Programme modulaire pour la résolution des jeux combinatoires : application au Sprouts et au Cram

Simon Viennot

Programme modulaire pour la résolution des jeux combinatoires

Application au Sprouts et au Cram

Thèse

par

Simon Viennot

présentée pour obtenir le grade de :

Docteur

Spécialité : **Informatique**

soutenue le 8 novembre 2011

Jury:

Jean-Paul Delahaye	Université Lille-I	*Directeur de thèse*
Bruno Bouzy	Université Paris-V	*Rapporteur*
Tristan Cazenave	Université Paris-Dauphine	*Rapporteur*
Sylvain Gravier	Université Grenoble-I	*Examinateur*
Philippe Mathieu	Université Lille-I	*Examinateur*
Éric Sopena	Université Bordeaux-I	*Examinateur*

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

Résumé

Nous cherchons dans cette thèse à calculer les stratégies gagnantes de jeux combinatoires avec un programme informatique. Nous montrons comment les découpages qui apparaissent au sein de certains jeux impartiaux peuvent être utilisés pour accélérer les calculs. Nous détaillons en particulier l'utilisation du concept d'arbre canonique réduit dans les calculs en version misère. Ces méthodes ont été appliquées avec succès au calcul de deux jeux impartiaux en apparence très différents : le Sprouts, où les joueurs relient des points par des lignes, et le Cram, qui consiste à remplir un plateau avec des dominos. Nous exposons ensuite une méthode originale de suivi des calculs de jeux, avec des interactions en temps réel par l'opérateur humain. Enfin, nous décrivons l'architecture du programme modulaire qui nous a permis de réaliser de nombreux calculs différents au sein d'un cadre commun, et qui pourrait être étendu à l'avenir à d'autres jeux ou algorithmes.

Mots-clefs : Théorie des jeux combinatoires, Jeu impartial, Nimber, Arbre canonique réduit, Sprouts, Cram.

Abstract

The goal of this thesis is to compute the winning strategies of combinatorial games. We show how to accelerate the computation of impartial games when some positions can be splitted into sums of independent components. We detail in particular the concept of reduced canonical tree in misere computations. We have applied these algorithms successfully to the game of Sprouts, where two players draw lines between spots, and the game of Cram, where the players fill a grid with dominoes. Then, we present an innovative method for monitoring the computation and allowing the human operator to interact in real-time. We also describe the architecture of the modular program that allowed us to compute a lot of different results in a single framework.

Keywords : Combinatorial game theory, Impartial game, Nimber, Reduced canonical tree, Sprouts, Cram.

Remerciements

Tout d'abord, je souhaite remercier Jean-Paul Delahaye, sans qui cette thèse n'aurait pas vu le jour. Les discussions que nous avons eu ont été le moteur de notre travail et ont débouché à chaque fois sur de nouvelles pistes de recherches.

Je remercie Tristan Cazenave, qui nous a orienté de façon décisive vers plusieurs concepts essentiels, ainsi que l'ensemble des membres du jury, dont les remarques et conseils nous ont apporté de nouvelles idées pour l'avenir.

Je tiens à remercier ma famille, en particulier mes parents, pour leur soutien pendant cette thèse. Je remercie aussi particulièrement ma femme, Yumiko, pour son soutien et sa patience.

Table des matières

1 Présentation de la thèse **11**
1.1 Contexte . . . 11
1.1.1 Théorie des jeux combinatoires . . . 11
1.1.2 Calculs des jeux combinatoires . . . 11
1.1.3 Positionnement de la thèse . . . 13
1.2 Historique de la thèse . . . 13
1.3 Méthodes de travail . . . 15
1.4 Plan de la thèse . . . 17

2 Théorie des jeux combinatoires **19**
2.1 Jeux combinatoires . . . 19
2.1.1 Qu'est-ce qu'un jeu combinatoire ? . . . 19
2.1.2 Jeux partisans et impartiaux . . . 19
2.2 Jeux étudiés dans cette thèse . . . 20
2.2.1 Jeu de Nim . . . 20
2.2.2 Jeu de Sprouts . . . 21
2.2.3 Jeu de Cram . . . 21
2.2.4 Dots-and-boxes . . . 22
2.3 Notion de jeu . . . 22
2.3.1 Définition formelle des jeux . . . 22
2.3.2 Déroulement du jeu . . . 22
2.3.3 Jeux courts . . . 23
2.3.4 Induction de Conway . . . 23
2.4 Stratégies gagnantes . . . 24
2.4.1 Issue d'une position . . . 24
2.4.2 Arbre de jeu . . . 24
2.4.3 Arbre solution . . . 25
2.4.4 Preuves par méthode . . . 26
2.5 Arbres canoniques . . . 26
2.5.1 Canonisation . . . 26
2.5.2 Arbres canoniques . . . 27
2.5.3 Lien avec la définition formelle des jeux . . . 28
2.6 Jeux découpables . . . 29
2.6.1 Somme de jeux . . . 29
2.6.2 Positions découpables . . . 29
2.6.3 Indistinguabilité . . . 30
2.7 Calculs informatiques . . . 31
2.7.1 Résolution des jeux . . . 31
2.7.2 Complexité spatiale d'un jeu . . . 32
2.7.3 Arbres de recherche . . . 33

2.7.4 Transpositions . . . 33
2.7.5 Espace et temps de calcul . . . 35

3 Jeux impartiaux en version misère **37**
3.1 Introduction . . . 37
3.1.1 Algorithme élémentaire commun . . . 37
3.1.2 Difficultés de la version misère . . . 38
3.1.3 Présentation de notre travail . . . 39
3.2 Arbres canoniques réduits . . . 39
3.2.1 Indistinguabilité . . . 39
3.2.2 Indistinguabilité en version normale . . . 39
3.2.3 Arbres canoniques . . . 40
3.2.4 Coups réversibles . . . 40
3.2.5 Arbres canoniques réduits . . . 41
3.2.6 Indistinguabilité en version misère . . . 42
3.2.7 Dénombrement . . . 43
3.2.8 Colonnes de Nim . . . 43
3.2.9 Rétablissement grâce aux coups réversibles . . . 44
3.2.10 Factorisation par $\mathbb{1}$ . . . 45
3.3 Calcul des arbres canoniques réduits . . . 45
3.3.1 Représentation et stockage . . . 45
3.3.2 Calcul de l'arbre canonique réduit d'une position . . . 47
3.3.3 Factorisation par $\mathbb{1}$ . . . 48
3.4 Algorithme de calcul utilisant les ACR . . . 48
3.4.1 Composantes des sommes . . . 49
3.4.2 Simplification des positions avec les ACR . . . 49
3.4.3 Exemple sur une position de Sprouts . . . 49
3.4.4 Calcul des enfants d'un nœud . . . 50
3.4.5 Intérêt des ACR . . . 50
3.4.6 Remplacement d'une position par son ACR . . . 50
3.4.7 Cas du Sprouts . . . 51
3.4.8 Sommes de positions . . . 51
3.5 Résultats . . . 51
3.5.1 Jeu de Sprouts . . . 52
3.5.2 Jeu de Cram . . . 52
3.6 Conclusion . . . 52

4 Suivi des calculs **53**
4.1 Affichage de la branche de calcul . . . 53
4.2 Zappage . . . 54
4.2.1 Positions bloquantes . . . 54
4.2.2 Principe du zappage manuel . . . 55
4.2.3 Implémentation multi-processus . . . 55
4.2.4 Sécurisation des objets partagés . . . 56
4.2.5 Déroulement du zappage . . . 57
4.2.6 Alternance des couleurs . . . 58
4.2.7 Avantages et inconvénients . . . 58
4.2.8 Automatisation du zappage . . . 59
4.3 Visualisation des arbres solutions . . . 59
4.4 Algorithmes de type Proof-number search . . . 60
4.4.1 Particularités de ces algorithmes . . . 60
4.4.2 Suivi du PN-search . . . 61

4.4.3 Interaction avec le PN-search . . . 62
4.4.4 Intérêt des interactions . . . 63
4.4.5 Recherche de nouveaux algorithmes . . . 64
4.5 Affichage des plateaux . . . 64
4.5.1 Position du problème . . . 64
4.5.2 Affichage en temps réel . . . 64
4.5.3 Influence sur le temps de calcul . . . 65
4.5.4 Affichage des tables de transpositions . . . 66

5 Architecture du programme **69**
5.1 Outils de programmation . . . 69
5.1.1 Langage C++ . . . 69
5.1.2 Bibliothèque STL . . . 69
5.1.3 Bibliothèque Qt . . . 70
5.1.4 Subversion . . . 70
5.1.5 Doxygen . . . 70
5.1.6 Dokuwiki . . . 70
5.1.7 Tuxfamily . . . 70
5.2 Principaux éléments du programme . . . 71
5.2.1 Figure d'ensemble . . . 71
5.2.2 Taille du programme . . . 73
5.2.3 Programme multi-processus . . . 74
5.3 Boucle de calcul principale . . . 74
5.3.1 Calculs avec ou sans suivi . . . 74
5.3.2 Boucle principale sans suivi . . . 75
5.3.3 Nœuds disponibles . . . 76
5.3.4 Boucle principale avec suivi . . . 77
5.3.5 Arbre de recherche . . . 77
5.4 Modularisation des jeux, nœuds et parcours . . . 78
5.4.1 Classes C++ et dérivation . . . 78
5.4.2 Polymorphisme . . . 79
5.4.3 Le cauchemar des pointeurs . . . 80
5.4.4 Polymorphisme avec des objets . . . 82
5.4.5 Destruction, copie et clonage . . . 84
5.4.6 Intérêts et inconvénients . . . 85
5.5 Tables de transpositions . . . 86
5.5.1 Types de bases de données . . . 86
5.5.2 Objet informatique de stockage . . . 86
5.5.3 Interface d'accès . . . 87
5.5.4 Fichiers de calculs . . . 88
5.6 Sérialisation . . . 88
5.6.1 Problème de la conversion entre chaînes et objets . . . 88
5.6.2 Classe StringConverter . . . 89
5.6.3 Exemple d'utilisation . . . 90
5.6.4 Paramétrage du format des chaînes . . . 90
5.6.5 Intérêt et limites . . . 91
5.7 Interface graphique . . . 91
5.7.1 Description de l'interface . . . 91
5.7.2 Création simple des interfaces . . . 92

6 Le jeu de Sprouts **95**
6.1 Introduction . . . 95
6.1.1 Versions normale et misère . . . 95
6.1.2 Résolution du jeu . . . 95
6.1.3 Historique des calculs de Sprouts . . . 96
6.2 Programme élémentaire de calcul . . . 96
6.2.1 Représentation des positions . . . 96
6.2.2 Options d'une position . . . 97
6.2.3 Algorithme élémentaire de calcul . . . 97
6.2.4 Transpositions . . . 98
6.3 Nimber . . . 99
6.3.1 Idée d'utilisation des nimbers . . . 99
6.3.2 Calcul du nimber d'une position . . . 99
6.3.3 Calcul de l'issue d'une somme avec les nimbers . . . 100
6.3.4 Nécessité des calculs de nimber . . . 100
6.3.5 Comparaisons des calculs avec et sans nimber . . . 100
6.4 Ordre des options . . . 101
6.4.1 Principe . . . 101
6.4.2 Evaluation du nombre d'options . . . 102
6.4.3 Ordre plus complexe . . . 102
6.4.4 Comparaison des ordres . . . 103
6.5 Interactions manuelles . . . 103
6.5.1 Suivi . . . 103
6.5.2 Zappage . . . 104
6.6 Version misère . . . 104
6.6.1 Algorithme misère . . . 104
6.6.2 Phase de précalcul . . . 105
6.6.3 Résolution forte . . . 105
6.6.4 Premier calcul de S_{17}^- . . . 106
6.7 Proof-number search . . . 106
6.8 Vérification . . . 107
6.8.1 Principe . . . 107
6.8.2 Records d'arbres solutions en version normale . . . 107
6.8.3 Records d'arbres solutions en version misère . . . 108
6.8.4 Solution du jeu à 5 points . . . 109
6.9 Les conjectures du Sprouts . . . 109
6.9.1 Conjecture normale forte . . . 109
6.9.2 Nouvelle conjecture misère . . . 109
6.9.3 Autres phénomènes de périodicité . . . 110
6.9.4 Conclusion . . . 111

7 Le jeu de Cram **113**
7.1 Introduction . . . 113
7.2 Résultats connus . . . 113
7.2.1 Stratégie de symétrie . . . 113
7.2.2 Plateaux de taille $1 \times n$ . . . 114
7.2.3 Calculs informatiques . . . 115
7.3 Découpages en positions indépendantes . . . 115
7.4 Représentation . . . 116
7.5 Canonisation . . . 116
7.5.1 Symétries . . . 116
7.5.2 Cases isolées . . . 117

7.5.3 Réduction de la taille du plateau . . . 118
7.5.4 Algorithme de canonisation . . . 118
7.6 Ordre des positions . . . 118
7.6.1 Priorité aux découpages . . . 119
7.6.2 Priorité aux transpositions . . . 120
7.7 Calculs des arbres canoniques . . . 120
7.7.1 Résolution forte . . . 121
7.7.2 Complexité spatiale du Cram . . . 121
7.7.3 Qualité de la canonisation . . . 121
7.8 Calculs en version normale . . . 123
7.8.1 Résultats . . . 123
7.9 Calculs en version misère . . . 124
7.9.1 Choix des arbres canoniques réduits . . . 124
7.9.2 Valeur de Grundy misère . . . 125
7.9.3 Résultats . . . 125
7.10 Conclusion . . . 125

8 Conclusion **127**
8.1 Résultats obtenus . . . 127
8.1.1 Sprouts . . . 127
8.1.2 Cram . . . 127
8.2 Fonctionnalités futures du programme . . . 128
8.2.1 Jeu homme-machine . . . 128
8.2.2 Représentation des arbres de jeu . . . 128
8.2.3 Calcul distribué . . . 129
8.3 Autres pistes de recherche . . . 129
8.3.1 Perspectives spécifiques au Sprouts et au Cram . . . 129
8.3.2 Algorithmes de parcours . . . 129
8.4 Limite de calculabilité . . . 130

A Résolution du jeu de Sprouts à 2 points **131**

B Résolution du jeu de Sprouts à 5 points **133**

C Représentation Chocolat-Toile-Forêt **137**
C.1 Problématique . . . 137
C.2 Chocolat, Toile et Forêt . . . 138
C.2.1 Chocolat . . . 138
C.2.2 Forêt . . . 139
C.2.3 Toile . . . 139
C.3 Représentation du graphe . . . 140

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 [5]. On trouvera un historique du développement de ce domaine dans l'article de 2009 par Richard J. Nowakowski [27], 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 [6]. Sprague et Grundy ont ensuite découvert indépendamment en 1935 [36] et 1939 [17] 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 [28] 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 [30], 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 [16]. 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 [7], puis 9×9. Nathan Bullock est parvenu à atteindre ensuite le plateau 10×10 en 2002 [9].

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 [13] 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 [31].

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 [3].

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 [29] 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 [22].

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* [11]. 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 [23].

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 [24].

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 [26].

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 [33], 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 [39], 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. 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 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* [10], 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.

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.

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 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 [33].

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 3.

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 [5] que [32], 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 $\mathbb{n}$ la colonne de Nim à n allumettes et l'on utilisera l'opérateur $+$ pour indiquer les différentes colonnes. Ainsi, la position $\mathbb{7}+\mathbb{5}+\mathbb{4}+\mathbb{2}$ 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 $\mathbb{7}+2+\mathbb{4}+2$. Ou alors, il pourrait enlever toutes les allumettes de la troisième colonne, et la nouvelle position serait $\mathbb{7}+\mathbb{5}+\mathbb{0}+2$.

La résolution du jeu de Nim a été décrite pour la première fois par Bouton, en 1902 [6]. 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).

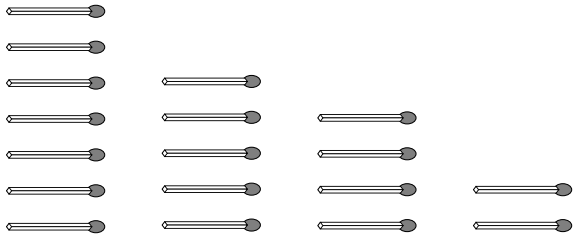


FIGURE 2.1 – Position $7 + 5 + 4 + 2$ du jeu de Nim.

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 [14]. Outre cet article, on peut également trouver une présentation de ce jeu dans *Winning Ways* [5]. 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. Notamment, le chapitre 3 sur la théorie des jeux impartiaux en version misère s'applique directement au Sprouts, et le chapitre 6 décrit les calculs réalisés sur ce jeu.

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* [5].

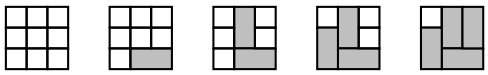


FIGURE 2.3 – Exemple de partie de Cram sur une grille 3×3 (le deuxième joueur gagne).

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 7 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 [4], un des trois co-rédacteurs de *Winning Ways* [5].

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.

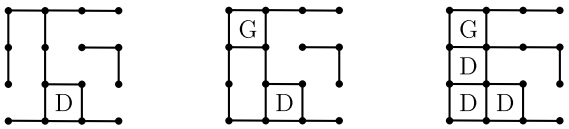


FIGURE 2.4 – Deux coups dans une partie de Dots-and-boxes.

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 [32], 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 $\mathcal{G}$ donné sont $\mathcal{G}$ (la *position de départ*), et toutes les positions des différentes options de $\mathcal{G}$. 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* [10] 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 ([10] 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 [32], 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 $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, ...\}$ dès que P est vraie pour tout $\mathcal{G}_i$. Alors, P est vraie pour tout jeu $\mathcal{G}$.*

Démonstration. Supposons qu'il existe un jeu $\mathcal{G}^0$ pour laquelle P est fausse. Alors, il existe une option $\mathcal{G}^1$ de $\mathcal{G}^0$ telle que P est fausse pour $\mathcal{G}^1$. En réappliquant cet argument, on peut construire une suite de jeux $\mathcal{G}^0$, $\mathcal{G}^1$, $\mathcal{G}^2$, ... 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 $\mathcal{G}$ dès lors qu'elle est vraie pour toutes les options de $\mathcal{G}$. 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.

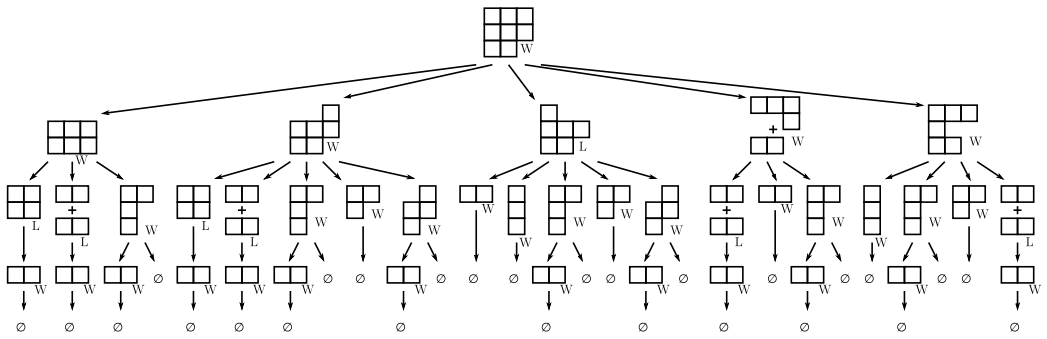


FIGURE 2.5 – Arbre de jeu d'une position de Cram.

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.

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.

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).

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.

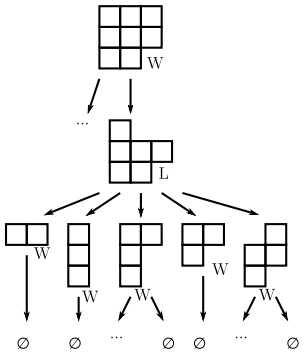


FIGURE 2.6 – Arbre solution d'une position de Cram.

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* $\mathscr{G}_1$ *est une option perdante de* $\mathscr{G}$ *et* $\mathscr{S}_1$ *un arbre solution de* $\mathscr{G}_1$, *alors* $\mathscr{S} = \{\mathscr{S}_1\}$ *est un arbre solution de* $\mathscr{G}$.

* *Si toutes les options $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \ldots$ 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 $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3 \ldots\}$ 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.

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 » [8]), 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 [6] 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.

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

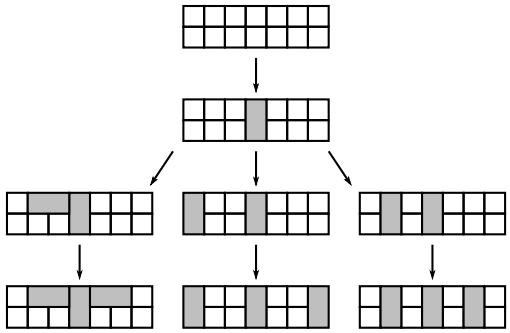


FIGURE 2.7 – Stratégie de symétrie.

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.

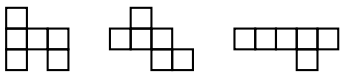


FIGURE 2.8 – Positions de Cram équivalentes.

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.

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 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.

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,*

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

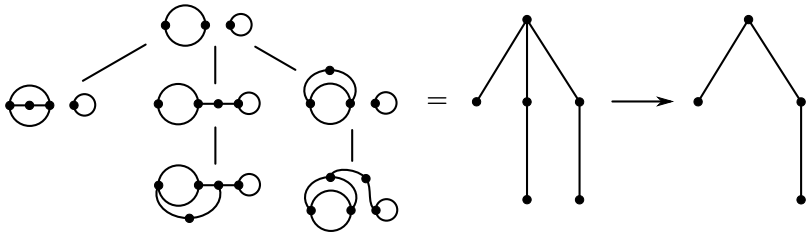


FIGURE 2.9 – Arbre de jeu d'une position de Sprouts, et arbre canonique associé.

* *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).

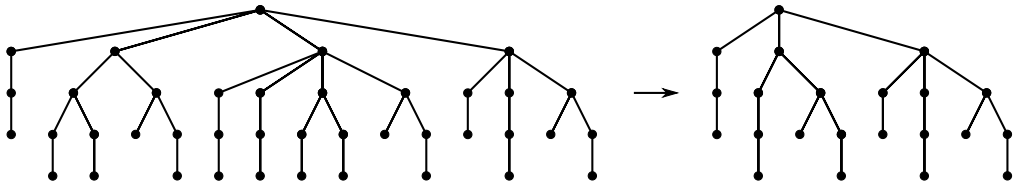


FIGURE 2.10 – Canonisation d'un arbre de jeu.

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 3 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* [5] ou *ONAG* [10]) : 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.

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

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

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* $\mathscr{G} = \{\mathscr{G}_1, \mathscr{G}_2, \mathscr{G}_3, ...\}$ *et* $\mathscr{H} = \{\mathscr{H}_1, \mathscr{H}_2, \mathscr{H}_3, ...\}$. *On définit récursivement le jeu somme* $\mathscr{G}+\mathscr{H}$ *par* $\mathscr{G}+\mathscr{H} = \{\mathscr{G}_1+\mathscr{H}, \mathscr{G}_2+\mathscr{H}, \mathscr{G}_3+\mathscr{H}, ..., \mathscr{G}+\mathscr{H}_1, \mathscr{G}+\mathscr{H}_2, \mathscr{G}+\mathscr{H}_3, ...\}$.

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

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

Le jeu somme $\mathscr{G} + \mathscr{H}$ se comporte en fait comme si les deux jeux $\mathscr{G}$ et $\mathscr{H}$ étaient placés côte à côte, et le joueur dont c'est le tour peut jouer un coup soit dans $\mathscr{G}$, soit dans $\mathscr{H}$, 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* $\mathscr{G}$ *donné, on dit qu'une position* $\mathscr{P}$ *du jeu* $\mathscr{G}$ *est* découpable *si celle-ci peut s'écrire comme une somme de jeux* $\mathscr{P} = \mathscr{G}_A + \mathscr{G}_B + ...$

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

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

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

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

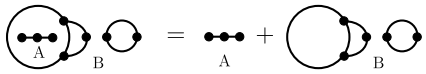


FIGURE 2.12 – Position de Sprouts découpable.

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.

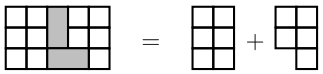


FIGURE 2.13 – Position de Cram découpable.

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. Le chapitre 3 s'attachera donc à montrer comment exploiter efficacement les découpages pour accélérer les calculs 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.

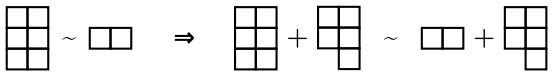


FIGURE 2.14 – Simplification d'une somme de positions de Cram en utilisant l'indistinguabilité.

Plus formellement, nous pouvons donner la définition suivante de 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 est l'objet de l'article [25].

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 3, 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 [34], 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 [31], 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 [21] et indépendamment par John Tromp [37]. 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 [38].

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

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.

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.

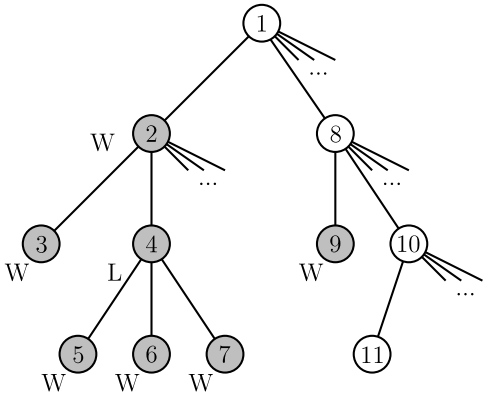


FIGURE 2.15 – Parcours d'un arbre en profondeur.

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.

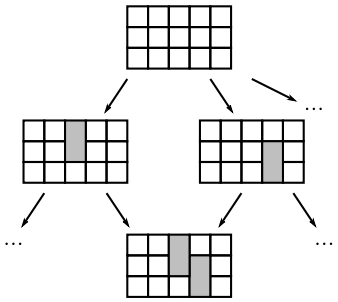


FIGURE 2.16 – Transposition dans un arbre de jeu.

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.

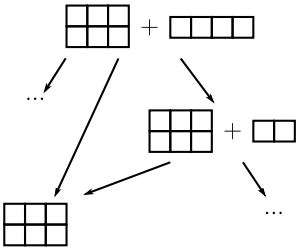


FIGURE 2.17 – Transposition suite à l'identification de positions équivalentes.

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.

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 » [8], 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 misère

3.1 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* [5] 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.

3.1.1 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 1 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 1 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.

3.1.2 Difficultés de la version misère

L'algorithme 1 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 notre article [26] 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 !

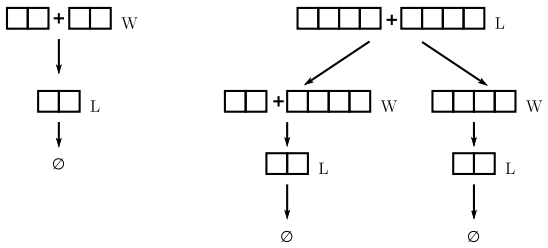


FIGURE 3.1 – Arbres solutions de positions de Cram.

La figure 3.1 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.

3.1.3 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.

3.2 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* [10], le livre de Conway (chapitre 12, p. 136–152), ou bien le fameux *Winning Ways* [5] de Berlekamp, Conway et Guy (chapitre 13, p. 413–452).

3.2.1 Indistinguabilité

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

Suivant [28], 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.

3.2.2 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 3. *(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 l'article [25] et le chapitre 6 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 [10] que ce qui joue le rôle du nimber dans le cas des jeux misère est le concept, décrit ci-après, d'*arbre canonique réduit.*

3.2.3 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 3.2 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.

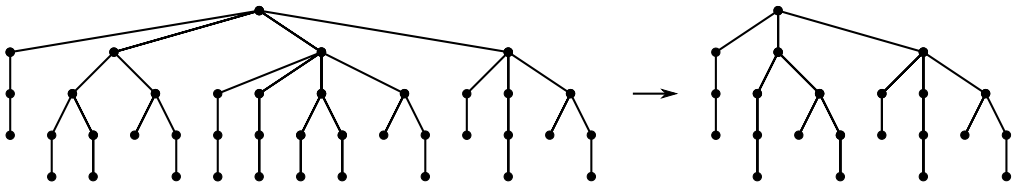


FIGURE 3.2 – Canonisation d'un arbre de jeu

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.

3.2.4 Coups réversibles

Coups réversibles

Définition 10. *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*).

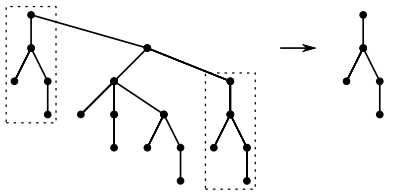


FIGURE 3.3 – Exemple de coup réversible

La figure 3.3 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.

3.2.5 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 $\mathscr{R}_1,\mathscr{R}_2,\mathscr{R}_3,\ldots$ 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.

3.2.6 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 [10] (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 [28]) 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 [35].

3.2.7 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 3.4, qui montre les 16 arbres canoniques de hauteur ≤ 3.

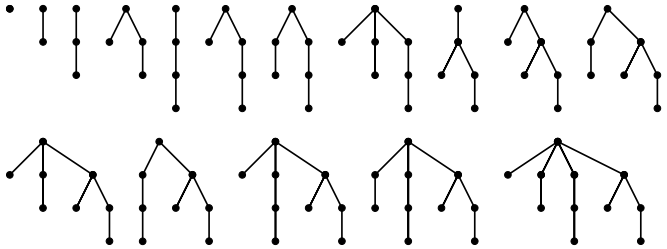


FIGURE 3.4 – Arbres canoniques de hauteur ≤ 3

Proposition 2. *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...

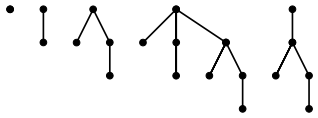


FIGURE 3.5 – Arbres canoniques réduits de hauteur ≤ 3 : $\mathbb{0}$; $\mathbb{1}$; $\mathbb{2}$; $\mathbb{3}$ et $\{2\}$

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 [18]). 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 [10] p. 140).

3.2.8 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 11. *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 [10] p. 139) :

Théorème 4. *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 5. *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 $\{1;\{2\}\}$.

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.

3.2.9 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 3.3, qui correspond à la position de Sprouts `0*2.A|1A`.

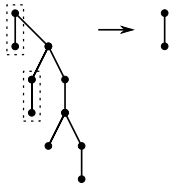


FIGURE 3.6 – Arbre canonique réduit de la position de départ à 3 points

3.2.10 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 $\mathbb{3} \sim 2 + \mathbb{1}$ et que $\{\mathbb{3}; \{2\}\} \sim \mathbb{1} + \{2\}$, on obtient : $\mathbb{3} + \{\mathbb{3}; \{2\}\} \sim 2 + \mathbb{1} + \mathbb{1} + \{2\} \sim 2 + \{2\}$, 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 $\{\mathbb{0}; \{2\}; \{\mathbb{3}; \{2\}\}\} + 2 \sim \{2\}$ dans [10] p. 151. Mais ces simplifications semblent trop rares pour être utiles, et nous détaillerons dans le paragraphe 3.4.8 pourquoi, dans notre programme, nous avons uniquement implémenté la simplification : $\mathbb{1} + \mathbb{1} \sim \mathbb{0}$.

3.3 Calcul des arbres canoniques réduits

3.3.1 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 $\{\mathbb{3}; \{\mathbb{1}; 2; \{\mathbb{3}; \{2\}\}\}\}$ [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 {{}}, 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 3.3.2).

Nous allons maintenant donner un exemple avec l'ACR de la figure 3.7, qui est celui de la position de Sprouts `1ABC|BCDE|ADE`. La représentation usuelle de cet ACR est $\{\mathbb{0};\mathbb{2};\{\mathbb{3}\};\{\mathbb{1};\mathbb{3};\{\mathbb{2}\}\}\}$.

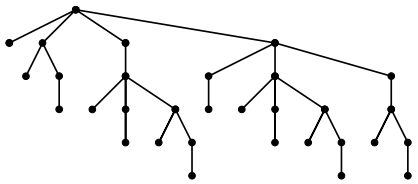


FIGURE 3.7 – Arbre canonique réduit de hauteur 5

Dans le tableau 3.1, 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`
$\mathbb{2}$	`2-0-W`	`0-0-W 1-0-L`
$\mathbb{3}$	`3-0-W`	`0-0-W 1-0-L 2-0-W`
$\{\mathbb{2}\}$	`3-1-L`	`2-0-W`
$\{\mathbb{3}\}$	`4-1-L`	`3-0-W`
$\{\mathbb{1};\mathbb{3};\{\mathbb{2}\}\}$	`4-2-W`	`1-0-L 3-0-W 3-1-L`
$\{\mathbb{0};\mathbb{2};\{\mathbb{3}\};\{\mathbb{1};\mathbb{3};\{\mathbb{2}\}\}\}$	`5-1-W`	`0-0-W 2-0-W 4-1-L 4-2-W`

TABLE 3.1 – 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 3.4.6 que dans nos calculs, nous avons produit une base de données d'ACR une fois pour toutes, de façon à éviter ce problème.

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

La définition du paragraphe 3.2.5 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 4 : 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 3.2.4 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 3.2.5 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 3.3.1, 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 3.3.2). 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).

3.3.3 Factorisation par $\mathbb{1}$

Nous avons vu dans le paragraphe 3.2.10 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 $\mathbb{3} \sim 2 + \mathbb{1}$ (ou, symétriquement, que $2 \sim \mathbb{3} + \mathbb{1}$). Le plus petit arbre des deux est 2. Donc 2 conserve son identifiant : 2-0-W, tandis que $\mathbb{3}$ aura désormais l'identifiant : 2-0+1-W.

On a vu également que $\{\mathbb{3}; \{2\}\} \sim \mathbb{1} + \{2\}$. Comme l'identifiant de $\{2\}$ est : 3-1-L, celui de $\{\mathbb{3}; \{2\}\}$ 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 3.3.1. 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 $\mathbb{1}$.

3.4 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.

3.4.1 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.

3.4.2 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 1, 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.

3.4.3 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 3.1 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`}.

3.4.4 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.

3.4.5 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 3.4.1 : 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.

3.4.6 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 3.2 : 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 3.2 – 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.

3.4.7 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`.

3.4.8 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 3.4.3, 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 3.2.10.

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 3.2.10 ne compensent pas ce problème : lors des tests que nous avons menés, nous n'en avons observé aucune.

3.5 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.

3.5.1 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 [20].

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 3.3 – Issues gagnantes/perdantes calculées.

Les résultats concernant le jeu de Sprouts sont détaillés dans le chapitre 6.

3.5.2 Jeu de Cram

Les meilleurs résultats connus précédemment en version misère étaient ceux de Martin Schneider en 2009 [33], 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 3.4 – 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 7.

3.6 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 4

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.

4.1 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 4.1 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 4.1). Chacun des cadres dessinés sur la figure 4.1 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 4.1 montre l'interface de suivi correspondant à la figure 4.1.

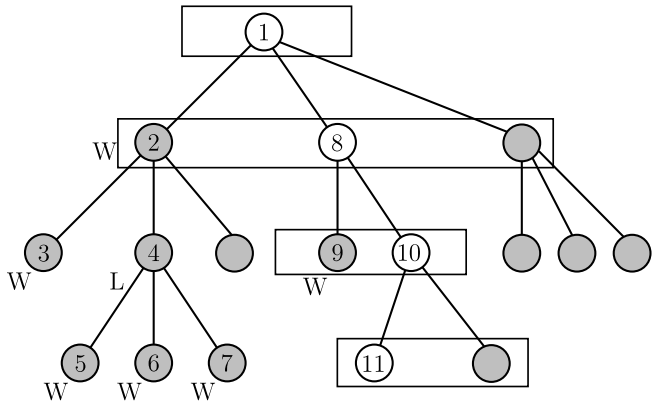


FIGURE 4.1 – Branche de calcul (nœuds en blancs).

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 4.1 – Interface de suivi correspondant à la figure 4.1.

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.

4.2 Zappage

4.2.1 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/`

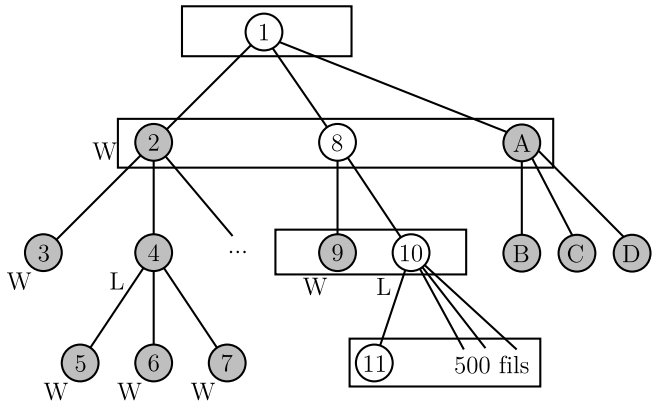


FIGURE 4.2 – Exemple d'arbre avec une position bloquante.

La figure 4.2 illustre comment une position bloquante pourrait apparaître dans l'arbre de recherche de la figure 4.1. 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.

4.2.2 Principe du zappage manuel

Le suivi correspondant à la figure 4.2 est le même que celui de la table 4.1 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.

4.2.3 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 4.3 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.

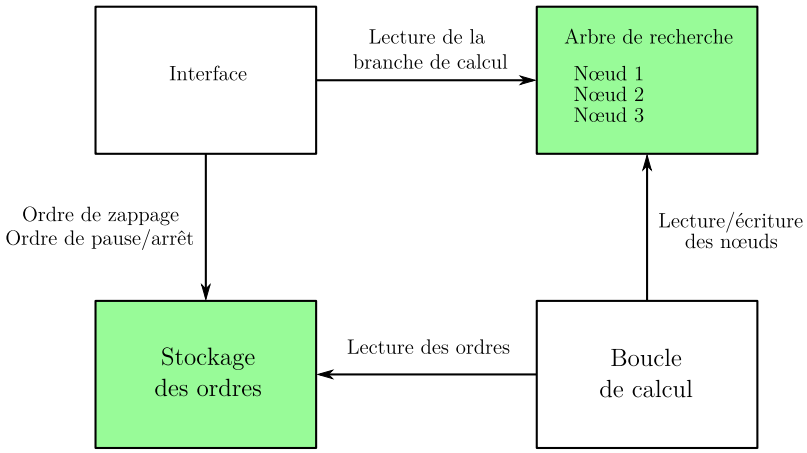


FIGURE 4.3 – Implémentation du suivi et du zappage.

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 (§4.5.3). 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.

4.2.4 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 2.

La simplicité de l'algorithme 2 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 2 Utilisation d'un objet partagé $\mathscr{O}$

```
Appel de mutex.lock() pour l'objet O
Utilisation de 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.

4.2.5 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 :

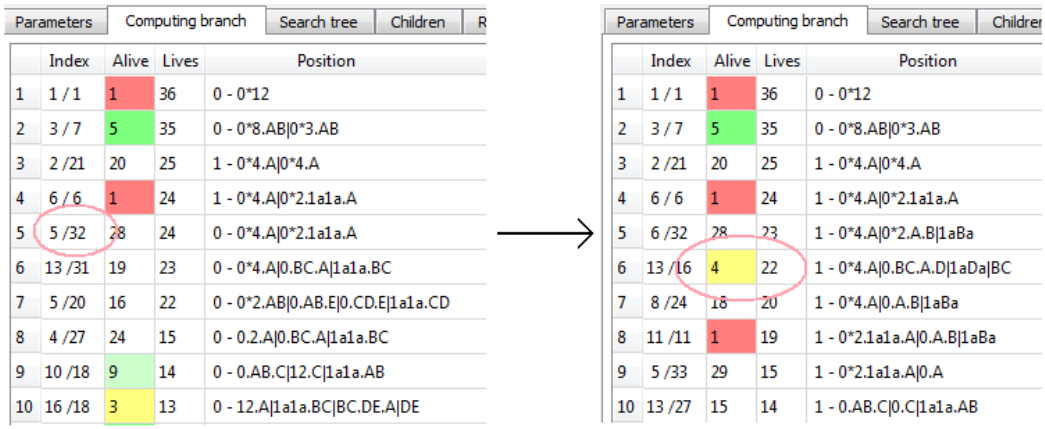
* utiliser des connaissances spécifiques au jeu qui n'ont pas encore été programmées, en se basant sur les chaînes de caractères qui représentent les positions, pour choisir des positions que l'on suppose perdantes et faciles à étudier.
* observer la forme du sous-arbre pour comparer la difficulté relative 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 4.4 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 4.1. 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 4.4 – 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 5e fils au 6e, 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.

4.2.6 Alternance des couleurs

L'alternance de couleurs obtenue sur la figure 4.4 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.

4.2.7 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.

4.2.8 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.

4.3 Visualisation des arbres solutions

Nous avons implémenté des méthodes de vérification des calculs, qui vérifient et épurent à l'aide d'un second calcul la table de transpositions d'un premier calcul. Cela permet 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.2 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 4.5 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

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

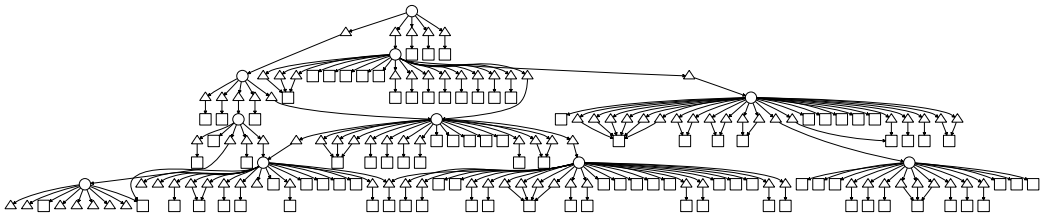


FIGURE 4.5 – Partie supérieure d'un arbre solution.

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 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.

4.4 Algorithmes de type Proof-number search

4.4.1 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.

3. en juillet 2011.

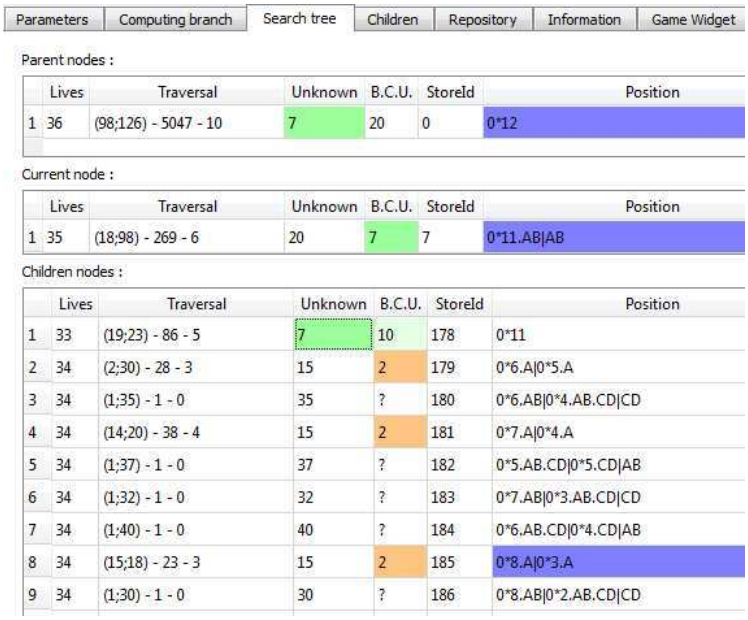
∗ le PN-search est capable de détecter qu'une branche devient bloquante par rapport à ses voisines, et va donner la priorité aux branches les plus faciles.

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.

4.4.2 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 4.6 – 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 4.6 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 9^{e} 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 8^{e}) 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.

4.4.3 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 4.7 montre un exemple d'interaction peu après la capture d'écran de la figure 4.6. 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

Parameters | Computing branch | Search tree | Children | Repository | Information | Game Widget

Parent nodes :

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

Current node :

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

Children nodes :

	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 4.7 – 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.

4.4.4 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.

4.4.5 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.

4.5 Affichage des plateaux

4.5.1 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 4.8 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.

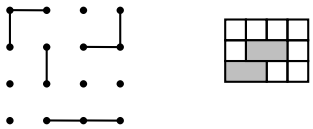


FIGURE 4.8 – Position de Dots-and-boxes et position de Cram.

4.5.2 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 4.9 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.

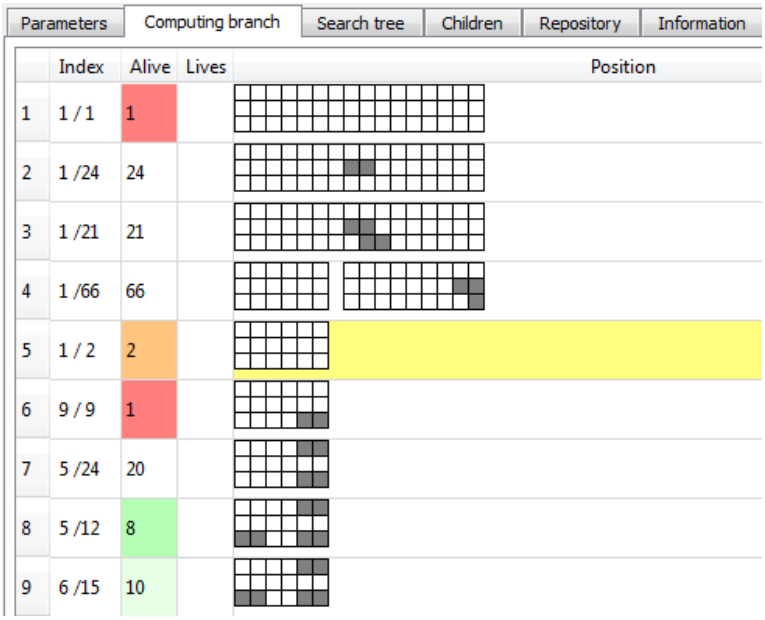


FIGURE 4.9 – Suivi graphique d'un calcul de Cram 3×16.

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.

4.5.3 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.

4.5.4 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.

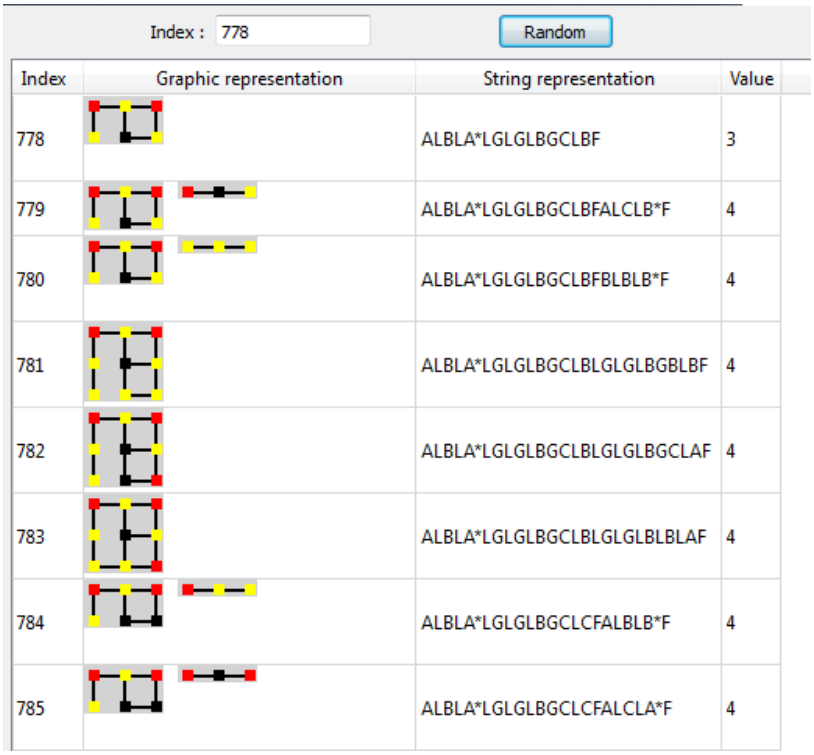


FIGURE 4.10 – Extrait d'une table de transpositions de Dots-and-boxes.

La figure 4.10 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 5

Architecture du programme

5.1 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.

5.1.1 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 5.4.

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.

5.1.2 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/`

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.1.6 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.

5.1.7 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.

5.2 Principaux éléments du programme

5.2.1 Figure d'ensemble

La figure 5.1 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.

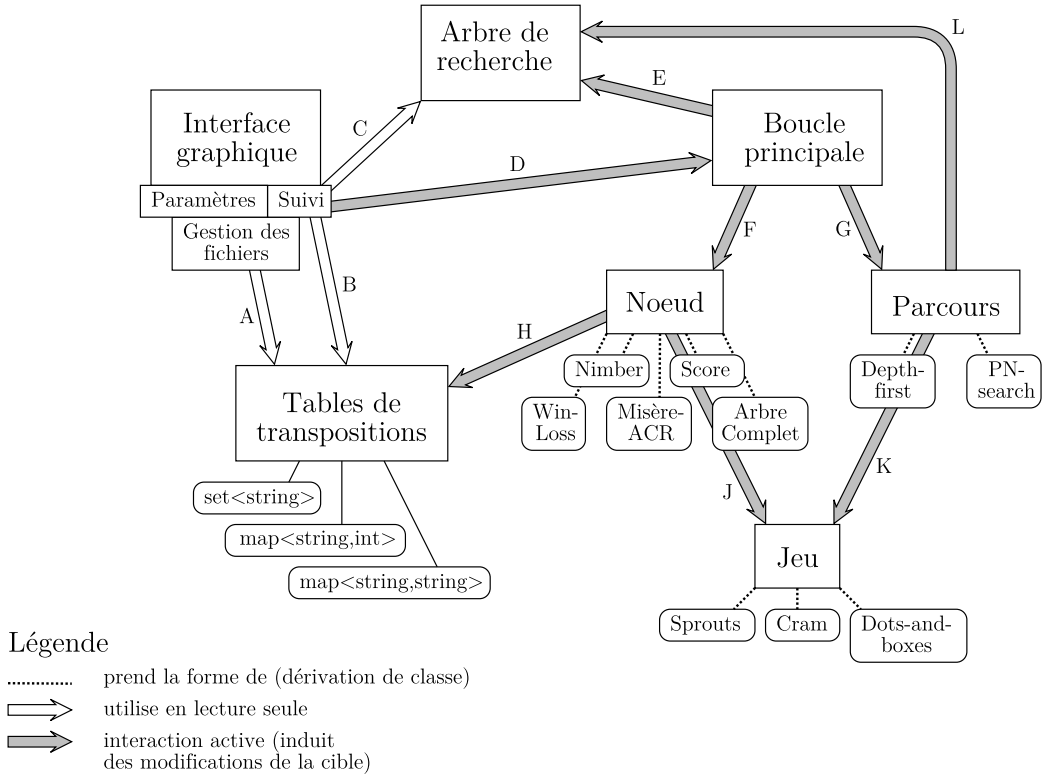


FIGURE 5.1 – Architecture modulaire du programme.

Les différents blocs de la figure 5.1 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 5.3. 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 5.4.

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 5.3 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 5.1.

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 5.5.

Les nœuds accèdent aux tables de transpositions (flèche H de la figure 5.1) 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 5.6.

Par ailleurs, la flèche A de la figure 5.1 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 5.7. Le suivi en temps réel des calculs est un sujet suffisamment riche pour être traité séparément, et fait l'objet du chapitre 4. Nous décrivons simplement ici les principales interactions avec les autres éléments du programme :

* La flèche B de la figure 5.1 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.

5.2.2 Taille du programme

La figure 5.1 permet de distinguer quatre grands ensembles de code assez distincts dans le programme.

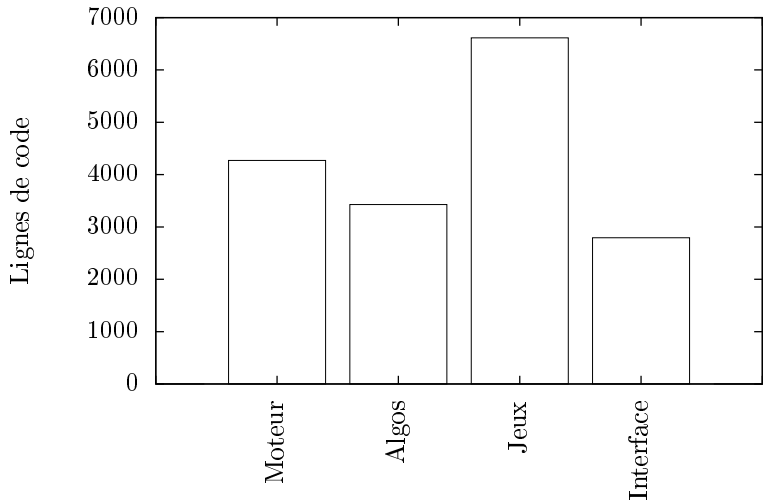


FIGURE 5.2 – Nombre de lignes de code des quatre principaux ensembles du programme.

* 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 5.1, 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 5.2 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 5.3 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.

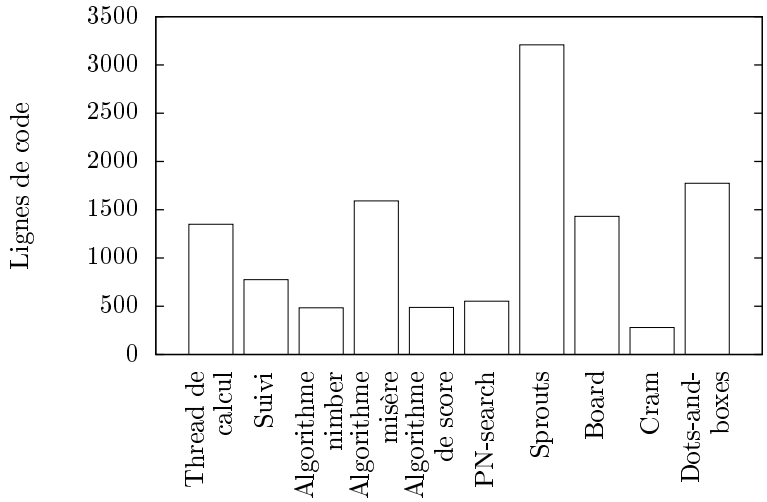


FIGURE 5.3 – Nombre de lignes de code des principaux modules du programme.

5.2.3 Programme multi-processus

Comme expliqué dans le chapitre 4 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 5.1, 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 §4.2.4 du chapitre 4.

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.

5.3 Boucle de calcul principale

5.3.1 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 4m 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 5.1). 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.

5.3.2 Boucle principale sans suivi

Le listing 5.1 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 5.1 – 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 5.1.

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.

5.3.3 Nœuds disponibles

Nous décrivons dans la table 5.1 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 5.1 – 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.

5.3.4 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.

5.3.5 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 5.1. 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 5.1.
* la liste des ids des nœuds enfants
* l'id du nœud parent
* un objet Traversal , qui correspond à un bloc « Parcours » de la figure 5.1, 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

5.4 Modularisation des jeux, nœuds et parcours

5.4.1 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.

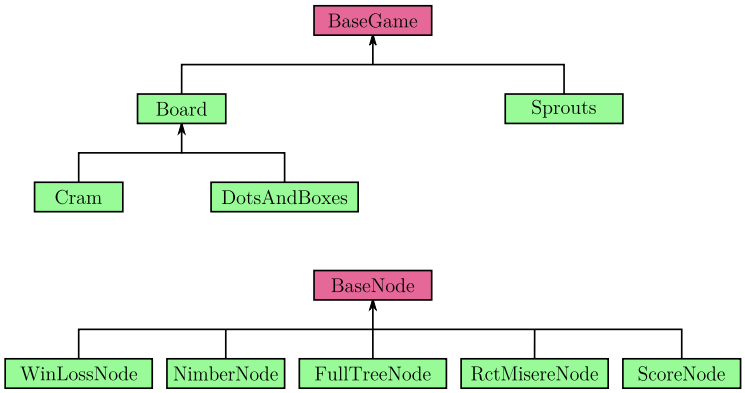


FIGURE 5.4 – Hiérarchie des classes de jeux et de nœuds.

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 5.4 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.

5.4.2 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 5.4, 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 5.2.

Listing 5.2 – 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 5.3 – 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 5.3 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.

5.4.3 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 5.4 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 5.4 – 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 5.5 – 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 5.5 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.

5.4.4 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 5.6 montre le principe de cette technique en reprenant l'exemple de la classe BaseGame déjà discuté en 5.2.

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 5.3 est, lui, inchangé.

Listing 5.6 – 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 5.7 est une réécriture du code 5.3 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 5.7 – 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 5.7, 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 5.7 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.

5.4.5 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 5.7), 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 [15].

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 5.6).

Listing 5.8 – 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 5.9.

Listing 5.9 – 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 5.10.

Listing 5.10 – 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 5.11 de clonage de la classe Cram, qui tient en une seule ligne.

Listing 5.11 – 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 5.6, 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.

5.4.6 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.

5.5 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.

5.5.1 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 5.6.

5.5.2 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.

5.5.3 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 5.12 – 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 5.2 – 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.

5.5.4 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 5.2 le fichier obtenu lors du calcul de Dots-and-boxes d'un plateau de taille 2×3, avec un contrat de 2.

5.6 Sérialisation

5.6.1 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.

5.6.2 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 5.13 – 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 >>.

5.6.3 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 3. 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 5.14 – 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 5.13, et c'est ce template qui appelle sur chaque élément du multi-ensemble la fonction 5.14 de conversion d'un arbre canonique réduit en chaîne de caractères.

Listing 5.15 – 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();
}
```

5.6.4 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).

5.6.5 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.

5.7 Interface graphique

5.7.1 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.

5.7.2 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 6

Le jeu de Sprouts

6.1 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.

6.1.1 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 12. *Dans ce chapitre, on notera S_p^+ le jeu de Sprouts à p points en version normale, et S_p^- en version misère.*

6.1.2 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.

6.1.3 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 [3] 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 [29]. 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 [19], puis l'issue de S_{16}^- en janvier 2009 [20]. 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.

6.2 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.

6.2.1 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. 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
```

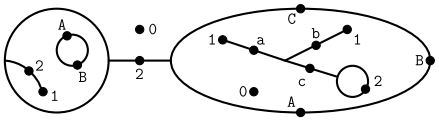


FIGURE 6.1 – Position de Sprouts et représentation en chaîne correspondante.

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).

6.2.2 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.

6.2.3 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 3 (alpha-bêta) qui découle directement de la définition même de l'issue, comme décrit dans la section 2.4 du chapitre 2.

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.

Algorithme 3 Calcul récursif de l'issue de $\mathscr{P}$

```
calculer la liste des options de P
Pour chaque option P_i faire
    calculer l'issue de P_i, et si celle-ci est perdante, renvoyer gagnant
fin Pour
renvoyer perdant (car toutes les options sont gagnantes)
```

6.2.4 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 6.2.

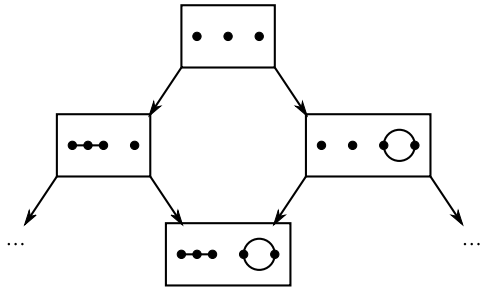


FIGURE 6.2 – Une transposition.

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.

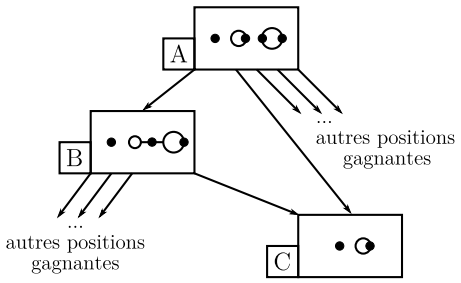


FIGURE 6.3 – Une transposition entre des étages de parités différentes.

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.

6.3 Nimber

6.3.1 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.

6.3.2 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.

6.3.3 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 4 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 4 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

6.3.4 Nécessité des calculs de nimber

Ce n'est qu'en 2009 que nous avons compris que l'algorithme 4 était inévitable en un certain sens. Nous expliquons les raisons théoriques qui le justifient dans notre article [26], publié sur arXiv en 2010 : en fait, même un calcul d'issue gagnante/perdante qui n'utiliserait pas les nimbers calcule autant d'informations que l'algorithme 4.

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 4. 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 3, 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 paradoxale : l'information en apparence « complexe » des nimbers est déjà présente dans les « simples » bases de positions perdantes.

6.3.5 Comparaisons des calculs avec et sans nimber

La figure 6.4 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 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.

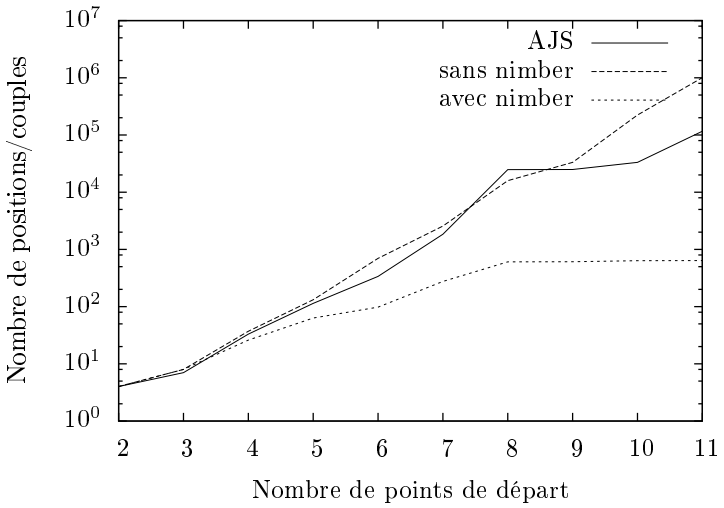


FIGURE 6.4 – Nombre de positions ou de couples perdants pour les calculs avec et sans nimbers (échelle logarithmique).

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).

6.4 Ordre des options

6.4.1 Principe

Lors du parcours en profondeur de l'arbre de jeu, que ce soit avec l'algorithme élémentaire (algorithme 3) ou avec l'algorithme des nimbers (algorithme 4), 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 [3], 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.

6.4.2 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 6.5 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 6.2.2 : 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.

6.4.3 Ordre plus complexe

Nous avons essayé plusieurs autres idées d'ordre. En particulier, l'algorithme 4 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 4.

Dans notre article [25], 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.

6.4.4 Comparaison des ordres

La figure 6.5 montre le nombre de couples (position, nimber) perdants stockés à la fin de l'algorithme 4 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}^+.

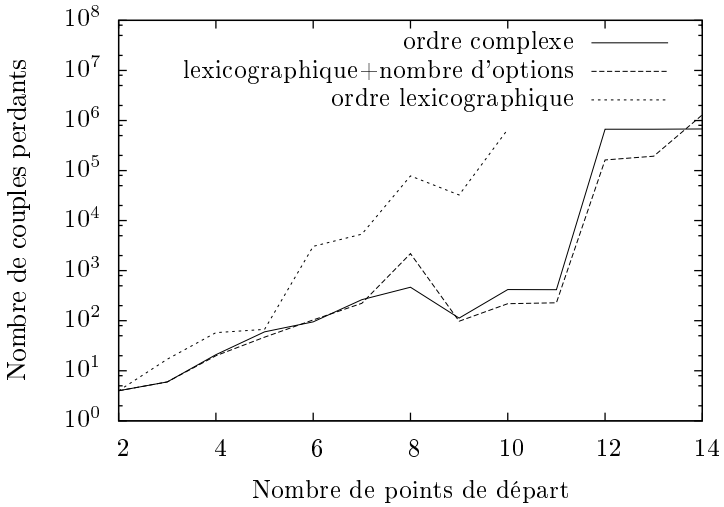


FIGURE 6.5 – Nombre de couples (position, nimber) perdants pour différents ordres possibles en fonction du nombre de points de départ.

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.

6.5 Interactions manuelles

6.5.1 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 6.5 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 4, qui permet d'afficher en temps réel la branche de calcul en cours d'étude dans l'arbre de recherche.

6.5.2 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 4. La plus grande valeur que l'algorithme 4 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.

6.6 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.

6.6.1 Algorithme misère

Nous avons détaillé dans le chapitre 3 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 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

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 6.1 – Nombres d'ACR dans les arbres de jeu des positions de départ.

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.

6.6.2 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 6.1 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 $\texttt{0*4.2} \sim \mathbb{0}$. 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 $\texttt{ABC|ABD|CE|DE} \sim \{2\}$.

6.6.3 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. L'étude détaillée des bases d'arbres canoniques obtenues peut également permettre de découvrir de nouvelles équivalences de positions.

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.

6.6.4 Premier calcul de S_{17}^-

Une fois l'ensemble des ACR de S_6 calculés, nous avons alors utilisé les algorithmes du chapitre 3 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.

6.7 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 4, 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 4.

6.8 Vérification

6.8.1 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.

6.8.2 Records d'arbres solutions en version normale

La table 6.2 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	p	taille	p	taille	p	taille	p	taille
1	1	11	113	21	5 312	31	5 463	41	42 663
2	3	12	316	22	1 581	32	58 204	42	98 947
3	6	13	369	23	1 058	33	62 389	43	98 961
4	15	14	1 017	24	5 327	34	21 107	44	99 095
5	15	15	1 986	25	2 497	35	4 265	45	?
6	46	16	669	26	4 458	36	80 001	46	80 473
7	76	17	329	27	12 768	37	80 009	47	54 542
8	139	18	1 997	28	2 549	38	80 281	...	?
9	60	19	1 736	29	2 172	39	98 905	53	73 225
10	110	20	1 831	30	12 800	40	45 782	...	?

TABLE 6.2 – Plus petits arbres solutions connus en version normale.

La figure 6.6 reprend le contenu de la table 6.2 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.

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

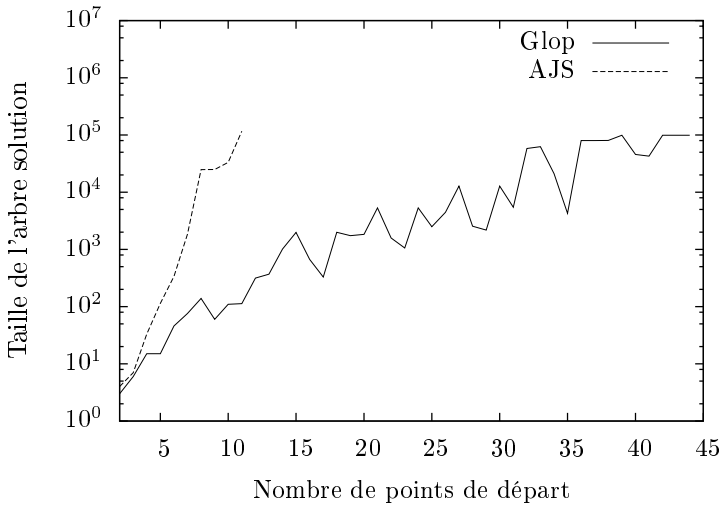


FIGURE 6.6 – Nombre de positions de l'arbre solution en fonction du nombre de points, en version normale (échelle logarithmique).

auparavant admettent en réalité un arbre solution de taille réduite, et qu'il est possible de les calculer en un temps raisonnable.

6.8.3 Records d'arbres solutions en version misère

Le tableau 6.3 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 6.3 – Nombre de positions nécessaires pour démontrer S_p^-.

Nous avons reporté le tableau 6.3 sur la figure 6.7.

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 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}^-.

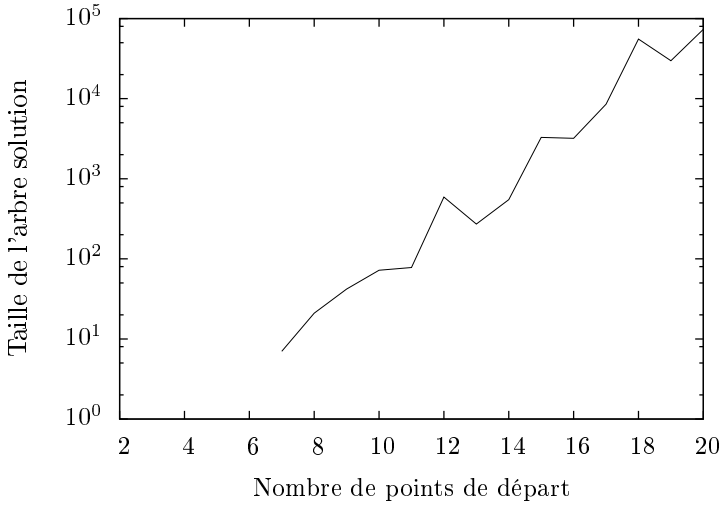


FIGURE 6.7 – Nombre de positions de l'arbre solution en fonction du nombre de points, en version misère (échelle logarithmique).

6.8.4 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.

6.9 Les conjectures du Sprouts

6.9.1 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.

6.9.2 Nouvelle conjecture misère

Le tableau 6.4 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.

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,

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 6.4 – Issues gagnantes/perdantes calculées de S_p^-.

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.*

6.9.3 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 :

$$\mathbb{1};\mathbb{0};2;\mathbb{1};\mathbb{0};\mathbb{3};\mathbb{1};\mathbb{0};\mathbb{5};\mathbb{1};\mathbb{0};\mathbb{3};\mathbb{1};\mathbb{0};\mathbb{3};\mathbb{1};\mathbb{0};\mathbb{3};\mathbb{1};\mathbb{0};\mathbb{3};\mathbb{1};\mathbb{0};\mathbb{3};\mathbb{1};\mathbb{0};\mathbb{3};\mathbb{1};\mathbb{0}$$

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

1. Mais pas insurmontable ?

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.

6.9.4 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 !

Chapitre 7

Le jeu de Cram

7.1 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 [5].

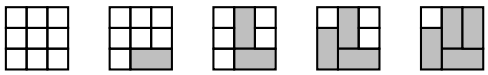


FIGURE 7.1 – Exemple de partie de Cram sur une grille 3×3 (le deuxième joueur gagne en version normale).

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, déjà abordées dans la section 6.3 du chapitre 6 pour le jeu de Sprouts, et dans le cas de la version misère celles des jeux impartiaux en version misère, décrites dans le chapitre 3.

7.2 Résultats connus

7.2.1 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 7.2 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 7.2 – 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 7.3 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.

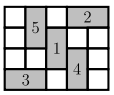


FIGURE 7.3 – Stratégie de symétrie sur un plateau de taille pair×impair.

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.

7.2.2 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 [5], 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 [12] p. 13, la version misère du Dawson's Kayles n'est par contre pas résolue entièrement.

7.2.3 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 [33], 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.

7.3 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 7.4 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.

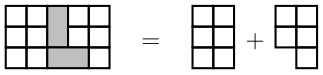


FIGURE 7.4 – Position de Cram découpable.

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 7.5 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.

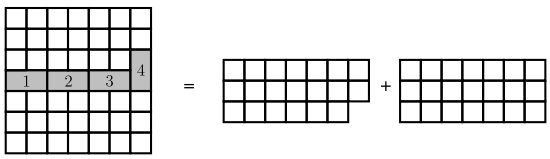


FIGURE 7.5 – Un découpage minimal du plateau de taille 7×7.

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 7.9.2 pour une définition.

7.4 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 7.6 montre un exemple de plateau de Cram et la représentation en tableau correspondante.

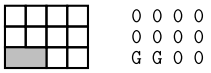


FIGURE 7.6 – Position de Cram et représentation en tableau.

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 5, sur les bases de données à la section 5.5, et sur la sérialisation à la section 5.6). 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 7.6 serait alors `00000000GG00`.

FIGURE 7.7 – 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 7.7 conduit également à `00000000GG00`, la même chaîne que la position 7.6. 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 7.6 est `0000*0000GG00` tandis que celle de la figure 7.7 est `000000*00GG00`.

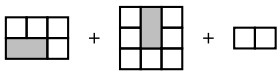


FIGURE 7.8 – Position de Cram contenant plusieurs composantes.

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 7.8 est alors représentée par la chaîne `000*GG0E0G0*0G0000E00*E`.

7.5 Canonisation

7.5.1 Symétries

Les positions de Cram sont équivalentes à symétrie près. Dans le cas général, comme sur la figure 7.9, 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.

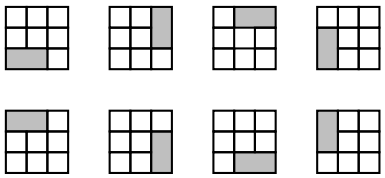


FIGURE 7.9 – Les 8 symétries d'une position de Cram.

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.

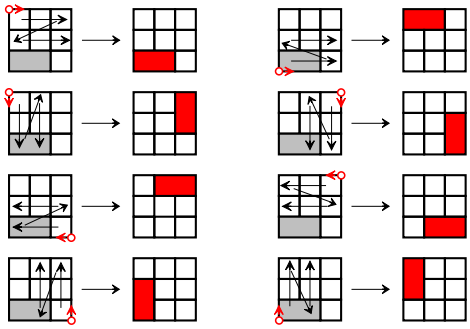


FIGURE 7.10 – Parcours du plateau correspondants aux 8 symétries.

Il y a en fait moyen de calculer la symétrie canonique bien plus efficacement. La figure 7.10 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 7.10.

L'algorithme 5 décrit le calcul rapide de la symétrie canonique d'une position $\mathscr{P}$.

7.5.2 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 7.11, la position de droite est équivalente à celle de gauche.

Algorithme 5 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**

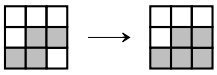


FIGURE 7.11 – Position équivalente en grisant les cases isolées.

7.5.3 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 7.11, obtenue après grisage des cases isolées, contient une ligne inutile entièrement grise. La figure 7.12 montre la position équivalente obtenue après suppression de cette ligne inutile.

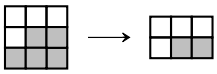


FIGURE 7.12 – Position équivalente en supprimant une ligne inutile.

7.5.4 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 6.

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.

7.6 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 6 Canonisation d'une position $\mathscr{P}$

```
Pour chaque composante de P faire
    griser les cases isolées désormais inutilisables
    découper si possible la composante en composantes indépendantes
fin Pour
Pour chaque composante de P faire
    supprimer les lignes ou colonnes inutiles (celles entièrement grises)
    calculer la symétrie canonique
fin Pour
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.

7.6.1 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.

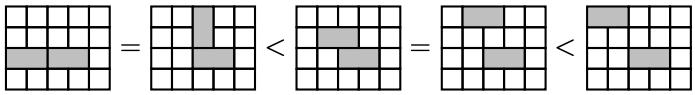


FIGURE 7.13 – Longueur de la ligne de coupe minimale de différentes positions.

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 7.13 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.

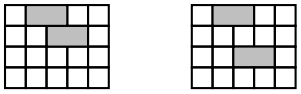


FIGURE 7.14 – Positions avec une même longueur de coupe.

Cependant, cet ordre pour favoriser l'apparition des découpages pourrait être amélioré. Par exemple, les deux positions de la figure 7.14 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.

7.6.2 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 [9] 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.

7.7 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.

7.7.1 Résolution forte

Le tableau 7.1 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 7.2 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	1 222	286	64
3×7	5 011	1 070	170
3×8	20 445	4 152	1 191
3×9	83 418	14 889	5 145

TABLE 7.1 – 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	3 749	874	205
4×6	29 300	6 401	2 136
5×5	32 221	7 540	1 780

TABLE 7.2 – 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.

7.7.2 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 7.15 réunit les résultats des deux tableaux 7.1 et 7.2, 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.

7.7.3 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

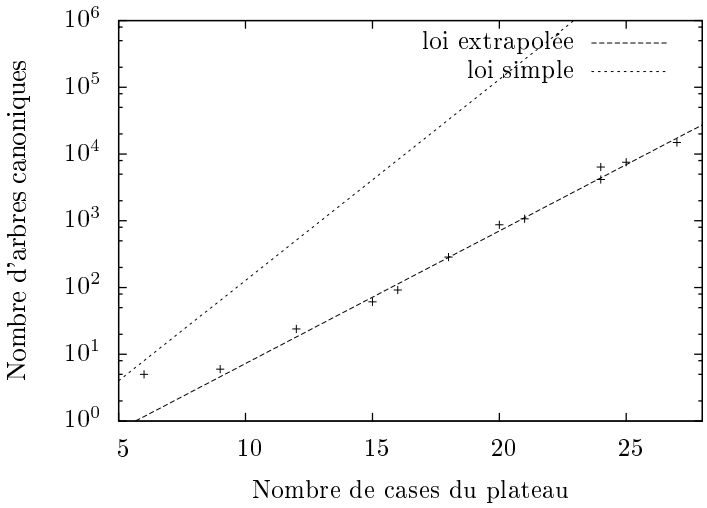


FIGURE 7.15 – Nombre d'arbres canoniques en fonction du nombre de cases du plateau (échelle logarithmique).

dans la section 7.5 était parfaite, le nombre de positions canonisées serait égal au nombre d'arbres canoniques.

La figure 7.16 montre le nombre de positions canonisées en fonction du nombre de cases du plateau (en utilisant les données des tableaux 7.1 et 7.2). 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 7.17 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* [5] p. 502–504, qui permettraient de découper plus rapidement les positions.

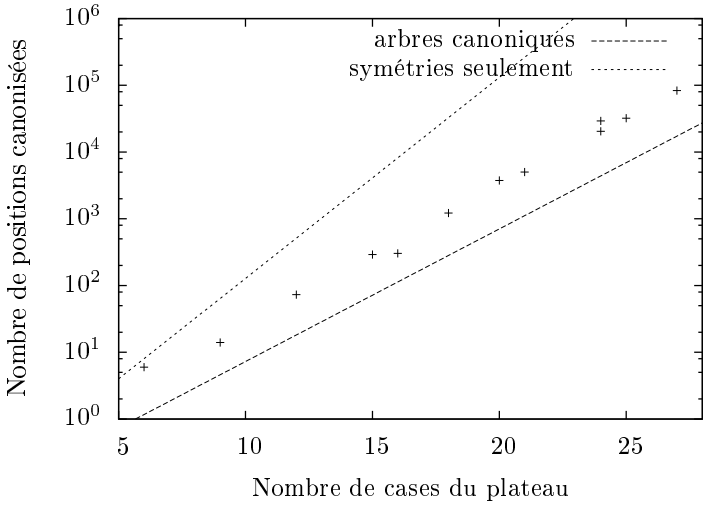


FIGURE 7.16 – Nombre de positions canonisées en fonction du nombre de cases du plateau (échelle logarithmique).

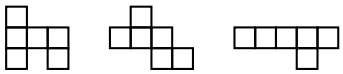


FIGURE 7.17 – Positions de Cram équivalentes.

7.8 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 [33] que nous avons indiqués par un astérisque dans les tableaux.

7.8.1 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 7.3 – 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.

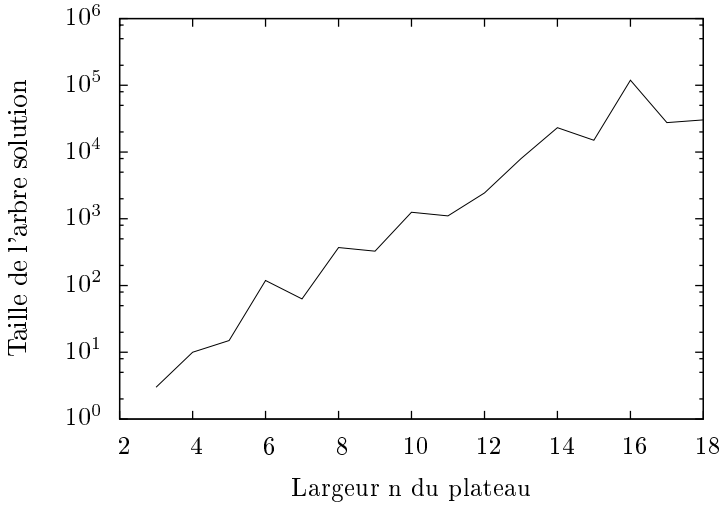


FIGURE 7.18 – Nombre de positions de l'arbre solution du nimber des plateaux $3 \times n$ (échelle logarithmique)

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

7.9 Calculs en version misère

7.9.1 Choix des arbres canoniques réduits

Dans le cas de la version misère, nous avons appliqué les techniques du chapitre 3, 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 7.1 et 7.2 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.

7.9.2 Valeur de Grundy misère

La *valeur de Grundy misère* est définie dans *ONAG* [10] p. 140, ou dans *Winning Ways* [5] p. 422.

Définition 13. *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 3. *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 3, 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 [33], 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.

7.9.3 Résultats

Comme pour la version normale du jeu, les meilleurs résultats connus précédemment en version misère étaient ceux de Martin Schneider en 2009 [33], 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 7.5 – 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 7.6 – Résultats misère obtenus sur les plateaux de taille $n \times m$, avec $n \geq 4$ et $m \geq 4$.

7.10 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 8

Conclusion

8.1 Résultats obtenus

Nous revenons tout d'abord sur les principaux résultats de calculs obtenus jusqu'ici dans le cadre de cette thèse.

8.1.1 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-bêta 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.

8.1.2 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).

8.2 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.

8.2.1 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, il 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.

8.2.2 Représentation des arbres de jeu

La notion d'arbre de jeu est définie au paragraphe 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.

8.2.3 Calcul distribué

Les calculs que nous avons menés sont en partie distribués dans le sens où nous avons fréquemment effectué 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[31].

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.

8.3 Autres pistes de recherche

8.3.1 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.

8.3.2 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-bêta 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.

8.4 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, créent l'image populaire de l'ordinateur omnipotent, et ont 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 nécessaires 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 soient 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.

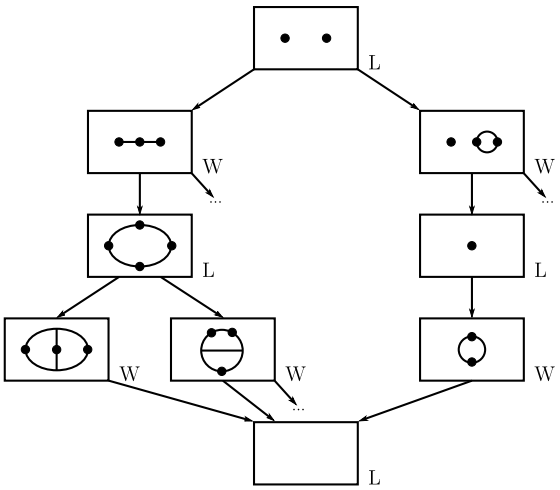


FIGURE A.1 – Arbre solution pour le jeu de Sprouts à deux points.

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 grâce à des considérations topologiques. 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.2, 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.

La table B.1 explique à quelles positions correspondent les numéros des nœuds de la figure B.2. 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 de l'article [25] 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 [3], 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 l'article [25] 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, qui consiste à effectuer des vérifications successives avec un ordre aléatoire des options pour détecter certaines transpositions, 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.

L'étude des positions de Sprouts avec un grand nombre de points de départ ressemble

FIGURE B.1 – Positions 13f et 11e (difficiles à étudier).

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`.

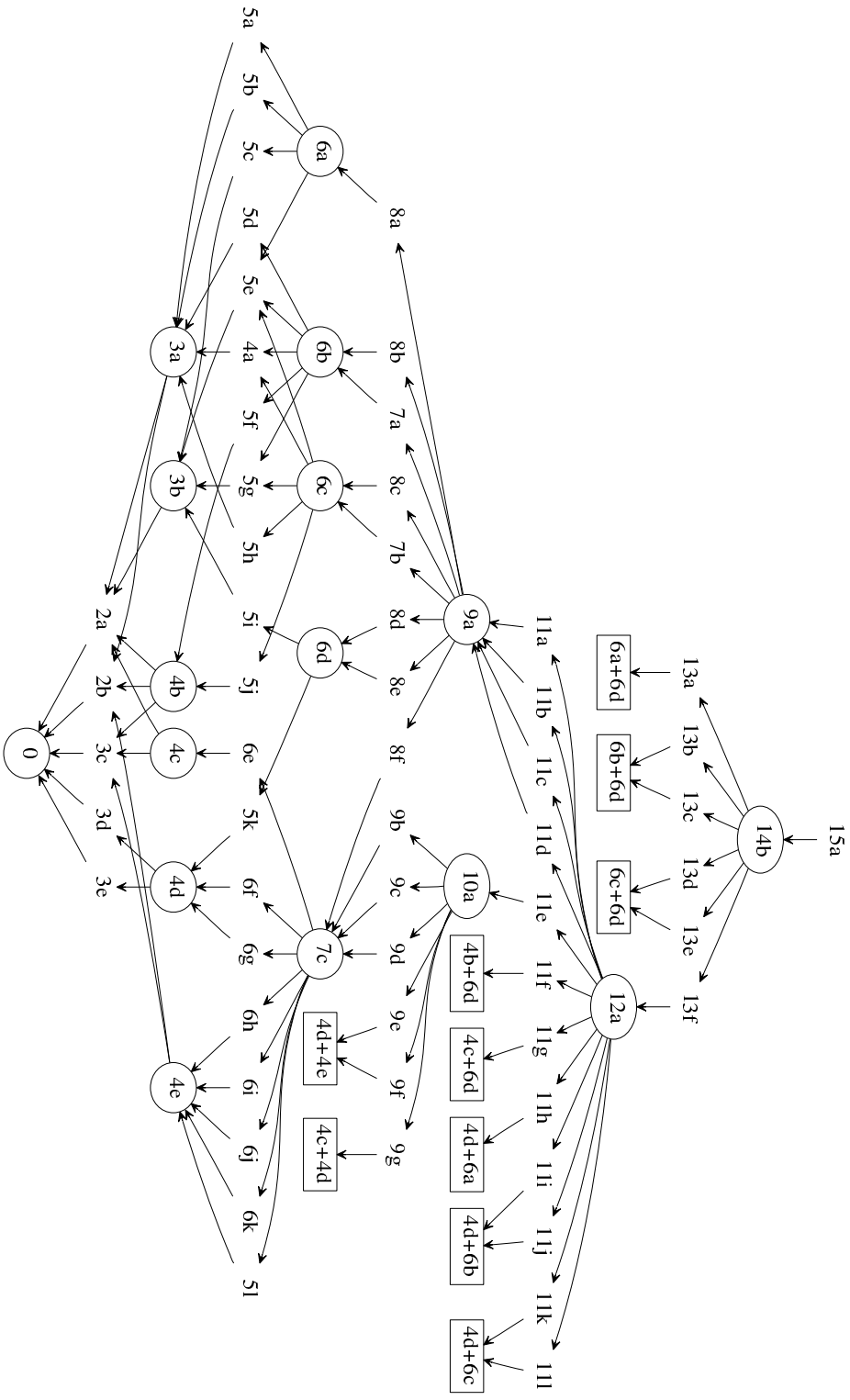


FIGURE B.2 – Arbre solution pour le jeu de Sprouts à 5 points.

#	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.2.

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*.

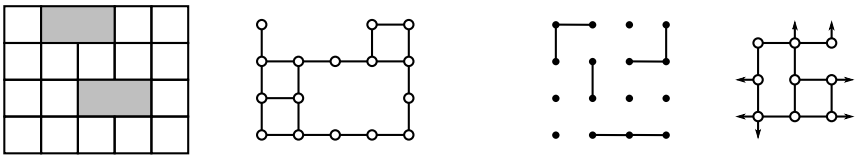


FIGURE C.1 – Positions de Cram et de Dots-and-boxes, et leurs graphes duaux.

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 7) qui ont le même graphe dual, représenté à droite de la figure. Ces trois positions sont donc équivalentes.

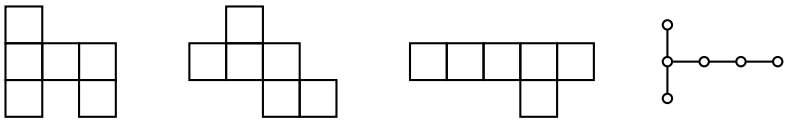


FIGURE C.2 – Trois positions de Cram avec le même graphe dual.

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 14. *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 15. *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.*

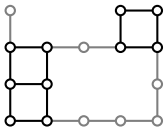


FIGURE C.3 – Graphe à deux tablettes.

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.

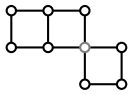


FIGURE C.4 – Sommet commun à deux tablettes.

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.

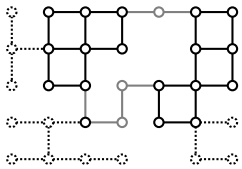


FIGURE C.5 – Graphe à 3 tablettes, 3 filaments et 3 arbres.

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.

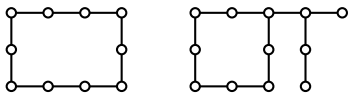


FIGURE C.6 – Cycle isolé, et cycle à une extrémité.

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).

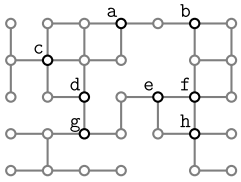


FIGURE C.7 – Sommets désignés par des lettres dans un graphe.

Chocolat

Pour chaque tablette, on peut utiliser une représentation sous forme de tableau, de même type que celle décrite dans le chapitre 7 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{{{}{{{}{{}{}}}}{{{}{}}}}}
```

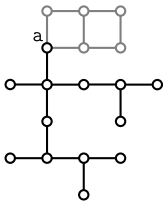


FIGURE C.8 – Arbre contenu dans un graphe.

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 l'article [25]). 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] 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.

[4] Elwyn Berlekamp, *The dots-and-boxes game : sophisticated child's play*, A K Peters, 2000.

[5] Elwyn Berlekamp, John Conway, and Richard Guy, *Winning ways for your mathematical plays*, A K Peters, 2001.

[6] C. L. Bouton, *Nim, a game with a complete mathematical theory*, Ann. of Math. **3** (1902), 35–39.

[7] Dennis Breuker, Jos Uiterwijk, and Jaap van den Herik, *Solving* 8×8 *domineering*, Theoretical Computer Science **230** (2000), 195–206.

[8] Dennis M. Breuker, *Memory versus search in games*, Ph.D. thesis, Universiteit Maastricht, 1998.

[9] Nathan Bullock, *Domineering : solving large combinatorial search spaces*, Master's thesis, University of Alberta, Canada, 2002.

[10] John H. Conway, *On numbers and games (second edition)*, A K Peters, 2001.

[11] Jean-Paul Delahaye, *Le jeu des pousses*, Pour la Science **371** (Septembre 2008), 90–95.

[12] Robert A. Hearn Erik D. Demaine, *Playing games with algorithms : algorithmic combinatorial game theory*, Games Of No Chance **3** (2008).

[13] Jonathan Schaeffer et Robert Lake, *Solving the game of checkers*, Game of No Chance **1** (1996), 119–133.

[14] Martin Gardner, *Mathematical games : of sprouts and brussels sprouts, games with a topological flavor*, Scientific American **217** (July 1967), 112–115.

[15] Richard Gillam, *The anatomy of the assignment operator*, `http://icu-project.org/docs/papers/cpp_report/the_anatomy_of_the_assignment_operator.html`, 1997.

[16] Ralph Grasser, *Solving nine men's morris*, Games of No Chance **1** (1996), 101–113.

[17] P. M. Grundy, *Mathematics and games*, Eureka **2** (1939), 6–8.

[18] 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.

[19] Roman Khorkov Josh Jordan Purinton, *Computation of misere sprouts with 15 spots*, `http://groups.google.com/group/sprouts-theory/browse_thread/thread/effb79c5fa99d096`, 2007.

[20] ———, *Computation of misere sprouts with 16 spots*, http://groups.google.com/group/sprouts-theory/browse_thread/thread/637b123af0c95f45, 2009.

[21] Peter Kissmann, *Symbolic search in planning and general game playing*, http://users.cecs.anu.edu.au/~ssanner/ICAPS_2010_DC/Abstracts/kissmann.pdf, 1950.

[22] Julien Lemoine and Simon Viennot, *A further computer analysis of sprouts*, http://download.tuxfamily.org/sprouts/sprouts-lemoine-viennot-070407.pdf, 2007.

[23] ———, *Sprouts game on compact surfaces*, http://arxiv.org/abs/0812.0081, 2008.

[24] ———, *Analysis of misère Sprouts game with reduced canonical trees*, http://arxiv.org/abs/0908.4407, 2009.

[25] ———, *Computer analysis of sprouts with nimbers*, http://arxiv.org/abs/1008.2320, 2010.

[26] ———, *Nimbers are inevitable*, http://arxiv.org/abs/1011.5841, 2010.

[27] Richard J. Nowakowski, *The history of combinatorial game theory*, http://www.mathstat.dal.ca/~rjn/papers/HistoryCGT.pdf, 2009.

[28] Thane E. Plambeck, *Taming the wild in impartial combinatorial games*, INTEGERS : Electronic Journal of Combinatorial Number Theory **5** (2005), G5.

[29] Josh Jordan Purinton, *Computation of normal sprouts with 14 spots*, http://groups.google.com/group/sprouts-theory/browse_thread/thread/191ff66a8db8c13f/4620c2e11248e264, 2006.

[30] Jonathan Schaeffer, *The games computers (and people) play*, Advances in computers **52** (2000), 189–266.

[31] Jonathan Schaeffer and al., *Checkers is solved*, Science **317** (2007), 1518–1522.

[32] Dierk Schleicher and Michael Stoll, *An introduction to conway's games and numbers*, http://arxiv.org/abs/math/0410026, 2005.

[33] Martin Schneider, *Das spiel juvavum*, Master's thesis, 2009, http://www.mschneider.cc/papers/masterthesis.pdf.

[34] Claude E. Shannon, *Programming a computer for playing chess*, Philosophical Magazine **41** (1950).

[35] Aaron N. Siegel, *Misère games and misère quotients*, http://arxiv.org/abs/math.CO/0612616, 2006.

[36] R. Sprague, *Uber mathematische kampfspiele*, Tohoku Math. J. **41** (1935-36), 438–444.

[37] John Tromp, *John's connect four (web page)*, http://homepages.cwi.nl/~tromp/c4/c4.html.

[38] John Tromp and Gunnar Farnebäck, *Combinatorics of go*, http://homepages.cwi.nl/~tromp/go/gostate.ps, 2009.

[39] David Wilson, *Dots-and-boxes analysis*, http://homepages.cae.wisc.edu/~dwilson/boxes/, 2002.