
Université Paris 8 de Vincennes à Saint-Denis
Département d'Informatique
UFR MITSIC
2, rue de la liberté
93526 Saint Denis CEDEX 02
Tél : +33(0)1 49 40 67 89

Parallélisation d'un *General Game Player* sur une architecture MPPA

ALINE HUFSCHMITT
<alinehuf@gmail.com>

sous la direction de JEAN MÉHAT

Mémoire de Master 2
Informatique et Sciences Humaines

26 Septembre 2014

Résumé

Dans ce mémoire, nous présentons une étude pour la parallélisation d'un *General Game Player*, *LeJoueur* de Jean-Noël Vittaut, sur une architecture MPPA.

Dans le cadre des compétitions de GGP organisées par Stanford, les joueurs programmés reçoivent les règles de jeux originaux sous la forme de descriptions en *Game Definition Language* (GDL). Ces joueurs utilisent des algorithmes de parcours d'arbre comme le *Upper Confidence Bounds applied to Trees* (UCT) pour déterminer les coups les plus prometteurs pouvant mener à une victoire. Ces algorithmes permettent une évaluation d'autant plus juste des coups qu'un grand nombre de simulations de parties (*playouts*) est effectué. Plusieurs techniques de parallélisation des algorithmes UCT ont été décrites dans la littérature mais aucune ne prend en compte l'utilisation de *propnets*, une nouvelle technique proposée par le [Stanford Logic Group \[2013\]](#) pour accélérer le calcul des *playouts*.

Dans le cadre d'un partenariat entre la société Kalray et le laboratoire LIASD de Paris 8, une machine dotée d'une carte MPPA-256 regroupant 256 processeurs en clusters de 16 a été mise à notre disposition. Notre but est de tester cette nouvelle architecture et de déterminer quelles options elle peut offrir pour la parallélisation de *LeJoueur* de Jean-Noël Vittaut, un *General Game Player* utilisant à la fois un algorithme UCT et la nouvelle technique des *propnets*.

Dans un premier temps nous réalisons une *parallélisation aux feuilles* de l'algorithme UCT en utilisant des communications synchrones. Nous envisageons plusieurs options pour le calcul des *playouts* : calcul distribué entre les différents processeurs élémentaires d'un cluster ou calcul d'un seul *playout* par processeur élémentaire. Nous déterminons la meilleure manière de gérer des threads et examinons le comportement des temps de communication en fonction de la taille des données transmises.

Lors de cette première phase d'expérimentation, nous nous heurtons à un problème de limitation de la mémoire au niveau des clusters : pour les jeux intéressants, les données des *propnets* devant être transmises aux clusters ne tiennent pas en mémoire. Ces *propnets* sont des circuits logiques décrivant la mécanique des jeux. Nous proposons d'utiliser une méthode d'optimisation stochastique pour réduire la taille de ces circuits.

Dans un second temps nous utilisons donc l'algorithme du recuit simulé pour ré-

duire les circuits. Nous présentons diverses transformations élémentaires que nous avons utilisées pour simplifier les circuits et montrons leur influence sur la convergence de l'algorithme. Nous présentons quelques développements futurs possibles de notre programme de simplification de circuits logiques mais expliquons que dans le cas particulier de *LeJoueur* une autre solution est trouvée en amont.

Nous concluons que la limitation de la mémoire à 2Mo au niveau des clusters sur la carte MPPA, fait de la *parallélisation aux feuilles* la seule parallélisation possible sur la carte MPPA. Cette mémoire limitée représente une contrainte majeure de cette architecture. Nous concluons également qu'il est plus efficace en terme de temps de communication et synchronisation d'effectuer le calcul d'un *playout* complet par processeur élémentaire et de relancer les threads à chaque demande de calcul. Nous relevons la présence de buggs dans les éléments logiciels de la carte MPPA et indiquons que l'amélioration du pack logiciel de la carte sera nécessaire pour permettre d'en tirer tout le potentiel. Notre programme devra être modifié pour utiliser des communications asynchrones. Cette implémentation de communications asynchrones sur le MPPA devra être évaluée pour déterminer si d'autres limitations propres à cette architecture ne sont pas mises en évidence et si une parallélisation efficace de *LeJoueur* peut être obtenue.

Remerciements

Je remercie Monsieur Jean Méhat pour ses précieux conseils et sa disponibilité sans réserve.

Je remercie Monsieur Jean-Noël Vittaut pour avoir mis à ma disposition le code source de son programme *LeJoueur* et m'avoir apporté explications et aide technique.

Je remercie Nicolas Jouandeaupour ses conseils et son aide concernant la prise en main de l'architecture MPPA.

Je remercie Jean-Philippe Hourcade, mon mari, pour ses encouragements et son soutien quotidien.

Je remercie mes parents pour leur soutien logistique sans lequel je n'aurais pas pu reprendre mes études.

Table des matières

Introduction	1
1 Le General Game Playing (GGP)	5
1.1 Présentation du GGP	5
1.2 Le Game Definition Language (GDL)	7
1.3 Compétitions de GGP	8
1.4 Exploration d'arbre	11
1.4.1 Famille Mini-Max	12
1.4.2 Techniques de Monte Carlo	14
1.4.3 Améliorations de MCTS	16
1.5 Parallélisation de MCTS	18
1.5.1 Parallélisation au niveau de l'arbre	19
1.5.2 Parallélisation à la racine	20
1.5.3 Parallélisation au niveau des feuilles	22
1.5.4 Parallélisation avec arbre distribué	23
1.5.5 Parallélisation de Nested Monte Carlo	24
1.6 Les <i>Propositional Networks</i> (<i>propnet</i>)	25
1.6.1 Le <i>tabling</i> et le <i>grounding</i>	26
1.6.2 Principe du <i>propnet</i>	27
1.6.3 Implémentation d'un <i>propnet</i> dans <i>LeJoueur</i> (version 02/2014)	28
1.7 Conclusion	32
2 Parallélisation de <i>LeJoueur</i>	35
2.1 Architecture utilisée (MPPA-256)	35
2.1.1 Le Multi Purpose Processor Array 256	36
2.1.2 Structure d'un programme pour le MPPA	38
2.1.3 Les communications	40
2.1.4 Analyse critique du MPPA	43
2.2 Hypothèses de parallélisation	44
2.3 Choix de communications synchrones	45
2.4 Parallélisation aux feuilles	46
2.4.1 Les <i>Reasoner</i> de <i>LeJoueur</i>	46
2.4.2 Outils pour tester les <i>Reasoner</i>	47
2.4.3 Structure de notre programme	49
2.4.4 Communications et transmission d'ordres	50
2.4.5 Codage des données à transmettre	51

2.5	Partage ou multiplication des <i>playouts</i>	53
2.6	Threads gardés actifs ou relancés	55
2.7	Temps de communication et de calcul	58
2.8	Évaluation du temps en fonction de la taille des données	62
2.9	Problème de la taille des circuits	66
3	Le recuit simulé	71
3.1	Le recuit thermique	72
3.2	Le recuit simulé	73
3.3	Programme de recuit	74
3.3.1	Température initiale	75
3.3.2	Longueur des paliers	76
3.3.3	Schéma de décroissance de la température	78
3.3.4	Le critère d'arrêt	80
4	Utilisation du recuit pour la simplification des circuits	83
4.1	Transformation des circuits en graphe	83
4.2	Garantir l'unicité des nœuds	84
4.3	Évaluation d'une solution	87
4.3.1	Problème d'ambiguïté	88
4.3.2	Propagation des effets	90
4.3.3	Évaluation finale des circuits	92
4.4	Choix du programme de recuit	93
4.5	Choix des transformations élémentaires	94
4.5.1	Transformations élémentaires principales	95
4.5.2	Non uniformité des transformations élémentaires	96
4.5.3	Transformations élémentaires optionnelles	100
4.5.4	Problème de convergence de l'algorithme	101
4.5.5	Guider la découverte des sous-expressions communes	104
4.5.6	Traitement des négations	106
4.6	Résultats	111
4.7	Futurs développements	115
4.8	Solution en amont	117
	Conclusion	118
	Annexe : règles des jeux évoqués	123
	Bibliographie	129

Introduction

Nous présentons dans cette introduction les motivations qui nous ont conduits à réaliser la parallélisation d'un *General Game Player* sur une architecture MPPA. Nous commençons par présenter le domaine du *General Game Playing*, les techniques les plus performantes utilisées par les joueurs à ce jour et le défi posé par la parallélisation d'un joueur utilisant ces techniques. Dans le cadre d'un partenariat avec la société Kalray, une machine dotée d'une carte MPPA-256 a été mise à notre disposition. Nous proposons de tester cette architecture matérielle dont les outils logiciels sont encore en développement en réalisant dessus la parallélisation de *LeJoueur* de Jean-Noël Vittaut.

Le *General Game Playing*

Le *General Game Playing* est une des branches de l'intelligence artificielle ayant pour but la réalisation de programmes polyvalents capables de jouer à n'importe quel jeu. Ces programmes doivent être capables d'analyser les règles d'un jeu inconnu, d'en comprendre les objectifs, de découvrir les enchaînements de coups menant à la victoire et d'y jouer avec expertise.

Pour stimuler la recherche dans ce domaine, des compétitions de *General Game Playing* sont organisées [Genesereth et al., 2005] pour confronter les programmes développés par différentes équipes. Les organisateurs utilisent le *Game Definition Language* (GDL) pour décrire la large gamme de jeux étudiés et transmettre les règles aux concurrents. Ce langage de type Prolog, fondé sur la logique du premier ordre, présente les règles du jeu sous forme d'une liste de faits logiques et de théorèmes sur ces faits. Les joueurs programmés analysent ces faits pour établir la situation du jeu, connaître les différents coups légaux et les résultats en fin de partie.

Pour découvrir quel coup choisir à chaque étape du jeu, différentes techniques de parcours d'arbre ont été développées pour explorer chaque position du jeu et les différentes branches de possibilités qui dépendent du coup choisi. La racine de l'arbre représente l'état initial du jeu et l'objectif est d'atteindre la feuille correspondant à la fin de partie dont le score est le plus élevé possible pour le joueur.

Techniques de Monte Carlo et *Propnets*

Depuis quelques années, les techniques de *Monte Carlo Tree Search* (MCTS) [Browne et al., 2012] ont remplacé les algorithmes de type *Minimax* avec coupures *Alpha Beta* pour l'exploration d'arbre. Ces nouvelles techniques de MCTS améliorent considérablement l'efficacité des explorations d'arbre en proposant un équilibre entre *exploration*, recherche de nouvelles branches plus prometteuses présentant l'inconvénient d'être coûteuse en temps, et *exploitation*, examen plus approfondi d'une zone de l'arbre qui semble intéressante, i.e. susceptible d'amener à un meilleur score.

Du nombre de simulations de parties (*playouts*) effectuées dépend la qualité de l'évaluation d'un nœud dans l'arbre du jeu. La simulation de ces parties est assez lente si l'on utilise un interprète Prolog pour analyser les règles GDL. Récemment, un nouveau modèle nommé *propnet* été proposé par l'équipe de Stanford [Stanford Logic Group, 2013] qui permet d'interpréter beaucoup plus rapidement les règles du jeu. L'usage d'un *propnet* permet d'augmenter considérablement le nombre de simulations effectuées dans un temps donné et donc la qualité des évaluations.

Différentes techniques de parallélisation des algorithmes MCTS ont été explorées jusqu'à ce jour, dont certaines sont appliquées au *General Game Playing* [Méhat et Cazenave, 2011a; Finnsson, 2012b] mais aucune à notre connaissance ne réalise cette parallélisation sur un joueur utilisant un *propnet*. Les temps de calcul étant plus rapides avec un *propnet*, il est à prévoir que les temps de communication et de synchronisation seront proportionnellement plus importants, rendant la réalisation d'une parallélisation performante plus délicate.

Parallélisation sur une architecture MPPA

Dans le cadre d'un partenariat entre le laboratoire LIASD de l'université Paris 8 et la société Kalray, nous avons eu accès à une machine dotée d'une carte MPPA-256 proposant 256 processeurs de calcul regroupés en 16 clusters. Les outils logiciels permettant d'exploiter cette architecture sont en cours de développement, toutes les fonctionnalités ne sont pas encore exploitées ou les éléments logiciels ne sont pas totalement opérationnels. Nous proposons donc de tester cette architecture tout en réalisant la parallélisation du *General Game Player* de Jean-Noël Vittaut qui utilise à la fois les techniques MCTS et les *propnets*.

Notre but est d'explorer les capacités de la carte MPPA et les contraintes techniques qu'elle impose tout en examinant les possibilités qu'elle offre pour la parallélisation de *LeJoueur* de Jean-Noël Vittaut¹.

1. Le joueur de Jean-Noël Vittaut a fini 2^{ème} à la compétition de *General Game Playing* organisée par Stanford en 2014, cependant nous avons utilisé la version de *LeJoueur* de février 2014 qui était moins aboutie que celle présentée à la compétition en juillet.

Plan de lecture

Dans le premier chapitre nous présentons le domaine du *General Game Playing*, les techniques de Monte Carlo utilisées pour l'exploration des arbres des jeux ainsi que les différentes parallélisations possibles de ces algorithmes. Nous présentons également le principe des *propnets* utilisés pour interpréter les règles des jeux et montrons comment cette technique est implémentée dans *LeJoueur* de Jean-Noël Vittaut.

Dans le chapitre 2 nous présentons l'architecture MPPA, sa structure et ses limitations. A la lumière de ces contraintes techniques nous expliquons notre choix de parallélisation et comment nous avons modifié *LeJoueur* pour l'implémenter. Enfin nous présentons diverses expérimentations et leurs résultats permettant de choisir les options d'implémentation les plus performantes pour le calcul des *playouts* et la gestion des threads, et d'estimer l'évolution des temps de communication. Nous achevons ce chapitre par la présentation d'un problème majeur qui nous a bloqués dans le développement de notre application : les données représentant les *propnets* de certains jeux, sont trop volumineuses pour tenir en mémoire dans les clusters de la carte MPPA.

Nous consacrons le chapitre 3 à la présentation de la technique du recuit simulé, technique d'optimisation que nous souhaitons utiliser pour simplifier les circuits logiques constituant les *propnets* et réduire ainsi le volume des données à transmettre à la carte MPPA.

Dans le chapitre 4 nous présentons notre implémentation du recuit simulé et consacrons une grande partie du chapitre à la présentation des différentes transformations élémentaires que nous avons utilisées pour simplifier les circuits et leur influence sur la convergence de l'algorithme. Nous présentons quelques développements futurs possibles de notre programme de simplification de circuits logiques mais expliquons que dans le cas particulier de *LeJoueur* une autre solution est trouvée en amont.

Dans la conclusion nous faisons le bilan de nos expérimentations et présentons une amélioration possible de notre parallélisation en utilisant des communications asynchrones. Jean-Noël Vittaut faisant évoluer son joueur constamment, nous présentons les dernières nouveautés qui rendent la réalisation d'une parallélisation performante encore plus complexe.

1 Le General Game Playing (GGP)

Ce chapitre commence par une brève présentation du *General Game Playing*. Nous parlons des compétitions organisées dans le but de faire progresser ce domaine de recherche et du langage de définition permettant de décrire les règles des jeux étudiés. Nous présentons ensuite des techniques d'exploration d'arbre utilisées pour explorer le déroulement des jeux et différentes approches tentées pour paralléliser une de ces techniques nommée MCTS. Enfin, nous concluons avec la présentation des réseaux de propositions (*propnets*), une nouvelle technique utilisée pour accélérer l'interprétation des règles des jeux.

1.1 Présentation du GGP

Le *General Game Playing* (GGP) est un champ de recherche faisant partie de l'intelligence artificielle et visant à réaliser des programmes capables de jouer à n'importe quel jeu en partant uniquement de la description de ses règles. Le concept de *General Game Playing* est assez récent¹ et n'a pris sa véritable ampleur qu'avec le projet initié par le *Stanford Logic Group* de l'université de Stanford. Leur projet d'une compétition de *General Game Playing* [Genesereth et al., 2005] a débouché également sur la mise en place de plateformes de jeu en ligne sur lesquelles les différents logiciels développés peuvent s'affronter.

La recherche dans le domaine de la programmation des jeux était motivée au départ par l'idée que la victoire d'un programme sur un joueur humain pourrait être une mesure fiable de son *intelligence* telle que recherchée dans le domaine de

1. Dans sa thèse, Quenault [2010] indique que l'étude des jeux représente tout un pan de l'intelligence artificielle. Les premiers programmes étaient des modèles spécialisés capables de battre des joueurs humains dans des jeux particuliers. Le principe de *General Game Playing* est apparu progressivement par une complexification croissante des modèles mis en place. Un système de jeu générique est défini par la présence de programmes joueurs réunis grâce à un programme capable de les mettre en relation et de les informer régulièrement sur l'état du jeu ; ceci nécessite l'usage d'un langage générique pour décrire le jeu. Quenault [2010] présente les travaux de Pitrat comme la première tentative pour créer un langage générique de description de règles de jeux. Le langage, que Pitrat [1971] décrit, permet de spécifier les règles de jeux à deux joueurs sur plateaux rectangulaires comme le *Tictactoe* ou les *Échecs*. Pitrat [1976] décrit également un programme capable d'apprendre des procédures ou enchaînements de coups favorables aux échecs. Les travaux de Pell [1993] sont similaires à ceux de Pitrat et toujours limités à une catégorie restreinte de jeux de plateau.

l'IA. Pell, dans sa thèse présentant le paradigme de *Metagame* [Pell, 1993], illustre clairement cette idée par une citation de Emanuel Lasker :

« In mathematics, if I find a new approach to a problem, another mathematician might claim that he has a better, more elegant solution. In chess, if anybody claims he is better than I, I can checkmate him. »

Avec la victoire de Deep Blue sur Garry Kasparov en 1997 et la réalisation d'autres programmes spécialisés comme Chinook (*Dames anglaises*) cette idée a montré ses limites : ces succès sont plus le résultat de l'intelligence du programmeur que d'une quelconque avancée en intelligence artificielle. Pell rappelle cela dans l'introduction de sa thèse :

« we have been able to produce strong programs for some games through specialized engineering methods, the extreme case being special-purpose hardware, and through analysis of the games by humans instead of by programs themselves. Consequently, it now appears that increased understanding and automation of intelligent processing is neither necessary nor sufficient for strong performance in game-playing. That is, it appears that we can construct strong game-playing programs without doing much of interest from an AI perspective, and conversely, we can make significant advances in AI that do not result in strong game-playing programs. » [Pell, 1993]

Pell propose alors de changer les données du problème de manière à forcer les chercheurs à concevoir des programmes faisant preuve d'une intelligence plus générale : en ne fournissant que les règles du jeu², le programme devrait alors être capable de les analyser pour en déduire une stratégie. Pour Pell, comme pour Clune [Clune, 2008], la généralité semble être une caractéristique fondamentale de l'intelligence et pour illustrer sa pensée il cite Robert A. Heinlein :

« A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects. »

Cette citation sera d'ailleurs reprise dans un article de 2005 [Genesereth et al., 2005] présentant la première compétition de General Game Playing.

2. Précisons que pour éviter la tentation d'injecter dans le programme un savoir propre au domaine, son auteur ne doit pas avoir connaissance du jeu en question. Le programme doit être capable de traiter une grande diversité de jeux inconnus uniquement à partir des règles qui les décrivent.

1.2 Le Game Definition Language (GDL)

Afin de pouvoir travailler sur une base commune et permettre aux différents joueurs de communiquer entre eux (par l'intermédiaire de l'arbitre dans le cas du GGP), il était nécessaire de concevoir un langage commun pour décrire les jeux. Plusieurs langages ont été proposés : Pell dans le cadre de son *Metagamer* [Pell, 1993] propose une grammaire décrivant les jeux de type échecs. Kaiser [Kaiser, 2005] propose RGL, un langage de définition de jeux fondé sur des expressions régulières décrivant des chemins dans un graphe. Il rappelle également l'existence de différentes notations propres à des jeux spécifiques, comme la *Portable Game Notation* (PGN) pour les mouvements aux échecs ou le *Smart Game Format* (SGF) surtout utilisé pour le jeu de Go et évoque d'autres tentatives pour créer des langages plus ou moins génériques de description des jeux.

C'est finalement le *Game Definition Language* (GDL), un langage de description des règles des jeux proposé par Michael Genesereth dans le cadre du projet de compétitions de General Game Playing initié par Stanford qui semble s'être imposé grâce à sa généralité. Conçu pour les jeux déterministes à information complète, Thielscher [Thielscher, 2011] montre néanmoins qu'il est suffisamment général pour décrire des jeux très différents et peut être facilement étendu aux jeux à information incomplète et soumis à une part de hasard. Il propose d'ailleurs dans ce but une version GDL-ii³ [Thielscher, 2010].

Ce *Game Definition Language* (version 1) dont la description est donnée par les créateurs de la compétition [Genesereth et al., 2005] et sur le site de Stanford [Stanford Logic Group, 2013], est un langage logique similaire à Prolog. Il est fondé sur la logique du premier ordre sans la notion d'arithmétique, bien que les nombres de 0 à 100 soient définis comme constantes pour décrire les scores en fin de partie. Certaines restrictions permettent d'assurer que toutes les règles logiques d'un jeu décrit en GDL sont totalement décidables. Un exemple de jeu décrit en GDL est donné à la figure 1.1. Le style du GDL est purement déclaratif et il possède quelques mots réservés pour décrire le jeu⁴ :

- (role r) signifie que r est un joueur dans le jeu.
- (base b) permet de préciser qu'un fait b est une position.
- (input r a) permet de préciser que a peut être un coup légal dans le jeu pour le joueur r.
- (init p) signifie que la proposition p est vraie à l'état initial, i.e. au début du match.
- (true p) signifie que la proposition p est vraie à l'état courant.
- (does r a) signifie que le joueur r fait l'action a dans l'état courant.

3. Les lettres "ii" figurent ici pour "information incomplète".

4. D'autres termes ont été ajoutés par la suite. (base a) et (input r a) font partie du standard utilisé par Stanford mais n'étaient pas présents dans la version de GDL utilisée pendant les premières années de la compétition. La version 2 de GDL utilise également les termes (sees r f) et random pour indiquer respectivement un fait f perçu par le joueur r dans la position suivante et un élément soumis au hasard.

- (next p) signifie que la proposition p sera vraie à l'état suivant.
- (legal r a) signifie que l'action a est un *coup légal* pour le joueur r dans l'état courant.
- (goal r n) signifie que le joueur r obtient la récompense n .
- terminal signifie que l'état courant est un état final.

D'autres termes constituent le langage logique :

- ($\leq x$ y_0 y_1 ... y_n) signifie que x est impliqué par les faits y_0 à y_n . Les ET logiques sont implicites entre les éléments y_0 à y_n .
- $?x$ signifie que x est une variable.
- (not x) permet d'obtenir la négation d'un fait.
- (distinct x y) indique que les éléments x et y ne peuvent être identiques.
- Les OU logiques sont implicites entre plusieurs règles de même conséquent. La première version du GDL comprenait un OU explicite ($\text{or } x$ y) qui a été supprimé par la suite⁵.

1.3 Compétitions de GGP

Depuis 2005, une compétition est organisée chaque année à l'occasion de la conférence de l'association pour l'avancement de l'intelligence artificielle *AAAI Conference* pour récompenser les meilleurs programmes réalisés (voir figure 1.2). Les conditions de la compétition ont évolué depuis sa conception, par exemple une partie de la compétition se déroule désormais en ligne, seule la finale se dispute à l'issue de la conférence AAAI ou IJCAI selon les années. Cependant le principe reste globalement le même.

Une machine-arbitre délivre aux machines des joueurs les règles d'un jeu inconnu des participants ; elle indique à chaque programme quel rôle il doit tenir dans le jeu. La diversité des jeux proposés est grande : ils peuvent être pour un seul joueur, deux ou multi-joueurs, à coups simultanés ou alternés avec une récompense numérique pour chaque joueur définie entre 0 et 100. Ces jeux peuvent également être à somme nulle ou non-nulle.⁶ Chaque programme joueur dispose d'un temps limité pour analyser les règles. À la fin de ce temps de préparation, l'arbitre donne aux programmes un temps limite pour indiquer le coup qu'ils décident de jouer. À la fin de chaque délai, les différents joueurs indiquent l'action choisie. Tous les joueurs jouent à chaque tour mais, pour les jeux à coups alternés, un seul joueur donne un coup *actif*, les autres se contentent d'indiquer un coup sans effet. Pour dissuader les

5. Il est à noter que malgré la suppression du OU explicite, devenu obsolète dans les spécifications GDL, il apparaît encore dans certains jeux de la compétition : il était présent par exemple dans le jeu Eight-Puzzle (pousse pousse 3×3) proposé le premier jour de la compétition de 2014.

6. Dans un jeu à somme nulle, ce que gagne un joueur est perdu par l'autre, les gains de l'un correspondent aux pertes de l'autre (exemple : score 100/0, 50/50 ou 0/100), dans un jeu à somme non-nulle, chaque joueur peut accumuler des points indépendamment de l'autre (exemple : score 25/25, 100/30, 75/50)

```

(role xplayer)
(role oplayer)

(index 1) (index 2) (index 3)
(<= (base (cell ?x ?y b)) (index ?x) (index ?y))
(<= (base (cell ?x ?y x)) (index ?x) (index ?y))
(<= (base (cell ?x ?y o)) (index ?x) (index ?y))
(<= (base (control ?p)) (role ?p))

(<= (input ?p (mark ?x ?y)) (index ?x) (index ?y) (role ?p))
(<= (input ?p noop) (role ?p))

(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control xplayer))

(<= (next (cell ?m ?n x)) (does xplayer (mark ?m ?n)) (true (cell ?m ?n b)))
(<= (next (cell ?m ?n o)) (does oplayer (mark ?m ?n)) (true (cell ?m ?n b)))
(<= (next (cell ?m ?n ?w)) (true (cell ?m ?n ?w)) (distinct ?w b))
(<= (next (cell ?m ?n b)) (does ?w (mark ?j ?k)) (true (cell ?m ?n b))
(or (distinct ?m ?j) (distinct ?n ?k)))
(<= (next (control xplayer)) (true (control oplayer)))
(<= (next (control oplayer)) (true (control xplayer)))

(<= (row ?m ?x) (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
(<= (column ?n ?x) (true (cell 1 ?n ?x)) (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
(<= (diagonal ?x) (true (cell 1 1 ?x)) (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
(<= (diagonal ?x) (true (cell 1 3 ?x)) (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))

(<= (line ?x) (row ?m ?x))
(<= (line ?x) (column ?m ?x))
(<= (line ?x) (diagonal ?x))

(<= open (true (cell ?m ?n b)))

(<= (legal ?w (mark ?x ?y)) (true (cell ?x ?y b)) (true (control ?w)))
(<= (legal xplayer noop) (true (control oplayer)))
(<= (legal oplayer noop) (true (control xplayer)))

(<= (goal xplayer 100) (line x))
(<= (goal xplayer 50) (not (line x)) (not (line o)) (not open))
(<= (goal xplayer 0) (line o))
(<= (goal oplayer 100) (line o))
(<= (goal oplayer 50) (not (line x)) (not (line o)) (not open))
(<= (goal oplayer 0) (line x))

(<= terminal (line x))
(<= terminal (line o))
(<= terminal (not open))

```

FIGURE 1.1 – Description du jeu de tictactoe en GDL
(fichier tictactoe.kif extrait du serveur Tiltyard [Schreiber, 2014]).

Année	Programme	Développeur
2005	ClunePlayer	Jim Clune
2006	FluxPlayer	Stephan Schiffel, Michael Thielscher
2007	CadiaPlayer	Yngvi Bjornsson, Hilmar Finnsson
2008	CadiaPlayer	Yngvi Bjornsson, Hilmar Finnsson
2009	Ary	Jean Mehat
2010	Ary	Jean Mehat
2011	TurboTurtle	Sam Schreiber
2012	CadiaPlayer	Yngvi Bjornsson, Hilmar Finnsson
2013	TurboTurtle	Sam Schreiber
2014	Sancho	Steve Draper, Andrew Rose

FIGURE 1.2 – Gagnants de la compétition de *General Game Playing* organisée chaque année depuis 2005.

interventions humaines lors des parties en ligne, les règles du jeu sont brouillées de manière à ce qu’aucun mot clef comme *cell* ou *succ* ne puisse servir de guide par son sens. À la place, des mots dénués de sens comme *foo* et *bar* sont utilisés.

Les résultats de cette compétition sont une assez mauvaise évaluation de la réelle valeur des programmes réalisés. Toutes sortes de problèmes peuvent survenir pendant les compétitions : problèmes d’ingénierie ou défaillances des machines qui n’ont pas forcément de rapport direct avec la qualité des modèles développés. Le hasard du tirage des adversaires qui devront s’affronter peut également biaiser les résultats. Il est à noter par exemple que *Ary* n’a jamais affronté *CadiaPlayer* pendant une de ces compétitions. Le nombre de matchs pendant cette compétition est également très faible et il est difficile d’obtenir une idée fiable de la performance d’un programme sur un nombre si restreint d’épreuves. Les programmes réalisés ne sont pas forcément en adéquation avec les jeux présentés une année donnée. La nature des jeux peut favoriser certains concurrents : en 2010, par exemple, aucun jeu pour un seul joueur n’a été proposé.

Cependant, malgré tous ces inconvénients, les compétitions de *General Game Playing* de Stanford sont quasiment les seules rencontres internationales organisées dans le monde et les résultats de ces compétitions sont à peu près⁷ les seules évaluations disponibles pour se faire une idée de la validité des démarches utilisées.

Depuis sa création en 2005, la compétition de *General Game Playing* a permis de révéler différentes approches intéressantes dont celles de Kuhlmann, Dresner et

7. Divers matchs amicaux sont disputés en ligne notamment sur les serveurs de Stanford et de Dresde. On peut noter l’organisation de quelques compétitions nationales ou limitées à une université donnée. La *German General Game Playing Competition* (GGGPC) a été organisée pour la première fois en 2009 ; les participants étaient tous des chercheurs d’universités allemandes. En 2011 une compétition ouverte aux équipes internationales s’est tenue pendant la 34ème Conférence Allemande d’intelligence artificielle (KI-11) ; il est à noter que le GDL-ii y a été utilisé pour la première fois afin de proposer des jeux à information incomplète et comportant une part de hasard. Les compétitions de GGP organisées depuis 2006 à l’université de Dresde sont organisées pour leurs étudiants. Une rencontre entre l’université de Dresde et celle de Reykjavik a été organisée en 2009 entre 6 équipes d’étudiants.

Stone [Kuhlmann *et al.*, 2006], Clune [Clune, 2007], Schiffel and Thielscher [Schiffel et Thielscher, 2007a], Finnsson and Björnsson [Finnsson et Björnsson, 2008]. On peut remarquer deux types d'approches distinctes, les plus anciennes mettaient l'accent sur l'extraction de connaissances et la création de fonctions d'évaluation fondées sur des heuristiques ; néanmoins elles utilisaient également des algorithmes de parcours d'arbre de la famille *Mini-Max* [Kuhlmann *et al.*, 2006; Schiffel et Thielscher, 2007b; Clune, 2007; Kaiser, 2007; Kirci *et al.*, 2011]. Les plus récentes sont fondées sur les algorithmes de Monte Carlo et ses variantes [Finnsson et Björnsson, 2011; Méhat et Cazenave, 2008] qui ont considérablement amélioré la performance des explorations des arbres des jeux. Associée aux algorithmes de Montecarlo, une nouvelle approche, utilisant un réseau de propositions (*propnet*) pour accélérer la simulation des parties pendant l'exploration d'arbre, a assuré la victoire à *TurboTurtle* en 2013.

1.4 Exploration d'arbre

Un jeu peut être représenté sous la forme d'un arbre⁸ dont chaque nœud représente un état de la partie à un moment donné et chaque lien représente un coup légal, i.e. un coup autorisé par les règles du jeu. La racine représente l'état initial du jeu et chaque feuille représente un état terminal où le score final peut enfin être connu.

À chaque nœud, le joueur doit choisir le coup qui lui sera le plus favorable pour obtenir le meilleur score possible. Dans ce but, diverses techniques ont été mises au point, à commencer par celles de la famille *Mini-Max* que nous exposons dans un premier temps. Ces méthodes étaient utilisées par les premiers joueurs programmés pour le *General Game Playing* et deux d'entre eux, *Clune Player* [Clune, 2007] et *Flux Player* [Schiffel et Thielscher, 2007a,b] ont gagné les compétitions de 2005 et 2006.

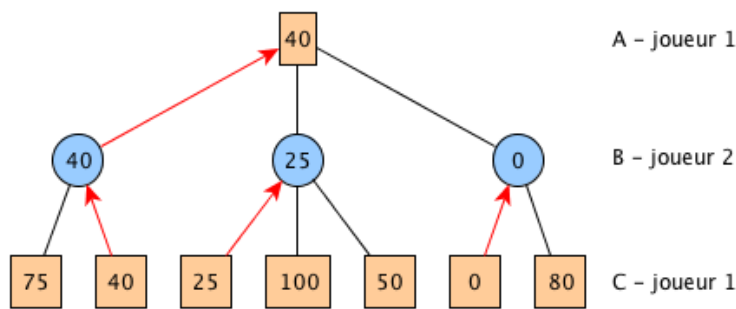
Cependant depuis 2007, les techniques de Monte-Carlo ont été appliquées avec grand succès à l'exploration d'arbre et elle se sont imposées avec la victoire de plusieurs joueurs les utilisant : *Cadia Player* [Finnsson et Björnsson, 2011; Finnsson, 2012a], *Ary* [Méhat et Cazenave, 2008, 2010] et *Turbo Turtle*. Nous verrons ces techniques dans un second temps et quelques améliorations utilisées par ces joueurs de GGP.

8. En réalité, dans le cadre des jeux, l'arbre est plutôt un graphe acyclique⁹ orienté possédant une racine unique. En effet, plusieurs branches peuvent mener à une même position par le jeu des transpositions. Pour la simplicité de la représentation, les jeux sont représentés ici sous formes d'arbres simples et les positions identiques sont dupliquées pour apparaître dans chaque branche.

9. Les spécifications GDL indiquent qu'un jeu *bien formé* doit être :

- *fini*, chaque séquence de coups légaux doit conduire, à partir de l'état initial, à un état terminal en un nombre fini de coups,
- *jouable*, à chaque position du jeu chaque joueur doit disposer d'au moins un coup légal,
- *gagnable*, au moins une série de coups doit permettre d'atteindre un état terminal de score maximum.

Il ne peut donc pas y avoir de cycles dans le graphe représentant le jeu.

FIGURE 1.3 – Exploration d'un arbre avec *Mini-Max*

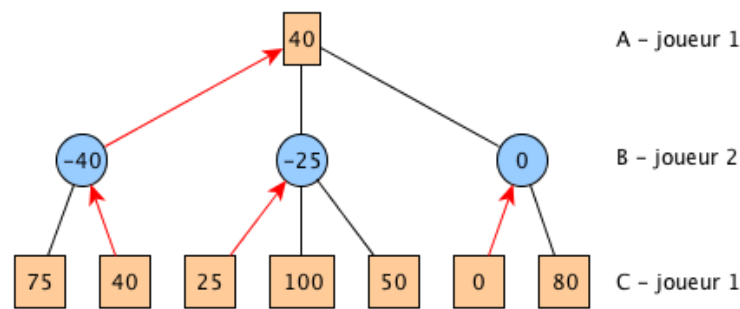
1.4.1 Famille Mini-Max

Avant l'apparition des techniques de Monte-Carlo (voir section 1.4.2), ce sont les algorithmes de la famille *Mini-Max* qui étaient utilisés pour le parcours d'arbre. *Mini-Max* consiste à chaque étape à maximiser le score du joueur courant (joueur 1) ou à minimiser celui de l'adversaire (joueur 2). Il part du principe que l'adversaire va choisir judicieusement à chaque étape du jeu le coup qui va minimiser le score de l'adversaire : ce qui implique à la fois que l'adversaire est capable de bien jouer et que minimiser le score du joueur 1 est directement corrélé à l'augmentation de son propre score, donc que le jeu est à somme nulle.

Une branche de l'arbre est d'abord parcourue en profondeur jusqu'à un état terminal et le score du joueur 1, associé à ce nœud, est relevé (voir étape *C* sur la figure 1.3). On considère que l'adversaire, à l'étape *B*, essaye d'optimiser son propre score, donc il va choisir les branches qui amènent au score minimal pour le joueur 1. À l'étape *A* le joueur 1 peut donc déduire qu'il a intérêt à choisir la branche de gauche pour assurer un score de 40 car, s'il vise le score 100 (au milieu) mais que le joueur 2 joue bien, il risque de se retrouver avec seulement 25.

Néga-Max est une variante de *Mini-Max* qui apporte essentiellement une simplification du code. Le principe est le même, mais il repose sur le fait que le score du joueur 1 est la valeur du handicap du joueur 2 : $\max(score) = -\min(-score)$ (voir figure 1.4). En lieu et place de deux fonctions de maximisation et de minimisation, on peut avoir une seule fonction :

```
negamax(position) :
    si la partie est terminée :
        renvoyer le score (score du joueur 1 ou handicap du joueur 2)
    res ← -∞
    pour chaque coup :
        tmp ← - negamax(nouvelle_position)
        if (tmp > res)
            res ← tmp
    renvoyer res
```

FIGURE 1.4 – Exploration d'un arbre avec *Néga-Max*

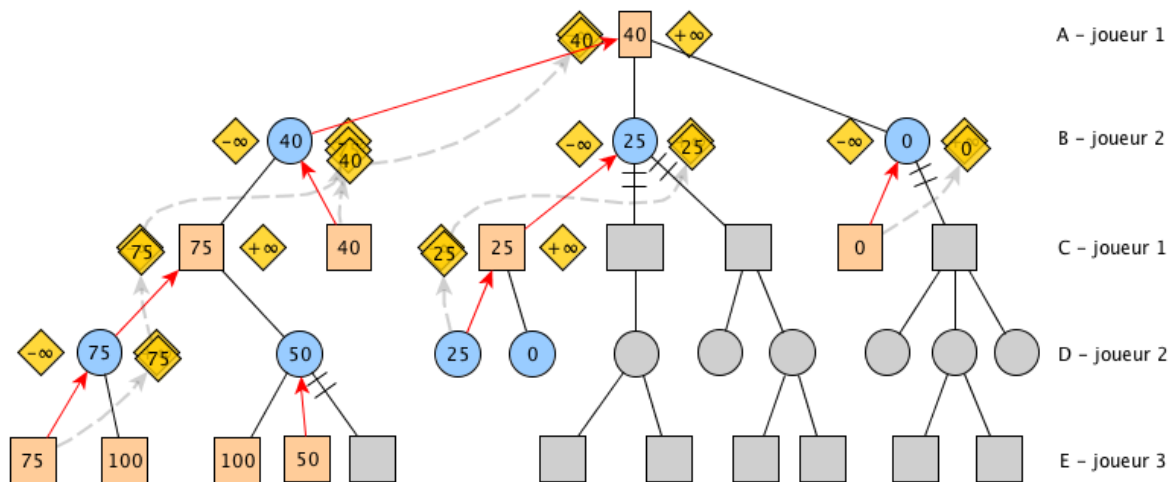
Mini-Max implique l'exploration complète de l'arbre du jeu, ce qui est impossible pour des jeux un tant soit peu intéressants et donc complexes¹⁰. Afin de simplifier l'exploration de l'arbre, *Alpha-Beta* permet de couper certaines branches dont il semble évident qu'elles ne mèneront pas à un meilleur score. À chaque position dans le jeu, le joueur courant tente d'évaluer entre quelles bornes minimales et maximales se situe son score. Ces bornes sont initialisées par défaut à $-\infty$ et $+\infty$, mais elles peuvent être initialisées en fonction des bornes connues du score pour le jeu testé. Une branche est explorée jusqu'à une fin de partie et le score est relevé (voir figure 1.5). Le score est alors remonté dans les nœuds de l'arbre et les bornes minimales et maximales sont ajustées en conséquence. Sur la figure 1.5 on voit par exemple que l'exploration de la branche de gauche assure au joueur 1 un score compris entre 40 et $+\infty$, l'exploration partielle des deux autres branches indique que le score sera de ce côté inférieur ou égal à 25, il est donc inutile d'explorer le reste de ces branches puisqu'il est évident qu'elles ne peuvent apporter un score meilleur.

La simplification vue pour *Néga-Max* peut également être adaptée à *Alpha-Beta*.

Ces deux techniques ont été utilisées par les premiers programmes de General Game Playing, par exemple le *Metagamer* de Pell capable de jouer à un groupe de jeux de plateau de type échecs. Pell utilise en plus la technique de l'approfondissement itératif qui consiste à explorer l'arbre en largeur d'abord et jusqu'à une profondeur donnée. Les nœuds non terminaux sont alors évalués avec différentes

10. L'intérêt d'un jeu est une notion relative. Certains jeux très simples peuvent s'avérer intéressants pour des joueurs humains en raison de la récompense (financière ou autre) qui en constitue l'enjeu. Un jeu très simple comme *Monty Hall*, dans lequel trois portes masquent deux récompenses sans intérêt et un lot prestigieux, peut également intéresser le joueur humain grâce à l'espoir d'obtenir le troisième lot, mais pose également un casse-tête probabiliste qui a suscité l'intérêt des mathématiciens (cf : *The Monty Hall Problem : Beyond Closed Doors* de Robert Deaves). Pour un jeu comme le *Tictactoe*, lorsque les deux joueurs ont compris la mécanique du jeu, l'intérêt subsiste uniquement dans la possibilité de déconcentrer l'adversaire pour le piéger. Sinon la partie s'achève invariablement par un match nul et l'intérêt du jeu disparaît. D'un autre côté certains jeux complexes peuvent s'avérer dénués d'intérêt. C'est le cas du jeu *Add Game* joué sur un plateau de 2x2 cases : les règles sont ridiculement complexes alors que le jeu est trivial.

La complexité à laquelle nous faisons référence n'est pas celle provenant de l'apprentissage du jeu, i.e. la complexité des règles, mais celle qui résulte du grand nombre de coups possibles et de la diversité des stratégies qui en découle. Dans des jeux présentant cette complexité, c'est dans l'exploration de ces possibilités et la mise au point de stratégies que l'intérêt réside.

FIGURE 1.5 – Élagage *Alpha-Beta* ($\alpha\beta$)

heuristiques (nombre de pions en possession des joueurs, qui contrôle le centre, etc.). Si le programme dispose de temps pour affiner son évaluation, il explore l'arbre un peu plus profondément et remonte sa nouvelle évaluation dans les nœuds précédents pour mettre à jour leur évaluation. Un classement des coups est effectué pour examiner en premier les coups les plus prometteurs ; ceci permet à *Alpha-Beta* d'être plus efficace en favorisant les coups.

Kuhlmann, Dresner et Stone [Kuhlmann *et al.*, 2006] et Clune [Clune, 2007] utilisent également *MiniMax* et *Alpha-Beta* pour leur joueurs respectifs *UTexas LARG* et *Clune Player* et y ajoutent le principe des tables de transposition. Ces tables permettent de mémoriser les différentes positions du jeu et leur évaluation de manière à ne pas devoir réévaluer une position qui peut se présenter au bout de différentes successions de coups. Pour retrouver rapidement une position et son évaluation, les positions sont associées à une clef de hashage à la Zobrist.

Le joueur de Schiffel et Thielscher, nommé *FluxPlayer* [Schiffel et Thielscher, 2007a,b], utilise également ces techniques classiques d'élagage *Alpha-Beta* et des tables de transposition avec une recherche itérative en profondeur d'abord.

1.4.2 Techniques de Monte Carlo

Le principe de base de la technique de *Monte Carlo* brille par sa simplicité. À chaque position du jeu, la liste des coups légaux est établie pour le joueur 1 dont le score doit être maximisé. Un coup est tiré au hasard dans cette liste et un *playout* est réalisé. Le *playout* est une simulation de la partie : tous les coups suivants sont joués au hasard, dans le respect des règles du jeu, jusqu'à atteindre un état terminal, i.e. jusqu'à ce la partie soit finie et qu'un score puisse être attribué à chaque joueur. Le score obtenu pour le joueur 1 est alors reporté dans une table en correspondance avec le premier coup choisi. Les simulations se succèdent de manière à

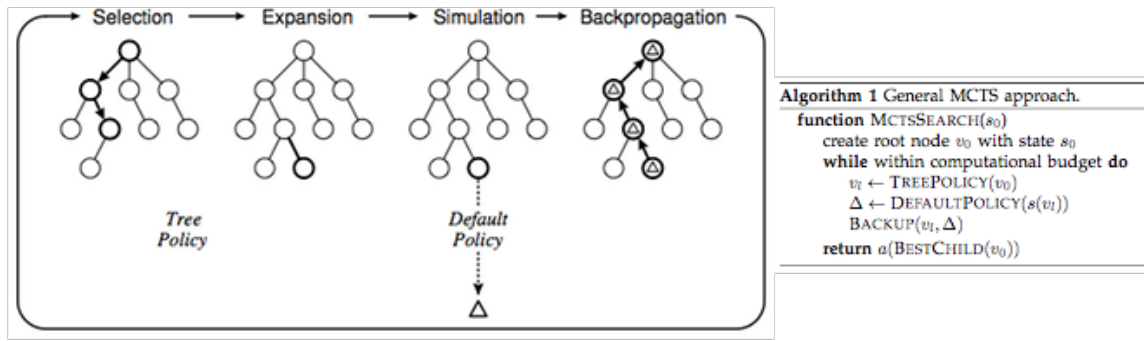


FIGURE 1.6 – Principe général de la méthode Monte Carlo Tree Search [Browne et al., 2012]

ce que chaque coup légal de la liste de départ se voit attribuer un score moyen. Un temps limite est accordé pour la réalisation de ces simulations : à la fin du temps imparti, le coup associé avec le gain moyen le plus fort est joué.

Cependant, cette utilisation basique peut facilement amener à un mauvais choix. Par exemple, si une branche de l'arbre du jeu mène à un ensemble de score moyens (50/100) tandis que l'autre branche amène à un seul score optimal (100/100) et une grande quantité de défaites (0/100), la moyenne obtenue sur la première branche sera plus élevée et le coup choisi annihilera tout espoir d'aboutir au score optimal.

Pour remédier à ce problème, les méthodes de *Monte Carlo Tree Search* (MCTS) proposent une manière plus efficace d'explorer l'arbre du jeu. Un arbre est construit localement de manière incrémentale et asymétrique [Browne et al., 2012]. À chaque itération un nœud est choisi parmi les nœuds déjà explorés et un nouveau nœud est ajouté selon une stratégie donnée (*TreePolicy*). À partir de ce nœud une simulation est effectuée qui peut être guidée et non effectuée au hasard.

Par exemple, la version actuelle de *CadiaPlayer* (2011-2012) [Finnsson et Björnsson, 2011; Finnsson, 2012a] utilise UCT pour choisir un nœud de départ pour l'exploration de l'arbre et pendant les *playouts*, les nœuds ne sont pas choisis selon un hasard uniforme, mais selon la distribution de Gibbs (ou Boltzmann) :

$$P(a) = \frac{e^{Qh(a)/\tau}}{\sum_{b=1}^n e^{Qh(b)/\tau}} \quad (1.1)$$

avec $P(a)$ la probabilité qu'une action a de désirabilité $Qh(a)$ soit choisie. Le paramètre τ permet de doser l'influence de $Qh(a)$: quand τ tend vers 0 la probabilité favorise un nombre plus restreint de coups, et si τ augmente $P(a)$ tend vers une distribution uniforme. La désirabilité d'un coup est évaluée en fonction d'heuristiques. Une fois un état terminal atteint, le résultat est reporté dans les nœuds visités par rétro-propagation (voir figure 1.6).

Les stratégies de sélection (*TreePolicy*) sont fondées sur un compromis entre exploration et exploitation. Pour avoir une idée fiable de l'ensemble du jeu, il est nécessaire d'explorer la plus grande partie possible de l'arbre, mais cette exploration est coûteuse en temps. D'un autre côté, exploiter immédiatement les coups les plus

prometteurs peut sembler une meilleure idée, mais elle risque de conduire à écarter des coups qui auraient pu se révéler gagnants à long terme e.g. il peut être nécessaire de jouer un coup défavorable à court terme (sacrifier une pièce) pour obtenir une victoire. Il faut donc équilibrer ces deux aspects.

L'algorithme *Upper Confidence bound for Tree* (UCT) [Kocsis et Szepesvári, 2006] est le plus populaire parmi les méthodes MCTS et généralement MCTS sous-entend l'utilisation de UCT en l'absence de précision. Le principe d'UCT est de choisir comme départ de chaque *playout* un nœud enfant j qui maximise la valeur de U_j [Browne et al., 2012] :

$$U_j = \bar{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}} \quad (1.2)$$

$\bar{X}_j$ représente le gain moyen obtenu pour ce nœud, C est une constante permettant de pondérer les deux aspects *exploration* et *exploitation*, n est le nombre de fois que le nœud parent a été visité et n_j le nombre de visites pour le nœud j .

Si $n_j = 0$ alors la valeur UCT est indéfinie. Pour chaque nœud parent, les nœuds enfants non explorés sont visités au moins une fois avant que le choix d'exploiter le meilleur ne survienne. L'autre élément notable, est la nécessité de choisir une valeur pour C , choix qui va déterminer le poids des deux aspects *exploration* et *exploitation* de l'algorithme.

1.4.3 Améliorations de MCTS

Jean Mehat et Tristan Cazenave [Méhat et Cazenave, 2008, 2010] utilisent UCT comme base de leur joueur *Ary*. Afin d'augmenter les performances d'UCT, ils utilisent deux améliorations : la technique des tables de transposition et le Nested Monte-Carlo Search Algorithm pour les jeux à un seul joueur. De leur côté Yngvi Björnsson et Hilmar Finnsson [Finnsson, 2007; Finnsson et Björnsson, 2008; Finnsson et Björnsson, 2009, 2011; Finnsson, 2012a] utilisent également UCT pour leur joueur *CadiaPlayer* et ils ont ajouté successivement, entre 2007 à 2009, plusieurs améliorations : Move-Average Sampling Technique (MAST), Rapid Action Value Estimation (RAVE) et Features-to-Action Sampling (FAST).

Pendant une partie, différentes successions de coups peuvent amener à la même situation. Afin de ne pas avoir à explorer deux fois une branche qui est présente à l'identique dans deux parties de l'arbre, les positions et leur évaluation sont stockées dans une *table de transposition*, ce qui permet de diminuer significativement le temps de calcul en réutilisant des informations déjà acquises. Ce temps peut alors être utilisé pour réaliser un nombre plus important de *playouts*. C'est en utilisant UCT avec une table de transposition que *Ary* a gagné la compétition de GGP de 2009.

Le *Nested Monte-Carlo Search Algorithm* (NMC) [Cazenave, 2009], utilisé pour les jeux à un seul joueur, consiste à orienter les *playouts* successifs en fonction des

playouts précédents en mémorisant le chemin qui a mené au meilleur score lors des simulations précédentes et à jouer le premier coup qui y mène. À chaque coup joué, une nouvelle série de *playouts* est effectuée et la suite du meilleur chemin est éventuellement remplacée par un enchaînement de coups encore meilleurs. Cette approche apporte un bon équilibre entre exploration et exploitation [Méhat et Cazenave, 2010] et présente l'avantage de ne nécessiter aucun ajustement de paramètre contrairement à UCT. Les performances de UCT et NMC se surpassent l'une l'autre en fonction des jeux, si bien que Mehat et Cazenave ont utilisé pour leur joueur une fonction d'évaluation mêlant les deux approches dans le cas des jeux à un seul joueur. À chaque position, le coup joué est celui recevant la meilleure évaluation combinée des deux algorithmes. Ary a également gagné la compétition de General Game Playing en 2010 mais, cette compétition n'ayant pas proposé de jeux pour un seul joueur, l'efficacité de ces améliorations n'a pu être mise à l'épreuve.

La *Move-Average Sampling Technique* (MAST) consiste, après chaque *playout* lorsque le score final est remonté à travers les nœuds de l'arbre, à reporter la valeur moyenne de chaque action dans une *lookup table* ; ceci permet d'évaluer la valeur d'une même action quel que soit le moment de la partie où elle est faite. Pour de nombreux jeux cette stratégie présente un avantage clair (occupation du centre au tic-tac-toe ou des angles à Othello). La valeur estimée de chaque action est utilisée pour biaiser le choix des coups pendant les *playouts* selon la distribution de Boltzmann vue avant.

TO-MAST [Finnsson, 2012a] est une petite variation de MAST où les nœuds mis à jour sont limités à ceux de l'arbre déjà exploré plutôt que tous les nœuds du *playout* ce qui permet de baser les décisions sur des données plus robustes.

La *Rapid Action Value Estimation* (RAVE) hérite d'une méthode appliquée aux programmes de Go connue sous le nom de *All-Moves-As-First* (AMAF) [Browne et al., 2012]. Elle consiste à faire remonter la valeur d'un état final non seulement sur les actions sélectionnées pendant le *playout*, mais également sur les actions identiques qui peuvent avoir été écartées lors des choix successifs pendant la descente. Contrairement à AMAF qui utilise une évaluation par action, seules les actions de la branche courante sont mises à jour ; ceci permet de distinguer les coups propices dans une situation et néfaste dans d'autres. Pour chaque coup, la valeur estimée par UCT et la valeur RAVE sont toutes les deux calculées séparément. Les valeurs RAVE et UCT sont pondérées de manière à ce que RAVE soit favorisé pour sélectionner l'action de départ d'un *playout* tant que le nombre de *playouts* est trop faible pour que la valeur UCT soit fiable. À mesure que la valeur UCT se précise, le poids de RAVE est diminué. Une constante est définie pour indiquer le nombre d'évaluations à partir duquel la valeur UCT est considérée suffisamment fiable pour remplacer totalement la valeur RAVE.

La *Predicate-Average Sampling Technique* (PAST) fonctionne comme MAST à la différence que c'est la valeur moyenne de chaque couple action-prédicat qui est calculée. Selon le principe du GDL, chaque état du jeu est caractérisé par un certain nombre de prédicats qui sont vrais. Pendant la phase de rétro-propagation, pour chaque nœud ou l'action a a été choisie, la valeur de chaque prédicat vrai est mise à jour. Pendant un *playout*, pour le calcul de la probabilité dans la formule de Gibbs,

c'est la moyenne la plus élevée parmi les prédicats qui est utilisée par *CadiaPlayer*. Cette valeur maximale est utilisée plutôt que la moyenne sur tous les prédicats pour limiter le temps de calcul. PAST permet de choisir les coups qui sont bons en fonction d'un certain contexte.

Comme preuve de l'efficacité de MCTS, *CadiaPlayer* a gagné les compétitions de 2007 et 2008, cependant, les concepteurs de *CadiaPlayer* indiquent que malgré l'efficacité des algorithmes de Monte Carlo, il reste nécessaire d'utiliser des heuristiques pour introduire une forme de compréhension du jeu de plus haut niveau :

« The aforementioned schemes do not use any game-specific domain knowledge. Although this has the benefit of allowing effective deployment over a wide range of disparate games, this approach seems simplistic in contrast to human players, which use high-level features such as piece types and board geometry in their reasoning. The lack of understanding of such high-level game concepts does indeed severely handicap GGP players using simple search-control schemes in certain types of games, for example chess-like games where a good understanding of the different piece type values is essential for competitive play » [Finnsson et Björnsson, 2011]

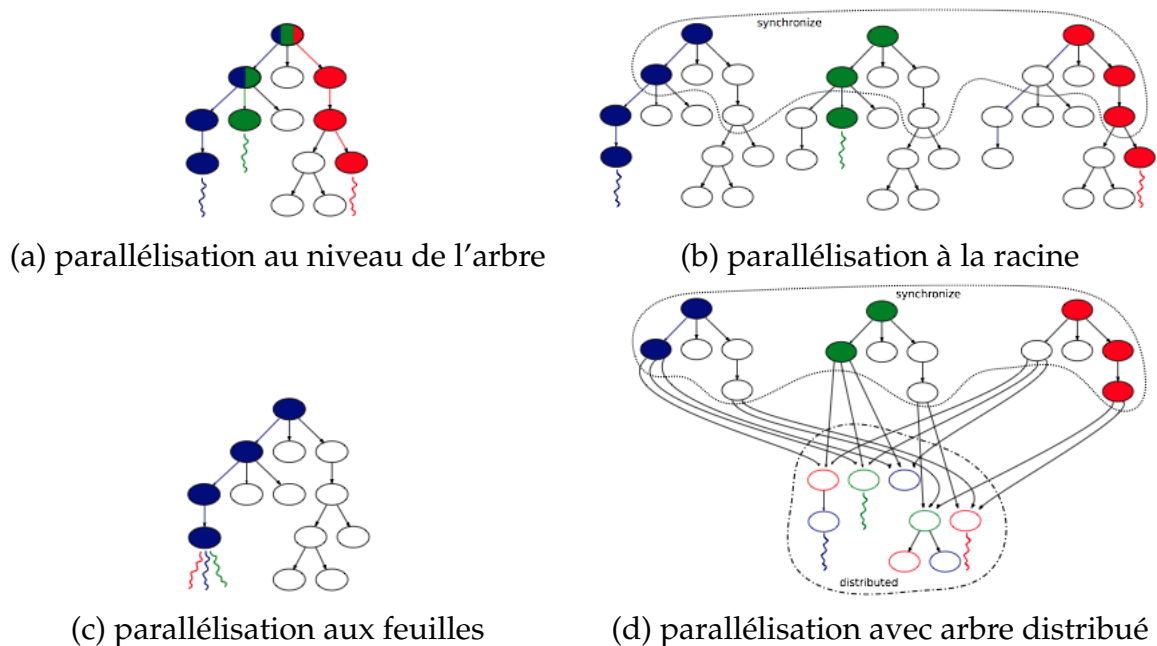
Björnsson et Finnsson ont donc ajouté à leur joueur la Features-to-Action Sampling Technique (FAST) qui utilise le Temporal Difference Learning $TD(\lambda)$ [Sutton, 1988] pour évaluer différentes caractéristiques du jeu (position ou déplacement d'une pièce dans une case, etc.). Pendant la phase préparatoire d'analyse du jeu, dans certains cas¹¹, FAST détermine des caractéristiques pertinentes pour l'évaluation des positions. Cet ensemble de caractéristiques est utilisé durant le reste du jeu pour estimer la valeur des actions et orienter la recherche.

Finnsson indique que MAST et FAST peuvent être utilisés conjointement en pondérant les résultats des deux mais la valeur optimale de cette pondération reste à déterminer. MAST, TO-MAST et PAST sont des variantes inconciliables de la même technique ; elle ne peuvent être utilisées ensemble ; c'est donc une seule de ces variantes qui doit être intégrée à la fonction d'évaluation de *CadiaPlayer* en fonction du type de jeu mais là aussi le critère de choix est à déterminer et nécessite de futures recherches.

1.5 Parallélisation de MCTS

L'amélioration de l'évaluation d'une position fournie par une recherche MCTS peut être obtenue non seulement en augmentant la taille limite de l'arbre construit, ce qui permet d'explorer une plus grande partie de la branche prometteuse de l'espace de recherche, mais également en augmentant le nombre de *playouts* ce qui permet d'obtenir une évaluation plus juste des positions. Kocsis et Szepesvári [2006] indiquent qu'en théorie la probabilité de choisir le meilleur coup avec l'algorithme

11. Nous sommes sceptiques quant à l'efficacité réelle de FAST

FIGURE 1.7 – Aperçu des différentes techniques de parallélisation de MCTS [Graf *et al.*, 2011]

UCT tend vers 1 quand le nombre de *playouts* tend vers l'infini. Pour augmenter le nombre de simulations lors de l'exploration de l'arbre du jeu, différentes techniques de parallélisation ont été tentées dont certaines permettent aussi la construction d'un arbre plus important. Nous allons maintenant en décrire quelques unes.

1.5.1 Parallélisation au niveau de l'arbre

Pour augmenter le nombre de simulations lors de l'exploration de l'arbre du jeu, Gelly *et al.* [2006] sont les premiers à proposer une parallélisation de MCTS appliquée au jeu de Go (Mogo) en construisant un arbre unique dans une mémoire partagée (voir figure 1.7 (a)) par plusieurs threads réalisant les *playouts*. L'accès à l'arbre est protégé par un mutex. Cette technique de parallélisation, nommée *tree parallelization* dans Chaslot *et al.* [2008] et Chaslot [2010] est limitée dans sa performance par le poids de cette synchronisation par mutexes. D'après les résultats obtenus par Gelly *et al.* [2006] on constate que l'algorithme passe mal à l'échelle au delà de 4 threads exécutés sur 4 processeurs distincts.

Pour remédier à cela, Chaslot *et al.* [2008] proposent d'utiliser des mutexes locaux : chaque nœud de l'arbre possède son propre mutex ce qui permet de verrouiller une zone limitée de l'arbre et permet aux threads de travailler en même temps. Pour éviter le surcoût dû aux fréquents verrouillages et déverrouillages des mutexes, ils utilisent un système de verrouillage fondé sur une attente active (*spinlock*).

Cependant, cette amélioration est au prix d'une augmentation de la taille de la structure de données représentant chaque nœud [Enzenberger et Müller, 2010]. De

plus, puisque les résultats des $N-1$ *playouts* précédents ne sont pas encore remontés quand le thread N démarre une simulation à partir d'un nœud, les N threads risquent de tous explorer la même zone prometteuse de l'arbre [Gelly *et al.*, 2008].

Pour éviter qu'un nœud ne soit ainsi trop exploré et que tous les threads attendent le déblocage du même mutex local, Chaslot *et al.* [2008] et Chaslot [2010] proposent d'utiliser une perte virtuelle (*virtual loss*). Cette perte virtuelle consiste à baisser l'évaluation d'un nœud quand il est choisi comme point de départ d'un *playout*. Ainsi ce nœud ne sera choisi par le thread suivant que si son évaluation est toujours supérieure à celle des autres nœuds. Sinon c'est le nœud possédant la seconde meilleure évaluation qui sera choisi comme point de départ d'une simulation par le thread suivant. Cette perte virtuelle est supprimée lorsque l'évaluation du nœud est remontée à la fin du *playout*.

Enzenberger et Müller [2010] indiquent que l'utilisation de mutexes provoque rapidement des temps d'attente qui empêchent le passage à l'échelle. Ils proposent un algorithme de *tree parallelization* sans mutexes qui permet de limiter les erreurs provoquées par des accès concurrents à l'arbre du jeu. En limitant la perte de temps due aux mutexes, Enzenberger et Müller [2010] montrent que sur le jeu de Go la recherche effectuée avec N threads est équivalente en terme de parties gagnées à la version séquentielle avec N fois plus de temps.

1.5.2 Parallélisation à la racine

Une autre méthode de parallélisation nommée *root parallelization* [Chaslot *et al.*, 2008; Chaslot, 2010] est proposée par Cazenave et Jouandeau [2007] (voir figure 1.7 (b)). Elle consiste à développer N arbres UCT distincts sur chacun des esclaves à partir de la racine. À la fin du temps de réflexion, les valeurs calculées par les différents esclaves pour les nœuds enfants de la racine sont additionnées par le processus maître et le meilleur nœud est choisi.

En plus de cette procédure nommée *single-run*, Cazenave et Jouandeau [2007] testent une approche *multiple-run* où le temps de réflexion est découpé en plusieurs phases à la fin de chacune desquelles la valeur des nœuds est actualisée : le processus maître regroupe les valeurs calculées par les différents esclaves, calcule une moyenne et relance la recherche de chaque esclave avec les valeurs actualisées. Cette approche *multiple-run* implique une augmentation des temps de communication mais donne de meilleurs résultats car elle apporte un partage des connaissances qui permet de mieux orienter la recherche. Gelly *et al.* [2008] étendent cette technique en synchronisant à intervalles de temps réguliers les évaluations des nœuds enfants de la racine qui ont été suffisamment visités sur k niveaux de profondeur.

Les travaux présentés jusqu'à présent testent les différentes techniques de parallélisation uniquement sur le jeu de Go. Méhat et Cazenave [2011a] proposent l'application de la parallélisation à la racine au domaine du General Game Playing. Ils utilisent la parallélisation de type *single-run* proposée par Cazenave et Jouandeau [2007] mais expérimentent différentes approches pour fusionner les résultats des

différentes recherches UCT effectuées par les esclaves.

Ils proposent une stratégie nommée *Best* qui consiste à choisir le coup ayant la meilleure évaluation toutes recherches confondues. La stratégie *Sum* consiste à fusionner les différentes évaluations e_s d'un nœud s en les tempérant par le nombre de simulations p_s correspondant, puis d'en faire la moyenne par rapport au nombre total de simulations :

$$\sum e_s \times p_s / \sum p_s \quad (1.3)$$

Ils indiquent que cette stratégie risque de favoriser les *faux bons coups*. Pour vérifier cela, ils testent une stratégie *Sum10* similaire mais ne prenant en compte que les 10 meilleurs coups de chaque recherche UCT. Dans cette approche, les mauvaises évaluations correspondant à des *faux bon coups* ne seront pas prises en compte et ne pourront pas contrer les bonnes évaluations des recherches UCT n'ayant pas détecté ces coups comme mauvais .

La dernière stratégie *Rave* est celle utilisée par Cazenave et Jouandeau [2007] et consiste à cumuler les évaluations sans tenir compte du nombre de simulations :

$$\sum e_s / \sum p_s \quad (1.4)$$

Théoriquement cette approche est un moyen simple d'éviter le biais en faveur des *faux bon coups*. Les résultats expérimentaux montrent que les meilleurs performances sont finalement obtenues avec *Sum* et *Sum10* dont les résultats sont équivalents. Comparé à une recherche séquentielle effectuée sur un même laps de temps, le gain apporté par la parallélisation sur N esclaves en terme de niveau de jeu pour les différents jeux testés¹² est positif bien que parfois inférieur ou identique à une recherche séquentielle bénéficiant de N fois plus de temps. Contrairement à cette parallélisation à la racine qui ne fonctionne pas pour tous les jeux, Méhat et Cazenave [2011b] indiquent que la parallélisation au niveau de l'arbre améliore les résultats pour tous les jeux testés par rapport à un joueur séquentiel bénéficiant de N fois plus de temps. La performance de cette parallélisation au niveau de l'arbre pour le GGP dépend principalement de la capacité du joueur à garder les processus esclaves occupés.

Soejima *et al.* [2010] proposent encore une autre approche pour la parallélisation à la racine en comparant la stratégie de sélection par moyenne (*Rave*) et un vote à la majorité : chaque recherche UCT aboutit à la sélection d'une action à jouer qui constitue un vote, le processus maître collecte alors ces votes et l'action remportant la majorité est choisie. Soejima *et al.* [2010] indiquent que la sélection par moyenne peut amener à choisir ce que la majorité des recherches a identifié comme le deuxième meilleur coup, la sélection par vote à la majorité peut donc éviter ce problème. Leur expérimentation sur le jeu de Go (Fuego vs. Mogo) montre qu'il y a

12. Les jeux testés sont blocker, breakthrough, checkers, connect4, othello, pawn whopping, pentago et skirmish.

peu de différence entre les deux stratégies sur une grille 9x9 car les deux stratégies amènent très souvent au choix des mêmes coups. Cependant sur une grille 19x19, le vote à la majorité améliore significativement le niveau de jeu.

Soejima *et al.* [2010] remarquent que la parallélisation au niveau de l'arbre est considérée plus efficace que la parallélisation à la racine à cause du partage de connaissance que permet la première approche et le fait que l'arbre construit est potentiellement plus grand. Cependant la parallélisation au niveau de l'arbre n'est pas applicable telle quelle dans le cas d'un système distribué.

1.5.3 Parallélisation au niveau des feuilles

Une troisième technique de parallélisation nommée *leaf parallelization* (voir figure 1.7 (c)) est proposée pour la première fois par Cazenave et Jouandeau [2007] pour une architecture avec une mémoire distribuée. L'arbre est construit par le processus maître et seuls les *playouts* sont exécutés par les processus esclaves. À partir d'une feuille, chaque esclave fait un certain nombre de simulations et retourne l'évaluation moyenne obtenue. Le processus maître effectue alors une moyenne de ces moyennes.

Le processus maître doit attendre le résultat d'un groupe de *playout* avant d'en lancer un autre et à chaque nouveau nœud une nouvelle communication avec les processus esclaves est nécessaire : ceci provoque un surcoût important en temps de synchronisation et de communications. Un autre inconvénient vient du fait que chez Cazenave et Jouandeau [2007] les *playouts* sont tous effectués à partir de la même position sans ajustement de l'évaluation comme dans la version séquentielle d'UCT. Ces deux inconvénients font que les résultats obtenus avec cette technique sont inférieurs en terme de rapidité et de niveau de jeu par rapport aux autres techniques de parallélisation [Cazenave et Jouandeau, 2007; Chaslot *et al.*, 2008].

Pour améliorer la performance de la parallélisation aux feuilles, Chaslot *et al.* [2008] tentent de stopper les threads restants si les résultats des premiers indiquent un coup peu intéressant, ce qui permet de relancer une recherche sur un autre nœud. Cependant cette modification ne permet pas un bon passage à l'échelle, beaucoup de temps étant perdu pour des *playouts* non menés à terme.

Pour son joueur GGP *Cadiaplayer*, Finnsson [2012b] utilise une parallélisation aux feuilles où chaque processus esclave lance plusieurs *playouts* à partir d'une même position pour améliorer l'équilibre entre le poids des communications et les temps de calcul.

Cazenave et Jouandeau [2008] tentent d'utiliser des communications étagées : chaque processus enfant reçoit une position à partir de laquelle il réalise une simulation et retourne le résultat dès qu'il est prêt. Le processus maître lui envoie alors une autre position à partir de laquelle travailler. Ainsi le processus maître n'attend pas de collecter les résultats renvoyés par tous les processus esclaves et chaque esclave peut travailler sur une position distincte. Cette technique testée sur le jeu de

Go permet un bon passage à l'échelle jusqu'à 16 esclaves.

Kato et Takeuchi [2010] utilisent la même approche et proposent un algorithme adapté à une architecture de type client-serveur permettant à des clients hétérogènes de se connecter à la volée¹³ au serveur pour participer au calcul des simulations.

Malgré ces améliorations, la technique de parallélisation aux feuilles reste moins performante que la parallélisation au niveau de l'arbre ou à la racine car elle souffre de limitations provoquées par le processus maître et par les communications [Soejima *et al.*, 2010].

1.5.4 Parallélisation avec arbre distribué

Sur un système distribué, seules les parallélisations aux feuilles ou à la racine étaient envisagées puisqu'aucun accès à une mémoire commune n'est possible pour stocker l'arbre. Cependant, la mémoire disponible sur les différentes machines n'est pas utilisée de manière optimale : dans la parallélisation aux feuilles, l'arbre occupe la mémoire du serveur et celle des clients est sous-exploitée, tandis qu'avec la parallélisation à la racine, un même arbre est dupliqué sur chaque client amenant à stocker des informations redondantes. De plus, avec la parallélisation à la racine sur un système distribué, le système de *virtual loss* ne peut pas être appliqué ce qui peut expliquer un passage à l'échelle limité [Yoshizoe *et al.*, 2011].

Graf *et al.* [2011]; Schaefers *et al.* [2011] proposent une solution en distribuant le stockage de l'arbre sur différents clusters (voir figure 1.7 (d)). Comme nous l'avons vu précédemment, MCTS peut être associé à l'utilisation d'une table de transposition (voir section 1.4.3) dans laquelle chaque nœud est stocké en l'associant à une clef de hashage pour y accéder plus rapidement. Graf *et al.* [2011] proposent d'utiliser cette clef de hashage pour choisir dans la mémoire de quel cluster le nœud devra être stocké.

Lors de la descente dans l'arbre jusqu'à la feuille la plus prometteuse, un cluster peut recevoir une requête de déplacement vers un nœud. Si ce nœud n'existe pas dans sa table de transposition, le cluster crée ce nouveau nœud et lance un *playout*. Dans le cas où le nœud existe dans la table, le cluster sélectionne un de ses nœuds enfants. Si ce nœud enfant est stocké localement, le cluster entame une nouvelle simulation, sinon il envoie une requête de déplacement vers le cluster chargé du stockage de ce nœud. Lorsqu'un *playout* retourne une évaluation pour un nœud, un message de mise à jour est envoyé à tous les clusters stockant les nœuds parents.

Chaque cluster possède un processeur d'entrée/sortie dédié aux communications avec les autres clusters. Les messages reçus par un cluster sont accumulés dans une queue et traités par l'un des processeurs de calcul dès qu'il se libère de la tâche précédente. Les échanges d'informations entre les clusters sont ainsi effectués en parallèle avec les simulations.

13. Cette fonctionnalité est essentiellement utilisée pour permettre au client de se reconnecter en cours de route en cas de perte de la connexion.

Pour limiter le nombre de messages, certains nœuds très souvent visités (la racine notamment) sont dupliqués et stockés dans le cache local de chaque cluster. Un paramètre contrôle la quantité de nœuds dupliqués sur les différents clusters : au pire, avec le partage de la totalité des nœuds, on peut obtenir l'équivalent d'une parallélisation à la racine avec synchronisation régulière. Un autre paramètre contrôle le nombre de simulations effectuées pour un nœud avant synchronisation. Ce paramètre permet de limiter les communications : au pire, en réduisant au minimum les communications, on peut obtenir l'équivalent de la parallélisation à la racine de type *single-run* avec une unique synchronisation à la fin.

Lors de la recherche, le nombre de requêtes de déplacement peut vite faire exploser les temps de communication et empêche un bon passage à l'échelle. Pour éviter de redescendre dans l'arbre depuis la racine à chaque ajout d'un nouveau nœud, Yoshizoe *et al.* [2011] améliorent la technique de parallélisation de Graf *et al.* [2011] en proposant d'effectuer une recherche en profondeur d'abord. Chaque fois qu'un *playout* est effectué et une évaluation remontée, l'évaluation d'un nœud parent est comparée à celle de ses frères. Si le nœud possède toujours la meilleure évaluation de la *fratrie*, l'évaluation n'est pas remontée davantage et la recherche suivante démarre à partir de ce nœud. Cette façon de procéder donne un résultat globalement équivalent à la recherche UCT habituelle mais permet d'économiser beaucoup de déplacements dans l'arbre et de communications pour mettre à jour l'évaluation des nœuds.

Yoshizoe *et al.* [2011] proposent également une modification de l'évaluation UCB pour ajouter le principe du *virtual loss*, ce qui permet d'éviter qu'un nœud ne soit trop visité suite aux mises à jour moins fréquentes de l'évaluation des nœuds :

$$UCB(i) = \frac{w_i}{s_i + d_i} + c \sqrt{\frac{\ln(t + d_T)}{s_i + d_i}} \quad (1.5)$$

Avec d_i le nombre de recherches en cours à partir du nœud i , s_i le nombre de simulations déjà effectuées, w_i le nombre de victoires obtenues, d_T le nombre de recherches en cours pour tous les nœuds apparentés, t le nombre total de visites déjà effectuées pour tous les nœuds enfants et c la constante équilibrant exploitation et exploration.

1.5.5 Parallélisation de Nested Monte Carlo

Cazenave et Jouandeau [2009] proposent une parallélisation de l'algorithme Nested Montecarlo [Cazenave, 2009]. Comme dans la technique de parallélisation à partir des feuilles, un processus maître construit l'arbre du jeu, tandis que les simulations sont réalisées par des processus esclaves. Le processus maître envoie des nœuds à explorer à des processus *médians*. Un processus *médian* fait une descente dans l'arbre et demande, à chaque niveau, aux *esclaves* de faire une simulation pour chaque nœud enfant. Ensuite le processus *médian* choisit le nœud qui a la meilleure évaluation pour passer au niveau suivant.

Avant l'envoi d'une tâche à un esclave, chaque processus *médian* communique la nature de cette tâche à un processus *répartiteur* qui doit lui indiquer en retour l'identifiant de l'esclave auquel envoyer le travail. Un algorithme *Round-Robin* est utilisé pour classer les travaux en fonction de leur taille estimée et les distribuer aux différents esclaves. Dans une autre version *LastMinute*, un esclave informe le *répartiteur* quand il n'a plus de travail à faire, le travail le plus long de la file d'attente lui est alors donné, sinon, si aucune tâche n'est en attente, l'identifiant de l'esclave est enregistré dans la liste des esclaves disponibles.

Cazenave et Jouandeau [2009] testent leur système sur un jeu de morpion solitaire disjoint et obtiennent une multiplication de la vitesse des calculs par 56 pour 64 processus esclaves. Leurs expérimentations montrent que la version *LastMinute* est plus efficace sur un cluster hétérogène. Leur programme utilisant une recherche de profondeur 4 arrive à atteindre un score égal au record mondial.

Dans toutes ces techniques de parallélisation, le passage à l'échelle peut être entravé par trois types de surcoût [Soejima *et al.*, 2010] :

- un *surcoût en calcul* quand une partie non prometteuse de l'arbre est développée ou qu'un excédent de simulations est effectué par différents processus en parallèle ;
- un *surcoût en synchronisation* quand le résultat des calculs de plusieurs threads est attendu par un processus maître ou quand le blocage d'un mutex génère une attente ;
- un *surcoût en communications* quand de nombreux échanges d'information sont nécessaires, dans une architecture distribuée notamment.

Cependant, Bourki *et al.* [2010] indiquent que ces difficultés ne sont pas les seules à empêcher le passage à l'échelle. Il signale qu'au jeu de Go, le passage à l'échelle de MCTS n'est pas constant. En faisant s'affronter deux joueurs identiques effectuant N et $N * 2$ simulations, la différence de niveau de jeu n'est pas aussi marquée entre eux quand la valeur de N augmente. Ceci est à prendre en compte lors de l'augmentation des simulations durant la parallélisation.

« As we will see, the limited efficiency also occurs in the sequential case - the trouble is not due to communication costs or overheads of the parallel implementation. » [Bourki *et al.*, 2010]

1.6 Les *Propositional Networks* (propnet)

Pendant l'exploration d'arbre, il est nécessaire d'utiliser un programme capable d'examiner les règles du jeu, de déterminer quelle sera la position du jeu après l'application d'un coup donné, de déterminer si cette nouvelle position est terminale, si oui quel est le score de chaque joueur, sinon établir la liste des coups légaux pour chacun d'eux, et tout ceci afin de construire l'arbre du jeu.

Les programmes utilisés pour interpréter les règles étaient jusqu'à présent généralement fondés sur le langage logique Prolog¹⁴. D'autres interprètes spécialisés existent comme JavaProver ou C++Reasoner, qui, comme leur nom l'indique, sont basés sur d'autres langages que Prolog mais Björnsson et Schiffel [2013] indique que les performances de ces interprètes sont bien moindres que celles des interprètes fondés sur Prolog.

Bien qu'ils aient été jusqu'à présent les plus efficaces, les langages logiques utilisant les principes d'unification, de récursivité et de retour sur trace (*backtracking*) restent assez lents et limitent l'exploration des arbres de jeux surtout pour les jeux dont les règles sont complexes et comportent beaucoup de variables.

Le concept de réseau de propositions nommé *Propositional Network* en anglais ou plus brièvement *propnet* a apporté une amélioration conséquente de la vitesse d'interprétation des règles et de ce fait le nombre de simulations de parties réalisables par seconde. Par exemple, pour le jeu de *Tictactoe*, la version de mai 2014 de *LeJoueur* peut réaliser 313216 *playouts* par seconde¹⁵ sur un Core I7 à 2,7GHz avec un *propnet* contre 756 seulement avec YAP Prolog. Les *propnets* sont issus de l'Automate Propositionnel, un framework inventé par le *Stanford Logic Group* [Stanford Logic Group, 2013] pour les systèmes multi-agents discrets et dynamiques tels que les jeux [Cox et al., 2009].

1.6.1 Le *tabling* et le *grounding*

Pour créer un *propnet*, une étape fondamentale est l'instanciation des règles GDL du jeu : les règles contenant des variables doivent être transformées en un ensemble de propositions logiques ne contenant que des constantes ; cette étape est désignée en anglais sous le terme *grounding*. Kissmann et Edelkamp [2010] suggèrent cette instanciation pour accélérer le fonctionnement des joueurs existants (utilisant un interprète Prolog). Vittaut et Mehat [2014] décrivent une technique utilisant Prolog pour instancier efficacement et rapidement les règles GDL dans le but de réaliser ensuite un *propnet*.

Les règles GDL originales subissent un ensemble de pré-traitements visant à les simplifier et à préparer leur instanciation. L'interprète prolog est ensuite utilisé pour unifier les variables avec toutes les valeurs possibles. Certains interprètes comme BProlog, XSB ou Yap proposent un système de mémorisation nommé *tabling*. Ce système permet de stocker les règles unifiées pour ne pas avoir à rechercher chaque fois les solutions possibles en examinant chaque règle. Le *tabling* est utilisé ici pour regrouper toutes les instanciations possibles des règles.

Voici par exemple une règle extraite du jeu de *Tictactoe*, indiquant qu'il est légal

14. *Ary* et *LeJoueur* utilisent YAP Prolog, développé par l'université de Porto, qui, en plus d'être open-source, multi-plateforme et gratuit d'utilisation dans un environnement académique, présente l'avantage d'être une implémentation de Prolog particulièrement rapide.

15. *LeJoueur* réalise en fait 4894 simulations par seconde en codant chaque fait logique sur 64 bits ce qui permet de faire 64 *playouts* distincts en parallèle donc $4894 * 64$ *playouts* par seconde.

pour un joueur w de mettre une marque dans une case (x, y) si c'est à son tour de jouer et si la case est initialement vide (b) :

```
(<= (legal ?w (mark ?x ?y)) (true (control ?w)) (true (cell ?x ?y b)))
```

La règle règle traduite en prolog se présente ainsi :

```
legal(P, mark(X,Y)) :- true(control(P)), true(cell(X,Y,b)).
```

Une fois instanciée, cette règle donne naissance à différentes propositions logiques en fonction des valeurs possibles des variables :

```
legal(xplayer, mark(1,1)) :- true(control(xplayer)), true(cell(1,1,b)).
legal(xplayer, mark(1,2)) :- true(control(xplayer)), true(cell(1,2,b)).
legal(xplayer, mark(1,3)) :- true(control(xplayer)), true(cell(1,3,b)).
...
legal(oplayer, mark(1,1)) :- true(control(oplayer)), true(cell(1,1,b)).
legal(oplayer, mark(1,2)) :- true(control(oplayer)), true(cell(1,2,b)).
legal(oplayer, mark(1,3)) :- true(control(oplayer)), true(cell(1,3,b)).
...
```

Suite au *grounding*, les règles GDL donnent naissance à de nombreuses expressions logiques. Pour des jeux complexes, le temps et la mémoire nécessaires à l'instanciation des règles peuvent être conséquents et dépasser le temps alloué dans le cadre des compétitions ou la mémoire disponible. Jean-Noël Vittaut indique¹⁶ que son joueur est capable, pour la majorité des jeux disponibles dans les dépôts [Schreiber, 2014; Stanford Logic Group, 2013], de réaliser le *grounding* dans un temps compatible avec les conditions de la compétition. Cependant, comme dans le cadre du GGP toutes sortes de jeux peuvent être proposées, le *grounding* des règles d'un jeu peut ne pas être réalisable dans un temps raisonnable. L'utilisation d'un programme mixte capable d'utiliser un interprète Prolog classique pour raisonner sur le jeu en cas d'échec du *grounding* (et de la création du *propnet*) est donc nécessaire dans le cadre des compétitions de GGP.

1.6.2 Principe du *propnet*

A partir des règles instanciées, un réseau logique de propositions *propnet* est construit sous forme d'un graphe (voir figure 1.8). Chaque fait nommé *fluent* qui peut être vrai ou faux à un instant donné du jeu constitue un nœud dans le graphe. A ces nœuds s'ajoutent des connecteurs ET, OU, NON qui permettent d'incarner la logique du jeu et un connecteur de transition simulant la dynamique du jeu : le connecteur de transition recopie son entrée sur sa sortie à chaque itération et permet ainsi de simuler le passage d'une position du jeu à une autre. Le *Stanford Logic*

16. Conversation du 26 juin 2014

Group [Genesereth et Thielscher, 2014] indique qu'un ensemble N de faits décrivant un jeu peut permettre de décrire 2^N états différents, ce qui permet au *propnet* de représenter la dynamique d'un jeu d'une manière plus économique qu'un automate à états.

Les fluents représentés par les nœuds du graphe sont divisés en trois groupes : les *inputs* qui représentent les actions possibles des joueurs (nœuds n'ayant pas d'entrée), les *bases* qui correspondent aux faits décrivant la position courante (possédant un arc entrant en provenance d'un connecteur de transition), et les *views* qui regroupent tous les autres faits décrivant la situation, par exemple les scores des joueurs ou l'indicateur de fin de partie.

Au début de la partie, les *bases* sont initialisés d'après les informations données dans la description GDL du jeu. En propageant le signal dans le graphe, il est possible de savoir si le nœud terminal prend la valeur *vrai* et donc si la partie est achevée. Dans ce cas, les nœuds, correspondant aux goals des différents joueurs, qui ont pris une valeur vraie donnent le score. Si la partie n'est pas terminée, les *views* correspondant aux coups légaux ayant une valeur vraie indiquent les actions possibles. Une action peut alors être choisie et l'*input* correspondant est basculé à *vrai*. Les connecteurs de transition sont alors activés pour recopier leur valeur d'entrée sur leur sortie et faire passer le *propnet* à l'état suivant.

1.6.3 Implémentation d'un *propnet* dans *LeJoueur* (version 02/2014)

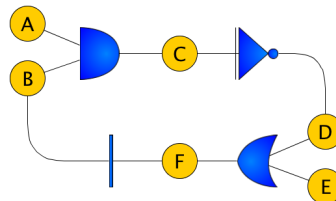
Jean-Noël Vittaut a développé en C++ un joueur GPP nommé *LeJoueur*. Son programme, positionné à la 7ème place sur 16 à la compétition de 2013 (734 points contre 1098 pour le vainqueur TurboTurtle) [Méhat, 2013], utilise un objet *Grounder* implémentant la technique de *grounding* qu'il décrit dans son article *Fast Instantiation of GGP Game Description* [Vittaut et Mehat, 2014]. Il utilise ce *Grounder* comme base pour générer un *propnet*.

La description du *propnet* par le *Stanford Logic Group* (voir section 1.6.2) montre que le calcul des coups légaux ou de la position suivante du jeu par exemple sont réalisés par la propagation de signaux dans le graphe de la même manière que dans un circuit électrique. L'implémentation naïve d'un tel graphe et de la propagation des signaux amènerait un temps de calcul important, il est donc nécessaire de s'éloigner un peu de ce modèle.

Dans la version de février 2014 de *LeJoueur*, pour implémenter ce *propnet*, Jean-Noël Vittaut le décompose en deux structures de données. La première est une table de valeurs qui contient les valeurs booléennes de tous les fluents intervenant dans la dynamique du jeu (les nœuds du graphe), la seconde représente l'ensemble des règles logiques à appliquer (les liens dans le graphe) et est constituée de différents *Circuits*¹⁷.

17. *LeJoueur* est en constante évolution, Jean-Noël Vittaut apportant à son programme de fréquentes améliorations, les éléments décrits ici sont donc spécifiques à la version de février 2014. La version de mai 2014 utilise encore deux structures pour stocker les valeurs booléennes des faits

Exemple de *propnet* : les fluents A et E sont des actions possibles (*input*), le fluent B décrit la position courante (*base*), C, D et F sont déduits d'après la valeur des faits précédents (*view*).



Dans les figures suivantes, les valeurs 1 et 0 indiquent respectivement les faits vrais et faux.

1	Position initiale du jeu : la valeur des <i>bases</i> est initialisée, ici B est un fait vrai en début de partie.	
2	Indication du coup joué : la valeur des <i>inputs</i> est modifiée pour indiquer le coup joué par chacun des joueurs (dans un jeu alterné, une seule véritable action est effectuée par l'un des joueurs, les autres jouant <i>noop</i>), ici l'action A est vraie et l'action E est fausse.	
3	Propagation du signal dans le <i>propnet</i> : les autres faits (<i>views</i>) sont déduits de la valeur des <i>bases</i> et des <i>inputs</i> .	
4	Itération/position suivante : les connecteurs de transition modifient la valeur des <i>bases</i> et indiquent ainsi l'état du jeu à la position suivante. Ici, le fait B devient faux. La partie se poursuit en indiquant une nouvelle action des joueurs (étape 2).	

FIGURE 1.8 – Fonctionnement d'un *propnet* : schéma inspiré de [Genesereth et Thielscher, 2014] chapitre 9.

La table des valeurs

Lors de l’instanciation des règles GDL grâce à son *Grounder*, tous les fluents intervenant dans la description du jeu sont extraits et stockés, dans leur représentation GDL, dans un dictionnaire (map) :

```
(legal xplayer (mark 1 1))  
(legal xplayer (mark 1 2))  
...  
(true (control xplayer))  
(true control oplayer)  
(true (cell 1 1 b))  
(true (cell 1 2 b))  
(true (cell 1 3 b))
```

Conserver la représentation GDL des fluents permet par exemple de communiquer au Game Master l’action choisie à chaque coup, ou bien de récupérer la valeur numérique du score dans un fait de type *goal* (exemple : (goal playerx 50)).

Jean-Noël Vittaut crée une table de pointeurs sur chacun de ces fluents dans un ordre précis dont nous verrons la raison d’être un peu plus loin (voir figure 1.9). Les pointeurs sur les faits décrivant la position courante (commençant par *true*) sont énumérés en premier, suivis des pointeurs sur les faits décrivant la position suivante (commençant par *next*) dans un ordre correspondant. Par exemple, si le fluent (true (control xplayer)) est le premier fait de la liste des *true*s, le fluent correspondant (next (control xplayer)) sera le premier de la liste des *next*s. Ensuite vient un pointeur sur l’unique fait *terminal*, les pointeurs sur les faits décrivant les coups légaux (commençant par *legal*) du premier joueur puis des autres dans l’ordre et les pointeurs sur les coups effectivement joués (commençant par *does*) dans un ordre correspondant aux coups légaux associés. Enfin viennent les pointeurs sur les fluents décrivant le score (commençant par *goal*) et les autres faits propres au jeu décrit.

A cette table de pointeurs Jean-Noël Vittaut associe une table de booléens de longueur égale où chaque booléen correspond au pointeur de même index et indique la valeur vraie ou fausse du fluent correspondant à un moment donné du jeu (voir figure 1.9). L’ordonnancement des faits permet de recopier très rapidement la liste des booléens décrivant la position suivante (portion de la table correspondant aux *next*s) sur la zone mémoire correspondant à la position courante (aux *true*s) et de simuler ainsi le connecteur de transition que nous avons vu dans le *propnet* décrit par le *Stanford Logic Group* (voir figure 1.8). Cet ordonnancement permet également de consulter rapidement les scores ou la liste des coups légaux. Pour jouer le coup correspondant à un coup légal donné, il suffit de connaître l’index de ce coup par rapport au premier coup légal de la table et de basculer à *vrai* la valeur située à l’index correspondant à la même position dans le segment des *does*.

et les règles du jeu, mais le codage des faits et des règles a été amélioré pour économiser l’espace

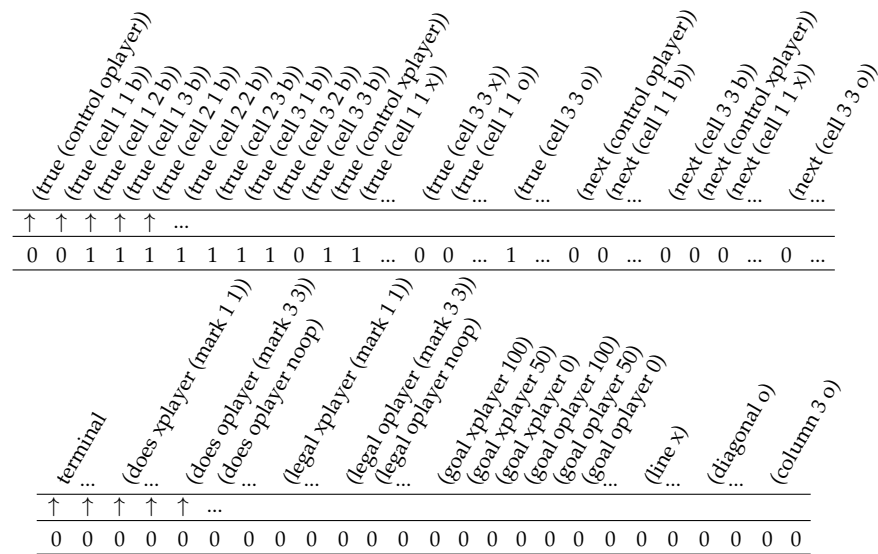


FIGURE 1.9 – Cette figure représente les structures de données utilisées pour gérer les valeurs des fluents durant le jeu et pour récupérer la représentation GDL de ces faits lorsque, par exemple, une communication avec le *Game Master* est nécessaire. Les fluents (couche du haut) sont stockés dans un dictionnaire (map). Une table de pointeur (couche du milieu) permet d’y accéder dans un ordre précis (*trues*, *nexts*, *terminal*, *does*, *legals*, *goals* et *rest*). Une table de booléens (couche du bas) permet de stocker la valeur de chaque fait désigné par le pointeur de même index.

Les circuits

Pendant la phase d’instanciation des règles, toutes les règles GDL sont transformées sous forme normale disjonctive puis traduites en listes de conjonctions. En Prolog, la disjonction est implicite et représentée par une succession de règles. Les règles de même conséquent sont regroupées ensemble. Comme une dépendance peut apparaître entre les règles, i.e. le conséquent d’une règle peut intervenir dans le corps d’une autre règle, Jean-Noël Vittaut regroupe les règles par *strates*, chaque *strate* devant être évaluée complètement avant la suivante. Chacune de ces *strates* est implémentée par un objet *Circuit*.

La liste des règles obtenue grâce au *Grounder* est donc au final un ensemble de listes de conjonctions simples regroupées en circuits. Pour pouvoir calculer séparément les faits *terminal*, *legal*, *goal* et *next*, la liste est scindée en quatre parties distinctes constituées chacune de un ou plusieurs *Circuits*. A partir de la table de valeurs et des tables de pointeurs et de faits GDL pointés, Jean-Noël Vittaut traduit toutes les règlesinstanciées du jeu en listes de pointeurs sur les valeurs booléennes correspondantes (voir figure 1.10). Les règles peuvent utiliser la négation de certains faits ; plutôt que de doubler la liste des valeurs pour stocker les faits et leur contraire, Jean-Noël Vittaut tire parti du fait que les pointeurs adressent la mémoire par pas de 8 bits : la valeur des pointeurs est toujours un multiple de 8. En ajoutant 1 à la valeur d’un pointeur désignant un fluent dans une règle, il code économiquement la négation de ce fait : cette approche permet de minimiser la mémoire utilisée pour

mémoire et accélérer le calcul des règles.

stocker le *propnet*.

1.7 Conclusion

Les meilleurs programmes de *General Game Playing* actuels utilisent pour la plupart la technique Monte Carlo Search Tree (MCTS) avec les diverses améliorations décrites par Cameron Browne [Browne *et al.*, 2012]. La qualité des coups choisis par les algorithmes MCTS augmentant avec la quantité de simulations effectuées, différentes techniques de parallélisation au niveau de l'arbre, de la racine ou des feuilles ont été expérimentées. En 2011, Méhat et Cazenave affirmaient :

« *As the Tree parallelization of Monte-Carlo Tree Search works well when playouts are slow, it is of interest for General Game Playing programs, as the interpretation of game description takes a large proportion of the computing time, when compared with program designed to play specific games.* » [Méhat et Cazenave, 2011b]

Avec l'utilisation des *propnets*, la proportion de temps utilisée pour l'interprétation de la description des jeux a considérablement diminué, les *playouts* sont donc bien plus rapides. Le coût des temps de synchronisation et de communication étant proportionnellement plus important, l'apport de la parallélisation pour un joueur GGP utilisant un *propnet* nécessite d'être ré-exploré.

Les architectures multi-cœurs et multi-threads se sont considérablement développées ces dernières années : la société Kalray a commercialisé en 2013¹⁸ une machine pourvue d'un processeur MPPA (Multi Purpose Processor Array) à 256 cœurs. Dans le chapitre suivant nous explorons la parallélisation de *LeJoueur* de Jean-Noël Vittaut sur cette architecture spécifique.

18. La première mention du MPPA-256 dans les informations de la société Kalray <http://www.kalray.eu/news-7/> date de 2011. La commercialisation et l'accès au public du MPPA-256 a été annoncée à la conférence HiPEAC de Berlin le 21/01/2013.

```

– TERMINAL –
– circuit 0 –
214 :- 2.           open :- (true (cell 1 1 b)).
214 :- 4.           open :- (true (cell 1 2 b)).
214 :- 6.
(...)
214 :- 18.
222 :- 22, 24, 26.   (row 1 x) :- (true (cell 1 1 x)), (true (cell 1 2 x)), (true (cell 1 3 x))
224 :- 28, 30, 32.
(...)
258 :- 44, 50, 56.
– circuit 1 –
210 :- 222.          (line x) :- (row 1 x).
210 :- 224.
(...)
212 :- 238.
– circuit 2 –
116 :- 210.
116 :- 212.
116 :- 215.          terminal :- (not (open)).
– LEGAL –
(...)
– GOAL –
(...)
– NEXT –
(...)

```

FIGURE 1.10 – Cette figure représente la manière dont les règles sont codées sous forme de *Circuits* dans *LeJoueur* de Jean-Noël Vittaut. Ici pour rendre la figure plus lisible, les pointeurs ont été remplacés par les index des valeurs booléennes correspondantes aux faits, multipliés par 2. Les index impairs permettent de coder les faits négatifs : par exemple le fluent *open* est codé par 214 (fluent d'index 107) et *(not open)* est codé par 215. Quatre ensembles de règles permettent de calculer les faits *terminal*, *legals*, *goals* et *nexts*. Chaque ensemble est découpé en *strates* permettant de gérer les dépendances : on voit par exemple que la valeur du fluent *(row 1 x)* calculée dans le circuit 0 contribue ensuite à déterminer la valeur du fluent *(line x)* dans le circuit 1.

2 Parallélisation de *LeJoueur*

Dans ce chapitre nous commençons par présenter l'architecture MPPA sur laquelle nous avons réalisé la parallélisation de *LeJoueur*. Nous expliquons ensuite les options de parallélisation possibles ou impossibles à implémenter en fonction des contraintes imposées par le MPPA. Nous justifions notre choix d'utiliser des communications synchrones et présentons la structure de notre implémentation d'une *parallélisation aux feuilles*. Nous décrivons ensuite différentes expérimentations et leurs résultats. La première a pour but de déterminer s'il est préférable de partager le calcul des *playouts* entre différents processeurs élémentaires (PE) du MPPA ou d'utiliser chaque PE pour réaliser un *playout* distinct. La seconde permet de comparer différentes approches pour la gestion des threads et de déterminer l'approche la plus efficace entre relancer un thread à chaque *playout* ou garder le thread actif et lui communiquer les *playouts* à calculer. La troisième vise à évaluer la proportion de temps consacrée aux communications face aux temps de calcul des *playouts*. La quatrième est une étude de la variation des temps de communication en fonction de la taille des données à transmettre. Enfin, nous exposons le problème posé par la taille du *prophet* à transmettre au MPPA.

2.1 Architecture utilisée (MPPA-256)

La machine MPPA-256 que nous avons utilisée pour paralléliser *LeJoueur* de Jean-Noël Vittaut est un serveur équipé d'un processeur Intel core I7 à 3.6GHz tournant sous Linux OS et d'une carte PCIe Application Board AB01 équipée d'une puce MPPA-256 (voir figure 2.1) [Kalray, 2013a]. Cette machine développée par la société Kalray est commercialisée depuis 2013. La société Kalray (Essonne, France), créée en 2008, conçoit des semi-conducteurs et des logiciels. Elle développe et vend cette nouvelle génération de processeurs multi-cœurs pour l'imagerie, les infrastructures de télécommunications, les applications de calcul intensif et les applications embarquées de sécurité des données et réseau.

Nous commençons par présenter les caractéristiques de la puce MPPA-256 puis la structure d'un programme développé pour cette architecture. Nous décrivons ensuite les différents types de communications disponibles avec le MPPA-256 et concluons par une analyse critique de cette architecture.



FIGURE 2.1 – Carte MPPA, image tirée de Kalray [2013a]

2.1.1 Le Multi Purpose Processor Array 256

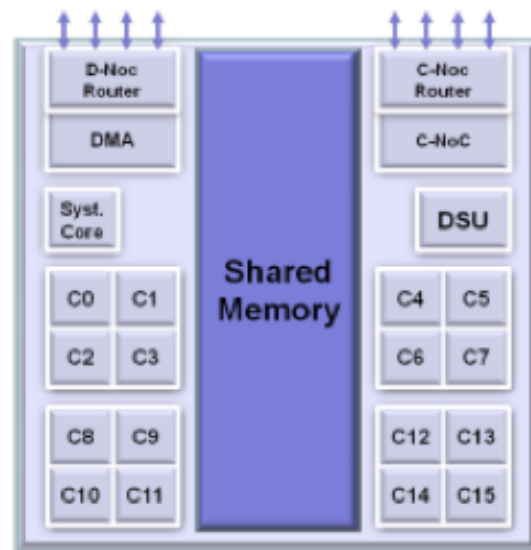
Le Multi Purpose Processor Array (MPPA-256) est un processeur multi-cœurs, premier membre de la famille *MPPA MANYCORE* ; il comprend 256 processeurs de calcul, les autres MPPA pouvant aller jusqu'à 1024 processeurs en une seule puce silicium. Ils sont conçus pour fournir une importante puissance de calcul pour une faible consommation énergétique : le MPPA-256 consomme moins de 10 W¹ (un Intel Core i7 consomme typiquement 130 W).

Le MPPA-256 est composé d'une grille de 16 clusters connectés à travers un réseau haute vitesse (Network-on-Chip, NoC). Chaque cluster (voir figure 2.2) contient 16 processeurs élémentaires utilisant une architecture VLIW (Very Long Instruction Word) et un processeur système exécutant un système d'exploitation spécifique (NodeOS). Ce processeur système a pour rôle de lancer le programme dédié au cluster et de gérer les communications, ce qui laisse chaque processeur élémentaire totalement disponible pour exécuter le calcul lui-même (voir figure 2.3). Chaque cluster dispose d'une mémoire partagée de 2Mo. Le MPPA-256 est une architecture MIMD (Multiple Instruction streams, Multiple Data) avec une mémoire distribuée accessible uniquement localement, i.e. chaque cluster est encapsulé, il n'accède qu'à sa propre mémoire et peut exécuter un code distinct des autres clusters.

Quatre interfaces d'entrées/sorties (I/O) permettent de gérer les communications entre les clusters et la machine hôte pour deux d'entre eux, entre les clusters et le réseau Ethernet pour les deux autres. Cependant la version 1.01² des outils logiciels permettant l'exploitation de la carte ne permet d'utiliser qu'une seule interface d'I/O comme intermédiaire entre la machine hôte et les clusters de calcul (voir figure 2.4). Cette interface d'entrées/sorties possède un processeur quadri-cœur SMP

1. Selon l'affirmation de Joël Monier, PDG de Kalray, dans une interview donnée à silicon.fr le 26 août 2013 : <http://www.silicon.fr/joel-monier-kalray-a-decide-de-reinventer-le-processeur-88705.html?ModPagespeed=noscript>

2. Nous avons utilisé la version 1.0.1 [Kalray, 2013b] pendant la majeure partie de notre travail. Cette version présentant un défaut se manifestant par un blocage inopiné de l'application au lancement des binaires sur la carte MPPA, nous avons dû passer à la version 1.2.1 [Kalray, 2014] pour nos dernières expérimentations. La version 1.2.1 permet l'accès à une deuxième interface d'I/O mais nous n'avons pas utilisé cette fonctionnalité.



- 16 PE cores + 1 RM core
- NoC Tx and Rx interfaces
- Debug Support Unit (DSU)
- 2 MB of shared memory

FIGURE 2.2 – Un cluster de la carte MPPA-256, schéma tiré de Kalray [2013a]

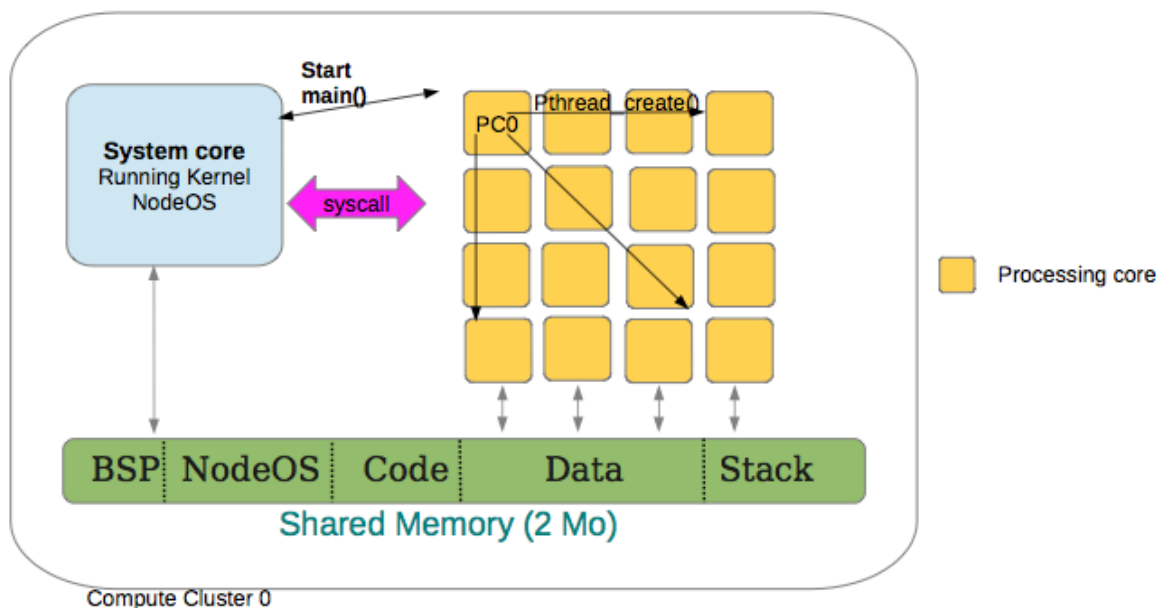


FIGURE 2.3 – Exécution d'un programme sur un cluster, schéma extrait de Kalray [2013a]

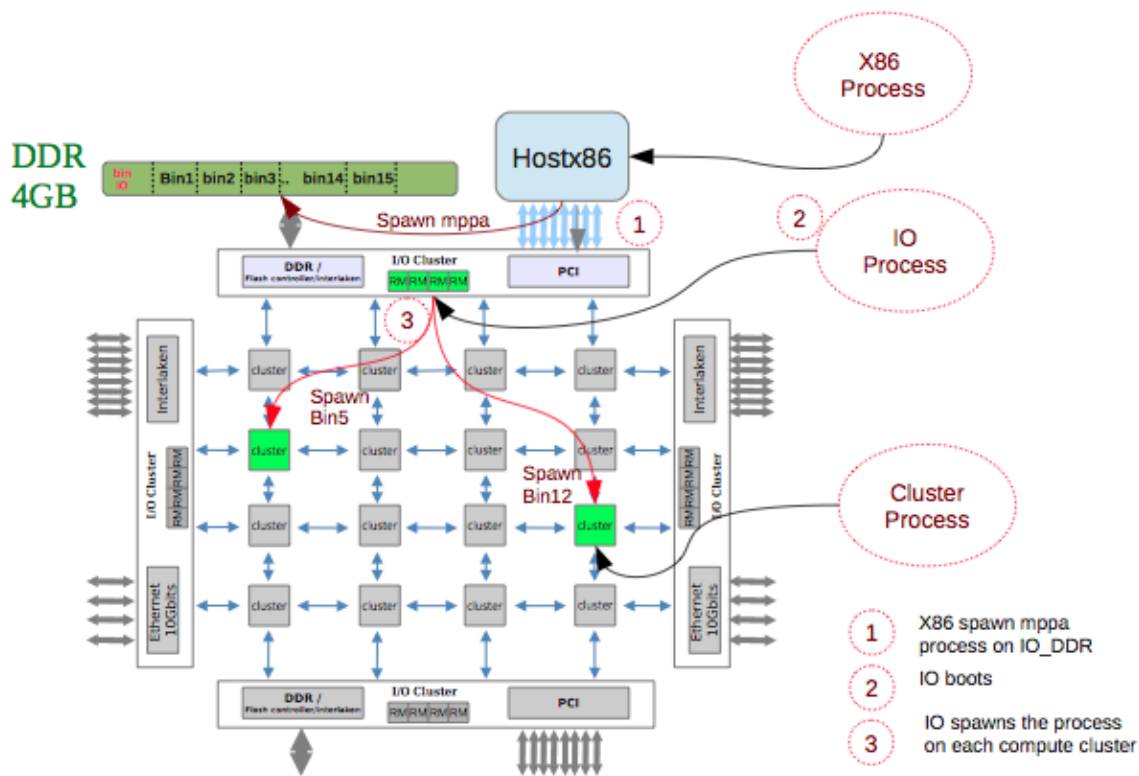


FIGURE 2.4 – Schéma du serveur équipé d'une carte MPPA-256 tiré de Kalray [2013b], trois programmes distincts sont déployés sur l'hôte, l'I/O et les clusters.

avec une mémoire DDR3 de 4Go et une interface PCIe Gen3 pour les échanges de données avec la machine hôte.

2.1.2 Structure d'un programme pour le MPPA

Le développement d'un programme pour le MPPA nécessite la création de trois exécutables pour chacune des 3 parties : l'hôte, l'I/O et les clusters. Chaque cluster peut exécuter un programme distinct, le nombre total d'exécutables peut donc s'élever à 18.

L'exécutable destiné à l'hôte est lancé par l'utilisateur. Une instruction `mppa_load` à l'intérieur de ce programme permet de charger un fichier multi-binaire contenant les exécutables destinés à l'interface d'I/O et aux clusters dans la mémoire de la carte MPPA. L'instruction `mppa_spawn` permet ensuite de lancer l'exécutable destiné à l'I/O. Le programme de l'I/O se charge à son tour avec la même instruction `mppa_spawn` de lancer l'exécutable voulu sur le ou les clusters choisis. Un même programme peut être exécuté sur chacun des 16 clusters en parallèle.

Kalray fournit différents outils logiciels pour le MPPA regroupés sous le nom de *MPPA ACCESSCORE*. Cet ensemble logiciel comprend entre autre un environnement de développement C/C++, des outils destinés aux tests, traçage et au débogage, un système d'exploitation spécifique pour l'I/O nommé *RTEMS* et pour les

clusters nommé *NodeOS*, les outils de compilation des sources et un environnement de programmation POSIX [Kalray, 2013c].

Le *Makefile* [Kalray, 2013d] créé par l'utilisateur consiste en une déclaration du nom de chaque exécutable à compiler avec la liste de ses sources. Un ou plusieurs fichiers multi-binaires sont définis en déclarant la liste des executables qui les constituent, destinés à l'I/O et aux clusters. Des flags indiquant les directives de compilation et les bibliothèques à lier peuvent être déclarés pour les différents niveaux : pour tous les éléments à compiler, uniquement pour l'hôte, pour tous les executables destinés à la carte MPPA, uniquement pour l'I/O, uniquement pour les clusters, ou pour un seul exécutable voire même un seul fichier source particulier. Les plateformes pour lesquelles la compilation doit être réalisée sont déclarées par l'utilisateur : *native* pour une émulation native (émulation x86 MPPAIPC), *simulator* pour une exécution sur le simulateur, *hardware* pour une exécution sur la carte MPPA. Des directives peuvent être définies pour effectuer divers traitements avant et après la compilation. Les commandes de compilation pour chaque OS sont regroupées dans un fichier contenu dans le *MPPA ACCESSCORE* et inclus dans le *Makefile* de l'utilisateur par la commande :

```
include $(K1_TOOLCHAIN_DIR)/share/make/Makefile.mppaipc3.
```

L'utilisateur peut finalement déclarer différentes cibles pour compiler et lancer son ou ses programmes sur les différentes plateformes mais aussi pour ajouter des informations destinées à l'utilisateur ou pour effectuer un nettoyage spécifique de certains fichiers. La compilation peut être réalisée pour lancer un programme destiné uniquement aux clusters, un programme utilisant l'I/O et les clusters ou un programme complet pour les 3 parties : hôte, I/O et clusters. Ces programmes peuvent être exécutés sur l'architecture MPPA ou bien sur différents simulateurs (simulateur complet, simulateur de clusters avec IO ou simulateur d'un seul cluster) tournant sur la machine hôte.

Le programme hôte peut récupérer des arguments spécifiques dans la table `argv[]` passée en argument du `main()` et il peut lui-même transmettre des arguments au `main()` de l'I/O en les passant à la commande `mppa_spawn`. L'I/O peut à son tour transmettre des arguments lors du lancement des executables des clusters. Ces arguments sont généralement la description des canaux de communications ou autres informations nécessaires pour la coordination des 3 parties : hôte, I/O et clusters.

Le programme destiné à un cluster est en fait exécuté sur le processeur élémentaire PE0. Ce programme peut lancer des threads, chacun d'eux est alors automatiquement lancé sur un des 15 autres processeurs élémentaires. Pour pouvoir lancer un thread, il est nécessaire que l'un des PE soit disponible, il est donc impossible d'exécuter plus de 15 threads en même temps sur un cluster.

3. `$(K1_TOOLCHAIN_DIR)` contient le chemin d'accès vers le dossier d'installation des outils MPPA ACCESSCORE.

Type	Pathname	Tx :Rx
Buffer	/mppa/buffer/rx_node#number/tx_node#number	1 :1
MQueue	/mppa/mqueue/rx_node#number/tx_node#number/mcount.tsize	1 :1

FIGURE 2.5 – Canaux de communication entre l’hôte et l’I/O. *rx_node* et *tx_node* indiquent les identifiants des nœuds lecteurs et écrivains : *hote*, ou *pcie0* et *pcie1* pour les clusters d’I/O. *number* est un entier destiné à désambiguïser les connecteurs. *mcount* et *tsize* permettent respectivement d’indiquer la taille maximale de la queue (en octets) et le nombre maximal de messages stockés dans la queue. [Kalray, 2014]

Type	Pathname	Tx :Rx
Sync	/mppa/sync/rx_nodes :cnoc_tag	N :M
Portal	/mppa/portal/rx_nodes :dnoc_tag	N :M
RQueue	/mppa/rqueue/rx_node :dnoc_tag/tx_nodes :cnoc_tag/credits.msize	N :1
Channel	/mppa/channel/rx_node :dnoc_tag/tx_node :cnoc_tag	1 :1
Sampler	/mppa/sampler/rx_node :dnoc_tag/tx_node :dnoc_tag	1 :M

FIGURE 2.6 – Canaux de communication de type NoC entre l’I/O et les clusters ou entre les clusters. *rx_node* et *tx_node* indiquent les identifiants des nœuds lecteurs et écrivains : 128 pour l’I/O ou un entier entre 0 et 15 pour les clusters. Un groupe de N ou M clusters peut être indiqué par une expression du type *[first..last]* ou *[first,second,third]*. *cnoc_tag* est compris dans l’intervalle [1, 127] et *dnoc_tag* dans l’intervalle [1, 255], les *cnoc_tag* et *dnoc_tag* à 0 sont réservés pour la coordination des systèmes d’exploitation de l’I/O et des clusters. [Kalray, 2014]

2.1.3 Les communications

Pour le transfert des données entre les trois parties (hôte, I/O et clusters), différents modes de communication sont disponibles⁴. Entre l’hôte et l’I/O des communications de type *buffer* ou *mqueue* sont utilisées. Entre l’I/O et les clusters, ou entre les clusters, quatre types sont disponibles : communication par *crédits* avec la queue, communication par échantillonnage avec le *sampler*⁵, communication avec *rendez-vous* pour le *channel* ou communication de grande taille utilisant des *offsets* pour le *portal*. Un type de communication spécifique *sync* est utilisé pour transmettre les messages de synchronisation.

Chaque connecteur pour les communications est décrit par une chaîne de caractères (voir figures 2.5 et 2.6). La fonction système *mppa_open* permet d’ouvrir un canal de communication en lui indiquant le connecteur souhaité et un sens unique de communication : lecture ou écriture. C’est la chaîne de caractères décrivant le connecteur qui doit être transmise en argument lors du lancement d’un binaire avec *mppa_spawn* : chaque processus se charge d’appeler *mppa_open* pour obtenir un descripteur de fichier correspondant à l’une des deux extrémités d’un canal de communication.

4. Les modes de communication décrits ici correspondent à la version 1.2.1 de MPPA ACCESSCORE. Par rapport à la version 1.0.1, un nouveau type de communication *sampler* est ajouté et l’usage des connecteurs existants est légèrement modifié, notamment les descripteurs des canaux de communication sont uniformisés.

5. Ce connecteur n’est pas disponible dans la version 1.0.1 de MPPA ACCESSCORE ; c’est une nouveauté de la version 1.2.1 [Kalray, 2014].

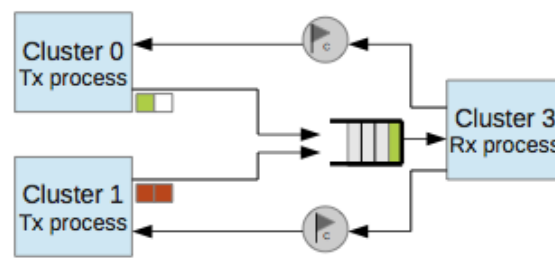


FIGURE 2.7 – Communication par queue entre 3 clusters. Le cluster 3 donne un crédit de 2 messages aux clusters 0 et 1. Quand un cluster a envoyé ses 2 messages, il doit attendre que l'un d'eux soit consommé par le cluster 3 pour pouvoir en envoyer un autre. Image tirée de Kalray [2013a].

Dans une communication entre l'hôte et l'I/O utilisant un connecteur de type buffer, le nœud écrivain cible une zone mémoire du nœud lecteur pour y inscrire les données. A la fin d'une écriture, il envoie une notification. De son côté, le nœud lecteur se prépare à la lecture puis commence à lire les données jusqu'à ce que la notification indique la fin de l'opération. Le processus écrivain reste bloqué jusqu'à ce que les données soient lues.

Le connecteur `mqueue` est utilisé uniquement entre l'hôte et l'I/O. La queue est caractérisé par une taille maximale *tsize*, en octets, et un nombre maximal de messages *mcount* qui peuvent être stockés. La queue peut donc contenir un seul message de taille *tsize*, deux messages de taille *tsize/2* ou *mcount* messages de taille *tsize/mcount*. Ce connecteur permet une communication asynchrone, les messages du nœud écrivain pouvant être consommés à une vitesse variable. Si le nombre maximal de messages est atteint ou si la queue est pleine, l'écriture est retentée à intervalles réguliers.

Un connecteur `rqueue`⁶ permet une communication du même type entre un lecteur (I/O ou cluster) et un ou plusieurs écrivains (I/O ou/et clusters). Cette fois la queue est caractérisée uniquement par la taille d'un message fixée par *msize* et limitée à 120 octets maximum. Un nombre fixé de *credits* est attribué aux écrivains. La figure 2.7 illustre le fonctionnement de la queue. Chaque écrivain disposant de *N* *credits* peut envoyer *N* messages dans la queue. Quand un message est consommé, l'écrivain récupère un *credit* pour envoyer un nouveau message.

Le connecteur de type `sampler` permet une communication par échantillonnage entre un seul écrivain et plusieurs lecteurs. Une zone mémoire dans le processus lecteur permet d'accumuler des échantillons de taille fixe.

Une communication utilisant un `channel` permet un *rendez-vous*, un échange de données entre un seul écrivain et un seul lecteur. Un message possédant la taille minimale requise par la lecture ou l'écriture est transmis sans copie intermédiaire. La figure 2.8 illustre le fonctionnement de ce type de communication. Le processus écrivain envoie un message terminé par EOT (End Of Transmission). Le processus lecteur attend une quantité de données *N*. Si EOF est reçu la communication prend

6. Ce connecteur est nommé simplement `queue` dans la version 1.0.1 de *MPPA ACCESSCORE*. Dans la version 1.2.1 il a été renommé `rqueue` pour éviter toute confusion avec `mqueue`.

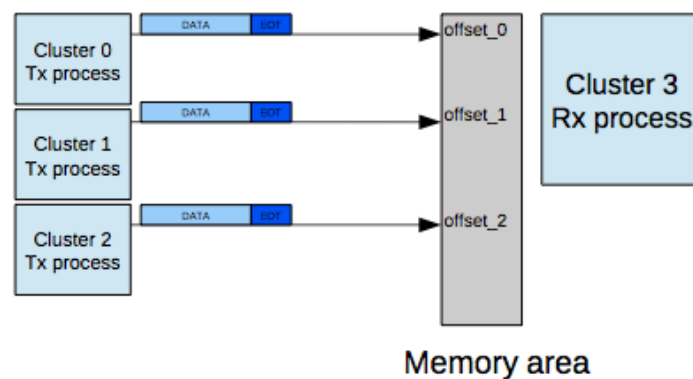


FIGURE 2.9 – Communication par portal. Les trois clusters 0 à 2 envoient chacun un paquet de données. Ces données sont regroupées dans une zone mémoire du cluster 3. Image tirée de Kalray [2013c].

possible doit être défini.

Si l'I/O veut envoyer des données aux clusters, il n'est pas possible de s'assurer que les clusters sont prêts à lire quand l'envoi des données commence. Dans ce cas, il est nécessaire d'utiliser un canal de communication de type sync. L'I/O définit une clef et attend les signaux indiquant que tous les clusters sont prêts à lire les données avant de commencer à écrire dans le portal.

2.1.4 Analyse critique du MPPA

Le discours commercial de la société Kalray met en avant la faible consommation énergétique du MPPA-256 ainsi que la facilité de développement d'applications C/C++ dédiées à cette architecture *MPPA MANYCORE* grâce à la chaîne de compilation *MPPA ACCESSCORE*. Cependant, peu d'informations sont disponibles sur sa performance de calcul.

Un rapport technique de Jouandeau [2013], indique que « les capacités de calcul d'un processeur Intel i7 3820 avec 8 cœurs et d'un processeur MPPA avec 256 cœurs sont proches ». Son estimation est fondée sur les performances constatées pour un cœur sur la résolution du problème du verre de spin et multipliée par le nombre de cœurs, ce qui suppose des algorithmes parallèles avec des temps de communication nuls. Cette estimation laisse supposer donc que le MPPA n'offre pas des capacités de calcul vraiment intéressantes et que peu de gain en rapidité d'exécution est à espérer par rapport à une application séquentielle exécutée sur l'Intel core I7 de la machine hôte.

La limitation de la mémoire dans les clusters à 2Mo représente également une grosse contrainte pour le développement d'applications destinées au MPPA. Dans notre cas, nous verrons dans la section suivante que cela diminue radicalement les choix possibles de parallélisation.

Le MPPA est encore une machine expérimentale et ses outils logiciels sont en

cours de développement. Plusieurs fonctionnalités prévues par l'architecture ne sont pas utilisables avec les outils de MPPA ACCESSCORE. Par exemple, il n'est possible d'utiliser que deux des clusters d'I/O sur les quatre existants. Durant notre travail avec la version 1.0.1 de *MPPA ACCESSCORE* nous avons rencontré un bogue majeur survenant au lancement de l'exécutable destiné à l'I/O (`mppa_spawn`) : l'application se fige parfois sans générer aucun message d'erreur. Ce problème est résolu dans la nouvelle version 1.2.1. Nous avons donc du mettre à jour la suite logicielle du MPPA en cours de travail, ce qui implique que nous avons du également faire différentes modifications dans notre code pour refléter les changements survenus dans l'utilisation des connecteurs pour les communications et dans le prototype des fonctions. La version 1.2.1. présente également un bogue survenant à la fermeture des descripteurs de fichiers utilisés pour les communications à partir de l'hôte (`mppa_close`) : l'application se fige systématiquement sans générer de message d'erreur. Ceci nous a contraint à arrêter notre programme sans fermer les descripteurs de fichiers, ce qui n'est pas très satisfaisant.

Le fait que *MPPA ACCESSCORE* soit en cours de développement laisse espérer de nouvelles fonctionnalités dans les prochaines versions et une amélioration des performances des fonctionnalités existantes. Si la carte MPPA-256 semble limitée dans ses performances, ce n'est que le premier membre de la famille *MPPA MANYCORE*. Un processeurs *Coolidge* à 1024 cœurs est prévu pour 2015⁷. De plus, 4 cartes MPPA-256 peuvent être utilisées ensemble dans la même machine hôte pour augmenter la capacité de calcul disponible.

2.2 Hypothèses de parallélisation

Comme nous l'avons vu dans la section 1.5, il est possible de paralléliser une recherche UCT au niveau de l'arbre, de la racine ou des feuilles. L'architecture MPPA est une architecture distribuée, cependant, chaque cluster dispose d'une mémoire partagée entre les 16 processeurs élémentaires qui le composent, ce qui pourrait permettre une parallélisation au niveau de l'arbre à l'intérieur de chaque cluster. La parallélisation à la racine pourrait, elle, être réalisée avec la participation des 256 processeurs élémentaires. Une hybridation de ces deux techniques serait même envisageable si la limitation de la mémoire des clusters à 2 Mo ne rendait pas toute construction d'un arbre UCT impossible.

Cette mémoire très réduite est déjà en partie utilisée pour stocker l'exécutable destiné au cluster et les données du jeu (les circuits et le vecteur de booléens décrivant l'état du jeu), il n'est donc pas envisageable de construire un arbre UCT dans la mémoire des clusters. En conséquence, nous avons écarté la possibilité de réaliser une parallélisation au niveau de l'arbre ou au niveau de la racine avec le MPPA.

La seule parallélisation possible sur le MPPA est une parallélisation au niveau des feuilles. L'arbre UCT est construit par *LeJoueur* sur la machine hôte et les de-

7. Annonce faite sur la page <http://www.kalray.eu/products/mppa-manycore/>

mandes de *playouts* sont transmises et traitées par le MPPA. Plusieurs approches sont alors possibles pour le calcul des *playouts* : chaque processeur élémentaire peut calculer un ou plusieurs *playouts* à partir d'une position donnée ou bien, à l'intérieur d'un cluster les différents processeurs élémentaires peuvent collaborer au calcul d'un seul et même *playout*.

Si chaque processeur calcule un *playout* distinct, ce *playout* peut être effectué à partir de la même position ou bien à partir de positions différentes. Réaliser 256 *playouts* à partir de la même position peut présenter un intérêt moindre pour l'exploration UCT qu'à partir de positions distinctes [Chaslot *et al.*, 2008]. Il conviendra de voir quel surcoût en communications est engendré par la transmission de N positions distinctes plutôt qu'une seule.

Pour diminuer le poids des communications par rapport au temps de calcul, il est également possible d'effectuer plusieurs *playouts* par processeur élémentaire. Cependant, là encore, l'intérêt de multiplier les *playouts* est discutable pour une exploration UCT.

Plutôt que de multiplier les *playouts*, la collaboration des processeurs élémentaires d'un cluster peut permettre d'accélérer le calcul d'un *playout*. Chaque strate d'un *Circuit* devant être calculée avant la suivante (voir section 1.6.3), les N processeurs élémentaires devront calculer une règle sur N et s'attendre avant de passer au calcul de la strate suivante. Nous supposons que cette attente risque de générer un surcoût en synchronisation ; cependant, les règles étant de longueurs relativement similaires, ce surcoût peut s'avérer minime : une expérimentation est nécessaire pour évaluer le gain apporté par le partage du *playout* ou la perte due aux synchronisations.

2.3 Choix de communications synchrones

L'objectif de notre recherche n'est pas d'obtenir une parallélisation performante par rapport à la version séquentielle de *LeJoueur*. L'étude de Jouandeau [2013] montre que la vitesse de fonctionnement d'une application parallélisée sur la carte MPPA est au mieux équivalente à la vitesse de fonctionnement de la même application séquentielle exécutée sur la machine hôte (Core I7). Notre étude vise plutôt, pour le moment, à permettre un choix entre les deux stratégies : partage d'un *playout* ou multiplication des *playouts* et à évaluer les coûts des transmissions en fonction de la taille des données.

Pour réaliser nos tests, nous avons donc choisi d'utiliser des communications synchrones dans un premier temps bien que ce type de communication soit susceptible d'amener un surcoût en synchronisation. Dans le cadre de ces tests, nous avons lancé des séries de *playouts* uniquement à partir de la position initiale du jeu. Les *playouts* sont donc de longueur maximale : le poids relatif des temps de calcul est donc maximisé par rapport aux temps de synchronisation. Tous les *playouts* effectués en parallèle à partir de la position initiale sont moins susceptibles de présenter

des durées disparates que des *playouts* effectués à partir de positions distinctes ; le surcoût en synchronisation est donc minimisé.

L'utilisation de communications asynchrones permettra, dans un second temps, de superposer les temps de communication et de calcul et d'accélérer la vitesse globale de l'application. Cependant, le nombre de communications sera multiplié : au lieu d'une seule communication de grande taille pour envoyer un lot de positions, plusieurs communications de petites tailles seront nécessaires pour envoyer les positions de manière échelonnée à mesure que les processeurs élémentaires auront fini les calculs précédents. Durant une partie, les *playouts* seront susceptibles de beaucoup varier en durée, rendant l'utilisation de ces communications asynchrones nécessaire pour minimiser les coûts en synchronisation.

L'utilisation de communications synchrones présente l'avantage de la simplicité et nous permet dans un premier temps de nous concentrer sur les éléments que nous désirons tester. De plus, pour estimer l'apport de communications asynchrones il est nécessaire de disposer au préalable d'une base de comparaison.

2.4 Parallélisation aux feuilles

Comme nous l'avons évoqué dans la section précédente, pour effectuer nos tests nous avons lancé des séries de *playouts* à partir de positions initiales et mesuré leur temps d'exécution en fonction du nombre de clusters et de threads par cluster pour les différentes approches utilisées. Pour cela nous avons utilisé uniquement la fonctionnalité de *LeJoueur* permettant de raisonner sur le jeu : le *Reasoner*.

Dans cette section, nous présentons les classes *Reasoner* utilisées par *LeJoueur* et les outils de test fournis par Jean-Noël Vittaut. Nous décrivons ensuite la structure de notre programme et les modifications apportées aux classes de *LeJoueur* pour effectuer la parallélisation. Nous présentons enfin les types de communications utilisés ainsi que la nature et le codage des données transmises au MPPA.

2.4.1 Les *Reasoner* de *LeJoueur*

Pour tester diverses approches pour le traitement des règles du jeu avec *LeJoueur*, Jean-Noël Vittaut a créé une classe abstraite nommée *Reasoner* (voir figure 2.10). Cette classe réunit les fonctionnalités nécessaires pour raisonner sur le jeu et effectuer une simulation (*playout*). *LeJoueur* possède différentes classes dérivées du *Reasoner* et utilisables pendant le jeu, parmi elles le *YAPReasoner* utilisant Yap prolog, le *PropnetReasoner* utilisant un *propnet* et l'*HybridReasoner* utilisant les deux pendant une phase initiale de test puis continuant à utiliser le plus rapide par la suite. Pour traiter les règles du jeu, *LeJoueur* utilise donc une classe dérivée du *Reasoner* capable de réaliser les opérations suivantes :

- Consulter un fichier GDL et le charger en mémoire : le *PropnetReasoner* fait ap-

pel à l'objet *Grounder* (voir section 1.6.3) pour instancier les règles et les transformer en circuits puis crée le vecteur de booléens ; le *YAPReasoner* transforme les règles GDL en Prolog et peut également utiliser le *tabling*, le stockage de règles instanciées, pour accélérer les calculs.

- Donner la correspondance entre l'index d'un joueur et le terme GDL correspondant (exemple : 0 = (role player0) ; 1 = (role playerX)).
- Charger la position initiale du jeu indiquée par les règles.
- Mettre à vrai une liste de faits : modifier l'état du jeu.
- Donner la liste des faits vrais : indiquer la position courante.
- Calculer si la position courante est terminale.
- Calculer les scores (si la position est terminale).
- Calculer la liste des coups légaux.
- Jouer un coup.
- Calculer la position suivante.
- Jouer tout un *playout* et retourner les scores.

Le *Reasoner* peut aussi enregistrer le temps limite fixé par le *Game Master* et indiquer si ce temps est écoulé. La méthode `interrupt()` permet de stopper le *Reasoner* pendant son travail : ceci est utile si on lui demande d'effectuer un long groupe de *playouts* qu'on souhaite interrompre avant la fin du temps imparti pour obtenir le résultat déjà calculé.

2.4.2 Outils pour tester les *Reasoner*

LeJoueur propose un exécutable de test nommé `TestReasoner` et permettant d'évaluer les différents *Reasoner*. Le fichier `TestReasoner.cpp` contient une fonction `CheckPropnet()` permettant de vérifier le bon fonctionnement d'un *Reasoner* en comparant son fonctionnement avec celui du *YapReasoner* standard utilisant Prolog. La position initiale du jeu est chargée dans les deux *Reasoner*, puis un *playout* est exécuté en comparant pas à pas le résultat des calculs effectués par les *Reasoners* : la position doit être terminale ou non pour les deux ; la liste des coups légaux doit être la même ; pour un même coup joué l'état suivant doit être le même et ainsi de suite jusqu'au score final. Jean-Noël Vittaut utilise cette fonction pour vérifier le bon fonctionnement de son *propnet* et conseille son utilisation pour vérifier la cohérence des calculs de tout autre *Reasoner* implémenté.

Le fichier `TestReasoner.cpp` contient également une fonction `testDirectory()` permettant de charger un par un les fichiers d'un répertoire contenant les règles GDL de différents jeux, de créer un *Reasoner* et de lancer une série de *playout* depuis la position initiale du jeu. La création et la destruction du *Reasoner* entre chaque test pose un problème dans notre situation. A la création du *Reasoner*, l'application est déployée sur le MPPA et les canaux de communication avec l'I/O sont ouverts. Il faut donc fermer ces canaux de communication et stopper l'application exécutée sur le MPPA à la destruction du *Reasoner*. Cependant, il n'est pas possible de déployer le même exécutable deux fois de suite sur le MPPA à partir du même processus : ceci est détecté comme une erreur.

```
class Reasoner {
public:
    virtual ~Reasoner() {}

    virtual void consult(const char* filename) = 0;

    virtual GDLTermPtr role(RoleId i) = 0;
    virtual RoleId id_role(GDLTermPtr) = 0;

    virtual GDLCompoundTermVecPtr init() = 0;
    virtual void set_dbtrue(GDLCompoundTermVecPtr& facts) = 0;
    virtual GDLCompoundTermVecPtr get_db_true() = 0;

    virtual bool terminal() = 0;
    virtual vector<GDLCompoundTermVec> legal() = 0;
    virtual void goal(GDLScores& s) = 0;

    virtual void set_dbdoes(GDLCompoundTermVecPtr& facts) = 0;

    virtual vector<GDLCompoundTermVec> sees() = 0;
    virtual void next() = 0;

    virtual void playout(GDLScores& s) = 0;

    virtual GDLParserPtr get_parser() = 0;

    virtual void interrupt() = 0;

    virtual bool more_time() = 0;
    virtual void set_timeout(size_t ms) = 0;

    inline bool start_playout() { playout(scores); return true; }
    inline bool get_playout(GDLScores& s) { s = scores; return true; }

private:
    GDLScores scores;
};
```

FIGURE 2.10 – Classe abstraite *Reasoner* de *LeJoueur* version 02/2014.

Pour éviter ce problème, notre première implémentation utilisait une classe *MP-PAInterface* chargée de communiquer avec le MPPA. Cette interface était créée au démarrage de l'application *TestReasoner* et transmise à chaque *Reasoner* créé. Une procédure de ré-initialisation permettait de nettoyer les données du jeu précédent avant de communiquer les données d'un autre jeu. Cependant, ceci contraignait à déporter l'implémentation de chaque méthode du *Reasoner* dans l'interface et à dupliquer une partie du code ce qui était propice à l'apparition de bogues et rendait difficile la mise au point du programme. Cette approche risquait également de rendre notre code incompatible avec les évolutions éventuelles du joueur de Jean-Noël Vittaut⁸.

Nous avons donc créé une seconde implémentation et utilisé une classe *MPPA-Reasoner* communiquant directement avec le MPPA pour conserver un code plus simple. Ce choix implique que notre application doit être relancée entre chaque jeu testé.

Enfin le fichier *TestReasoner* contient une fonction permettant de lancer des *playouts* en boucle à partir de la position initiale du jeu. Le nombre d'itérations est fixé par une limite de temps de 15 secondes. Cette fonction permet d'évaluer la vitesse du *Reasoner* en nombre de *playouts* par seconde. Comme nous avons choisi d'utiliser des communications synchrones, lorsque le nombre de clusters et threads utilisés augmente, tout un groupe de *playouts* risque d'être en cours d'exécution à l'expiration du délai. Ce problème est encore plus marqué si plusieurs *playouts* sont exécutés par chaque PE. Pour éviter que cela ne fausse notre évaluation des performances du MPPA, nous avons choisi de remplacer ce délai par un nombre fixe de *playouts* et d'estimer ensuite le nombre de *playouts* par seconde en divisant le temps total d'exécution par le nombre de *playouts*. Pour adapter le programme aux conditions des compétitions de *General Game Playing* il faudra rétablir un délai chronométré et utiliser des communications asynchrones pour récupérer les résultats des *playouts* à mesure de leur calcul.

2.4.3 Structure de notre programme

Notre *MPPAReasoner* que nous avons évoqué dans la section précédente, est fondé sur le *PropnetReasoner* de Jean-Noël Vittaut et constitue la partie hôte de notre programme destiné au MPPA. Il utilise le *Grounder* pour instancier les règles GDL du jeu, récupère les circuits correspondant aux règles transformées, crée la liste de booléens destinée à conserver les valeurs des faits, charge le fichier multi-binaire dans la mémoire du MPPA et lance l'exécutable destiné à l'I/O. Les circuits et les caractéristiques du vecteur de booléens sont alors transmises à l'I/O.

Pour chacun des *playouts* demandés, l'hôte transmet un groupe de positions de

8. Par la suite Jean-Noël Vittaut a lui même modifié le prototype de sa classe abstraite *Reasoner*, et opéré d'importantes modifications de son *Grounder* (classe réalisant l'instanciation et divers traitements de simplification des règles GDL) nous avons donc décidé de continuer à travailler avec la version de février 2014 de *LeJoueur*, les autres n'étant plus compatibles.

départ à l'I/O et attend le résultat de la simulation calculé par le MPPA. Pour tester les deux parallélisations possibles (calcul de *playouts* distincts ou partage d'un *playout*) nous avons prévu d'envoyer une position de départ distincte pour chaque cluster ou pour chaque processeur élémentaire (PE), soit respectivement 16 ou 256 positions au maximum. Le prototype de la méthode `playout()` de *LeJoueur* (version 02/2014) n'étant pas prévu pour recevoir différentes positions, nous avons dupliqué la position passée en argument autant de fois que nécessaire pour simuler l'envoi de plusieurs positions et évaluer les temps de transmission. Le nombre de résultats reçus est donc égal au nombre de positions envoyées, i.e. de *playouts* effectués.

Le rôle de l'I/O se limite à celui d'un répartiteur. Le programme destiné à l'I/O reçoit les informations en provenance de l'hôte et les retransmet aux clusters : les règles du jeu et les caractéristiques du vecteur de booléens sont retransmises à chaque cluster, tandis que les positions de départ des *playouts* sont réparties entre les clusters. Les résultats en provenance des différents clusters sont regroupés avant d'être transmis en une seule communication à l'hôte.

Le programme destiné aux clusters attend donc la description d'un jeu et du vecteur de booléens puis effectue un ou plusieurs *playouts* en fonction des versions que nous avons implémentées et retourne le ou les résultats.

Chaque programme est écrit en C++. Nous avons utilisé les environnements *PO-SIX* et *pthread* fournis par le MPPA *ACCESSCORE*.

2.4.4 Communications et transmission d'ordres

Pour la transmission des données entre l'hôte et l'I/O, nous avons utilisé une communication de type `buffer` permettant l'envoi de grandes quantités de données. Les circuits étant susceptibles d'atteindre de grandes tailles selon les jeux, ce type de communication est le plus adapté. Les positions étant communiquées de manière synchrone, nous avons aussi utilisé le `buffer` pour les transmettre. Dans le cas de communications asynchrones, il serait nécessaire d'utiliser le type `mqueue` pour les transmettre et récupérer les résultats.

Entre l'I/O et les clusters, nous avons utilisé une communication de type `portal` permettant l'envoi de messages de grande taille. Là encore c'est le type le plus adapté pour transmettre les circuits. Nous l'avons utilisé également pour la transmission des positions et des résultats. Dans *LeJoueur* version 02/2014, les booléens sont codés par des `short int`. Dans le cas de communications asynchrones, les types `sampler` et `rqueue`, permettraient de créer une liste de travaux à effectuer et d'accumuler les résultats. Le type `rqueue` est limité à des messages de 120 octets et nécessiterait de modifier le codage des positions, les positions des jeux (liste des *true*s) dépassant fréquemment les 60 booléens. Les communications par `channel` sont limitées à un seul lecteur et un seul écrivain mais elles ne présentent pas cette limitation de taille. Le choix entre `sampler/rqueue` et `channel` devrait donc être soigneusement étudié pour l'implémentation de communications asynchrones.

Pour vérifier la cohérence des calculs effectués au niveau des clusters, nous avons utilisé la méthode `CheckPropnet()` proposée par *LeJoueur*. Pour demander au MPPA de ne calculer qu'une partie d'un *playout* plutôt qu'un *playout* complet (réception d'une position initiale, vérification si la position est terminale, scores obtenus ou liste des coups légaux, calcul de la position suivante) il est nécessaire de communiquer l'opération souhaitée au MPPA. Pour cela nous avons utilisé une liste d'entiers décrivant chacun une des opérations possibles. Une première communication indique donc chaque fois le type d'opération demandée : initialisation des données du jeu, un *playout* ou une opération élémentaire. Un ordre permet également de demander l'arrêt des applications des clusters et de l'I/O à la destruction de l'objet *MPPAReasoner*. Une fois un ordre reçu, chaque partie (hôte, I/O, clusters) peut donc se préparer à recevoir ou/et envoyer les données correspondantes.

2.4.5 Codage des données à transmettre

Dans *LeJoueur* version 02/2014, les circuits sont stockés dans des structures de données complexes. Les règles sont des vecteurs de pointeurs sur les booléens stockés dans une liste ordonnée (voir section 1.6.3). Pour transférer les circuits dans l'espace mémoire de l'I/O puis des clusters, les pointeurs doivent être remplacés par des index : nous avons donc réécrit la classe *Circuit*. Pour pouvoir coder les faits négatifs, nous avons utilisé uniquement des index pairs (l'index réel est donc multiplié par deux) et ajouté 1 pour signaler la négation (voir figure 1.10 ou 2.11). Il aurait aussi été possible d'utiliser des index négatifs pour les faits négatifs mais la négation de l'index 0 étant impossible il aurait fallu opérer un décalage d'une unité de tous les index ce qui n'était pas plus simple au final.

Pour envoyer la totalité des circuits en une seule transmission au MPPA, nous avons sérialisé les données. Les quatre groupes de circuits *TERMINAL*, *LEGAL*, *GOAL*, *NEXT* sont placés les uns à la suite des autres dans un tableau *C*. Une valeur indique le nombre de circuits de chaque groupe ; chaque strate est précédée d'un nombre indiquant le nombre de règles qu'elle contient et chaque règle est précédée d'un nombre indiquant sa taille (voir figure 2.11). Chaque valeur est codée par un entier non signé. Ce type de données que nous avons nommé *idx* et choisi pour garantir que tous les index pourront être codés est défini dans un fichier d'entête global de manière à pouvoir être modifié facilement.

Le vecteur de booléen ne nécessite pas d'être envoyé, seule la taille de chaque section (*TRUE*, *NEXT*, *TERMINAL*, *DOES*, *LEGAL*, *GOAL*, *REST*) doit être communiquée pour construire la liste de booléens dans les clusters et permettre par exemple la copie des *NEXTS* sur les *TRUES* au moment d'un changement de position dans le jeu (voir section 1.6.3).

La position courante et la suivante (*TRUE* et *NEXT*) sont logiquement codées par des listes de booléens de même taille, de même pour les actions effectuées ou possibles (*DOES* et *LEGAL*). *TERMINAL* est toujours codé par un seul booléen donc la liste des valeurs à transmettre peut être réduite aux indications suivantes :

```

-- TERMINAL -----
-- circuit 0 --
214 :- 2.
214 :- 4.
214 :- 6.
[...]
222 :- 22, 24, 26.
224 :- 28, 30, 32.
226 :- 34, 36, 38.
[...]
-- circuit 1 --
210 :- 222.
210 :- 224.
[...]
-- circuit 2 --
116 :- 210.
116 :- 212.
[...]
-- LEGAL -----
-- circuit 0 --
188 :- 12, 0.
168 :- 12, 20.
190 :- 14, 0.
[...]
-- GOAL -----
-- circuit 0 --
200 :- 211, 213, 215, 213, 211.
206 :- 211, 213, 215, 213, 211.
[...]
-- NEXT -----
-- circuit 0 --
78 :- 0.
58 :- 20.
80 :- 118, 2.
...

```

Circuits sérialisés :

```

3 25 2 214 2 2 214 4 2 214 6 [...] 4 222 22 24 26 4 224 28 30 32 [...]
14 2 210 222 [...] 3 2 116 210 [...] 1 20 3 188 12 0 3 168 12 20 [...]
1 6 6 200 211 213 215 213 211 [...] 1 181 2 78 0 2 58 20 [...]

```

FIGURE 2.11 – Liste des règles (voir figure 1.10) telles que représentées dans les circuits modifiés : les pointeurs ont été remplacés par les index des valeurs booléennes correspondantes aux faits, multipliés par 2. Les index impairs permettent de coder les faits négatifs : par exemple le fait (line o) d'index 106 est codé par 212 et (not (line o)) est codé par 213. Pour sérialiser chaque *Circuit*, nous indiquons en premier (valeurs signalées en bleu) la taille de chaque groupe de circuits. Ensuite une valeur indique le nombre de règles de chaque strate (valeurs signalées en rouge). Enfin, la taille de chaque règle (valeurs signalées en vert) précède les index représentant la règle elle-même (un conséquent suivi de ses antécédents).

- taille des TRUES
- taille des DOES pour chaque joueur
- taille des GOALS
- taille du REST

Toutes ces informations sont envoyées sous forme d'un tableau d'idx et comme la taille du tableau dépend du nombre de joueurs, ce dernier est communiqué au préalable.

Dans une première implémentation nous avons communiqué chaque information séparément. Pour minimiser le nombre de communications, nous avons ensuite regroupé tout dans une seule table d'idx, une première communication étant utilisée pour signaler au processus lecteur la taille des données à attendre. Nous avons constaté une différence de 3 centièmes de seconde entre les deux approches, temps qui reste somme toute négligeable, l'envoi des données étant effectué une seule fois en début de jeu.

2.5 Partage ou multiplication des *playouts*

Les expériences que nous décrivons ici ont été menées sur le jeu de Tictactoe. Ce jeu présente des *playouts* assez courts (entre 5 et 9 coups) ce qui permet d'effectuer de nombreuses séries de tests dans un temps acceptable. Cependant, il faut garder en mémoire que la brièveté des *playouts* et donc des temps de calcul rend les temps de synchronisation et de communication plus importants en proportion.

Dans une première expérience, nous avons partagé le calcul de chaque *playout* entre les threads d'un cluster, chaque processeur élémentaire exécutant un thread. Nous avons effectué 100000 *runs*, i.e. envois d'un groupe de positions. Pour chaque groupe de positions envoyé, l'I/O distribue une position par cluster. Chaque cluster effectue un seul *playout*. L'I/O attend les résultats calculés par chaque cluster et retourne l'ensemble à l'hôte. Cette expérience a été répétée en utilisant 1, 2, 4, 8, 16 clusters et 1, 2, 4, 8 et 16 threads par cluster.

La figure 2.12 montre, comme attendu que le nombre de *playouts* effectués augmente régulièrement avec le nombre de clusters, mais la meilleure performance est obtenue avec 1 seul thread par cluster. La figure 2.13 met en évidence l'énorme coût en synchronisation engendré par le partage du calcul des *playouts*.

Grâce au *propnet*, le calcul de chaque phase du *playout* est considérablement accéléré, le partage du calcul d'un *playout* ne peut donc apporter aucun bénéfice face à l'importance des coûts en synchronisation.

Dans une seconde expérience, nous avons exécuté un *playout* sur chaque processeur élémentaire. Chaque cluster reçoit un groupe de positions à partir de chacune desquelles chaque thread réalise un *playout*. L'I/O regroupe les résultats de tous les calculs et retourne l'ensemble à l'hôte à la fin de chaque *run*. Un *run* correspond à l'envoi d'un groupe de positions par l'hôte. Nous avons effectué là aussi 100000 *runs*

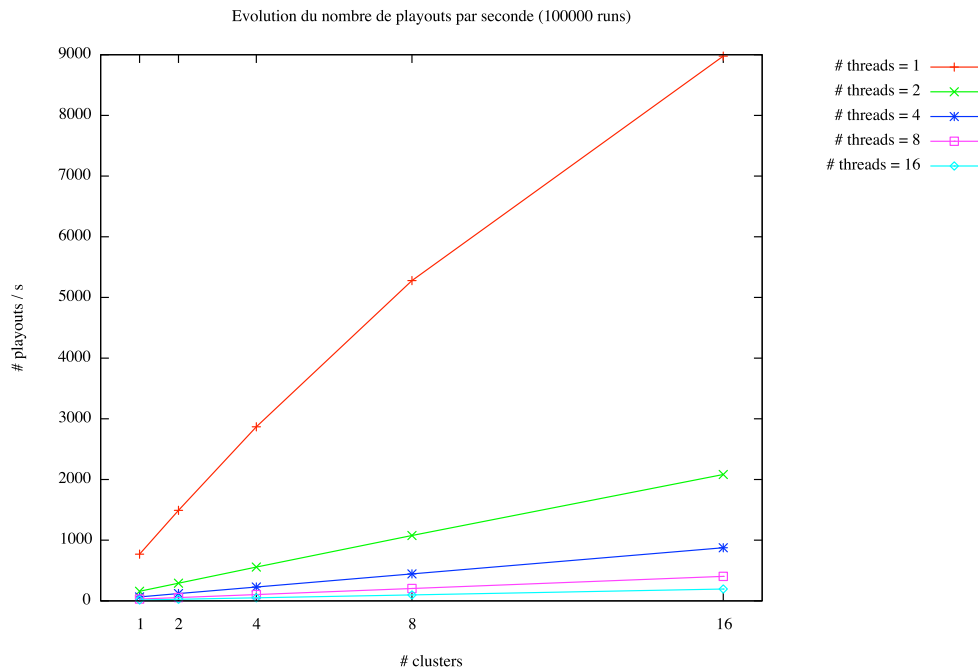


FIGURE 2.12 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de clusters utilisés pour le jeu de Tictactoe en partageant le calcul des circuits entre les threads de chaque cluster. Chaque cluster effectue un seul *playout* pour chaque position envoyée. L'envoi d'une position correspond à 1 *run*. La figure montre que la meilleure performance est obtenue avec 1 seul thread.

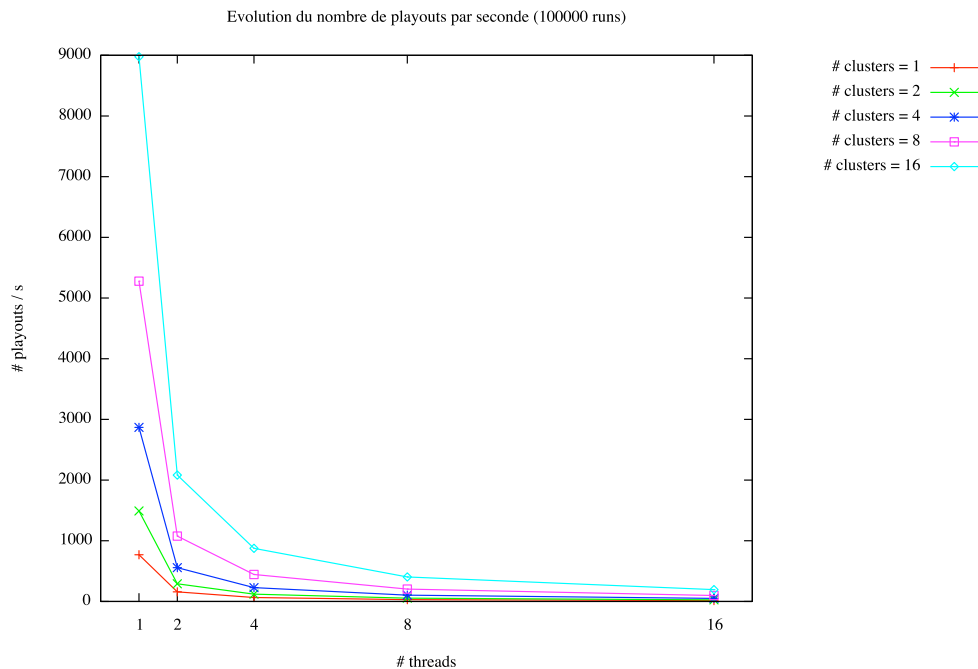


FIGURE 2.13 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de threads utilisés pour le jeu de Tictactoe en partageant le calcul des circuits entre les threads de chaque cluster. Chaque cluster effectue un seul *playout* pour chaque position envoyée. L'envoi d'une position correspond à 1 *run*. La figure met en évidence l'énorme coût en synchronisation engendré par le partage du calcul des *playouts*.

en faisant varier le nombre de clusters et de threads par cluster.

La figure 2.15 montre que cette fois le passage à l'échelle est satisfaisant. Cependant nous constatons que le poids en communications et synchronisation reste important. Avec un thread par cluster, le nombre de *playouts* n'est multiplié que par 11,68 en passant de 1 à 16 clusters, et multiplié par 10,99 avec 16 threads par cluster. Nous émettons l'hypothèse que ce coût est dû aux temps de synchronisations engendrés par l'usage de communications synchrones. L'I/O doit attendre les résultats de tous les clusters avant de renvoyer les résultats à l'hôte et les *playouts* étant de longueurs variables, chaque cluster doit attendre la fin du *playout* le plus long.

La figure 2.16 donne les mêmes résultats, cette fois en fonction du nombre de threads. En passant de 1 à 16 threads, le nombre de *playouts* n'est multiplié que par 8,92 avec un cluster, et multiplié par 8,40 avec 16 clusters. Les temps de communication et synchronisation semblent peser plus lourdement au niveau des threads.

La même expérience réalisée avec un nombre variable de *runs* donne des résultats constants (voir figure 2.14) sauf pour les nombres de *runs* inférieurs à 100 pour lesquels on constate une moins bonne performance. Cette différence peut être attribuée à la variation de la longueur des *playouts*. Elle devient négligeable à partir de 100 *runs*, nous avons donc réalisé nos expériences avec au minimum 1000 *runs* pour avoir des résultats fiables.

Pour cette expérience nous avons relancé les threads à chaque demande de *playout* (`pthread_create` / `pthread_join`). Une autre stratégie consiste à lancer les threads au démarrage de l'application puis à communiquer les demandes de *playouts* quand nécessaire. Les threads sont stoppés à la réception de l'ordre demandant l'extinction du MPPA. Pour déterminer laquelle de ces deux stratégies minimise les coûts en communications nous avons procédé à d'autres expérimentations.

2.6 Threads gardés actifs ou relancés

Pour tester différents types de communications, nous avons repris la seconde expérience de la section précédente en modifiant la gestion des threads. Chaque cluster reçoit un groupe de position qu'il distribue aux différents threads. Les tests sont réalisés sur le jeu de *Tictactoe* en faisant varier le nombre de clusters et de threads par cluster utilisés ; 100000 *runs* sont lancés chaque fois.

Plutôt que de relancer les threads à chaque *run*, les threads sont lancés au démarrage de l'application destinée aux clusters et stoppés à l'arrêt du programme. Les threads sont placés en attente avec la commande `pthread_cond_wait()` et à chaque demande de *playout*, il sont réveillés par la commande `pthread_cond_signal()`. Un mutex est utilisé pour protéger la variable de type `pthread_cond_t` permettant de contrôler chaque thread.

Les résultats obtenus sont présentés sur les figures 2.17 en fonction du nombre

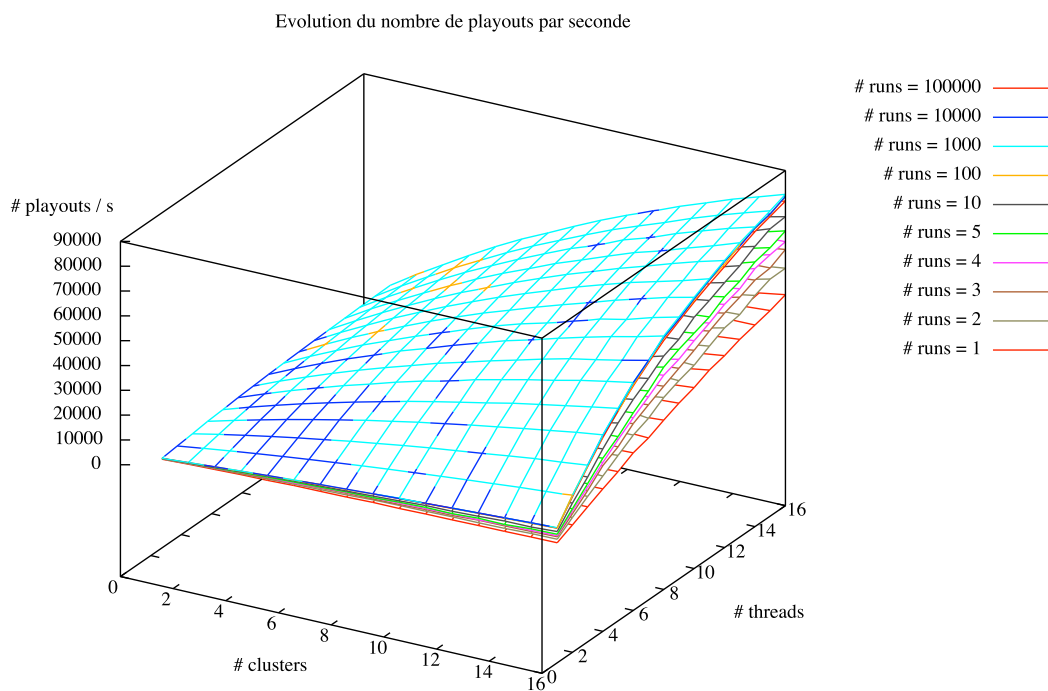


FIGURE 2.14 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de clusters et threads utilisés pour le jeu de Tictactoe en réalisant un *playout* complet par PE. Chaque thread effectue un seul *playout* pour chaque position envoyée. L’envoi d’une position correspond à 1 *run*. Les threads sont relancés pour chaque *playout*. Chaque *couche* du graphe correspond à une exécution du programme avec un nombre de *runs* spécifique. Les résultats sont constants sauf pour les nombres de *runs* inférieurs à 100, la différence peut être attribuée à la variation de longueur des *playouts*.

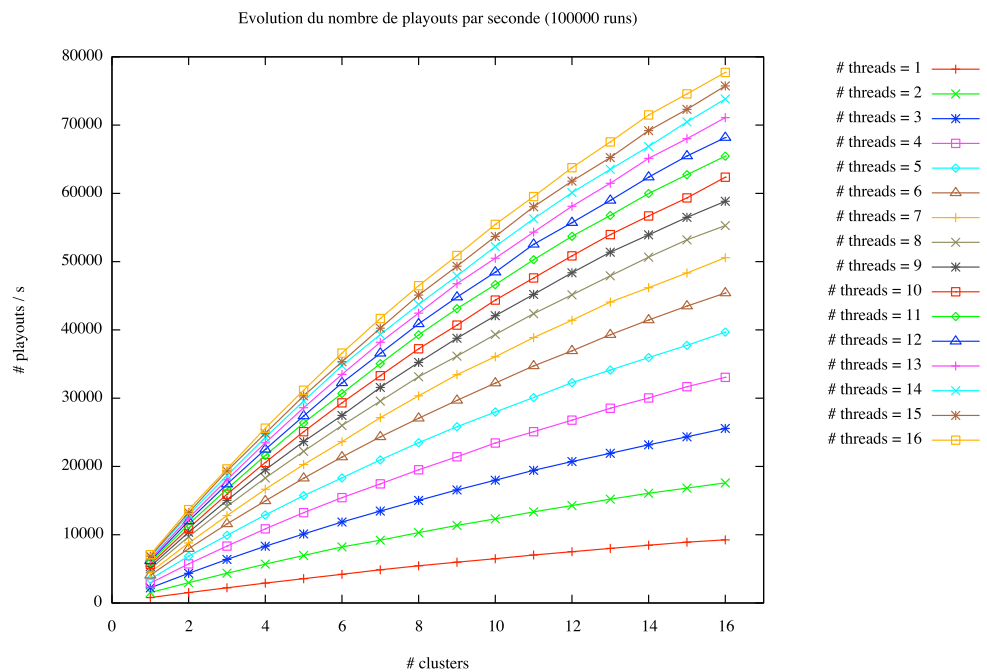


FIGURE 2.15 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de clusters utilisés pour le jeu de Tictactoe en réalisant un *playout* complet par PE. Chaque thread effectue un seul *playout* pour chaque position envoyée. L’envoi d’une position correspond à 1 *run*. Les threads sont relancés pour chaque *playout*. Malgré le bon passage à l’échelle, nous constatons que le poids en communications et synchronisation reste important.

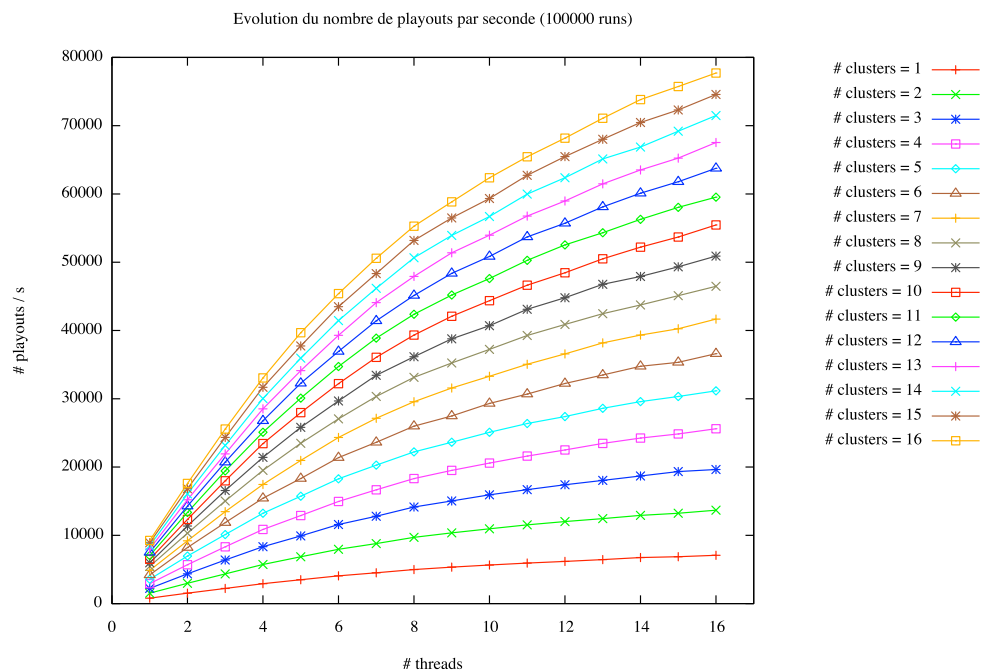


FIGURE 2.16 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de threads utilisés pour le jeu de Tictactoe en réalisant un *playout* complet par PE. Chaque thread effectue un seul *playout* pour chaque position envoyée. L’envoi d’une position correspond à 1 *run*. Les threads sont relancés pour chaque *playout*. Les courbes présentent une inflexion autour de 8 threads : les temps de communication et de synchronisation semblent peser plus lourdement au niveau des threads.

de clusters et 2.18 en fonction du nombre de threads. Il est visible qu'à partir de 14 threads le passage à l'échelle ne se fait plus. Étrangement, il semble plus long de garder les threads actifs que de les relancer pour chaque *playout*.

Afin de vérifier que l'implémentation des commandes `pthread_cond_signal()` et `pthread_cond_wait()` sur le MPPA n'est pas à l'origine de cette mauvaise performance, nous les avons remplacées par une attente active. Chaque thread réalise une boucle infinie testant la valeur d'une variable `signal` à chaque itération : quand la variable contient une demande de *playout*, un *playout* est exécuté et la variable est réinitialisée. Là encore, l'utilisation d'un mutex est nécessaire pour protéger l'accès à la variable `signal`.

Les figures 2.19 et 2.20 montrent que les résultats obtenus avec une attente active sont tout à fait similaires aux précédents. Nous en concluons que les commandes `pthread_cond_signal()` et `pthread_cond_wait()` sont probablement implémentées avec une attente active dans le MPPA.

L'usage d'un mutex semble donc particulièrement coûteux pour contrôler les threads. Nous avons songé à utiliser des *pipes* pour établir des canaux de communication entre le PE0 exécutant l'application destinée au cluster et les autres PE. Un *pipe* aurait permis une communication sans utiliser de mutex, malheureusement les *pipes* ne sont pas implémentés sur le MPPA. Nous avons également tenté d'établir une communication entre différents threads d'un même cluster grâce à des connecteurs de type *portal* et *sync*⁹ mais sans succès¹⁰.

Il ne semble donc pas possible d'améliorer le coût en synchronisation au niveau des threads. Relancer les threads pour chaque *playout* semble la stratégie la plus économique¹¹.

2.7 Temps de communication et de calcul

Dans le but d'évaluer la part relative des temps de communication et de calcul, nous avons exécuté trois versions de notre programme, testées sur le jeu de *Tictactoe*. La version *A* exécute 1000 *runs* sans effectuer de *playout* au niveau des threads : le but est d'avoir une mesure du temps de communication seul. Dans la seconde version *B*, les threads effectuent une attente fixe de 1 centième de seconde au lieu de calculer un *playout* : une fois le temps soustrait (*B'*), cette mesure devrait confirmer la première. Dans la troisième expérience *C*, chaque thread lance 1 *playout* : une fois soustrait le temps de communication, cette expérience pourra nous indiquer le temps de calcul,

9. Rien n'indique dans la documentation si un même cluster peut (ou ne peut pas) utiliser un connecteur en étant à la fois lecteur et écrivain.

10. Gailhard [2014] de la société Kalray nous a affirmé que les *portal* devaient normalement pouvoir être utilisés entre deux threads/PE d'un même cluster. Cependant notre code génère une erreur "[EE] (portal) < 0> bad receiver" dont nous n'avons pas encore pu établir l'origine.

11. Gailhard [2014] de la société Kalray s'est fortement étonné de ces résultats et suggère de plus amples tests pour les confirmer, assurant que, d'après lui, relancer les threads ne peut en aucun cas être plus rapide que les autres approches.

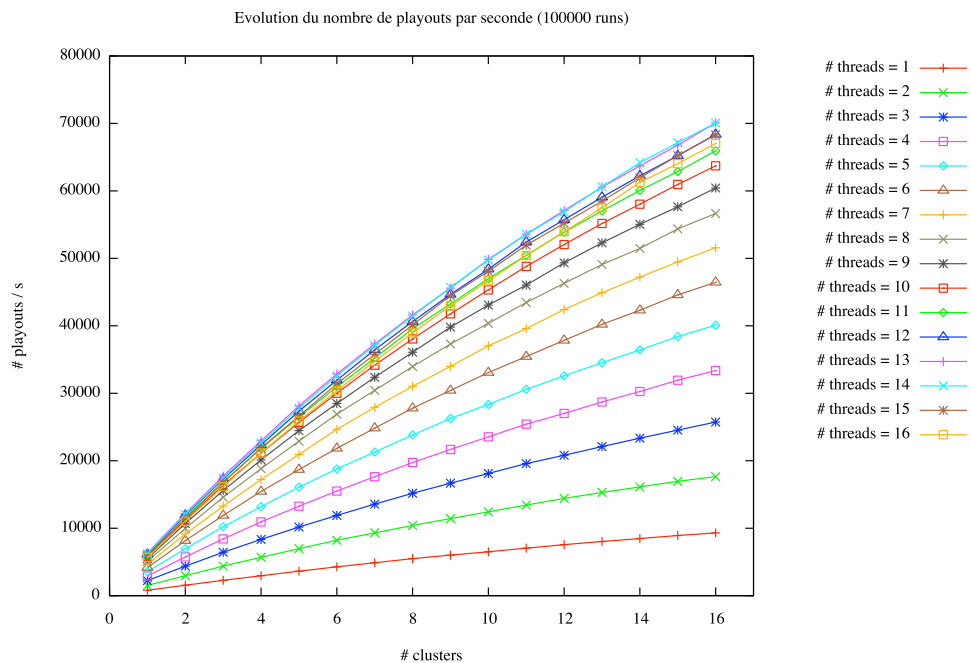


FIGURE 2.17 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de clusters utilisés pour le jeu de Tictactoe en réalisant un *playout* complet par PE. Chaque thread effectue un seul *playout* pour chaque position envoyée. L’envoi d’une position correspond à 1 *run*. Les threads sont lancés à l’initialisation du MPPA puis contrôlés avec `pthread_cond_signal()` et `pthread_cond_wait()`. La figure indique que le passage à l’échelle est correct au niveau des clusters.

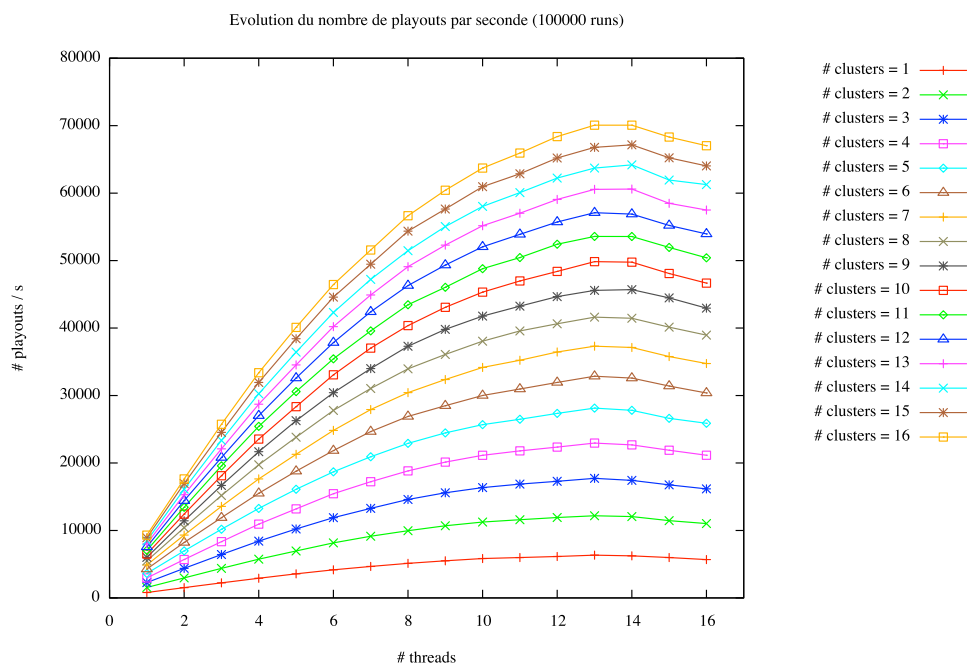


FIGURE 2.18 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de threads utilisés pour le jeu de Tictactoe en réalisant un *playout* complet par PE. Chaque thread effectue un seul *playout* pour chaque position envoyée. L’envoi d’une position correspond à 1 *run*. Les threads sont lancés à l’initialisation du MPPA puis contrôlés avec `pthread_cond_signal()` et `pthread_cond_wait()`. La figure indique que le passage à l’échelle ne se fait plus à partir de 14 threads.

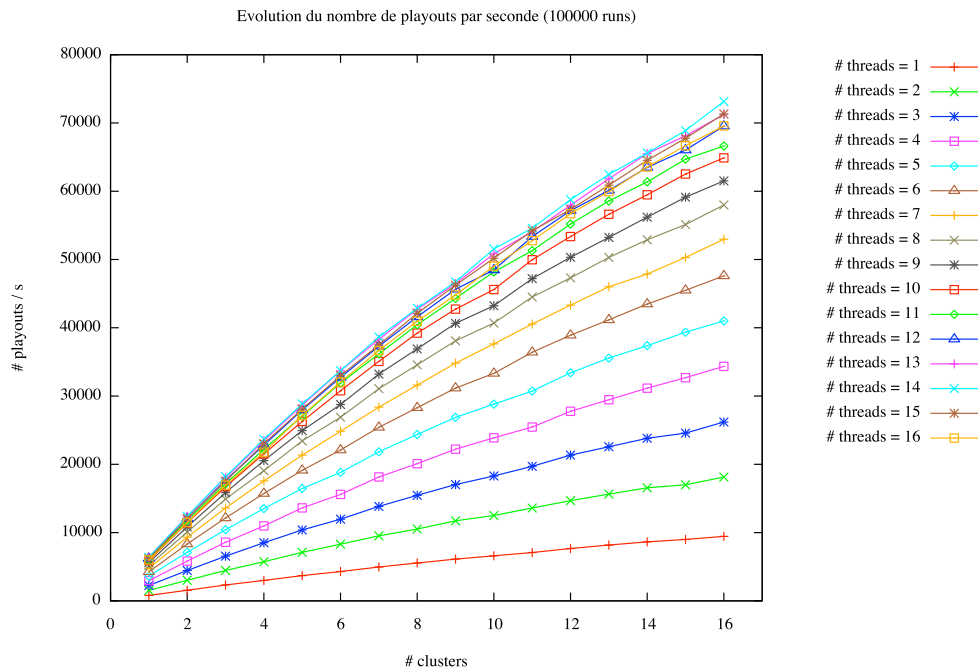


FIGURE 2.19 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de clusters utilisés pour le jeu de Tictactoe en réalisant un *playout* complet par PE. Chaque thread effectue un seul *playout* pour chaque position envoyée. L'envoi d'une position correspond à 1 *run*. Les threads sont lancés à l'initialisation du MPPA puis procèdent à une attente active. Les résultats sont similaires à ceux obtenus sur la figure 2.17.

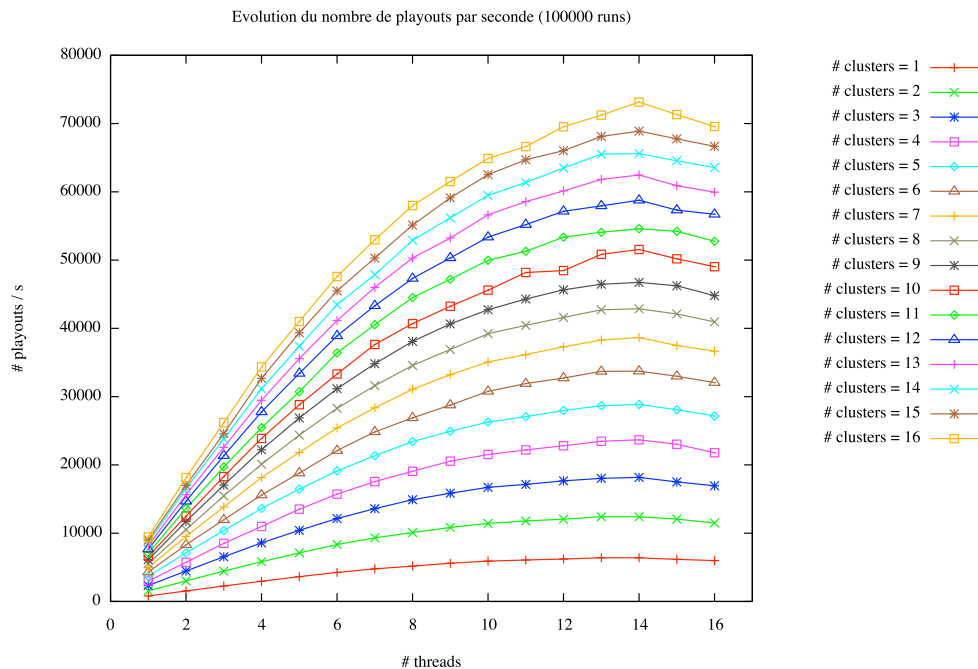


FIGURE 2.20 – Évaluation du nombre de *playouts* par seconde en fonction du nombre de threads utilisés pour le jeu de Tictactoe en réalisant un *playout* complet par PE. Chaque thread effectue un seul *playout* pour chaque position envoyée. L'envoi d'une position correspond à 1 *run*. Les threads sont lancés à l'initialisation du MPPA puis procèdent à une attente active. Les résultats sont similaires à ceux obtenus sur la figure 2.18.

		A	B'-A	B'	B	C-A	C-B'	C
clusters	threads	sans <i>playout</i>			Attente 0.01s			1 <i>playout</i>
1	1	0.96 (± 0.01)	0.44	1.40	11.40 (± 0.01)	0.31	-0.14	1.27 (± 0.01)
1	16	1.18 (± 0.01)	0.42	1.60	11.60 (± 0.02)	1.42	0.99	2.60 (± 0.02)
4	1	1.29 (± 0.01)	0.21	1.50	11.50 (± 0.01)	0.10	-0.10	1.39 (± 0.01)
4	16	1.39 (± 0.01)	0.49	1.88	11.88 (± 0.01)	1.50	1.01	2.89 (± 0.02)
16	1	2.58 (± 0.02)	-0.64	1.94	11.94 (± 0.01)	-0.60	0.04	1.98 (± 0.01)
16	16	3.09 (± 0.02)	-0.35	2.74	12.74 (± 0.01)	0.70	1.05	3.79 (± 0.03)

FIGURE 2.21 – Évaluation des temps de communication et de calcul. La version A du programme n'effectue que des communications. La version B permet de confirmer les premières mesures en effectuant une attente fixe de 0.01s qui est soustraite ensuite (B') pour pouvoir comparer les résultats avec les autres expériences. Dans la version C chaque thread réalise 1 *playout*. Le programme effectue 1000 runs, chaque mesure correspond à une moyenne que 10000 exécutions. L'écart type est donné entre parenthèses. Les valeurs sont arrondies à deux décimales après calcul des écarts $B' - A$, $C - A$ et $C - B'$ ce qui explique des valeurs comme $C - B' = 0.99$ pour 1 cluster et 16 threads. Les incohérences signalées en rouge rendent difficile l'évaluation des temps de communication et de calcul.

i.e. consacré aux *playouts*. Chaque expérience est lancée 10000 fois avec un nombre de clusters de 1, 4 ou 16 et un nombre de threads par cluster de 1 ou 16. Les résultats obtenus sont présentés dans le tableau de la figure 2.21.

Un examen rapide des mesures laisse entrevoir plusieurs incohérences. Les mesures B' (attente fixe soustraite pour obtenir le temps de communication seul), censées confirmer les mesures effectuées sans *playouts* (A), sont en fait très différentes : les communications prennent plus de temps jusqu'à 4 clusters et la tendance s'inverse ensuite jusqu'à être inférieure pour 16 clusters. Plus surprenant encore, pour 16 clusters avec un seul thread chacun, l'application met moins de temps à effectuer 1 *playout* par thread (C) qu'à ne rien faire (A).

Dans la version B, avec une attente fixe, les différences entre les temps d'exécution lors du passage de 1 à 16 threads pour 1, 4 et 16 clusters (0.20s, 0.38s et 0.80s) sont cohérentes : l'augmentation du temps correspond au temps utilisé pour les opérations *create* et *join* pour chaque thread et à l'augmentation de la taille des communications entre l'I/O et les clusters. En passant de 1 à 4 puis 4 à 16 clusters on a une multiplication par 2 de l'écart qui correspond à l'augmentation des communications entre hôte et I/O et au temps de répartition des données entre les clusters.

Pour la version C, avec 1 *playout*, les écarts de temps d'exécution entre 1 et 16 threads (1.33s, 1.5s et 1.81s) sont plus importants ce qui s'explique par l'utilisation de communications synchrones : chaque cluster doit attendre la fin de l'exécution du *playout* le plus long qui varie entre 5 et 9 coups pour une partie de *Tictactoe*.

Dans la version A, sans *playout*, les écarts de temps d'exécution entre 1 et 16 threads sont surprenants : 0.22s pour 1 cluster, 0.10s pour 4 clusters et 0.51s pour 16 clusters. Nous ne pouvons expliquer pourquoi l'écart est moindre pour 4 clusters alors que nous nous attendions à une augmentation régulière proportionnelle au

nombre de clusters utilisés.

L'évolution du temps d'exécution est similaire entre les versions *B* et *C* : pour 1 thread, les temps de communication augmentent un peu plus que linéairement en passant de 1 à 4 puis 4 à 16 clusters (écarts de $0.10s$ et $4 \times 0.11s$ pour *B*, $0,12s$ et $4 \times 0.15s$ pour *C*) ; pour 16 threads, ils augmentent un peu moins que linéairement (écarts de $0.28s$ et $4 \times 0.21s$ pour *B*, $0.29s$ et $4 \times 0.22s$ pour *C*).

Pour la version *A*, avec 1 thread, l'évolution des temps d'exécution est un peu en dessous d'une augmentation linéaire avec des écarts de $0.33s$ entre 1 et 4 clusters et $4 \times 0,3225s$ entre 4 et 16 clusters. En revanche pour 16 threads, les écarts de 1 à 4 puis 4 à 16 clusters ($0,21s$ et $4 \times 0,425s$) évoluent deux fois plus vite que linéairement ce qui amène à des temps supérieurs à ceux calculés en *B'* en soustrayant l'attente fixe. Une fois de plus nous ne pouvons nous expliquer cette évolution des temps d'exécution.

Les incohérences observées dans les résultats de ce banc de test rendent difficile l'évaluation des temps de communication et de calcul. Nous avons refait cette expérimentation une deuxième fois un autre jour pour nous assurer qu'un dysfonctionnement temporaire du MPPA ne pouvait pas expliquer ces résultats, cependant nous avons obtenu exactement les mêmes temps d'exécution.

Nous pouvons juste constater que le temps de calcul des 1000 *playouts* doit être relativement faible comparé au temps de communication et que l'utilisation de communications synchrones augmente beaucoup les temps de synchronisation, même pour un jeu comme *Tictactoe* ou les *playouts* présentent peu de variation de longueur.

2.8 Évaluation du temps en fonction de la taille des données

Selon les jeux, la taille des positions à transmettre aux clusters pour effectuer les *playouts* est variable. Dans le cas de communications synchrones, si l'on souhaite effectuer un *playout* par thread, la taille du message à envoyer correspond à la taille d'une position multipliée par le nombre de threads. La documentation du MPPA indique qu'aucune latence supplémentaire n'est induite par l'augmentation de la taille des transmissions. Afin de nous en assurer nous avons procédé à une expérimentation.

Nous avons fait varier la taille des messages transmis entre l'hôte et l'I/O puis entre l'I/O et les clusters en définissant des messages de longueur fixe, les positions réellement envoyées n'occupant que les premiers octets du message. Pour éviter que le temps variable nécessaire au calcul des *playouts* ne perturbent les mesures, aucun *playout* n'est calculé et un résultat vide (tout les bits à 0) est retourné. Chaque exécution du programme effectue 1000 *runs*, i.e. 1000 envois de position avec attente du résultat. Pour estimer séparément l'impact de la variation de longueur des messages entre hôte et l'I/O et entre l'I/O et les clusters, les messages sont gardés fixes pour l'une des communications quand l'autre varie. La taille par défaut d'une position

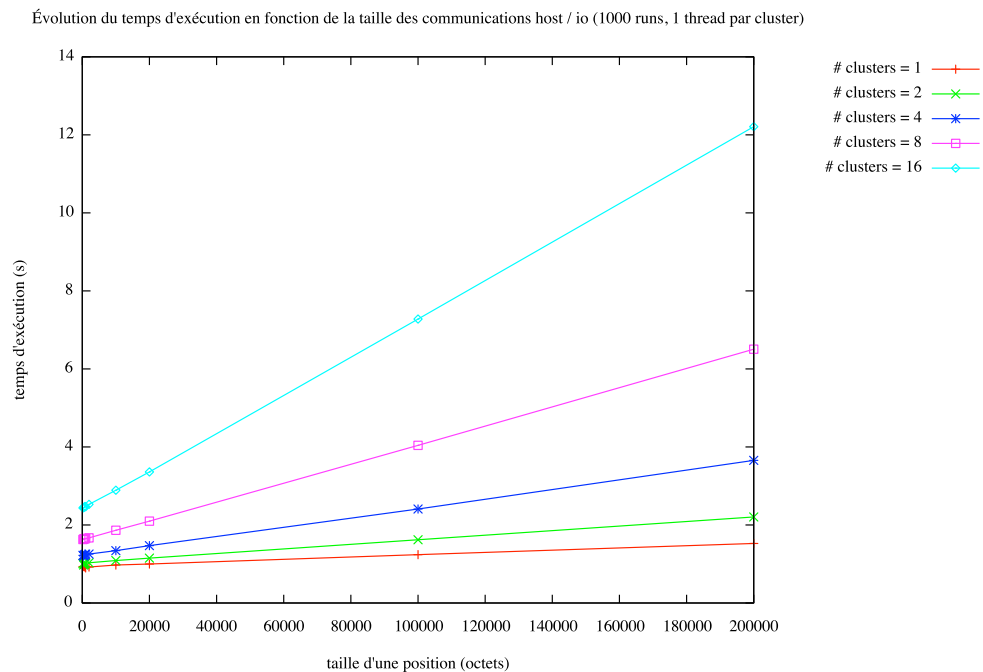


FIGURE 2.22 – Variation du temps d'exécution de l'application, testée sur le jeu de *Tictactoe* pour 1000 runs, sans calculer de *playout*, en faisant varier la taille des communications entre l'hôte et l'I/O. Chaque courbe correspond à une exécution du programme pour 1, 2, 4, 8 et 16 clusters avec 1 thread par cluster. Le passage à l'échelle est assez bon pour les communications entre l'hôte et l'I/O.

est de 200 octets (100 short int représentant les booléens) ; cette taille est à multiplier par le nombre de clusters et de threads par cluster, une position étant envoyée à chaque processeur élémentaire. Les résultats sont donnés sur les figures 2.22, 2.23 et 2.24.

Sur la figure 2.22 nous constatons que le temps des communications entre l'hôte et l'I/O augmente peu quand leur taille augmente. Par exemple pour 1 cluster et 1 thread, le temps de fonctionnement de l'application augmente de 50% quand la taille des communications est multipliée par 10. Pour 16 clusters et 1 thread, le temps de fonctionnement de l'application est multiplié par 3.6 quand la taille des communications est multipliée par 10. Le passage à l'échelle est donc assez bon pour les communications entre l'hôte et l'I/O.

La figure 2.23 montre que les communications entre l'I/O et les clusters présentent également un bon passage à l'échelle. Par exemple pour 1 cluster et 1 thread, le temps d'exécution augmente de 66% quand la taille des communications est multipliée par 10. Pour 16 clusters et 1 thread, le temps de fonctionnement de l'application est multiplié par 3.9 quand la taille des communications est multipliée par 10. Le passage à l'échelle est un peu moins bon que pour les communications entre hôte et I/O, nous supposons que cela est dû au temps nécessaire pour répartir les fragments du message reçu par l'I/O entre les différents clusters.

Sur la figure 2.24 nous constatons qu'en augmentant le nombre de threads par

Évolution du temps d'exécution en fonction de la taille des communications io / clusters (1000 runs, 1 thread par cluster)

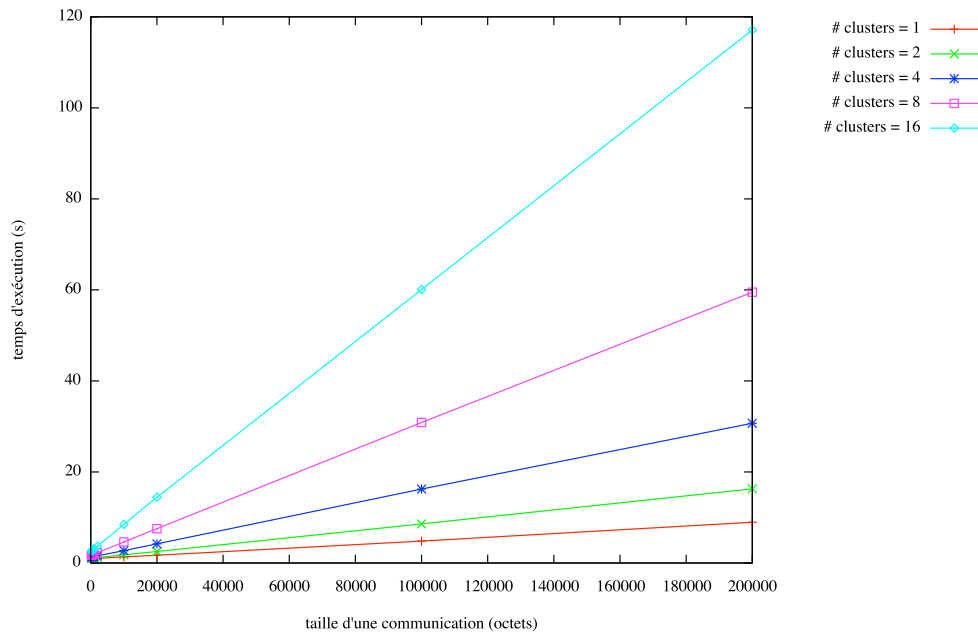


FIGURE 2.23 – Variation du temps d'exécution de l'application, testée sur le jeu de *Tictactoe* pour 1000 runs, sans calculer de *playout*, en faisant varier la taille des communications entre l'I/O et les clusters. Chaque courbe correspond à une exécution du programme pour 1, 2, 4, 8 et 16 clusters avec 1 thread par cluster. Le passage à l'échelle est un peu moins bon que pour les communications entre hôte et I/O, nous supposons que cela est dû au temps nécessaire pour répartir les fragments du message reçu par l'I/O entre les différents clusters.

Évolution du temps d'exécution en fonction de la taille des communications io / clusters (1000 runs, 16 clusters)

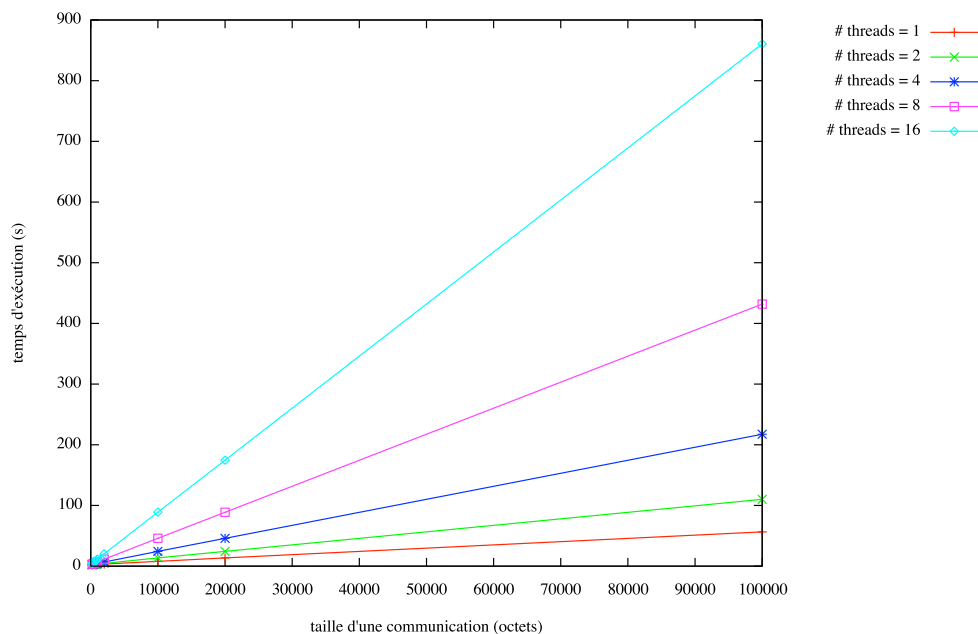


FIGURE 2.24 – Variation du temps d'exécution de l'application testée sur le jeu de *Tictactoe* pour 1000 runs, sans calculer de *playout*, en faisant varier la taille des communications entre l'I/O et les clusters. Chaque courbe correspond à une exécution du programme pour 1, 2, 4, 8 et 16 threads par cluster avec 16 clusters. Le passage à l'échelle reste acceptable bien que les temps de lancement des threads fassent baisser les performances.

clusters, le passage à l'échelle se dégrade encore : pour 16 clusters et 16 threads, le temps d'exécution augmente d'un facteur supérieur à 8 quand la taille d'une position est multipliée par 10.

Parmi les jeux que nous avons testés, une position est codée au maximum par 325 booléens (*CheckersTiny*), ce qui représente 650 octets en reprenant le codage de chaque booléen par un `short int` utilisé par Jean-Noël Vittaut. Dans une dernière expérience, nous avons fait varier la taille des communications, globalement cette fois, pour transmettre les messages d'une taille variant entre 200 et 2000 octets de l'hôte jusqu'aux threads (voir figure 2.25). Nous constatons que l'augmentation des temps de fonctionnement de l'application est moins marquée.

Pour un cluster et un thread, le temps d'exécution augmente d'environ 6% quand la taille des messages est multipliée par 10. En passant le nombre de threads à 16, le temps d'exécution pour des messages de 200 octets est très légèrement supérieur, ce qui correspond au temps de lancement et d'arrêt des threads. Quand la taille des messages est multipliée par 10, le temps d'exécution est multiplié par 2 ce qui constitue un très bon passage à l'échelle.

Pour transmettre des messages de 200 octets à 16 clusters avec 1 thread par cluster, on remarque que le temps d'exécution est deux fois plus important que pour un cluster avec 16 threads, cette augmentation correspond au découpage du message au niveau de l'I/O pour envoyer les données à chaque cluster. Cependant, lorsque la taille des messages est multipliée par 10, le temps d'exécution pour 16 clusters avec 1 thread par cluster n'augmente que de 50% environ.

Pour 16 clusters et 16 threads, l'augmentation du temps d'exécution est par contre beaucoup plus marquée. Le temps de lancement des threads ajouté au temps de répartition des données entre les clusters diminue la qualité du passage à l'échelle. Il reste néanmoins assez satisfaisant puisque le temps d'exécution est multiplié environ par 5 quand la taille des messages est multipliée par 10.

Globalement nous voyons que le passage à l'échelle est assez bon : les temps de communication augmentent plus faiblement que la taille des communications. Avec les portals nous constatons également que la seule limite à la taille des messages est la mémoire disponible dans les clusters.

Nous avons vu dans la section 2.6 qu'il n'est pas possible de gagner du temps en gardant les threads actifs ; cependant en n'ayant pas à partager les messages entre les différents clusters, il serait peut-être possible d'accélérer le fonctionnement de l'application. L'utilisation de communications asynchrones amènerait à cette situation, une position unique étant transmise à la demande de l'I/O à un cluster donné.

L'utilisation d'un message unique de plus grande taille plutôt que plusieurs messages de petite taille est recommandée dans la documentation du *MPPA* mais pour accélérer encore les communications, nous pourrions modifier le codage des positions utilisé par *LeJoueur*. En codant chaque booléen sur un seul bit au lieu d'utiliser un `short int` nous pourrions diviser la taille des messages par 16 et accélérer les communications. Un système de masque binaire permettrait d'encoder les données

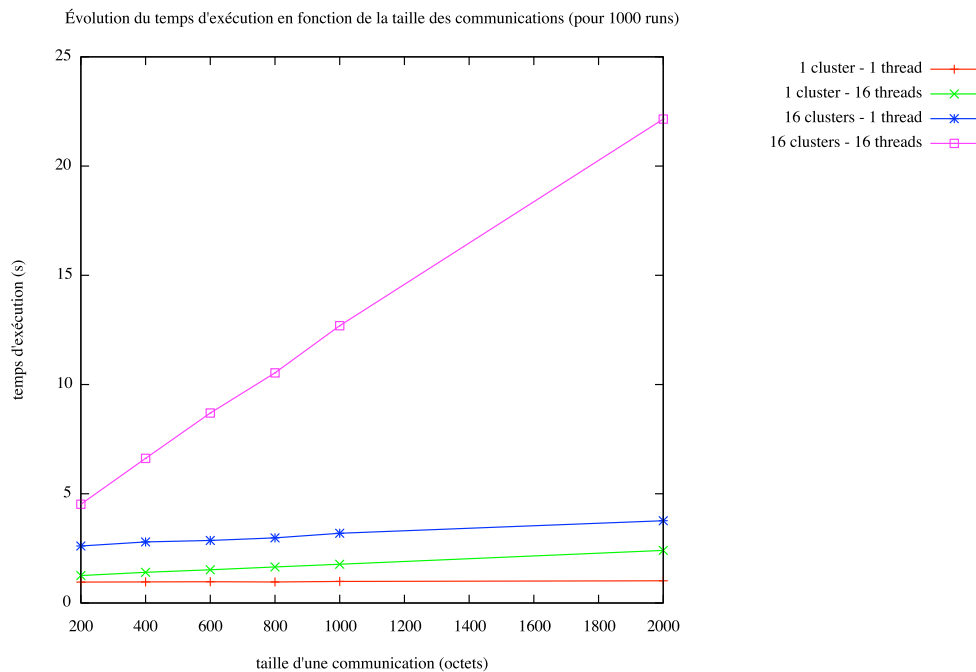


FIGURE 2.25 – Variation du temps d'exécution de l'application testée sur le jeu de *Tictactoe* pour 1000 runs, sans calculer de *playout*, en faisant varier la taille des communications. Chaque courbe correspond à une exécution du programme pour 1 ou 16 clusters et 1 ou 16 threads par cluster. Le passage à l'échelle est assez bon malgré le temps nécessaire au lancement des threads et à la répartition des données entre les clusters.

au niveau de l'hôte et de les décoder dans les threads.

2.9 Problème de la taille des circuits

Durant nos tests avec le *MPPA*, nous nous sommes heurtés à la limitation de la mémoire des clusters. Nous avons pu effectuer des tests sur des jeux simples comme *Tictactoe*, mais l'intérêt de tels jeux est limité. Les jeux intéressants, non triviaux, sont généralement représentés par des *propnets* d'une taille supérieure à 2Mo et qui ne tiennent donc pas dans la mémoire des clusters du *MPPA*. Le tableau de la figure 2.26 montre que les circuits de *ChineseCheckers*, *Breakthrough* et *SheepAndWolf* font respectivement plus de 9Mo, 16Mo et 25Mo.

Comme la majorité du poids est située dans les circuits NEXT et que parfois NEXT est réduit à un seul et unique circuit, il n'est pas envisageable de diviser les circuits pour en communiquer les fragments à différents clusters. La division du calcul d'un *playout* entre différents clusters engendrerait de toute manière des temps de communication trop importants.

Comme nous l'avons vu précédemment (voir section 1.6.3), les règles du jeu sont transformées par *LeJoueur* en circuits. Après avoir été instanciée, chaque règle est codée sous forme d'une liste de pointeurs. Pour transmettre ces données au *MPPA*,

Jeux	Position (# trues)	# TERMINAL (octets)	# LEGAL (octets)	# GOAL (octets)	# NEXT (octets)	# total (octets)
Blocker	48	465 (1860)	98 (392)	18 (72)	354 (1416)	935 (3740)
Tictactoe	29	162 (648)	80 (320)	28 (112)	710 (2840)	980 (3920)
ConnectFour	98	1110 (4440)	73 (292)	43 (172)	986 (3944)	2212 (8848)
EightPuzzle	142	20 (80)	74 (296)	249 (996)	27929 (111716)	28272 (113088)
CheckersTiny	325	2826 (11304)	8 (32)	1354 (5416)	145224 (580896)	149412 (597648)
ChineseCheckers	211	40 (160)	8193 (32772)	135 (540)	2327414 (9309656)	2335782 (9343128)
Breakthrough	130	448 (1792)	1869 (7476)	14 (56)	4088840 (16355360)	4091171 (16364684)
SheepAndWolf	194	8360 (33440)	1478 (5912)	14 (56)	6279176 (25116704)	6289028 (25156112)

FIGURE 2.26 – Tailles des positions à transmettre au MPPA pour chaque *playout* (en nombre de booléens) et taille des groupes de circuits. Le détail des quatre groupes de circuits (TERMINAL, LEGAL, GOAL et NEXT) montre que le poids n'est pas uniformément réparti et que pour la plupart des jeux les règles nécessaires au calcul de l'état suivant sont bien plus nombreuses. La taille totale de certains circuits dépasse les 2Mo disponibles sur les clusters du MPPA.

nous avons remplacé ces pointeurs par des index et pour pouvoir coder les négations nous avons multiplié chaque index par 2 et ajouté 1 aux faits négatifs. Toutes les règles sont sérialisées en ajoutant devant chacune une valeur indiquant la taille de la règle.

En examinant ces règles nous avons constaté qu'il était possible de les factoriser ou d'extraire des sous-expressions communes à plusieurs règles¹². La figure 2.27 (3) montre un exemple de circuit constitué de 10 règles. La factorisation de ces règles (voir figure 2.27 (4)) et l'extraction d'une sous-expression commune (voir figure 2.27 (5)) permet de réduire la taille totale de ce circuit de 20%.

Pour les jeux *ChineseCheckers*, *Breakthrough* et *SheepAndWolf* qui ne tiennent pas dans la mémoire du MPPA, une réduction de 20% de la taille des circuits ne serait pas suffisante (voir figure 2.26), cependant la présence de nombreuses sous-expressions communes peut améliorer la réduction.

La difficulté posée par une automatisation de cette simplification réside dans le choix de la bonne factorisation quand plusieurs factorisations sont possibles et le choix de la bonne sous-expression à extraire pour minimiser la taille du circuit. Prenons par exemple les expressions suivantes nécessitant 34 valeurs selon notre codage :

$$\begin{aligned}
 x &= a.b.f + b.g + a.h + a.i \\
 y &= c.b.f + b.g + c.h + c.i
 \end{aligned}$$

En factorisant par le terme b puis par a , on obtient un circuit codé par 36 valeurs :

12. Par exemple, une seule action *does* pouvant être réalisée pour chaque joueur, il est souvent possible de factoriser par les *does*.

(1)	index	0	1	2	3	4	5	6	7	8	9	10	11	12
	symbole	a	b	c	d	e	f	g	w	x	y	h	z	i
(2)	symbole	a	b	c	d	e	f	not(g)	w	x	y	h	z	i
	codage	0	2	4	6	8	10	13	14	16	18	20	22	24
expression								règles		codage			taille du circuit	
(3)	$a = b.c + b.d + b.e + b.f + b.g$ $w = x.c + x.d + x.e + x.f + x.y$							a :- b, c.		3, 0, 2, 4			(40)	
								a :- b, d.		3, 0, 2, 6				
								a :- b, e.		3, 0, 2, 8				
								a :- b, f.		3, 0, 2, 10				
								a :- b, not(g).		3, 0, 2, 13				
								w :- x, c.		3, 14, 16, 4				
								w :- x, d.		3, 14, 16, 6				
								w :- x, e.		3, 14, 16, 8				
								w :- x, f.		3, 14, 16, 10				
								w :- x, y.		3, 14, 16, 18				
(4)	$h = c + d + e + f + g$ $a = b.h$ $z = c + d + e + f + y$ $w = x.z$							h :- c.		2, 20, 4			(38)	
								h :- d.		2, 20, 6				
								h :- e.		2, 20, 8				
								h :- f.		2, 20, 10				
								h :- not(g).		2, 20, 13				
								a :- b, h.		3, 0, 2, 20				
								z :- c.		2, 22, 4				
								z :- d.		2, 22, 6				
								z :- e.		2, 22, 8				
								z :- f.		2, 22, 10				
z :- y.		2, 22, 18												
w = x, z		3, 14, 16, 22												
(5)	$i = c + d + e + f$ $h = i + g$ $a = b.h$ $z = i + y$ $w = x.z$							i :- c.		2, 24, 4			(32)	
								i :- d.		2, 24, 6				
								i :- e.		2, 24, 8				
								i :- f.		2, 24, 10				
								h :- i.		2, 20, 24				
								h :- not(g).		2, 20, 13				
								a :- b, h.		3, 0, 2, 20				
								z :- i.		2, 22, 24				
								z :- y		2, 22, 18				
								w :- x, z		3, 14, 16, 22				

FIGURE 2.27 – Cette figure représente la manière dont les circuits peuvent être simplifiés. La table (1) en haut présente les faits (symboles) utilisés avec leurs index respectifs. Les faits en rouges sont ceux créés lors de la simplification du circuit. Dans la table (2) figure le code correspondant à chaque symbole : l'index multiplié par 2, plus 1 pour signaler la négation. Dans la section (3) de la table du bas, les deux expressions a et w sont représentées par 10 règles : ce circuit nécessite 40 valeurs pour être codé. En factorisant respectivement par b et par x , le circuit en (4) peut alors être codé avec 38 valeurs. En extrayant la sous-expression commune i , le circuit en (5) peut être réduit à 32 valeurs.

$$\begin{aligned}m &= a.f + g \\n &= h + i \\o &= c.f + g \\x &= b.m + a.n \\y &= b.o + c.n\end{aligned}$$

Alors qu'en factorisant d'abord par a , le circuit peut être réduit à 26 valeurs :

$$\begin{aligned}l &= b.f + h + i \\x &= a.l + b.g \\y &= c.l + b.g\end{aligned}$$

Pour obtenir une simplification optimale, une exploration stochastique de l'espace des solutions possibles nous semble nécessaire. Nous avons choisi d'utiliser la technique du recuit simulé car c'est une méthode simple et dont l'efficacité pour résoudre des problèmes d'optimisation a été démontrée.

Dans le chapitre suivant nous présentons les principes du recuit simulé. Ensuite, dans un ultime chapitre, nous présentons notre implémentation de l'algorithme de recuit simulé dans le but de réduire la taille des circuits et les résultats obtenus.

3 Le recuit simulé

Le recuit simulé est une méta-heuristique d'optimisation combinatoire issue de la mécanique statistique qui s'inspire de la technique du recuit thermique. Le principe du recuit est de chauffer puis refroidir lentement une substance pour obtenir, à la suite d'une réorganisation progressive des atomes constituant la matière, une solide structure cristalline.

L'algorithme de recuit simulé a été proposé pour la première fois par [Metropolis et al. \[1953\]](#) sous la forme d'une simulation de Monte Carlo du comportement d'une collection d'atomes à une température donnée, puis généralisé par [Kirkpatrick et al. \[1983\]](#) et [Černý \[1985\]](#) pour la résolution de problèmes d'optimisation. L'algorithme de recuit simulé modélise la recherche par une chaîne de Markov, i.e. une chaîne d'états où l'état suivant ne dépend que de l'état courant. Il consiste à autoriser sous certaines conditions une dégradation de la solution d'un problème, ceci afin d'éviter une convergence prématurée vers un minimum local et de permettre l'optimisation de la solution jusqu'au minimum global.

Dans les années 80, le recuit simulé a été très populaire pour résoudre des problèmes d'optimisation combinatoire grâce à son efficacité et sa simplicité de mise en œuvre. Il a été appliqué au partitionnement de graphe ou de nombres et coloration de graphes [[Johnson et al., 1989, 1991](#)], à la restauration d'images bruitées [[Geman et Geman, 1984](#)], au problème du voyageur de commerce [[Kirkpatrick et al., 1983](#); [Černý, 1985](#)], au problème des plannings scolaires [[Abramson et al., 1997](#)], au partitionnement, placement des composants et câblage de systèmes électroniques (VLSI) [[Kirkpatrick et al., 1983](#); [Sechen et Sangiovanni-Vicentelli, 1985](#)]. Par la suite il a été étendu pour pouvoir également traiter des problèmes d'optimisation continue [[Corana et al., 1987](#); [Ali et al., 1996](#); [Dekkers et Aarts, 1991](#); [Locatelli, 2000](#); [Özdamar et Demirhan, 2000](#)]. [Ingber \[1993\]](#) et [Suman et Kumar \[2006\]](#) présentent de nombreuses applications du recuit simulé.

La convergence de la technique du recuit simulé a été démontrée par différents auteurs [[Aarts et al., 1997](#); [Van Laarhoven et Aarts, 1987](#)] en s'appuyant sur le formalisme des chaînes de Markov homogènes ou in-homogènes. Les propriétés mathématiques concernant cette convergence vers une solution optimale globale ont été démontrées et discutées par [Pincus \[1970\]](#); [Hajek \[1985, 1988\]](#); [Bertsimas et Tsitsiklis \[1993\]](#). La convergence pour des problèmes d'optimisation continue a été également traitée par [Dekkers et Aarts \[1991\]](#); [Gelfand \[1987\]](#); [Gelfand et Mitter \[1993\]](#).

Geman et Geman [1984] montrent qu'il y a une garantie de convergence vers la solution optimale en suivant un programme de refroidissement lent :

$$T_n = c/\log(1 + n) \quad (3.1)$$

avec T_n la température à l'itération n et c représentant la profondeur maximum d'une *cuvette* ne contenant pas le minimum global [Hajek, 1988]. Ce programme permet au processus de recuit d'atteindre un équilibre énergétique plus rapidement que la température ne varie [Hajek, 1985]. Cependant ces conditions de convergence sont malheureusement asymptotiques et sont donc en pratiques irréalisables.

Plusieurs schémas pratiques d'application du recuit simulé ont été proposés. Ces schémas ne garantissent pas l'obtention d'une solution optimale, cependant leurs applications ont montré qu'il était possible d'obtenir une solution assez proche, voire égale à l'optimum global dans un temps fini [Rajasekaran, 1990; Lecchini-Visintini *et al.*, 2007].

Après une présentation du recuit thermique, nous expliquons l'algorithme du recuit simulé. Nous décrivons ensuite différents programmes de recuit proposés dans la littérature.

3.1 Le recuit thermique

En thermodynamique, une substance physique placée à une température élevée passe à l'état liquide. A cette température, les atomes qui la composent s'agencent librement. En abaissant la température, les atomes se réorganisent pour atteindre un état d'équilibre et d'énergie moindre. En la refroidissant totalement, la substance se fige.

Selon le schéma de décroissance de la température adopté il est possible d'obtenir différents états de la substance. Avec une décroissance rapide, la substance peut se figer dans un état méta-stable qui correspond à un minimum local de l'énergie : c'est la technique de la trempe. Avec une décroissance lente, la substance peut atteindre une structure cristalline qui correspond à un état minimum global de l'énergie.

Pour obtenir l'état d'énergie minimale optimum, on démarre à une température élevée. A chaque étape du recuit, à une température T , on laisse le système atteindre un équilibre thermique. La probabilité P de se trouver dans un état d'énergie E est donnée par la distribution de Boltzmann :

$$P(E) = \frac{e^{-E/K_B T}}{Z(T)} \quad (3.2)$$

$Z(T)$ est un facteur de normalisation, nommé fonction de partition et représen-

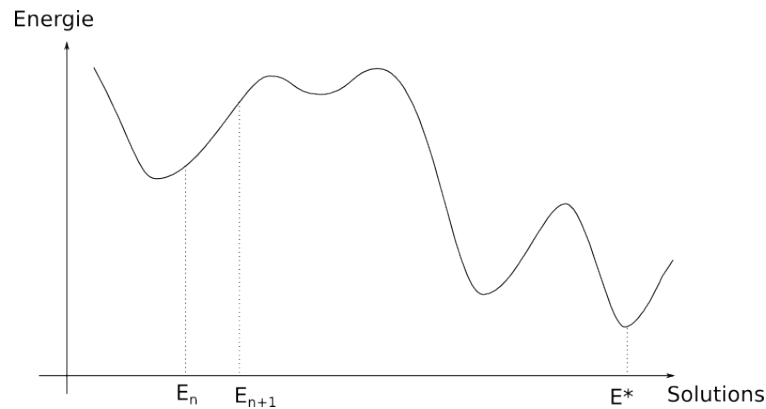


FIGURE 3.1 – L’acceptation de transformations augmentant l’énergie du système est nécessaire pour passer du point d’énergie E_n à E_{n+1} , se rapprocher de la solution optimale d’énergie E^* et échapper à un minimum local.

tant les propriétés statistiques d’un système à l’équilibre thermodynamique.

K_B est la constante de Boltzmann, elle peut s’interpréter comme le facteur de proportionnalité reliant la température d’un système à son énergie thermique.

Plus la température baisse, plus la probabilité d’existence d’état d’énergie plus faible augmente jusqu’à ce qu’à une température nulle, seuls les états d’énergie minimale aient une probabilité non nulle. Néanmoins, si la température est abaissée trop vite, des imperfections peuvent subsister dans la structure cristalline. Il est alors nécessaire de réchauffer à nouveau la substance pour que les atomes puissent à nouveau se réorganiser et sortir du minimum local atteint pour passer à un état d’énergie minimal optimal lors d’une nouvelle phase de refroidissement lent. Plusieurs séances de recuit peuvent être nécessaires, en marquant des paliers de température de durée suffisante, pour permettre d’atteindre une structure stable d’énergie minimale.

3.2 Le recuit simulé

Le recuit simulé s’inspire du recuit thermique pour la résolution de problèmes d’optimisation combinatoire complexes. Chaque état du système est une solution possible du problème. L’énergie du système est représentée par une fonction objectif représentant le coût d’une solution. La température est un paramètre de contrôle qui permet d’amener progressivement le système vers un état de coût minimal. L’idée est de simuler numériquement une phase de recuit thermique (souvent unique) par une descente lente du paramètre de contrôle (la température).

L’algorithme proposé par [Metropolis *et al.* \[1953\]](#) s’appuie sur une règle déterminant le passage du système d’un état à un autre, connue sous le nom de *critère de Monte Carlo*. Le recuit simulé consiste à itérer un grand nombre de fois et à proposer à chaque itération une modification aléatoire de l’état courant du système qui change l’énergie (ou le coût évalué) d’une quantité ΔE (voir figure 3.1). Si l’énergie

```

Choisir une solution initiale  $S$  à optimiser
Choisir une température initiale  $T_0 > 0$ 
Tant que la condition d'arrêt n'est pas atteinte pour  $i$  de 1 à  $n$ 
  Itérer à la température  $T_i$ 
    Choisir une transformation à appliquer à  $S$  pour obtenir  $S'$ 
    Soit  $\Delta E$  la différence d'énergie entre  $S$  et  $S'$ 
    Si  $\Delta E \leq 0$  :  $S = S'$ 
    Si  $\Delta E > 0$  :  $S = S'$  avec la probabilité  $e^{-\Delta E/T_i}$ 
  Calculer la température  $T_{i+1}$ 

```

FIGURE 3.2 – Algorithme du recuit simulé

diminue, on accepte automatiquement le changement et le nouvel état devient l'état courant. Sinon le changement est effectué avec la probabilité P suivante :

$$P = e^{-\Delta E/T} \quad (3.3)$$

Ainsi, à une température T donnée, la probabilité d'accepter un changement augmentant le coût de la solution courante est inversement proportionnel à l'ampleur de cette augmentation. Plus la température baisse, plus l'ampleur de l'augmentation doit être faible pour avoir une probabilité d'acceptation non nulle. Quand la température s'approche de zéro, seuls les changements permettant une baisse du coût de la solution ont une probabilité supérieure à zéro.

3.3 Programme de recuit

Kirkpatrick *et al.* [1983] et Černý [1985] ont généralisé le principe du recuit simulé en proposant un programme de recuit, c'est à dire une baisse progressive de la température à mesure des itérations selon un *schéma* ou *programme* de recuit (voir Figure 3.2). A chaque palier de température le système tend progressivement vers un équilibre.

Toute implémentation du recuit simulé nécessite donc de définir un programme de recuit, i.e. un schéma d'évolution de la température. La température est généralement initialisée à une valeur élevée de manière à accepter la majorité des transformations et éviter une convergence prématurée vers un minimum local. Une stratégie est alors nécessaire pour la réduction de la température et le choix du nombre d'itérations effectuées à chaque palier (longueur des chaînes de Markov). Enfin, le programme de recuit définit un critère d'arrêt pour mettre fin à la recherche.

Le programme de recuit ou schéma de refroidissement définit donc la manière de choisir les paramètres de base :

- La valeur initiale de la température
- La longueur des chaînes de Markov, i.e. le critère de changement de palier de température

- La loi de décroissance de la température
- La condition d'arrêt

3.3.1 Température initiale

En démarrant avec une température élevée, la recherche consiste plus ou moins en une exploration aléatoire de l'espace des solutions. D'un autre côté, une température initiale basse revient à une optimisation locale aboutissant à une solution non optimale. Il est donc nécessaire de choisir une température initiale T_0 de manière à ce qu'au départ tous les changements d'état du système soit acceptés [Kirkpatrick *et al.*, 1983] c'est à dire tel que $e^{-\Delta E/T_0} \approx 1$. Cependant, ce choix peut amener un coût important en calcul et plusieurs manières de choisir la température initiale ont été proposées qui permettent de limiter le temps passé dans une exploration aléatoire ; nous allons maintenant en présenter quelques unes.

Dans le programme de recuit introduit par Van Laarhoven et Aarts [Van Laarhoven et Aarts, 1987; Suman et Kumar, 2006; Dekkers et Aarts, 1991], la température initiale T_0 est choisie de manière à ce que presque toutes les transformations proposées soient acceptées. Pour déterminer cette température, un nombre de transformations m_0 suffisamment important est généré durant une expérimentation préliminaire. Le choix de T_0 est fondée sur le ratio d'acceptation initial χ_0 souhaité et sur la variation moyenne de la fonction d'évaluation ΔE^+ :

$$T_0 = \Delta E^+ \left(\ln \frac{m_2}{m_2 \chi_0 + (1 - \chi_0) m_1} \right)^{-1} \quad (3.4)$$

avec m_1 et m_2 le nombre de solutions qui font décroître ou croître l'énergie dans l'expérimentation préliminaire, ΔE^+ la croissance moyenne de l'énergie pour les transformations augmentant le coût de la solution, χ_0 le nombre de transformations acceptées sur le nombre de transformations proposées.

Cardoso *et al.* [1994] proposent une démarche similaire avec :

$$\chi_0 = \frac{m_1 + m_2 e^{-\Delta E^+/T_0}}{m_1 + m_2} = 0.95 \quad (3.5)$$

Locatelli [2000] propose de choisir T_0 de manière à accepter 60% des transformations proposées augmentant le coût de la solution. Il suggère également d'utiliser deux phases de recuit avec une température initiale élevée pour la première, pour favoriser l'exploration de l'espace des solutions, puis une température plus basse pour raffiner la meilleure solution trouvée lors de la première phase.

Siarry *et al.* [1997] proposent une méthode expérimentale simple pour déterminer la température de départ. Pour commencer, il faut choisir un taux initial d'acceptation τ_0 des transformations augmentant le coût de la solution : Siarry *et al.* [1997] indiquent que généralement $\tau_0 = 0.5$. D'après Ghédira [2007] le taux initial

d'acceptation dépend de la qualité supposée de la configuration initiale : si la qualité est médiocre $\tau_0 = 50\%$ (démarrage à haute température), si la qualité est bonne $\tau_0 = 20\%$ (démarrage à basse température)¹. Une fois la valeur de τ_0 fixée, [Siarry et al. \[1997\]](#) proposent de faire de 50 à 100 transformations au hasard et évaluer la moyenne $\langle \Delta E \rangle$ des variations de l'énergie. Il est possible alors de déduire T_0 de la relation $\tau_0 = e^{-\langle \Delta E \rangle / T_0}$, c'est à dire $T_0 = -\langle \Delta E \rangle / \log(\tau_0)$.

[Aarts et al. \[1997\]](#) proposent de choisir $T_0 = \Delta E_{max}$ avec ΔE_{max} la différence maximale d'énergie entre deux solutions. Cette valeur étant coûteuse à calculer de manière exacte, elle peut être estimée durant une recherche préliminaire en choisissant une petite valeur initiale pour T_0 et en exécutant un nombre N de transitions. Si le rapport χ entre le nombre de transformations acceptées et le nombre de transformations proposées est inférieur à une valeur donnée χ_0 , alors on multiplie la valeur de T_0 par une constante supérieure à 1 et on recommence jusqu'à ce que le ratio χ soit supérieur à χ_0 . Ils indiquent que la valeur de χ_0 est généralement comprise entre 0.9 et 0.99. [Van Laarhoven et Aarts \[1987\]](#) et [Geman et Geman \[1984\]](#) indiquent que le choix de $T_0 = \Delta E_{max}$ peut amener à une convergence inutilement longue de l'algorithme.

[Suman et Kumar \[2006\]](#) indiquent également qu'une méthode simple pour choisir la température initiale (chez Kouvelis and Chiang, 1992) est de la déterminer en fonction de la probabilité P d'acceptation souhaitée entre 0.5 et 0.95 :

$$P = e^{-\Delta E / T} \quad (3.6)$$

[Johnson et al. \[1989\]](#) indiquent qu'un taux d'acceptation initial de 40%, déterminé expérimentalement, permet de réduire la durée de la recherche sans sacrifier la qualité des solutions trouvées. Dans l'implémentation originale de Kirkpatrick, une option nommée *cutoff*, fondée sur l'idée que c'est le nombre de modifications acceptées plus que le nombre d'itérations qui est important, est utilisée pour supprimer les itérations inutiles au début du processus de recuit. [Johnson et al. \[1989\]](#) montrent que démarrer à une température plus basse en utilisant un taux d'acceptation initial de 40% donne des résultats globalement équivalents.

3.3.2 Longueur des paliers

Pour atteindre un état d'équilibre énergétique à chaque palier de température un nombre suffisant de transformations doit être appliqué, i.e. un nombre suffisant d'itérations doit être effectué. La théorie indique que ce nombre d'itérations (longueur de la chaîne de Markov) doit être exponentiellement proportionnel à la taille du problème [[Aarts et al., 1997](#)] ce qui est inapplicable en pratique. Néanmoins, ce nombre d'itérations peut être fixé a priori de manière à être proportionnel à la taille

1. [Ghédira \[2007\]](#) laisse entendre que cette information est donnée par [Siarry et al. \[1997\]](#) mais nous ne l'avons pas retrouvée dans leur article.

du voisinage de solutions qui peuvent être atteintes, à une transformation près, à chaque position [Aarts *et al.*, 1997].

Deux stratégies sont applicables pour limiter la longueur des paliers : statique ou adaptative. Une stratégie statique consiste à déterminer au préalable le nombre d'itérations effectuées à chaque palier de température. Plus ce nombre d'itérations est important, plus le nombre d'états voisins visités est important et le résultat meilleur, au prix cependant d'un coût en calcul plus élevé.

Aarts *et al.* [1997] proposent de faire en sorte qu'un minimum de transformations soit accepté pour chaque palier de température. Cependant cela implique que les chaînes de Markov seront de plus en plus longues à mesure que la température tend vers 0. Pour limiter la taille des paliers, ils proposent de fixer une valeur seuil déterminée à priori.

La taille du voisinage est fortement dépendante du problème et peut varier au cours de la recherche. Une approche adaptative (Non-Equilibrium Simulated Annealing, NESA) proposée par Cardoso *et al.* [1994] consiste à passer au palier suivant de température dès qu'un état d'énergie plus basse est atteint, sans attendre donc d'atteindre un équilibre énergétique. Cette démarche accélère beaucoup le temps de calcul tout en permettant d'atteindre un minimum local très proche de la solution optimale.

Ali *et al.* [1996] utilisent eux aussi une approche adaptative fondée sur la mémorisation des solutions de plus grande et de plus basse énergie, E_h et E_l . Comme Dekkers et Aarts [1991], ils utilisent un nombre d'itérations fonction de la dimension N du problème traité et de la longueur L_0 de la chaîne de Markov initiale :

$$L_n = L_0 \cdot N \quad (3.7)$$

avec N le plus grand entier inférieur à $1 - e^{-(E_h - E_l)/E_h}$ ².

Johnson *et al.* [1989] utilisent aussi un nombre d'itérations variable à chaque palier de température et fonction de la dimension N du voisinage estimé d'une solution. Cependant le paramètre L_0 est remplacé par un paramètre SIZEFACTOR. Dans une étude expérimentale ils font varier la valeur de SIZEFACTOR et étudient son influence sur les résultats, associée à une variation du taux de décroissance de la température : ils déduisent que la longueur des paliers et la vitesse de décroissance de la température sont fortement liées. Pour obtenir un équilibre énergétique il est possible d'associer de longs paliers à une décroissance rapide de la température ou inversement des paliers très courts avec une décroissance très lente de la température. Cette dernière option semble préférable car elle apporte des résultats similaires en un temps de recherche plus court.

2. Cette définition de N est donnée par Talbi [2009]. Il indique que cette information est tirée de Ali *et al.* [1996] mais nous ne l'avons pas retrouvée dans leur article, Ali *et al.* [1996] indiquent juste que N correspond à la dimension du problème

3.3.3 Schéma de décroissance de la température

Geman et Geman [1984] font la démonstration que la température doit décroître de manière logarithmique pour assurer la convergence de l'algorithme vers une solution optimale globale. La progression doit être effectuée selon la règle :

$$T_n = c/\log(1 + n) \quad (3.8)$$

avec T_n la température à l'itération n et c une constante indépendante de n .

Cette progression est bien trop lente pour être effectivement applicable. Ainsi, pour une application pratique du recuit simulé, plusieurs schémas de décroissance de la température ont été proposés, aucun ne garantissant l'obtention d'un minimum global.

Décroissance linéaire : La température suivante est définie par $T_{n+1} = T_n - \alpha$ avec α une constante comprise entre 0 et T_0 .

Décroissance géométrique : Ce schéma est sans doute le plus utilisé dans les applications du recuit simulé. La décroissance de la température est définie par $T_{n+1} = \alpha T_n$ avec α une constante inférieure à 1, généralement comprise entre 0.9 [Kirkpatrick *et al.*, 1983] et 0.99.

Sechen et Sangiovanni-Vicentelli [1985] utilisent une variante de ce schéma avec une valeur moindre de α (0.80) pendant les phases initiales et finales de la recherche de manière à passer moins de temps sur ces périodes où l'énergie varie peu. Johnson *et al.* [1989] présentent 0.95 comme un ratio de refroidissement *normal*.

Décroissance rapide : La décroissance la plus rapide est une décroissance exponentielle donnée par $T_n = T_0 e^{-Cn}$ avec C constante. Bien qu'il n'y ait pas de preuve rigoureuse que ce programme permette la convergence vers une solution optimale, de bon arguments heuristiques ont été avancés dans ce sens [Ingber, 1989; Suman et Kumar, 2006].

Décroissance très lente : Comme Johnson *et al.* [1989], Talbi [2009] indique que pour assurer une bonne convergence, le schéma de décroissance de la température doit utiliser un grand nombre d'itérations sur quelques paliers de température, ou bien peu d'itérations à un grand nombre de températures différentes. Il est possible d'utiliser le schéma de décroissance très lent suivant ³, avec une seule itération pour chacune des températures :

3. Talbi [2009] indique que ce schéma est proposé par Lundy et Mees, *Convergence of an Annealing Algorithm* (1986).

$$T_{n+1} = \frac{T_n}{1 + \beta T_n} \quad (3.9)$$

avec $\beta = T_0 - T_f / ((L - 1)T_0T_f)$ et T_f la température finale.

Décroissance non-monotone : [Hu et al. \[1995\]](#) proposent un schéma d'évolution dans lequel la température est abaissée pour chaque amélioration de la solution et très légèrement rehaussée à chaque rejet d'une transformation augmentant le coût de la solution.

[Abramson et al. \[1997\]](#) propose plusieurs schémas non-monotones pour l'évolution de la température. Le premier consiste à remonter la température au bout d'un certain nombre de chaînes de Markov effectuées sans améliorations afin d'échapper à un minimum local. Cependant cette technique peut induire des cycles. Une autre approche consiste à mettre la température à jour selon le schéma :

$$T_{n+1} = \max[T_r/2, T_b] \quad (3.10)$$

avec T_r la dernière température utilisée lors d'un réchauffement (ou la température initiale lors du premier réchauffement), et T_b la température correspondant à la meilleure solution trouvée.

[Boese et Kahng \[1994\]](#) proposent, dans l'optique d'améliorer la meilleure solution trouvée plutôt que la solution courante, un programme de recuit non-monotone où chaque palier est démarré à partir de la meilleure solution trouvée.

Décroissance dynamique : La plupart des programmes de refroidissement sont statiques dans le sens où ils sont déterminés à priori. Une décroissance dynamique tient compte d'informations obtenues pendant la recherche pour ajuster la température.

La décroissance de la température proposée par Aarst et Van Laarhoven en 1985 [[Ali et al., 1996](#); [Cardoso et al., 1994](#); [Suman et Kumar, 2006](#); [Dekkers et Aarts, 1991](#)] est calculée selon l'expression suivante :

$$T_{n+1} = \frac{T_n}{1 + \frac{T_n \ln(1+\delta)}{3\sigma}} \quad (3.11)$$

avec σ la déviation standard de l'énergie entre les points de la chaîne de Markov à la température courante T_n et δ le paramètre de distance, une constante déterminant la décroissance de la température T et représentant la proximité désirée avec l'équilibre. Une faible valeur de δ (< 1) permet une convergence lente alors qu'une valeur élevée (> 1) précipite la convergence vers un minimum local.

Siarry *et al.* [1997] proposent de déterminer une décroissance rapide ou lente de la température en fonction du ratio entre les valeurs minimale et moyenne de l'énergie durant le palier précédent. Ceci permet de faire décroître rapidement la température au début de la recherche puis plus lentement à l'approche d'une transition entre un état instable et un état d'équilibre.

Comme raffinement de leur schéma de décroissance non-monotone, Abramson *et al.* [1997] proposent d'ajuster la température en fonction de l'évolution de la qualité des solutions trouvées.

Dans l'algorithme de recuit simulé adaptatif (ASA) [Ingber, 1997, 1993, 1996; Chen et Luk, 1999; Locatelli, 2000], un schéma de refroidissement adaptatif ajuste la température pour maintenir la recherche proche de l'équilibre tout en minimisant le nombre d'itérations pour atteindre cet équilibre. Le taux de décroissance de la température s'ajuste en fonction de l'évolution de la recherche [Suman et Kumar, 2006]. La température est ajustée pour favoriser l'exploration pendant la phase initiale puis diriger la recherche vers la zone la plus favorable par la suite.

Lam et Delosme [1988] proposent un programme de recuit encore plus efficace [Aarts *et al.*, 1997] qui est fondé sur l'optimisation du schéma de décroissance de la température de manière à maintenir en permanence un quasi-équilibre. Cependant la complexité de ce schéma le rend difficile à utiliser en pratique [Aarts *et al.*, 1997].

Décroissance programmée : Nahar *et al.* [1985] proposent de fixer le nombre d'étapes à une valeur K et les valeurs de la température à chacune de ces étapes est déterminée expérimentalement.

Après comparaison des résultats obtenus avec différents schémas de refroidissement, Johnson *et al.* [1989] concluent : « *There seems no reason to replace the standard geometric cooling method by any of the nonadaptive alternatives we have examined (logarithmic cooling, linear temperature cooling, etc.)* ».

3.3.4 Le critère d'arrêt

La séance de recuit simulé peut être stoppée au bout d'un nombre prédéterminé d'itérations à chaque palier de température, jusqu'à atteindre une température minimale déterminée a priori. Elle peut être également stoppée quand plus aucune modification du système n'est possible : les transformations augmentant le coût sont systématiquement refusées et aucune amélioration n'est possible.

Dekkers et Aarts [1991] proposent un critère d'arrêt introduit par Van Laarhoven et Aarts (1985) et fondé sur l'idée que la moyenne de l'énergie du système décroît en même temps que la température et converge vers un minimum quand T tend vers 0. Le critère d'arrêt est alors évalué par :

$$\frac{\Delta\langle E_n \rangle}{\Delta T_n} \frac{T_n}{\langle E_0 \rangle} < \epsilon \quad (3.12)$$

avec $\langle E_0 \rangle$ la valeur moyenne des points situés dans la chaîne de Markov initiale, $\Delta\langle E_n \rangle$ la moyenne de l'énergie sur plusieurs chaînes pour atténuer les fluctuations de la fonction d'évaluation et ϵ , le paramètre d'arrêt, un petit nombre réel.

Ingber [1993] propose de stopper la recherche quand le taux d'acceptation tombe en dessous d'un seuil fixé a priori. Dans [Johnson *et al.*, 1989], à chaque température, si le pourcentage de transformations acceptées tombe en dessous d'un tel seuil, un compteur est augmenté de 1. Si une solution de plus basse énergie que la meilleure solution trouvée est atteinte, le compteur est réinitialisé à 0. L'algorithme est stoppé quand le compteur atteint la valeur limite 5 fixée a priori. Le seuil choisi expérimentalement par Johnson *et al.* [1989] est de 2%, il constitue un bon compromis pour préserver la qualité de la solution tout en limitant la durée de la recherche.

Lorsque la température tend vers zéro, les transformations sont acceptées avec une probabilité décroissante, les chaînes de Markov sont alors très longues pour un gain peu significatif. Comme peu d'améliorations sont apportées pendant les derniers paliers de température, Cardoso *et al.* [1994] associent à leur schéma de convergence rapide de la température (voir section 3.3.3) un critère d'arrêt fondé sur le gradient de la fonction d'évaluation de l'énergie. Quand la variation de l'énergie durant une période de 1000 itérations tombe au dessous d'une valeur seuil déterminée a priori, la recherche est stoppée. Ceci implique l'obtention d'une solution non optimale bien que proche de l'optimum global, mais permet d'économiser beaucoup de temps de calcul.

Ali *et al.* [1996] proposent plus simplement de stopper le processus si aucun changement n'est intervenu durant deux chaînes de Markov, i.e. deux paliers de température successifs. Aarts *et al.* [1997] proposent également de stopper au bout de N chaînes consécutives sans variation de la fonction d'évaluation.

4 Utilisation du recuit pour la simplification des circuits

Dans le but de réduire la taille des circuits à transmettre au *MPPA*, nous avons implémenté en C++ un algorithme de recuit simulé. Comme l'indique [Johnson *et al.*, 1989] la mise en œuvre de cet algorithme pour traiter un problème particulier nécessite de faire différents choix d'implémentation :

« The first problem faced by someone preparing to use or test simulated annealing is that the procedure is more an approach than a specific algorithm. Even if we abide by the basic outline (...) we still must make a variety of choices for the values of the parameters and the meanings of the undefined terms. The choices fall into two classes : those that are problem-specific and those that are generic to the annealing process. »

Dans un premier temps nous décrivons la manière dont nous avons représenté les données sous forme de graphe et présentons notre choix de garantir l'unicité des nœuds de ce graphe. Nous expliquons ensuite comment nous évaluons une solution et quels paramètres génériques nous avons choisis pour notre programme de recuit. Enfin nous présentons les différentes transformations élémentaires que nous avons choisies pour simplifier notre graphe, comment notre postulat d'unicité des nœuds influence l'implémentation de ces transformations et quels problèmes nous avons rencontrés. Nous terminons par une présentation des résultats, nous suggérons quelques développements futurs et indiquons comment, dans le cas spécifique de *LeJoueur*, une meilleure solution a été trouvée en amont.

4.1 Transformation des circuits en graphe

Les règles des quatre groupes de circuits *TERMINAL*, *LEGAL*, *GOAL*, *NEXT* ne sont pas totalement indépendantes d'un groupe à l'autre. Pendant un *playout*, *TERMINAL* est toujours évalué en premier, puis en fonction du résultat, *GOAL* est évalué ou bien *LEGAL* et *NEXT* sont évalués l'un après l'autre, le choix d'un coup à jouer étant effectué entre les deux. Plutôt que de redéfinir toutes les règles dans chaque groupe, *GOAL* peut donc utiliser dans le corps des règles des faits calculés dans *TERMINAL*. De même *LEGAL* peut utiliser des faits calculés dans *TERMINAL* et *NEXT* des faits calculés dans *LEGAL* et *TERMINAL*.

La simplification des circuits nécessite donc de travailler sur l'ensemble des règles. Pour pouvoir simplifier les circuits en tenant compte des dépendances, nous avons transformé l'ensemble des circuits en un graphe.

Nous avons utilisé *LeJoueur* pour générer un *propnet* pour chacun des jeux testés puis exporter les circuits (voir figure 4.1) et la liste de fluents (voir figure 4.2) sous forme de fichiers textuels. Le codage des règles est celui utilisé pour la communication des informations au MPPA : un fait est représenté par son index dans la liste des fluents multiplié par 2, un fait négatif est signalé en ajoutant 1.

Notre programme construit le graphe à partir de ces fichiers textuels en traitant chaque règle dans l'ordre : à chaque fois qu'un fluent apparaît comme conséquent d'une règle, l'existence du nœud correspondant est vérifiée dans une liste de tous les nœuds ; il est alors réutilisé s'il existe déjà ou créé dans le cas contraire. Le regroupement des règles en différentes strates garantit qu'un fluent utilisé dans le corps d'une règle (antécédent) correspondra à un nœud déjà créé. Les strates restent ensuite présentes dans la structure même du graphe. Ce graphe constitue la solution initiale que l'algorithme de recuit simulé doit optimiser.

Le graphe contient des nœuds ET, OU, NON qui découlent naturellement des règles mais également des nœuds que nous avons nommés LATCH, qui correspondent aux fluents de type *input* et *base* et qui constituent les entrées du circuit, et des nœuds que nous avons nommés ALIAS, dont la valeur est une copie de celle d'un autre nœud (voir figure 4.3).

4.2 Garantir l'unicité des nœuds

L'examen des circuits nous a révélé que deux actions, la factorisation et la détection de sous-expressions communes, peuvent permettre de les simplifier (voir section 2.9). L'identification d'un facteur peut être effectuée en un temps linéaire. En revanche, la détection de sous-expressions communes nécessite d'examiner toutes les combinaisons possibles de termes.

Pour identifier automatiquement les sous-expressions communes, nous avons choisi de laisser l'exploration stochastique, effectuée par le recuit simulé, faire apparaître ces sous-expressions, et afin de s'assurer que ces sous-expressions communes soient automatiquement fusionnées, nous avons utilisé un mécanisme garantissant l'unicité des nœuds.

Notre programme maintient une liste de nœuds uniques représentée par une structure C++ de type `std::set` : deux nœuds sont considérés identiques quand ils sont du même type (ET, OU, NON) et possèdent les mêmes entrées. Les entrées de chaque nœud sont également placées dans des `std::set` ; elles sont ainsi uniques et ordonnées, et la comparaison des entrées de deux nœuds peut être réalisée simplement en itérant sur le contenu de ces `std::set`. A chaque transformation effectuée sur le circuit, les nouveaux nœuds créés sont comparés à ceux pré-existant, si un

```

-- TERMINAL -----
-- strate 0 --
214 :- 2.
214 :- 4.
214 :- 6.
(...)
258 :- 44, 50, 56.
-- strate 1 --
210 :- 222.
210 :- 224.
(...)
212 :- 238.
-- strate 2 --
116 :- 210.
116 :- 212.
116 :- 215.

-- LEGAL -----
-- strate 0 --
188 :- 12, 0.
168 :- 12, 20.
190 :- 14, 0.
(...)
164 :- 8, 20.
186 :- 10, 0.

-- GOAL -----
-- strate 0 --
200 :- 211, 213, 215, 213, 211.
(...)
208 :- 210.

-- NEXT -----
-- strate 0 --
78 :- 0.
58 :- 20.
(...)
```

FIGURE 4.1 – Extrait d'un fichier de règles utilisé en entrée de notre programme de simplification de circuits (jeu de *Tictactoe*).

```
0 - (true (control oplayer))
1 - (true (cell 1 1 b))
2 - (true (cell 1 2 b))
3 - (true (cell 1 3 b))
4 - (true (cell 2 1 b))
5 - (true (cell 2 2 b))
6 - (true (cell 2 3 b))
(...)
16 - (true (cell 2 3 x))
17 - (true (cell 3 1 x))
18 - (true (cell 3 2 x))
(...)
26 - (true (cell 3 1 o))
27 - (true (cell 3 2 o))
28 - (true (cell 3 3 o))
29 - (next (control oplayer))
(...)
49 - (next (cell 1 1 o))
50 - (next (cell 1 2 o))
51 - (next (cell 1 3 o))
(...)
58 - terminal
59 - (does xplayer (mark 1 1))
60 - (does xplayer (mark 1 2))
(...)
83 - (legal xplayer (mark 2 2))
84 - (legal xplayer (mark 2 3))
85 - (legal xplayer (mark 3 1))
(...)
99 - (goal xplayer 100)
100 - (goal xplayer 50)
101 - (goal xplayer 0)
(...)
105 - (line x)
106 - (line o)
107 - open
108 - (row 1 b)
(...)
129 - (column 3 o)
```

FIGURE 4.2 – Extrait d’un fichier de fluents utilisé en entrée de notre programme de simplification de circuits (jeu de *Tictactoe*).

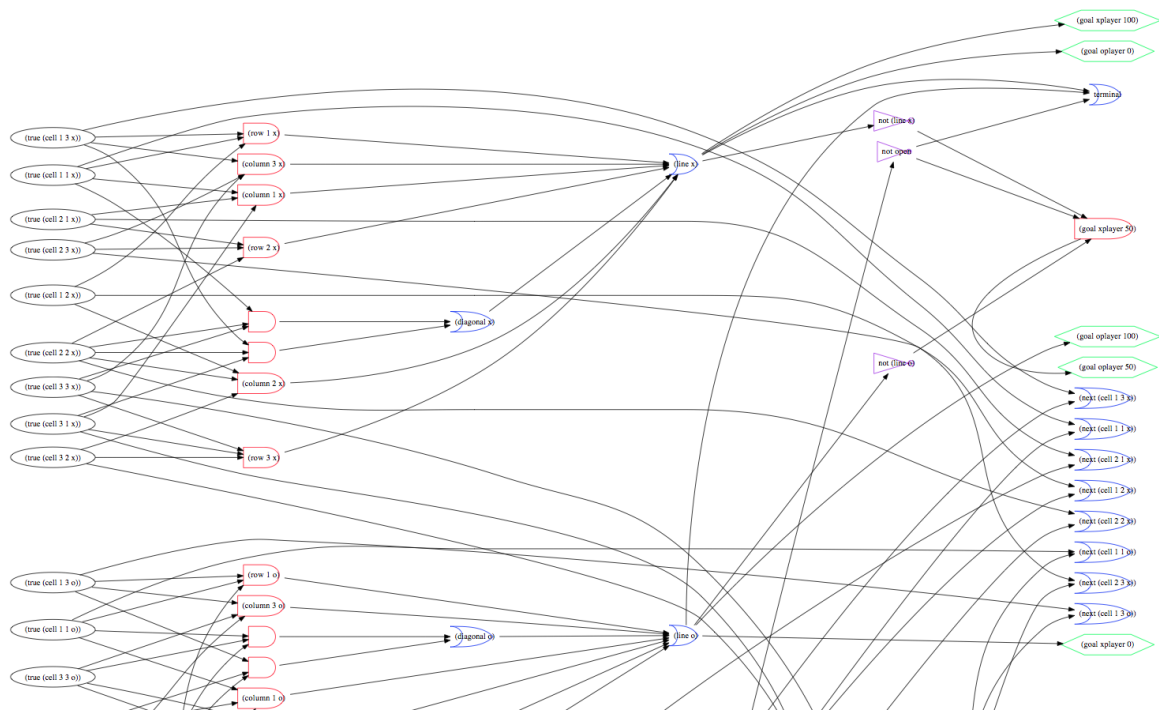


FIGURE 4.3 – Extrait du graphe généré pour le jeu de *Tictactoe*. Les ovales noirs signalent les LATCH, Les portes ET sont signalée en rouge, les OU en bleu, les NON en violet, les ALIAS en vert.

nœud existe déjà il est alors réutilisé et non dupliqué.

Certains fluents en sortie du circuit peuvent avoir des valeurs dépendant des mêmes faits. Par exemple dans un jeu à deux joueurs et à somme nulle, les fluents (goal playerA 100) et (goal playerB 0) sont représentables par deux nœuds identiques. C'est pour s'assurer que des sorties du circuit ne risquent pas d'être éliminées par le mécanisme garantissant l'unicité des nœuds que nous avons introduit le type ALIAS mentionné précédemment. Ces ALIAS peuvent également apparaître au milieu d'un circuit quand un conséquent ne dépend que d'un unique antécédent, par exemple dans *ConnectFour* :

```
(columnopen 3) :- (cellopen 3 6).
```

4.3 Évaluation d'une solution

Notre but étant de minimiser la taille des circuits que nous souhaitons transmettre au MPPA, il est tout naturel d'utiliser le nombre de valeurs nécessaires pour coder les circuits comme évaluation d'une solution. Étant donné que notre programme maintient une liste de tous les nœuds du graphe, il est possible d'évaluer la taille du circuit correspondant en donnant un poids à chaque nœud et en faisant la somme de ces poids. La base de notre évaluation est la suivante :

- **Une porte de type ALIAS** ne fait que copier la valeur d'un terme. Si par exemple

il existe un terme G défini au préalable, l'ALIAS H de G sera défini par la règle $H :- G$, son poids sera donc 3 : une valeur pour indiquer la taille de la règle et 2 valeurs pour l'antécédent et le conséquent.

- Une **porte** OU avec N entrées donne lieu à la création de N règles. Son poids est donc $N * 2$ pour coder la taille et le conséquent de chaque règle, plus la somme des entrées de ses entrées de type ET et plus 1 pour chacune des autres entrées (de type NON ou LATCH).
- Une **porte** ET peut être comptabilisée dans les portes OU qui l'utilisent, dans ce cas, elle n'apporte aucun poids supplémentaire au circuit. Cependant si la conjonction est utilisée dans une autre conjonction ou si la porte ET est une sortie, alors son poids est égal au nombre de ses entrées plus 2 pour coder sa taille et le conséquent de la règle.



Par exemple, dans le cas A ci-dessus, la porte y est utilisée par une porte OU. Cette porte est traduite par les deux règles $x :- c$ et $x :- a.b$ de poids total 7. L'expression $a.b$ étant incluse dans les règles codant x , la porte y n'apporte aucun poids supplémentaire au circuit. En revanche dans le cas B , la porte x est une sortie traduite par la règle $x :- c.y$ de poids 4 et la porte y est traduite par la règle $y :- a.b$ de poids 4 également : chacune de ces deux portes ET apporte un poids égal au nombre de ses entrées plus 2.

- Les **portes de type LATCH et de type NON** sont codées dans les conjonctions et disjonctions qui les utilisent, elle n'apporte donc aucun poids supplémentaire au circuit, sauf dans le cas d'une double négation, la seconde porte NON aura une valeur de 3 (taille de la règle, antécédent et conséquent).

4.3.1 Problème d'ambiguïté

Lors de nos premiers tests, un problème est apparu lors de la reconversion du graphe en liste de règles : il est possible de convertir une porte ET de différentes manières dont l'une est plus économique que l'autre selon les circonstances. La conjonction peut être isolée et donner lieu à la création d'un nouveau terme conséquent, ou bien elle peut être l'antécédent d'une règle faisant partie d'une disjonction. Par exemple, il est possible d'écrire :

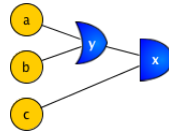
$x :- a.b$
 $x :- c$

ou bien :

$y :- a.b$

x :- y
x :- c

Ces deux ensembles de règles correspondent à un seul et même graphe :



Selon la taille et le nombre d'utilisations de la conjonction, un codage peut être plus économique que l'autre :

x :- a.b (4)
x :- c.d (4)
y :- e.f (4)
y :- c.d (4)

Cette formulation permet de coder le circuit avec 16 valeurs.

Z :- c.d (4)
x :- a.b (4)
x :- Z (3)
y :- e.f (4)
y :- Z (3)

Celle-ci, équivalente, nécessite 18 valeurs : ici l'isolement de la conjonction n'est pas optimal. Prenons un autre exemple :

x :- a.b (4)
x :- h.i.j.k.l.m.n.o.p.q (12)
y :- e.f (4)
y :- h.i.j.k.l.m.n.o.p.q (12)

La répétition de la conjonction alourdit le circuit, son poids total est de 32.

Z :- h.i.j.k.l.m.n.o.p.q (12)
x :- a.b (4)
x :- Z (3)
y :- e.f (4)
y :- Z (3)

Ce circuit a un poids de 26. Dans le cas où la conjonction est de taille importante et où elle est réutilisée plusieurs fois, il est visible que l'isoler devient plus économique.

La solution à adopter dépend donc de la taille de la conjonction et du nombre d'utilisations dont elle fait l'objet. Nous avons donc choisi, lors de l'évaluation du circuit, d'évaluer le poids des deux représentations pour choisir la plus économique des deux.

Cependant, dans le cas où une conjonction est utilisée comme entrée d'une porte NON, d'une autre porte ET ou constitue une sortie du circuit, elle est forcément conservée comme un terme isolé. Il est donc inutile de tester la différence de poids des deux représentations possible de cette conjonction : le terme existant déjà, il est plus économique de le réutiliser dans une disjonction.

4.3.2 Propagation des effets

Afin de ne pas devoir réévaluer entièrement une solution à chaque itération, les algorithmes de recuit simulé évaluent juste la différence d'énergie que la transformation élémentaire a apporté par rapport à la solution précédente. C'est cette façon de faire que nous avons voulu utiliser dans notre programme mais nous avons constaté que la contrainte que nous avons posée comme point de départ, i.e. vérifier l'unicité de chaque nœud, rend cette méthode d'évaluation terriblement complexe à mettre en œuvre.

Nous avons constaté qu'une transformation élémentaire peut avoir des effets bien au delà de la section de circuit où elle est appliquée (voir figure 4.4) et peut entraîner une simplification en chaîne de tout le circuit. Ces effets peuvent aussi simplement se traduire par un changement de la solution optimale à adopter pour la transformation d'un nœud ET (voir section 4.3.1).

Ce cas de figure intervenant quand deux nœuds distincts se révèlent en fait équivalents d'un point de vue logique, par exemple $((b + c).d)$ et $(b.d + c.d)$, nous avons envisagé¹, comme identifiant de chaque nœud, d'utiliser une clef de hashage qui serait une fonction logique de ses entrées. Les LATCH possédant un nom (le terme GDL correspondant), leur clef de hashage aurait été fonction de ce nom unique. En comparant les nœuds grâce à cette clef de hashage, il aurait été possible de détecter directement des nœuds probablement équivalents et pas seulement identiques (de même type et d'entrées identiques). Cependant, les circuits étant de grande taille pour certains jeux, l'unicité des clefs ne serait pas garantie, surtout en cas d'enchaînement de nombreux nœuds de type OU : la fonction logique OU appliquée à de nombreuses entrées amènerait à une clef de hashage dont tous les bits sont à 1. Nous avons donc écarté cette option.

Plutôt que d'effectuer une évaluation complexe à mesure de la propagation des

1. Jean-Noël Vittaut nous a indiqué avoir utilisé cette approche dans une version préliminaire de *LeJoueur* mais l'a abandonnée à notre connaissance.

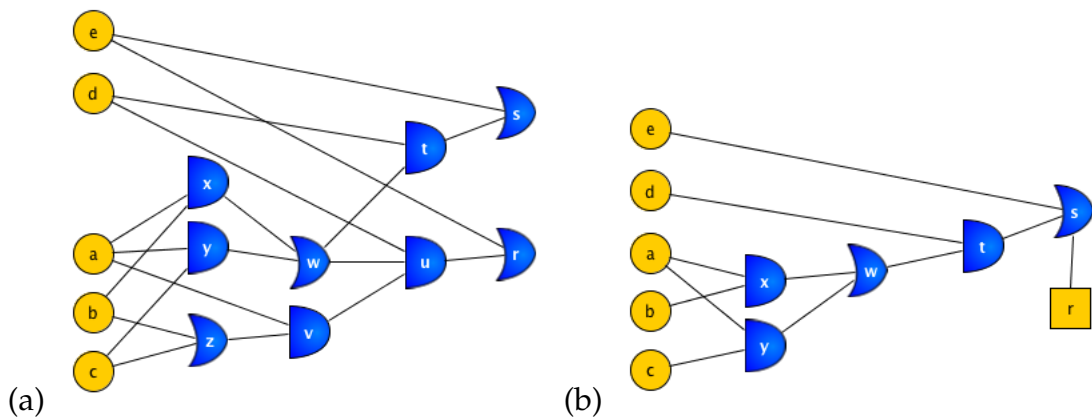


FIGURE 4.4 – Une transformation élémentaire de type EXPAND est appliquée sur le nœud v du circuit (a). Au lieu de se limiter au remplacement des nœuds v et z les effets de cette transformation se répercutent sur les nœuds suivants. Après développement, la nouvelle porte OU qui remplace v devient identique à w , le nœud u perd donc une entrée et devient identique à t , les nœuds s et r deviennent donc également identiques : si r est une sortie, r devient un ALIAS de s . Cet exemple trivial montre qu'il peut y avoir une propagation des effets d'une transformations à tout un circuit.

effets d'une transformation, nous avons finalement choisi de réévaluer le circuit complet en itérant sur tous les nœuds stockés dans notre liste de nœuds uniques. Une conséquence de ce choix est qu'il devient nécessaire d'appliquer les transformations pour pouvoir estimer leur effet. Au cas où l'algorithme de recuit amène au refus de la transformation, une transformation inverse doit être appliquée pour restituer l'état précédent du circuit.

Nos différents tests nous ont amenés à créer une douzaine de transformations élémentaires distinctes dont certaines se sont finalement révélées superflues. Cependant, certaines transformations utiles que nous avons finalement retenues entraînent des effets qui sont parfois difficiles à inverser. Les fonctions permettant de réaliser les transformations inverses se révèlent donc complexes et lourdes car elles doivent prévoir une multitude de cas particuliers.

Par exemple si l'on veut simplifier l'expression $((A.B).A)$ en fusionnant les deux portes ET, on obtient $(A.B)$. Notre postulat d'unicité des nœuds implique que le nœud $(A.B)$ de la première expression est réutilisé. Cependant notre fonction de fusion détruit les deux nœuds de départ pour les remplacer par un nouveau nœud qui dans ce cas particulier, si l'on y prend pas garde, a été également détruit. Ce genre de situation peut donc provoquer un bogue dans le programme. Quand un cas particulier comme celui-ci est pris en compte, l'inversion de l'opération est également problématique car pour rescinder la porte $(A.B)$ en deux, il faut garder une trace des entrées et sorties des portes d'origine. Quand nous utilisons deux transformations élémentaires inverses l'une de l'autre, chacune doit donc retourner une structure contenant toutes les informations nécessaires à l'inversion de la transformation et prendre en argument la structure éventuellement retournée par l'autre fonction dans le but d'effectuer une restauration².

2. On peut faire ici un parallèle avec les fonctions utilisées pour jouer et dé-jouer un coup dans les programmes de jeu. En poussant plus loin ce parallèle, on pourrait imaginer notre programme

En plus de ces problèmes d'inversion des transformations, quand il y a propagation des effets, nous sommes partis du principe que cette propagation entraîne forcément une grosse simplification du circuit et que la transformation n'a pas besoin d'être annulée, mais nous n'avons aucune garantie que ce soit toujours le cas. Après avoir utilisé des fonctions qui tentent d'inverser les transformations localement, nous avons rencontré deux grandes difficultés : de nombreux bogues dûs aux nombreux cas particuliers auxquels il n'est pas évident de penser à priori et l'impossibilité d'inverser certaines transformations. Finalement nous avons choisi de créer une copie du circuit complet avant chaque transformation pour pouvoir restaurer son état de manière plus sûre. Cette solution a l'inconvénient d'utiliser deux fois plus de mémoire pour stocker le circuit et sa copie mais permet une importante simplification des fonctions réalisant les transformations élémentaires.

Cette solution qui consiste à appliquer les transformations, calculer le poids de tout le circuit et restaurer une copie le cas échéant est loin d'être optimale en terme de temps de calcul et d'usage de la mémoire, mais dans le cadre de notre test préliminaire pour estimer l'efficacité du recuit simulé pour la simplification de circuits, nous avons privilégié la simplicité de cette solution, une autre approche permettant l'optimisation du programme pouvant être recherchée par la suite.

4.3.3 Évaluation finale des circuits

Notre évaluation pendant le fonctionnement de l'algorithme de recuit ne tient pas compte des valeurs nécessaires pour indiquer le nombre de strates et la taille de chaque strate dans la version sérialisée transmise au MPPA. Nous avons considéré que la différence du nombre de strates était négligeable entre le circuit de départ et sa version simplifiée. Cependant pour nous en assurer, nous avons effectué une évaluation finale de la taille des solutions générées.

Cette évaluation supplémentaire nous a permis d'identifier une erreur dans notre implémentation. Dans les premières versions de notre programme, nous n'avions pas pris conscience du dernier point évoqué dans la section 4.3.1 : pendant l'évaluation, des conjonctions de termes étaient comptabilisées dans l'antécédent des règles de différentes disjonctions alors qu'une règle définissant une sous-expression correspondant à cette conjonction avait déjà été comptabilisée.

Par exemple :

$$\begin{aligned}e &= a.b \text{ (4)} \\g &= \neg e + d \text{ (4)} \\f &= c \text{ (3)} \\f &= a.b \text{ (4 au lieu de 3)} \\h &= g.f \text{ (4)}\end{aligned}$$

de simplification sous forme d'un jeu, mais que serait alors la description GDL d'un jeu consistant à simplifier la description d'un jeu ?

La répétition de $a.b$ dans l'antécédent de f aurait dû être interdite puisque le terme e existe déjà : e est rendu nécessaire par son utilisation comme entrée de la négation $\neg e$.

De plus notre calcul du poids des deux versions possibles de la représentation des ET présentait un bogue. Ces erreurs n'étaient en revanche pas présentes dans la routine finale re-transformant notre graphe optimisé en circuit car la présence des règles définissant les conjonctions était testée avant de choisir si elles devaient être incluses dans l'antécédent d'une règle.

Cette mauvaise évaluation entravant la bonne convergence de l'algorithme de recuit, il était difficile d'estimer le véritable effet des modifications successives que nous effectuions sur notre programme. Les ajustements du programme de recuit et la modification des transformations élémentaires appliquées au circuit ne permettaient pas d'obtenir une solution meilleure que celle de départ et les effets observés n'étaient pas cohérents avec nos hypothèses.

L'incohérence entre la qualité des circuits obtenus, mesurée par notre évaluation finale, et l'évolution de l'évaluation des circuits durant le recuit, observée sur nos graphiques, nous a permis d'identifier cette erreur et de réévaluer les effets de notre programme de recuit et des différentes transformations élémentaires appliquées au graphe.

4.4 Choix du programme de recuit

Afin de fixer la température de départ nous avons procédé empiriquement en réalisant une expérimentation préalable. En fixant une température élevée de manière à accepter toutes les transformations, nous avons itéré 200 fois et calculé la variation moyenne de l'énergie $\langle \Delta E \rangle$. Nous avons ensuite évalué la température initiale T_0 selon la formule de [Siarry et al. \[1997\]](#) : $T_0 = - \langle \Delta E \rangle / \log(\tau_0)$ avec un taux d'acceptation initial entre 50% et 60%.

[Johnson et al. \[1989\]](#) indiquent qu'un taux d'acceptation initial de 40%, déterminé expérimentalement, permet de réduire la durée de la recherche sans sacrifier la qualité des solutions trouvées. Cependant d'autres chercheurs utilisent des taux allant jusqu'à 95% voire 99%. Nous avons fait de choix d'augmenter légèrement le taux conseillé par [Johnson et al. \[1989\]](#) pour nous assurer d'une bonne convergence de l'algorithme.

Les expérimentations de [Johnson et al. \[1989\]](#) sur les différents schémas de décroissance de la température montrent que la décroissance géométrique donne les meilleurs résultats en pratique et présente l'avantage de la simplicité ; nous avons donc choisi ce schéma de décroissance. La rapidité de convergence n'est pas notre priorité. La qualité de la solution étant recherchée avant tout, nous avons donc choisi un taux de décroissance de 0.99 qui assure une décroissance lente. Le temps d'exécution de notre programme est donc important mais nous pouvons ainsi estimer la

qualité de la solution obtenue dans le meilleur des cas.

La manière de déterminer la longueur des chaînes de Markov n'est pas clairement définie dans la littérature. Nous savons qu'elle est fonction de la dimension du problème mais cette *dimension* est complexe à estimer. Pour nous assurer d'une bonne exploration de l'espace des solutions, nous avons fixé les paliers de manière statique entre 10000 et 1000000 itérations en fonction du temps disponible pour les expériences.

Lorsque plus aucune transformation n'est appliquée à la solution durant un nombre d'itérations équivalent à un dixième du palier (entre 1000 et 100000 itérations), notre programme baisse la température et entame le palier suivant. Lorsque 3 paliers de température sont passés sans changement la recherche est stoppée. Certaines transformations élémentaires peuvent ne pas modifier l'évaluation du circuit ; dans ce cas, comme elles n'augmentent pas le coût, elles sont acceptées mais ne produisent pas de véritable changement. Pour éviter que l'algorithme de recuit ne se poursuive indéfiniment, nous avons donc fixé une température limite de 0.001 au delà de laquelle aucune transformation strictement positive n'est à espérer.

4.5 Choix des transformations élémentaires

Afin d'appliquer l'algorithme de recuit simulé, il est nécessaire de définir des transformations élémentaires. Pour déterminer ces transformations, nous avons testé empiriquement diverses simplifications et leurs équivalents inverses de manière à ce que l'exploration stochastique puisse faire marche arrière quand une simplification ne mène pas à une zone satisfaisante de l'espace des solutions³.

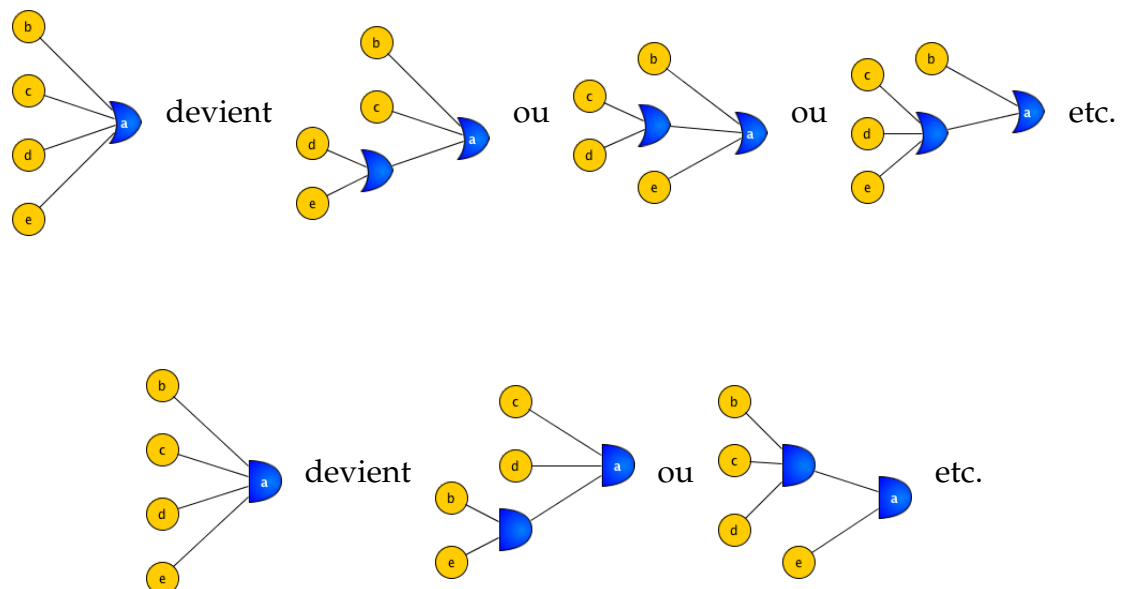
Nous commençons par présenter les quatre transformations principales que nous avons utilisées puis expliquons la solution que nous avons apportée au problème posé par les différences de probabilité d'application de ses transformations. Nous présentons ensuite quelques transformations optionnelles qui après expérimentation se sont révélées superflues. Nous expliquons le problème de convergence de l'algorithme que nous avons rencontré et la solution apportée en guidant la découverte des sous-expressions communes. Enfin nous présentons trois transformations supplémentaires ajoutées pour traiter spécifiquement les négations.

3. Comme indiqué dans la section 4.3.2, dans la première version de notre programme nous utilisons les fonctions effectuant les transformations inverses également pour annuler les transformations augmentant le coût de la solution qui étaient refusées par le recuit. Dans la version finale de notre programme, ces fonctions sont grandement simplifiées puisqu'elles n'ont plus besoin de prendre en arguments les informations nécessaires à une éventuelle annulation d'une transformation inverse, de gérer les cas particuliers et de produire en sortie les informations éventuellement nécessaires à une annulation de la transformation courante.

4.5.1 Transformations élémentaires principales

Nous avons isolé quatre transformations élémentaires principales que nous avons nommées *split*, *merge*, *factorize* et *expand* (voir figure 4.5) permettant d'opérer les simplifications souhaitées sur nos circuits.

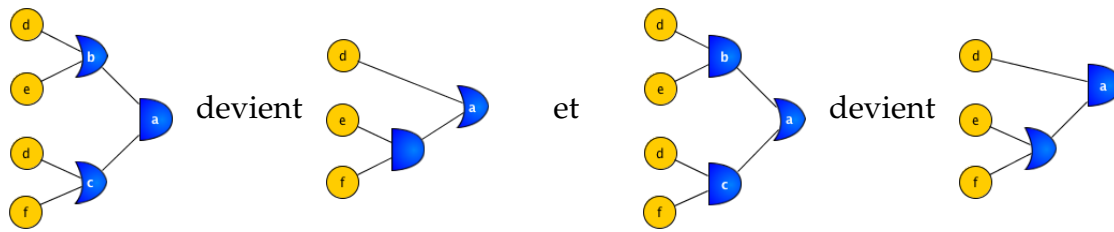
L'opération *split* consiste à dédoubler une porte logique ET ou OU. Une porte logique possédant plus de deux entrées peut être dédoublée en sélectionnant au maximum N-1 entrées et au minimum 2 parmi les N qu'elle possède, en attribuant ces entrées à une nouvelle porte et en ajoutant cette nouvelle porte comme entrée de la première :



L'opération *merge* est l'inverse de *split*. Lorsqu'une porte logique possède une entrée de même type qu'elle, les deux portes peuvent être fusionnées en réunissant toutes leurs entrées sur une seule et même porte. Une porte de type ALIAS, peut également être fusionnée avec n'importe quelle porte qui n'est pas une sortie du circuit, puisque cet ALIAS présente en sortie une simple copie de son entrée.

L'opération *factorize* consiste à factoriser un terme. Cette opération n'est appliquée que sur des portes à deux entrées⁴. L'opération *split* permet de dédoubler les portes jusqu'à ce qu'une factorisation possible apparaisse. Lorsqu'une porte logique ET (resp. OU) possède deux entrées de type OU (resp. ET) et que ces portes ont un terme commun en entrée, ce terme peut être factorisé :

4. Cette restriction était initialement imposée pour faciliter la restauration en cas de transformation augmentant le coût de la solution. Elle a été conservée car elle simplifie également la détection d'une factorisation possible.



L'opération *expand* est l'opération inverse de *factorize*. Lorsqu'une porte ET (resp. OU) possède deux entrées dont une entrée de type OU (resp. ET), cette porte peut être développée de telle manière que $(a.b + c)$ devient $((a + c).(b + c))$ et $((a + b).c)$ devient $(a.c + b.c)$.

La figure 4.5 montre comment, avec différentes transformations élémentaires successives utilisant uniquement ces quatre opérations que nous avons définies, il est possible de simplifier un circuit en factorisant certains termes et en isolant une sous-expression commune.

4.5.2 Non uniformité des transformations élémentaires

A chaque itération une transformation élémentaire est tirée au hasard et le programme itère sur la liste de tous les nœuds pour en trouver un sur lequel la transformation est applicable. Si la transformation ne peut être appliquée le programme passe à l'itération suivante. L'algorithme de recuit permet, en appliquant toujours les transformations aux effets positifs et en refusant les transformations aux effets négatifs avec une certaine probabilité, d'optimiser progressivement la solution de départ. Nos premiers tests montraient au contraire une complexification importante du circuit que nous souhaitions simplifier (voir figure 4.6). L'augmentation de la température initiale ne permettait pas d'améliorer la découverte de solutions optimales, au contraire, la complexification du circuit était encore plus importante.

Pour contrer cet effet, nous envisagions comme suggéré par Boese et Kahng [1994] de repartir de la meilleure solution trouvée à chaque changement de palier. Ainsi si des transformations augmentant le coût ne remplissaient pas leur rôle, i.e. sortir d'un minimum local pour trouver une zone plus prometteuse de l'espace de recherche mais complexifiaient le circuit de manière non maîtrisée, il aurait été possible de revenir en arrière et tester une autre piste de recherche. Une autre solution, pour éviter de trop s'éloigner de la meilleure solution trouvée, nous semblait être de calculer la probabilité d'appliquer une transformation augmentant le coût à partir de la différence d'énergie entre la meilleure solution et la solution suivante plutôt qu'entre la solution courante et la solution suivante.

Cependant cette complexification non maîtrisée du circuit laissait supposer que le problème provenait d'un dysfonctionnement dans la génération des solutions voisines. Nous avons pu observer que les transformations augmentant peu la taille du circuit avaient une forte probabilité d'être appliquées pendant les premiers paliers de température, et cumulées, ces transformations complexifiaient le circuit de

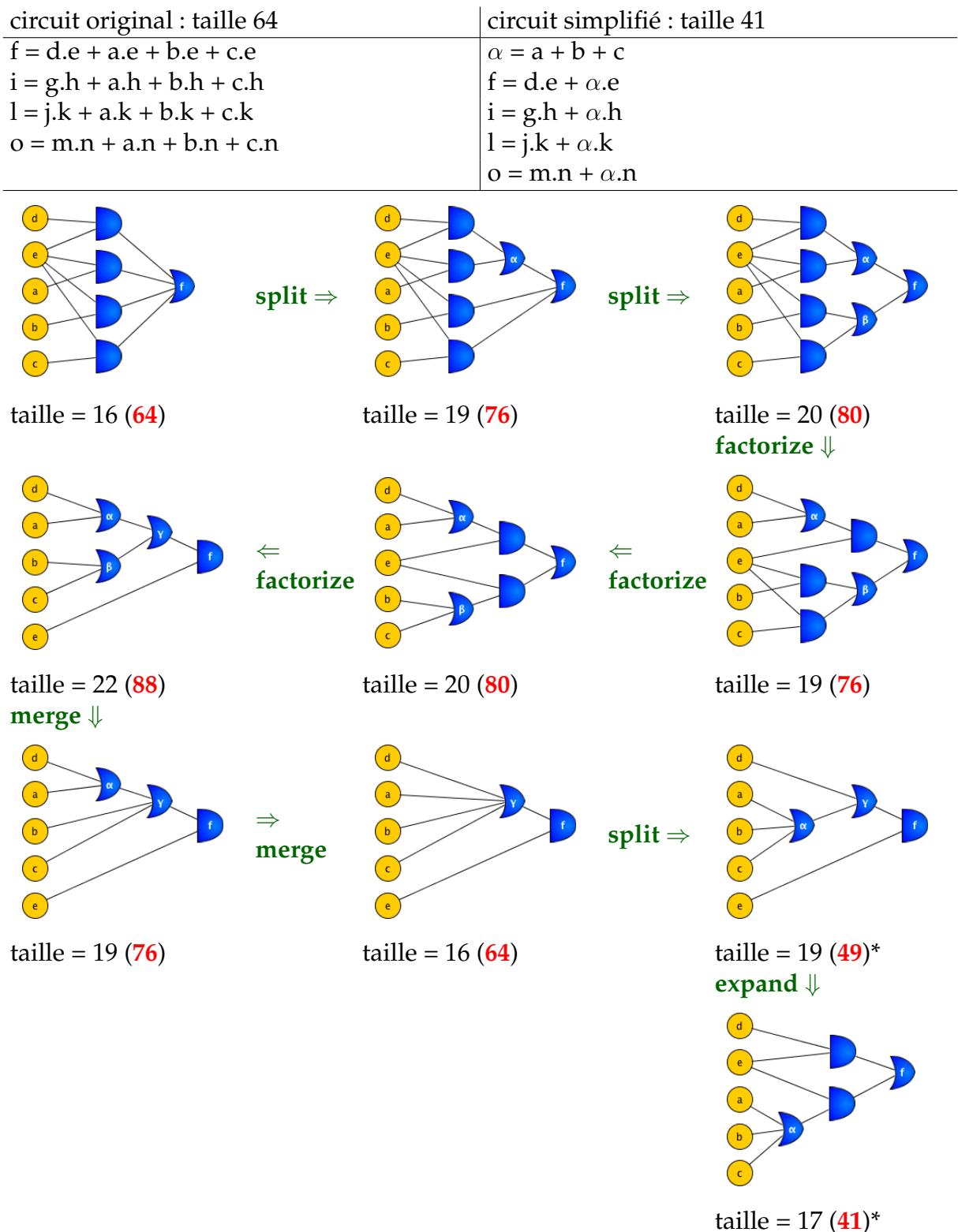


FIGURE 4.5 – Simplification d'un circuit avec les opérations *split*, *merge*, *factorize* et *expand*. Seule la simplification de l'expression f est détaillée. Sous chaque graphe figure la taille de sa représentation en circuit. La taille totale du circuit en appliquant la même transformation à chaque expression est signalée en rouge. Les deux dernières étapes signalées par * montrent le gain apporté par la détection d'une sous-expression α commune.

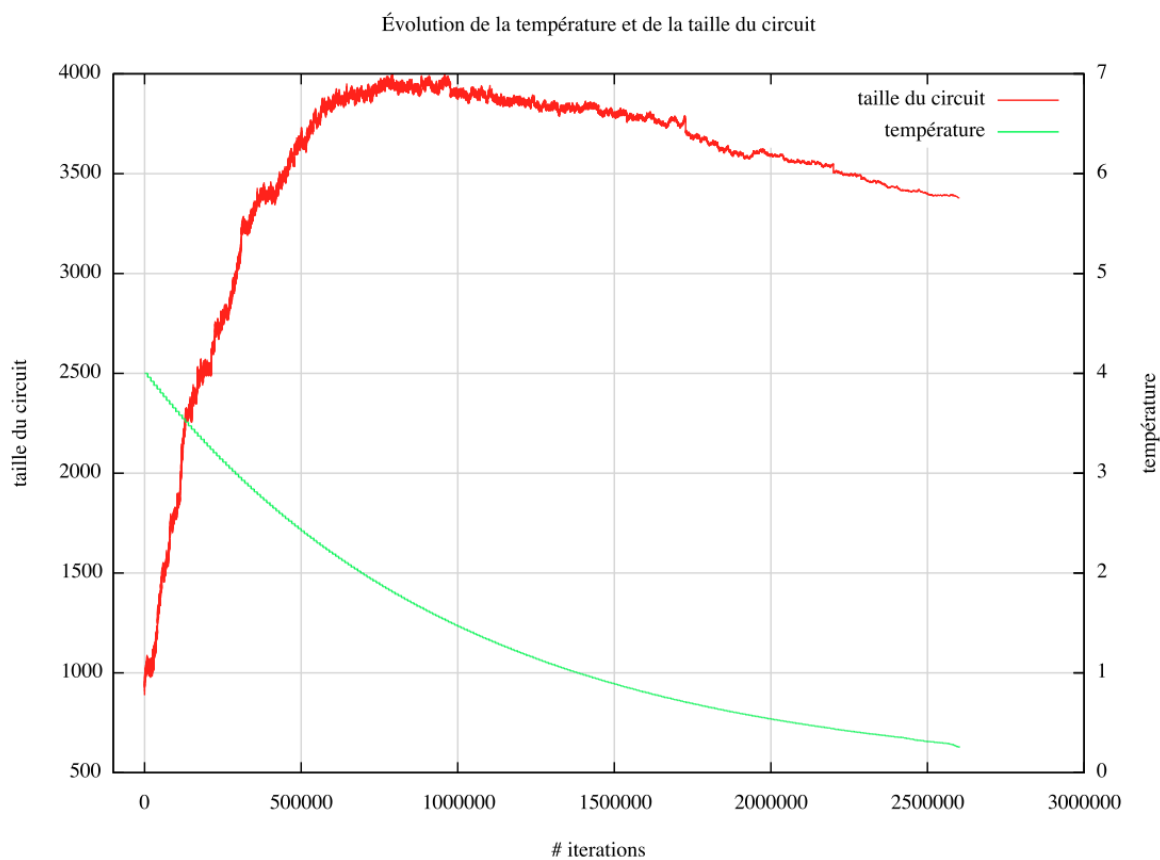


FIGURE 4.6 – Une session de l’algorithme de recuit testée sur le jeu *Blocker* avec les transformations élémentaires *split*, *merge*, *factorize* et *expand* appliquées avec une probabilité égale. L’algorithme ne converge pas et le circuit se complexifie de manière incontrôlée.

manière incontrôlée. Les factorisations et les sous-expressions détectées ne permettaient pas de re-simplifier le circuit par la suite.

Dans l'algorithme de recuit on accepte une transformation négative avec une certaine probabilité pour se rapprocher progressivement de l'état optimal en acceptant plus de transformations positives que négatives. Cependant, cela implique qu'à la base la probabilité de tenter une transformation négative soit de 0,5 (équivalente à la probabilité que la transformation soit positive). Ou bien, il faut que la probabilité de refuser une transformation négative soit suffisante pour permettre d'avoir moins de transformations négatives que de positives au final. Or il apparaissait dans notre première implémentation que ce ne soit pas le cas, il y avait très peu de transformations positives par rapport à la quantité de transformations négatives.

Un examen plus approfondi de l'évolution des circuits nous a permis d'identifier l'origine de ce problème : il provient de la différence entre les possibilités d'application de nos transformations élémentaires. Le dédoublement d'un nœud ou le fusionnement de deux nœuds ont des probabilités équivalentes d'être appliquées et, quand tous les nœuds sont dédoublés ou fusionnés, ces transformations ne sont plus applicables. Cependant, alors que la factorisation peut être effectuée dans les cas peu nombreux où un facteur existe, l'opération *expand* est toujours applicable et peut complexifier le circuit à l'infini.

Henderson *et al.* [2010] indiquent que les solutions voisines générées par transformations élémentaires peuvent l'être de manière non uniforme pour guider la recherche : « Fox (1993) suggests that instead of blindly generating neighbors uniformly, adopt an intelligent generation mechanism that modifies the neighborhood and its probability distribution to accommodate search intensification or diversification, in the same spirit of the tabu search meta-heuristic. Fox (1993) also notes that simulated annealing convergence theory does not preclude this idea. »

Pour limiter l'application de *expand*, une solution naïve consiste à biaiser le hasard en diminuant la probabilité de choisir la transformation de type *expand* ; cependant quand le nombre d'itérations augmente, le nombre d'applications de cette transformation augmente aussi et l'exploration non maîtrisée de l'espace des données réapparaît. Pour résoudre le problème nous avons choisi de procéder de la manière suivante :

- L'opération *expand* n'est choisie que dans moins de 2% des cas, les 3 autres transformations étant choisies avec une probabilité équivalente.
- Si la différence entre le nombre d'applications de *factorize* et de *expand* est supérieure à un centième de la taille initiale du circuit, alors l'opération *expand* est remplacée par *factorize*

De cette manière, l'opération n'est que très rarement appliquée et permet d'obtenir l'effet souhaité : le développement intervient pour supprimer une factorisation non optimale ou pour faire apparaître une sous-expression commune.

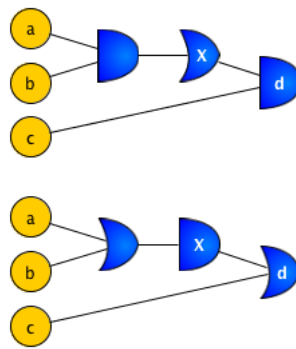
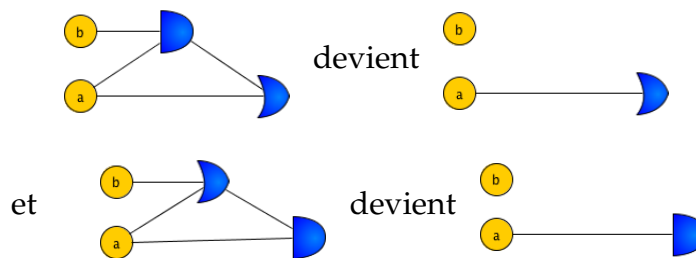


FIGURE 4.7 – Exemple de configurations pouvant apparaître suite à l’application des transformations optionnelles à un circuit. Les nœuds X ne peuvent pas être éliminés par les transformations de type *split*, *merge*, *factorize* ou *expand*

4.5.3 Transformations élémentaires optionnelles

L’examen des circuits générés durant les étapes du recuit nous ont révélé la présence d’une expression qui semblait ne pas pouvoir être simplifiée grâce aux quatre transformations élémentaires définies précédemment. L’expression du type $((a.b) + a)$ ne peut être transformée grâce aux opérations *split*, *merge* ou *factorize*. Soumise à la transformation *expand* elle devient $((a + b).a)$ qui, soumise à nouveau à *expand*, redonne l’expression de départ.

Pour pouvoir simplifier cette expression nous avons donc ajouté une transformation nommée *triangleTip* qui consiste à supprimer l’entrée inutile de la porte ET ou OU :



L’application de cette transformation est susceptible de provoquer deux effets : l’apparition de zones non connectées du graphe et l’apparition de portes à une seule entrée qui ne peuvent être éliminées par les quatre transformations principales (voir figure 4.7).

Pour éliminer les zones non connectées du graphe nous avons ajouté une transformation nommée *orphan* éliminant tout nœud non utilisé, i.e. ne possédant aucune sortie ou ne constituant pas lui-même une sortie.

Pour éliminer les portes à une seule entrée, nous avons tenté une première transformation consistant à supprimer le nœud et à transmettre son unique entrée aux

nœuds parents qui l'utilisent. Cependant cette transformation était à moitié redondante avec *merge*, elle aussi capable d'éliminer certains nœuds à une seule entrée. Nous avons donc utilisé une autre transformation nommée *switch* qui a en plus l'avantage de ne pas nécessiter de transformation inverse⁵. Elle consiste simplement à basculer le type d'une porte ET (resp OU) à OU (resp. ET), l'opération *merge* permet alors d'éliminer ce nœud en le faisant fusionner avec un autre.

L'application de ces transformations sur le jeu *Blocker* nous a permis de constater que la configuration permettant l'application de *triangleTip* apparaît rarement et se trouve être en fin de compte la conséquence de l'application des transformations principales que nous avons définies (voir figure 4.8). Les expressions du type $((a.b) + a)$ et $((a + b).a)$ peuvent donc être simplifiées en appliquant les transformations principales inverses et *triangleTip* n'est finalement pas nécessaire⁶. Nous avons donc éliminé *triangleTip*, *orphan* et *switch* des versions suivantes de notre programme.

4.5.4 Problème de convergence de l'algorithme

Malgré l'ajout d'un tirage non uniforme des transformations à appliquer, nos premières expérimentations montraient que l'algorithme de recuit présentait des difficultés importantes de convergence. La figure 4.9 montre l'évolution de la taille du circuit pendant le recuit pour le jeu *Blocker* ; nous pouvons voir que la taille du circuit stagne pendant de longues périodes durant lesquelles aucune simplification du circuit n'est trouvée.

De plus, lorsqu'une solution meilleure que celle de départ est finalement trouvée, elle est souvent perdue. Ali *et al.* [1996] indiquent que ce problème est courant avec le principe du recuit simulé : « *It is therefore quite possible that at some fixed temperature level the procedure will visit the optimal solution but due to the acceptance/rejection mechanism it will leave the best solution and arrive at a worse solution.* ». Un remède simple à ce problème est de mémoriser la meilleure solution trouvée durant la recherche pour la conserver dans le cas où la solution finale serait moins bonne.

Notre hypothèse concernant la difficulté de l'algorithme de recuit à trouver une simplification des circuits est que le voisinage de chaque solution comporte beaucoup plus d'états menant à des solutions non optimales que le contraire, ce qui rend la découverte de circuit plus simple très improbable. Prenons l'exemple donné à la figure 4.5 : une suite de transformations précises permet de simplifier le circuit de taille 64 en un circuit de taille 41. Cependant, à chaque étape la probabilité de trouver la transformation qui mènera vers la solution finale est faible comparée au nombre de transformations qui mèneront à une autre solution non optimale.

L'augmentation du nombre d'itérations pour augmenter les chances de l'algorithme stochastique de trouver une solution optimale ne nous semble pas une solu-

5. A ce stade de notre expérimentation nous n'avions pas encore opté pour une copie du circuit et une restauration totale.

6. Nous n'avons donc par conséquent jamais créé la transformation inverse de *triangleTip* permettant à l'exploration stochastique de faire marche arrière.

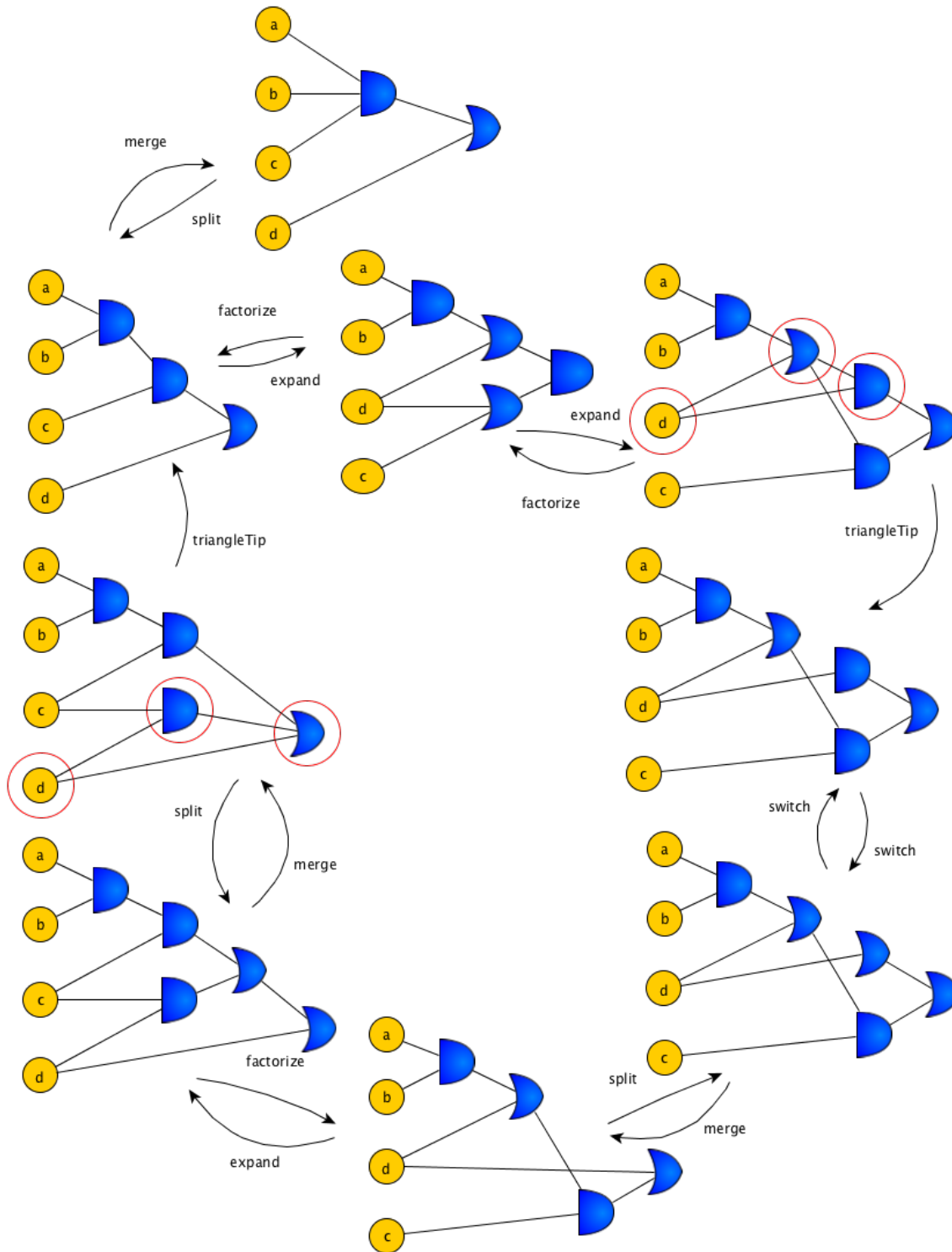


FIGURE 4.8 – Cette figure montre comment à partir d'un circuit simple du type $a.b.c + d$ (en haut), les transformations élémentaires *split-expand-expand* peuvent faire apparaître une expression du type $(d + X).d$ qui paraît à priori non simplifiable si elle est examinée de manière isolée. La transformation *triangleTip* permet de re-simplifier le circuit en 6 étapes mais le même résultat aurait pu être obtenu en 2 étapes en inversant les transformations initiales, *triangleTip* n'est donc pas une transformation utile.

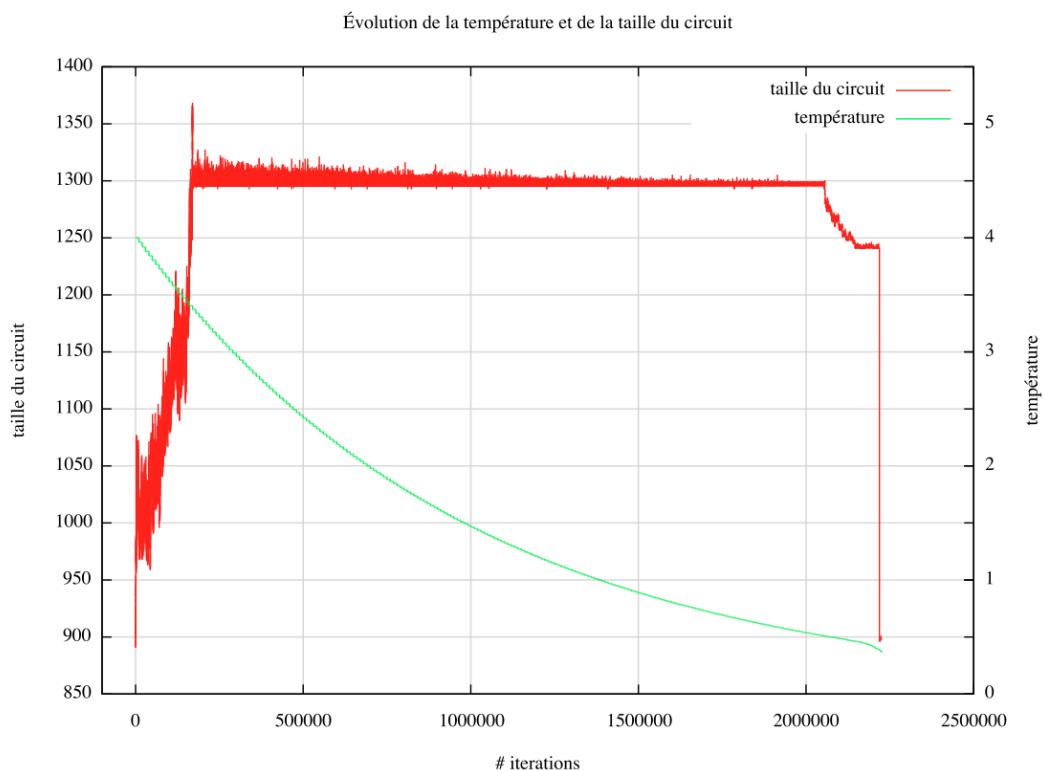
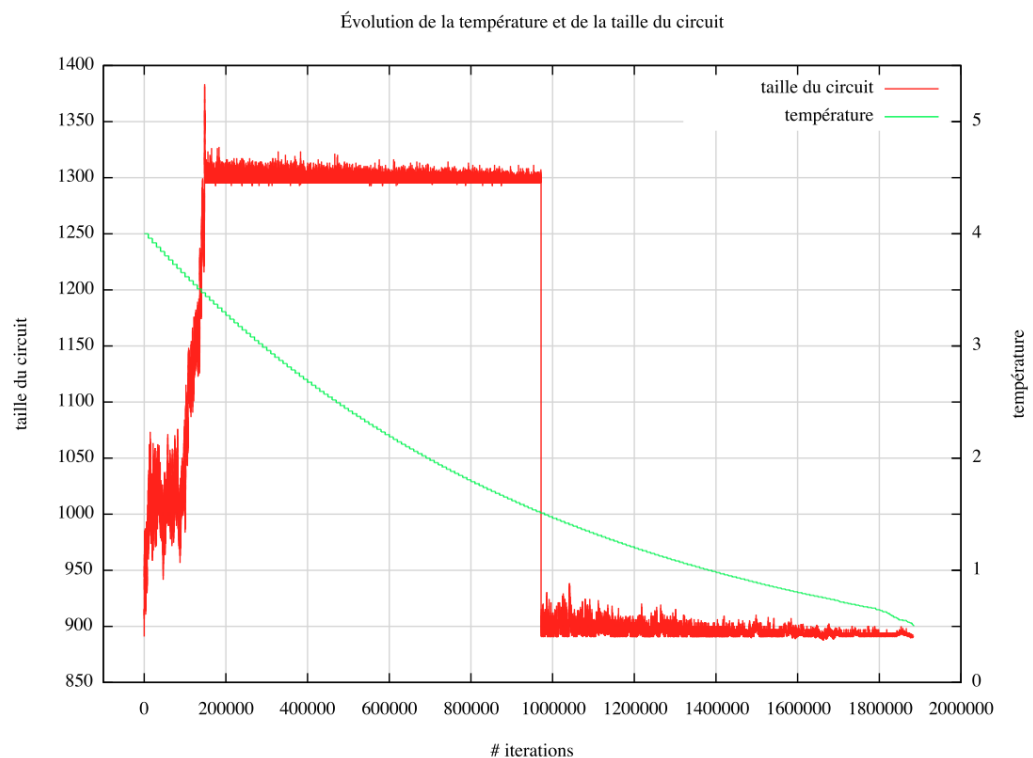


FIGURE 4.9 – Deux sessions de recuit simulé effectuées sur le jeu *Blocker* en appliquant les 4 transformations élémentaires *split*, *merge*, *factorize* et, avec une faible probabilité, *expand*. Le circuit se complexifie beaucoup durant les premiers paliers à température élevée puis, sur les deux graphes nous pouvons observer un long palier pendant lequel aucune transformation ne permet d’apporter une simplification majeure du circuit. Une simplification est finalement trouvée, plus ou moins vite selon les sessions de recuit, mais elle ne permet pas d’obtenir au final une solution meilleure que celle de départ.

tion satisfaisante parce qu'elle entraînerait une augmentation considérable du temps de convergence et qu'une grande partie de ce temps serait perdue à tester des transformations n'apportant aucune amélioration ni à court ni à long terme. L'ensemble des combinaisons de transformations à tester risquerait d'augmenter exponentiellement avec la taille des circuits à simplifier.

Černý [1985] évoque un problème similaire lors de l'application du recuit simulé au problème du voyageur de commerce : « *the computation ended in a local minimum. The same behavior was observed in other examples. We investigated the problem more closely, and our conjecture is that the algorithm using Step 4' is not able to untie efficiently the loops in the route [...]. To undo such a loop, it might be necessary to go through many configurations of higher energy. This is in principle possible, but highly improbable. The construction of Step 4 is chosen with the aim of undoing such loops quickly.* »

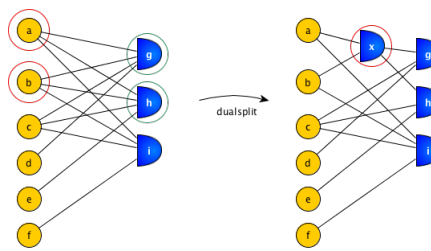
Dans le circuit du jeu de *Tictactoe*, une disjonction `lineX` permet de déterminer l'alignement de trois symboles `X`. Ce terme `lineX` est utilisé pour déterminer les scores et pour déterminer si la position courante du jeu est terminale. La transformation *merge* peut faire fusionner le nœud `lineX` avec `terminal` ; cette transformation augmente la taille du circuit mais elle est parfois acceptée à haute température. Pour re-simplifier le circuit obtenu, l'opération *split* devrait restaurer la sous-expression `lineX`, mais la probabilité que notre transformation *split* choisisse d'isoler les bonnes entrées est très faible.

Černý [1985] indique qu'il a créé une transformation élémentaire spécifique pour gérer les boucles dans le problème du voyageur de commerce. Nous avons donc cherché une solution en modifiant le comportement de la transformation élémentaire *split*.

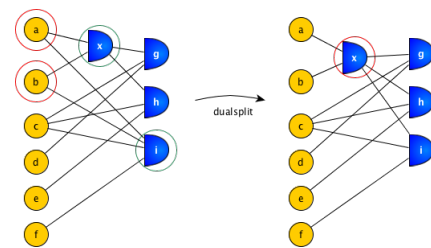
4.5.5 Guider la découverte des sous-expressions communes

Pour permettre à l'algorithme de recuit simulé de trouver directement des sous-expressions communes, nous avons remplacé la transformation élémentaire *split* par une transformation nommée *dualSplit*. Cette transformation consiste à identifier deux portes ET ou OU ayant deux entrées en commun et à isoler ces deux entrées pour constituer un nouveau terme.

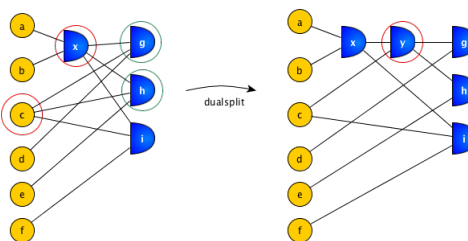
Par exemple sur le circuit suivant, la transformation permet d'isoler une sous-expression $a.b$: les entrées communes sont entourées en rouges et deux portes les utilisant sont entourées en vert. Le nouveau circuit généré possède une nouvelle porte x isolant la sous-expression $a.b$:



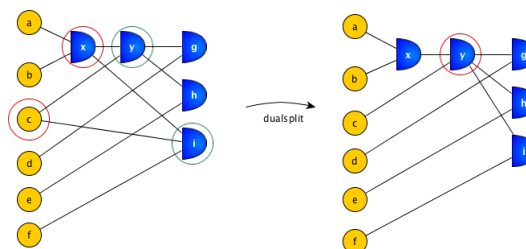
Appliquée à nouveau, l'usage de $a.b$ par i peut être détecté. La porte x correspondant déjà à l'expression $a.b$, aucune nouvelle porte n'est créée. Les entrées a et b de i sont remplacées par x :



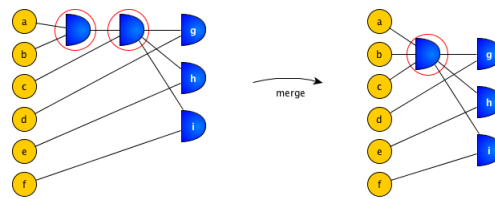
La transformation peut alors identifier que x et c sont utilisés par les deux portes g et h , ce qui entraîne la création d'un nouveau terme y :



Comme précédemment, si une autre porte utilise la même sous-expression, elle est réutilisée et aucune nouvelle porte n'est créée :



La transformation *merge* permet finalement de regrouper les différentes sous-expressions en une seule :



Les tests effectués sur *Blocker* montrent que cette transformation permet d'obtenir une bien meilleure convergence de l'algorithme de recuit simulé. En comparant la courbe de la figure 4.9 correspondant à l'utilisation de *split* et celle de la figure 4.10 correspondant à l'usage de *dualSplit*, nous constatons que les sous-expressions sont bien mieux détectées, ce qui se manifeste par une simplification du circuit dès les premières itérations du recuit. L'algorithme tente ensuite de raffiner la solution et converge vers un équilibre énergétique. La solution finale (taille 879) bien que non-optimale est meilleure que la solution de départ (taille 935). La meilleure solution trouvée a une taille de 875 ce qui représente 6,4% de gain.

Le même effet est visible sur la figure 4.11 pour le jeu *Eightpuzzle* (pousse-pousse 3x3 en français). Le circuit initial a une taille de 28269. Avec *split*, la meilleure solution trouvée a une taille de 23627, ce qui représente un gain de 16,4%. L'examen des circuits durant la recherche indique que la grande conjonction *inorder*⁷ qui permet de déterminer si les tuiles numérotées sont dans l'ordre est perdue suite à une transformation *merge* augmentant le coût et elle n'est que partiellement reconstituée par la suite. Avec *dualSplit*, la convergence de l'algorithme est bien meilleure. Lorsque *inorder* est perdue, la conjonction est rapidement reconstituée et la meilleure solution trouvée a une taille de 22336, ce qui représente presque 21% de gain.

Comme *merge* n'est pas exactement l'inverse de *dualSplit*, une transformation *removeDualSplit* était initialement utilisée pour annuler *dualSplit* quand la transformation augmentait le coût et devait être inversée. Cependant nous avons constaté différents cas particuliers où les effets de *dualSplit* ne pouvaient être correctement inversés. Cette transformation *dualSplit* rend l'utilisation d'une copie du circuit indispensable pour correctement restaurer son état après la transformation.

4.5.6 Traitement des négations

Jusqu'à l'obtention d'une bonne convergence de notre algorithme de recuit, nous avons mis de côté le traitement des négations. Certes les négations n'apportent aucun poids supplémentaire dans les circuits puisqu'elles sont simplement codées par des indices négatifs plutôt que positifs. Cependant, la transformation des expressions comportant des négations grâce aux lois de De Morgan peut permettre de

7. la règle originale est :

```
inorder :- (true (cell 1 1 1)), (true (cell 1 2 2)), (true (cell 1 3 3)),
           (true (cell 2 1 4)), (true (cell 2 2 5)), (true (cell 2 3 6)),
           (true (cell 3 1 7)), (true (cell 3 2 8)), (true (cell 3 3 b)).
```

elle est utilisée plus de 60 fois pour déterminer le fait terminal et les différents scores possibles.

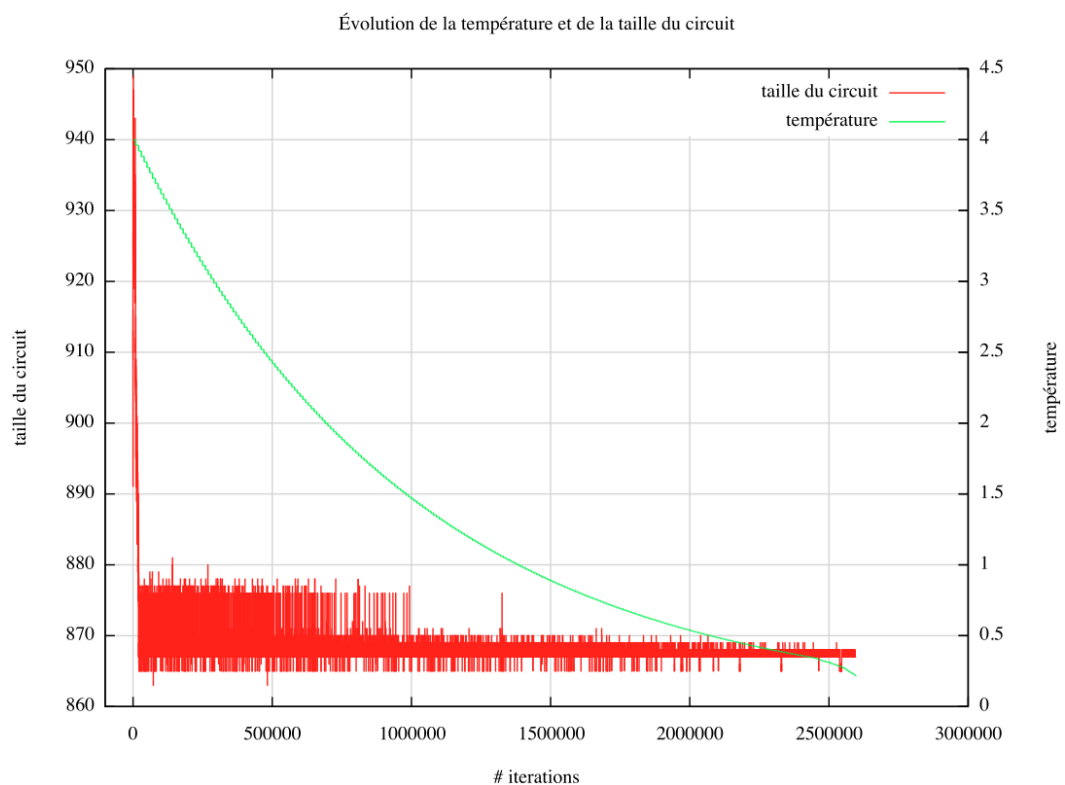


FIGURE 4.10 – Une session de recuit simulé appliquée au jeu *Blocker* et effectuée en appliquant les 4 transformations élémentaires *diagsplit*, *merge*, *factorize* et, avec une faible probabilité, *expand*. Le circuit se complexifie durant les tous premiers paliers à température élevée puis, rapidement, des sous-expressions communes sont identifiées et le circuit est simplifié. La meilleure solution obtenue est un circuit de taille 875 une fois sérialisé, la solution de départ ayant une taille de 935, ce qui représente une réduction de 6,4%.

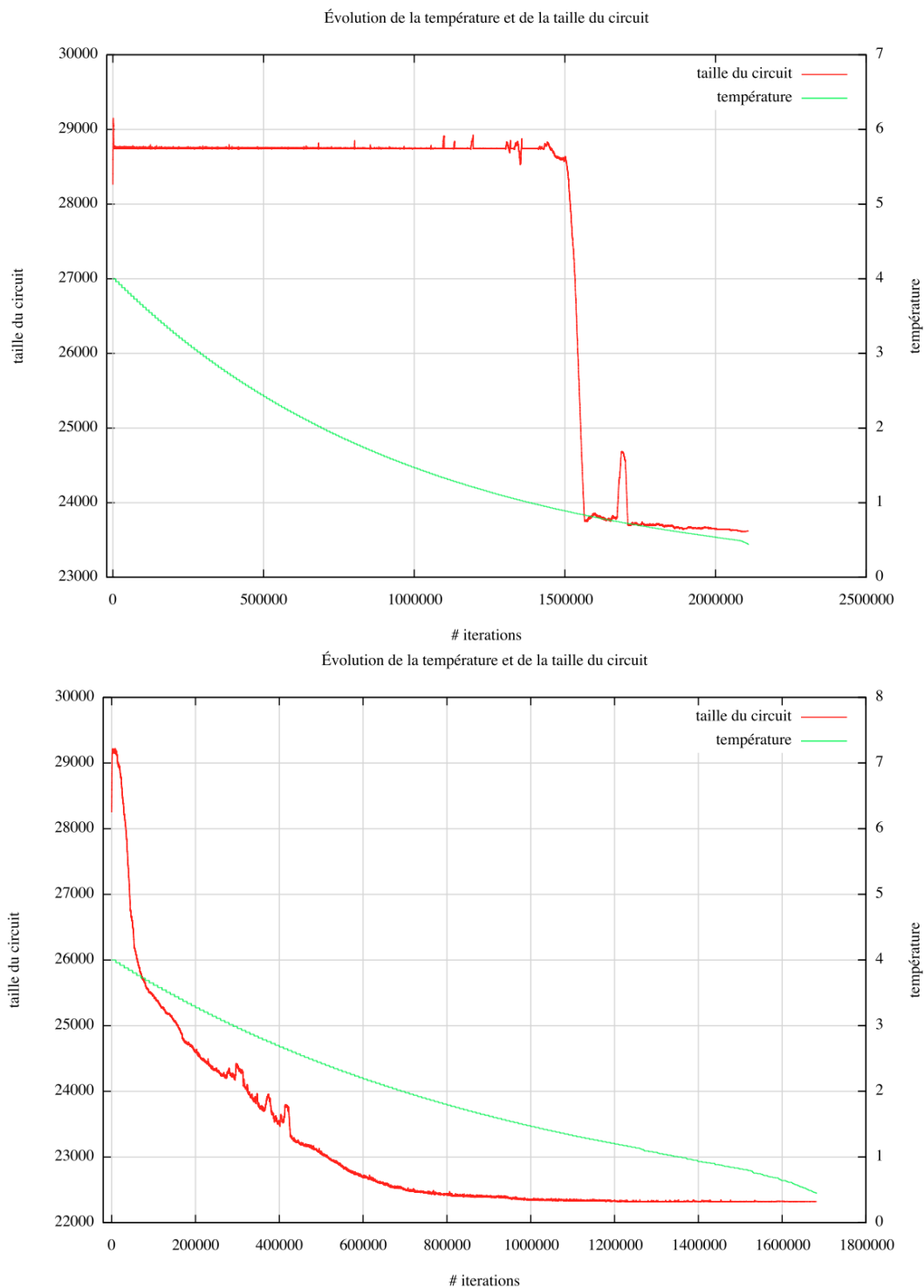
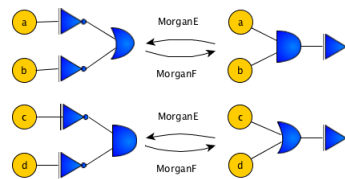


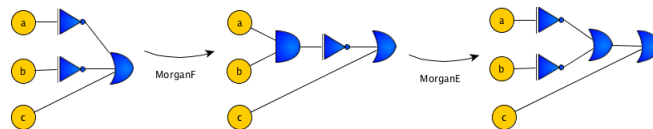
FIGURE 4.11 – Deux sessions de recuit simulé appliquées au jeu *EightPuzzle* et effectuées en appliquant les 3 transformations élémentaires *merge*, *factorize* et, avec une faible probabilité, *expand*. En haut, la 4^{ème} transformation élémentaire est *split*, en bas *dualSplit*. Nous constatons qu’avec *split* l’algorithme peine à trouver une simplification du circuit et stagne longtemps en explorant des solutions non-optimales. En revanche avec *dualSplit* la découverte de solutions de plus basse énergie est guidée et l’algorithme converge mieux.

faire apparaître des termes déjà utilisés et d'économiser ainsi plusieurs règles dans le circuit.

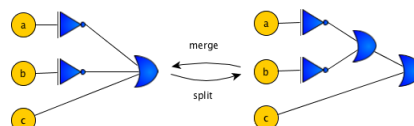
De plus, dans la version de février 2014 de LeJoueur, les disjonctions étant implicites et codées par plusieurs règles, leur poids est plus important que celui des conjonctions. Les lois de De Morgan peuvent nous permettre de profiter des négations pour transformer des disjonctions en conjonctions. Nous avons donc ajouté deux transformations élémentaires *MorganF* et *MorganE* qui correspondent respectivement à la factorisation d'une négation ou à son développement :



Ici encore, la restauration de l'état du circuit par une transformation inverse est problématique et l'utilisation d'une copie du circuit s'avère plus simple et moins lourde. En effet, pour pouvoir restaurer l'état du circuit quand une transformation est appliquée, il faut que les deux transformations soient exactement l'inverse l'une de l'autre, mais, dans le cas de *MorganF*, la porte ET ou OU originale comporte une entrée qui n'est pas une négation, l'inversion de la transformation ne permet pas de revenir au circuit d'origine :



Pour pouvoir inverser simplement une transformation *MorganF*, il faudrait donc en restreindre l'application à des portes logiques possédant uniquement des négations en entrée. Cependant ceci nécessiterait de disposer d'une transformation capable d'isoler les termes négatifs pour faire apparaître de telles portes. Il faudrait alors réutiliser la transformation *split* (avec *merge* comme transformation inverse), en lui donnant la liste des termes négatifs à isoler lors du dédoublement d'une porte.



L'usage d'une copie du circuit permet donc d'éviter l'utilisation d'une transformation élémentaire supplémentaire et la complexification des fonctions de transformation pour collecter les informations indispensables pour une restauration.

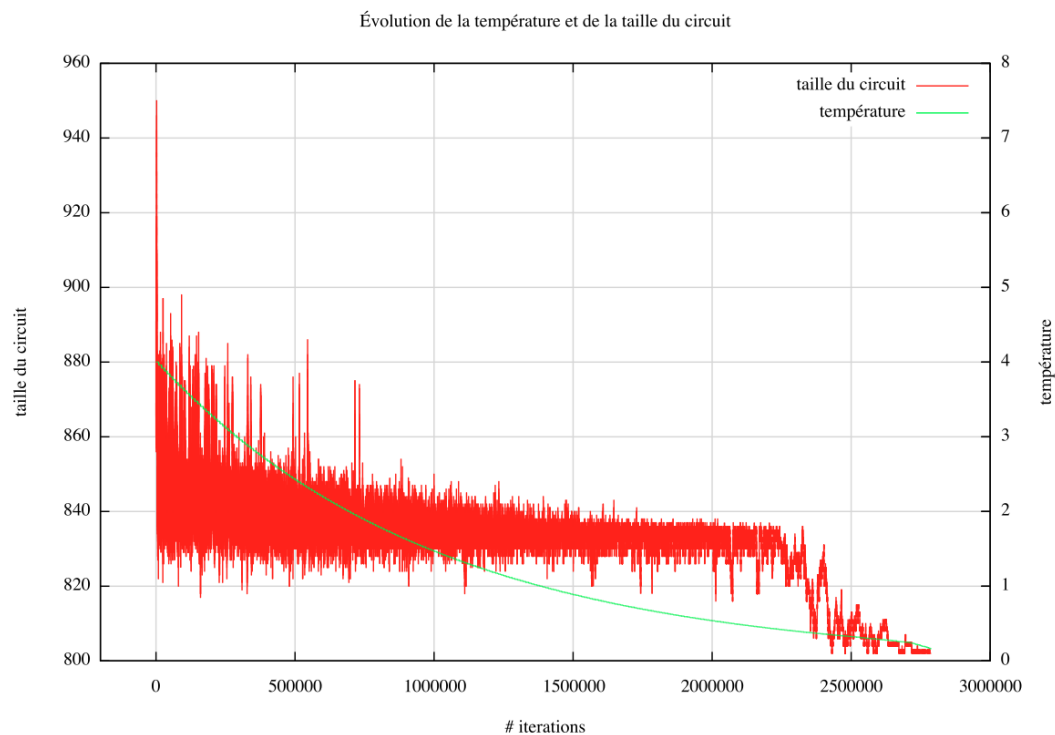


FIGURE 4.12 – Une session de recuit simulé appliquée au jeu *Blocker*. Les 4 transformations élémentaires *merge*, *dualSplit*, *factorize* et, avec une faible probabilité, *expand* sont appliquées ainsi que les transformations *MorganF*, *MorganE* et *doubleNot*. La meilleure solution obtenue est un circuit de taille 813 une fois sérialisé, la solution de départ ayant une taille de 935, ce qui représente une réduction de 13,05%, deux fois plus qu'en l'absence des transformations *MorganF*, *MorganE* et *doubleNot* (voir figure 4.10).

Une autre transformation rendue nécessaire par l'application des lois de De Morgan est l'élimination des doubles négations. Pour les supprimer nous avons créé une transformation spécifique nommée *doubleNot*.

Le jeu *Blocker* présente deux disjonctions importantes : *bridge* qui définit si un pont a été construit par le joueur devant traverser le plateau de jeu et *open* qui indique s'il reste une case vide sur ce plateau. Ces deux disjonctions sont utilisées sous leur forme négative mais *bridge* est également utilisé sous sa forme positive. En testant les transformations *MorganF*, *MorganE* et *doubleNot* sur *Blocker*, nous souhaitons voir si *open* est bien développé pour transformer la négation de la disjonction en conjonction de négations et si *bridge* est bien préservé.

La figure 4.12 montre l'évolution de l'algorithme de recuit avec l'ajout des nouvelles transformations. Nous constatons que pour une température initiale de 4 la convergence de l'algorithme est bonne et que la meilleure solution trouvée est meilleure que précédemment (voir figure 4.10). L'examen du meilleur circuit montre que la disjonction *bridge* a bien été préservée et que de nombreuses sous-expressions communes à plusieurs règles ont été trouvées. La disjonction *open* utilisée sous sa forme négative a quant à elle été transformée en une conjonction de négations comme attendu.

Jeux	taille initiale	avec 4 transformations principales	gain	toutes les transformations	gain
Blocker	935	875	6.42 %	813	13.05 %
Tictactoe	980	926	5.51 %	910	7.14 %
ConnectFour	2212	2120	4.16 %	2143	3.12 %
EightPuzzle	28272	22250	21.30 %	22566	20.18 %
CheckersTiny	149412	117396	21.43 %	117932	21.07 %
ChineseCheckers	2335782	-	-	-	-
Breakthrough	4091171	-	-	-	-
SheepAndWolf	6289028	-	-	-	-

FIGURE 4.13 – Pour chaque jeu testé, la taille initiale du circuit est comparée au meilleur circuit obtenu avec notre programme d’optimisation par recuit simulé en appliquant 4 transformations élémentaires *merge*, *dualSplit*, *factorize* et avec une faible probabilité *expand*. Une autre session de recuit est effectuée en ajoutant à ces transformations les transformations *MorganF*, *MorganE* et *doubleNot*, la taille indiquée est encore une fois celle du meilleur circuit obtenu. Le temps nécessaire à la convergence de l’algorithme de recuit ne nous a pas permis d’obtenir des résultats dans un temps raisonnable pour les circuits de grande taille.

4.6 Résultats

Nous avons testé notre algorithme de recuit simulé sur 8 jeux dont 3 ont des circuits qui ne tiennent pas dans la mémoire du Mppa. Les résultats sont présentés dans le tableau de la figure 4.13. Nous avons effectué deux tests différents avec uniquement les 4 transformations principales *merge*, *dualSplit*, *factorize* et *expand* appliqué avec une faible probabilité puis avec l’ensemble des transformations : les 4 transformations principales plus *MorganF*, *MorganE* et *doubleNot* pour gérer les négations.

Pour les jeux *Blocker*, *Tictactoe*, *ConnectFour* et *EightPuzzle* nous avons réalisé une dizaine de tests ; les résultats présentés correspondent au meilleur résultat trouvé durant ces différents tests. Pour le jeu *CheckersTiny*, le temps nécessaire au test ne nous a pas permis de renouveler l’expérimentation. Pour *ChineseCheckers*, *Breakthrough* et *SheepAndWolf* nous avons dû interrompre les tests avant la fin faute de temps.

Le temps de convergence du recuit va de quelques minutes pour *Blocker*, à quelques heures pour *ConnectFour*, deux jours pour *EightPuzzle* et plusieurs semaines voire plusieurs mois pour les jeux les plus lourds. Notre programme, comme nous l’avons dit, n’est pas du tout optimisé. Cependant la technique du recuit est connue pour sa lenteur de convergence et la découverte d’une solution optimale pour les jeux de grande taille semble devoir rester un processus assez long.

Nous constatons que les gains de taille obtenus par la simplification des circuits sont très variables d’un jeu à l’autre et dépendent des possibilités de simplifications offertes par les règles logiques. Les règles du jeu *eightPuzzle* présentent de nombreuses sous-expressions qui peuvent être isolées ce qui permet un gain de plus de 20% sur la taille du circuit en appliquant les 4 transformations élémentaires principales. En revanche, le circuit du jeu *ConnectFour* n’offre que très peu de possibilité de simplification et le meilleur gain obtenu n’est que de 4.16%.

Les simplifications opérant sur les négations apportent beaucoup au jeu *Blocker*

(voir section 4.5.6) et permettent de doubler le gain de taille obtenu par la simplification. Par contre, elles n'apportent rien pour les jeux *ConnectFour* et *eightPuzzle*.

Les règles du jeu *EightPuzzle* (*Pousse-Pousse 3x3*) ne présentent qu'une seule négation (not inorder) : le terme inorder indique si les tuiles sont placées dans l'ordre. Il est utilisé sous sa forme positive pour déterminer une partie gagnante et sous sa forme négative dans le cas contraire. Le terme étant utilisé également sous sa forme positive, la transformation de la négation ne peut apporter aucun gain.

Il en est de même pour le jeu *ConnectFour* (*Puissance 4*) qui possède pourtant un circuit contenant beaucoup de négations : pour qu'un pion soit dans une case il faut que la case au dessous ne soit pas vide, pour qu'une case soit considérée vide elle ne doit pas contenir de pion noir ni de pion rouge, pour pouvoir jouer dans une colonne elle ne doit pas être pleine, etc. Ces faits négatifs se retrouvent principalement dans des conjonctions et sont également utilisés sous leur forme positive, la transformation de ces négations n'apporte donc pas d'amélioration dans le poids du circuit.

Pour *eightPuzzle*, *ConnectFour* et *CheckersTiny* la diversification des transformations a même un effet négatif : elle augmente l'espace de recherche et entrave la convergence de l'algorithme. Ceci est visible dans la table des résultats : les meilleures solutions obtenues en appliquant toutes les transformations élémentaires sont moins bonnes qu'avec les 4 transformations principales seulement. La figure 4.14 montre qu'à haute température, les transformations élémentaires supplémentaires entraînent une exploration plus importante et l'apparition de solutions plus éloignées de l'optimale en début de recherche. Par la suite l'algorithme de recuit peine à trouver des simplifications du circuit, la taille du circuit stagne longtemps et la solution finale est bien moins bonne que celle de départ.

La température initiale calculée par une expérimentation préalable est de 2. Nous constatons cependant qu'avec cette température initiale et seulement les 4 transformations principales le meilleur circuit obtenu présente encore de nombreuses expressions qui auraient pu être simplifiées, essentiellement des ALIAS qui n'ont pas fusionné. Nous avons donc augmenté la température et allongé les chaînes de Markov dans le but de favoriser la découverte d'une solution plus proche de l'optimal.

En augmentant la température initiale à 7 et la longueur des chaînes de Markov à 100000 (voir figure 4.15), nous constatons que le meilleur circuit obtenu avec les 4 transformations principales est quasiment optimal, mais l'algorithme se perd encore pendant une longue période dans l'exploration de solutions non optimales. L'ajout des transformations supplémentaires provoque le même phénomène de ralentissement de la convergence de l'algorithme que précédemment.

Finalement, nous constatons qu'avec un gain de taille maximum autour de 20% pour les circuits représentant les règles des jeux utilisés en General Game Playing, notre programme ne nous permet pas de réduire suffisamment leur taille pour transmettre ces circuits au *Mppa*. En revanche, nous constatons que nous pouvons simplifier des circuits logiques de grande taille présentant un grand nombre d'entrées et de sorties (voir figure 4.16) ce qui peut être utile pour d'autres domaines de re-

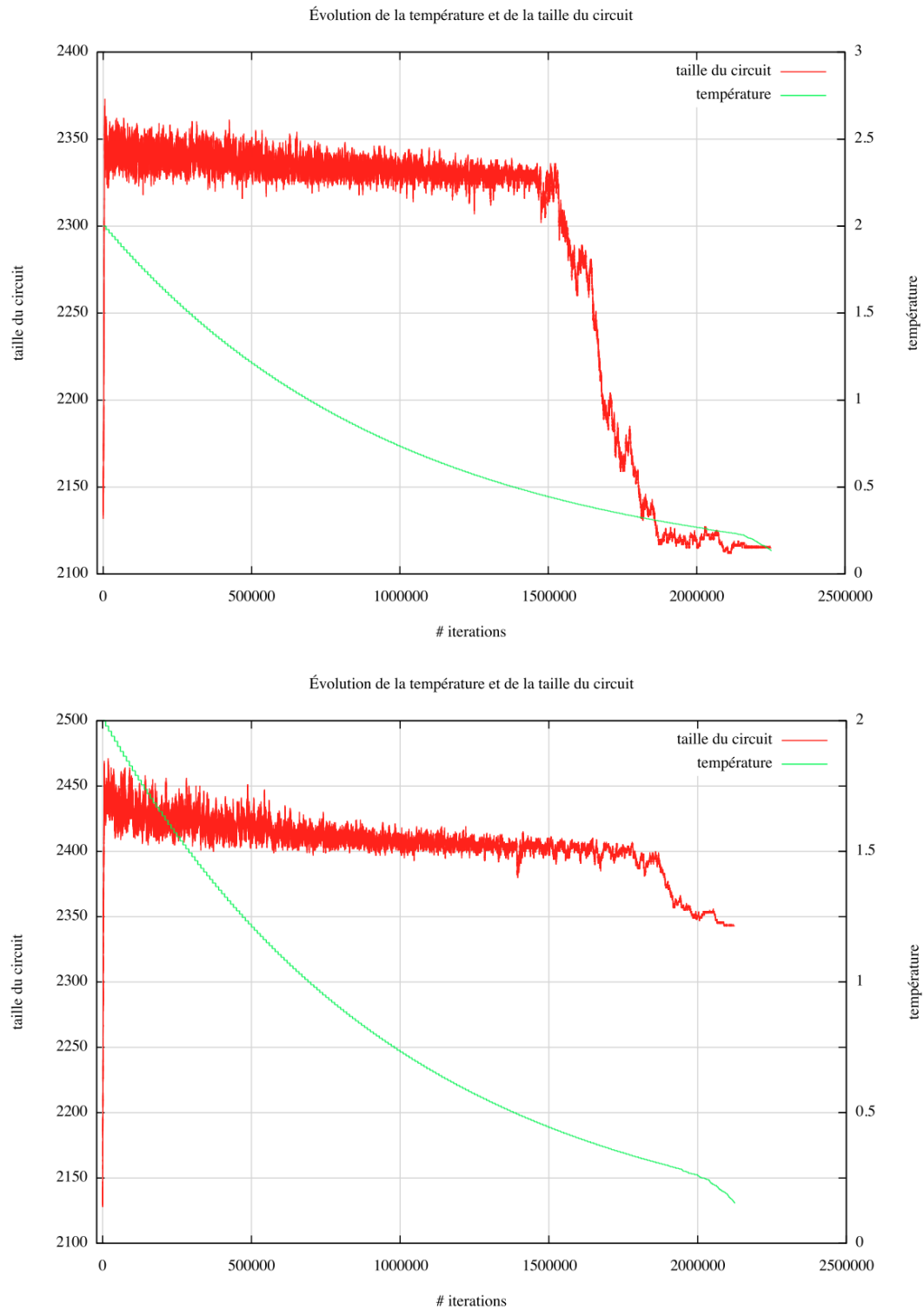


FIGURE 4.14 – Deux sessions de recuit simulé appliquées au jeu *ConnectFour*. En haut, les 4 transformations élémentaires *merge*, *dualSplit*, *factorize* et, avec une faible probabilité, *expand* sont appliquées. En dessous, les transformations *MorganE*, *MorganF* et *doubleNot* sont ajoutées. La température initiale est de 2 et 10000 itérations sont réalisées par palier. Nous constatons que les transformations supplémentaires ne permettent pas de réduire davantage la taille du circuit et entravent au contraire la convergence de l'algorithme.

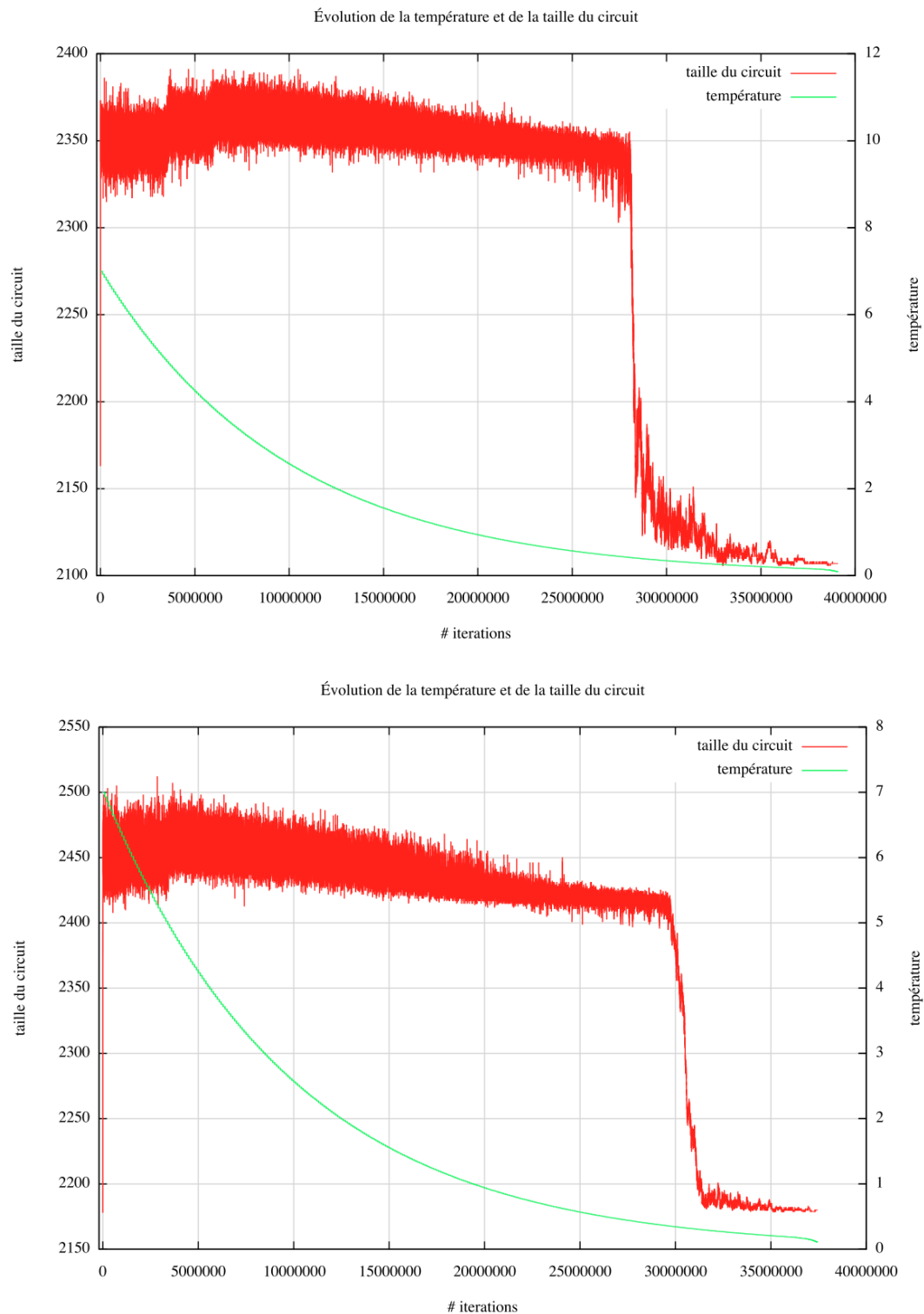


FIGURE 4.15 – Deux sessions de recuit simulé appliquées au jeu *ConnectFour*. En haut, les 4 transformations élémentaires *merge*, *dualSplit*, *factorize* et, avec une faible probabilité, *expand* sont appliquées. En dessous, les transformations *MorganE*, *MorganF* et *doubleNot* sont ajoutées. La température initiale est de 7 et 1000000000 itérations sont réalisées par palier. Nous constatons que l'augmentation de la température initiale et l'allongement des chaînes de Markov permettent à l'algorithme de trouver un circuit de taille plus réduite avec 4 transformations (voir figure 4.14). Cependant l'ajout des transformations *MorganE*, *MorganF* et *doubleNot* a toujours un effet négatif sur la convergence de l'algorithme.

Jeux	nombre d'entrées (<i>true, does</i>)	nombre de sorties (<i>next, legal, terminal, goal</i>)
Blocker	79	83
Tictactoe	48	54
ConnectFour	115	121
EightPuzzle	151	153
CheckersTiny	597	601
ChineseCheckers	4320	4332
Breakthrough	16646	8457
SheepAndWolf	8387	8391

FIGURE 4.16 – Nombre d'entrées et de sorties pour les circuits des différents jeux testés.

cherche.

4.7 Futurs développements

Pendant nos tests, nous avons utilisé des probabilités équivalentes pour le choix de chaque transformation élémentaire sauf pour *expand* qui doit être appliqué plus rarement pour ne pas entraver la convergence de l'algorithme. Nous avons vu que les transformations des négations ne sont pas toujours utiles voire néfastes pour la découverte d'une solution optimale. De plus amples recherches sont nécessaires pour évaluer dans quelle proportion chaque transformation doit être appliquée, le dosage de chaque transformation pouvant être fonction de la nature du circuit. La probabilité de transformer les négations pourrait être, par exemple, proportionnelle à la quantité de négations de disjonctions ou de disjonctions de négations présentes dans le circuit.

Nous avons empiriquement fixé un écart de 1/100^{ème} de la taille du circuit entre les applications de *expand* et de *factorize* ; d'autres manières de limiter l'application de cette transformation doivent être expérimentées. Par exemple, cette transformation pourrait ne plus être applicable quand la taille du circuit s'éloigne d'un pourcentage donné de la meilleure solution trouvée.

D'autres transformations peuvent également être explorées pour aider la découverte de nouvelles simplifications. Nous n'avons pas implémenté l'équivalent inverse de *doubleNot*, or la suppression des doubles négations est susceptible de bloquer la recherche dans un minimum local. L'ajout de doubles négations peut inversement permettre une transformation des règles aboutissant à une réduction du poids des circuits (voir figure 4.17).

Notre postulat de départ, i.e. garantir l'unicité des nœuds du graphe, engendre de nombreuses complications pour l'évaluation de la solution et l'application des transformations. Lors des transformations, si deux nœuds à modifier sont parents l'un de l'autre, la modification du premier peut entraîner une mise à jour du second lors de la propagation des effets ; le second nœud a alors changé d'adresse avant que la modification ne lui soit appliquée. Le code du programme est alors alourdi par de nombreux tests nécessaires pour garantir la cohérence du circuit.

Pour simplifier notre programme, nous pourrions renoncer à ce postulat d'uni-

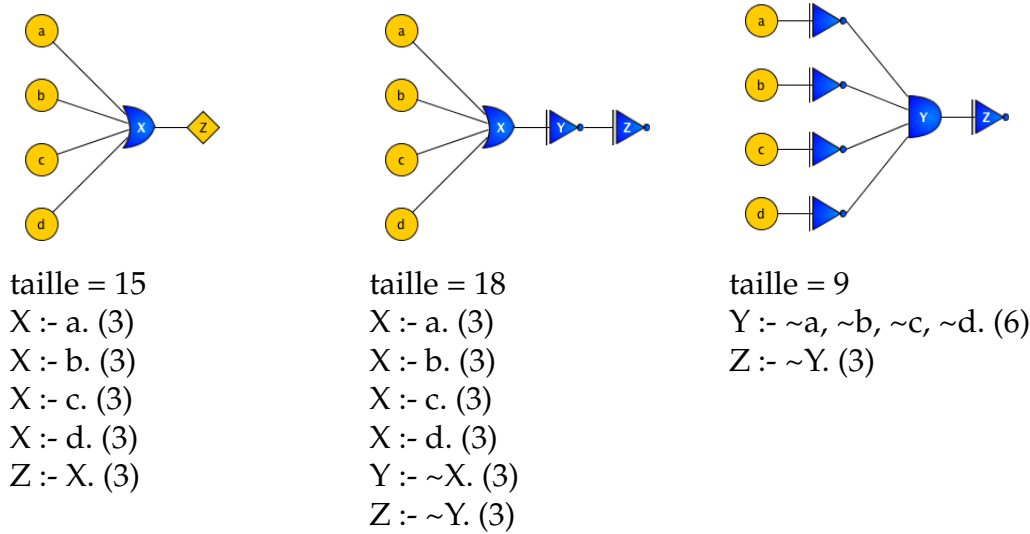


FIGURE 4.17 – Réduction de la taille d'un circuit (à gauche) par injection d'une double négation (au centre) puis simplification avec la transformation élémentaire *MorganE* (à droite). Entre parenthèses figure la taille de chaque règle.

citée et ajouter une transformation élémentaire qui réalise la fusion des nœuds identiques. Il ne serait alors plus nécessaire de se préoccuper de la propagation des effets au moment d'appliquer les autres transformations élémentaires et il serait possible d'évaluer localement leur effet avant de les appliquer. Cependant, certaines transformations augmenteraient systématiquement la taille du circuit puisqu'elles provoqueraient l'apparition de doublons.

L'implémentation de cette solution alternative est nécessaire pour évaluer dans quelle mesure elle peut préserver la convergence de l'algorithme de recuit simulé et dans quelle proportion elle peut accélérer la vitesse de fonctionnement du programme.

Une autre approche proposée par [Ruijl et al. \[2013\]](#) pour la simplification d'expressions algébriques de grande taille pourrait être appliquée à la simplification de nos circuits. L'approche de [Ruijl et al. \[2013\]](#) consiste à utiliser le principe du recuit simulé sur l'algorithme UCT. La température du recuit agit directement sur le paramètre C (voir equation 1.2) et fait varier l'équilibre entre exploration et exploitation dans le temps. A température élevée, l'exploration est plus importante et elle diminue jusqu'à concentrer UCT uniquement sur l'exploitation de la meilleure branche. Cette approche pourrait permettre d'obtenir une solution optimale plus rapidement en concentrant la recherche sur les séquences de transformations les plus prometteuses.

4.8 Solution en amont

Il nous semble peu probable malgré les différentes améliorations que nous avons évoquées, que notre programme de simplification des circuits puisse opérer cette simplification dans les conditions strictes des compétitions de *General Game Playing*.

Durant une conversation avec Vittaut [2014] nous avons constaté que l'utilisation des règles de type *input* et *base*, données dans la description GDL des jeux pouvaient provoquer la création de nombreuses règles inutiles lors du *Grounding*. Les *inputs* indiquent les faits qui constituent des actions possibles des joueurs. Les *bases* sont les faits décrivant l'état du jeu ; ils sont déduits de la position précédente et des actions choisies. Ces *inputs* et *bases* peuvent être déduits des règles du jeu et ne sont donnés que pour simplifier l'étape du *grounding* pour les joueurs utilisant des *propnets*.

Pour un jeu comme *Tictactoe*, si le fait qu'une case peut être vide est indiqué comme *base*, une règle permettant de déterminer l'alignement de 3 symboles peut provoquer la création de règles indiquant des alignements de cases vides ce qui n'a aucun intérêt pour la logique du jeu et alourdit le circuit inutilement. Pour *Breakthrough* nous avons ainsi constaté qu'en prenant en compte les *inputs* et les *bases* donnés dans la règle du jeu du serveur *Tiltyard* [Schreiber, 2014] le circuit atteint environ 1 million de règles, alors qu'en recalculant les *inputs* et les *bases* à partir des règles du jeu, le circuit passe à environ 45000 règles.

La version de *LeJoueur* de février 2014 que nous avons utilisée présente un bogue qui ne nous a pas permis de forcer le re-calcul des *inputs* et des *bases* pour estimer le gain possible sur la taille des circuits. Cependant, dans la nouvelle version de son joueur, Jean-Noël Vittaut a amélioré sa technique de *Grounding*. Le calcul des *inputs* et des *bases* prend un peu de temps mais le gain obtenu dans la taille des circuits et donc dans la vitesse de calcul des *playouts* contrebalance cet inconvénient.

La version de *LeJoueur* de mai 2014 présente de nombreuses autres modifications améliorant significativement la taille des circuits. Chaque règle a désormais une taille fixe et représente une porte logique à deux entrées pour les OU et ET et à une seule entrée pour les NON. Les OU ne sont donc plus codés implicitement comme en Prolog et les NON ne sont plus codés par une modification des pointeurs. Chaque strate du circuit est constituée d'une liste de disjonctions (OU), de conjonctions (ET) et de négations (NON), chaque booléen étant désigné par son index dans la liste de valeurs et non par son adresse. L'amélioration qui réduit le plus la taille des circuits vient du fait que le *grounding* n'est plus effectué pour toutes les règles du jeu mais uniquement pour les règles dont le conséquent contient *TERMINAL*, *LEGAL*, *GOAL* ou *NEXT*. Les autres règles ne sont examinées que si elles sont réellement nécessaires pour effectuer l'instanciation de ces dernières. Le circuit est ensuite simplifié en effectuant des factorisations : le terme le plus fréquent est factorisé en premier, la simplification obtenue n'est donc pas forcément optimale mais elle s'en approche. Au final, tous ces traitements effectués en amont permettent d'obtenir des circuits de taille bien moindre.

Jeux	taille du circuit (02/2014)	taille du circuit (05/2014)
Blocker	935	933
Tictactoe	980	790
ConnectFour	2212	2967
EightPuzzle	28272	2559
CheckersTiny	149412	28495
ChineseCheckers	2335782	134242
Breakthrough	4091171	269752
SheepAndWolf	6289028	398854

FIGURE 4.18 – Taille des circuits obtenus avec le nouveau codage de *LeJoueur* version 05/2014, l'ajout d'optimisations au niveau du *grounding* et la non-prise en compte des *inputs* et des *bases* quand ils provoquent la création de nombreuses règles inutiles.

Comme chaque règle à une taille fixe, seules les tailles des listes de ET, OU et NON de chaque strate doivent être codées et les valeurs codant la taille de chaque règle sont économisées. Les tailles des circuits des deux versions de *LeJoueur* sont comparées dans la table de la figure 4.18. Le jeu *ConnectFour* est le seul dont le circuit se trouve plus important avec le nouveau codage. Ceci s'explique, comme nous l'avons vu dans la section 4.6, par le fait que le circuit de ce jeu présente peu de simplifications possibles par factorisation et, malgré l'économie de la taille de chaque règle, le dédoublement des règles pour n'obtenir que des expressions à 1 ou 2 antécédents amène une augmentation de la taille du circuit.

Conclusion

Nous avons présenté une première approche pour la parallélisation d'un *General Game Player* sur une architecture MPPA-256 de Kalray.

Le joueur utilisé est *LeJoueur* de Jean-Noël Vittaut. Il fait usage de *propnets* pour interpréter les règles GDL des jeux et utilise l'algorithme UCT pour raisonner sur le jeu et choisir les coups les plus prometteurs.

La parallélisation de UCT permet de réaliser un plus grand nombre de simulations de parties (*playouts*) dans un même laps de temps et de construire ainsi un arbre du jeu de plus grande taille et dont les nœuds, représentant les coups possibles à chaque étape du jeu, sont évalués de manière plus fiable. Cependant, la parallélisation induit des coûts en synchronisation au niveau des différents processus, des temps de communication pour les échanges de données et peut entraîner un surcoût en calcul si elle oblige à recalculer certains éléments plusieurs fois. Nous avons présenté plusieurs techniques qui ont été proposées dans la littérature pour la parallélisation de UCT : parallélisation à la racine, au niveau de l'arbre, ou au niveau des feuilles, ainsi que des techniques utilisant un arbre distribué.

Nous avons présenté la technique des *propnets* qui permet d'accélérer l'interprétation des règles des jeux et réduit les temps de calcul des *playouts*. Ces temps de calcul plus courts rendent les coûts en synchronisation et en communication plus importants en proportion et compliquent l'implémentation d'une parallélisation performante.

Nous avons exploré les capacités offertes par la carte MPPA-256 de Kalray et les options de parallélisation disponibles pour *LeJoueur*.

Parallélisation de *LeJoueur*

L'architecture MPPA est composée d'une machine hôte, d'un processeur d'I/O chargé des échanges de données et de 16 clusters de 16 processeurs élémentaires chacun. Le MPPA étant une machine encore expérimentale, nous avons utilisé des communications synchrones afin de simplifier l'implémentation des communications et l'analyse des résultats. Ceci nous permet également d'obtenir des résultats de référence pour pouvoir comparer plus tard les performances d'autres types de

communications.

Nous avons montré que les limitations de l'architecture MPPA, plus particulièrement la mémoire des clusters limitées à 2Mo, nous contraint à implémenter une *parallélisation aux feuilles* de l'algorithme UCT et interdit tout autre type de parallélisation, ce qui s'avère problématique étant donné que la *parallélisation aux feuilles* est présentée comme la moins performante dans la littérature.

Nous avons montré que l'approche la plus efficace est de confier le calcul d'un *playout* complet à un seul processeur élémentaire. Le partage du calcul d'un *playout* entre les différents PE d'un cluster entraîne des temps de synchronisation trop importants dûs en partie au fait que les règles calculées par chaque PE sont de longueurs variables.

Nous avons mis en évidence le fait que les threads doivent être relancés à chaque demande de calcul sur les processeurs élémentaires. Si les threads sont gardés actifs depuis le démarrage de l'application, les temps de communication avec les threads sont trop importants, même avec une attente active, et le nombre de *playouts* calculés par seconde devient inférieur à celui obtenu par la première approche à partir de 12 threads⁸.

Nous avons constaté que le MPPA permet un bon passage à l'échelle quand les communications augmentent en taille : les temps de communication sont multipliés par 5 quand le volume des données transmises est multiplié par 10. La taille des positions augmentant avec la complexité des jeux, ceci nous permet d'espérer qu'une parallélisation d'un joueur GGP sur MPPA restera performante même pour des jeux complexes.

Les résultats que nous avons obtenus lors de nos expériences visant à évaluer les temps de communication et synchronisation comparés au temps de calcul des *playouts* se sont avérés incohérents. Nous avons obtenu des temps de fonctionnement de notre application plus courts en réalisant 1000 envois de données accompagnés du calcul d'un *playout* qu'en réalisant les communications seules. Nous avons également été surpris de constater qu'avec les communications seules, la différence de temps de fonctionnement de l'application est bien plus importante lorsque l'on passe de 1 à 16 threads sur un seul cluster ou sur seize clusters que lorsque l'on réalise la même augmentation sur quatre clusters. De plus amples expérimentations et une analyse de l'architecture MPPA seront nécessaires pour comprendre l'origine de ces résultats.

Nos expérimentations nous ont permis de mettre en évidence quelques dysfonctionnements du pack logiciel MPPA ACCESSCORE fourni avec le MPPA. Avec la version 1.0.1. nous avons rencontré des blocages inopinés de l'application hôte lors du lancement des binaires sur la carte MPPA (`mppa_spawn`). La version 1.2.1. ne présente plus ce problème mais provoque un blocage systématique de l'application à la fermeture des descripteurs de fichiers correspondant aux canaux de communication

8. Gailhard [2014] de la société Kalray s'est montré très surpris de ces résultats et suggère de plus amples tests pour en comprendre l'origine.

ouverts entre la machine hôte et la carte MPPA.

La limitation de la mémoire au niveau des clusters, constitue un obstacle insoluble pour une parallélisation vraiment intéressante de *LeJoueur*⁹. Une parallélisation à la racine ou utilisant un arbre distribué nous semble plus prometteuse pour améliorer à la fois la taille de l'arbre UCT, le nombre de simulations et le niveau de jeu de *LeJoueur*. Cependant la carte MPPA-256 n'est que le premier membre de la famille MPPA MANYCORE et son développement est en cours. De plus nous n'avons pas encore testé l'utilisation de communications asynchrones qui peuvent réduire de manière conséquente les temps de communication et de synchronisation.

Simplification des circuits

Au cours de nos expérimentations, nous avons été bloqués par la limitation de la mémoire à 2Mo au niveau des clusters. Cette mémoire très réduite ne permet pas de transmettre les règles de jeux intéressants au MPPA : les jeux non triviaux présentent des *propnets* de taille trop importante pour tenir dans une mémoire aussi limitée.

Pour résoudre ce problème nous avons implémenté un algorithme de recuit simulé dans le but de réduire la taille des circuits logiques composant le *propnet*. Le recuit s'est avéré efficace pour réduire les circuits logiques mais bien trop lent pour que cette technique puisse être utilisée dans le cadre strict des compétitions de *General Game Playing* (GGP). De plus la réduction obtenue est totalement dépendante des possibilités de simplifications offertes par chaque circuit et reste insuffisante pour permettre au circuit optimisé de tenir en mémoire dans les clusters du MPPA.

Une solution a finalement été trouvée en amont par Jean-Noël Vittaut. Une simplification au moment du *grounding* permet d'éliminer dès l'origine les règles inutiles et permet de procéder à diverses optimisations lors de la transformation des règles GDL en *propnet*. En plus de l'amélioration du *grounding*, la version de mai 2014 de *LeJoueur* présente d'importantes modifications des structures de données utilisées pour représenter les *propnets*. La taille des circuits n'est désormais plus un problème pour la transmission des données sur le MPPA. Des optimisations pourraient éventuellement être encore apportées aux circuits mais la limitation de temps imposée par le GGP oblige à se contenter d'un circuit de taille acceptable mais non optimale.

Futurs développements

Ayant été bloqués dans nos recherches par le problème de la taille des circuits, nous n'avons pas eu le temps de tester l'emploi de communications asynchrones.

9. Nous sommes d'avis qu'une mémoire aussi réduite au niveau des clusters constitue une contrainte technique majeure pour un grand nombre d'applications parallèles et pas seulement pour la parallélisation de UCT.

Le premier développement à apporter est donc la modification des communications pour traiter les *playouts* à la demande et la comparaison des résultats avec ceux obtenus en utilisant des communications synchrones.

Il faut maintenir une queue de positions au niveau de l'hôte, à partir desquelles le calcul de *playouts* est souhaité. L'I/O doit récupérer ces positions et les accumuler dans une liste dans laquelle tous les clusters peuvent piocher. Dans chaque cluster, une liste de positions à traiter peut être maintenue pour fournir un travail à 14 des processeurs élémentaires (PE), le 15^{ème} PE étant occupé à récupérer les positions auprès de l'I/O de manière à maintenir le cache et éviter une famine. Le 16^{ème} PE peut envoyer les résultats des *playouts* à l'I/O à mesure de leur disponibilité. L'I/O possédant un processeur 4 cœurs, l'envoi des positions et la récupération des résultats peuvent être effectués en parallèle par deux threads distincts. La superposition des temps de calcul et des temps de communication doit permettre un gain de temps conséquent.

Pour les communications entre l'hôte et l'I/O, nous avons vu que le MPPA fournit un type `mqueue` permettant de gérer des listes de travaux. La taille des messages et leur nombre sont fixés par l'utilisateur mais ne sont pas limités par une valeur seuil.

Entre l'I/O et les clusters nous avons vu que le nouveau type de communication `sampler` permet à l'I/O de remplir une liste de travaux consommés par différents clusters. Cependant ce type de communication, conçu pour communiquer un flux d'échantillons continu, est prévu pour que les échantillons non consommés avant l'arrivée des échantillons suivants soient écrasés par ceux-ci. La documentation du MPPA indique que ce connecteur peut être configuré pour éviter la perte d'échantillons non consommés à temps : il convient de vérifier cette fonctionnalité dans la pratique.

Le type `rqueue` permet de récolter les résultats des *playouts* en provenance des différents clusters dans une queue dont chaque message peut être traité ensuite par l'I/O. Une difficulté posée par le type `rqueue` est la limitation des messages à 120 octets. Ceci rend nécessaire le codage des booléens sur 1 seul bit plutôt que d'utiliser des `short int` et nécessite la création d'un objet C++ spécifique pouvant convertir les vecteurs de booléens en table de booléens codés bit à bit, et permettant d'accéder aux valeurs de chacun de ces bits pendant le calcul des circuits.

Le codage des données utilisé par *LeJoueur* a été complètement remanié par Jean-Noël Vittaut dans la nouvelle version de mai 2014¹⁰. Ce nouveau codage des circuits permet une accélération conséquente des calculs pendant les *playouts*¹¹ (position terminale, score, coups légaux, position suivante), les temps de communication et

10. Il est à noter que *LeJoueur* est en constante évolution. Alors que *LeJoueur* avait terminé 7^{ème} sur 16 en 2013, la nouvelle version de *LeJoueur* a fini 2^{ème} sur 17 concurrents à la compétition de GGP de 2014.

11. Chaque règle représente une porte logique ET ou OU avec un conséquent et seulement deux antécédents ou une porte NON avec un conséquent et un seul antécédent. Le calcul des règles peut donc être codé par des opérations binaires très rapides, ce qui accélère considérablement l'évaluation des circuits malgré l'augmentation du nombre de règles.

de synchronisation risquent donc de peser encore plus lourdement et la réalisation d'une parallélisation performante est un véritable challenge.

De plus, Jean-Noël Vittaut a testé une version de son joueur utilisant des *playouts* 64bit : la liste de booléens est remplacée par une matrice de booléens, chaque indice d'un tableau C correspond à une variable de 64 bits dont chaque bit représente la valeur du fait dans un *playout* distinct. Un groupe de 64 *playouts* peut donc être calculé en parallèle.

Cependant Jean-Noël Vittaut a constaté avec son nouveau joueur que développer l'arbre pour réaliser 64 *playouts* simultanés n'apportait pas d'amélioration du niveau de jeu, des *playouts* inutiles étant réalisés dans des zones sous-optimales de l'arbre. Dans une réalisation séquentielle des *playouts*, les résultats des *playouts* précédents indiquent si une zone de l'arbre est peu intéressante et les *playouts* suivants ne sont pas exécutés dans cette zone.

De plus, les *playouts* 64 bits de Jean-Noël Vittaut sont réalisés à partir de positions différentes, mais quand un *playout* est plus long que les autres, les *goals* des *playouts* achevés sont collectés et les étapes des *playouts* continuent d'être calculées uniquement pour les bits correspondant aux parties non achevées. Il y a donc une attente générée par les *playouts* les plus longs.

Utiliser la technique de *virtual loss* [Chaslot, 2010] avec des communications asynchrones sur le MPPA permettrait de lancer les *playouts* suivants sur d'autres positions en attendant le résultat du *playout* lancé à partir de la meilleure position et pourrait éviter une baisse du niveau de jeu. Il reste néanmoins à tester si la superposition des temps de communication avec les temps de calcul permet d'obtenir un meilleur résultat que les *playouts* 64 bits tentés par Jean-Noël Vittaut.

Une toute autre approche pourrait également être envisagée. Cazenave [2014] suggère l'utilisation et la parallélisation de la technique Nested Monte-Carlo qui demande peu de mémoire. Cependant cette technique n'est applicable qu'aux jeux à un seul joueur ce qui en limite l'intérêt. Les 2Mo disponibles au niveau des clusters étant déjà partiellement utilisés pour stocker le *propnet*, cette approche nous semble de toute façon difficile à mettre en œuvre.

Annexe : règles des jeux évoqués

Blocker : Sur un damier 4x4 cases, deux joueurs *blocker* et *crosser* choisissent simultanément dans quelle case ils souhaitent poser leur marque. Si le joueur *blocker* a choisi la même case que *crosser*, alors c'est *blocker* qui pose sa marque dans la case. Deux marques sont considérées contigües si elles sont voisines sur la même ligne ou en diagonale. Si *crosser* arrive à rejoindre les deux bords droit et gauche du damier avec une série contigüe de marques, il met fin à la partie et gagne 100 points (voir image ci-dessous). Si aucune case vide ne subsiste, le jeu prend fin et *blocker* remporte 100 points. Il s'agit d'un jeu asymétrique qui, pour deux joueurs de force égale, offre beaucoup plus de chances de victoire à *blocker*.

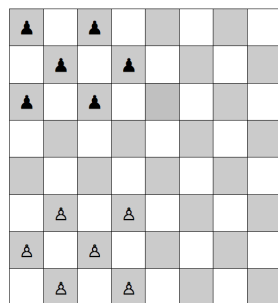
	B		
C	B		C
C	B	C	B
	C	B	

Breakthrough : Chaque joueur dispose de deux rangées de pions à chaque extrémité d'un damier de 8x8 cases (voir image ci-dessous). A chaque tour l'un des joueurs peut avancer un pion d'une case en avant ou en diagonale. Une case ne peut contenir qu'une seule pièce à la fois. La capture d'une pièce adverse ne peut être effectuée qu'en se déplaçant en diagonale, le pion occupe alors la place devenue vacante. Le premier joueur à atteindre l'autre côté du damier met fin au jeu¹² et remporte 100 points, l'autre 0. *Breakthrough* est un jeu difficile pour UCT à cause de la profondeur et de l'important facteur de branchement de l'arbre du jeu.

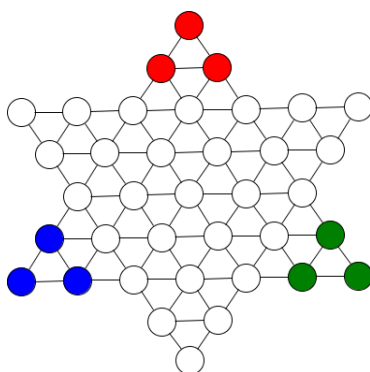
▲	▲	▲	▲	▲	▲	▲	▲
▲	▲	▲	▲	▲	▲	▲	▲
△	△	△	△	△	△	△	△
△	△	△	△	△	△	△	△

12. Michael Genesereth a apporté une correction aux règles GDL de ce jeu en précisant que le jeu prend également fin si l'adversaire s'est fait prendre tous ses pions.

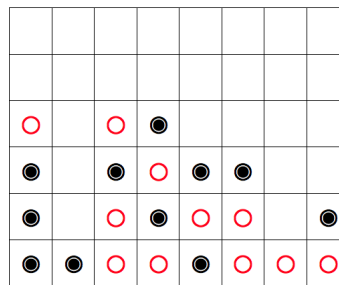
CheckersTiny : Il s'agit d'un jeu de *Dames Anglaises* où chaque joueur commence avec seulement 6 pions (voir image ci-dessous). Le jeu prend fin au bout de 102 coups ou bien lorsque l'un des joueurs n'a plus de coup légal. Si l'un des joueurs possède plus de pièces que l'autre en fin de jeu, il gagne 100 points et l'autre 0. Si les joueurs sont ex-æquo ils gagnent chacun 50 points.



ChineseCheckers : Il s'agit d'un jeu de *Dames Chinoises* à 3 joueurs, disputé sur un plateau en forme d'étoile à 6 branches (voir image ci-dessous). Les joueurs possèdent chacun 3 pions placés sur l'une des branches de l'étoile, la branche opposée restant libre. Chaque joueur, à son tour, peut avancer un de ses pions de la case où il se trouve à l'une des cases voisines reliées par un trait. Si l'un de ses pions ou un pion adverse se trouve dans l'une des cases voisines, il peut alors sauter par dessus et placer son pion dans la case suivante ; il peut enjambrer de cette manière deux pions tour à tour, comme aux *Dames Anglaises*, mais sans les prendre. Le but de chaque joueur est d'amener ses trois pions dans la branche opposée de l'étoile. S'il y parvient il obtient 100 points. S'il n'amène que 2 pions à destination il obtient 50 points, 25 s'il n'en amène qu'un seul et 0 si aucun de ses pions n'est arrivé à la fin du jeu. Le jeu prend fin au bout de 60 coups ou plus tôt si l'un des joueurs a atteint les 100 points.



ConnectFour : Il s'agit du jeu connu en France sous le nom de *Puissance 4* qui consiste à empiler des pions sur 7 ou 8 colonnes (voir image ci-après) jusqu'à former un alignement de 4 pions. Les joueurs jouent tour à tour et placent un pion de leur couleur (noir ou rouge) sur une des piles. Le premier joueur qui arrive à aligner 4 pions à l'horizontale, verticale ou en diagonale gagne 100 points et l'autre 0. En cas de match nul chacun reçoit 50 points.



EightPuzzle : Il s'agit d'un jeu à un seul joueur connu en France sous le nom de *pousse-pousse 3x3*. Il est constitué d'une grille de 3x3 cases contenant 8 tuiles numérotées de 1 à 8 et un espace vide. Au début du jeu, les tuiles sont dans le désordre. A chaque coup, le joueur peut faire glisser une tuile située à côté de l'emplacement vide vers celui-ci, le but étant de replacer les tuiles dans l'ordre comme indiqué sur l'image ci-dessous. Si le joueur y parvient en exactement 30 coups, il obtient 100 points. Sinon, s'il y parvient en moins de 60 coups il obtient 99 points. Le jeu prend fin quand le joueur a réussi, sinon il est stoppé au bout de 60 coups et il obtient un score de 0.

1	2	3
4	5	6
7	8	

Tictactoe : Il s'agit du jeu également connu en France sous le nom de *Morpion*¹³ qui consiste à aligner trois croix ou trois cercles sur une grille de 3x3 cases (voir image ci-dessous). Par convention X est considéré comme étant le joueur 1 et commence. Les joueurs jouent chacun à leur tour et déposent leur symbole dans une des cases encore vide. Le premier joueur qui aligne trois symboles dans une ligne, une colonne ou une diagonale remporte 100 points et l'autre 0. En cas de match nul chacun remporte 50 points. Si les deux joueurs jouent parfaitement, l'issue du match est toujours nulle¹⁴ ce qui limite l'intérêt du jeu. Cependant, la petite taille de l'arbre du jeu le rend pratique comme terrain de tests. La description GDL de ce jeu utilise le (*or ...*) pourtant interdit dans les versions récentes des spécifications GDL.

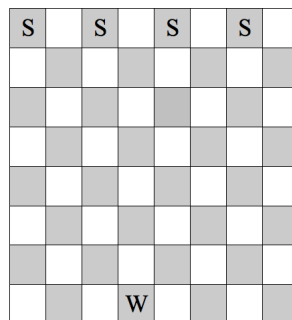
×		○
○	×	
×	○	×

SheepAndWolf : Il s'agit d'un jeu asymétrique. Sur un damier de 8x8 cases, un joueur dirige un loup (W) et l'autre joueur 4 moutons (S). Au début du jeu les

13. Le nom *Morpion* est moins précis puisqu'il désigne plusieurs jeux différents qui n'ont rien à voir entre eux.

14. Ce fait constitue un élément majeur de l'intrigue du film *WarGames*.

pions de chaque joueur sont aux deux extrémités du damier (voir image ci-dessous). Les moutons peuvent se déplacer d'une case à la fois, en diagonale et vers l'avant uniquement. Le loup peut se déplacer en diagonale dans toutes les directions. Pour un joueur comme pour l'autre, la case de destination doit être vide pour pouvoir s'y déplacer. Le but des moutons est d'encercler le loup pour l'empêcher de bouger. Le jeu prend fin quand le loup ne peut plus bouger : les moutons gagnent alors 100 points et le loup 0. Si le loup dépasse les moutons et qu'ils n'ont donc plus aucun espoir de l'encercler, le jeu prend fin, le loup remporte 100 points et les moutons 0.



Bibliographie

- AARTS, E. H. L., KORST, J. et VAN LAARHOVEN, P. J. M. (1997). *Local Search in Combinatorial Optimization*, chapitre 4, Simulated Annealing, pages 91–120. Princeton University Press.
- ABRAMSON, D. A., KRISHNAMOORTHY, M. et DANG, H. (1997). Simulated annealing cooling schedules for the school timetabling problem. *Asia-Pacific Journal of Operational Research*, 16:1–22.
- ALI, M., TÖRN, A. et VIITANEN, S. (1996). A direct search simulated annealing algorithm for optimization involving continuous variables. TUCS 97, Turku Center for Computer Science.
- BERTSIMAS, D. et TSITSIKLIS, J. (1993). Simulated annealing. *Statistical Science*, 8(1): 10–15.
- BJÖRNSSON, Y. et SCHIFFEL, S. (2013). Comparison of gdl reasoners. In *Proceedings of the IJCAI-13 Workshop on General Game Playing (GIGA'13)*, pages 55–62.
- BOESE, K. D. et KAHNG, A. B. (1994). Best-so-far vs. where-you-are : Implications for optimal finite-time annealing. *Systems and Control Letters*, 22:71–78.
- BOURKI, A., CHASLOT, G., COULM, M., DANJEAN, V., DOGHMEN, H., HOOCK, J.-B., HÉRAULT, T., RIMMEL, A., TEYTAUD, F., TEYTAUD, O., VAYSSIÈRE, P. et YUT, Z. (2010). Scalability and parallelization of monte-carlo tree search. In *Computers and Games*, pages 48–58.
- BROWNE, C. B., POWLEY, E., WHITEHOUSE, D., LUCAS, S. M., COWLING, P. I., ROHLFSHAGEN, P., TAVENER, S., PEREZ, D., SAMOTHRAKIS, S. et COLTON, S. (2012). [A survey of monte carlo tree search methods](#). *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.
- CARDOSO, M. F., SALCEDO, R. L. et de AZEVEDO, S. F. (1994). Nonequilibrium simulated annealing : A faster approach to combinatorial minimization. *Ind. Eng. Chem. Res.*, 33(8):1908—1918.
- CAZENAVE, T. (2009). [Nested Monte-Carlo Search](#). In BOUTILIER, C., éditeur : *IJCAI*, pages 456–461.
- CAZENAVE, T. (2014). communication personnelle du 07/07/2014.

- CAZENAVE, T. et JOUANDEAU, N. (2007). On the parallelization of uct. In *Proceedings of the Computer Games Workshop*, pages 93–101.
- CAZENAVE, T. et JOUANDEAU, N. (2008). A parallel monte-carlo tree search algorithm. In van den HERIK, H. J., XU, X., MA, Z. et WINANDS, M. H. M., éditeurs : *Computers and Games*, volume 5131 de *Lecture Notes in Computer Science*, pages 72–80. Springer.
- CAZENAVE, T. et JOUANDEAU, N. (2009). Parallel nested monte-carlo search. In *IPDPS*, pages 1–6.
- CHASLOT, G. (2010). *Monte-Carlo Tree Search*. Thèse de doctorat, Universiteit Maastricht.
- CHASLOT, G., WINANDS, M. H. M. et van den HERIK, H. J. (2008). Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71.
- CHEN, S. et LUK, B. (1999). Adaptive simulated annealing for optimization in signal processing applications. *Signal Processing*, 79:117–128.
- CLUNE, J. (2007). [Heuristic evaluation functions for general game playing](#). In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2, AAAI'07*, pages 1134–1139. AAAI Press.
- CLUNE, III, J. E. (2008). [Heuristic evaluation functions for general game playing](#). Thèse de doctorat, University of California, Los Angeles, Los Angeles, CA, USA. AAI3346923.
- CORANA, A., MARCHESI, M., MARTINI, C. et RIDELLA, S. (1987). Minimizing multimodal functions of continuous variables with the "simulated annealing"; algorithm. *ACM Trans. Math. Softw.*, 13(3):262–280.
- COX, E., SCHKUFZA, E., MADSEN, R. et GENESERETH, M. (2009). Factoring general games using propositional automata. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 13–20.
- DEKKERS, A. et AARTS, E. (1991). Global optimization and simulated annealing. *Mathematical Programming*, 50(1-3):367–393.
- ENZENBERGER, M. et MÜLLER, M. (2010). A lock-free multithreaded monte-carlo tree search algorithm. In *Proceedings of the 12th International Conference on Advances in Computer Games, ACG'09*, pages 14–20, Berlin, Heidelberg. Springer-Verlag.
- FINNSSON, H. (2007). Cadia-player : A general game playing agent. Mémoire de D.E.A., Reykjavík University, School of Computer Science, Reykjavík.
- FINNSSON, H. (2012a). [Simulation-Based General Game Playing](#). Doctor of philosophy, School of Computer Science, Reykjavík University.
- FINNSSON, H. (2012b). *Simulation-Based General Game Playing*. Thèse de doctorat, Reykjavík University.

- FINNSSON, H. et BJÖRNSSON, Y. (2008). [Simulation-based approach to general game playing](#). In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, AAAI'08, pages 259–264. AAAI Press.
- FINNSSON, H. et BJÖRNSSON, Y. (2009). [Simulation Control in General Game Playing Agents](#). In *IJCAI'09 Workshop on General Intelligence in Game Playing Agents*.
- FINNSSON, H. et BJÖRNSSON, Y. (2011). [CadiaPlayer: Search Control Techniques](#). *KI Journal*, 25(1):9–16.
- GAILHARD, S. (2014). communication personnelle du 10/09/2014. société Kalray.
- GELFAND, S. B. (1987). *Analysis of Simulated Annealing Type Algorithms*. Thèse de doctorat, Massachusetts Institute of Technology, Cambridge, Laboratory for Information and Decision Systems.
- GELFAND, S. B. et MITTER, S. K. (1993). Metropolis-type annealing algorithm for global optimization in $\mathbb{R}$. *Siam J. Control and Optimization*, 31(1):111–131.
- GELLY, S., HOOCK, J.-B., RIMMEL, A., TEYTAUD, O. et KALEMKARIAN, Y. (2008). The parallelization of monte-carlo planning - parallelization of mc-planning. In FILIPE, J., ANDRADE-CETTO, J. et FERRIER, J.-L., éditeurs : *ICINCO-ICSO*, pages 244–249. INSTICC Press.
- GELLY, S., WANG, Y., MUNOS, R. et TEYTAUD, O. (2006). Modification of uct with patterns in monte-carlo go. Rapport technique 6062, Inria.
- GEMAN, S. et GEMAN, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741.
- GENESERETH, M. et THIELSCHER, M. (2014). Stanford Logic Group.
- GENESERETH, M. R., LOVE, N. et PELL, B. (2005). [General Game Playing: Overview of the AAAI Competition](#). *AI Magazine*, 26(2):62–72.
- GHÉDIRA, K. (2007). *Optimisation combinatoire par métaheuristiques : origines, concepts et éléments de base, algorithmes canoniques et étendus*. Sciences et technologies. Éditions Technip.
- GRAF, T., LORENZ, U., PLATZNER, M. et SCHAEFERS, L. (2011). Parallel monte-carlo tree search for hpc systems. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 365–376, Berlin, Heidelberg. Springer-Verlag.
- HAJEK, B. (1985). A Tutorial Survey of Theory and Applications of Simulated Annealing. In *Proceedings of the 24th IEEE Conference on Decision and Control*, pages 755–760, Ft. Lauderdale.

- HAJEK, B. (1988). Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2):311–329.
- HENDERSON, D., JACOBSON, S. H. et JOHNSON, A. W. (2010). *Handbook of Metaheuristics*, chapitre 10, Theory and Practice of simulated Annealing, pages 287–319. Springer Publishing Company, Incorporated, 2nd édition.
- HU, T. C., KAHNG, A. B. et ALBERT TSAO, C. (1995). Old bachelor acceptance : A new class of non-monotone threshold accepting methods. *ORSA Journal on Computing*, 7:41742–5.
- INGBER, L. (1989). Very fast simulated re-annealing. Lester Ingber Papers 89vf, Lester Ingber.
- INGBER, L. (1993). Simulated annealing : Practice versus theory. *Mathl. Comput. Modelling*, 18:29–57.
- INGBER, L. (1996). Adaptive simulated annealing (asa) : Lessons learned. *Control and Cybernetics*, 25:33–54.
- INGBER, L. (1997). Adaptive simulated annealing (asa).
- JOHNSON, D. S., ARAGON, C. R., MCGEOCH, L. A. et SCHEVON, C. (1989). Optimization by simulated annealing : An experimental evaluation ; part i, graph partitioning. *Operations research*, 37(6):865–892.
- JOHNSON, D. S., ARAGON, C. R., MCGEOCH, L. A. et SCHEVON, C. (1991). Optimization by simulated annealing : an experimental evaluation ; part ii, graph coloring and number partitioning. *Operations research*, 39(3):378–406.
- JOUANDEAU, N. (2013). Intel versus mppa. Rapport technique, LIASD Université Paris8.
- KAISER, D. M. (2005). The structure of games. In *Proceedings of the 43rd annual Southeast regional conference - Volume 1*, ACM-SE 43, pages 61–62, New York, NY, USA. ACM.
- KAISER, D. M. (2007). [Automatic feature extraction for autonomous general game playing agents](#). In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems, AAMAS '07*, pages 93 :1–93 :7, New York, NY, USA. ACM.
- KALRAY (2010-2013c). *POSIX LEVEL - On the Kalray MPPA Developer with AB01*. Kalray SA.
- KALRAY (2013a). *Getting started reviewed 1.01b*. Kalray SA.
- KALRAY (2013b). *MPPA® ACCESSCORE 1.0.1 - POSIX Programming Reference Manual - KETD-325 W08*. Kalray SA. 142 pages.
- KALRAY (2013d). *Using the Kalray Makefiles - KETD 323 W10*. Kalray SA.

- KALRAY (2014). *MPPA® ACCESSCORE 1.2.1 - POSIX Programming Reference Manual - KETD-325 W08*. Kalray SA. 166 pages.
- KATO, H. et TAKEUCHI, I. (2010). Parallel monte-carlo tree search with simulation servers. In *Proceedings of the 2010 International Conference on Technologies and Applications of Artificial Intelligence, TAAI '10*, pages 491–498, Washington, DC, USA. IEEE Computer Society.
- KIRCI, M., STURTEVANT, N. et SCHAEFFER, J. (2011). [A GGP Feature Learning Algorithm](#). *KI - Künstliche Intelligenz*, 25(1):1–8.
- KIRKPATRICK, S., GELATT, C. D. et VECCHI, M. P. (1983). Optimization by simulated annealing. *Science*, 220:671–680.
- KISSMANN, P. et EDELKAMP, S. (2010). Instantiating general games using prolog or dependency graphs. In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence, KI'10*, pages 255–262, Berlin, Heidelberg. Springer-Verlag.
- KOCSIS, L. et SZEPESVÁRI, C. (2006). Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning, ECML'06*, pages 282–293, Berlin, Heidelberg. Springer-Verlag.
- KUHLMANN, G., DRESNER, K. et STONE, P. (2006). [Automatic Heuristic Construction in a Complete General Game Player](#). In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1457–62.
- LAM, J. et DELOSME, J.-M. (1988). An efficient simulated annealing schedule : Derivation. Technical Report 8816, Yale Electrical Engineering Department, New Haven, CT.
- LECCHINI-VISINTINI, A., LYGEROS, J. et MACIEJOWSKI, J. M. (2007). Simulated annealing : Rigorous finite-time guarantees for optimization on continuous domains. In PLATT, J. C., KOLLER, D., SINGER, Y. et ROWEIS, S. T., éditeurs : *NIPS*. Curran Associates, Inc.
- LOCATELLI, M. (2000). Simulated annealing algorithms for continuous global optimization : Convergence conditions. *Journal of Optimization Theory and Applications*, 104(1):121–133.
- MÉHAT, J. et CAZENAVE, T. (2011a). A parallel general game player. *KI*, 25(1):43–47.
- MÉHAT, J. et CAZENAVE, T. (2011b). Tree parallelization of aryl on a cluster. In *GIGA 2011, IJCAI 2011*, Barcelona.
- METROPOLIS, N., ROSENBLUTH, A. W., ROSENBLUTH, M. N., TELLER, A. H. et TELLER, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092.

- MÉHAT, J. (2013). Rapport sur la compétition de general game playing de 2013 première partie : déroulement de la compétition. Rapport technique, LIASD, Université de Paris 8.
- MÉHAT, J. et CAZENAVE, T. (2008). [An Account of a Participation to the 2007 General Game Playing Competition](#).
- MÉHAT, J. et CAZENAVE, T. (2010). [Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing](#). *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):271–277.
- NAHAR, S., SAHNI, S. et SHRAGOWITZ, E. (1985). Experiments with simulated annealing. In *Proceedings of the 22Nd ACM/IEEE Design Automation Conference, DAC '85*, pages 748–752, Piscataway, NJ, USA. IEEE Press.
- PELL, B. D. (1993). [Strategy generation and evaluation for meta-game playing](#). Thèse de doctorat, Trinity College, University of Cambridge.
- PINCUS, M. (1970). Letter to the editor — a monte carlo method for the approximate solution of certain types of constrained optimization problems. *Operations Research*, 18(6):1225–1228.
- PITRAT, J. (1971). A general game playing program. In FINDLER et MELTZER, éditeurs : *Artificial Intelligence and Heuristic Programming*, pages 125–155. Edinburgh University Press.
- PITRAT, J. (1976). A program to learn to play chess. In *Pattern Recognition and Artificial Intelligence*, pages 399–419. Academic Press Ltd. London, UK.
- QUENAULT, M. (2010). *Une application de jeux génériques pour les jeux symboliques*. Thèse de doctorat, Université Paris Dauphine.
- RAJASEKARAN, S. (1990). On the convergence time of simulated annealing. Rapport technique, University of Pennsylvania Department of Computer and Information Science Technical.
- RUIJL, B., VERMASEREN, J., PLAAT, A. et van den HERIK, H. J. (2013). Combining simulated annealing and monte carlo tree search for expression simplification. *CoRR*, abs/1312.0841.
- SCHAEFERS, L., PLATZNER, M. et LORENZ, U. (2011). Uct-treesplit - parallel mcts on distributed memory. In *MCTS Workshop*, Freiburg, Germany.
- SCHIFFEL, S. et THIELSCHER, M. (2007a). [Fluxplayer: A successful general game player](#). In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 1191–1196. AAAI Press.
- SCHIFFEL, S. et THIELSCHER, M. (2007b). [Automatic Construction of a Heuristic Search Function for General Game Playing](#). Hyderabad, India.

- SCHREIBER, S. (2014). [GGP.org - Tilyard Gaming Server](http://tiltyard.ggp.org/), <http://tiltyard.ggp.org/>. CS227b class at Stanford University.
- SECHEN, C. et SANGIOVANNI-VICENTELLI, A. (1985). The timber wolf placement and routing package. *IEEE, Journal on Solid State Circuits*, SC-20:510–522.
- SIARRY, P., BERTHIAU, G., DURDIN, F. et HAUSSY, J. (1997). Enhanced simulated annealing for globally minimizing functions of many-continuous variables. *ACM Trans. Math. Softw.*, 23(2):209–228.
- SOEJIMA, Y., KISHIMOTO, A. et WATANABE, O. (2010). Evaluating root parallelization in go. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):278–287.
- STANFORD LOGIC GROUP (2013). [General Game Playing](http://games.stanford.edu/), <http://games.stanford.edu/>. Stanford university.
- SUMAN, B. et KUMAR, P. (2006). – a survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the Operational Research Society*, 57(10):1143–1160.
- SUTTON, R. S. (1988). [Learning to Predict by the Methods of Temporal Differences](#). *Mach. Learn.*, 3(1):9–44.
- TALBI, E.-G. (2009). *Metaheuristics : From Design to Implementation*.
- THIELSCHER, M. (2010). [A General Game Description Language for Incomplete Information Games](#). In *AAAI*.
- THIELSCHER, M. (2011). [The general game playing description language is universal](#). In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI’11, pages 1107–1112. AAAI Press.
- VAN LAARHOVEN, P. et AARTS, E. (1987). *Simulated Annealing : Theory and Applications*. Ellis Horwood series in mathematics and its applications. Springer.
- VITTAUT, J.-N. (2014). communication personnelle du 28/03/2014.
- VITTAUT, J.-N. et MEHAT, J. (2014). Fast instantiation of ggp game descriptions using prolog with tabling. Rapport technique, LIADS, Paris8, France.
- YOSHIZOE, K., KISHIMOTO, A., KANEKO, T., YOSHIMOTO, H. et ISHIKAWA, Y. (2011). Scalable distributed monte-carlo tree search. In *SOCS*.
- ÖZDAMAR, L. et DEMIRHAN, M. (2000). Experiments with new stochastic global optimization search techniques. *Computers & OR*, 27(9):841–865.
- ČERNÝ, V. (1985). Thermodynamical approach to the traveling salesman problem : An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51.