UNIVERSITÉ PARIS 8 - VINCENNES-SAINT-DENIS
U.F.R. 6

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PARIS 8
Spécialité Informatique

Bernard Helmstetter

Analyses de dépendances et méthodes de Monte-Carlo dans les jeux de réflexion

Jury

Bruno Bouzy	Rapporteur	Université Paris 5
Jean-Paul Delahaye	Rapporteur	Université Lille 1
Tristan Cazenave	Directeur	Université Paris 8
Rémi Coulom	Examinateur	Université Lille 3
Michel Gondran	Examinateur	Conseiller scientifique EDF
Patrick Greussay	Examinateur	Université paris 8

Résumé

Nous explorons deux familles de techniques de programmation des jeux de réflexion : des techniques de Monte-Carlo et des techniques de recherche permettant d'exploiter la faible dépendance entre différentes parties d'un jeu. Ces techniques sont appliquées à des jeux à un ou deux joueurs : un jeu de patience à un joueur appelé Montana, le jeu de Go, et un puzzle à un joueur appelé le Morpion solitaire.

Sur le plan théorique, nous formalisons quelques notions : celles de jeu, de coup, et d'indépendance entre sous-jeux. Ces notions sont définies différemment pour des jeux à un et à deux joueurs. À un joueur, un sous-jeu est typiquement défini par les coups qui sont permis ; à deux joueurs, il est défini par un but à atteindre. Dans l'idéal, rechercher un jeu en le décomposant en sous-jeux peut permettre une large réduction du nombre de nœuds cherchés, mais ceci est au prix d'un travail supplémentaire pour étudier les dépendances entre les sous-jeux. Le problème de la décomposition en sous-jeux est aussi lié à celui de la détection des transpositions. Nous présentons un algorithme que nous appelons algorithme de transpositions incrémentales, et qui vise à détecter une certaine classe de transpositions avec des ressources mémoire très faibles.

Le premier domaine d'étude est une patience du nom de Montana, déterministe et à information complète. Nous l'abordons par des algorithmes de recherche classiques, et nous observons de particulièrement bonnes performances pour un algorithme d'échantillonnage itératif. Nous analysons ensuite des algorithmes de décomposition en sous-jeux : nous décrivons et évaluons un algorithme que nous appelons *recherche par blocs*.

Le domaine suivant est le jeu de Go. Nous étudions des algorithmes de recherche dans des problèmes tactiques décomposables en sous-problèmes : la transitivité des connexions. Les problèmes sont abordés par une combinaison de recherche à base de menace et d'un Aplha-Beta au niveau transitif. Toujours sur le jeu de Go, mais en l'abordant dans sa globalité, nous développons l'approche de Monte Carlo. En partant des travaux antérieurs, nous simplifions la méthode et étudions quelques améliorations. En particulier, nous combinons avec la méthode avec des algorithmes de recherche, en faisant des statistiques en fonction de buts associés à ces recherches. Les performances sont particulièrement bonnes compte tenu de la simplicité de la méthode.

Le troisième domaine d'étude est le Morpion Solitaire. Notre contribution principale est l'application de l'algorithme des transposition incrémentales, combiné à une parallélisation de la recherche, et qui mène à un record pour une variante de ce jeu.

Table des matières

Chapitre 1

Introduction

Nous explorons deux familles de techniques de programmation des jeux de réflexion : des techniques basées sur l'utilisation du hasard (ou techniques de Monte-Carlo), et des techniques de recherche permettant d'exploiter la faible dépendance entre différentes parties d'un jeu. Ces techniques sont appliquées à des jeux à un ou deux joueurs : le jeu de Go, un jeu de patience à un joueur appelé Montana, et une sorte de puzzle à un joueur appelé le Morpion solitaire. Ces deux familles de techniques, sans être vraiment nouvelles dans le domaine des jeux, ne font pas partie des techniques les plus fréquemment utilisées, qui sont des techniques reposant sur la rapidité de calcul des ordinateurs et que nous appelons des techniques de force brute. Nous expliquons dans un premier temps les raisons d'étudier de telles techniques, puis les raisons de nos choix de domaines d'application.

1.1 Recherche, connaissances, et leurs combinaisons

Il est généralement possible de séparer le contenu d'un moteur de jeu en deux parties : le code dédié à la recherche, et le code ou les données dédiées aux connaissances.

La recherche recouvre principalement des algorithmes de recherche arborescentes, tels que les recherches Alpha-Beta pour les jeux à deux joueurs ([54], chap. 6), ou les recherches IDA^* pour les jeux à un joueur ([54], chap. 4). Nous y classons aussi la génération automatique de bases de données, comme l'analyse rétrograde utilisée pour la génération de bases de données de finales aux échecs ou aux dames. Certains jeux ont été parfaitement résolus uniquement par la recherche : Nine Men's Morris [31] ; le Go 5x5 [64] ; l'Awari [53] ; les Dames anglaises seront probablement resolues dans quelques années [57].

Les connaissances peuvent avoir la forme de code ou de données. Quand il s'agit de code, c'est généralement du code peu coûteux à exécuter. Des exemples typiques sont les fonctions d'évaluation dans une recherche Alpha-Beta, ou la

fonction heuristique dans une recherche en IDA^*. D'autres exemples sont les motifs (*patterns*) utilisés au Go pour indiquer où jouer dans des positions particulières. Plus généralement, un programme de jeu comporte souvent de petits bouts de code destinés à optimiser des cas particuliers ; c'est une façon de donner des connaissances à un programme.

En première analyse, les ordinateurs sont très bons pour la recherche, alors que les humains ont l'avantage au niveau des connaissances. Souvent, les connaissances qu'on essaye de donner en premier à des programmes sont copiées sur les raisonnements humains. Ceci est cependant à relativiser à cause de l'existence de méthodes d'acquisition automatique de connaissances. Il est parfois possible de régler divers paramètres par des méthodes de différences temporelles ou par des algorithmes génétiques, en faisant jouer des programmes entre eux ou en analysant des parties de maître. Le cas du backgammon est particulièrement frappant : il est possible de construire et d'entraîner par apprentissage non supervisé un réseau de neurones pour l'évaluation positionnelle ; la qualité de cette fonction d'évaluation suffit à donner un niveau expert au programme, la recherche étant presque absente [62].

Au delà de cette distinction entre recherche et connaissance, il y a la façon dont ces deux aspects s'agencent dans un programme de jeu. Certains algorithmes sont particulièrement attractifs parce qu'ils permettent une séparation claire entre la recherche et les connaissances. L'Alpha-Beta est dans ce cas. La recherche arborescente peut être optimisée avec un certain nombre de techniques bien étudiées et largement indépendantes du domaine. Les connaissances interviennent principalement dans la fonction d'évaluation. Pour certains jeux comme les diverses variantes d'échecs ou de dames, l'Alpha-Beta est presque le seul algorithme utilisé en pratique. Dans le domaine des jeux à un joueur, les algorithmes A^* ou IDA^* ont des avantages similaires. Les connaissances interviennent essentiellement dans la fonction heuristique qui évalue le nombre minimal de coups nécessaires pour atteindre une solution. D'une certaine façon, un « bon » algorithme est un algorithme dont au moins la partie « centrale » est très simple. Ainsi, nous pouvons dire que même les meilleurs programmes d'échecs restent « simples » (ceci malgré, par exemple, les 45.000 lignes de code de Crafty, un des meilleurs programmes libres d'échecs), puisque tout reste subordonné à une recherche centrale Alpha-Beta. En comparaison, un programme qui doit assembler les résultats de recherches hétérogènes pour évaluer une position ou choisir un coup est « compliqué » ; sa mise en oeuvre et son réglage sont difficiles.

Un algorithme en IDA^* a par exemple été appliqué au jeu de Sokoban par A. Junghanns [42] avec de bon résultats ; ceci bien que l'approche utilisée soit très différente du genre de techniques utilisées par des joueurs humains pour résoudre les mêmes problèmes. Le programme en IDA^* aborde le problème par une recherche globale, alors que les joueurs humains cherchent plus à décomposer le problème en sous-problèmes localisés dans l'espace. Ceci illustre l'avantage à utiliser des méthodes adaptées aux ordinateurs, et non à transposer les méthodes des joueurs humains.

Parfois, les méthodes de recherche utilisées sont moins « monolithiques », et

il n'y a pas d'algorithme simple auquel tout est soumis. Un exemple frappant est le jeu de Go. D'autres exemples, dans une moindre mesure, sont le Shogi ou Amazons. Le fait qu'un jeu soit abordé par des techniques compliquées est un signe de sa difficulté, dans le sens où les techniques simples n'ont pas suffi. Il est bien sûr possible que des algorithmes simples et efficaces pour ces jeux n'aient pas encore été trouvés. On peut citer l'exemple du Bridge où les techniques utilisées actuellement font plus appel à la recherche (dans ce cas, une combinaison de recherche arborescente et de simulations de Monte-Carlo) ; elles sont plus simples et plus efficaces que les techniques précédentes basées sur les connaissances.

Notre travail, par rapport à la distinction entre recherche et connaissances, est résolument placé du côté de la recherche. Par rapport à l'aspect monolithique ou non des programmes, il se situe des deux côtés. Certaines des techniques, principalement des techniques de Monte-Carlo, visent à simplifier les programmes. Pour d'autres techniques, nous acceptons qu'il faut parfois des algorithmes de recherches bien adaptés, même si ils sacrifient la simplicité.

1.2 Méthodes basées sur l'analyse de dépendances

Dans le domaine des jeux, nous appelons *méthodes de dépendances* des méthodes de recherche qui analysent les relations entre les coups ou des ensembles de coups, et s'en servent d'une façon ou d'une autre pour améliorer la recherche. Ce genre de méthodes cherche à exploiter des particularités de la structure du graphe des états, plutôt que de le rechercher comme un graphe amorphe. Notre travail est en partie inspiré des travaux de V. Allis sur un algorithme appelé *dependency-based search* [2]. Il a des similarités avec des algorithmes de recherche sélective pour les jeux à deux joueurs, du type des *menaces généralisées* [20]. L'étude de ces méthodes est la motivation première de notre travail.

Une sous-classe de ces méthodes consiste en ce que nous appelons des *méthodes de recherche par décompositions en sous-jeux*. Ces sous-jeux ne sont en général pas totalement indépendants, mais suffisament pour qu'il puisse être avantageux de les rechercher séparément, et d'utiliser ces recherches locales pour aider la recherche globale. Ce genre de situations arrive assez fréquemment. Un jeu où ce genre de raisonnement est particulièrement attractif est le jeu de Go. D'autres exemples sont Sokoban ou Amazons. À l'inverse, des jeux commes d'échecs ou de dames peuvent difficilement être abordés ainsi, principalement parce que l'espace du plateau de jeu est trop petit pour qu'une décomposition spatiale soit utile, et à cause des pièces à longue portée.

En fait, la quasi-totalité des programmes de Go actuels font des recherches locales et s'en servent pour évaluer la position globale. Cependant, l'évaluation globale ne consiste finalement qu'en un nombre, les détails des recherches locales n'étant pas réutilisés dans la recherche globale (si du moins elle a lieu, tous les programmes de Go ne le faisant pas). Là où notre recherche sur les dépendances

est novatrice (dans les applications au Go ou ailleurs), c'est que l'analyse globale utilise les recherches locales tout en restant au niveau de la recherche et non des connaissances.

Une première remarque est que l'idée d'analyser un problème en le décomposant en sous-problèmes est typiquement une tentative de transformer en programme les raisonnements humains. Ceci est plutôt annonciateur de difficultés : les raisonnements humains sont souvent mal formalisés, et peu adaptés aux ordinateurs dont ils n'utilisent pas les forces. Il nous semble cependant qu'un but tel que l'évaluation d'une position de Go est une motivation suffisante pour entreprendre de telles recherches.

Une deuxième remarque est que l'idée en question est similaire à des méthodes très courantes en algèbre : étudier une structure algébrique en tentant de la décomposer en structures plus simples. On peut penser par exemple à la décomposition de groupe en produits directs ou semi-directs, ou bien à la diagonalisation de matrices ou d'un ensemble de matrices simultanément. Il y a même un sous-domaine de la programmation des jeux qui est dans l'intersection avec l'algèbre, il s'agit de la théorie combinatoire des jeux [5]. Cette théorie donne un sens algébrique précis à une *somme* de sous-jeux.

Cependant, nous devons préciser que notre recherche est en fait assez éloigné de ce genre de décompositions algébriques. Les problèmes que nous considérons n'admettent pas de décomposition aussi propre. Afin de mieux définir notre domaine de travail, nous précisons qu'il s'agit de décomposition d'un jeu en parties relativement indépendantes, pour laquelle l'analyse au niveau global reste au niveau de la recherche et non des connaissances.

On remarque enfin que des approches semblables sont utilisées en planification. Les problèmes sont donnés dans des langages tels que STRIPS [29] qui définissent précisément les conditions d'application des différentes actions. Des planificateurs tels que Graphplan utilisent la structure des problèmes ainsi définis. Peut-on s'inspirer des méthodes de planification pour la programmation des jeux ? Notre avis est que le genre de problèmes abordés en planification est assez différent que ce que nous considérons comme des jeux, et les méthodes sont difficilement transposables. Malgré le fait qu'un jeu à un joueur peut toujours être défini comme un problème de planification, un domaine « ludique » a quelque chose de différent en pratique.

1.3 Techniques de Monte-Carlo

Si l'analyse de dépendances est la motivation première, nous avons été amenés à plusieurs reprises à étudier des méthodes de Monte-Carlo pour étudier les mêmes jeux.

Par rapport aux techniques d'analyse de dépendances, les techniques de Monte-Carlo sont à l'opposé sur plusieurs points : l'analyse est directement au niveau global au lieu de chercher à décomposer le jeu ou a exploiter des particularités de ses règles ; elle fournit des résultats probabilistes plutôt que des analyses complètes.

Il est facile de sous-estimer l'efficacité des méthodes de Monte-Carlo. Notre travail en donnera des exemples. Les méthodes de recherche arborescente peuvent sembler plus systématiques et plus sûres. Cependant, du fait de l'explosion exponentielle du nombre de nœuds avec la profondeur, l'horizon d'une recherche arborescente est souvent limité. L'utilisation du hasard apporte plus de souplesse, permet une évaluation rapide des choix de recherche qui sont effectués et permet de mieux aborder les problèmes dans leur globalité.

1.4 Domaines d'application

Notre travail porte sur le domaine des jeux de réflexion [1]. Dans ce domaine, les jeux peuvent être classés selon divers critères, tels que le nombre de joueurs, jeu à somme nulle ou non, jeu de hasard ou déterministe, jeu à information complète ou non. La difficulté de la programmation du jeu dépend souvent de critères tels que le facteur de branchement, la taille de l'espace de recherche, ou celle de l'espace des états [2]. On peut encore qualifier les jeux selon le type de raisonnement par lesquels les joueurs humains les approchent : certains nécessitent une bonne capacité de lecture, d'autres font appels à des concepts plus stratégiques comme une bonne évaluation positionnelle.

Le jeu de Go est central dans notre travail. Nous en abordons d'autres, parce qu'ils sont intéressants en eux-mêmes, et aussi parce qu'ils permettent d'utiliser des techniques similaires, et ainsi de contourner les difficultés que présentent ce jeu.

Nous ne nous concentrons pas spécialement sur une sous-classe de jeux. Plutôt, la sélection des jeux étudiés a été faite selon la possibilité d'appliquer les techniques que nous développons. Nous recherchons la performance, mais nous sommes limités par la complexité nécessaire pour programmer efficacement certains jeux complexes. Ainsi, quand nous abordons le jeu de Go, nous le faisons en délimitant une classe de sous-problème du Go, ou bien nous abordons le jeu globalement par des algorithmes simples, dont l'efficacité est bonne compte tenu de leur simplicité mais limitée par rapport à des programmes plus complexes.

Dans la pratique, il y a des limites aux jeux que nous considérerons : il s'agit de jeux à un ou deux joueurs, déterministes, à information complète. Nous considérons les jeux suivants :

– Une patience (jeu à un joueur utilisant des jeux de cartes) appelé *Montana*. L'intérêt de ce jeu est d'être à la fois simple à aborder et de posséder des caractéristiques rendant appropriée une analyse basée sur une décomposition en sous-jeux. Simultanément, nous considérons aussi des approches en Monte-Carlo.

[1] jeux de réflexion par opposition principalement aux jeux vidéos ; même si ces jeux incorporent de plus en plus des IA développées, les buts recherchés sont suffisamment différents pour faire une distinction nette.

[2] Il est souvent écrit, par exemple, qu'une raison de la difficulté de la programmation du Go tient au grand nombre de positions possibles : environ 10^{170} (des estimations très précises ont été réalisées [63]) contre 10^{50} pour les échecs ou 10^{17} pour les checkers [13].

- Le jeu de Go. Nous nous intéressons au problème de transitivité des connexions : déterminer par la recherche si deux connexions entre des chaînes A et B d'une part, B et C d'autre part résultene en une connexion entre A et C. Simultanément, nous nous intéressons à des approches en Monte-Carlo résultant en un programme jouant des parties entières sur 9x9.
- Le *Morpion solitaire*, qui est un jeu à un joueur. Nous montrons l'applicatino de plusieurs méthodes de dépendances. Les meilleurs résultats à ce jeu ont été obtenus par des méthodes de Monte-Carlo, et sont antérieures à notre travail.

Le chapitre 2 présente des méthodes générales, alors que les trois chapitres suivants portent sur chacun des jeux en particulier, dans l'ordre ci-dessus : Montana, jeu de Go, Morpion solitaire. Le chapitre deux est particulièrement important pour l'exposé de la méthode des transpositions incrémentales, utilisée dans les chapitres trois et cinq. Il fournit aussi quelques définitions utiles, et un état de l'art des techniques de Monte-Carlo dans les jeux. En dehors de cela, les chapitres 3, 4 et 5 peuvent être lus dans un ordre quelconque.

On trouvera en annexe la présentation d'un programme de *Lines of action*, développé originellement pour tester certaines idées de notre travail sur des algorithmes Alpha-Beta, et qui a obtenu de bons résultats en tournois.

Chapitre 2

Formalisation et méthodes générales

Nous présentons dans ce chapitre les états de l'art, techniques et définitions communes aux différents domaines d'applications que nous exposerons dans les chapitres suivants. Dans le cadre de ce que nous avons appelé recherches de dépendances, nous faisons notamment un exposé complet d'une méthode que nous appelons *transpositions incrémentales*, permettant dans certains cas de détecter des transpositions dans un jeu en utilisant moins de ressources mémoires qu'une table de transposition.

2.1 Analyses de dépendances

Pour notre travail d'analyse de dépendances, nous avons besoin de donner un cadre formel adapté à nos domaines d'application. Ce formalisme concerne surtout les jeux à un joueur, sauf mention contraire.

2.1.1 Définitions

Nous avons besoin de définir ce que nous entendons par *jeu*, *position* d'un jeu ou *coup* dans un jeu. Un des tous premiers livres sur les jeux, le *Theory of Games and Economic Behavior* de J. von Neumann [65], présente un cadre général, valable pour des jeux très généraux, non déterministes, à information incomplète et à n'importe quel nombre de joueurs. Ces travaux, essentiellement théoriques, laissent de côté certains aspects des jeux, du moins tels que nous les considérons. Pour nous, un jeu prend généralement place, concrètement, dans un espace de faible taille (un échiquier ou un goban par exemple), avec des règles précises qui donnent une certaine structure à l'espace des états.

Par exemple, le terme de coup dans un jeu est ambigu. À quelles conditions peut-on dire que deux coups, joués dans des positions différentes, sont les mêmes ? Un formalisme dans lequel un coup ne peut être joué que dans une

position précise est peu intéressant. D'un autre côté, il est souhaitable que le même coup, joué dans des positions différentes, produise les mêmes effets. Par exemple, considère-t-on que deux coups au Go sont les mêmes, si ils sont joués sur la même intersection, mais que l'un capture une chaîne et pas l'autre ? C'est le cas dans notre formalisme. Cependant, le formalisme utilisé dans [20] tend plutôt vers l'autre solution.

Les définitions que nous donnons sont proches de celles de STRIPS ([29]), mais plus restrictives et mieux adaptées aux jeux que nous considérerons. Remarquons que nous appelons *position* et *coup* ce qui est appelé *état* et *opérateur* dans la terminologie de STRIPS.

Position

Nous représentons une position dans un jeu (à un ou plusieurs joueurs) comme un élément d'un produit d'ensembles, ceux-ci étant indexés par un ensemble I :

$$E = \prod_{i \in I} E_i$$

Notons que, parmi les éléments de E, tous ne correspondent pas forcément à des positions légales.

Cette définition convient à tous les jeux que nous considérerons. Par exemple, pour le jeu de Go, on pourrait avoir (sans tenir compte des règles de *ko*) :

$$\forall x, y \in \{1, 2, \ldots, 19\}, E_{x,y} = \{\text{Vide}, \text{Blanc}, \text{Noir}\},$$
$$E_{\text{joueur}} = \{\text{Blanc}, \text{Noir}\},$$
$$E_{\text{ko}} = \{1, 2, \ldots, 19\} \times \{1, 2, \ldots, 19\}$$

Coup

Un coup c, associé à un jeu dont les positions sont représentables comme ci-dessus, est représenté par :

- un ensemble $c^{\text{pre}} \subset I$, et pour tout $i \in c^{\text{pre}}$, une valeur $e_i^{\text{pre}} \in E_i$,
- un ensemble $c^{\text{mod}} \subset c^{\text{pre}}$, et pour tout $i \in c^{\text{mod}}$, une valeur $e_i^{\text{mod}} \in E_i$ telle que $e_i^{\text{mod}} \neq e_i^{\text{pre}}$.

Les valeurs e_i^{pre} pour $i \in c^{\text{pre}}$ sont appelées *préconditions*. Les valeurs e_i^{mod} pour $i \in c^{\text{mod}}$ sont appelées *modifications*. Le coup c est légal dans une position $(e_i)_{i \in I}$ si, pour tout $i \in c^{\text{pre}}$, $e_i = e_i^{\text{pre}}$. La position d'arrivée est la famille $(e'_i)_{i \in I}$, avec $e'_i = e_i^{\text{mod}}$ si $i \in c^{\text{mod}}$, et $e'_i = e_i$ sinon.

Par exemple, pour un coup au Go, c, qui capture un certain nombre de pierres, c_{mod} contient l'intersection jouée et les intersections des pierres capturées, et c_{pre} contient en plus toutes les intersections adjacentes aux chaînes capturées. De plus, à cause de la règle interdisant le suicide, c_{pre} doit aussi contenir au moins une liberté (qui peut être choisie arbitrairement) de la case jouée.

Jeu

Pour nos besoins, nous nous contentons de faire correspondre à un jeu un graphe orienté dont les sommets sont des positions et les arêtes correspondent à des coups, comme définis précédemment.

Nous ne formalisons pas d'autres aspects des jeux tels que les multiples joueurs ou les valeurs des positions. Ce sont des aspects importants (ils interviennent dans notre travail sur les connexions transitives) qu'on pourrait formaliser (voir par exemple [65]) mais nous n'en n'aurions pas l'utilité. En outre, nous n'imposons pas de contrainte de connexité.

Sous-jeu

Nous avançons une définition possible de sous-jeu ; malgré cela, ce terme reste un peu flou dans notre pratique.

Un sous-jeu est défini par un sous-ensemble de coups du jeu original et par un sous-ensemble de positions stable par le sous-ensemble de coups.

Une définition incorrecte serait de définir un sous-jeu à partir d'un sous-graphe orienté quelconque. On ne pourrait pas alors forcément donner au sous-jeu une structure telle que nous l'avons définie. En effet, si un coup c est présent dans le sous-graphe à partir d'une certaine position, il devrait l'être dans n'importe quelle position du sous-graphe qui vérifie les préconditions c^{pre}.

Il y cependant des cas où la notion intuitive de sous-jeu ne correspond pas à la définition que nous avons donnée. Il est, par exemple habituel de parler de « sous-jeu » pour un problème tactique locolisé dans un jeu à deux joueurs, comme par exemple un problème de connexion au Go. Ce sous-jeu n'est pas défini par un sous-ensemble de coups, car nous ne savons pas *a priori* jusqu'où va mener la recherche. La recherche définit bien un sous-graphe, mais il n'y a pas de raison pour que celui-ci corresponde à un sous-jeu tel que nous l'avons défini. Le sous-jeu est alors plutôt défini par un but à atteindre. C'est une autre signification de « sous-jeu », que nous utiliserons, mais que nous ne savons pas formaliser.

Trace

La *trace* d'un arbre de recherche est définie, de façon informelle, comme l'ensemble des propriétés du jeu dont dépend le résultat de la recherche. Cette notion aide à définir l'indépendance entre des sous-jeux.

Dans un jeu à deux joueurs, la trace d'une séquence S est égale à l'union des préconditions des coups de la séquence :

$$\text{trace}(S) = \bigcup_{\text{coup } c \text{ de } s} c^{\text{pre}}$$

Dans le cas d'un arbre de recherche à deux joueurs, cette notion est utilisée dans le formalisme de [20]. On peut le formaliser avec la définition récursive suivante. Soit un jeu à deux joueurs, appelés joueur *min* et joueur *max*, dont

le résultat est binaire, et soit une recherche de ce jeu qui aboutit à un résultat gagné (c-à-d gagné pour le joueur max). On définit résursivement la trace d'un nœud de l'arbre :

– À un nœud p où le joueur max joue, et où un coup gagnant est le coup c, qui mène au nœud p_c :

$$\text{trace}(p) = c^{\text{pre}} \bigcup \text{trace}(p_c)$$

– À un nœud min, tous les coups du du joueur min sont perdants, et on a :

$$\text{trace}(p) = \bigcup_{\text{coup min } c} \text{trace}(p_c)$$

À un nœud max, le choix du coup gagnant est libre. Des choix différents peuvent mener à des traces plus ou moins grandes.

Sous-jeux indépendants

L'indépendance entre sous-jeux est encore une notion un peu floue ; nous lui donnons des sens différents selon que le jeu est à un ou deux joueurs.

Jeux à un joueur Soient deux sous-jeux $\mathcal{J}_1$ et $\mathcal{J}_2$ d'un jeu à un joueur $\mathcal{J}$. Nous disons qu'ils sont indépendants si :

$$\left(\bigcup_{c_1} c_1^{\text{pre}}\right) \bigcap \left(\bigcup_{c_2} c_2^{\text{mod}}\right) = \left(\bigcup_{c_1} c_1^{\text{mod}}\right) \bigcap \left(\bigcup_{c_2} c_2^{\text{pre}}\right) = \emptyset$$

Cette formule fait intervenir les traces des sous-jeux, et l'analogue avec les modifications remplaçant les préconditions. Autrement dit, deux sous-jeux $\mathcal{J}_1$ et $\mathcal{J}_2$ sont indépendants si la faisabilité d'un coup dans un sous-jeu ne dépend pas de l'avancement dans l'autre sous-jeu.

La définition implique aussi que tous les coups de $\mathcal{J}_1$ « commutent » avec tous les coups de $\mathcal{J}_2$; c'est-à-dire qu'une position atteinte par des coups de $\mathcal{J}_1$ et $\mathcal{J}_2$ ne dépend que de la l'avancement dans $\mathcal{J}_1$ et $\mathcal{J}_2$ et pas de l'ordre d'exécution. En effet, on a toujours, par définition d'un coup, $c^{\text{mod}} \subset c^{\text{pre}}$ quel que soit le coup c ; la définition de l'indépendance implique donc que les modifications se font sur des parties disjointes :

$$\left(\bigcup_{c_1} c_1^{\text{mod}}\right) \bigcap \left(\bigcup_{c_2} c_2^{\text{mod}}\right) = \emptyset$$

Une conséquence est que le graphe du jeu contient le produit cartésien des graphes des deux sous-jeux. Si les sous-jeux contiennent respectivement N_1 et N_2 positions, le produit cartésien contient donc $N_1 \times N_2$ positions. Par exemple, si les jeux $\mathcal{J}_1$ et $\mathcal{J}_2$ sont des séquences sans branchement, on peut représenter $\mathcal{J}_1$, $\mathcal{J}_2$ et $\mathcal{J}$ comme dans la figure 2.1.

L'intérêt d'une décomposition en sous-jeu est, idéalement, de se contenter de rechercher les $N_1 + N_2$ positions des deux sous-jeux, plutôt que les les $N_1 \times N_2$ positions du produit.

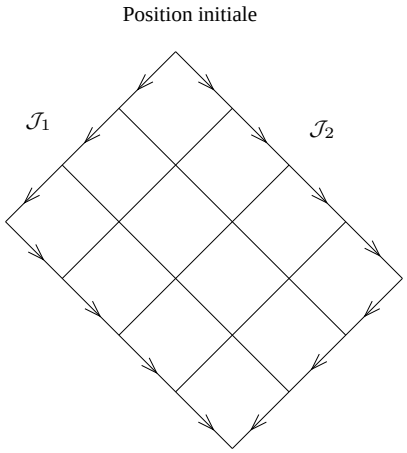


FIG. 2.1 – Deux sous-jeux sans branchements, indépendants.

Jeux à deux joueurs Dans le cadre de jeux à deux joueurs, la définition précédente n'est pas appropriée. Il ne sert pas à grand chose de savoir quels sont les positions atteignables par un joueur, alors que l'autre joueur peut jouer en fonction des coups du premier joueur. Cependant, une décomposition en sous-jeux peut être utile avec une approche différente. Dans notre travail sur la transitivité des connexions (voir pour plus de détails), nous avons été amenés à définir l'indépendance, de façon imparfaite, à partir de l'intersection des traces :

$$\left(\bigcup_{c_1} c_1^{\text{pre}}\right) \bigcap \left(\bigcup_{c_2} c_2^{\text{pre}}\right) = \emptyset$$

2.1.2 Transpositions

Une *transposition* dans une recherche arborescente est la visite à multiples reprises d'une même position par des chemins différents. Les transpositions sont en général nuisibles aux performances, quoique l'impact sur l'efficacité de la recherche est très variable selon les domaines.

Les transpositions peuvent être évitées en marquant les positions parcourues avec des structures de données adaptées (qu'on appelle alors *tables de transposition*. De nombreuses structures, comme les arbres binaires de recherche, permettent des opérations d'insertion et de recherche en $\mathcal{O}(\log(n))$. En pratique, une structure de données très utilisée est une table de hachage, avec des fonctions de hachage de Zobrist [16]. Cette méthode est très efficace en terme de temps et de mémoire.

Les fonctions de hachage de Zobrist sont bien adaptées aux jeux que nous considérons. Avec la définition précédente d'une position d'un jeu 2.1.1, une

fonction de hachage de Zobrist sur n bits serait :

$$f((e_i)_{i\in I}) = \bigoplus_{i\in I} r_{i,e_i},$$

où $\oplus$ représente l'opération ou exclusif bit à bit sur n bits, et où les $r_{i,j}$ sont des nombres aléatoires de n bits précalculés. Le calcul peut être fait incrémentalement et très rapidement quand un coup est joué dans une position, puisque seuls certains des e_i sont modifiés.

En choisissant un nombre de bits assez grand pour la fonction de hachage, la valeur de hachage d'une position peut être considérée en pratique comme un identifiant unique. Pour cela, 64 bits est souvent bien adapté. Ainsi, les collisions ont une probabilité d'apparition négligeable jusqu'à un nombre de positions de l'ordre de $\sqrt{2^{64}} = 2^{32}$, ce qui n'est de toute façon pas atteint à cause de la limitation de la mémoire. Une position prend donc 8 octets en mémoire dans la table de hachage, plus la taille des données qui lui sont attachées (selon les cas, la profondeur de recherche, la valeur de la position, ...). Si la table de hachage est de taille 2^k, l'index d'une position dans la table est fourni par les k premiers bits de la fonction de hachage.

Si deux positions insérées ont le même indice dans la table, plusieurs stratégies existent. En utilisant des listes chaînées pour chaque case de la table de hachage, la table peut grandir autant que souhaité. Ceci est attractif si le but est de compter précisément le nombre de positions dans un jeu, mais est sinon peu pratique puisque les ressources mémoire sont difficiles à maîtriser. Il est plus courant de ne garder qu'une des deux positions ; diverses heuristiques pour choisir la position à garder ont été étudiées. Des variantes existent qui font intervenir deux positions pour chaque case de la table de hachage.

2.1.3 Décomposition en sous-jeux et transpositions

Nous établissons ici un lien entre décomposition en sous-jeux et transpositions : une décomposition en sous-jeux indépendants est une des raisons d'apparition de transpositions, et cela même si il n'y en a pas dans chacun des sous-jeux. En effet, toutes les positions du produit $\mathcal{J}_1 \times \mathcal{J}_2$ qui ne sont ni dans $\mathcal{J}_1$ ni dans $\mathcal{J}_2$ peuvent être atteintes de plusieurs façons.

Nous reprenons l'exemple de la figure 2.1, où $\mathcal{J}_1$ et $\mathcal{J}_2$ sont des séquences sans branchement. Notons que, dans l'exemple précédent de deux sous-jeux indépendants, pour effectuer une recherche des $N_1 \times N_2$ positions du jeu $\mathcal{J}$, il est très utile d'utiliser une table de transpositions. Sans cela, la recherche pourrait nécessiter le parcours d'un nombre de positions encore supérieur.

Il y a deux problèmes liés à des décompositions en sous-jeux indépendants : éviter de parcourir toutes les positions, ou bien détecter les transpositions dues à cette décomposition. Le premier permet des gains potentiels plus grands, mais il est beaucoup plus compliqué que le second, qui admet une solution algorithmique simple. Cet algorithme que nous allons présenter est en fait basé uniquement sur l'indépendance de coups.

2.1.4 Coups commutatifs

Nous utilisons indifféremment les termes de coups *indépendants* ou *commutatifs* ; il s'agit d'un cas particulier de celui de sous-jeux indépendants, dans le cas ou les deux sous-jeux ne sont constitués chacun que d'un coup. Soient p une position et c_1, c_2 deux coups ; les propositions suivantes sont équivalentes et définissent des coups commutatifs :

1. Les coups c_1 et c_2 sont légaux en p et $c_1^{\text{mod}} \cap c_2^{\text{pre}} = c_1^{\text{pre}} \cap c_2^{\text{mod}} = \emptyset$.
2. Les deux séquences ($c1$, $c2$) et ($c2$, $c1$) sont légales dans la position p.

En effet, avec la définition de coup que nous avons donnée, il suffit que les deux séquences ($c1$, $c2$) et ($c2$, $c1$) soient indépendantes pour qu'elles mènent à la même position (ceci repose toujours sur la propriété que $c^{\text{mod}} \subset c^{\text{pre}}$)).

2.2 Transpositions incrémentales

Nous décrivons un algorithme que nous appelons algorithmes de transpositions incrémentales. L'utilité de cet algorithme est de pouvoir détecter certains types de transpositions que nous appelons par la suite transpositions incrémentales, avec une utilisation mémoire très limitée. Une description de cet algorithme peut être trouvée dans l'article [37].

2.2.1 L'algorithme de transpositions incrémentales

La fonction, appelée `dfs_ti` est montrée ci-dessous, prend en argument un ensemble de coups T dont on sait qu'ils mènent à des transpositions incrémentales, c'est-à-dire à des positions qui ont déjà été complètement cherchées. La fonction est appelée initialement avec $T = \emptyset$.

```
procédure dfs_ti(T) {
  Pour chaque coup légal c, c ∉ T,
     T' = { t ∈ T, t et c sont commutatifs} ;
     jouer_coup(c) ;
     dfs_ti(T') ;
     déjouer_coup(c) ;
     T = T ∪ {c} ;
}
```

Pour chaque coup légal c, l'ensemble T' est construit de telle façon qu'il est restreint aux coups qui seront légaux après c. Pour chaque appel récursif, l'ensemble T est donc toujours un sous-ensemble de l'ensemble des coups légaux. En conséquence, cet algorithme nécessite très peu de mémoire.

Nous appelons *transposition incrémentale* une transposition qui peut être détectée par l'algorithme des transpositions incrémentales. De façon équivalente, une transposition est incrémentale si on peut passer d'une séquence à l'autre par une suite d'interversions de coups commutatifs, adjacents dans la séquence.

2.2.2 Preuve de la complétude

Nous montrons que l'algorithme de transpositions incrémentales est complet. Pour cela, il faut montrer que, en supposant que l'ensemble T passé en argument de la fonction `dfs_ti` ne contient que des coups qui mènent à des transpositions, la même propriété vaut pour les appels récursifs sur les positions filles.

Soit c un coup dans la position courante p_1 et $t \in T$, tels que c et t sont commutatifs avec la définition de 2.1.4. Nous avons donc le diagramme commutatif de la figure 2.2. Dans la position p_1, nous savons que le coup t mène à une position, p_2, qui a déjà été complètement cherchée. La position p_4, qui peut être atteinte à partir de la position p_2 avec le coup c, a donc également été complètement cherchée, elle aussi. Puisque la position p_4 peut aussi être atteinte à partir de la position p_3 avec le coup t, on en déduit que le coup t, joué dans la position p_3, mène à une transposition. C'est ce qu'il fallait montrer.

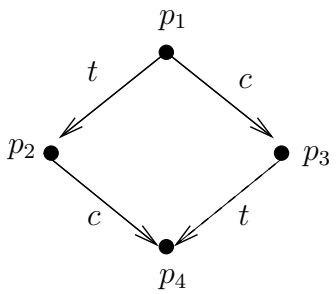


FIG. 2.2 – Transposition de base

2.2.3 Domaines d'applications

L'algorithme de transpositions incrémentales est applicable dès que le jeu contient des coups commutatifs. C'est le cas de nombreux jeux. Cependant, cet algorithme ne détecte que les « transpositions incrémentales ». Il faut donc comparer ce qu'il apporte par rapport à une table de transposition. Un autre problème est la compatibilité de cet algorithme avec d'autres algorithmes de recherche qu'une simple recherche en profondeur d'abord.

Comparaison avec une table de transposition

La première question qui se pose ici est d'estimer la proportion de transpositions qui sont dues à des permutations de coups commutatifs. Ceci dépend beaucoup du domaine ; nous analysons quelques exemples.

Au jeu du taquin, il n'y a absolument aucune paire de coups commutatifs dans aucune position. En effet, si on considère deux coups dans une certaine position, ces coups nécessitent que le trou soit dans sur une certaine case ; aucun des deux coups n'est donc légal après que l'autre ait été joué. La situation

serait différente si, au lieu d'un trou, il y en avait plusieurs. Il existe de nombreux de puzzles de ce genre, tels que *L'âne rouge*, ou *Rush hour*. Dans cette classe de jeux, il est possible que des coups utilisant des trous différents soient commutatifs.

Dans le Rubik's cube, il y a très peu de paires de coups commutatifs : seulement les paires de coups consistant en des rotations sur des faces opposées. L'algorithme des transpositions incrémentales serait donc inefficace.

Les deux domaines que nous allons étudions dans la suite, *Gaps* et le morpion solitaire, contiennent un grand nombre de coups commutatifs. Le morpion solitaire est un cas extrême : nous verrons que toutes les transpositions sont dues à des permutations de coups commutatifs.

Deuxièmement, l'algorithme de transpositions incrémentales n'est intéressant que pour de grandes recherches, quand une table de transposition seule ne suffit pas. De plus, dans certains domaines, il arrive qu'une table de transposition reste efficace même pour des recherches de tailles nettement supérieures. Par exemple, aux échecs, une expérience de D. Breuker montre que, pour une recherche qui prendrait 100×10^6 nœuds de façon optimale, le surcoût n'est que d'un facteur deux pour une table de transposition de 8000 entrées. Ceci dépend des domaines et des types de recherche, et nous verrons que le surcoût peut être beaucoup plus important.

Compatibilité avec d'autres algorithmes que la recherche en profondeur d'abord

Nous avons présenté l'algorithme des transpositions incrémentales comme un ajout à un algorithme de recherche en profondeur d'abord. Est-il possible de l'adapter à d'autres algorithmes de recherche ? Par exemple, il serait souhaitable de pouvoir l'appliquer à des jeux à deux joueurs dans un algorithme de minimax. Il y a cependant une raison simple pour laquelle cette adaptation est difficile, voire impossible : l'algorithme de transpositions incrémentales est capable de détecter que des coups mènent à des transpositions, mais pas de retourner la valeur de la position après ce coup ! Cette valeur est nécessaire pour l'algorithme du minimax. Avec une table de transposition classique, cette valeur (ou, plus précisément, des bornes sur cette valeur) aurait été stockée dans la table.

2.3 Méthodes de Monte-Carlo

L'appellation *méthodes de Monte-Carlo* est assez large ; elle désigne tout algorithme qui utilise des nombres aléatoires ou pseudo-aléatoires pour calculer un résultat de façon approché. Ces méthodes s'opposent donc aux méthodes *déterministes*. Les méthodes de Monte-Carlo se présentent souvent, dans le domaine des jeux, comme une alternative à des méthodes basées sur des connaissances.

Des méthodes de Monte-Carlo sont souvent utilisées dans la programmation des jeux, mais surtout dans des jeux non déterministes ou à information in-

complète. Leur utilisation se justifie cependant aussi dans des jeux déterministes à information complète. C'est dans ce cadre que nous travaillons.

Typiquement, les méthodes de Monte-Carlo sont utilisées pour l'évaluation positionnelle. Il s'agit d'évaluer une position en calculant la moyenne sur plusieurs séquences de jeu aléatoires. Ceci peut alors être combiné avec divers algorithmes de recherche. Typiquement, une évaluation positionnelle par cette méthode est lente mais permet de saisir des aspects du jeu qu'il est difficile de coder statiquement.

D'autres méthodes stochastiques, même si elles ne sont pas à proprement parler des méthodes de Monte-Carlo, sont assez proches. Plutôt que de calculer la moyenne des valeurs des échantillons, d'autres quantités peuvent être intéressantes : le maximum, la médiane, les déciles ou les centiles. Ceci s'applique par exemple à l'algorithme appelé *échantillonnage itératif* dans les jeux à un joueur, qui repose sur le maximum des échantillons.

2.3.1 Cadre mathématique

Dans le domaine des jeux, la théorie mathématique nécessaire pour l'application de méthodes de Monte-Carlo est limitée. Nous utiliserons essentiellement les notions d'espérance, d'écart-type, et le théorème central limite. Particulièrement importante est la décroissance en $1/\sqrt{n}$ de l'écart-type, quand une même variable aléatoire est échantillonnée n fois de façon indépendante.

Les applications de méthodes de Monte-Carlo dans le calcul scientifique (simulations physiques par exemple) se ramènent en général à des calculs d'intégrales de fonctions, sur des domaines D munis de distributions de probabilité :

$$\int_D g(x)dx$$

Une théorie mathématique existe pour améliorer la précision de ces calculs [46]. Les difficultés proviennent de la dimension élevée du domaine d'intégration D, et du fait que la plus grande partie de l'intégrale est souvent concentrée sur une très faible partie du domaine. Nous n'utilisons pas de théorie aussi poussée pour nos calculs, et nous pensons que ce ne serait pas nécessaire, car les difficultés sont différentes. Dans nos expériences sur le Go, la convergence est en général raisonnablement rapide, et le problème tient plutôt à la formulation du problème et à l'interprétation des résultats.

2.3.2 Évaluation positionnelle

L'idée générale consiste à évaluer une position en calculant la moyenne sur plusieurs séquences de jeu. Ces séquences sont parfois jouées jusqu'à la conclusion de la partie qui sert d'échantillon ; ainsi, l'évaluation de cet échantillon est triviale.

Cette idée est naturelle si le jeu est non déterministe. Elle est intéressante aussi si le jeu est à information incomplète, pour calculer une moyenne sur les divers paramètres inconnus, dont on peut estimer la distribution. Même si elle

moins naturelle dans le cadre de jeu déterministes à information complète, elle reste possible. Le « hasard » porte alors sur les coups des joueurs. Notre travail comporte des exemples sur un jeu à un joueur (le go) et un jeu à un joueur (le morpion solitaire).

Nous commençons, ici, par donner des exemples de programmes utilisant des simulations de Monte-Carlo.

Backgammon

Au Backgammon, la force des meilleurs programmes est généralement basée sur une évaluation positionnelle par réseaux de neurones. Des simulations de Monte-Carlo peuvent être utilisées pour simuler le hasard des dés, et ainsi améliorer la précision. Les coups joués sont donnés par le réseau de neurones. L'utilisation est cependant difficile en partie réelle pour des contraintes de temps [62].

Scrabble

Le meilleur programme de Scrabble actuel est MAVEN. La programmation de ce jeu fait appel à diverses techniques. Par dessus, l'utilisation de simulations de Monte-Carlo est aussi possible pour améliorer la précision de l'évaluation, de façon similaire au Backgammon [58]. Une différence, cependant, est que les simulations portent à la fois sur le hasard des tirages à venir, et sur l'incertitude du jeu courant de l'adversaire.

Bridge

Un des meilleurs programmes de Bridge est actuellement GIB, écrit par M. Ginsberg [34, 33]. La force de ce programme provient de l'utilisation qu'il fait d'une méthode de Monte-Carlo. Il est devenu plus fort que d'autres programmes qui essayaient plutôt de copier le raisonnement humain et contenaient beaucoup de connaissances expertes.

Ce programme est capable de faire des recherches complètes, dès le début de la partie, sur des donnes ouvertes (c'est à dire que les jeux des quatre joueurs sont connus de tous les autres). Ces recherches sont, de plus, réalisées relativement rapidement : environ 200.000 noeuds en moyenne. Cette performance est due à l'utilisation de la méthode de recherche *Partition Search*. En situation réelle, un joueur doit choisir quelle carte jouer sans connaître les jeux des trois autres joueurs. Il dispose cependant de quelques indications à cause des enchères, et des cartes jouées dans les plis précédents. Le programme construit donc un certain nombre de donnes cohérentes avec ces indications, résout ces donnes comme si il s'agissait de donnes ouvertes, et joue le coup qui maximise l'espérance.

Cette méthode présente quelques défauts inhérents. Le programme n'est pas capable de *bluffer*, c'est à dire de jouer de façon à induire les adversaires en erreur sur son propre jeu. Il n'est pas capable non plus de jouer de façon à obtenir des indications sur les jeux adverses. En effet, ces stratégies n'ont pas

lieu d'être dans les recherches à donnes ouvertes, et l'utilisation du Monte-Carlo ne peut pas corriger cela.

La méthode utilisée dans GIB peut être généralisée à d'autres domaines à information imparfaite. Ainsi, une méthode similaire a été appliquée au Tarok (un jeu semblable au tarot, joué en Europe centrale) [47].

Poker

Au poker, Le programme *Poki* utilise des simulations de Monte-Carlo de façon assez similaire à GIB au Bridge [7].

Go

Le jeu de Go est notre domaine de travail principal, dans le cadre de l'évaluation positionnelle par Monte-Carlo. Ces techniques ont été introduites dans le domaine du Go par B. Bruegmann [17]. Le programme s'appelait Gobble. Plus précisemment, la méthode appliquée était celle du recuit simulé, qui est une méthode générale d'optimisation [43].

Les travaux antérieurs de B. Bruegmann, ainsi que les nôtres et ceux d'autres programmeurs, seront présentés dans le chapitre correspondant.

2.3.3 Autres applications

L'algorithme d'*échantillonage itératif* est un algorithme de recherche stochastique particulièrement simple, dans un jeu à un joueur dont le résultat est binaire (gagné ou perdu) [45]. Dans ce cadre, un échantillon consiste à joueur une séquence aléatoire, à partir de la position initiale, jusqu'à être bloqué. Si la partie est gagnée, l'algorithme se finit, sinon un autre échantillon est réalisé en recommençant à la position initiale. La différence principale avec un algorithme de Monte-Carlo classique est donc qu'on calcule le maximum sur les échantillons, plutôt que la moyenne.

Il existe des méthodes plus évoluées de recherches stochastiques dans des jeux à un joueur. La méthode générale consistant, de façon informelle, à recommencer la recherche à zero avec d'autres réglages aléatoires, lorque les précédent réglages ont échoué, est souvent utilisée. Elle l'est par exemple dans le programme de Sokoban *Rolling stone*, sous le nom *Rapid random restart* [42]. Ce genre de méthodes n'est cependant pas à proprement parler des méthodes de Monte-Carlo.

Des variations sont possibles si le résultat du jeu n'est pas binaire mais peut prendre toute une gamme de valeurs. Il est possible de considérer le maximum sur les échantillons, ou bien la médiane, les déciles ou les centiles. . .Par exemple, l'utilisation du décile supérieur peut se justifier par le fait qu'il est à la fois statistiquement plus stable que le maximum, et plus significatif que la moyenne dans un jeu où on cherche à maximiser. L'évaluation positionnelle ainsi obtenue peut alors servir pour la sélection des branches les plus intéressantes dans une recherche arobrescente, bâtie au dessus du Monte-Carlo.

Les possibilités sont nombreuses. Nous avons utilisé plusieurs fois des algorithmes de cette sorte pour Montana et pour le Morpion solitaire, mais nous ne le détaillerons pas (les résultats obtenus sur Montana avec de tels algorithmes ne sont pas meilleurs qu'avec un simple échantillonage itératif, et les résultats sur le Morpion solitaire sont inférieurs à des résultats obtenus par d'autres personnes avec des méthodes similaires).

Chapitre 3

Montana

Nous étudions une *patience*, que nous appellons par un de ses noms les plus courants : *Montana*. Le terme patience désigne un jeu de cartes en solitaire ; on emploie aussi le mot *réussite*.

Le livre de P. Crépeau [26] recense 179 patiences. Le logiciel libre *pysol* en propose 213. Il s'agit donc d'un domaine vaste. La patience que nous avons choisie se caractérise, d'abord par le fait que c'est une patience *ouverte* (comme 52 des patiences proposées dans pysol) ; c'est à dire que toutes les cartes sont visibles à tout moment. Il n'y a donc pas de hasard ou d'incertitude dans le jeu, sauf pour le tirage de la position initiale. Ensuite, la patience choisie possède des règles particulièrement simples.

Quelques autres noms de cette patience sont : *Gaps*, *Blue moon*, *Spaces*, *Rangoon*. Souvent, ces noms désignent des variantes légèrement différentes, mais les différentes sources ne s'accordent pas sur la nature et les noms précis des variantes.

La patience Montana a également été étudiée sous le nom de *Superpuzz* par H. Berliner [6], puis par T. Shintani [59, 60]. Nous l'ignorions quand nous avons commencé nos recherches sur cette patience ; en effet, Superpuzz n'est pas un nom courant pour le jeu. Compte tenu du grand nombre patiences existantes, il est remarquable que la même ait été étudiée deux fois indépendamment.

Hormis la patience Montana, les patiences du type de *freecell* ont aussi été l'objet de recherches académiques. Cette patience doit sa popularité au fait qu'elle a été incluse dans des distribution de Microsoft Windows. Elle a été utilisée comme domaine de test en planification [39]. Il s'agit également d'une patience ouverte.

L'intérêt de ce domaine d'étude est de présenter des caractéristiques rendant une décomposition en sous-jeux intéressante, tout en restant relativement simple. Nous présentons les règles du jeu, les travaux antérieurs, et deux approches différentes de recherche dans ce domaine, l'une basée sur une décomposition en sous-jeux, l'autre sur l'algorithme des transpositions incrémentales.

3.1 Règles du jeu

Nous expliquons les règles de ce que nous appelons la variante de base, puis celles de quelques autres variantes. Nous donnons ensuite quelques propriétés de la patience.

3.1.1 Variante de base

La patience Montana se joue avec 52 cartes. Les cartes sont placées en 4 rangées de 13 cartes chacune. Les 4 As sont retirés du jeu, ce qui crée 4 trous dans la position ; les As sont ensuite placés sur une nouvelle colonne à gauche dans un certain ordre, qui est le même pour toutes les parties (par exemple, pique sur la 1ère ligne, cœur sur la 2e, carreau sur la 3e, trèfle sur la 4e). Le but est de créer des séquences ascendantes, de même famille, sur chaque ligne, du Deux jusqu'au Roi.

Un coup consiste à déplacer une carte dans un trou ; le trou se retrouve alors là où était la carte. Un trou peut être rempli uniquement par le successeur, dans la même famille, de la carte à gauche du trou. Si le trou est sur la colonne de gauche, si il y a un trou à sa gauche ou si la carte à sa gauche est un roi, aucune carte ne peut être placée dans ce trou. La figure 3.1 montre une position initiale avec seulement 4 cartes par famille (pour simplifier), avant et après le déplacement des As. Les coups possibles sont aussi indiqués.

FIG. 3.1 – Une position initiale avec 4x4 cartes, avant et après le déplacement des As.

La figure 3.2 représente le graphe acyclique orienté obtenu par une recherche à partir de la position de la figure 3.1. On voit que cette position est gagnable. L'observation de ce graphe montre qu'il y a essentiellement deux façons de perdre, vers le début de la partie. La première façon est de jouer la séquence 4 de Pique - 3 de Pique - 3 de Carreau avant de déplacer le 4 de Carreau. La deuxième est de jouer la séquence 4 de Carreau - 2 de Pique avant de jouer la séquence 4 de Pique - 3 de Pique.

3.1.2 Autres variantes

La variante de base présentée ci-dessus n'est probablement pas la plus courante. Habituellement, les As ne sont pas placés dans une nouvelle colonne à gauche mais sont définitivement retirés du jeu. Pour compenser, il est permis de déplacer n'importe quel Deux dans un trou si celui-ci se trouve sur la première

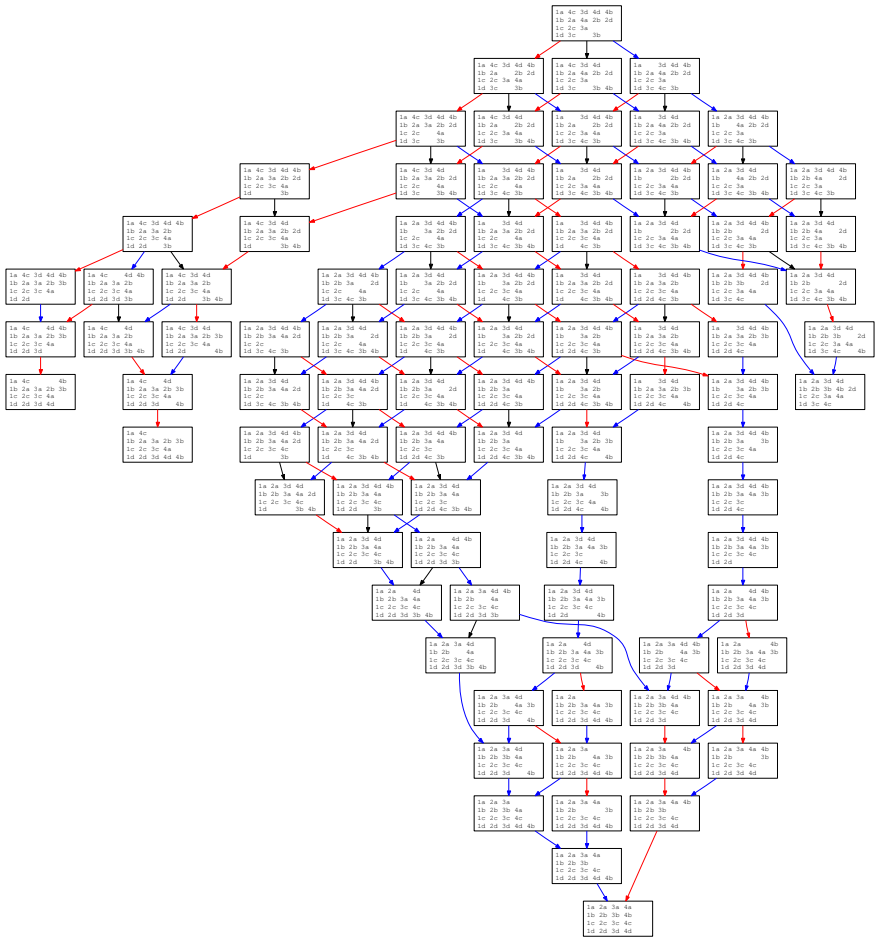


FIG. 3.2 – Arbre de recherche complet pour une position initiale de 4x5 cartes

colonne. Il y a donc plus de coups possibles que dans la variante de base, dans laquelle un seul Deux pouvait être placé sur une trou dans la première colonne. Cette différence a une grande influence a la fois sur la taille moyenne de l'espace de recherche et sur la probabilité qu'une partie soit gagnable. Nous appelons cette variante la variante commune.

Notre travail se concentrera sur la variante de base plutôt que sur la variante commune. Les règles de la variante commune sont moins « propres » que celles de la variante de base. Certaines des techniques que nous montrerons ne seront pas applicables à la variante commune sans modifications. Il s'agit cependant de la variante qui a été étudiée sous le nom de *Superpuzz*.

Certaines autres variantes donnent droit à redistribuer les cartes mal placées en cas de blocage. Jusqu'à trois redistributions sont permises. Une redistribution

consiste à retirer du jeu toutes les cartes mal placées (c-à-d les cartes qui ne font pas partie d'une séquence ascendante partant de la première colonne), les mélanger (avec les As) et les redistribuer, puis de nouveau enlever les As. Nous n'avons pas étudié de variantes de ce type.

Finalement, il est évidemment possible de changer le nombre de cartes par familles ou le nombre de familles.

3.1.3 Propriétés

La variante de base a une propriété remarquable : la profondeur de l'espace de recherche est bornée. En particulier, il n'y a pas de cycle.

Pour montrer cette propriété, nous montrons par récurrence sur v que, dans toute position, une carte de valeur v ne peut être déplacée au plus que $v-1$ fois dans la suite de la partie. C'est vrai pour les As, qui sont sur la première colonne et ne peuvent pas être déplacés. En supposant que c'est vrai pour toute carte de valeur $v-1$, alors c'est vrai pour une carte de valeur v, puisqu'elle doit forcément être déplacée à droite de la carte précédente, de valeur $v-1$, et que celle-ci, en comptant sa position originale, pourra être en tout dans $1+(v-2) = v-1$ emplacements. Finalement, le nombre total de mouvements pour le jeu avec 52 cartes est borné par $4 \times (1+2+3+\ldots+12) = 312$.

3.2 Techniques de base et travaux antérieurs

Dans cette partie, nous appliquons des méthodes de recherche classiques à la patience Montana. Cette partie s'applique aussi bien à la variante de base qu'à la variante commune. Nous montrons des résultats pour un algorithme de recherche en profondeur d'abord et un algorithme d'approfondissement itératif.

3.2.1 Recherche en profondeur d'abord

D'après des expériences, une recherche en profondeur d'abord complète est réalisable dans la variante de base avec 52 cartes. En supposant que la recherche est interrompue dès qu'une solution est trouvée, la taille moyenne de l'espace de recherche est d'environ 250.000. La taille dépasse rarement 2×10^6. C'est suffisamment petit pour que toutes les positions puissent être stockées dans une table de transposition.

Un test sur 10.000 positions initiales a montré qu'une position aléatoire a environ 24.8% de chances d'être gagnable. La longueur des solutions gagnantes est habituellement entre 90 et 130 coups. Ces calculs ont été effectués sur un athlon 1600+ avec 1Go de RAM ; la recherche précédente prenait environ 0.2s par problème en moyenne.

Par contre, une recherche en profondeur d'abord complète n'est pas réalisable dans la variante de base avec plus de cartes, ou dans la variante commune.

3.2.2 Échantillonage itératif

Quand une recherche en profondeur d'abord complète n'est pas possible, il faut faire une recherche incomplète, guidée soit par des heuristiques, soit simplement par le hasard.

Un des algorithmes les plus simples envisageable est l'algorithme d'échantillonage itératif 2.3.3. Un *échantillon* consiste ici à jouer une partie en choisissant tous les coups au hasard, à partir de la position initiale, jusqu'à gagner ou être bloqué. Cet algorithme est en fait particulièrement efficace. [1]

Nous donnons dans la table 3.1 des résultats de cet algorithme pour la variante de base et la variante commune. Les expériences ont été faites sur 10.000 positions initiales aléatoires pour les expériences avec moins de 1000 échantillons, et sur 1000 positions initiales pour les autres. Autant que possible, l'ensemble des positions initiales est le même pour les différentes expériences. Un échantillon (c-à-d une partie aléatoire à partir de la position initiale) prend environ 4.5μs sur un athlon 1600+. En pratique, le nombre maximum d'échantillons n'est atteint que si une solution n'a pas été trouvée. Pour 10^8 échantillons maximum, les temps moyens des expériences sont de 425s et 164s respectivement pour la variante de base et la variante commune.

nb. max d'échantillons	taux de réussite variante de base	taux de réussite variante commune
1	0.046%	0.041%
10	0.373%	0.396%
10^2	1.37%	2.96%
10^4	7.1%	26.6%
10^5	9.7%	43.1%
10^6	12.8%	53.0%
10^7	14.5%	60.3%
10^8	16.4%	66.9%

TAB. 3.1 – Résultat de l'échantillonnage itératif

L'algorithme d'échantillonage itératif est donc plutôt efficace. Ceci est en contraste avec d'autres jeux plus complexes dans lesquels, même si il existe un grand nombre de solutions, toutes contiennent un certain nombre de coups qui doivent être joués dans un ordre précis. Il serait presque impossible de les trouver au hasard.

Une raison pour l'efficacité de l'algorithme dans le cas de Montana tient au fait que les position initiales sont aléatoires, et non pas choisies en fonction

[1] Pour voir à quel point l'algorithme d'échantillonage itératif est efficace comparé à d'autres, il est intéressant d'évoquer le processus par lequel nous avons pensé à cette méthode. Nous travaillions sur des méthodes de recherche sélectives basées sur des heuristiques pour sélectionner les coups. Pour traiter les fins de partie rapidement, nous avons vu qu'une méthode d'échantillonage itératif était efficace. Graduellement, nous nous sommes aperçus que cette méthode était efficace aussi en milieu de partie, et même en début de partie – si bien que nous avons remplacé notre algorithme compliqué, par de l'échantillonnage itératif, avec des résultats comparables.

de leur difficulté. Les positions initiales gagnables mais pas facilement sont, finalement, relativement rares parmi toutes les positions possibles.

3.2.3 Combinaison d'une recherche à profondeur bornée avec l'échantillonnage itératif

L'algorithme d'échantillonage itératif est efficace parce que, statistiquement, il parcourt une partie assez bien distribuée de l'espace de recherche. Il y a cependant des irrégularités dans cette distribution, et elles sont particulièrement sensibles en début de partie. Par exemple, si on considère toutes les positions atteignables à une certaine profondeur, elles seront explorées par l'algorithme d'échantillonage itératif avec une probabilité proportionnelle au nombres de façons de les atteindre depuis la position initiale. Idéalement, on voudrait que ce soit proportionnel à la taille des sous-arbres issus de ces positions.

On peut au moins améliorer l'algorithme en prenant des probabilités uniformes pour toutes les positions à une certaine profondeur. Ceci mène à l'algorithme suivant, qui est une combinaison d'une recherche à profondeur bornée et d'un échantillonage itératif.

Nous faisons une recherche en largeur d'abord à partir de la position initiale, jusqu'à l'épuisement des ressources mémoire ; à chaque nœud de cette recherche, nous lançons une partie aléatoire.

Nous avons testé cet algorithme sur 100 positions initiales aléatoires pour la variante commune. Nous étions limités par le nombre de positions qui pouvaient être stockées en mémoire ; avec notre implémentation, avec 1Go de mémoire disponible, il était possible de stocker cinq millions de positions. Le programme a pris 144s par problème en moyenne. Il a trouvé des solutions pour 88 des positions initiales et a prouvé 4 positions impossibles. Ceci est donc à la fois plus rapide et plus performant que la méthode d'échantillonage itératif.

3.2.4 Différences entre la variante de base et la variante commune

Dans la variante commune, les règles permettent plus de mouvements. Ceci se traduit par une taille plus grande de l'espace de recherche et de plus grandes chances que les parties soient gagnables. Mais ces deux variantes ont également des propriétés différentes.

Remarquons d'abord que le variante de base est un cas particulier de la variante commune. En effet, dans la variante commune, si les 4 Deux ont été placés sur la première colonne, il ne peuvent plus être déplacés ; les règles sont alors similaire à la variante de base, juste avec une colonne en moins.

Ensuite, dans la variante commune, il existe des cycles dans le graphe de recherche. Par exemple, si un Deux et un trou se trouvent sur la première ligne, bouger ce Deux dans le trou est un coup inversible. T. Shintani a étudié plus en détail ces phénomènes [59, 60]. Ces travaux n'ont été publiés qu'en japonais. Nous présentons rapidement les principaux résultats obtenus.

Il s'agit principalement d'une étude des *composantes fortement connexes* du graphe de recherche. Ces composantes ont la propriété que, pour toute paire de positions u et v, il existe des chemins de u à v et de v à u ; de plus ces composantes sont maximales par rapport à cette propriété. Il est possible de décrire complètement ces composantes fortement connexes. Sans rentrer dans les détails, nous donnons ici un exemple de position avec une composante fortement connexe de taille maximale :

□	□	□	□	...
2♡	3♡	4♡	5♡	...
2♠	3♠	4♠	5♠	...
2♣	3♣	4♣	5♣	...

Les cartes à droite ont été omises. Dans cette position, il est possible de permuter entre elles toutes les cartes sur une même colonne, tout en restant dans la même composante fortement connexe (pour cela, commencer par placer les cinqs, puis les quatres, jusqu'aux deux). Cette composante est donc de taille $(4!)^4 = 331776$. D'autres dispositions des trous donnent des composantes fortement connexes plus compliquées.

Étant capable de trouver rapidement la composante fortement connexe de toute position, il est alors possible de les remplacer par de simples nœuds. On obtient alors un arbre acyclique orienté, de taille inférieure à l'arbre original. Dans l'optique de réaliser une recherche en profondeur d'abord complète, ceci permet de réduire les besoins en mémoire. Dans ses travaux, T. Shintani implémente une table de transposition avec un arbre de Patricia là où nous utilisons une table de hachage. Hormis cela, il fait des recherches en profondeur d'abord de la même façon que nous.

3.2.5 Comparaison avec l'approche de H. Berliner

Avant T. Shintani, le domaine de Montana a été étudié par H. Berliner [6]. Cette publication nous était inconnue lors de nos travaux. Elle consiste en une comparaison d'algorithmes de recherche en profondeur d'abord, en meilleur d'abord et en A*. Des heuristiques sont utilisées pour l'A* et pour la recherche en meilleur d'abord ; nous n'en connaissons pas les détails. Une recherche auxiliaire basée sur des satisfactions de contraintes est utilisée pour tester rapidement des blocages et couper la branche courante. Les détails ne sont pas non plus expliqués dans l'article. Les expériences sont limitées à des tailles de jeu inférieures à 4x8.

Une autre particularité de l'approche est la séparation du jeu en deux phases, la première consistant à placer les as sur la première colonne (ce qui revient donc, pour nos définitions, à se ramener à la variante de base), et la deuxième à placer les autres cartes. Les recherches de blocages ne sont faites qu'en deuxième phase.

L'article conclut globalement sur l'efficacité des heuristiques et de la recherche de blocage, pour éviter une explosion combinatoire ; les conclusions sont donc différentes des notres, puisque nous prétendons qu'un des meilleurs algo-

rithmes est une recherche basée essentiellement sur de l'échantillonnage itératif.

Les conclusions de H. Berliner sont toutefois sujettes à une critique : observer une amélioration avec l'utilisation de coupes heuristiques n'est pas forcément seulement une indication de la qualité des heuristiques, mais aussi plus simplement de la quantité de coupes qui sont pratiquées. Certaines des méthodes utilisées par H. Berliner sont proches de méthodes que nous avons essayées, et les résultats n'ont pas été meilleurs qu'un simple échantillonage itératif.

3.2.6 Discussions sur les possibilités d'amélioration

Nous avons montré des résultats obtenus par quelques algorithmes de base. Comment les améliorer ?

Une idée naturelle est de rajouter des heuristiques pour la sélection ou l'ordonnancement des coups. Divers algorithmes de recherche sont alors possibles, tels que la recherche par faisceau (*beam search*), *limited discrepancy search* [36], ...

Les essais que nous avons menés pour chercher de bonnes heuristiques n'ont pas été concluants. Nous n'avons pas pu faire mieux ainsi que la méthode basée sur l'échantillonnage itératif. Parmi les heuristiques que nous avons essayées, il y a :

- Le nombre de cartes qui sont au même emplacement que dans la position gagnante. Cette heuristique n'est pas bien bonne. D'une part, il est tout à fait possible d'avoir presque toutes les cartes bien placées, mais d'être quand même bloqué. D'autre part, il n'est pas rare d'avoir peu de cartes bien placées, mais de pouvoir finir le jeu rapidement par une suite de mouvement simple ; comme par exemple dans la position de la figure 3.3.

1♢	□	5♠	6♠	7♠	7♠	9♠	10♠	V♠	D♠	R♠	□	□	□
1♠	2♠	3♠	4♠	2♢	3♢	4♢	5♢	6♢	7♢	8♢	9♢	10♢	V♢
1♡	2♡	3♡	4♡	5♡	6♡	7♡	8♡	9♡	10♡	V♡	D♡	R♡	D♢
1♣	2♣	3♣	4♣	5♣	6♣	7♣	8♣	9♣	10♣	V♣	D♣	R♣	R♢

FIG. 3.3 – Une fin de partie de Montana

- Le nombre de séquences croissantes dans la même famille à l'intérieur de la position (y compris les séquences de longueur 1, c-à-d les cartes *isolées*). Ce nombre ne peut que décroître au fur et à mesure de l'avancement de la partie, puisqu'un coup peut supprimer une séquence à la case de départ si la carte était isolée, mais ne peut pas en rajouter sur la case d'arrivée puisque la carte appartient alors à la même séquence que la carte de gauche. Ce nombre est donc, en quelque sorte, un assez bon indicateur de l'avancement de la partie. Par contre, il n'est pas clair qu'il puisse servir à évaluer une position. Il faudrait au moins l'utiliser en combinaison avec d'autres.
- Le nombre de trous utilisables. En première approximation, on peut dire qu'un trou n'est pas utilisable si il est précédé d'un Roi (un trou précédé d'un trou est souvent plus facilement utilisable).

Nous n'avons pas trouvé ces heuristiques très utiles. Il est possible qu'on puisse en trouver de meilleures.

Dans la suite, nous explorons une voie différente, orthogonale à l'approche heuristique. Notre but est de simplifier l'espace de recherche, par exemple pour pouvoir faire une recherche complète plus vite.

3.3 Recherche par blocs

Dans cette partie, nous présentons une nouvelle méthode qui vise à montrer l'indépendance entre certaines parties d'un problème, pour un nombre restreint de coups, et à exploiter ces indépendances tout en gardant la recherche complète.

À partir de maintenant, nous nous concentrons uniquement sur la variante de base. En effet, nous utiliserons la propriété que, pour chaque trou, il y a une seule carte qui peut être placée dedans. La variante commune n'a pas cette propriété. Il serait sans doute possible avec plus de travail d'adapter la méthode pour cette variante.

3.3.1 Motivations

La figure 3.4 montre l'espace de recherche pour une certaine position initiale, dans la variante de base. Le graphe est montré en entier, mais l'image a été fortement réduite en taille. Il s'agit d'une position initiale de taille plutôt petite : 4564 nœuds (comme on l'a vu, la moyenne est vers 250,000). La partie encadrée dans cette figure est agrandie dans la figure 3.5.

Cet arbre a été construit à l'aide du programme *graphviz* à partir d'une définition brute du graphe de la recherche (qui a lui-même été obtenu par une recherche en profondeur d'abord). On ne contrôle pas directement le placement des nœuds, mais le programme est conçu pour les placer de façon à avoir une bonne visibilité du graphe, de sorte que, parfois, certaines régularités du graphe sont visibles sur la représentation qu'en fait graphviz.

Dans la partie du graphe qui a été agrandie, on observe, dans différentes zones, des arrangements en réseau, avec des arêtes dans deux directions différentes. Ces directions correspondent en fait à des coups utilisant différents trous ; ces différentes séquences sont, au moins sur quelques coups, indépendantes. Ces arrangements ont cependant des étendues limitées, et il y a des phénomènes de transition particuliers aux frontières.

Cet agrandissement est assez typique des autres régions de ce graphe, et des autres graphes correspondant à différentes positions initiales. En général, on pourrait aussi trouver des arrangements avec 3 ou 4 séquences indépendantes.

Le but de la recherche par bloc est d'exploiter cette régularité dans l'espace de recherche.

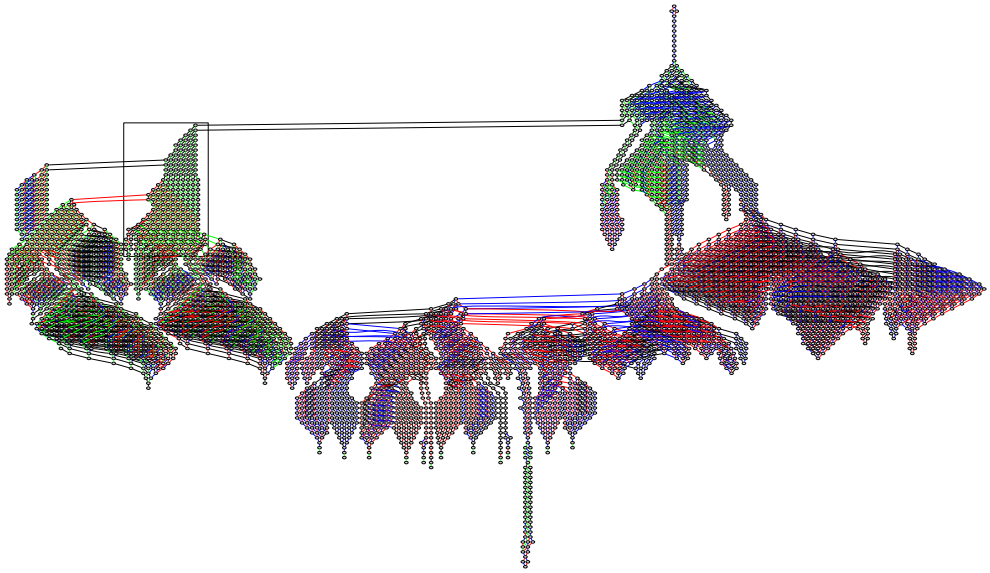


FIG. 3.4 – Graphe de recherche complet pour une certaine position initiale de 4x13 cartes

3.3.2 Travaux antérieurs : *Dependency based search*

Nous avons conçu la recherche par blocs après des essais infructueux pour appliquer une méthode existante, appelée *Dependency based search* (*dbs*), de V. Allis [2]. Nous pensons que cette méthode a des limitations qui n'étaient pas mentionées.

La méthode est exposée dans un cadre plus général que le notre : une position est définie comme un ensemble d'axiomes. Des conditions d'application sont indiquées : *monotonie*, *non-redondance* et *singularité*. Singularité signifie qu'il existe une seule position gagnante ; ceci n'est pas vérifié dans Montana, mais ce n'est pas très gênant, d'après [2]. Non-redondance signifie que les positions gagnantes n'ont pas de fils ; ceci est vérifié dans Montana. Monotonie signifie que, dans la suite de positions d'une séquence de coups, un même axiome ne peut pas être supprimé puis rajouté de nouveau. Ceci n'est pas vérifié pour Montana : il est par exemple possible qu'une carte soit placée plusieurs fois sur le même emplacement. Ceci est gênant, mais on pourrait espérer adapter l'algorithme.

Ce que nous avons trouvé le plus gênant n'est en fait pas une condition mentionné explicitement dans [2], mais est lié à une fonction qui est difficile à écrire efficacement. Un pseudo-code est donné pour *dbs*, mais il dépend d'une fonction appelée *NotInConflict* qui n'est pas donnée et doit être codée au cas par cas. Le rôle de cette fonction est de tester la compatibilité entre deux nœuds, en cherchant si les séquences qui mènent depuis la racine à ces deux nœuds

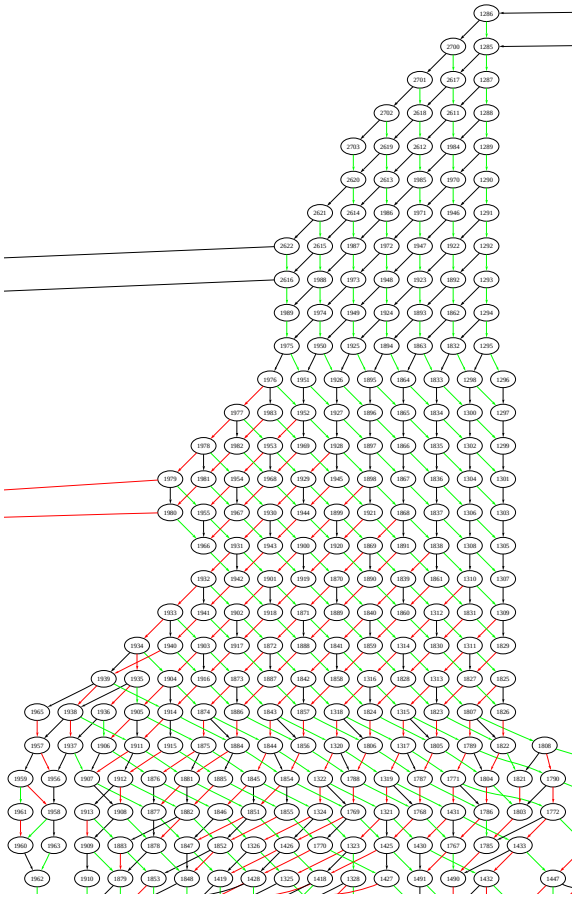


FIG. 3.5 – Graphe de recherche, agrandissement

peuvent être fusionnées en une seule séquence légale, éventuellement en jouant sur l'ordre des coups. Or, si il est facile de voir comment est écrite cette fonction dans chacun des domaines d'application de *dbs*, il ne parait pas possible d'écrire une telle fonction dans le cas de Montana.

Dbs a été appliqué, par exemple, pour la résolution de jeux comme Qubic et Gomoku. Il s'agit de jeux à deux joueurs, mais qui peuvent être ramenés à des jeux à un joueur si on se limite aux séquences de coups forçants, qui sont très courantes dans ces jeux. La compatibilité des séquences dans ces jeux est facile à décider, puisqu'il suffit de tester l'intersection des traces des deux séquences.

Principe de la méthode

Nous appelons les quatre trous A, B, C, D, et nous divisons le jeu en quatre « sous-jeux », que nous appelons aussi A, B, C, D. Les coups permis dans un sous-jeu sont ceux qui utilisent le trou correspondant.

Puisque nous sommes dans la variante de base, dans toute position, il n'y a au plus qu'un coup possible dans chaque sous-jeu ; si on ne joue que des coups dans un sous-jeu, on obtient une séquence de coups menant à une position où ce sous-jeu est bloqué. Il y a deux raisons possibles pour cela : soit la case précédente est un trou, soit c'est un Roi. L'intérêt de cette décomposition est que les séquences associées au différents sous-jeux sont souvent relativement indépendantes les unes des autres.

Nous définissons un *bloc* par :

- sa position initiale,
- pour chaque sous-jeu X, une séquence S_X, éventuellement vide, à partir de la position initiale, avec la contrainte que les quatres séquences sont totalement indépendantes les unes des autres.

Un bloc représente l'ensemble des positions qui peuvent être atteintes à partir de la position initiale du bloc avec des coups des séquences S_X. Ces séquences étant indépendantes, l'ordre des coups entre les différentes séquences n'a pas d'importance. Si l_X est la longueur de la séquence S_X, le bloc représente $l_A \times l_B \times l_C \times l_D$ positions. On peut imaginer un bloc comme étant plongé dans un espace de dimension 4. Nous appelons F_X la « face » du bloc qui consiste en les positions du bloc où la séquence S_X a été complètement jouée.

L'idée principale de l'algorithme est de chercher un bloc à la fois au lieu de chercher une position à la fois. Nous faisons une recherche en profondeur d'abord au niveau des blocs, en construisant de nouveaux blocs à la frontière et en les cherchant récursivement.

Nous voulons construire des blocs les plus gros possibles. Ainsi, avant de lancer les appels récursifs sur les blocs fils, nous essayons d'étendre les blocs le plus possible dans chacun des quatre sous-jeux. La figure 3.6 montre un pseudo-code pour l'algorithme.

```
procédure recherche_bloc(bloc) {
  pour chaque sous-jeu X
    étendre le bloc dans le sous-jeu X, tant
    que les séquences restent indépendantes ;
  pour chaque sous-jeu X
    construire des nouveaux blocs près de la face F_X du bloc,
    tels que chaque coup jouable sur F_X mène à un de ces blocs
    et les chercher récursivement ;
  tester l'existence d'une position gagnante dans le bloc ;
}
```

FIG. 3.6 – Pseudo-code pour la recherche par blocs

Il reste à expliquer comment étendre les blocs et construire de nouveaux blocs aux frontières. On montrera aussi comment adapter ce pseudo-code pour utiliser une table de transposition.

L'interaction de base

Nous étudions en détail le cas d'une interaction simple entre deux séquences. Cette interaction est schématisée sur la figure 3.7. Pour simplifier, toutes les cartes sont de la même famille. Nous supposons que les deux séquences commencent quelques coups avant l'interaction et continuent quelques coups après ; nous n'avons représenté que les coups intéressants. La flèche pointillée dans le diagramme de droite indique l'action de la séquence S_B sur la séquence S_A. Nous supposons que nous sommes juste après le coup a_1 dans S_A. Si b_1 n'a pas encore été joué, le coup a_2 peut être joué dans le sous-jeu A ; si le coup b_2 a déjà été joué, le coup a'_2 peut être joué dans le sous-jeu A ; si le coup b_1 a été joué mais pas le coup b_2, aucun coup ne peut être joué dans le sous-jeu A.

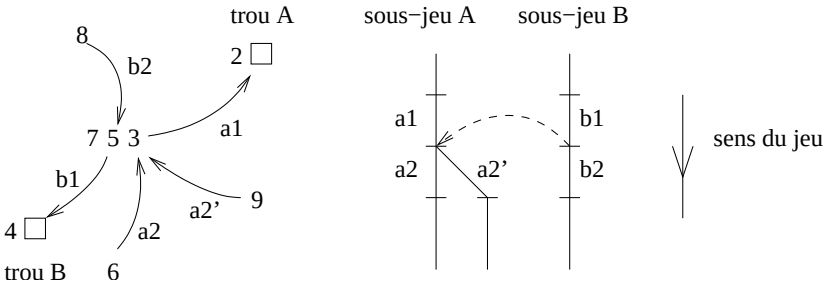


FIG. 3.7 – Une interaction de base

La figure 3.8 montre une représentation dans le plan de l'espace de recherche. Les positions sont aux intersections. On peut comparer avec des séquences sans interactions ; il y aurait alors un grand rectangle avec les séquences A et B sur les côtés. L'effet de l'interaction est de couper ce rectangle le long d'une ligne partant du point p vers le côté droit (la double ligne dans la figure), et de coller un autre rectangle le long de cette coupe. Ce rectangle correspond aux positions dans lesquelles la séquence S_A a « pris la bifurcation ». La position en p est particulière : les deux trous sont adjacents, et aucun coup ne peut donc être joué dans la séquence S_A.

Dans la figure 3.8, nous nous sommes concentrés sur les deux séquences avec une interaction. On peut rajouter les autres, et on pourrait alors représenter l'espace de recherche en rajoutant autant de dimensions. Si ces séquences ne rajoutent pas d'interaction, l'espace de recherche complet est un simple produit entre le graphe de la figure et les séquences S_C et S_D.

Nous cherchons à partionner l'espace de recherche en blocs. Il y a plusieurs façons de le faire ; les figures 3.9 et 3.10 montrent les deux que nous utiliserons. Elles correspondent aux deux formes possibles du premier bloc. On utilisera l'une

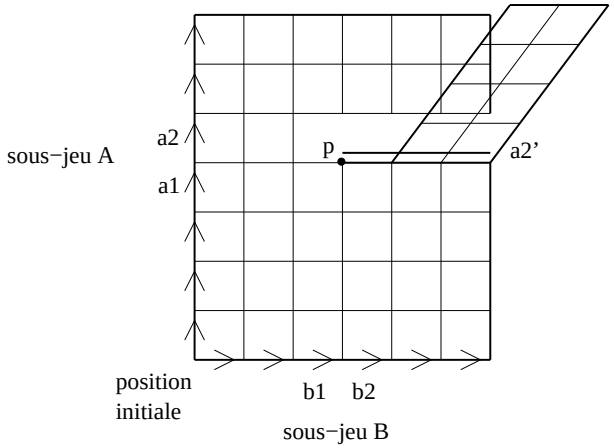


FIG. 3.8 – Espace de recherche correspondant à l'interaction de base

ou l'autre selon l'ordre dans lequel le premier bloc a été étendu : d'abord dans le sous-jeu A ou B. Nous expliquerons plus tard précisément comment détecter des interactions quand les blocs sont étendus, et comment construire les nouveaux blocs à la frontière. Remarquons, pour le moment, que dans la décomposition a, les blocs 2 et 3 sont des fils du bloc 1, alors que dans la décomposition b, les blocs 2 et 4 sont des fils du bloc 1 mais le bloc 3 est le fils du bloc 2.

Pourquoi l'interaction de base est la seule à considérer

Nous avons décrit ce que nous avons appelé l'interaction de base, et nous avons montré comment la traiter. Nous montrons maintenant que c'est la seule que nous avons besoin de considérer.

Soient S_X et S_Y deux séquences dans les sous-jeux X et Y. Quelles sont toutes les façons dont un coup y de la séquence S_Y peut influencer un coup x de la séquence S_X ? Supposons que le coup x consiste à déplacer la carte c de l'emplacement p_1 à l'emplacement p_2. Les prérequis pour ce coup sont :

1. il y a un trou en p_2,
2. la carte c est en p_1,
3. La carte à gauche de p_2, c_L, est le prédecesseur de c.

Ces prérequis sont vérifiés dans la position de la séquence S_X, juste après le coup x, mais elles pourraient être cassées par des coups de la séquence S_Y. Nous supposons que nous avons déjà établi l'indépendance des séquences S_X et S_Y jusqu'aux coups x et y.

Le prérequis 1 n'introduit pas d'interactions, car, par définition, seuls les coups de S_X changent l'emplacement du trou X.

Le prérequis 2 est aussi automatiquement vérifié. En effet, la carte c peut seulement aller à droite de c_L où que soit cette carte. Si cette carte était déplacée

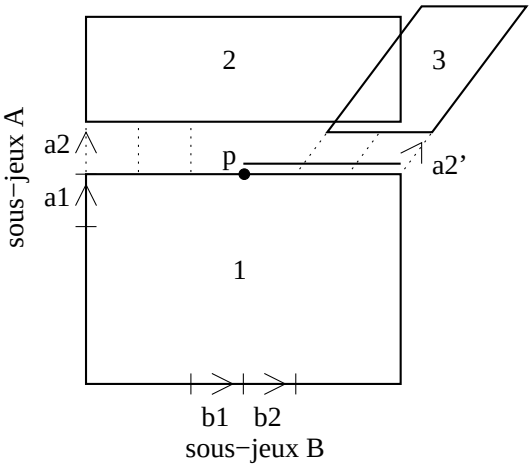


FIG. 3.9 – Décomposition en blocs : cas 1

par la séquence S_Y, alors il y aurait déjà une interaction à cause du prérequis 3.

Détection des interactions

L'intérêt de la méthode repose sur la possibilité de détecter des interactions entre les différentes séquences *sans* parcourir toutes les positions à l'intérieur des blocs, mais seulement en essayant les coups des séquences S_X séparément. Une solution à ce problème est d'utiliser la *trace* des séquences déjà construites.

Dans le contexte de montana, la trace des séquences S_X est un tableau de même taille que les positions (4 lignes de 13 colonnes). Ce tableau indique, pour chaque emplacement p, si une des séquences passe par p, et à quel avancement dans la séquence. La trace est maintenue incrémentalement quand les blocs sont étendus.

La trace permet de détecter les interactions de la façon suivante. Supposons que nous étendons le bloc courant dans un certain sous-jeu X avec un nouveau coup m, qui déplace une carte de p_1 à p_2. En regardant la trace à l'emplacement à gauche de p_2, on peut savoir si une autre séquence produit une interaction sur le coup m. En regardant la trace à droite de p_1, on peut savoir si le coup m produit une interaction sur une des autres séquences.

Construction et extension des blocs

Dans ce paragraphe, nous détaillons la méthode pour construire des nouveaux blocs aux frontières d'un bloc. Nous savons que, entre deux séquences, il ne peut y avoir qu'un seul type d'interactions. Cependant, ces interactions peuvent arriver dans les deux configurations décrites figures 3.9 et 3.10, et il

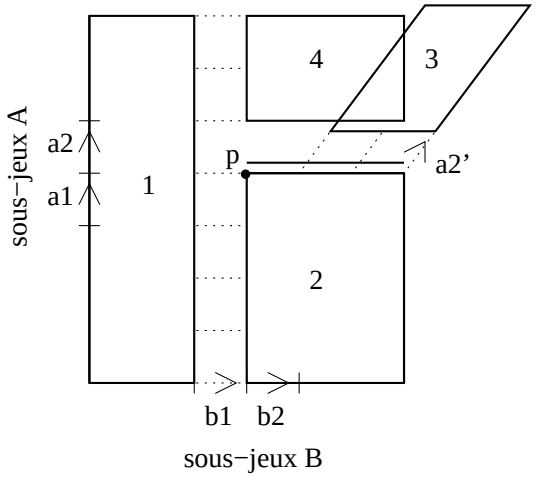


FIG. 3.10 – Décomposition en blocs : cas 2

peut y avoir des interactions entre différentes paires de séquences. Dans certains cas, if faut traiter plusieurs interactions sur la même face.

La figure 3.11 montre un exemple un peu compliqué, où on essaye de construire de nouveaux blocs sur la face B d'un bloc. Comme précédemment, pour simplifier, on suppose qu'il y a une seule famille de cartes. Il y a deux interactions à considérer. La première est due au fait que le roi peut être déplacé (coup c) et remplacé par la carte 6 (coup d), interagissant ainsi avec le sous-jeu B. Avant le coup c, aucun coup n'est possible dans le sous-jeu B à cause du roi. Après le coup d, le coup e devient possible, mais celui-ci lui-même produit une action sur le sous-jeu A.

Compte tenu de ces remarques, nous proposons la méthode suivante pour construire les blocs à la frontière d'un bloc b, sur la face F_X.

1. Nous cherchons si il y a une action d'une autre séquence qui cause une bifurcation sur S_X. C'est le cas si et seulement si la trajectoire du trou pour une autre séquence passe par l'emplacement à gauche du trou X. Ceci peut être détecté rapidement en utilisant la trace. Un exemple est l'interaction 1 dans la figure 3.11. Si une interaction est détectée, la face est séparée en deux parties avant de passer à l'étape suivante.
2. Arrivés ici, nous nous sommes restreint à une partie de la face dans laquelle le coup possible dans la séquence X, si il existe, ne dépend pas de la position sur la face S_X. Il reste à déterminer si le coup est bien possible, c-à-d si il y a bien une carte à droite du trou X et si ce n'est pas un Roi. Sans cela, aucun bloc ne sera construit à la frontière.
3. Nous savons maintenant qu'un coup est possible à la frontière ; ce coup déplace une carte d'un emplacement p dans le trou X. Y a-t-il une action de ce coup sur une autre séquence ? C'est le cas si et seulement si la

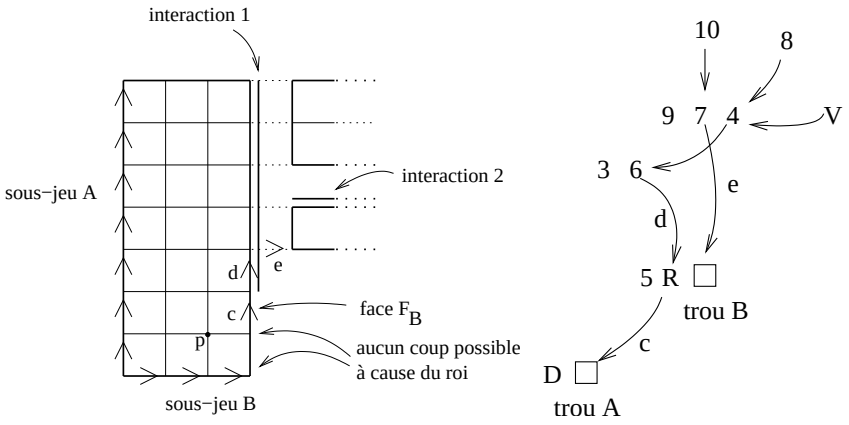


FIG. 3.11 – Plusieurs interactions sur une face

trajectoire du trou pour une autre séquence passe à droite de p. Ceci peut également être décidé rapidement en utilisant la trace. Un exemple est l'interaction 2 dans la figure 3.11.

Dans les étapes 1 et 3, la face peut donc être coupée en deux parties ; il peut donc y avoir jusqu'à 4 parties à la fin. Pour chacune de ces parties, nous sommes assurés que nous pouvons jouer un certain coup m indépendamment de la position dans la partie, et que ce coup n'interagit pas avec d'autres séquences. Nous pouvons donc créer un nouveau bloc en jouant le coup m.

Quand un bloc c est créé sur la face F_X, il a initialement une profondeur nulle dans le sous-jeu X. D'après l'algorithme de la recherche par blocs, nous essaierons ensuite d'étendre ce bloc dans chacun des sous-jeux. Il sera généralement possible de l'étendre dans la direction X. En comparaison, l'extension sera en général impossible dans les autres sous-jeux. En effet, si l'extension du bloc b (le bloc père) a été bloquée dans ces sous-jeux, c'est pour des raisons qui risquent de rester valables pour le bloc c.

Ajout d'une table de transposition

Pour rendre l'algorithme de recherche par blocs efficace, il est nécessaire de rajouter une table de transposition. Remarquons tout de même que, par rapport à une simple recherche en profondeur d'abord, l'algorithme de recherche par blocs détecte déjà un certain nombre de transpositions : toutes celles qui arrivent à l'intérieur d'un même bloc. Cependant, il peut y avoir des positions communes dans des blocs différents.

Rajouter une table de transposition pose un problème. Il serait mauvais de parcourir toutes les positions du bloc et de les marquer dans la table de transposition, car l'intérêt de la méthode de recherche par blocs est justement de ne pas parcourir toutes les positions des blocs. À la place, nous cherchons

un compromis : nous ne marquons dans la table qu'une partie des positions de chaque bloc. Nous étudions deux possibilités : la première est de ne marquer que les racines des blocs, la deuxième est de ne marquer que les positions des blocs atteignables, à partir de la racine, en ne jouant des coups que dans une seule séquence.

Pour évaluer ces deux possibilités, nous étudions le rapport entre le nombre de positions cherchées par la recherche par blocs (que nous définissons comme la somme des tailles des blocs) et la taille réelle de l'espace de recherche, comme on peut le mesurer avec une recherche en profondeur d'abord classique et une table de transposition suffisamment grande. Nous appelons ce rapport R ; nous avons $R \geq 1$. L'idéal est $R = 1$.

Dans la première possibilité (on marque seulement les racines des blocs), nous avons fait des recherches sur 100 positions initiales et obtenu $R = 3.95$. Dans la deuxième possibilité, nous obtenons $R = 1.34$. Nous considérons que la deuxième possibilité est un bon compromis, et c'est celui que nous utilisons dans la suite.

3.3.3 Résultats expérimentaux

Complétude

La méthode a été concue pour être complète ; nous avons vérifié expérimentalement que c'est bien le cas. Pour chaque position initiale, nous avons lancé une recherche en profondeur d'abord en marquant les positions parcourues dans une table de transposition, puis nous avons lancé une recherche par blocs et vérifié que toutes les positions marquées dans la table appartiennent à un seul des blocs cherchés. Nous avons ainsi vérifié 1000 positions initiales aléatoires. La table de transposition était réglée pour pouvoir contenir autant de positions que nécessaire dans les limites de la mémoire disponible (table de hachage sous forme d'un tableau de liste de positions).

variante de base, 4x13 cartes

La table 3.2 montre des statistiques pour une expérience sur 1000 positions initiales aléatoires pour la variante de base. Ici, la table de transposition est implémentée avec une table de transposition de taille fixe : 64 millions d'entrées.

nb. moyen de positions, DFS	502,000
nb. moyen de blocs	36,200
taille moyenne des blocs	18.6
R	1.34
temps moyen, DFS	2.28s
temps moyen, recherche par blocs	2.04s

TAB. 3.2 – Variante de base(4 familles, 13 cartes/famille)

La taille moyenne des blocs est de 18.6. C'est un avantage pour la recherche par blocs, mais il est réduit par plusieurs facteurs. D'abord, comme nous l'avons déjà mentionné, le nombre total de positions cherchées par la recherche par blocs, en comptant toutes les positions des blocs, est supérieur au nombre de positions cherchées par une recherche en profondeur d'abord – par environ 34%. Le résultat final est un gain de 11% pour la recherche par blocs.

Variante de base, 6x13 cartes

Ces premières expériences montrent donc la faisabilité de l'algorithme, mais le gain est faible. Les gains possibles dépendent directement de la taille moyenne des blocs. La taille d'un bloc est égale au produit des longueurs des séquences qui le composent. On peut agir là-dessus en modifiant des paramètres du jeu, comme le nombre de familles (et donc le nombre de trous) et le nombre de cartes par familles. Nous nous intéressons dans la suite à une variante avec 6 familles de 13 cartes.

Il est, dans la variante avec 6x13 cartes, difficile de donner des statistiques moyennes car la taille de problèmes varie beaucoup, et certains sont trop grands pour être cherchés même avec une recherche par bloc. Nous avons gardé 15 positions initiales qui peuvent être cherchées complètement. La figure 3.12 montre le gain en temps pour la recherche par blocs, en fonction de la taille du problème. Nous montrons également, table 3.3, des statistiques détaillées pour un de ces problèmes, pour un cas assez typique. Les blocs construits sont en moyenne plus gros et le gain devient en temps devient intéressant.

nb. de positions, DFS	289×10^6
nb. de blocs	5.00×10^6
taille moyenne des blocs	59.7
R	1.03
temps, DFS	437s
temps, recherche par blocs	44s

TAB. 3.3 – 6 familles, 13 cartes/famille, table de transposition de 64M entrées

Détection des transpositions

Dans l'expérience précédente, table 3.3, le rapport R a baissé à 1,03. Cette expérience a été faite avec une table de transposition de 64 millions d'entrées. Comment expliquer cela ? L'espace de recherche étant plus grand, et comme nous n'avions pas les ressources mémoires suffisantes pour augmenter la taille de la table de transposition, les collisions sont plus nombreuses. Mais ceci vaut surtout pour la recherche en profondeur d'abord ; nous utilisons pour la recherche par blocs un procédé qui limite beaucoup le nombre de positions stockées dans la table (voir 3.3.2). Ce qui change donc, ce n'est pas que les transpositions sont

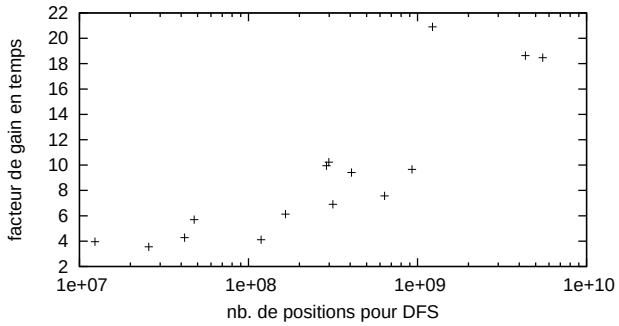


FIG. 3.12 – gain en temps de la recherche par blocs par rapport à DFS, pour 15 positions initiales

mieux détectées dans la recherche par blocs, c'est qu'elles le sont moins bien dans le cas d'une recherche en profondeur d'abord.

Pour tester cette idée, nous faisons une expérience avec la même position initiale, mais en variant la taille de la table de transposition. Nous utilisons une table avec seulement 8 millions d'entrées. Les résultats sont montrés dans la table 3.4. On observe une large hausse du nombre de nœuds pour la recherche en profondeur d'abord, alors que le nombre de nœuds pour la recherche par bloc est presque constant.

nb. de positions, DFS	1550×10^6
nb. de blocs	5.88×10^6
taille moyenne des blocs	61.1
R	0.23
temps, DFS	1730s
temps, recherche par blocs	46s

TAB. 3.4 – 6 familles, 13 cartes/famille, table de transposition de 8M entrées

Perspectives pour la recherche par blocs

Nous avons défini l'algorithme de recherche par blocs dans le cas de Montana. Nous avons utilisé un certain nombre de spécificités de ce domaine. Quelles sont les possibilités de généralisation ? Ce qu'on utilise dans le domaine, en premier lieu, c'est la possibilité de séparer le jeu en sous-jeux relativement indépendants les uns des autres.

Cependant, dans le domaine de Montana, et particulièrement dans la variante de base, les sous-jeux en question sont de nature très simple. Il s'agit de séquences linéaires. Fondamentalement, rien n'empêche de considérer des sous-jeux plus complexes représentés par des graphes quelconques. Les « blocs »

seraient alors définis comme des produits de ces graphes. Le reste de l'algorithme pourrait être adapté : il s'agit encore de construire de nouveaux blocs à la frontière et de les chercher récursivement.

La généralisation de la recherche par blocs à d'autres domaines pose cependant quelques problèmes. D'abord, l'analyse des résultats pour Montana montre que de bons résultats ne sont obtenus que dans des variantes avec des blocs suffisament gros en moyenne, c-à-d une assez grande indépendance entre les sous-jeux.

Deuxièmement, il y a des difficultés, au niveau de la conception du programme, pour l'analyser précises des interactions entre les sous-jeux et les méthodes de construction des blocs aux frontières. Dans le cas de Montana, on a vu que les interactions sur la frontière peuvent être assez compliquées.

En pratique, nos tentatives d'application à d'autres domaines n'ont pas abouti. Nous avons considéré des domaines tels que :

- Les jeux semblables à l'Âne rouge. Cette famille, appelée en anglais *sliding block puzzles*, comprend de très nombreux jeux : le taquin en fait partie, ainsi que le puzzle *Rush hour* et de nombreux autres dont la description peut être trouvée sur le Web. Les meilleurs candidats pour la recherche par bloc sont ceux avec beaucoup de trous, afin d'avoir des sous-jeux indépendants.
- Sokoban. Ce jeu parait un bon candidat. Cependant, l'investissement nécessaire pour réaliser un bon programme dans ce domaine est de toute façon élevé. Il est nécessaire d'ajouter des heuristiques dépendantes du domaine [42]. Un approche visant à rechercher des problèmes de Sokoban en se basant sur des recherches localisées a été développée par A. Botea [8]. Cette approche est sans doute plus adaptée. La décomposition en sous-jeux utilisée correspond aux « salles », comme on en trouve dans la plupart des problèmes, avec des liaisons par des « corridors ». Les positions atteignables dans chaque salles sont précalculés avant la recherche globale. Remarquons qu'il s'agit d'un exemple de décomposition plus fort que dans Montana, puisqu'elle est indépendante de l'avancement de la recherche globale, alors que la décomposition dans Montana est plus dynamique.
- D'autres patiences similaires à Montana. Un candidat est la patience *Freecell*. Il serait au moins nécessaire de séparer entre le début de la patience, avec une combinatoire faible et beaucoup de possibilités de blocage, et la fin de partie, relativement facile avec de simples heuristiques mais dont la combinatoire explose. D'autre candidats possibles seraient des patiences comme (terminologie de pysol).

3.4 Transpositions incrémentales

L'algorithme des transpositions incrémentales a été présenté en 2.2.1.

3.4.1 Étude des transpositions

Nous avons montré, sur la figure 3.5, un exemple de graphe de recherche pour une certaine position initiale du jeu. C'est une image de taille très réduite par rapport à la taille du graphe. On peut observer certaines parties fortement connectées, et des zones où des séparations ont lieu. Une chose n'est pas claire sur la représentation de ce graphe : certains nœuds sont représentés très proches alors qu'il n'y a pas de connections entre eux. En fait, une fois qu'une « séparation » a eu lieu, il n'y a pas de connection entre les différentes parties. Autrement dit, il n'a pas de « grand cycle » dans le graphe qui ne puisse être décomposé en cycles élémentaires, dus à la commutativité de deux coups. Avec la terminologie de 2.2.1, ceci indique que toutes les transpositions sont dues à des transpositions incrémentales.

Ce qu'on a observé sur l'exemple de la figure 3.5 n'est pas une loi générale ; c'est faux pour d'autres positions initiales, particulièrement quand l'espace de recherche est plus grand.

L'exemple le plus simple de transposition non incrémentale est montré figure 3.13. Les séquences m_1, m_2 et m_3, m_1, m_4 aboutissent à la même position. Il y a des cas un peu plus compliqués de transpositions incrémentales ; certaines tiennent surtout du hasard. Comme cas particulier, il y a la position finale ou les positions proches de la position finale : souvent, on peut y arriver par de nombreuses séquences très différentes, ce qui cause l'existence de transpositions non incrémentales.

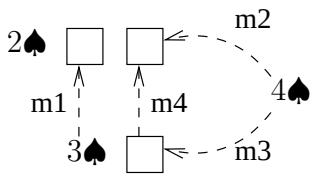


FIG. 3.13 – exemple simple de transposition non incrémentale

Notons que, dans la variante commune, il y a encore d'autres types de transpositions non incrémentales. Elles sont dues aux mouvements des Deux sur la première colonne. Un exemple est montré figure 3.14. La séquence m_1, m_2 est équivalente au coup m_3. Ce type de transpositions est beaucoup plus fréquent que le précédent. En conséquence, il y a dans la variante commune beaucoup plus de transpositions non incrémentales

3.4.2 Résultats expérimentaux pour la variante de base

Nous comparons expérimentalement l'algorithme des transpositions incrémentales avec l'utilisation d'une table de transposition, et avec la combinaison des deux. Les expériences ont été faites sur la variante de base avec 52 cartes. Nous faisons

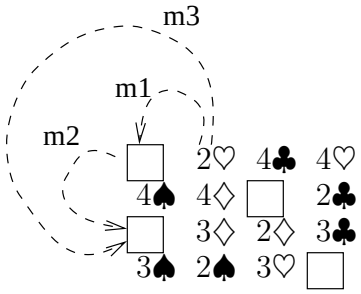


FIG. 3.14 – exemple de transpositions non incrémentales, variante commune uniquement

une recherche complète. Pour avoir des résultats plus faciles à exploiter, nous n'interrompons pas les recherches si une solution a été trouvée.

La table de transposition a été implémentée avec une table hachage avec un nombre fixé d'entrées, et une position par entrée. En cas de collision, l'ancienne position est remplacée par la nouvelle. Des méthodes de remplacement plus complexes sont possibles et donneraient des résultats légèrement meilleurs . Cependant, les variations sont faibles par rapport à ce qu'apporte l'algorithme des transpositions incrémentales ; les travaux de D. Breuker montrent, dans certaines positions d'échecs, des écarts de l'ordre de 20% [16] entre la méthode que nous utilisons et la meilleure (appelée TWOBIG 1).

Les expériences ont été faites sur un pentium 3GHz avec 1Go de mémoire. La figure 3.15 montre la vitesse, en nombre de nœuds par seconde, d'un algorithme de recherche en profondeur d'abord avec ou sans les transpositions incrémentales, et avec ou sans une table de transposition. L'algorithme de recherche en profondeur d'abord pur n'est pas utilisable en pratique. L'algorithme avec les transpositions incrémentales et la table de transposition est plus rapide que sans les transpositions incrémentales ; ceci paraît étonnant mais s'explique par le fait que les transpositions incrémentales ne nécessitent pas d'accès à la table de transpositions. Avec notre implémentation, l'accès à la table de transposition est nettement plus lent que le test de détection des transpositions incrémentales.

	sans transp. incrémentales	avec transp. incrémentales
sans table de transp.	20.4×10^6	9.5×10^6
avec table de transp.	1.7×10^6	3.4×10^6

FIG. 3.15 – nombre de positions cherchées par seconde pour les différents algorithmes

Les expériences suivantes ont été réalisées sur une série de 3000 positions

initiales ; la même série pour tous les algorithmes. Nous notons ainsi le nombre de nœuds cherchés avec les différents algorithmes :

- N_{tt_1M}, N_{tt_4M} et N_{tt_32M} : avec une table de transposition seule de taille 1M, 4M ou 32M.
- N_{ti} : algorithme de transpositions incrémentales sans table de transposition.
- $N_{ti\_tt\_64K}$, $N_{ti\_tt\_1M}$, $N_{ti\_tt\_32M}$: algorithme de transpositions incrémentales avec une table de transposition de taille 64K, 1M ou 32M.

Dans nos expérences, N_{tt_32M} a toujours été nettement inférieur à 32M ; en conséquence, les recherches avec une table de transposition de cette taille donnent des résultats proches de la taille de l'espace de recherche. Dans la suite, nous faisons l'approximation $N = N_{tt\_32M} = N_{ti\_tt\_32M}$.

Dans tous les expériences, les recherches ont été interrompues si le nombre de nœuds dépasse un milliard.

Table de transposition seule

En premier lieu, il est intéressant d'évaluer les performances d'une table de transposition indépendamment de l'utilisation des transpositions incrémentales. Les performances dépendent fortement de la taille de la table, et de la taille de l'espace de recherche.

Nous montrons sur le graphe 3.16 le rapport N_{tt_4M}/N, en fonction de la taille de l'espace de recherche N, et sur la figure 3.17 le rapport le rapport N_{tt_1M}/N. Un point correspond à une des 3000 positions initiales. Pour ces deux graphiques, les ordonnées sont en échelle logarithmique.

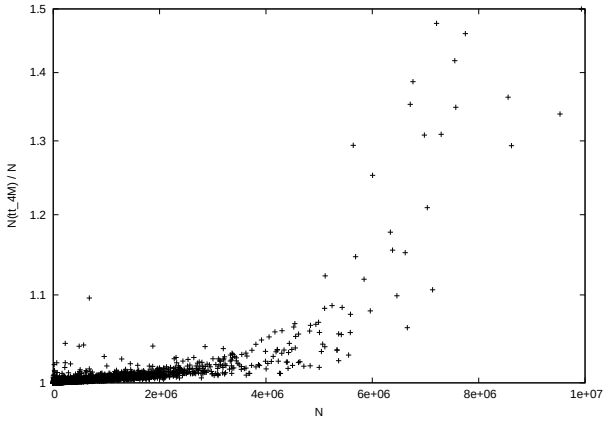


FIG. 3.16 – table de transposition seule, 4M entrées

Ces deux graphiques montrent que les nombres N_{tt_4M}, N_{tt_1M} sont très proches de N tant que la taille des problèmes est inférieure à celle de la taille de la table de transposition. Le rapport augmente nettement pour des tailles

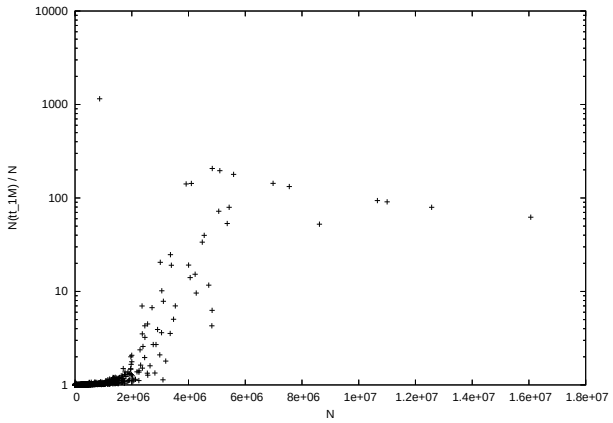


FIG. 3.17 – table de transposition seule, 1M entrées

supérieures.

Ces résultats invitent une comparaison avec ceux obtenus par D. Breuker sur les échecs et sur domineering [16]. Ses résultats montrent, par exemple, que pour des recherches de l'ordre de plusieurs millions de nœuds, l'accroissement du nombre de nœuds cherchés reste faible pour des tailles de la table supérieures à 128K. Cette différence est peut-être due aux coupes produites par l'alpha-beta, ou bien à des propriétés du domaine des échecs.

Transpositions incrémentales seules

Le graphe 3.18 montre le rapport N_{ti}/N en fonction de N. Ce graphe est tracé en coordonnées logarithmiques en abscisses et en ordonnées. Cette expérience montre que la recherche sans table de transposition est possible, mais le coût en nombre de positions cherchées est d'un facteur entre 1 et 1000. On voit que ce facteur est fortement lié à la taille de l'espace de recherche. Ces résultats confirment l'idée que la plupart des transpositions sont incrémentales.

Transpositions incrémentales avec une table de transposition

Nous placons ici dans un cas où la table de transposition est de taille nettement inférieure à l'espace de recherche. Nous fixons la taille de la table à 1M entrées. Nous évaluons l'apport de l'algorithme des transpositions incrémentales par rapport à la table de transposition seule.

Le graphe 3.19 montre le rapport $N_{ti\_tt\_1M}/N$ en fonction de N. La comparaison avec le graphe 3.17 (sans les transpositions incrémentales) montre que, avec l'algorithme des transpositions incrémentales, l'explosion du nombre de nœuds est nettement amoindrie. Le graphe 3.20 montre que le surcout reste assez limité même avec une table de transposition de taille 64K.

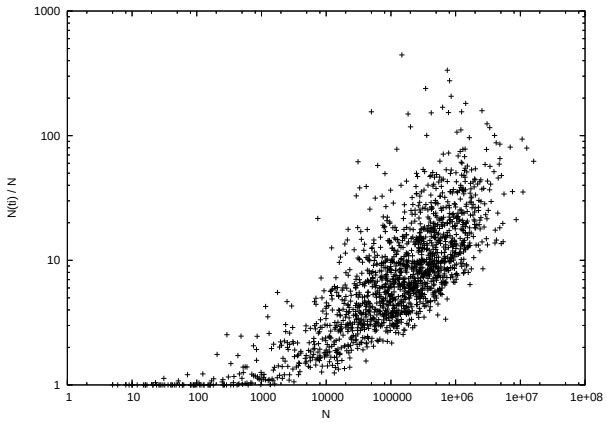


FIG. 3.18 – transpositions incrémentales seules

Conclusion

Les expériences montrent que, par rapport à une recherche en profondeur d'abord simple avec une table de transposition, l'ajout de la detection des transpositions incrémentales est utile pour améliorer les performances de la table ; ceci est particulièrement vrai si la taille de l'espace de recherche est nettement supérieur à la taille de la table.

3.5 Comparaison entre recherche par blocs et transpositions incrémentales

Nous avons étudié séparément deux algorithmes de recherche que nous avons appliqué à Montana. Nous les avons comparé séparément à une recherche en profondeur d'abord avec table de transposition. On peut se demander comment ils se comparent entre eux ; tout d'abord, nous nous demandons si il est possible de les combiner. Ceci est effectivement possible ; par contre, rajouter en plus une table de transposition pose des difficultés que nous n'avons pas résolues.

Pour pouvoir dire qu'un coup M, dans un sous-jeu X, mène à une transposition à partir d'un *bloc* (et non plus simplement une position), il faut que ce coup mène à une transposition si il est joué sur n'importe quelle position de la face S_X. Un cas simple est celui de la figure 3.21. À partir du bloc b_1, le bloc b_2 est construit sur la face S_A et cherché récursivement, puis le bloc b_3 est construit sur la face S_B. Nous rappelons que, initialement, ce bloc est construit avec une longueur nulle dans la direction du sous-jeu de S_B – disons que ce bloc a pour dimension $n-1$, où n est le nombre de trous. Le bloc des transpositions, de dimension $n-2$, qu'on pourrait construire sur la face S_A du bloc b_2 a déjà été complètement cherché par la recherche récursive du bloc b_1, en cas de commutativité des coups a et b. Pour le bloc b_3, le coup a mène donc

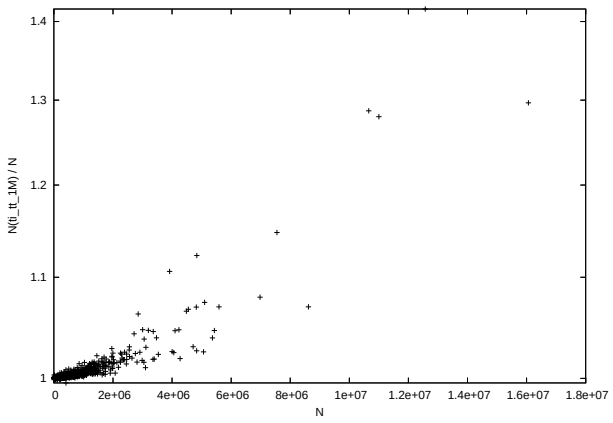


FIG. 3.19 – transpositions incrémentales et table de transposition, 1M entrées

à une transposition. Quand le bloc b_3 est étendu avec un coup b' du sous-jeu B, en cas de commutativité de a et b', le bloc des transpositions peut également être étendu avec le coup b'. Cette commutativité est en fait automatique en cas d'absence d'interactions entre les deux sous-jeux. Après l'extension du bloc b_3, on peut donc dire que le coup b mène à une transposition pour le bloc étendu.

Un algorithme basé sur ces idées a été implémenté avec succès. Cependant, l'ajout d'une table de transposition mène à des problèmes non résolus.

3.6 Conclusion

Pour une taille standard de 4x13 cartes, nos expériences, basées essentiellement sur des recherches par échantillonnage itératif, aboutissent à une probabilité de victoire de 24.8% pour la variante de base, et dans les environs de 90%-95% pour la variante commune. Nous pensons que ce genre d'algorithmes, malgré leur simplicité, est bien adapté pour ce domaine, et que l'usage d'heuristiques n'est pas indispensable.

La recherche par blocs a montré son utilité pour des tailles de jeu supérieures, en permettant d'accélerer des recherches complètes de l'espace de recherche. Plus de travail serait nécessaire pour appliquer la méthode à d'autres domaines. Cependant, nous retrouverons l'utilisation de traces de recherche dans notre travail sur la transitivité des connexions au Go.

Quant à l'algorithme des transpositions incrémentales, nous verrons qu'il est particulièrement bien adapté au morpion solitaire.

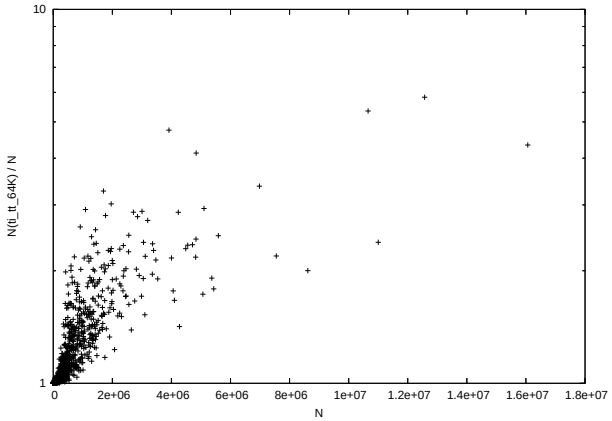


FIG. 3.20 – transpositions incrémentales et table de transposition, 64K entrées

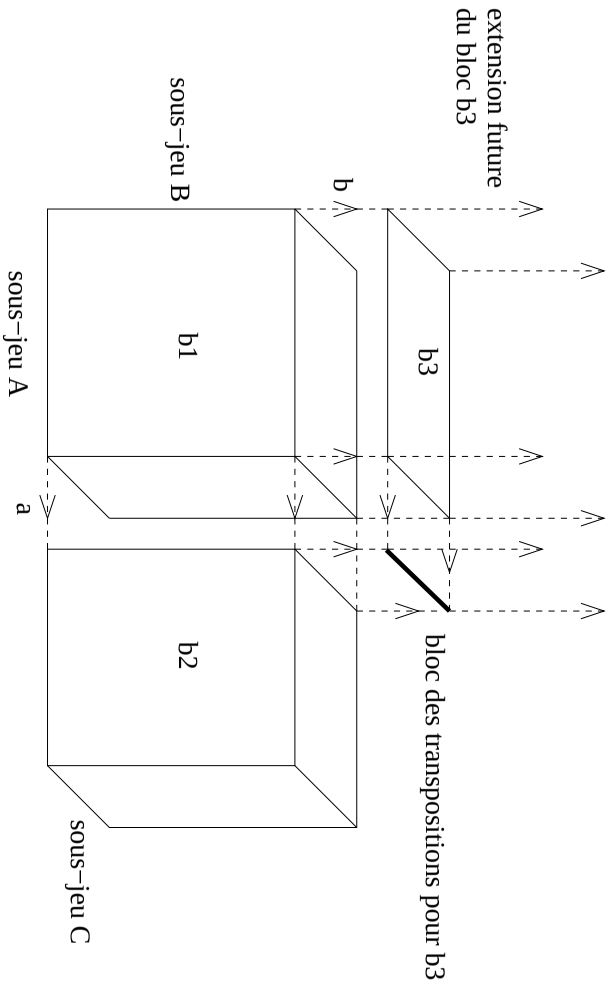


FIG. 3.21 – blocs avec transpositions incrémentales

Chapitre 4

Programmation du jeu de go

Le jeu de *Go* est né en Chine, probablement entre 700 et 1000 avant J.C., même si certaines légendes lui donnent une existence plus ancienne [15]. Les japonais ont beaucoup développé le jeu à partir du XVI^{e} siècle, et ont longtemps dominé la scène internationale devant les autres pays asiatiques. Depuis une vingtaine d'années, la Corée et la Chine ont rattrapé leur retard, et ils ont maintenant dépassé le Japon. Il existe quelques centaines de joueurs professionnels au Japon, en Chine et en Corée.

En Europe, le jeu ne s'est développé que depuis les années 1950. En France, le premier club date de 1969. Les joueurs européens sont presque tous amateurs [1].

Le jeu de Go est depuis longtemps reconnu comme un défi majeur dans le domaine de l'intelligence artificielle. Le programme GNU Go, actuellement un des meilleurs, joue régulièrement sur le serveur de Go KGS ; le bot principal est actuellement environ 13 kyu. C'est le niveau d'un joueur de club plutôt faible, très loin même des bons joueurs amateurs. Plusieurs raisons ont été invoquées pour expliquer la difficulté de programmer ce jeu [13]. Première difficulté, le nombre de positions possibles est particulièrement élevé (environ 10^{170}, contre 10^{50} pour les Échecs). De plus, il n'existe pas de fonction d'évaluation simple : une évaluation globale repose sur une analyse des positions locales et des dépendances entre celles-ci. Ces problèmes rendent presque impossible une approche basée sur une recherche en Alpha-Beta envisageant tous les coups possibles ; approche qui a en revanche été utilisée avec succès dans de nombreux autres jeux.

La question se pose des savoir si le Go est vraiment un cas à part pour la difficulté de programmation. Il existe quelques jeux qui se placent en intermédiaires

[1] Cinq européens sont devenus professionnels après avoir étudié le Go au Japon ou en Corée : Ronald Schlemper (qui n'est plus actif dans le monde du Go), Catalin Taranu, Hans Pietsch (décédé en 2003), Alexandr Dinerstein et Svetlana Shikshina.

entre les échecs et le Go ; certains ont des similarités avec le Go au niveau des règles et des méthodes de programmation utilisées. Nous commençons par nous limiter aux jeux à deux joueurs et à information complète, car des jeux comme le bridge ou le poker sont difficiles mais pour d'autres raisons. Il reste un certain nombre d'autres jeux :

- Les échecs japonais (*Shogi*), et dans une moindre mesure les échecs chinois (*Xiangqi*) sont plus difficiles à programmer que les échecs occidentaux. Au shogi, la possibilité de réutiliser les pièces capturées à l'adversaire ralonge la durée de la partie. Cela augmente également beaucoup le nombre des coups possibles dans une position. Des stratégies à plus long terme sont employées et le matériel a une importance moindre. Certains programmes de shogi utilisent un Alpha-Beta sélectif. Le niveau atteint est celui d'amateur fort.
- Le jeu d'*Hex*, inventé en 1942, repose sur l'idée de connexion entre des pions. C'est une similitude avec le Go, même si le Go, avec la règle de capture, a plus de profondeur. Une grande avancée à hex a été le travail de V. Anshelevich [3]. Son approche vise à construire des connexions de plus en plus complexes en utilisant certaines règles logiques. Comme au Go, l'évaluation globale repose donc sur des résultats locaux. Le programme de Anshelevich utilise un Alpha-Beta au niveau global. Les programmes d'Hex sont maintenant proches des meilleurs joueurs humains.
- Le jeu d'*Amazons* est relativement récent. Comme le Go, c'est un jeu de territoire. Il se distingue aussi par le grand nombre de coups possibles.
- Le jeu d'*Arimaa* a été inventé en 2002, par O. Syed, dans le but précis qu'il soit difficile à programmer. L'inventeur offre un prix de 10000$ pour le premier programme qui bat un des meilleurs humains dans un série de parties. Le meilleur programme actuel est développé par D. Fotland (auteur également du programme *Many faces of Go*) [30]. Puisque ce jeu est très récent, les joueurs humains et les programmes progressent en même temps, à mesure que de nouvelles stratégies sont découvertes. A ce jour, les meilleurs joueurs humains gardent une longueur d'avance sur le programme de D. Fotland.

Les difficultés de la programmation du Go se manifestent dans la structure des programmes de Go actuels. Typiquement, ces programmes consistent en différents modules, chacun en charge d'un certain type de problèmes (connexion, vie et mort, reconnaissance de territoires...). Il faut ensuite assembler les suggestions et les évaluations fournies par chacun de ces modules pour choisir un coup. En comparaison, un programme d'échecs, même s'il peut être compliqué, reste bâti autour d'un algorithme en Alpha-Beta global.

Notre travail sur le Go s'est fait dans deux directions différentes.

Premièrement, nous avons écrit un module spécialisé dans une certaine classe de problèmes tactiques, à savoir les problèmes de transitivité dans les connexions entre différentes chaînes. Ce module est destiné à être intégré dans un programme de Go complet. Nous avons choisi ce domaine pour deux raisons. D'abord, parce que c'est un domaine où la plupart des programmes de Go actuels ont des faiblesses. Ensuite, parce que ce domaine demande de traiter des sous-problèmes

autant que possible séparément, tout en analysant les interactions entre les sous-problèmes. Notre travail a été étendu par J. Ramon et T. Croonenborghs pour analyser des dépendances plus générales [51].

Deuxièmement, nous avons étudié une architecture pour les programmes de Go qui se différencie totalement des architectures classiques. Il s'agit de construire un programme de Go autour d'un algorithme de Monte-Carlo. Cette approche n'est pas nouvelle : elle a été décrite par B. Bruegmann dès 1993 [17]. Cependant, elle n'a presque pas été étudiée jusque récemment. Nous montrons comment l'approche initiale peut être simplifiée et nous proposons des améliorations. Bruno Bouzy est parvenu à combiner une approche de Monte-Carlo avec une ancienne version de son programme Indigo, et ainsi à améliorer sensiblement le niveau du programme.

4.1 État de l'art

Le niveau des meilleurs programmes de Go n'a que faiblement augmenté ces dernières années, et certains programmes conçus dans les années 80 ou 90 sont donc toujours actuellement dans les meilleurs. Le lecteur pourra trouver une description plus complète de ces programmes dans [9] [19], ou [50]. Nous nous concentrons ici sur l'évolution récente de la compétition.

Développer un programme de Go compétitif est un gros travail. Il faut pour cela aborder différents sous-problèmes du Go : la vie et la mort, la connexion, la reconnaissance des zones d'influence et des territoires, le début de partie, la fin de partie... De plus, ces différentes parties doivent être assemblés de façon cohérente dans un programme global. Certains programmes atteignent cependant un bon niveau avec une architecture plus simple : par exemple NeuroGo (réseau de neurones), ou Crazy Stone (Monte-Carlo go).

Une liste sommaire des meilleurs programmes de Go contient les programmes suivants (par ordre alphabétique) : Crazy Stone, de Rémi Coulom ; GNU Go ; Go++, de Michael Reiss ; Go Explorer, de Martin Müller ; Go Intellect, de Ken Chen ; Goliath, de Mark Boon (inactif depuis 1993) ; Golois, de Tristan Cazenave ; Handtalk/Goemate, de Chen Zhixing ; Haruka, de Ryuichi Kawa ; Indigo, de Bruno Bouzy ; KCC, d'une équipe nord-coréenne. Many faces of Go, de David Fotland ; NeuroGo, de Markus Enzenberger.

Une nouveauté des dernières années est la monté en puissance du logiciel libre GNU Go. Les origines de GNU Go remontent à 1989, mais le niveau était très faible ; le processus de développement a vraiment commencé en 1998. L'architecture du programme est relativement classique. La force de ce programme provient du grand nombre de programmeurs et autres personnes impliquées dans le projet. La version 3.6 contient presque 85000 lignes de code `C`.

Les compétitions actuelles comportent les *Computer Olympiads*, les tournois sur le serveur de Go KGS, ou les parties sur CGOS (*9x9 computer go server*).

4.2 Connexions transitives

Nous définissons le problème de la transitivité des connexions, et nous étudions un algorithme de recherche combinant des recherches sur les connexions séparément.

Ce travail fait suite à l'approche de la patience Montana par la recherche par blocs. Il s'agit aussi d'un cas de recherche par dépendances. La méthode est cependant assez différente. Elle a été décrite dans [24].

Notre algorithme est évalué sur une base de 22 problèmes (4.2.4), et il est comparé à un algorithme utilisant un Alpha-Beta simple.

4.2.1 Définition

Soient A, B et C trois pierres de la même couleur, C_{AB} la connexion entre A et B et C_{BC} la connexion entre B et C. Nous appelons joueur *max* le joueur qui cherche à connecter et joueur *min* celui qui cherche à déconnecter. Nous considérons comme un prérequis que les connexions C_{AB} et C_{BC} soient gagnées dans l'état initial ; sinon, *a fortiori*, la connexion transitive serait aussi perdue.

Le cas le plus simple est celui de deux connexions indépendantes, comme dans la figure 4.1.

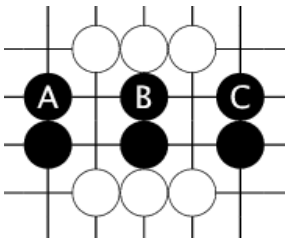


FIG. 4.1 – Deux connexions indépendantes

Les cas intéressants sont quand les connexions sont dépendantes l'une de l'autre ; c'est-à-dire quand il existe au moins un coup par le joueur min qui menace les deux connexions. La connexion transitive peut alors être gagnée ou perdue. La figure 4.2 montre un exemple où la connexion transitive est perdue si le joueur min commence : le coup de déconnexion est le point $\triangle$. La figure 4.3 montre un autre exemple : les connexions C_{AB} and C_{BC} sont dépendantes (en effet, un coup blanc en 1 menace les deux), mais la connexion transitive est quand même gagnée (par exemple, après blanc 1, noir 2 répare les deux connexions).

4.2.2 Recherche de connexions simples

Nous expliquons l'algorithme utilisé pour la recherche des connexions simples ; cet algorithme est utilisé préalablement pour vérifier que les deux connexions simples sont gagnées, et à l'intérieur de la recherche transitive.

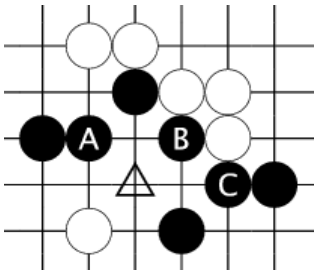


FIG. 4.2 – Connexion non transitive

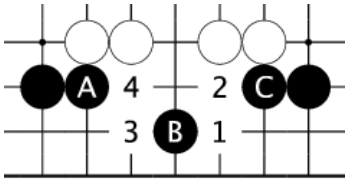


FIG. 4.3 – Connexion transitive

Nous utilisons l'algorithme de recherche par menaces généralisées (*Generalized threat search*, ou *GTS*) ; cet algorithme est décrit dans [20]. Il n'est pas forcément nécessaire de comprendre les détails de l'algorithme en question. Pour nos besoins, l'avantage de cet algorithme est qu'il renvoie, en plus du statut de la connexion, la *trace* de la recherche. Nous avons introduit cette notion en 2.1.1.

La trace est un ensemble contenant au moins tous les coups qui changent le statut de la connexion. Cet algorithme reprend, en grande partie, du code présent dans la *Tail* (Templates for artificial intelligence library), qui est une bibliothèque pour la programmation du Go développée par Tristan Cazenave.

Nous détaillons les paramètres et connaissances du domaine utilisées dans l'algorithme des menaces généralisées. La menace utilisée est (8, 5, 2, 0). La fonction d'évaluation est simple (notons que, dans l'algorithme des menaces généralisées, l'évaluation ne doit pas dépendre du joueur qui a le trait) ; elle renvoie :

- *Perdu* si une des deux chaînes à connecter est capturée en *shicho*, [2],
- *Gagné* si les deux pierres à connecter appartiennent à la même chaîne,
- *Inconnu* sinon.

Nous utilisons des fonctions spécialisées pour la génération des coups du joueur *max*. Ces fonctions dépendent de l'*ordre* de la menace considérée, c'est-à-dire le nombre de coups à la suite que le joueur max a besoin de jouer pour gagner [20]. Puisque nous utilisons à la racine la menace (8, 5, 2, 0), l'ordre

[2] Terme technique du Go, *shicho* désigne une séquence aboutissant à la capture d'une chaîne par une suite ininterrompue d'*atari* ; on parle aussi de chemin à une liberté

maximal est 3. Nous avons donc des fonctions pour trouver tous les coups qui connectent en un, deux ou trois coups. Ces fonctions sont écrites en faisant un inventaire des différents cas possibles, en considérant en particulier toutes les possibilités de connexion par capture de chaînes ennemies.

En ce qui concerne la génération des coups du joueur *min*, nous utilisons la trace de la menace qui a été vérifiée. Ceci est général à l'algorithme GTS : seuls les coups du joueur max ont besoin d'être spécifiés.

4.2.3 Recherche de connexions transitives

Au niveau de la recherche de connexions transitives, nous utilisons un algorithme en Alpha-Beta. Nous utilisons aussi les optimisations suivantes, classiques de l'Alpha-Beta : une table de transposition [16], deux *killer-move* [1] et l'heuristique de l'historique [56].

Contrairement au cas des connexions simples, un algorithme de menaces généralisées ne serait pas adapté au niveau global. Nous aurions besoin pour cela de fonctions efficaces pour calculer l'ordre de la connexion au niveau transitif, et pour générer les coups. Pour cela, les méthodes que nous avions utilisées précédemment seraient difficilement transposables, en premier lieu parce que l'ordre de la connexion devient plus élevé.

Nous décrivons dans la suite les fonctions spécifiques au domaine dans l'Alpha-Beta : la fonction d'évaluation, la fonction de génération de coups pour le joueur min, et celle pour le joueur max.

Fonction d'évaluation

La fonction d'évaluation commence par rechercher les deux connexions C_{AB} et C_{BC} séparément, avec l'algorithme présenté en 4.2.2. Selon les statuts des deux connexions et le joueur qui a le trait, il est parfois possible de connaître le statut de la connexion transitive. Nous écrivons qu'une connexion est gagnée ou perdue si le statut ne dépend pas du joueur qui a le trait ; nous disons qu'elle est gagnable sinon. Il faut utiliser deux fois l'algorithme GTS, en changeant le joueur qui a le trait, pour obtenir le statut complet de chaque connexion.

Nous considérons d'abord le cas où le joueur max (*i.e.* le joueur qui connecte) commence. Les connexions C_{AB} et C_{BC} sont d'abord cherchées deux fois, en donnant le trait à chacun des joueurs. Ces recherches fournissent, en plus des statuts de la connexion, les traces dont ils dépendent. Il est parfois possible d'en déduire le statut de la connexion transitive :

- Si une des connexions C_{AB} ou C_{BC} est perdue, alors la connexion transitive est perdue.
- Si une des connexions, par exemple C_{AB}, est gagnée, et que l'autre, C_{BC}, est gagnable, et si les traces dont dépendent les résultats sont disjointes, alors la connexion transitive est gagnable. En effet, il suffit pour gagner de jouer un coup gagnant pour C_{BC}.

De façon similaire, dans le cas ou le joueur min commence :

– Si une des connexions C_{AB} ou C_{BC} est perdue, alors la connexion transitive est perdue.
– Si les deux connexions C_{AB} et C_{BC} sont gagnées, et si les traces dont dépendent les résultats sont disjointes, alors la connexion transitive est gagnée.

Génération des coups pour le joueur min

Hors des cas présentés ci-dessus, la fonction d'évaluation ne peut pas décider du statut de la connexion transitive. L'étape suivante est donc la génération des coups à essayer ; nous commençons par le cas du joueur min. Nous cherchons à tirer parti des recherches préalables qui ont été menées sur les connexions simples.

Nous savons que les deux connexions simples sont gagnées, séparément. Nous voulons considérer comme ensemble de coups pour le joueur min les coups qui menacent de casser une des deux connexions C_{AB} ou C_{BC}. Pour des raisons pratiques, nous considérons en fait l'union des traces des deux recherches des connexions simples. Cet ensemble, calculé par le programme, peut être plus grand que strictement nécessaire, à cause de diverses approximations. Il serait trop lent de calculer une trace minimale. Un exemple de ce type d'ensembles, tel qu'il est trouvé par notre programme, est donné figure 4.4. Un ensemble minimal ne contiendrait que les 5 coups centraux.

Dans cet exemple, les deux coups qui déconnectent A et C pour blanc sont à gauche et à droite de noir B. On peut remarquer que ces deux coups menacent en fait les deux connexions simultanément. Ceci n'est cependant pas général : par exemple, dans le problème 13 de notre base de tests (figure 4.2.4), l'unique coup gagnant ne menace qu'une des deux connexions. Il est donc bien nécessaire de considérer au moins l'union des traces.

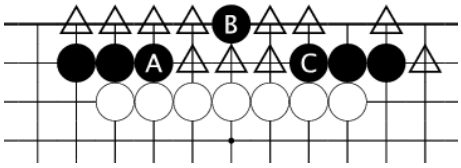


FIG. 4.4 – coups pour le joueur min

Cependant, de façon surprenante, même considérer l'union des traces n'est peut-être pas totalement sûr non plus. Ceci a bien fonctionné pour tous les problèmes de notre base de tests, mais nous connaissons un exemple artificiel de connexion transitive où l'algorithme manque un coup (figure 4.5). Le coup 1 ne menace aucune des deux connexions simples, mais il casse la connexion transitive car noir ne peut pas se défendre contre à la fois blanc 2 et blanc 3. On peut noter que blanc 3 aurait aussi cassé la connexion transitive, donc dans ce cas l'algorithme aurait quand même trouvé un coup gagnant.

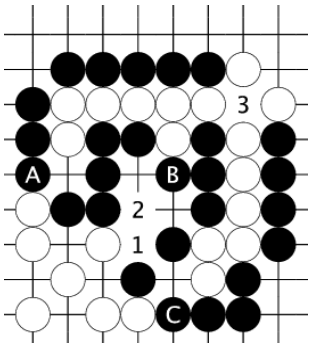


FIG. 4.5 – Position pathologique

Génération des coups pour le joueur max

Contrairement aux coups pour le joueur min, les coups pour le joueur max sont trouvés par des fonctions spécialisées. Ces ensembles de coups dépendent de l'ordre d auquel sont cherchées chacune des connexions simples dans les recherches secondaires. On peut aller jusqu'à chercher des coups d'ordre $d + 1$, puisque un problème d'ordre $d + 1$ aux nœuds max correspond à l'ordre d aux nœuds min. En pratique, nous prendrons une menace d'ordre 3 dans les recherches secondaires, et nous nous limitons à des coups d'ordre 3 pour la génération des coups max. Une raison pour se limiter est la difficulté d'écrire des fonctions de générations de coups à des ordres plus élevés, à cause de nombreux cas particuliers (se connecter directement ou en capturant des chaînes ennemies...).

Un exemple d'ensemble de coups max tel qu'il est trouvé par notre programme est montré sur la figure 4.6. Dans ce cas, cet ensemble n'est évidemment pas optimal, puisque le programme sélectionne des coups d'ordre 3 malgré le fait que les deux connexions sont d'ordre 2. Une solution simple, mais que nous n'avons pas essayée dans le cadre des connexions transitives, serait de faire un élargissement itératif sur l'ordre des connexions simples [22]. Il faudrait partir de l'ordre 1 et l'augmenter par étapes successives.

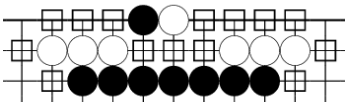
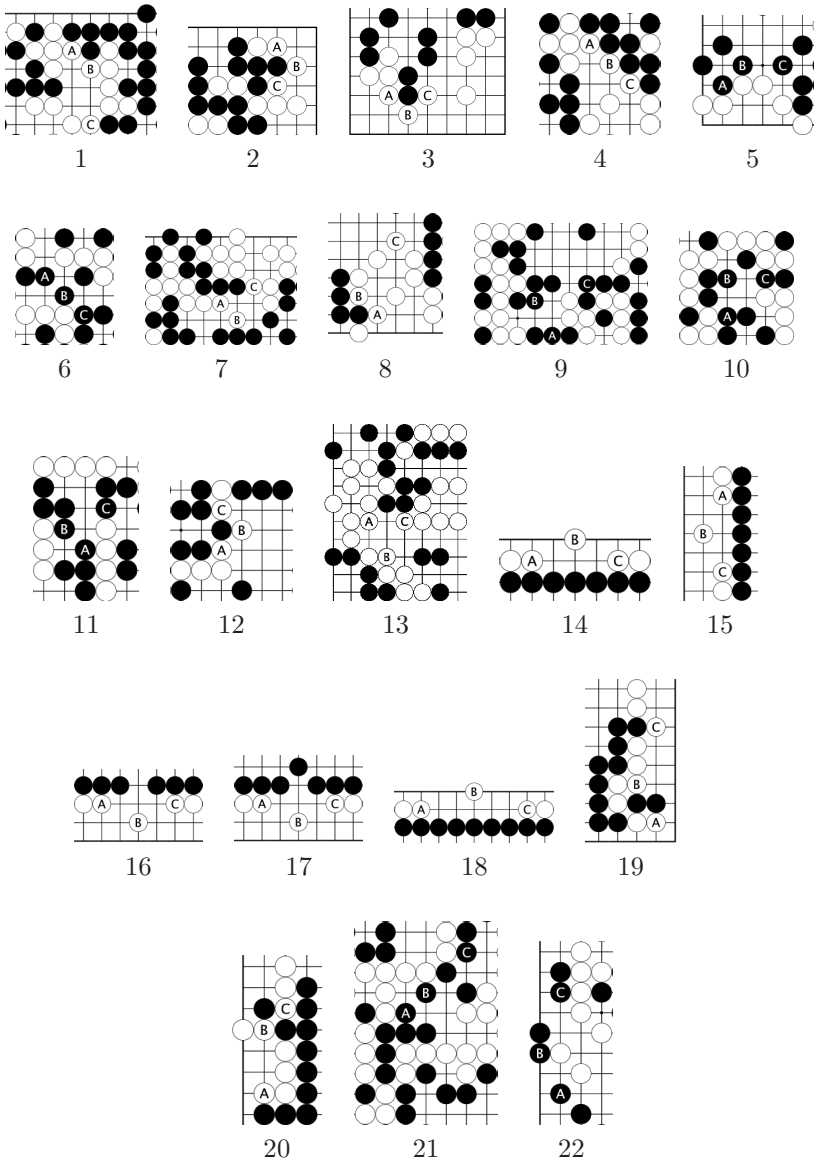


FIG. 4.6 – Un exemple d'ensemble de coups max

4.2.4 Base de problèmes

La plupart des problèmes proviennent de parties jouées par Golois, dans lesquelles des erreurs liées à la transitivité des connexions ont été commises. D'autres sont des problèmes classiques. Les problèmes numéros 20 et 21 sont à la limite entre problème de connexion et problème de *damezumari* (réduction de libertés).



4.2.5 Résultats expérimentaux

Dans nos expériences, la menace maximale utilisée pour les recherches de connexions simples est la menace $(8, 5, 2, 0)$, qui est d'ordre 3. Les coups max, pour la connexion transitive, sont donc donnés pat les coups d'ordre inférieur ou égal à 3 dans chaque connexion simple. L'algorithme de connexion transitive dépend de deux paramètres : le nombre maximum de nœuds par recherche secondaire, et le nombre total maximum de nœuds. Pour commencer, ces paramètres sont fixés à 100 000 et 10 000 000 respectivement.

Comparaison d'algorithmes

Nous comparons les algorithmes suivants.

Le premier est un algorithme sans recherches secondaires sur les connexions simples. Il s'agit d'un algorithme en Alpha-Beta, avec des optimisations classiques. Il réutilise des générateurs de coups de Golois, à la fois pour le joueur min et pour le joueur max. La fonction d'évaluation renvoie Perdu si une des deux connexions est perdue ; les connexions simples elles-mêmes sont considérées perdues si leur ordre devient strictement supérieur à 3.

Le second algorithme est celui de la recherche transitive que nous avons présenté. Le troisième est une variation dans laquelle les coups pour le joueur min sont donnés non pas par l'union des traces, mais par leur union. Nous avons expliqué que rigoureusement, il fallait prendre au moins l'union des traces, mais nous verrons qu'il est en fait plus efficace de prendre seulement l'intersection.

La table 4.1 donne, pour chaque problème de la base, le nombre de nœuds et la durée de la recherche, et ce pour chacun des trois algorithmes. Les expériences ont été faites sur un Pentium 3.0 GHz avec 2Go de RAM.

Les nombres de problèmes résolus sont à peu près similaires. Les échecs sont dus aux raisons suivantes. Le problème 3 parce que, dans une des variantes, une chaîne de noir qui coupe les pierres blanches atteint quatre libertés, et est donc considérée stable à tort. Le problème 17 parce que l'algorithme, même si il trouve le bon coup, ne voit pas que le résultat dépend d'un ko. Le problème 20, parce que le programme arrive à connecter les pierres mais ne voit pas qu'elles sont capturées en shicho (en fait, il n'est pas clair si ce problème doit être classé comme problème de connexion ou de capture). Le problème 16, à cause de la limite sur le nombre de nœuds total, et le problème 13 à cause de la limite sur le nombre de nœuds dans les recherches secondaires.

En ce qui concerne les temps de recherche, l'avantage n'est pas clair entre les deux premiers ; mais le troisième, avec les intersections des traces au lieu des unions, est nettement plus rapide dans l'ensemble, et se révèle en fait aussi bon au niveau des résultats. En effet, les seuls problèmes de la série pour lequel l'union des traces serait nécessaire sont les problèmes 13 et 21. Le problème 13 n'est de toute façon pas résolu à cause d'une limite sur le nombre de nœuds. Le problème 21 est résolu même avec l'intersection des traces, parce que les traces calculées sont parfois plus grandes que strictement nécessaire.

TAB. 4.1 – Comparaison des algorithmes

	Alpha-Beta seul			Recherche transitive union des traces			Recherche transitive intersection des traces		
Problème	nœuds	durée(ms)	résolu	nœuds	durée(ms)	résolu	nœuds	durée(ms)	résolu
1	61188	190	oui	115065	180	oui	104025	150	oui
2	348950	1390	oui	56130	90	oui	2558	10	oui
3	85582	330	non	397880	560	non	359609	490	non
4	147	10	oui	1857	0	oui	1502	0	oui
5	2687471	6850	oui	384933	620	oui	5962	10	oui
6	8655	40	oui	11670	20	oui	10463	10	oui
7	234806	1910	oui	86656	160	oui	23268	50	oui
8	43537	180	oui	5920	10	oui	1230	0	oui
9	72800	320	oui	6231	0	oui	6231	10	oui
10	101	0	oui	6503	10	oui	5471	0	oui
11	8307	100	oui	31742	50	oui	14262	20	oui
12	88152	400	oui	93959	160	oui	93959	130	oui
13	10000108	39510	non	1906627	3330	non	1913120	2720	non
14	21266	40	oui	27442	40	oui	19165	20	oui
15	198569	610	oui	372134	490	oui	194050	250	oui
16	10000256	33300	non	10437826	14320	non	1214948	1660	oui
17	10000183	29530	non	783545	1160	non	382636	490	non
18	3605	10	oui	269258	430	oui	67642	90	oui
19	3828385	16360	oui	109155	220	oui	99858	150	oui
20	22871	50	oui	10186430	14900	non	145989	250	non
21	988	0	oui	6436	260	oui	4298	10	oui
22	13703	20	oui	18846	40	oui	7723	10	oui
Total			18			17			17

TAB. 4.2 – Nombre de problèmes résolus en fonction de la durée maximale et du nombre maximal de nœuds par recherche secondaire

	Recherche transitive					Alpha-Beta seul
durée(ms)	1000	5000	10000	30000	100000	
10	7	6	9	9	8	4
30	7	10	10	11	11	5
100	7	13	14	13	13	9
300	7	13	15	15	17	11
1000	7	13	15	15	17	15

Influence des paramètres

Nous étudions l'influence du nombre maximal de nœuds par recherche secondaire. Les résultats sont représentés dans le graphe 4.7.

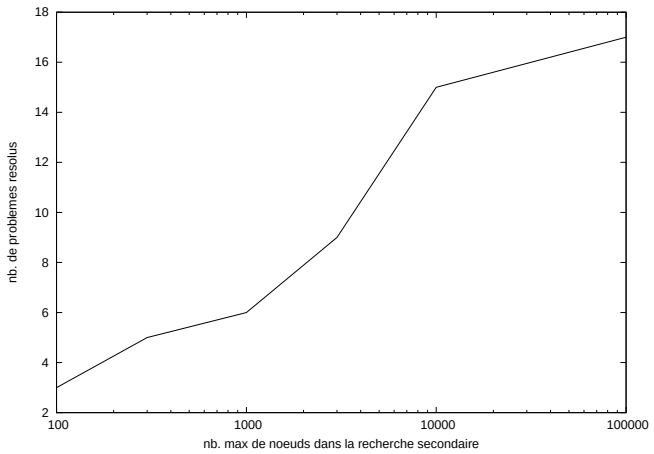


FIG. 4.7 – Influence du nombre maximum de nœuds secondaires

Nous étudions les résultats obtenus, en fixant, en plus, une borne sur le temps de recherche, et en comparant avec un Alpha-Beta seul. La table 4.2 montre le nombre de problèmes résolus. Il en ressort que 100 000 nœuds par recherche secondaire (ou peut-être plus) est bien adapté, et donne des résultats nettement meilleurs que l'Alpha-Beta seul, particulièrement pour les limites de temps réduites.

4.3 Monte-Carlo Go

L'utilisation d'une méthode de Monte-Carlo pour le jeu de Go remonte à B. Bruegmann, et à son programme *Gobble* [17].

L'idée d'utiliser une méthode de Monte-Carlo au Go peut sembler étrange. Même si ses méthodes de Monte-Carlo sont maintenant utilisées couramment pour certains jeux, il s'agit de jeux non déterministes ou à information incomplète.

4.3.1 Architecture originale de Gobble

Le programme Gobble, dans sa version originale, utilise une méthode de Monte-Carlo ; plus précisément, il utilise une méthode de *recuit simulé*. Le nom de cette méthode provient de la métallurgie : pour améliorer la disposition des atomes dans un métal, on le réchauffe puis on le laisse refroidir lentement. Au fur et à mesure que la température baisse, les atomes se fixent dans une position de plus faible énergie. La méthode du recuit simulé peut être utilisée, en général, pour trouver un optimum global d'une fonction dans un large espace de recherche.

Cette méthode est cependant appliquée d'une façon inhabituelle dans le programme Gobble. Elle est appliquée dans un jeu à deux joueurs : un des joueurs cherche à maximiser le résultat en même temps que l'autre cherche à le minimiser. Le programme Gobble applique donc le recuit simulé *simultanément* sur deux séquences de coups : une séquence de coups noirs et une séquence de coups blancs. Les deux joueurs s'efforcent *simultanément* de modifier les coups de leurs séquences pour maximiser ou minimiser le résultat.

Afin de choisir un coup dans la position courante, un grand nombre de parties aléatoires sont jouées ; en pratique, 10000 est un bon compromis. Les statistiques obtenues avec ces parties servent à modifier l'ordre des coups dans les deux séquences, et donc les parties aléatoires futures. Dans une partie aléatoire, chaque joueur joue les coups de sa séquence dans l'ordre. Si un coup d'une séquence est impossible, soit parce qu'il a déjà été joué par l'adversaire, soit à cause de la règle du ko ou du suicide, alors le coup suivant de la séquence est essayé, jusqu'à trouver un coup possible. En cas de captures, il se peut qu'un coup soit joué plusieurs fois ; pour gérer cela, il est nécessaire que les séquences initiales contiennent plusieurs fois tous les coups possibles.

Afin d'assurer la terminaison de cette partie, il faut rajouter au moins une contrainte : interdire aux joueurs de se boucher leurs yeux [3]. Sans cela, aucun groupe ne pourrait vivre, et la partie continuerait avec des captures de groupes indéfiniment.

Quand un joueur ne peut plus jouer sans se boucher ses yeux, il passe ; quand les deux joueurs passent, la partie est finie. La résultat de la partie est utilisé pour évaluer les coups des deux séquences.

Le programme Gobble évalue les coups en tirant profit de la propriété suivante : la valeur d'un coup est relativement indépendante du moment de la partie où ce coup a été joué. Dans une vraie partie, cette loi serait très critiquable ; mais dans des parties aléatoires, elle a une certaine validité. Ainsi, le

[3] Au Go, un coup qui se bouche un œil est presque toujours mauvais. Il existe de rares problèmes ou cette règle est mise en défaut, mais ce ne sont que des curiosités.

résultat d'une partie basée sur les deux séquences de coups permet d'évaluer simultanément tous les coups de la partie, pour les deux joueurs.

Ainsi, à tout moment de l'algorithme de recuit simulé, l'évaluation d'un coup est définie comme la moyenne des scores finaux des parties dans lesquelles ce coup a été joué.

L'évaluation des coups influence l'ordre des coups dans les deux séquences. Entre deux parties aléatoires, les coups de chacune des séquences sont triés selon leurs évaluations, en commençant par le meilleur pour chaque joueur. Ensuite, cet ordre est perturbé aléatoirement. Les modifications sont faites avec des probabilités qui sont fonction de la *température* courante T. Plus précisément, un coup est décalé vers le début de la séquence de n cases avec la probabilité $\exp(-n/T)$.

Au fur et à mesure de l'avancement de l'algorithme, la température diminue, et ainsi les modifications de la séquence se font plus rares. Au début, la température est élevée et les séquences sont donc presque complètement aléatoires, de même que les parties jouées. Vers la fin, les séquences sont beaucoup moins aléatoires. Finalement, le coup renvoyé par l'algorithme est celui dont l'évaluation, telle qu'elle est définie plus haut, est la meilleure pour le joueur dont c'est le tour.

4.3.2 Présentation des différents programmes de Monte-Carlo Go

Après l'article de B. Bruegmann, rien n'a été publié sur le Monte-Carlo Go, et Gobble est resté le seul programme utilisant l'idée. Nous pensons avoir été les premiers à avoir sérieusement repris l'idée de Monte-Carlo Go, à partir de fin 2001. Cependant, il existe aujourd'hui plusieurs programmes de Monte-Carlo Go, dont voici la liste par ordre approximatif de création :

Gobble Gobble, de B. Bruegmann, a été le premier programme de Monte-Carlo Go, en 1993. Il a participé a participé à la *Computer Go Ladder*, une compétition sur internet, entre 1994 et 2001, et les parties jouées sont disponibles sur la toile.

Oleg Oleg est notre programme de Monte-Carlo Go. Ce nom regroupe un certain nombre d'expérimentations différentes. En particulier, dans une certaine configuration, Oleg ressemble beaucoup à la forme originale de Gobble. Initialement, Oleg était basé sur le programme GNU Go. Nous avons surtout réutilisé les fonctions de base de gestion du goban, et nous avons remplacé les fonctions de génération de coups (qui sont la partie principale de GNU Go) par des algorithmes de Monte-Carlo. Plus tard, Oleg s'est transformé et a été réécrit à l'intérieur de la *Tail* (Templates for artificial intelligence library), qui est une bibliothèque pour la programmation des jeux développée par Tristan Cazenave. Oleg a participé aux Computer Olympiads de 2003 en 9x9, et a fini 9ème sur

10, avec 5 points sur 18. Ce score est loin d'être un échec, puisque les autres programmes présents ont été développés sur des périodes plus longues.

Olga/Indigo Olga [4] est le nom d'un programme expérimental de Bruno Bouzy, créé à partir de la version précédente de son programme Indigo. Olga reprend et développe des idées d'Oleg. Surtout, Olga réutilise une partie de la fonction d'évaluation d'Indigo pour faire du pré-traitement et se limiter aux coups les plus prometteurs. Olga s'est révélé meilleur que Indigo et est donc devenu la version suivante d'Indigo. Sous ce nom, il a fini 5ème sur 11 sur 19x19 et 4ème sur 10 sur 9x9 aux Computer Olympiads de 2003, 3ème sur 5 sur 19x19 et 4ème sur 9 sur 9x9 aux Computer Olympiads de 2004. Indigo s'est encore développé récemment avec de la recherche arborescente sélective globale [11], [12]

Vegos Vegos a été écrit par P. Kaminski vers 2003, en java, et le programme est librement téléchargable sur internet. Il est très proche de l'architecture de Gobble telle qu'elle est décrite dans [17].

Dumbgo Dumbgo, écrit par J. Hamlen, a participé aux Computer Olympiads de 2004 sur 9x9 et finit 8ème sur 9. Il reprend certaines des idées d'Olga, comme l'élagage progressif.

Crazy Stone Crazy Stone, écrit par Rémi Coulom, reprend des idées de Indigo [25]. Comme ce dernier, il combine les simulations de Monte-Carlo avec une recherche arborescente globale. Il utilise cependant des méthodes différentes pour la sélection des coups dans la recherche arborescente globale ainsi que pour faire redescendre les valeurs à la racine. Ce programme a gagné le 10^{e} tournoi de programmes de Go sur KGS.

4.3.3 Exemple de partie

Nous donnons un exemple de partie joué en 95 entre Gobble (blanc) et NeuroGo (noir). C'est une bonne partie pour Gobble, qui gagne la partie en tuant tous les groupes noirs.

Cette partie montre certaines caractéristiques qu'on retrouve dans d'autres programmes de Go utilisant des simulations de Monte-Carlo :

1. Tendance à jouer des formes solides, fortement connectées (coups 6, 8, 10, 12).
2. Faiblesses tactiques. Par exemple, le coup 28 a probablement été joué en « espérant » que l'adversaire ne réponde pas (ce qui, dans les parties aléatoires arrive effectivement souvent), mais est mauvais en cas de réponse de l'adversaire.

[4] Le nom *Olga* utilise les premières lettres de l'expression *go aLéatoire*. Le nom Oleg, inventé plus tard, est formé de façon similaire.

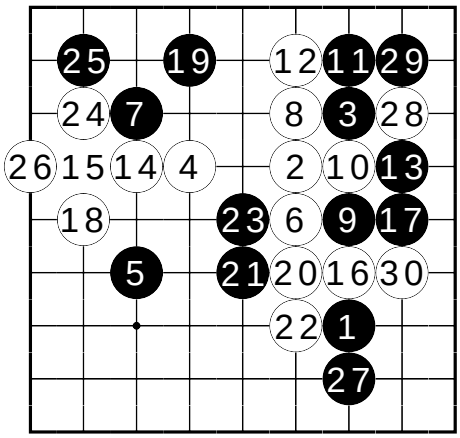


FIG. 4.8 – Gobble (B) contre NeuroGo (N)

4.3.4 Critique de l'architecture de Gobble

Le programme Gobble utilise une méthode de recuit simulé, qui est une méthode générale pour trouver des optimums de fonctions et dont l'efficacité a été prouvé. La façon dont cette méthode a été appliqué est cependant inhabituelle sur plusieurs points :

- Le programme cherche simultanément à maximiser le résultat des parties pour un joueur et à minimiser le résultat pour l'autre.
- Ce ne sont pas les séquences qui sont évaluées, mais les coups séparément.
- L'évaluation des coups ne dépend pas seulement des l'état courant des deux séquences, mais de la *moyenne* des évaluations pour toutes les parties précédentes.

4.3.5 Simplification de la méthode

Nous avons vu que l'utilisation d'une méthode de recuit simulé est un peu discutable. Nous montrons dans la suite qu'un programme de force similaire à Gobble peut être construit en utilisant des simulations de Monte-Carlo mais pas de recuit simulé. Nous aboutissons à un algorithme particulièrement simple. Cet algorithme n'est d'ailleurs pas vraiment nouveau : des variantes ont déjà été appliquées à d'autres jeux (2.3). Ce qui change ici, c'est que le hasard intervient pour modéliser les coups des joueurs et non le hasard du jeux ou l'incertitude des positions. Nous donnons un algorithme en pseudo-code pour la méthode :

```
procédure mcgo_gen_coup(position initiale P0, couleur coul) {
initialiser les évaluations des coups : e[c] = 0 pour tout coup c
pour chaque coup légal c :
   P = position obtenue en jouant le coup c à partir de P0 ;
   pour i de 1 à N :
      jouer une partie aléatoire à partir de la position P
         jusqu'à la fin, et calculer le score S ;
      e[c] = e[c] + S
si coul = NOIR :
   renvoyer le coup c tel que e[c] est maximal
sinon
   renvoyer le coup c tel que e[c] est mininal
}
```

Cette méthode est malheureusement très lente, même sur 9x9. Il existe deux méthodes pour accélérer l'algorithme. La première est la méthode de Gobble : faire des statistiques sur tous les coups joués dans une même partie aléatoire. La seconde a été proposée par B. Bouzy et s'appelle l'élagage progressif.

Sauf mention contraire, toutes les expériences suivantes ont été faites sur un goban 9x9.

4.3.6 Améliorations de la méthode

À partir de la méthode simplifiée, nous proposons et évaluons quelques améliorations. En particulier, nous analysons les quelques différences qui existent entre la méthode simplifiée et la méthode originale : les statistiques par interversion de coups, et le recuit simulé.

Statistiques par interversion de coups

Cette amélioration était déjà présente dans Gobble. Elle permet de résoudre le problème de vitesse de la variante de base. Une méthode alternative utilisée dans Olga est l'élagage itératif.

Le programme Gobble évalue les coups en tirant profit de la propriété suivante : la valeur d'un coup est relativement indépendante du moment de la partie où ce coup a été joué. Dans une vraie partie, cette loi serait très critiquable ; mais dans des parties aléatoires, elle a une certaine validité. Ainsi, le résultat d'une partie basée sur les deux séquences de coups permet d'évaluer simultanément tous les coups de la partie, pour les deux joueurs.

L'idée consiste à utiliser le résultat d'une partie aléatoire pour faire des statistiques sur tous les coups joués dans cette partie comme si ils avaient été joués en premier. Le nombre de parties aléatoires nécessaires devient donc indépendant du nombre de coups possibles ; les gains en vitesse sont très intéressants.

Cette méthode de statistiques par interversion de coups a clairement des défauts. Dire que la valeur d'un coup ne dépend pas du moment où il est joué

dans la partie est une approximation grossière. Ceci est particulièrement vrai dans des positions impliquant des captures, comme dans la figure 4.9. La valeur du coup noir A ou blanc B dépend beaucoup de ce que l'autre a déjà été joué ou pas.

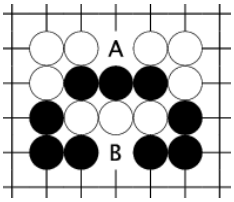


FIG. 4.9 – importance de l'ordre des coups

L'évaluation de la perte de niveau induite par l'utilisation de cette heuristique a été réalisée sur l'ancien programme Olga de B. Bouzy ; une confrontation de 100 parties entre Olga avec et sans l'heuristique aboutit à un écart moyen de 13.7 points en faveur de la variante sans l'heuristique.

En comparaison, la méthode d'élagage progressif utilisée par B. Bouzy induit une perte inférieure, de l'ordre de 5 à 10 points. Le temps de jeu est en revanche supérieur [14].

Température et recuit simulé

Nous avons expliqué que le recuit simulé, dans Gobble, est réalisé d'une façon peu courante. Par rapport à la méthode simplifiée, utiliser le recuit simulé consiste à choisir les coups des parties aléatoires en tenant compte des évaluations courantes de ces coups. De plus, la force de cette influence dépend de la température courante : au début de l'algorithme, les parties sont presque complètement aléatoires ; à la fin, presque plus.

Dans Oleg, le choix des coups en fonction de la température se fait différemment par rapport à Gobble. Nous avons expliqué (4.3.1) comment ce programme maintient une liste de tous les coups (avec des répétitions), comment cette liste est modifiée et comment les coups sont choisis dans cette liste à l'intérieur des parties aléatoires. Notre méthode est beaucoup plus simple (mais a, nous pensons, des effets semblables). Dans Oleg, un coup d'une partie aléatoire est choisi avec une probabilité proportionnelle à $\exp(Kv)$, où v est l'évaluation courante du coup, est K est une constante. En mécanique statistique, il est courant de considérer qu'un état survient avec une probabilité proportionnelle à $\exp(-E/kT)$, où E est l'énergie de l'état, k la constante de Boltzmann, et T la température. Notre constante K peut donc être vue comme l'inverse de la température. La valeur $K = 0$ correspond donc à une température infinie, où tous les états sont équiprobables.

L'utilisation de cette amélioration a un coût en terme de vitesse : il faut calculer les évaluations des coups avant les parties aléatoires, et surtout la sélection

des coups avec la loi de probabilité précédente est beaucoup plus lente qu'avec une probabilité uniforme. Ceci fait passer Oleg de 10.000 parties/s à environ 2.000, ce qui reste acceptable.

Température constante Le programmeur est libre de fixer l'évolution de la température au fil de l'algorithme. Nous étudions donc d'abord le cas d'une température constante.

L'expérience suivante a été réalisée avec Oleg, en prenant comme référence Oleg($K = 2$), et en faisant jouer 100 parties entre les deux programmes pour chaque confrontation. Avec $K = 2$ par exemple, un coup dont l'évaluation est 1 point meilleure a une probabilité d'être jouée plus grande d'un facteur $exp(2)$. En pratique, les écarts entre les meilleurs coups sont plutôt de l'ordre du dixième de point.

K	0	5	10	20
moyenne	+8.1	+2.6	+4.9	+11.3

TAB. 4.3 – Score moyen de Oleg($K = 2$) par rapport à Oleg(K)

Ces expériences montrent qu'utiliser une température constante améliore le niveau du programme. Nous n'avons pas déterminé la valeur optimale de K, mais $K =$ semble un bon compromis. Le niveau est amélioré probablement parce que les meilleurs coups sont joués plus tôt dans les parties aléatoires, et donc parce que les parties aléatoires sont de meilleure qualité. Par contre, une valeur trop élevée de K est mauvaise, puisque les parties deviennent trop peu aléatoires.

Recuit simulé Si il est intéressant d'utiliser une valeur de K strictement positive, il n'est pas évident qu'on gagne quelque chose à le faire varier au cours de l'algorithme. Par exemple, nous avons expérimenté une confrontation entre Oleg($K = 5$) et Oleg($K = 1 \rightarrow 5$) (une variante « avec recuit simulé » dans laquelle K varie graduellement entre 0 au début et 5 à la fin. Nous rappelons que K varie en sens inverse de la température). La variante avec recuit simulé a gagné de 1.6 points. C'est un écart faible qui n'est pas très significatif.

Nous avons déjà expliqué que la méthode de recuit simulé utilisée par Gobble nous paraissait critiquable (4.3.4). Notre conclusion est que cette méthode aboutit à des bons résultats par rapport aux simulations de Monte-Carlo de base, mais ce n'est pas vraiment pour les raisons habituelles associées au recuit simulé.

Recherche à profondeur 2

Dans cette variante, au lieu de faire des statistiques sur des coups, on fait des statistiques sur des paires de coups : un coup pour le joueur qui a le trait, puis un coup pour son adversaire. En principe, un nombre N de parties aléatoires est donc joué après chacune de ces paires de coups ; les valeurs moyennes sont

ensuite remontées par minimax et donnent donc une évaluation de la racine et des différents coups à la racine.

Malheureusement, cet algorithme est beaucoup trop coûteux en temps, encore plus que l'algorithme original de 4.3.5. Si n est le nombre de coups possibles à la racine, le nombre de parties aléatoires est de l'ordre de $n^2 * N$ au lieu de $n * N$. Dans Oleg, la solution a été, comme précédemment, de faire des statistiques par interversion de coups. Pour toute paire de coups joués dans la partie, on fait comme si ils avaient été joués au début.

Cette variante de Oleg à profondeur 2, et avec interversion de coups, a été testée contre la variante de Oleg à profondeur 1 avec interversion de coups. Sur 100 parties, le résultat est de 2.4 points de moins en moyenne pour la profondeur 2. Cette « amélioration » fait donc baisser le niveau du programme.

Les résultats sont donc décevants. On pourrait penser que la responsabilité vient des statistiques par interversion de coups. Or, une expérience de Monte-Carlo Go à profondeur 2 menée par B. Bouzy sur Olga, avec de l'élagage progressif au lieu des interversions de coups, a donné un résultat similaire : 2.1 points de moins en moyenne [14].

Comment expliquer ces résultats ? Nous commençons par une analyse statistique sommaire. Si on considère n variables aléatoires indépendantes $(X_i)_i$, de loi normale, centrée en 0, et d'écart-type 1, quel est la loi de la variable aléatoire $X = \max_i(X_i)$? Pour $n = 10$, on obtient empiriquement une moyenne de 1,58 et un écart-type de 0,58. Une opération de maximum sur une dizaine d'évaluations aléatoires a donc plusieurs effets : elle introduit un biais positif ; elle réduit l'ampleur des variations ; elle fait aussi que les valeurs réelles des différentes positions ont du mal à s'exprimer : en supposant qu'un coup soit légèrement meilleur que la plupart des autres coups médiocres, sa valeur ne s'exprime que si il est capable de battre *tous* les coups médiocres.

Des études sur des situations un peu semblables ont en fait déjà été faites, par exemple par A. Sadikov et al. [55]. Il s'agit d'évaluer les performances du minimax quand l'évaluation des feuilles est imprécise. Dans l'article cité, l'expérience porte sur les finales Roi+Tour contre Roi aux échecs, dans lesquelles les évaluations des feuilles ont été volontairement perturbées. L'expérience mène à une étude empirique du biais introduit en fonction de la quantité de bruit et de la profondeur du minimax. La conclusion est néanmoins, dans ce cas, qu'une recherche profonde a des effets positifs sur le niveau de jeu – au contraire donc de notre Monte-Carlo Go à profondeur 2. Nous n'avons donc pas d'explication définitive pour les mauvaise performances de la recherche à profondeur 2.

Finalement, des travaux plus récents montrent que, avec certaines modifications, la recherche arborescente à profondeur 2, ou plus, peut donner de bons résultats. Par exemple, le programme Indigo de B. Bouzy [11] aborde le problème en introduisant une sélectivité dans la recherche arborescente globale.

4.3.7 Statistiques sur les buts

La recherche tactique est une grande faiblesse de l'algorithme de base de Monte-Carlo Go. Peut-on combiner cet algorithme avec des recherches tac-

tiques fiables ? Ce problème a été abordé de différentes façons par différentes personnes. Pour notre part, nous montrons comment les simulations de Monte-Carlo peuvent être utilisées pour estimer la valeur de différents problèmes tactiques, que nous appelons des buts. Nous commençons par évoquer d'autres combinaisons possibles avec des recherches tactiques.

Autres combinaisons possibles avec des recherches tactiques

Génération pseudo-aléatoire des coups Nous avons expliqué qu'il est nécessaire d'utiliser des connaissances de base sur les yeux à l'intérieur des parties aléatoires, pour assurer la fin des parties. On peut penser que le niveau du Monte-Carlo Go peut être amélioré en améliorant la qualité des parties aléatoires, par exemple en rajoutant des connaissances supplémentaires ou en faisant des recherche simples. On parle alors de parties *pseudo-aléatoires*. D'après notre expérience, rajouter des connaissances dans la parties aléatoires est plutôt dangereux. Il est de toute façon vital d'assurer que les parties restent suffisamment aléatoires.

Une méthode possible est l'utilisation de motifs. Dans [10], il est expliqué comment utiliser des motifs 3x3 pour modifier les probabilités des coups à l'intérieur des parties pseudo-aléatoires.

Présélection des coups Cette possibilité consiste à ne faire des statistiques que sur un certain nombre de coups présélectionnés, parmi lesquels le coup joué sera choisi. C'est une solution assez simple car les problèmes sont bien séparés.

Cette méthode a été utilisée dans Indigo [10]. Des écarts jusqu'à 85 points sur 19x19 ont été observés expérimentalement, en fonction du nombre de coup présélectionnés. Le présélecteur de coup est dérivé du générateur de coups d'une version précédente de Indigo.

Principe des statistiques sur les buts

Dans cette méthode, des recherches tactiques sont préalablement menées sur des buts tactiques : capturer une chaîne, connecter des pierres, faire des yeux... Contrairement à la méthode précédente, les résultats de ces recherches ne sont pas utilisées immédiatement pour éliminer des coups. À la place, les parties aléatoires servent à déterminer quel est le but le plus important à remplir, parmi ceux dont le résultat a été déterminé incertain par la recherche tactique. Un coup est alors joué parmi les coups gagnants pour ce but. Ces travaux ont été présentés dans [23].

La méthode de base est un cas particulier de celle-ci : il suffit de considérer l'ensemble des buts qui consistent, pour chaque intersection, à jouer le premier dessus. Dans le cas général, nous considérons en plus des buts liés à des problèmes tactiques.

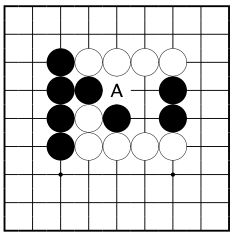


FIG. 4.10 – Connexion utile

Motivation

Dans la figure 4.10, le coup A permet de connecter les deux groupes noirs, en formant ce qui est appelé un nœud de bambou. Ceci est détecté par une recherche préalable. La valeur du but associé à la connexion est particulièrement élevée dans cette position. En comparaison, le but consistant à jouer en premier en A n'est, en pratique, pas aussi intéressant. En effet, à cause de l'absence de recherche tactique dans les parties aléatoires, le coup A ne connecte les deux groupes que, en moyenne, dans environ 3 cas sur 4 – blanc a encore statistiquement environ une chance sur 4 de couper le nœud de bambou. La valeur de ce coup serait donc sous-évaluée. Ainsi, faire des statistiques sur des buts tactiques est une voie plausible pour pallier aux faiblesses tactiques des algorithmes de Monte-Carlo Go.

Recherches tactiques des buts

Nous donnons la liste des buts tactiques que nous considérons. Nous devons définir ces buts de deux façons : pour la recherche tactique préalable, et pour les parties aléatoires. Il y a inévitablement des différences entre les deux, les recherches tactiques ayant un horizon limité.

Les buts suivants sont considérés :

- Jouer le premier sur une intersection. C'est le seul but utilisé dans l'algorithme de Monte-Carlo de base. Il n'y a pas de recherche tactique associée.
- Posséder une intersection à la fin de la partie (c-à-d que ce point est soit occupé par le joueur, ou bien est un point de territoire pour ce joueur). La recherche tactique sur ce but n'est considérée que pour les pierres déjà posées. Elle consiste en un test d'encerclement et, le cas échéant, une recherche de vie et de mort sur les groupes correspondants.
- Capturer une chaîne ennemie. Dans la recherche tactique, ce but est considéré comme perdu dès que la chaîne a quatre libertés.
- Connecter deux chaînes. Dans les parties aléatoires, il est possible que les deux chaînes, après avoir été connectées, soient capturées plus tard. Dans ce cas, le but est quand même considéré comme atteint. Dans les recherches tactiques, ce but est considéré comme perdu si la connexion devient d'ordre suffisamment grand (en pratique, d'ordre supérieur ou égal

à quatre).
- Connecter une chaîne avec une intersection vide. Ce but est similaire au précédent.
- Faire un œil sur une intersection ou sur une des voisines.

Les recherches sont réalisées par des algorithmes à base de menaces [21]. Nous utilisons les implémentations de ces méthodes qui ont été réalisées dans une bibliothèque appelée *Tail*, écrite principalement par T. Cazenave. Pour chacun des buts non triviaux présentés ci-dessus, le but des recherches est de déterminer le statut du but, ainsi que l'ensemble des coups changeant cet état dans un sens ou dans l'autre.

Nous avons utilisé la menace (6,3,2,0). Elle est d'ordre 3, ce qui signifie qu'un but (capturer, connecter, faire un œil) est considéré comme perdu si il ne peut pas être gagné en jouant 3 coups de suite.

Résultats expérimentaux

Les résultats expérimentaux indiquent que les buts tactiques les plus utiles sont la connexion et la connexion au vide. Ce sont des jeux dont il y a de nombreuses instances dans presque chaque position. Le jeu de la capture est moins utile ; une raison est que les captures liées à des problèmes de connexion sont déjà trouvés par le jeu de la connexion, et ce sont souvent les captures les plus importantes. Le jeu de la construction d'œil est également peu utile.

En utilisant seulement les jeux de la connexion et de la connexion au vide, avec un algorithme de Monte-Carlo jouant 10 000 parties aléatoires par coup, le résultat moyen sur 20 parties est de 52.1 points pour la nouvelle variante.

4.3.8 Développements récents sur les méthodes de Monte-Carlo

Les programmes de Monte-Carlo Go se sont beaucoup amélioré ces deux dernières années. Par exemple, nous avions observé de faibles performances pour une recherche à profondeur 2 (4.3.6), mais les programmes récents ont eu, au contraire, beaucoup de réussite dans cette voie. Il y a eu d'abord les recherches de R. Coulom sur les opérateurs de *backup* ; puis l'invention d'un algorithme appelé UCT (*Upper confidence bounds applied to trees*) par L. Kocsis et C. Szepesvári [44]. Cet algorithme est simple et efficace ; il a été adopté par de nombreux programmes en 2006. L'un deux est Mogo, développé par S. Gelly et Yizao Wang [32]. Il semble que, avec ce programme, un nouveau cap ait encore été franchi : d'après les parties jouées sur le serveur de Go KGS, son niveau sur 9x9 serait celui d'un joueur en Dan.

4.3.9 Conclusion

La programmation du jeu de Go passe à la fois par la résolution de problème locaux et par des approches globales. Nous avons exploré les deux de façon indépendante.

La recherche des connexions transitives utilise la notion de trace dans un cadre à deux joueurs. Les algorithmes utilisés sont considérablement différents du cas à un joueur. Notre programme est un succès car il est raisonnablement rapide et plus général qu'un programme basé sur des connaissances et des analyses de cas.

L'étude des algorithmes de Monte-Carlo appliqués au Go s'est beaucoup développé ces dernières années. Notre contribution de base a été de simplifier l'algorithme original de B. Bruegmann, et d'étudier les points qui étaient vraiment nécessaires. Ainsi, la méthode de recuit simulé n'est pas vraiment nécessaire. La simplification qui consiste à utiliser une température constante, même si elle améliore légèrement le niveau de notre programme Oleg, n'est en général pas utilisé dans les autres programmes. Avec ces recherches les les méthodes de Monte-Carlo, nous avons conttribué à un domaine de recherche qui s'est déjà révélé particulièrement fécond.

Chapitre 5

Morpion Solitaire

Le Morpion solitaire présenté ici est un bon domaine d'application pour les types d'algorithmes que nous étudions. Il a déjà été abordé par des méthodes de Monte-Carlo [40, 41]. Pour notre part, nous y appliquons avec succès la méthode des transpositions incrémentales. En parallélisant la recherche, nous avons établi un nouveau record pour une des variantes. Nous montrons aussi que l'algorithme *dbs* [2] peut être appliqué, même si son efficacité est limitée.

5.1 Présentation du jeu

5.1.1 Histoire du jeu

Les règles du Morpion Solitaire ont été publiées pour la première fois dans Science & Vie, en avril 1974. Le jeu a aussi été discuté dans plusieurs numéros de Jeux & Stratégie en 1982 et 1983. Le record actuel date de cette époque et a été obtenu à la main.

5.1.2 Règles du jeu

Variante principale

Le jeu se joue sur une feuille de papier quadrillée, idéalement infinie. La position initiale, avec 36 points, est représentée sur la figure 5.1.

Les points sont rajoutés sur les intersections de la grille. Un point peut être rajouté si il forme un alignement de cinq points horizontalement, verticalement ou diagonalement avec les points déjà présents. Un coup consiste à rajouter ce point ainsi que le segment de droite, de longueur quatre, qui passe par les cinq points. Cependant, un alignement ne peut pas être tracé si il a strictement plus d'un point en commun avec un alignement déjà tracé. Autrement dit, les alignements peuvent se croiser en un point seulement si ils ne sont pas parallèles. La figure 5.2 montre un exemple de coup légal.

Le but du jeu est de maximiser le nombre de coups.

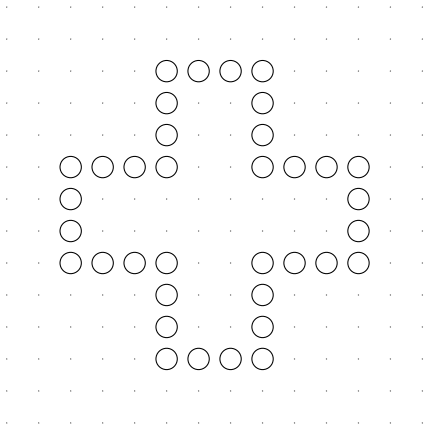


FIG. 5.1 – position initiale

Autres variantes

De nombreuses variantes peuvent être construites en changeant certains paramètres du jeu, comme la configuration initiale ou la longueur des alignements. Une modification plus importante des règles est la suivante : imposer que deux alignements parallèles ne puissent pas se toucher aux extrémités. Nous parlons alors de variante *disjointe* (*disjoint model*). La variante principale s'appelle aussi variante *avec contact touching model*.

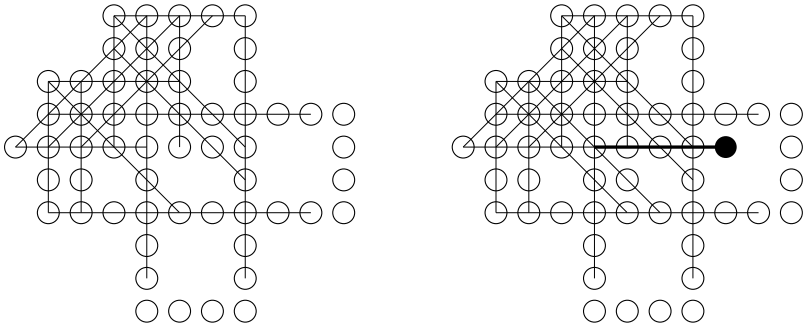
Ces différentes variations sont présentées dans [27], analysées surtout d'un point de vue mathématique, et les records actuels sont présentés. En ce qui nous concerne, nous gardons les paramètres de base et nous limitons, en plus de la variante principale, et à la variante disjointe. c'est la variante sur laquelle nous avons pu améliorer le précédent record.

5.1.3 Borne sur la longueur de la partie

Il a été montré que la longueur du jeu est bornée [27, 49], bien que nous ne connaissions pas l'auteur original de la preuve. Nous exposons rapidement cette preuve.

Chaque point peut être utilisé pour faire un alignement dans 8 directions ; dans une position donnée, certaines de ces directions sont libres et d'autres sont déjà utilisées par des alignements. Le nombre total de directions libres pour tous les points est un invariant du jeu. Quand un coup est joué, il se crée 8 directions libres nouvelles correspondant au nouveau point dessiné ; en même temps, 8 directions libres sont utilisées par l'alignement qui est dessiné : les 3 points internes de l'alignement perdent deux directions libres chacun, et les deux points extrémaux un chacun.

On sait qu'il y a des directions libres au moins pour les points sur la frontière extérieure de l'ensemble des points dessinés. Ainsi la taille de cette frontière est



(a) (b)

FIG. 5.2 – exemple de coup

limitée, et donc aussi le nombre de points qui peuvent être encerclés par cette frontière.

Des majorations précises peuvent être menées : un calcul simple aboutit à une borne de 704 coups [27].

Records

Actuellement, le record est détenu par Charles-Henri Bruneau, qui a trouvé à la main une séquence de 170 coups. Ce record date des années 1970 [49]. Il est reproduit figure 5.3.

Pour ce qui est des recherches par ordinateur, mentionnons les travaux de Hugues Juillé [40, 41]. Il a appliqué une méthode qu'il appelle *evolving non-determinism* au jeu. Cet algorithme consiste à faire évoluer une population de séquences préfixes ; les meilleures sont sélectionnées en fonction des résultats moyens des parties aléatoires qui commencent par ces séquences préfixes. Les séquences préfixes sont allongées graduellement jusqu'à atteindre la fin de la partie. Le meilleur résultat obtenu par Juillé avec cet algorithme est de 122. Une amélioration de l'algorithme par Pascal Zimmer a permis d'atteindre 147 coups [1] .

Dans la variante disjointe, il est beaucoup plus difficile de faire de longues séquences. Le record est de 69 coups ; il a été obtenu par nous-même en améliorant le précédent record de 68 coups [27].

[1] P. Zimmer a corrigé un défaut simple dans l'algorithme de Juillé. Dans l'algorithme original, les parties aléatoires jouées pour choisir le coup suivant d'une séquence préfixe étaient « oubliées » dès que le coup suivant était choisi. Pourtant, il est clairement avantageux de garder la meilleure de ces parties en mémoire, car elle est toujours valable après que la séquence ait été prolongée d'un coup. Ces informations proviennent d'une correspondance personnelle avec P. Zimmer.

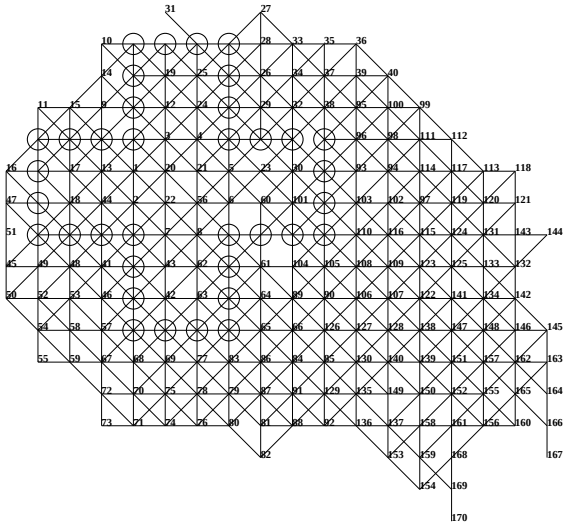


FIG. 5.3 – record de Charles-Henri Bruneau

Intérêt du jeu

Le problème de trouver de longues séquences au Morpion solitaire peut être approchés de plusieurs façons. Dans le cadre de notre travail, deux de ces approches sont particulièrement intéressantes : les méthodes de Monte-Carlo, et les méthodes basées sur des recherches complètes avec analyses de dépendances.

Certains des algorithmes que nous présentons cherchent les séquences à partir de la position initiale. D'autres servent à faire des recherches complètes et ne peuvent donc pas être utilisées à partir de la position initiale. Une utilité de ces algorithmes, cependant, est de chercher à améliorer les de bonnes séquences déjà trouvées. Nous traiterons par exemple le cas du record de 170 coups. Les recherche sont lancéees à partir de positions intermédiaires de la séquence initiale : nous partons de la position finale et remontons vers le début de la séquence.

5.2 Méthodes de Monte-Carlo

Les techniques de Monte-Carlo sont certainement un outil performant pour le Morpion solitaire, comme le montre le travail de Juillé, et comme le montre nos propres expériences. Cependant, notre travail n'est ni plus original que celui de H. Juillé, ni plus efficace puisque notre record avec ces techniques n'est que de 136 coups.

Ce qui est intéressant, c'est de remarquer la similarité entre la méthode de H. Juillé et d'autres algorithmes de jeux utilisant du Monte-Carlo. Le point

commun est que, à la base de l'algorithme, pour évaluer une séquence préfixe, on calcule la moyenne sur des simulations de Monte-Carlo.

À partir de là, on se retrouve dans le cadre d'une recherche arborescente avec une évaluation heuristique des nœuds. De nombreuses méthodes classiques peuvent donc être utilisées : recherche en *hill climbing*, recherche en faisceau (*beam search*), A*, ...([54], chapitre 4). L'algorithme de Juillé ne correspond à aucun de ceux-là : il consiste à développer simultanément plusieurs séquences préfixes, avec un mécanisme pour faire se reproduire plus vite les meilleures séquences. En ce sens, on peut dire que cet algorithme est en fait une combinaison deux idées : l'utilisation de simulations de Monte-Carlo, et la méthode de recherche arborescente (qui pourrait s'appliquer indépendamment des simulations de Monte-Carlo, avec une heuristique de n'importe quel type).

Les recherches que nous avons faites nous-mêmes sont principalement des variations autour de la recherche en faisceau, avec des heuristiques basées sur des simulations de Monte-Carlo. [2] Parmi les variations essayées, il y a :

- Évaluation des coups par simulations de Monte-Carlo, une simulation consistant en une partie aléatoire jouée jusqu'à la fin. [3]. L'évaluation est obtenue en calculant la moyenne des longueurs des parties obtenues,
- Le même algorithme en remplaçant la moyenne par le maximum, le décile ou le centile supérieur. Nous pensons que ces dernières méthodes sont un peu meilleures.
- Le même algorithme, mais dans lequel les simulations sont elles-mêmes des recherches en *hill climbing*, réglées de façon à être suffisamment rapides.

Notre record de 136 coups nous place un peu derrière le record obtenu par P. Zimmer sur une variation de l'algorithme de Juillé. Nous ne savons plus quelle est la configuration sur laquelle notre record avait été obtenu. Il serait difficile d'évaluer précisément toutes ces méthodes, à cause des nombreuses variantes possibles, et parce que les résultats sont de toute façon très aléatoires. Nous pensons cependant que la méthode Juillé est un peu meilleure à cause du mécanisme utilisé pour développer simultanément plusieurs séquences et propager les meilleures.

5.3 Application de *Dependency based search*

Dans la partie 3.3.2 à propos de la patience *Montana*, nous nous sommes intéressés à l'algorithme *Dependency based search* de V. Allis [2]. Nous avons expliqué pourquoi cet algorithme n'est pas applicable dans ce domaine. Cet algorithme nous avait servi d'inspiration pour créer l'algorithme de recherche par blocs, plus adapté dans ce cas.

[2] Nous présentons ici nos travaux comme des variations des travaux de Juillé, mais nous les avons en fait réalisés indépendamment, sans connaissance de ses travaux.

[3] L'heuristique de statistiques par interversion de coups, qui posait des problèmes au Go, marche parfaitement ici et permet de faire des statistiques sur plusieurs coups avec la même simulation

Dans le cas du Morpion solitaire, nous nous demandons de nouveau si cet algorithme est applicable et quelle est son efficacité.

5.3.1 conditions d'application

Dans [2], partie 3.3.3, les conditions suivantes sont nécessaires pour pouvoir appliquer l'algorithme *Dependency based search* :

- *monotonie* : signifie qu'un *attribut* [4] de la position ne peut pas être supprimé puis recréé. En général, cette propriété est utile pour démontrer que, pour toue séquence, si on peut changer l'ordre des coups dans la séquence, alors la position atteinte n'est pas changée. Dans notre cas, selon une formalisation naturelle du domaine du Morpion solitaire, un attribut peut être : le fait qu'un point ait été dessiné sur une intersection, ou le fait qu'un segment de ligne ait été dessiné entre deux intersections voisines en 8-connexité. Ceci étant, la monotonicité est évidente : les règles ne permettent que de dessiner des points ou des lignes, pas de les effacer.
- *singularité* : signifie que chaque position gagnante est constituée d'un seul attribut. Cette condition n'est pas vérifiée, d'abord parce que le but n'est pas de trouver une position gagnante mais de maximiser le nombre de coups. Il serait peut-être possible de contourner ce problème. On pourrait commencer par définir une position gagnante comme une position telle que le nombre de coups dépasse un seuil ; mais même ainsi, on n'aurait pas la propriété de singularité, puisqu'une position gagnante aurait de nombreux attributs.
- *absence de redondance* : un chemin est dit redondant si il est l'extension d'une solution. Le jeu lui-même n'est pas redondant si aucune solution ne l'est. Si on prend la définition précédente d'une solution, alors cette condition probablement n'est pas vérifiée.

Les conditions ne sont donc pas vérifiées. Elles pourraient peut-être l'être avec une modification de la façon de formaliser le domaine. Ceci n'est pas tellement grave, car l'algorithme reste applicable *partiellement*. Nous pouvons faire tourner l'algorithme ; le problème vient de ce que nous ne pourrons pas savoir si une position « gagnante » a été atteinte, ou quel est le maximum de coups atteignable. Tout en ayant conscience de cette limitation, nous pouvons continuer l'étude de cet algorithme et évaluer ses performances.

Dans la partie 3.3.2 sur la patience Montana, nous avons expliqué que l'algorithme *dbs* était difficilement applicable, car il était difficile de déterminer facilement la compatibilité entre plusieurs nœuds de *dbs*. Heureusement, ce problème ne se pose pas pour le Morpion solitaire (de même qu'il ne se posait pas dans les problèmes étudiés dans [2]).

[4] Selon la terminologie utilisée dans [2], les états sont définis comme des ensembles d'*attributs*. Dans la terminologie que nous utilisons (2.1.1), l'ensemble des états est un produit direct d'ensemble $\prod_{i\in I} E_i$. C'est une définition plus restrictive. Pour faire la conversion entre ce type et le précédent, on peut associer à la position $(e_i)_{i\in I}$ l'ensemble d'attributs $\{(i, e_i), i \in I\}$.

5.3.2 Performances

Nous comparons l'algorithme *Dependency based search* avec un algorithme de recherche en profondeur d'abord. Nous faisons des recherches sur la fin de partie, à partir du record de 170 coups ; c-à-d que, à partir de la position finale du record, nous défaisons successivement les coups et nous lançons à chaque fois une recherche avec les deux algorithmes.

Les recherches ont été faites sur un PC avec processeur Intel 3GHz. La recherche en profondeur d'abord utilise une table de hachage avec 64M éléments.

Les recherches par *dbs* permettent un gain en nombre de nœuds, mais ce gain est plus que perdu par le travail supplémentaire qui est nécessaire pour chaque nœud.

Il faut tenir compte évidemment de l'implémentation de l'algorithme ; des optimisations sont peut-être possibles. Il semble cependant que le domaine du Morpion solitaire n'est pas adaptée à *dbs*. Ceci est sans doute dû au relativement faible nombre de coups possibles dans une position donnée : on cherche à exploiter l'indépendance entre les coups alors que les coups ne sont pas assez indépendants. Des recherches en début de partie seraient sans doute plus favorable pour l'algorithme *dbs*, car il y a plus de coups indépendants les uns des autres. Cependant, la comparaison avec *dfs* serait difficile à faire, car on ne peut pas faire en sorte que les ensembles effectifs de positions cherchés soient les mêmes avec les deux algorithmes.

5.3.3 Exemples de graphes de recherche

Pour illustrer le fonctionnement de l'algorithme *dbs*, nous montrons en figure 5.4 le graphe correspondant à une certaine recherche ; celle-ci commence au coup 150 dans la séquence record de 170 coups. La figure 5.5 montre une recherche en profondeur d'abord à partir de la même position. La recherche par *dbs* prend 22 nœuds, et la recherche en profondeur d'abord 88.

Dans la figure 5.4, un nœud correspond à un coup ; les parents d'un nœud sont les coups qui doivent être joués pour que ce coup devienne possible. Les nœuds sans parents correspondent donc aux coups possibles dans la position initiale de la recherche (dans l'exemple, les nœuds 1 et 7). Une position correspond à un *sous-ensemble* des nœuds de ce graphe, tel que pour tout nœud du sous-ensemble, les parents du nœud sont aussi dans le sous-ensemble. La réciproque n'est pas vraie : le graphe ne contient pas, en lui-même, une information suffisante pour décrire l'espace des états. On sait quels sont tous les coups possibles, comment les réaliser séparément, mais on ne sait pas quelles combinaisons de ces coups sont possibles. Dans l'exemple, il y a 22 nœuds, mais il n'est possible d'en faire que 20 (170-150) simultanément.

Il est possible qu'un même coup puisse être atteint de deux façons différentes. Il apparaîtrait alors plusieurs fois dans le graphe. Ce phénomène ne se produit pas dans le cas présent, mais il apparaîtrait certainement dans le graphe correspondant à la position initiale.

	dfs		dbs	
coup de départ	nb. de nœuds	temps (s)	nb. de nœuds	temps (s)
170	1	0.00	1	0.00
169	2	0.00	2	0.01
168	3	0.00	3	0.00
167	5	0.00	4	0.00
166	15	0.00	6	0.00
165	20	0.00	7	0.00
164	22	0.00	8	0.00
163	34	0.00	9	0.01
162	46	0.00	10	0.01
161	48	0.00	11	0.00
160	50	0.00	12	0.00
159	52	0.00	13	0.00
158	55	0.00	14	0.00
157	57	0.00	15	0.00
156	59	0.00	16	0.00
155	61	0.00	17	0.00
154	63	0.00	18	0.00
153	74	0.00	19	0.00
152	82	0.00	20	0.00
151	85	0.00	21	0.00
150	88	0.00	22	0.00
149	91	0.00	23	0.00
148	96	0.00	24	0.00
147	104	0.00	25	0.00
146	306	0.00	68	0.01
145	334	0.00	69	0.01
144	390	0.00	70	0.00
143	652	0.00	71	0.00
142	1388	0.00	105	0.00
141	1428	0.00	106	0.00
140	2936	0.00	207	0.01
139	5892	0.00	363	0.01
138	7908	0.01	444	0.01
137	10952	0.00	567	0.01
136	14024	0.01	574	0.01
135	22132	0.02	623	0.01
134	58193	0.04	2109	0.04
133	142704	0.11	4418	0.18
132	193480	0.15	5434	0.30
131	259467	0.21	6961	0.57
130	268911	0.20	7133	0.61
129	391209	0.29	10131	1.43
128	447026	0.32	10664	1.70
127	925368	0.60	25421	23.69
126	1065476	0.61	29869	39.43
125	1065712	0.67	29870	38.10
124	1745712	1.00	58930	286.49
123	2211122	1.32	79858	616.57

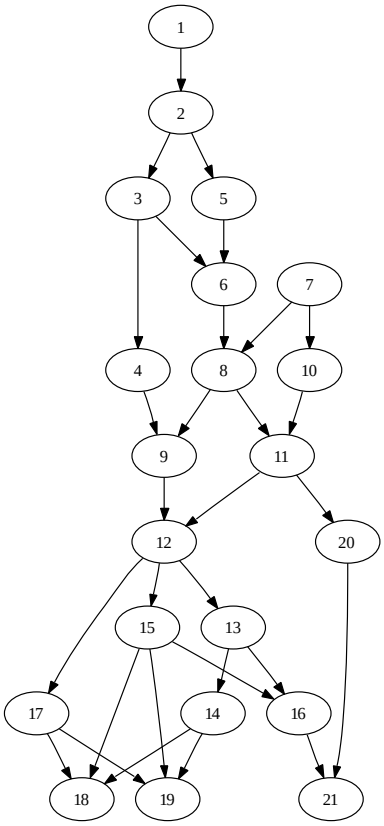


FIG. 5.4 – *dbs*, à partir du coup 150

5.4 Transpositions incrémentales

L'algorithme des transpositions incrémentales, présenté en 2.2.1, est particulièrement bien adapté pour les recherches de fin de parties au Morpion solitaire. Nous montrons que, dans ce domaine, toutes les transpositions sont dues à des transpositions de coups commutatifs ; de sorte que toutes les transpositions peuvent être détectées sans avoir même avoir recours à une table de transposition.

5.4.1 Transpositions au morpion solitaire

Nous montrons que, dans le domaine du Morpion solitaire, toutes les transpositions sont incrémentales (c-à-d qu'elles sont sont dues à une suite d'interversion de coups commutatifs adjacents). Ceci découle directement des deux lemmes suivants.

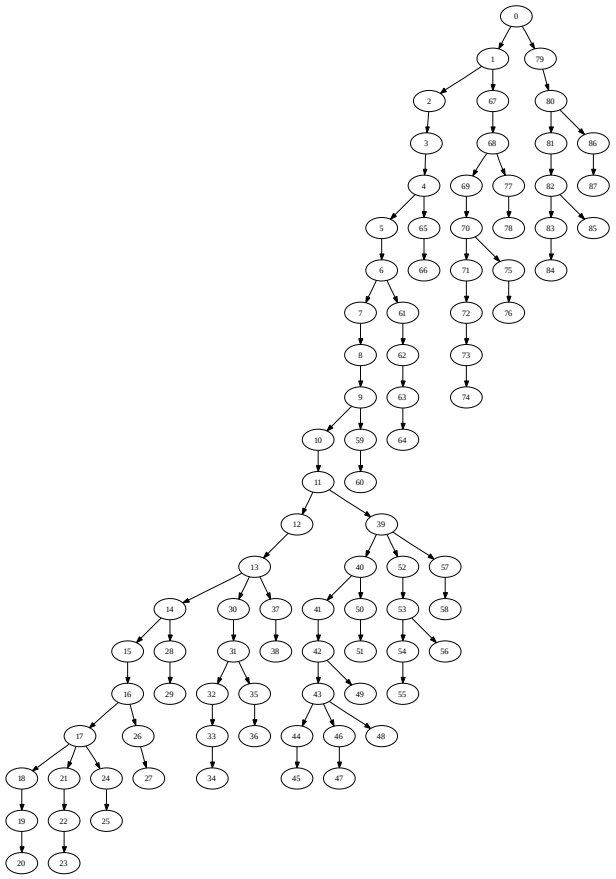


FIG. 5.5 – *dfs*, à partir du coup 150

Lemme 1 *Soit une position du Morpion solitaire. On suppose qu'elle peut être obtenue par au moins une séquence, à partir de la position initiale. Alors on peut retrouver l'ensemble (non ordonné) des coups de cette séquence.*

Dém. Nous commençons par grouper tous les segments élémentaires (c-à-d de longueur 1) dessinés en des segments de longueur 4. On peut faire cette opération séparément pour tous les segments appartenant à une même droite, et dans ce cas-là le problème est facile : il suffit de regrouper les segments 4 par 4 dans l'ordre de la droite. Les coups joués correspondent nécessairement aux alignements trouvés.

Pour chacun des alignements précédemment trouvés, il reste à trouver les positions exactes où des points ont été rajoutés. La procédure est un algorithme glouton. Nous partons de la position initiale, et, à chaque fois qu'un coup correspondant à un des alignements est possible, nous le jouons. On sera de toutes

façons obligés de le jouer, et on ne perd rien en le jouant le plus vite possible. Puisqu'il existe au moins une séquence qui mène à la position finale, l'algorithme glouton pourra également être mené jusqu'à la position finale, et l'ensemble des coups trouvés sera une permutation de l'ensemble des coups de la séquence. □

Lemme 2 *Soient deux séquences du Morpion solitaire, qui sont des permutations du même ensemble de coups. Alors on peut passer d'une séquence à l'autre par une suite d'interversions de coups adjacents.*

Dém. Soient $S = (c_1, c_2, \ldots, c_n)$ une séquence, et $S_\varphi = (c_{\varphi(1)}, c_{\varphi(2)}, \ldots, c_{\varphi(n)})$, pour une permutation φ de $\{1, \ldots, n\}$. Si $k = \varphi^{-1}(1)$, alors le coup $c_{\varphi(k-1)}$ intervient après c_1 dans la séquence S et avant c_1 dans la séquence S_φ. Les deux coups c_1 et $c_{\varphi(k-1)}$ sont donc commutatifs ; on peut les échanger dans la séquence S_φ. En répétant cette opération avec les coups $c_{\varphi(k-2)}, c_{\varphi(k-3)}, \ldots, c_{\varphi(1)}$, on peut placer le coup c_1 au début de la séquence S_φ. On conclut en raisonnant par récurrence sur la longueur des séquences, en considérant les deux séquences privées du coup c_1. □

Si on considère une transposition au Morpion solitaire, le premier lemme montre donc que les deux séquences sont des permutations du même ensemble de coups, et le deuxième montre que la transposition est incrémentale.

5.4.2 Parallélisation

L'utilisation de recherches parallèles en programmation des jeux est courant. On peut citer la parallélisation de l'Alpha-Beta dans le programme d'échecs Deep Blue [18], ou plus récemment Hydra, ou la parallélisation de l'analyse rétrograde pour la construction de bases de données de fin de parties aux checkers [57], ou pour la résolution complète du jeu Awari [53].

Nous avons montré que toutes les transpositions peuvent être détectées sans table de transposition. Les recherches dans différents sous-arbres peuvent donc être réalisées indépendamment sur des machines différentes. Notre algorithme de recherche est donc essentiellement une recherche en profondeur d'abord dans un arbre. Nous ne connaissons pas d'exemple de parallélisation d'une recherche en profondeur d'abord dans un jeu à un joueur, mais cela ne pose pas de grosses difficultés.

Nous expliquons comment nous avons résolu deux problèmes d'implémentation : le découpage de la recherche en différentes tâches, et les communications client-serveur.

Découpage en tâches

Le but est de séparer l'arbre en sous-arbres qui seront explorés indépendamment. Le problème est qu'on ne connaît pas a priori la taille des sous-arbres. Expérimentalement, on observe de grandes variations pour des sous-arbres issus de nœuds à la même profondeur. Il est cependant nécessaire de limiter le temps passé sur une tâche ; sinon on risque de perdre beaucoup en cas de problèmes sur les machines. La méthode consistant à construire autant de tâches que de

nœuds à une certaine profondeur n'est pas très bonne, car elle ne permet pas de limiter la taille des tâches autrement que par essais et erreurs.

Nous procédons de la façon suivante. Une tâche est déterminée par le nœud racine du sous-arbre, et par l'ensemble T qui est passé en argument de la fonction `dfs_ti` (2.2.1) pour ce nœud. La recherche se fait par l'algorithme des transpositions incrémentales, avec la modification suivante : si l'un des appels récursifs, *à lui seul*, a exploré plus de un milliard de nœuds au moment où il se termine, alors tous les appels récursifs suivants sont annulés. De nouvelles tâches sont crées à la place, et renvoyées au serveur. La figure 5.6 explique ce procédé. Au tout début de la recherche parallèle, une seule tâche existe (et donc un seul client travaille), mais d'autres tâches apparaissent rapidement, si bien que les clients ont généralement toujours du travail.

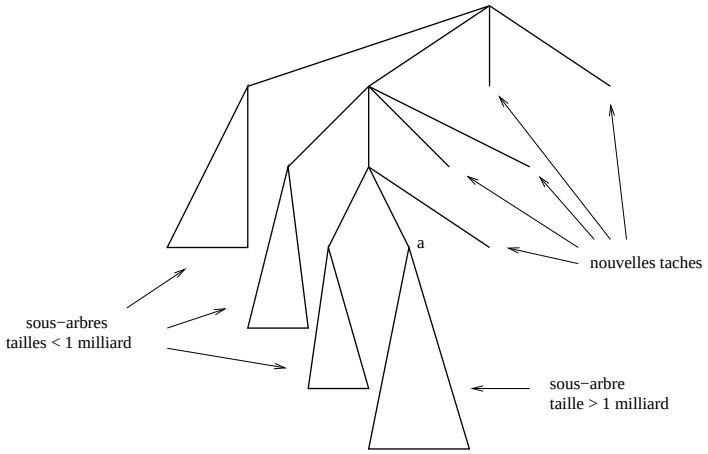


FIG. 5.6 – exemple de répartition des tâches. La recherche se fait en profondeur d'abord, de gauche à droite

L'idée derrière cette décomposition est la suivante. Étant données les relations de parenté entre les racines des nouvelles tâches et le nœud a (voir figure 5.6), ces nouvelles tâches ont des chances d'être au moins aussi grandes en nombre de nœuds que le sous-arbre issu de a. Or celui-ci, par construction, a au moins un milliard de nœuds. On évite donc d'avoir trop de petites tâches.

En pratique, sur une des expériences menées, mous avons mesuré le nombre moyen de nœuds par tâche à 180 millions, avec un maximum à 4.49 milliards. Il existe des tâches très petites, mêmes avec un seul nœud, mais elles ne sont pas trop nombreuses. Nous concluons que la solution apportée au problème de la répartition des tâches est tout à fait satisfaisante.

Architecture client-serveur

Les tâches à réaliser ainsi que les résultats des tâches déjà réalisées sont stockés sur le serveur ; celui-ci se charge d'envoyer les tâches aux clients et de

récupérer les résultats. La tâche envoyée est celle qui, parmi toutes les tâches à traiter à l'instant courant, respecte le mieux l'ordre de la recherche en profondeur d'abord. Deux implémentations différentes ont été essayées.

La première a été codée en bas niveau, avec des sockets TCP. Les tâches sont stockées sur disque sous la forme de fichiers indépendants, qui sont déplacés dans divers répertoires en fonction de leur statut (à traiter, en traitement, traité).

La deuxième remplace le serveur fait-main par un serveur de bases de données. Nous avons choisi le serveur *postgreSQL*. Les clients se connectent directement au serveur de bases de données par l'intermédiaire d'une bibliothèque standard de *postgreSQL* (*libpq*).

Après programmation et utilisation des deux implémentations, si les deux se sont révélées viables, il est évident que la solution utilisant des bases de données est nettement meilleure, pour les raisons suivantes :

- Facilité de programmation. Seul le client a besoin d'être écrit. L'écriture du serveur est remplacée par la configuration du serveur SQL.
- Bien meilleure résistance en cas de dysfonctionnement machine ou réseau. [5]
- Facilité de construction de statistiques, par l'utilisation des fonctions SQL standards.
- Possibilité de généraliser plus facilement la méthode de recherche, en changeant l'ordre de traitement des tâches. On peut par exemple introduire du hasard ou des évaluations heuristiques. Puisque la recherche n'est plus une recherche en profondeur d'abord, le nombre de tâches en attente de traitement peut être nettement plus grand ; cependant, avec une bonne configuration du serveur SQL (utilisation d'index), celui-ci reste efficace.

Résultats expérimentaux

Nous appliquons la version parallélisée de l'algorithme des transpositions incrémentales pour déterminer si la séquence record de 170 coups est optimale.

En appliquant la version parallélisée de l'algorithme des transpositions incrémentales, nous pouvons étudier si une séquence donnée est optimale dans la fin de partie. Nous lançons successivement des recherches à partir de toutes les positions traversées par la séquence, en partant de la fin. Nous faisons en sorte qu'une recherche à partir d'une certaine position ne soit pas refaite lors des recherches à partir des positions parents.

Nous avons appliqué ce processus de recherche sur la séquence record de 170 coups dans la variante principale, et dans la séquence record de 68 coups de la variante disjointe 5.1.2.

Un client tournant sur un Pentium 3GHz peut chercher environ 3.7×10^6 positions/s. Les recherches ont été faites sur une vingtaine d'ordinateurs différents,

[5] Divers problèmes se sont posés avec la première implémentation. Le premier de ces problèmes a été quand le répertoire contenant les tâches traitées a atteint quelques centaines de milliers de fichiers, ce qui a provoqué un crash machine et la perte des données. La solution a été d'assembler tous ces petits fichiers en de plus gros. D'autres problèmes matériels ont abouti à des résultats de tâches incohérents, et ils ont du être détectés et réparés *a posteriori*.

mais la plupart sont des stations de travail Alpha ou Sun relativement peu puissantes.

Variante principale Dans la variante principale (avec contact), en étudiant l'optimalité de la séquence record de 170 coups, la recherche a été menée jusqu'au coup 61 de cette séquence. Un total de 3.97×10^{13} nœuds ont été cherchés. [6] La figure 5.7 montre l'évolution du nombre de nœuds cherchés en fonction de la profondeur de départ. La figure 5.8 montre une comparaison entre les positions après les coups 66 et 170.

Nous avons montré que le record de 170 coups est optimal sur les 109 derniers coups. Ceci est remarquable, étant donné que ce record a été obtenu à la main. Il existe un grand nombre de séquences de même longueur, mais celles-ci sont des variation mineures par rapport au record original.

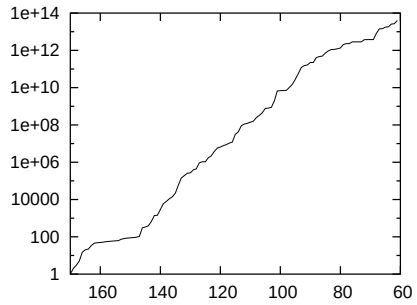


FIG. 5.7 – nombre de positions cherchées en fonction de la profondeur de départ

Variante disjointe Dans la version disjointe, l'ancien record de 68 coups a été amélioré. En remontant au coup 28 de la séquence, une séquence de 69 coups peut être trouvée. Les positions après le coup 28, pour l'ancien et le nouveau record sont montrées figure 5.10.

La table 5.9 montre le nombre de positions cherchées en fonction de la profondeur de départ. Un total de 5.79×10^{13} ont été cherchées. Pour les recherches aux profondeurs 26 et 27, nous avons volontairement fait des recherches incomplètes (le nombre de nœuds à ces profondeurs est donc une borne inférieure). Toutes les tâches traitées l'ont été complètement, mais toutes n'ont pas été traitées. À ces profondeurs, nous avons introduit du hasard dans l'ordonnancement des tâches qui sont envoyées aux clients.

[6] Cette recherche a été réalisée avec la première implémentation client-serveur. À plusieurs reprises, nous avons observé que la recherche avait été corrompue. La plupart de ces corruptions ont été réparées, mais il est probable que certaines existent encore, particulièrement sur la recherche à profondeur 61. À cette profondeur, une faible erreur sur le nombre de nœuds est probable.

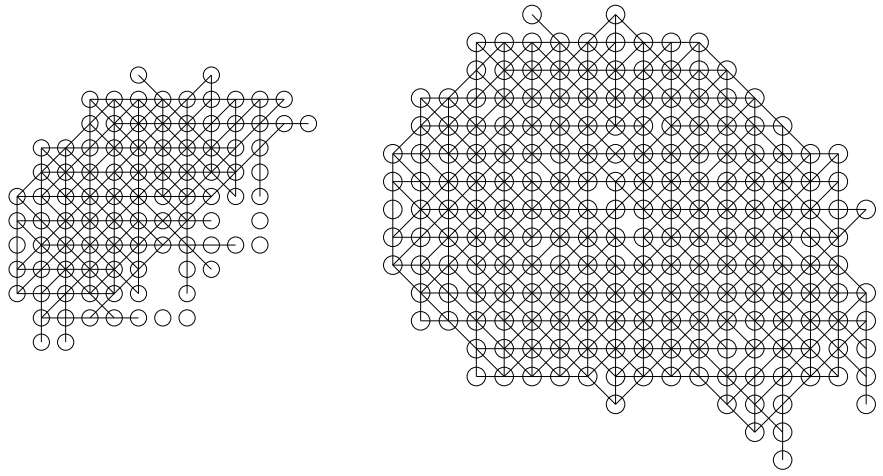


FIG. 5.8 – positions après le coup 61 et après le coup 170

5.4.3 Conclusion et perspectives

Actuellement, les algorithmes les plus efficaces pour la construction, à partir de zéro, de longues séquences au Morpion solitaire sont basés sur des simulations de Monte Carlo. Notre apport principal à ce jeu concerne les analyses de fin de partie, pour essayer d'améliorer des séquences existantes. Nous avons ainsi analysé des fins de partie pour les deux séquences record de la variante principale et de la variante disjointe.

Ceci a d'abord été tenté en appliquant l'algorithme *Dependency base search*. À notre connaissance, il s'agit de la première application de cet algorithme à un nouveau domaine depuis les travaux de V. Allis. Les résultats sont cependant

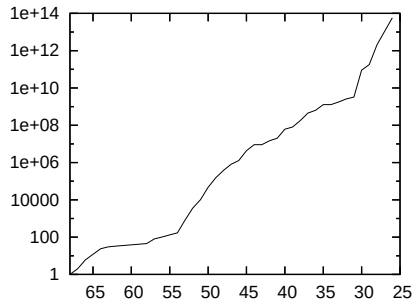


FIG. 5.9 – nombre de positions cherchées en fonction de la profondeur de départ

moins bon qu'une simple recherche en profondeur d'abord.

Nous avons eu beaucoup plus de réussite en appliquant simplement l'algorithme des transpositions incrémentales. La recherche pour la variante disjointe a abouti a une amélioration du précédent record ; ceci montre l'intérêt de la méthode. Il est tout de même frappant que, malgré les profondeurs de recherche atteintes, le record de la variante principale n'ait pas pu être amélioré.

Dans le but de battre les records, il faut probablement rajouter des heuristiques ou du hasard dans la recherche, comme nous avons commencé à le faire. Tout en restant dans le cadre présenté, on peut jouer sur les méthodes de choix pour la prochaine tâche à chercher. Il est envisageable, par exemple, d'utiliser des recherches de Monte-Carlo rapides pour évaluer les différentes tâches.

Même si les recherches complètes sont fondamentalement peu efficaces, elles permettent de faire des statistiques précises. Nous avons obtenu beaucoup de données sur les recherches effectuées. Par exemple, nous connaissons l'arbre des tâches qui ont été traitées, et nous avons des statistiques sur chacune d'elles, telles que le nombre de nœuds cherchés ou la profondeur maximum atteinte. Il s'agit donc d'une vue synthétique sur l'arbre de recherche total, suffisamment simplifiée pour pouvoir être cherchée rapidement (environ 500 000 tâches sur la première recherche). Dans la perspective d'étudier des algorithmes plus sophistiqués, ces données peuvent être utiles.

prof. de départ	nb. de nœuds cumulés	prof. de départ	nb. de nœuds cumulés
170	1	115	31867703
169	2	114	43381885
168	3	113	90451292
167	5	112	110753604
166	15	111	122907880
165	20	110	140961218
164	22	109	160359754
163	34	108	247814711
162	46	107	332114369
161	48	106	453159128
160	50	105	770689327
159	52	104	800312869
158	55	103	884572569
157	57	102	1966072161
156	59	101	6.71333×10^{9}
155	61	100	6.89888×10^{9}
154	63	99	7.05741×10^{9}
153	74	98	7.18415×10^{9}
152	82	97	1.02616×10^{10}
151	85	96	1.53168×10^{10}
150	88	95	2.78013×10^{10}
149	91	94	5.72995×10^{10}
148	96	93	1.19628×10^{11}
147	104	92	1.55476×10^{11}
146	306	91	1.63556×10^{11}
145	334	90	2.25626×10^{11}
144	390	89	2.25626×10^{11}
143	652	88	4.11194×10^{11}
142	1388	87	4.67248×10^{11}
141	1428	86	4.99047×10^{11}
140	2936	85	7.23905×10^{11}
139	5892	84	9.36143×10^{11}
138	7908	83	1.11926×10^{12}
137	10952	82	1.13405×10^{12}
136	14024	81	1.23083×10^{12}
135	22132	80	1.35373×10^{12}
134	58193	79	2.05288×10^{12}
133	142704	78	2.32884×10^{12}
132	193480	77	2.32885×10^{12}
131	259467	76	2.89103×10^{12}
130	268911	75	2.89103×10^{12}
129	391209	74	2.89103×10^{12}
128	447026	73	2.89103×10^{12}
127	925368	72	3.75872×10^{12}
126	1065476	71	3.78317×10^{12}
125	1065712	70	3.78978×10^{12}
124	1745712	69	3.78979×10^{12}
123	2211122	68	8.04926×10^{12}
122	3801787	67	1.41974×10^{13}
121	5726525	66	1.45221×10^{13}
120	6597732	65	1.77244×10^{13}
119	7828295	64	1.83049×10^{13}
118	8882005	63	2.59876×10^{13}
117	10622362	62	2.70392×10^{13}
116	12049820	61	3.97674×10^{13}

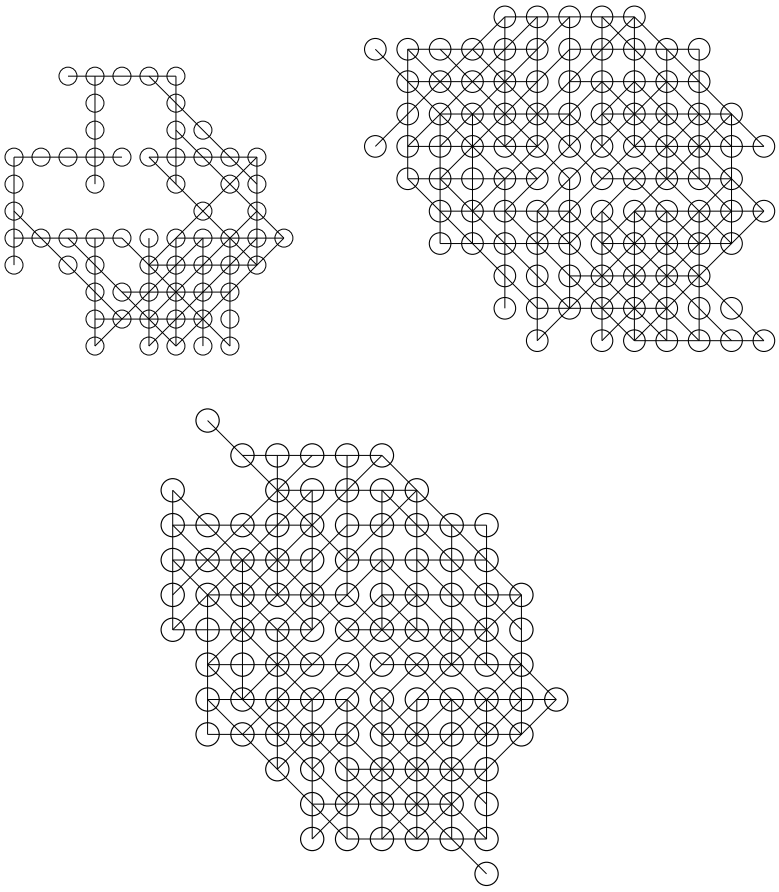


FIG. 5.10 – positions après le coups 28, après le coup 68 de l'ancien record, et après le coup 69 du nouveau record

Chapitre 6

Conclusion

Notre recherche s'est menée dans deux directions : ce que nous avons appelé des recherches par analyses de dépendances, et des algorithmes basés sur des simulations de Monte-Carlo. Ces familles de méthodes sont très différentes. Cependant, notre expérience montre qu'elles peuvent fournir des approches efficaces dans de mêmes domaines. Peut-être que des approches idéales combineraient les deux ; mais ceci est au delà de notre travail.

Ce que nous avons appelé analyse de dépendances repose sur quelques idées communes. L'idée principale est d'analyser les relations entre les coups pour accélérer une recherche. En général, cela se fait par une décomposition du jeu en sous-jeux. Nous avons partiellement formalisé cette notion. Il y a au moins deux façons de définir un sous-jeu : en caractérisant les coups jouables dans chaque sous-jeu, ou selon les buts à atteindre dans chaque sous-jeu. Une notion importante, pour cela, est celle de la trace d'une recherche.

Dans ce cadre, trois choses ressortent de notre travail. Pour notre recherche sur Montana, nous avons exposé un algorithme que nous avons appelé recherche par blocs. Un avantage de la recherche par blocs est qu'elle permet de minimiser les recherches sur les sous-jeux en ne recalculant que ceux qui ont changé quand la recherche globale avance. À Montana, une décomposition naturelle en sous-jeux est donnée par les coups utilisant les différents trous. Nous avons utilisé la structure sans branchement des sous-jeux. L'algorithme pourrait probablement fonctionner aussi si les sous-jeux ont des branchements. Les résultats sont positifs mais mitigés : des gains significatifs ne peuvent être atteints que pour un niveau d'indépendance entre les sous-jeux qui est supérieur à celui des règles standards.

Pour la transitivité des connexions au Go, le passage à deux joueurs a conduit à un algorithme différent. Les différents sous-jeux sont définis par des buts à atteindre, à savoir les connexions simples. Le programme repose sur un Alpha-Beta au niveau global, et sur des recherches sélectives avec des menaces généralisées sur les connexions simples. L'intérêt de cet algorithme est de fournir une trace des recherches qui permet dans certains cas de conclure au niveau global, et de trouver les coups à jouer aux nœuds min. Les résultats obtenus sur une base de

problèmes montrent qu'il est souvent meilleur de chercher les connexions simples indépendamment et de se servir des résultats pour guider la recherche globale.

L'algorithme des transpositions incrémentales est différent des précédents car il ne repose pas sur une décomposition en sous-jeux ; son but est d'améliorer la détection des transpositions. On peut ainsi accélérer la recherche, mais indirectement, dans les cas où les ressources mémoires sont insuffisantes pour une table de transposition suffisamment grande. Nous avons montré l'efficacité de cet algorithme dans le domaine de Montana, et surtout dans nos recherches au Morpion solitaire. Nos recherches sur la variante « sans contact » de ce jeu nous a permis d'améliorer le record précédent, de 68 à 69 coups. Dans la variante commune, le record de 170 coups a été prouvé optimal sur les 109 deniers coups, avec une recherche de presque 4×10^{13} positions.

L'autre direction de notre recherche a été les recherches par simulations de Monte-Carlo. Ces méthodes ont l'avantage d'être assez simples à coder, et elles permettent de saisir des aspects globaux des jeux qu'il est difficile de coder directement.

Dans le domaine du Go, l'application de ces méthodes donne rapidement un programme dont le jeu a certaines qualités intéressantes. Il cherche naturellement à faire des formes solides et connectées. Les faiblesses sont évidemment les recherches tactiques. Nous avons proposé une combinaison avec de la recherche, dans laquelle les simulations de Monte-Carlo servent à détecter les buts intéressants, et des recherches tactiques spécialisées déterminent l'état de ces buts et les meilleurs coups associés.

À Montana, l'utilisation de l'échantillonage itératif est une façon simple de rechercher de façon bien répartie dans tout l'espace. Cette méthode est proche de simulations de Monte-Carlo, avec la différence que les simulations ne servent pas à calculer une moyenne mais à trouver une solution. Le succès de cet algorithme peut s'expliquer par une particularité de ce domaine, à savoir qu'il ne nécessite pas forcément le recours à des heuristiques.

Au Morpion solitaire, des méthodes de Monte-Carlo ont été utilisées à la fois par nous et d'autres chercheurs, de façon indépendantes et avec de bons résultats. Ces méthodes permettent à la fois une bonne évaluation positionnelle, et une recherche bien répartie dans l'espace des états.

Nous envisageons des développements dans différentes parties de notre travail.

Un but qui parait atteignable est l'amélioration du record du morpion solitaire. La situation actuelle est très particulière, avec un record humain (170 coups) très supérieur au record par ordinateur (147 coups). Une combinaison de méthodes de Monte-Carlo et de recherche arborescente est prometteuse. Une possibilité serait de reprendre le même algorithme de recherche avec les transpositions incrémentales, mais de rendre la recherche sélective et d'évaluer les positions par des simulations de Monte-Carlo.

L'amélioration des méthodes de Monte-Carlo au Go est aussi particulièrement intéressante. Il est cependant claire que d'autres programmes de Monte-Carlo Go sont devenus meilleurs que ce que nous sommes capables de faire actuellement. La première chose à faire serait de les rattraper. Au delà, beaucoup

de pistes restent à explorer. Notre travail de combinaison avec des recherches tactiques est un bon début, mais très imparfait. Parmi toutes les recherches tactiques possibles, seules les recherches de connexions ont été vraiment utiles. Et encore, la recherche se fait de façon presque indépendante des simulations de Monte-Carlo : les simulations indiquent les buts importants, la recherche indique les meilleurs coups pour ces buts. On peut penser à des interactions plus fortes entre simulations et recherches tactiques, dans lesquelles les coups des simulations seraient dépendants des recherches tactiques. Ceci présente cependant le risque de biaiser les parties pseudo-aléatoires. Selon notre expérience, cela est assez difficile à régler.

En contraste avec ces deux directions de recherches précédentes, d'autres parties sont moins attractives à développer : les recherches de dépendances à Montana ou pour les connexions transitives au Go. Le problème de ces méthodes est le manque de cadre général solide. Il y a des idées communes, mais l'adaptation à un nouveau domaine n'a jamais été facile. Similairement, le travail de V. Allis sur l'algorithme *dependency based search* n'a pas, à notre connaissance, été appliqué de nouveau depuis. Le travail de J. Ramon [51], qui généralise nos travaux sur les connexions transitives, n'a pas encore été appliqué ailleurs, même si le cadre théorique est général. Les algorithmes produits sont, en général, dépendants de certaines propriétés des jeux. Un apport de notre recherche est, au moins, de contribuer à les identifier. On peut espérer, dans le futur, la découverte de méthodes plus générales... Un algorithme utile, assez généralement applicable, produit par notre recherche sur les dépendances, est l'algorithme des connexions transitives.

Annexe A

Annexe : Un programme de Lines of Action

Nous avons été amenés, pour diverses raisons, à écrire un programme de *Lines of Action*. La motivation première était de tester si la méthode des transpositions incrémentales était applicable sur des jeux à deux joueurs (nous avons expliqué pourquoi ce n'est en fait pas possible 2.2.3). Une autre motivation était la participation à un tournoi interne au laboratoire d'Intelligence Artificielle de Paris 8. Le programme écrit s'est appelé Bing (Bing Is Not GNU chess). Ce programme est un bon exemple de l'approche classique des jeux à deux joueurs avec une recherche en alpha-beta. Il est décrit dans [38].

Notre programme a été écrit en réutilisant une grande partie du code de GNU Chess. L'algorithme de recherche alpha-beta a été gardé sans grandes modifications. La fonction d'évaluation a été réécrite ; elle est basée sur six composantes qui sont données en entrée à un petit réseau de neurones, dont l'apprentissage a été fait par l'algorithme des différences temporelles.

Notre programme utilise donc des méthodes relativement classiques. Sa particularité est d'avoir été développé rapidement (environ un mois et demi), et d'avoir malgré cela atteint un niveau proche des meilleurs programmes de Lines of action. Il a d'abord été développé pour participer à un tournoi interne à Paris 8. Il a fini premier ex-aequo de ce tournoi, avec un revers inattendu dans une de ses deux parties contre un programme de T. Goossens. Il a ensuite fini second sur 3 participants aux 8^e *Computer Olympiads* de 2003 à Graz, avec un score de 5/8. Il a réussi la performance de gagner une partie contre le vainqueur, MIA de M. Winands. Aux 9^e *Computer Olympiads* de 2004 à Ramat-Gan, il a encore fini 2e sur 4 participants, avec un score de 8/12 ; derrière MIA mais devant YL de Y. Bjornsson, le vainqueur des 5^e, 6^e et 7^e *Computer Olympiads*.

A.1 Le jeu de Lines of action

Le jeu de Lines of action est relativement récent ; il s'est un peu développé depuis les années 90, à la fois chez des joueurs humains et chez les programmeurs. Actuellement, les programmes ont un net avantage sur les joueurs humains.

Lines of action se joue entre deux joueurs sur un damier 8x8. La position initiale est celle de la figure A.1. Les joueurs jouent à tour de rôle, en commençant par Noir. Un coup consiste à déplacer un pion de sa couleur sur une ligne horizontale, verticale ou diagonale, du nombre exact de pions total qui se trouvent sur cette ligne. Un pion peut sauter par dessus des pions amis, mais pas par dessus des pions ennemis ; il ne peut pas atterrir sur un pion ami mais peut atterrir sur un pion ennemi, et dans ce cas ce pion est capturé (figure A.2). Le but pour chaque joueur est de former une seule composante 8-connexe avec ses pions. La partie est nulle en cas de répétition de position, ou si les deux joueurs forment une seule composante en même temps.

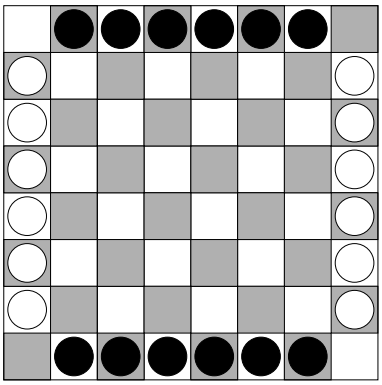


FIG. A.1 – Position initiale

A.2 Réutilisation de GNU Chess

Nous avons réutilisé une grande partie du code de la version 5.07 de GNU Chess. GNU Chess n'est pas parmi les meilleurs programmes d'échecs. Même parmi les programmes libres, il est largement battu par des programmes comme Crafty ou Fruit. Son attrait pour nous est la simplicité du code, qui est bien commenté. Il consiste en 13700 lignes de C, avec essentiellement un seul gros fichier d'entêtes. L'adaptation du code pour Lines of action n'a pas posé de difficultés, et a résulté en un gain de temps considérable.

L'interface de jeu a été gardée avec peu de modifications. L'essentiel du code de recherche a été gardé, c'est-à-dire : un alpha-beta avec negamax, approfondissement itératif, *principal variation search*, le coup nul, deux *killer moves*, l'*history heuristic* et une table de transposition [48]. Certaines optimisations

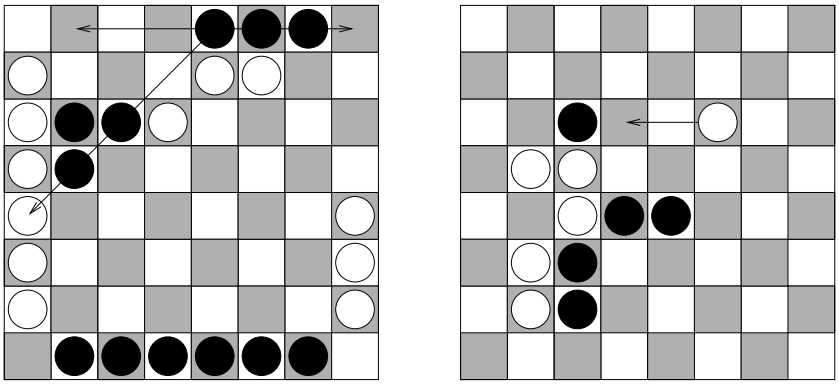


FIG. A.2 – Exemples de coups, et une position finale gagnée après le coup blanc

auraient demandé du travail pour être adaptées à Lines of action et ont été supprimées :

- la recherche de quiescence,
- le livre d'ouvertures,
- les extensions de la profondeur de recherche : dans GNU Chess, la recherche est étendue en cas de menaces sur les rois, de certaines captures et de menaces de promotions de pions.
- le *razoring* et le *futility pruning* qui consistent, lorsque l'évaluation est particulièrement mauvaise et que la profondeur de recherche restante est faible, à diminuer la profondeur de recherche ou à supprimer certains types de coups.

Au niveau des optimisations utilisées, notre programme est donc similaire à MIA [67], avec la recherche de quiescence et le livre d'ouvertures en moins.

Les plus grandes modifications ont porté sur la représentation du jeu, la gestion des coups et la fonction d'évaluation. La modification des règles du jeu a été facilitée par les similitudes entre les deux jeux, notamment la taille du damier (8x8). GNU Chess utilise une représentation du jeu basée sur les *bitboards*, qui sont en fait des entiers de 64 bits. Plusieurs bitboards sont utilisés pour représenter soit l'ensemble des pièces d'une couleur, soit seulement des pièces d'un certain type. La plupart des opérations sont accélérées par des manipulations booléennes sur les bits. Ceci est aidé par le précalcul de divers bitboards liés aux divers déplacements possibles. Ceci est aussi aidé par le maintien simultané de bitboards « tournés » de 45, 90 ou 315 degrés par rapport aux bitboards initiaux. Il est ainsi possible d'extraire rapidement des lignes, mais aussi des colonnes et même des diagonales de la position, et de représenter leur contenu comme des nombres binaires d'au plus 8 bits. Ces nombres peuvent, par exemple, être utilisés comme indices dans des tableaux précalculés, pour obtenir les mouvements possibles sur cette ligne, colonne ou diagonale. Les lignes peuvent être extraites trivialement à partir du bitboard initial, les colonnes à partir d'un

bitboard changeant le rôle des lignes et des colonnes, et les diagonales à partir d'un des deux bitboards correspondant aux permutations des bits de la figure A.3.

0	2	5	9	14	20	27	35
1	4	8	13	19	26	34	42
3	7	12	18	25	33	41	48
6	11	17	24	32	40	47	53
10	16	23	31	39	46	52	57
15	22	30	38	45	51	56	60
21	29	37	44	50	55	59	62
28	36	43	49	54	58	61	63

28	36	43	49	54	58	61	63
21	29	37	44	50	55	59	62
15	22	30	38	45	51	56	60
10	16	23	31	39	46	52	57
6	11	17	24	32	40	47	53
3	7	12	18	25	33	41	48
1	4	8	13	19	26	34	42
0	2	5	9	14	20	27	35

FIG. A.3 – permutations des cases pour les bitboards tournés de 45 et 315 degrés

Tout ceci est facile à adapter pour les règles de Lines of action. Nous utilisons dans BING deux bitboards pour les pions de chacune des couleurs. Par exemple, la position initiale est représentée par les deux bitboards 0x7E0000000000007E pour blanc et 0x0081818181818100 pour noir. Nous utilisons aussi trois bitboards tournés pour chaque couleur. Dans notre cas, l'intérêt des bitboards tournés et d'éviter, quand on cherche les mouvements possibles d'un pion dans une direction, de devoir exécuter une boucle coûteuse sur les cases dans cette direction.

Notre programme recherche environ 100.000 nœuds par seconde sur un Pentium 4 3.2 GHz, contre environ 300.000 pour GNU Chess. Cette différence est due à la lenteur relative de notre fonction d'évaluation. La fonction d'évaluation de GNU Chess est pourtant plus complexe, avec beaucoup plus de cas particuliers traités, mais notre fonction d'évaluation évalue des composantes assez coûteuses (il serait sans doute possible d'accélérer le calcul de ces composantes en le rendant incrémental, mais ce serait nuisible à la simplicité du code). De plus, GNU Chess utilise des coupes d'évaluation paresseuse (*lazy evaluation cuts*), permettant de se contenter d'une évaluation rapide si elle est suffisante.

A.3 Fonction d'évaluation

Nous définissons les composantes de notre fonction d'évaluation, le réseau de neurones qui fournit l'évaluation finale et les méthodes d'apprentissage.

A.3.1 Les différentes composantes

La fonction d'évaluation repose sur plusieurs composantes qui sont ensuite données en entrée d'un réseau de neurones. Nous nous sommes inspirés des composantes décrites dans [68]. Nous commençons par calculer le centre de gravité des pions des ensembles de pions de chaque couleur. Nous définissons l'*excentrisme* d'un pion comme la distance au centre de gravité de la couleur. Quand nous considérons une distance, il s'agit toujours de la distance $d_\infty((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|)$. En effet. il s'agit du nombre minimal de déplacement nécessaires en 8-connexité pour se déplacer d'une case à une autre.

Pour chacun des joueurs, les composantes sont :

1. Le nombre de pions de la couleur.
2. La distance moyenne au centre de gravité des pions de la couleur.
3. La distance entre le centre de gravité de la couleur et le centre du damier.
4. Une évaluation du nombre de composantes 8-connexes de la couleur et de leurs excentrismes (l'excentrisme d'une composante connexe est défini comme l'excentrisme minimal des pions qui la composent). Il est évidemment bon d'avoir peu de composantes connexes ; cependant, une composante connexe excentrée, composée de n pions, est presque aussi mauvaise que n pions séparés, puisque ces pions devront probablement être déplacés séparément pour les connecter au reste des pions. Ces idées amènent à la formule suivante : une composante connexe de n pions compte pour 1 si elle est peu excentrée (distance inférieure à 1,5), pour n si elle est très excentrée (distance supérieure à 2,5), avec une transition affine pour les distances intermédiaires.
5. Un évaluation des connexions entre pions adjacents de la couleur, une connexion comptant plus si elle est proche du centre de gravité de la couleur. Plus précisément, nous donnons à chaque connexion la valeur $\max(2, 5 - d, 0)$, où d est la distance au centre de gravité de la couleur, et nous calculons la somme sur toutes les paires de pions adjacents de la couleur. Cette composante donne l'avantage à des formes ramassées, qui sont connectées de nombreuses façons différentes et donc plus « solides ».
6. Une évaluation du nombre de pions de la couleur « bloqués » par des pions adverses. Nous considérons qu'un pion est bloqué si il est à distance supérieure à 2 du centre de gravité, et si les directions qui lui permettraient de se rapprocher du centre de gravité sont bloquées par des pions adverses adjacents. L'évaluation précise est un peu compliquée ; le blocage d'un pion dépend de la distance exacte au centre de gravité et de la meilleure direction libre qu'il possède, dans le sens où elle le rapproche ou au moins ne l'éloigne pas trop du centre de gravité. La formule précise est :

$$\max(d - 2, 0) \times \max(0.5 + \min(\min_{\text{dir. libre}}(d_2 - d), 1), 0),$$

où d est la distance originale au centre de gravité, d_2 la distance au centre de gravité d'une case adjacente libre, et où on considère que le minimum

d'un ensemble vide est infini. Ainsi, la partie droite de l'expression varie entre 1,5 pour des pions ne pouvant même pas reculer et 0 pour des pions dont une des cases adjacentes libres lui permet d'avancer d'au moins 0,5.

Les composantes 4, 5 et 6 ressemblent à des composantes utilisées dans d'autres programmes tels que MIA [68], mais les détails sont un peu originaux. Nous avons fait en sorte, pour chacune des composantes, de ne pas introduire de variations brusques mais d'utiliser des fonctions continues. Il y a des paramètres un peu arbitraires qu'il aurait été fastidieux de tester rigoureusement. Une composante qui aurait pu être rajoutée, et qui est présente dans YL et MIA, est le nombre de coups possibles pour chaque joueur.

Il n'est pas évident de décider si la composante sur le nombre de pions doit intervenir positivement ou négativement (il est plus facile de se connecter avec peu de pions, mais plus facile de gêner l'adversaire avec beaucoup de pions). De plus, les composantes 4 et 5 évaluent des choses un peu similaires, et sont fortement corrélées au nombre de pions. Nous avions d'abord utilisé une combinaison linéaire de ces composantes avec des poids qui nous ont paru raisonnables, mais nous l'avons ensuite remplacée par un réseau de neurones, avec apprentissage des poids par différences temporelles. L'amélioration du niveau a été très claire.

A.3.2 Calcul du nombre d'Euler avec les quads

Le *nombre d'Euler* d'une partie finie d'un plan discrétisé est la différence entre le nombre de composantes connexes (dans notre cas, 8-connexes) et le nombre de trous. Ce nombre peut être calculé efficacement, et surtout incrémentalement, avec les *quads* [35].

En pratique, ce nombre est une bonne estimation du nombre de composantes connexes, puisque les trous dans les composantes dont rares. Le nombre de composantes brut n'est pas utilisé dans notre fonction d'évaluation (puisque la composante 4 fait intervenir l'excentrisme des composantes), mais le nombre d'Euler reste utile pour les tests de victoire. En effet, si le nombre d'Euler est strictement supérieur à 1, il y a certainement plus d'une composante connexe, et le calcul exact n'est pas nécessaire.

Le nombre d'Euler se calcule par une somme sur chacun des carrés 2x2 du plateau (y compris sur les bords en dépassant d'une case), avec les poids de la figure A.4 (les autres carrés qui se déduisent par rotation ont les mêmes poids). Ces poids correspondent au nombre de quarts de tours que font les frontières des composantes connexes. Diviser la somme par quatre donne donc le nombre d'Euler. L'intérêt de ce calcul est la possibilité de calcul incrémental quand un coup est joué.

A.3.3 Apprentissage

Nous présentons notre méthode d'apprentissage par différences temporelles, pour l'apprentissage des poids d'un petit réseau de neurones. Nous comparons avec des programmes similaires dans d'autres jeux. Un succès notable de l'algorithme des différences temporelles est le programme de Backgammon *TD-*

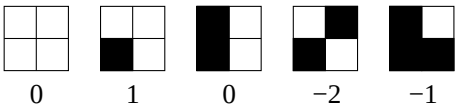


FIG. A.4 – Poids des quads pour le calcul du nombre de composantes connexes

Gammon [62] ; cependant le domaine est stochastique et les résultats doivent donc être comparés avec précaution. Une tentative d'appliquer la méthode dans des jeux reposant plus sur la recherche, comme les échecs, est le programme *KnightCap* [4]. Le programme de Lines of action MIA a aussi utilisé des différences temporelles pour l'apprentissage des poids d'une combinaison linéaire [66]. Un autre exemple d'application est *Neurogo* [28].

Le réseau de neurones

Nous utilisons un petit réseau de neurones, prenant en entrées les composantes ci-dessus, pour calculer l'évaluation d'une position. La taille de ce réseau est limitée par des contraintes de temps de calcul ; et de toute façon l'intérêt d'utiliser un gros réseau de neurones n'est pas clair, puisqu'il est plutôt habituel d'utiliser de simples combinaisons linéaires dans ce genre de programmes en alpha-beta (comme le font par exemple MIA ou KnightCap). Nous avons choisi un réseau avec une couche cachée, mais seulement 2 neurones dans cette couche. Nous aurions aussi obtenu de bons résultats avec une simple combinaison linéaire ; en fait, des essais avec un seul neurone dans la couche cachée (donc essentiellement une combinaison linéaire) ont donné des résultats très proches.

Les entrées de ce réseau de neurones sont au nombre de 14 : les 6 composantes pour chacun des joueurs, une entrée indiquant le trait (± 1) et une entrée de biais fixée à 1.

Les poids du réseau de neurones sont modifiés par l'algorithme classique de rétro-propagation. Nous avons également implémenté l'algorithme RPROP [52] qui est une amélioration, mais ce n'était pas forcément nécessaire puisque l'apprentissage est suffisamment rapide avec l'algorithme classique (quelques minutes).

Les différences temporelles

L'apprentissage s'est fait par l'algorithme des différences temporelles TD(λ) [61]. Étant donnée une partie, cet algorithme fait apprendre au réseau de neurones, dans chacune des positions de la partie, l'évaluation telle qu'elle est fournie par le programme lui-même mais dans les positions ultérieures. La méthode dépend d'un paramètre λ. La valeur $\lambda = 0$ signifie que le programme apprend l'évaluation de la position immédiatement suivante, alors que la valeur $\lambda = 1$ signifie que c'est la valeur finale de la partie qui est apprise, et toutes les valeurs intermédiaires sont possibles. Nous avons choisi de fixer $\lambda = 0,9$, ce qui

est une valeur plutôt élevée. Le programme TD-Gammon a utilisé $\lambda = 0.7$ et $\lambda = 0$; Neurogo a utilisé $\lambda = 0$; MIA a utilisé $\lambda = 0.8$. G. Tesauro déconseille des valeurs de λ trop proches de 1 [62], mais le jeu Lines of action étant moins stochastique que le Backgammon, nous pensons qu'une valeur de λ plus élevée est justifiée.

L'apprentissage de notre programme s'est fait à partir d'une base statique de quelques centaines de parties. Les différentes bases que nous avons utilisées contenaient entre 600 et 4000 parties. Les parties sont jouées en boucle jusqu'à stabilisation des poids du réseau de neurones. En cela, notre approche diffère de celle, plus habituelle, consistant à faire apprendre au programme des parties jouées dynamiquement par la version courante du programme contre lui-même (comme TD-Gammon, Neurogo ou MIA) ou bien entre le programme et des joueurs humains (comme KnightCap, 308 parties jouées sur le serveur Internet FICS). Un avantage de notre méthode est la plus grande rapidité d'apprentissage, puisque moins de parties doivent être construites. Les poids du réseau de neurones de BING se stabilisent après environ 500 lectures de la base de parties.

Notre application de l'algorithme des différences temporelles a une différence avec l'algorithme original : les coups qui sont appris sont produits par une recherche en alpha-beta, et non par une simple recherche à profondeur 1. Avec la terminologie de [4], l'algorithme que nous utilisons s'appelle en fait TD-DIRECTED(λ). Les résultats sur KnightCap indiquent que cet algorithme, dans le domaine des échecs, a une efficacité supérieure à TD(λ) et inférieure à l'algorithme appelé TDLEAF(λ).

Construction des bases de parties

La principale difficulté provient de la construction de la base de parties. Nous avons construit des bases qui donnent un bon niveau au programme, et d'autres beaucoup moins bonnes. Il n'est pas encore clair pour nous quelles sont les bonnes façons de construire une base de parties pour l'apprentissage. Toutes les parties de nos bases ont été jouées par BING contre lui-même, en fixant le temps total d'une partie à seulement quelques secondes.

Nous pensons qu'une bonne base de parties doit offrir une certaine diversité, c'est pourquoi nous avons mélangé des parties jouées à différents stades du développement de BING, en commençant par la version avec une combinaison linéaire réglée à la main, et en passant par des versions avec des composantes différentes en entrée du réseau de neurones. Différentes bases de parties ont produit, après apprentissage, des programmes avec des styles de jeu différents. Par exemple, certaines versions de BING accordent plus ou moins d'importance au nombre de pions. Ceci n'empêche pas des niveaux de jeu similaires, comme il a été vérifié expérimentalement.

Les expériences faites avec une base de parties tirées uniquement de la dernière version du programme ont abouti à des résultats souvent nettement inférieurs. Ceci est en accord avec les résultats sur KnightCap [4], indiquant que des parties d'un seul programme contre lui-même introduisent trop peu de diversité, et que le niveau après apprentissage s'en ressent.

Bibliographie

[1] S.G. Akl and M.M. Newborn. The principal continuation and the killer heuristic. In *ACM Annual Conference*, pages 466–473, 1977.

[2] L. Victor Allis. Searching for solutions in games an artificial intelligence. Phd thesis, Vrije Universitat Amsterdam, Maastricht, 1994.

[3] Vadim V. Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134(1-2) :101–120, 2002.

[4] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to play chess using temporal-differences. *Machine Learning*, 40(3) :243–263, 2000.

[5] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways*, volume 1-2. Academic Press, 1982.

[6] Hans Berliner and Gordon Goetsch. A study of search methods : The effect of constraint satisfaction and adventurousness. In *IJCAI*, pages 1079–1082, 1985.

[7] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2) :210–240, 2002.

[8] Adi Botea, Martin Muller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. In *Proceedings of Computers and Games, Edmonton, Canada*, 2002.

[9] Bruno Bouzy. *Modélisation cognitive du joueur de Go*. PhD thesis, Université Paris 6, 1995.

[10] Bruno Bouzy. Associating domain-dependent knowledge and monte-carlo approaches within a go program. In *Joint Conference on Information Sciences, Heuristic Search and Computer Game Playing Session*, pages 505–508, 2003.

[11] Bruno Bouzy. Associating shallow and selective global tree search with monte-carlo for 9x9 go. In *Computer and Games*, 2004.

[12] Bruno Bouzy. Move pruning techniques for monte-carlo go. In *Advances in Computer Games 11*, 2005.

[13] Bruno Bouzy and Tristan Cazenave. Computer go : an ai-oriented survey. *Artificial Intelligence*, 132 :39–103, octobre 2001.

[14] Bruno Bouzy and Bernard Helmstetter. Monte-carlo go developments. In Kluwer, editor, *Advances in Computer Games (ACG10)*, pages 159–174, 2003.

[15] Richard Bozulich. *The Go Players's Almanac*. Kiseido Publishing Company, 2001.

[16] Dennis M. Breuker. *Memory versus Search in Games*. PhD thesis, Maastricht University, 1998.

[17] B. Bruegmann. Monte-carlo go. ftp ://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z, Max-Planck-Institute of Physics, 1993.

[18] M. Campbell, A. J. Hoane, and F. h Hsu. Deep blue. *Artificial Intelligence*, 134(1-2), 2002.

[19] Tristan Cazenave. *Systeme d'Apprentissage par Auto-Observation. Application au Jeu de Go*. PhD thesis, Université Pierre et Marie Curie, Paris 6, 1996.

[20] Tristan Cazenave. A generalized threats search algorithm. In *Proceedings of Computers and Games, Edmonton, Alberta*, 2002.

[21] Tristan Cazenave. Gradual abstract proof search. *ICGA journal*, Vol. 25 (1) :3–15, 2002.

[22] Tristan Cazenave. Generalized widening. In *ECAI 2004*, pages 156–160, August 2004.

[23] Tristan Cazenave and Bernard Helmstetter. Combining tactical search and monte-carlo in the game of go. In *IEEE CIG 2005*, pages 171–175, 2005.

[24] Tristan Cazenave and Bernard Helmstetter. Search for transitive connections. *Information Sciences*, 175(4) :284–295, November 2005.

[25] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, Torino, Italy, May 2006.

[26] Pierre Crépeau. *Le grand livre des patiences*. Les éditions de l'homme, 1999.

[27] Erik D. Demaine, Martin L. Demaine, Arthur Langerman, and Stefan Langerman. Morpion solitaire. In *Proceedings of the 3rd International Conference on Fun with Algorithms*, pages 53–64, 2004.

[28] Markus Enzenberger. Evaluation in go by a neural network using soft segmentation. In Kluwer, editor, *Advances in Computer Games (ACG10)*, pages 97–108, 2003.

[29] Richard E. Fikes and Nils J. Nilsson. Strips : a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2 :189–208, 1971.

[30] David Fotland. Arimaa... a completer. In *Computer and Games*, 2004.

[31] Ralph Gasser. Solving nine men's morris. *Computational Intelligence*, 12 :24–41, 1996.

[32] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. Technical report, INRIA, 2006.

[33] M. Ginsberg. Gib : Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 2001.

[34] M. L. Ginsberg. GIB : Steps toward an expert-level bridge-playing program. In *IJCAI-99*, pages 584–589, Stockholm, Sweden, 1999.

[35] S. B. Gray. Local properties of binary images in two dimensions. In *IEEE Transactions on Computers*, volume C-20, pages 551–561, 1971.

[36] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of IJCAI ; Vol. 1*, pages 607–615, 1995.

[37] Bernard Helmstetter and Tristan Cazenave. Incremental transpositions. In *Computer and Games*. Springer Verlag, 2004.

[38] Bernard Helmstetter and Tristan Cazenave. Architecture d'un programme de lines of action. *Intelligence Artificielle et Jeux*, pages 107–116, 2006.

[39] Jorg Hoffmann. Local search topology in planning benchmarks : An empirical analysis. In *Proceedings of IJCAI*, pages 453–458, 2001.

[40] Hugues Juillé. Incremental co-evolution of organisms : A new approach for optimization and discovery of strategies. In *European Conference on Artificial Life*, pages 246–260, 1995.

[41] Hugues Juillé. *Methods for Statistical Inference : Extending the Evolutionary Computation Paradigm*. PhD thesis, Brandeis University, 1999.

[42] Andreas Junghanns. *Pushing the Limits : New Developments in Single-Agent Search*. PhD thesis, University of Alberta, Department of Computing Science, 1999.

[43] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220 :671–681, 1983.

[44] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML 2006*, 2006.

[45] P. Langley. Systematic and nonsystematic search strategies. In *Artificial Intelligence Planning Systems : Proceedings of the First International Conference*, 1992.

[46] Jun S. Liu. *Monte-Carlo Strategies in Scientific Computing*. Springer-Verlag, 2001.

[47] M. Lustrek, M. Gams, and I. Bratko. A program for playing tarok. *ICGA Journal*, 26(3) :190–197, 2003.

[48] T. A. Marsland. A review of game-tree pruning. *ICGA journal*, 9(1) :3–19, 1986.

[49] Jean-Charles Meyrignac. Morpion solitaire progress. http ://euler.free.fr/morpion.htm.

[50] Martin Müller. Review : Computer go 1984 - 2000. In *Lecture Notes in Computer Science*, pages 426–435. Springer Verlag, 2001.

[51] Jan Ramon and Tom Croonenborghs. Searching for compound goals using relevancy zones in the game of go. In *Computers and Games*, volume 3846 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, 2006. URL = http ://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl ?id=41320.

[52] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning : the rprop algorithm. In *IEEE International Conference on Neural Networks*, volume 1, pages 586–591, 1993.

[53] J. Romein and H. Bal. Solving the game of awari using parallel retrograde analysis, 2003.

[54] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Pearson US imports, 1995.

[55] A. Sadikov, I. Bratko, and I. Kononenko. Search versus knowledge : an empirical study of minimax on krk. In *Advances in Computer Games 10*, pages 33–44, 2003.

[56] Jonathan Schaeffer. The history heuristic and the performance of alpha-beta enhancements. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 11, pages 1203–1212, 1989.

[57] Jonathan Schaeffer, Yngvi Bjornsson, Neil Burch, Robert Lake, Paul Lu, and Steve Sutphen. Building the checkers 10-piece endgame databases. In *Advances in Computer Games 10*, pages 193–210. Kluwer Academic Publishers, 2003.

[58] B. Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1-2) :241–275, 2002.

[59] Toshio Shintani. Consideration about state transition in Superpuzz. In *Information Processing Society of Japan Meeting*, pages 41–48, Shonan, Japan, 2000. (en japonais).

[60] Toshio Shintani and Hiroyuki Miyake. Consideration on search for the solutions of superpuzz. In *Game Programming Workshop in Japan 2002*, 2002. (en japonais).

[61] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1) :9–44, August 1988.

[62] G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2) :181–199, 2002.

[63] J. Tromp and G. Farnebäck. Combinatorics of go. In *Proceedings of 5th International Conference on Computer and Games*, Torino, Italy, May 2006.

[64] E.C.D. van der Werf, H.J. van den Herik, and J.W.H.M. Uiterwijk. Solving go on small boards. *ICGA Journal*, 26(2) :92–107, 2003.

[65] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[66] M. H. M. Winands, L. Kocsis, J. W. H. M. Uiterwijk, and H. J. van den Herik. Learning in lines of action. In H. Blockeel and M. Denecker, editors,

Proceedings of the Fourteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC), pages 371–378, 2002.

[67] Marc Winands. Analysis and implementation of lines of action. Master's thesis, Faculty of General Sciences of the Universiteit Maastricht, 2000.

[68] M.H.M. Winands, H.J. van den Herik, and J.W.H.M. Uiterwijk. An evaluation function for lines of action. In H. Iida H. van den Herik and E. Heinz, editors, *Advances in Computer Games 10*, pages 249–260. Kluwer Academic Publishers/Boston, 2003.