THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS 8
ED 401

Spécialité
Informatique

Sujet de la thèse :

LeJoueur : un programme de General Game Playing pour les jeux à information incomplète et/ou imparfaite

Présentée par
Jean-Noël VITTAUT

Pour obtenir le grade de
DOCTEUR de l'UNIVERSITÉ PARIS 8

Soutenue le 1er décembre 2017

devant le jury composé de :

Bruno Bouzy	(Maitre de conférences HDR)	Rapporteur
Tristan Cazenave	(Professeur)	Rapporteur
Ludovic Denoyer	(Professeur)	Examinateur
Maria Rifqi	(Professeur)	Examinatrice
Nicolas Jouandeau	(Maître de conférences HDR)	Directeur de thèse
Jean Méhat	(Maître de conférences)	Co-directeur de thèse

Table des matières

Remerciements

Mes premiers remerciements vont à Jean Méhat, mon directeur de thèse, qui m'a proposé de travailler avec lui sur sa thématique de recherche et qui m'a convaincu de «passer ma thèse». Son immense culture scientifique, technique et historique de l'informatique qu'il partageait à chaque occasion a été précieuse pour progresser dans mon activité d'enseignant et de chercheur. Je remercie Nicolas Jouandeau d'avoir accepté d'être aussi mon co-directeur et dont les discussions, les conseils et les encouragements me permettent de présenter ces travaux.

Je remercie Bruno Bouzy et Tristan Cazenave d'avoir accepté d'être les rapporteurs de cette thèse ainsi que Maria Rifqi et Ludovic Denoyer d'en être les examinateurs/trices.

Je remercie Aline Hufschmitt avec qui j'ai le plaisir de collaborer sur la même thématique.

J'adresse aussi ma gratitude aux enseignant·e·s-chercheur·e·s du LIASD, mais aussi aux secrétaires, ingénieur·e·s et techniciens de l'UFR MITSIC qui ont rendu possible ce travail, chacun·e·s à leur façon. Je remercie chaleureusement Vincent Boyer d'avoir relu mon manuscrit ainsi que toutes celles et ceux qui m'ont proposé de le faire : leur avis m'importe beaucoup même si je leur ai épargné ce travail fastidieux.

Je remercie mes ami·e·s qui se reconnaîtront pour leurs nombreuses invitations à se changer les idées. Je pense que le moment est venu de leur consacrer plus de temps.

Enfin, mes remerciements vont aussi à mes parents qui ont introduit un ordinateur déjà obsolète à la maison lorsque j'étais encore à l'école primaire et dont le principal intérêt était d'être programmé. J'imagine que cet événement et la liberté qu'ils m'ont toujours accordée y sont pour beaucoup dans mon parcours.

À Jean

Introduction

Motivation

Cette thèse est une contribution au domaine du *General Game Playing* (GGP), une problématique de l'Intelligence Artificielle qui s'intéresse au développement d'agents autonomes capables de jouer à une grande variété de jeux et que nous appelons les jeux généraux. Le GGP se distingue des recherches sur les algorithmes permettant de bien jouer à des jeux spécifiques et offre de ce fait la possibilité d'évaluer l'efficacité de méthodes développées en Intelligence Artificielle sans perturbation par ajout de connaissances spécifiques à un jeu fournies par des experts.

La recherche en Intelligence Artificielle sur les jeux est motivée par l'impossibilité pour les machines d'explorer en un temps raisonnable tous les coups possibles pouvant intervenir dans le déroulement d'une partie d'un jeu dont la combinatoire dépasse les possibilités de représentation complète en mémoire.

De nombreux travaux de recherche portent sur des jeux spécifiques, notamment les Échecs et le Go. Les intelligences artificielles pour ces jeux reposent sur des évaluations des positions par des heuristiques développées à partir de connaissances d'experts et de bases de données de parties recueillies dans des tournois où s'affrontent des professionnels ou des amateurs. Ces connaissances expertes ou la constitution de ces bases de données n'existent pas pour tous les jeux, ce qui motive le développement de méthodes qui n'en font pas usage.

Un aspect important de nos travaux porte sur l'utilisation d'une représentation implicite de l'arbre de jeu sous la forme de règles logiques, une représentation explicite étant trop volumineuse pour être stockée sur une machine. Dans ces conditions, l'ordre de parcours de cet arbre et la rapidité d'évaluation des règles logiques sont des aspects cruciaux qui influent sur le niveau de jeu des intelligences artificielles.

Contributions

Dans ce contexte, nous avons proposé une méthode efficace d'instanciation des règles du *Game Description Language* permettant la génération d'un circuit logique avec lequel nous pouvons effectuer une analyse statique du jeu et simuler rapidement des parties aléatoires [72, 73]. Une parallélisation de l'évaluation du circuit logique nous a permis d'accélérer significativement la recherche dans l'arbre de jeu. Dans des travaux avec Aline Hufschmitt, nous avons utilisé une analyse statique du circuit qui nous permet d'envisager la décomposition automatique d'un jeu en plusieurs sous-jeux afin de réduire la taille de l'espace de recherche [32]. Une méthode de recherche des éléments de l'ensemble d'information que nous proposons permet à lejoueur de jouer à des jeux à information incomplète et imparfaite. Nous avons de plus proposé des adaptations des méthodes de recherche Monte-Carlo dans les arbres aux contraintes du GGP ainsi qu'une méthode permettant d'utiliser une estimation RAVE (*Rapid Action Value Estimation*) en début de recherche lorsque peu d'estimations sont disponibles avant de basculer automatiquement sur une méthode de Monte-Carlo [48].

Organisation du mémoire

Dans le chapitre 1, nous présentons l'historique du *General Game Playing* jusqu'à la définition qu'en a donné l'équipe logique de l'Université de Stanford, notamment à travers la définition d'un langage de description des jeux généraux finis, déterministes et à information complète : le *Game Description Langage* (GDL).

Le chapitre 2 est consacré à l'interprétation du GDL par le langage logique PROLOG et sa transformation en circuit logique combinatoire dont l'analyse statique permet d'étudier les caractéristiques d'un jeu. La compilation du circuit en un *bytecode* permet la simulation rapide de parties aléatoires et offre la possibilité d'une parallélisation présentée au chapitre 5. L'analyse du circuit logique permet aussi d'envisager une décomposition des jeux.

Dans le chapitre 3, nous présentons les jeux généraux à information incomplète ou imparfaite et la manière dont nous les traitons dans le cas des descriptions en GDL ainsi qu'en GDL-II, une extension de ce langage aux jeux finis, déterministes et à information imparfaite.

Dans le chapitre 4, nous présentons le cadre des algorithmes Monte-Carlo dans les arbres qui ont permis une amélioration significative du niveau des joueurs au jeu de Go et la manière dont nous les avons adaptés au domaine des jeux généraux.

Enfin, le chapitre 5 présente les différentes formes de parallélisation que nous avons développées dans les différents modules de notre programme LEJOUEUR décrits dans les chapitres précé-

dents. La manière dont nous avons conçu la machine de Warren de LeJoueur pour interpréter le GDL permet d'envisager simplement sa parallélisation et permettre une instanciation plus rapide des règles. Une parallélisation sur un seul thread de la simulation de parties aléatoires permet une amélioration du niveau de jeu à ressources de calcul identiques. Enfin, la parallélisation des algorithmes de recherche Monte-Carlo dans les arbres permet d'explorer davantage l'arbre de jeu.

Chapitre 1

GGP / GDL

Dans ce chapitre, nous présentons le domaine du General Game Playing. Nous définissons tout d'abord le modèle de représentation des jeux que nous utilisons ensuite pour définir des classes de jeux. Par la suite nous présentons l'historique du domaine depuis le système de Pitrat jusqu'à la définition proposée par Genesereth et le groupe logique de Stanford et qui constitue l'objet principal de notre étude. Nous abordons ensuite l'extension de ce domaine aux jeux vidéos.

1.1 Les jeux

Nous présentons ici le modèle de représentation des jeux que nous utilisons et la façon dont il fait apparaître la classification classique des jeux.

1.1.1 Un modèle de représentation des jeux

Les jeux que nous étudions dans ce mémoire sont modélisables sous la forme d'un système de transitions d'états. Nous apporterons par la suite des restrictions à ce modèle afin de représenter les jeux décrits dans les langages que nous avons utilisés en General Game Playing.

Définition 1. *Un* **système de transitions d'états** *est un couple* $(E, \rightarrow)$ *où* E *est un ensemble d'états et* $\rightarrow$ *un ensemble de transitions, sous ensemble de* $E \times E$*. L'existence d'une transition entre un état a et un état b, c'est-à-dire* $(a, b) \in \rightarrow$ *sera noté pas la suite* $a \rightarrow b$ *pour plus de clarté.*

Dans cette définition, il n'y a aucune hypothèse sur E et $\rightarrow$. Ces ensembles peuvent être finis ou infinis, dénombrables ou non.

Les éléments essentiels d'un jeu que nous définissons ci-après sont les **joueurs**, les **coups**, les **paiements** et l'**information** [58]. Le joueur va chercher à maximiser ses paiements à l'aide de

stratégies qui déterminent ses coups en fonction des informations qu'il a en sa possession à un moment donné. Ces éléments sont communément appelés les *règles du jeu*.

Définition 2. *Un* **joueur** *est un individu qui choisit ses actions afin de maximiser ses paiements*[1].

Définition 3. *La* **nature** *est un pseudo-joueur qui choisit ses actions aléatoirement en fonction de l'état où dans lequel elle se trouve et suivant une distribution de probabilités donnée.*

Définition 4. *Un* **coup** *est le choix qu'un joueur peut faire.*

Définition 5. *Un* **jeu** *est l'ensemble des éléments suivants :*

1. $S = (E, \rightarrow)$ *: un système de transitions d'états.*
2. J *: un ensemble fini de joueurs.*
3. C *: un ensemble de coups.*
4. P *: un ensemble de paiements.*
5. r *: un état de* E *qui détermine l'état initial du jeu.*
6. *Une application* τ *de* E *dans* J *qui étiquette chaque état par le joueur qui a la main.*
7. *Pour chaque état* $e \in E$*, une application injective de* λ_e *de* T_e *dans* C *où* T_e *est l'ensemble des transitions ayant pour origine* e*.* λ_e *définit les coups légaux associés à chaque transition.*
8. *Une application* π *de* T *dans* P^J *où* T *est l'ensemble des états terminaux (sans successeur), qui détermine les paiements de chaque joueur.*

Dans cette définition, nous faisons le choix que pour chaque état, seul un joueur peut jouer afin de définir la notion d'ensemble d'information que nous verrons plus loin de la manière la plus simple. Cette modélisation se rapproche de la représentation extensive en théorie des jeux. C'est pour cette raison aussi de nous choisissons que l'ensemble des joueurs soit fini. S'il était infini, une difficulté apparaîtrait dans le cas où les joueurs jouent simultanément car le système de transitions devrait contenir une chaîne infinie depuis l'état où ils doivent jouer et l'état où ils ont tous joué.

Définition 6. *Un* **match** *est un chemin dans le système de transitions de l'état initial à un état terminal.*

Ce modèle permet de modéliser une grande variété de jeux car il ne fait aucune hypothèse sur les différents ensembles introduits à part l'ensemble des joueurs qui doit être fini. La modélisation

[1] Nous ne distinguerons pas la notion de paiement de celle d'utilité qui modélise la manière dont un joueur particulier perçoit son paiement. Le paiement est une qualification universelle du but atteint alors que l'utilité est attachée à la perception qu'en a le joueur. Cette notion est utile dans le cas où la psychologie du joueur intervient.

sous forme de système de transitions impose tout de même qu'un match soit composé d'une suite d'états. Autrement dit, il n'y a pas de possibilité d'évolution continue de l'état du jeu. Cela nous permet de définir un temps discret qui détermine à quel rythme le système peut changer d'état.

Nous allons à présent définir différentes classes de jeux selon les hypothèses que nous faisons sur le système de transitions, les joueurs, les paiements ou la perception qu'ils ont de l'état du système.

1.1.2 La classification des jeux

Jeux infiniment longs Un jeu est infiniment long lorsque qu'il est possible de construire un chemin infini dans le système de transitions d'états. Même si l'ensemble des états est fini, un jeu peut être infiniment long à partir du moment où le graphe du système de transitions comporte des circuits. Nous pouvons aussi construire un jeu infiniment long en itérant un jeu qui ne l'est pas et en fixant pour but aux joueurs de gagner un maximum de parties intermédiaires.

Jeux discrets La plupart des jeux étudiés en théorie des jeux sont discrets voire finis. Un jeu est discret lorsque le nombre d'états, de transitions et de paiements possibles sont discrets.

Jeux à somme nulle Un jeu est à somme nulle lorsque pour chaque état terminal, la somme des paiements des joueurs est nul.

Jeux à coups simultanés Dans le modèle que nous avons adopté, seul un joueur a la main dans un état donné du système. Pour modéliser un jeu à coups simultanés avec n joueurs, il suffit d'imposer qu'un tour de jeu se fasse par une séquence de n coups effectués par des joueurs deux à deux distincts et que l'information du coup joué par un joueur soit totalement cachée aux autres au moins pendant la durée de la séquence. En particulier, l'ensemble des coups légaux d'un joueur durant cette séquence sera toujours le même quels que soient les choix effectués par les autres joueurs auparavant, ceci afin de ne pas dévoiler d'information. Cette notion de coups alternés ou coups simultanés correspond à un choix de modélisation. C'est plutôt la notion d'information qui importe comme nous allons voir dans les paragraphes suivants.

Jeux symétriques Un jeu symétrique est un jeu dont les récompenses en jouant une stratégie particulière dépendent seulement des autres stratégies employées et pas des joueurs qui la jouent. Par exemple, le jeu de la poule mouillée ou le dilemme du prisonnier sont des jeux symétriques. Les échecs ou le go ne sont pas symétriques car un des deux joueurs joue en premier.

Jeux à information parfaite ou imparfaite Un jeu à coups alternés est à information parfaite si tous les joueurs connaissent l'ordre et les coups effectués précédemment par tous les autres joueurs. Par conséquent, en parcourant le système de transitions à partir de l'état initial, les joueurs connaissent parfaitement dans quel état se trouve la partie. Un jeu est à information imparfaite s'il n'est pas à information parfaite.

Nous pouvons aussi aborder l'imperfection de l'information grâce à la notion d'ensemble d'information.

Définition 7. L'ensemble d'information *d'un joueur est l'ensemble des états du jeu dans lequel il est possible qu'il se trouve sans qu'il puisse les distinguer par observation directe.*

Le fait que le joueur sache parfaitement dans quel état se trouve le jeu peut alors s'énoncer simplement :

Définition 8. *Un jeu est à* **information parfaite** *pour un joueur si pour chaque état du jeu son ensemble d'information est réduit à un singleton.*

Les jeux où les joueurs peuvent effectuer des coups simultanés avec plus d'un coup possible pour chacun sont donc à information imparfaite.

Jeux à information complète ou incomplète Un jeu est à information incomplète lorsque chaque joueur ne connaît pas les stratégies des autres joueurs ou les gains obtenus par chacun à la fin du jeu. Tout se passe comme si il n'y avait pas un mais plusieurs jeux et que le joueur ne savait pas dans lequel il joue. Par conséquent, un jeu à information incomplète est aussi à information imparfaite.

Cette idée a permis à Harsanyi de montrer que tout jeu à information incomplète peut être transformé en jeu à information complète mais imparfaite [30]. Pour cela, le jeu initial dont les paiements ne sont pas connus est décomposé en plusieurs sous-jeux dont les paiements sont connus. Il attribue un type à chaque sous-jeu et introduit un coup initial de la nature qui décide du sous-jeu qui sera employé ce qui nous conduit à cette définition alternative :

Définition 9. *Un jeu est à* **information complète** *si la nature ne joue pas en premier ou si tous les joueurs ont observé le coup initial de celle-ci.*

Jeux déterministes ou bayésiens Les jeux bayésiens sont ceux pour lesquels les caractéristiques des adversaires sont modélisées sous la forme de distributions de probabilités qui sont mises à jour en fonction de leurs actions durant une partie selon la règle de Bayes. De ce fait, il existe une incertitude sur les gains des adversaires comme pour les jeux à information incomplète.

Jeu en temps réel Dans notre modèle, il n'y a pas d'évolution continue de l'état du système du fait du caractère discret du système de transitions. Nous pouvons tout de même modéliser les jeux en temps réel en supposant que le rythme avec lequel le système peut changer état est très rapide. Cette modélisation suppose l'existence d'une horloge discrète commune à tous les joueurs.

1.2 Les langages de description de jeux

Une fois épuisées les définitions des jeux des dictionnaires et des encyclopédies et les travaux des philosophes sur l'essence du jeu (Johan Huizinga, 1938 - Homo Ludens - Essai sur la fonction sociale du jeu), un fait demeure : même en ignorant le jeu chez les animaux, le jeu recouvre un large spectre d'activités extrêmement diverses depuis le football professionnel jusqu'au jeu de l'oie ; pourtant leurs éléments possèdent des caractéristiques communes.

Ce fait est sans doute un élément important à l'origine des multiples tentatives de définition d'un système de *jeu général* qu'a connu le domaine de l'informatique : extraire les caractéristiques communes de plusieurs jeux et en construire un nouveau jeu général dans lequel les différences entre les jeux particuliers ne sont traitées que comme des paramètres ; construire un programme joueur pour ce jeu général, qui possédera une compétence pour jouer correctement à tous ces jeux particuliers.

Ce programme se heurte à une difficulté : les programmes de jeu particuliers sont dotés non seulement de la logique nécessaire pour appliquer les règles d'un jeu particulier mais également de connaissances sur les caractéristiques de plus haut niveau de ce jeu — souvent sous la forme de fonctions heuristiques d'évaluation d'une position et de modifications ad'hoc de l'exploration de l'espace des coups possibles — qui compliquent sérieusement la réalisation d'un joueur général qui possède un niveau de jeu équivalent à celui des joueurs particuliers. Pour cette raison, la plupart des systèmes de jeux généraux que nous présentons ici limitent en pratique le domaine auquel ils s'appliquent, souvent à celui du jeu d'échecs et de leurs variantes, de façon à permettre d'obtenir un joueur général compétent en utilisant des heuristiques et des fonctions de recherche qui s'appliquent toujours à ce domaine ou bien dont il est faisable de reconnaître s'ils s'y appliquent.

Une exception est celle du General Game Playing défini par le groupe de logique de l'université de Stanford, qui constitue notre principal objet d'étude : son domaine est vaste puisqu'il englobe tous les jeux finis à information complète. La difficulté pour obtenir un joueur compétent y est principalement de laisser le joueur déterminer les heuristiques et les algorithmes de recherche qui donneront de bons résultats dans un jeu particulier.

Nous présentons ici les plus importants langages ou systèmes destinés à décrire des ensembles de jeux. Ces systèmes sont le résultat d'un compromis entre la généralité des jeux qu'ils sont ca-

pables de décrire et l'utilisation d'éléments communs à une grande variété de jeux pour lesquels il est possible d'utiliser des heuristiques. Les systèmes décrits dans cette section ont en commun d'être une généralisation des échecs. À l'extrême, n'importe quel langage de programmation Turing complet est un langage de description de jeux.

1.2.1 Le programme de Jacques Pitrat

Les premiers travaux rapportés appartenant au domaine du General Game Playing sont dus à Pitrat qui a conçu un programme pouvant jouer à plusieurs jeux [55]. Le programme reçoit les règles d'un jeu particulier en entrée et doit pouvoir étudier ces règles. Pitrat mentionne toutefois que le programme n'est pas totalement général et cite trois limitations : il ne peut jouer que sur des grilles bidimensionnelles ; les règles ne peuvent pas décrire tous les jeux ; les heuristiques générales utilisées permettent de jouer mais avec des performances insuffisantes pour certains jeux.

Son point de départ est qu'un programme de General Game Playing doit connaître les règles du jeu qu'il traite, ces règles étant : un algorithme donnant l'état *gagnant* ; un algorithme qui énumère les coups légaux, un coup étant un ensemble de changements qui fait passer de l'état courant à l'état suivant [56]. Il cite deux approches pour réaliser un tel programme : écrire des sous-programmes implantant ces algorithmes et fonctionnant comme une boite noire ; définir un langage que le programme de GGP pourrait analyser afin d'y trouver des indications pour améliorer sa stratégie. C'est la seconde approche qu'il a choisi en définissant un langage permettant de décrire des jeux similaires aux échecs.

1.2.2 Le système Metagame

L'idée de Metagame est de susciter le développement de programmes qui peuvent prendre en entrée les règles de n'importe quel jeu appartenant à une classe bien définie et jouer à ces jeux contre des adversaires [49]. C'est un système qui comprend un langage de description de jeux, un joueur général et un générateur de jeux.

Il dispose d'un langage de définition qui impose que les développeurs de jeux ne se focalisent plus sur un ensemble de règles spécifiques mais soient obligés de représenter leurs connaissances dans le formalisme général d'une classe de jeux. Pell définit une grammaire pour la classe des *symetric chess-like games* qui permet de représenter une grande partie des jeux à deux joueurs, à information complète, à tablier rectangulaire [50]. Les définitions de jeux peuvent aussi être produites par un générateur qui possède des distributions de probabilités pour des paramètres comme la complexité d'une pièce, le type de mouvement, la taille du tablier et la localité.

Le langage de définition n'est pas directement exploité par le joueur qui passe par l'intermédiaire d'un langage de description. La syntaxe et la sémantique de ce langage est très similaire à PROLOG, avec en plus des constructions pour accéder à un *état courant implicite*, *le joueur qui a la main* et le *jeu courant* [51, page 100]. Ces constructions sont les suivantes :

`true(P)` : une propriété `P` qui est vraie dans l'état courant du jeu.

`add(P)` : ajoute une propriété `P` à l'état courant.

`del(P)` : supprime une propriété `P` de l'état courant.

`control(Player)` : est vrai si `Player` à la main dans l'état courant.

`game:Pred` : *Pred* est une propriété dépendant du jeu et qui est vraie dans le jeu courant pour le joueur qui a la main.

`transfers_control` : passe la main à l'adversaire.

Nous pouvons y voir les prémisses du langage de description de jeux qui sera défini par le groupe logique de Stanford, notamment dans l'utilisation d'un langage logique et la représentation des états par un prédicat **`true`**. Le fait que les prédicats soient explicitement ajoutés et retirés permet d'éviter le problème du cadre qui est une des difficultés des descriptions en GDL.

1.2.3 Le logiciel Zillions of Games

Zillions Rules Files (ZRF) est un format propriétaire de description de jeux pour *Zillions of Games*, un logiciel commercial pour modéliser et jouer à des jeux se déroulant principalement sur une grille avec des pièces [42]. Les utilisateurs de ce programme ont la possibilité de jouer aux 375 jeux et puzzles inclus et de créer leurs propres descriptions de jeux dans le langage ZRF.

Le programme dispose d'une interface homme-machine qui permet de visualiser le jeu. Le rendu du jeu est assuré par des images stockées dans des fichiers et qui sont en lien dans la description ZRF.

Zillions of Games inclut aussi une intelligence artificielle pour pouvoir jouer contre la machine. L'algorithme utilisé semble apparemment utiliser des algorithmes efficaces aux échecs mais d'un faible niveau pour des jeux comme le Go, Hex ou Twixt [57].

1.2.4 Le cadre applicatif Ludi

Ludi est un système pour jouer, mesurer et synthétiser des jeux modélisables dans son langage de description [9]. Les principaux composants de ce système sont : un langage de description de jeux qui définit une classe de jeux, un joueur général qui interprète le jeu et coordonne son

déroulement, un module stratégique qui décide des coups à jouer, un module critique qui mesure la qualité du jeu et un module de synthèse qui génère de nouveaux jeux.

1.2.5 Le General Game Playing de Stanford

Le General Game Playing, tel qu'il a été défini par le groupe logique de Stanford vers 2005 constitue le cadre principal de notre étude [40]. Nous le présentons en détail dans la section suivante.

1.3 General Game Playing

Nous présentons dans cette section le *General Game Playing* qui constitue le cadre principal dans lequel notre étude se déploie. Après une description générale, nous présentons le *Game Description Language* (GDL) qui sert à décrire les jeux puis le protocole utilisé pour la communication entre l'arbitre et les joueurs. Nous détaillons les serveurs sur lesquels des humains peuvent affronter des programmes ou les faire jouer entre eux et nous terminons par une description et un bref historique des compétitions internationales qui permettent d'évaluer les progrès considérables du niveau des programmes de jeux du domaine depuis une dizaine d'années.

1.3.1 Le domaine du *General Game Playing*

En dehors du principe général de chercher à extraire le substrat commun entre les différents jeux, implémenté dans les divers systèmes décrits dans la section précédente, le *General Game Playing* (GGP) sert principalement à identifier le sous-domaine défini par le groupe de logique de l'université de Stanford vers 2005 et structuré autour d'une compétition annuelle organisée initialement dans le cadre de la conférence AAAI et/ou IJCAI. Nous décrivons dans les sections suivantes les aspects les plus importants du GGP.

Le principal est sans doute le langage utilisé pour décrire les jeux *Game Description Language* (GDL) qui délimite la classe des jeux à laquelle le GGP s'applique et influence significativement la manière de réaliser des joueurs.

Nous décrivons également le protocole de communication entre l'arbitre et les joueurs qui présente quelques particularités inhabituelles en essayant d'expliquer les contraintes techniques qui ont conduit ses concepteurs à le spécifier de cette manière.

Nous esquissons un historique de la compétition annuelle organisée principalement par le groupe de logique de l'université de Stanford à l'origine du GGP et tout particulièrement par

Michael Genesereth, compétition dont les modalités ont considérablement varié jusqu'à converger vers une forme stable.

Un autre aspect important du GGP est l'existence de serveurs permanents sur lesquels les programmes peuvent jouer à la demande ou dans le cadre d'un tournoi permanent. Ces serveurs ont permis l'émergence d'une communauté de personnes et d'équipes de recherches intéressées par le GGP à travers des contacts permanents, plus approfondis que ce qui peut se faire lors de la compétition annuelle.

Depuis 2009, la communauté du GGP se rencontre également lors d'un atelier (*workshop*) de la conférence IJCAI ; les communications qui y sont faites permettent des échanges sur les progrès des différentes équipes sous la forme d'articles scientifiques qui complètent utilement les discussions informelles lors des compétitions et sur les serveurs.

Depuis 2013, Michael Genesereth et le groupe de logique de Stanford organisent un cours en ligne ouvert massif (*Massive Open Online Course*, MOOC) sur le sujet. Cela a attiré un grand nombre d'inscriptions au cours et a débouché sur un accroissement significatif de la participation aux compétitions qui constitue une mesure de l'existence et de la taille de la communauté autour du GGP ; celle-ci, initialement composée de chercheurs spécialisés dans le domaine, s'est élargie à des amateurs enthousiastes — dont certains ont des capacités techniques impressionnantes — et a permis la révélation de certaines méthodes employées par des joueurs qui ont conduit à une amélioration générale du niveau des programmes.

1.3.2 Le Game Description Language (GDL)

Le GDL est un langage logique déclaratif dont la sémantique est très proche celle de Datalog. Une description de jeu se compose d'axiomes et de théorèmes. Les axiomes décrivent des aspects du jeu qui sont toujours vrais quelle que soit la partie jouée. Les théorèmes permettent de déterminer les faits qui sont vrais en fonction de l'état courant du jeu, notamment si la position est terminale, les coups légaux, les récompenses et l'état suivant en fonction des coups choisis par les joueurs.

1.3.2.1 Les origines du GDL

Le GDL est défini par un document originel d'une part formalisé ensuite dans un document de spécifications mais également dans la pratique des personnes qui décrivent des jeux et de celles qui réalisent des joueurs.

La première définition qui date de 2005 a pour auteurs Genesereth et al. [29]. Elle a ensuite été formalisée dans un document de spécification en 2008 [40]. Le principal inconvénient est que cette

définition n'a pas vraiment connu d'évolution au cours des années alors que dans la pratique, la manière de décrire des jeux a évolué, sous l'influence des auteurs de descriptions et du règlement de la compétition annuelle principale qui a été aménagé pour répondre aux besoins des joueurs dont les organisateurs connaissaient les détails.

Du fait de ces déficiences et aussi sans doute du caractère un peu étonnant pour certains d'un langage purement déclaratif, sans arithmétique, avec une syntaxe de langage fonctionnel, les propositions de modification ou de remplacement du langage GDL sont nombreuses. L'importance de la compétition annuelle de AAAI organisée par les auteurs de la première spécification dans un premier temps, et maintenant responsables du MOOC sur le General Game Playing garantit dans la pratique que ces modifications seront le plus souvent ignorées ; on peut considérer ceci comme un coté positif qui garantit la stabilité du langage.

Les paragraphes qui suivent abordent en détail certains des points dans lesquels se manifestent ces faiblesses.

La monotonie est une caractéristique du langage décrite dans la spécification : la description d'un jeu doit permettre le calcul du score de chaque joueur dans toutes les positions valides (avec le prédicat **`goal`** décrit plus loin) ; pour qu'un jeu soit monotone, le score d'un joueur dans une position ne peut pas être inférieur au score d'une des positions précédentes. Un jeu bien formé doit être monotone. Il s'agit évidemment de faciliter l'exploration de l'espace des positions d'un jeu en garantissant qu'une recherche fondée sur la descente de gradient ne restera jamais bloquée sur un minimum local.

En pratique, les descriptions de jeux actuelles, vues sur les serveurs aussi bien qu'utilisées dans les différentes compétitions ne respectent pas cette contrainte ; il ne faut utiliser le score des joueurs que dans les positions terminales[2]. Pourtant, la monotonie n'a jamais été retirée des spécifications du GDL.

Jeux infinis Les auteurs du GDL ont souhaité limiter différents aspects des jeux qu'il est possible de décrire. Dans le document de spécification, ils se proposent de décrire uniquement les jeux représentables par un automate fini dont chaque état représente une position du jeu et chaque

[2]Parmi les descriptions de jeux qui ne respectent pas la monotonie, deux cas peuvent se rencontrer : pour celles qui ne fournissent pas de score dans les positions intermédiaires, il est facile d'ajouter un théorème pour que le score de tous les joueurs soit égal à 0 dans toutes les positions non-terminales ; pour les descriptions qui fournissent des scores intermédiaires qui ne respectent pas la monotonie, une solution radicale est de les éliminer en ajoutant la prémisse **`terminal`** dans tous les théorèmes calculant les scores, ce qui nous ramène au premier cas. Évidemment, forcer les scores à zéro dans toutes les positions intermédiaires ne donne aux joueurs aucune information quant au score minimal garanti qu'ils pourraient obtenir en cours de partie. Le problème de décision relatif à la monotonie des descriptions GDL est certainement exponentiel en fonction de la taille de la description.

transition un coup. Pour y parvenir, ils ont tenté d'imposer que certains aspects du langage soient finis : les descriptions doivent garantir que toute partie se terminera en un nombre fini de coups ; elles doivent aussi garantir que tous les faits qui peuvent être produits par la description sont aussi en nombre fini.

La limite imposée au nombre de faits interdit dans la pratique l'utilisation de listes générales dans une description de jeux, si rien ne permet de garantir que cette liste sera de longueur finie. Il n'est pas possible de définir l'arithmétique en se fondant sur le 0 et la relation *successeur* parce que nous aurions alors accès à tous les nombres. Les descriptions de jeux définissent donc une arithmétique ad'hoc, beaucoup moins générales et concises ou bien ne sont pas du GDL légal.

Pendant des années, la description des jeux dans lesquels des groupes de pièces de tailles variables forment une unité, comme les chaînes de pions à Hex ou les groupes de pierres au Go, a posé problème : soit la description utilisait des listes pour représenter ces groupes de pièces et était alors du GDL illégal[3] ou bien la description garantissait la finitude des listes à l'aide de constructions artificielles et lourdes pour l'auteur. La transformation automatique de description pour ajouter explicitement une limite à la longueur des listes a été évoquée par les organisateurs de la compétition de l'AAAI mais n'a à ma connaissance jamais été réalisée. [M. Genesereth, liste de diffusion GGP, 2008] [4]

[3] Il y a longtemps eu sur le serveur de Stanford une description d'Atari Go fondée sur ces listes de tailles variables, non délimitées, qui commençait par le commentaire *Illegal GDL*, sans explication supplémentaire. Soulignons que les listes effectivement construites en cours de partie étaient nécessairement de tailles limitées puisqu'elles se composaient de pierres dont le nombre ne pouvaient excéder le nombre fini de positions, mais la description de ces listes en revanche ne limitait pas leur longueur.

[4] Ce problème a été résolu par Peter Pham pour le jeu de Hex : au lieu de construire explicitement les listes de pions qui constituent une chaîne, sa description contient un tableau de marqueurs équivalent à celui du damier ; la pose d'un nouveau pion conduit à la création d'un nouveau groupe dont le nom est déterminé par le compteur de tours ; les positions adjacentes contenant des pions de même couleur sont également marquées comme élément du même groupe. La chose se décrit simplement en GDL par les deux prédicats :

```
(<= (next (connected ?k ?m ?n))
  (does ?role (place ?m ?n))
  (true (step ?k)))

(<= (next (connected ?k ?m ?n))
  (transfer ?e)
  (true (connected ?e ?m ?n))
  (true (step ?k)))
```

```
(<= (next (connected ?k ?m ?n))
  (true (connected ?k ?m ?n))
  (not (transfer ?k)))

(<= (transfer ?k)
  (does ?role (place ?x ?y))
  (true (owner ?k ?role))
  (true (connected ?k ?m ?n))
  (adjacent ?x ?y ?m ?n))
```

Cette méthode a permis de décrire des jeux comme No Go ou Atari Go d'une façon élégante sans recourir à du GDL illégal. Elle s'apparente à la structure de données d'ensembles disjoints utilisée dans l'algorithme de Kruskal. Les connexions entre les pions sont calculés au fur et à mesure des frames.

1.3.2.2 Les syntaxes du GDL

Les descriptions de jeux en GDL peuvent s'exprimer dans deux syntaxes très différentes utilisées toutes les deux dans des contextes différents : KIF et Datalog.

La notation KIF Cette notation s'inspire de celle employée pour le *Knowledge Interchange Format* (KIF). Le KIF est un standard — peu utilisé à ma connaissance — à la définition duquel a participé Michael Genesereth qui est un des chercheurs pionniers du GGP[5]. KIF a une syntaxe qui évoque celle du LISP : toutes les expressions sans exception sont représentées soit sous la forme d'un mot (un *atome* dans la terminologie LISP) ou d'une liste composée d'une parenthèse ouvrante, d'expressions séparées par des espaces et d'une parenthèse fermante. On les appelle des *S-expressions*, une dénomination employée par John McCarthy dans son article original de description du langage LISP [43][6]. Cet article est antérieur à son arrivée à l'université de Stanford, un des endroits où le langage LISP a été largement utilisé comme outil en intelligence artificielle et en logique.

La simplicité est le principal avantage de cette notation. Les seuls éléments lexicaux sont les parenthèses et les espaces auxquels s'ajoutent les mots formés par toutes les suites d'autres caractères autorisés[7]. Elle présente en revanche l'inconvénient d'être passablement déroutante pour les lecteurs peu familiers avec le langage LISP.

Dans cette notation, les variables sont caractérisées par leur premier caractère qui est un point d'interrogation.

La notation KIF est la seule qui soit employée sur les serveurs de jeux et est parfois utilisée dans les articles scientifiques.

La notation Datalog Une autre notation est couramment employée dans les articles scientifiques. Elle s'inspire de celle de Datalog, un autre langage de représentation des connaissances dont la syntaxe est largement inspirée de celle du langage de programmation PROLOG.

[5]Les concepteurs de KIF s'étaient fixés pour but de fournir un moyen d'*échanger* de l'information entre systèmes plutôt que de la représenter comme le font les systèmes plus communs. La distinction entre les deux n'apparaît pas clairement à la lecture du document de description mais le succès assez limité du standard KIF me laisse croire qu'il ne s'agit pas d'un point crucial. Le standard XML avec ses lourdeurs et ses limitations évidentes occupe une telle place hégémonique, dans l'échange de données pour lequel il a été conçu et dans d'autres domaines auxquels il est encore moins bien adapté, que les autres moyens d'échanger de la connaissance sont condamnés à un rôle confidentiel.

[6]Dans son article de description du langage LISP, John McCarthy décrit une autre notation des expressions qu'il appelle des *M-expressions* ; elles ont été peu employées par les programmeurs LISP [43].

[7]Nous ignorons ici les commentaires qui sont introduits par le caractère ; et terminés par une fin de ligne.

(a)

```
(<= (blocks_rook_threat ?x ?y ?tx ?y ?kx ?y)
  (rook_move ?tx ?y ?x ?y)
  (rook_move ?x ?y ?kx ?y))
```

(b)

```
blocks_rook_threat(X, Y, TX, Y, KX, Y) :-
    rook_move(TX, Y, X, Y),
    rook_move(X, Y, KX, Y).
```

FIGURE 1.1 – Un extrait de la règle du jeu d'échecs notée à la KIF (a) et à la Datalog (b).

Dans cette notation, les éléments lexicaux sont plus nombreux : le (`:-`), le point (`.`), la virgule (`,`) qui joue deux rôles distincts en fonction du contexte où elle apparaît, les parenthèses ; les variables sont indiquées par des mots qui commencent par une lettre majuscule ou par un blanc souligné. Certains traducteurs depuis la notation KIF utilisent également le point-virgule pour la disjonction et les apostrophes afin d'introduire des espaces dans les symboles[8].

Cette notation est familière pour les programmeurs PROLOG. Il n'est pas certain qu'ils soient beaucoup plus nombreux que les programmeurs LISP mais le PROLOG partage au moins certains éléments syntaxiques rencontrés dans presque tous les langages de programmation courants et en mathématiques comme la représentation des appels de fonctions : `foo(a, b, c)`. La figure 1.1 présente la même règle du jeu exprimée dans les deux variantes.

Dans ce contexte, un *atome* désigne les axiomes, les prémisses et la conclusion d'un théorème : tout ce à quoi PROLOG attache une valeur de vérité. Ce mot possède donc un sens très différent de celui qu'il porte en LISP, ce qui complique parfois les communications.

La table 1.1 donne la liste des mots clés utilisés en GDL. Elle contient l'ensemble de prédicats de GDL que nous décrivons dans les paragraphes suivants et qui sont spécialisés dans la description des jeux : les joueurs, les états, les transitions et les coups, la fin de partie.

[8] Dans leur ouvrage sur le General Game Playing, Genesereth et Thielscher remplacent la virgule (`,`) par un (`&`) et une ligne blanche à la place du point (`.`) [28].

Prédicats logiques	
<=	déclaration d'une clause
or	disjonction
not	négation
distinct	compare deux termes instanciés
Prédicats statiques	
role	définit les noms des joueurs
init	définit l'état initial du jeu
input	définit l'ensemble des coups possibles du jeu
base	définit l'ensemble des éléments décrivant les états du jeu
Prédicats dynamiques	
terminal	vrai si l'état du jeu est terminal
goal	score des joueurs
legal	coups légaux dans l'état courant du jeu
next	transitions vers l'état suivant
does	choix des coups par les joueurs
true	définit les éléments décrivant l'état courant du jeu
Constantes numériques	
Nombres entiers de 0 à 100	définis comme scores possibles

TABLE 1.1 – Mots clés du GDL

1.3.2.3 Le prédicat `role`

Le prédicat **`role`** sert à énumérer les joueurs quand il est exprimé sous la forme d'un axiome. Dans toutes les descriptions de jeux que j'ai pu examiner, **`role`** n'apparaît jamais en conclusion d'une règle.

L'unique argument de **`role`** est toujours un symbole au sens de Datalog (un atome au sens de KIF). L'ordre dans lequel apparaissent ces axiomes joue un role puisque c'est celui utilisé par l'arbitre pour communiquer les différents coups joués à l'ensemble des joueurs au moment où il notifie un **`PLAY`** comme décrit à la section 1.3.3.4[9].

`role` avec une variable s'utilise en partie droite des théorèmes pour décrire des règles qui s'appliquent à tous les joueurs ou assurer la sécurité (*safety*) au sens de la définition du langage GDL [40] qui impose que les variables apparaissant dans une règle dans la conclusion ou dans une prémisse négative, apparaissent également dans une prémisse positive[10].

Les seuls **`legal`** utiles sont ceux qui prennent en premier argument un des joueurs défini par **`role`** mais certaines descriptions de jeux permettent d'attribuer des coups légaux à des entités qui ne sont pas des joueurs. Par exemple, une description inhabituelle mais compacte de TicTacToe décrit les coups possibles du non-joueur *case vide*.

Le nom des joueurs peut être n'importe quelle constante légale en GDL[11]. Dans la description du jeu *Beatmania*, un des deux joueurs chargé de composer la séquence de notes porte le nom **`random`**. En GDL-II, ce nom est réservé pour indiquer que ce joueur a le comportement prévisible de tirer au sort de manière équiprobable parmi les coups légaux à sa disposition. Il joue le rôle de la *nature*. Les coups de ce joueur spécial peuvent être choisis par l'arbitre alors que dans le GDL classique, il s'agit d'un joueur comme les autres.

---

[9] On aurait pu imaginer une description de jeu qui décrive un nombre arbitrairement grand mais fini de joueurs en faisant apparaître le prédicat **`role`** dans la conclusion d'un théorème, par exemple avec `(<= (role ?x) (gt ?x 0))` après avoir défini les opérations arithmétiques sur une partie des entiers mais cette possibilité a été écartée dans les spécification du GDL [40, page 13]. Le nombre maximal de joueurs que j'ai rencontré dans une description étant de six (pour les dames chinoises), on comprend que l'usage de théorèmes ait été considéré comme inutile. De plus, en l'absence de méthode imposée pour l'évaluation des règles qui produirait un ordre canonique, cela poserait un problème pour déterminer l'ordre des joueurs qui sert à communiquer les coups.

[10] L'ajout d'un théorème dont les prémisses contiennent `(role ghost)` alors qu'il n'y a pas de joueur `ghost` ne change rien à la description du jeu puisque cette prémisse sera toujours fausse. On peut obtenir des prémisses toujours fausses par d'autres moyens. Cette méthode permet de construire des descriptions de jeux arbitrairement complexes de façon à rendre leur évaluation et leur analyse automatique particulièrement ardue.

[11] La description d'un joueur nommé **`role`** par `(role role)` est probablement légale mais sans doute de nature à poser des problèmes à certains programmes d'évaluation du GDL.

1.3.2.4 Le prédicat `terminal`

Le prédicat **`terminal`** indique si l'état courant du jeu correspond à une fin de partie.

Quoi que la spécification GDL ne l'interdise pas formellement, le prédicat **`terminal`** n'est jamais utilisé comme prémisse d'un théorème dans les descriptions GDL présentes sur les serveurs[12].

Les prémisses permettant d'évaluer **`terminal`** ne peuvent dépendre que de l'état courant de la partie décrit par le prédicat **`true`** présenté dans la suite, avant que les joueurs aient choisi leurs coups. En aucun cas il ne peut dépendre directement ou indirectement des coups joués décrits par le prédicat **`does`** [40, page 12]. En effet, sa valeur de vérité détermine si l'étape suivante pour un raisonneur consiste à calculer les scores des différents joueurs ou bien à déterminer leurs coups légaux[13].

1.3.2.5 Les prédicats `true`, `init` et `next`

Ces trois prédicats servent à décrire l'ensemble dynamique des faits vrais qui décrivent une position donnée dans la partie : **`init`** pour la position initiale, **`true`** pour la position courante et **`next`** pour la position suivante.

Une situation donnée d'une partie est décrite par un ensemble fini de faits vrais donnés par le prédicat **`true`**[14] [15].

Le prédicat **`init`** sert à spécifier les faits présents dans cet ensemble au début de la partie[16].

---

[12] Fluxplayer, en 2009 au moins, ne pouvait pas traiter les descriptions de jeux utilisant **`terminal`** en prémisse (S Schiffer, Communication personnelle). On peut facilement réécrire une telle description, en remplaçant le symbole **`terminal`** par `done` et en ajoutant le théorème `(<= terminal done)`.

[13] Dans le cas où la partie est terminée comme indiqué par **`terminal`**, le prédicat **`legal`** qui décrit les coups possibles n'a pas de sens. On peut même imaginer que sa résolution soit impossible par exemple en faisant entrer le raisonneur dans une boucle infinie sans enfreindre les spécifications qui les interdisent.

[14] L'ensemble des faits qui sont vrais pour une position donnée du jeu peut être décrit de deux manières : soit l'ensemble des faits vrais au sens du raisonneur qui sont de la forme `(true fluent)` ; soit par l'ensemble de *f*luents qui sont les arguments possibles du prédicat **`true`**. Ces deux ensembles sont trivialement en bijection. Parler de *fluent* permet de parler des mêmes faits qu'ils apparaissent dans **`init`**, **`true`** ou **`next`**. La distinction entre ces deux ensembles n'est pas si importante car le contexte permet facilement de savoir si l'on parle du *fluent* ou du fait `(true fluent)`.

[15] Le *fluent calculus* est utilisé entre autres pour résoudre le *frame problem* [44] qui se pose dans les descriptions de jeux comme des automates en GDL. Nous commettons donc un abus de langage lorsque nous utilisons le terme *fluent* pour décrire les faits qui apparaissent à l'intérieur des **`true`**.

[16] Dans les premières descriptions de jeux, **`init`** était uniquement utilisé sous la forme d'axiomes. Dans les descriptions plus récentes, qui souvent décrivent des jeux plus complexes, **`init`** apparaît également en conclusion de règles.

Les spécifications GDL imposent que le prédicat **true**, qui décrit la position courante, apparaît toujours en prémisse dans les règles alors que le prédicat **next** apparaît seulement en conclusion des règles[17].

Une autre manière de décrire les transitions entre les états successifs de la partie peut se faire en utilisant un unique prédicat et une variable temporelle. Si l'arithmétique avait été inclue dans le langage, on aurait pu remplacer chaque fait `(init f)` par `(true f 0)`, chaque conclusion `(next f)` par `(true f (+ t 1))` et chaque prémisse `(true f)` par `(true f t)`[18].

1.3.2.6 Les prédicats does et legal

Les prédicats **legal** et **does** servent respectivement à indiquer, pour un état courant de la partie, les coups légaux pour chacun des joueurs et le coup choisi par chaque joueur. **legal** ne s'utilise normalement qu'en tant que conclusion des théorèmes et pas en tant que prémisse[19]. **does** ne peut apparaître que comme prémisse dans les théorèmes utilisés pour démontrer directement ou indirectement les faits **next**[20].

Ces prédicats sont d'arité 2 : le premier argument, lorsqu'il est instancié, désigne un des joueurs définis par le prédicat **role**[21] ; le second argument doit s'instancier en un terme représentant un coup légal.

---

[17] Un esprit taquin pourrait s'étonner qu'on ne puisse pas rencontrer un axiome concernant le prédicat **next** à cause de l'obligation de n'utiliser le prédicat **next** qu'en conclusion d'une règle. Pourtant, un axiome comme `(next cestparti)` peut être utile pour signaler que l'on a passé le premier coup. Il est trivial cependant d'obtenir la même information avec la règle sans prémisse `(<= (next cestparti))` qui elle respecte l'obligation.

[18] Même si l'arithmétique n'est pas incluse dans le langage GDL, elle pourrait être remplacée par une arithmétique de Peano si celle-ci n'était pas interdite par la limitation de récursion.

[19] Les restrictions de GDL n'imposent pas formellement que **legal** n'apparaisse qu'en tant que conclusion. Cependant, je n'ai jamais rencontré de description de jeu utilisée dans la compétition ou présente sur les serveurs dans laquelle ce soit le cas. Dans les jeux où c'est l'absence de coup légal qui marque la fin du jeu, comme NIM, AMAZONS ou SHEEPANDWOLF, il y a toujours un prédicat intermédiaire souvent nommé `legalMove` qui joue ce role. Si une description utilisait le prédicat **legal** dans la partie droite d'un théorème, on pourrait facilement la réécrire en introduisant un tel prédicat annexe :

```
(<= (legal ?x ?y) (legalbis ?x ?y))
(<= terminal (role ?r) (legalbis ?r ?m))
```

[20] Cette restriction est explicitement mentionnée dans la description formelle du GDL [40, Définition 20, page 11] qui spécifie également qu'il ne peut pas servir au calcul de **terminal**, **goal** ou **legal**. Le prédicat **does** ne peut servir qu'à calculer le passage d'une position à une autre ; **terminal**, **goal** et **legal** ne sont calculés qu'à partir d'une position.

[21] Si le premier argument du **legal** ou du **does** est une variable, il sera forcément instancié lors de l'évaluation comme l'impose les spécifications du GDL. Cette instanciation devrait être un des joueurs décrits par le prédicat **role** mais cette restriction n'est pas imposée dans les spécifications, tout au plus elle est suggérée pour **does**. On peut penser que la plupart des joueurs seraient déconcertés par l'arrivée en cours de partie via **legal** d'un joueur qui n'était pas convié au départ par le prédicat **role**. L'absence du nom du joueur dans le protocole utilisé par l'arbitre pour notifier les coups joints complique l'arrivée d'un nouveau joueur via **does**.

Le prédicat **legal** doit être appelé une fois qu'on a vérifié que **terminal** s'évalue à faux. En effet, rien ne garantit dans les spécifications du GDL que les ensembles de coups légaux seront vides dans le cas où **terminal** est vrai. En revanche, elles précisent que tant que la partie n'est pas terminée, l'ensemble de coups légaux de chaque joueur n'est jamais vide. Cette restriction renforce l'obligation implicite que chaque transition entre deux états de la partie se produit en fonction d'un *coup joint* composé d'un coup joué par chacun des joueurs[22].

En utilisant les coups joints, le langage GDL permet de décrire des jeux à coups simultanés aussi bien que des jeux à coups alternés dans lesquels l'alternance entre les joueurs ne se réduit pas nécessairement à un simple tour de table.

Puisqu'un joueur qui n'a qu'un seul coup légal à un moment de la partie est obligé de choisir ce coup, les jeux à coups alternés sont usuellement décrits en imposant un unique coup légal à tous les joueurs dont ça n'est pas le tour de jouer. Ce coup est souvent désigné par `noop` y compris dans les spécifications et associé à un prédicat `control` qui sert à identifier le joueur qui a la main[23].

Le prédicat **does** est aussi d'arité deux : son premier argument désigne le joueur effectuant le coup et le second est un terme représentant le coup joué. Le joueur doit obligatoirement faire partie de l'ensemble des joueurs définis par **role**. Le coup choisi par un joueur donné doit être unique et obligatoirement faire partie de l'ensemble des coups légaux de ce joueur indiqués par le

[22] Les spécifications ne précisent que le fait que le langage GDL modélise un automate dont les transitions sont étiquetées par les coups joints.

[23] Dans la description GDL du jeu de Nim, les deux joueurs sont définis avec les prédicats suivants :

```
(role player1)
(role player2)
```

Le passage de la main d'un joueur à l'autre est déterminé par le prédicat `control` qui n'est *pas* défini sur la base du coup `noop` :

```
(init (control player1))
(<= (legal ?p noop) (role ?p) (not (true (control ?p))))
(<= (next (control ?p)) (role ?p) (does ?p noop))
```

Le prédicat `control` peut être vu comme un compteur modulo 2 mais cette définition ne permet pas de le découvrir avec une analyse statique comme celle évoquée dans la note 26 alors que la définition originale le permet :

```
(init (control player1))
(<= (next (control ?p2)) (true (control ?p1)) (next_player ?p1 ?p2))
(next_player player1 player2)
(next_player player2 player1)
```

prédicat **legal**[24]. Le comportement du jeu est indéfini lorsqu'un coup illégal est joué et toutes les restrictions imposées par les spécifications ne sont plus garanties, en particulier sa terminaison en un nombre fini de coups. L'arbitre veille à ce que les coups soient toujours légaux en substituant le cas échéant un coup légal à un coup illégal ou à une absence de réponse d'un des joueurs[25].

Pour un état donné du jeu, l'ajout des coups des joueurs via le prédicat **does** est un préalable au calcul du prédicat **next** car celui-ci en dépend[26].

1.3.2.7 Le prédicat goal

Le résultat d'une partie est décrit par un prédicat **goal** pour chaque joueur. Le prédicat indique le score obtenu par chacun avec un nombre entier compris entre 0 (*score minimum*) et 100 (*score maximum*). La description des règles du jeu de Nim contient un théorème qui indique que le joueur qui ne peut pas jouer a perdu (*règle normale*) :

```
(<= (goal ?p 0) (true (control ?p)))
```

Le théorème doit être accompagné d'un autre théorème qui lui indique la récompense obtenue par l'autre joueur, ici 100, par exemple avec

```
(<= (goal ?p 100) (true (control ?p1)) (next_player ?p ?p1))
```

L'obligation de déclarer la récompense obtenue par chaque joueur permet de décrire avec GDL les jeux qui ne sont pas à somme nulle, comme des jeux coopératifs où les deux joueurs doivent collaborer pour obtenir tous les deux des scores élevés.

Les entiers de 0 à 100 qui apparaissent comme second argument de **goal** sont à proprement parler des mots réservés du langage GDL : l'obfuscateur, utilisé pendant la compétition pour éviter que les joueurs utilisent les noms des prédicats ou des variables pour deviner leurs rôles, doit laisser ces nombres intacts ; la relation *supérieur* doit conserver son sens usuel sur ces nombres pour permettre aux joueurs de chercher à maximiser leurs scores. Cependant, les nombres qui peuvent apparaître ailleurs dans la règle ne sont pas eux des mots réservés du langage : un obfuscateur pourrait remplacer par des mots arbitraires ou que leur ordre ne soit pas l'ordre usuel des entiers.

[24] Tant que la partie n'est pas terminée, chaque joueur doit jouer un coup parmi ceux qui sont légaux. Cependant, la description GDL peut-être utilisée à d'autres fins que de simplement jouer une partie. Par exemple, pour déterminer incrémentalement l'ensemble des coups qui peuvent être légaux au cours d'une partie, on peut éliminer les négations et jouer simultanément tous les coups légaux. C'est un manière d'obtenir un ensemble ayant la même utilité que celui donné par le prédicat **input** lorsqu'il n'est pas présent dans la description ou bien qu'on ne souhaite par l'utiliser.

[25] Des simulations dans lesquelles des joueurs jouent des coups illégaux pourraient-elles être utiles à la découverte de certaines propriétés du jeu ?

[26] Le fait de ne pas jouer un seul coup peut être utilisé dans une utilisation de la description GDL à d'autres fins que de jouer une partie. Par exemple, le calcul systématique de tous les états successifs décrits par **next** sans jouer de coup permet dans les descriptions de jeux usuelles d'identifier le compteur utilisé pour garantir la terminaison du jeu en limitant le nombre de coups joués. En effet, celui-ci est en général de la forme :

Toutefois, identifier automatiquement quelles occurrences des nombres seront utilisées en définitive par **goal** et lesquelles ne le seront pas est une tâche un peu délicate qui n'est pas réalisée par les obfuscateurs que j'ai pu observer. De ce fait, des nombres sont souvent définis par la règle GDL avec leur représentation décimale usuelle, obéissent aux règles de l'arithmétique, et ne sont pas renommés par l'obfuscateur. À ma connaissance, aucun des programmes joueurs qui participent à la compétition n'utilise cet aspect pour accélérer les calculs[27].

La description de GDL impose qu'un **goal** soit défini pour chacun des joueurs. Du fait que les descriptions de règles ne sont pas toujours correctes, ce n'est pas toujours le cas : des descriptions de jeux dans lesquelles certains joueurs n'ont pas de score ou bien obtiennent plusieurs scores différents sont encore présentes sur des serveurs de GGP[28].

1.3.2.8 Les prédicats `input` et `base`

Les prédicats **input** et **base** servent à décrire d'une part l'ensemble des coups légaux (pour **input**) et d'autre part l'ensemble des faits qui pourront être vrais au cours du déroulement d'une partie et donc apparaître dans un **true** (pour **base**).

input et **base** n'apparaissent pas dans la spécification originelle de GDL. Ils ont été introduit dans les compétitions organisées par Stanford vers 2012. Je suppose que c'était pour permettre aux joueurs développés à Stanford (sur lesquels on dispose de fort peu d'informations, d'autant qu'ils

```
1 (init (step 0))
2 (<= (next (step ?n+1)) (true (step ?n)) (++ ?n ?n+1))
3 (++ 0 1) (++ 1 2) ... (++ 50 51)
4 (<= terminal (true (step 51)))
```

Cette procédure permet de déterminer que chaque état successif du jeu comprendra un terme du type `(step ?i)` où `?i` est le numéro de l'état avec une complexité linéaire en fonction de la durée d'une partie au lieu d'une complexité exponentielle si l'on considérait tous les coups possibles.

[27] S'il était connu que des joueurs utilisent le fait que les entiers portent le plus souvent leur nom décimal usuel dans les règles GDL, il ne serait pas très difficile de modifier un obfuscateur pour lui faire remplacer ces noms par d'autres quand ils ne sont pas employés par **goal** : l'utiliser dans un joueur serait donc un effort qui ne pourrait apporter qu'un avantage temporaire en compétition sans faire progresser la connaissance dans le domaine.

[28] La description de PEG souffrait d'une telle déficience : quand un joueur terminait avec un seul pion, son score était de 99 ; quand ce pion était au centre du damier, il obtenait un score de 100 ; du coup, un joueur pouvait obtenir *à la fois* les scores 99 et 100. Cette erreur dans la description est passée inaperçue jusqu'en 2009, quoi que le jeu ait été utilisé aussi bien pendant les phases de qualification que dans les finales de la compétition officielle depuis la première compétition de 2005 [Jean Méhat, *A bug in peg description*, general-game-playing@lists.stanford.edu, 10 nov. 2009]. Pendant une compétition, il n'est pas toujours facile d'identifier qu'une description présente une de ces déficiences ; pour cette raison, LEJOUEUR reconnaît ces situations et tente d'y remédier : si un joueur n'a pas de score ou bien obtient plusieurs scores différents, il attribue un score de 0 à ce joueur ; cela permet de traiter ces cas normalement et d'encourager les joueurs à ne pas se placer dans ces situations dont l'issue est mal définie pour eux.

ne sont pas autorisés par le règlement à participer à la compétition principale organisée à AAAI) de pré-traiter les descriptions en GDL – sans doute en construisant des propnets – pour accélérer leur traitement.

Comme d'autres modifications de la spécification de GDL (comme la suppression du prédicat **or** par exemple), celle-ci s'est faite d'une façon quasi-subreptice, par un simple paragraphe dans le règlement de la compétition annuelle organisée par Stanford. De ce fait, les descriptions de jeux antérieures à cette modification n'ont pas été mises à jour et il est possible de rencontrer aussi bien des descriptions qui utilisent **input** et **base** que d'autres qui ne l'utilisent pas[29].

De plus, **input** et **base** doivent contenir tous les coups légaux et tous les faits qui peuvent être vrais mais on n'est pas obligé de s'y limiter : ils peuvent également comprendre des coups qui ne seront jamais légaux ou des faits qui ne seront jamais vrais. Quand les descriptions de jeux anciennes ont été modifiées pour y ajouter les **input** et **base**, cela a parfois été réalisé en incluant un grand nombre de coups qui ne seraient jamais légaux ou de faits qui ne seraient jamais vrais. J'ai constaté lors des mesures de temps nécessaire pour instancier les règles décrites à la section 2.3.1 qu'il était dans certains cas plus rapide de recalculer l'ensemble des coups possibles et des faits susceptibles d'être vrais plutôt que d'utiliser les **input** et **base** fournis par la règle du jeu.

1.3.2.9 Les constantes numériques

Les nombres entiers de 0 à 100 sont des mots clés réservés uniquement en second paramètre du prédicat **goal**. Ils indiquent le score obtenu par les joueurs. Ainsi en GDL, le nombre d'issues possibles pour un jeu est limité à 101^n où n est le nombre de joueurs. La relation d'ordre usuelle sur les entiers permet de graduer la performance de chaque joueur[30].

1.3.3 Le protocole de communication avec le Game Manager et le Game Master

La donnée de la description GDL détermine complètement le système de transitions d'états du jeu. Ce n'est cependant pas suffisant pour définir totalement le jeu car il n'y est pas précisé entièrement les règles que doivent suivre les joueurs notamment la manière dont ils peuvent jouer

[29] Seuls dix pour cent des jeux présents sur le serveur de jeux Tiltyard utilisent **input** et **base**. En revanche, les descriptions du serveur de Stanford l'utilisent toutes ; cependant leur utilisation est souvent un ajout a posteriori qui souffre des déficiences décrites dans le paragraphe suivant. Pire encore, certaines descriptions anciennes peuvent utiliser un prédicat auxiliaire **input** ou **base** qui n'a rien à voir avec le rôle que les descriptions plus modernes leur attribue.

[30] Il est à noter que dans la compétition annuelle AAAI, ces scores sont cumulés pour déterminer les joueurs qui passeront la première phase de qualifications. Ce sont ensuite les victoires et défaites qui déterminent le déroulement du tournoi de la seconde phase où s'affrontent les joueurs deux à deux. C'est alors la comparaison des scores des deux joueurs qui détermine le gagnant et le perdant.

leurs coups par rapport aux autres joueurs et les contraintes temporelles qu'ils doivent respecter. Le protocole de communication des règles, des coups et des scores fait partie de la spécification du GDL. Dans ce protocole, en plus des joueurs, il y a un agent supplémentaire appelé Game Manager (Responsable du jeu)[31], chargé d'envoyer aux joueurs une description du jeu en GDL ainsi que les contraintes temporelles qu'ils devront respecter, réceptionner les coups joués et informer des coups joués par les autres joueurs et enfin donner le résultat du jeu. Ce Game Master dispose aussi de commandes techniques permettant de vérifier si un joueur est disponible pour un match ou pour abandonner un jeu.

1.3.3.1 Transmission des messages

Les joueurs et le Game Manager sont des programmes s'exécutant sur des machines connectées en réseau et communiquent entre eux à travers une connexion HTTP. Le protocole suppose que chaque joueur joue le rôle de serveur et écoute sur un port particulier les requêtes du Game Manager. Le Game Manager fait office de client pour ces serveurs. Les joueurs n'ont donc pas la possibilité de solliciter directement le Game Manager et n'ont que la possibilité de répondre à une sollicitation de celui-ci.

Les messages HTTP ont un entête standard qui est composé d'un identifiant de l'expéditeur qui sera toujours `GAMEMASTER`, d'un identifiant du joueur tel qu'il a été enregistré auprès du Game Master, du type de contenu `text/acl` et de la longueur du message. Le joueur conserve la connexion ouverte le temps de préparer sa réponse qu'il envoie selon ce même protocole. La figure 1.2 montre un exemple de communication entre le Game Manager et un joueur.

1.3.3.2 Les commandes du Game Manager

Il n'existe à notre connaissance que cinq commandes qu'un joueur est susceptible de recevoir et nous les décrivons ci-dessous : trois sont détaillées dans la spécification du GDL, deux ont été introduites plus tard et n'ont pas de spécification officielle. Les commandes sont envoyées par le Game Master ou le Game Manager, les joueurs n'ayant pas la possibilité d'initier une communication avec eux.

La commande `START` prend cinq arguments qui informent notamment le joueur des règles du jeu, son rôle dans ce jeu ainsi que le temps maximal accordé pour l'analyse des règles et le temps

[31] Ce Game Manager est souvent appelé Game Master (Maître du jeu) dans la communauté GGP, appellation issue des jeux de rôle. Ce que la spécification définit par Game Master est l'ensemble du cadre applicatif permettant le GGP. Il stocke les données de connexion des joueurs, les descriptions de jeux en GDL, l'historique des parties et des joueurs et facilite les communications entre le Game Manager et les joueurs.

```
POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: LEJOUEUR
Content-type: text/acl
Content-length: 41
(PLAY MATCH.3316980892 ((MARK 2 2) NOOP))
```

(a)

```
HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 10
(MARK 3 2)
```

(b)

FIGURE 1.2 – Le Game Manager envoie une commande **PLAY** au joueur LEJOUEUR (a) et celui-ci répond en choisissant le coup (MARK 3 2) (b).

maximal autorisé par coup :
(**START** <MATCHID> <**ROLE**> <DESCRIPTION> <STARTCLOCK> <PLAYCLOCK>)
où

<MATCHID> est un identifiant unique utilisé par le serveur de GGP pour identifier la partie qui débute. Les commandes suivantes faisant référence à cette partie utiliseront le même identifiant.

<DESCRIPTION> est constitué d'une parenthèse ouvrante, suivie de l'ensemble des règles de la description GDL puis d'une parenthèse fermante.

<**ROLE**> est une constante désignant le rôle qu'à le joueur recevant la commande parmi ceux déclarés par le prédicat **role** dans la description.

<STARTCLOCK> et <PLAYCLOCK> sont des nombres entiers représentant le nombre de secondes accordées pour répondre respectivement à la commande **START** et aux commandes **PLAY** qui suivront. C'est le Game Manager qui contrôle le temps. Ainsi, le temps de propagation des messages sur le réseau n'est pas pris en compte. Les joueurs doivent donc en tenir compte et ajouter une marge de sécurité pour ne pas être sanctionné par le Game Manager pour un coup envoyé hors délai. L'absence de réponse du joueur est traitée comme les réponses trop tardives : le Game Manager choisit un coup à la place du joueur pour que la partie puisse continuer avec les autres joueurs.

La commande PLAY sert à informer les joueurs des coups qui viennent d'être joués. Elle prend deux arguments : l'identifiant du match et une liste des coups des joueurs dans un ordre identique à celui du prédicat **role** dans la description GDL :
(**PLAY** <MATCHID> (<C1> <C2> ... <Cn>)
où <Ci> est le coup joué par le joueur *i*. Par exemple, si dans la description GDL, les rôles sont déclarés dans l'ordre (**role** red) (**role** black), cela signifie que red a joué <C1> et black

a joué <C2>. Lorsqu'il s'agit de la première commande **PLAY** d'un match, étant donné qu'aucun coup n'a été joué, elle prend une forme différente :
(**PLAY** <MATCHID> NIL)[32].

La commande STOP est similaire dans sa forme à la commande **PLAY** :
(**STOP** <MATCHID> (<C1> <C2> ... <Cn>)
Elle informe les joueurs des derniers coups joués et signale que la partie est terminée. Après cette commande, le Game Manager n'enverra plus aucun message avec l'identifiant de cette partie. Les scores obtenus par les joueurs, qui sont pourtant calculés par le Game Manager, ne sont pas communiqués. C'est donc à eux qu'il revient de le calculer à partir de la dernière position et des coups envoyés par cette commande **STOP**.[33]

1.3.3.3 Les commandes du Game Master

La commande INFO ne fait pas partie de la spécification alors qu'elle est utilisée sur certains serveurs de GGP et lors des compétitions depuis 2013. C'est une commande du Game Master et non du Game Manager. Elle lui permet d'interroger un joueur sur sa disponibilité à jouer un nouveau match.[34]

La commande ABORT ne fait pas partie non plus de la spécification et sert à mettre fin à un match avant d'avoir atteint une position terminale du jeu :
(**ABORT** <MATCHID>)
Elle a sans doute été introduite car la commande **STOP** n'est pas prévue pour arrêter un match en cas de problème technique. D'une manière générale, utiliser une commande en dehors du cadre pour lequel elle est prévue est un risque de faire planter le joueur qui la reçoit. Ça pourrait être le cas pour **STOP** si le Game Manager n'avait pas de série de coups légaux à envoyer.

1.3.3.4 Les réponses du joueur

Il n'y a que deux types de réponse que peuvent apporter les joueurs aux commandes précédentes : il s'agit soit d'un mot, soit d'un coup au format KIF.

[32] Cette information se trouve dans l'annexe des spécifications où un exemple de communication est donné.

[33] Nous verrons qu'en GDL-II, ce calcul final des scores par un joueur n'est pas toujours possible dans les cas où il ne dispose pas de suffisamment d'information.

[34] Lors de la préparation de la compétition de 2013, nous devions faire valider notre joueur pour la compétition en permettant aux organisateur de faire une batterie de tests sur celui-ci. Cette commande **INFO** est alors apparue (elle s'appelait alors PING) et nulle part il n'était précisé ce que le joueur devait répondre.

La réponse `READY` est la seule possible à une commande **`START`** et elle peut être envoyée tant que le temps indiqué par `STARTCLOCK` n'est pas épuisé. En cas d'absence de réponse, le Game Manager poursuivra tout de même avec la commande **`PLAY`**. Il n'y a pas d'intérêt pour un joueur à répondre trop tôt avant la fin du temps accordé sauf dans le cas où celui-ci a déjà découvert une stratégie gagnante.

Le coup choisi par le joueur est envoyé à toute commande **`PLAY`**. Il s'agit du second argument du prédicat **`legal`**. Par exemple, si parmi ses coups légaux, le joueur `red` a choisi `(legal red (mark 1 2))`, alors il enverra simplement `(mark 1 2)`.

La réponse `DONE` est la seule possible à une commande **`STOP`**.

La réponse `ABORTED` est de même apportée à toute commande **`ABORT`** même si aucun document de spécification ne le précise. De toute manière, cette commande n'est envoyée qu'une seule fois par match à notre connaissance et par conséquent, n'importe quelle réponse peut faire l'affaire.

Les réponses `AVAILABLE` et `BUSY` sont apportées par le joueur à toute commande **`INFO`** pour signaler respectivement s'il est disponible pour un match ou bien occupé.

1.3.4 Les limites du GDL

1.3.4.1 Expressivité du GDL

D'après la spécification, les jeux que permet de représenter le langage GDL sont multijoueurs finis, discrets, déterministes et à information complète [40, page 1]. Le modèle sous jacent que ses auteurs utilisent est celui d'une machine à états finis qui est un cas particulier de notre modèle de système de transitions d'états introduit au début de ce chapitre.

Cependant, la représentation explicite de cet automate serait peu pratique à la fois pour la communication entre les différents joueurs et le Game Manager et pour raisonner dessus. À titre d'exemple, une machine à état finis pour représenter explicitement les échecs nécessiterait plus de 10^{28} états distincts. Un autre désavantage de la machine à états finis est qu'elle manque de modularité en ce sens que la fonction de transition ne rend pas compte directement des effets des actions des joueurs sur le jeu (particulièrement dans le cas des jeux à coup simultanés). Le langage GDL propose donc de représenter implicitement cette machine à états finis en décrivant les jeux de manière plus compacte et modulaire.

Les spécifications décrivent la sémantique du GDL comme étant celle de Datalog munie de la négation par l'échec et des symboles de fonctions et y ajoutent deux restrictions.

La première restriction sert à garantir que lors de l'application d'un théorème, toutes les variables seront instanciées. Elle impose que toute variable apparaissant dans la conclusion d'un théorème ou dans une prémisse négative (**`not`** ou **`distinct`**), apparaisse aussi dans une prémisse positive.

La seconde restriction sert à borner la croissance de la profondeur des termes pour tous les calculs d'inférence entre deux états successifs du jeu. Elle s'énonce ainsi : étant donnée une règle

$$p(t_1, ..., t_n) :- b_1, ..., q(v_1, ..., v_k), ..., b_m$$

pour laquelle le graphe de dépendance entre les prédicats fait apparaître un circuit entre p et q. Alors, chaque v_j doit soit être totalement instancié, soit être l'un des t_i, soit apparaître dans un des b_i tel que $b_i = r(...v_j...)$ avec r n'apparaissant pas dans un circuit avec p.

La figure 1.3 présente deux définitions de l'arithmétique de Peano. En (a), la définition est illégale car la profondeur des termes n'est pas bornée par la seconde restriction. En effet, `entier` dépend de lui même et la variable `?x` n'est ni instanciée, ni en paramètre de la conclusion et n'apparaît pas dans une autre prémisse car il n'y en a pas. En revanche, la définition (b) est légale car la variable `?x` apparaît dans `(true (int ?x))` qui n'est pas dans un circuit avec le prédicat `entier`. Pour que l'arithmétique soit totalement définie en (b), il faut qu'une partie soit infiniment longue car cette manière de procéder ne produit qu'un seul nouvel entier à chaque tour. La notion de description GDL *bien formée* que nous allons voir à présent empêche en réalité de définir l'arithmétique de Peano.

Les spécifications imposent trois propriétés à une description pour qu'elle soit qualifiée de *bien formée* : elle ne doit pas produire de jeux infiniment longs ; chaque joueur dispose d'un moins un coup légal pour chaque état non terminal accessible depuis l'état initial ; pour chaque joueur, il existe une suite de coups-joints qui mène à un état terminal où la valeur de **`goal`** est maximale pour ce joueur. Elles suggèrent aussi qu'il est possible vérifier qu'une description est bien formée par force brute en développant tout l'arbre de jeu. Or, sans ces trois propriétés, Saffidine a prouvé que le GDL était Turing complet [63] et par conséquent que le problème de savoir si une description conduit à un jeu infiniment long est indécidable. Par conséquent, il n'existe aucun algorithme général permettant de vérifier qu'une description est bien formée. Cependant, cette découverte n'implique pas que le GDL décrit aussi les jeux infiniment longs car une description GDL doit être bien formée. Il aurait peut-être été plus prudent de restreindre le champ des descriptions GDL utilisables en GGP à celles accompagnées d'une preuve de terminaison.

```
(<= (entier (successeur ?x))
    (entier ?x))
(entier 0)
```

(a)

```
(<= (entier ?x)
    (true (int ?x)))
(init (int 0))
(next (int 0))
(<= (next (int (successeur ?x)))
    (int ?x))
```

(b)

FIGURE 1.3 – Définition illégale (a) et légale (b) de l'arithmétique de Peano en GDL.

1.3.4.2 Complétude et perfection de l'information en GDL

Un jeu représenté en GDL est à information complète car les gains des différents joueurs sont connus de tous. Il est à information parfaite pour les jeux à coups alternés mais s'il existe des états du jeu où deux joueurs ont plus d'un coup légal, alors le jeu est à information imparfaite.

1.3.4.3 Communication des coups

Le protocole de communication entre le Game Manager et les joueurs impose que les coups soient synchrones. L'horloge est par coup et non sur la durée totale du jeu. Un joueur peu évidemment envoyer son coup sans utiliser la totalité du temps imparti mais cela n'a aucun effet sur la durée du jeu si les adversaires jouent plus lentement (même pour envoyer un unique coup possible noop). Il n'y a donc pas de possibilité de jouer en temps réel, c'est à dire un mode de jeu dans lequel un même coup possède un effet différent suivant le moment où il est joué.

1.3.4.4 Sens des communications

Comme nous l'avons vu à la section 1.3.3, le Game Manager se comporte comme un client et le joueur comme un serveur. En cours de partie, puisque le Game Manager envoie la liste des coups précédemment joués et attend en réponse le coup suivant, la liaison TCP doit rester ouverte pendant la durée maximale fixée en début de jeu. Pour de longues périodes de réflexion, il est parfois compliqué de garder la liaison ouverte. Lorsque cette liaison est interrompue, le joueur n'a aucune possibilité de recontacter le Game Manager pour lui envoyer son coup.

1.3.5 Les variantes du GDL

Le GDL-II été introduit en 2011 en tant qu'extension du GDL pour représenter les jeux à information incomplète ou imparfaite [71][35].

Cette extension ne comporte que deux ajouts : un joueur **`random`** (appelé plus communément la nature en théorie des jeux) peut avoir un rôle dans le jeu et choisit toujours son coup parmi ses coup légaux en effectuant un tirage aléatoire uniforme ; un nouveau prédicat **`sees`** est introduit pour communiquer aux joueurs des informations sur la partie (perceptions).

Le protocole de communication du GDL-II a aussi été modifié pour que le Game Manager ne révèle pas aux joueurs les coups des autres adversaires et pour qu'il puisse lui communiquer les perceptions. Une description plus détaillée est présentée à la section 3.1.2.

Une variante permettant aux joueurs d'envoyer leurs coups en temps réel a aussi été proposée [36, 37].

1.3.6 Les compétitions de GGP

1.3.6.1 La compétition IGGPC

L'IGGPC (*International General Game Playing Competition*) est une compétition annuelle organisée au mois d'août depuis 2005 par le groupe logique de l'université de Stanford. Après les premières compétitions plutôt confidentielles, ce rendez-vous s'est installé comme l'événement qui mesure les avancées dans le domaine du GGP de Stanford. Pendant plusieurs années, la compétition a eu lieu pendant la conférence AAAI ou IJCAI avec la possibilité d'y participer à distance. Depuis 2014, elle n'a lieu qu'à distance, ce qui coïncide avec l'arrivée de nombreux participants ayant suivi le cours de GGP du MOOC *Coursera*.

La liste des vainqueurs par année de la compétition IGGPC est présentée à la figure 1.4. Cluneplayer, le gagnant de la première compétition utilise un algorithme alpha-beta à partir d'heuristiques construites par une analyse des règles du jeu [19]. À cet effet, il cherche à détecter un tablier de jeu, les pièces, les compteurs de tours et d'autres caractéristiques typiques des jeux utilisés en GGP même si la méthode de détection repose souvent sur des habitudes communes aux rédacteurs de descriptions GDL. Le second gagnant, FluxPlayer, utilise la logique floue pour estimer la valeur des positions intermédiaires à partir de la relation *goal* du GDL qui détermine les scores en fin de partie [65]. La victoire de CadiaPlayer en 2007 marque l'installation en GGP des méthodes reposant sur des simulations de parties via l'algorithme MCTS et la politique UCT [25].

[35] C'est sans doute par choix délibéré que le II représente à la fois les initiales des mots incomplet et imparfait et aussi le nombre deux en chiffre romains.

Année	Nom du joueur	Auteur	Institution
2005	Cluneplayer	Jim Clune	University of California, Los Angeles
2006	FluxPlayer	Stephan Schiffel Michael Thielscher	Dresden University of Technology
2007	CadiaPlayer	Yngvi Björnsson Hilmar Finnsson	Reykjavík University
2008	CadiaPlayer	Yngvi Björnsson Hilmar Finnsson Gylfi Þór Guðmundsson	Reykjavík University
2009	Ary	Jean Méhat	Université Paris 8
2010	Ary	Jean Méhat	Université Paris 8
2011	TurboTurtle	Sam Schreiber	indépendant
2012	CadiaPlayer	Hilmar Finnsson Yngvi Björnsson	Reykjavík University
2013	TurboTurtle	Sam Schreiber	indépendant
2014	Sancho	Steve Draper Andrew Rose	indépendant
2015	Galvanise	Richard Emslie	indépendant
2016	Woodstock	Éric Piette	Université d'Artois

FIGURE 1.4 – Liste des joueurs ayant remporté la compétition IGGPC depuis 2005 avec leurs auteurs et leur université d'affiliation le cas échéant.

Pour interpréter le règles GDL, CadiaPlayer utilise l'interpréteur YapProlog. Il a intégré plusieurs améliorations à UCT qui estiment la valeurs des actions en plus de celles des positions. Ces améliorations qui lui ont permis de remporter la compétition à trois reprises avant son retrait en 2013 et il aurait été intéressant qu'elles puissent être utilisées avec les interpréteurs de GDL significativement plus rapide qui se sont développés depuis. Il fonctionne sur une architecture distribuée. Ary, le joueur de Jean Méhat développé à l'université Paris 8 possède des points commun avec CadiaPlayer comme l'utilisation de YapProlog pour l'interprétation des règles GDL, la recherche MCTS avec UCT et l'architecture distribuée sur laquelle il fonctionne [47]. Il a remporté à deux reprises la compétition en 2009 et 2010. Même s'il en est complètement distinct, les choix effectués dans Ary en termes d'interprétation du GDL et d'utilisation des méthodes MCTS ont contribué à la création rapide de LeJoueur qui a participé à sa première compétition en 2012. TurboTurtle est le premier joueur non affilié à une université à remporter la compétition en 2013, même si son auteur est un des contributeurs les plus actifs au GGP et qu'il est issu de l'université de Stanford. Il est le premier joueur à notre connaissance à avoir utilisé la simulations de circuit logiques pour accélérer significativement l'interprétation du GDL. Nous n'avons pas de sources décrivant ce joueur même si nous supposons que le code GGPBase[36] mis à disposition lors des cours du MOOC sur le GGP en est extrait. Les joueurs gagnants des années 2014 et 2015 ont été conçu par des auteurs indépendants ayant suivi ce cours et possèdent des similitudes avec TurboTurtle, son auteur étant un intervenant de ce cours. Enfin, la compétition de 2016 a vu le retour d'un gagnant conçu par des universitaires. Le joueur Woodstock de l'Université d'Artois utilise une approche originale dirigée par la résolution de réseaux de contraintes stochastiques combinée à une méthode d'échantillonage de type Monte-Carlo [54].

1.3.6.2 La compétition *Tiltyard Open*

Le *Tiltyard Open* est une nouvelle compétition qui a été crée en 2015 à l'initiative d'Alex Landau et qui utilise le serveurs Tiltyard sur lequel a lieu un tournoi permanent de GGP. Elle a lieu au mois de décembre est constitue un second rendez-vous annuel lors duquel la communauté GGP peut mesurer les avancées du domaine en faisant s'affronter des programmes joueurs. La liste des vainqueurs par année de la compétition Tiltyard Open est présentée à la figure 1.5. Le programme LeJoueur qui est décrit dans ce mémoire a remporté l'édition de 2015.

[36] `https://github.com/ggp-org/ggp-base`

Année	Nom du joueur	Auteur	Institution
2015	LeJoueur	Jean-Noël Vittaut	Université Paris 8
2016	NeuralNed	Ed Holland	indépendant

FIGURE 1.5 – Liste des joueurs ayant remporté la compétition Tiltyard Open depuis 2015 avec leurs auteurs et leur université d'affiliation le cas échéant.

1.4 Extensions aux jeux vidéo

1.4.1 Arcade Learning Environment

L'Arcade Leaning Environment (ALE) est une interface pour un émulateur d'Atari 2600, une console de jeux commercialisée à partir de 1977 et qui dispose d'une vaste collection de jeux [4]. Cette interface permet au joueur d'envoyer des mouvements du Joystick et recevoir le contenu de l'écran ou de la RAM. Il inclut aussi la possibilité de transformer le jeu en un problème standard d'apprentissage par renforcement par identification du score et par la détection de la fin du jeu.

En temps réel, l'émulateur génère 60 frames par seconde. Pour les besoins de simulation, il était capable de générer 6000 frames par seconde au moment de sa publication en 2013.

1.4.2 General Video Game Playing

S'inscrivant clairement dans la continuité du General Game Playing tel que défini par le groupe logique de Stanford, le General Video Game Playing (GVGP) propose un cadre applicatif pour développer des intelligences artificielles généralistes où l'ordre de grandeur du temps entre deux coups est en dizaines de millisecondes [38, 53].

La motivation pour créer un langage spécifique pour le GVGP est que le GDL n'est pas bien adapté pour décrire les jeux vidéos et les auteurs en indiquent plusieurs raisons. Les jeux vidéo comportent souvent une part de non-déterminisme due à la présence de personnages non-joueurs (PNJ) ou au hasard[37]. Les actions qui déterminent les changements d'état du jeu proviennent soit des entrées des joueurs, soit des comportements des PNJ, qui peuvent avoir lieu simultanément[38]. Les environnements de jeux vidéo peuvent faire appel à des effets continus ou temporels, une physique des objets en temps réel et des collisions en fonction du contexte ou des interactions entre plusieurs joueurs et artefacts. La tâche du joueur est en partie d'apprendre à prédire ces dynamiques. Dans les jeux vidéo, les joueurs évoluent dans un environnement beaucoup plus

[37] Le GDL-II permet pourtant de représenter des jeux comportant du hasard.
[38] Le GDL permet pourtant les actions simultanées.

large que celui des jeux de société. Cependant, comparativement, les interactions entre le joueur et l'environnement sont plus rares. Autrement dit, les décisions du joueur sont rarement voire jamais influencées par la complexité de l'environnement.

1.5 Conclusion

Nous avons présenté le domaine du General Game Playing dont le but est de développer des approches permettant de jouer efficacement à n'importe quel jeu de stratégie sans utilisation de connaissances d'experts ou de bases de parties. Le GDL, le langage de description de jeux de Stanford permet de décrire une grande variété de jeux fini, déterministes et à information complète. Son extension GDL-II l'étend aux jeux à information imparfaite.

Le serveur Tiltyard proposant un tournoi permanent permet aux joueurs de mesurer leur niveau entre les deux principales compétitions annuelles : l'*International General Game Playing Competition* et l'*Open Tiltyard*. Le niveau des joueurs de GGP s'est significativement amélioré depuis la première compétition en 2005. Les deux principales ruptures ont eu lieu lors de l'introduction des méthodes MCTS basées sur UCT et lors de la généralisation de l'utilisation de circuits logiques permettant d'accélérer de manière importante l'interprétation du GDL et dont l'analyse permet de détecter des caractéristiques du jeu en termes de topologie, de symétrie et de composition.

Chapitre 2

Interprétation du GDL

Nous présentons dans ce chapitre les caractéristiques de l'interprétation des descriptions de jeux en GDL. Cette description est cruciale dans la mesure où d'une part une interprétation déficiente conduira bien sur à ignorer des coups légaux ou bien à jouer des coups illégaux (voire, comme détaillé plus loin, à ne pas être en mesure de répondre) ; d'autre part, la vitesse d'interprétation des descriptions de jeux conditionne directement la taille du sous-ensemble de l'espace du jeux que le joueur pourra explorer dans un temps donné et donc la qualité de son jeu. Björnsson et Schiffel ont comparé la vitesse d'exécution de plusieurs raisonneurs GDL[39] et ont montré que ceux-ci étaient au moins deux ou trois ordres de grandeur plus lent par rapport à des versions des jeux codés en dur. Les deux raisonneurs les plus rapides qu'ils ont testés utilisent un interpréteur Prolog [5].

Après la définition des termes employés pour décrire les constituants du langage, nous décrivons la manière dont l'interprétation du GDL diffère de celle d'un programme PROLOG et les transformations qui sont nécessaires pour garantir qu'un interpréteur PROLOG donnera des résultats valides. Nous présentons notre implémentation des faits vrais et des coups joués par les joueurs sous la forme de bases de données et nous détaillons notre utilisation du *tabling* pour optimiser la vitesse d'obtention des résultats.

Nous présentons ensuite les techniques que nous avons utilisées pour transformer la description GDL en un circuit d'opérations logiques élémentaires qui permet en général une interprétation rapide des descriptions de jeux [72, 73].

[39] Flux Player, Cadia Player, Java Eclipse, Java Prover, Ggpbase Prover, C++ Reasoner

2.1 Vocabulaire de description des éléments des langages logiques

Nous utilisons ici une notation proche de celle utilisée dans la description du langage PROLOG, que la plupart des lecteurs trouveront plus familière que celle fondée sur les S-expressions (voir 1.3.2.2 page 16).

Un *terme* du premier ordre est soit une *constante*, représentée par un mot écrit en minuscules ; soit une *variable* représentée par une lettre majuscule[40] ; soit une *structure* de la forme $f(t_1, t_2, \cdots, t_n)$ où le *symbole de fonction* f d'arité n (noté $f_{/n}$) est représenté par un mot en minuscules et où les t_i sont des sous-termes.

Un terme est dit *instancié* lorsqu'il ne comporte aucune variable.

Une constante est un cas particulier d'une structure formée avec un symbole de fonction d'arité 0. Suivant l'usage, nous omettons les parenthèses qui entourent une liste vide d'arguments ; il n'en reste pas moins qu'une requête composée seulement d'une constante est traitée par l'interpréteur de la même manière qu'une structure formée d'un symbole de fonction d'arité zéro donc sans arguments.

Un programme logique écrit en GDL est un ensemble de *clauses* de la forme

$$a_0 :\!- a_1, a_2, \cdots, a_n.$$

où $n \geqslant 0$ et les termes a_i sont des *atomes*.

Nous distinguons deux types de clauses : d'une part les *faits* dans le cas où $n = 0$ et pour lesquels le symbole d'implication ':–' peut être omis ; d'autre part les *règles* pour lesquelles $n \geqslant 1$.

Dans le cas des règles, a_0 s'appelle la *conclusion* et les a_i à droite du symbole ':–' s'appellent les *prémisses*.

Nous appelons *procédure* tout ensemble de clauses dont les conclusions utilisent le même symbole de fonction, comme en PROLOG.

Deux procédures sont prédéfinies en GDL : 'distinct' qui est désignée ici par le symbole '≠' en position infixe pour laquelle $t_1 \neq t_2$ est vrai si deux termes t_1 et t_2 obligatoirement instanciés sont différents ; 'not' qui est désignée ici par le symbole '¬' et pour laquelle $\neg t$ est vrai si le terme t obligatoirement instancié n'est pas prouvable par l'interpréteur. PROLOG et GDL n'interprètent pas tout à fait de la même manière ces deux procédures ; ce point est développé aux sections 2.2.2.2 et suivantes.

[40] PROLOG reconnaît n'importe quel mot commençant par une majuscule ou un '_' comme étant une variable mais nous nous limitons aux noms composés d'un seul caractère majuscule.

2.2 Évaluation des règles par PROLOG

En dehors des différences de syntaxes évidentes entre GDL et PROLOG, les deux langages présentent quelques différences de sémantique plus subtiles qu'il faut prendre en compte en traduisant le GDL en PROLOG pour garantir que son interprétation produit des résultats corrects[41]. Certaines structures de GDL peuvent se traduire de plusieurs manières en PROLOG, avec des performances et des difficultés d'implémentation propres.

Nous présentons ici les problèmes posés par le prédicat *or*, l'ordre des prémisses et la négation, la consultation et la mise à jour des fluents, ainsi que l'utilisation de tables de résultats pré-calculés.

2.2.1 Le prédicat `or` du langage GDL

Le prédicat **`or`** était présent dans les premières versions du GDL et les descriptions de jeux l'utilisaient. Il a par la suite été supprimé des spécifications mais ces descriptions anciennes sont toujours présentes sur les serveurs et sont parfois utilisées, y compris dans le cadre formel des compétitions officielles. Nous devons donc garantir que son évaluation est correcte.

Il y a deux manières de procéder pour traduire le **`or`** du GDL en PROLOG : utiliser le ';' de PROLOG ou réécrire les règles.

2.2.1.1 L'utilisation du *ou* de PROLOG

En PROLOG, il y a une procédure permettant d'écrire des disjonctions au sein d'une règle usuellement notée avec un point-virgule (';'), par exemple dans
$menu :- entree, plat, (fromage; dessert)$.

La sémantique est la même en PROLOG qu'en GDL : son utilisation produit le même résultat que celle du *ou* implicite décrit dans la section suivante. En revanche, la complexité en temps peut être différente.

[41] La plupart des descriptions de jeux sont le fruit du travail de programmeurs plus familiers du langage PROLOG que du Datalog qui correspond mieux au GDL ; de ce fait, les descriptions n'utilisent que rarement les idiotismes de GDL qui le différencient de PROLOG et une traduction naïve du GDL conduit souvent à un programme PROLOG qui donne les résultats attendus. Les subtilités décrites dans la section n'apparaissent que lors d'une analyse détaillée de la sémantique des deux langages ou lorsqu'on constate que le programme PROLOG conduit un joueur à jouer des coups illégaux ou insensés. Au quotidien, les détails de la sémantique du GDL sont en pratique principalement déterminés par le consensus des joueurs GGP et des arbitres présents sur les serveurs ; ainsi lors de la compétition annuelle, les membres du groupe de logique de Stanford déterminent en pratique les points litigieux qui apparaissent du fait des interprétations divergentes de la sémantique du GDL.

Le *ou* pose des problèmes spécifiques pour le tabling décrit à la section 2.2.4 : il n'est pas possible de sauvegarder des tables de résultats de *ou* sauf à réécrire chaque *ou* comme une procédure, ce qui reviendrait à effectuer une des transformations décrites à la section suivante.

2.2.1.2 L'utilisation du *ou* implicite de PROLOG

La mécanique de la résolution d'un programme logique conduit l'interpréteur PROLOG à utiliser chacune des clauses d'une procédure jusqu'à obtenir un succès ou énumérer les différentes solutions[42]. Nous pouvons utiliser cette caractéristique pour traduire les **`or`** des descriptions GDL. Cette traduction peut se faire en dupliquant la clause ou bien en définissant deux clauses supplémentaires pour chacune des branches de l'alternative.

Dans la suite de l'exposé, nous utiliserons l'exemple suivant :

$$gain(J) :- domine(J,X,Y),(case(X,Y);colonne(Y))$$

qui permet d'indiquer que le joueur J est dans une position gagnante s'il domine la position XY, soit via la case XY soit via la colonne Y.

Nous pouvons dupliquer entièrement la clause en remplaçant dans chacune des nouvelles clauses l'alternative par une de ses branches : on obtient alors

$$\begin{aligned} gain(J) &:- domine(J,X,Y), case(X,Y) \\ gain(J) &:- domine(J,X,Y), colonne(X) \end{aligned}$$

Cette méthode présente un problème de performance dans certains cas : la procédure *domine* sera calculée deux fois pour énumérer toutes les solutions alors qu'elle ne l'était qu'une seule fois dans la formulation originale[43]. En revanche, il est possible de réordonner différemment les prémisses selon les branches de l'alternative ce qui peut conduire à une amélioration des performances.

Une autre méthode consiste à définir une nouvelle procédure destinée à résoudre uniquement la disjonction. Nous obtenons avec notre exemple :

[42] Un interpréteur PROLOG possède une pile des *et* qui sauvegarde le contexte nécessaire à l'interrogation des prémisses et une pile des *ou* chargée de sauvegarder les points de choix entre les différentes clauses d'un même prédicat. Le prédicat *or* de PROLOG utilise la pile des *ou* de la même manière pour créer un point de choix entre les requêtes qu'il possède en arguments.

[43] Le problème a été mis en évidence dans une description du jeu de Reversi par Alex Landau : le calcul du score conduit à 2^{64} évaluations d'un prédicat. En pratique, les joueurs dont l'interprétation est purement fondée sur la traduction du GDL en PROLOG ne peuvent pas l'effectuer .

$$\begin{aligned} gain(J) &:\!- domine(J,X,Y), aux(X,Y) \\ aux(X,Y) &:\!- case(X,Y) \\ aux(X,Y) &:\!- colonne(X) \end{aligned}$$

Cette réécriture conduit PROLOG à produire les mêmes résultats dans le même ordre que l'utilisation du prédicat **or**[44]. L'appel à un nouveau prédicat entraîne seulement un léger surcoût. Cependant, cette réécriture peut conduire à produire des clauses qui ne respectent pas la spécification GDL que la conclusion ne contienne que des variables instanciées. Ici c'est le cas de la clause $aux(X,Y) :\!- colonne(X)$ puisque la variable Y ne peut pas être instanciée par $colonne(X)$.

2.2.1.3 Le traitement du *ou* par LEJOUEUR

Dans LEJOUEUR nous avons choisi de dupliquer complètement les clauses pour permettre la mise en œuvre de notre procédure d'instanciation qui nécessite que toutes les variables de toutes les clauses soient instanciables.

D'une façon générale, LEJOUEUR met donc en œuvre la réécriture :

$$\begin{aligned} & f(t_0,\cdots,t_m) :\!- a_1,\cdots,a_{k-1},or(b_1,\cdots,b_n),a_{k+1},\cdots,a_r \\ \rightarrow & \begin{cases} f(t_0,\cdots,t_m) :\!- a_1,\cdots,a_{k-1},b_1,a_{k+1},\cdots,a_r \\ \vdots \\ f(t_0,\cdots,t_m) :\!- a_1,\cdots,a_{k-1},b_n,a_{k+1},\cdots,a_r \end{cases} \end{aligned}$$

Par la suite, nous supposerons que les **or** ont été éliminés de cette manière des descriptions du jeu, qui ne l'utilisent donc pas.

2.2.2 Le choix de l'ordre des prémisses

Nous décrivons ici les problèmes que pose l'ordre des prémisses qui apparaissent dans la partie droite des clauses. Cet ordre, indifférent dans la sémantique du GDL, peut conduire à des résultats faux avec une traduction naïve en PROLOG et a un impact sur les performances.

2.2.2.1 La différence entre GDL et PROLOG

La spécification du langage GDL indique qu'il s'agit d'une variante de Datalog [40, page 2]. La différence entre GDL et PROLOG est similaire à celle entre Datalog et PROLOG. Un programme écrit

[44] Un interpréteur PROLOG qui utilise une machine de Warren entrelace une pile des *et* et une pile des *ou*. L'exécution d'une procédure comportant plusieurs clauses conduit l'interpréteur à empiler pour chaque clause le contexte de la même manière que pour l'interprétation d'un **or**.

en Datalog est purement déclaratif alors que ce n'est pas le cas en Prolog. L'ordre des clauses et des prémisses dans les clauses contraignent l'ordre dans lequel Prolog effectue ses opérations alors qu'en Datalog, la description est indépendante de l'algorithme chargé d'en trouver un modèle. Donc une traduction naïve du GDL vers le Prolog ne fonctionne pas toujours. Elle peut entraîner notamment des récursions infinies pour certains ordonnancements des prémisses.

Les auteurs de descriptions GDL utilisent assez souvent une définition d'un opérateur de comparaison arithmétique sous la forme des clauses :

$$succ(0,1)succ(1,2)succ(2,3)\cdots$$
$$less(X,Y) :- succ(X,Y)$$
$$less(X,Y) :- less(X,Z), succ(Z,Y)$$

En effectuant la requête $less(A,B)$, la deuxième clause va tenter d'unifier la première prémisse $less(A,Z)$ avec Z non instanciée, ce qui constitue exactement la même requête qu'initialement au renommage des variables près et conduit donc à une boucle infinie.

Présenté plus loin, le tabling permet d'éviter ces boucles infinies.

2.2.2.2 Le problème du `distinct` avec les variables non instanciées

En Prolog, les prémisses sont évaluées de gauche à droite. Une prémisse qui utilise '$\neq$' avec des variables qui ne sont pas instanciées par les prémisses plus à gauche peut conduire à des erreurs d'interprétation.

Par exemple, le programme suivant évalué par Prolog pose un problème :

$$f(0).$$
$$g(X,Y) :- f(X), X \neq Y, f(Y).$$

Ici, $g(X,Y)$ peut s'unifier avec $g(0,0)$ malgré la prémisse $X \neq Y$. En effet, $f(X)$ s'unifie avec $f(0)$ et unifie donc X avec 0. Ensuite, l'évaluation de $0 \neq Y$ est aussi vraie en utilisant le prédicat '\==' de Prolog car les deux termes ne sont pas strictement identiques et la variable Y reste non instanciée[45]. Enfin, $f(Y)$ s'unifie avec $f(0)$, unifiant Y avec 0.

Le problème peut être corrigé en repoussant $X \neq Y$ vers la droite :

$$f(0).$$
$$g(X,Y) :- f(X), f(Y), X \neq Y.$$

Dans ce cas, l'évaluation de $g(X,Y)$ est bien fausse car lors de l'examen de la prémisse $X \neq Y$, les deux variables X et Y sont instanciées du fait de l'unification des deux prémisses plus à gauche.

[45]La ménagerie des opérateurs de comparaison du langage Prolog ne permet pas de résoudre le problème : soit ils évaluent à vrai et n'instancient rien, soit ils unifient les deux termes et évaluent à faux et ignorent le reste des prémisses.

2.2.2.3 Le problème du `not` avec les variables non instanciées

Un problème similaire se pose lors de l'utilisation de la négation par l'échec. En PROLOG, une manière de l'implémenter utilise la coupure comme dans la définition suivante de '$\neg_{/1}$' :

$$\neg X \text{ :- } execute(X), !, fail.$$
$$\neg X.$$

Ici $execute_{/1}$ exécute la requête passée en argument. En cas d'échec, l'évaluation de cette clause s'arrête et la clause suivante est interprétée, ce qui conduit à évaluer la négation à vrai. En cas de succès, il faut pouvoir renvoyer faux tout en empêchant l'évaluation de la clause suivante en cas de retour arrière. L'opération $!_{/0}$ appelée *coupure* permet d'obtenir ce comportement en élaguant l'arbre de recherche au point d'appel à '$\neg$'. L'évaluation des prémisses se poursuit avec $fail_{/0}$ qui s'évalue toujours à faux.

Comme le succès de la négation n'est obtenu que par la deuxième clause, la négation par l'échec n'instancie aucune variable. Cela peut conduire à des erreurs d'interprétation du GDL comme dans l'exemple suivant :

$$f(0).$$
$$g(1).$$
$$h(X) \text{ :- } \neg f(X), g(X).$$

Dans cet exemple, la requête $h(X)$ ne donnera pas la solution évidente $h(1)$: l'évaluation de $\neg f(X)$ échoue puisque $f(0)$ est vrai et cela fait échouer définitivement la clause.

Le problème peut à nouveau être corrigé en repoussant la négation vers la droite :

$$f(0).$$
$$g(1).$$
$$h(X) \text{ :- } g(X), \neg f(X).$$

Dans ce cas, l'évaluation de $h(X)$ donne bien la solution $h(1)$ puisqu'au moment de l'évaluation de $\neg f(X)$, la variable X est liée à 1 et $f(1)$ est faux.

La solution de repousser vers la droite les négations afin de garantir l'absence de variables non instanciées permet toujours d'obtenir une évaluation correcte de la description GDL en PROLOG. En effet, les spécifications du langage GDL imposent que l'évaluation d'une règle conduise à l'instanciation de toutes les variables de la conclusion. Nous avons donc la garantie que les variables utilisées dans la négation apparaissent au moins une fois dans un terme positif.

2.2.2.4 L'instanciation des variables par `distinct` et `not`

Un **`distinct`** qui instancierait ses opérandes possédant des variables non instanciées avec toutes les valeurs possibles distinctes de ces variables résoudrait le problème mais ruinerait les performances. Ce mécanisme nécessiterait en plus de connaître le domaine des termes instanciés possibles pour ces variables.

De la même façon, le **`not`** pourrait instancier les variables avec toutes les valeurs pour lesquelles la procédure en argument échoue. Là aussi, le coût rend cette façon de procéder sans intérêt.

2.2.2.5 Le réordonnancement des prémisses par LeJoueur

Pour les raisons exposées dans les sections précédentes, nous procédons à une opération de reformulation des règles GDL avant de les faire interpréter par Prolog.

Nous conservons l'ordre des prémisses positives et nous insérons les prémisses négatives à la position la plus à gauche telle que les variables de la prémisse négative apparaissent au moins une fois dans les prémisses précédentes. Cela garantit que les prémisses négatives sont toujours évaluées avec des variables totalement instanciées. Cela a également comme effet de repousser les **`distinct`** vers la gauche ce qui permet de faire échouer rapidement une clause puisque son évaluation est peu coûteuse.

Nous supposons que les rédacteurs de descriptions GDL écrivent leurs règles de manière optimale dans le cas d'une évaluation des prémisses de gauche à droite. Même si cette hypothèse n'est pas vraie pour certains rédacteurs, elle conduit à modifier la description GDL initiale a minima tout en garantissant une évaluation correcte par Prolog.

Nous n'avons pas poussé plus loin l'étude du problème de l'ordre des prémisses pour améliorer la vitesse d'évaluation car celle-ci peut dépendre de l'état du jeu qui est très variable.

2.2.3 La représentation de l'état courant de la partie

Les règles d'une description de jeu en GDL comportent des prémisses qui dépendent de l'état courant de la partie : d'une part celles formées avec $true_{/1}$ et d'autre part celles avec $does_{/2}$. L'état de la partie avant les actions des joueurs est décrite grâce à un ensemble de faits appelés *fluents*. Dans la description GDL, les règles qui consultent l'état courant font appel à une requête formée avec le symbole de fonction $true_{/1}$. De même, les coups choisis par les joueurs sont interrogés par une requête formée sur $does_{/2}$.

2.2.3.1 L'utilisation naïve de la base de données des prédicats dynamiques de PROLOG

Une première approche consiste à ne pas faire de distinction entre d'une part les faits décrivant l'état courant du jeu et les coups joués et d'autre part le reste de la description.

Cela nécessite de pouvoir ajouter et retirer des fluents afin de prendre en compte l'évaluation de l'état de la partie. En PROLOG, `assert`, `retract` ou une de leurs variantes dynamiques permettent de faire ces ajouts et ces retraits[46].

L'inconvénient principal de cette méthode est que la base des faits n'est pas la seule à occuper la base de données ce qui nous oblige à stocker aussi ailleurs cet ensemble de faits afin de pouvoir les retirer un par un à la fin du coup. Ceci est surtout pénalisant lors du calcul d'une partie Monte-Carlo pour lequel il n'est pas nécessaire de stocker toutes les positions intermédiaires.

2.2.3.2 L'utilisation de bases de données auxiliaires

Dans l'interpréteur YAPPROLOG que nous utilisons, nous stockons les fluents dans des bases de données auxiliaires (*recorded databases*) qui sont aussi courantes dans la plupart des implémentations de PROLOG fondées sur la tradition d'Edimbourg. Nous conservons l'état courant de la partie et les coups joués dans deux bases de données spécifiques.

Les procédures $true_{/1}$ et $does_{/2}$ sont définies comme des interrogations de ces bases de données avec le prédicat *recorded* de PROLOG :

$$true(X) :- recorded(dbtrue, X, \_).$$
$$does(X, Y) :- recorded(dbdoes, does(X, Y), \_).$$

où *dbtrue* et *dbdoes* sont des identifiants de ces bases de données.

L'ensemble des fluents calculés pour l'état suivant est placé dans une troisième base de données *dbnext* avec :

$$forall(X, next(X), recordzifnot(dbnext, X, \_)).$$

qui est basculée dans le base *dbtrue* pour faire de l'état suivant l'état courant. Le prédicat PROLOG *recordzifnot* permet d'ajouter des faits à une base de donnée en éliminant les doublons.

L'interpréteur YAPPROLOG permet de compiler un programme pour qu'il soit ensuite exécuté sur une version dérivée de la *machine abstraite de Warren*. L'exécution d'une version compilée permet un gain de performance par rapport à une simple interprétation sans compilation juste à temps. Il convient donc de compiler la description GDL tout en gardant la possibilité de modifier l'état du jeu.

[46] Il existe des variantes statiques du prédicat `assert` : un fait ou une règle est ajouté de façon définitive. Nous ne pouvons plus le retirer avec `retract`. Cela permet à l'interpréteur d'optimiser l'évaluation des règles, en les compilant.

L'état du jeu n'est pas compilé mais stocké dans deux bases de données nommées *dbtrue* et *dbdoes* dans lesquelles sont stockés les fluents correspondant à la position courante du jeu et des coups des jours.

2.2.4 L'utilisation du tabling

Afin de réduire le temps d'évaluation par PROLOG, nous pouvons réduire une partie des unifications redondantes grâce à la technique du tabling. Cette technique est implémentée dans l'interpréteur YAPPROLOG que nous utilisons qui utilise la résolution SLG. Le tabling consiste à stocker les faits prouvés dans une table et évite ainsi de les calculer plusieurs fois. Une explication de cette technique est présentée à la section 5.2.2.1. Cependant, pré-calculer certains faits ne vaut que pour ceux dont la valeur de vérité ne va pas changer en fonction de la position courante du jeu ou des coups des joueurs.

Nous avons donc besoin de faire la distinction entre deux types de procédures : les procédures statiques et les procédures dynamiques. Une procédure est dynamique si le symbole de fonction de ses conclusions appartient à l'ensemble

$$\mathcal{D}_{\text{init}} = \{true_{/2}, does_{/2}, terminal_{/0}, legal_{/2}, sees_{/2}, goal_{/2}, next_{/1}\}$$

ou si elle possède une règle dans laquelle une prémisse utilise le symbole de fonction d'une procédure dynamique (éventuellement à l'intérieur d'une négation). Les procédures statiques sont toutes celles qui ne sont pas dynamiques.

Pour cela, nous construisons un graphe des dépendances inversées où les procédures constituent l'ensemble des sommets et qui possède un arc de p_1 vers p_2 s'il existe une règle dans laquelle la conclusion utilise le symbole de fonction de p_1 et une prémisse utilise le symbole de fonction de p_2 (éventuellement à l'intérieur d'une négation). Un parcours en largeur à partir des procédures de $\mathcal{D}_{\text{init}}$ permet de visiter l'ensemble des procédures dynamiques.

Lors de l'évaluation, nous n'appliquons le tabling que sur les procédures statiques ce qui évitera les calculs redondants pour celles-ci. La pénalité en temps engendrée par la création des tables est largement compensée par la consultation rapide des tables. Nous n'utilisons pas le tabling sur les procédures dynamiques car pour que l'évaluation soit correcte, nous devrions vider les tables à chaque changement de position de jeu. Il y a néanmoins un cas où le tabling est nécessaire sur une procédure dynamique : lorsqu'un ordre particulier des prémisses conduit à des évaluations répétées de la même requête qui ne permet pas à l'interpréteur de renvoyer sa réponse dans des temps compatibles avec le GGP comme c'est le cas par exemple pour le calcul des scores dans la description du jeu REVERSI.

2.3 Évaluation rapide par circuit

La vitesse d'évaluation des règles par PROLOG est limitée par les nombreuses unifications qu'il met en œuvre. Par ailleurs, pour deux positions distinctes du jeu, il y a des évaluations qui sont sans cesse recalculées. Afin de réduire une partie des calculs redondants mais aussi afin de permettre une analyse statique de la description en vue de son optimisation, nous transformons le programme logique en un circuit logique. Cette transformation passe par les étapes suivantes : correction de la description GDL ; instanciation des règles ; construction du circuit ; optimisation du circuit. L'étape de correction de la description GDL a déjà été détaillée à la section 2.2. Nous allons donc à présent décrire les opérations suivantes puis nous expliquerons comment nous procédons à l'évaluation avec ce circuit.

2.3.1 Instanciation des règles

L'étape d'instanciation des règles consiste à les substituer par d'autres règles dans lesquelles les variables sont remplacées par les valeurs instanciées qu'elles pourraient prendre. Le but est d'obtenir une description GDL dans le formalisme du calcul propositionnel à partir de la description initiale en logique du premier ordre. Cette tâche est aussi assurée par un interpréteur PROLOG.

La technique d'instanciation que nous utilisons consiste à réécrire la description GDL en un nouveau programme capable de générer toutes les règles instanciées. Nous désignons par $\mathcal{P}$ l'ensemble des procédures de la description GDL initiale et par $\mathcal{S}$ et $\mathcal{D}$ celles qui sont respectivement statiques et dynamiques. Par extension, on écrira aussi $t \in \mathcal{S}$ (resp. $t \in \mathcal{D}$) si le terme t (dont on aura supprimé l'éventuelle négation) possède un symbole de fonction de procédure statique (resp. dynamique).

La transformation de la description initiale en programme d'instanciation fait appel à un ensemble de réécritures élémentaires des règles que nous définissons ci-dessous. Lorsque les conditions requises pour effectuer la réécriture ne sont pas vérifiées, la transformation laisse la règle inchangée.

2.3.1.1 Formatage des règles instanciées

Les transformations R_1 et R_2 servent uniquement à déterminer la forme des règles qui seront sauvegardées dans le programme instancié. Leur valeur de vérité ne sera donc pas calculée par le programme d'instanciation. Il s'agit simplement de réécritures. Une fois que l'instanciation est réalisée, les requêtes sur les prédicats statiques ont une valeur de vérité constante. Il n'est donc pas nécessaire de les conserver.

La transformations R_1 permet l'élimination du premier appel à une procédure statique, y compris lorsqu'elle apparaît dans une négation :

$$\begin{array}{rll} R_1 : & \text{si} & a_k \in \mathcal{S} \text{ et } \forall i \in [\![1, k-1]\!], a_i \in \mathcal{D} \\ & & a_0 \coloneq a_1, \cdots, a_{k-1}, a_k, a_{k+1}, \cdots, a_n \\ & \rightarrow & a_0 \coloneq a_1, \cdots, a_{k-1}, a_{k+1}, \cdots, a_n \end{array}$$

Cette règle sert à éliminer tous les appels aux procédures statiques lors de la sauvegarde des règles du programme instancié. En effet, étant statiques, elles sont toujours vraies quel que soit l'état courant du jeu et donc aucune vérification n'est nécessaire. Cela évite de faire un traitement supplémentaire sur le programme instancié qui consisterait à éliminer ces termes toujours vrais. Dans le cas d'une négation de procédure statique, il est également inutile de vérifier que cette négation est toujours vraie.

La transformation R_2 permet d'éliminer le premier test d'inégalité dans un règle :

$$\begin{array}{rll} R_2 : & \text{si} & \forall i \in [\![1, k-1]\!], \nexists X, Y : a_i = X \neq Y \\ & & a_0 \coloneq a_1, \cdots, a_{k-1}, A \neq B, a_{k+1}, \cdots, a_n \\ & \rightarrow & a_0 \coloneq a_1, \cdots, a_{k-1}, a_{k+1}, \cdots, a_n \end{array}$$

Comme précédemment, il est inutile de conserver les tests d'inégalité dans le programme instancié car les variables seraient remplacées par des termes instanciés distincts et ce test serait alors toujours vrai. Pour une règle de réécriture $R \in \{R_1, R_2\}$, l'application en série de la règle $R \circ \cdots \circ R$ jusqu'à l'obtention d'un point fixe est notée $R^\star$. Pour chaque règle r du programme initial, $R_2(R_1(r))$ est la forme de la règle qui sera sauvegardée dans le programme instancié, débarrassée des tests inutiles.

2.3.1.2 Modification des règles pour forcer les instanciations

Contrairement aux réécritures précédentes qui n'ont aucun impact sur l'ensemble des faits prouvés, les transformations suivantes sont celles qui permettent vraiment d'instancier la description GDL initiale. La transformation T_1, permet d'éliminer la première négation d'un appel à une procédure dynamique :

$$\begin{array}{rll} T_1 : & \text{si} & a_k \in \mathcal{D} \text{ et } \forall i \in [\![1, k-1]\!], a_i \in \mathcal{S} \text{ ou } \nexists a \in \mathcal{D} : a_i = \neg a \\ & & a_0 \coloneq a_1, \cdots, a_{k-1}, \neg a_k, a_{k+1}, \cdots, a_n \\ & \rightarrow & a_0 \coloneq a_1, \cdots, a_{k-1}, \mathit{instance\_not}(a_k), a_{k+1}, \cdots, a_n \end{array}$$

où le prédicat $\mathit{instance\_not}(T)$ est défini ainsi :

$$\mathit{instance\_not}(T) \coloneq \mathit{execute}(T); \mathit{succeed}.$$

avec le prédicat $execute_{/1}$ qui évalue la requête passée en argument et le prédicat *succeed* qui est toujours évalué à vrai.

En effet, les procédures dynamiques dépendent de l'état courant du jeu. Ce dernier étant variable et inconnu à ce stade, il faut prendre en compte toutes les possibilités. Nous considérons donc que le fait en argument de la négation peut être faux, cette vérification étant différée au moment où nous évaluerons les règles instanciées car la règle qui sera sauvegardée dans le programme instancié conservera cette négation. En remplaçant $\neg a_k$ par $instance\_not(a_k)$, le domaine des termes pouvant être en argument de cette négation est exploré en totalité.

Pour les deux transformation suivantes, nous avons le choix entre utiliser $input_{/2}$ et $base_{/1}$: ce sont les transformations T_2 et T_3 ou de ne pas utiliser $input_{/2}$ et $base_{/1}$: ce sont les transformations T'_2 et T'_3.

Les transformations T_2 et T_3 transforment le symbole de fonction $true_1$ en $base_1$ ainsi que le symbole de fonction $does_2$ en $input_2$.

$$\begin{aligned} T_2 : \quad & true(a,b) :\!- \, a_1, \cdots, a_n \\ \rightarrow \quad & base(a,b) :\!- \, a_1, \cdots, a_n \end{aligned}$$

$$\begin{aligned} T_3 : \quad & legal(a,b) :\!- \, a_1, \cdots, a_n \\ \rightarrow \quad & input(a,b) :\!- \, a_1, \cdots, a_n \end{aligned}$$

Les transformations T'_2 et T'_3 transforment les symboles de fonction $init_1$ et $next_1$ en $true_1$ ainsi que le symbole de fonction $legal_2$ en $does_2$.

$$\begin{aligned} T'_2 : \quad \text{si} \quad & f \in \{init_{/1}, next_{/1}\} \\ & f(a) :\!- \, a_1, \cdots, a_n \\ \rightarrow \quad & true(a) :\!- \, a_1, \cdots, a_n \end{aligned}$$

$$\begin{aligned} T'_3 : \quad & legal(a,b) :\!- \, a_1, \cdots, a_n \\ \rightarrow \quad & does(a,b) :\!- \, a_1, \cdots, a_n \end{aligned}$$

Ces deux dernières transformations permettent d'obtenir, à partir de l'état du jeu initial défini par la procédure $init_{/1}$, l'ensemble des faits pouvant apparaître dans une position du jeu et l'ensemble des coups possibles autorisés dans toutes les configurations possibles d'une partie. Cette méthode n'utilise pas les prédicats $base_{/1}$ et $input_{/2}$ qui permettraient d'obtenir directement tous les faits et les coups possibles car ils ne font pas partie de la spécification GDL officielle.

La dernière transformation S prend en entrée une expression x et une règle et crée deux règles. Dans la première, on renomme le symbole de fonction de conclusion et on ajoute le terme x en paramètre supplémentaire. La seconde règle permet de relier l'ancienne procédure à celle nouvel-

lement créée.

$$S : \quad \text{si } f \in \mathcal{D} \quad x; f(t_0, \cdots, t_m) \text{ :- } a_1, \cdots, a_n \rightarrow \begin{cases} f_k(t_0, \cdots, t_m, x) \text{ :- } a_1, \cdots, a_n, store(x) \\ f(t_0, \cdots, t_m) \text{ :- } f_k(t_0, \cdots, t_m, x). \end{cases}$$

où k est le premier entier naturel non utilisé lors un renommage du symbole de fonction f et x est le terme qui détermine la forme que nous souhaitons donner à la règle instancié. La procédure $store_{/1}$ est toujours vraie et a pour effet de bord d'enregistrer son argument dans une base de données.

L'ajout du paramètre x à la procédure nouvellement introduite sert aussi à forcer le tabling à rechercher de nouvelles solutions dans le cas où cette procédure est appelée avec des variables $t_0, \cdots, t_m$ déjà totalement instanciées. Par exemple, la règle : $goal(robot, 100)$:- $true(victoire(X))$ lorsqu'elle est tablée, n'a besoin que d'une seule preuve de $true(victoire(X))$ pour fermer ensuite la table car le fait $goal(robot, 100)$ est prouvé. Si $victoire(nord)$ et $victoire(sud)$ sont prouvables, nous n'obtenons alors qu'une seule des deux règles instanciées, par exemple $goal(robot, 100)$:- $true(victoire(nord))$: l'ajout du paramètre x garantit que $goal(robot, 100)$:- $true(victoire(sud))$ sera ajouté à la table lui aussi.

2.3.1.3 Programme d'instanciation

Pour obtenir le nouveau programme GDL que l'on va utiliser pour l'instanciation, nous combinons l'ensemble des transformations que nous venons de décrire. La transformation $T(r) = T_3(T_2(T_1^\star(r)))$ détermine la forme que doivent prendre les instanciations de la règle r dans le programme instancié. La transformation $R(r) = R_2^\star(R_1^\star(r))$ permet d'obtenir un programme logique dont le modèle contient toutes les instanciations possibles de toutes les variables du programme initial. La transformation S permet d'ajouter un effet de bord aux règles $R(r)$ pour récupérer les instanciations et d'éviter que le tabling ignore les variantes des règles ayant la même conclusion instanciée.

Pour obtenir le nouveau programme logique que nous utilisons pour l'instanciation, nous appliquons donc sur chaque règle de la description GDL initiale la transformation

$$G : r \rightarrow S\ (R(r), T(r)))$$

La figure 2.1 montre l'effet de la transformation G sur trois exemples de règles issues du jeu CONNECT5. Dans le premier exemple, la règle est déjà complètement instanciée. Nous voyons que la règle à sauvegarder W est bien identique à la règle initiale. En revanche, la règle d'instanciation ne vérifie plus la négation du terme dynamique $line\_of\_5_{/0}$ car il pourrait être vrai dans certains

r	$G(r)$
$goal(x,50) :- \neg line\_of\_5$	$W = (goal(x,50) :- \neg line\_of\_5)$ $goal_1(x,50,W) :- store(W)$ $goal(x,50) :- goal_1(x,50,W)$
$legal(R,noop) :- role(R), \neg true(ctrl(R))$	$W = (legal(R,noop) :- \neg true(ctrl(R)))$ $legal_2(R,noop)) :- role(R), store(W)$ $legal(R,noop) :- legal_2(R,noop,W)$
$next(cell(X,Y,R)) :- does(R,mark(X,Y))$	$W = (next(cell(X,Y,R)) :- does(R,mark(X,Y)))$ $next_3(cell(X,Y,R),W) :- legal(R,mark(X,Y)), store(W)$ $next(cell(X,Y,R)) :- next_3(cell(X,Y,R),W)$

FIGURE 2.1 – Exemple de transformation de trois règles de la description du jeu CONNECT5. À gauche se trouve la règle initiale et à droite les règles utilisées dans le programme d'instanciation.

états du jeu et faux dans d'autres. Dans le deuxième exemple, la règle fait appel à la procédure statique $role_{/1}$ et à la procédure dynamique $true_{/1}$. Nous constatons que dans la règle instanciée à sauvegarder, $role(R)$ a été éliminée car une fois R instancié, ce fait est toujours vrai quel que soit l'état du jeu. Dans la règle d'instanciation, c'est l'appel à une négation de la procédure dynamique $true_{/1}$ qui a été éliminé pour la même raison que dans le premier exemple. Enfin le troisième exemple illustre le renommage de *does* en *legal* dans la règle d'instanciation, alors qu'il est conservé dans la règle à sauvegarder W.

Nous activons le tabling sur les règles où la conclusion s'écrit avec un nouveau symbole de fonction p_k ainsi que sur les procédures statiques. Nous générons ensuite toutes les solutions des buts suivants :

$$init(X); terminal; role(X), legal(X,Y); role(X), goal(X,Y); next(X)$$

L'ensemble des règles qui ont été sauvegardées par la procédure *store* constituent un programme GDL totalement instancié équivalent à la description initiale après l'ajout des faits du prédicat *role* dans le même ordre que dans la description initiale car cet ordre est utilisé dans le protocole de transmission des coups des joueurs en GGP.

Une évaluation dans la nouvelle description fournit exactement les mêmes résultats que dans l'ancienne. Il est ainsi possible d'utiliser cette description dans PROLOG, cependant le nombre de règles étant significativement plus grand que dans la description initiale, l'évaluation peut s'avérer plus lente.

2.3.2 Construction et simplification du graphe des règles

2.3.2.1 Définition du graphe des règles

Nous représentons le programme instancié par un graphe orienté. Nous définissons dans un premier temps les sommets et un étiquetage de ceux-ci, ensuite nous définissons les arcs entre les sommets.

L'ensemble des sommets sont les atomes, les négations d'atomes et les règles. Les sommets n'ont pas tous les mêmes rôles dans l'interprétation du programme. Une règle est un sommet qui correspond à une conjonction de ses prémisses. La conclusion d'une règle correspond à une disjonction de toutes les règles ayant la même conclusion. Les prémisses du type $\neg a$ correspondent à une négation. Pour les distinguer, nous introduisons une fonction *type* qui à chacun de ces sommets associe une étiquette Et, Ou et Non. Nous n'avons pas précisé d'étiquette pour les prémisses car soit elles apparaissent en conclusion d'une règle du programme et sont étiquetées par Ou, soit elles ne proviennent d'aucun calcul comme l'ensemble des faits définis par le prédicat *true*.

Par exemple, la règle instanciée :

$$goal(x, 50) :- \neg line\_of\_5$$

fait référence à quatre sommets : $goal(x, 50)$ étiqueté Ou ; $line\_of\_5$; $\neg line\_of\_5$ étiqueté Non ; la règle entière étiquetée Et. Deux sommets a et b du graphe sont reliés par un arc $a \rightarrow b$ dans trois cas :

1. si b est une règle et a est une prémisse de cette règle ;
2. si b est un atome conclusion et a est une règle ayant cette conclusion ;
3. si b est une négation et a est l'atome nié.

Le graphe étant ainsi complètement défini, nous introduisons une autre fonction d'étiquetage des sommets *atome* qui associe à chaque sommet l'atome dont il est issu uniquement si cet atome possède un symbole de fonction parmi *true*, *does*, *terminal*, *legal*, *next* et *goal*. La plupart des sommets du graphe n'ont pas d'étiquette *atome*.

Avec ce graphe et les deux étiquetages, il est possible de générer une description GDL équivalente à la description GDL initiale au sens où elle décrit exactement le même jeu. Pour cela, nous prolongeons la fonction *atome* en associant une constante distincte à chaque sommet puis nous générons une ou plusieurs règles pour chaque sommet étiqueté par Et, Ou et Non qui auront pour conclusion son étiquette *atome*. Un sommet étiqueté Et ou Non générera une unique règle dont les prémisses sont les étiquettes *atome* de ses prédécesseurs. Un sommet Ou générera autant de règles qu'il possède de prédécesseurs, chaque règle ayant pour unique prémisse l'étiquette *atome* de ce prédécesseur.

Le graphe est donc une description du jeu équivalente au GDL. Les étapes suivantes transforment ce graphe tout en conservant cette équivalence.

2.3.2.2 Simplification du graphe

Le graphe décrit précédemment modélise aussi un circuit logique. Nous avons représenté à la figure 2.2 le circuit logique équivalent à l'instanciation de la description GDL du jeu ButtonsAndLights présentée à la figure 2.3. En l'analysant, nous pouvons détecter les sommets inutiles ou redondants et réaliser des simplifications logiques. Chaque altération que nous décrivons transforme le graphe en un graphe équivalent, au sens où il peut être converti en un programme GDL instancié décrivant exactement le même jeu. En particulier, les sommets possédant une étiquette *atome* doivent toujours être présents après une altération du graphe.

Sommet inutile Tout sommet non étiqueté par *atome* n'ayant pas de successeur est supprimé du graphe avec ses arcs incidents, puisque sa valeur de vérité n'est utilisée dans le calcul d'aucune des sorties du graphe.

Sommet redondant Tout sommet s étiqueté Et ou Ou n'ayant qu'un seul prédécesseur p est redondant et supprimé du graphe en remplaçant les arcs $s \rightarrow x$ par des arcs $p \rightarrow x$ à condition que s ne possède pas d'étiquette *atome*. Si s a cette étiquette et que p n'en a pas, alors p récupère l'étiquette et s est supprimé.

Sommet vrai et faux On appelle *sommet vrai* tout sommet étiqueté par Et et n'ayant pas de prédécesseur. On appelle *sommet faux* tout sommet étiqueté par Ou et n'ayant pas de prédécesseur. Ces sommets peuvent être générés par l'instanciation ou apparaître à la suite d'une transformation du graphe.

Simplification avec les sommets vrais et faux Nous supprimons tous les arcs reliant un sommet *vrai* à un sommet de type Et sans étiquette atome ou reliant un sommet *faux* à un sommet de type Ou sans étiquette atome. Lorsqu'un arc relie un sommet *vrai* à un sommet de type Non, ce dernier est remplacé par un sommet *faux* et l'arc est supprimé. De même, lorsqu'un arc relie un sommet *faux* à un sommet de type Non, ce dernier est remplacé par un sommet *vrai* et l'arc est supprimé.

Sommets équivalents Deux sommets s et t sont équivalents s'ils ont le même *type* et les mêmes prédécesseurs. Si un des deux ne possède pas d'étiquette *atome* et supposons que c'est s, alors s est supprimé et les arcs $s \rightarrow x$ sont remplacés par des arcs $t \rightarrow x$. Dans le cas où les deux sommets

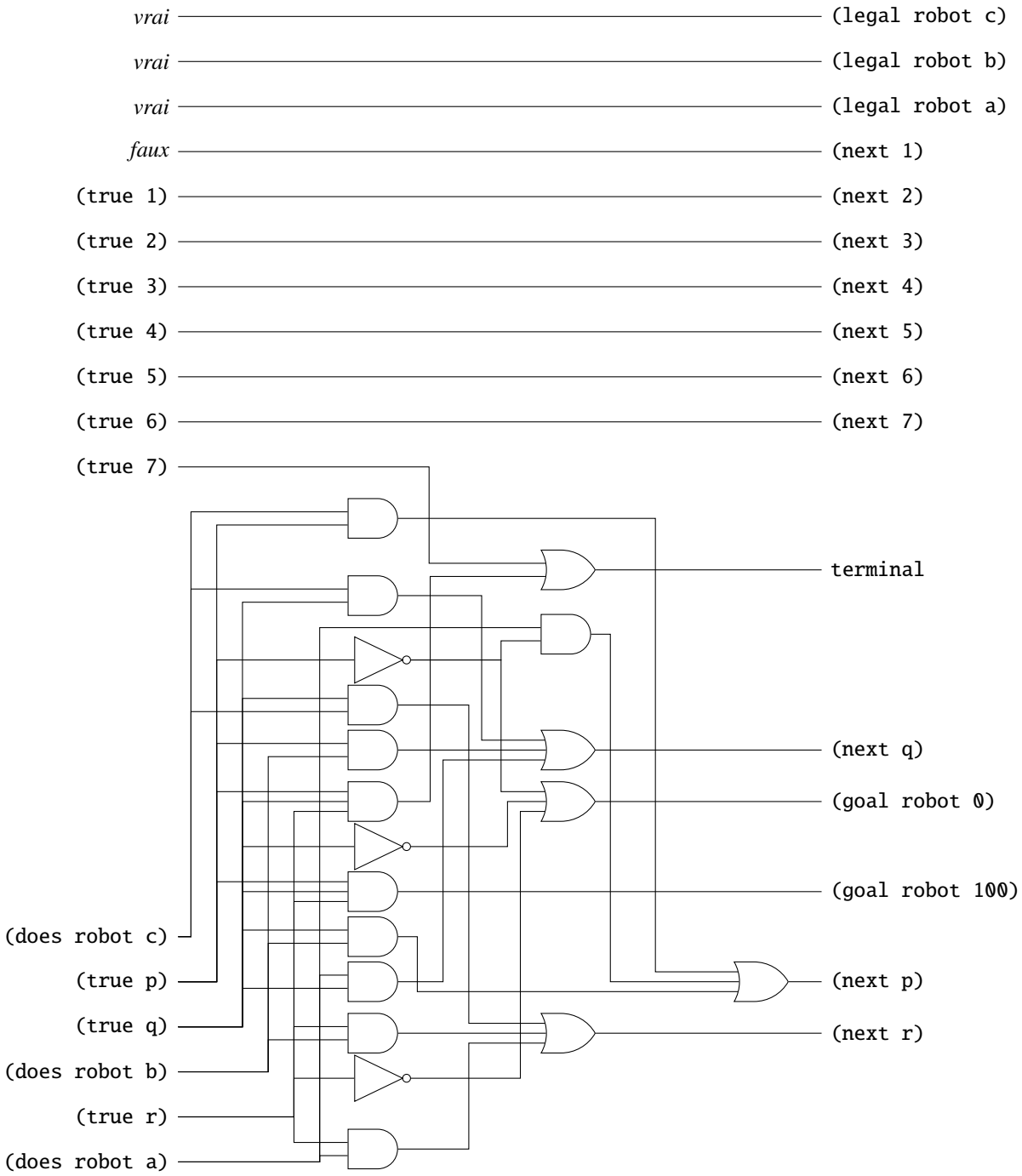


FIGURE 2.2 – Circuit logique du jeu BUTTONSANDLIGHTS généré à partir de la description GDL instanciée. Les fils qui se croisent en formant un '+' ne sont pas connectés.

```
; Components
(role robot)

(base p)
(base q)
(base r)
(base 1)
(base 2)
(base 3)
(base 4)
(base 5)
(base 6)
(base 7)

(input robot a)
(input robot b)
(input robot c)

; init
(init 1)

; legal
(legal robot a)
(legal robot b)
(legal robot c)

; next
(<= (next p)
  (does robot a)
  (not (true p)))
(<= (next p) (does robot b) (true q))
(<= (next p) (does robot c) (true p))

(<= (next q) (does robot a) (true q))
(<= (next q) (does robot b) (true p))
(<= (next q) (does robot c) (true q))

(<= (next r) (does robot a) (true r))
(<= (next r) (does robot b) (true r))
(<= (next r) (does robot c) (true q))

(<= (next ?y)
  (true ?x)
  (successor ?x ?y))

; goal
(<= (goal robot 100)
  (true p)
  (true q)
  (true r))
(<= (goal robot 0) (not (true p)))
(<= (goal robot 0) (not (true q)))
(<= (goal robot 0) (not (true r)))

; terminal
(<= terminal (true p) (true q) (true r
    ))
(<= terminal (true 7))

; Data
(successor 1 2)
(successor 2 3)
(successor 3 4)
(successor 4 5)
(successor 5 6)
(successor 6 7)
```

FIGURE 2.3 – Description du jeu BUTTONSANDLIGHTS en GDL.

possèdent une étiquette *atome*, alors s devient un sommet ayant pour unique prédécesseur t et son étiquette *type* devient Ou[47].

Les simplifications précédentes ont toujours pour conséquence de diminuer le nombre de sommets et d'arcs du graphe. Ça n'est pas forcément le cas pour celles que nous allons décrire à présent et qui reposent sur les propriétés d'une l'algèbre de Boole $\{B, \wedge, \vee, \neg\}$ où $\wedge$,$\vee$ et $\neg$ sont respectivement les opérations *et*, *ou* et *non*. Cependant, ces transformations peuvent être utiles lorsqu'elles font apparaître des sommets équivalents qui réduisent la taille du graphe. Dans l'algèbre de Boole B, le *ou* et le *et* jouent des rôles symétriques. Il en est de même pour nos transformations qu'elles portent sur un sommet Et ou un sommet Ou. Nous présentons donc une seule des deux alternatives sachant qu'il suffit de remplacer Et par Ou et Ou par Et pour obtenir les altérations équivalentes.

Factorisation Le *et* est distributif par rapport au *ou* :

$$(a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c).$$

Cette propriété permet de développer ou factoriser les expressions logiques.

Dans le graphe, les sommets Et et Ou peuvent avoir plus de deux prédécesseurs. De manière plus générale, on a donc :

$$(a \wedge Q_1) \vee ... \vee (a \wedge Q_n) \vee R = a \wedge (Q_1 \vee ... \vee Q_n) \vee R$$

où les Q_i sont des conjonctions et R est une disjonction. En notant $P_i = a \wedge Q_i$, $T = Q_1 \vee ... \vee Q_n$ et $U = a \wedge T$, on a :

$$P_1 \vee ... \vee P_n \vee R = U \vee R.$$

Cela nous permet de définir la factorisation suivante sur le graphe. Soit s un sommet Ou et $p_1, ..., p_n$ ses prédécesseurs étiquetés Et. S'il existe un sommet a prédécesseur de chaque p_i, alors :

1. ajouter n sommets $q_1, ..., q_n$ étiquetés Et
2. pour chaque p_i, changer les arcs $x \rightarrow p_i$ en $x \rightarrow q_i$ pour tout $x \neq a$
3. ajouter les arcs $q_i \rightarrow p_i$
4. supprimer les arcs $p_i \rightarrow s$
5. ajouter un sommet t étiqueté Ou
6. ajouter les arcs $q_i \rightarrow t$
7. ajouter un sommet u étiqueté Et
8. ajouter les arcs $a \rightarrow u$, $t \rightarrow u$ et $u \rightarrow s$.

[47] Son étiquette pourrait être aussi Et car il ne possède qu'un seul prédécesseur.

Après cette transformation, il y a $n + 2$ sommets et $n + 3$ arcs supplémentaires. Dans le meilleur des cas, si les sommets p_i modifiés sont devenus inutiles et sont supprimés, le bilan est alors de 2 sommets supplémentaires et $n-3$ arcs en moins. Pour s'assurer d'être dans ce cas, nous limitons la factorisation aux sommets p_i n'ayant qu'un seul successeur. Si de plus l'opération a créé des sommets équivalents ou redondants, ils pourront aussi être supprimés et réduire encore la taille du graphe.

Fusion La transformation suivante utilise l'associativité du *et*. Si un sommet s sans étiquette *atome* est étiqueté par ET et possède un unique successeur t étiqueté lui aussi par ET alors :

1. changer tous les arcs $x \to s$ en $x \to t$
2. supprimer l'arc $s \to t$
3. supprimer le sommet s.

Le bilan de cette opération est d'un sommet et un arc en moins.

Double négation La transformation suivante utilise l'involutivité du *non*. Si un sommet s possède un prédécesseur q tous deux étiquetés par NON, alors :

1. remplacer l'arc $q \to s$ par l'arc $p \to s$ où p est le prédécesseur de q
2. changer l'étiquette de s en OU

Le bilan de cette opération est nul. Cependant, si s ou q devient inutile, il est alors supprimé.

Deux prédécesseurs au maximum Nous ajoutons une dernière opération que nous n'effectuons qu'une fois le graphe simplifié. Elle sert à transformer le graphe de sorte que les sommets étiquetés ET et OU n'aient que deux prédécesseurs au maximum. Si un sommet s étiqueté ET possède plus de deux prédécesseurs, alors :

1. diviser l'ensemble des prédécesseurs en deux ensembles P_1 et P_2 de tailles égales à un près
2. créer deux sommets p_1 et p_2 étiquetés ET
3. pour chaque p_i dans P_j, changer les arcs $x \to s$ en $x \to p_j$
4. ajouter les arcs $p_j \to s$.

Le bilan de cette opération est de deux sommets et deux arcs supplémentaires lorsque ces nouveaux sommets ne sont pas équivalents à des sommets déjà présents dans le graphe.

Implémentation pour découvrir les sommets équivalents Le graphe est implémenté par l'ensemble de ses sommets et des listes d'adjacence. Chaque sommet contient les informations suivantes : son étiquette *type*, son étiquette *atome*, l'ensemble des références à ses prédécesseurs et l'ensemble des références à ses successeurs. Des références à tous les sommets du graphe sont stockées dans une table de hachage. Deux sommets sont considérés comme identiques pour la table s'ils ont les mêmes prédécesseurs, le même type, la même étiquette atome mais pas forcément les mêmes successeurs. Les doublons sont traités lors de l'insertion dans la table. Lors de l'insertion d'un sommet, si un sommet équivalent occupe déjà une entrée dans la table, les successeurs du sommet à insérer sont ajoutés à celui déjà présent. Selon le choix qui est fait pour la fonction de hachage, une telle insertion peut nécessiter un rehachage de certains sommets. Nous discutons dans les paragraphes suivants le choix de la fonction de hachage et ses conséquences en terme de rehachage de sommets.

Si nous souhaitons créer d'une fonction de hachage qui ne dépend que des valeurs de hachage des prédécesseurs, du type de sommet et qui prend les mêmes valeurs sur des sommets codant la même fonction logique alors nécessairement, celle-ci doit prendre ses valeurs dans une algèbre de Boole. Nous calculons alors la valeur de hachage d'un sommet en effectuant sur les valeurs de hachage de ses prédécesseurs l'opération correspondante à son étiquette ET, OU ou NON. Ainsi deux sommets équivalents, du point de vue de la fonction logique qu'ils calculent, auront toujours la même valeur de hachage. Une telle connaissance peut orienter nos optimisations, étant donné que pour deux sommets ayant des valeurs de hachage identiques, il est possible qu'ils soient équivalents.

Par exemple si les valeurs de hachage sont des mots de 64 bits, les valeurs de hachage des sommets sans prédécesseur peuvent être tirées aléatoirement et nous pouvons utiliser les opérations logiques *et*, *ou* et *non* bit à bit. Cependant, une telle fonction de hachage est assez pauvre car les conjonctions successives font passer peu à peu tous les bits à 0, alors que les disjonctions les font passer à 1, la négation ne faisant qu'inverser la proportion de 1 et de 0. Le nombre de collisions est élevé pour des graphes de grande taille[48]. La seule solution dans ce cas est d'augmenter la taille du mot. À l'extrême, nous pouvons obtenir un hachage parfait en ayant des valeurs de hachage qui représentent de manière unique l'opération logique calculée. Par exemple, la valeur de hachage pourrait être la table de vérité sans variable inutile à partir des entrées du circuit. Cela reviendrait à avoir des mots d'une taille de 2^n bits où n est le nombre d'entrées du circuit pour la fonction précédente à la place de mots de 64 bits. Une telle fonction de hachage serait trop lourde à calculer du fait que la complexité en mémoire d'une valeur de hachage et sa complexité en temps de calcul sont exponentielles en fonction du nombres de fluents.

[48] Avec 64 bis, il n'est possible de distinguer que l'ensemble des tables de vérités des circuits à 6 entrées. L'immense majorité des circuits générés à partir de jeux décrits en GDL comportent davantage d'entrées.

Si nous n'imposons plus la contrainte que la fonction de hachage prend des valeurs identiques pour des sommets codant la même fonction logique alors celle-ci peut simplement dépendre des prédécesseurs (et pas de leur valeur de hachage) et du type de sommet. Pour ce type de fonction, chaque remplacement d'un sommet par un autre équivalent pourrait nécessiter un re-hachage de tous les successeurs.

Afin de limiter cet effet, nous pouvons utiliser comme fonction de hachage une combinaison des valeurs des emplacements en mémoire des prédécesseurs. La table de hachage ne contient pas directement les sommets mais des références vers les sommets qui sont eux stockés à des adresses stables en mémoire.

Les opérations de transformation du graphe décrites à la section 2.3.2.2 peuvent modifier des sommets existants. Dans ce cas, il faut sortir leur référence de la table de hachage, les modifier puis les réinsérer.

L'insertion d'une référence à un sommet s dans la table se fait de la manière suivante : si un sommet équivalent x existe déjà, alors les références aux successeurs t_i de s sont retirées de la table de hachage et ajoutées à une file d'attente pour une réinsertion ultérieure ; les références aux t_i sont ajoutées à x et nous modifions les prédécesseurs des t_i en remplaçant la référence vers s par une référence vers x[49]. Le processus est poursuivi jusqu'à l'épuisement de la file d'attente. Par cette méthode, les valeurs de la fonction de hachage restent assez stables pendant les opérations d'optimisation.

Pour limiter davantage ces modifications de la table de hachage, nous avons choisi dans le LeJoueur de traiter les sommets équivalents par lots. Les factorisations sont effectuées par un algorithme glouton (voir l'algorithme 2.1). Nous classons les sommets par ordre décroissant du nombre de prédécesseurs factorisables puis nous factorisons chacun de ces sommets en mettant dans une file d'attente les sommets ajoutés ou modifiés et en supprimant le cas échéant leurs références dans la table de hachage. Une fois que tous les sommets sont traités, nous mettons à jour la table de hachage en insérant les références aux sommets dans la file d'attente.

2.3.2.3 Partitionnement du graphe en fonction des prédicats

Le graphe décrit précédemment modélise un circuit logique.[50] Ses entrées sont les sommets sans prédécesseurs et les sorties sont les sommets sans successeurs. Nous voulons évaluer le circuit logique de sorte que la séquence d'opérations à effectuer ne dépende pas de l'état courant du

[49] Les rôles de s et x peuvent être inversés en ne gardant dans la table de hachage que celui des deux qui a le moins de successeurs. Cependant il n'y a pas de garantie que ça sera plus efficace car le sommet conservé pourrait avoir davantage de descendants nécessitant un rehachage que le sommet éliminé.

[50] Nous avons vu à la section 2.3.2.1 qu'il modélise aussi une description du jeu en GDL.

```
Fonction FactoriserGraphe(g : graphe, h : table de hachage)
    f : file d'attente
    tant que il existe des sommets factorisables faire
        pour chaque sommet s de g par ordre décroissant du nombre de prédécesseurs
         factorisable faire
            factoriser s
            mettre dans la file f les sommets modifiés
            retirer de h les références vers les sommets ajoutés ou modifiés
        fin
        tant que f n'est pas vide faire
            prendre s dans la file f
            si s n'est pas dans h alors
                insérer dans h la référence vers s
            fin
            sinon
                t ← l'élément de h équivalent à s
                pour chaque u : successeur de s faire
                    ajouter u aux successeurs de t
                    retirer la référence à u dans h
                    ajouter u à f
                fin
            fin
        fin
    fin
    retourner g
```

Algorithme 2.1 : Factorisation avec mise à jour de la table de hachage par lots.

jeu. Pour cela nous adoptons une évaluation qui va des entrées vers les sorties. Cette manière de procéder peu sembler moins efficace qu'une approche où l'on irait des sorties vers les entrées car elle conduit à effectuer des évaluations inutiles pour un état donné. En réalité, cette méthode indépendante de l'état courant du jeu va nous permettre plus tard de paralléliser plus efficacement ce traitement. Il permet aussi de limiter le nombre de branchements dans le programme et organiser les emplacements mémoires des variables de façon à limiter les ralentissements liés à une absence des données dans le cache du processeur.

Le circuit doit nous servir à déterminer si la position courante est terminale. Si c'est le cas, il doit permettre de calculer les scores. Dans le cas contraire, nous souhaitons obtenir la liste des coups légaux et après que les joueurs ont choisi leur coup, calculer la position suivante et les perceptions. Pour chacune de ces opérations, il est inutile d'évaluer la totalité du circuit et pour déterminer les parties du circuit nécessaires à chacune de ces opérations, nous allons étiqueter chaque sommet du graphe par l'ensemble des opérations qui en dépend. Nous utilisons quatre étiquettes : T pour tous les sommets nécessaires au calcul de la sortie *terminal*, L pour ceux nécessaires aux sorties $legal(X, Y)$, G pour $goal(X, Y)$ et enfin N pour $next(X)$[51].

Pour effectuer un étiquetage, nous initialisons les sorties d'une des opérations en lui attribuant son étiquette puis nous propageons par un parcours en profondeur à partir des sorties du graphe en parcourant les arcs dans le sens inverse. Une fois l'étiquetage réalisé, chaque sommet est marqué par un sous-ensemble non vide de $\{T; L; G; N\}$ et nous obtenons une partition des sommets modulo ces 15 sous-ensembles. Nous disposons ainsi d'ordre partiel grossier pour l'évaluation du circuit. En effet, pour calculer un groupe étiqueté par un ensemble E, il faut au préalable avoir calculé tous les groupes étiquetés par un sur-ensemble de E. En résumé, l'ordre de calcul sur les groupes est un ordre partiel déterminé par la relation $\supset$ sur leurs étiquettes et possède donc une structure de treillis. Sur la figure 2.4, nous avons représenté une vue partielle de ce treillis en se limitant aux étiquettes T, L et G. Ainsi, pour calculer la sortie *terminal*, nous effectuons les opérations dans l'ordre suivant : celles étiquetées par $\{T; L; G\}$, puis (dans un ordre quelconque) celles des groupes $\{T; L\}$ et $\{T; G\}$, puis enfin celle du groupe $\{T\}$.

Si par exemple la position n'est pas terminale, le calcul des scores est inutile et on peut calculer les coups légaux en évaluant les groupes $\{L; G\}$ puis $\{L\}$. Nous constatons qu'il n'a pas été nécessaire de calculer le groupe $\{G\}$. Si la description du jeu est correcte, les groupes dont l'ensemble étiquette contient T, L ou G ne doivent pas dépendre des entrées $does(X, Y)$.

Cette partition est un peu trop fine. Les opérations pour calculer la terminaison, les coups légaux et scores et la position suivante sont toujours effectuées dans un ordre précis. Nous propo-

[51] Pour le traitement des descriptions en GDL-II que nous traitons à la section 3.1.2, il faut ajouter aussi une étiquette S pour le prédicat *sees*.

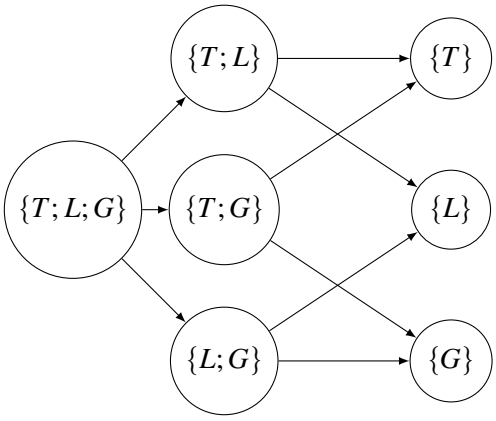


FIGURE 2.4 – Vue partielle du treillis représentant l'ordre partiel entre les éléments de la partition du graphe selon leur groupe d'étiquettes. L'arc de $\{T;L\}$ vers $\{T\}$ signifie que la partie $\{T;L\}$ doit être calculée avant la partie $\{T\}$.

sons donc une partition plus grossière en quatre sous-ensembles. Pour le calcul de la terminaison, nous prenons les sommets dont l'étiquetage contient $\{T\}$, $\{L;G\}$ ou $\{G;N\}$ car quelle que soit la terminaison, il faut calculer soit les coups légaux et la position suivante, soit les scores. Pour le calcul des coups légaux, seuls les sommets étiquetés par $\{L\}$ et $\{L;N\}$ sont nécessaires. Pour les scores, seuls ceux étiquetés par $\{G\}$ sont nécessaires. Enfin, pour le calcul de la position suivante, il ne reste que les sommets étiquetés par $\{N\}$. À présent, nous allons décrire comment déterminer l'ordre des opérations un sein d'un de ces quatre sous-ensembles de sommets que nous notons respectivement $E_{terminal}$, E_{legal}, E_{goal} et E_{next}.

2.3.2.4 Tri topologique pour déterminer l'ordre des opérations logiques

Nous déterminons l'ordre d'évaluation des sommets au sein d'un sous-ensemble E_p où $p \in \{terminal, legal, goal, next\}$ par l'algorithme bien connu de tri topologique. Le sous-graphe composé des sommets de E_p peut comporter des circuits. L'algorithme 2.2 permet à la fois de déterminer un ordre et aussi de détecter les circuits dans les sous-graphes de sommets E_p en utilisant une coloration des sommets. Dans l'algorithme, les sommets, initialement tous blancs, deviennent gris à leur première visite et leurs prédécesseurs sont visités immédiatement après lorsqu'ils sont blancs. Lorsqu'il y a un circuit, certains prédécesseurs sont déjà gris. Dans ce cas, l'arc reliant le sommet avec son prédécesseur gris est stocké dans un ensemble pour un traitement ultérieur. Lorsqu'un sommet est visité une seconde fois, il prend la couleur noire, signifiant que tous ses prédécesseurs ont déjà été traités.

```
Fonction TriTopologique(g : graphe)
    p ← ∅
    circuits ← ∅
    ordre ← ∅
    pour chaque sommet s de g faire
        colorier s en blanc
        si s n'a pas de successeur alors
            empiler s dans p
        fin
    fin
    tant que p n'est pas vide faire
        s ← dépiler p
        si s est blanc alors
            colorier s en gris
            circ ← ∅
            pour chaque prédécesseur x de s faire
                si x est blanc alors
                    empiler x dans p
                fin
                si x est gris alors
                    insérer x dans circ
                fin
            fin
            si circ n'est pas vide alors
                insérer {s, circ} dans circuits
            fin
        fin
        sinon
            si s est gris alors
                insérer s dans ordre
                colorier s en noir
            fin
        fin
    fin
    retourner {ordre, circuits}
```

Algorithme 2.2 : Tri topologique des sommets et de détection des circuits dans un graphe.

La quasi-totalité des descriptions GDL présentes sur les serveurs ont un graphe qui ne comporte pas de circuits. Nous supposons à partir de maintenant que le graphe est sans circuit et nous étudierons plus tard sur le cas des graphes ayant des circuits.

En traitant les sommets dans l'ordre de la liste *ordre* de l'algorithme 2.2, on est assuré que lors du calcul de la valeur de vérité d'un sommet, les valeurs de vérité de ses prédécesseurs ont déjà été calculées. Ça n'est pas le seul ordre possible et nous allons voir à présent que nous pouvons permuter des opérations au sein des E_p.

2.3.2.5 Stratification des opérations logiques

La connaissance des ensembles opérations qui peuvent s'effectuer dans un ordre indifférent permet d'envisager une parallélisation des calculs et une optimisation du placement des données en mémoire. Nous partitionnons l'ensemble des sommets en strates. La strate de niveau 0 correspond à l'ensemble des sommets sans prédécesseurs et la strate de niveau i contient des sommets qui peuvent être calculés uniquement à partir des sommets des strates inférieures. Au sein d'une strate, les valeurs de vérité des sommets peuvent être calculées dans n'importe quel ordre.

```
Fonction CalculStrates(g : graphe, ordre : liste)
    niveau ← étiquetage des sommets par des entiers
    pour chaque s dans g faire
        niveau[s] ← 0
    fin
    pour chaque s de la liste ordre faire
        niveau[s] ← −1
        pour chaque x prédécesseur de s faire
            niveau[s] ← max(niveau[s], niveau[x])
        fin
        niveau[s] ← niveau[s] + 1
    fin
    retourner niveau
```

Algorithme 2.3 : Calcul des strates.

L'algorithme 2.3 permet d'attribuer un niveau à chaque sommet égal au maximum des niveaux de ses prédécesseurs. Il utilise l'ordre calculé par l'algorithme 2.2. Nous pouvons ensuite partitionner chaque strate en fonction de la nature des sommets, c'est-à-dire s'ils sont étiquetés par ET, OU et NON.

Les strates du sous-graphe nous permettent d'envisager une parallélisation de l'évaluation du circuit logique car les opérations en leur sein sont indépendantes. Cette possibilité est décrite à la section 5.3.4. Elle pourrait aussi nous permettre d'optimiser le placement en mémoire des variables (contenant les valeurs de vérités des sommets) que nous introduisons à la section 2.3.2 afin de maximiser leur présence dans le cache du processeur.

Le graphe simplifié représente un circuit logique avec seulement deux types de portes à deux entrées : la porte *ou* et la porte *et* et un type de porte à une entrée : la porte *non*. Les sommets du graphe correspondant à ces portes sont structurés en sous-ensembles. L'ensemble E de tous les sommets est partitionné en $E_{terminal}$, E_{legal}, E_{goal} et E_{next}. Chaque ensemble E_p, qui détermine la partie du circuit à évaluer pour le calcul du prédicat p, est lui même découpé en strates E_{p_i} indiquant que les sommets E_{p_i} doivent être évalués avant les sommets de $E_{p_{i+1}}$. Enfin, chaque strate E_{p_i} est constituée d'un ensemble de négations, de conjonctions et de disjonctions et nous notons ces ensembles $E^{\neg}_{p_i}$, $E^{\wedge}_{p_i}$ et $E^{\vee}_{p_i}$.

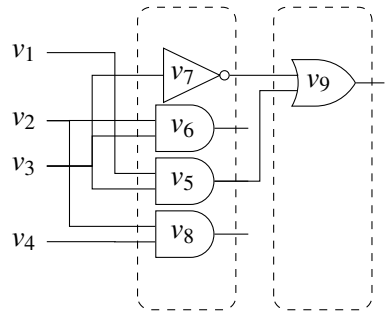
2.3.3 Compilation du circuit logique en un *bytecode*

À partir du graphe transformé de la façon décrite dans les sections précédentes, nous construisons un simulateur de circuits logiques qui permet d'interpréter le GDL.

Nous attribuons une variable booléenne à chaque sommet du graphe et notons V cet ensemble de variables. Parmi toutes ces variables, certaines correspondent à des entrées ou des sorties du circuit. Nous définissons donc des sous-ensembles de V afin de repérer les entrées correspondant aux prédicats *true* décrivant l'état courant du jeu et *does* pour les coups choisis par les joueurs. De même, nous repérons les variables correspondant aux sorties indiquant la fin de partie avec *terminal*, les coups légaux avec *legal*, les scores avec *goal* et l'état suivant du jeu avec *next*.

Nous notons $V_f \subset V$ le sous-ensemble de variables attribuées aux sommets étiquetés par les atomes dont le symbole de fonction est f, où f appartient à $\{true, does, terminal, legal, goal, next\}$. Lorsque le premier argument des atomes formés avec $f \in \{does, legal, goal\}$ est un des joueurs `j` déclaré par `(role j)`, alors nous notons V_{f_j} le sous-ensemble de variables correspondant.

Les opérations booléennes à effectuer pour un sous-graphe E_p sont compilées dans un *bytecode*. Il commence par le nombre de strates de E_p suivi du *bytecode* de chaque strate E_{p_i} dans l'ordre. Chaque strate est constituée du code compilé pour les conjonctions $E^{\wedge}_{p_i}$, les disjonctions $E^{\vee}_{p_i}$ et les négations $E^{\neg}_{p_i}$. Enfin, le code pour les conjonctions ou les disjonctions est constitué du nombre d'opérations suivi de triplets de variables dont le premier élément est la variable de destination et les deux suivants sont les opérandes. Pour les négations, le code contient le nombre d'opérations suivi de couples de variables dont le premier élément est la variable de destination et le second est l'opérande.



bytecode	commentaire
	début du code
2	2 strates
3	début de la première strate, 3 conjonctions
6 2 3	$v_6 \leftarrow v_2 \wedge v_3$
5 1 3	$v_5 \leftarrow v_1 \wedge v_3$
8 2 4	$v_8 \leftarrow v_2 \wedge v_4$
0	0 disjonction
1	1 négation
7 3	$v_7 \leftarrow \neg v_3$
0	début de la seconde strate, 0 conjonction
1	1 disjonction
9 7 5	$v_9 \leftarrow v_7 \vee v_5$
0	0 négation
	fin du code

FIGURE 2.5 – Exemple de circuit (à gauche) compilé en *bytecode* (à droite). Les cadres en pointillés indiquent les strates qui doivent être évaluées dans l'ordre, de gauche à droite. Au sein d'une strate, l'ordre des évaluations n'a pas d'importance.

La figure 2.5 donne un exemple de *bytecode* produit à partir d'un graphe et de sa stratification. Les variables sont numérotées ce qui peut correspondre à leur indice dans un tableau ou leur adresse en mémoire. L'algorithme 2.4 permet d'évaluer ce *bytecode* sur un ensemble de variables booléennes.

Afin de faciliter le passage de l'état courant du circuit à son état suivant, nous explicitons des associations entre certaines entrées et certaines sorties. Un sommet s étiqueté par `(next x)` et un sommet t étiqueté par `(true x)` avec le même terme `x` sont dits associés. De même un sommet s étiqueté par `(legal j c)` et un sommet t étiqueté par `(does j c)` avec les mêmes termes `j` et `c` sont dits associés. Une variable v attribuée à un sommet étiqueté par `(next x)` représente l'état suivant du jeu et sa valeur devra être recopiée dans la variable w attribuée au sommet étiqueté par `(true x)`. De la même manière que nous notons $w \leftarrow v$ pour cette recopie, nous notons $V_{true} \leftarrow V_{next}$ pour la recopie des variables associées entre les sommets **next** et les sommets **true**. On notera de la même manière $V_{does} \leftarrow V_{legal}$ pour la recopie des variables associées entre **does** et **legal**.

L'initialisation des variables d'entrée du circuit est la donnée d'une application f de $E \subset V$ dans l'ensemble des booléens. Nous noterons plus simplement cette initialisation par $V \leftarrow f$.

Procédure `EvalBytecode`($p = \{p_i, i \text{ entier}\}$: bytecode, $v = \{v_k, k \text{ entier}\}$: *booléens*)

$i \leftarrow 0$
pour $s \leftarrow p_0$ **à** 1 **faire**
  $i \leftarrow i + 1$
  pour $j \leftarrow p_i$ **à** 1 **faire**
    $v_{p_i} \leftarrow v_{p_{i+1}} \wedge v_{p_{i+2}}$
    $i \leftarrow i + 3$
  fin
  $i \leftarrow i + 1$
  pour $j \leftarrow p_i$ **à** 1 **faire**
    $v_{p_i} \leftarrow v_{p_{i+1}} \vee v_{p_{i+2}}$
    $i \leftarrow i + 3$
  fin
  $i \leftarrow i + 1$
  pour $j \leftarrow p_i$ **à** 1 **faire**
    $v_{p_i} \leftarrow \neg v_{p_{i+1}}$
    $i \leftarrow i + 2$
  fin
fin

Algorithme 2.4 : Évaluation du *bytecode*.

De même, la lecture des sorties du circuit est modélisée par une fonction g du même type. Nous notons $E \leftarrow f$ et $g \leftarrow E$ ces opérations. Par exemple, l'initialisation des variables de V_{true} à partir d'un état du jeu donné par une application *init* s'écrit simplement $V_{true} \leftarrow init$.

Procédure `CoupAleatoire(`*j : joueur,* V_{legal_j}, V_{does_j} *: variables booléennes*`)`

```
l ← liste vide
pour chaque v ∈ V_legal_j faire
    si v = 1 alors
        insérer une référence à v dans l
    fin
fin
choix ← tirage aléatoire dans l
pour chaque v ∈ V_does_j faire
    si v est la variable de V_does_j associée à variable choix de V_legal_j alors
        v ← 1
    fin
    sinon
        v ← 0
    fin
fin
```

Algorithme 2.5 : Choix aléatoire d'un coup pour le joueur j.

2.3.4 Analyse statique du circuit pour détecter des caractéristiques du jeu

Une analyse statique du graphe représentant le circuit logique nous permet de déterminer des caractéristiques du jeu qui nous permettent d'envisager une décomposition de celui-ci en plusieurs sous-jeux. L'intérêt de cette décomposition est de permettre une découverte plus rapide des positions gagnantes dans chacun des sous-jeux et d'améliorer ainsi le niveau de notre joueur pour le jeu initial.

2.3.4.1 Décomposition d'un jeu en sous-jeux

Dans des travaux avec Aline Hufschmitt, nous avons identifié plusieurs type de jeux composés ainsi que la manière de les détecter [32].

Un jeu parallèle est composé de plusieurs sous-jeux qui sont joués en parallèle. Ces sous-jeux peuvent être à un seul joueur et dans ce cas, chaque joueur du jeu principal ne joue qu'à un et un

Fonction `Playout`(*e : état du jeu,* $P_{terminal}, P_{legal}, P_{next}, P_{goal}$ *:* bytecodes*,* V *: booléens*)
  $V_{true} \leftarrow e$
  `EvalBytecode`($P_{terminal}$, V)
  tant que $t \neq 1$ *où t est l'unique variable de* $V_{terminal}$ **faire**
    `EvalBytecode`(P_{legal}, V)
    pour chaque *joueur* j **faire**
      `CoupAleatoire`(j, V_{legal}, V_{does})
    fin
    `EvalBytecode`(P_{next}, V)
    $V_{true} \leftarrow V_{next}$
    `EvalBytecode`($P_{terminal}$, V)
  fin
  `EvalBytecode`(P_{goal}, V)
  retourner V_{goal}

Algorithme 2.6 : Calcul du résultat d'un *playout* à partir d'un état donné du jeu.

seul sous-jeu. Sinon, il peut s'agir de sous-jeux à plusieurs joueurs et dans ce cas, chaque joueur joue alternativement dans les différents sous-jeux.

Un jeu à actions composées est un jeu dans lequel plusieurs sous-jeux sont joués en même temps et pour lequel une seule action d'un joueur influence en même temps plusieurs de ces sous-jeux.

Un jeu en série est composé de sous-jeux joués les uns après les autres, la fin d'un sous-jeu signifiant le début d'un autre.

Un jeu multiple est composé de plusieurs instances du même sous-jeu dans lesquels le joueur peut jouer indifféremment. Seul un des sous-jeux détermine la terminaison du jeu composé et contribue au score. Cette notion de jeu multiple se généralise aux jeux comportant des actions assimilables au fait de passer son tour.

Un jeu comportant un compteur de coups est assimilable à un jeu composé. Les compteurs de coups sont ajoutés par les concepteurs de descriptions GDL afin d'assurer que le jeu se termine toujours en un nombre fini de coups.

L'analyse statique du graphe représentant le circuit logique permet d'identifier les actions qui sont utilisées en même temps que des fluents et aussi les actions responsables d'un effet dans

le cas de coups-joints. Il permet aussi d'identifier facilement les compteurs par la recherche des composantes connexes du graphe dans lequel les sorties correspondant au prédicat **`next`** ont été reliées aux entrées associées du prédicat **`true`**. Cette analyse statique ne permet de découvrir que les jeux parfaitement décomposables, c'est-à-dire ceux dont l'automate de transition représenté par le circuit logique reflète cette propriété.

2.3.4.2 Détection de verrous

Lors de la simulation du déroulement d'une partie par le circuit logique, les portes logiques le constituant passent par des états successifs vrais ou faux. Un verrou (*latch* en anglais) est une de ces portes logiques qui, lorsqu'elle atteint un de ses deux états possibles, le conservera jusqu'à la fin de la partie. La détection de ces verrous est particulièrement importante car elle permet de simplifier le circuit logique en lui retirant des parties qui resteront constantes jusqu'à la fin de la partie. En plus d'accélérer la simulation de parties aléatoires, cette simplification peut aussi permettre de découvrir de nouvelles décomposition du jeu qui émergent en cours de partie.

2.3.5 Les implémentations de circuits logiques dans les autres joueurs

La plupart des joueurs de la compétition annuelle organisée par l'université de Stanford et de la compétition Tiltyard Open ont adopté ces trois dernières années une transformation de la description GDL en circuit logique combinatoire pour accélérer la simulation de parties aléatoires. Ces circuits logiques sont appelés Propnets (pour réseaux de propositions). Lors des compétitions précédentes seul un joueur, les utilisait : TurboTurtle. Sam Schreiber, l'auteur de ce joueur n'a jamais publié ses recherches dans le domaine mais la création du MOOC sur le General Game Playing en 2014 auquel il a participé à permis la divulgation d'une partie des techniques qu'il utilisait notamment du fait de la mise à disposition du code source d'un joueur rudimentaire : GGP-Base.

2.4 Évaluation de la vitesse de l'instanciation et du *bytecode*

2.4.1 Temps nécessaire à l'instanciation

Nous avons mesuré le temps nécessaire à l'instanciation des règles de la totalité des 246 descriptions GDL disponibles sur le serveur de GGP de Dresden en février 2014. Nous avons exécuté le programme sur un seul thread d'un processeur Intel Xeon E5-4610 2.40GHz doté de 520Gio de RAM. Nous avons utilisé l'interpréteur YapProlog 6.2.2 en tant que bibliothèque pour LeJoueur.

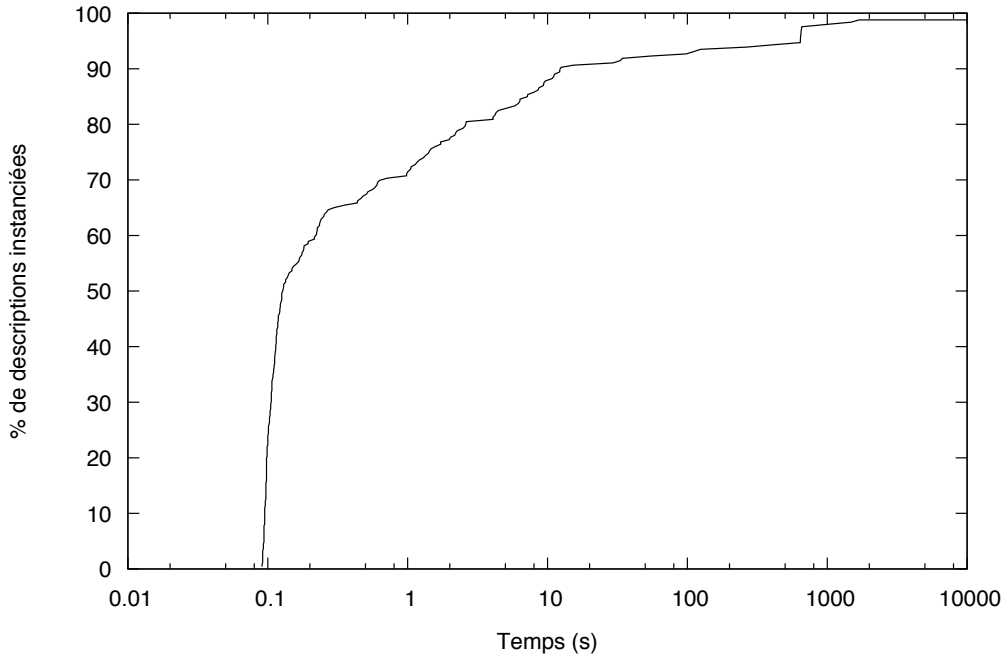


FIGURE 2.6 – Pourcentage de descriptions GDL instanciées en un temps inférieur à celui précisé sur l'axe des abscisses qui est en échelle logarithmique.

Le calcul de l'instanciation a été interrompue lorsque le temps nécessaire dépassait 30 minutes : seuls trois descriptions n'ont pas été instanciées : RACER, RULEDEPTHQUADRATIC et LAIKLEE HEX.

La figure 2.6 résume la performance de notre méthode d'instanciation dans le cas où les prédicats **`input`** et **`base`** sont pré-calculés. Nous avons représenté le pourcentage de descriptions de jeux qui ont été instanciées dans le temps représenté par l'axe des abscisses. 24% des descriptions sont instanciées en moins de 100ms, 72% en moins d'une seconde et 94% en moins d'une minute. Les 6% restants qui sont instanciées en plus d'une minute sont BATTLEBRUSHES, MERRILLS, AMAZONS, RACER4, FARMERS, les deux instances de BATTLESNAKES et 8 des 13 instances de VACUUMCLEANER. Ces descriptions qui sont instanciées en plus d'une minutes contienne plus de 10^7 règles.

2.4.2 Comparaison entre les temps d'évaluation de YAPPROLOG et du *bytecode*

Nous avons comparé les performances du *bytecode* par rapport à YAPPROLOG sur les 157 descriptions GDL qui étaient disponibles sur le serveur Tiltyard à la fin du mois d'août 2017. Nous avons compilé chaque description en *bytecode* et mesuré le nombre de *playouts* à la racine qu'il pouvait calculer pendant 10 secondes sur un thread de processeur Intel Xeon E5-4610 à 2.40GHz. Nous avons réalisé les mêmes mesure en utilisant l'interpréteur YAPPROLOG.

Les résultats sont présentés à la figure 2.7 en échelle logarithmique pour les deux axes. L'axe des abscisses représente le nombre de *playouts* par seconde réalisés par YAPPROLOG et en ordonnée, le nombre de *playouts* réalisés par le *bytecode*. Les points situés au dessus de la diagonale en trait plein correspondent aux jeux pour lesquels le *bytecode* est plus performant. Le *bytecode* est plus rapide pour 85% des jeux, 10 fois plus rapide pour 61% des jeux et 100 fois plus rapide dans 10% des cas. Les 15% de jeux pour lesquels YAPPROLOG est plus rapide correspond aux *bytecodes* les plus longs comme par exemple ceux de CHESS ou d'ATARIGO_7X7. La meilleure performance de YAPPROLOG s'explique par le fait que celui-ci n'a pas toujours à prouver tous les termes présents dans une conjonction ou une disjonction.

2.5 Conclusion

Nous avons présenté dans ce chapitre les caractéristiques de l'interprétation des descriptions de jeux en GDL. lejoueur dispose de deux raisonneurs pour réaliser cette interprétation : YAPPROLOG et un *bytecode* compilé à partir de la description GDL instanciée. Nous avons montré qu'il est possible d'instancier la majorité des descriptions de jeux présents sur les serveurs dans un temps compatible avec les contraintes du GGP. L'analyse statique du circuit logique correspondant à l'instanciation permet de découvrir certaines formes de décompositions de jeux et d'envisager des simplifications de ce circuit en cours de match pour accélérer encore son évaluation. Le *bytecode* compilé à partir de la description GDL instanciée permet de réaliser des *playouts* plus rapidement que YAPPROLOG pour 85% des jeux. Nous décrivons au chapitre 5 une méthode de parallélisation permettant d'améliorer encore les performances.

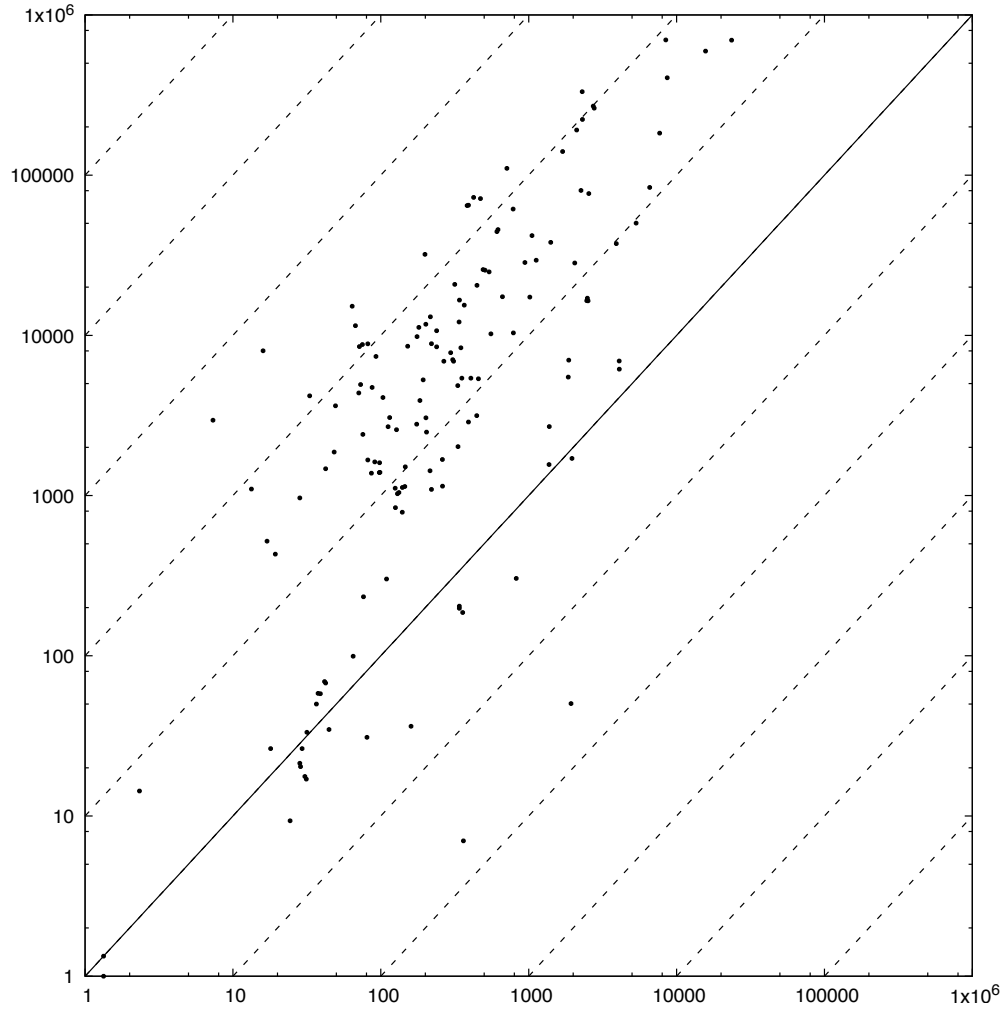


FIGURE 2.7 – Comparaison du nombre de parties par seconde réalisées par le *bytecode* (en ordonnée) en fonction du nombre de parties par seconde réalisées par YAPPROLOG (en abscisse). Les abscisses et les ordonnées sont en échelle logarithmique. La diagonale en trait plein correspond aux points où les performances sont les mêmes entre le *bytecode* et YAPPROLOG. Les lignes en pointillés au dessus correspondent aux points où le *bytecode* est plus rapide d'un facteur 10, 100, 1000, etc.

Chapitre 3

Jeux à information incomplète et/ou imparfaite

Nous présentons ici le General Game Playing pour les jeux à information incomplète ou imparfaite. Après avoir décrit la problématique spécifique de ces jeux dans le contexte du GGP, nous présentons la manière dont ces jeux sont traités par LeJoueur.

3.1 Descriptions de jeux à information incomplète/imparfaite

3.1.1 Les jeux à information imparfaite en GDL

Les jeux décrits en GDL sont multijoueurs finis, discrets, déterministes et à information complète [40, page 1]. Parmi ces jeux, le langage est aussi capable de décrire une sous-classe des jeux à information imparfaite : ceux à coups simultanés. L'imperfection vient du fait qu'au moment de choisir son coup, un joueur ne connaît pas les coups des adversaires. Après chaque coup joint, chaque joueur est informé des coups des adversaires et donc sait parfaitement dans quel état se trouve le jeu.

3.1.2 GDL pour les jeux à information incomplète/imparfaite (GDL-II)

Le langage GDL-II est une extension par Thielscher du langage GDL pour les jeux à information incomplète et imparfaite. Il n'y a pas de document de spécification comme c'est le cas en GDL. Les informations au sujet de ce langage de description sont dispersées dans trois sources principales. Dans l'article initial de présentation, l'auteur introduit un nouveau mot clé **`random`** et un nouveau prédicat **`sees`** [71]. Dans le livre de Genesereth et Thielscher sur le General Game

Playing, les auteurs introduisent un nouveau prédicat **percept** [28]. La spécification du protocole de communication est quant à elle disponible sur le serveur GGP de l'université de Dresden[52]. Nous présentons dans les sections suivantes ces différents ajouts, le protocole de communication et analysons ensuite les classes de jeux que l'on peut décrire en GDL-II. Peu après la création de cette extension, l'auteur a aussi proposé une solution pour traiter ces descriptions [66].

3.1.2.1 Le pseudo-joueur `random`

En GDL, le prédicat **role** sert à nommer les joueurs dans la description du jeu. Par exemple, `(role red)` permet d'indiquer qu'un joueur `red` est un acteur du jeu et l'interprétation de la description permet de connaître pour chaque état du jeu, ses coups légaux `(legal red ?coup)` et son paiement `(goal red ?nombre)`. Ces noms sont choisis librement par l'auteur de la description. Le GDL-II ajoute un nouveau mot-clé au langage : **random** qui sert à nommer un joueur qui représente ce que l'on appelle *la nature* en théorie des jeux. Les descriptions ayant besoin de ce pseudo-joueur contiennent simplement le fait `(role random)`.

La nature (**random**) choisit ses coup aléatoirement suivant une distribution de probabilités connue à l'avance. Dans le cas du GDL-II, la distribution est toujours uniforme. Par exemple, imaginons un lancé de pièce, la nature n'a que deux choix possibles : `(legal random pile)` et `(legal random face)`. Nous savons alors qu'elle a 50% de choisir pile et 50% de choisir face. Il est toutefois possible d'obtenir des distributions discrètes quelconques en créant des coups possibles dont les effets sont identiques à un autre coup pour le passage d'un état à un autre du jeu. Par exemple, une pièce pipée peut être modélisée en donnant trois choix à la nature : `(legal random pile)`, `(legal random face)` et `(legal random face_bis)`. Dans ce cas, la nature a une chance sur trois de choisir `pile`, `face` ou `face_bis`. Si ces deux derniers coups ont exactement les mêmes effets lors du passage de l'état courant à l'état suivant, cela revient à choisir pile avec une probabilité de $\frac{1}{3}$ et face avec une probabilité de $\frac{2}{3}$. Nous remarquons cependant que lorsque la distribution utilisée par la nature n'est pas uniforme, le nombre d'actions possibles pour celle-ci peut augmenter significativement.

Comme les autres acteurs du jeu, **random** apparaît en paramètre des prédicats hérités du GDL : **role** comme nous l'avons vu, mais aussi **input**, **legal**, **does** et **goal**. Ces prédicats gardent le sens qu'ils avaient en GDL. Le concept de nature dans les jeux n'ayant pas de but, la valeur indi-

[52] *The GDL-II communication protocol* : `http://ggpserver.general-game-playing.de/ggpserver/public/gdlii_protocol.jsp`

quée par le prédicat **goal** n'a pas d'importance[53]. Le pseudo-joueur **random** est géré directement par le Game Manager et connaît donc toujours parfaitement l'état courant du jeu.

3.1.2.2 Le prédicat sees

Le prédicat **sees** permet au Game Manager de communiquer des informations, appelées *perceptions* en GDL-II, sur l'état du jeu à un joueur. Ce prédicat est appelé à la fin de chaque tour lorsque tous les coups des joueurs ont été spécifiés par le prédicat **does**. Son calcul a donc lieu en même temps que celui du prédicat **next**. Il prend deux arguments : le premier est le joueur qui doit recevoir l'information ; le second est l'information en elle même sous la forme d'une expression KIF.

Les informations qui sont communiquées via le prédicat **sees** peuvent être de n'importe quelle nature. Ça peut être par exemple révéler certains coups joués précédemment par les autres joueurs ou encore informer qu'une ligne ou une colonne est vide pour un jeu se jouant sur un échiquier.

Le prédicat **sees** ne peut apparaître que dans la conclusion d'une règle donc jamais dans une prémisse. Ce prédicat n'a donc aucune influence sur le calcul par le Game Manager des états de la partie qui se fait à information complète et parfaite comme en GDL. Il n'y a pas non plus de relation directe entre la forme d'une information introduite par le prédicat **sees** en GDL-II et les termes utilisés pour décrire l'état du jeu avec le prédicat **true**. Par exemple, l'état d'une pièce dans un jeu pourrait être décrit par `(true pile)` ou `(true face)` et le prédicat **sees** pourrait informer un joueur de l'état de la pièce respectivement par `(sees vert)` et `(sees bleu)`.

Il y a cependant un aspect qui reste sous silence dans l'article présentant le GDL-II. Dans un jeu à information imparfaite, le joueur ne sait pas dans quel état se trouve le jeu. Il y a un ensemble d'états où il pourrait se trouver qui constitue l'ensemble d'information. Si pour tous ces états, les coups légaux sont identiques, alors le joueur est assuré qu'un choix parmi ces coups sera légal quel que soit l'état réel dans lequel il se trouve et qui est connu par le Game Manager. Si cette condition n'est pas vérifiée, par exemple si son ensemble d'information ne comporte que deux états, qu'il n'y a qu'un seul coup légal par état et que ces coups sont distincts, alors il se peut qu'il joue un coup illégal[54]. Ce cas de figure serait particulièrement fâcheux car il ne révèle pas une faiblesse du joueur à notre sens mais un problème dans la description du jeu.

[53] Pour que la description reste bien formée au sens des spécification GDL, il doit exister des parties où **random**, comme n'importe quel autre joueur, obtient un score maximal. La manière la plus simple est de déclarer un unique théorème qui permet à `ramdom` d'obtenir toujours un score de 100 quel que soit l'état du jeu : `(goal random 100)`.

[54] Ce que nous appelons *coup illégal* est ce que le Game Manager, qui connaît précisément l'état du jeu, considère comme illégal. Le joueur avec sa vision partielle de l'état du jeu peut ignorer que le coup est impossible dans l'état courant.

Dans leur ouvrage sur le GGP, Genesereth et Thielscher font état de ce problème en affirmant simplement qu'un joueur pourrait jouer un coup illégal dans le cas où il ne connaîtrait pas l'intégralité de l'ensemble d'information [28, page 182]. Cela suggère que les seuls coups légaux garantis sont ceux qui appartiennent à l'intersection des ensembles de coups légaux de tous les états de l'ensemble d'information, les autres coups légaux étant potentiellement risqués. Mais comme nous l'avons vu dans un exemple au paragraphe précédent, cette intersection peut être vide. En GDL, pour une description bien formée, envoyer un coup illégal au Game Manager est considéré comme une erreur technique du joueur (il ne respecte pas les règles du jeu) et elle est sanctionnée d'une part par le fait que le Game Manager peut choisir n'importe quel coup légal en remplacement mais cela peut aussi entraîner une élimination du joueur lors d'une compétition. Il serait plus raisonnable de conserver cet esprit en GDL-II et considérer qu'une description GDL-II est invalide lorsqu'il existe un ensemble d'information dans lequel deux états distincts ont des ensembles de coups légaux distincts.

3.1.2.3 Le prédicat `percept`

Le prédicat **`percept`** a été introduit dans le même esprit que **`input`** et **`base`** (voir la section 1.3.2.8). Il définit l'ensemble de toutes les perceptions possibles que pourraient recevoir les joueurs via le prédicat **`sees`** dans n'importe quelle partie. Il apparaît dans certaines descriptions de jeu du serveur GGP de l'université de Dresden et est décrit uniquement dans l'ouvrage consacré au GGP de Genesereth et Thielscher [28]. Comme pour **`input`** et **`base`**, ce prédicat a sans doute été introduit pour faciliter l'instanciation des descriptions de jeux en GDL-II.

3.1.2.4 Protocole de communication entre le Game Manager et les joueurs

Deux modifications sont apportées au protocole de communication en GDL-II par rapport à celui du GDL (voir section 1.3.3). Seules les commandes **`PLAY`** et **`STOP`** sont modifiées, les autres commandes, notamment **`START`**, restant inchangées.

La commande `PLAY` qui en GDL était de la forme :

(**`PLAY`** <MATCHID> (<C1> <C2> ... <Cn>)

où <Ci> est le coup joué par le joueur *i*, est en GDL-II de la forme :

(**`PLAY`** <MATCHID> <TURN> <LASTMOVE> <PERCEPTS>).

Le paramètre <TURN> est un entier qui compte le nombre de tours. Il vaut zéro à la première commande **`PLAY`** et est incrémenté de un à chaque commande **`PLAY`** suivante et à la commande **`STOP`**.

Le paramètre <LASTMOVE> est le coup exécuté par le joueur au dernier tour (sauf pour le premier message **PLAY** où il vaut NIL). Dans le cas où le joueur a envoyé un coup illégal, s'il n'a pas envoyé de coup ou s'il l'a envoyé trop tard, cela lui permet de connaître le coup que le Game Manager a choisi à sa place.

Le paramètre <PERCEPTS> sert à envoyer des informations au joueur sur la partie en cours conformément aux faits produits par le prédicat **sees**.[55]

La commande STOP a exactement la même forme que la commande **PLAY** décrite ci-dessus. C'est la dernière commande du match. Nous pouvons regretter qu'elle n'informe pas les joueurs des scores obtenus par tous, cette information étant connue du Game Manager mais parfois impossible à évaluer pour les joueurs étant donné que le jeu est à information incomplète ou imparfaite. En conservant le même protocole, il est tout de même possible d'informer les joueurs de leur score en envoyant une perception dont l'effet est de réduire l'ensemble d'information de chaque joueur à des états finaux du jeux qui possèdent les mêmes scores. Ceci n'est cependant pas imposé dans la définition du GDL-II et n'a pas été mis en œuvre dans les compétitions de GDL-II.

3.1.2.5 Exemple d'une partie en GDL-II

Le problème de Monty Hall est un jeu à information incomplète à un seul joueur librement inspiré d'un jeu télévisé américain, le jeu portant le nom de son présentateur. Le problème s'énonce simplement. Le joueur a devant lui trois portes fermées, l'une cachant une voiture, les deux autres cachant une chèvre. Le but du joueur est de gagner la voiture en ouvrant la porte derrière laquelle elle se trouve. Le jeu se déroule en trois étapes : le joueur choisit une des portes qui reste fermée ; une des deux autres portes cachant une chèvre est ouverte, le joueur décide alors de conserver son premier choix ou de choisir l'autre porte fermée. Le présentateur demande alors l'ouverture de toutes les portes pour constater si le joueur a gagné la voiture ou non.

La solution à ce problème est contre-intuitive, c'est pourquoi il est souvent appelé paradoxe de Monty Hall. L'analyse mathématique de ce problème montre que la stratégie maximisant le gain consiste à changer de porte à la troisième étape quels que soient les événements ayant eu lieu aux deux étapes précédentes. En suivant cette stratégie, la probabilité de gagner la voiture est de $\frac{2}{3}$. Ce résultat peut aussi être vérifié expérimentalement en utilisant cette stratégie dans un grand nombre de parties simulées.

La description du jeu en GDL-II introduit le pseudo-joueur **random** pour effectuer les choix de la nature (voir figure 3.1, ligne 1). Celui-ci place la voiture et les chèvres au tour de jeu numéro 0.

[55] La mise en forme de plusieurs perceptions n'est pas précisée mais l'on suppose que l'ensemble des perceptions est mise entre parenthèses et donc qu'une unique perception est aussi mise entre parenthèses.

Il a trois possibilités selon la position de la voiture et il choisit de manière équiprobable. Il joue une seconde fois au tour de jeu numéro 2 pour ouvrir une porte qui cache une chèvre parmi celles qui n'ont pas été choisies par le joueur. Il n'influence plus le jeu ensuite.

La figure 3.2 montre un exemple de communication entre le Game Manager et un candidat au problème du MONTYHALL. On peut y voir les actions effectuées par le candidat : il choisit la porte 2, puis ne fait rien, puis modifie son choix initial.

À la ligne 1, le joueur reçoit une commande **START** du Game Manager qui lui fournit le nom du match `match_montyhall` et la description du jeu en GDL-II, l'informe qu'il jouera le rôle `candidate` et qu'il dispose de 20 secondes pour répondre à cette commande et de 10 secondes pour chaque commande **PLAY** qui suivra. À la ligne suivante, nous constatons qu'il répond `ready`. Il reçoit ensuite une commande **PLAY** lui précisant qu'il s'agit du tour 0 du match et les deux `nil` indiquent qu'aucun coup n'a été joué et qu'il n'y a pas de perceptions. Le joueur répond en indiquant son premier coup : `(choose 2)`. La commande **PLAY** suivante indique qu'il s'agit du tour 1, rappelle le coup qu'à choisi le joueur au tour précédent et lui fournit une liste avec une unique perception (**does**`(candidate (choose 2))`)[56]. Il répond en choisissant le coup `noop`. La commande **PLAY** suivante fonctionne de la même manière avec une liste de perceptions à deux éléments dont le plus important `(open_door 3)` qui fournit l'information que la voiture ne se trouve pas derrière la porte 3. Le joueur répond en indiquant qu'il veut changer son choix initial avec `switch`. La dernière commande **STOP** a le même modèle que les commandes **PLAY** précédentes. Elle indique en particulier la perception `(car 1)` qui permet au joueur de déduire qu'il a gagné[57].

3.2 Le jeu de l'ensemble d'information

Nous présentons le premier modèle que nous avons développé pour LEJOUEUR. Ce programme a participé à la compétition de décembre 2012 organisée par l'université de Nouvelles Galles du sud en Australie[58]. Nous n'avions pas encore développé les circuits pour une évaluation rapide des règles GDL et la principale difficulté était que notre joueur puisse jouer en GDL-II sans envoyer de coups illégaux. Il s'agissait de la première compétition avec des jeux à information incomplète et imparfaite et le langage de description venait tout juste d'être divulgué.

[56]Cette perception est totalement inutile, elle ne fournit aucune information que le joueur ne connaisse déjà. Elle pourrait être remplacée par n'importe quoi. Il en est de même pour les perceptions qui rappellent le coup joué des tours précédents.

[57]Les spécifications n'imposent pas que la dernière perception contienne suffisamment d'information pour que le joueur sache dans quel état du jeu il se trouve.

[58]`https://wiki.cse.unsw.edu.au/ai2012/GGP/`

```
(role random)
(role candidate)

(init (closed 1))
(init (closed 2))
(init (closed 3))
(init (step 1))

(<= (legal random (hide_car ?d))
   (true (step 1))
   (true (closed ?d)))
(<= (legal random (open_door ?d))
   (true (step 2))
   (true (closed ?d))
   (not (true (car ?d)))
   (not (true (chosen ?d))))
(<= (legal random noop)
   (true (step 3)))

(<= (legal candidate (choose ?d))
   (true (step 1))
   (true (closed ?d)))
(<= (legal candidate noop)
   (true (step 2)))
(<= (legal candidate switch)
   (true (step 3)))
(<= (legal candidate noop)
   (true (step 3)))

(<= (sees candidate
         (does candidate ?m))
   (does candidate ?m))
(<= (sees candidate (open_door ?d))
   (does random (open_door ?d)))

(<= (next (car ?d))
   (does random (hide_car ?d)))
(<= (next (car ?d))
   (true (car ?d)))
(<= (next (closed ?d))
   (true (closed ?d))
   (not (does random (open_door ?d))))
(<= (next (chosen ?d))
   (next_chosen ?d))

(<= (next_chosen ?d)
   (does candidate (choose ?d)))
(<= (next_chosen ?d)
   (true (chosen ?d))
   (not (does candidate switch)))
(<= (next_chosen ?d)
   (does candidate switch)
   (true (closed ?d))
   (not (true (chosen ?d))))

(<= (next (step 2))
   (true (step 1)))
(<= (next (step 3))
   (true (step 2)))
(<= (next (step 4))
   (true (step 3)))

(<= (sees candidate (car ?d))
   (true (step 3))
   (true (car ?d))
   (next_chosen ?d))

(<= terminal
   (true (step 4)))

(goal random 100)
(<= (goal candidate 100)
   (true (chosen ?d))
   (true (car ?d)))
(<= (goal candidate 0)
   (true (chosen ?d))
   (not (true (car ?d))))
```

FIGURE 3.1 – Description du jeu MONTYHALL en GDL-II par Thielscher.

```
1  (START match_montyhall ((role random)(role candidate)...) candidate 20 10)
2    ready
3  (PLAY match_montyhall 0 nil nil)
4    (choose 2)
5  (PLAY match_montyhall 1 (choose 2) ((does candidate (choose 2)))
6    noop
7  (PLAY match_montyhall 2 noop ((does candidate noop) (open_door 3)))
8    switch
9  (STOP match_montyhall 3 switch ((does candidate switch) (car 1)))
10   done
```

FIGURE 3.2 – Exemple de communication entre le Game Manager et le candidat durant une partie de MONTYHALL. Les envois du Game Manager commencent par une commande en majuscules et chaque réponse du candidat est donnée à la ligne suivante indentée.

3.2.1 Utilisation de descriptions GDL-II avec les spécifications du GDL

Un joueur peut interpréter les descriptions de jeux en GDL-II comme s'il s'agissait de description en GDL. Les deux nouveaux mots-clés **random** et **sees** ne sont pas réservés en GDL, par conséquent le pseudo joueur **random** est traité comme un joueur classique et le prédicat **sees** est simplement ignoré car il n'a pas de signification particulière et n'apparaît jamais dans les prémisses des règles.

La figure 3.3 montre un exemple de communication en GDL entre le Game Manager et un candidat avec la description GDL-II du problème de MONTYHALL. Le protocole de communication GDL permet au joueur de connaître les coups effectués par le joueur **random** et grâce à cette information, le candidat connaît parfaitement à chaque commande **PLAY** dans quel état du jeu il se trouve. Dans le cas du MONTYHALL, le jeu devient trivial car quel que soit le choix de **random**, le candidat est toujours informé de la porte derrière laquelle est cachée la voiture.

C'est donc le protocole de communication du GDL-II qui permet au Game Manager de ne dévoiler qu'une partie de l'information sur la partie grâce au prédicat **sees**. Le Game Manager sait toujours dans quel état précis du jeu se trouve la partie.

La construction du graphe du jeu avec une description en GDL-II se fait donc de la même manière qu'en GDL classique. Les joueurs peuvent aussi effectuer des playouts pour évaluer les sommets de ce graphe. Il n'y a que deux différences introduites par le GDL-II : au début de chaque tour du match, le joueur peut ne pas savoir précisément le sommet du graphe dans lequel il se trouve ; le joueur connaît la stratégie du pseudo-joueur **random** qui choisit toujours un coup en tirant parmi ses coups légaux de manière uniforme. Pour modéliser le fait que le joueur est

```
1  (START match_montyhall_gdl ((role random)(role candidate)...) candidate 20 10)
2    ready
3  (PLAY match_montyhall_gdl nil nil)
4    (choose 2)
5  (PLAY match_montyhall_gdl ((hide_car 1) (choose 2)))
6    noop
7  (PLAY match_montyhall_gdl ((open_door 3) noop))
8    switch
9  (STOP match_montyhall_gdl (noop switch))
10   done
```

FIGURE 3.3 – Exemple de communication en GDL entre le Game Manager et le candidat durant un match avec la description GDL-II du problème de MONTYHALL. Les envois du Game Manager commencent par une commande en majuscules et chaque réponse du candidat est donnée à la ligne suivante indentée.

susceptible de se trouver dans plusieurs sommets du graphe, nous utilisons la notion d'*ensemble d'information*.

3.2.2 L'ensemble d'information

Nous rappelons la définition 7 que nous avions donnée à la section 1.1.2 :

Définition. L'ensemble d'information *d'un joueur est l'ensemble des états du jeu dans lequel il est possible qu'il se trouve sans qu'il puisse les distinguer par observation directe.*

En GDL-II, l'ensemble d'information est donné de manière implicite par le Game Manager à travers les perceptions qu'il envoie à chaque tour du match. Supposons que lors d'un match, le joueur ait reçu n commandes **PLAY** et envoyé n coups. À la commande **PLAY** suivante, il dispose des informations suivantes sur la partie : les n coups qu'il a joué et les n ensembles de perceptions. Il existe plusieurs chemins dans le graphe de jeu dans lesquels le joueur a joué les mêmes coups et reçu les mêmes perceptions.

Par exemple au MONTYHALL, lorsque le candidat reçoit la commande **PLAY** à la ligne 7 de la figure 3.2, il sait qu'il a déjà joué `(choose 2)` puis `noop` et a reçu deux séries de perceptions : `(does candidate (choose 2))` au tour 1 puis `(does candidate noop)` et `(open_door 3)` au tour 2. Compte tenu de ces informations, il n'y a que deux historiques de jeu dans lesquels le candidat a joué les mêmes coups et **random** a joué des coups qui donnent les mêmes perceptions : soit **random** a joué `(hide_car 1)` puis `noop` ; soit il a joué `(hide_car 2)` puis `noop`. La voiture n'a pas pu être cachée derrière la porte 3 car cette hypothèse ne peut pas donner la perception

`(open_door 3)` compte tenu du fait que dans la description du jeu, la porte cachant la voiture n'est jamais ouverte à ce moment du jeu.

Nous pouvons ainsi définir précisément l'ensemble d'information en GDL-II :

Propriété 1. *L'ensemble d'information d'un joueur à un tour donné d'un match en GDL-II est l'ensemble des états du jeu qui ont le même historique de coups de ce joueur et pour lesquels il a reçu exactement la même série de perceptions.*

La première condition pour au moins jouer des coups légaux est de découvrir au moins un élément de cet ensemble d'information. C'est le rôle du jeu de l'ensemble d'information de les découvrir et que nous présentons dans la section suivante. En jouant à ce jeu, nous déterminons l'ensemble d'information qui permet de jouer de façon valide au jeu à information imparfaite.

3.2.3 La suite des jeux de l'ensemble d'information

Nous proposons de découvrir les éléments de la suite des ensembles d'information en définissant une suite de méta-jeux à information complète et parfaite dans lesquels le but est de trouver ces éléments.

Propriété 2. *Si deux états sont dans le même ensemble d'information au tour n, alors tous les états parents du tour* $n-1$ *appartenaient déjà à un même ensemble d'information.*

Les suites d'ensembles d'information pour les tours successifs d'un match ont donc une structure. En connaissant l'ensemble d'information au tour n, nous savons déjà que l'ensemble d'information au tour $n+1$ sera inclus dans l'ensemble des enfants des états de l'ensemble d'information du tour n.

Au tour n d'un jeu G, un joueur j a déjà joué une suite de coups $(c_1, c_2, \cdots c_n)$ et a reçu une suite de perceptions $(p_1, p_2, \cdots, p_n)$. Nous définissons le méta-jeu de l'ensemble d'information $\tilde{G}_n$ ayant les propriétés suivantes :

- c'est un jeu à information complète et parfaite à un seul joueur ;
- les états du jeu $\tilde{G}$ sont les mêmes que les états du jeu G jusqu'à une profondeur de $n+1$;
- un coup légal au tour k de $\tilde{G}$ est un coup-joint de G constitué du coup c_k qu'à joué le joueur j et de n'importe quelle combinaison de coups légaux des autres joueurs seulement si le choix de cette action dans le jeu G produit la perception p_k ;
- les états terminaux de $\tilde{G}$ sont : les états terminaux de G ; les états de profondeur $n+1$; les états n'ayant pas de successeur ;
- les récompenses sont minimales pour les états terminaux de profondeur inférieure ou égale à n et maximales pour les états terminaux de profondeur $n+1$.

En comparant deux jeux successifs, nous constatons que les positions terminales de $\tilde{G}_n$ restent terminales dans $\tilde{G}_{n+1}$ à l'exception de celles de profondeur $n+1$.

3.2.4 Exploration du jeu de l'ensemble d'information

Pour chaque joueur, l'évaluation que nous utilisons pour les positions terminales de $\tilde{G}_n$ est égale à la profondeur de ces nœuds dans l'arbre de jeu. À ce stade, nous pouvons utiliser n'importe quel algorithme d'exploration pour trouver les éléments de l'ensemble d'information. Chaque exploration d'une position terminale est suivie d'une remontée dans l'arbre et d'une mise à jour des évaluation des nœuds par le maximum entre leur valeur courante et l'évaluation de la position terminale. En particulier, si l'évaluation de la racine de $\tilde{G}_n$ vaut $n+1$ alors au moins un élément de l'ensemble d'information a été découvert et il suffit de suivre un chemin constitué de nœuds évalués à $n+1$ pour reconstituer un historique possible de la partie.

La découverte de ces historiques possibles de la partie doit être aussi complète que possible. À cette fin, nous marquons tous les sous-arbres explorés en totalité. L'étiquetage que nous utilisons est la valeur n lorsque de la recherche de l'ensemble d'information du jeu $\tilde{G}_n$. Ainsi, pour les générations suivantes de jeux $\tilde{G}_{n+i}$, un nœud marqué n signifie qu'il a été complètement évalué à la génération n.

Puisque les évaluations des nœuds terminaux restent constantes entre deux jeux de l'ensemble d'information successifs, l'évaluation de chaque nœud dans le jeu $\tilde{G}_n$ est une borne inférieure pour son évaluation dans le jeu $\tilde{G}_{n+1}$ et peut donc être utilisée pour guider la recherche d'une génération à la suivante. N'importe quelle fonction d'évaluation croissante avec la profondeur garantit cette propriété.

Quant aux nœuds complètement explorés de $\tilde{G}_n$, il le restent dans les générations suivantes $\tilde{G}_{n+i}$ dès lors que leur évaluation n'est pas plus grande que l'étiquette indiquant la génération à laquelle ils ont été complètement explorés. Cette propriété permet de connaître les nœuds qui ne peuvent pas faire partie d'un historique possible dans les générations suivantes de jeux de l'ensemble d'information.

3.2.5 Utilisation du jeu de l'ensemble d'information en information parfaite

En complétant un jeu de l'ensemble d'information $\tilde{G}_n$ à partir de ses feuilles avec l'arbre de jeu associé à sa description en GDL-II comme s'il s'agissait d'une description en GDL, nous obtenons un arbre de jeu complet que nous pouvons explorer avec les algorithmes que nous utilisons pour jouer aux jeux à information parfaite décrits en GDL à condition de partager les évaluations des

actions issues des positions appartenant au même ensemble d'information. Ces algorithmes sont présentés au chapitre suivant.

C'est cette approche qui a été utilisée lors de l'unique compétition internationale de GDL-II qui a eu lieu en Australie en décembre 2012 et qui a été remportée par CadiaPlayer. La principale limite de cette approche est que LeJoueur choisit ses actions en supposant que ses adversaires ont rétrospectivement choisi les meilleurs actions compte tenu de la connaissance de l'ensemble d'information courant ce qui se traduit par un comportement très défensif.

3.3 Conclusion

Nous avons présenté l'approche utilisée dans LeJoueur pour traiter les jeux à information imparfaite décrits en GDL-II. Elle repose sur la construction de l'ensemble d'information permettant de jouer correctement et le partage des évaluations des actions issues des positions appartenant au même ensemble d'information.

Chapitre 4

MCTS et UCT

Nous décrivons dans ce chapitre les algorithmes de recherche Monte-Carlo dans les arbres (MCTS) que nous utilisons dans LeJoueur. Nous décrivons dans un premier temps la classe des algorithmes MCTS et la politique UCT (*Upper Confidence bound applied to Trees*). Nous présentons ensuite la manière dont nous utilisons ces algorithmes dans le cas des jeux décrits en GDL qui sont à coup simultanés. Nous évoquons enfin l'utilisation de RAVE (*Rapid Action Value Evaluation*) en début de partie.

4.1 La recherche Monte-Carlo dans les arbres (MCTS)

4.1.1 Le développement des MCTS

La recherche Monte-Carlo dans les arbres ou MCTS (*Monte-Carlo Tree Search*) est une classe d'algorithmes dont le but est d'évaluer les sommets non-terminaux de l'arbre des états du jeu. L'évaluation fait appel à des simulations dans lesquelles les coups sont tirés aléatoirement jusqu'à l'obtention d'un état terminal du jeu. Par la suite, nous appellerons ces simulations des *playouts*. L'évaluation d'un sommet repose sur les résultats des playouts effectués à partir de ses descendants. Les MCTS ont gagné en popularité ces dernières années en permettant une amélioration significative des programmes de Go par rapport aux approches antérieures [27, 7, 8][59]. Depuis 2015, la combinaison des MCTS avec le Deep Learning a permis d'atteindre un niveau suffisant pour battre les meilleurs professionnels [68].

Des variantes ont été développées pour prendre en compte les jeux dans lesquels le facteur de branchement est important [6] ou lorsque peu d'évaluations sont disponibles, notamment en

[59] C'est l'utilisation des MCTS depuis 2006 qui a permis aux programmes de Go de passer du niveau amateur au niveau amateur fort (5ème dan).

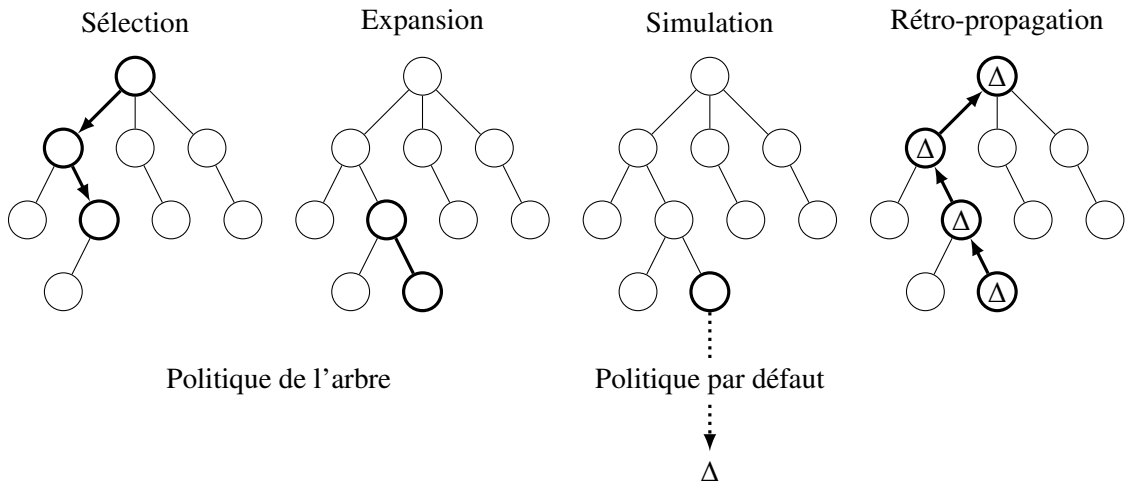


FIGURE 4.1 – Une itération de l'algorithme MCTS telle que présentée dans [10].

début de partie [27]. Dans une étude, Browne et al. fait un panorama des méthodes MCTS et de leur supériorité sur les méthodes alpha-beta dans la résolution de la plupart des problèmes [10][60]. Des algorithmes de recherche MCTS dans le cadre des jeux à information imparfaite ont aussi été développés [22, 39].

4.1.2 Modèle général d'un algorithme MCTS

Un algorithme MCTS permet la construction de l'arbre de jeu itérativement dans un ordre qui tient compte du résultat de parties jouées aléatoirement. Chaque cycle suit quatre phases illustrées par la figure 4.1 : la *sélection*, l'*expansion*, la *simulation* et la *rétro-propagation* [14].

La sélection s'effectue lors de la descente dans l'arbre à partir de la racine. À chaque sommet correspondant à un état du jeu, l'algorithme doit choisir parmi plusieurs actions possibles pour atteindre le sommet suivant selon les statistiques accumulées lors des itérations précédentes. Ce choix doit opérer un compromis entre l'exploitation et l'exploration des résultats accumulés. D'une part, l'exploitation permet de visiter le plus souvent les branches de l'arbre qui ont conduit à de bonnes évaluations. D'autre part, l'exploration doit aussi visiter des branches moins prometteuses du fait de l'incertitude des évaluations. Cette phase de sélection et la phase d'expansion qui suit constitue la *politique de l'arbre*.

[60] Les Échecs sont un des derniers remparts.

L'expansion a lieu lorsque l'étape précédente aboutit à un état du jeu absent de l'arbre. Dans ce cas, un nouveau sommet est ajouté à l'arbre[61].

La simulation consiste à poursuivre la partie sans ajouter de nouveaux sommets, en choisissant les actions aléatoirement. Cette étape s'achève lorsqu'un état terminal du jeu est atteint. Une évaluation de cet état terminal est alors disponible.

Pour chaque état du jeu rencontré, le tirage parmi les actions n'est pas forcément équiprobable. Des heuristiques peuvent servir à déterminer des probabilités pour ces actions. Cela permet de ne pas commencer à explorer tous les coups dans les jeux où ils sont trop nombreux et de prendre en compte les caractéristiques de la situation pour favoriser le tirage des coups qui semblent les plus prometteurs. Le choix des actions lors d'une simulation est déterminé par une *politique par défaut* dans l'algorithme.

La rétro-propagation met à jour l'évaluation des sommets visités lors de la phase de sélection ainsi que le sommet ajouté lors de l'expansion. Le nombre de visites est incrémenté et les espérances de gain sont mises à jour grâce à l'évaluation de l'état terminal de la simulation.

Le modèle général d'un algorithme MCTS est donné par l'algorithme 4.1 qui construit un arbre à partir d'une politique de l'arbre et d'une politique par défaut, en répétant les quatre étapes décrites précédemment.

Fonction `RechercheMCTS`(e_0)
  Créer un arbre de racine s_0 avec l'état e_0
  tant que *les ressources de calcul ne sont pas épuisées* **faire**
    $s_i \leftarrow$ `PolitiqueArbre`(s_0)
    $\Delta \leftarrow$ `PolitiqueParDéfaut`($e(s_i)$)
    `Rétropropagation`(s_i, Δ)
  fin
  retourner Δ

Algorithme 4.1 : Modèle général d'un algorithme MCTS.

4.1.3 La politique de l'arbre et les résultats de la théorie de la décision

Le choix d'un coup au niveau de chaque sommet de l'arbre peut être assimilé au problème classique du bandit manchot de la théorie de la décision. Le problème du bandit manchot à K bras

[61] Plusieurs sommets peuvent être ajoutés en même temps.

est défini par un ensemble de variables aléatoires $X_{i,n}$, où $i = 1..K$, indépendantes et identiquement distribuées. Les tirages successifs sur le bras i correspondent aux réalisations $X_{i,1}, X_{i,2}, \cdots$. Choisir un coup revient donc à choisir un bras i avec une espérance de gain $\mu_i = \mathbb{E}_n(X_{i,n})$ qui correspond à la valeur théorique du match commençant par ce coup.

4.1.4 *Upper Confidence Bound*

En 2002, Auer et al. propose une politique optimale de sélection des bras appelée UCB comme *Upper Confidence Bound* [2]. Elle consiste à choisir le bras qui maximise :

$$\mu_i + C \times \sqrt{\frac{\log N}{n_i}}$$

où N est le nombre total de tirages et n_i est le nombre de fois qu'a été tiré le bras de la machine i.

Le terme μ_i encourage l'utilisation des machines dont les tirages précédents ont donné de bons résultats en moyenne, qui correspond à l'exploitation. Le terme $\sqrt{\frac{\log N}{n_i}}$ autorise les essais sur les autres machines quand elles ont été peu choisies, ce qui correspond à l'exploration. La meilleure machine est jouée exponentiellement plus souvent que les autres.

La constante C qui n'apparaît pas dans l'article original permet d'ajuster le compromis entre exploration et exploitation[62].

Auer et al. montre qu'UCB permet d'approcher le regret logarithmique :

$$R_n = \mu^\star n - \sum_{j=1}^{n} \mu_j \mathbb{E}[T_j(n)]$$

uniformément et sans hypothèse préalable sur la loi de probabilité des machines. Ce regret mesure la perte issue du tirage sur les différents bras par rapport au gain cumulé maximal qui est obtenu en tirant tout le temps sur le meilleur bras.

4.1.5 Politique UCT pour l'exploration de l'arbre de jeu

En 2006, Kocsis et Szepesvári propose d'utiliser UCB comme politique de l'arbre dans l'algorithme MCTS sous le nom UCT : *Upper Confidence bound applied to Trees* [35]. Il applique UCB pour la sélection de la branche à explorer à chacun des sommets rencontrés lors de la descente[63].

[62] La recherche du meilleur compromis exploration-exploitation a permis le développement de nombreuses méthodes de «variation de la constante» qui laissent penser qu'il s'agit davantage d'un paramètre que d'une constante. Une des raisons de l'intérêt suscité par le *Sequential Halving* est qu'il n'y a pas de constante à ajuster [34].

[63] Le premier programme de Go à populariser les MCTS était *Crazystone* de Rémi Coulom et n'utilisait pas UCT [21].

Il démontre que le glissement des espérances de gain mesurées par les playouts à mesure que l'arbre est construit n'empêche pas la convergence d'UCT vers la politique optimale : le coup choisi par minimax. Le terme $\sqrt{\frac{\log N}{n_i}}$ diminue en effet avec le nombre d'explorations dans la branche la plus explorée et augmente dans les autres ce qui garantit asymptotiquement une exploration intégrale de l'arbre. Le terme $\log N$ de son côté garantit que la branche optimale sera explorée exponentiellement plus souvent que les autres, ce qui fait converger l'évaluation du nœud parent vers celle de son meilleur enfant.

Des alternatives ont été proposées utilisant le *Sequential Halving*, qui minimise la probabilité de choisir la mauvaise machine dans le problème du bandit, en remplacement de la politique UCB [52, 11].

LeJoueur utilise UCT pour explorer le graphe du jeu combiné à une évaluation exacte des positions par l'algorithme alpha-beta dans le cas des jeux à information parfaite, une combinaison déjà proposée sous le nom de MCTS-Solver [75].

4.2 Adaptation des méthodes MCTS pour les jeux à coups simultanés

Les méthodes MCTS ont été principalement développées pour les jeux à coups alternés à deux joueurs. Les jeux en GGP peuvent présenter des coups simultanés ce que nécessite une adaptation que nous décrivons ici.

4.2.1 Problème de l'arité de l'arbre de recherche pour les jeux à coups simultanés

Le langage GDL modélise les jeux pour lesquels les joueurs jouent de manière synchrone et chacun doit obligatoirement fournir un coup pour chaque position non terminale afin de passer à la suivante. Il en résulte que la transition entre deux positions d'un jeux se fait par un coup simultané de chacun des joueurs ou *coup joint*. Dans la construction de l'arbre de jeu qui en découle, chaque position possède un nombre d'enfants égal au nombre de coups joints.

Dans ces conditions, les concepteurs de descriptions de jeux modélisent les jeux à coup alternés en faisant en sorte qu'à chaque tour, au plus un joueur dispose de plus d'un coup légal. Par exemple, dans le cas d'un jeu à deux joueurs, chacun d'entre eux aura un tour sur deux un seul coup légal possible, souvent appelé `noop` dans les descriptions GDL. Si tous les joueurs n'ont qu'un seul coup légal, alors il n'y a qu'un seul nœud enfant et les joueurs n'ont donc aucune influence sur le déroulement du jeu à cette position. Sinon, il n'y a qu'un seul joueur qui a plus d'un coup légal et donc le nombre de coups joints est égal au nombre de coups de ce joueur.

Pour les jeux à coups simultanés, le nombre de coups joints légaux à chaque tour est égal au produit des nombres de coups légaux de chacun des joueurs. Comparé à l'arbre d'un jeu à n joueurs à coups alternés d'arité moyenne l, l'arbre d'un jeu similaire à coups simultanés possédera une arité moyenne égale à a^n. En utilisant une méthode de Monte-Carlo pour visiter cet arbre, chaque nœud découvert servira de position initiale pour une partie aléatoire dont le résultat sera utilisé pour déterminer le nœud suivant à développer. Lorsque l'arité de l'arbre devient trop grande relativement à la vitesse de simulation, la profondeur d'exploration se réduit et la méthode perd de son efficacité. C'est d'autant plus préjudiciable dans le cas des jeux à coups simultanés pour lesquels les coups légaux d'un joueur sont indépendants de ceux des autres.

C'est le cas de CHINOOK, un jeu composé de deux variantes du jeu de dames, la première se jouant sur les cases blanches et la seconde sur les cases noires[64]. À chaque position, un joueur joue avec les cases blanches pendant que l'autre joue sur les cases noires, la situation s'inversant à chaque tour. Il suffit de gagner dans l'une des composantes pour remporter la victoire. Pour ce jeu, l'arité moyenne de l'arbre de jeu est le carré de l'arité moyenne d'une composante et pourtant les deux composantes sont clairement indépendantes à la terminaison du jeu près. La seule dépendance qui existe est que la victoire dans une des composantes met fin à la partie dans la seconde et détermine son score. Pour un jeu comme CHINOOK, estimer la valeur d'un coup joint est clairement préjudiciable car les coups de chaque joueur sont indépendants.

Nous allons à présent exposer la méthode que nous utilisons pour augmenter la profondeur d'exploration dans le cas de jeux à coups simultanés indépendants et les conséquences de cette méthode pour les autres types de jeux.

4.2.2 Recherche MCTS pour les jeux à coups simultanés

Pour réduire l'arité des jeux à coups simultanés, nous construisons l'arbre en alternant les coups et en partageant l'information recueillie par l'exploration MCTS. Décrivons tout d'abord ce qui se passe dans le cas d'un arbre construit avec les coups joints. Nous nous limiterons au cas de deux joueurs (que nous appelons Alice et Bob), la généralisation à n joueurs n'étant pas compliquée.

Pour une position donnée, imaginons qu'Alice a deux coups légaux (a_1 et a_2) et que Bob en a trois (b_1, b_2 et b_3). Il y a donc six coups joints légaux (par exemple (a_2, b_1) en est un) qui mènent à six autres positions. Le tableau suivant résume le nombre de fois qu'à été joué chaque coup joint

[64] CHINOOK a été conçu explicitement pour pouvoir rendre difficile la séparation des deux sous-jeux. Le programme ne peut pas identifier deux damiers différents où se dérouleraient les deux sous-parties. Les deux sous-parties se déroulent sur des cases différentes du même damier en exploitant le fait que les pièces au jeu de dame se déplacent sur des cases ayant toujours la même couleur.

lors de l'exploration MCTS ainsi que les sommes en ligne et en colonne des résultats obtenus.

	b_1	b_2	b_3	somme
a_1	21	79	5	105
a_2	7	12	2	21
somme	28	91	7	126

Le tableau indique par exemple que le coup joint (a_2, b_1) a été joué sept fois. Alice et Bob doivent chacun choisir un coup sans connaître celui de leur adversaire. Supposons qu'ils choisissent leur coup en estimant que le meilleur est celui ayant reçu le plus de visites. Alice choisit le coup a_1 qui a été visité 105 fois au total alors que Bob choisit le coup b_2 visité 91 fois. Nous constatons que le détail du nombre de visites pour chaque coup joint n'est pas nécessaire pour prendre la décision, seules les sommes sont utiles.

Il en est de même lors d'une recherche Monte-Carlo pure. En se plaçant du point de vue de Bob et en considérant le nombre de victoires obtenues pour chacun des coups joints, nous obtenons le tableau suivant des probabilités estimées de victoires pour Bob :

	b_1	b_2	b_3	somme
a_1	$P(a_1, b_1)$	$P(a_1, b_2)$	$P(a_1, b_3)$	$P(a_1)$
a_2	$P(a_2, b_1)$	$P(a_2, b_2)$	$P(a_2, b_3)$	$P(a_2)$
somme	$P(b_1)$	$P(b_2)$	$P(b_3)$	1

Lors de l'exploration, Bob choisit son coup en fonction des probabilités de gain estimées $P(b_1)$, $P(b_2)$ et $P(b_3)$[65]. Bob n'utilise pas les probabilités jointes $P(a_i, b_j)$ pour prendre sa décision ; il s'en suit qu'il n'a besoin de connaître que trois probabilités plutôt que six. Pour une recherche MCTS, il est donc suffisant d'estimer les probabilités $P(a_i)$ pour Alice et $P(b_i)$ pour Bob. Il n'y a pas besoin d'estimer les probabilités jointes $P(a_i, b_j)$. En pratique, lors de l'exploration MCTS, il suffit de faire remonter et partager les victoires au niveau du coup de chaque joueur et non au niveau du coup joint.

4.2.3 Réduction de l'arité par linéarisation des coups-joints

Une autre manière d'aborder ce problème est de linéariser les coups joints. Cela consiste à modifier l'arbre en faisant jouer Alice et Bob l'un après l'autre, en créant des positions intermédiaires. L'arbre initial comporte la position courante et six arcs vers les six positions suivantes. L'arbre modifié possède la même position courante d'où partent deux arcs qui aboutissent à deux

[65] Si le résultat du jeu n'est pas binaire (victoire ou défaite) mais un score entre 0 et 100 comme c'est le cas en GGP, il suffit de remplacer les probabilités estimées de gain par l'espérance du score.

positions intermédiaires matérialisant le fait qu'Alice a choisi son coup. De chaque position intermédiaire partent trois arcs correspondant au choix de Bob et aboutissent aux six positions suivantes de l'arbre initial. Une exploration Monte-Carlo sur cet arbre revient à considérer que Bob choisit son coup en connaissant préalablement le coup d'Alice et le tableau des probabilités précédent est modifié ainsi :

	b_1	b_2	b_3	somme
a_1	$P(b_1\|a_1)$	$P(b_2\|a_1)$	$P(b_3\|a_1)$	1
a_2	$P(b_1\|a_2)$	$P(b_2\|a_2)$	$P(b_3\|a_2)$	1

En linéarisant les coups joints, la méthode MCTS estime pour Alice les probabilités $P(a_i)$, alors que pour Bob, elle estime les probabilités jointes $P(b_j|a_i)$. Évidemment un joueur GGP ayant le rôle de Bob ne connaît pas la décision d'Alice. Il ne pourra donc estimer sa probabilité de gagner en choisissant choisissant le coup b_j que par :

$$P(b_j) = P(b_j|a_1).P(a_1) + P(b_j|a_2).P(a_2)$$

En procédant ainsi, nous sommes ramenés au cas décrit à la section précédente mais seulement du fait que Bob joue en second.

En effet, imaginons qu'un joueur GGP a le rôle d'Alice qui joue en premier. Il peut estimer directement ses probabilités de gagner $P(a_i)$. Il pourra aussi estimer les choix de Bob selon les probabilités $P(b_j|a_i)$. Par conséquent, tout se passe comme dans un jeu à coups alternés ou pour chaque coup-joint, Bob connaît déjà la décision d'Alice. Avec cette manière de jouer, Alice essaie de parer au pire en supposant que Bob triche en connaissant son coup.

Dans un jeu comme CHINOOK, si nous faisons l'hypothèse raisonnable que les coups de Bob sont indépendants des coups d'Alice, la probabilité de gain pour Bob devient alors :

$$P(b_j) = P(b_j|a_1) = P(b_j|a_2)$$

Dans le tableau précédent, cela revient à dire qu'il n'y a pas six probabilités conditionnelles distinctes mais seulement trois. Dans ce cas, la connaissance du coup d'Alice par Bob n'a pas d'influence sur son choix. Nous pouvons donc réaliser une exploration MCTS sur l'arbre où les coups joints ont été linéarisés et obtenir exactement les mêmes résultats que pour l'arbre initial à trois conditions :

1. les probabilités de gain $P(b_j|a_1)$ et $P(b_j|a_2)$ pour Bob doivent être partagées (ce qui revient à estimer directement $P(b_j)$) ;
2. les parties aléatoires (*playouts*) ne doivent pas débuter sur une des positions intermédiaires introduites lors de la linéarisation des coups joints ;
3. toutes les positions intermédiaires introduites doivent avoir été explorées avant de descendre plus bas dans l'arbre.

La condition nº1, en estimant directement la probabilité $P(b_j)$, fait qu'Alice n'essaie plus de parer au pire car Bob ignore la décision d'Alice pour prendre la sienne. Nous avons choisi de respecter cette condition dans LeJoueur car parer au pire en considérant que les autres joueurs connaissent notre décision conduit à des estimations trop pessimistes et à l'extrême, l'estimation que la partie est toujours perdante[66].

Nous respectons la condition nº2 dans LeJoueur afin de garantir qu'il n'y ait pas plus d'un *playout* par coup joint.

C'est en ne respectant pas la condition nº3 que la recherche MCTS permet une exploration à une profondeur plus importante. C'est le choix que nous avons fait dans LeJoueur. En effet, l'exploration avec UCT suppose d'avoir visité toujours les enfants d'un nœud avant de pouvoir explorer plus bas dans l'arbre[67]. Par conséquent, sur l'arbre initial, lorsque le premier joueur dispose de a coups légaux, et que le second en a b, il faut explorer $a \times b$ positions avant de pouvoir explorer l'arbre plus en profondeur. Sur l'arbre avec les coups-joints linéarisés, il suffit d'avoir exploré tous les coups de chaque joueur pour pouvoir explorer plus bas, ce qui correspond à $a + b$ positions.

4.2.4 Conséquences de la linéarisation des coups-joints dans LeJoueur

Dans le cas de jeux à coups simultanés comme chinook qui sont en fait une combinaison de deux jeux, le gain en exploration est important. Cette approche est une solution intermédiaire entre l'exploration coûteuse de tous les coups joints et la décomposition du jeu. Il n'y a pas besoin d'analyser les règles et découvrir que le jeu est décomposable pour pouvoir exploiter cette propriété. Plus généralement, cette méthode est efficace pour tous les jeux à coups simultanés où il y a une forte indépendance entre les coups de chaque joueur dans un coup-joint.

Dans les jeux où cette indépendance n'est pas vérifiée, il peut se produire qu'un bon coup joint ne soit pas exploré du fait que chacun des coups qui le compose a une estimation de victoire faible dans l'estimation d'autres coups joints. C'est un compromis que nous avons fait dans LeJoueur, sans doute du fait que la plupart des jeux à coups simultanés dans les compétitions étaient en fait des combinaisons de jeux sur le modèle de chinook[68]. Lorsque l'hypothèse d'indépendance est violée, il est nécessaire d'utiliser une stratégie mixte, ce que LeJoueur ne fait pas à ce jour. Avec

[66] Pour les jeux à plusieurs joueurs et à coups simultanés, il est courant d'adapter une stratégie *paranoïaque* dans laquelle les adversaires ont connaissance de notre coup avant de choisir le leur. Ceci permet de multiplier les coupes alpha-beta, d'agrandir la profondeur de recherche et débouche sur une amélioration du niveau de jeu [70, 69].

[67] Le calcul du gain moyen $\mu = \frac{\sum s_i}{n_i}$ n'a pas de sens si $n_i = 0$.

[68] Les jeux composés ont été très à la mode dans les compétitions IGGPC en 2013 et 2014.

la méthode actuelle, il découvre les coups dominants, rejette les coups dominés et oscille entre plusieurs coups lorsque la solution est une stratégie mixte[69].

4.3 Les tables de transposition

4.3.1 Définition des tables de transposition

Les tables de transposition sont une technique de programmation dynamique qui permet de mettre en commun des positions qui sont identiques mais qui n'ont pas le même historique de coups. C'est par exemple souvent le cas dans des jeux se jouant sur un damier où une séquence de coups non perturbée par le joueur adverse aboutit à la même position quel que soit l'ordre dans lequel ils sont joués.

Quelle que soit l'implémentation, l'utilisation de table de transpositions permet d'explorer plus rapidement l'arbre de jeu. Cela a cependant des conséquences sur les algorithmes d'exploration d'arbres de jeux qui doivent alors prendre en compte ces transpositions.

4.3.2 Unicité des positions dans LeJoueur

Par l'application des règles GDL, nous envisageons deux manières de construire un graphe représentant toutes les parties qu'il est possible de jouer. La première approche permet donc d'obtenir un graphe orienté sans circuit et la seconde un arbre.

La première approche consiste à considérer qu'une position est l'ensemble des fluents `(true ?x)` vrais au début du jeu et chaque fois que les joueurs ont choisi leur coup. Nous pouvons alors construire un graphe qui a pour sommets les différentes positions possibles du jeu. Ces sommets sont reliés par des arcs représentant les coups-joints permettant de passer d'une position à une autre. Dans ce cas, le langage GDL impose que le graphe ainsi construit soit fini, orienté et acyclique. Il est fini car la largeur du graphe, qui correspond au nombre de coups joués dans la plus longue partie, doit être fini. Il est trivialement orienté. Il est acyclique car sinon, il serait possible de jouer une partie infinie alors que c'est interdit par les spécifications du langage. Il est tout à fait possible qu'un sommet possède plus d'un arc incident issu de positions distinctes. Autrement dit, deux séries de coups distinctes passant par des positions distinctes peuvent aboutir à une position commune. Notons qu'en GDL, l'historique d'une position n'a aucune importance car toute

[69] Une stratégie mixte est un ensemble de stratégies auxquelles sont attribuées des probabilités. Le choix d'un coup avec la stratégie mixte consiste d'abord à tirer au sort une des stratégies selon leurs probabilités puis de sélectionner le coup conformément à la stratégie sélectionnée.

l'information nécessaire à l'application des règles réside dans la position[70]. Par exemple dans le cas des Échecs, deux positions identiques sur l'échiquier mais où pour l'une des deux, l'historique contient un roque, seront distinctes en GDL car l'information sur le roque est stockée dans la position.

La seconde manière d'envisager une représentation de l'ensemble de parties possibles est de considérer cette fois-ci qu'une position est réduite à l'historique des coups joués. Dans ce cas, nous obtenons un arbre. Il y a une bijection entre l'ensemble des graphes orientés sans circuits (DAG pour *directed acyclic graph*) et l'ensemble des arbres de jeu décrits précédemment. En effet, pour passer de l'arbre au DAG, il suffit de contracter les sommets dont la suite de coups conduit à la même position au sens de l'ensemble des fluents `(true ?x)`. Inversement, pour passer du DAG à l'arbre, il suffit de dupliquer tous les sommets ayant plus d'un prédécesseur jusqu'à ce qu'il ne leur en reste plus qu'un.

La représentation sous la forme d'un arbre revient à considérer comme distinctes des positions du jeu qui possèdent les mêmes fluents `(true ?x)`. Étant donné que cet ensemble de fluents représente un état du jeu et que l'automate modélisé par la description GDL n'a pas de mémoire, ces positions ont par conséquent exactement la même valeur.

LeJoueur utilise la première approche et stocke les positions dans une table de hachage ce qui garantit leur unicité et permet d'optimiser l'encombrement en mémoire. De ce fait, il utilise par défaut ce qui est communément appelé une table de transposition.

Nous allons étudier à présent les conséquences de l'utilisation d'un graphe orienté sans circuit sur les algorithmes prévus pour des arbres.

4.3.3 Adaptation de la politique UCT pour les graphes orientés sans circuit

Avec les tables de transposition, le calcul du score UCT se fait en utilisant un nombre de visites de l'enfant qui peut être différent du nombre de rétro-propagations des valeurs du nœud parent à partir de celle de l'enfant : de ce fait, nous perdons la garantie que l'utilisation d'UCT minimise le regret.

Childs et al. analyse la situation sur des jeux synthétiques : ils montrent que les valeurs mesurées par les playouts doivent être attachées aux arcs et non aux nœuds et proposent une mise à jour systématique des valeurs de tous les prédécesseurs possibles ce qui pose des problèmes d'implémentation et ne résout pas celui des transpositions découvertes a posteriori [18].

[70]Comme souvent en GGP, la formulation de la description est cruciale. Il est facile de conserver dans l'état courant les coups précédemment joués avec les autres fluents. Les transpositions deviennent alors impossibles sans une analyse difficile qui permette de détecter les fluents qui sont inutiles dans la description de l'état courant.

Saffidine et al. étudient une version dans laquelle les transpositions ne sont utilisées que pour la politique de l'arbre ; ils paramétrisent l'algorithme avec des profondeurs différentes pour la valeur moyenne, le nombre de visites du successeur et la valeur RAVE. Les résultats sur le jeu Hex, Go6x6 et différents jeux communs du GGP montrent des résultats significativement différents selon la valeur des paramètres mais n'aboutissent pas à une conclusion nette quant aux paramètres à choisir pour un jeu donné [64].

Dans LeJoueur, les statistiques sur le nombre de visites et les scores accumulés sont stockées au niveau des arcs. Lorsqu'est découverte une position qui possède parmi ses successeurs une position transposée, les statistiques stockées dans l'arc reliant ces deux sommets sont accumulées comme d'ordinaire. Par conséquent, le coup menant à la position transposée bénéficie d'une évaluation dans laquelle les scores ont davantage dérivé que pour les coups menant aux autres successeurs.

4.4 L'évaluation RAVE

Dans les programmes de Go, UCT est souvent enrichi et même parfois remplacé par une variante nommée RAVE. Le principe de RAVE est une extension de *All Moves As First* [48].

Avec un UCT classique, chaque coup depuis un nœud doit être exploré au moins une fois avant de commencer à obtenir des mesures significatives sur les résultats des playouts : RAVE, acronyme de *Rapid Action Value Evaluation* est destiné à obtenir beaucoup plus rapidement des évaluations, moins précises que celle d'UCT.

Chaque nœud parent contient une table RAVE où tous les coups légaux sont associés avec les résultats des playouts qui incluent ce coup, même si ce n'est pas le premier coup du playout. Comme un playout est composé usuellement de plusieurs coups, cette table est remplie beaucoup plus rapidement que la table UCT qui ne collecte qu'une seule valeur pour chaque playout. Dans les jeux où les transpositions de coups sont fréquemment possibles sans modifier significativement l'issue de la partie, la moyenne des résultats des playouts associés à un coup dans la table RAVE sont proches de la valeur associée avec le coup dans la table UCT.

L'évaluation d'un coup par UCT e_u et son évaluation par RAVE e_r sont combinés pour choisir le successeur à explorer en prenant principalement en compte e_r quand le nombre de visites du successeur est faible et e_u quand il devient plus important.

En pratique, RAVE donne d'excellents résultats au jeu de Go, au point que certains programmes l'ont utilisé exclusivement en remplacement d'UCT.

Dans le cadre du GGP, les transpositions peuvent avoir une influence significative sur l'issue d'une partie et l'analyse de la description pour le déterminer est difficile. Nous avons procédé à

des expériences pour déterminer en cours de partie si l'utilisation de RAVE peut être utile, en se fondant sur les rapports entre e_r et e_u mais les résultats obtenus sur des jeux synthétiques ne se reproduisent pas bien sur les jeux réels utilisés en GGP [48]. Pour ces raisons, LeJoueur n'utilise pas RAVE.

4.5 Conclusion

Nous avons présenté dans ce chapitre la classe des algorithmes MCTS et la politique UCT que nous utilisons dans LeJoueur pour l'exploration de l'arbre de jeu. Nous avons décrit la manière dont nous les adaptons au cas des jeux décrits en GDL qui sont à coups simultanés. Nous avons enfin testé la politique RAVE sur des jeux synthétiques mais sans envisager de l'intégrer dans LeJoueur.

Chapitre 5

Parallélisation du programme LeJoueur

Nous présentons dans ce chapitre les différents aspects de la parallélisation que nous avons introduit dans LeJoueur dans le but d'accélérer la construction du circuit logique permettant d'interpréter les règles du jeu, le calcul des *playouts* et la construction de l'arbre de jeu. Nous distinguons pour cela les Traitements Avant un Match (TAM) des Traitements Pendant un Match (TPM).

Une fois introduite la problématique de la parallélisation, nous présentons la machine de Warren que nous avons développée pour l'interprétation du GDL et ses potentialités en termes de parallélisation de l'instanciation des règles : il s'agit d'un TAM. Nous exposons ensuite la parallélisation de la simulation de parties par le circuit décrit à la section 2.3.2 : un TAM analysant le circuit logique dans le but d'évaluer simultanément plusieurs portions de ce circuit ; un TPM qui permet une parallélisation sur un seul thread du calcul des *playouts* permettant de l'accélérer d'environ deux ordres de grandeur par rapport à l'interprétation par Prolog ; un autre TPM permettant une parallélisation multi-threads sur CPU, GPU et sur un processeur *many cores*. La dernière partie du chapitre décrit un TPM permettant une parallélisation multi-threads de l'exploration de l'arbre de jeu avec une méthode MCTS.

5.1 Problématique de la parallélisation

Flynn propose une taxonomie des architectures des ordinateurs en quatre classes [26] suivant qu'elles utilisent un ou plusieurs flux d'instruction (instruction stream) et un ou plusieurs flux de données (data stream) : SISD (single-instruction stream, single-data stream) ; SIMD (single-instruction stream, multiple-data stream) ; MISD (multiple-instruction stream, single-data stream) ; MIMD (multiple-instruction stream, multiple-data stream). De nos jours, les architectures basées sur le x86 ou PowerPC comportent à la fois des jeux d'instructions permettant de traiter plusieurs données à la fois et de multiples cœurs qui traitent des flux d'instructions indépendants donc sont

de type MIMD. En fonction du programme qui s'exécute sur ces machines, nous pourrons alors dire que celui-ci appartient à une de ces quatre classes.

5.1.1 Parallélisation *instruction unique sur des données multiples*

Les architectures SIMD peuvent traiter plusieurs flux de données avec un seul flux d'instruction. Les processeurs vectoriels conçus dès les années 70 sont alors réservés à des supercalculateurs. À partir des années 90, devant le besoin croissant de traitement des images et du son, cette architecture devient accessible au grand public. C'est le cas en particulier des processeurs graphiques (GPU) qui sont dotés d'instructions qui manipulent des vecteurs et pour lesquels un seul programme permet de traiter plusieurs pixels d'une image simultanément et de manière synchrone. Les processeurs de la famille x86 et PowerPC sont étendus pour incorporer des jeux d'instructions SIMD notamment pour accélérer le traitement des données comme les images et les sons. Ces jeux d'instructions additionnels s'accroissent au fil des générations de processeurs. Pour ceux basés sur le x86, après les instructions MMX introduites en 1997, se sont ajoutées entre autres SSE et AVX. Le jeu d'instructions Altivec a été ajouté aux processeurs PowerPC à partir du processeur G4 en 1999.

La plupart des instructions SIMD de ces extensions permettent de manipuler des données vectorielles. Un registre peut contenir plusieurs données (par exemple 4 nombres à virgule flottante de 32 bits dans un registre de 128 bits). Il est alors possible d'effectuer le même calcul sur chacune de ces données indépendantes en une seule instruction. Lorsque le flux de données est constitué de bits, on peut aussi considérer que les instructions logiques bit à bit sont des instructions SIMD. Lorsque qu'elles opèrent sur des mots de n bits, il est possible de réaliser n opérations en parallèle avec une seule instruction.

Tous les algorithmes ne se prêtent pas facilement à une implémentation sur une architecture SIMD. Les principales contraintes étant l'indépendance des données traitées simultanément et que ces données sont traitées de manière synchrone.

5.1.2 Processeurs multi-cœurs et multi-threading

Un processeur multi-cœurs est un composant possédant au moins deux unités de calcul appelées cœurs. Chaque cœur est capable de lire et d'exécuter ses propres instructions et permet donc des calculs en parallèle. Chaque cœur est une unité de calcul complète. Il dispose de ses propres registres, son pointeur d'instructions et son unité de traitement. Ils accèdent tous à la même mémoire centrale (RAM) à travers des caches qui peuvent être partagés ou distribués.

Un processus est un programme s'exécutant sur un des cœurs composé notamment d'un ensemble d'instructions à exécuter et d'un espace d'adressage en mémoire. L'espace d'adressage est généralement séparé en deux parties : la pile d'exécution qui permet d'allouer de la mémoire lors de l'exécution d'une fonction et le tas qui permet de stocker des données entre les appels de fonctions. Plusieurs processus peuvent être en exécution en même temps et c'est l'ordonnanceur qui gère le temps pendant lequel ils pourront accéder à l'unité de calcul. Les processus peuvent communiquer entre eux mais ne partagent en général pas leurs ressources. Au sein d'un processus, il est possible d'exécuter des threads (ou processus légers) qui partagent les ressources du processus qui les a lancés mais qui disposent de leur propre pile. La programmation parallèle à processus (ou threads) multiples nécessite de partager de la mémoire, de synchroniser et faire communiquer les processus entre eux [46].

Le multi-threading est une technique matérielle permettant d'exécuter simultanément plusieurs threads sur un même cœur physique. Lorsque le processeur le permet, le système d'exploitation ordonnance les exécutions sur des cœurs logiques qui partagent un même cœur physique. Un cœur physique dispose d'une seule unité de traitement mais de plusieurs jeux de registres et pointeurs d'instruction. Deux instructions peuvent alors s'exécuter en parallèle lorsqu'elles font appel à deux parties distinctes de l'unité de traitement.

5.2 Parallélisation de l'instanciation des règles GDL

5.2.1 Parallélisation de l'interprétation Prolog

Nous traitons dans cette section de parallélisation implicite de programmes logiques, ce qui signifie que des programmes traités sont écrits dans la sémantique séquentielle de Prolog mais que la machine qui les exécute utilise le parallélisme pour accélérer leur exécution.

Il y a deux principales forme de parallélisme implicite en Prolog [59]. La première est la *parallélisation-et* qui correspond à l'évaluation en parallèle des différentes prémisses d'une clause et la *parallélisation-ou* qui correspond à l'évaluation en parallèle des différentes clauses d'une procédure. Cette dernière est en principe plus simple car elle correspond à des branches distinctes de l'arbre de recherche et impose donc moins de synchronisation que la *parallélisation-et*.

5.2.1.1 Fonctionnement général d'une machine de Warren

Nous avons montré aux sections 2.2 et 2.3.1 que nous utilisons un interpréteur Prolog pour l'instanciation des règles nécessaires à la construction d'un circuit logique et pour l'évaluation

de la description GDL lorsque le circuit n'est pas prêt ou trop coûteux à calculer dans des temps compatibles avec le GGP.

Nous utilisons deux interpréteurs : YAPPROLOG [20] et une implémentation d'une machine de Warren que nous avons réalisée [1, 74]. La motivation qui nous a conduit à créer notre propre machine de Warren pour l'interprétation du GDL est que YAPPROLOG est un système Prolog complet et qu'à ce titre, il est optimisé pour le langage Prolog. Même s'il s'agit d'un projet *open source*, sa modification n'est pas aisée[71]. Ces deux interpréteurs fonctionnent de manière similaire car ils implémentent une machine abstraite dérivée de celle proposée par Warren dont nous esquissons à présent le fonctionnement général.

Une machine de Warren exécute du code compilé dans un jeu d'instructions spécifique qui manipule des données en mémoire. Celle-ci est constitué de plusieurs piles[72] :

— la *pile de registres* qui permettent de spécifier les arguments des requêtes ;
— *le tas* (heap) qui permet de stocker des termes structurés ;
— *la pile* (stack) qui permet de mémoriser les frames d'environnement qui correspondent à l'évaluation d'une clause et les points de choix qui correspondent aux disjonctions lorsqu'une procédure est constituée de plusieurs clauses ;
— *la pile de suivi* (trail) qui sauvegarde les variables qui ont été unifiées pour pouvoir défaire ces unifications lors d'un retour en arrière.

L'ensemble des données stockées dans ces structures permet de parcourir un ensemble de recherche pour toute requête formulée à l'interpréteur.

Le code pour la machine est constitué d'instructions qui permettent de construire des termes structurés, d'unifier des termes et de faire des branchements.

5.2.1.2 Exemple de code compilé pour une machine de Warren

La figure 5.1 présente un exemple de compilation de la procédure **`next`** extraite de la description GDL du jeu CONNECT5. La procédure est constituée de trois clauses. La première est compilée aux adresses 1818–1834, la deuxième aux adresses 1837–1861 et la dernière aux adresses 1863–1906.[73]

[71] Nous avons tout de même porté une version de YAPPROLOG qui est un programme écrit en C en un programme C++ dans lequel il est possible de lancer plusieurs instances de YAPPROLOG en même temps car tout le code a été intégré dans une unique classe.

[72] Nous ne mentionnons pas le *PDL* dont parle Ait-Kaci car cette pile sert temporairement lors de l'unification de deux termes et la pile des environnements et des points de choix peut servir à cela

[73] La ligne 1803 est réservée pour y placer une instruction si l'on souhaite tabler la procédure. Les adresses 1804–1815 contiennent du code qui permet d'optimiser l'accès à une clause grâce à une indexation du premier paramètre de la conclusion. Dans ce code, le premier paramètre est une structure avec deux possibilités de symbole de fonction : `control` et `cell`.

```
1 (<= (next (control ?r)) (does ?r noop))
2 (<= (next (cell ?x ?y ?r)) (does ?r (mark ?x ?y)))
3 (<= (next (cell ?x ?y ?c)) (true (cell ?x ?y ?c)) (does ?r (mark ?x1 ?y1))
4         (or (distinct ?x ?x1) (distinct ?y ?y1)))
```

```
;[next/1]
; adr. operation

  1803 noop
  1804 switch_on_term 1816
  1806 switch_on_functor 2
        [cell/3]
        [control/1]
        1812
  1812 try 1837
  1814 trust 1863
  1816 try_me_else 1835
  1818 get_structure [control/1] A1
  1821 unify_variable X3
  1823 get_variable X4 A1
  1826 put_value X3 A1
  1829 put_constant [noop/0] X2
  1832 execute [does/2]
  1834 proceed
  1835 retry_me_else 1862
  1837 get_structure [cell/3] A1
  1840 unify_variable X4
  1842 unify_variable X5
  1844 unify_variable X3
  1846 get_variable X6 A1
  1849 put_structure [mark/2] A2
  1852 set_value X4
  1854 set_value X5
  1856 put_value X3 A1
  1859 execute [does/2]
  1861 proceed
  1862 trust_me
  1863 allocate
  1864 get_structure [cell/3] A1
  1867 unify_variable Y3
  1869 unify_variable Y4
  1871 unify_void 1
  1873 call [true/1] 4
  1876 put_structure [mark/2] A2
  1879 set_variable Y1
  1881 set_variable Y2
  1883 put_variable X1 A1
  1886 call [does/2] 4
  1889 put_structure [distinct/2] A1
  1892 set_local_value Y3
  1894 set_local_value Y1
  1896 put_structure [distinct/2] A2
  1899 set_local_value Y4
  1901 set_local_value Y2
  1903 deallocate
  1904 execute [or/2]
  1906 proceed
```

FIGURE 5.1 – Description de la procédure **next** extraite de la description GDL de CONNECT5 et son code compilé pour la machine de Warren de LEJOUEUR.

Pour compiler une clause, on compile successivement la conclusion et la suite des prémisses. Des variables sont associées à chaque terme et chaque sous terme de la clause. Il y en a deux types : les variables temporaires notées `Xi` et les variables permanentes notées `Yi`. `Ai` est un autre nom donné à `Xi` lorsqu'il est question d'un argument.

La compilation de la conclusion d'une clause (ou d'un simple fait) s'effectue argument par argument du terme en entier vers ses sous-termes. Elle utilise des instructions commençant par `get` et `unify` qui unifient les arguments passés par les registres `Ai` avec les paramètres de la conclusion. Le compilateur a attribué les variables suivantes pour la conclusion de la troisième clause : `A1` pour `(cell ?x ?y ?c)` ; `Y3` pour `?x` ; `Y4` pour `?y`. Il n'attribue pas de variable à `?c` car cela n'est pas nécessaire.[74]

La compilation d'une prémisse s'effectue aussi argument par argument mais en commençant par les sous-termes les plus intérieurs. Elle utilise des instructions commençant par `put` et `set` qui construisent les arguments de la requête. Le compilateur a attribué les variables suivantes à la deuxième prémisse de la troisième clause : `A1` pour `?r` ; `Y1` pour `?x1` ; `Y2` pour `?y1` ; `A2` pour `(mark ?x1 ?y1)`.

La disjonction entre les trois clauses de la procédure **`next`** est assurée par trois instructions aux lignes 1816, 1835 et 1862 : `try_me_else`, `retry_me_else` et `trust_me`. L'instruction `try_me_else` crée un point de choix dans la pile dans lequel elle sauvegarde les informations nécessaires pour pouvoir passer au choix suivant en cas de retour en arrière. L'instruction associée `retry_me_else` met à jour le point de choix et défait les unifications qui ont eu lieu depuis le dernier point de choix. L'instruction `trust_me` supprime le point de choix et défait aussi les unifications pour l'évaluation de la dernière clause.

La conjonction entre les prémisses est assuré par l'instruction `allocate` qui crée une frame d'environnement dans la pile. Cette frame sert à stocker entre autres les variables permanentes `Yi` qui apparaissent dans plusieurs prémisses (en considérant que la conclusion fait partie de la première prémisse) et l'adresse de retour des procédures appelée par l'instruction `call`. Lorsqu'il n'y a qu'une seule prémisse, il n'y a pas besoin de frame d'environnement car il s'agit seulement d'une modification d'arguments et d'un transfert de contrôle à une autre procédure. La compilation d'une clause se termine par l'instruction `proceed` qui renvoie le contrôle à la procédure appelante.

[74] La variable `?c` apparaît aussi dans `(true (cell ?x ?y ?c))` mais il n'est pas nécessaire d'y faire référence car l'argument de **`true`** est déjà référencé entièrement par `A1`. C'est différent pour `?x` et `?y` qui apparaissent plus loin dans un autre contexte : `(distinct ?x ?x1)` et `(distinct ?y ?y1)`.

5.2.1.3 Parallélisation-ou

YapProlog possède un module de parallélisation YapOr au niveau des points de choix qui permet de traiter les différentes clauses d'une même procédure dans des processus distincts [59]. Nous ne l'avons pas utilisé pour principalement trois raisons. La première est que le temps nécessaire au lancement des threads peut dépasser le gain obtenu par la parallélisation au niveau des points de choix et nous n'avons pas identifié de critère simple, autre que comparer les temps d'exécution, permettant de déterminer le sous-ensemble des descriptions GDL qui pourraient bénéficier d'une telle parallélisation. La deuxième raison est que ce module n'est pas compatible avec le tabling que nous décrivons à la section 5.2.2.1 et qui nous est indispensable pour instancier les descriptions GDL. La troisième raison est que la nature procédurale de Prolog impose un ordre dans l'évaluation des prémisses d'une règle, ce qui n'est pas le cas du GDL ; le module de parallélisation utilise donc un mécanisme de synchronisation pour garantir cet ordre et utilise des ressources de calcul alors que cela n'est pas nécessaire. Nous avons donc plutôt étudié la possibilité de paralléliser le tabling.

5.2.2 Parallélisation du *tabling*

5.2.2.1 Le *tabling*

La résolution SLD avec la négation par l'échec (SLD-NF) reflète l'interprétation procédurale du calcul des prédicats en tant que langage de programmation et constitue la base de calcul pour les systèmes Prolog. En dépit de ses avantages concernant la mémoire gérée sous la forme d'une pile, la résolution SLD-NF n'est souvent pas bien adaptée à l'évaluation de requêtes pour trois raisons : elle peut ne pas s'arrêter à cause d'une récursion positive infinie ; elle peut ne pas s'arrêter à cause d'une récursion infinie à travers une négation ; elle peut évaluer à de nombreuses reprises le même littéral[75] parmi les prémisses d'une clauses, dégradant les performances de façon inacceptable.

La résolution SLG [17] permet de résoudre ces trois problèmes en utilisant une évaluation tablée[76] et paresseuse. Cette méthode d'évaluation appelée *tabling* en Prolog consiste à mémoriser dans des tables des faits déjà prouvés. Le programmeur peut choisir de tabler certains prédicats et pas d'autres.

YapProlog utilise des tables distinctes pour les requêtes qui ne sont pas équivalentes, en considérant que deux requêtes sont équivalentes lorsqu'elles sont structurellement identiques à un renommage bijectif près des variables. Par exemple la requête `(req a ?x (foo ?y ?z))` est

[75] Un littéral est un atome ou la négation d'un atome.

[76] Nous utilisons ce néologisme pour traduire le mot anglais *tabled* qui signifie placé dans un dépôt ou une table.

équivalente à `(req a ?g (foo ?h ?k))` mais pas à `(req a ?x (foo ?x ?z))` [60, 61]. Nous avons fait le même choix pour l'interpréteur GDL de LeJoueur.

Lorsqu'au niveau d'un nœud de son arbre de recherche, l'interpréteur YapProlog doit évaluer une requête sur une procédure tablée, il vérifie tout d'abord s'il existe déjà une table pour cette requête. Si ça n'est pas le cas, le nœud devient un *producteur* et crée une table spécifique à cette requête. Ce producteur peut alors être invoqué à tout moment pour démontrer de nouveaux faits qui sont stockés dans la table. Lorsque dans un autre contexte, c'est-à-dire à une autre position de l'arbre de recherche, la même requête est soumise à l'interpréteur, celui-ci constate qu'une table existe et le nœud devient un *consommateur* de la table. Il utilise les faits déjà prouvés par le producteur et évite ainsi des redémonstrations coûteuses. Cela nécessite dans YapProlog une modification de la pile afin que l'environnement du producteur puisse être gelé lors de la poursuite de l'évaluation. En effet, le producteur doit pouvoir être réinvoqué à tout moment pour produire des faits réclamés par les consommateurs.

Nous avons utilisé le même principe avec un producteur et des consommateurs dans la machine de Warren de LeJoueur mais en utilisant plusieurs machines. En effet, la nature très légère de l'implémentation de notre machine permet d'en créer à la demande très rapidement. Ainsi, lorsque l'interpréteur doit évaluer une requête sur une procédure tablée pour la première fois, il crée une nouvelle machine qui joue le rôle de producteur et stocke les faits prouvés dans une table. Lorsque la même requête est rencontrée à nouveau dans un autre contexte, les faits déjà prouvés sont simplement consommés et lorsque la table est épuisée, la machine du producteur peut à nouveau être invoquée pour produire un nouveau fait.

Le critère d'arrêt pour un producteur, qui entraîne la clôture de la table, intervient lorsque celui-ci s'invoque lui-même.

5.2.2.2 Exécution des producteurs dans des threads distincts

Comme nous l'avons mentionné à la section 5.2.1.3, le module de tabling YapTab n'est pas compatible avec le module YapOr de parallélisation au niveau des points de choix. Pourtant, le module YapTab a été conçu pour être facilement parallélisable [61] mais cette implémentation n'a encore été faite à notre connaissance.

Le tabling de la machine de Warren de LeJoueur est aussi conçu pour permettre aisément sa parallélisation. En effet, contrairement au tabling de YapProlog qui n'utilise qu'une seule pile et un mécanisme qui gèle des parties de la pile pour préserver les producteurs, notre implémentation qui utilise plusieurs machines de Warren permet facilement de lancer celles-ci dans des threads distincts. Une parallélisation du tabling permet d'obtenir plus rapidement l'instanciation de la

description GDL avant le début d'un match et ainsi de disposer plus rapidement du circuit logique permettant d'accélérer les simulations de parties en cours de match.

5.3 Parallélisation de la simulation du circuit logique

Le circuit logique construit à la section 2.3.2 permet de réaliser des parties aléatoires ou *playouts* permettant d'évaluer la valeur des positions d'un jeu, notamment dans le cadre d'un algorithme MCTS. Dans cette section, nous présentons différentes approches que nous avons testées pour paralléliser et ainsi accélérer la simulation du circuit. Nous avons dans tous les cas adopté une approche dans laquelle les calculs sont effectués des entrées du circuit vers ses sorties d'une part parce que des tests préliminaires nous avaient montré que l'évaluation était plus rapide et d'autre part parce que certaines des méthodes présentées ne fonctionnent que dans ce sens.

Après avoir rappelé les caractéristiques du circuit logique, nous nous intéressons à trois types de parallélisation indépendantes dont le but est d'accélérer la simulation de ce circuit. À la section 5.3.2 nous présentons un TAM nécessaire à une parallélisation SIMD permettant de simuler simultanément plusieurs portions du circuit à partir d'un seul état du jeu donné en entrée. La seconde parallélisation de type SIMD que nous décrivons à la section 5.3.3 est celle que nous utilisons dans LeJoueur ; il s'agit d'un TPM de type SIMD pouvant traiter plusieurs états du jeu en parallèle et ainsi réaliser plusieurs *playouts* simultanés. Nous présentons enfin un autre TPM pour une parallélisation de type MIMD, en tirant parti du fait que des groupes d'opérations pour simuler le circuit peuvent être effectuées dans un ordre indifférent.

5.3.1 Rappels sur la simulation du circuit logique

Le circuit logique décrit à la section 2.3.2 est représenté par un graphe. Les sommets sont étiquetés par trois types de portes logiques : les portes *ou*, *et* et *non*. Ce graphe a été construit à partir d'une description GDL instanciée et nous avons présenté un algorithme glouton permettant de réduire la taille de ce circuit.

À partir de ce circuit logique, nous avons généré un ensemble constitué de variables booléennes et de *bytecodes* afin de simuler son comportement. Il y a quatre *bytecodes* chargés chacun de simuler la partie du circuit permettant de calculer les prédicats `terminal`, `legal`, `next` et `goal`. Le *bytecode* dont un exemple est donné à la figure 5.2 est constitué d'opérations logiques sur les variables à effectuer dans un certain ordre. Les strates dans le *bytecode* correspondent à des ensembles d'opérations logiques qui peuvent s'effectuer simultanément mais une strate d'un certain niveau ne peut être calculée que lorsque la strate de niveau inférieur l'a été en totalité. Pour

bytecode	commentaire
	début du code
2	2 strates
3	début de la première strate, 3 conjonctions
6 2 3	$v_6 \leftarrow v_2 \wedge v_3$
5 1 3	$v_5 \leftarrow v_1 \wedge v_3$
8 2 4	$v_8 \leftarrow v_2 \wedge v_4$
0	0 disjonction
1	1 négation
7 3	$v_7 \leftarrow \neg v_3$
0	début de la seconde strate, 0 conjonction
1	1 disjonction
9 7 5	$v_9 \leftarrow v_7 \vee v_5$
0	0 négation
	fin du code

FIGURE 5.2 – Exemple de *bytecode* constitué de plusieurs strates comportant une liste de conjonctions, de disjonctions et de négations. Au sein d'une strate, l'ordre des opérations n'a pas d'importance.

générer ce *bytecode*, nous avons préalablement modifié le circuit afin que les portes logiques ne comportent que deux entrées pour les conjonctions et les disjonctions.

La figure 5.2 reprend l'exemple de *bytecode* de la figure 2.5.

5.3.2 Parallélisation des opérations logiques au sein d'un mot

La parallélisation de type SIMD que nous présentons dans cette section est la seule qui n'utilise pas le *bytecode* et nécessite un pré-traitement du circuit avant le début d'un match. Le but est d'utiliser le fait que les opérations logiques sur des mots traitent les bits en parallèle et ainsi calculer en parallèle plusieurs portes logiques du circuit.

Il n'y a que trois types d'opérations logiques dans ce circuit : les *et*, le *ou* et le *non* sur des variables booléennes qui n'ont que deux états : 0 ou 1. Les processeurs que nous utilisons ne sont pas optimisés pour adresser directement les bits, et dans la plupart des langages de programmation, ces opérations sont en réalité effectuées sur un mot de n bits pour lequel nous n'utilisons qu'un seul bit. C'est l'approche que nous avons utilisée pour générer le *bytecode* en associant une variable de la taille d'un mot à chaque sommet pour représenter son état 0 ou 1.

En revanche, si nous décidons d'associer un bit par sommet, cela nous permet d'effectuer plusieurs opérations logiques en une seule opération portant sur des mots. Par exemple, si deux bits a_0 et a_1 sont dans le même mot et que b_0 et b_1 sont dans un autre mot, on peut calculer en une seule opération $a_0 \wedge b_1$ et $b_0 \wedge b_1$. Cependant, réaliser les opérations $a_0 \wedge b_1$ ou $a_1 \wedge b_0$ nécessiterait un déplacement des bits au sein des mots et ces opérations sont particulièrement coûteuses. Optimiser le placement des bits dans un ensemble de mots, afin de minimiser les opérations de déplacement de ces bits, est une tâche complexe. Nous avons étudié deux approches pour minimiser ces déplacements.

La première nécessite de trouver des sous-graphes indépendants et isomorphes du circuit les plus grands possibles. Dans ce cas, les bits associés aux sommets mis en relation par l'isomorphisme peuvent occuper le même mot et il n'est pas nécessaire de les déplacer. Le problème des graphes isomorphes, bien qu'étant polynomial dans le cas où le degré est borné [41], est trop coûteux pour être réalisé dans des temps compatibles avec les contraintes du GGP.[77].

La seconde approche consiste à pré-calculer de larges portions du circuit en utilisant des tables. Pour que les tables restent de taille raisonnable, le nombre d'entrées est limité. En se limitant à 256 entrées, le problème revient à trouver 8 sommets dans le graphe qui permettent de calculer un maximum d'autres sommets. Les combinaisons possibles des valeurs de vérité associées à ces 8 sommets sont de 256 et chacune de ces possibilités est une entrée de la table. Les valeurs qui y sont stockées sont les valeurs de vérité des sommets dont la valeur peut s'exprimer uniquement en fonction de ces entrées. Là encore, cette approche nécessite de préparer les valeurs de vérité des sommets pour les regrouper dans un mot de 8 bits.

La complexité du pré traitement nécessaire sur le circuit avant le début d'un match ne permet pas de le réaliser dans des temps compatibles avec le GGP. De plus la pénalité en temps de calcul engendrée par le déplacement des bits au sein des mots lors des simulations en cours de match ne nous a pas permis d'accélérer la simulation du circuit. Nous n'avons donc pas non plus exploré plus avant la possibilité de simuler plusieurs portions distinctes du circuit en parallèle. Nous présentons donc dans la section suivante une autre parallélisation de type SIMD permettant d'effectuer plusieurs simulations simultanées en cours de match.

[77] Nous avons imaginé un cas où cette technique pourrait être utilisée plus facilement : le cas des jeux à deux joueurs ou plus où la description GDL est symétrique. Prenons une description GDL dont on retire les prédicats `init`, `input` et `base` et dont on permute les symboles des rôles. S'il existe une permutation des autres constantes et symboles de fonction (à l'exception des prédicats du langage) et que l'on retombe sur une description identique, nous avons détecté des sous-graphes isomorphes de grande taille.

5.3.3 Simulation de *n playouts* simultanés sur un seul thread

Les sections qui suivent décrivent la parallélisation de type SIMD que nous utilisons dans LeJoueur après que le match a commencé et qui permet de réaliser *n playouts* simultanés sur un seul thread en utilisant le *bytecode*. Elle exploite le fait qu'une opération logique sur un mot est aussi rapide que lorsqu'on n'utilise qu'un seul bit de ce mot de n bits. Sur l'architecture x86-64 que nous utilisons, les registres font 64 bits et il est donc possible de faire jusqu'à $n = 64$ calculs booléens en simultané. Nous présentons d'abord comment utiliser le *bytecode* pour traiter 64 états de jeux en simultané, puis nous montrons comment utiliser cette propriété aussi pour un tirage aléatoire des coups. Nous abordons enfin le problème lié à l'initialisation de n états distincts du jeu.

5.3.3.1 Utilisation du *bytecode* pour traiter simultanément n états distincts

Les variables utilisées par le *bytecode* sont des booléens. Nous remplaçons ces variables par des mots de n bits et nous modifions la fonction d'évaluation de l'algorithme 5.1, qui est simplement l'algorithme 2.4 dans lequel nous avons remplacé les opérations booléennes $\wedge, \vee, \neg$ par les opérations logiques bit à bit correspondantes notées $\wedge_n, \vee_n, \neg_n$. Il est alors possible d'utiliser le même *bytecode* pour traiter n états distincts du jeu en simultané. Si n est la taille du mot mémoire pour stocker un booléen pour une architecture de machine particulière, alors le temps d'exécution est à peu près le même car ce sont les mêmes instructions du processeur qui sont utilisées.

Tout se passe comme si nous disposions de n circuits identiques. La seule contrainte est que les évaluations des n circuits s'effectuent de manière synchrone. Il n'est pas possible de calculer le prédicat `terminal` en même temps que le prédicat `next` par exemple. De même, si nous avons besoin de calculer les scores pour un seul des circuits qui est dans un état terminal, alors le calcul sera fait sur les n circuits.

L'exécution du *bytecode* ne concerne que la simulation de n circuits logiques en parallèle, il n'assure pas le choix des coups, ni l'initialisation des états initiaux ou le recueil des scores. Par la suite, lorsque nous parlerons du circuit i avec $i \in [\![1, n-1]\!]$, nous ferons référence aux calculs réalisés sur le $i^{\text{ème}}$ bit des mots.

5.3.3.2 Tirage aléatoire séquentiel des coups par état

Le tirage aléatoire des coups dans le cadre d'un *playout* est une action qu'il faut particulièrement optimiser car nous avons mesuré que la majorité du temps de calcul y est consacré par rapport à l'exécution du *bytecode*.

Procédure `EvalBytecodeParallèle(`$P = \{p_i, i \textit{ entier}\}$ *:* bytecode*,* $V = \{v_k, k \textit{ entier}\}$ *:* *mots de n bits*)

$i \leftarrow 0$
pour $s \leftarrow p_0$ **à** 1 **faire**
  $i \leftarrow i + 1$
  pour $j \leftarrow p_i$ **à** 1 **faire**
    $v_{p_i} \leftarrow v_{p_{i+1}} \wedge_n v_{p_{i+2}}$
    $i \leftarrow i + 3$
  fin
  $i \leftarrow i + 1$
  pour $j \leftarrow p_i$ **à** 1 **faire**
    $v_{p_i} \leftarrow v_{p_{i+1}} \vee_n v_{p_{i+2}}$
    $i \leftarrow i + 3$
  fin
  $i \leftarrow i + 1$
  pour $j \leftarrow p_i$ **à** 1 **faire**
    $v_{p_i} \leftarrow \neg_n v_{p_{i+1}}$
    $i \leftarrow i + 2$
  fin
fin

Algorithme 5.1 : Évaluation du *bytecode*

Nous avons noté V_{legal_j} l'ensemble des variables associées aux coups du joueur j pouvant être légaux au cours de n'importe quelle partie. Après évaluation du circuit, ces variables valent 1 lorsque le coup est légal et 0 sinon. Nous avons déjà décrit une procédure de choix aléatoire d'un coup légal pour un joueur à l'algorithme 2.5. Elle consiste à répertorier les sorties du circuit correspondant aux coup légaux qui sont à 1, puis de tirer aléatoirement une de ces sorties. Cette opération doit être répétée pour chaque joueur. En notant $l = \text{Card}(V_{legal_j})$, la complexité de cet algorithme est en $O(l)$, c'est-à-dire linéaire en fonction du nombre total de coups, le temps nécessaire au tirage étant constant.

En utilisant des mots de n bits, il y a n choix aléatoires à faire par joueur pour chacun des n circuits. L'algorithme 5.2 est une adaptation où l'algorithme 2.5 est répété n fois et où seul un bit du mot est lu à chaque fois. Sa complexité est en $O(l \times n)$ et le temps d'exécution du choix aléatoire des coups est au moins n fois plus élevé.

Procédure `CoupAleatoireXn`(*j : joueur,* V_{legal_j}, V_{does_j} *: variables (mots de n bits)*)
- **pour** $b \leftarrow 0$ **à** $n-1$ **faire**
  - $l \leftarrow$ liste vide
  - **pour chaque** $v \in V_{legal_j}$ **faire**
    - **si** $bit_b(v) = 1$ **alors**
      - insérer une référence à v dans l
    - **fin**
  - **fin**
  - *choix* $\leftarrow$ tirage aléatoire dans l
  - **pour chaque** $v \in V_{does_j}$ **faire**
    - **si** *v est la variable de* V_{does_j} *associée à variable choix de* V_{legal_j} **alors**
      - $bit_b(v) \leftarrow 1$
    - **fin**
    - **sinon**
      - $bit_b(v) \leftarrow 0$
    - **fin**
  - **fin**
- **fin**

Algorithme 5.2 : Tirages aléatoires de n coups pour le joueur j linéaire en fonction de la taille n du mot.

Cet algorithme, contrairement à celui de l'évaluation du circuit, ne tire aucun bénéfice des opérations de calcul simultané sur n bits. Nous avons donc réalisé aussi une parallélisation SIMD

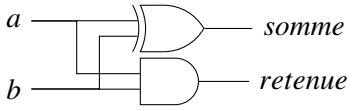


FIGURE 5.3 – Demi-additionneur à 1 bit.

du tirage aléatoire des coups dans laquelle les différents bits du mot ne sont pas traités individuellement.

5.3.3.3 Parallélisation du tirage aléatoire des coups

Pour décrire cette parallélisation, nous allons présenter une méthode pour choisir un coup parmi tous les coups légaux de manière équiprobable en n'utilisant que le bit comme unité d'information. Pour simplifier notre propos, nous présentons les algorithmes sur une taille de mot égale à 1 bit et nous mentionnons comment les adapter pour n bits en remplaçant les opérations sur un bit par les opérations correspondantes agissant sur n bits simultanément et les quelques précautions qu'il faut prendre.

Pour effectuer un tirage sur le circuit, nous réalisons les opérations suivantes : nous déterminons le nombre de coups légaux par des additions sur 1 bit, nous effectuons plusieurs tirages aléatoires sur un bit, la suite de ces bits représentant un entier x indiquant la position relative du coup choisi parmi toutes les sorties légales à 1 ; nous repassons un nouvelle fois sur les coups légaux en décrémentant x pour chaque sortie à 1 ; lorsque x s'annule, le coup choisi est identifié et nous le reportons sur l'entrée `does` correspondante.

5.3.3.3.1 Calcul du nombre de coups légaux La première étape est de calculer le nombre de coup légaux. Nous effectuons des incrémentations sur le modèle du demi-additionneur à 1 bit présenté à la figure 5.3. La somme est représentée par une suite de bits $s_k, \ldots, s_1, s_0$. Pour chaque sortie de V_{legal_j} à 1 représentant un coup légal pour le joueur j, nous incrémentons cette suite de bits en incrémentant tout d'abord s_0 ; si cette incrémentation produit une retenue à 1, alors nous incrémentons s_1 et nous réitérons le procédé jusqu'à ce qu'il n'y ait plus de retenue. Ce calcul est présenté à l'algorithme 5.3. Le pire cas pour cet algorithme correspond au fait que tous les coups sont légaux. Dans ce cas, la complexité en temps d'exécution est en $O(l \times \log(l))$.

Pour passer à n bits, il suffit simplement de remplacer les suites de bits en suites de mots de n bits et de remplacer les opérations logiques par leurs équivalents sur n bits.

5.3.3.3.2 Tirage aléatoire des coups La seconde étape consiste à tirer aléatoirement un entier positif ou nul strictement inférieur au nombre de coup légaux. Encore une fois, nous n'allons

Fonction `NombreCoupsLegaux`(*j : joueur,* V_{legal_j} *: variables booléennes*)

$s \leftarrow ()$
pour chaque $b \in V_{legal_j}$ **faire**
$retenue \leftarrow b$
$i \leftarrow 0$
tant que $i < taille(s)$ **et** $retenue \neq 0$ **faire**
$(s_i, retenue) \leftarrow (s_i \oplus retenue, s_i \wedge retenue)$
$i \leftarrow i + 1$
fin
si $retenue \neq 0$ **alors**
$s_i \leftarrow retenue$
fin
fin
retourner $s = (s_{taille(s)-1}, ..., s_1, s_0)$

Algorithme 5.3 : Calcul du nombre de coups légaux avec uniquement des variables booléennes. Pour un calcul simultané du nombre de coups légaux pour les n rangées de bits, il suffit de remplacer les opérations sur un bit par l'opération équivalente sur n bits.

utiliser que des opérations sur un bit. Cela revient à tirer aléatoirement un entier par le tirage individuel de chacun de ses bits. Si $s = (s_{taille(s)-1}, ..., s_1, s_0)$ est le nombre de coups légaux, nous devons tirer un entier entre 0 et $s-1$ et pour cela il faut tirer au moins $\lfloor \log_2(s-1) + 1 \rfloor$ bits. Si l'entier obtenu n'est pas strictement inférieur à s, alors il faut recommencer l'opération pour garantir un tirage équiprobable. Dans le pire cas, lorsque $s - 1 = 2^k$, il y a une chance sur deux de tirer un entier en dehors de l'intervalle. Pour k tirages, la probabilité de n'obtenir aucun entier valide est alors de $\left(\frac{1}{2}\right)^k$. Cette probabilité d'échouer est inférieure à 1% pour 7 tirages et à 0,1% pour 10 tirages. La complexité du tirage est donc en pratique constante.

L'algorithme 5.4 effectue ce tirage. Il tire aléatoirement un nombre de bits égal au nombre de bits utiles pour représenter le nombre de coups légaux s puis vérifie que le nombre représenté est strictement inférieur à s. Si ça n'est pas le cas, il réitère l'opération[78].

Pour adapter cet algorithme sur n bits, mis à part le passage aux opérations logiques sur les mots, il faut veiller à ne pas refaire les tirages une fois qu'ils sont valides. Il suffit pour cela d'introduire un masque de tirage qui détermine quels sont les bits, parmi les n que contient le mot,

[78] L'algorithme n'est pas tout à fait optimal car si par exemple $s = 1000$ en binaire alors il va tirer quatre bits alors que le plus grand tirage admissible est 111 en binaire. Dans LeJoueur, nous décrémentons s de 1 pour déterminer le nombre de bits à tirer aléatoirement. Nous n'avons pas traité ce cas pour éviter de compliquer davantage l'explication.

qui peuvent être modifiés. La complexité est alors la même que pour un seul bit, c'est-à-dire en $O(\log(l))$ en moyenne.

Fonction `TirageParBits`($s = (s_{taille(s)-1}, ..., s_1, s_0)$ *: suite de bits*)
- $a \leftarrow ()$
- $valide \leftarrow 0$
- **tant que** $valide = 0$ **faire**
  - **pour** $i \leftarrow 0$ **à** $taille(s) - 1$ **faire**
    - $a_i \leftarrow rand(\{0,1\})$
  - **fin**
  - **pour** $i \leftarrow 0$ **à** $taille(s) - 1$ **faire**
    - $identique \leftarrow \neg a_i \oplus s_i$
    - $inferieur \leftarrow \neg a_i \wedge s_i$
    - $valide \leftarrow (valide \wedge identique) \vee inferieur$
  - **fin**
- **fin**
- **retourner** $a = (a_{taille(a)-1}, ..., a_1, a_0)$

Algorithme 5.4 : Tirage aléatoire d'un entier par tirage individuel de chacun de ses bits.

5.3.3.3.3 Report du coup légal sur l'entrée `does` La troisième étape consiste à reporter le coup choisi parmi V_{legal_j} vers sa variable associée de V_{does_j}. Pour cela, nous parcourons à nouveau les variables V_{legal_j} en décrémentant la suite de bits a représentant le tirage à chaque fois qu'un coup légal est rencontré. Lorsque a s'annule, c'est que le prochain coup légal est le coup choisi et nous mettons à 1 la variable associée de V_{does_j}. Le détail de cette opération est donné à l'algorithme 5.5.

Comme précédemment, pour utiliser cet algorithme sur des mots de n bits, il suffit de remplacer les opérateurs logiques par leur équivalent sur n bits. La complexité de cet algorithme est la somme de celles des deux précédentes fonctions qui étaient $O(l \times \log(l))$ et $O(l)$ et de celle de la boucle principale qui est en $O(l \times \log(l))$. L'algorithme de tirage de n coups aléatoires possède donc une complexité moyenne en $O(l \times \log(l))$ où $l = \text{Card}(V_{legal_j})$ est le nombre total de coups du joueur j qui pourraient être légaux dans n'importe quel match.

La complexité en temps de calcul de l'algorithme de tirage séquentiel décrit à la section 5.3.3.2 est en $O(l \times n)$. Les deux complexités varient donc linéairement en fonction du nombre de variables $l = \text{Card}(V_{legal_j})$. Celle du tirage séquentiel varie aussi linéairement en fonction de la taille n du mot alors que celle du tirage en parallèle y est insensible.

Procédure `CoupAleatoireParallèle`(j : *joueur,* V_{legal_j}, V_{does_j} : *variables booléennes*)

$s \leftarrow$ `NombreCoupsLegaux`(V_{legal_j})
$a \leftarrow$ `TirageParBits`(s)
$fait \leftarrow 0$
pour chaque $b \in V_{legal_j}$ **faire**
  $estnul \leftarrow \neg fait$
  pour $i \leftarrow 0$ **à** $taille(a) - 1$ **faire**
    $estnul \leftarrow estnul \wedge \neg a_i$
  fin
  d est à la variable de V_{does_j} associée à la variable b de V_{legal_j}
  $d \leftarrow b \wedge estnul$
  $fait \leftarrow fait \vee d$
  pour $i \leftarrow 0$ **à** $taille(a) - 1$ **faire**
    $(a_i, b) \leftarrow (a_i \oplus b, \neg a_i \wedge b)$
  fin
fin

Algorithme 5.5 : Choix aléatoire d'un coup pour le joueur j avec uniquement des opérations sur un bit. Pour paralléliser cette procédure, il suffit de remplacer les opérations logiques par leurs équivalents sur les mots.

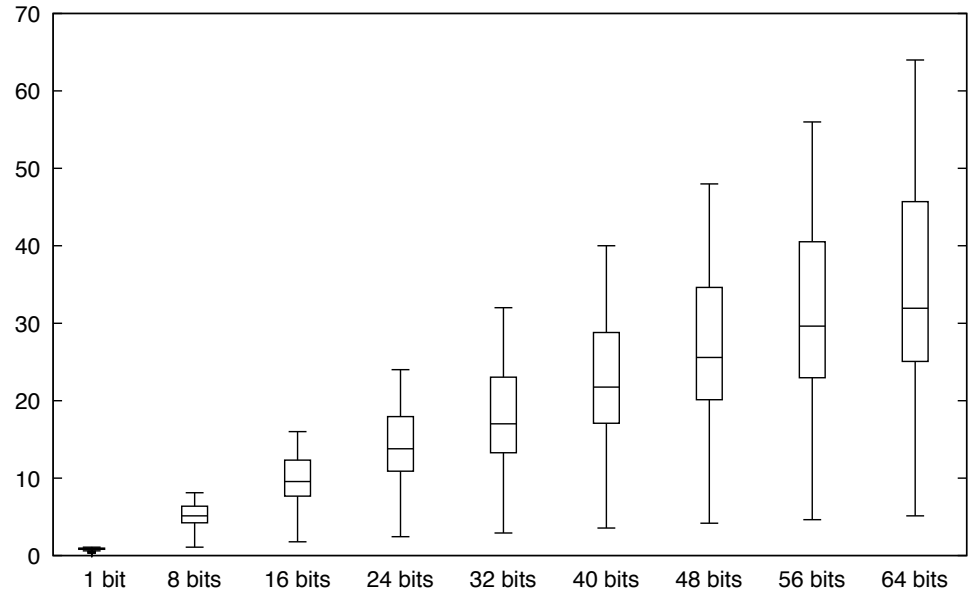


FIGURE 5.4 – Multiplicateur du nombre de *playouts* par seconde effectués en utilisant la parallélisation SIMD en fonction du nombre de bits utilisés par mot pour 154 des 157 jeux disponibles sur le serveur Tiltyard. La valeur 1 correspond au nombre de *playouts* par seconde effectués avec le *bytecode* non parallélisé. Chaque boite à moustaches représente la distribution des jeux selon le multiplicateur obtenu par l'application de la parallélisation (médiane, quartiles et déciles).

5.3.3.4 Évaluation du gain lié au calcul de *n playouts* simultanés

Nous avons mesuré le comportement de notre parallélisation SIMD en fonction du nombre de bits utilisés. Pour cela, nous avons collecté les 157 descriptions GDL qui étaient disponibles sur le serveur Tiltyard à la fin du mois d'août 2017. Nous avons compilé chaque description en *bytecode* et mesuré pour des tailles de mots de 8 à 64 bits, le nombre de *playouts* réalisés pendant 10 secondes sur un thread de processeur Intel Xeon E5-4610 à 2.40GHz. Nous présentons les résultats à la figure 5.4. Pour chaque taille de mot présent en abscisses, nous avons représenté une boite à moustaches qui donne les proportions de jeux pour lesquels la parallélisation conduit à multiplier le nombre de *playouts* réalisés par la valeur située en ordonnée. La valeur centrale d'une boite indique le facteur multiplicatif au moins atteint pour 50% des jeux, ce qui correspond à la médiane ; le rectangle indique les quartiles et les extrémités des moustaches indiquent les déciles.

Nous observons que la progression du facteur multiplicatif est linéaire en fonction de la taille du mot utilisée ce qui indique que la parallélisation est efficace. En utilisant des mots de 64 bits, le nombre de *playouts* réalisés est multiplié par au moins 25 pour les trois quarts des jeux, par au moins 32 pour la moitié des jeux et par au moins 46 pour le quart des jeux.

Nous avons aussi comparé rapidité du calcul de 64 *playouts* en simultané en la comparant au temps moyen utilisé par YAPPROLOG pour réaliser un *playout* en utilisant les mêmes données. Les résultats sont présentés à la figure 5.5 de la même manière qu'à la section 2.4.2. L'axe des abscisses représente le nombre de *playouts* par seconde réalisés par YAPPROLOG et en ordonnée, le nombre de *playouts* réalisés par le *bytecode* parallélisé. Le *bytecode* réalise davantage de *playouts* que YAPPROLOG dans tous les cas sauf pour le jeu RACER4 qui comporte plus de 10^6 opérations. Le *bytecode* est réalise 10 fois plus de *playouts* dans 93% des cas, au moins 100 fois plus dans 76% des cas et au moins 1000 fois plus dans 33% des cas.

5.3.3.5 Choix d'implémentation liée à l'évaluation synchrone des circuits

La parallélisation SIMD que nous venons de présenter permet de simuler n circuits avec seulement un léger surcoût par rapport à la simulation d'un seul circuit. Elle présente cependant le désavantage que les évaluations de ces circuits sont toujours synchrones et qu'il faut donc trouver une utilisation à ces n circuits.

Le désavantage principal de cet aspect synchrone est que le calcul des scores est effectué sur les n circuits, y compris ceux pour lesquels cette information n'est pas pertinente. Lors de la simulation de n *playouts* en simultané, certains se terminent plus rapidement que d'autres. Nous avons donc fait le choix dans LEJOUEUR d'attendre que le *playout* le plus long s'achève pour calculer simultanément les scores pour les n circuits.

Chaque circuit doit être initialisé avec un état du jeu. Une initialisation avec n états de départ distincts est n fois plus longue que pour un seul. C'est pourquoi dans LEJOUEUR, nous initialisons toujours les n circuits avec le même état de départ. La mise à jour de l'état des n circuits lors d'un *playout* n'est pas pénalisante car l'algorithme de tirage aléatoire des coups présenté dans la section précédente est efficace et que les sorties de V_{next} sont simplement des copies de mots de n bits sur les entrées V_{true}, ce qui est une opération rapide.

Nous obtenons donc actuellement n *playouts* à partir de la même position mais nous envisageons, toujours avec les n circuits initialisés avec le même état, d'extraire les états suivants lors des premières itérations du *playout*. Ceci nous permettrait de construire un arbre dans lequel nous pourrions ajouter n sommets à la fois, chaque état étant associé à un *playout* indépendant des autres. Ceci pourrait se faire sans surcoût car en l'état, il y a une initialisation de circuit par sommet qui permet d'obtenir n *playouts*. Chaque initialisation serait remplacée par une extraction d'un état, ces deux opérations ayant la même complexité. Ceci nous permettrait donc de construire un arbre avec un *playout* par sommet pratiquement n fois plus rapidement que l'arbre actuel avec ses n *playouts* par sommet.

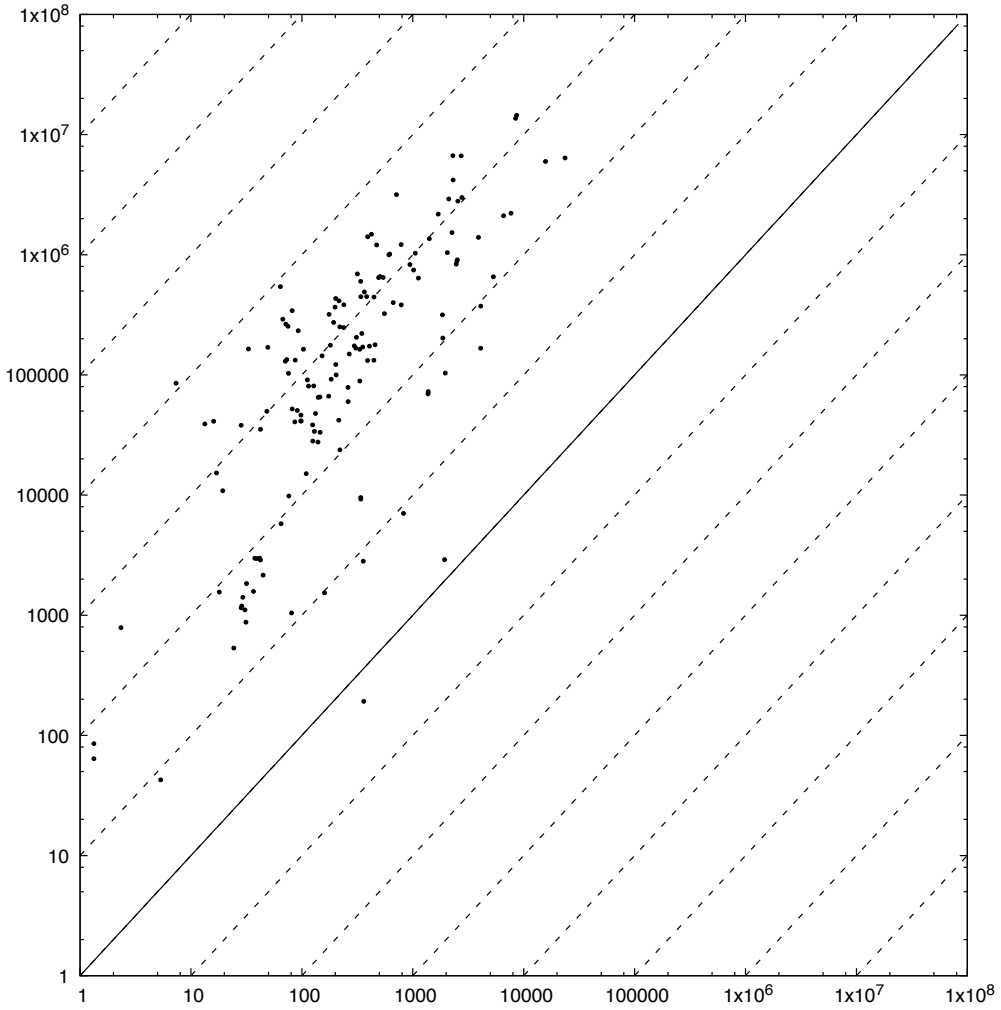


FIGURE 5.5 – Comparaison du nombre de parties par seconde réalisées par le *bytecode* (en ordonnée) parallélisé sur 64 bits en fonction du nombre de parties par seconde réalisées par YAPPROLOG (en abscisse). Les abscisses et les ordonnées sont en échelle logarithmique. La diagonale en trait plein correspond aux points où les performances sont les mêmes entre le *bytecode* et YAPPROLOG. Les lignes en pointillés au dessus correspondent aux points où le *bytecode* est plus rapide d'un facteur 10, 100, 1000, etc.

5.3.4 Parallélisation multi-thread du calcul d'un *playout*

Le *bytecode* défini à la section 2.3.3 est constitué d'une suite de strates et chaque strate doit être évaluée après les strates de niveaux inférieurs. Chaque strate est constituée d'opérations logiques qui effectuent des conjonctions, disjonctions et négations entre des variables booléennes stockées en mémoire. Ces opérations logiques peuvent être effectuées dans n'importe quel ordre au sein d'une strate.

Cette propriété nous permet d'envisager une parallélisation MIMD dans laquelle chaque thread est chargé du calcul d'une partie du *bytecode*. Pour une parallélisation à n threads, nous créons n *bytecodes* partiels à partir du *bytecode* initial. Chaque *bytecode* partiel possède le même nombre de strates que le *bytecode* initial mais seulement une partie de ses opérations logiques. Pour une même strate, la réunion des opérations logique des *bytecodes* partiels est identique à l'ensemble des opérations logique du *bytecode* initial. Avec une répartition équitable, les *bytecodes* partiels contiennent à peu près un $n^{\text{ème}}$ de l'ensemble des opérations. Nous avons testé cette parallélisation MIMD sur trois architectures : un CPU Intel x86-64, un GPU NVidia et un processeur *many-core* développé par la société Kalray.

5.3.4.1 Multi-threads sur un CPU

L'exécution concurrente par les threads des *bytecodes*, même s'ils ont exactement le nombre et le même type d'opérations logiques ne prennent pas le même temps. Ceci est dû à plusieurs facteurs : le système d'exploitation peut donner plus ou moins de temps de calcul à un thread ; le temps d'accès à la mémoire dépend du cache ; la durée des instructions en nombre de cycles sur les processeurs x86-64 que nous utilisons n'est plus donnée qu'en moyenne car elle est affectée par d'autres tâches que le processeur effectue simultanément. Il est donc nécessaire de synchroniser les threads pour qu'ils débutent le calcul de chaque strate en même temps. Nous avons donc introduit une barrière de synchronisation à la fin de chaque strate dans LeJoueur.

Les processeurs Intel multi-cœurs x86-64 possèdent plusieurs niveaux de cache. Le cache L1, qui est le plus rapide n'est pas partagé entre les cœurs. Par conséquent, deux cœurs ne peuvent pas accéder en même temps à une ligne de cache L1, ce qui provoque des défauts de cache et des attentes [23]. Il faut donc veiller à ce que les *bytecodes* partiels fassent des calculs sur des segments de mémoire chargés dans des lignes de cache distinctes. Ce problème de placement des variables utilisées par les *bytecodes* est loin d'être trivial. De plus, il n'est pas toujours possible d'obtenir un agencement parfait car les variables apparaissant souvent dans le *bytecode* initial sont présentes dans plusieurs *bytecodes* partiels.

Cette approche n'a pas été concluante d'autant plus que le temps de lancement d'un thread est trop important en comparaison avec le temps d'exécution du bytecode. Même en maintenant les threads en vie, nous n'avons pas constaté d'amélioration.

5.3.4.2 Utilisation d'un GPU

Les capacités de calcul des processeurs graphiques (GPU) ont très rapidement évolué au cours de ces dix dernières années. Afin de répondre à l'exigence de produire un rendu temps réel (60 à 100 frames par seconde) pour un scène composée d'objets modélisés par des maillages triangulaires, les constructeurs des cartes graphiques ont proposé des architectures spécifiques. Les opérations à réaliser étant les mêmes sur des données différentes, une architecture SIMD a été proposée par les deux principaux fabricants de GPU : Nvidia et ATI.

Cette architecture a évolué afin de permettre la gestion d'un nombre croissant de triangles (plus de dix millions) et un affichage sur des périphériques de résolution plus importante (4096 pixels en largeur) en moins d'un vingt-cinquième de seconde pour satisfaire les exigences qualitatives des éditeurs de jeux vidéos principalement.

Nvidia fut le premier avec le processeur *GeForce 3* à permettre la programmation partielle des GPU en utilisant un assembleur spécifique. L'objectif était d'offrir la possibilité de programmer des rendus propres aux développeurs et d'autres chercheurs ont alors utilisé cette possibilité pour réaliser du *General Purpose computing on Graphical Processing Units* (GPGPU).

Dès lors, en 2007, Nvidia a proposé une API[79] dédié au calcul scientifique nommée CUDA (Compute Unified Device Architecture).

Comparé à la taxonomie des architectures des ordinateurs proposée par Flynn [26] et évoqué à la section 5.1, cette architecture est généralement nommée SIMT (Single Instruction Multiple Thread) car l'approche consiste à stocker un grand nombre de threads en parallèle directement sur le processeur, chaque thread opérant sur son jeu de données.

Dans cet environnement, chaque thread est référencé par un identifiant et dispose de registres et d'entrées/sorties. Chaque thread est exécuté sur un sous-processeur. Les threads sont organisés par blocs uniformes eux-mêmes référencés par un identifiant unique. L'ensemble de ces blocs constitue une grille. Un bloc est exécuté sur un multi-processeur. Chaque multi-processeur dispose d'une mémoire partagée de petite taille (48Ko) permettant aux threads d'un même bloc de communiquer entre eux. Des éléments de synchronisation existent afin de garantir par exemple que l'accès à la mémoire partagée soit réalisé sans décalage par l'ensemble des threads d'un même

[79] Une autre API, OpenCL existe également mais son utilisation est sous-optimale par rapport à une API dédiée telle que CUDA.

Threads	1	128	256	512	1024
Temps CPU (ms)	14680	-	-	-	-
Temps GPU (ms)	-	2864	2781	2738	2719

TABLE 5.1

bloc. Enfin une mémoire globale (DRAM) est disponible sur la carte graphique permettant aux threads de différents blocs de communiquer entre eux et permettant d'échanger des données entre la RAM et DRAM. Les threads peuvent être organisés en 1D, 2D ou 3D au sein d'un bloc et les blocs peuvent être organisés en 1D ou 2D au sein de la grille. Cette organisation est une possibilité laissée au développeur afin de faire concorder l'organisation de la grille à son problème (e.g des blocs 2D sont généralement utilisés dans des calculs matriciels). Les identifiants du thread dans le bloc et du bloc dans la grille permettent d'obtenir un numéro unique pour chaque thread et de déterminer généralement les données sur lesquelles il devra opérer. Notons qu'un thread ne peut accéder qu'à ces registres, la mémoire partagée de son bloc et la DRAM de la carte graphique[80].

La parallélisation de la simulation du circuit consistant à réaliser une évaluation du *bytecode* a été réalisée pour l'architecture CUDA sur un GPU Nvidia Tesla K80. Nous stockons l'ensemble du *bytecode* en DRAM, les threads accèdent à des mots de 128 bits afin d'avoir un accès non-concurrentiel à la mémoire et opèrent sur une donnée. Nous avons effectué un ensemble d'essais en faisant varier le nombre de threads par blocs et en déterminant en conséquence le nombre de blocs dans la grille.

Le tableau 5.1 présente l'ensemble des résultats en temps d'exécution pour différentes configurations sur un *bytecode* de 3 millions d'opérations logiques. Nous présentons également le temps d'exécution sur un processeur CPU Intel Xeon E5-2620 à 2.10GHz pour le même échantillon. Les résultats montrent que l'utilisation du GPU n'apporte aucun gain en temps comparé au CPU en tenant compte du fait que seul un seul thread de celui-ci (parmi 16) est utilisé.

En effet les opérations étant une succession d'opérations logiques chacune est réalisée sur un cycle d'horloge du GPU. Dans ce cas, le temps de transfert des données de la RAM vers la DRAM est extrêmement pénalisant et aboutit à ce résultat. Par ailleurs, cette évaluation du *bytecode* est un préalable à d'autres et nous ne pouvons utiliser le fait que CPU et GPU sont asynchrones pour réaliser ces opérations sur le GPU pendant que le CPU effectue d'autres calculs. Enfin, étant donné que les threads effectuent à un instant t la même opération dans l'architecture CUDA (i.e

[80] Ainsi au préalable à tout calcul sur ce type d'architecture, une copie des données utiles de la RAM vers la DRAM est réalisé. Réciproquement à posteriori les données devant être utilisées par le CPU devront être copiées de la DRAM vers la RAM.

les threads n'étant pas concernés se retrouvent bloqués), notre approche n'est pas adaptée en l'état à ce type d'architecture et il ne nous a pas semblé opportun d'explorer plus en avant cette voie.

5.3.4.3 Utilisation d'une machine spécialisée : MPPA-developper

Collaboration scientifique La société Kalray a développé et commercialisé depuis 2013 une famille de processeurs avec un grand nombre de cœurs (PC) appelée *Multi-Purpose Processor Array* (MPPA). Une collaboration entre Kalray et le Laboratoire d'Informatique Avancée de Saint-Denis de l'Université Paris 8 (LIASD) nous a permis de disposer d'une machine équipée de ce processeur afin d'étudier les possibilités de parallélisation de nos outils de recherche sur cette architecture particulière.

Architecture du MPPA Le processeur mis à notre disposition dispose de 256 cœurs organisés au sein d'une grille de 16 clusters et connectés entre eux par *réseau sur puce* à très haute vitesse. Il est intégré à une carte d'extension dont est équipée une machine équipée d'un processeur Intel i7 (machine hôte). Chaque cluster dispose d'une mémoire dédiée de 2Mo partagée entre les 16 PCs et le cœur *système*. Ce dernier est chargé de superviser l'exécution des tâches et exécute le système NodeOS, alors que chaque PC est dédié à l'exécution d'une seule tâche dont le code est stocké dans la mémoire partagée. La machine hôte et le MPPA communiquent via le bus PCI Express 3. Une suite de logiciels de développement fournie permet d'utiliser différent types de connexions synchrones ou asynchrones utilisant un simple tampon ou des files d'attente.

Dans un rapport technique, Jouandeau a mesuré que les capacités de calcul du MPPA à 256 cœurs était proches de celle de la machine hôte équipée d'un processeur Intel i7 à 8 cœurs [33].

Dans un travail conjoint avec Hufschmitt et al., nous avons étudié la possibilité d'un déport du calcul des *playouts* sur un MPPA. La quantité limitée de mémoire (2Mo) dont disposent les clusters ne permet pas de stocker des arbres de grande taille. Nous n'avons donc pas envisagé de parallélisation au niveau de la racine ou au niveau l'arbre. En revanche, le circuit logique de calcul des transitions entre les états du jeu est constitué d'un grand nombre d'opérations logiques indépendantes. Nous nous sommes donc tournés vers une parallélisation au niveau des feuilles pour le calcul de ces transitions [31] et des *playouts*.

Expériences menées sur le MPPA Nous avons réalisé des expériences sur le TICTACTOE dont les *playouts* sont courts (entre cinq et neuf positions) et donc *bytecode* possède une taille permettant d'être stocké dans la mémoire du MPPA. La performance du programme parallélisé est mesurée en nombre de *playouts* par seconde estimé à partir du calcul de 10000 *playouts* pour chaque jeu.

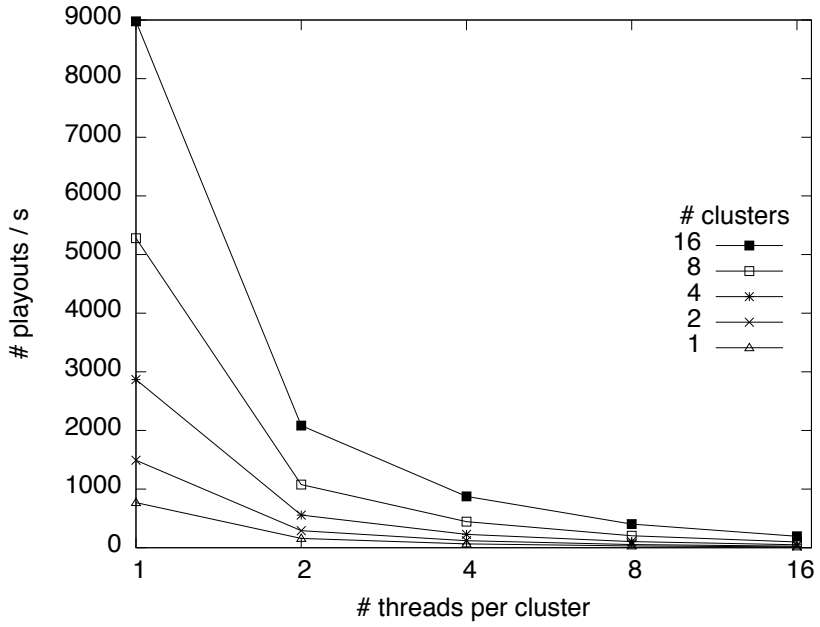


FIGURE 5.6 – Évolution du nombre de *playouts* par seconde (moyenne sur 10000 exécutions) en fonction du nombre de threads utilisés pour le jeu TICTACTOE.

La figure 5.6 présente la vitesse d'exécution en *playouts* par seconde en fonction pour chacun des trois jeux en fonction du nombre de clusters et de threads utilisés. Nous observons que la parallélisation est clairement défavorable car quel que soit le nombre de cluster utilisés, le nombre de *playouts* effectués par seconde s'effondre dès l'utilisation de deux threads. Ce résultat s'explique par le fait que le temps d'évaluation du *bytecode* pour ce jeu très simple est négligeable devant celui des synchronisations entre les threads. Néanmoins, la mémoire limitée du MPPA ne permet pas d'envisager son utilisation avec des jeux plus complexes pour lesquels le temps d'évaluation du *bytecode* deviendrait significatif.

5.3.5 Bilan de la parallélisation de la simulation du circuit

Nous avons présenté deux parallélisations SIMD et une parallélisation MIMD sur trois architectures dont le but est d'accélérer la simulation du circuit logique modélisant les transitions entre les différents états du jeu. Pour cette tâche, nous n'avons retenu dans LEJOUEUR que la parallélisation de type SIMD décrite à la section 5.3.3. Cette parallélisation SIMD qui n'utilise qu'un seul thread nous permet d'utiliser une parallélisation de type MIMD pour l'exploration du graphe du jeu que nous décrivons dans la section suivante.

5.4 Parallélisation de l'exploration du graphe de jeu

Nous présentons à présent les techniques de parallélisation que nous avons mises en œuvre dans LeJoueur pour accélérer l'exploration du graphe de jeu dans le cadre d'un algorithme MCTS. Nous présentons ces différentes techniques en suivant la classification établie par Chaslot et al. en trois catégories : la parallélisation au niveau des feuilles, de la racine et de l'arbre [15]. La parallélisation aux feuilles consiste à n'utiliser qu'un seul thread ou processus pour la politique de l'arbre et d'utiliser plusieurs politiques par défaut en parallèle pour évaluer une position du jeu. Avec une parallélisation à la racine, les différents threads ou processus développent leur propre arbre indépendemment les uns des autres et partagent périodiquement les résultats qu'ils ont obtenus pour la partie haute de l'arbre. La parallélisation de l'arbre utilise plusieurs threads ou processus qui développent le même arbre : dans ce cas, c'est à la fois la politique de l'arbre et la politique par défaut qui sont parallélisées [24, 45]. Les difficultés à paralléliser UCT sont analysées dans [67]. Sur GPU, les méthodes combinant une parallélisation au niveau de l'arbre avec une parallélisation aux feuilles permettent de tirer parti de cette architecture particulière [62, 3].

Nous avons déjà détaillé à la section 5.3 notre technique de parallélisation SIMD sur un seul thread permettant d'obtenir jusqu'à 64 *playouts* en simultané au niveau de chaque feuille. Il s'agit d'une parallélisation aux feuilles. Dans un premier temps, nous expliquons comment nous agrégeons les résultats de ces *playouts* multiples. Nous présentons ensuite la parallélisation avec plusieurs arbres que nous avions mise en place dans une première version de LeJoueur. Enfin nous présentons la parallélisation au niveau de l'arbre qui est celle qui a été utilisée lors des dernières compétition de GGP dans LeJoueur.

5.4.1 Parallélisation au niveau de chaque feuille

La parallélisation au niveau des feuilles de l'algorithme UCT a déjà été testée sur une plateforme distribuée pour le jeu de Go [12]. Un processus maître développe une feuille et délègue à des processus esclaves le calcul d'un ou plusieurs *playouts* à partir de cette feuille. L'ensemble des résultats de ces *playouts* agrégés sous la forme d'une moyenne arithmétique qui sert d'évaluation pour la rétro-propagation. Les résultats montrent que pour ce type de parallélisation, les temps de communication sont importants et n'apportent pas de gain par rapport à d'autres techniques de parallélisation au niveau de l'arbre. Les auteurs mentionnent que ce type de parallélisation est mieux adaptée pour des architectures parallèles à mémoire partagée.

La parallélisation aux feuilles nécessite de faire un choix quant à la méthode d'agrégation des multiples résultats obtenus au niveau de chaque feuille. Nous venons de voir que la moyenne arithmétique peut être utilisée. La moyenne présente l'inconvénient de s'éloigner des conditions de la

politique UCT car elle favorise moins clairement certains nœuds de l'arbre au détriment d'autres. Une méthode de distillation, sur plusieurs itérations de l'algorithme MCTS, des évaluations multiples disponibles au niveau de chaque feuille a été proposée [16].

Notre technique de parallélisation SIMD que nous avons présentée à la section 5.3 permet d'obtenir jusqu'à 64 *playouts* en simultané au niveau d'une seule feuille. Elle s'apparente par conséquent à une parallélisation au niveau des feuilles sans le surcoût induit par les communications. Les 64 *playouts* constituent la politique par défaut de l'algorithme MCTS que nous utilisons. Lors de la phase de rétro-propagation, nous agrégeons ces 64 évaluations en utilisant la médiane ou la moyenne pour les jeux à plusieurs joueurs, et le maximum pour les jeux à un seul joueur.

Nous avons comparé l'agrégation avec la moyenne et la médiane. Pour cela, nous avons mesuré le taux de victoires ces deux approches pour cinq jeux disponibles sur le serveur Tiltyard, en effectuant 500 matchs face à un algorithme UCT classique pour chaque mesure. Chaque joueur ne joue en premier que pour la moitié des matchs. Les conditions de ces matchs sont celles du GGP, à savoir une horloge par coup que nous avons fixée à 10 secondes. Les deux joueurs s'affrontant disposent des mêmes ressources de calcul, à savoir un seul thread de processeur Intel Xeon E5-4610 à 2.40GHz.

Les résultats sont présentés à la table 5.2 et montrent que pour une constante UCT fixée à 0.4[81], nous obtenons des taux de victoire supérieurs pour la médiane pour tous les jeux sauf HEX. Nous expliquons cette meilleure performance de la médiane par le fait qu'il s'agit d'un estimateur plus robuste que la moyenne et qu'elle conduit l'algorithme UCT a explorer plus profondément dans l'arbre avec une meilleure confiance. S'agissant du jeu HEX, la longueur du *bytecode* et le facteur de branchement important ne permet qu'un nombre limité de *playouts* par coup (environ 300 en dix secondes en début de match). Par conséquent, la valeur du meilleur coup ne comporte que moins d'une dizaine d'estimations. Dans ces conditions, la médiane ne permet pas de distinguer finement la valeur des coups et le joueur qui l'utilise possède un comportement proche d'un joueur aléatoire tout comme le joueur utilisant l'algorithme UCT classique contre lequel il joue.

En divisant la constante UCT de 0.4 par 64 pour favoriser l'exploitation, la moyenne permet d'obtenir les meilleurs taux de victoire comme le montrent les résultats de la troisième colonne du tableau. Cette valeur de la constante permet d'obtenir le même effet que la médiane en favorisant l'exploitation et permet de distinguer plus finement la valeur des coups dans le cas des jeux où peu d'estimations sont disponibles. Elle semble un meilleur compromis que la médiane qui a été

[81] Il s'agit de la valeur utilisée par Ary ayant montré le meilleur compromis entre exploitation et exploration et que nous avons reprise dans LEJOUEUR.

	Moyenne $C = 0.4$	Médiane $C = 0.4$	Moyenne $C = 0.4/64$
BREAKTHROUGH	34%	67%	79%
BREAKTHROUGHSUICIDE	64%	74%	76%
HEX	100%	52%	100%
CONNECTFOUR	72%	72%	76%
CHECKERS	90%	92%	98%

TABLE 5.2 – Taux de victoires entre une agrégation de 64 résultats aux feuilles avec la moyenne ou la médiane et deux valeurs de la constante UCT contre UCT avec un seul *playout* par position et $C = 0.4$.

utilisée dans LEJOUEUR dans ses dernières compétitions et a l'avantage d'estimer correctement les probabilités de gain des positions[82].

5.4.2 Parallélisation au niveau de la racine

La parallélisation au niveau de la racine consiste à développer plusieurs arbres UCT indépendants dans des processus distincts à partir d'une position donnée et d'agréger les résultats au moment de choisir un coup. Cette approche permet d'augmenter le niveau de jeu par rapport à un algorithme UCT non parallélisé dans le cas d'une horloge fixe, notamment au Go [13].

Nous avons utilisé la parallélisation à la racine dans les premières versions de LEJOUEUR mais nous l'avons abandonnée au profit d'une parallélisation au niveau de l'arbre, moins gourmande en mémoire, du fait qu'elle ne développe qu'un seul arbre.

5.4.3 Parallélisation au niveau de l'arbre

La parallélisation MIMD au niveau de l'arbre que nous utilisons dans LEJOUEUR consiste à utiliser plusieurs threads qui vont explorer des parties distinctes du même arbre de jeu et mettre en commun leurs résultats.

L'arbre de jeu contient au niveau de chaque sommet son nombre de visites et au niveau de chaque arc reliant deux positions, le nombre de visites de chaque enfant ainsi que le cumul des évaluations renvoyées par la politique par défaut (scores à l'issue d'un *playout*). Ces informations sont utilisées dans le cadre de la politique UCT pour déterminer les sommets à visiter lors de la phase de sélection. La politique de l'arbre est modifiée pour intégrer le *virtual loss* [76] qui

[82]Les moyennes de médianes effectuées lors de la rétro-propagation de l'algorithme surestime la probabilité de victoire du joueur ayant l'avantage.

permet d'éviter que tous les threads aboutissent à la même feuille lors de la phase de sélection, ce qui reviendrait à une parallélisation aux feuilles. Elle consiste à modifier la politique UCT afin que l'incrémentation du nombre de visites soit faite lors de la phase de sélection. Le cumul des évaluations n'étant pas mis à jour, cela revient à considérer immédiatement que l'évaluation de la politique par défaut est nulle. Lors de la phase de rétro-propagation, seul le cumul des évaluations est mis à jour, les nombres de visites étant déjà corrects. Le *virtual loss* permet d'introduire de la diversité lors de la phase de sélection tout en favorisant les branches de l'arbre qui semblent prometteuses. Deux threads sélectionneront plutôt les mêmes sommets en haut de l'arbre et ne divergeront qu'en s'approchant des feuilles.

La parallélisation au niveau de l'arbre permet à LeJoueur d'explorer davantage l'arbre de jeu sur un processeur multi-cœurs. Cette parallélisation est très efficace pour les jeux dont l'interprétation est lente avec une augmentation linéaire du nombre d'explorations en fonction du nombre de threads utilisés. Pour les jeux où l'interprétation est rapide, l'accès concurrent à la mémoire reste un facteur limitant et nécessite une analyse plus approfondie.

5.5 Conclusion

Pour le traitement de la description GDL avant le début du match, nous avons développé une machine de Warren implémentant la résolution SLG avec le tabling. La particularité de cette implémentation est qu'elle crée une machine de Warren pour chaque requête à une procédure tablée et offre ainsi la possibilité d'être facilement parallélisable sur une architecture MIMD.

En ce qui concerne les traitements effectués pendant un match, nous avons testé une parallélisation de l'évaluation du circuit logique de type MIMD sur des architectures CPU, GPU et *many cores* mais celles-ci n'ont pas été concluantes du fait des temps de synchronisation trop important par rapport au temps nécessaire à la traversée du circuit.

Pendant le déroulement d'un match, la parallélisation est utilisée à deux niveau dans LeJoueur. La parallélisation SIMD de l'évaluation du circuit logique permet d'accélérer significativement l'interprétation du GDL et permet d'obtenir jusqu'à 64 *playouts* simultanés sur un seul thread. La parallélisation au niveau de l'arbre utilisant le *virtual loss* permet d'explorer plus rapidement l'arbre de jeu.

Conclusion et perspectives

Dans cette thèse, nous avons présenté LeJoueur, un programme de *General Game Playing* (GGP) que nous avons développé et qui est capable de jouer à une grande variété de jeux décrits en GDL et en GDL-II.

Dans la première partie de ce travail, nous avons proposé des algorithmes permettant l'interprétation rapide du GDL. La méthode d'instanciation rapide que nous avons développée permet de traiter 90% des jeux présents sur les serveurs dans des temps compatibles avec les contraintes du GGP. Grâce à une approche gloutonne, nous avons montré qu'il est possible de transformer cette instanciation en circuit logique ou réseau de propositions et d'en faire une simplification. Enfin, la compilation en *bytecode* du circuit logique permet d'obtenir une interprétation du GDL plus rapide que l'approche reposant sur une traduction du GDL en Prolog. L'analyse statique du circuit logique que nous générons permet de découvrir certaines formes de décomposition des jeux.

Nous avons ensuite proposé une méthode permettant de jouer aux jeux à information incomplète et/ou imparfaite qui s'appuie sur un méta-jeu dont le but est de découvrir les éléments de l'ensemble d'information d'un jeu.

Dans le cadre des méthodes MCTS, nous avons présenté notre approche permettant de jouer aux jeux à coups simultanés et nous détaillons notre contribution dans laquelle nous utilisons RAVE en début de recherche lorsque peu d'estimations sont disponibles avant de basculer automatiquement sur une méthode de Monte-Carlo.

Dans la dernière partie, nous détaillons notre contribution à la parallélisation de l'interprétation du GDL et de la recherche MCTS. Nous avons proposé un algorithme permettant de réaliser plusieurs playouts en simultané sur un seul cœur de processeur plus rapidement qu'un calcul séquentiel de ces playouts. La parallélisation *multi-threads* de l'évaluation du circuit logique présenté dans la première partie sur CPU, GPU et MPPA n'a pas abouti à un gain de performance. Nous avons alors utilisé ces ressources de calcul pour la parallélisation au niveau de l'arbre des MCTS.

LeJoueur possédant à présent sa machine de Warren pour l'interprétation du GDL, nous pouvons envisager d'autres formes de parallélisation de l'instanciation et d'analyse des descriptions

GDL du fait de son utilisation beaucoup moins lourde qu'un interpréteur PROLOG complet. Notre travail sur la parallélisation multi-threads de l'interprétation du GDL offre des perspectives de recherche dans l'optimisation de l'ordonnancement des calculs qui permettrait une meilleure affinité au cache du CPU ou aux architectures particulières comme les GPU. Même si les premiers résultats concernant l'évaluation rapide par circuit sur un GPU ne sont pas concluants, le déport sur le GPU de la construction de l'arbre de recherche MCTS permettrait de s'affranchir du problème de transfert des données entre la RAM et la DRAM.

L'analyse dynamique du circuit logique lors de simulations ouvre des perspectives dans la découverte de nouvelles formes de décomposition des jeux exactes ou approximatives. En émettant des hypothèses sur la décomposition d'un jeu, il est possible de jouer plus rapidement à chacun des sous-jeux tant que l'hypothèse n'est pas infirmée. Si cette décomposition approximative ne dénature pas significativement le jeu initial, on peut alors envisager que l'augmentation du nombre de simulations augmente le niveau du joueur.

Bibliographie

[1] H. Ait-Kaci. *Warren's Abstract Machine : A Tutorial Reconstruction*. MIT Press, Cambridge, MA, USA, 1991.

[2] P. Auer, P. Fischer, and J. Kivinen. Finite-time Analysis of the Multiarmed Bandit Problem. In *Machine Learning*, 2002.

[3] N. A. Barriga, M. Stanescu, and M. Buro. Parallel UCT Search on GPUs. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–7. IEEE, 2014.

[4] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment : An evaluation platform for general agents. *Journal of Artificial Intelligence Research (JAIR)*, 47 :253–279, 2013.

[5] Y. Björnsson and S. Schiffel. Comparison of GDL Reasoners. In *Proceedings of the IJCAI-13 Workshop on General Game Playing (GIGA'13)*, 2013.

[6] B. Bouzy. Monte-Carlo Fork Search for Cooperative Path-Finding. In *Workshop on Computer Games*, pages 1–15. Springer, 2013.

[7] B. Bouzy and T. Cazenave. Computer Go : an AI oriented survey. *Artificial Intelligence*, 132 (1) :39–103, 2001.

[8] B. Bouzy and B. Helmstetter. Monte-Carlo Go Developments. In *Advances in computer games*, pages 159–174. Springer, 2004.

[9] C. Browne and F. Maire. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1) :1–16, March 2010.

[10] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1) :1–43, March 2012.

[11] T. Cazenave. Sequential Halving Applied to Trees. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1) :102–105, 2015.

[12] T. Cazenave and N. Jouandeau. On the Parallelization of UCT. In *Proceedings of the Computer Games Workshop, CGW 2007*, pages 93–101, 2007.

[13] T. Cazenave and N. Jouandeau. A Parallel Monte-Carlo Tree Search Algorithm. In *Computers and Games : 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*, pages 72–80, 2008.

[14] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo Tree Search : A New Framework for Game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008.

[15] G. Chaslot, M. H. Winands, and H. J. van Den Herik. Parallel Monte-Carlo Tree Search. *Computers and Games*, 5131 :60–71, 2008.

[16] M. Chee, A. Saffidine, and M. Thielscher. A Principled Approach to the Problem of Chunking in UCT. In *Computer Games Workshop @ IJCAI (CGW)*, Buenos Aires, Argentina, July 2015.

[17] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *J. ACM*, 43(1) :20–74, Jan. 1996. ISSN 0004-5411.

[18] B. E. Childs, J. H. Brodeur, and L. Kocsis. Transpositions and Move Groups in Monte Carlo Tree Search. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 389–395. IEEE, 2008.

[19] J. Clune. Heuristic Evaluation Functions for General Game Playing. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, 2007.

[20] V. S. Costa, R. Rocha, and L. Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2) :5–34, 2012.

[21] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International Conference on Computers and Games*, pages 72–83, 2006.

[22] P. I. Cowling, E. J. Powley, and D. Whitehouse. Information Set Monte Carlo Tree Search. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(2) :120–143, 2012.

[23] U. Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat, Inc, 2007.

[24] M. Enzenberger and M. Müller. A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm. In *Advances in Computer Games : 12th International Conference, ACG 2009, Pamplona Spain, May 11-13, 2009. Revised Papers*, pages 14–20, 2010.

[25] H. Finnsson. CADIA-Player : A General Game Playing Agent. Master's thesis, Reykjavík University - School of Computer Science, 2007.

[26] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9) :948–960, 1972.

[27] S. Gelly and D. Silver. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, 175(11) :1856–1875, 2011.

[28] M. Genesereth and M. Thielscher. *General Game Playing*. Morgan & Claypool Publishers, 2014.

[29] M. Genesereth, N. Love, and B. Pell. General Game Playing : Overview of the AAAI Competition. *AI magazine*, 26(2) :62, 2005.

[30] J. C. Harsanyi. Games with Incomplete Information Played by Bayesian Players, I-III. *Management Science*, 14(3) :159–182, 1967.

[31] A. Hufschmitt, J. Méhat, and J.-N. Vittaut. MCTS Playouts Parallelization with a MPPA Architecture. In *Proceedings of the IJCAI-15 Workshop on General Game Playing (GIGA'15)*, pages 63–69, 07 2015.

[32] A. Hufschmitt, J.-N. Vittaut, and J. Méhat. A General Approach of Game Description Decomposition for General Game Playing. In *Computer Games : 5th Workshop on Computer Games (CGW 2016), and 5th Workshop on General Intelligence in Game-Playing Agents (GIGA 2016) @IJCAI, New York, USA, July 9-10, 2016*, 2017.

[33] N. Jouandeau. Intel versus MPPA. Technical report, LIASD Université Paris8, 11 2013.

[34] Z. Karnin, T. Koren, and O. Somekh. Almost optimal exploration in multi-armed bandits. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1238–1246, 2013.

[35] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo Planning. In *In : ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[36] J. Kowalski and A. Kisielewicz. Game description language for real-time games. In *4th Workshop on General Intelligence in Game-Playing Agents (GIGA '15)*, pages 23–30, 2015.

[37] J. Kowalski and A. Kisielewicz. Towards a Real-time Game Description Language. In *ICAART (2)*, pages 494–499, 2016.

[38] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson. General Video Game Playing. In *Artificial and Computational Intelligence in Games*, volume 6, pages 77–83. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[39] V. Lisý, M. Lanctot, and M. Bowling. Online Monte Carlo Counterfactual Regret Minimization for Search in Imperfect Information Games. In *Proceedings of the Fourteenth International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 27–36, 2014.

[40] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing : Game Description Language Specification. Technical report, 2008.

[41] E. M. Luks. Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time. *Journal of Computer and System Sciences*, 25 :42–65, 1982.

[42] J. Mallet. ZRF Language Reference. Technical report, 2003.

[43] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Communications of the ACM*, 3(4) :184–195, 1960.

[44] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Readings in artificial intelligence*, pages 431–450, 1969.

[45] S. A. Mirsoleimani, A. Plaat, J. van den Herik, and J. Vermaseren. Parallel Monte Carlo Tree Search from Multi-Core to Many-Core Processors. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 3, pages 77–83. IEEE, 2015.

[46] J. Méhat. *Contrôle et programmation des architectures pyramidales*. PhD thesis, Université Paris 8, 1989.

[47] J. Méhat and T. Cazenave. A Parallel General Game Player. *KI 2011 : Advances in Artificial Intelligence*, 25(1) :43–47, 2011.

[48] J. Méhat and J.-N. Vittaut. Online Adjustment of Tree Search for GGP. In *GIGA 2013 - General Intelligence in Game-Playing Agents - The IJCAI-13 Workshop on General Game Playing - Beijing, China, August 2013*, 2013.

[49] B. Pell. METAGAME : A new challenge for games and learning. In *Euristic Programming in Artificial Intelligence : The Third Computer Olympiad*. Ellis Horwood, 1992.

[50] B. Pell. METAGAME in Symmetric Chess-Like Games. In *Euristic Programming in Artificial Intelligence : The Third Computer Olympiad*. Ellis Horwood, 1992.

[51] B. Pell. *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, University of Cambridge, 1993.

[52] T. Pepels, T. Cazenave, M. H. Winands, and M. Lanctot. Minimizing Simple and Cumulative Regret in Monte-Carlo Tree Search. In *Workshop on Computer Games*, pages 1–15. Springer, 2014.

[53] D. Perez-Liebana, S. Samothrakis, J. Togelius, S. M. Lucas, and T. Schaul. General Video Game AI : Competition, Challenges and Opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[54] E. Piette. *Une nouvelle approche au General Game Playing dirigée par les contraintes*. PhD thesis, Université d'Artois, 2016.

[55] J. Pitrat. Realization of a General Game-Playing Program. In *Information Processing*, pages 1570–1574, 1968.

[56] J. Pitrat. A General Game Playing Program. In *Artificial Intelligence and Heuristic Programming*, pages 125—155, 1971.

[57] M. Quenault. *Une application de jeux génériques pour les jeux symboliques*. PhD thesis, LAMSADE, Université Paris Dauphine, 2010.

[58] E. Rasmusen. *Games and information : an introduction to Game Theory*. Wiley-Blackwell, 4th edition, 2006.

[59] R. Rocha, F. Silva, and V. S. Costa. YapOr : an Or-Parallel Prolog System based on Environment Copying. Technical report, Universidade do Porto, 1997.

[60] R. Rocha, F. Silva, and V. S. Costa. A tabling engine for the YAP Prolog system. In *Proceedings of the 2000 APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP 2000)*, December 2000.

[61] R. Rocha, F. Silva, and V. Santos Costa. YapTab : A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[62] K. Rocki and R. Suda. Large-Scale Parallel Monte Carlo Tree Search on GPU. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2034–2037. IEEE, 2011.

[63] A. Saffidine. The Game Description Language is Turing Complete. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2) :320–324, December 2014.

[64] A. Saffidine, T. Cazenave, and J. Méhat. UCD : Upper Confidence Bound for Rooted Directed Acyclic Graphs. *Knowledge-Based Systems*, 34 :26–33, 2012.

[65] S. Schiffel and M. Thielscher. Fluxplayer : A successful General Game Player. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2007.

[66] M. J. Schofield, T. J. Cerexhe, and M. Thielscher. HyperPlay : A Solution to General Game Playing with Imperfect Information. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[67] R. B. Segal. On the Scalability of Parallel UCT. *Computers and Games*, 6515 :36–47, 2010.

[68] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587) :484–489, 2016.

[69] N. Sturtevant. A Comparison of Algorithms for Multi-Player Games. In *In Proc. of the Third International Conference on Computers and Games*, 2002.

[70] N. Sturtevant and R. E. Korf. On Pruning Techniques for Multi-Player Games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 201–207. AAAI Press, 2000.

[71] M. Thielscher. GDL-II. *KI - Künstliche Intelligenz*, 25(1) :63–66, Mar 2011.

[72] J.-N. Vittaut and J. Méhat. Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic*, 2014.

[73] J.-N. Vittaut and J. Méhat. Efficient Grounding of Game Descriptions with Tabling. In *Computer Games : Third Workshop on Computer Games, CGW 2014, 2014*, 2014.

[74] D. H. D. Warren. An abstract Prolog instruction set. Technical report, Menlo Park, CA, USA : Artificial Intelligence Center at SRI International, October 1983.

[75] M. H. Winands, Y. Björnsson, and J.-T. Saito. Monte-Carlo Tree Search Solver. *Computers and Games*, 5131 :25–36.

[76] I.-C. Wu, H.-H. Lin, P.-H. Lin, D.-J. Sun, Y.-C. Chan, and B.-T. Chen. Job-Level Proof-Number Search for Connect6. In *International Conference on Computers and Games*, pages 11–22. Springer, 2010.