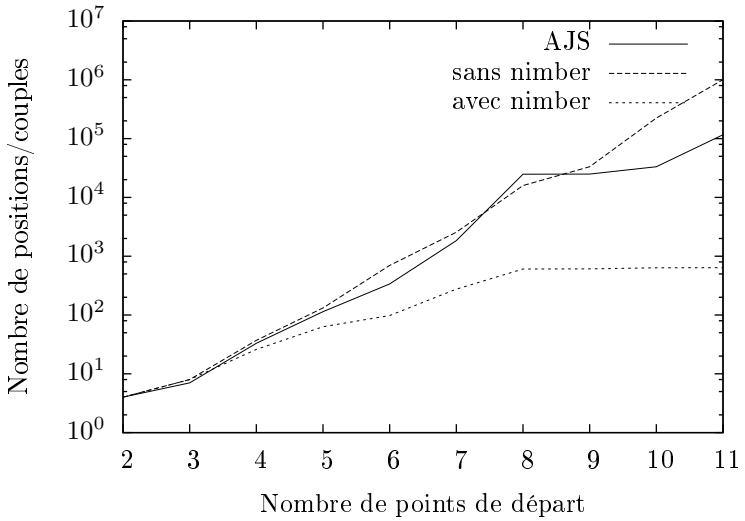


FIGURE 10.4 – Nombre de positions ou de couples perdants pour les calculs avec et sans nimbers (échelle logarithmique).

graphe le nombre de positions perdantes lors des calculs faits par AJS, le nombre de positions perdantes avec notre programme sans la théorie des nimbers, et enfin le nombre de couples (position, nimber) perdants avec notre programme lorsque l'on utilise la théorie des nimbers.

Le calcul fait avec notre programme sans la théorie des nimbers n'exploite pas du tout les découpages en positions indépendantes, alors que le calcul d'AJS utilise par contre la technique intermédiaire qui consiste à supprimer les composantes perdantes. On voit que cette technique intermédiaire de suppression des composantes perdantes permet de gagner tout de même un facteur 10 sur le nombre de positions stockées pour S_{11}^+.

On notera que l'échelle du nombre de positions est logarithmique. L'efficacité des calculs à base de nimbers est vraiment remarquable : dès 11 points de départ, il y a un facteur 1 500 par rapport aux calculs sans nimbers et un facteur 180 par rapport aux calculs d'AJS (à ceci près qu'une partie du gain par rapport aux calculs d'AJS vient aussi de la meilleure représentation et de la meilleure canonisation des positions).

10.4 Ordre des options

10.4.1 Principe

Lors du parcours en profondeur de l'arbre de jeu, que ce soit avec l'algorithme élémentaire (algorithme 15) ou avec l'algorithme des nimbers (algorithme 16), on est amené à effectuer des choix sur l'ordre dans lequel on calcule les options. Dès qu'une option perdante est trouvée, cela permet de déterminer que la position parente est gagnante, et il n'est donc plus utile d'effectuer de calculs sur les autres options. Le calcul est donc d'autant plus rapide que les options perdantes sont calculées prioritairement. Idéalement, si les options perdantes sont toujours calculées en premier, tout se passe comme si un étage sur deux de l'arbre de jeu était supprimé.

Le Sprouts pose cependant une difficulté : le jeu est impartial, et nous ne connaissons pas de critère fiable permettant de déterminer si telle ou telle option a plus de chances que telle autre d'être perdante. Dans les jeux partisans, où les joueurs peuvent accumuler un avantage, certaines heuristiques apparaissent de façon naturelle : par exemple, aux échecs, une option où la reine du joueur qui a le trait a été capturée a plus de chances d'être perdante qu'une

option où cette reine est encore disponible. Au Sprouts, les joueurs ne peuvent pas accumuler d'avantage de ce type.

Comme le soulignent AJS [4], l'une des stratégies les plus efficaces dans le cas d'un jeu impartial est de chercher à diriger le calcul vers les zones de l'arbre qui semblent plus faciles à calculer que les autres, indépendamment de leur caractère gagnant ou perdant.

L'ordre lexicographique sur les chaînes de caractères représentant le Sprouts est un premier critère efficace. Il joue un double rôle : d'une part, il permet d'orienter les calculs vers une même partie de l'arbre de jeu, ce qui augmente l'efficacité de la table de transpositions, et d'autre part, il permet de trier correctement les options d'une position pour éliminer les doublons qui peuvent apparaître lors de la génération des options.

10.4.2 Evaluation du nombre d'options

Un excellent critère pour évaluer la difficulté d'une position est de prendre en compte le nombre de ses options : très souvent, plus ce nombre est petit, et plus cette position est facile à calculer. Si la position est perdante, il faut calculer l'issue de toutes ses options, donc il vaut mieux qu'elles soient les moins nombreuses possibles. Inversement, si la position est gagnante, il faut trouver une issue perdante, et là encore mieux vaut avoir à la chercher dans un nombre restreint de candidats.

Cependant, calculer le nombre d'options d'une position de Sprouts n'est pas une opération facile : la nature topologique du Sprouts rend impossible le dénombrement exact des options autrement que par leur calcul effectif, qui est très coûteux en temps. Nous avons donc utilisé la même stratégie qu'AJS, à savoir une évaluation seulement approximative de ce nombre d'options. Ce nombre n'a en effet pas besoin d'être exact, puisqu'il s'agit seulement de donner la priorité aux branches qui semblent plus faciles que les autres. Le nombre d'options est estimé de la façon suivante :

* Quand une frontière est reliée à elle-même, le nombre d'options possibles est $(\textit{nombre de sommets})^2 \times (\textit{nombre de partitions})$, où $(\textit{nombre de partitions})$ est le nombre de façons de partitionner les autres frontières en deux ensembles.
* Quand deux frontières F_i et F_j sont reliées entre elles, le nombre d'options possibles est $(\textit{nombre de sommets de } F_i) \times (\textit{nombre de sommets de } F_j)$.

La figure 10.5 montre que cette évaluation du nombre d'options permet un gain énorme sur le nombre de couples perdants stockés en fin de calcul. Sur S_{10}^+, on passe de 666 715 couples perdants en fin de calcul avec l'ordre lexicographique seul à 219 seulement quand on ajoute une priorité aux positions avec un faible nombre d'options.

Cette énorme différence est liée au phénomène décrit dans le paragraphe 10.2.2 : le nombre d'options d'une position de Sprouts est extrêmement variable à cause de la nature topologique du jeu. Des options d'une même position parente peuvent avoir elles-mêmes un nombre d'options très différent, rendant ce critère intéressant pour les trier.

10.4.3 Ordre plus complexe

Nous avons essayé plusieurs autres idées d'ordre. En particulier, l'algorithme 16 à base de nimbers est d'autant plus efficace que les découpages sont nombreux. Pour augmenter l'efficacité de l'algorithme, on peut donner la priorité aux positions qui comportent un nombre élevé de composantes indépendantes (appelées *pays* dans le cadre du Sprouts). Nous avons également essayé des critères impliquant à la fois le nombre de vies d'une position et la valeur de la partie nimber de l'algorithme 16.

Dans notre article [30], nous décrivons l'ordre suivant :

* priorité aux couples avec une valeur minimale de (nombre de vies + nimber).
* priorité aux positions avec un nombre élevé de pays.
* priorité aux positions avec un petit nombre estimé d'options.

∗ ordre lexicographique.

10.4.4 Comparaison des ordres

La figure 10.5 montre le nombre de couples (position, nimber) perdants stockés à la fin de l'algorithme 16 pour les trois ordres que nous venons d'exposer : ordre lexicographique seul, ordre lexicographique précédé d'une priorité à un faible nombre estimé d'options, et enfin l'ordre complexe décrit ci-dessus prenant en compte en plus le nombre de vies et le nombre de pays. Les calculs de S_2^+ à S_{12}^+ ont été réalisés en partant d'une base initiale vide. Les calculs de S_{13}^+ et S_{14}^+ ont ensuite été réalisés en repartant respectivement de la base obtenue avec S_{12}^+ et S_{13}^+. Enfin, les calculs avec l'ordre lexicographique seul ont été limités à S_{10}^+.

L'aspect aplati de l'ordre complexe de S_{12}^+ à S_{14}^+ vient du fait que les issues de S_{13}^+ et S_{14}^+ s'obtiennent presque immédiatement une fois calculée l'issue de S_{12}^+.

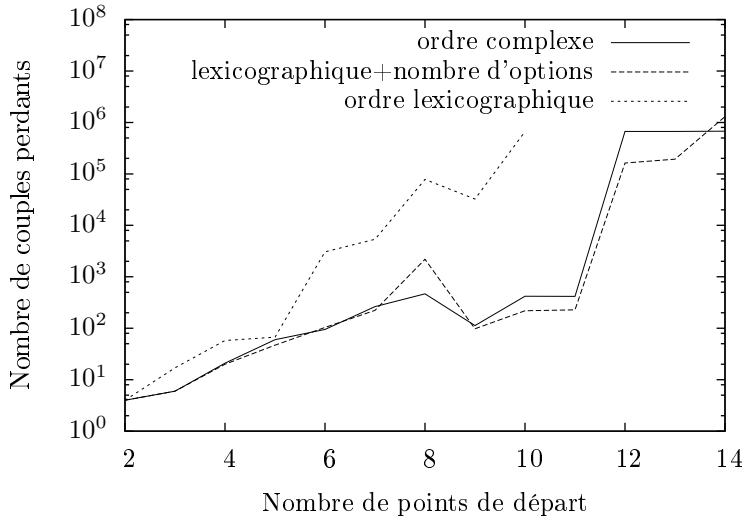


FIGURE 10.5 – Nombre de couples (position, nimber) perdants pour différents ordres possibles en fonction du nombre de points de départ.

Comme on le voit sur le graphe, l'ordre complexe semble moins bon que l'ordre n'utilisant que les deux dernières priorités jusqu'à 13 points de départ, avant de montrer un petit avantage pour 14 points de départ. En fait, autant il est évident que l'estimation du nombre d'options apporte un excellent gain par rapport à l'ordre lexicographique seul, autant il est difficile d'évaluer précisément les autres petites variations imaginables dans l'ordre.

10.5 Interactions manuelles

10.5.1 Suivi

C'est en essayant d'améliorer l'ordre des positions de Sprouts qu'est apparue l'idée du suivi des calculs. Les arbres de jeu de Sprouts sont en fait très déséquilibrés, et étant donné un certain calcul, une variation mineure de l'ordre peut conduire à une accélération ou un ralentissement très marqué sans raison apparente, uniquement à cause d'un choix différent sur une ou deux positions dans le haut de l'arbre. Le cas de S_8^+ sur la figure 10.5 est assez représentatif : l'ordre lexicographique couplé à l'estimation du nombre d'options est soudainement nettement moins bon que l'ordre complexe, alors qu'il est grossièrement équivalent pour toutes les autres positions de départ jusqu'à S_{11}^+.

Dès lors, nous avions besoin d'une méthode de visualisation des calculs pour déterminer plus précisément pourquoi le calcul paraissait si lent pour certains variantes mineures de l'ordre et si rapides pour d'autres. C'est ainsi que nous avons mis en place un système de suivi, détaillé dans le chapitre 5, qui permet d'afficher en temps réel la branche de calcul en cours d'étude dans l'arbre de recherche.

10.5.2 Zappage

La mise en place du suivi des calculs a révélé l'existence de positions bloquantes dans l'arbre de Sprouts : certaines positions, qui pourtant semblent similaires à leurs voisines, se révèlent beaucoup plus difficiles à calculer. La rencontre d'une seule de ces positions bloquantes à un moment donné peut suffire à modifier complètement le temps de calcul. Malheureusement, ces positions bloquantes semblent impossibles à éliminer uniquement par des améliorations de l'ordre. Les quelques règles systématiques de l'ordre sont impuissantes pour détecter et éviter un petit pourcentage de positions pathologiques qui n'obéissent - du moins à notre connaissance - à aucune règle particulière.

Cela a naturellement débouché sur l'idée du *zappage* manuel : plutôt que d'essayer vainement d'affiner un ordre qui finit systématiquement par rencontrer un contre-exemple catastrophique, autant donner la possibilité à l'utilisateur de « zapper » une position trop difficile lorsque celle-ci apparaît. Cette idée d'interaction manuelle ne serait peut-être pas apparue si nous avions étudié d'autres jeux que le Sprouts en premier lieu. De tous les jeux que nous avons essayé de calculer, le Sprouts reste celui dont les arbres de jeu sont les plus déséquilibrés, et donc où le zappage est le plus efficace.

Nous décrivons ce processus de suivi et de zappage des calculs en temps réel dans le chapitre 5. La plus grande valeur que l'algorithme 16 est capable d'atteindre sans zappage est (seulement !) S^+_{14}. Nous avons essayé récemment par curiosité de calculer S^+_{15} avec cet algorithme, sans zappage, mais nous avons finalement interrompu le calcul après plus de 5 millions de positions calculées...

Le zappage s'est révélé très efficace : nous sommes passé de S^+_{14} à S^+_{32}, au prix cependant de nombreuses heures de guidage humain dans l'interface. En particulier, nous avons obtenu pour la première fois S^+_{21} après une véritable traque qui s'est étalée au total sur plusieurs semaines et sur deux ordinateurs.

En fait, la différence d'efficacité avec et sans zappage est tellement forte qu'il ne faut jamais laisser les calculs s'effectuer seuls : une heure de calcul guidé est en général plus efficace que plusieurs jours de calculs en roue libre. Et, de manière contre-intuitive, laisser tourner un calcul en roue libre, bien que cela augmente la quantité d'information disponible, peut s'avérer contre-productif, en augmentant le taille des tables de transpositions, ce qui ralentit les calculs et peut saturer la RAM.

10.6 Version misère

Les algorithmes à base de nimber, combinés au zappage manuel dans les calculs ont permis de dépasser largement, en version normale, le record de 1991 d'AJS et celui de 2006 de Purinton. Il se posait naturellement la question de la version misère, que nous avons commencé à étudier fin 2008.

10.6.1 Algorithme misère

Nous avons détaillé dans le chapitre 4 la théorie des calculs de jeux impartiaux découpables en version misère. Contrairement à la version normale, il est difficile de mener des calculs séparés sur les composantes indépendantes d'une somme. Les algorithmes misère que nous

points de départ	nombre d'ACR	positions stockées
2	5	18
3	7	157
4	35	1 796
5	1 203	24 784
6	25 458	393 103
7	598 685	6 849 627

TABLE 10.1 – Nombres d'ACR dans les arbres de jeu des positions de départ.

avons développés adoptent donc une stratégie différente : dans une première phase, nous étudions « totalement » certaines petites composantes qui reviennent fréquemment dans les calculs, en calculant leur arbre canonique réduit (abrégé « ACR »). Dans une deuxième phase, nous réalisons le calcul classique d'issue sur la position de départ qui nous intéresse, en remplaçant les petites composantes étudiées dans la première phase par leur ACR dès qu'elles apparaissent dans une somme.

L'intérêt de remplacer ces petites composantes par leur ACR vient du fait que l'arbre canonique réduit est en général nettement plus simple que la composante initiale. Les branches redondantes (coups identiques entre eux), ainsi que les coups réductibles (coups inutiles car l'adversaire possède une réponse qui annule le coup) ont été éliminés de l'arbre de jeu. Manipuler cet arbre canonique réduit au lieu de l'arbre de jeu normal de la composante permet de diminuer le nombre de positions rencontrées et ainsi d'accélérer les calculs.

10.6.2 Phase de précalcul

Nous avons besoin dans cette phase de précalcul de déterminer les ACR d'un ensemble bien choisi de positions de Sprouts. Il faut que ces positions interviennent fréquemment dans les positions de Sprouts de plus grande taille. Une possibilité est tout simplement de choisir l'ensemble des positions de Sprouts qui apparaissent à partir d'un certain nombre de points de départ. Ce choix a aussi l'avantage d'être calculable en une seule fois : il suffit de calculer l'ACR de la position de départ en question, car cela implique le calcul des ACR de toutes les positions apparaissant dans son arbre de jeu.

Le tableau 10.1 récapitule le nombre d'ACR différents qui interviennent dans les arbres de jeu de certaines positions de départ de Sprouts. S_6 est calculable avec seulement 45 Mo de RAM, mais S_7 est à la limite des capacités de nos ordinateurs avec 1,4 Go de RAM.

L'ensemble des positions de Sprouts issues de S_6 est en fait le candidat idéal : le nombre de positions (393 103) n'est pas trop élevé, la surcharge de RAM est de 45 Mo seulement, et les positions apparaissent fréquemment dans les sommes de positions de plus grande taille.

Le choix de S_7 n'était pas raisonnable à l'époque de nos calculs, du fait de la surcharge trop forte de RAM par rapport à la capacité mémoire dont nous disposions, mais son utilisation est désormais envisageable. Rien n'interdirait également d'utiliser un ensemble intermédiaire, par exemple l'ensemble des ACR de S_6 couplé à l'ensemble des ACR de l'une des options de S_7 seulement.

Nous avons remarqué que beaucoup de positions presque terminales sont indistinguables de colonnes de Nim, mais que certaines positions assez complexes le sont également. On peut citer notamment `0*4.2`$\sim \mathbb{0}$. Cela montre que les simplifications liées aux colonnes de Nim dans la théorie des ACR sont loin d'être négligeables.

Inversement, la position avec un nombre minimal de vies qui n'est pas une colonne de Nim est `ABC|ABD|CE|DE`$\sim \{2\}$.

10.6.3 Résolution forte

On remarquera que le calcul de l'arbre canonique (réduit ou non) d'une position donnée est équivalent à une résolution forte de cette position, puisque cela nécessite de calculer la totalité des positions apparaissant dans son arbre de jeu. Nos premiers calculs de résolution forte sur le Sprouts n'étaient donc initialement qu'une pré-étape de calcul pour la version misère.

Ce n'est qu'ensuite que nous ne nous sommes rendus compte de la richesse des informations que l'on pouvait obtenir à partir des arbres canoniques (réduits ou non) indépendamment de toute référence au jeu misère. En particulier, cela débouche sur une notion de complexité spatiale exacte, dont on peut déduire une évaluation de la qualité de la canonisation des positions (voir §9.4.4). L'étude détaillée des bases d'arbres canoniques obtenues peut également permettre de découvrir de nouvelles équivalences de positions (voir à ce sujet la section 9.6).

Nous avons donc résolu fortement le jeu de Sprouts jusqu'à S_6 au début de l'année 2009, lors de l'étude du jeu misère. Puis nous avons résolu fortement S_7 en janvier 2011 grâce à une nouvelle machine de calcul possédant plusieurs Go de RAM. Nous évaluons à environ 25 Go la RAM nécessaire pour résoudre fortement S_8, ce qui serait d'ores et déjà possible sur une station de travail. L'utilisation d'une méthode de compression pourrait rendre ce calcul rapidement possible avec nos ordinateurs de bureau.

10.6.4 Premier calcul de S_{17}^-

Une fois l'ensemble des ACR de S_6 calculés, nous avons alors utilisé les algorithmes du chapitre 4 pour effectuer des calculs d'issue en version misère de positions de départ avec un nombre de points de départ plus élevé. En cours de calcul, nous remplaçons par leur ACR les composantes des sommes qui se révèlent être une position apparaissant dans l'arbre de jeu de S_6. Cela nous a permis de calculer en 2009 que S_{17}^- est gagnante. Ce premier calcul de S_{17}^- a nécessité le stockage de 170 000 nœuds environ. Parmi eux, la moitié environ avaient une partie position vide, ce qui montre que l'on peut explorer l'arbre de jeu de sorte qu'assez rapidement, la partie position soit démantelée en plusieurs positions indépendantes assez petites pour que leur ACR soit dans l'arbre de jeu de S_6.

Toutes les techniques décrites sur la version normale (ordre des options, table de transpositions, et surtout suivi du calcul et zappage), à l'exception bien sûr de la théorie du nimber, ont été réutilisées sur la version misère.

Le temps de calcul lors de ce premier record en 2009 était d'une vingtaine d'heures seulement sur un processeur à 1,8 GHz, et la consommation de RAM inférieure à 100 Mo, ce qui permettait d'espérer des améliorations supplémentaires.

10.7 Proof-number search

La dernière évolution des calculs de Sprouts, à partir de 2010, vient de la recherche de meilleurs parcours de l'arbre de jeu. Comme expliqué dans le chapitre 5, le zappage manuel est en fait assez proche, par bien des aspects, des algorithmes de type best-first, comme le PN-search. Ces algorithmes explorent l'arbre de jeu d'une manière radicalement différente du depth-first (alpha-bêta), en favorisant les calculs vers la branche de jeu qui semble la plus facile. Cette branche est réévaluée entre chaque étape de l'algorithme, ce qui permet d'éviter automatiquement les branches qui se révèlent trop difficiles. Ce comportement est proche du zappage, car il permet de détecter les positions bloquantes.

L'implémentation du PN-search et de variantes nous a permis de calculer sans interaction manuelle les positions jusqu'à S_{21}^+, à comparer avec S_{14}^+ pour l'alpha-bêta seul. Mais l'algorithme de PN-search n'a pas réussi à égaliser seul les résultats de Sprouts obtenus grâce au

zappage. La raison tient à un développement trop en largeur de l'arbre de recherche, ce qui sature rapidement la RAM sur certaines positions de Sprouts qui possèdent parfois plusieurs centaines d'options différentes.

Nous avons donc naturellement appliqué au PN-search les deux techniques qui s'étaient montrées si efficaces sur l'alpha-bêta : le suivi et les interactions manuelles. Cela a été fructueux : non seulement nous avons pu retrouver plus rapidement nos résultats de 2007 jusqu'à S^+_{32}, mais nous avons pu pousser les calculs plus loin, jusqu'à S^+_{44}, plus diverses valeurs éparses, la plus grande étant S^+_{53}. En version misère, cela a permis de calculer les issues de S^-_{18} à S^-_{20}.

Le suivi et les interactions dans le PN-search sont décrits dans le chapitre 5, et les variantes du PN-search que nous avons implémentées sont décrites dans le chapitre 6.

10.8 Vérification

10.8.1 Principe

L'idée de la vérification est en grande partie une conséquence de l'introduction des zappages manuels. Ces zappages sont certes très efficaces pour accélérer les calculs de Sprouts, mais ils ont un inconvénient majeur : ils ne sont pas reproductibles, car l'utilisateur humain ne clique jamais exactement au même moment d'un calcul à l'autre.

Nous avons donc introduit la notion de calcul de vérification, pour contrôler, sans intervention manuelle, que le résultat du calcul est bien correct. La vérification effectue simplement le calcul récursif de l'issue, en se basant sur les résultats des premiers calculs pour se guider dans l'arbre de jeu. Contrairement au calcul initial avec zappage, le calcul de vérification est reproductible.

Ces calculs de vérification se sont ensuite révélés très utiles pour diminuer la taille des arbres solutions en éliminant les branches redondantes ou inutiles. La vérification est traitée en détail dans le chapitre 7.

10.8.2 Records d'arbres solutions en version normale

La table 10.2 indique le nombre de couples (position, nimber) perdants du plus petit arbre solution que l'on connaît actuellement pour les différentes positions de départ en version normale.

p	taille
1	1
2	3
3	6
4	15
5	15
6	46
7	76
8	139
9	60
10	110

p	taille
11	113
12	316
13	369
14	1 017
15	1 986
16	669
17	329
18	1 997
19	1 736
20	1 831

p	taille
21	5 312
22	1 581
23	1 058
24	5 327
25	2 497
26	4 458
27	12 768
28	2 549
29	2 172
30	12 800

p	taille
31	5 463
32	58 204
33	62 389
34	21 107
35	4 265
36	80 001
37	80 009
38	80 281
39	98 905
40	45 782

p	taille
41	42 663
42	98 947
43	98 961
44	99 095
45	?
46	80 473
47	54 542
...	?
53	73 225
...	?

TABLE 10.2 – Plus petits arbres solutions connus en version normale.

La figure 10.6 reprend le contenu de la table 10.2 jusqu'à S^+_{45}, première valeur inconnue. Pour comparaison, nous avons également reporté sur le graphe le nombre de positions perdantes de la base en fin de calcul d'AJS, de 2 à 11 points de départ.

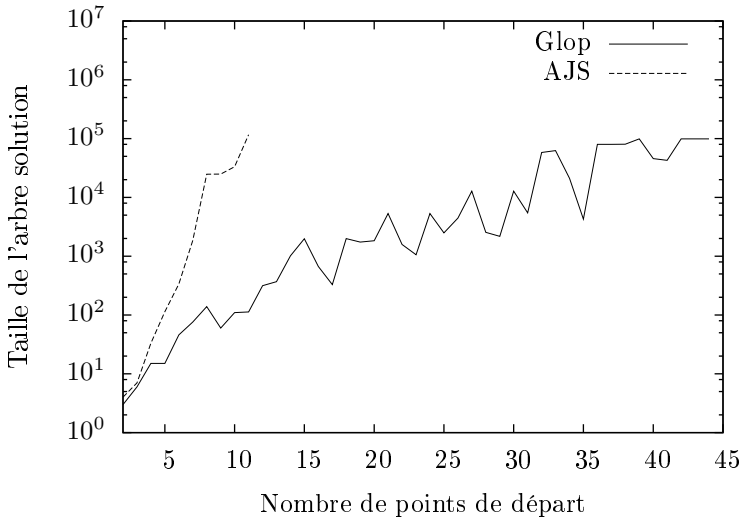


FIGURE 10.6 – Nombre de positions de l'arbre solution en fonction du nombre de points, en version normale (échelle logarithmique).

On remarquera l'échelle logarithmique, qui montre que dans l'ensemble, l'augmentation de la taille de l'arbre solution est exponentielle par rapport au nombre de points de départ. Cependant, bien qu'exponentielle, la croissance est très lente. Le plus petit arbre solution connu pour S_{44}^+ demande moins de couples (position, nimber) perdants que la base de fin de calcul d'AJS lors de la première obtention de S_{11}^+. C'est très surprenant. Cela signifie que même des positions de Sprouts qui paraissaient totalement inatteignables quelques années auparavant admettent en réalité un arbre solution de taille réduite, et qu'il est possible de les calculer en un temps raisonnable.

10.8.3 Records d'arbres solutions en version misère

Le tableau 10.3 récapitule le nombre de positions du plus petit arbre solution que nous connaissons actuellement pour démontrer S_p^- (après utilisation de l'algorithme de vérification). Les nombres indiqués correspondent aux nombres de positions dont nous avons besoin *en plus* des 393 103 positions de l'arbre de jeu de S_6 pour calculer l'issue de S_p^-. Bien sûr, on ne peut pas démontrer l'issue de S_7^- avec seulement 7 positions.

p	7	8	9	10	11	12	13
Positions	7	21	42	72	78	591	272

p	14	15	16	17	18	19	20
Positions	548	3 281	3 200	8 535	55 532	29 801	73 258

TABLE 10.3 – Nombre de positions nécessaires pour démontrer S_p^-.

Nous avons reporté le tableau 10.3 sur la figure 10.7.

En version misère, nous n'avons pu atteindre que S_{20}^-, à comparer avec S_{44}^+ en version normale. Cette différence vient entièrement des difficultés théoriques de la version misère, qui ne permet pas de simplifier le calcul des sommes de positions aussi bien qu'en version normale. Pour un même nombre de points de départ, il faut donc explorer beaucoup plus de nœuds en version misère qu'en version normale. Nous avons également remarqué qu'en version misère, la difficulté du calcul semble augmenter plus régulièrement avec le nombre

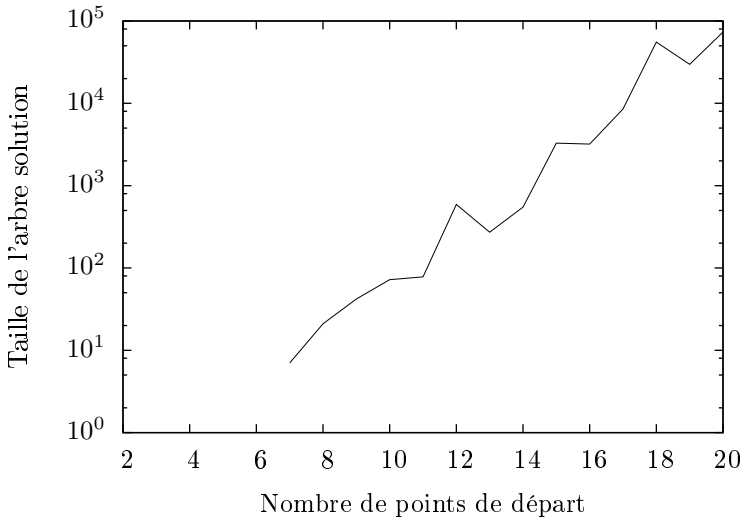


FIGURE 10.7 – Nombre de positions de l'arbre solution en fonction du nombre de points, en version misère (échelle logarithmique).

de points de départ qu'en version normale, (il est par exemple bien plus rapide de calculer S^+_{17} que S^+_{15} en version normale). Il est probable que cet écart entre la version misère et la version normale continue de se creuser dans les années à venir.

Enfin, comme en version normale, on peut observer une régularité de période 6. Ainsi, de même que S^-_{12} est plus difficile à calculer que S^-_{13}, on observe que S^-_{18} est plus difficile à calculer que S^-_{19}.

10.8.4 Solution du jeu à 5 points

Nous présentons en annexe B la solution détaillée du jeu à 5 points. C'est un exemple de cas où la programmation vient au secours du joueur humain, en produisant une preuve très simple : il suffit au premier joueur de retenir une quinzaine de positions perdantes pour s'assurer de la victoire. Nous expliquons dans l'annexe quelles sont les méthodes qui ont permis d'aboutir à une preuve aussi compacte, la vérification en faisant partie.

10.9 Les conjectures du Sprouts

10.9.1 Conjecture normale forte

Toutes les valeurs calculées en version normale sont conformes à la conjecture d'AJS en 1991. Nous avons également calculé les nimbers jusqu'à S^+_{32}. Les positions de départ perdantes ne nécessitent pas de calcul supplémentaire car elles sont de nimber nul, quant aux positions de départ gagnantes, elles se sont toutes révélées de nimber $\mathbb{1}$. Cela conduit à formuler une conjecture plus forte que celle d'AJS, sur les valeurs des nimbers des positions de départ du Sprouts :

Conjecture 3. *Le nimber de S^+_p vaut $\mathbb{1}$ si p est égal à 3, 4 ou 5 modulo 6 et $\mathbb{0}$ sinon.*

Cette conjecture contient strictement celle d'AJS, et exprime que la suite des nimbers en fonction du nombre de points de départ est périodique, de période 6.

10.9.2 Nouvelle conjecture misère

Le tableau 10.4 montre les issues gagnantes (W) ou perdantes (L) calculées jusqu'à S_{19}^-. AJS n'avaient calculé les valeurs que jusqu'à S_9^-, ce qui les avait amenés à conjecturer une suite périodique de période 5. L'issue de S_{12}^- a été calculée gagnante par Josh Purinton et Roman Khorkov en 2006, invalidant la conjecture.

p	1	2	3	4	5	6	7	8	9	10
S_p^-	W	L	L	L	W	W	L	L	L	W

p	11	12	13	14	15	16	17	18	19	20
S_p^-	W	W	L	L	L	W	W	W	L	L

TABLE 10.4 – Issues gagnantes/perdantes calculées de S_p^-.

Nous disposons maintenant de suffisamment de valeurs pour formuler une nouvelle conjecture. Les valeurs pour 1 et 4 points de départ ressemblent à des exceptions des petites valeurs, avant la stabilisation de l'aspect périodique. Comme dans le cas du jeu normal, on retrouve une période de longueur 6, mais décalée, et l'on peut donc conjecturer :

Conjecture 4. *Le premier joueur a une stratégie gagnante sur S_p^- si et seulement si p est égal à 0, 4 ou 5 modulo 6 – sauf si $p = 1$ ou 4.*

10.9.3 Autres phénomènes de périodicité

Au cours des calculs, nous avons remarqué de nombreux aspects périodiques ou quasi-périodiques du Sprouts, ne concernant pas uniquement les positions de départ.

Par exemple, considérons les positions dont la représentation en chaîne est du type `222...2`. En commençant par `22`, puis `222`, `2222`... le début de la suite des nimbers de ces positions est :

$$\mathbb{1}\,;\mathbb{0}\,;2\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}\,;5\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}\,;\mathbb{3}\,;\mathbb{1}\,;\mathbb{0}$$

De manière plus compacte, on peut énoncer :

Conjecture 5. *Le nimber de la position* `222...2` *contenant n fois « 2 » est :*
- $\mathbb{0}$ *si* $n \equiv 0 \mod 3$
- $\mathbb{3}$ *si* $n \equiv 0 \mod 3$
- $\mathbb{1}$ *si* $n \equiv 0 \mod 3$

sauf si $n = 4$ ou 10.

Autre exemple, celui des positions du type `0*m.A|0*n.A`, qui interviennent fréquemment en haut des arbres de jeu des positions de départ du Sprouts (de telles positions apparaissent après seulement deux coups joués). La conjecture peut cette fois s'énoncer ainsi :

Conjecture 6. *Le nimber de la position* `0*m.A|0*n.A` *est $\mathbb{0}$ si et seulement si l'on est dans une des situations suivantes :*
- $m + n \equiv 0$ *ou* $5 \mod 6$
- $m + n \equiv 4 \mod 6$ *et* $m \not\equiv 2$ *ou* $5 \mod 6$
- $m + n \equiv 1 \mod 6$ *et* $m \equiv 2$ *ou* $5 \mod 6$
- $m = n = 2$

Dans le cas contraire, son nimber est $\mathbb{1}$.

Nos calculs ont montré que toutes les positions de ce type avec $n + m \leq 17$ vérifient cette conjecture, ainsi que plusieurs dizaines d'autres positions avec $n + m > 17$. On a donc,

ici encore, une périodicité modulo 6 qui s'installe, le seul contre-exemple étant la position `0*2.A|0*2.A` qui est de nimber $\mathbb{1}$ là où $\mathbb{0}$ était attendu.

De tels phénomènes de quasi-périodicité ont été observés sur de nombreux types de positions, la périodicité est cependant parfois plus longue à s'installer. Ce comportement est à rapprocher de celui des jeux octaux, à ceci près que dans le cas du Sprouts, les différentes périodicités observées ressemblent plus à des îlots de régularité perdus dans le désordre général : au contraire des jeux octaux, il semble difficile[1] d'imaginer une preuve de toutes les positions de départ du Sprouts qui repose sur un catalogue de positions pour lesquelles on prouve que la périodicité s'installe.

Ces aspects quasi-périodiques du Sprouts permettent sans doute d'expliquer l'excellent niveau de certains joueurs humains. En particulier, nous avons joué plusieurs parties en version normale contre Roman Khorkov en 2007. Dans chaque partie où il était initialement en position de force, il a joué parfaitement. Il est probable qu'il connaisse un grand nombre de ces phénomènes périodiques et qu'il les utilise pour jouer.

10.9.4 Conclusion

La question principale que l'on peut se poser est de savoir s'il n'existerait pas un argument simple permettant de démontrer les conjectures du Sprouts pour un nombre quelconque de points de départ. La très petite taille des arbres solutions comparativement à la taille de l'arbre de jeu, en particulier en version normale, penche partiellement en faveur de l'existence d'une stratégie simple, que l'on pourrait peut-être écrire de façon exacte pour un nombre quelconque de points de départ.

Inversement, nos calculs ont régulièrement montré des aspects que l'on pourrait qualifier de chaotiques. Il n'est pas exclu que l'on puisse éviter totalement ces parties chaotiques de l'arbre de jeu, si l'on disposait d'une cartographie de l'aspect régulier ou chaotique des différentes zones de l'arbre de jeu. Mais dans tous les cas, si une stratégie simple existe, elle fait certainement intervenir une analyse détaillée d'un très grand nombre de types de positions différentes.

Une autre question est de savoir jusqu'où l'on pourra pousser les calculs à l'avenir. Dans le cas du Sprouts, la réponse est peut-être hasardeuse. Jusque dans les années 2000, les connaisseurs du Sprouts - dont Conway - pensaient que la position de départ à 12 points était très difficile. Et pourtant un arbre solution de 316 positions perdantes seulement existe. Personne n'aurait imaginé non plus que la résolution de la position à 53 points serait possible en 2011, et encore moins que cette solution serait vérifiable sur l'ordinateur utilisé par AJS en 1991 !

1. Mais pas insurmontable ?

Chapitre 11

Le Sprouts sur les surfaces compactes

Ce chapitre présente une généralisation du jeu de Sprouts, lorsqu'il est joué sur d'autres surfaces que le plan. Sa bonne compréhension nécessite la lecture préalable d'autres chapitres de cette thèse.

Concernant l'étude des jeux combinatoires impartiaux, le chapitre 3 est relatif à l'étude des jeux en version normale et présente la notion importante de *nimber*, et le chapitre 4 présente des techniques permettant d'étudier les jeux impartiaux en version misère.

Concernant le jeu de Sprouts, le chapitre 9 permet de se familiariser avec la représentation des positions de ce jeu sous une forme utilisable par un programme informatique, et le chapitre 10 détaille l'étude du jeu classique sur le plan.

11.1 Une généralisation naturelle

Nous avons vu dans le chapitre 10 que l'étude du jeu de Sprouts fait émerger une périodicité qui rappelle celle des jeux octaux. En effet, une fois que le nombre p de points de départ dépasse la quinzaine, l'étude du jeu à $p+6$ points ressemble à celle du jeu à p points. Les calculs menés n'ont révélé que peu d'irrégularités relativement à cette observation. Aussi, il est tentant de chercher à altérer les règles, de façon à ramener un peu de chaos dans l'étude du jeu, chaos qui pourrait augmenter son intérêt.

Dans ce chapitre, nous développons ainsi l'étude du jeu de Sprouts sur les surfaces compactes. Conformément aux objectifs que nous venons d'énoncer, le jeu sur les surfaces nous réservera de nombreuses surprises. Mais avant de commencer son étude, nous allons expliquer en quoi cette généralisation du jeu classique est légitime.

Le jeu de Sprouts est un jeu topologique : seule la forme des positions considérées est importante, les longueurs ne le sont pas. Si l'on désire conserver le principe qui consiste à relier deux points par une ligne, et à en rajouter un sur la ligne, alors il ne reste que peu de latitude pour modifier les règles du jeu. En effet, la variation qui consiste à décréter que celui qui joue le dernier coup a perdu, la version misère, a déjà été étudiée dans le chapitre 4. Autoriser le croisement des lignes ne mènerait pas à grand chose, il serait toujours possible de relier deux points quelconques, du moment que leur degré n'excède par 2. Ainsi, chaque partie à p points de départ se terminerait en $3p-1$ coups, le nombre maximal, et le gagnant serait le même quelle que soit la partie jouée.

Il ne reste guère que le terrain de jeu que l'on puisse modifier. Un terrain de jeu tridimensionnel ne serait pas non plus très motivant. Là encore, deux points quelconques pourraient toujours être reliés en slalomant entre les lignes existantes, et la partie se terminerait tou-

jours en $3p-1$ coups. Un terrain de jeu unidimensionnel ne laissant aucune place au jeu pour se développer, il est nécessaire de considérer un terrain à deux dimensions, c'est-à-dire une surface.

C'est ainsi assez naturellement que l'on est amené à considérer le jeu sur d'autres surfaces que le plan. Avant de voir quelles surfaces pourraient modifier le déroulement du jeu de manière acceptable, rappelons quelques notions relatives à l'étude du jeu classique, qui se déroule sur le plan. Certaines de ces notions ont été détaillées dans le chapitre 9.

On appelle *position* une partie de Sprouts à un instant donné : c'est un graphe $\mathscr{G}$, composé de sommets et d'arêtes, plongé dans le plan $\mathscr{P}$. Une *région* $\mathscr{R}$ est une composante connexe de $\mathscr{P}-\mathscr{G}$, c'est donc un ouvert du plan. On note $\overline{\mathscr{R}}$ l'adhérence de $\mathscr{R}$, et l'on appelle *frontières* de la région les différentes composantes connexes de la frontière (au sens topologique) de $\mathscr{R}$, à savoir $\overline{\mathscr{R}}-\mathscr{R}$. Concrètement, les frontières de $\mathscr{R}$ sont les éléments de $\mathscr{G}$ en contact avec la région.

Il est assez intuitif qu'une « petite » déformation d'une position, c'est-à-dire, qui ne va pas changer la forme globale de la position (et ne pas induire de croisement entre les arêtes, par exemple), n'altère pas les parties jouables à partir de cette position. Cette idée peut être formalisée par la notion d'*homéomorphisme*. Un homéomorphisme est une bijection continue, de réciproque continue. Toute déformation acceptable d'une position correspond en fait à un homéomorphisme du plan sur lui-même. Et si deux positions sont homéomorphes, alors ces deux positions sont *équivalentes* au sens du paragraphe 2.5.3 : elles ont le même arbre canonique.

Intéressons-nous maintenant aux régions. Une région intérieure à f frontières est homéomorphe à un disque, borné par l'unique frontière qui entoure la région, auquel on rajoute $f-1$ *bords*, les autres frontières. Or, un disque est homéomorphe à une sphère à un bord. On en déduit qu'une région intérieure à f frontières est homéomorphe à une sphère à f bords. L'unique région extérieure, si elle admet f' frontières, est homéomorphe à une sphère à $f'+1$ bords, le bord supplémentaire correspondant au pôle créé lorsque l'on projette cette région sur une sphère avec la projection stéréographique. Toute région est ainsi homéomorphe à une sphère à bords. En particulier, la position de départ à p points sur le plan est homéomorphe à une sphère à $p+1$ bords. Rappelons enfin que rajouter des bords, par exemple avec des frontières mortes (avec lesquelles on ne peut plus jouer) n'influence pas le déroulement du jeu.

Maintenant, imaginons que le jeu commence non pas sur le plan, ou de manière équivalente, sur la sphère, mais sur une autre surface compacte, par exemple, le tore. Cela va t-il changer quelque chose ? Les figures 11.1 et 11.2 vont nous montrer que oui.

Sur la figure 11.1, nous avons commencé à développer l'arbre de jeu d'une position de Sprouts sur le plan. Pour chaque fils de la racine, nous avons indiqué en pointillés le coup que le deuxième joueur va répondre. En poussant un peu l'étude, on se rend compte que le deuxième joueur a une stratégie gagnante. La position de départ a 6 vies, le deuxième joueur fait en sorte que la partie se termine avec 2 points isolés, et donc, la partie se termine en $6-2=4$ coups.

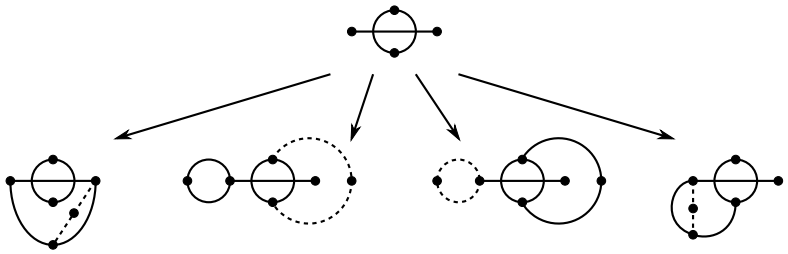


FIGURE 11.1 – Arbre de jeu d'une position de Sprouts.

Si l'on étudie la même position sur le tore, on se rend compte que cette fois-ci, c'est le premier joueur qui dispose d'une stratégie gagnante. Son premier coup, représenté sur la figure 11.2, consiste à faire le tour du tore. Une étude détaillée montrerait qu'il peut ensuite forcer la partie à se terminer avec un seul point isolé, donc en $6 - 1 = 5$ coups, ce qui lui assure la victoire.

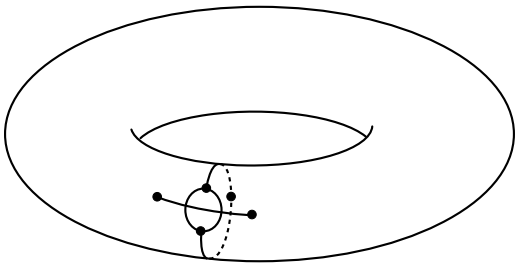


FIGURE 11.2 – Position de Sprouts sur le tore.

Jouer sur le tore produit des arbres de jeu qui contiennent ceux sur le plan, c'est-à-dire que cela conduit à rajouter des coups, sans en enlever. En effet, toute partie jouée sur le plan peut être également jouée sur le tore, en restreignant l'aire de jeu à un disque ouvert prélevé sur le tore. Dans l'exemple que nous venons d'étudier, le fait de jouer sur le tore a rajouté des options à la position de départ, et une de ces options permettait au premier joueur de gagner, ce qui n'était pas le cas sur le plan.

On peut imaginer commencer le jeu sur un tore, mais aussi sur une autre surface orientable (tore à n trous), ou même sur des surfaces non orientables (ruban de Möbius, bouteille de Klein...). Cette généralisation est suffisamment naturelle pour qu'elle soit apparue dès les premiers jours de l'histoire du jeu. Dans son article de 1967 qui suivit de quelques mois l'invention du jeu [19], Martin Gardner cite John Conway : « The day after sprouts sprouted, it seemed that everyone was playing it. At coffee or tea times there were little groups of people peering over ridiculous to fantastic sprouts positions. Some people were already attacking sprouts on toruses, Klein bottles and the like, while at least one man was thinking of higher-dimensional versions. »

Dans la suite de ce chapitre, nous allons commencer par énoncer à la section 11.2 quelques notions élémentaires sur les surfaces compactes qui seront utiles à notre exposé. Ensuite, dans la section 11.3, nous étudierons en détail comment une surface autre que le plan ou la sphère affecte les coups disponibles. Nous verrons que la principale difficulté réside dans les coups qui relient une frontière à elle-même, modifiant de façon intéressante le type de surface sur laquelle le jeu évolue. Dans la section 11.4, nous détaillerons les différences entre l'implémentation sur le plan et celle sur les surfaces, qui se produisent exclusivement dans le calcul des options. Puis nous expliquerons la notion théorique de *genre limite* dans la section 11.5 : cette notion traduit le fait qu'il est inutile, si l'on fixe les frontières d'une région, d'associer une surface trop complexe à cette région. Enfin, nous récapitulerons les résultats obtenus à l'aide de notre programme dans la section 11.6.

11.2 Notions de base sur les surfaces compactes

11.2.1 Surfaces orientables

On notera $\mathbb{S}$ la sphère, $\mathbb{T}$ le tore. Étant données deux surfaces $\mathscr{S}_1$ et $\mathscr{S}_2$, on obtient leur somme connexe $\mathscr{S}_1 \sharp \mathscr{S}_2$ en supprimant un disque de chacune des surfaces, puis en recollant les bords d'un cylindre sur les deux bords ainsi créés. Remarquons que quelle que soit la

surface $\mathscr{S}$, on a $\mathscr{S}\sharp\mathbb{S} = \mathscr{S}$, c'est-à-dire que la sphère est un élément neutre pour la somme connexe.

La figure 11.3 montre que la somme connexe de deux tores donne un *tore à deux trous*. On notera cette surface $\mathbb{T}^2$. Plus généralement, la somme connexe de n tores sera notée $\mathbb{T}^n$, pour $n \geq 1$. Pour simplifier l'énoncé de certains résultats, on s'autorisera à écrire $\mathbb{T}^0 = \mathbb{S}$.

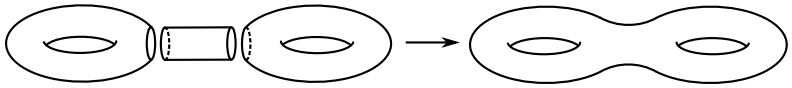


FIGURE 11.3 – Somme connexe de deux tores.

11.2.2 Surfaces non orientables

La surface non orientable la plus célèbre est le *ruban de Möbius*. Elle s'obtient en recollant deux côtés opposés d'un rectangle, après lui avoir fait subir une torsion d'un demi-tour. Cette surface n'est cependant pas une surface compacte, puisqu'elle a un bord (ce bord est unique, ce qui est surprenant quand on compare le ruban au cylindre qui, lui, en a deux).

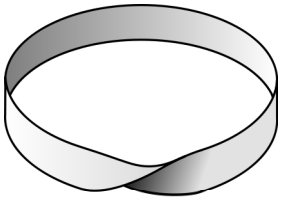


FIGURE 11.4 – Le ruban de Möbius.

On obtient une surface compacte à partir du ruban de Möbius en collant un disque le long de son bord. La surface ainsi obtenue est appelée *plan projectif réel*, et sera notée $\mathbb{P}$. Cette surface ne peut pas être plongée dans $\mathbb{R}^3$, mais seulement dans $\mathbb{R}^4$, et comme beaucoup d'objets de ce type, elle est difficile à représenter.

Nous verrons au paragraphe 11.2.5 une astuce permettant de jouer sur cette surface avec une feuille de papier. En attendant, on peut tout de même jouer sur cette surface si l'on se rappelle que rajouter un bord ne change pas le déroulement du jeu. Ainsi, il suffit de fabriquer un ruban de Möbius en scotchant du papier transparent, et de jouer la partie au feutre sur ce ruban, pour attaquer l'étude du Sprouts sur le plan projectif.

La somme connexe de deux plans projectifs sera notée $\mathbb{P}^2$, et, plus généralement, la somme connexe de n plans projectifs sera notée $\mathbb{P}^n$, pour $n \geq 1$. La surface $\mathbb{P}^2$ porte un nom particulier, il s'agit de la *bouteille de Klein*. Comme toute surface compacte non orientable, elle ne peut être plongée dans $\mathbb{R}^3$, mais cette fois, une de ses représentations est relativement facile à appréhender. La figure 11.5 montre cette représentation dans un contexte qui explique pourquoi cette surface porte le nom de *bouteille*.

11.2.3 Classification des surfaces compactes

On rappelle que toute surface compacte connexe sans bord est homéomorphe soit à $\mathbb{T}^n$ (pour un $n \geq 0$) si elle est orientable, soit à $\mathbb{P}^n$ (pour un $n \geq 1$) si elle n'est pas orientable [18]. De plus, la somme connexe de deux surfaces compactes s'effectue ainsi :

$$\begin{array}{lcll} \mathbb{T}^m\sharp\mathbb{T}^n & = & \mathbb{T}^{m+n} & (m,n \geq 0) \\ \mathbb{P}^m\sharp\mathbb{P}^n & = & \mathbb{P}^{m+n} & (m,n \geq 1) \\ \mathbb{P}^m\sharp\mathbb{T}^n & = & \mathbb{P}^{m+2n} & (m \geq 1,\ n \geq 0) \end{array}$$

FIGURE 11.5 – La bouteille de Klein.

11.2.4 Caractéristique d'Euler

On notera $\chi(\mathscr{S})$ la caractéristique d'Euler de la surface $\mathscr{S}$. On rappelle la caractéristique d'Euler des surfaces compactes :

$$\begin{array}{lcll} \chi(\mathbb{T}^n) & = & 2-2n & (n \geq 0) \\ \chi(\mathbb{P}^n) & = & 2-n & (n \geq 1) \end{array}$$

Le nombre n est appelé le *genre* de la surface.

Si la surface $\mathscr{S}$ n'est pas connexe, sa caractéristique d'Euler est la somme des caractéristiques d'Euler de ses composantes connexes. Si une surface $\mathscr{S}$ est homéomorphe à une surface compacte $\mathscr{S}_C$ à b bords, alors $\chi(\mathscr{S}) = \chi(\mathscr{S}_C) - b$. Par exemple, une surface $\mathscr{S}$ ayant deux composantes connexes, l'une homéomorphe à un tore à 2 trous, l'autre à une sphère avec 3 bords, a une caractéristique d'Euler valant : $\chi(\mathscr{S}) = (-2) + (2-3) = -3$.

« Souder » une surface selon un lacet, ou inversement, « découper » une surface selon un lacet ne change pas la caractéristique d'Euler de la surface. Par exemple, quand on réalise la somme connexe de deux surfaces $\mathscr{S}_1$ et $\mathscr{S}_2$, on rajoute un bord à $\mathscr{S}_1$ et un bord à $\mathscr{S}_2$, puis on soude selon ces bords. Ainsi, $\chi(\mathscr{S}_1 \sharp \mathscr{S}_2) = (\chi(\mathscr{S}_1) - 1) + (\chi(\mathscr{S}_2) - 1)$.

11.2.5 Représentation plane des surfaces compactes

S'il est facile d'utiliser une feuille de papier pour jouer au Sprouts sur le plan, il ne paraît pas très confortable d'utiliser une bouée et un feutre pour jouer au Sprouts sur le tore... Un outil mathématique vient à notre secours : le schéma. On a représenté dans la figure 11.6 le schéma d'un plan projectif et celui d'un tore. Sur chaque schéma, on identifie les côtés marqués par une même lettre, la flèche nous renseignant sur le sens de l'identification. On a représenté à chaque fois une unique ligne en pointillés sur le schéma, de façon à se rendre compte de la façon dont il fonctionne.

Si l'on découpait une feuille de papier ayant la forme d'un de ces schémas, et qu'on la recollait en respectant l'orientation, on retrouverait la surface associée (en gardant à l'esprit qu'une telle réalisation n'est pas possible avec seulement 3 dimensions dans le cas des surfaces non orientables).

On peut représenter un schéma par une suite de caractères, en partant d'un sommet, puis en énonçant les arêtes lors d'un tour du polygone. Par exemple, le schéma de $\mathbb{P}$ donné en figure 11.6 est aa, et celui de $\mathbb{T}$ est $aba^{-1}b^{-1}$ (partir du sommet en haut à gauche puis tourner dans le sens des aiguilles d'une montre). On inscrit a quand on rencontre l'arête dans le sens de la flèche, et a^{-1} sinon.

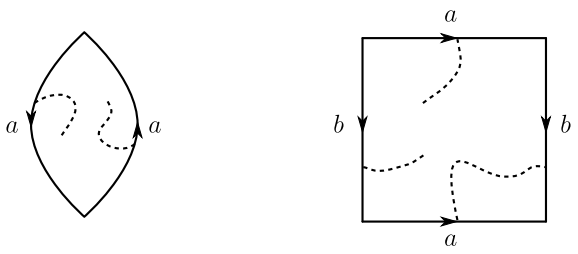


FIGURE 11.6 – Schémas d'un plan projectif et d'un tore.

Ensuite, pour obtenir les schémas d'autres surfaces compactes, on peut utiliser le fait que le schéma de la somme connexe de deux surfaces est la mise bout à bout des schémas de ces deux surfaces. Par exemple, un schéma de $\mathbb{P}^2$ est le carré $aabb$, un schéma de $\mathbb{T}^2$ est l'octogone $aba^{-1}b^{-1}cdc^{-1}d^{-1}$.

Il faut noter qu'il n'y a pas unicité des schémas. Par exemple, $aabb$ est un schéma de la bouteille de Klein $\mathbb{P}^2$, mais $abab^{-1}$ est également un schéma correct pour $\mathbb{P}^2$.

11.2.6 Surface associée à une région

Dans une partie de Sprouts sur une surface compacte, toute région est homéomorphe à une surface compacte à bords, le nombre de bords étant égal au nombre de frontières de la région. On appellera *surface associée à une région* la surface compacte homéomorphe à la région privée de ses bords.

Comme le nombre de bords n'influence pas le déroulement du jeu, les seules données topologiques importantes sont les types des surfaces associées aux régions. Dans la section suivante, nous décrivons à quelles surfaces peuvent être associées les nouvelles régions obtenues lorsque l'on joue un coup.

11.3 Types de coups sur les surfaces

Nous allons présenter dans cette section les différents types de coups que l'on peut jouer sur les surfaces compactes. Dans le jeu sur le plan, deux types de coups seulement existent : ou bien on relie une frontière à une autre, ou bien on relie une frontière à elle-même en créant une nouvelle région. Lorsque l'on généralise aux surfaces compactes, beaucoup plus de types de coups sont possibles.

11.3.1 Description des coups

On suppose que l'on joue un coup de Sprouts dans une région $\mathscr{R}$. La surface associée à $\mathscr{R}$ sera notée $\mathscr{S}$.

Coup (type I)**.** *On relie deux frontières.*

Ce coup est une généralisation du coup sur le plan. Il aboutit à la fusion des deux frontières, et la région $\mathscr{R}$ reste homéomorphe à la même surface compacte.

Coup (type II)**.** *On relie une frontière à elle-même.*

En reliant une frontière à elle-même, on crée un lacet $\mathscr{L}$ qui va modifier la structure de la région $\mathscr{R}$. Dans le cas du jeu sur le plan, la région était séparée en deux. Ce n'est pas forcément le cas avec les surfaces compactes, et l'on va détailler les différentes possibilités.

Coup (type II.A)**.** $\mathscr{R} - \mathscr{L}$ *a deux composantes connexes.*

Ce coup correspond à scinder la région $\mathscr{R}$ en deux surfaces compactes dont la somme connexe est homéomorphe à $\mathscr{R}$. En inversant les relations du paragraphe 11.2.3, on obtient :

$$\begin{array}{lcll} \mathbb{T}^n & \to & \mathbb{T}^k + \mathbb{T}^{(n-k)} & (n \geq 0,\ 0 \leq k \leq n) \quad (a) \\ \mathbb{P}^n & \to & \mathbb{P}^k + \mathbb{P}^{(n-k)} & (n \geq 2,\ 0 < k < n) \quad (b) \\ \mathbb{P}^n & \to & \mathbb{T}^k + \mathbb{P}^{(n-2k)} & (n \geq 1,\ 0 \leq k < \frac{n}{2}) \quad (c) \end{array}$$

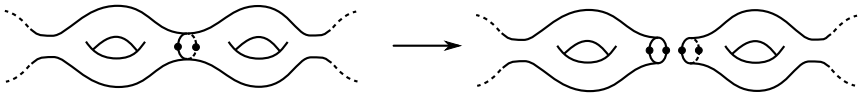


FIGURE 11.7 – Coup de type II.A.(a).

Coup (type II.B)**.** *$\mathscr{R} - \mathscr{L}$ a une seule composante connexe.*

La surface associée à $\mathscr{R} - \mathscr{L}$ sera notée $\mathscr{S}'$. Deux cas émergent :

Coup (type II.B.1)**.** *Couper $\mathscr{R}$ selon $\mathscr{L}$ engendre deux bords.*

On a la relation $\chi(\mathscr{S}) = \chi(\mathscr{S}') - 2$, et comme de plus, découper une région orientable ne change pas l'orientabilité de la région, il suffit de considérer la caractéristique d'Euler pour voir que les seuls cas possibles sont :

$$\begin{array}{lcll} \mathbb{T}^n & \to & \mathbb{T}^{(n-1)} & (n \geq 1) \quad (a) \\ \mathbb{P}^n & \to & \mathbb{P}^{(n-2)} & (n \geq 3) \quad (b) \\ \mathbb{P}^n & \to & \mathbb{T}^{\frac{n-2}{2}} & (n = 2k,\ k \geq 1) \quad (c) \end{array}$$

Ce coup correspond au cas où l'on « casse » l'anse[1] d'un tore dans les deux premiers cas, ou le goulot d'une bouteille de Klein dans les deux derniers cas (le deuxième cas pouvant être vu des deux façons).

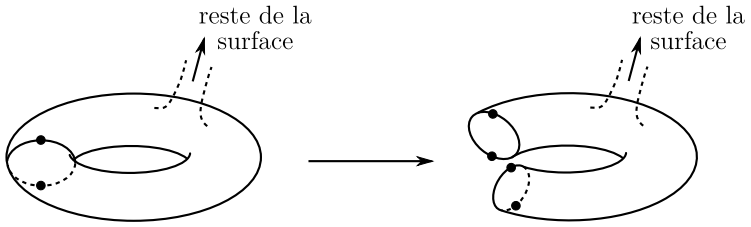


FIGURE 11.8 – Coup de type II.B.1.(a) ou II.B.1.(b).

Coup (type II.B.2)**.** *Couper $\mathscr{R}$ selon $\mathscr{L}$ engendre un unique bord.*

On obtient la relation $\chi(\mathscr{S}) = \chi(\mathscr{S}') - 1$, ce qui conduit cette fois à :

$$\begin{array}{lcll} \mathbb{P}^n & \to & \mathbb{P}^{(n-1)} & (n \geq 2) \quad (a) \\ \mathbb{P}^n & \to & \mathbb{T}^{\frac{n-1}{2}} & (n = 2k+1,\ k \geq 0) \quad (b) \end{array}$$

Ce coup correspond à ce qui se passe quand on casse un plan projectif. Pour le visualiser, on a représenté un tel coup en gris sur la figure 11.10, joué sur un ruban de Möbius, une surface qui est homéomorphe à un plan projectif à un bord.

C'est ce coup qui permet au premier joueur de gagner le jeu à deux points de départ sur le plan projectif, situation qui diffère du jeu sur le plan, où c'est le deuxième joueur qui gagne (cf annexe A). Nous expliquons la stratégie gagnante au paragraphe suivant.

1. Selon le point de vue adopté, la surface compacte orientable de genre n peut en effet être désignée comme étant un tore à n *trous*, ou comme une sphère à n *anses*.

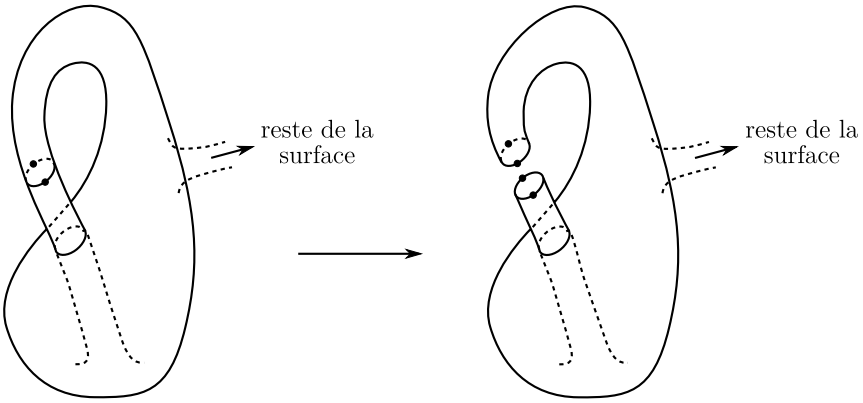


FIGURE 11.9 – Coup de type II.B.1.(b) ou II.B.1.(c).

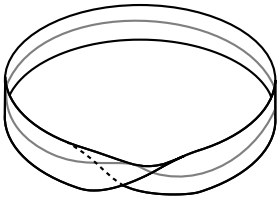


FIGURE 11.10 – Coup de type II.B.2 sur un ruban de Möbius.

11.3.2 Jeu à deux points sur le plan projectif

Le jeu commence avec deux points, soit un total de 6 vies. Le premier joueur peut gagner en faisant en sorte qu'il n'y ait qu'un seul point isolé à la fin de la partie, qui ainsi se sera terminée en $6 - 1 = 5$ coups. Pour gagner, il n'a pas le choix de son premier coup, il doit jouer le seul coup de type II.B.2.(b). Ceci est représenté en haut de l'arbre solution de la figure 11.11.

Le deuxième joueur a ensuite trois réponses possibles. Nous avons représenté en pointillés le coup que doit jouer le premier joueur au tour suivant, après quoi, quelle que soit la façon dont la partie se termine, il est sûr de gagner.

On notera en particulier qu'après le coup de type II.B.2.(b) du premier joueur, la surface associée à l'unique région de la position n'est plus un plan projectif : conformément au paragraphe 11.3.1, elle se comporte ensuite comme une sphère, et il n'y a plus de coup de type II.B.2.(b) disponible.

11.3.3 Cas particuliers

Lorsque le jeu se déroule sur un plan ou une sphère, toute région créée est homéomorphe à une sphère à bord. Par conséquent, parmi tous les coups présentés au paragraphe 11.3.1, on n'utilise que les coups de type I et de type II.A.(a) avec $n = k = 0$.

De même, si un jeu débute sur une surface orientable, toutes les régions qui apparaissent lors du déroulement de la partie sont homéomorphes à une surface orientable. On n'utilise que les coups de type I, de type II.A.(a), et de type II.B.1.(a).

Remarquons d'ailleurs que seul le dernier type de coup, II.B.1.(a), qui correspond à « casser une anse », est susceptible de changer le déroulement de la partie sur une surface orientable par rapport au jeu sur le plan. Si l'on interdisait de jouer les coups de ce type, le jeu sur une surface orientable serait exactement le même que celui sur le plan. Les positions qui ne

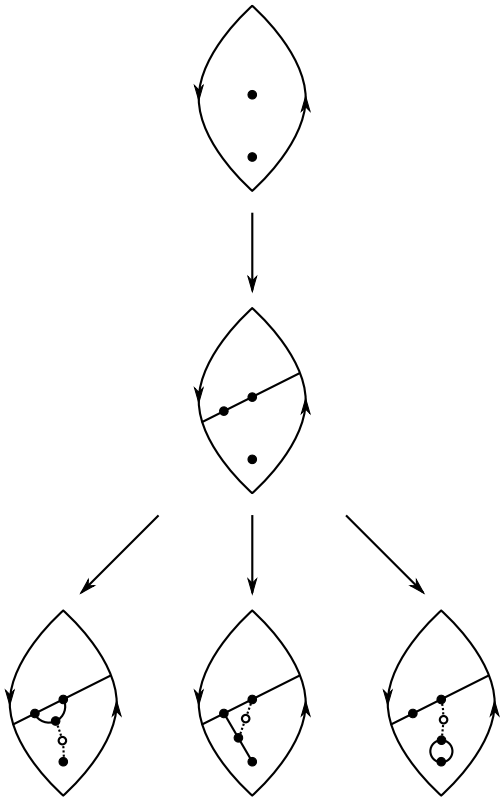


FIGURE 11.11 – Arbre solution partiel pour le jeu à 2 points sur le plan projectif.

diffèrent que par les surfaces orientables associées à leurs régions seraient toutes équivalentes (avec le même arbre canonique). Ceci implique que le chaos induit par l'introduction des surfaces orientables reste somme toute assez limité.

11.4 Représentation adaptée aux surfaces

Au chapitre 9, nous avons expliqué comment représenter les positions du jeu de Sprouts avec des chaînes de caractères. Dans cette section, nous expliquons comment généraliser cette représentation pour qu'elle fonctionne sur les surfaces compactes.

11.4.1 Représentation en chaîne

La principale modification par rapport au jeu sur le plan vient de la nécessité de noter quelle est la surface associée à chacune des régions composant la position. Pour cela, nous avons choisi de simplement associer à chaque région un nombre entier qui correspond au genre. Ainsi, les régions orientables sont notées avec un nombre positif, et les régions non-orientables avec un nombre négatif. Les régions planes (homéomorphes à une sphère à bords) sont elles notées sans nombre, ce qui permet de garder la compatibilité avec le jeu classique.

Le caractère « `@` » est placé avant l'indication du nombre, ce qui donne par exemple `0*2.AB@-1|0.CD@2|AB.CD` pour une position composée de trois régions homéomorphes respectivement à un plan projectif, un tore à deux trous, et une sphère.

Par ailleurs, avec les surfaces compactes, un nouveau type de sommet apparaît. Dans une partie sur le plan, les sommets à une vie sont de deux types seulement : ou bien ils

n'appartiennent qu'à une seule région, et alors ils n'appartiennent également qu'à une seule frontière, ou bien ils appartiennent à deux régions et donc à deux frontières différentes. Mais dans le cas des surfaces compactes, les coups de type II.B.1 engendrent des sommets à une vie qui sont sur deux frontières tout en n'appartenant qu'à une seule région. Bien que la distinction de ces trois types de sommets ne soit pas nécessaire pour programmer le Sprouts, celle-ci nous permet d'accélérer certaines fonctions liées à la canonisation.

Comme dans le cas des sommets qui n'appartiennent qu'à une seule frontière, nous notons ces nouveaux sommets en lettres minuscules, mais en commençant par `z` et dans l'ordre inverse pour les différencier. Par exemple, en partant de la position de départ à 2 points sur le tore, notée `0*2@1`, et en jouant un coup de type II.B.1.(a), on obtient une position notée `0.zy.zy`.

Enfin, détaillons un piège technique dans le cas des sommets à une vie qui apparaissent deux fois à l'intérieur d'une même frontière. Lorsque l'on joue une partie dans le plan, il est possible de simplifier les notations en remplaçant de tels sommets par des parenthèses [2]. Par exemple, dans la figure 11.12, à gauche, on peut noter la frontière de deux façons équivalentes : `abcdedcafgf` ou `(b((e)))(g)`.

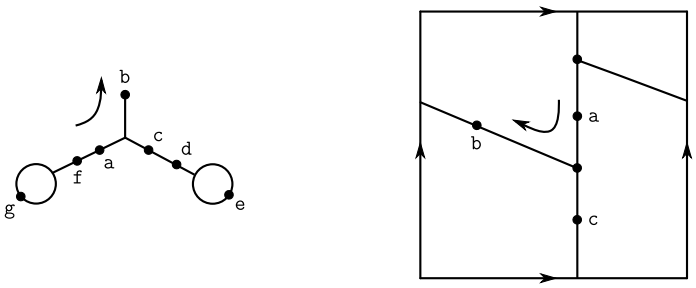


FIGURE 11.12 – Positions parenthésable et non parenthésable.

Par contre, cela n'est plus possible si l'on joue une partie sur une surface autre que le plan. L'exemple de la figure 11.12, à droite, correspond à une partie à deux points de départ sur le tore, dans laquelle 3 coups ont été joués. Si l'on parcourt la frontière dans le sens indiqué par la flèche, on rencontre dans l'ordre les sommets `abcabc`, ce qui ne correspond pas à un parenthésage correct.

Autre exemple, la position obtenue après un coup sur la figure 11.11 se note `0.abab` et n'est pas non plus parenthésable.

11.4.2 Effets de l'orientabilité

La notation d'une région associée à une surface orientable obéit à la même règle que pour une région classique du jeu sur le plan. Quand on note une frontière, on énumère ses sommets en tournant autour. Il faut alors toujours tourner dans le même sens autour de toutes les frontières de cette région (soit direct, soit indirect).

Dans le cas d'une région associée à une surface non orientable, on ne doit plus tenir compte de l'orientation, et donc, on peut tourner dans des sens différents suivant les frontières quand on énumère leurs sommets. La figure 11.13 montre que sur un plan projectif, quand on énumère les sommets dans un même sens, une même frontière peut se lire `abc` ou `acb`.

Par contre, quand on joue un coup du type II.A.(c), II.B.1.(c) ou II.B.2.(b), on crée des régions orientables. Il faut donc choisir une orientation pour chaque frontière de la région orientable nouvellement créée. Chaque choix d'orientation correspond à un coup possible.

2. Une idée de Dan Hoey.

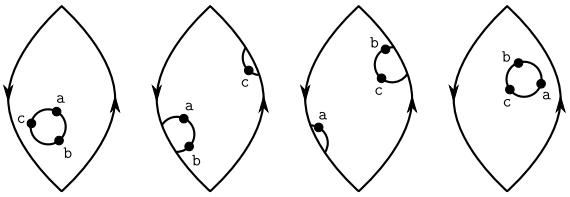


FIGURE 11.13 – Effet de la non-orientabilité.

La figure 11.14 montre un exemple de coup de type II.B.2.(b), on voit que l'orientation des frontières dans la nouvelle surface change suivant la façon dont le coup serpente.

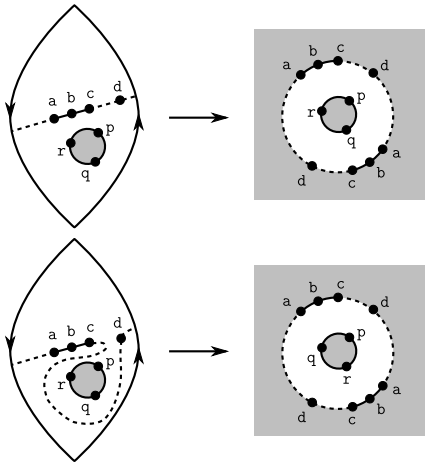


FIGURE 11.14 – Passage d'une surface non orientable à une surface orientable.

Ce choix de l'orientation, lorsqu'une région orientable émerge d'une région non orientable, augmente encore le nombre d'options dans le cas du jeu sur des surfaces non orientables. Ainsi, si une région orientable à n frontières est créée, il y a 2^{n-1} façons distinctes d'orienter les frontières.

11.4.3 Programmation des coups

Nous avons décrit dans le paragraphe 9.2.5 comment transformer les chaînes de caractères pour calculer les représentations des différents coups jouables à partir d'une position. Nous allons maintenant généraliser ces explications au jeu sur les surfaces. Pour décrire les différents types de coups, nous suivons la terminologie du paragraphe 11.3.

Coup de type I

Si l'on suppose que dans les frontières $\mathtt{x}_1...\mathtt{x}_m$ avec $m \geq 2$ et $\mathtt{y}_1...\mathtt{y}_n$ avec $n \geq 2$, on relie $\mathtt{x}_\mathrm{i}$ à $\mathtt{y}_\mathrm{j}$ en créant $\mathtt{z}$, alors :

* la frontière issue de la fusion sera $\mathtt{x}_\mathrm{i}...\mathtt{x}_m\mathtt{x}_1...\mathtt{x}_i\mathtt{z}\mathtt{y}_\mathrm{j}...\mathtt{y}_n\mathtt{y}_1...\mathtt{y}_j\mathtt{z}$ si l'on joue dans une région orientable.

* elle pourra également être $\mathtt{x}_\mathrm{i}...\mathtt{x}_m\mathtt{x}_1...\mathtt{x}_i\mathtt{z}\mathtt{y}_\mathrm{j}...\mathtt{y}_1\mathtt{y}_\mathrm{n}...\mathtt{y}_j\mathtt{z}$ si l'on joue dans une région non orientable.

Notons que dans le cas particulier où la frontière $\mathtt{x}_1...\mathtt{x}_m$ n'a qu'un seul sommet $\mathtt{x}_1$, alors $\mathtt{x}_\mathrm{i}...\mathtt{x}_m\mathtt{x}_1...\mathtt{x}_i = \mathtt{x}_1$.

Coup de type II.A, ou II.B.1.(a) ou (b)

Si l'on suppose que dans la frontière $\mathtt{x}_1...\mathtt{x}_n$, on relie $\mathtt{x}_\mathrm{i}$ à $\mathtt{x}_\mathrm{j}$ $(i \leq j)$ en créant $\mathtt{z}$, on obtient deux frontières : $\mathtt{x}_\mathrm{j}...\mathtt{x}_n\mathtt{x}_1...\mathtt{x}_i\mathtt{z}$ et $\mathtt{x}_\mathrm{i}...\mathtt{x}_j\mathtt{z}$.

Coup de type II.B.1.(c)

Si l'on suppose que dans la frontière $\mathtt{x}_1...\mathtt{x}_n$, on relie $\mathtt{x}_\mathrm{i}$ à $\mathtt{x}_\mathrm{j}$ $(i \leq j)$ en créant $\mathtt{z}$, on obtient deux frontières : $\mathtt{x}_\mathrm{j}...\mathtt{x}_n\mathtt{x}_1...\mathtt{x}_i\mathtt{z}$ et $\mathtt{x}_\mathrm{j}...\mathtt{x}_i\mathtt{z}$.

Coup de type II.B.2.

Si l'on suppose que dans la frontière $\mathtt{x}_1...\mathtt{x}_n$, on relie $\mathtt{x}_\mathrm{i}$ à $\mathtt{x}_\mathrm{j}$ $(i \leq j)$ en créant $\mathtt{z}$, on obtient une unique frontière : $\mathtt{x}_\mathrm{j}...\mathtt{x}_n\mathtt{x}_1...\mathtt{x}_i\mathtt{z}\mathtt{x}_\mathrm{j}...\mathtt{x}_i\mathtt{z}$.

11.4.4 Difficulté de la programmation

Les nombreuses modifications qui interviennent dans le jeu sur les surfaces rendent la programmation du calcul des options difficile. Le risque d'erreurs est assez important.

Contrairement au jeu sur le plan, la vérification manuelle de certaines positions manque d'efficacité pour détecter d'éventuelles erreurs de programmation, vu le grand nombre d'options des positions de Sprouts sur les surfaces, et vu la complexité de leur représentation graphique. Dès que les positions se compliquent un peu, il est difficile de générer à la main sans erreur toutes les options possibles d'une position, en particulier sur les surfaces non orientables. La vérification manuelle n'est envisageable que pour de petits cas (avec un petit nombre de points de départ).

Il serait probablement plus efficace, pour détecter des erreurs de programmation, de confronter les bases de données engendrées par notre programme à celles d'une autre implémentation du jeu de Sprouts sur les surfaces. Le problème est qu'à notre connaissance, notre programme est le premier à être capable d'effectuer ce type de calculs, et il faudra donc attendre une implémentation indépendante pour une vérification plus complète.

11.5 Genre limite des surfaces

Dans cette section, nous discutons le concept théorique de *genre limite*. Ce concept exprime que si l'on fixe les frontières d'une région et que l'on fait varier la surface qui lui est associée, alors il est inutile de considérer des surfaces de genre trop important.

Le genre limite a des implications théoriques, en décrivant comment se comportent les positions lorsque l'on fait varier les surfaces associées à leurs régions, mais également des applications pratiques, en permettant de simplifier les calculs sur les surfaces.

11.5.1 Cas des régions à 3 vies ou moins

Pour commencer, nous étudions le cas particulier des très petites régions. Étant donnée une position qui contient une région à 3 vies ou moins, la proposition suivante exprime que la surface associée à cette région n'a aucune importance.

Proposition 13. *Si dans une position, une région comporte 3 vies ou moins, changer la surface associée à cette région ne modifie pas l'arbre canonique de cette position.*

En effet, l'argument exposé au paragraphe 9.3.5 reste valide quelle que soit la surface associée à la région : il est toujours possible de relier toute paire de sommets de la région, quels que soient les coups joués (ce qui aboutit à la création de sommets génériques 2). Modifier la surface ne peut donc pas rajouter de coup.

Là encore, les sommets notés en minuscules, mais à partir de `z` (ceux qui appartiennent à une seule région mais à deux frontières) peuvent être remplacés par le sommet générique `2`. Par exemple, la région `2z.Az` est équivalente à la région `22A`.

L'intérêt pratique de cette proposition est important. Dès lors qu'une région a 3 vies ou moins, nous pouvons supprimer les informations relatives à la surface qui lui est associée. Par exemple, `0*4.AB|0*2.C|ABC@3` est remplacée par `0*4.AB|0*2.C|ABC`. Ce faisant, on identifie des positions équivalentes, donc on réduit la taille des arbres de jeu, et l'on accélère le calcul.

Remarquons que 3 est la meilleure valeur : nous avons représenté sur la figure 11.15 une position de Sprouts à 4 vies dont l'arbre canonique dépend de la surface.

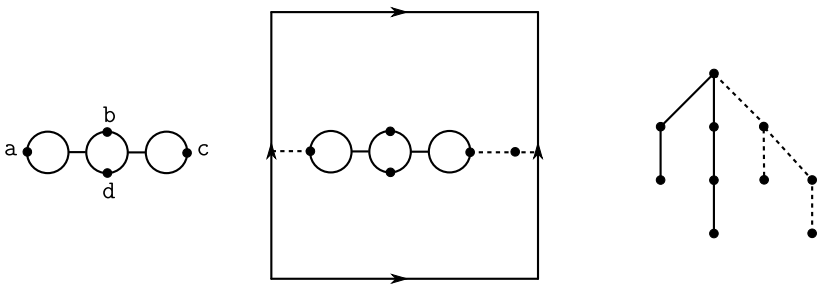


FIGURE 11.15 – Position dont l'arbre canonique dépend de la surface de jeu.

Si la région extérieure est un plan, l'arbre canonique obtenu est représenté en traits pleins à droite de la figure : ou bien on relie `a` à `c` ou `b` à `d`, et la partie se termine en deux coups. Ou bien on relie `a` à `b`, `b` à `c`, `c` à `d` ou `d` à `a`, et la partie se termine en trois coups. Par contre, si l'on joue sur $\mathbb{T}$, le coup représenté en pointillés donne la branche de l'arbre canonique représentée en pointillés : ou bien on relie `b` à `d` à l'intérieur de la petite région circulaire, et la partie est finie, ou bien on joue tout autre coup, et la partie se termine en un coup supplémentaire.

Ainsi, suivant que cette région est associée à un plan ou à un tore, la position n'a pas le même arbre canonique. La proposition 13 ne marche pas en général avec les régions à 4 vies ou plus. Elle fonctionne néanmoins avec certaines régions particulières, comme par exemple les régions dont la représentation en chaîne est du type `A.B.C.D`.

Les représentations en chaînes des positions de la figure 11.15 sont `2A2B|AB` (sur le plan) et `2A2B@1|AB` (sur le tore). Remarquons que ces positions non seulement n'ont pas le même arbre canonique, mais qu'elles n'ont pas le même nimber. On peut déterminer facilement leurs nimbers à partir de leurs arbres canoniques : le nimber de `2A2B|AB` est 2, celui de `2A2B@1|AB` est $\mathfrak{Z}$. Ceci montre que ces deux positions sont fondamentalement différentes en version normale. Par exemple, `2A2B|AB+2AB|AB` est perdante, tandis que `2A2B@1|AB+2AB|AB` est gagnante. Les différences observées ne sont donc pas superficielles.

11.5.2 Genre limite sur les surfaces orientables

On peut énoncer un résultat moins fort, mais plus général, qui délimite l'influence des surfaces sur le jeu. On se restreint pour commencer aux surfaces orientables.

Proposition 14. *Étant donnée une région $\mathscr{R}$, il existe un entier naturel $g(\mathscr{R})$, le* genre limite*, tel que quel que soit l'entier $n \geq g(\mathscr{R})$, et quelle que soit la position $\mathscr{P}$ contenant la région $\mathscr{R}$, l'arbre canonique de $\mathscr{P}$ où $\mathscr{R}$ est homéomorphe à $\mathbb{T}^n$ est identique à l'arbre canonique de $\mathscr{P}$ où $\mathscr{R}$ est homéomorphe à $\mathbb{T}^{n+1}$.*

D'une manière moins formelle, ce résultat exprime que lorsqu'il y a trop d'anses dans une région $\mathscr{R}$, on n'a pas le temps de toutes les utiliser avant que la partie ne se termine. L'arbre

canonique de $\mathscr{P}$ est donc constant à partir d'une certaine valeur $g(\mathscr{R})$ de n : on obtient un *arbre canonique limite*.

Démonstration. On travaille par récurrence sur le nombre de vies de la région. Pour une région $\mathscr{R}$ incluse dans une position $\mathscr{P}$, quatre types de coups peuvent la modifier. Chaque coup va transformer $\mathscr{R}$ en une ou deux nouvelles régions, chacune ayant strictement moins de vies que $\mathscr{R}$, de sorte que nous puissions affirmer par hypothèse de récurrence que ces nouvelles régions ont un genre limite.

1. coups de type I dans la région : chaque coup de ce type transforme $\mathscr{R}$ en une région $\mathscr{R}_i$, avec $g(\mathscr{R}_i) = a_i$ (l'indice i ne peut prendre qu'un nombre fini de valeurs, parce qu'il n'existe qu'un nombre fini de coups à partir d'une position donnée). La surface associée à chaque $\mathscr{R}_i$ est la même que la surface associée à $\mathscr{R}$. Par conséquent, si la surface associée à $\mathscr{R}$ est $\mathbb{T}^n$ avec $n \geq \max\{a_i\}$, alors l'arbre canonique de toute option de $\mathscr{P}$ obtenue après avoir joué un coup de type I est son arbre canonique limite.
2. coups de type II.B.1.(a) dans la région : on obtient des régions $\mathscr{R}_j$, avec $g(\mathscr{R}_j) = b_j$. Comme le genre de la région diminue d'une unité lors d'un tel coup, il faut que $n \geq \max\{b_j + 1\}$ pour s'assurer d'avoir l'arbre canonique limite.
3. coups de type II.A.(a) dans la région : la région est scindée en deux nouvelles régions $\mathscr{R}_k$ et $\mathscr{R}'_k$, avec $g(\mathscr{R}_k) = c_k$ et $g(\mathscr{R}'_k) = d_k$. Il faut cette fois-ci que $n \geq \max\{c_k + d_k\}$, car un coup de ce type partage le genre de $\mathscr{R}$ entre les deux régions créées.
4. coups extérieurs à la région : il suffit de considérer les coups qui tuent une ou deux vies sur les frontières extérieures de la région (il faut considérer toutes les façons de supprimer une ou deux vies, y compris lorsqu'il y a deux vies sur deux frontières différentes). On obtient des régions $\mathscr{R}_l$, avec $g(\mathscr{R}_l) = e_l$. Comme la surface associée à $\mathscr{R}$ n'est pas modifiée, il suffit d'avoir $n \geq \max\{e_l\}$.

Si la surface associée à $\mathscr{R}$ est $\mathbb{T}^n$, avec $n = \max\{a_i; b_j + 1; c_k + d_k; e_l\}$, alors l'arbre canonique de la position $\mathscr{P}$ est l'arbre canonique limite. Donc $g(\mathscr{R})$ existe, et $g(\mathscr{R}) \leq \max\{a_i; b_j + 1; c_k + d_k; e_l\}$. □

Un exemple d'arbre canonique limite a été donné sur la figure 11.15 : l'arbre du jeu sur le plan est représenté en traits pleins, et celui du jeu sur le tore s'obtient en rajoutant la partie en pointillés. Dans le cas de cette position, le genre limite est 1, donc à partir du tore, l'arbre canonique est constant, c'est l'arbre canonique limite.

En général, les arbres canoniques évoluent au fur et à mesure que l'on augmente le genre de la surface associée à une région, mais pas toujours en gagnant des branches, comme celui de la figure 11.15. Car si augmenter le genre fait systématiquement augmenter la taille de l'arbre de jeu, ceci peut aboutir à rendre identiques certaines de ses branches, qui dès lors fusionnent dans l'arbre canonique.

11.5.3 Conséquences du genre limite

Selon la proposition 14, si l'on fixe une position, que l'on sélectionne une de ses régions, et que l'on augmente le genre de la surface qui lui est associée, alors au bout d'un moment, l'arbre canonique de la position ne change plus. Ceci implique que l'issue de la position (gagnante ou perdante) reste constante au-delà d'un certain genre, que ce soit en version normale ou en version misère. Le même phénomène se produit aussi avec le nimber, ce qui s'avère utile si la position apparaît dans une somme de positions pour un jeu en version normale, ainsi qu'avec l'arbre canonique réduit (pour les sommes de positions en version misère).

En particulier, une fois un entier p fixé, nous pouvons dire qu'il existe une valeur limite n_0, telle que si $n \geq n_0$, le jeu à p points sur $\mathbb{T}^n$ a le même vainqueur que le jeu à p points

sur $\mathbb{T}^{n_0}$. Nous démontrons dans le paragraphe suivant 11.5.4 que $n_0 = 0$ si $p = 2$, c'est-à-dire que la position de départ à 2 points a le même gagnant quelle que soit la surface orientable sur laquelle on joue. Les résultats que nous avons obtenus informatiquement nous ont même permis de conjecturer que $n_0 = 0$ pour tout p.

Beaucoup de résultats du type de celui du paragraphe 11.5.4 peuvent être établis en corollaires de la proposition 14, mais leur démonstration ne nécessite pas forcément d'avoir recours aux arbres canoniques. Notamment, la proposition 13 est très utile pour obtenir de telles démonstrations.

Par exemple, on peut démontrer rapidement que la position de départ à 4 points est gagnante sur toute surface orientable. En effet, `0*4@n` a pour option `0*3.AB|AB@n` (c'est l'option obtenue en reliant un sommet à lui-même, en mettant d'un côté les autres sommets, et de l'autre côté les n trous de la surface. Or la proposition 13 implique que `0*3.AB|AB@n` est équivalente à `0*3.AB|AB`, qui est une position perdante.

11.5.4 Jeu à 2 points sur les surfaces orientables

Nous allons montrer de manière élémentaire que si l'on joue avec deux points sur une surface orientable, c'est le deuxième joueur qui dispose d'une stratégie gagnante. Il y a 6 vies : le deuxième joueur gagne en faisant en sorte que la partie se termine avec 2 points isolés, donc en $6 - 2 = 4$ coups.

Au premier coup, le premier joueur a 3 possibilités :

* coup de type I : il relie les deux points entre eux. Le deuxième joueur forme alors un quadrilatère avec un coup de type II.A.(a), séparant le terrain de jeu en deux régions. Aux coups suivants, lorsque le premier joueur jouera dans une région, le deuxième joueur jouera dans l'autre pour gagner.
* coup de type II.A.(a) : il sépare la surface en deux régions, en reliant un point à lui-même. Le deuxième joueur joint alors les deux points de la frontière dans la région qui ne comporte pas le point inutilisé. Il ne restera alors plus que deux coups à jouer, avec ce point inutilisé.
* coup de type II.B.1.(a) : il casse une anse, en reliant un point à lui-même. Le deuxième joueur relie alors l'autre point à lui-même selon une trajectoire parallèle, séparant la surface en deux parties, un cylindre, et le reste de la surface. Pour gagner, si le premier joueur joue dans le cylindre, le deuxième jouera le dernier coup dans le reste de la surface, et réciproquement.

La figure 11.16 présente le début de ce raisonnement appliqué au tore. Le premier coup est en trait plein, la réponse du deuxième joueur est en pointillés. Notons que la réponse apportée au coup de type II.B.1.(a) est la seule possible, toutes les autres sont gagnantes pour le premier joueur.

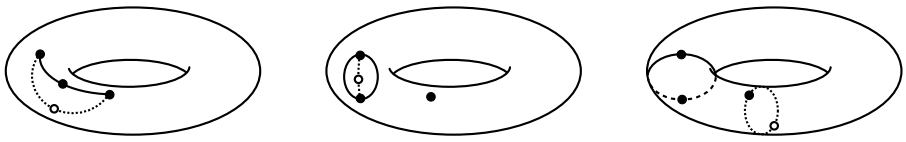


FIGURE 11.16 – Solution du jeu à deux points sur le tore.

11.5.5 Genre limite sur les surfaces non orientables

La propriété précédente de genre limite s'applique également aux surfaces non orientables, avec une variante : à partir du genre limite, l'arbre canonique reste constant sur $\mathbb{P}^n$ si et seulement si n conserve la même parité.

Proposition 15. *Étant donnée une région $\mathscr{R}$, il existe un entier naturel $g'(\mathscr{R})$ tel que quel que soit l'entier $n \geq g'(\mathscr{R})$, et quelle que soit la position $\mathscr{P}$ contenant la région $\mathscr{R}$, l'arbre canonique de $\mathscr{P}$ où $\mathscr{R}$ est homéomorphe à $\mathbb{P}^n$ est identique à l'arbre canonique de $\mathscr{P}$ où $\mathscr{R}$ est homéomorphe à $\mathbb{P}^{n+2}$.*

La différence avec la propriété précédente provient, d'une part, de ce que l'existence de coups de type II.B.1.(c) et II.B.2.(b) dépend de la parité de n, et d'autre part, des coups de type II.A.(c). En effet, dans un tel coup, le genre de la surface associée à la région initiale et le genre de la surface associée à la région non orientable engendrée par le coup ont forcément la même parité.

Deux cas sont donc envisageables : à partir du genre limite, soit l'arbre canonique reste constant, soit il alterne entre deux arbres canoniques limites. On peut dire la même chose de l'issue, voire du nimber ou de l'arbre canonique réduit : ils peuvent être constants à partir d'un certain rang, ou alterner entre deux valeurs différentes.

En pratique, les calculs de petites positions ne permettent pas d'observer d'alternance. Par exemple, les positions de départ à 2, 3 et 4 points sont gagnantes sur l'ensemble des surfaces non orientables. Nous avons donc commencé par tester de nombreuses petites positions, mais nous avons été bien en peine d'observer la moindre alternance d'issue. Cette situation était assez frustrante : la théorie prévoit la possibilité d'alternance entre positions perdantes et gagnantes, mais nous étions incapable d'en trouver une seule.

Notre programme est alors venu à notre secours. Tout d'abord avec un résultat peu spectaculaire, mais rassurant : l'alternance entre deux arbres canoniques différents pour certaines petites positions. C'est le cas de `0.0.2@-n`, dont le genre limite est 3 sur les surfaces non orientables. L'étude manuelle de cet exemple est envisageable, mais est très longue et fastidieuse, aussi nous l'avons omise dans cet exposé. C'est également le cas de la position de départ à 3 points `0*3@-n`, dont le genre limite est 8, mais cette fois, ce résultat est beaucoup trop complexe pour l'étudier à la main.

Nous avons observé un certain nombre de résultats similaires sur de petites positions. Cependant, ces résultats sont d'un intérêt purement théorique : dans chaque cas, seul l'arbre canonique alterne, ce n'est pas le cas du nimber, ni de l'arbre canonique réduit. Ainsi, impossible d'espérer les convertir en une position où l'issue alterne (en version normale ou en version misère), ce qui aurait été bien plus spectaculaire.

Nous avons finalement trouvé une telle position, sans la chercher : il s'agit tout simplement du jeu à 11 points de départ. Notre programme nous a permis d'observer que pour la position `0*11@-n`, c'est le premier joueur qui a une stratégie gagnante si `n` est impair, et le second si `n` est pair. Le coup gagnant pour le premier joueur n'existe que si le genre `n` de la surface non-orientable est impair. C'est un coup de type II.A.(c), qui consiste à jouer vers la position `0*8.AB@-1|0*2.AB@m` où `m` est un entier positif ou nul.

Cela fournit ainsi la validation expérimentale d'un résultat théorique : nous avons d'abord déterminé qu'il était théoriquement possible que les arbres canoniques, et même les issues, alternent avec le genre des surfaces sur certaines positions, avant d'en trouver des exemples concrets avec notre programme.

11.6 Résultats obtenus par la programmation

11.6.1 Surfaces orientables en version normale

Nous avons mené le calcul pour toutes les positions de départ jusqu'à 14 points avec un genre allant jusqu'à 9. Pour aucune de ces positions, le joueur disposant d'une stratégie gagnante ne change par rapport au jeu sur le plan. Et même, résultat plus fort, le nimber de ces positions de départ est $\mathbb{1}$ lorsque c'est le premier joueur qui dispose d'une stratégie gagnante. On peut donc exprimer une conjecture plus générale que celle de [4] :

Conjecture 7. *Le nimber de la position de départ à n points sur une surface orientable est* $\mathbb{0}$ *si* $n \equiv 0$, 1 *ou* 2 mod 6, *et* $\mathbb{1}$ *sinon.*

Ce résultat n'est pas totalement surprenant, si l'on se souvient que seuls les coups de type II.B.1.(a) induisent une modification par rapport au jeu sur le plan. Néanmoins, il ne faut pas espérer une démonstration reposant sur le fait qu'il serait équivalent de jouer sur le plan ou sur une surface orientable quelle que soit la position, car nous avons vu avec la figure 11.2 un exemple de position dont l'issue change suivant qu'elle est jouée sur le plan ou sur le tore.

De tels résultats ne sont pas si rares. Outre la position de la figure 11.15, dont le nimber vaut 2 sur le plan et $\mathbb{3}$ sur le tore, la figure 11.17 présente d'autres positions dont le nimber est différent sur le plan et sur les autres surfaces orientables. Elles sont suffisamment simples (5 ou 6 vies) pour que l'on puisse vérifier les résultats assez rapidement à la main. La première en partant de la gauche est de nimber 2 si la région extérieure est un plan, $\mathbb{4}$ si cette région est toute autre surface orientable. Pour la deuxième, ces nimbers sont respectivement $\mathbb{1}$ et $\mathbb{4}$, et pour la dernière, ces nimbers sont respectivement $\mathbb{0}$ et $\mathbb{3}$.

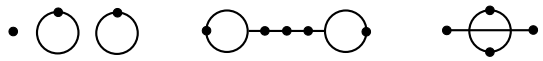


FIGURE 11.17 – Positions dont le nimber change sur le tore.

Pour de petites valeurs de n (jusqu'à 6), nous ne nous sommes pas contentés de calculer le nimber jusqu'à $\mathbb{T}^9$, mais nous avons calculé ce nimber sur $\mathbb{T}^n$ pour tout n, par une méthode identique à celle expliquée sur la position de départ à 4 points dans le paragraphe 11.5.3. Il est probablement possible d'obtenir bien mieux que $n = 6$, via la programmation d'une automatisation du procédé.

Plus généralement, il est envisageable de programmer des algorithmes de recherche qui favoriseraient les positions où les surfaces ont disparu, du fait de la proposition 13, ce qui contribuerait à accélérer les calculs.

11.6.2 Surfaces non orientables en version normale

Jouer sur des surfaces non orientables change les résultats de manière beaucoup plus significative. Il suffit de considérer le jeu à deux points sur $\mathbb{P}$ (§11.3.2) pour avoir un premier exemple de résultat qui change par rapport au jeu sur le plan (en effet, c'est le deuxième joueur qui a une stratégie gagnante sur toute surface orientable, et le premier joueur sur toute surface non orientable).

	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathbb{S}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$
$\mathbb{P}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$
$\mathbb{P}^2$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$
$\mathbb{P}^3$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{4}$	$\mathbb{0}$	$\mathbb{0}$	> 2
$\mathbb{P}^4$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$> \mathbb{0}$
$\mathbb{P}^5$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$> \mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	> 2
$\mathbb{P}^6$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$> \mathbb{0}$
$\mathbb{P}^7$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$> \mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	> 2
$\mathbb{P}^8$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{0}$	$> \mathbb{0}$

TABLE 11.1 – Nimbers des positions de départ à p points sur $\mathbb{P}^n$.

La table 11.1 présente des résultats obtenus à l'aide de notre programme. Les positions de départ ont entre 2 et 14 points, et l'on joue sur les surfaces $\mathbb{P}^n$ avec n entre 1 et 8.

Pour pouvoir comparer plus aisément, nous avons rappelé les résultats sur le plan en haut du tableau. Une différence entre le jeu sur le plan et celui sur les surfaces non orientables apparaît dès 2 points de départ, comme nous l'avons vu dans le paragraphe 11.3.2, mais il y a de nombreuses autres irrégularités. Le cas du jeu à 8 points, perdant sur $\mathbb{P}$, mais gagnant à partir de $\mathbb{P}^2$, montre qu'il ne faut pas mettre toutes les surfaces non orientables dans le même panier.

Autre résultat surprenant, la découverte de positions de départ dont le nimber n'est ni $\mathbb{0}$ ni $\mathbb{1}$, la plus simple étant la position de départ à 11 points sur $\mathbb{P}^3$, dont nous avons réussi avec difficulté à calculer le nimber, qui vaut $\mathbb{4}$.

On remarque également l'alternance pour 11 points de départ, discutée au paragraphe 11.5.5. Nous n'avons pas pu établir formellement que cette alternance se poursuit pour tout n, mais il est probable que l'automatisation de la méthode du paragraphe 11.5.3 que nous avons discutée au paragraphe précédent puisse en venir à bout.

Enfin, contrairement au jeu sur les surfaces orientables, il semble plus difficile d'émettre une conjecture concernant les nimbers ou même les issues. Les résultats dont nous disposons pour l'instant sont trop partiels pour qu'émerge une régularité.

11.6.3 Version misère

L'implémentation du Sprouts sur les surfaces compactes est assez technique à cause de la difficulté théorique du sujet, mais les modifications nécessaires du programme sont limitées uniquement au code de génération des options d'une position de Sprouts. Une fois cette modification faite, tous les types de calculs qui étaient possibles pour le Sprouts sur le plan le deviennent pour le Sprouts sur les surfaces compactes. Nous avons donc pu mener des calculs de Sprouts sur les surfaces compactes en version misère.

Il n'y a pas grand-chose de spécifique à indiquer sur le sujet : la méthode de calcul est exactement la même que celle utilisée sur le plan en version misère, avec les arbres canoniques réduits (voir chapitre 4). Par ailleurs, les résultats sur le genre limite de la section 11.5 ont été établis sur les arbres canoniques, et sont donc valables aussi bien en version normale qu'en version misère. Sur les surfaces orientables, l'issue misère des positions de départ doit donc se stabiliser au-delà d'un certain genre limite. Sur les surfaces non orientables, l'issue misère doit se stabiliser, ou bien alterner avec la parité du genre de la surface.

Nous n'avons pas cherché à pousser les calculs très loin, avec simplement quelques valeurs jusqu'à 11 points de départ. Sur les surfaces orientables, nous avons calculé les issues en version misère de toutes les positions de départ de 2 à 11 points pour des tores de 1 à 3 trous. Nous avons fait la même constatation qu'en version normale, à savoir que pour un nombre de points initiaux fixés, le gagnant en version misère sur une surface orientable semble être le même que sur le plan.

Pour les surfaces non orientables, nous avons calculé les issues en version misère de toutes les positions de départ de 2 à 11 points sur les surfaces $\mathbb{P}^n$ avec n entre 1 et 4. Les résultats obtenus sont donnés dans la table 11.2. On constate, comme en version normale, que l'issue de certaines positions est différente sur le plan et sur certaines surfaces non orientables. Par exemple, la position de départ à 4 points en version misère est perdante sur le plan, mais gagnante sur la bouteille de Klein.

Par contre, l'issue misère de toutes ces positions semble se stabiliser quand le genre augmente. Théoriquement, il est tout à fait possible d'après les résultats du paragraphe 11.5.5 d'obtenir une alternance plutôt qu'une stabilisation pour certaines positions de départ, mais il faudrait effectuer des calculs avec un nombre plus élevé de points de départ pour espérer en trouver une.

On notera tout de même un résultat intéressant sur la position de départ à 11 points : d'après les résultats, on peut conjecturer que l'issue alterne avec le genre en version normale, mais qu'elle se stabilise avec le genre en version misère. Ces résultats ne sont pas

	2	3	4	5	6	7	8	9	10	11
$\mathbb{S}$	L	L	L	W	W	L	L	L	W	W
$\mathbb{P}$	L	L	W	W	W	L	L	L	W	W
$\mathbb{P}^2$	L	L	W	W	W	L	L	W	W	W
$\mathbb{P}^3$	L	L	W	W	W	L	L	W	W	W
$\mathbb{P}^4$	L	L	W	W	W	L	L	W	W	W

TABLE 11.2 – Issues en version misère des positions de départ sur $\mathbb{P}^n$.

contradictoires. La conjecture en version normale implique forcément de conjecturer aussi une alternance de l'arbre canonique, mais des arbres canoniques différents peuvent bien sûr avoir la même issue en version misère.

11.7 Conclusion

Le jeu de Sprouts sur les surfaces orientables est une variante dont l'étude n'est pas si triviale. Nous avons étudié en détail le jeu à deux points, et nous avons constaté que le nouveau coup induit par les surfaces ne possède qu'une réponse valide, ce qui en fait un coup intéressant à jouer face à un novice. Nous avons également observé plusieurs positions dont l'issue ou le nimber change sur le tore. Cependant, les différences observées avec le jeu plan ne semblent pas suffisantes pour changer l'issue d'une position de départ, que ce soit en version normale, ou en version misère.

Avec les surfaces non orientables, ce bémol n'existe plus. Les différences avec le jeu sur le plan sont beaucoup plus importantes, les arbres de jeu beaucoup plus complexes avec de très nombreuses options, si bien que l'on a pu observer des positions de départ dont l'issue sur certaines surfaces non orientables change par rapport au plan, que ce soit en version normale ou misère. Les éventuelles régularités dans les listes d'issues, similaires à celles des jeux octaux, et qui ont été observées sur les surfaces orientables, n'apparaissent pas encore de façon évidente sur les surfaces non orientables avec les seules valeurs que nous avons calculées.

Plus surprenant encore, on observe sur la position de départ à 11 points un phénomène étonnant : le vainqueur dépend de la parité du genre de la surface. La théorie du genre limite, qui a su prévoir qu'un tel phénomène pourrait se produire, ouvre également la voie pour établir des résultats généraux du type : « La position de départ à 6 points est perdante sur toute surface orientable », et permet d'envisager des améliorations substantielles de la rapidité de nos algorithmes de calcul.

Le Sprouts sur les surfaces est un jeu très riche, et nous sommes loin d'avoir poussé les calculs et la théorie autant qu'il est possible.

Chapitre 12

Le jeu de Cram

12.1 Introduction

Le jeu de Cram se joue sur un quadrillage avec des règles très simples. Les joueurs posent alternativement un domino sur deux cases vides adjacentes, jusqu'à ce que l'un d'entre eux ne puisse plus jouer. Un domino est simplement constitué de deux carrés gris, sans indication particulière. On trouve par exemple une présentation de ce jeu dans le volume 3 de *Winning Ways* p. 502-506 [6].

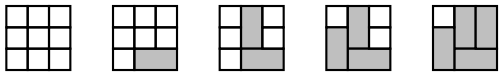


FIGURE 12.1 – Exemple de partie de Cram sur une grille 3×3 (le deuxième joueur gagne en version normale).

À partir de toute position, les mêmes coups sont disponibles pour les deux joueurs, ce qui rend le jeu de Cram impartial. Comme pour le jeu de Sprouts, il existe deux versions possibles de jeu suivant la convention de victoire choisie : version *normale* si celui qui ne peut plus jouer perd, version *misère* si au contraire il est déclaré vainqueur.

Il est intéressant d'indiquer pour quelle raison nous avons effectué des calculs sur le jeu de Cram. Initialement, nos travaux ne concernaient que le jeu de Sprouts en version normale, et étaient centrés avant tout sur les astuces de représentation liées au Sprouts. Ils ont ensuite évolués vers la théorie du nimber qui s'est montrée très efficace grâce aux nombreux découpages existants dans le jeu de Sprouts. Il était donc naturel de tenter d'appliquer les mêmes algorithmes à un autre jeu, qui possèderait lui aussi cette propriété de découpage. C'est ainsi que le Cram est apparu comme un candidat idéal.

Dans le cas de la version normale de jeu, nous allons appliquer les techniques de calculs des jeux impartiaux en version normale, décrites dans le chapitre 3, et dans le cas de la version misère celles des jeux impartiaux en version misère, décrites dans le chapitre 4.

12.2 Résultats connus

12.2.1 Stratégie de symétrie

Dans la version normale du jeu, où celui qui joue le dernier coup a gagné, il existe une stratégie de symétrie sur les plateaux de taille pair×pair, qui sont perdants (et donc de nimber $\mathbb{0}$), et sur les plateaux de taille pair×impair, qui sont gagnants.

Les plateaux de taille pair×pair sont symétriques par rapport au point central, et donc chaque fois que le premier joueur joue un coup, l'adversaire répond avec un coup symétrique par rapport à ce point. La figure 12.2 illustre cette stratégie. Le premier joueur a joué les coups 1 et 3, et le second joueur a répondu avec les coups symétriques 2 et 4. Avec cette stratégie, le deuxième joueur est certain de pouvoir jouer le dernier coup, et donc de gagner en version normale. Le plateau de départ est donc perdant.

FIGURE 12.2 – Stratégie de symétrie sur un plateau de taille pair×pair.

Inversement, les plateaux de taille pair×impair sont gagnants, car c'est cette fois le premier joueur qui dispose d'une stratégie de symétrie. Il effectue son premier coup avec un domino au centre du plateau, rendant le plateau symétrique vis-à-vis du point central de ce domino. Chaque fois que le deuxième joueur joue un coup, il répond avec un coup symétrique vis-à-vis de ce point central, s'assurant ainsi de pouvoir jouer le dernier coup. La figure 12.3 illustre cette stratégie. Après le coup 1 au centre, les coups 3 et 5 sont les symétriques des coups 2 et 4 du deuxième joueur.

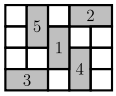


FIGURE 12.3 – Stratégie de symétrie sur un plateau de taille pair×impair.

Cette stratégie ne fonctionne pas sur les plateaux de taille impair×impair, et n'indique rien non plus sur le nimber des plateaux de taille pair×impair (en dehors du fait que ce nimber est non nul). Ce sont donc les deux principaux éléments que l'on va chercher à calculer informatiquement.

Enfin, cette stratégie de symétrie ne fonctionne que dans la version normale du jeu. Dans la version misère, elle n'a pas d'intérêt, car le but des joueurs est inverse, à savoir de ne pas jouer le dernier coup. Tous les plateaux sont donc intéressants à calculer en version misère.

12.2.2 Plateaux de taille $1 \times n$

Le jeu de Cram dans le cas des plateaux de taille $1 \times n$ est en fait une version déguisée d'un jeu impartial connu sous le nom de Dawson's Kayles. On peut en trouver une présentation dans [6], p. 92. Le jeu de Dawson's Kayles se joue avec des allumettes, comme le jeu de Nim, et le seul coup autorisé consiste à prendre deux allumettes de l'un des tas d'allumettes et à séparer éventuellement ce tas en deux tas différents. Cette règle est exactement celle du Cram jouée sur des plateaux de taille $1 \times n$. La position de départ de Dawson's Kayles avec un unique tas de n allumettes correspond à la position de départ d'un plateau de Cram $1 \times n$.

Le jeu de Dawson's Kayles est entièrement résolu en version normale. La suite des nimbers pour des tas d'allumettes de taille croissante (en partant d'un tas de 2 allumettes) commence par 1, 1, 2, 0, 3, 1, 1, 0, 3, 3, 2, 2, 4, 0, 5, 2, 2, 3, 3, 0, 1, 1, 3, 0, 2, 1, 1, 0, 4, 5, 2, 7, 4, 0, 1, 1, 2, 0, 3, 1... C'est une suite périodique, de période 34.

D'après la liste de Demaine et Hearn établie en 2008 [15] p. 13, la version misère du Dawson's Kayles n'est par contre pas résolue entièrement.

12.2.3 Calculs informatiques

Le jeu de Cram a été nettement moins étudié que la version partisane du jeu, appelée Domineering, où l'un des joueurs ne peut joueur que des dominos verticaux et l'autre que des dominos horizontaux. Nous n'avons trouvé en fait qu'une seule autre étude informatique du jeu de Cram, par Martin Schneider en 2009, dans le cadre de sa thèse de master [39], malheureusement disponible en allemand uniquement.

Les principaux résultats obtenus par Schneider, listés p. 111 de son mémoire, sont le nimber du plateau 5×7 en version normale, et la valeur de Grundy misère [1] du plateau 5×5 en version misère. Les calculs réalisés par Schneider sont particulièrement importants, car ils sont la seule source indépendante pour des plateaux de grande taille. Nous confirmons d'ailleurs tous les résultats obtenus par Schneider sur le jeu de Cram.

12.3 Découpages en positions indépendantes

Nous avons déjà vu à la section 2.6 que le Cram est un jeu découpable. Certaines positions peuvent se découper en positions indépendantes. Par exemple, la figure 12.4 montre une position qui est découpable en deux composantes indépendantes, car les joueurs ne peuvent jouer un coup que dans l'une ou l'autre des composantes. Il n'est pas possible de placer un domino qui serait à cheval sur les deux composantes.

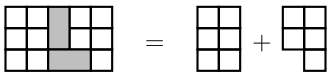


FIGURE 12.4 – Position de Cram découpable.

C'est l'existence de ces découpages qui justifie de réappliquer au Cram les algorithmes développés initialement pour le Sprouts. Par contre, la proportion de positions découpables varie notablement entre les deux jeux. Dans le cas du Sprouts, des découpages sont susceptibles de se produire en 2 coups seulement à partir de n'importe quelle position. Dans le cas du Cram, cela dépend de la taille du plateau.

La figure 12.5 montre par exemple que le découpage le plus rapide possible du plateau de taille 7×7 nécessite de jouer au moins 4 coups. Les découpages interviennent d'autant plus tard que le plateau est grand.

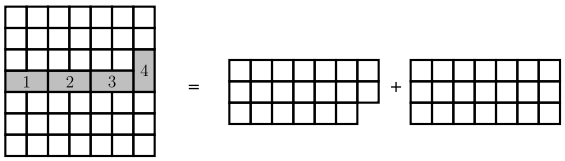


FIGURE 12.5 – Un découpage minimal du plateau de taille 7×7.

Il est bien sûr souhaitable d'explorer en priorité la partie de l'arbre de jeu où ces découpages se produisent le plus rapidement. Pour cela, il est nécessaire de définir un critère permettant de trier les positions et de donner la priorité à celles qui semblent plus faciles à découper que les autres. La qualité de l'ordre choisi pour favoriser les découpages influence notablement la vitesse des calculs.

1. Voir le paragraphe 12.9.2 pour une définition.

12.4 Représentation

La représentation d'un plateau de Cram de taille $n \times m$ est faite simplement sous la forme d'un tableau de taille $n \times m$, les cases vides étant codées par un zéro, et les cases grises par la lettre `G`. Les algorithmes complexes, comme celui du calcul de la symétrie canonique, ou celui du calcul de l'ensemble des options d'une position, sont effectués sur cette représentation naturelle sous forme d'un tableau. La figure 12.6 montre un exemple de plateau de Cram et la représentation en tableau correspondante.

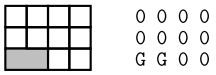


FIGURE 12.6 – Position de Cram et représentation en tableau.

Suivant le contexte, nous avons également besoin d'une représentation sous la forme d'une chaîne de caractères, en particulier pour le stockage dans les bases de données (voir en particulier les explications du chapitre 8, sur les bases de données à la section 8.5, et sur la sérialisation à la section 8.6). On peut représenter un plateau sous forme d'une chaîne de caractères en mettant simplement bout à bout les lignes du plateau. La représentation en chaîne de la position 12.6 serait alors `00000000GG00`.

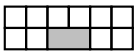


FIGURE 12.7 – Position de Cram à ne pas confondre.

Il se pose cependant une difficulté, car des plateaux de taille différente pourrait conduire à la même chaîne de caractères. Par exemple, la position de la figure 12.7 conduit également à `00000000GG00`, la même chaîne que la position 12.6. Il faut donc une méthode pour indiquer dans la représentation quelle est la taille du plateau. En fait, la longueur des lignes est suffisante, et nous avons choisi dans notre représentation d'indiquer simplement la fin de la première ligne avec le caractère « `*` ». Ainsi, la chaîne représentant la position de la figure 12.6 est `0000*0000GG00` tandis que celle de la figure 12.7 est `000000*00GG00`.

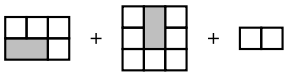


FIGURE 12.8 – Position de Cram contenant plusieurs composantes.

Enfin, du fait de l'existence de positions découpables en sommes de positions indépendantes, il est nécessaire de stocker des listes de plateaux de Cram. Lorsque l'on représente une liste de plateaux sous la forme d'une chaîne de caractères, nous avons besoin d'un caractère terminal pour indiquer la fin de chaque plateau. Nous avons choisi la lettre `E` (abbréviation du mot anglais « end » en anglais). La position de la figure 12.8 est alors représentée par la chaîne `000*GG0E0G0*0G0000E00*E`.

12.5 Canonisation

12.5.1 Symétries

Les positions de Cram sont équivalentes à symétrie près. Dans le cas général, comme sur la figure 12.9, une position donnée possède 8 symétries possibles. La canonisation d'une position

de Cram consiste à choisir un représentant canonique parmi ces 8 possibilités. Un choix possible consiste simplement à prendre la symétrie minimale pour l'ordre lexicographique.

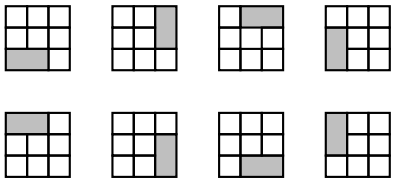


FIGURE 12.9 – Les 8 symétries d'une position de Cram.

L'implémentation naïve des symétries consiste à calculer les 8 chaînes de caractères correspondant aux symétries possibles, puis à les trier selon l'ordre lexicographique, avant de retenir la meilleure. Cela provoque au moins 8 copies de chaînes lors du calcul des symétries, avec 8 parcours complets de la chaîne représentant la position, suivi de plusieurs parcours partiels lors des comparaisons de chaînes. Ces opérations sont très coûteuses, à tel point que le calcul de la symétrie canonique s'est révélé l'une des opérations les plus coûteuses lors des premiers profils des calculs de Cram.

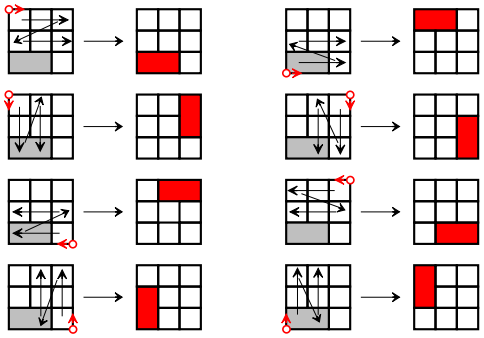


FIGURE 12.10 – Parcours du plateau correspondants aux 8 symétries.

Il y a en fait moyen de calculer la symétrie canonique bien plus efficacement. La figure 12.10 montre comment on peut obtenir les 8 symétries sans avoir besoin de les calculer vraiment, simplement avec des parcours différents du tableau représentant la position. Les flèches sur les plateaux montrent comment parcourir le plateau de gauche pour obtenir la symétrie correspondant au plateau de droite.

Il est commode de visualiser une symétrie comme le choix d'un coin de départ (quatre possibilités) suivi du choix d'un côté partant de ce coin pour l'ordre de parcours (deux possibilités). Nous avons matérialisé ce moyen mnémotechnique par un cercle et une petite flèche sur chacun des plateaux de la figure 12.10.

L'algorithme 17 décrit le calcul rapide de la symétrie canonique d'une position $\mathscr{P}$.

12.5.2 Cases isolées

Les cases isolées ne sont plus utilisables par les joueurs, car un domino nécessite au moins deux cases vides adjacentes pour pouvoir être posé. On obtient donc une position équivalente en grisant toutes les cases isolées. Par exemple, sur la figure 12.11, la position de droite est équivalente à celle de gauche.

Algorithme 17 Calcul de la symétrie canonique d'une position $\mathscr{P}$

1: $n \leftarrow$ nombre de cellules de $\mathscr{P}$
2: **Pour** $i = 1$ à n **faire**
3: lire la i-ème cellule de chacun des parcours de symétrie encore possibles
4: ne garder comme parcours possibles que ceux dont la i-ème cellule est minimale pour l'ordre lexicographique
5: **fin Pour**
6: **Si** le parcours minimal n'est pas celui de la position $\mathscr{P}$ **alors**
7: $\mathscr{P} \leftarrow$ symétrie correspondant au parcours minimal déterminé
8: **fin Si**

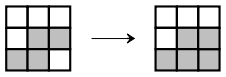


FIGURE 12.11 – Position équivalente en grisant les cases isolées.

12.5.3 Réduction de la taille du plateau

Par ailleurs, il est évident que l'on obtient une position équivalente en supprimant les lignes ou les colonnes entièrement grisées. Par exemple, la position à droite de la figure 12.11, obtenue après grisage des cases isolées, contient une ligne inutile entièrement grise. La figure 12.12 montre la position équivalente obtenue après suppression de cette ligne inutile.

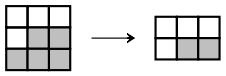


FIGURE 12.12 – Position équivalente en supprimant une ligne inutile.

12.5.4 Algorithme de canonisation

L'algorithme complet de canonisation d'une position $\mathscr{P}$, éventuellement constituée de plusieurs composantes, prend alors la forme de l'algorithme 18.

On notera que les composantes de $\mathscr{P}$ lors de la deuxième boucle peuvent être plus nombreuses que celles de la première boucle. Ce cas se produit si certaines composantes de la première boucle ont pu être elles-mêmes découpées en composantes indépendantes.

12.6 Ordre des positions

L'un des points-clefs de l'efficacité des calculs réside dans un ordre adéquat des positions. La stratégie la plus classique dans les calculs de jeux combinatoires consiste à définir une heuristique pour guider le calcul en priorité vers les positions perdantes. Dans le cas idéal, cela équivaut à diviser par deux la hauteur de l'arbre de recherche lors d'un calcul alpha-bêta. Malheureusement, comme pour le Sprouts, cette stratégie ne marche pas pour le Cram. La nature impartiale du jeu rend très difficile la définition d'une telle heuristique. Nous ne connaissons en fait aucun critère théorique, même approximatif, permettant de dire que telle position a plus de chances que telle autre d'être perdante.

La stratégie pour ordonner les options va donc être plutôt d'orienter le calcul autant que possible vers la partie de l'arbre qui semble la plus facile à calculer. Par exemple, on peut donner la priorité aux plateaux de petite taille (la taille du plateau étant exprimée par le nombre de cases) : plus les plateaux sont petits, plus ils sont faciles à calculer.

Algorithme 18 Canonisation d'une position $\mathscr{P}$

```
1: Pour chaque composante de P faire
2:     griser les cases isolées désormais inutilisables
3:     découper si possible la composante en composantes indépendantes
4: fin Pour
5: Pour chaque composante de P faire
6:     supprimer les lignes ou colonnes inutiles (celles entièrement grises)
7:     calculer la symétrie canonique
8: fin Pour
9: trier les composantes suivant l'ordre lexicographique
```

Nous utilisons les critères suivants (par ordre décroissant de priorité) :

* priorité aux plateaux de petite taille.
* priorité aux positions avec un grand nombre de composantes indépendantes.
* priorité aux positions qui sont « plus symétriques » que les autres.
* priorité aux positions qui semblent plus faciles à découper.
* priorité aux positions avec des cases utilisées au centre.
* ordre lexicographique sur la représentation en chaîne.

12.6.1 Priorité aux découpages

La priorité aux positions avec un grand nombre de composantes indépendantes permet de favoriser les découpages quand ceux-ci se produisent. Cependant, cela permet uniquement de favoriser les positions découpées par rapport aux positions non découpées parmi les options d'une position donnée. Cela ne permet pas de jouer successivement des coups qui vont conduire à des découpages.

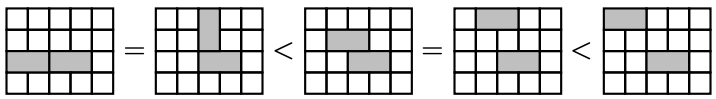


FIGURE 12.13 – Longueur de la ligne de coupe minimale de différentes positions.

Pour jouer successivement des coups qui mènent à un découpage, il faut une priorité supplémentaire, plus complexe. Un critère possible pour évaluer si la position est facile ou non à découper est de calculer la *longueur de coupe minimale* : on compte le nombre de cases vides sur chaque ligne et chaque colonne, et l'on retient la plus petite valeur.

La figure 12.13 montre la longueur de la ligne de coupe minimale de plusieurs positions de Cram. Pour chaque position, il existe une ligne ou une colonne dont le nombre de cases vides correspond au nombre indiqué en dessous.

Cet ordre permet de jouer des coups qui réduisent petit à petit la longueur de la ligne de coupe minimale, et donc de se diriger rapidement vers les parties de l'arbre de jeu où les découpages en composantes indépendantes se produisent.

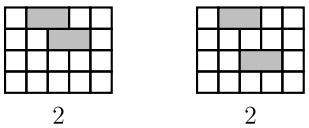


FIGURE 12.14 – Positions avec une même longueur de coupe.

Cependant, cet ordre pour favoriser l'apparition des découpages pourrait être amélioré. Par exemple, les deux positions de la figure 12.14 ont toutes les deux une longueur de coupe minimale en ligne droite de 2. Mais les deux positions ne se découpent pourtant pas aussi facilement l'une que l'autre : la position de gauche peut être découpée en deux composantes en un seul coup, alors que la position de droite nécessite de jouer deux coups.

Ce problème vient du fait que le simple fait de compter les cases vides le long d'une ligne ou d'une colonne ne tient pas compte de la dispersion éventuelle de ces cases vides. Une amélioration de l'ordre consisterait à donner la priorité à la position de gauche par rapport à celle de droite.

12.6.2 Priorité aux transpositions

Dans les calculs de Cram que nous avons effectués, nous stockons les positions déjà calculées dans une table de transpositions, de façon à réutiliser les résultats connus chaque fois que l'on rencontre une même position une nouvelle fois. Le calcul est d'autant plus rapide que l'on rencontre les mêmes positions un grand nombre de fois.

Une idée pour accélérer les calculs est alors d'augmenter le nombre de transpositions dans l'arbre de jeu grâce à un ordre de parcours adéquat. Cette idée a par exemple été utilisée avec succès par Nathan Bullock, sur le jeu de Domineering [10] p. 38, en favorisant les positions symétriques. De façon générale, le nombre de transpositions a tendance à augmenter dès lors que l'on explore préférentiellement une partie seulement de l'arbre de jeu. Cela peut être réalisé en donnant systématiquement la préférence à certains types de positions plutôt qu'à d'autres.

Dans le cas du Cram, nous avons retenu deux types de priorité basées sur cette idée. D'une part, une priorité aux positions qui sont « plus symétriques » que les autres, et d'autre part, une priorité aux positions avec un nombre élevé de cases occupées au centre du plateau.

Dans chacun de ces deux cas, nous définissons un *degré de symétrie* et un *degré d'occupation centrale*. Par exemple, pour obtenir le degré de symétrie, nous calculons les différentes symétries de la position et accordons d'autant plus de points à la position que ses cases conservent la même couleur (blanche ou grise) pour les différentes symétries.

L'algorithme précis pour calculer le degré de symétrie ou le degré d'occupation centrale n'est pas vraiment important. On peut imaginer de nombreuses variantes, sans raison particulière au niveau théorique d'en choisir une plutôt qu'une autre. Les variantes ont souvent une influence positive dans certains cas et négative dans d'autres, si bien qu'il est difficile de faire un choix définitif. L'important est surtout de définir des critères, quels qu'ils soient, qui dirigent les calculs préférentiellement vers certaines parties de l'arbre de jeu.

La dernière priorité avec l'ordre lexicographique relève d'ailleurs du même principe : l'ordre lexicographique permet en dernier recours d'explorer préférentiellement une certaine partie de l'arbre de jeu.

12.7 Calculs des arbres canoniques

Le calcul des arbres canoniques permet d'obtenir de nombreuses informations sur certaines positions de départ. Tout d'abord, il s'agit d'un calcul exhaustif de l'arbre de jeu de ces positions. En ce sens, il s'agit donc d'une résolution forte, au sens décrit dans le paragraphe 2.7.1 du chapitre d'introduction. Par ailleurs, les calculs d'arbres canoniques permettent d'obtenir des informations sur la complexité spatiale réelle du jeu, et sur la qualité de la canonisation utilisée par le programme.

12.7.1 Résolution forte

Le tableau 12.1 montre les nombres de positions, d'arbres canoniques et d'arbres canoniques réduits contenus dans l'arbre de jeu de différents plateaux $3 \times n$, et le tableau 12.2 montre les valeurs obtenues pour des plateaux de départ de dimensions supérieures.

plateau	positions	arbres canoniques	arbres canoniques réduits
3×2	6	5	3
3×3	14	6	3
3×4	73	24	6
3×5	292	61	5
3×6	1222	286	64
3×7	5011	1070	170
3×8	20445	4152	1191
3×9	83418	14889	5145

TABLE 12.1 – Nombre d'arbres canoniques différents obtenus à partir de plateaux $3 \times n$.

plateau	positions	arbres canoniques	arbres canoniques réduits
4×4	304	92	21
4×5	3749	874	205
4×6	29300	6401	2136
5×5	32221	7540	1780

TABLE 12.2 – Nombre d'arbres canoniques différents obtenus à partir de plateaux de grande taille.

Nous avons pu calculer l'arbre canonique des plateaux de taille 3×9 et 5×5, qui étaient les plus grands plateaux calculés précédemment en version misère par Martin Schneider.

12.7.2 Complexité spatiale du Cram

Le nombre d'arbres canoniques peut être considérée comme la vraie complexité spatiale d'un plateau de jeu, dans le sens où toutes les branches redondantes de l'arbre de jeu ont été éliminées.

La figure 12.15 réunit les résultats des deux tableaux 12.1 et 12.2, en indiquant le nombre d'arbres canoniques en fonction du nombre de cases du plateau.

On en déduit une loi exponentielle approximative (tracée sur la figure) du nombre d'arbres canoniques en fonction du nombre n de cases du plateau :

$$0,075 \times 1{,}58^n$$

Cette loi est à comparer avec une analyse naïve du nombre de positions différentes. Si l'on considérait uniquement les 8 symétries possibles d'un plateau de jeu, le nombre de positions différentes d'un plateau de n cases serait de $0,125 \times 2^n$, que nous avons tracé en pointillés sur la figure. L'écart entre ces deux lois est important car l'échelle des ordonnées est logarithmique.

12.7.3 Qualité de la canonisation

En comparant maintenant le nombre de positions canonisées et le nombre d'arbres canoniques, on peut évaluer la qualité de la canonisation des positions. Si la canonisation décrite

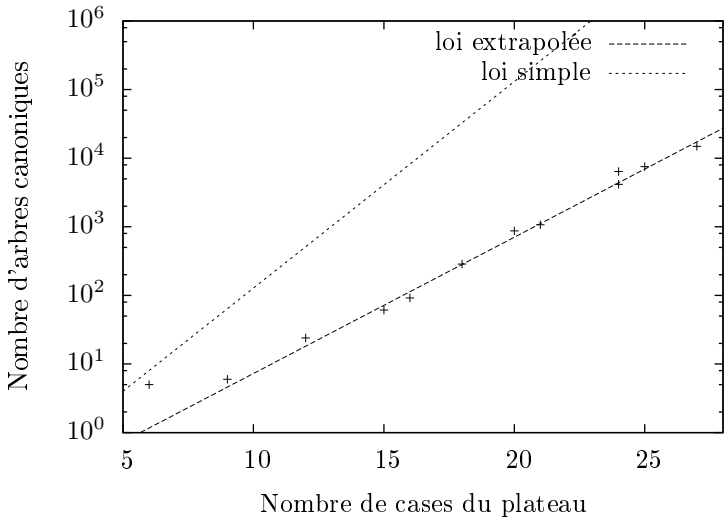


FIGURE 12.15 – Nombre d'arbres canoniques en fonction du nombre de cases du plateau (échelle logarithmique).

dans la section 12.5 était parfaite, le nombre de positions canonisées serait égal au nombre d'arbres canoniques.

La figure 12.16 montre le nombre de positions canonisées en fonction du nombre de cases du plateau (en utilisant les données des tableaux 12.1 et 12.2). Nous avons également reporté sur cette figure la droite correspondant au nombre de positions que l'on obtiendrait si l'on ne tenait compte que des symétries (en pointillés) et la droite limite des arbres canoniques si la canonisation était parfaite.

On constate que les opérations de découpages en plateaux indépendants, de remplissage des cases isolées, et de suppression des lignes inutiles sont loin d'être négligeables. On s'est déjà nettement rapproché de la droite limite des arbres canoniques. Cependant, la figure peut être trompeuse : comme l'échelle des ordonnées est logarithmique, le petit écart entre le nombre de positions canonisées et la droite limite des arbres canoniques est en fait assez grand. Il reste une marge d'amélioration importante sur la qualité de la canonisation.

La piste de recherche la plus prometteuse pour améliorer la canonisation est la prise en compte du graphe dual des positions de Cram. Pour obtenir ce graphe, on associe un sommet à chaque case vide, et l'on relie deux sommets par une arête si les cases vides correspondantes sont placées à côté l'une de l'autre. Deux positions qui ont le même graphe dual sont alors équivalentes.

La figure 12.17 présente trois positions qui sont équivalentes car elles ont le même graphe dual.

L'implémentation du graphe dual n'est cependant pas évidente, car il faut définir une représentation du graphe qui ne soit pas trop coûteuse en mémoire, tout en étant capable de facilement reconnaître deux graphes isomorphes, pour éviter l'apparition de trop de positions équivalentes. Une implémentation possible est discutée en annexe C, la théorie « Chocolat-Toile-Forêt », qui peut également s'avérer utile pour le Dots-and-boxes.

D'autres améliorations sont possibles en implémentant les *Crackers*, décrits dans *Winning Ways* [6] p. 502–504, qui permettraient de découper plus rapidement les positions.

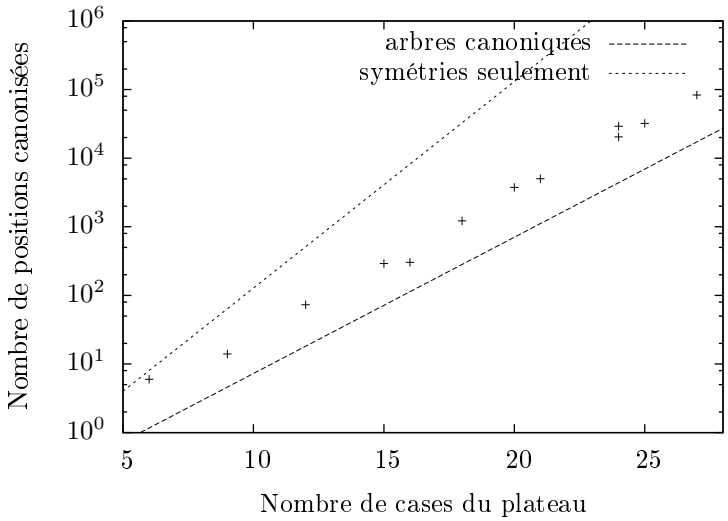


FIGURE 12.16 – Nombre de positions canonisées en fonction du nombre de cases du plateau (échelle logarithmique).

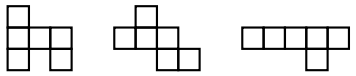


FIGURE 12.17 – Positions de Cram équivalentes.

12.8 Calculs en version normale

Dans les tableaux ci-dessous, nous avons indiqué entre parenthèses les résultats qui ne nécessitent pas de calcul grâce à la stratégie de symétrie. Par ailleurs, nous avons indiqué par un « – « les plateaux $n \times m$ avec $n > m$, car la valeur est identique par symétrie à celle du plateau $m \times n$.

À notre connaissance, les meilleurs résultats connus précédemment étaient ceux de Martin Schneider en 2009 [39] que nous avons indiqués par un astérisque dans les tableaux.

12.8.1 Résultats

3	4	5	6	7	8	9	10
*0	*1	*1	*4	*1	*3	*1	2

11	12	13	14	15	16	17	18
0	1	2	3	1	4	0	1

TABLE 12.3 – Résultats obtenus sur les plateaux de taille $3 \times n$.

Les résultats nouveaux sur les plateaux de taille supérieure à 4 sont donc les nimbers des plateaux 4×7, 4×9, 5×6, et 5×8, et l'issue gagnante des plateaux 5×9 et 7×7. Par ailleurs, l'algorithme de calcul du nimber par incrémentation de la valeur potentielle du nimber permet d'obtenir des résultats partiels. Nous avons pu montrer avec cette méthode que le nimber du plateau 6×7 est strictement supérieur à 3, mais sans parvenir pour l'instant à calculer la valeur exacte.

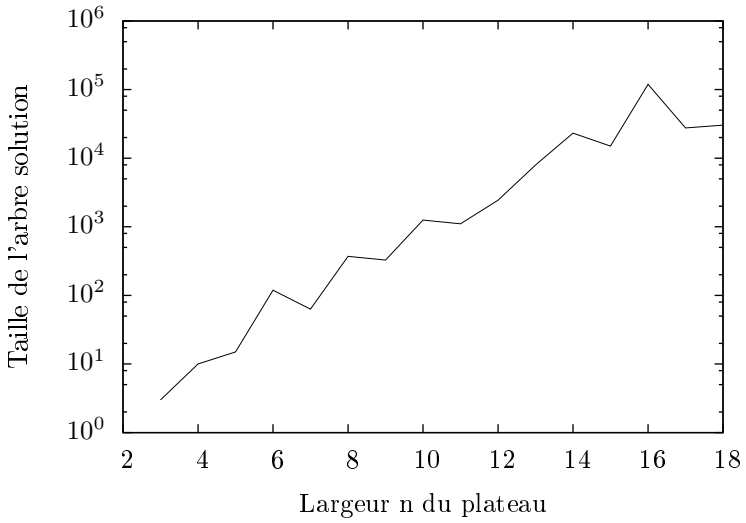


FIGURE 12.18 – Nombre de positions de l'arbre solution du nimber des plateaux $3 \times n$ (échelle logarithmique)

	4	5	6	7	8	9
4	$(\mathbb{0})$	$*2$	$(\mathbb{0})$	$\mathbb{3}$	$(\mathbb{0})$	$\mathbb{1}$
5	–	$*\mathbb{0}$	2	$*\mathbb{1}$	$\mathbb{1}$	W
6	–	–	$(\mathbb{0})$	$> \mathbb{3}$	$(\mathbb{0})$	(W)
7	–	–	–	W	(W)	

TABLE 12.4 – Résultats obtenus sur les plateaux de taille $n \times m$, avec $n \geq 4$ et $m \geq 4$.

12.9 Calculs en version misère

12.9.1 Choix des arbres canoniques réduits

Dans le cas de la version misère, nous avons appliqué les techniques du chapitre 4, avec les arbres canoniques réduits. Les calculs en version misère sont constitués de deux grandes étapes : la première étape consiste à calculer l'arbre canonique réduit d'une position de départ bien choisie. Ce calcul d'arbre canonique réduit implique de calculer les arbres canoniques réduits de toutes les positions apparaissant dans l'arbre de jeu de cette position de départ, et il n'est donc possible que pour des positions d'assez petite taille.

Cette position de départ pour laquelle on calcule l'arbre canonique réduit doit être choisie de façon à ce que les positions de son arbre de jeu apparaissent fréquemment dans les calculs que l'on souhaite faire lors de la deuxième étape. Dans le cas des calculs de Sprouts misère, la position choisie était celle à 6 points de départ, dont les sous-positions apparaissent très fréquemment dans toutes les positions de Sprouts.

Dans le cas des calculs de Cram misère, nous avons choisi de distinguer les plateaux de taille $3 \times n$ et les autres, car il y a peu de positions communes entre des plateaux de taille trop différente. Les tableaux 12.1 et 12.2 de la section précédente ont montré les calculs réalisés d'arbres canoniques et d'arbres canoniques réduits. Nous avons retenu l'arbre canonique réduit du plateau de taille 3×8 pour effectuer les calculs misère des plateaux de taille $3 \times n$ et l'arbre canonique réduit du plateau de taille 5×5 pour les calculs de plateaux de grande taille.

12.9.2 Valeur de Grundy misère

La *valeur de Grundy misère* est définie dans *ONAG* [12] p. 140, ou dans *Winning Ways* [6] p. 422.

Définition 15. *La valeur de Grundy misère d'une position $\mathscr{P}$ est l'unique colonne de Nim $\mathbb{n}$ telle que $\mathscr{P} + \mathbb{n}$ soit perdante.*

L'existence et l'unicité de la valeur de Grundy misère découlent de la caractérisation suivante, qui permet de calculer récursivement cette valeur.

Proposition 16. *La valeur de Grundy misère d'une position terminale est $\mathbb{1}$, et la valeur de Grundy misère d'une position non terminale $\mathscr{P}$ est le* mex *des valeurs de Grundy misère des options de $\mathscr{P}$.*

La seule différence avec le nimber est que la valeur de Grundy misère d'une position terminale est $\mathbb{1}$ au lieu de $\mathbb{0}$. Et comme pour le nimber, en dépit de la proposition 16, il n'est pas nécessaire de calculer tout l'arbre de jeu de la position $\mathscr{P}$ pour calculer sa valeur de Grundy misère : il suffit de calculer l'issue de $\mathscr{P} + \mathbb{0}$, $\mathscr{P} + \mathbb{1}$, $\mathscr{P} + 2$... jusqu'à ce que l'on trouve celle qui est perdante.

Dans les résultats qui suivent, nous donnons les valeurs de Grundy misère de certaines positions, ce qui est plus précis que la simple donnée de leur issue. La raison en est que Martin Schneider [39], qui avait auparavant mené des calculs sur le Cram, avait lui aussi calculé ces valeurs. Refaire ces calculs nous a permis de vérifier que nous trouvions les mêmes résultats.

12.9.3 Résultats

Comme pour la version normale du jeu, les meilleurs résultats connus précédemment en version misère étaient ceux de Martin Schneider en 2009 [39], indiqués par un astérisque dans les tableaux.

3	4	5	6	7	8	9	10	11	12	13	14	15
$*\mathbb{1}$	$*\mathbb{0}$	$*\mathbb{0}$	$*\mathbb{1}$	$*\mathbb{0}$	$*\mathbb{0}$	$*\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$	$\mathbb{0}$	$\mathbb{0}$	$\mathbb{1}$

TABLE 12.5 – Résultats misère obtenus sur les plateaux de taille $3 \times n$.

De façon surprenante, les plateaux de taille $3 \times n$ se comportent de façon plus régulière en version misère qu'en version normale. On peut conjecturer que la suite des valeurs de Grundy misère est périodique, de période 3.

	4	5	6	7	8	9
4	$*\mathbb{0}$	$*\mathbb{0}$	$*\mathbb{0}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$
5	–	$*2$	$\mathbb{1}$	$\mathbb{1}$		
6	–	–	$\mathbb{1}$			

TABLE 12.6 – Résultats misère obtenus sur les plateaux de taille $n \times m$, avec $n \geq 4$ et $m \geq 4$.

12.10 Conclusion

Le jeu de Cram présente plusieurs caractéristiques qui en font un jeu intéressant à calculer informatiquement. L'existence de positions découpables nécessite d'utiliser la théorie du nimber en version normale pour effectuer des calculs efficaces. Inversement, les découpages en positions indépendantes sont plus difficiles à faire apparaître que dans le jeu de Sprouts.

L'existence de nombreuses positions non découpables limite la puissance de l'utilisation du nimber, et demande de ne pas négliger les ordres de parcours de l'arbre de jeu.

Pour réaliser des calculs efficaces, il faut donc faire appel simultanément à des techniques de plusieurs domaines relativement distincts : informatique pratique bien sûr (canonisation rapide des positions, réutilisation des algorithmes du Sprouts), théorie combinatoire des jeux (nimber, arbres canoniques réduits), intelligence artificielle (méthodes de parcours de jeu).

Il est intéressant aussi de constater que des jeux comme le Cram et le Sprouts, qui semblent très différents à première vue, partagent en fait une théorie commune, et qu'ils peuvent donc être calculés efficacement avec les mêmes algorithmes. Comme pour le Sprouts, ces algorithmes nous ont permis de dépasser les meilleurs résultats connus précédemment, et d'atteindre avec des moyens informatiques raisonnables (simple PC de bureau) des positions qui semblaient inaccessibles auparavant, comme l'issue du plateau 7×7 en version normale.

Dans le cadre de cette thèse, nous avons consacré une partie importante de notre temps à la généralisation des algorithmes du Sprouts pour pouvoir les appliquer à d'autres jeux. Nous avons implémenté relativement peu de théories spécifiques au Cram, et il reste encore une marge notable d'amélioration sur certains points, comme la canonisation ou l'ordre des positions. Les résultats obtenus jusqu'ici seront donc probablement améliorés assez rapidement.

Cependant, la difficulté de calcul des positions de Cram croît nettement plus vite que celle des positions du Sprouts. Les découpages se produisent d'autant plus tardivement que le plateau est grand, limitant d'autant l'efficacité des algorithmes à base de nimber. Il semble probable qu'après une amélioration des résultats grâce à des améliorations spécifiques au jeu de Cram, les calculs se heurtent ensuite à la combinatoire intrinsèque du jeu et qu'il soit difficile d'aller plus loin.

Chapitre 13

Le Dots-and-boxes

13.1 Introduction

13.1.1 Historique

Le *Dots-and-boxes* est un jeu combinatoire, dont la première description connue remonte à 1889. Dans une revue éphémère de sa fabrication, intitulée sobrement « Jeux scientifiques pour servir à l'histoire, à l'enseignement et à la pratique du calcul et du dessin » [1], Édouard Lucas présente un jeu intitulé *Pipopipette*, dont il attribue la paternité à des élèves de l'école Polytechnique. Il présente le cas particulier du jeu à 5×5 cases.

Il semble que le jeu se soit ensuite répandu, on en retrouve des variantes diverses dans le monde entier, où changent le nombre de cases, le fait que les arêtes du bord soit ou non présentes en début de partie, voire la forme des boîtes qui peuvent tout aussi bien être triangulaires ou hexagonales. On peut citer par exemple la boîte de jeu signée MB datée de 1976 : le jeu s'intitule « Les petits carrés » et se joue sur un plateau rectangulaire comportant 9×8 cases.

13.1.2 Règles du jeu

Le Dots-and-boxes est un jeu combinatoire qui se joue sur une grille composée de points. Chacun leur tour, les deux joueurs joignent deux points adjacents par une arête horizontale ou verticale. Dès qu'un joueur complète un carré, il le marque de son initiale, puis rejoue (à moins bien sûr qu'il ne s'agisse du dernier carré de la grille). Il passe son tour dès qu'il ne peut plus compléter de carré. À la fin, le joueur qui a remporté le plus de carrés gagne la partie.

La figure 13.1 présente deux coups joués lors d'une partie entre les joueurs « Gauche » et « Droite ». Au premier coup, Gauche complète un carré, puis rejoue sans compléter de carré. Au deuxième coup, Droite complète deux carrés avec une seule arête, puis rejoue. Au prochain tour, Gauche pourra compléter les 5 carrés restants.

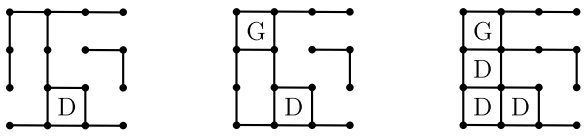


FIGURE 13.1 – Deux coups dans une partie de Dots-and-boxes.

1. C'est dans le numéro suivant de sa revue qu'Édouard Lucas introduisit le problème devenu fameux des Tours de Hanoï.

Le Dots-and-boxes n'est pas un jeu impartial, comme le Sprouts ou le Cram, mais un jeu partisan. Il n'est cependant pas loin d'être impartial, puisque étant donnée une position, les mêmes coups sont disponibles pour les deux joueurs. La seule chose qui l'en différencie est que l'initiale sur chaque carré complété dépend du joueur qui a joué le coup.

13.1.3 Analyses existantes

Le Dots-and-boxes tient une place importante dans *Winning Ways* [6], où un chapitre d'une quarantaine de pages, vraisemblablement dû à Elwyn Berlekamp, lui est consacré. Elwyn Berlekamp qui, en 2000, a publié à part un approfondissement de ce chapitre [5].

La plupart des méthodes décrites dans ces ouvrages sont destinées à des joueurs humains. Elles fonctionnent dans un grand nombre de positions, mais pas systématiquement. Notamment, le fait que le Dots-and-boxes soit presque un jeu combinatoire a permis à Berlekamp de mettre au point une variante du nimber qui permet d'exploiter les sommes de positions, et qui permet de trouver le vainqueur dans un grand nombre de cas. Cependant, la programmation de cette méthode (le « Nimstring ») pour la résolution parfaite du jeu serait probablement assez difficile.

L'analyse informatique la plus poussée dont nous ayons connaissance est due à David Wilson en 2002 [45]. Il s'agit d'une résolution forte, il a calculé l'intégralité des positions qui émergent des jeux à 4×4 et 5×3 carrés. Le plus gros de son travail a ainsi porté sur la compression de ses données. Contrairement au cas d'une résolution faible, l'ordre dans lequel on explore l'arbre de jeu n'a pas d'importance, puisque l'intégralité de l'arbre est de toute façon parcourue.

La force de l'analyse de Wilson est sa simplicité. C'est également sa faiblesse : chaque nouvelle arête sur le plateau multiplie par 2 le nombre de positions qu'il doit étudier, si bien qu'il estime à 2034 la date à laquelle son programme sera capable d'analyser le jeu à 5×5 carrés, en supposant que la loi de Moore sur l'augmentation de la vitesse des microprocesseurs et l'augmentation de la RAM des ordinateurs continue à être valable d'ici-là.

13.1.4 Présentation de notre travail

Après avoir travaillé sur le Sprouts et le Cram, et pour tirer parti de l'architecture modulaire de notre programme, nous avons décidé de mener des calculs sur un autre jeu. C'est ainsi que nous avons entrepris l'étude du Dots-and-boxes, avec pour objectif d'obtenir la résolution faible des plateaux les plus grands possibles. Le travail présenté est le résultat de quatre mois de travail seulement, ce qui montre la facilité avec laquelle nous pouvons adapter notre programme pour l'étude d'un nouveau jeu combinatoire.

L'avantage que procure le fait de se limiter à la résolution faible, par rapport à la résolution forte de Wilson, c'est que nous n'explorons qu'une partie de l'arbre de jeu. La table de transpositions comporte donc moins de positions. Mais cet avantage est à double tranchant : Wilson calcule toutes les positions, sur un jeu de plateau dont la représentation est simple, ce qui rend particulièrement aisé le travail de compression des données. À l'inverse, les positions que nous rencontrons sont plus aléatoires, donc difficiles à compresser.

Comme Wilson, nous avons plafonné sur les plateaux classiques au niveau des tailles 4×4 et 5×3. Cependant, nous avons ramené ces calculs à des tailles plus raisonnables : chacun peut être mené en moins de 24 heures sur un ordinateur de bureau actuel, en ne stockant qu'une dizaine de millions de positions dans la table de transpositions (là où Wilson en stockait plusieurs dizaines de milliards). Il est désormais envisageable de mener un calcul sur plusieurs mois afin d'obtenir la résolution du jeu 5×4. Et ce, d'autant plus que si avec une résolution forte, le passage de 4×4 à 5×4 nécessite de calculer 512 fois plus de positions, avec une résolution faible, les résultats obtenus sur les autres plateaux laissent penser que ce facteur devrait être plus petit, plutôt de l'ordre de 50 (voir §13.8.3).

Enfin, nous avons également résolu un certain nombre de plateaux de tailles diverses, dont certains auraient été impossibles à calculer avec le programme de Wilson. En effet, l'efficacité de son programme est liée directement au nombre d'arêtes potentielles de la position de départ, alors que le nôtre est plutôt lié au nombre de coups potentiellement jouable, ce qui nous procure un avantage sur certains plateaux.

13.2 Terminologie

Dans cette section, nous établissons quelques définitions qui nous seront utiles pour la résolution du Dots-and-boxes.

13.2.1 Représentation des positions

À la place de positions du Dots-and-boxes, on considérera les positions du jeu dual, à savoir le *Strings-and-coins*, pour lequel la visualisation de certaines propriétés est plus aisée. Ce jeu équivalent est obtenu en remplaçant les boîtes par des *jetons*, et en reliant deux jetons par une *arête* si, sur la position correspondante du Dots-and-boxes, l'arête située entre les deux boîtes n'est pas tracée. Dans le cas particulier d'une arête située entre une boîte et l'extérieur de la position, si cette arête n'est pas tracée, on représente la position correspondante de Strings-and-coins avec une *flèche* (représentation héritée de *Winning Ways* [6] et du livre de Berlekamp [5]). La figure 13.2 présente à gauche une position de Dots-and-boxes, et au milieu la position correspondante de Strings-and-coins.

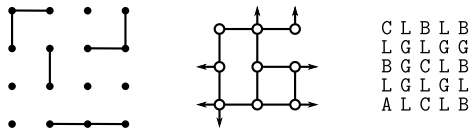


FIGURE 13.2 – Position du Dots-and-boxes, position duale de Strings-and-coins, et représentation en tableau.

Si un coup de Dots-and-boxes consiste à tracer une nouvelle arête, un coup de Strings-and-coins consiste au contraire à en supprimer une. En début de partie de Strings-and-coins, les arêtes relient soit deux jetons entre eux (*arêtes internes*), soit un jeton au bord du plateau (flèches). Remarquons que les arêtes ne changent pas de type au cours de la partie : les flèches restent des flèches, et les arêtes internes restent des arêtes internes.

Les positions seront représentées informatiquement par des tableaux, formés des caractères suivants :

* « `A` », « `B` » et « `C` » représentent un jeton, avec respectivement 2 flèches, 1 flèche ou aucune flèche lui étant raccordées. On ne peut pas dépasser 2 pour une position de Dots-and-boxes classique.
* « `L` » représente une arête interne (« `L` » étant l'abbréviation de *link* en anglais).
* « `G` » représente une case grise, inutilisable.

La figure 13.2 illustre le fait qu'une position du Dots-and-boxes 3×3 est représentée par un tableau de taille 5×5, et, plus généralement, une position du Dots-and-boxes $n \times m$ sera représentée par un tableau de taille $(2n-1) \times (2m-1)$. Cette représentation avec un tableau de $(2n-1) \times (2m-1)$ cases au lieu de $n \times m$ cases, n'est pas la plus compacte possible. Nous l'avons choisie principalement pour permettre une écriture plus facile de certains algorithmes, en particulier ceux communs au jeu de Cram.

13.2.2 Positions de départ

On peut distinguer 3 types de positions de départ (en utilisant la terminologie de Berlekamp) :

* *américaine*, la plus classique, où les arêtes extérieures ne sont initialement pas tracées dans le cas du Dots-and-boxes. C'est-à-dire qu'il y a des flèches dans la représentation du Strings-and-coins.
* *suédoise*, où les arêtes extérieures sont initialement tracées pour le Dots-and-boxes. C'est-à-dire que dans la représentation du Strings-and-coins, il n'y a que des arêtes internes, et pas de flèche.
* *islandaise*, où seules les arêtes extérieures du haut et de la gauche sont initialement tracées pour le Dots-and-boxes. C'est le cas intermédiaire.

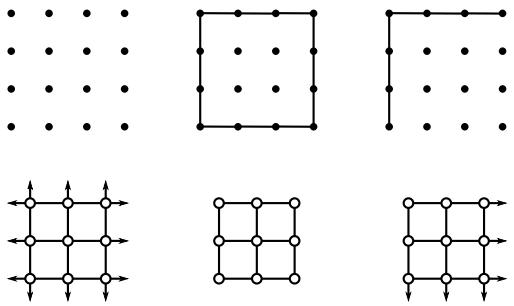


FIGURE 13.3 – Positions de départ américaine, suédoise et islandaise du Dots-and-boxes, et leurs positions duales de Strings-and-coins.

13.2.3 Coups et Tours

Considérant une position de Strings-and-coins, on appelle *coup* la suppression d'une arête. Mais dans un *tour* de jeu, plusieurs coups peuvent être joués, car si un joueur capture un (ou deux) jeton(s) suite à son coup, il doit rejouer.

Détaillons un peu plus à quoi ressemble un tour. On peut distinguer deux types de coups : les coups *capturants*, où l'on capture un jeton, et les coups *non capturants*. Le joueur joue des coups tant qu'il veut (et peut) jouer des coups capturants, et au premier coup non capturant, il passe son tour. Il faut bien préciser que c'est tant qu'il *veut* jouer des coups capturants, car contrairement au jeu de dames, on peut choisir de ne pas capturer un jeton. D'autant plus qu'il existe des positions où il vaut mieux ne pas prendre tous les jetons que l'on peut ; c'est le cas de la position à gauche de la figure 13.1. Gauche a choisi de ne capturer qu'une boîte au lieu de trois. En jouant ainsi, il gagne la partie 6/3, alors qu'en capturant les trois boîtes, il aurait perdu 3/6.

Dans les arbres de recherche que nous allons développer pour déterminer les stratégies gagnantes du Dots-and-boxes, chaque sommet correspondra à une position, et chaque arête, à un tour de jeu. Cette implémentation est plus économe qu'une implémentation où chaque arête correspondrait à un coup.

Dans l'étude des fins de parties, dès lors qu'il y a des jetons que l'on peut capturer, il faudrait donc lister toutes les façons possibles de prendre un certain nombre de jetons, puis d'enfin jouer un coup où l'on ne prend pas de jeton. S'il y a beaucoup de jetons que l'on peut capturer dans une position, le nombre de tours possibles est très élevé. Heureusement, le théorème 12 énoncé plus loin permettra de fortement limiter l'explosion combinatoire.

13.2.4 Chaînes

On appelle *chaîne* une suite de jetons et d'arêtes consécutifs. Les chaînes considérées seront supposées *élémentaires*, c'est-à-dire qu'elles ne passent pas deux fois par le même jeton.

FIGURE 13.4 – Ceci n'est pas une chaîne.

La *longueur* d'une chaîne est son nombre d'arêtes. Voici les différents types de chaînes :
* un *cycle* : la chaîne n'a pas d'extrémité. Tous les jetons sont de degré 2.
* une chaîne *fermée* : elle commence et finit par un jeton.
* une chaîne *ouverte* : elle commence et finit par une arête.

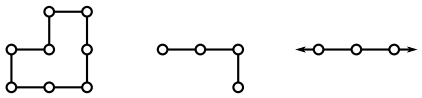


FIGURE 13.5 – Cycle, chaîne fermée, chaîne ouverte.

Remarquons que la longueur d'un cycle est forcément paire et ≥ 4.
On pourra considérer des chaînes *indépendantes*, ou *plongées* dans une position.

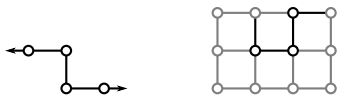


FIGURE 13.6 – Chaînes ouvertes : indépendante, plongée dans une position.

13.2.5 Sommes de positions indépendantes

Le découpage d'une position de Strings-and-coins en positions indépendantes est très facile : les positions indépendantes sont exactement les composantes connexes du graphe formé par la position de Strings-and-coins. Ainsi, chaque coup joué se déroule dans une et une seule de ces composantes.

Cependant, étant donnée une position qui est somme de positions indépendantes, un tour de jeu peut se dérouler dans plusieurs composantes de cette somme. En effet, lorsqu'il rejoue, le joueur peut décider de jouer son coup dans une autre composante de la somme. La théorie classique des sommes de jeux partisans ne s'applique donc pas.

Il est néanmoins utile de programmer ces sommes. Tout d'abord, reconnaître les composantes connexes et scinder les plateaux en conséquence, sans oublier de trier ces composantes, permet d'identifier des positions équivalentes. La figure 13.7 présente une équivalence de ce type.

Nous verrons également au paragraphe 13.4.6 que l'on peut obtenir un résultat partiel permettant d'accélérer le calcul de certaines sommes particulières.

13.3 Score et contrat

Nous présentons dans cette section les notions de *score* et de *contrat*, qui permettent de décrire l'état d'une position de manière plus précise que l'issue (gagnante ou perdante), en

FIGURE 13.7 – Deux sommes de positions indépendantes équivalentes.

spécifiant combien de jetons chaque joueur peut s'assurer de capturer, s'il joue de façon à maximiser ce nombre de jetons.

Il serait possible d'obtenir des résultats équivalents en se limitant à la notion usuelle d'issue, mais cette formulation sous forme de score et contrat nous paraît plus claire et naturelle.

13.3.1 Score théorique

Le *score théorique* d'une position est le nombre maximal de jetons que le joueur dont c'est le tour est sûr de capturer, s'il joue parfaitement. Étant donnée une position à n jetons, le score théorique est donc un nombre s tel que $0 \leq s \leq n$. Selon les cas, on pourra également noter le score $s/n-s$, pour signifier que le deuxième joueur est sûr de capturer $n-s$ jetons en jouant parfaitement [2].

Par exemple, le score théorique de la position suivante est $3/0$, car le premier joueur peut capturer tous les jetons.

Le score théorique de cette nouvelle position est $2/3$:

Le premier joueur, faute de mieux, capture la paire de jetons, puis est obligé de jouer un coup dans la chaîne ouverte. Suite à ce coup, le second joueur capture tous les jetons restants.

Si l'on souhaite déterminer le score de positions plus compliquées à étudier, la proposition suivante énonce comment calculer récursivement le score d'une position à partir de celui de ses options :

Proposition 17. *Le score théorique d'une position $\mathscr{P}$ à n jetons et d'options $\mathscr{F}_i$ se détermine par la formule :*

$$score(\mathscr{P}) = \max_i \{n - score(\mathscr{F}_i)\}$$

La notion d'*option* correspond ici aux tours de jeu.

On pourrait croire suite à cette formule que calculer le score d'une position nécessite de calculer le score de chaque position de son sous-arbre. Nous verrons au paragraphe 13.3.3 que, comme pour le nimber dans un autre contexte (cf chapitre 3), ce n'est pas le cas.

Donnons maintenant en tant qu'illustration quelques scores de positions simples issues du jeu suédois (il n'y a donc pas de flèche, uniquement des arêtes internes).

Proposition 18. *Le score d'une chaîne fermée de longueur n ($n \geq 1$) est $n+1/0$.*

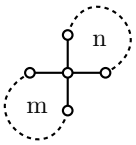
En effet, le joueur dont c'est le tour peut capturer la totalité des jetons durant son tour de jeu.

2. « / » ne représente pas une division !

Proposition 19. *Le score d'un cycle de longueur n est $0/n$.*

Tout coup joué conduit à la formation d'une chaîne fermée de longueur $n-1$, donc le deuxième joueur peut capturer tous les jetons au tour suivant.

Proposition 20. *Le score d'un cycle de longueur n jetons collé à un cycle de longueur m par un sommet est $2/m+n-3$ ($m, n \geq 4$ et sont pairs).*



En effet, le premier joueur est obligé de casser un des deux cycles. Le deuxième joueur en profite et capture tous les jetons de ce cycle, sauf une paire. Le premier joueur n'a d'autre choix que de ramasser la paire, marquant ses deux points, puis de casser l'autre cycle, et le deuxième joueur capture les jetons restants.

13.3.2 Contrat

On peut imaginer de nombreuses implémentations du Dots-and-boxes. Notre objectif est de développer des arbres de recherche de façon à calculer l'issue (*gagnante*, *perdante*, voire *nulle* si la convention est ainsi faite), ou le score théorique d'une position. Ainsi, il est légitime de se demander ce qu'il est pertinent de conserver comme information dans les nœuds de l'arbre de recherche. Nous allons décrire l'implémentation que nous avons choisie avant de préciser ses avantages.

Dans un nœud, nous stockerons 2 paramètres :

* le paramètre « position ».
* le paramètre « contrat ».

Le terme de *contrat* est emprunté au bridge. Étant donné un nœud, il sera gagnant si à partir de sa position, le joueur dont c'est le tour peut remplir son contrat, donc si le score théorique de la position est supérieur ou égal à ce contrat. Sinon, le nœud est perdant. En résumé :

Proposition 21. *Soit une position $\mathscr{P}$ de score s.*

* *Le nœud $(\mathscr{P}; c)$ est gagnant si et seulement si $c \leq s$.*
* *Le nœud $(\mathscr{P}; c)$ est perdant si et seulement si $c > s$.*

Par exemple, soit $\mathscr{P}$ la position de départ 5×2 du jeu suédois, représentée sur la figure 13.8. On peut démontrer que son score théorique est 6, moyennant un calcul non trivial, mais que l'on peut effectuer à la main[3]. Par conséquent, les nœuds $(\mathscr{P}; 1)$ ... $(\mathscr{P}; 6)$ sont gagnants, tandis que les nœuds $(\mathscr{P}; 7)$ ... $(\mathscr{P}; 10)$ sont perdants.

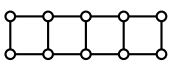


FIGURE 13.8 – Position de score théorique 6/4.

Pour pouvoir développer un arbre de recherche, il faut expliquer comment calculer les fils d'un nœud. La proposition suivante s'en charge simplement :

3. Prévoir quelques dizaines de minutes et un bon café.

Définition 16. *Soit une position $\mathscr{P}$ à n jetons, et soit le contrat c.*
Les fils du nœud $(\mathscr{P}; c)$ sont les nœuds de la forme $(option(\mathscr{P}); n - c + 1)$.

On peut justifier cette formule avec un exemple. Si l'on considère l'arbre de recherche[4] de la figure 13.9, la racine a pour paramètres une position de 12 jetons et un contrat de 8, c'est-à-dire que le joueur dont c'est le tour espère capturer 8 jetons. Pour le faire échouer, l'autre joueur doit en capturer $12 - 8 + 1 = 5$, donc tous les fils de la racine ont pour contrat 5.

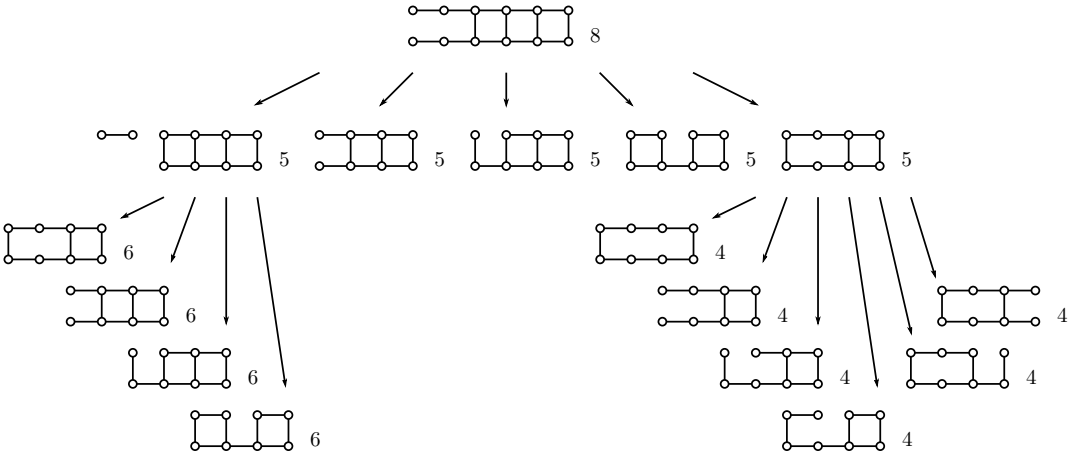


FIGURE 13.9 – Arbre de recherche.

Cette définition des fils d'un nœud permet de disposer de la relation classique énoncée dans la proposition 22.

Proposition 22. ∗ *Un nœud est gagnant s'il a un fils perdant.*
∗ *Un nœud est perdant si tous ses fils sont gagnants.*

Le premier avantage de cette implémentation est qu'elle ne casse pas cette relation, nous pourrons donc utiliser les algorithmes de parcours classiques comme le PN-search.

Le deuxième avantage est que cette implémentation tient compte de la nature quasiment impartiale du Dots-and-boxes : les nœuds sont impartiaux, il n'est pas nécessaire de savoir si c'est le tour du premier ou du deuxième joueur pour étudier un nœud. Ceci va permettre d'augmenter le nombre de transpositions dans l'arbre de recherche, donc d'augmenter la vitesse de calcul, et de diminuer le nombre de nœuds stockés. La figure 13.10 présente un exemple de telle transposition se produisant dans un arbre de recherche.

Enfin, dans les nœuds de l'arbre de recherche, on n'a pas besoin de garder une trace du score de la partie qui se déroule entre la racine et le nœud (une information équivalente étant véhiculée par le contrat). En particulier, dans les positions, on n'a pas besoin de conserver l'information de savoir qui a capturé tel ou tel jeton : une fois un jeton capturé, il disparaît tout simplement. Ceci augmente encore le nombre de transpositions, notamment par rapport à une implémentation calquée sur le jeu papier-crayon, où l'on indiquerait par une lettre si c'est le premier ou le deuxième joueur qui a capturé tel jeton.

13.3.3 Lien entre score et issue

Le score théorique et l'issue (*gagnante* ou *perdante*) sont deux notions distinctes, mais on peut obtenir des résultats équivalents quelles que soit la notion retenue dans l'implémenta-

4. Certaines options sont absentes, elles ont été éliminées par des considérations de symétrie, ou l'application du théorème 12 que nous verrons plus loin.

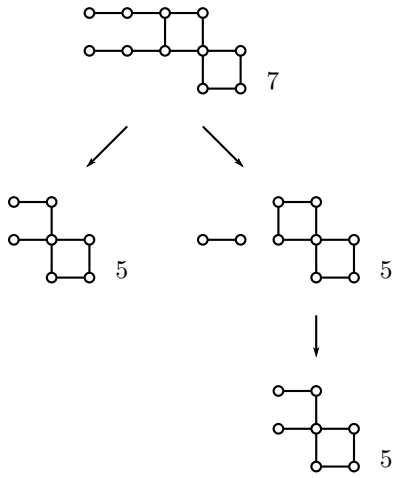


FIGURE 13.10 – Arbre de recherche présentant une transposition.

tion. Nous allons décrire dans cette section les rapports entre ces deux notions.

Commençons par reformuler la proposition 21.

Proposition 23. *Soit $\mathscr{P}$ une position à n jetons, alors :*
- *le nœud $(\mathscr{P};c)$ est trivialement gagnant si $c \leq 0$.*
- *le nœud $(\mathscr{P};c)$ est trivialement perdant si $c > n$.*

De plus, si le score théorique est s (donc $0 \leq s \leq n$) :
- *les nœuds $(\mathscr{P};1)$ ... $(\mathscr{P};s)$ sont gagnants.*
- *les nœuds $(\mathscr{P};s+1)$ ... $(\mathscr{P};n)$ sont perdants.*

Par « trivialement » , on entend qu'il n'est pas nécessaire de mener le moindre calcul. En effet, étant donné une position à 10 jetons, le premier joueur aura du mal à en capturer 13...

Au contraire, la détermination de l'issue des nœuds $(\mathscr{P};k)$ pour $1 \leq k \leq n$ est non triviale. La démonstration du fait que $(\mathscr{P};1)$ est gagnant est généralement assez rapide, car il suffit de prouver que le premier joueur peut capturer au moins un jeton. Dans ce cas, le deuxième joueur n'a pas beaucoup de latitude, et il est donc possible de le démontrer même pour des positions de départ très compliquées.

Puis, la démonstration de $(\mathscr{P};2)$ est un peu plus compliquée que celle de $(\mathscr{P};1)$, celle de $(\mathscr{P};3)$ encore plus... Symétriquement, la démonstration du fait que $(\mathscr{P};n)$ est perdante est facile, celle de $(\mathscr{P};n-1)$ un peu moins... Les plus dures des démonstrations étant celle de $(\mathscr{P};s)$ pour les nœuds gagnants, et celle de $(\mathscr{P};s+1)$ pour les nœuds perdants.

Par exemple, on démontre très facilement que la position $\mathscr{P}$ de la figure 13.11, position à 10 jetons, est de score au plus 9 : quel que soit le coup du premier joueur, le deuxième joueur peut prendre un jeton dès le tour suivant. Par contre, démontrer que son score est au plus 5 nécessite de développer un arbre de recherche beaucoup plus conséquent.

Pour disposer d'un exemple chiffré, nous avons calculé informatiquement l'issue de $(\mathscr{P};c)$, où $\mathscr{P}$ représente la position de départ américaine 3×3, en faisant varier le contrat c entre 0 et 10. Cette position étant de score théorique 3/6, le nœud $(\mathscr{P};c)$ est gagnant si $c \leq 3$, et perdant si $c > 3$. Nous avons mené le calcul avec un algorithme de type depth-first, avant d'utiliser la vérification (décrite dans le chapitre 7) de sorte à optimiser l'arbre solution obtenu. Les résultats sont résumés dans la table 13.1.

Les résultats pour $c = 0$ ou 10 sont les résultats triviaux, c'est pourquoi leur arbre solution ne comporte aucun nœud. Les calculs extrêmes sont $c = 3$ (issue gagnante), et $c = 4$ (issue perdante). À gauche de $c = 3$, la difficulté diminue quand le contrat diminue. À droite de $c = 4$, la difficulté diminue quand le contrat augmente.

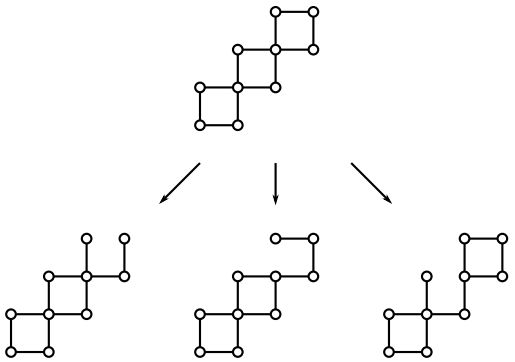


FIGURE 13.11 – Position de score inférieur ou égal à 9.

contrat	0	1	2	3	4	5	6	7	8	9	10
positions	0	241	380	533	1410	884	825	808	673	427	0

TABLE 13.1 – Taille des arbres solutions sur le jeu 3×3 américain, en faisant varier le contrat.

Ainsi, calculer le score s d'une position $\mathscr{P}$ nécessite de calculer l'issue de 2 nœuds : il faut démontrer que le nœud $(\mathscr{P}; s)$ est gagnant, mais aussi que le nœud $(\mathscr{P}; s+1)$ est perdant. Il faut donc trouver deux arbres solutions distincts. Sur l'exemple précédent, il faut calculer les nœuds pour $c = 3$ et 4. Autre exemple, la figure 13.12 montre que le score de la position étudiée est 5.

Ceci nous montre qu'en dépit de la proposition 17, il n'est pas nécessaire de disposer de tout l'arbre de jeu pour connaître le score d'une position.

En général, un joueur de Dots-and-boxes ne souhaite déterminer que l'issue d'une position, et non pas son score, l'objectif étant de battre l'adversaire. Ainsi, pour démontrer que la position 3×3 américaine, qui comporte 9 jetons, est perdante, il suffit de démontrer que son score est strictement inférieur à 5, c'est-à-dire que le nœud $(\mathscr{P}; 5)$ est perdant. Son score exact est plus difficile à obtenir. Pour cela, il faut non seulement démontrer que le nœud $(\mathscr{P}; 4)$ est perdant, ce qui est plus dur que pour $(\mathscr{P}; 5)$, mais aussi que le nœud $(\mathscr{P}; 3)$ est gagnant.

Enfin, il reste le cas où le nombre n de jetons est pair, et où le score est $\frac{n}{2}/\frac{n}{2}$, lorsque chaque joueur peut s'assurer de capturer la moitié des jetons. L'issue du match dépend alors de la convention utilisée. Dans *Winning Ways* [6] (p. 547), Berlekamp choisit de déclarer le deuxième joueur gagnant, pour contrer l'avantage qu'a le premier joueur de pouvoir choisir le premier coup. Mais rien n'interdit de décider que ce serait un match nul.

13.4 Coloriages

L'analyse développée dans cette section se concentre sur la capture des jetons, qui peut conduire un tour de jeu à être composé de plusieurs coups. L'objectif est d'éliminer le plus possible d'options équivalentes ou dominées, pour éviter que ces options surnuméraires ne saturent les arbres de recherche.

13.4.1 Coloriage de la position

On appelle *jeton-feuille* un jeton relié à une seule arête (arête interne ou flèche). C'est donc un jeton que l'on peut capturer au coup suivant, en supprimant l'arête qui lui est

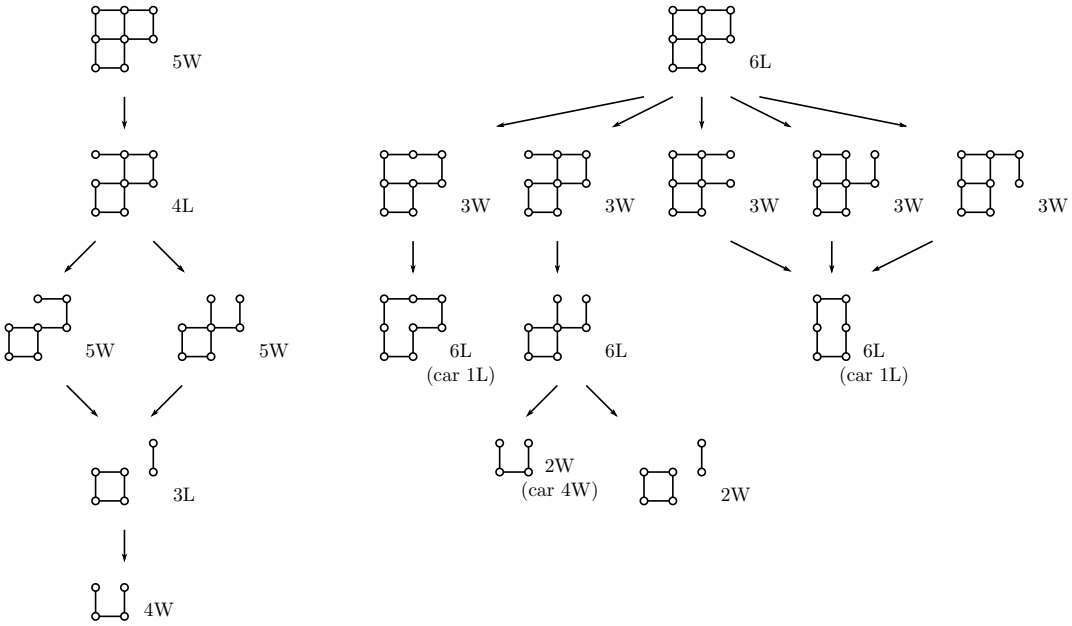


FIGURE 13.12 – Arbres solutions prouvant que le score est 5.

reliée. Et réciproquement, les seuls jetons que l'on peut capturer au coup suivant sont les jetons-feuilles.

Étant donnée une position de Strings-and-coins, nous allons lui appliquer l'algorithme de coloriage 19.

Algorithme 19 Coloriage blanc-noir

1: **Tant que** il reste des jetons-feuilles dans la position **faire**
2: supprimer tous les jetons-feuilles, ainsi que toutes les arêtes qui leur sont reliées.
3: **fin Tant que**

Revenant à la position initiale, les jetons et les arêtes qui n'ont pas été supprimés par l'algorithme 19 sont alors coloriés en *noir*, et ceux qui ont été supprimés sont coloriés en *blanc*. La figure 13.13 présente quelques exemples de coloriages de positions.

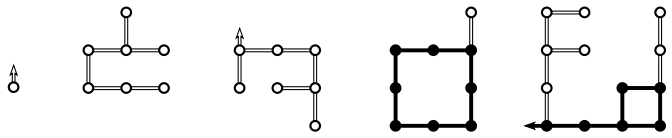


FIGURE 13.13 – Coloriage en blanc et noir de diverses positions.

La position est ainsi décomposée en deux parties disjointes par l'algorithme 19, que nous appellerons *parties blanche, et noire*.

Pour simplifier les raisonnements ultérieurs, nous considérerons que la pointe de chaque flèche est reliée à un même jeton spécial, le *sol*. Le sol ne pouvant être capturé, il est noir.

Les composantes connexes de la partie blanche sont des arbres [5], que nous appellerons *arbres blancs*. Un arbre blanc est relié à au plus un élément noir : un jeton noir, via une

5. Pas stricto sensu pour les arbres liés : ce sont des arbres si l'on leur rajoute le sommet auquel ils sont liés.

arête interne blanche, ou le sol, via une flèche blanche. Si c'est le cas, on parle d'arbre blanc *lié*, par opposition à un arbre blanc *libre*. Un arbre blanc libre est constitué uniquement de jetons et d'arêtes internes blancs. Sur la figure 13.13, on observe de gauche à droite :

* Un arbre blanc réduit à un jeton, lié au sol.
* Un arbre blanc libre.
* Un arbre blanc lié au sol.
* Un arbre blanc réduit à un jeton, lié à un jeton noir.
* Deux arbres blancs, chacun lié à un jeton noir différent.

L'intérêt de l'algorithme 19 réside dans la proposition suivante.

Proposition 24. *Les jetons blancs sont ceux qu'il est possible de capturer au tour suivant, tandis que les jetons noirs sont ceux que l'on ne peut pas capturer durant le tour suivant, quels que soient les coups joués.*

On appelle *coup blanc* la suppression d'une arête blanche, et *coup noir* la suppression d'une arête noire. Un coup noir ne permet pas de capturer de jeton, et signifie donc la fin du tour de jeu. Un coup blanc, en revanche, peut conduire ou non à la capture d'un jeton, suivant le moment où il est joué durant le tour de jeu.

13.4.2 Théorème des jetons blancs

Nous venons de voir qu'il est possible de capturer tous les jetons blancs durant un tour de jeu. On pourrait penser que la meilleure façon de jouer le tour implique justement de tous les capturer. Cependant, il est alors nécessaire de rejouer un coup noir, et, pour certaines positions, jouer un coup noir peut conduire à la défaite.

C'est le cas par exemple sur la figure 13.14. Si l'on capture tous les jetons blancs, on perd 2/6. Inversement, prendre la flèche et laisser les jetons blancs à l'adversaire conduit à une victoire 6/2.

FIGURE 13.14 – Pour gagner ici, il ne faut pas capturer tous les jetons blancs.

Il est ainsi parfois nécessaire de ne pas capturer tous les jetons blancs. Le théorème ci-après exprime que le tour optimal consiste soit à prendre tous les jetons blancs (et donc à jouer ensuite un dernier coup qui sera noir), soit à prendre « presque » tous les jetons blancs, de sorte qu'au tour suivant, l'adversaire soit forcé de jouer un coup noir.

Théorème 12 (des jetons blancs)**.** *Étant donnée une position de Strings-and-coins, le meilleur tour de jeu possible est de l'un des deux types suivants :*

* *type A : « capturer tous les jetons blancs, puis jouer un coup noir ».*
* *type B : « capturer le maximum de jetons blancs possible, sans jouer de coup noir ».*

Commençons par décrire plus précisément les tours de type B.

1. S'il existe quelque part dans la figure un arbre blanc lié contenant au moins 2 jetons, alors, partant de l'élément noir lié à l'arbre (le sol ou un jeton noir), on va s'intéresser aux deux premiers jetons blancs que l'on rencontre dans l'arbre, une *paire*[6]. Les deux cas sont présentés sur la figure 13.15.

   Le tour qu'il faut retenir consiste à commencer par prendre tous les jetons blancs de la position, sauf la paire. Puis le dernier coup consiste à supprimer soit la flèche, soit

6. Si plusieurs paires conviennent, on choisit n'importe laquelle d'entre elles.

l'arête reliant la paire au jeton noir, de sorte qu'il ne reste que la paire de jetons blancs isolée à la fin du tour. Au tour suivant, l'adversaire commencera par prendre la paire blanche pour s'assurer au moins ces 2 points, puis il jouera forcément un coup noir, les seuls coups encore disponibles étant de ce type.

FIGURE 13.15 – Paires de jetons blancs reliées au sol ou à un jeton noir.

2. Sinon, les jetons blancs reliés aux jetons noirs ou aux flèches sont « isolés », comme dans la figure 13.16. Les autres jetons blancs de la position appartiennent à des arbres blancs libres. Là encore, il va falloir distinguer deux cas.

FIGURE 13.16 – Jetons blancs isolés.

   (a) S'il existe un arbre blanc libre contenant une chaîne fermée à 4 jetons, comme dans la figure 13.17, alors le tour à retenir consiste à capturer tous les jetons blancs de la figure, sauf ceux de la chaîne. Puis, on supprime l'arête centrale de la chaîne, ne laissant que 2 paires de jetons blancs à l'adversaire au tour suivant. Là encore, il commencera par prendre ces 2 paires pour s'assurer ces 4 points, avant de jouer un coup noir.

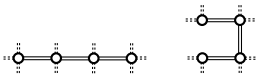


FIGURE 13.17 – Chaînes fermées à 4 jetons blancs, plongées dans des arbres blancs.

   On peut remarquer que les seuls arbres blancs libres ne contenant pas de chaîne fermée à 4 jetons sont (à symétrie près) ceux de la figure 13.18.

   (b) Sinon, c'est qu'il est impossible de « rendre la main » à l'adversaire. Tout tour de jeu débouche sur un coup noir, et donc le meilleur tour de jeu est forcément de type A.

En résumé, les seuls tours qu'il est utile de considérer sont les tours de type A, aussi nombreux que la position comporte d'arêtes noires, et éventuellement un unique tour de type B. Ce tour de type B n'existe que si la position contient soit un arbre blanc lié comportant au moins 2 jetons blancs, soit un arbre blanc libre contenant une chaîne fermée à 4 jetons.

Nous pouvons maintenant démontrer le théorème.

Démonstration. On a éliminé 2 types de tours.

Le premier type de tour éliminé est $\mathscr{T}_1$: « capturer une partie seulement des jetons blancs, puis jouer un coup noir $\mathscr{C}$ ». Or ce tour est dominé par $\mathscr{T}_1'$, le tour de type A suivant : « capturer tous les jetons blancs, puis jouer le même coup noir $\mathscr{C}$ ». En effet, si la meilleure réponse à $\mathscr{T}_1'$ consiste à jouer vers la position $\mathscr{P}$, alors on a la possibilité de répondre le tour suivant à $\mathscr{T}_1$: « capturer les jetons blancs restants, puis jouer vers la position $\mathscr{P}$ ». Ainsi, on peut renvoyer l'adversaire vers exactement la même position, sauf que l'on aura un meilleur score. On dispose ainsi d'une meilleure réponse à $\mathscr{T}_1$ qu'à $\mathscr{T}_1'$.

Le deuxième type de tour éliminé est $\mathscr{T}_2$: « capturer un nombre non maximal de jetons blancs, sans jouer de coup noir ». Ce type de tour est évidemment dominé par l'unique tour de type B. En effet, la réponse optimale au tour de type B consiste à prendre la (ou les)

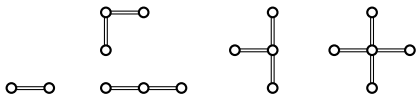


FIGURE 13.18 – Arbres blancs libres ne contenant pas de chaîne de 4 jetons.

paire(s) que le joueur a laissées, puis à jouer un coup noir. Or, on obtient une position plus favorable si l'on répond à $\mathscr{T}_2$ en capturant tous les jetons blancs restants, puis en jouant le même coup noir, puisque l'on aboutit à la même position mais avec un meilleur score. □

L'intérêt de ce théorème est évident si l'on se réfère à la figure 13.9. L'arbre de recherche a été tracé en utilisant le théorème, si bien que la racine n'a que 5 fils, alors que sans le théorème, elle en aurait 51. L'implémentation de ce théorème ne fait pas débat. Une fois programmé, le temps de calcul comme l'espace de stockage ont été divisés par un facteur 5 à 10, même sur de petits calculs.

Enfin, remarquons que l'implémentation de ce théorème implique que certaines positions ne pourront pas être rencontrées lors de nos calculs. C'est le cas des positions numéro 2, 3 et 5 de la figure 13.13.

13.4.3 Coup noir

Dans la partie noire d'une position, toute arête noire appartient à une unique chaîne. Si la chaîne n'est pas un cycle, les arêtes aux extrémités de la chaîne sont reliées à des *joints*, c'est-à-dire des jetons de degré 3 ou 4, ou le sol. En effet, les extrémités ne peuvent pas être des jetons de degré 1, car alors la chaîne serait blanche, ni de degré 2, car alors la chaîne se prolongerait au-delà.

Une position se décompose ainsi en trois parties disjointes : la partie blanche, les joints noirs, et les chaînes noires.

Soit une chaîne noire de longueur L, elle appartient à un de ces trois types de chaînes :

* un *cycle libre* : la chaîne ne comporte pas d'extrémité. Elle comporte donc L arêtes internes et L jetons internes. L est forcément pair et ≥ 4.
* un *cycle lié* : les deux extrémités de la chaîne sont un seul et même joint (autre que le sol). Elle comporte donc L arêtes internes, $L-1$ jetons internes, puis une extrémité, le joint. L est forcément pair et ≥ 4.
* une *chaîne ouverte* : les deux extrémités de la chaîne sont deux joints différents (dont éventuellement le sol), ou deux fois le sol. Elle comporte donc L arêtes (les arêtes extrêmes pouvant être des flèches), et $L-1$ jetons internes.

À titre d'exemple, nous avons représenté sur la figure 13.19 la partie noire d'une position. Les joints sont représentés en noir, et les chaînes en gris. On peut dénombrer :

* 6 joints.
* un cycle libre de longueur 6.
* un cycle lié de longueur 4.
* cinq chaînes ouvertes de longueur 1.
* deux chaînes ouvertes de longueur 2.
* une chaîne ouverte de longueur 3.
* une chaîne ouverte de longueur 4.

Suite à un coup noir, aucun jeton n'est capturé, mais certains jetons ou arêtes noirs peuvent devenir blancs. Plus précisément, lorsque l'on joue un coup dans une chaîne noire, tous les jetons et arêtes de cette chaîne deviennent blancs. De plus, si cette chaîne est un cycle lié à un jeton de degré 3, ce joint et la chaîne ouverte située de l'autre côté du joint deviennent blancs également.

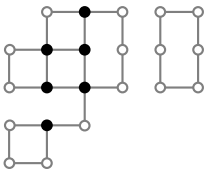


FIGURE 13.19 – Décomposition en joints et chaînes de la partie noire d'une position.

13.4.4 Équivalences lors de l'ouverture d'une chaîne

Nous allons voir avec la proposition 25 que dans la plupart des cas, les coups noirs joués dans une même chaîne sont équivalents, c'est-à-dire qu'ils conduisent à une position de même score. De plus, certains coups noirs sont dominés, c'est-à-dire qu'ils conduisent à une position moins favorable pour le joueur dont c'est le tour que d'autres coups. Ne pas étudier des coups équivalents ou dominés permet de gagner du temps de calcul.

Proposition 25. *1. Étant donnée une chaîne ouverte de longueur 3, les deux coups consistant à supprimer une arête extrême sont dominés par le coup qui consiste à supprimer l'arête centrale de la chaîne.*

2. Pour toute autre chaîne, tous les coups joués à l'intérieur de la chaîne sont équivalents.

Démonstration. 1. Si le premier joueur supprime l'arête centrale, le meilleur coup du second joueur consiste à capturer les deux jetons blancs créés, puis à jouer un coup noir. Il ne peut rendre la main à l'adversaire.

Maintenant, si le premier joueur supprime une des deux autres arêtes, le second joueur peut là encore capturer les deux jetons blancs, puis jouer un coup noir. Mais il peut aussi ne pas capturer la paire de jetons blancs, laissant au premier joueur l'obligation de jouer un coup noir. Il a donc plus de possibilités dans ce second cas, son score est donc au moins aussi bon que dans le premier cas.

2. Si la chaîne est de longueur 1, le cas est trivial. Si elle est de longueur 2, le tour du second joueur commence par la capture du jeton blanc créé, ce qui renvoie à la même situation quel que soit le coup choisi parmi les deux possibles. Si la chaîne est de longueur $L \geq 4$, quel que soit le coup choisi dans la chaîne, le tour du second joueur commence par la capture de $L-3$ jetons, puis il est toujours possible, soit de capturer les deux jetons restants puis de jouer un coup noir, soit de ne pas capturer les deux jetons restants, pour forcer le premier joueur à jouer un coup noir.

□

Nous illustrons l'implémentation de ces équivalences avec la figure 13.20, qui présente une amélioration de l'arbre de recherche de la figure 13.9. Par rapport à cet ancien arbre, le troisième fils de la racine, le troisième fils du premier fils de la racine, et le cinquième fils du cinquième fils de la racine ont disparu, car ces options étaient dominées.

De plus, nous avons modifié certains nœuds, en supprimant les jetons blancs qui sont obligatoirement capturés au tour suivant (en vertu du théorème des jetons blancs), et en mettant à jour le contrat de ces nœuds en conséquence. Cette modification a pour effet de fusionner les nœuds qui correspondent à des ouvertures de chaînes équivalentes.

Comme avec le théorème des jetons blancs, l'implémentation de ces équivalences permet un gain suffisamment net (de l'ordre d'un tiers en moins pour l'espace, et de deux tiers en moins pour le temps) pour ne pas avoir à remettre en cause le bien fondé de leur utilisation.

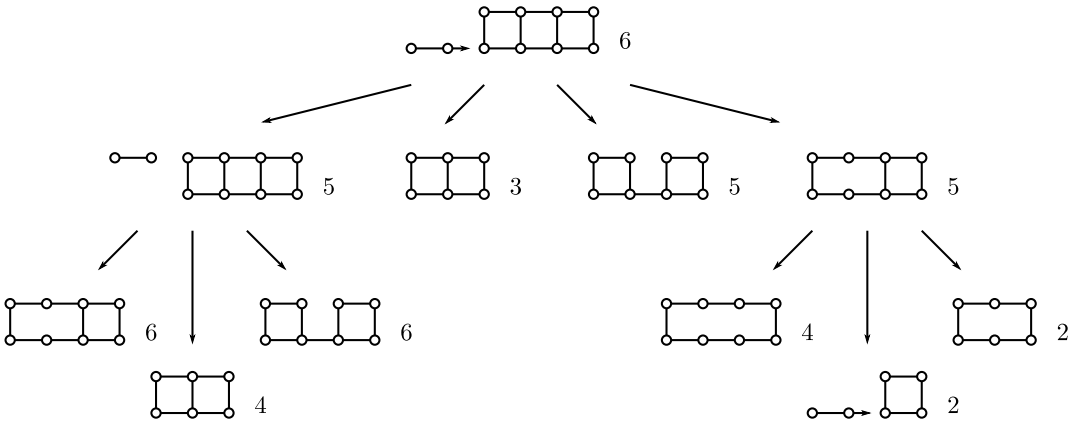


FIGURE 13.20 – Arbre de recherche simplifié.

13.4.5 Résolution ultra-faible

Si l'on se trouve dans une position à partir de laquelle on peut jouer un tour de type B, c'est-à-dire que l'on peut choisir de jouer ou non un coup noir, alors le résultat suivant va exprimer que l'on est assuré de capturer au moins la moitié des jetons présents sur le plateau. L'adversaire a donc été mal inspiré [7] au tour précédent, à tel point que Berlekamp dit dans [5] p. 24 qu'il vient de jouer un « loony move » (un coup de cinglé).

Théorème 13. *Soit $\mathscr{P}$ une position à n jetons. Si à partir de $\mathscr{P}$, on peut jouer un tour de type B, pendant lequel on capture k jetons, alors score$(\mathscr{P}) \geq \lceil \frac{n+k}{2} \rceil$*

Démonstration. Rappelons qu'à partir de $\mathscr{P}$, on peut jouer deux types de tours de jeu :

- type A : capturer tous les jetons blancs, puis jouer un coup noir.
- type B : ne capturer que k jetons blancs, en ne laissant qu'une ou deux paires de jetons blancs à l'adversaire, suivant les cas. L'adversaire est alors obligé de jouer un coup noir au tour suivant.

Appelons M le score minimal d'une position obtenue suite à un coup noir, c'est-à-dire, soit après un tour de type A, soit après un tour de type B puis la réponse de l'adversaire. Deux cas se distinguent.

- Si $M \leq \frac{n-k}{2}$, on choisit de jouer un tour de type A à partir de $\mathscr{P}$, celui justement qui permet d'obtenir un score de M. On a ainsi score$(\mathscr{P}) \geq (n - \frac{n-k}{2})$, car l'adversaire pourra prendre au maximum $\frac{n-k}{2}$ jetons sur les n jetons de $\mathscr{P}$.
- Si $M \geq \frac{n-k}{2}$, on choisit de jouer le tour de type B à partir de $\mathscr{P}$. On a ainsi score$(\mathscr{P}) \geq k + \frac{n-k}{2}$, k étant le nombre de jetons capturés lors du tour de type B, et $\frac{n-k}{2}$ le score minimal que l'on est assuré de faire après la réponse de l'adversaire.

Une fois l'expression simplifiée, on obtient dans chaque cas score$(\mathscr{P}) \geq \frac{n+k}{2}$, et donc score$(\mathscr{P}) \geq \lceil \frac{n+k}{2} \rceil$ car le score est un entier. □

Ce résultat est très fort : pour de nombreuses positions où une chaîne vient d'être ouverte, il permet d'affirmer que le joueur dont c'est le tour peut capturer au moins la moitié des jetons. Mais, revers de la médaille, il a deux faiblesses importantes.

La première est qu'il ne permet pas systématiquement de se passer de l'étude de la position $\mathscr{P}$. Il peut être nécessaire de calculer que le score de $\mathscr{P}$ est nettement supérieur à la moitié du nombre de jetons, et dans ce cas, le théorème ne permet pas de trancher. Par exemple, on peut vouloir étudier le contrat 12 pour la position de la figure 13.21. Or, le théorème assure

7. Si tant est qu'il avait le choix...

juste que son score est supérieur ou égal à $\lceil \frac{16+3}{2} \rceil = 10$, il ne permet donc pas de décider si ce contrat peut être rempli.

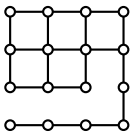


FIGURE 13.21 – Position de score au moins 10.

La deuxième faiblesse est que cette méthode n'est pas constructive : on peut s'assurer la moitié des jetons, mais on ne sait pas comment ! En utilisant cette astuce, on n'opère qu'une résolution *ultra-faible*. On peut ainsi connaître la valeur théorique des positions calculées, mais on ne dispose pas d'une stratégie permettant de jouer parfaitement toute partie à partir de ces positions.

Après implémentation, les performances de cette méthode sont un peu décevantes : pour les positions de départ américaines, entre 4 et 5 % de gain pour la mémoire, et entre 8 et 10 % de gain pour le temps de calcul. Les performances sont légèrement meilleures pour les positions de départ suédoises, car elles engendrent plus d'ouvertures de chaînes. Mais les gains restent limités, et l'on pourrait considérer que le jeu n'en vaut pas la chandelle, puisque l'on ne dispose plus d'une stratégie gagnante une fois le calcul achevé.

Cependant, l'utilisation de la vérification permet, dans un deuxième temps, de « reboucher les trous » générés par cette résolution ultra-faible, et de disposer à nouveau d'une résolution faible sitôt la vérification terminée. Nous avons ainsi pu nous débarrasser des scrupules qui nous retenaient d'utiliser cette méthode, et profiter de ses modestes gains.

13.4.6 Sommes de jetons isolés ou de paires

Nous avons vu que les théories classiques exploitant les sommes de positions indépendantes ne s'appliquent pas au Dots-and-boxes, un tour de jeu pouvant se dérouler dans plusieurs composantes. Dans ce paragraphe, nous présentons tout de même un résultat partiel qui permet de simplifier certaines sommes.

Proposition 26. *On note $\mathscr{J}$ un jeton isolé à deux flèches. Soit $\mathscr{P}$ une position de score a/b. Alors la position $\mathscr{P} + \mathscr{J} + \mathscr{J}$ a pour score $a + 1/b + 1$.*

Démonstration. Le premier joueur dispose d'une stratégie pour marquer au moins $a + 1$ points. Quand c'est son tour, il joue dans $\mathscr{P}$ la stratégie classique qui lui permet de marquer a points, et ce, tant que le deuxième joueur répond dans $\mathscr{P}$. Si soudain, le deuxième joueur décide de jouer dans un des deux jetons $\mathscr{J}$, ce jeton devient un jeton blanc isolé. Le premier joueur doit alors obligatoirement capturer ce jeton, puis il joue dans l'autre jeton $\mathscr{J}$. Le deuxième joueur capture alors l'autre jeton $\mathscr{J}$, puis le jeu classique sur $\mathscr{P}$ reprend.

Enfin, si le jeu dans $\mathscr{P}$ se termine avant que le deuxième joueur ait joué dans un jeton $\mathscr{J}$, le score d'une somme $\mathscr{J} + \mathscr{J}$ est $1/1$, donc le premier joueur marque un point supplémentaire.

La stratégie symétrique permet au deuxième joueur de marquer au moins $b + 1$ points, ce qui permet de conclure que le score théorique de $\mathscr{P} + \mathscr{J} + \mathscr{J}$ est bien $a + 1/b + 1$. □

Des résultats similaires existent avec des paires de jetons, c'est-à-dire deux jetons reliés par une arête interne.

Proposition 27. *On note $\mathscr{P}_1$ une paire de deux jetons à une flèche, $\mathscr{P}_2$ une paire constituée d'un jeton à une flèche et d'un jeton à deux flèches, et $\mathscr{P}_3$ une paire de deux jetons à deux flèches. Soit $\mathscr{P}$ une position de score a/b. Alors :*

∗ *la position $\mathscr{P} + \mathscr{P}_1 + \mathscr{P}_1$ a pour score $a + 2/b + 2$.*

* *la position* $\mathscr{P} + \mathscr{P}_2 + \mathscr{P}_2$ *a pour score* $a + 2/b + 2$.
* *la position* $\mathscr{P} + \mathscr{P}_3 + \mathscr{P}_3$ *a pour score* $a + 2/b + 2$.

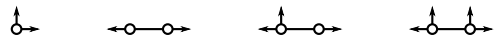


FIGURE 13.22 – Positions de type $\mathscr{J}$, $\mathscr{P}_1$, $\mathscr{P}_2$ et $\mathscr{P}_3$.

Démonstration. L'argument est similaire à celui de la proposition 26. Il faut là aussi vérifier que les sommes $\mathscr{P}_1 + \mathscr{P}_1$, $\mathscr{P}_2 + \mathscr{P}_2$, et $\mathscr{P}_3 + \mathscr{P}_3$ ont pour score $2/2$.

Il est à noter qu'aucun joueur ne devrait commencer à jouer dans $\mathscr{P}_1$ en prenant une arête extrême, ce coup étant dominé par celui qui consiste à prendre l'arête centrale.

On peut préciser que la preuve concernant $\mathscr{P}_2$ utilise les résultats de $\mathscr{J}$ et $\mathscr{P}_1$, et que la preuve concernant $\mathscr{P}_3$ utilise les résultats de $\mathscr{J}$ et $\mathscr{P}_2$. □

L'implémentation de ces simplifications nous a permis de diminuer de plusieurs pourcents la taille de nos tables de transpositions, et de diminuer le temps de calcul en conséquence.

13.5 Déformations

Nous allons présenter dans cette section des techniques de déformation des positions permettant d'identifier des positions équivalentes. Hormis les déformations, l'autre méthode classique pour reconnaître des positions équivalentes est la symétrie, que nous avons bien sûr implémentée. Cela n'a pas été difficile, car les symétries avaient déjà été implémentées pour le jeu de Cram, et cette partie du code est donc commune au Cram et au Dots-and-boxes. Sur un plateau carré, il faut prendre en compte 8 symétries par position, et seulement 4 sur un plateau rectangulaire, une fois que l'on a orienté le plateau de sorte que la longueur soit supérieure à la hauteur.

13.5.1 Redressement des chaînes indépendantes

La première méthode implémentée consiste à redresser les chaînes indépendantes : lorsque l'on rencontre une telle chaîne sous une forme non linéaire, on la remplace par la même chaîne, mais représentée linéairement. Une telle chaîne se reconnaît au fait qu'après suppression des flèches, elle n'est constituée que de jetons de degré 2, hormis les deux extrémités qui sont de degré 1. La figure 13.23 est suffisamment parlante pour arrêter ici les explications.

FIGURE 13.23 – Redressement d'une chaîne indépendante.

Cette méthode, comme toutes les méthodes de canonisation (§2.5.1) qui n'alourdissent pas trop le calcul des options, permet de gagner tant en espace mémoire qu'en temps de calcul. Le gain est ici variable suivant les positions étudiées, mais il est généralement de l'ordre de plusieurs dizaines de pourcents. En effet, cela concerne toutes les sommes de positions pour lesquelles un ou plusieurs termes de la somme sont des chaînes indépendantes, qui sont d'autant plus nombreuses que nous essayons justement de privilégier les sommes de positions dans le développement des arbres de recherche (voir plus loin le §13.7.3).

13.5.2 Orientation des angles droits

On appelle *angle droit* la figure obtenue lorsque deux arêtes internes reliées à un même jeton ne sont pas alignées. Il faut de plus que ce jeton soit de degré 2, et que les trois jetons formant l'angle droit soient aux sommets d'un carré dont le quatrième sommet est vide (jeton déjà capturé).

La figure 13.24 montre un tel angle droit. On obtient alors une position équivalente en plaçant le jeton central à l'emplacement du jeton déjà capturé. La canonisation est chargée de ne conserver qu'une seule position parmi ces deux positions équivalentes.

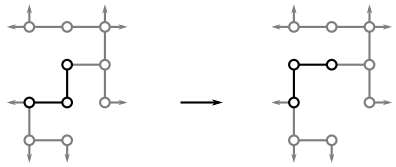


FIGURE 13.24 – Positions équivalentes via l'orientation d'un angle droit.

Cette méthode est un peu moins élémentaire que celle du paragraphe précédent. Son intérêt principal est qu'elle est facile à programmer, on peut détecter directement sur la représentation en plateau si un tel angle est présent dans la position. Le gain, moins important, est limité à quelques pourcents. Il peut cependant être plus important sur les grands plateaux (donc cette méthode a plus servi au jeu suédois). En effet, sur les petits plateaux, il n'y a pas assez de place pour que des angles apparaissent.

13.5.3 Extension

La théorie « Chocolat-Toile-Forêt », expliquée en annexe C et développée initialement pour le Cram, est susceptible de s'appliquer au Dots-and-boxes. Cette théorie contient les deux équivalences déjà expliquées dans ce chapitre, mais prend en compte de nombreuses autres équivalences de positions.

La figure 13.25 présente un exemple de position de Dots-and-boxes dans lequel nous avons identifié les trois éléments constitutifs de la théorie CTF : la partie chocolat, la plus interne, est représentée en noir, la partie toile est en gris, et la partie forêt, la plus externe, est en blanc. Il est à noter que les flèches n'ont pas d'importance pour savoir à quelle partie un jeton appartient.

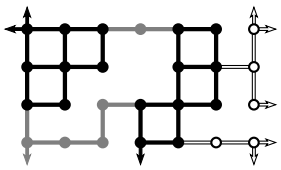


FIGURE 13.25 – Décomposition en Chocolat-Toile-Forêt.

Étant donnée la complexité du code nécessaire pour implémenter cette théorie, nous n'avons pas encore eu le temps de l'expérimenter. Nous pensons toutefois qu'elle est susceptible d'apporter des améliorations significatives, sans doute du même ordre que la méthode du redressement des chaînes indépendantes.

13.6 Table de transpositions

13.6.1 Stockage des positions

Une fois l'issue d'un nœud démontré, on le stocke dans une table de transpositions, de sorte que l'on n'ait pas à refaire le calcul si l'on rencontre à nouveau ce nœud plus tard. Ceci est particulièrement important, du fait du nombre élevé de transpositions ; d'autant plus qu'outre les transpositions « strictes », comme celle de la figure 13.10, on trouve également des transpositions où la position est identique, mais où le contrat est différent. Par exemple, dans la figure 13.9, si l'on commence par démontrer que le petit-fils de la racine situé le plus à gauche est gagnant, donc que le score de sa position est d'au moins 6, on en déduit que le fils situé le plus à droite est gagnant, car le score de cette même position sera d'au moins 5.

Étant donné une position $\mathscr{P}$ à n jetons, on sait initialement que son score théorique se trouve dans l'intervalle $[\![0;n]\!]$. Si l'on démontre que le nœud $(\mathscr{P};c)$ est gagnant, cela prouve que le score théorique de cette position est d'au moins c, donc on réduit l'intervalle des scores possibles à $[\![c;n]\!]$. Inversement, si l'on démontre ensuite que le nœud $(\mathscr{P};c')$ est perdant, on réduit l'intervalle des scores possibles à $[\![c;c'-1]\!]$. Le score théorique de $\mathscr{P}$ sera calculé lorsque les deux bornes de l'intervalle seront identiques.

Il suffira donc de stocker, dans la table de transpositions, les intervalles des scores possibles pour toutes les positions des nœuds que l'on aura démontrés.

13.6.2 Stockage des positions perdantes seulement

Pour économiser de la mémoire, on peut décider de ne stocker que les nœuds perdants. Dans ce cas, il suffit de stocker dans la table, pour chaque position, le plus petit contrat que l'on sait impossible à remplir. Par exemple, soit $\mathscr{P}$ la position suédoise 5×2 (voir figure 13.8), à 10 jetons, et de score théorique 6/4. On peut imaginer que l'on a d'abord démontré qu'il est impossible de remplir le contrat 10. On stocke alors $(\mathscr{P};10)$ dans la table. Si, un peu plus tard, on démontre que l'on ne peut pas remplir le contrat 9, on stocke ensuite $(\mathscr{P};9)$. On aura calculé le score de $\mathscr{P}$ quand on saura d'une part que le contrat 7 est impossible à remplir, et d'autre part que l'on peut remplir le contrat 6, c'est-à-dire qu'il existe un fils de $\mathscr{P}$ qui ne peut pas remplir le contrat $10-6+1=5$. Donc, dans la table, il y aura $(\mathscr{P};7)$ et $(\mathscr{F};5)$, où $\mathscr{F}$ est un certain fils de $\mathscr{P}$.

Nous avons utilisé cette astuce, car elle permet de diminuer la taille de la table de transpositions d'un facteur qui oscille entre 4 et 5,5 dans les calculs que nous avons menés. Les calculs les plus difficiles sont généralement ceux pour lesquels le facteur est le plus important, ce qui paraît logique, car ces calculs ont un facteur d'embranchement plus important, donc ils comportent en proportion plus de positions gagnantes.

Le gain en mémoire est donc important, pour une perte de temps assez limitée : de l'ordre de 50% de temps de calcul supplémentaire. La perte de temps vient de ce que lorsque l'on rencontre une position déjà calculée gagnante, elle n'est pas stockée en mémoire. Pour retrouver le fait que cette position est gagnante, il faut donc calculer ses options et vérifier que l'une d'entre elles est perdante.

13.6.3 Cache de positions gagnantes

Pour contrebalancer cette perte de temps, nous avons utilisé une astuce fonctionnant en sens inverse, à savoir, qui transforme l'espace mémoire en temps de calcul. Le principe est d'utiliser un cache de positions gagnantes. Le calcul fonctionne avec deux tables de transpositions, une pour les positions perdantes, une pour les positions gagnantes. Au départ, le calcul stocke à la fois les positions gagnantes et perdantes, puis, périodiquement, lorsque le nombre de positions gagnantes stockées atteint une valeur limite, on vide le cache.

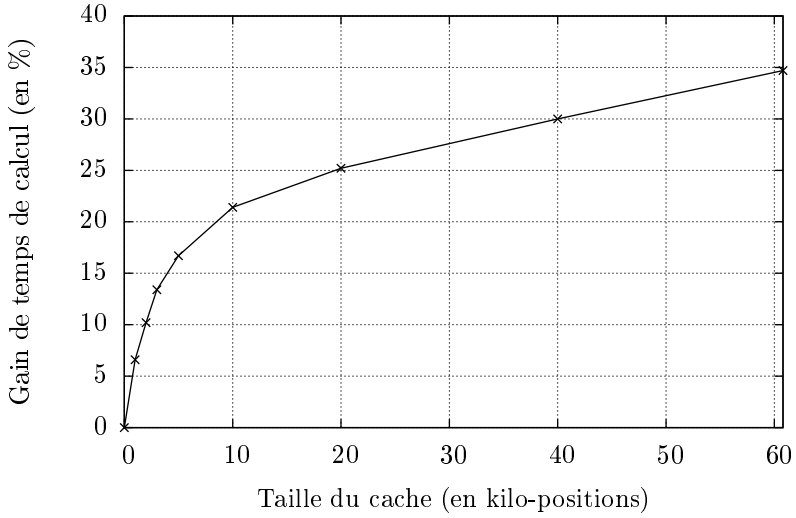


FIGURE 13.26 – Temps de calcul de $3 \times 4-$am6 en fonction de la taille du cache.

Un exemple est présenté sur la figure 13.26. Nous avons calculé l'issue de la position de départ américaine sur le plateau 3×4, avec un contrat de 6 (que nous notons en abrégé $3 \times 4-$am6). Ce calcul se termine normalement en 17 500 positions perdantes (17,5 kpos en abrégé), plus éventuellement 60,8 kpos gagnantes.

Si nous ne limitons pas la taille du cache, il termine avec les 60,8 kpos stockées, ce qui donne un gain en temps de 34,7% (inversement, cela donne une perte de temps de 53% sans cache, ce qui correspond à la valeur de 50% environ donnée précédemment). Avec un cache de 17,5 kpos, ce qui nécessiterait donc deux fois plus de mémoire que le calcul sans cache, le gain est proche de 25%, et avec un cache de 5 kpos, ce qui ne nécessiterait qu'un tiers de mémoire en plus, le gain est supérieur à 15%.

Une fois que l'on sait que le calcul que l'on souhaite mener ne nécessitera pas plus d'une certaine quantité de mémoire, l'utilisation de cette astuce permet d'utiliser la mémoire encore disponible pour la convertir en temps de calcul.

Enfin, nous pouvons préciser que nous avons gagné quelques dizaines de pourcents d'espace mémoire en optimisant la taille des chaînes de caractères qui représentent les positions.

13.6.4 Visualisation des positions

Les chaînes de caractères représentant le Dots-and-boxes sont difficiles à visualiser, ce qui gêne l'analyse des bases de données (ainsi que le suivi des calculs). Pour remédier à cet inconvénient, nous avons implémenté un outil de visualisation graphique des positions, qui peut être utilisé dans les tableaux d'affichage du programme. Comme expliqué dans la section 5.5 du chapitre sur le suivi, cela nous permet d'analyser les bases de données facilement et de suivre les calculs de Dots-and-boxes avec un affichage graphique en temps réel.

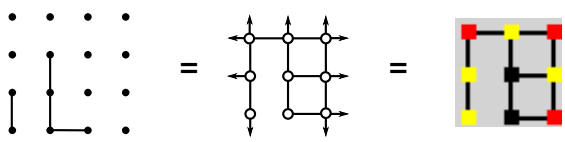


FIGURE 13.27 – Affichage graphique d'une position dans le programme.

La figure 13.27 montre une position de Dots-and-boxes, la position duale de Strings-

and-coins, et l'affichage graphique résultant dans le programme. On voit que l'affichage du programme est très proche de la position de Strings-and-coins : les nœuds et les arêtes de la position de Strings-and-coins sont représentés tels quels, par des carrés et des arêtes. Pour des questions de lisibilité (et aussi de facilité d'implémentation), nous représentons les flèches par des carrés de couleur. Un jeton avec une flèche est représenté par un carré jaune, tandis qu'un jeton avec deux flèches est représenté par un carré rouge.

La figure 13.28 montre le cas d'une position somme de deux composantes indépendantes.

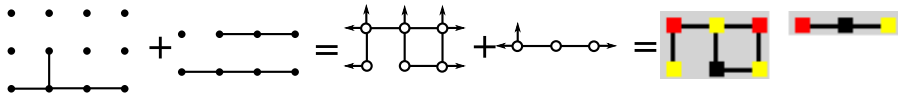


FIGURE 13.28 – Affichage graphique d'une position somme.

La programmation de cette méthode de visualisation nous a permis de découvrir des propriétés théoriques, comme celles du paragraphe 13.4.6, et aussi de déterminer certaines des heuristiques décrites dans le paragraphe 13.7 pour ordonner les calculs.

13.7 Heuristiques orientant le depth-first

13.7.1 Difficultés du jeu quasi-impartial

Comme sur les jeux de Sprouts et de Cram, l'ordre des options lors des calculs de depth-first (alpha-bêta) est un élément essentiel de l'efficacité des calculs. La mise au point d'heuristiques sur l'ordre des options s'est d'ailleurs révélée aussi difficile pour le Dots-and-boxes que pour ces deux autres jeux. L'une des raisons tient à la nature quasi-impartiale du Dots-and-boxes, ce qui empêche de créer des heuristiques pour déterminer quels sont les coups perdants ou gagnants.

Une exception notable cependant : ouvrir une chaîne est en général une mauvaise idée, car cela permet à l'adversaire de ramasser immédiatement des jetons. Nous avons donc mis une priorité aux positions qui ne contiennent pas de jeton de degré 1.

Toutes les autres règles relèvent d'une technique similaire aux heuristiques du Sprouts et du Cram, à savoir que l'on cherche à diriger le calcul prioritairement soit vers les parties les plus faciles de l'arbre, soit vers des parties toujours identiques (pour favoriser les transpositions). Le suivi des calculs, avec la visualisation graphique des positions qui permet de comprendre le déroulement du calcul, et le zappage qui permet de tester différents parcours, a joué un rôle important dans la mise au point de ces heuristiques. La plupart des idées qui suivent découlent en fait d'observations faites en réalisant et en manipulant directement de nombreux calculs de tests.

13.7.2 Équilibre de l'arbre de jeu

Si l'on programme le Dots-and-boxes de manière basique, et en associant les arêtes de l'arbre de recherche aux coups plutôt qu'aux tours de jeu, on obtient des arbres de recherche parfaitement équilibrés. En effet, si l'on étudie une position de départ à 40 arêtes, la racine de l'arbre a 40 fils, chacun de ces fils a 39 fils, qui chacun en ont 38...

En pratique, ce n'est pas tout à fait vrai. Par exemple, les équivalences de positions, comme la symétrie, diminuent le nombre d'options. Mais globalement, les arbres de jeu du Dots-and-boxes sont plutôt bien équilibrés, surtout quand on les compare à ceux du Sprouts, qui eux, sont excessivement déséquilibrés. Ainsi, dans un premier temps, nous avons concentré nos efforts sur une exploration de l'arbre de recherche qui se dirige le plus possible vers le même endroit de l'arbre, afin de faire apparaître le plus de transpositions possible.

L'ordre lexicographique s'est révélé particulièrement intéressant dans cette optique : il est simple à programmer et peu coûteux en temps de calcul, et a une bonne tendance à envoyer l'exploration dans les mêmes zones de l'arbre de jeu.

Pour améliorer cet ordre un peu basique, nous avons ensuite essayé de donner la priorité aux positions plus symétriques que les autres, car elles ont moins d'options que les autres, ce qui nous aurait permis d'exploiter une des rares manifestations du déséquilibre des arbres de jeu. Cela n'a pas donné de bons résultats, probablement à cause d'un conflit avec l'ordre lexicographique, provoquant l'exploration de zones très différentes de l'arbre de jeu.

Puis, à un certain moment, nous avons inversé l'ordre lexicographique. Notre premier ordre donnait en effet la priorité aux coups qui suppriment les flèches plutôt que les arêtes internes, mais nous nous sommes aperçus ensuite qu'il était préférable de supprimer d'abord les arêtes internes, puis les flèches.

Les lettres représentant les flèches, les arêtes, les jetons et les cases grises ont été choisies en conséquence : G < L, c'est-à-dire case grise < arête interne, donne la priorité aux cases grises sur les arêtes internes, favorisant donc leur suppression. Par ailleurs, A < B < C, c'est-à-dire deux flèches < une flèche < jeton normal, donne la priorité aux positions avec de nombreuses flèches, favorisant donc leur maintien.

13.7.3 Découpages

Cette inversion de l'ordre lexicographique était un premier indice du fait que le faible déséquilibre des arbres de jeu était exploitable. Nous avons pu également le constater avec la programmation d'une autre priorité, qui consiste à favoriser les découpages.

Car même si l'on ne sait pas tenir compte de ces découpages aussi bien que pour les jeux impartiaux, les positions découpées sont néanmoins plus simples à calculer. Les opérations de canonisation (dont le calcul de la symétrie canonique) sont réalisées sur chaque composante indépendamment et sont donc plus performantes. Cela favorise aussi l'apparition de jetons isolés ou de paires, sur lesquels on peut appliquer la théorie du paragraphe 13.4.6, ainsi que des chaînes indépendantes, qui peuvent être redressées (§13.5.1).

Pour aller plus loin, plutôt que de simplement favoriser les découpages au moment où ils apparaissent, nous avons fait en sorte de jouer en priorité les coups les plus susceptibles de faire apparaître des découpages. Nous avons essayé dans un premier temps de jouer les coups à l'endroit où le nombre de coups nécessaire à l'obtention d'un découpage était le plus faible. Cependant, le résultat n'était pas concluant, car ce faisant, l'exploration de l'arbre de jeu se faisait dans des zones trop différentes.

Nous avons donc réalisé un compromis : nous jouons des coups en priorité dans une des lignes, horizontale ou verticale, qui scinde un plateau le plus rapidement possible, et en deux parties de taille les plus proches possibles. Sur un plateau carré, il y a 2 lignes privilégiées si le nombre de jetons d'un côté du carré est pair, et 4 si ce nombre est impair. Sur un plateau rectangulaire, il y a 1 ou 2 lignes privilégiées, suivant que le nombre de jetons du plus grand côté du rectangle est pair ou impair.

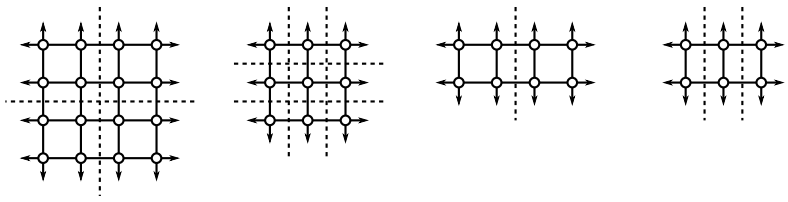


FIGURE 13.29 – Lignes privilégiées de découpage.

Enfin, nous favorisons les coups les plus proches possible du centre de la position, ce

qui permet de jouer en priorité des arêtes internes. Voici en définitive les priorités que nous utilisons, les plus importantes étant placées en haut :

* Priorité aux coups qui n'engendrent pas de jeton capturable.
* Priorité aux découpages.
* Priorité aux coups sur une des lignes privilégiées.
* Priorité aux coups les plus proches du centre de la position.
* Ordre lexicographique.

13.8 Résultats

13.8.1 Tableaux récapitulatifs

Les meilleurs résultats connus jusqu'ici étaient ceux de David Wilson en 2002 [45]. Les résultats nouveaux par rapport à Wilson ont été indiqués par un astérisque dans les tableaux. Il est à noter que Wilson n'a pas réalisé de calculs exhaustifs pour certains types de positions, en particulier les positions de taille $2 \times n$, même si son programme est théoriquement capable de calculer le score de n'importe quelle position jusqu'à une limite de 40 arêtes (sur les positions américaines). Dans les tableaux, nous n'avons donc pas considéré comme nouveaux des résultats qui auraient probablement pu être calculés par Wilson, même s'il ne sont pas indiqués sur son site web.

Pour des raisons évidentes de symétrie, le score du plateau $n \times m$ est le même que celui du plateau $m \times n$, et nous ne donnons que les valeurs des plateaux $n \times m$ avec $n \leq m$.

	2	3	4	5	6	7	8
2	3/1	2/4	5/3	4/6	7/5	7/7	*8/8
3	–	3/6	6/6	7/8			
4	–	–	8/8				

TABLE 13.2 – Scores calculés des positions américaines $n \times m$.

Le tableau 13.2 montre les résultats obtenus sur les positions de départ américaines. La seule position américaine qui dépasse la limite de 40 arêtes du programme de Wilson est la position de taille 2×8, avec 42 arêtes. Nous avons été capables de calculer cette position parce qu'à nombre d'arêtes équivalent, les plateaux aux dimensions déséquilibrées sont plus faciles à calculer. En effet, un déséquilibre des dimensions du plateau implique une apparition plus rapide des découpages, mais aussi un nombre de jetons plus élevé, donc un nombre de tours de jeu moins important. Ce phénomène se retrouve aussi dans les versions islandaises et suédoises.

	2	3	4	5	6	7	8	9	10
2	3/1	5/1	6/2	6/4	7/5	8/6	9/7	9/9	*10/10
3	–	4/5	8/4	7/8	10/8				
4	–	–	10/6	*11 ou 12					

TABLE 13.3 – Scores calculés des positions islandaises $n \times m$.

Le tableau 13.3 montre les résultats obtenus sur les positions de départ islandaises. Nous ne sommes pas certain de la limite de calcul du programme de Wilson en version islandaise. Une des astuces qu'il utilise en version américaine (la non-distinction des arêtes de coin) ne marche pas aussi bien en version islandaise, puisqu'il n'y a qu'un seul coin concerné au lieu de quatre. Nous avons donc considéré comme nouveau le score de la position islandaise

de taille 2×10, qui contient 40 arêtes, limite du programme de Wilson dans les conditions idéales de la version américaine.

Remarquons que nous avons obtenu un résultat partiel sur la position de taille 4×5 (40 arêtes également) : nous avons démontré que le premier joueur est capable de s'assurer 11 points, ce qui est suffisant pour prouver qu'il est victorieux, et également qu'il ne peut obtenir 13 points si son adversaire joue parfaitement. C'est-à-dire que cette position de taille 4×5 est gagnante avec le contrat 11, et perdante avec le contrat 13. Pour obtenir le score exact, il reste à calculer l'issue lorsque le contrat est 12.

position	2×2	2×3	2×4	2×5	2×6	2×7
arêtes	4	7	10	13	16	19
score	0/4	6/0	4/4	6/4	4/8	9/5

position	2×8	2×9	2×10	2×11	2×12	2×13
arêtes	22	25	28	31	34	37
score	8/8	10/8	9/11	12/10	12/12	14/12

position	2×14	2×15	2×16	2×17	2×18	2×19
arêtes	40	43	46	49	52	55
score	*14/14	*16/14	*16/16	*17/17	*18/18	*20/18

TABLE 13.4 – Scores calculés des positions suédoises de taille $2 \times n$.

Le tableau 13.4 montre les résultats obtenus sur les positions de départ suédoises de taille $2 \times n$. Comme pour la version islandaise, nous avons considéré comme nouveaux uniquement les scores de positions de départ contenant au moins 40 arêtes. La position 2×19 suédoise, avec 55 arêtes, est la position de départ de score connu qui contient le plus d'arêtes.

	3	4	5	6	7	8	9	10
3	8/1	8/4	10/5	11/7	11/10	12/12	*13/14	*14/16
4	–	8/8	10/10	13/11	*13/15			
5	–	–	*11/14					

TABLE 13.5 – Scores calculés des positions suédoises $n \times m$.

Enfin, le tableau 13.5 montre les résultats obtenus sur les positions de départ suédoises avec des dimensions supérieures à 3. Le score de la position suédoise 5×5 (40 arêtes) est sans doute le résultat nouveau le plus élégant. Les positions suédoises 3×9, 3×10 et 4×7 comportent respectivement 42, 47 et 45 arêtes différentes.

13.8.2 Exploitation des résultats

Quels que soient les types de positions de départ considérés (américaines, islandaises ou suédoises), il est difficile de voir émerger une régularité dans les résultats calculés. Ceci laisse penser qu'une résolution totale du jeu, sur les plateaux de toute taille, sera difficile à obtenir. Le problème le plus simple que l'on puisse imaginer, la résolution totale des positions suédoises de taille $2 \times n$, n'est visiblement pas trivial quand on regarde les scores obtenus sur les 18 premiers plateaux.

Le principal enseignement que l'on puisse tirer des scores calculés, c'est que mis à part les petits plateaux, les parties sont toutes relativement équilibrées. Hormis sur les positions suédoises de tailles 3×2 et 3×3, l'écart de points ne dépasse jamais 4, et n'est que rarement supérieur à deux.

Ces résultats sont à mettre en relation avec l'existence possible de stratégies permettant à chaque joueur de s'assurer un minimum de points quoi qu'il arrive. Une telle stratégie peut s'expliquer rapidement : un joueur découpe la position en un maximum de composantes indépendantes, en prenant garde de ne pas laisser de jeton capturable pour son adversaire. En fin de partie, s'il y a n composantes, il sera assuré de marquer au moins $2 \times (n-1)$ points, ceux des paires que l'autre joueur devra abandonner dans chaque composante pour conserver la main.

Établie de manière plus rigoureuse, voire adaptée à la programmation, cette stratégie pourrait permettre de minorer et majorer les scores des positions de départ.

13.8.3 Complexité

Notre programme est actuellement du niveau de celui de Wilson, et même légèrement plus performant. Notre principal avantage est que nous nous sommes limités à une résolution faible, alors qu'il pratiquait une résolution forte. Ce qui laisse d'ailleurs penser que plus les calculs que nous mèneront seront compliqués, et plus le fossé se creusera.

En effet, un programme du type résolution forte fait face à une complexité en $O(2^n)$, où n est le nombre d'arêtes. La complexité de notre résolution faible est expérimentalement moins importante. Nous pouvons l'évaluer à l'aide du graphique de la figure 13.30.

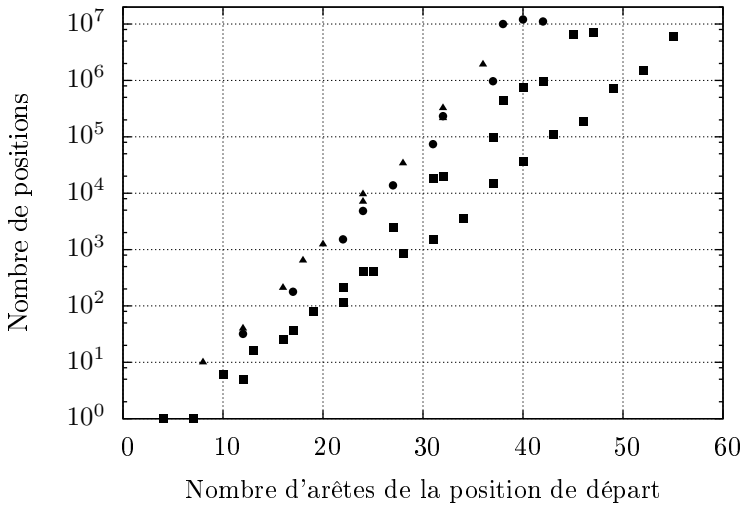


FIGURE 13.30 – Évaluation de la complexité des positions de départ en fonction du nombre d'arêtes.

Sur ce graphique, nous avons représenté par un point chaque calcul que nous avons mené. L'abscisse correspond au nombre d'arêtes de la position de départ, et l'ordonnée (en échelle logarithmique), au nombre de positions stockées dans la table de transpositions à la fin du calcul. Un carré symbolise une position de départ suédoise, un triangle symbolise une islandaise, et un cercle, une américaine.

Certains carrés sont relativement bien alignés le long d'une droite qui passe sous tous les points de ce graphique. Ces carrés correspondent aux positions suédoises de taille $2 \times n$, ce sont les calculs les plus faciles à mener. La complexité qui correspond à cette droite est environ de $O(2^{0,45n})$.

Quant aux triangles et aux cercles, ils sont également à peu près alignés sur une droite qui correspond à une complexité en $O(2^{0,6n})$: c'est la complexité approximative de la résolution faible des positions américaines et islandaises.

13.8.4 Pistes de recherche

Il ne manque plus grand chose à notre programme pour réussir à calculer la première position américaine inconnue, la position de taille 4×5. Entre la position 4×4 déjà calculée, et celle-ci, il y a 9 arêtes supplémentaires, ce qui donne un calcul $2^{0,6 \times 9} \simeq 42$ fois plus difficile, selon l'estimation développée au paragraphe précédent. Nous livrons dans ce paragraphe quelques pistes qui devraient permettre d'y parvenir.

Tout d'abord, les calculs les plus compliqués menés n'ont jamais dépassé 24 heures sur un ordinateur de bureau ordinaire. Des progrès sont donc envisageables simplement en menant des calculs plus longs. Toutefois, cela nécessiterait de régler le problème de la taille des tables de transpositions, qui saturent rapidement la mémoire disponible. Mais la grande quantité de temps de calcul disponible (on peut envisager de mener un calcul sur plusieurs mois, donc de disposer de 100 fois plus de temps de calcul) laisse de la latitude pour régler ce problème, le manque de mémoire pouvant être compensé par un temps de calcul plus long, moyennant une technique de programmation adéquate.

D'autres méthodes sont envisageables pour améliorer les performances du programme. Des progrès sont possibles dans la canonisation des positions avec la théorie CTF que nous avons déjà évoquée pour exploiter la déformation des positions. On pourrait également envisager une théorie qui prenne en compte les sommes de positions indépendantes de façon plus complète que ce qui est développé au paragraphe 13.4.6. Ce type d'améliorations est susceptible de réduire l'exposant dans la formule donnant la complexité du calcul.

Enfin, les algorithmes de parcours de l'arbre utilisés jusqu'ici restent assez élémentaires. De meilleures heuristiques pour le parcours en profondeur, voire un algorithme de type best-first (PN-search ou variante) seraient probablement à même d'accélérer les calculs.

L'estimation du paragraphe précédent, appliquée au jeu américain de taille 5×5, indique que ce calcul serait 4 000 fois plus difficile que celui de 4×4. Ce nombre n'est pas si important. Entre l'amélioration des performances des ordinateurs, et les améliorations théoriques potentielles, il est probable que dans quelques années, le vainqueur du jeu qu'Édouard Lucas avait présenté en 1889 soit enfin connu.

Chapitre 14

Conclusion

14.1 Résultats obtenus

Nous revenons tout d'abord sur les principaux résultats de calculs obtenus jusqu'ici dans le cadre de cette thèse.

14.1.1 Sprouts

Dans la version normale du jeu, une résolution faible (stratégie gagnante explicite) a été obtenue jusqu'à 44 points de départ, le record précédent de 2006 par Josh Jordan étant de 14 points de départ. Ces résultats ont été obtenus grâce à des innovations dans plusieurs directions complémentaires : une amélioration de la représentation du jeu avec des chaînes de caractères, un algorithme mélangeant les concepts d'issue et de nimber pour tirer partie au mieux des découpages en composantes indépendantes, un parcours de l'arbre de jeu avec des algorithmes variés comme l'alpha-beta et le Proof-number search, ainsi que des interactions manuelles permettant d'aider à diriger ces algorithmes vers les meilleures branches de la partie haute de l'arbre.

Dans la version misère du jeu, une résolution faible a été obtenue jusqu'à 20 points de départ, le record précédent de 2008 par Josh Jordan et Roman Khorkov étant de 16 points de départ. La représentation du jeu et l'algorithme de parcours sont les mêmes qu'en version normale, mais l'algorithme de calcul en lui-même est différent car le concept de nimber n'est pas disponible en version misère. Les découpages en sommes ont été exploités en remplaçant les petites composantes par leur arbre canonique réduit, d'où sont éliminés les coups redondants et les coups réversibles.

14.1.2 Cram

Nous avons appliqué au jeu de Cram l'ensemble des techniques utilisées sur le jeu de Sprouts, que ce soit au niveau des algorithmes de calcul ou des algorithmes de parcours, aussi bien en version normale qu'en version misère. Le seul élément qui a demandé une implémentation différente du jeu de Sprouts est la représentation des positions du jeu.

En version normale, une résolution faible a été obtenue pour le plateau de taille 7×7 (49 cases), le record précédent par Martin Schneider en 2009 étant le nimber du plateau 5×7 (35 cases). Comme dans le cas du Sprouts, les algorithmes en version misère sont moins efficaces qu'en version normale. Le plus grand plateau que nous avons pu résoudre en version misère est de taille 6×6 (36 cases), le record précédent par Martin Schneider étant le plateau de taille 5×5 (25 cases).

14.2 Fonctionnalités futures du programme

Nous décrivons dans les paragraphes qui suivent plusieurs fonctionnalités générales dont l'ajout permettrait d'améliorer sensiblement l'intérêt ou l'efficacité de notre programme.

14.2.1 Jeu homme-machine

Notre programme actuel ne permet pas de jouer directement contre la machine. L'absence de cette fonctionnalité est la conséquence d'avoir étudié en premier lieu le jeu de Sprouts. Sa nature topologique le rend difficile à représenter graphiquement, et il serait encore plus difficile d'exploiter cette représentation graphique pour permettre à la machine de jouer parfaitement contre un joueur humain.

Dans le cas des jeux de plateaux, une implémentation du jeu homme-machine serait envisageable plus facilement. Mais même si la représentation graphique et sa manipulation sont nettement plus simples que pour le Sprouts, il faudrait résoudre un problème délicat : les calculs sont systématiquement effectués sur des représentations canonisées des plateaux de jeu, dans le but d'identifier autant que possible des positions équivalentes. Cela peut induire des modifications très fortes de l'aspect du plateau de jeu, qui ne sont pas évidentes pour un joueur humain non initié au fonctionnement du programme.

Pour rendre le jeu homme-machine possible sans modifier brusquement la représentation du jeu suivant le bon vouloir de la machine, il faudrait donc un mécanisme qui permette de faire le lien entre le plateau non canonisé sur lequel se déroule la partie, et les bases de données de positions canonisées issues des calculs. Le problème serait simple si les algorithmes de canonisation étaient parfaits, mais pour des raisons de performance, ils s'agit presque toujours de pseudo-canonisations. Si la partie se déroule sur un plateau que l'on ne canonise pas au fur et à mesure, il n'est pas forcément trivial de retrouver à quelle position pseudo-canonisée de la base de données correspond la position du plateau.

14.2.2 Représentation des arbres de jeu

La notion d'arbre de jeu est définie dans la section 2.4.2. Des représentations graphiques de tels arbres ont été utilisées dans de nombreuses figures de ce mémoire, mais elles ont presque toutes été tracées à la main, avec le logiciel Inkscape, ou avec le logiciel Graphviz. Un tracé automatique par le programme permettrait de visualiser (et donc d'étudier) facilement les arbres de certaines positions.

Le tracé automatique des arbres solutions existait dans les premières versions de notre programme, lorsque celui-ci ne faisait que des calculs de Sprouts en version normale. La méthode employée consistait à créer un fichier texte décrivant l'arbre solution sous un format compatible avec le logiciel Graphviz. C'était ensuite le logiciel Graphviz qui traçait l'arbre solution. Malheureusement, cette fonctionnalité a été perdue lors de la généralisation à d'autres jeux et d'autres algorithmes.

Graphviz présente un intérêt de par ses algorithmes qui permettent de tracer efficacement des arbres comportant des transpositions. Cependant, les arbres étudiés comportent rapidement trop de sommets et de transpositions pour que leur tracé, ressemblant à un sac de nœuds inextricable, soit exploitable. De plus, pour des raisons de performance, il serait intéressant de ne pas dépendre de Graphviz, et de tracer les arbres de jeu directement avec les fonctionnalités de la bibliothèque graphique Qt sur laquelle est basée notre programme.

Idéalement, il faudrait ainsi ne représenter que des arbres sans transposition, soit en répétant les positions si nécessaire, ou bien en utilisant un système de références avec des numéros. Un outil de ce type permettrait d'effectuer des opérations évoluées, comme par exemple d'afficher graphiquement en temps réel les premiers niveaux de l'arbre de recherche

lors d'un parcours de type Proof-number search, voire même d'interagir avec cet arbre de recherche en cliquant directement dans sa représentation graphique.

14.2.3 Calcul distribué

Les calculs que nous avons menés sont en partie distribués dans le sens où nous avons fréquemment effectués certains calculs en plusieurs fois, souvent sur des ordinateurs différents, avant de fusionner les bases obtenues, puis de valider le résultat final grâce à une vérification unique sur un seul ordinateur. Les calculs de stratégies gagnantes se prêtent en fait particulièrement bien à la répartition sur plusieurs ordinateurs, car il est possible d'étudier sur chaque machine des parties différentes de l'arbre de jeu. Le calcul distribué a été utilisé avec succès par de nombreux auteurs, en particulier par Jonathan Schaeffer lors de la résolution du jeu de dames anglaises[36].

Notre programme possède plusieurs fonctionnalités qui pourraient servir de base à une implémentation complète du calcul distribué. Tout d'abord, la fusion des bases de calcul permet de regrouper les résultats des calculs séparés. Ensuite, la vérification permet non seulement de vérifier les calculs effectués sur un autre ordinateur (sans avoir besoin d'effectuer une nouvelle recherche dans l'arbre de jeu), mais aussi de réduire les bases de résultats. On peut donc imaginer un système où les ordinateurs clients vérifient leur calcul avant d'envoyer le résultat au serveur, pour diminuer la taille des arbres solution trouvés et réduire la taille des données à s'échanger et à stocker.

Nous avons commencé à développer une méthode de stockage des paramètres d'un calcul dans des fichiers XML, dans le but de mémoriser avec quels paramètres de calcul telle ou telle base a été obtenue. Ce système encore embryonnaire manque de stabilité et de facilité d'utilisation, et nous ne l'avons donc pas présenté dans le cadre de cette thèse. Il devrait permettre dans un futur proche d'échanger des fichiers de paramètres de calcul entre des ordinateurs, ce qui fournira une première clef vers le calcul distribué.

14.3 Autres pistes de recherche

14.3.1 Perspectives spécifiques au Sprouts et au Cram

Le Sprouts est le jeu sur lequel nous avons consacré le plus de temps, que ce soit au niveau de l'étude théorique du jeu ou des calculs informatiques, si bien que les améliorations spécifiques au Sprouts sont désormais plus difficiles. En particulier, les améliorations possibles de la représentation des positions sont désormais limitées, et leur impact sur le temps de calcul serait d'autant plus faible que la quasi-totalité des transpositions significatives sont déjà identifiées avec notre représentation actuelle. Les transpositions restantes sont marginales et donc rarement rencontrées dans les calculs réels. Une direction envisageable de recherches futures se situe au niveau des équivalences de régions, ou d'une amélioration du parcours de l'arbre de jeu grâce à une meilleure connaissance statistique des positions perdantes ou gagnantes.

Dans le cas du Cram, au contraire, nous avons consacré assez peu de temps à développer les aspects spécifiques de ce jeu, puisque que notre priorité était surtout de réutiliser les algorithmes de calcul et de parcours développés sur le Sprouts. Il reste donc une marge notable d'amélioration de la représentation des positions, en particulier avec la théorie chocolat-toile-forêt décrite en annexe C.

14.3.2 Algorithmes de parcours

Les algorithmes de parcours sont une direction prometteuse de recherche. En particulier dans le cas du Sprouts, le Proof-number search a montré de bons résultats, et de nombreuses

améliorations sont envisageables. Par exemple, l'algorithme Proof-number search ne tient compte d'aucune connaissance spécifique au Sprouts. Des heuristiques pour favoriser ou défavoriser certains types de positions statistiquement plus faciles ou plus difficiles pourraient permettre d'atteindre la solution plus rapidement. Le PN-search ne prend pas non plus en compte de façon optimale les spécificités des algorithmes de calcul sous-jacents, comme le nimber en version normale. Une meilleure combinaison des concepts de PN-search et de nimber est donc une piste intéressante.

Mais plus généralement, le suivi en temps réel des algorithmes de parcours, que ce soit celui de l'alpha-beta ou du Proof-number search, nous a convaincu qu'il reste une marge d'amélioration notable des idées théoriques concernant les algorithmes de parcours. Dans bien des situations, un simple coup d'oeil à l'état du calcul suffit à l'humain pour prendre une décision, comme par exemple d'interrompre le calcul de telle ou telle branche, ou au contraire de se concentrer sur telle ou telle autre. Cette force de l'analyse humaine montre en creux la faiblesse des algorithmes connus jusqu'ici et laisse espérer des améliorations importantes.

14.4 Limite de calculabilité

Terminons enfin par un mot sur l'intérêt des calculs de jeux combinatoires menés dans cette thèse. L'intérêt pratique pour les joueurs est immédiat, en dévoilant une stratégie parfaite pour jouer à partir de certaines positions de départ, quoiqu'à double tranchant, car cela peut diminuer l'intérêt du jeu. Mais on conviendra qu'il y a peu de joueurs de Sprouts, et encore moins de Cram.

La véritable motivation des calculs informatiques — et pas seulement des calculs de jeux combinatoires — se ramène en fait toujours à la même question fondamentale : déterminer ce qui est calculable, et ce qui ne l'est pas. Les progrès informatiques permanents des cinquante dernières années, que ce soit au niveau du matériel ou des algorithmes, crée l'image populaire de l'ordinateur omnipotent, et a tendance à nous faire croire que tout est calculable, ou finira par le devenir. Pourtant, il existe nécessairement une limite infranchissable, physique et algorithmique, qui place définitivement certains calculs hors de notre portée.

Les théories fondamentales de l'informatique apportent de premières réponses. La théorie de la calculabilité permet de montrer que certains problèmes sont insolubles, quels que soient les algorithmes utilisés, tandis que la théorie de la complexité, en quantifiant la relation entre la taille des problèmes et le temps ou l'espace nécessaire au calcul, permet d'évaluer l'efficacité asymptotique des algorithmes ainsi que la difficulté intrinsèque de certains problèmes. Mais ces théories, par nature, ne peuvent pas apporter de réponse sur le problème de savoir quelle est notre limite de calcul en pratique.

Cette limite de calculabilité, relative aux moyens informatiques et aux théories disponibles à une époque donnée, ne peut être déterminée que par des records de calcul, que ce soit les décimales de π, le nombre minimal de coups permettant de résoudre une position quelconque du Rubik's cube, ou la stratégie gagnante de tel ou tel jeu combinatoire.

Annexe A

Résolution du jeu de Sprouts à 2 points

Nous présentons dans la figure A.1 un arbre solution pour le jeu de Sprouts à 2 points, dans sa version normale, joué sur le plan (c'est-à-dire le jeu classique du Sprouts, pas sa généralisation sur des surfaces). Cet arbre résume une stratégie que peut appliquer le deuxième joueur s'il souhaite gagner à coup sûr la partie.

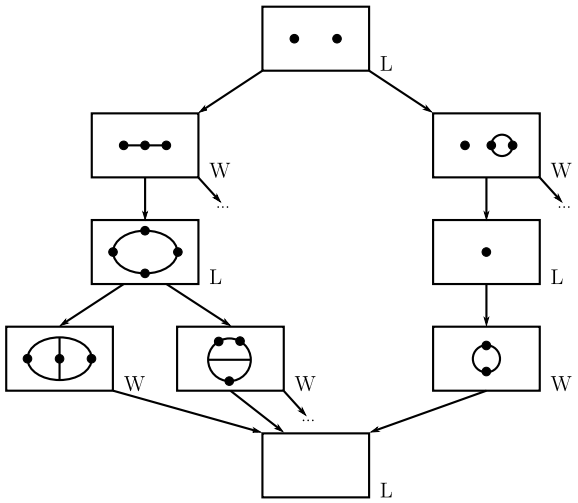


FIGURE A.1 – Arbre solution pour le jeu de Sprouts à deux points.

Les positions perdantes sont marquées de la lettre « L » (pour « Loss »), et les positions gagnantes de la lettre « W » (pour « Win »). Pour les positions gagnantes, il n'a pas été nécessaire d'indiquer toutes les options, seule suffit la donnée d'une position perdante.

Certaines positions équivalentes ont été identifiées avec les méthodes décrites dans le chapitre 9. Par exemple, le premier coup du premier joueur qui consisterait à relier un point à lui-même en enfermant l'autre point dans la région créée est équivalent à son premier coup de droite sur la figure A.1, en utilisant l'équivalence entre intérieur et extérieur que l'on déduit de la projection stéréographique.

Annexe B

Résolution du jeu de Sprouts à 5 points

Dans la figure B.1, nous présentons un arbre solution pour le jeu de Sprouts à 5 points, dans sa version normale, noté S_5^+. Cet arbre résume une stratégie que peut appliquer le premier joueur s'il souhaite gagner à coup sûr la partie.

Les positions entourées par une ellipse sont perdantes. Pour vérifier qu'elles sont perdantes, nous avons représenter chacune de leurs options. Celles entourées par un rectangle sont des sommes de deux positions perdantes, donc sont elle-mêmes perdantes. Celles qui ne sont pas entourées sont gagnantes, il nous suffit donc de représenter une de leurs options perdantes.

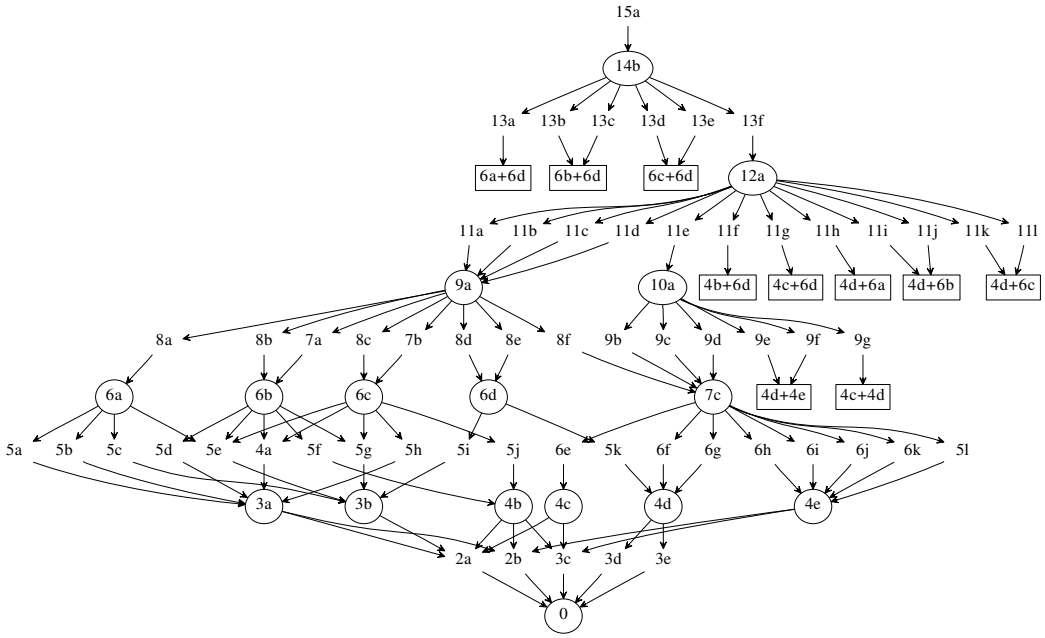


FIGURE B.1 – Arbre solution pour le jeu de Sprouts à 5 points.

La table B.1 explique à quelles positions correspondent les numéros des nœuds de la figure B.1. La numérotation utilise un nombre qui correspond au nombre de vies de la position, ainsi qu'une lettre arbitraire pour différencier les positions qui ont le même nombre de vies.

La lecture du chapitre 9 permet de comprendre quelles positions de Sprouts sont codées par les chaînes de caractères de la deuxième colonne de la table, ainsi que les astuces de canonisation qui permettent d'identifier des positions équivalentes.

Ce schéma ne comporte que 15 positions perdantes (sans compter la position terminale), nombre à comparer avec les 114 positions nécessaires à Applegate, Jacobson et Sleator en 1991 [4], et avec les 24 784 arbres canoniques différents dans l'arbre de jeu de S_5, qui traduisent la complexité de ce jeu. La réalisation d'une preuve aussi compacte a été rendue possible par l'utilisation conjointe de plusieurs techniques.

Tout d'abord, la canonisation décrite dans le chapitre 9 a permis d'identifier des positions équivalentes. L'algorithme PN-search a permis quant à lui une exploration de l'arbre de jeu qui se dirige rapidement vers la solution. Enfin, la vérification aléatoire décrite dans le chapitre 7 a permis de supprimer des positions inutiles dans l'arbre solution.

Cet arbre permet de bien visualiser certaines particularités du jeu de Sprouts. Ainsi, dans le haut de l'arbre, on observe que la plupart des positions peuvent se calculer grâce à un découpage en deux positions plus petites, faciles à calculer. C'est le cas des positions 13a à 13e, ou 11f à 11l.

On observe également que les positions 13f et 11e ne se démontrent qu'après une étude bien plus compliquée que ce qui se pratique ailleurs dans l'arbre : ce sont les positions obtenues lorsque le deuxième joueur a joué un coup qui relie entre eux deux points vierges, ce qui aboutit à une frontière dont la représentation est `1a1a`. Ce sont quasi-systématiquement les positions de ce type dont l'étude est la plus problématique dans les calculs de Sprouts.

FIGURE B.2 – Positions 13f et 11e (difficiles à étudier).

L'étude des positions de Sprouts avec un grand nombre de points de départ ressemble aux 6 premiers étages de cette figure : le facteur d'embranchement est très important, mais la plupart des options des positions perdantes se démontrent grâce à un découpage. Seules subsistent un petit nombre de positions dont la démonstration est difficile, notamment celles du type `1a1a`.

#	position perdante
14b	0*2.AB|0*2.AB
12a	0*2.AB|CDEF.AB|CDEF
10a	ABCD.EF|GHIJ.EF|ABCD|GHIJ
9a	0*2.AB|2AB
7c	2AB|CDEF.AB|CDEF
6a	0.A|1A
6b	0.A|ABC|BC
6c	0.AB|2AB
6d	0*2
4b	2A|ABC|BC
4c	ABC|BCD|AD
4d	ABCD|ABCD
4e	2AB|2AB
3a	12
3b	0

#	position gagnante
15a	0*5
13a	0*2.A|0.1aAa
13b	0*2+0.A|0.A
13c	0*2.AB|0.AB.CD|CD
13d	0*2.AB|0.CD|AB.CD
13e	0*2.2+0*2
13f	0*2.AB|1a1a.AB
11a	0*2.AB|2CD.AB|CD
11b	0*2.AB|2CD|AB.CD
11c	0*2.AB|ABC|CDE|DE
11d	0*2.AB|AB.CD|CE|DE
11e	1a1a.AB|CDEF.AB|CDEF
11f	0*2+2.ABCD|ABCD
11g	0*2.A|aAaBCD|BCD
11h	0.1aAa|BCDE.A|BCDE
11i	0.AB.CD|EFGH.AB|EFGH|CD
11j	0.A|0.A+ABCD|ABCD
11k	0*2.2+ABCD|ABCD
11l	0.AB|CDEF.GH|CDEF|AB.GH

#	position gagnante
9b	2AB.CD|EFGH.CD|EFGH|AB
9c	ABCD.EF|ABCD|EFG|GHI|HI
9d	ABCD.EF|ABCD|EF.GH|GI|HI
9e	2AB|CDEF.GH|CDEF|AB.GH
9f	2.ABCD|ABCD+ABCD|ABCD
9g	aAaBCD|EFGH.A|EFGH|BCD
8a	0.1aAa|2A
8b	0.AB.CD|2AB|CD
8c	0.AB|2CD|AB.CD
8d	0*2.A|2A
8e	0*2+22
8f	1a1a.AB|2AB
7a	0.A|0.A
7b	0*2.2
6e	2A|aAaBCD|BCD
6f	22+ABCD|ABCD
6g	2A|BCDE.A|BCDE
6h	2AB.CD|2CD|AB
6i	2AB|2CD|AB.CD
6j	2AB|ABC|CDE|DE
6k	2AB|AB.CD|CE|DE
5a	1A|ABC|BC
5b	12+1
5c	0+AB|AB
5d	12+AB|AB
5e	0.A|2A
5f	ABC|ADE|BC|DE
5g	0+22
5h	1aAa|2A
5i	0.AB|AB
5j	2AB|AB.CD|CD
5k	1a1a
5l	2.ABCD|ABCD
4a	0.2
3c	2A|2A
3d	AB|AC|BC
3e	2AB|AB
2a	AB|AB
2b	22

TABLE B.1 – Positions correspondant à la figure B.1.

Annexe C

Représentation Chocolat-Toile-Forêt

C.1 Problématique

Tant pour le Cram que pour le Dots-and-boxes, on peut obtenir des jeux équivalents en considérant leurs jeux duaux.

Pour le Cram, on remplace chaque case vide par un sommet, et l'on relie deux sommets par une arête lorsque les deux cases vides correspondantes sont situées à côté : on obtient un *graphe dual*.

Pour le Dots-and-boxes, on remplace chaque boîte par un sommet, puis on relie deux sommets par une arête lorsque les deux boîtes sont situées à côté l'une de l'autre, et que la ligne qui les sépare n'a pas encore été tracée. Le jeu équivalent obtenu est appelé *Strings-and-coins*.

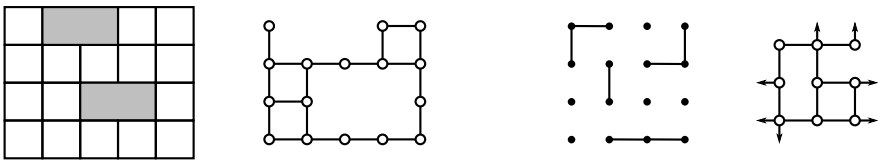


FIGURE C.1 – Positions de Cram et de Dots-and-boxes, et leurs graphes duaux.

Cette représentation sous forme de graphe revêt un intérêt : si deux graphes sont *isomorphes*, alors les positions correspondantes sont équivalentes. Considérer les graphes duaux permet donc de tenir compte des équivalences liées aux déformations des positions.

La figure C.2 présente à titre d'exemple trois positions de Cram (déjà rencontrées aux chapitres 2 et 12) qui ont le même graphe dual, représenté à droite de la figure. Ces trois positions sont donc équivalentes.

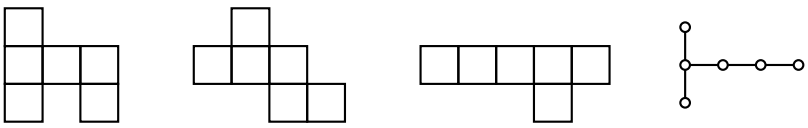


FIGURE C.2 – Trois positions de Cram avec le même graphe dual.

Ensuite se pose le problème de la représentation informatique de ces graphes. Une re-

présentation basée sur la définition des graphes (on affecte un numéro différent à chaque sommet, puis on stocke les paires de numéros correspondant aux arêtes) poserait quelques soucis. Outre le fait qu'une telle représentation consommerait beaucoup de mémoire, le principal problème serait de déterminer un algorithme rapide et efficace pour identifier les graphes isomorphes.

Or, les graphes duaux de Cram et les positions de Strings-and-coins ont de nombreux points communs : ce sont des graphes qui proviennent de plateaux, donc des graphes planaires, et dont tous les sommets sont de degré au plus 4. Ainsi, dans cette annexe, nous détaillons une représentation informatique qui nous semble plus adaptée à ces graphes particuliers.

Nous n'avons pas eu le temps d'implémenter cette représentation, mais nous pensons qu'elle permettrait d'améliorer nos résultats du fait de l'identification de nombreuses positions équivalentes. Cette représentation est d'autant plus efficace que les plateaux sont grands, ainsi, nous pensons qu'elles donnerait de meilleurs résultats sur le Cram, où nous traitons des plateaux plus grands, que sur le Dots-and-boxes. Et sur le Dots-and-boxes, les résultats seraient sans doute meilleurs sur les positions suédoises, en particulier celles dont les deux dimensions du plateau sont déséquilibrées.

C.2 Chocolat, Toile et Forêt

Le premier travail va consister à séparer les arêtes du graphe en trois ensembles disjoints : le chocolat, la toile et la forêt. Dans le cadre du Dots-and-boxes, seules les arêtes internes sont concernées [1].

C.2.1 Chocolat

Nous avons voulu décrire avec le *chocolat* une partie du graphe trop rigide pour laisser la moindre latitude lorsque l'on cherche à tracer la position duale du graphe. Ainsi, au contraire de la figure C.2 où le même graphe correspond à plusieurs positions, un graphe qui ne comporterait qu'une seule *tablette de chocolat* ne pourrait correspondre qu'à une seule position. Une définition plus formelle va nous permettre d'éclaircir ce propos.

Définition 17. *Étant donné un graphe, un* carré *est un ensemble de 4 sommets A, B, C et D, reliés entre eux par des arêtes : A—B ; B—C ; C—D et D—A.*

On peut maintenant définir le chocolat :

Définition 18. *On colorie les arêtes du graphe, de sorte que les 4 arêtes de tout carré soient de même couleur, et en utilisant un maximum de couleurs. Une* tablette de chocolat *est un ensemble de plusieurs arêtes ayant la même couleur.*

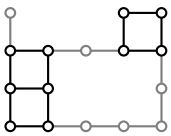


FIGURE C.3 – Graphe à deux tablettes.

Un graphe peut contenir plusieurs tablettes de chocolat. Par exemple, le graphe dual de la position de Cram de la figure C.1 contient 2 tablettes, représentées en noir sur la figure C.3. Remarquons également qu'il est possible qu'un sommet appartienne à deux tablettes, comme le sommet gris sur la figure C.4.

1. Il est également nécessaire de tenir compte des flèches (les arêtes non internes) dans la représentation décrite à la section C.3. Nous ne détaillons pas ce point dans un souci de concision.

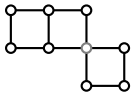


FIGURE C.4 – Sommet commun à deux tablettes.

C.2.2 Forêt

La *forêt* est la partie externe des graphes, constituée d'*arbres* au sens de la théorie des graphes. Il est facile de détecter les arêtes qui appartiennent à la forêt : on supprime les sommets de degré 1 du graphe, ainsi que les arêtes qui leur sont liées. On recommence tant qu'il reste des sommets de degré 1. Les arêtes qui se font supprimer lors de cet algorithme appartiennent à la forêt.

Remarquons que si cet algorithme termine en ne laissant qu'un sommet de degré nul, c'est que le graphe est un arbre. Hormis ce cas particulier, si l'on considère le graphe composé de tous les éléments qui se font supprimer par cet algorithme, les différentes composantes connexes obtenues sont des arbres.

Sur la figure C.5, on peut observer 3 arbres représentés en pointillés.

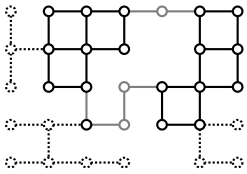


FIGURE C.5 – Graphe à 3 tablettes, 3 filaments et 3 arbres.

C.2.3 Toile

Les arêtes qui n'appartiennent ni au chocolat, ni à la forêt, appartiennent à la *toile*. Si un sommet du graphe est de degré 2, et si les deux arêtes qui le touchent sont des arêtes de toile, on colorie ces deux arêtes d'une même couleur. Ainsi, les arêtes de toile sont classées en différentes composantes unicolores appelées *filaments*. Un filament est au minimum de longueur 1, lorsqu'il ne contient qu'une arête.

Par exemple, la figure C.5 contient 3 filaments de longueurs respectives 1, 2 et 3, représentés en gris.

Un filament a toujours deux extrémités qui sont deux sommets de degré au moins 3, sauf dans un cas : si le filament est un cycle. Si ce cycle forme une composante connexe isolée du graphe, alors le filament n'a pas d'extrémité, sinon, il n'a qu'une seule extrémité, de degré au moins 3.

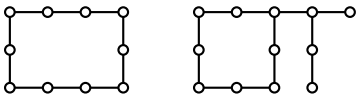


FIGURE C.6 – Cycle isolé, et cycle à une extrémité.

C.3 Représentation du graphe

Lettres

Dans la représentation que nous allons décrire, certains sommets particuliers seront désignés par des lettres, à partir de « `a` ». Il s'agit des sommets qui sont reliés à des arêtes d'au moins 2 objets parmi les tablettes, les filaments, et les arbres (il peut s'agir de tablette+tablette, ou tablette+filament+arbre, ou filament+filament+arbre... il y a de nombreuses possibilités).

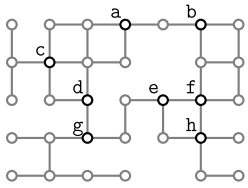


FIGURE C.7 – Sommets désignés par des lettres dans un graphe.

Chocolat

Pour chaque tablette, on peut utiliser une représentation sous forme de tableau, de même type que celle décrite dans le chapitre 12 pour les plateaux de Cram. Les sommets sont représentés par le chiffre `0`, ou par la lettre qui leur correspond si nécessaire, et les cases vides par la lettre `G`. Le caractère `*` est utilisé pour déterminer la longueur d'une ligne du tableau.

Toile

Chaque filament sera représenté sous la forme `ab4`, où `a` et `b` sont les lettres aux extrémités du filament, et où `4` est sa longueur.

Si le filament est un cycle isolé, on utilise la lettre spéciale `@` pour décrire ce cas particulier : `@10` désigne ainsi un cycle de longueur 10. Si par contre le filament est un cycle lié au reste du graphe, alors il est lié par une unique lettre, et sa représentation est du type `aa8`.

Forêt

Pour représenter un arbre, on commence par noter la lettre de sa racine, puis on réalise un parcours en profondeur : lorsque l'on rencontre un sommet pour la première fois lors du parcours, on écrit « `{` », et lorsqu'on le rencontre pour la dernière fois, on écrit « `}` ».

La représentation de l'arbre de la figure C.8 est donc :

```
a{{{}{{{}{{}{}}}}{{{}{}}}}}
```

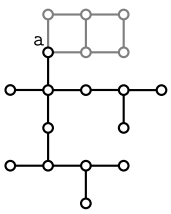


FIGURE C.8 – Arbre contenu dans un graphe.

Exemple

Une représentation correcte du graphe des figures C.5 et C.7 serait :
- `00a*c000dG b0*00f0 ef*0h` pour le chocolat.
- `ab2 dg1 eg3` pour la toile.
- `c{{{}{}}} g{{{}{}{}{}}}} h{{}{{}}}` pour la forêt.

Canonisation

Canoniser chaque objet indépendamment du reste de la position n'est pas très difficile. Pour canoniser une tablette, il suffit de considérer ses 4 symétries (8 si la tablette est carrée), et de ne garder que la plus petite pour l'ordre lexicographique. Il n'y a aucun travail à faire pour un filament, et enfin, on peut canoniser les arbres récursivement : un arbre est constitué d'une racine, reliée au maximum à 3 arbres fils. On trie chaque fils séparément avec un appel récursif à la fonction de tri, puis on trie les fils entre eux.

Le plus difficile est de canoniser l'ensemble de la position, car il se pose le même problème avec les lettres que dans le cas du Sprouts (voir le paragraphe 9.4.3). En effet, pour être sûr d'avoir une représentation canonique, il faudrait tester toutes les façons de renommer les lettres, canoniser chacune des représentations obtenues, puis garder la meilleure pour l'ordre lexicographique. Ceci introduirait un facteur en $n!$ dans la complexité de la canonisation, où n est le nombre de lettres de la représentation.

Un tel procédé n'est pas acceptable, car beaucoup trop coûteux en temps de calcul. Comme dans le cas du Sprouts, le recours à une pseudo-canonisation peut permettre de trouver un bon compromis entre rapidité d'exécution de la canonisation, et perte de performance liée à l'apparition de multiples représentations correspondant au même graphe.

Bibliographie

[1] Louis Victor Allis, *Searching for solutions in games and artificial intelligence*, Ph.D. thesis, University of Limburg, Maastricht, 1994.

[2] Victor Allis, *A knowledge based approach to connect-four. the game is solved*, Master's thesis, University of Vrije, Amsterdam, 1988.

[3] Piers Anthony, *Macroscope*, Avon Books, 1969.

[4] D. Applegate, G. Jacobson, and D. Sleator, *Computer Analysis of Sprouts*, Tech. Report CMU-CS-91-144, Carnegie Mellon University Computer Science Technical Report, 1991.

[5] Elwyn Berlekamp, *The dots-and-boxes game : sophisticated child's play*, A K Peters, 2000.

[6] Elwyn Berlekamp, John Conway, and Richard Guy, *Winning ways for your mathematical plays*, A K Peters, 2001.

[7] C. L. Bouton, *Nim, a game with a complete mathematical theory*, Ann. of Math. **3** (1902), 35–39.

[8] Dennis Breuker, Jos Uiterwijk, and Jaap van den Herik, *Solving* 8×8 *domineering*, Theoretical Computer Science **230** (2000), 195–206.

[9] Dennis M. Breuker, *Memory versus search in games*, Ph.D. thesis, Universiteit Maastricht, 1998.

[10] Nathan Bullock, *Domineering : solving large combinatorial search spaces*, Master's thesis, University of Alberta, Canada, 2002.

[11] Ralph M. Butler, Selden Y. Trimble, and Ralph W. Wilkerson, *A logic programming model of the game of sprouts*, Proceedings of the eighteenth SIGCSE technical symposium on Computer Science Education (1987), 319–323.

[12] John H. Conway, *On numbers and games (second edition)*, A K Peters, 2001.

[13] Jean-Paul Delahaye, *Le jeu des pousses*, Pour la Science **371** (Septembre 2008), 90–95.

[14] Joachim Draeger, Stefan Hahndel, Gerhard Köstler, and Peter Rossmanith, *Sprouts – Formalisierung eines topologischen Spiels*, Tech. Report TUM-I9015, Technische Universität München, Institut für Informatik, March 1990.

[15] Robert A. Hearn Erik D. Demaine, *Playing games with algorithms : algorithmic combinatorial game theory*, Games Of No Chance **3** (2008).

[16] Jonathan Schaeffer et Robert Lake, *Solving the game of checkers*, Game of No Chance **1** (1996), 119–133.

[17] Riccardo Focardi and Flaminia L. Luccio, *A modular approach to sprouts*, Discrete Applied Mathematics **144** (2004), no. 3, 303–319.

[18] George K. Francis and Jeffrey R. Weeks, *Conway's zip proof*, American Mathematical Monthly **106** (1999), 393–399.

[19] Martin Gardner, *Mathematical games : of sprouts and brussels sprouts, games with a topological flavor*, Scientific American **217** (July 1967), 112–115.

[20] Richard Gillam, *The anatomy of the assignment operator*, http://icu-project.org/docs/papers/cpp_report/the_anatomy_of_the_assignment_operator.html, 1997.

[21] Ralph Grasser, *Solving nine men's morris*, Games of No Chance **1** (1996), 101–113.

[22] P. M. Grundy, *Mathematics and games*, Eureka **2** (1939), 6–8.

[23] P.M. Grundy and C.A.B. Smith, *Disjunctive games with the last player losing*, Mathematical Proceedings of the Cambridge Philosophical Society **52** (1956), no. 3, 527–533.

[24] Roman Khorkov Josh Jordan Purinton, *Computation of misere sprouts with 15 spots*, http://groups.google.com/group/sprouts-theory/browse_thread/thread/effb79c5fa99d096, 2007.

[25] ———, *Computation of misere sprouts with 16 spots*, http://groups.google.com/group/sprouts-theory/browse_thread/thread/637b123af0c95f45, 2009.

[26] Peter Kissmann, *Symbolic search in planning and general game playing*, http://users.cecs.anu.edu.au/~ssanner/ICAPS_2010_DC/Abstracts/kissmann.pdf, 1950.

[27] Julien Lemoine and Simon Viennot, *A further computer analysis of sprouts*, http://download.tuxfamily.org/sprouts/sprouts-lemoine-viennot-070407.pdf, 2007.

[28] ———, *Sprouts game on compact surfaces*, http://arxiv.org/abs/0812.0081, 2008.

[29] ———, *Analysis of misère Sprouts game with reduced canonical trees*, http://arxiv.org/abs/0908.4407, 2009.

[30] ———, *Computer analysis of sprouts with nimbers*, http://arxiv.org/abs/1008.2320, 2010.

[31] ———, *Nimbers are inevitable*, http://arxiv.org/abs/1011.5841, 2010.

[32] Richard J. Nowakowski, *The history of combinatorial game theory*, http://www.mathstat.dal.ca/~rjn/papers/HistoryCGT.pdf, 2009.

[33] Thane E. Plambeck, *Taming the wild in impartial combinatorial games*, INTEGERS : Electronic Journal of Combinatorial Number Theory **5** (2005), G5.

[34] Josh Jordan Purinton, *Computation of normal sprouts with 14 spots*, http://groups.google.com/group/sprouts-theory/browse_thread/thread/191ff66a8db8c13f/4620c2e11248e264, 2006.

[35] Jonathan Schaeffer, *The games computers (and people) play*, Advances in computers **52** (2000), 189–266.

[36] Jonathan Schaeffer and al., *Checkers is solved*, Science **317** (2007), 1518–1522.

[37] Martin Schijf, *Proof-number search and transpositions*, Master's thesis, Universiteit Leiden, 1993.

[38] Dierk Schleicher and Michael Stoll, *An introduction to conway's games and numbers*, http://arxiv.org/abs/math/0410026, 2005.

[39] Martin Schneider, *Das spiel juvavum*, Master's thesis, 2009, http://www.mschneider.cc/papers/masterthesis.pdf.

[40] Claude E. Shannon, *Programming a computer for playing chess*, Philosophical Magazine **41** (1950).

[41] Aaron N. Siegel, *Misère games and misère quotients*, http://arxiv.org/abs/math.CO/0612616, 2006.

[42] R. Sprague, *Uber mathematische kampfspiele*, Tohoku Math. J. **41** (1935-36), 438–444.

[43] John Tromp, *John's connect four (web page)*, http://homepages.cwi.nl/~tromp/c4/c4.html.

[44] John Tromp and Gunnar Farnebäck, *Combinatorics of go*, http://homepages.cwi.nl/~tromp/go/gostate.ps, 2009.

[45] David Wilson, *Dots-and-boxes analysis*, http://homepages.cae.wisc.edu/~dwilson/boxes/, 2002.