UNIVERSITE PARIS 8
Vincennes - Saint-Denis

LIASD
Laboratoire Informatique Avancée de Saint-Denis
Université Paris 8

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS 8
ÉCOLE DOCTORALE ED401

Spécialité
Informatique

Pour obtenir le grade de docteur délivré par
Université Paris 8 Vincennes à Saint-Denis

Décomposition des jeux dans le domaine du *General Game Playing*

Présentée et soutenue publiquement par
Aline Hufschmitt
le **04 octobre 2018**

Jury

Tristan Cazenave	(Professeur, Université Paris Dauphine)	Rapporteur
Bruno Bouzy	(MCF-HDR, Université René Descartes)	Rapporteur
Myriam Lamolle	(Professeur, Université Paris 8)	Examinatrice
Lakhdar Saïs	(Professeur, Université d'Artois)	Examinateur
Sylvain Lagrue	(Professeur, Université de technologie de Compiègne)	Examinateur
Abdallah Saffidine	(Chercheur, Australian National University)	Examinateur
Nicolas Jouandeau	(MCF-HDR, Université Paris 8)	Directeur de thèse
Jean-Noël Vittaut	(PRAG - Docteur, Université Paris 8)	Co-directeur de thèse

Remerciements

Ever tried. Ever failed. No matter...
Try again. Fail again. But fail better.
[Samuel Beckett]

Je remercie Jean Mehat, que j'ai croisé par hasard dans les années 90 dans les couloirs de Paris 8, et qui en partageant son immense culture scientifique, technique et historique de l'informatique, a su éveiller ma curiosité. Je le remercie une seconde fois d'avoir nourri cette curiosité quand j'ai décidé d'entamer tardivement des études d'informatique et une troisième fois d'avoir accepté de diriger mes recherches en master puis en thèse.

Je remercie Nicolas Jouandeau pour son soutien et ses conseils depuis le début de mes recherches. Merci d'avoir prit la direction de ma thèse quand Jean nous a quittés et de m'avoir aidée à finaliser ces travaux.

Je remercie Jean-Noël Vittaut, avec qui j'ai travaillé sur la même thématique, pour ses idées, ses avis, ses conseils qui m'ont énormément aidée à avancer dans mes recherches. Je le remercie également d'avoir accepté de co-diriger ma thèse juste après avoir fini la sienne.

Je remercie Abdallah Safidine qui, en présentant une solution systématique au problème de la (dé-)composition [Cerexhe *et al.*, 2014] à Dauphine, m'a donné envie de travailler sur cet aspect particulier du GGP.

Je remercie Bruno Bouzy et Tristan Cazenave d'avoir accepté d'être les rapporteurs de cette thèse ainsi que Myriam Lamolle, Lakhdar Saïs, Sylvain Lagrue et Abdallah Saffidine d'en être les examinateurs/trices.

J'adresse mes remerciements à tous les membres du LIASD qui m'ont soutenue, conseillée et aidée.

Je remercie mon frère qui m'a initiée à la programmation et m'a un jour présenté Jean Mehat.

Je remercie mon mari qui a supporté avec patience mes absences, mes sautes d'humeur, mes inquiétudes et mes discours interminables sur les jeux (dé)composés, qui m'a réconfortée souvent, et qui m'a soutenue depuis le début de mes études d'informatique en 2010.

Merci à ma mère pour la correction des coquilles et toutes ces parties de scrabble qui m'ont changé les idées dans les moments de découragement (mon niveau au scrabble s'est bien amélioré ! ou alors elle m'aurait laissé gagner ?).

Merci à mon père qui, le fer à souder entre les dents, entre un ZX81 et un Apple II, m'a transmis sa passion pour toutes les sciences.

À Jean,

Table des matières

Introduction

Dans cette thèse nous présentons deux méthodes générales de décomposition des jeux dans le domaine du *General Game Playing* (GGP) et une méthode pour exploiter ces décompositions dans un joueur programmé. Le GGP est une branche de l'Intelligence Artificielle (IA) dont l'objet est la conception de programmes intelligents capables de jouer à une grande diversité de jeux. Ces programmes doivent définir une stratégie optimale par analyse d'une description logique de ces jeux, sans utiliser de connaissances expertes préétablies et spécifiques à un jeu.

Les jeux offrent un cadre idéal pour l'étude des algorithmes de recherche et de décision, regroupés dans le domaine de l'IA, en offrant une diversité de problèmes non triviaux dans un cadre fini et maîtrisé. En pratique, l'amélioration des performances des joueurs programmés porte sur deux aspects : l'augmentation de la rapidité d'évaluation des règles logiques des jeux et l'amélioration des algorithmes de décision permettant l'évaluation des états des jeux.

Depuis 2005, conjointement au développement des programmes GGP intelligents, des jeux composés sont proposés dans l'espoir de stimuler la recherche d'algorithmes fondés sur une analyse de haut niveau des règles plutôt que sur une exploration aveugle. Dans ce contexte, un joueur capable de décomposer un jeu, de résoudre chaque partie séparément et de synthétiser ces solutions en une solution globale, pourrait significativement diminuer le coût de l'exploration et de la résolution de celui-ci.

Malgré le gain potentiel que la décomposition d'un jeu laisse espérer, très peu de travaux sur le sujet ont été menés. Ces jeux composés sont très divers et plusieurs classes peuvent être identifiées. Les travaux menés entre 2009 et 2014 proposent des solutions plus ou moins ad hoc pour décomposer certaines de ces classes de jeux et quelques approches pour l'exploitation de ces décompositions. Cependant, aucune des méthodes de décomposition proposées n'est assez générale pour être utilisable dans un joueur programmé en compétition.

Contributions

Notre première contribution est l'évaluation d'une architecture spécifique, le *Multi Purpose Processor Array*, et l'étude des possibilités de parallélisation d'un joueur programmé, *LeJoueur* de Jean-Noël Vittaut, sur cette architecture [Hufschmitt *et al.*, 2015b,a].

Notre deuxième contribution est la mise au point d'une méthode générale de décomposition des jeux décrits en *Game Description Language* (GDL) [Hufschmitt *et al.*, 2016, 2017]. Dans la continuité des travaux existants, cette méthode s'appuie sur une analyse logique des règles. Bien que plus générale que les précédentes, cette méthode présente différents défauts qui en limitent la performance et la robustesse.

Notre troisième contribution est une autre méthode générale de décomposition [Hufschmitt *et al.*, 2018a,b], plus robuste, fondée sur une analyse statistique d'informations collectées pendant des simulations. Cette méthode résout les problèmes soulevés par l'approche précédente et permet d'envisager l'exploitation des décompositions dans un joueur programmé.

Notre quatrième et dernière contribution est une méthode pour exploiter ces décompositions dans un joueur programmé. Nous proposons une variation des méthodes de Monte-Carlo appliquées au GGP permettant d'explorer plusieurs sous-arbres d'un jeu décomposé afin de résoudre plus efficacement le jeu global.

Plan de lecture

La partie I est consacrée à une présentation du GGP. Nous commençons par une présentation générale du domaine, du GDL utilisé pour décrire les jeux, des compétitions permettant d'évaluer les joueurs programmés et des principaux joueurs qui s'y sont distingués. Nous présentons ensuite l'état de l'art concernant les deux facettes d'un joueur programmé : l'interprétation des règles, à travers notamment l'usage des *propnets* et leur transformation en circuit logique, et le choix d'une action optimale, effectué grâce aux méthodes de Monte-Carlo (MCTS) associées à l'usage d'heuristiques.

Dans la partie II, nous présentons les différentes techniques de parallélisation des algorithmes MCTS et nous présentons l'étude d'une architecture originale, le *Multi Purpose Processor Array* de la société Kalray, pour la parallélisation de *LeJoueur* de Jean-Noël Vittaut.

Dans la partie III, nous commençons par présenter un *bestiaire* des jeux composés puis nous dressons un état de l'art de la recherche concernant la décomposition d'un problème en sous-parties dans les domaines de la planification, de la satisfiabilité booléenne (SAT), de la satisfaction de contraintes (CSP), de la vérification de modèles (*model checking*) et du GGP. Ensuite, nous présentons la contribution majeure de cette thèse : deux méthodes générales de décomposition des jeux GGP, l'une fondée sur une analyse logique des règles, l'autre sur une analyse statistique de simulations de parties. Nous terminons par la présentation d'une idée originale permettant de jouer avec ces jeux décomposés : la réalisation de simulations globales pour construire différents sous-arbres. Nous expliquons toutes les pistes de recherche ouvertes par cette idée.

Première partie

General Game Playing

Dans cette partie nous présentons le domaine du *General Game Playing* et l'état de l'art concernant les joueurs programmés. Nous détaillons les techniques utilisées par les joueurs actuels pour interpréter les règles des jeux d'un côté et déterminer une stratégie optimale de l'autre.

Chapitre 1
Présentation du *General Game Playing*

Sommaire

Le *General Game Playing* (GGP) est un champ de recherche faisant partie de l'*Intelligence Artificielle*, visant à réaliser des programmes capables de jouer à des jeux inconnus en partant uniquement de la description de leurs règles. Le concepteur du programme n'a pas connaissance du jeu transmis à son programme et ne doit pas intervenir pendant le jeu. Les premiers travaux sur les jeux généraux, ayant pour objet la création de joueurs polyvalents, ont été initiés par Pitrat [1968, 1971, 1976], Sutton [1988] et Pell [1993], mais le concept de GGP n'a pris sa véritable ampleur qu'avec le projet initié en 2005 par le *Stanford Logic Group* [1] de l'université éponyme.

La recherche dans le domaine de la programmation des jeux était initialement motivée par l'idée que la victoire d'un programme sur un joueur humain pourrait être une mesure fiable de son intelligence telle que recherchée dans le domaine de l'IA. Cependant, avec la victoire de Deep Blue sur Garry Kasparov en 1997 et la réalisation d'autres programmes spécialisés comme Chinook (Dames anglaises) cette idée a montré ses limites. Pell [1993] indique que les programmes experts dans un jeu mettent en œuvre des méthodes d'ingénierie spécialisées allant jusqu'à des matériels dédiés, l'analyse du jeu étant plus le résultat de l'intelligence du programmeur que d'une quelconque avancée en intelligence artificielle. Pell propose de changer les données du problème de manière à forcer les chercheurs à concevoir des programmes faisant preuve d'une intelligence plus générale : à partir des seules règles du jeu, le programme devrait les analyser pour en déduire une stratégie sans intervention humaine. Swiechowski et Mandziuk [2014] confirment ce point de vue et présentent le GGP comme la tentative la plus récente et la mieux conçue pour remettre d'actualité les objectifs des débuts de l'IA visant à se rapprocher de l'intelligence

1. Stanford Logic Group : `http://ggp.stanford.edu/`

générale.

Depuis 2005 une compétition annuelle de GGP organisée en marge des conférences AAAI et/ou IJCAI permet d'évaluer les programmes joueurs et depuis 2009, le *workshop General Intelligence in Game Agent* (GIGA) de la conférence IJCAI est spécifiquement consacré à la présentation d'articles scientifiques concernant le GGP. Depuis 2013, un MOOC organisé par Michael Genesereth et le groupe de logique de Stanford est également consacré à ce sujet. Une communauté plus nombreuse de chercheurs et amateurs éclairés se sont intéressés au sujet, augmentant la participation aux compétitions et permettant le partage de nouvelles techniques accroissant le niveau des joueurs programmés. Les compétitions, peu à peu dématérialisées, ont débouchées également sur la mise en place de plate-formes de jeu en ligne sur lesquelles les différents joueurs programmés peuvent s'affronter.

Les jeux proposés dans le cadre du GGP sont des jeux discrets, finis, déterministes à information complète [Love *et al.*, 2006] qui peuvent être très variés [Schreiber, 2017] : jeux mono- ou multijoueurs, à coups simultanés ou alternés, à somme nulle ou non-nulle, avec une récompense numérique pour chaque joueur définie entre 0 et 100 [2]. Kowalski et Kisielewicz [2016] proposent une modification du langage de description des jeux utilisé pour le GGP afin d'y intégrer une composante temps réel. Cependant, le GGP ne s'intéresse toujours qu'aux jeux joués au tour à tour et non aux jeux en temps réel comme les jeux vidéo. Ces derniers constituent un autre domaine de recherche [Ebner *et al.*, 2013] qui sort du cadre de nos travaux.

Nous commençons par présenter le GDL utilisé pour décrire les jeux. Nous présentons ensuite les différentes compétitions, sur site ou à distance, permettant l'évaluation des joueurs. Enfin, nous présentons les spécificités des différents joueurs vainqueurs de la compétition et représentant l'état de l'art.

1.1 *Game Description Language* (GDL)

Afin de permettre aux différents joueurs programmés de jouer ensemble, il était nécessaire de concevoir un langage pour décrire les jeux. Plusieurs langages de description des jeux ont été proposés [Pell, 1993; Kaiser, 2005; Browne, 2016], mais c'est le GDL proposé par Genesereth *et al.* [2005] qui s'est imposé comme standard pour le GGP. Conçu pour les jeux déterministes à information complète, Thielscher [2011] montre néanmoins qu'il est suffisamment général pour décrire des jeux très différents. Saffidine [2014] indique même que le GDL est Turing complet. Le GDL peut être facilement étendu aux jeux à information incomplète, soumis à une part de hasard et aux jeux épistémiques. Thielscher [2010, 2017] propose d'ailleurs dans ce but les versions GDL-II [3] et GDL-III [4] que nous ne détaillerons pas ici car elles sortent du cadre de nos travaux [5] De très nombreuses descriptions de jeux en GDL sont proposées sur le site `http://games.ggp.org/` [Schreiber, 2017] qui regroupe les descriptions de jeux proposées sur les

2. La récompense maximale est généralement 100, mais il peut y avoir des exceptions. Dans le jeu *Hunter*, le joueur doit capturer tous les pions présents sur le plateau de jeu en seulement 14 coups, hors cet objectif est impossible à atteindre. Le score maximum qui peut être atteint dans ce jeu est donc inférieur à 100. Dans les jeux *Onestep* ou *Pearls* le score maximal définit dans la description du jeu est 90.
3. Les lettres "II" figurent ici pour *Information Imparfaite*.
4. Le troisième "I" figure ici pour *Introspection*.
5. Actuellement, il n'existe pas à notre connaissance de jeu composé, i.e. conçu explicitement pour être décomposable, en GDL-II et GDL-III. Il serait néanmoins facile d'en concevoir, comme par exemple un *Dual Blind Tictactoe* ou un *Serial Muddy Children*.

serveurs de Dresde [Günther et Schiffel, 2013], de Stanford [Genesereth, 2018] et de Tiltyard [Schreiber, 2014] [6].

1.1.1 Syntaxe du langage

Le GDL (version 1) dont la description est donnée par les créateurs de la compétition [Genesereth *et al.*, 2005; Love *et al.*, 2006] est un langage logique dont la sémantique est très proche de celle de datalog. Il est fondé sur la logique du premier ordre sans la notion d'arithmétique, bien que les nombres de 0 à 100 soient définis comme constantes pour décrire les scores en fin de partie. Un exemple de jeu décrit en GDL est donné à la figure 1.1. Le style du GDL est purement déclaratif et il possède quelques mots clefs pour décrire le jeu [7] :

- `(role r)` signifie que `r` est un joueur dans le jeu.
- `(base b)` permet de préciser qu'un terme `b` est une proposition décrivant un état du jeu.
- `(input r o)` permet de préciser que `o` est une proposition décrivant une option du joueur `r` dans le jeu.
- `(init p)` signifie que la proposition `p` est vraie à l'état initial.
- `(true p)` signifie que la proposition `p` est vraie à l'état courant.
- `(legal r o)` signifie que l'option `o` est un *coup légal* pour le joueur `r` dans l'état courant.
- `(does r o)` est une action signifiant que le joueur `r` a choisi de jouer l'option `o` dans l'état courant.
- `(next p)` signifie que la proposition `p` sera vraie à l'état suivant.
- `terminal` signifie que l'état courant est un état final.
- `(goal r n)` signifie que le joueur `r` obtient la récompense `n` si l'état courant est un état final.

Les mots clefs `true` et `does` ne sont utilisés que dans le corps des règles. Les règles dont la conclusion utilise le prédicat `legal`, `goal` ou `terminal`, ne peuvent accepter que le mot clef `true` parmi leurs prémisses. Les règles `next` [8] n'acceptent que les mots clefs `true` et `does` parmi leurs prémisses. Chaque description peut utiliser autant de *prédicats auxiliaires* que nécessaire pour décrire les différents éléments logiques du jeu (alignement de pions, plateau de jeu totalement rempli, etc.). Ces *prédicats auxiliaires* ne font pas partie des mots clefs du langage. Par exemple, sur la figure 1.1 `row`, `column`, `diagonal`, `line` sont des *prédicats auxiliaires*.

D'autre termes constituent le langage logique :

6. Les jeux issus du serveur de Tiltyard sont disponibles dans le dépôt *Base* de `http://games.ggp.org/`.

7. D'autres termes ont été ajoutés par la suite. `(base a)` et `(input r a)` font partie du standard utilisé par Stanford, mais n'étaient pas présents dans la version de GDL utilisée pendant les premières années de la compétition. Ils ont été ajoutés en 2008 pour faciliter la traduction du GDL vers d'autres modèles (les *propnets* notamment). La version 2 de GDL utilise également les termes `(sees r f)` et `random` pour indiquer respectivement un fait `f` perçu par le joueur `r` dans l'état suivant et un élément soumis au hasard. La version 3 ajoute les termes `(knows p)` et `(knows r p)` pour indiquer respectivement un fait `p` coonu de tous ou uniquement du joueur `r`

8. Pour éviter de lourdes répétitions nous utiliserons directement l'expression « règle `next` », « règle `legal` », etc. pour signifier « règle dont la conclusion utilise le prédicat `next` » ou « règle dont la conclusion utilise le prédicat `legal` », etc.

- `(<= x y0 y1 y2 ... yn)` signifie que les faits `y0` à `yn` impliquent `x`. Les `ET` logiques sont implicites entre les éléments `y0` à `yn`.

- `?x` signifie que *x* est une variable.

- `(not x)` permet d'obtenir la négation d'un fait.

- `(distinct ?x ?y)` indique que les éléments `?x` et `?y` ne peuvent être syntaxiquement identiques.

- Les `OU` logiques sont implicites entre plusieurs clauses de même conclusion. Pour désigner l'ensemble des *clauses* de même conclusion nous parlerons de *règle*. La première version du GDL comprenait un `OU` explicite `(or x y)` qui a été supprimé par la suite. Cependant on le trouve encore dans certaines descriptions [9] (fig.1.1).

Les prédicats `base` indiquent les différents termes représentant un état du jeu, aussi nommés `fluents`. Parmi ces termes, certains se retrouvent dans le prédicat `init` décrivant l'état initial ou le prédicat `true`, indiquant un fluent vrai dans l'état courant. À chaque `true` correspond un `next` indiquant l'état suivant du fluent. De même, le prédicat `input` fait écho aux prédicat `legal` et `does` concernant les actions. Certains fluents devenant vrais ou faux définitivement durant le jeu sont désignés comme étant des *latches*.

Le GDL fait l'hypothèse d'un monde clos. Tous les éléments du jeu sont finis (ou dénombrables) et ce qui n'est pas prouvé vrai est faux par défaut. Ceci implique que les règles `next` ne spécifient que les fluents qui seront vrais dans l'état suivant. Celui qui décrit un jeu doit donc spécifier des *axiomes du cadre* pour indiquer les fluents qui restent vrai d'un état à l'autre. Par exemple sur la figure 1.1, les deux premières règles de conclusion `next` sont des *axiomes du cadre*.

La description d'un jeu peut être réalisée de nombreuses manières différentes et il n'existe pas de forme canonique pour une description GDL. Par exemple, dans la première règle de la figure 1.1, le fluent garde sa valeur pour toute action qui ne le modifie pas. Cette première règle peut être remplacée par la suivante :

```
(<= (next (cell ?m ?n b)) (true (cell ?m ?n b))
    (not (does ?w (mark ?m ?n)))
```

Dans cette deuxième formulation des règles, le fluent garde sa valeur si on ne fait pas l'action qui peut le modifier. Concernant le jeu *Tictactoe*, une troisième formulation est possible en rendant implicite l'existence des cases blanches : une case blanche est simplement ni marquée d'un *x*, ni marquée d'un *o*.

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

```
(role xplayer)
(role oplayer)

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

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

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

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

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

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

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

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

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

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

FIGURE 1.1 – Description du jeu de *Tictactoe* en GDL (serveur Tiltyard [Schreiber, 2014]).

1.1.2 Description GDL bien formée

Un jeu décrit en GDL se déroule de l'état initial à un état terminal où le score (entre 0 et 100) est déterminé pour chaque joueur en fonction de la position [10] atteinte. À chaque tour de jeu, tous les joueurs choisissent une action (une seule action par joueur à chaque tour). L'ensemble des actions de tous les joueurs pour un tour du jeu constitue un *mouvement joint*. Certaines restrictions permettent d'assurer que toutes les règles logiques d'un jeu décrit en GDL sont totalement décidables. Les spécifications GDL garantissent que dans une position donnée, il est possible de calculer les actions légales pour chaque joueur, et pour un mouvement joint donné, un seul état suivant est possible. Cependant, pour éviter certains jeux problématiques, les descriptions doivent également respecter les contraintes suivantes pour être *bien formées* [Love *et al.*, 2006] :

Definition 1.1 (Terminaison)**.** *Une description GDL garantit la terminaison du jeu si toute séquence de coups légaux joués à partir de l'état initial atteint un état terminal après un nombre fini d'étapes.*

Definition 1.2 (Jouabilité)**.** *Une description GDL est jouable si et seulement si, à partir de l'état initial, chaque rôle dispose d'un coup légal dans chaque état non-terminal du jeu.*

Definition 1.3 (Gagnabilité)**.** *Une description GDL est gagnable faiblement* [11] *si et seulement si, pour chaque rôle, il existe une séquence de mouvements joints depuis la position initiale menant à un état terminal où le score est maximal pour ce rôle indépendamment des actions des autres joueurs.*

Definition 1.4 (Évaluation)**.** *Une description GDL garantit l'évaluation si est seulement si chaque rôle obtient un score et un seul dans chaque état terminal.* [12]

Definition 1.5 (Bien formé)**.** *Une description GDL est bien formée si elle garantit la terminaison, l'évaluation et si elle est à la fois jouable et gagnable faiblement.*

Les organisateurs des compétitions s'efforcent de proposer des jeux respectant ces contraintes, cependant Saffidine [2014] a démontré que déterminer si une description de jeu arbitraire est bien formée est en fait un problème indécidable.

1.1.3 Notation du GDL

La notation utilisée dans l'exemple donné dans la figure 1.1 s'inspire du *Knowledge Interchange Format* (KIF) et présente une syntaxe similaire à celle du Lisp : toutes les expressions sont représentées

10. Dans le domaine des jeux, une position est définie par un ensemble de variables matérielles, positionnelles et de contrôle. Une position est donc définie par application d'une séquence d'actions à partir d'une position initiale donnée. Une transposition désigne une même position obtenue par application de deux séquences d'actions différentes. En GGP, une position (également appelée état) correspond à un ensemble de fluents vrais ; ce qui implique qu'une position est également définie par un ensemble implicite de fluents faux et qu'une transposition reste une position obtenue par application de deux séquences d'actions différentes.

11. Une description GDL est gagnable fortement si et seulement si, pour un rôle, il existe une stratégie gagnante i.e. une séquence d'actions menant à un état terminal où le score est maximal pour ce rôle quelles que soient les actions des autres joueurs. Notons, que tout jeu à un seul joueur bien formé est gagnable fortement.

12. Nous avons remplacé la contrainte de *monotonicité* (Définition 23 de Love *et al.* [2006]) par la contrainte plus faible d'*évaluation*. Comme l'indique Saffidine [2014], la monotonicité n'est pas une contrainte judicieuse pour la modélisation des jeux, et en pratique de nombreux jeux décrits en GDL ne sont pas monotones, mais garantissent une évaluation.

```
(<= (legal player (move ?x ?y))
    (true (cell ?x ?v b))
    (or (succ ?y ?v) (pred ?y ?v)) )
```

(a) *notation KIF*

```
legal(player, move(X, Y)) :-
    true(cell(X, Y, b)),
    (succ(Y, V) ; pred(Y, V)).
```

(b) *notation Datalog*

FIGURE 1.2 – Une règle du jeu *Eight Puzzle* en notation KIF et en notation Datalog

sous forme d'un terme unique (atome) ou d'une liste d'expressions dont la première est un prédicat. C'est cette notation qui est utilisée pour transmettre les règles aux joueurs ou comme format d'échange pour les différentes règles disponibles dans les dépôts. Cependant, pour des raisons de simplicité de lecture, une autre notation proche de Datalog (ou Prolog) est fréquemment utilisée dans les publications scientifiques concernant le GGP (fig.1.2b).

1.1.4 Habitudes d'écriture

A chaque tour du jeu chaque joueur doit spécifier une action. Les actions des joueurs sont donc toujours simultanées. Dans les jeux à coups alternés, afin de simuler l'alternance des coups, les joueurs qui n'ont pas la main ont pour seule action légale une action sans effet. Cette action est souvent nommée *noop* dans les versions non obfusquées des jeux. Par référence à ces actions, nous désignerons toute action qui ne modifie pas la position du jeu comme étant une action *noop*.

Dans les jeux à coups alternés, pour garder trace du joueur `?x` qui a la main à un tour donné du jeu, le fluent `(control ?x)` (dans les jeux non obfusqués) est utilisé. Généralement des règles `next` indépendantes des actions indiquent l'état suivant de ce fluent. Le joueur 1 obtient la main si c'est le joueur 2 qui l'a actuellement et inversement :

```
(<= (next (control player1)) (true (control player2))
(<= (next (control player2)) (true (control player1))
```

Pour garantir la terminaison d'un jeu, un compteur ou *stepper* est généralement utilisé. Ce *stepper* est constitué d'un ensemble de fluents `(step 1)`, `(step 2)`, etc. Le premier *step* est ajouté à l'état initial du jeu et une règle `next` indépendante des actions décrit le passage d'un *step* à l'autre. Comme le GDL n'inclut pas de notion d'arithmétique, un ensemble de prédicats statiques est souvent utilisé pour décrire la notion de successeur. Le dernier *step* est utilisé comme condition de terminaison du jeu :

```
(init (step 1))
(<= (next (step ?y)) (true (step ?x)) (succ ?x ?y))
(succ 1 2) (succ 2 3) (succ 3 4)
(<= terminal (true (step 4))
```

1.2 Compétitions internationales de GGP

Une compétition internationale de GGP (IGGPC) est organisée tous les ans depuis 2005 par Stanford [Genesereth *et al.*, 2005]. La compétition était au départ organisée en marge de l'International Joint Conference on Artificical Intelligence (IJCAI) ou de la conférence de l'Association pour l'Avancement de l'Intelligence Artificielle (AAAI). Les conditions de la compétition ont évolué depuis sa conception. Ces dernières années, pour des raisons pratiques, bien qu'organisée en été aux mêmes dates que la conférence IJCAI (devenue annuelle), l'IGGPC s'est peu à peu dématérialisée et se déroule totalement en ligne depuis 2014.

Les résultats de cette compétition donnent une évaluation relative des programmes réalisés [Méhat, 2013]. Le hasard du tirage des adversaires, le faible nombre de matchs et la nature des jeux peut favoriser certains concurrents : en 2010, par exemple, aucun jeu pour un seul joueur n'a été proposé. Toutes sortes de problèmes peuvent survenir pendant les compétitions : problèmes d'ingénierie ou défaillances des machines qui n'ont pas forcément de rapport direct avec la qualité des modèles développés. Les dernières compétitions proposent une phase préparatoire pour permettre aux concurrents de tester leur joueur et aux organisateurs de vérifier le fonctionnement de leur serveur.

Les compétitions de GGP de Stanford sont quasiment les seules rencontres internationales organisées dans le monde et les résultats de ces compétitions étaient jusqu'en 2011 à peu près [13] les seules évaluations disponibles pour se faire une idée de la validité des démarches utilisées.

Depuis 2011, un tournoi permanent [Schreiber, 2014] est également organisé en ligne et permet actuellement aux joueurs de s'affronter sur plus de 150 jeux. Le système *Agon* est utilisé pour évaluer la difficulté de chaque rôle de chaque jeu et établir un classement entre les différents programmes joueurs à partir des données collectées pour chaque match. Ce système est dérivé du classement *Elo* utilisé aux échecs et modifié pour pouvoir évaluer également les jeux à un ou plus de deux joueurs, les jeux asymétriques, et la compétence généraliste des joueurs i.e. leur capacité à jouer correctement à des jeux très divers. Ce tournoi permet une évaluation plus fiable des joueurs sur un grand nombre de matchs. Depuis 2015, à l'initiative d'Alex Landau, la compétition *Tiltyard Open* est organisée en décembre sur les serveurs de Tiltyard [Schreiber, 2014]. Un autre tournoi permanent est disponible sur les serveurs de l'université de Dresde [Günther et Schiffel, 2013].

Lors d'un match en compétition, une machine-arbitre, le *Game Manager*, délivre aux machines des joueurs les règles en GDL d'un jeu théoriquement inconnu [14] ; elle indique à chaque programme quel rôle il doit tenir dans le jeu. Chaque programme joueur dispose d'un temps limité (*startclock*) pour ana-

13. Divers matchs amicaux sont disputés en ligne notamment sur les serveurs de Stanford et de Dresde. On peut noter l'organisation de quelques compétitions nationales ou limitées à une université donnée. La *German General Game Playing Competition* (GGGPC) a été organisée pour la première fois en 2009 ; les participants étaient tous des chercheurs d'universités allemandes. En 2011 une compétition ouverte aux équipes internationales s'est tenue pendant la 34ème Conférence Allemande d'intelligence artificielle (KI-11) ; il est à noter que le GDL-II y a été utilisé pour la première fois afin de proposer des jeux à information incomplète et comportant une part de hasard. Les compétitions de GGP de l'université de Dresde sont organisées depuis 2006 pour leurs étudiants. En 2012, une compétition a été organisée dans le cadre de la 32ème International Conference on Artificial Intelligence (AI'12) dont une part était également consacrée aux jeux en GDL-II.

14. Il peut s'agir aussi bien d'un jeu classique, nouvellement inventé, ou d'une variation d'un jeu connu. Dans tous les cas, les auteurs des programmes ne savent pas de quel jeu il s'agit et la description est offusquée pour rendre son identification difficile.

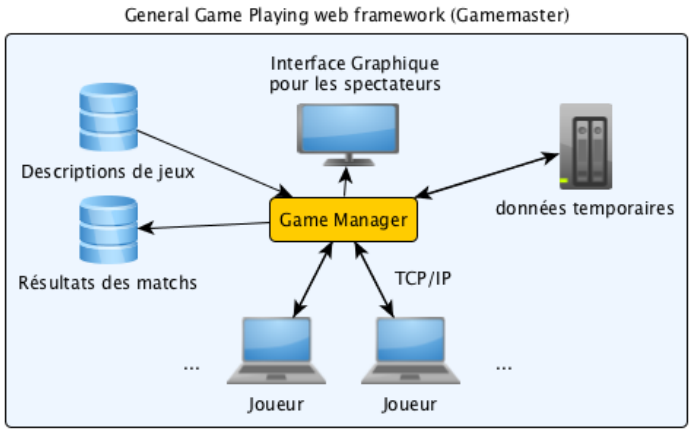


FIGURE 1.3 – Représentation du General Game Playing web framework (Gamemaster)

lyser les règles et effectuer les pré-traitements nécessaires à la mise au point d'une stratégie. À la fin de ce temps de préparation, l'arbitre donne aux programmes un temps limite (*playclock*) pour indiquer le coup qu'ils décident de jouer [15]. À la fin de chaque délai, les différents joueurs doivent indiquer l'action choisie. Tous les joueurs jouent à chaque tour mais, pour les jeux à coups alternés, un seul joueur donne un coup actif, les autres se contentent d'indiquer un coup sans effet. Si un joueur ne transmet pas l'action choisie avant la fin du délai, ou transmet une action illégale, le *Game Manager* joue une action légale au hasard à sa place afin de ne pas interrompre le match et lui permettre de jouer correctement au tour suivant. Pour dissuader les interventions humaines lors des parties en ligne, les règles du jeu sont offusquées de manière à ce qu'aucun mot clef comme *pawn* ou *board* ne puisse servir de guide par son sens.

Le *Game Manager* est au centre du *General Game Playing web framework* (*Gamemaster*) [Love *et al.*, 2006; Genesereth, 2017]. Ce *Game Manager* assure les communications avec les joueurs à travers une connexion HTTP via le protocole TCP/IP. Il maintient différentes bases de données qui permettent de stocker les descriptions de jeux ainsi que les résultats des matchs (description de jeu concernée, joueurs impliqués et leur rôle respectifs, scores obtenus). Il stocke également les données temporaires des matches en cours et fournit une interface graphique pour le suivi des matchs par les spectateurs humains (fig.1.3).

Chaque programme joueur tourne sur une machine connectée au *Gamemaster* via un port désigné pour la compétition. Le protocole GCL (*Game Communication Language*) est utilisé pour l'échange des messages entre le *Game Manager* et le joueur. Voici les différents messages qui peuvent être échangés lors d'un match :

`info` demande si le joueur est prêt (pour vérifier la connexion HTTP). Le joueur peut répondre par

15. Les *startclock* et *playclock* sont variables selon les jeux et varient de quelques secondes à quelques minutes. Plus de temps de réflexion est accordé pour les jeux plus complexes. Voici quelques exemples extraits du site `http://ggp.stanford.edu/` (les temps sont en secondes) :

Jeu	*startclock*	*playclock*
Tictactoe	10	10
Selective Sukoshi	30	10
Joint Connect Four	120	40

jeu	*startclock*	*playclock*
Platform Jumpers	180	40
Breakthrough	600	30
Connect4	600	30

`available` ou `busy` s'il est respectivement prêt ou non.

`start(id,role,description,startclock,playclock)` indique le départ d'un match identifié par un *id* unique, le rôle attribué au joueur, la description GDL complète du jeu ainsi que les délais accordés pour la réflexion. Le joueur peut répondre `ready` s'il est prêt avant la fin du *startclock*. Si tous les joueurs sont prêts, le match commence. Sinon il débute à l'expiration du *startclock*.

`play(id,move)` indique qu'une action du joueur est attendue. *move* est une liste des actions prises en compte pour chaque joueur au tour précédent (ou le mot clef NIL pour le premier coup) [16]. Le joueur répond par un message HTTP contenant juste l'option choisie, par exemple `(mark 2 2)` et non l'action complète (prédicat does avec ses arguments).

`stop(id,move)` joue le meme role que `play(id,move)` mais indique qu'un état terminal est atteint et que le match est terminé. Le joueur répond juste `done`.

`abort(id)` indique l'abandon du match. Le joueur peut alors se réinitialiser ; il répond `done` quand c'est fait.

1.3 *General Game Players*

Le GGP présente un défi d'une nature totalement différente de celui proposé par un joueur programmé spécialisé [Swiechowski et Mandziuk, 2014]. Le joueur programmé GGP ne dispose d'aucun savoir préalable sur le jeu. Il doit acquérir ce savoir par l'exploration ou l'analyse du jeu et ceci le plus rapidement possible puisque le temps est limité. Il doit utiliser ce savoir pour mettre au point une évaluation des positions ; cette évaluation passe parfois par l'utilisation d'heuristiques, le score n'étant connu que dans les états terminaux. Cependant, la diversité des jeux proposés en GGP rend la création d'heuristiques particulièrement complexe.

Depuis 2005, la compétition IGGPC (fig.1.4) a permis de révéler différentes approches intéressantes pour surmonter ces difficultés [Genesereth et Björnsson, 2013]. Les premiers joueurs mettaient l'accent sur l'extraction de connaissances et la création de fonctions d'évaluation, fondées sur des heuristiques, utilisées avec des algorithmes de parcours d'arbre de la famille *Minimax* (*UTexas LARG* [Kuhlmann *et al.*, 2006; Kuhlmann et Stone, 2007], *Ogre* [Kaiser, 2007], MaLigne [Kirci *et al.*, 2011]). Cette approche a assuré la victoire de *ClunePlayer* [Clune, 2007] en 2005 et de *Fluxplayer* [Schiffel et Thielscher, 2007b] en 2006.

A partir de 2007, les algorithmes de type Monte-Carlo remplacent ceux de la famille *Minimax* et s'imposent en assurant un avantage significatif aux joueurs qui les utilisent. *Cadiaplayer* [Finnsson, 2007, 2012; Finnsson et Björnsson, 2008, 2009, 2011] remporte la compétition en 2007 et 2008. Ce joueur utilise UCT, auquel se sont ajoutées successivement plusieurs améliorations : *Move-Average Sampling Technique* (MAST), *Rapid Action Value Estimation* (RAVE) et *Features-to-Action Sampling* (FAST). *Ary* [Méhat et Cazenave, 2008, 2010; Cazenave et Méhat, 2010] remporte la compétition les deux années suivantes en associant UCT avec la technique des tables de transposition et en utilisant l'algorithme *Nested*

16. Cette liste permet au joueur de connaitre l'action jouée par le ou les autres joueurs, mais aussi de constater si l'action enregistrée par le Game Manager est différente de celle qu'il a envoyée, ce qui l'informe d'un problème.

Année	Programme joueur	Auteur(s)	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 Thor Gudmundsson	Reykjavík University
2009	Ary	Jean Méhat	Université Paris 8
2010	Ary	Jean Méhat	Université Paris 8
2011	TurboTurtle	Sam Schreiber	indépendant
2012	Cadiaplayer	Yngvi Björnsson, Hilmar Finnsson	Reykjavík University
2013	TurboTurtle	Sam Schreiber	indépendant
2014	Sancho	Steve Draper, Andrew Rose	indépendant
2015	Galvanise	Richard Emslie	indépendant
2016	WoodStock	Eric Piette	Université d'Artois
2017	*pas de compétition*		

Figure 1.4 – Gagnants de la compétition de IGGPC depuis sa création

Monte-Carlo pour les jeux à un seul joueur [17].

La fiabilité de l'évaluation obtenue par les algorithmes de la famille MCTS augmente avec le nombre de simulations (*playouts*) effectuées. Cependant, le nombre de *playouts* effectués par seconde est limité pour les joueurs utilisant Prolog pour interpréter les règles des jeux. En 2011, *TurboTurttle* apporte une amélioration significative à UCT en utilisant un réseau de propositions (*propnet* [Cox *et al.*, 2009]) pour accélérer la simulation des parties pendant l'exploration d'arbre [18]. Les joueurs Sancho [Draper et Rose, 2015] et Galvanise [19] utilisent également cette amélioration. Sancho effectue différentes optimisations du *propnet* (factorisation et découpage pour évaluer les `goal` séparément) et utilise une version parallélisée de UCT (au niveau des feuilles) avec des heuristiques pour guider la recherche. Sancho utilise également le *propnet* pour détecter les jeux factorisables et identifier les latches (fluents qui ne changent qu'une fois de valeur durant tout le jeu).

En 2016, *Woodstock* [Koriche *et al.*, 2017a] remporte la compétition grâce à un nouvel algorithme MAC-UCB [Koriche *et al.*, 2016] capable d'utiliser la programmation par contraintes stochastiques (SCSP) à la place d'un *propnet* pour déterminer ses actions. Cette approche permet de détecter des symétries dans les jeux pour réduire l'espace de recherche.

Année	Programme joueur	Auteur(s)	Institution
2015	LeJoueur	Jean-Noël Vittaut	Université Paris 8
2016	NeuralNed [20]	Ed Holland	indépendant

Figure 1.5 – Gagnants de la compétition *Tiltyard Open* depuis sa création

17. Ary a également remporté la compétition *German Open in General Game Playing* organisée en marge de la conférence KI-11.
18. *TurboTurtle* a également rempoté la *AI'12 General Game Playing Competition*.
19. Ce projet n'est l'objet d'aucune publication. Le code source a été déposé sur bitbucket en 2016 et supprimé en 2017.

La première édition de la compétition en ligne Tiltyard Open en 2015 (fig.1.5) est remportée par *LeJoueur* [Vittaut, 2017] qui utilise un circuit logique dérivé du *propnet* : le *propnet* est factorisé, simplifié et transformé en circuit n'utilisant que des portes à 1 ou 2 entrées, évaluées très rapidement par des opérateurs binaires. L'utilisation d'opérateurs binaires permet de modifier en parallèle les 64bits des variables représentant la valeur de sortie de chaque porte logique et ainsi de réaliser 64 *playouts* en un. *LeJoueur* utilise ce circuit avec une version parallélisée d'UCT (au niveau des feuilles) associée à un élagage *alpha beta*.

Tous ces joueurs associent un *reasoner*, partie du joueur interprétant les règles avec un *player* permettant d'évaluer les positions. Pour la plupart, les meilleurs joueurs actuels utilisent un *reasoner* de type *propnet* auquel ils apportent diverses optimisations. La programmation par contrainte n'est pour le moment utilisée que par *Woodstock*. Les meilleurs joueurs utilisent un *player* associant les algorithmes de la famille Monte-Carlo avec différentes heuristiques guidant la recherche [21]. Ces algorithmes peuvent être parallélisés pour augmenter le nombre de *playouts* effectués par seconde. Enfin, certains joueurs utilisent la détection de symétries ou la factorisation (sous-jeux identiques) pour diminuer significativement le nombre de positions à explorer dans les jeux. Cependant, aucun de ces joueurs n'implémente une méthode de décomposition générale permettant d'identifier des sous-jeux différents.

20. Ce joueur n'a été l'objet d'aucune publication.

21. Notons que *AlphaGo*, programme spécialisé qui a battu Lee Sedol, un des meilleurs joueurs mondiaux de Go (9e dan professionnel) en mars 2016 et le champion du monde Ke Jie en mai 2017, associe lui aussi MCTS avec une fonction heuristique d'évaluation. Cette fonction est produite par un réseau de neurones profond entrainé au préalable avec des parties d'experts.

Chapitre 2
Interprétation des règles

Sommaire

L'interprétation des règles est réalisée par le *reasoner* dont les fonctions sont de déterminer quelle sera la position du jeu après l'application d'un coup donné, déterminer si cette nouvelle position est terminale, si oui quel est le score de chaque joueur, sinon établir la liste des coups légaux pour chacun d'eux. Plus ce *reasonner* est rapide et robuste, plus l'exploration de l'arbre est rapide et une meilleure évaluation des positions du jeu est obtenue dans le temps imparti en compétition. Une partie de la recherche sur le GGP s'est donc intéressée à la mise au point de *reasonner* efficaces. On peut noter par exemple des approches tentant une compilation des règles pour gagner en vitesse d'interprétation [Cazenave et Saffidine, 2011; Kowalski et Szykuła, 2013] ou bien des interprètes spécialisés comme JavaProver ou C++Reasoner. Cependant Björnsson et Schiffel [2013]; Schiffel et Björnsson [2014] indiquent que les performances de ces interprètes sont bien moindres que celles des interprètes fondés sur Prolog.

Après une présentation des interprètes fondés sur Prolog, nous présenterons le principe du *grounding* et du *tabling* et comment l'instanciation des règles permet de créer un *propnet* voire un circuit logique surpassant nettement les performances de Prolog.

2.1 Interprètes de type Prolog

Jusqu'en 2011, les différents champions de l'IGGPC utilisaient des *reasonners* employant Prolog (*Fluxplayer*, *Cadiaplayer*, *Ary*)[22]. Afin d'être interprétées, les règles GDL sont converties en langage Prolog.

Cette conversion nécessite différentes transformations pour s'assurer que les règles seront correctement interprétées par Prolog. L'ordre des clauses et des prémisses dans le corps des clauses n'est pas signifiant en GDL alors qu'il l'est en Prolog. Dans le cas d'une règle GDL récursive du type :

```
(succ 1 2) (succ 2 3) (succ 3 4)
(<= (less ?x ?y) (succ ?x ?y))
(<= (less ?x ?z) (less ?x ?y) (succ ?y ?z))
```

il convient de réordonner les prémisses de la troisième clause lors de sa traduction en Prolog pour éviter que ce dernier ne boucle indéfiniment en tentant d'unifier le terme `(less ?x ?y)`. Dans le cas des prédicats `not` et `distinct`, les prémisses correspondants doivent être réordonnés de sorte que les variables présentes dans leurs arguments soient unifiées par Prolog avant leur évaluation. Dans le cas contraire l'évaluation des clauses pourrait conduire à un résultat erroné [Vittaut, 2017].

Le programme Prolog obtenu, décrivant la logique du jeu, n'inclut pas la description de l'état courant. Celui-ci doit être spécifié et modifié à chaque tour du jeu ou d'une simulation. Deux approches distinctes peuvent être utilisées pour modifier la description de l'état courant du jeu. La première consiste à ajouter ou supprimer des fluents dans la position courante et à indiquer les actions effectuées grâce au mécanisme d'assertion de Prolog. Ce mécanisme d'assertion de Prolog est assez coûteux. La seconde, fondée sur l'utilisation de listes, vise à réduire ce coût et consiste à enrichir les prédicats d'arguments sous forme de listes indiquant l'état du jeu. Cependant, les inférences nécessitant l'examen de faits présents dans ces listes peuvent être plus lentes qu'avec des assertions [Schiffel et Björnsson, 2014].

Les principes d'unification, de récursivité et de retour sur trace (*backtracking*) sont assez lents et limitent l'exploration des arbres de jeux surtout pour les jeux dont les règles sont complexes et comportent beaucoup de variables. La réalisation de cette unification une fois pour toute en début de jeu (*grounding*) est apparue comme une solution pour accélérer significativement l'interprétation des règles.

2.2 *Grounding* et *Tabling*

L'instanciation ou *grounding* consiste à transformer chaque clause contenant des variables en un ensemble de clauses ne contenant que des constantes. Kissmann et Edelkamp [2010] proposent de réaliser cette instanciation via des graphes de dépendances pour accélérer le fonctionnement des *reasonners* utilisant Prolog.

Voici par exemple une règle extraite du jeu de *Tictactoe*, indiquant qu'il est légal pour un joueur w de mettre une marque dans une case de coordonnées (x, y) si c'est à son tour de jouer et si la case est

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

initialement vide (*b*) :

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

La règle règle traduite en Prolog se présente ainsi :

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

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

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

Vittaut et Méhat [2014] décrivent une technique utilisant Yap Prolog et le *tabling* pour instancier efficacement et rapidement les règles GDL : les règles GDL originales subissent un ensemble de prétraitements visant à les simplifier et à préparer leur instanciation. Les disjonctions explicites (normalement absentes du GDL) sont éliminées, les prémisses des clauses sont ré-ordonnées pour garantir une interprétation correcte des règles par Yap Prolog. Les prédicats statiques (ne dépendant pas de l'état du jeu) et dynamiques (les autres) sont identifiés et les règles sont réécrites de manière à constituer un programme logique dont chaque clause a pour effet de bord la génération de toutes les instances possibles de la clause. La transformation des règles en programme d'instanciation passe par un ensemble de réécritures élémentaires décrites en détail dans la thèse de Vittaut [2017]. Le programme permet d'obtenir toutes les instances des prédicats `input` et `base`, ou de les recalculer à partir des prédicats `true`/`init` et `does`/`legal`, et les utilise ensuite pour instancier toutes les règles.

Certains interprètes comme BProlog, XSB ou Yap proposent un système de mémorisation nommé *tabling*. Ce système permet de stocker les clauses unifiées pour ne pas avoir à rechercher chaque fois les solutions possibles en examinant chaque clause. Le *tabling* est utilisé ici pour regrouper toutes les instances possibles des clauses et accélère considérablement l'instanciation. L'utilisation du *tabling* permet également d'éviter les cas ou l'interprétation d'une règle récursive provoquerait une boucle infinie.

Schiffel [2016] propose une technique d'instanciation fondée sur ASP. Les règles GDL sont transformées en un programme ASP. Un générateur d'actions et un générateur de positions sont implémentés [Haufe *et al.*, 2012] (cf. §9.1.3) et ajoutés au programme. Enfin des règles permettant d'obtenir toutes les instances des prédicats `input` et `base` sont ajoutées. Le programme est optimisé par la suppression des variables existentielles (cf. §9.6) et par l'ajout de fluents pré-instanciés limitant le domaine de certaines variables pour éviter la génération de règles instanciées inutiles. Le programme d'instanciation du solveur ASP Potassco, particulièrement optimisé, est utilisé pour produire l'instanciation du programme ASP de laquelle est extraite l'instanciation des règles GDL. Schiffel ne compare pas son approche avec celle de Vittaut. Cependant il est possible de comparer les graphiques présentant les temps d'instanciation pour un important panel de jeu proposés par chacun d'eux. Il faut noter que Vittaut n'utilise pas

l'optimisation consistant à supprimer les variables existentielles dans la version de *LeJoueur* de 2014 alors que cette optimisation diminue drastiquement le nombre de règles instanciées et par conséquent le temps d'instanciation. Pourtant les résultats annoncés pour les deux approches sont très similaires, ce qui laisse penser que le programme d'instanciation utilisant ASP est moins performant que celui de Vittaut.

Suite au *grounding*, les règles GDL donnent naissance à de nombreuses expressions logiques. Le nombre des règles instanciées est important, dans un rapport exponentiel par rapport au nombre de règles originales. Pour des jeux complexes, le temps et la mémoire nécessaires à l'instanciation des règles peuvent être conséquents et dépasser le temps alloué dans le cadre des compétitions ou la mémoire disponible. Comme dans le cadre du GGP toutes sortes de jeux peuvent être proposées, le *grounding* des règles d'un jeu peut ne pas être réalisable dans un temps raisonnable [Vittaut, 2017].

L'instanciation des règles accélère leur interprétation par Prolog mais apporte également la possibilité d'utiliser une nouvelle approche : le réseau de propositions ou *propnet*. Pour bénéficier du gain en performance apporté par un *propnet* tout en s'assurant de pouvoir jouer même dans le cas ou l'instanciation des règles échoue, la version de *LeJoueur* victorieuse lors de la compétition Tiltyard Open de 2015, implémente un joueur mixte capable d'utiliser l'interprète Yap Prolog pour raisonner sur le jeu en cas d'échec du *grounding*.

La dernière version de *LeJoueur* [Vittaut, 2017] utilise une machine de Waren spécialement dédiée à l'instanciation des règles ainsi de diverses optimisations. Cependant, l'élimination des variables n'est toujours pas implémentée.

2.3 *Propositional Networks* (*propnet*)

En 2011, le concept de *propnet* [23] a apporté une amélioration conséquente de la vitesse d'interprétation des règles et de ce fait une augmentation significative du nombre de simulations de parties réalisables par seconde par les joueurs MCTS. Les *propnets* sont issus de l'Automate Propositionnel, un framework inventé par le *Stanford Logic Group* pour les systèmes multi-agents discrets et dynamiques tels que les jeux [Schkufza *et al.*, 2008].

A partir des règles instanciées, un *propnet* est construit sous la forme d'un graphe. Chaque fait nommé *fluent* qui peut être vrai ou faux à un instant donné du jeu constitue un sommet dans le graphe. À ces sommets s'ajoutent des connecteurs `ET`, `OU`, `NON` qui permettent d'incarner la logique du jeu et un connecteur de transition simulant la dynamique du jeu : le connecteur de transition recopie son entrée sur sa sortie à chaque itération et permet ainsi de simuler le passage d'une position du jeu à une autre. Le *Stanford Logic Group* [Genesereth, 2017] indique qu'un ensemble N de faits décrivant un jeu peut permettre de décrire 2^N positions différentes, ce qui permet au *propnet* de représenter la dynamique d'un jeu d'une manière plus économique qu'un automate à états.

La figure 2.1 représente un *propnet* créé d'après les règles suivantes :

```
(role r)
(base s)
(input r a)
```

23. Littéralement *réseau logique de propositions*.

```
(input r b)
(legal r a)
(<= (legal r b) (true c))
(<= (next c) (does r a))
(<= d (does r a))
(<= d (true f))
(<= (next e) d)
(<= (next f) (does r b) e)
(<= (goal r 100) (not (true e)))
(<= (goal r 0) (true e))
(<= terminal (true f))
```

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

Au début de la partie, la valeur des faits produits par le prédicat `base` est initialisée d'après les informations données dans la description GDL du jeu. Les actions toujours légales, peuvent être représentées par un circuit bouclant sur lui-même constitué d'un seul terme et d'une transition : le terme est initialisé à vrai et gardera cette valeur tout au long du jeu grâce au connecteur de transition.

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

Sironi et Winands [2016] proposent différentes optimisations pour diminuer la taille d'un *propnet* appliquées récursivement jusqu'à l'obtention d'une taille minimale : supprimer les termes internes sans signification particulière, éliminer les composants de valeur constante et ceux non reliés aux sorties. Schkufza [2007] et Schkufza *et al.* [2008] indiquent que les *propnets* peuvent être utilisés pour l'identification de structures spécifiques i.e. des propositions indépendantes et des parties disjointes du graphe.

Les *reasoners* utilisant les *propnets* sont beaucoup plus rapides. Depuis 2009, la majorité des vainqueurs de l'IGGPC utilisent des joueurs fondés sur les *propnets*. Schiffel et Björnsson [2014] indiquent que l'instanciation de toutes les règles peut mener à une explosion exponentielle de la taille de la représentation. Dans *LeJoueur*, Jean-Noël Vittaut propose la conversion d'un *propnet* en un *circuit* logique plus rapide car évalué avec des opérateurs binaires et permettant une parallélisation efficace des calculs. Ainsi le coût de l'instanciation est largement compensé par le gain en vitesse de calcul du *reasoner*.

2.4 Du *propnet* au circuit logique : *LeJoueur*

Vittaut [2017] propose la construction d'un graphe de règles similaire à un *propnet* à partir des règles

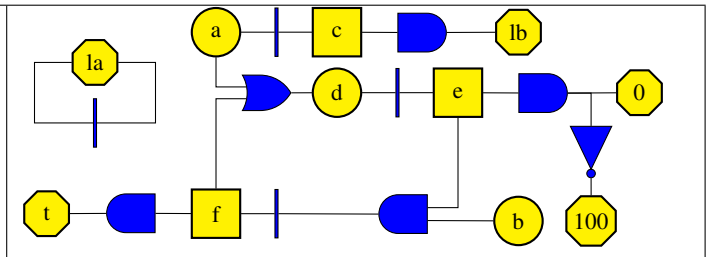
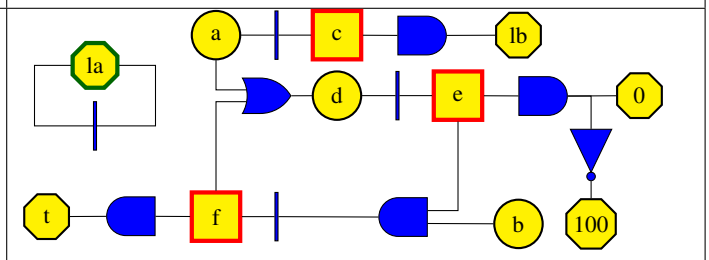
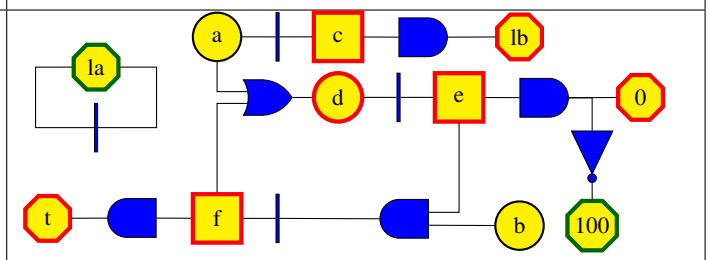
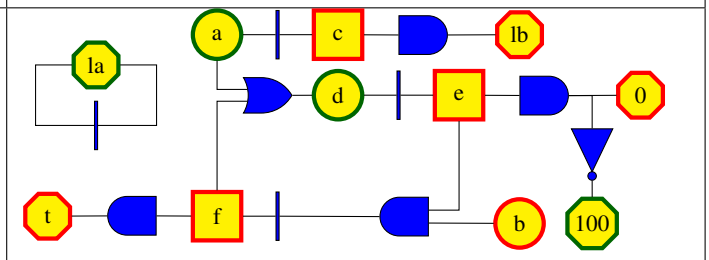
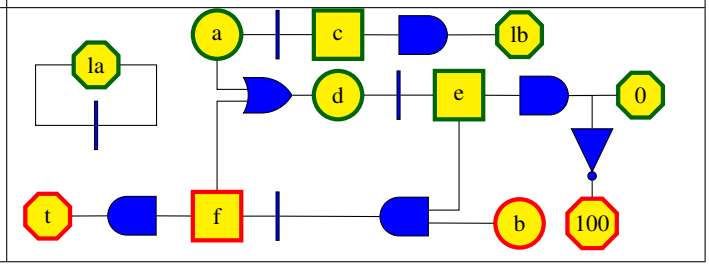
	Exemple de *propnet* : `a` et `b` représentent les actions possibles (`input`) et `la`, `lb` leur légalité. `t` représente le prédicat `terminal`, `0` et `100` les scores. Les fluents `c`, `e` et `f` décrivent la position courante (`base`) et `d` représente un prédicat auxiliaire.	
1	L'état initial du jeu : la valeur des fluents issus du prédicat `base` est initialisée, ainsi que celle de `la` pour indiquer que `a` est toujours légal. Les termes faux sont encadrés en rouge, les termes vrais en vert.	
2	Evaluation de `terminal` et `legal` : le signal est propagé dans le circuit. Si terminal est faux, la partie n'est pas terminée, la valeur de `la` et `lb` indiquent alors quels sont les coups légaux.	
3	Indication du coup joué : la valeur des `inputs` est modifiée pour indiquer le coup joué par chacun des joueurs. Ici l'action `a` est vraie et l'action `b` est fausse. La valeur de `d` est modifiée.	
4	Calcul de l'état suivant : l'activation des transitions provoque la transmission du signal. Les autres faits (`views`) sont déduits de la valeur des fluents issus du prédicat `base` et `input`. L'état est toujours non terminal et l'action `b` devient légale.	

FIGURE 2.1 – Fonctionnement d'un *propnet*. Les carrés, cercles et octogones représentent respectivement les fluents issus des prédicats `base` et `input`, et les *views*.

instanciées. L'ensemble des sommets de ce graphe est constitué des atomes utilisés en conclusion ou en prémisse des règles, des négations d'atomes et des règles. Comme dans un *propnet* des étiquettes `ET`, `OU`, `NON` sont utilisées pour distinguer le rôle des sommets et représenter les relations logiques entre eux. La conclusion d'une règle est étiquetée `OU` et correspond à une disjonction de toutes les clauses de même conclusion. Le corps d'une clause est étiquetée `ET` et correspond à une conjonction de prémisses. Les prémisses négatives sont étiquetées `NON`. Un arc $a \rightarrow b$ est ajouté entre

- une conclusion b et chaque clause a correspondante.
- une clause b et chaque prémisse a de cette clause
- une négation b et l'atome nié a

Les sommets correspondants à des atomes dont les symboles de fonction sont un des mots clefs du GDL (`true`, `does`, `terminal`, `legal`, `next` ou `goal`) sont également étiquetés avec le nom de l'atome auquel ils correspondent.

Le graphe est optimisé par suppression de différents sommets : les sommets redondants, les doubles négations, les sommets ne correspondant pas aux mots clefs du GDL et non reliés aux sorties. Le graphe est également factorisé. Une clef de hachage associée à chaque sommet permet d'identifier les sommets correspondant à une même expression tout en limitant le re-calcul des clefs durant l'optimisation du graphe.

Ce graphe est alors transformé en un circuit logique. Les sommets `ET` et `OU` du graphe sont dédoublés de manière à ne dépendre que de deux prédécesseurs. Les entrées du circuit sont les sommets sans prédécesseurs, les sorties, ceux sans successeurs. Chaque sommet représente le résultat d'une opération logique à effectuer sur ses prédécesseurs. Contrairement à un *propnet*, les opérateurs de transition ne sont pas représentés dans le circuit. Le passage d'un état à un autre se traduit par la recopie des sorties sur les entrées (`next` sur `true` ou `legal` sur `does`).

Les sommets du circuit sont partitionnés de manière à identifier les sous-circuits permettant le calcul des sorties `terminal`, `legal`, `next` et `goal` dans lesquels ils sont impliqués. Un tri topologique permet d'ordonner les sommets en fonction de l'ordre des opérations logiques. Le circuit est alors découpé en strates. Chaque strate est constituée d'un ensemble de sommets ou opérations logiques pouvant être évalués dans un ordre quelconque. L'ensemble est alors transformé en *bytecode* pouvant être évalué rapidement avec des opérateurs binaires.

Cette représentation permet d'effectuer plus de simulations par seconde qu'un *propnet* classique. En utilisant une variable de 64bits pour chaque sommet il est même possible de stocker une valeur différente dans chacun des bits et de réaliser 64 *playouts* en parallèle. Le gain de vitesse obtenu pour les simulations est encore plus marqué par rapport à un interprète de type Prolog [24].

24. Par exemple, pour le jeu de *Tictactoe*, la version de mai 2014 de *LeJoueur* pouvait réaliser 313216 *playouts* par seconde (4894 simulations par seconde en codant chaque fait logique sur 64 bits ce qui permet de faire 4894×64 *playouts* distincts en parallèle par seconde) sur un Core I7 à 2,7GHz avec un *circuit* contre 756 seulement avec YAP Prolog.

Chapitre 3
Exploration de l'arbre d'un jeu

Sommaire

Un jeu peut être représenté sous la forme d'un arbre [25] dont chaque nœud représente un état de la partie à un moment donné et chaque arc représente un coup légal, i.e. un coup autorisé par les règles du jeu. La racine représente l'état initial du jeu et chaque feuille représente un état terminal où le score final peut enfin être connu.

À chaque tour d'un jeu, le joueur doit choisir le coup qui lui sera le plus favorable pour obtenir le meilleur score possible. Pour évaluer chaque état du jeu, diverses techniques ont été mises au point, à commencer par *Minimax* et *Alpha-Beta*. Ces méthodes étaient utilisées par les premiers joueurs programmés pour le GGP. Kuhlmann, Dresner et Stone et Clune utilisaient *Minimax* et *Alpha-Beta* pour leur joueurs respectifs *UTexas LARG* [Kuhlmann *et al.*, 2006] et *Clune Player* [Clune, 2007] combinés à l'utilisation de tables de transpositions. Ces tables sont des tables de hachage permettant de retrouver les informations d'une position même si celle-ci est le résultat de différentes séquences d'actions i.e. si elle est atteinte par différents chemins dans l'arbre du jeu. Réutiliser ces informations permet d'accélérer la recherche. La clef permettant de retrouver les informations d'une position est une clef de hachage

25. Deux visions sont possibles, l'une sous forme d'un arbre incluant des sous-arbres dont les racines représentent la même position et partagent la même information, l'autre sous forme d'un graphe acyclique [26] orienté, possédant une racine unique, où l'unicité des positions est garantie.

26. Partant des définitions de *terminaison* (def.1.1), *gagnabilité* (def.1.3) et *jouabilité* (def.1.2), un jeu bien formé est sans cycles.

à la Zobrist [27]. *FluxPlayer* [Schiffel et Thielscher, 2007a,b] utilise également les techniques classiques d'élagage *Alpha-Beta* et des tables de transposition avec une recherche itérative en profondeur d'abord.

Depuis 2007, les méthodes de Monte-Carlo sont appliquées avec succès à l'exploration d'arbre. Tous les gagnants actuels de la compétitions utilisent des algorithmes de cette famille que nous allons présenter maintenant, ainsi que quelques améliorations utilisées par ces joueurs de GGP.

3.1 Méthodes de Monte-Carlo

Le principe des méthodes de Monte-Carlo repose sur la répétition d'expériences, qui sont des simulations de parties appelées *playouts* [28] dans le cadre des jeux. Pour une position à évaluer, on établit la liste des coups légaux pour lesquels des *playouts* sont réalisés. Pour chaque *playout* le score obtenu est associé au coup légal choisi au départ de cette position. À la fin du temps imparti pour l'ensemble des simulations, le score moyen permet de choisir le meilleur coup.

Abramson et Korf [1987] sont les premiers à proposer une fonction heuristique d'évaluation des positions utilisant une exploration au hasard. Brügmann [1993] propose d'utiliser les méthodes de Monte-Carlo appliquées en physique pour évaluer les actions au jeu de Go et implémente cette idée dans son programme *Gobble*. Cependant, cette exploration basique peut facilement amener à un mauvais choix. Par exemple, si une branche de l'arbre du jeu mène à un ensemble de score moyens (50/100) tandis que l'autre branche amène à un seul score optimal (100/100) et une grande quantité de défaites (0/100), la moyenne obtenue sur la première branche sera plus élevée et le coup choisi annihilera tout espoir d'aboutir au score optimal.

3.1.1 Méthodes MCTS

Pour remédier à ce problème, les méthodes *Monte-Carlo Tree Search (MCTS)* proposent une manière plus efficace d'explorer l'arbre du jeu. Un arbre est construit localement de manière incrémentale et asymétrique. À chaque itération, une phase de *sélection* consiste à choisir un chemin dans l'arbre i.e. une succession de nœuds parmi ceux déjà explorés selon une stratégie donnée (*politique de l'arbre*). La phase de sélection s'achève quand une position dont toutes les actions légales n'ont pas été testées est atteinte. Une action non testée est alors choisie et un nouveau nœud correspondant à l'état suivant du jeu est ajouté : c'est la phase d'*expansion*. Une *simulation* est effectuée à partir de ce nouveau nœud. Le score obtenu est utilisé pour mettre à jour l'évaluation des nœuds : c'est la phase de *rétro-propagation* (fig.3.1). Cette évaluation consiste à mémoriser dans chaque nœud visité pendant la descente (*sélection* et *expansion*) un cumul des scores associé à un nombre de visites.

Les méthodes MCTS constituent une famille d'algorithmes permettent d'obtenir une évaluation des nœuds non terminaux de l'arbre. [Browne *et al.*, 2012] en propose un état de l'art. Lors de la phase de sélection, la politique de l'arbre permet de faire un choix entre *exploration* de nouveaux états et *exploitation* des états précédemment explorés.

27. Le hachage à la Zobrist permet de minimiser la taille des clefs tout en minimisant le risque de collision i.e. réduire le risque d'associer deux positions différentes à un même identifiant.
28. Les *playouts* sont parfois nommés *rollout*.

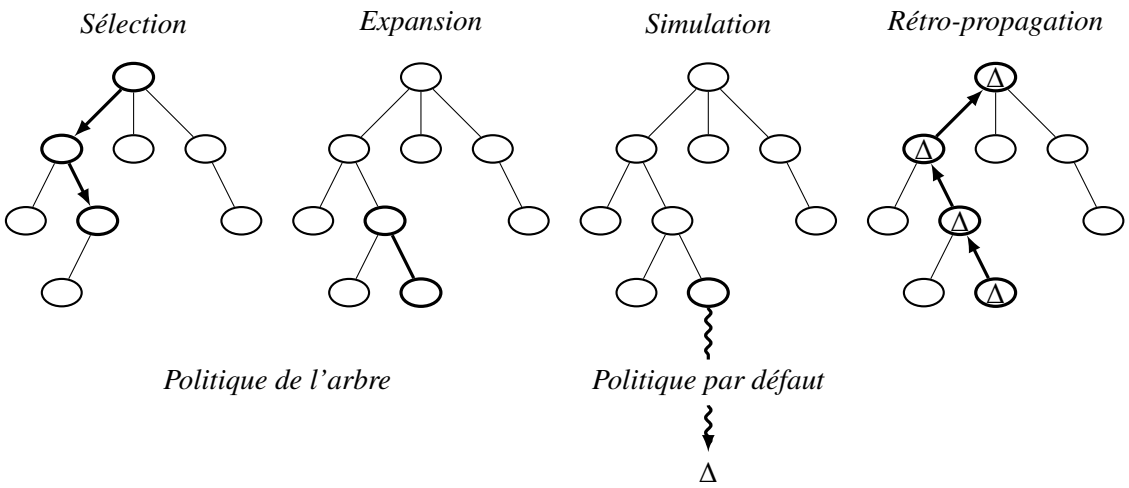


FIGURE 3.1 – Principe des méthodes MCTS [Browne *et al.*, 2012]. La descente regroupant sélection et expansion est définie par une *politique de l'arbre*. Les simulations sont définies par une *politique par défaut*.

3.1.2 Algorithme UCT

L'algorithme *Upper Confidence bound for Tree* (UCT) [Kocsis et Szepesvári, 2006] est le plus populaire parmi les méthodes MCTS [29]. Le principe d'UCT est de choisir, à chaque étape de la sélection, un nœud i qui maximise la valeur de U_i :

$$U_i = \frac{W_i}{N_i} + C\sqrt{\frac{\log N}{N_i}} \tag{3.1}$$

$\frac{W_i}{N_i}$ représente le gain moyen obtenu pour ce nœud : la somme des scores obtenus W_i sur le nombre de visites N_i. La constante C permet de pondérer les deux aspects *exploration* et *exploitation* ; N est le nombre de visites du nœud parent et N_i le nombre de visites pour ce nœud i.

On peut noter que si $N_i = 0$ alors la valeur UCT est $+\infty$ ce qui garantit que pour chaque nœud parent, les nœuds enfants non explorés seront visités au moins une fois avant que le choix d'exploiter le meilleur ne survienne. L'autre élément notable, est la nécessité de choisir une valeur pour C, choix qui va déterminer le poids des deux aspects *exploration* et *exploitation* de l'algorithme.

Dans le cas d'un joueur programmé exploitant les transpositions, les cumuls des scores W_i et des visites N_i sont stockés dans chaque arc correspondant à l'action effectuée pour atteindre un nœud enfant et non dans le nœud enfant lui même. Le nombre de visites N est quant à lui stocké dans le nœud parent à partir duquel est effectuée l'action.

Concernant la *politique par défaut*, la version de *CadiaPlayer* de 2011-2012 [Finnsson et Björnsson, 2011; Finnsson, 2012] utilise la distribution de Gibbs (ou Boltzmann) pour choisir les nœuds pendant les *playouts* :

$$P(a) = \frac{\mathbf{e}^{Qh(a)/\tau}}{\sum_{b=1}^{n} \mathbf{e}^{Qh(b)/\tau}} \tag{3.2}$$

29. Notons que de nombreux articles sous entendent UCT quand ils évoquent MCTS.

avec $P(a)$ la probabilité qu'une action a de désirabilité $Qh(a)$ soit choisie. Le paramètre τ permet de doser l'influence de $Qh(a)$: quand τ tant vers 0 la probabilité favorise un nombre plus restreint de coups, et si τ augmente $P(a)$ tend vers une distribution uniforme. La désirabilité d'un coup est évaluée en fonction d'heuristiques dont nous parlons plus en détail au paragraphe 3.4.

3.2 Alternatives aux méthodes MCTS

Différentes alternatives à MCTS ont été proposées pour résoudre le problème de l'équilibre entre *exploration* et *exploitation* dans certains problèmes.

L'algorithme *Nested Monte-Carlo Search* (NMCS) [Cazenave, 2009], utilisé pour les jeux à un seul joueur, consiste à effectuer une recherche à boucles (for) imbriquées à profondeur N. À partir d'un nœud de départ, chaque action légale est testée tour à tour. À partir de chaque nœud enfant atteint, une recherche à boucles imbriquées de profondeur $N-1$ est effectuée. L'action légale associée au meilleur score est jouée et une recherche à boucles imbriquées à profondeur N est relancée à partir du nœud enfant. La descente s'arrête lorsqu'une feuille i.e. une position terminale est atteinte. À profondeur 1, à chaque itération, un *playout* est lancé à partir de chaque nœud enfant. Le chemin ayant mené au meilleur score est mémorisé pour chacune de ces recherches. À chaque itération, la suite du meilleur chemin est éventuellement remplacée par un enchaînement de coups encore meilleurs. Cette approche apporte un bon équilibre entre exploration et exploitation [Méhat et Cazenave, 2010] et présente l'avantage de ne nécessiter aucun ajustement de paramètre contrairement à UCT. Les performances de UCT et NMC se surpassent l'une l'autre en fonction des jeux, si bien que Mehat et Cazenave ont utilisé pour leur joueur une fonction d'évaluation mêlant les deux approches dans le cas des jeux à un seul joueur : à chaque position, le coup joué est celui recevant la meilleure évaluation combinée des deux algorithmes. Leur joueur *Ary* a gagné la compétition de GGP en 2010 mais, cette compétition n'ayant pas proposé de jeux pour un seul joueur, l'efficacité de ces améliorations n'a pu être mise à l'épreuve.

L'algorithme *Nested Rollout Policy Adaptation* (NRPA) [Rosin, 2011] propose une extension de NMCS qui consiste à utiliser une politique P associant une probabilité à chaque action. Pour une recherche à profondeur $N \geq 1$, tous les enfants ne sont pas l'objet d'une recherche imbriquée. À la place, un enfant est choisi en fonction de la politique P et une recherche NRPA à profondeur $N-1$ est exécutée. Cette étape est itérée M fois pour augmenter les chances de bonne orientation des futurs *playouts*. À chaque itération, les probabilités des actions de la meilleure séquence sont renforcées et le meilleur chemin est éventuellement mis à jour. La politique à niveau N est adaptée en partant d'une politique uniformément aléatoire. Elle est transmise au niveau $N-1$, mais la recherche à niveau $N-1$ ne retourne que la meilleure séquence d'action et le score associé. Une recherche à profondeur 0 consiste à effectuer un *playout*. Pour choisir l'action au départ de ce *playout*, la politique P remplace la *politique par défaut* i.e. l'action n'est pas choisie selon une probabilité uniforme mais en fonction de son évaluation.

Dans la recherche d'un meilleur compromis entre exploration et exploitation, le *Sequencial Halving* [Karnin *et al.*, 2013] présente l'intérêt de ne pas avoir de constante à ajuster pour équilibrer ces deux aspects.

L'algorithme *Sequential Halving applied to Trees* (SHOT) [Cazenave, 2015b] est une alternative à

UCT qui consiste à allouer un nombre fixe de *playouts*, un budget, pour un ensemble d'actions à évaluer. Une partie du budget est répartie entre les actions à évaluer au début de chaque phase. Une phase consiste à jouer le nombre donné de *playouts* à partir de chaque action et à ne conserver pour la phase suivante que la moitié des actions ayant obtenu le score moyen le plus élevé. L'algorithme prend fin quand il ne reste qu'une seule action : celle-ci est retournée.

3.3 Politiques et heuristiques *online*

L'algorithme UCT permet de guider la recherche vers les branches les plus prometteuses en fonction du score moyen espéré. Cependant cette recherche est globalement *aveugle* et pour les jeux complexes i.e. avec un facteur de branchement important ou une profondeur importante dans l'arbre du jeu, cet algorithme échoue à produire une évaluation fiable des positions [30].

L'utilisation d'heuristiques permet aux joueurs spécialisés d'obtenir une évaluation pour les positions non terminales lors d'une recherche à profondeur limitée. Si ces joueurs spécialisés peuvent utiliser des heuristiques établies à partir de connaissances expertes, pour le GGP il est nécessaire d'utiliser des heuristiques plus générales. Généralement, les joueurs programmés n'utilisent pas ces heuristiques comme unique fonction d'évaluation mais s'en servent pour orienter la recherche vers les branches les plus pertinentes de l'arbre du jeu.

Il est possible de distinguer deux types d'heuristiques *online* et *offline* [Trutman et Schiffel, 2015]. Les heuristiques *online* consistent à collecter de l'information durant les *playouts* pour guider la recherche (MAST, PAST, N-Gram). Elles permettent d'établir une fonction d'évaluation qui est progressivement améliorée tout au long du jeu. D'autres approches (AMAF, RAVE, GRAVE) constituent des améliorations de la politique d'évaluation particulièrement utiles en début de jeu pour compenser le manque de fiabilité de UCT quand peu de *playouts* ont été effectués. Les heuristiques *offline* consistent à acquérir des connaissances de plus haut niveau (plateau, mobilité) pendant la phase préparatoire du jeu (*startclock*) et à utiliser les heuristiques générées pendant le reste du jeu ; nous en détaillerons certaines dans la prochaine section.

Les heuristiques utilisées pour le GGP peuvent intervenir à plusieurs niveaux dans l'exploration de l'arbre du jeu : pour définir ou compléter la *politique de l'arbre* (par exemple exclure certaines branches qui ne semblent pas devoir aboutir à un score intéressant) ou pour définir la politique par défaut (par exemple en ordonnant les coups légaux).

L'heuristique *Move-Average Sampling Technique* (MAST) [Finnsson et Björnsson, 2008; Tak *et al.*, 2012] consiste, après chaque *playout*, lorsque le score final est remonté à travers les nœuds de l'arbre, à mettre à jour une valeur moyenne globale dans une *lookup table* pour chaque action choisie pendant les phases de *sélection* et de *simulation* (fig.3.1) ; ceci permet d'évaluer une action quel que soit le moment de la partie où elle est jouée. Pour de nombreux jeux cette stratégie présente un avantage clair (occupation du centre au *Tictactoe* ou des angles à *Othello*). La valeur estimée de chaque action est utilisée pour biaiser le choix des coups pendant les *playouts* selon la distribution de Boltzmann vue précédemment

30. Aujourd'hui la transformation des règles en réseau de propositions (*propnet*) et/ou la parallélisation de leur interprétation permet de repousser les limites de ce qui est calculable dans un temps donné, mais elle ne change pas fondamentalement la complexité de la recherche

(eq. 3.2). Cette approche présente cependant un inconvénient : quand une même action s'avère très bonne ou très mauvaise selon la situation où elle est jouée, cette heuristique diminue la qualité du jeu.

L'heuristique *Tree-Only MAST* (TO-MAST) [Finnsson et Björnsson, 2009; Finnsson et Björnsson, 2010; Finnsson, 2012] est une petite variation de MAST où la mise à jour de l'évaluation des actions est limitée à celles choisies pendant la phase de *sélection* (dans l'arbre déjà exploré), plutôt que dans la totalité de la descente, ce qui permet de baser les décisions sur des données plus robustes.

L'heuristique *Predicate-Average Sampling Technique* (PAST) [Finnsson et Björnsson, 2009; Finnsson, 2012] fonctionne comme MAST à la différence que c'est la valeur moyenne de chaque couple action-prédicat qui est calculée. Selon le principe du GDL, chaque état du jeu est caractérisé par un certain nombre de prédicats qui sont vrais i.e. des fluents. Pendant la phase de rétro-propagation, pour chaque position où l'action a a été choisie, la valeur de chaque fluent vrai est mise à jour. Pendant un *playout*, *CadiaPlayer* utilise la moyenne la plus élevée parmi les fluents pour le calcul de la probabilité dans la formule de Gibbs (eq. 3.2). Cette valeur maximale est utilisée plutôt que la moyenne sur tous les fluents pour limiter le temps de calcul. PAST permet de choisir les coups qui sont bons en fonction d'un certain contexte. Cependant, le bénéfice apporté par une meilleure évaluation n'est pas suffisant comparé au surcoût impliqué par le suivi du contexte et PAST n'est pas vraiment supérieur à MAST.

La méthode N-Gram [Tak *et al.*, 2012] permet d'obtenir une évaluation des actions tenant compte du contexte en évitant le surcoût en calcul précédent. Elle consiste à remonter le score final pour mettre à jour l'évaluation des séquences de 1, 2 et 3 actions effectuées pendant le *playout*.

L'heuristique *All-Moves-As-First* (AMAF) [Brügmann, 1993; Browne *et al.*, 2012], proposée initialement pour le jeu de *Go*, consiste à faire remonter la valeur d'un état final non seulement sur les actions choisies pendant la phase de sélection mais également sur les actions qui peuvent avoir été écartées lors des choix successifs pendant la descente et qui ont été jouées pendant le *playout*. L'idée est qu'il existe des actions qui sont bonnes quelque soit le moment ou elles sont jouées.

L'heuristique *Rapid Action Value Estimation* (RAVE) [Gelly et Silver, 2007, 2011] hérite de AMAF. Pour chaque coup, la valeur estimée par UCT et la valeur RAVE sont toutes les deux calculées séparément. Les valeurs RAVE et UCT sont pondérées de manière à ce que la valeur RAVE soit favorisée pour sélectionner l'action de départ d'un *playout* tant que le nombre de *playouts* est trop faible pour que la valeur UCT soit fiable. À mesure que la valeur UCT se précise, le poids de RAVE est diminué. Une constante est définie pour indiquer le nombre d'évaluations à partir duquel la valeur UCT est considérée suffisamment fiable pour remplacer totalement la valeur RAVE.

La généralisation de RAVE (GRAVE) [Cazenave, 2015a] consiste à calculer la valeur AMAF d'un coup, non pas à partir des informations stockées dans la position courante, mais à partir de celles stockées dans le premier nœud en amont de la position courante à partir duquel un nombre de *playouts* supérieur à un seuil donné a été exécuté. Ceci dans le but de baser le calcul sur des informations plus robustes.

Les premiers joueurs ayant remporté la compétition IGGPC grâce à UCT utilisaient différentes améliorations afin d'en augmenter les performances. Par exemple, les concepteurs de *Cadiaplayer* ont ajouté

successivement, entre 2007 à 2009 les heuristiques MAST, RAVE et FAST [31].

3.4 Heuristiques *offline*

Un des premiers programmes de GGP nommé *Metagamer* [Pell, 1993] capable de jouer à un groupe de jeux de plateau de type échecs a inspiré plusieurs concurrents qui ont fondé leur joueur sur des heuristiques *offline*. Il utilise des heuristiques comme la mobilité (comparaison entre les nombres de coups légaux disponibles pour chaque joueur), le matériel (nombre de pièces dont chaque joueur dispose), la centralité (placer les pièces là ou elles ont la mobilité potentielle maximum i.e. occuper le centre), la promotion (pion devenant reine aux dames) et les menaces (valeur de la pièce menacée et effets à craindre).

Plusieurs travaux antérieurs au GGP ont suggéré des pistes pour l'élaboration d'heuristiques. Samuel [1959, 1967] proposait d'établir manuellement une liste d'heuristiques pour le jeu de *Dames anglaises* (contrôle du centre, mobilité des pièces, etc.) et suggérait de combiner ces heuristiques avec des opérateurs logiques. Les pondérations des heuristiques étaient ajustées en faisant jouer le programme contre lui-même ou en comparant les coups proposés avec ceux d'une partie modèle. L'interaction des heuristiques n'étant pas linéaire, et la combinatoire possible étant importante, des combinaisons d'heuristiques étaient établies à la main à priori.

Marcolino et Matsubara [2011] proposent de recombiner des heuristiques appliquées au jeu de Go en utilisant la programmation agent. Utgoff [2001] évoque l'utilisation de réseaux de neurones ou des algorithmes génétiques pour combiner des caractéristiques décrites par des règles logiques similaires aux règles GDL. Cependant, plusieurs problèmes se posent comme le choix de la topologie des réseaux, le risque de rester bloqué dans un minimum local et la lenteur de l'apprentissage. En outre, les heuristiques sont une fois de plus déterminées à l'avance manuellement.

Différentes phases interviennent dans la génération d'une fonction d'évaluation fondée sur les heuristiques : l'extraction de caractéristiques des règles GDL des jeux, l'utilisation de ces caractéristiques pour générer des heuristiques ce qui inclut le choix des bonnes caractéristiques et leur pondération. La plupart des heuristiques *offline* évaluent les fluents mais nous verrons que certaines proposent une évaluation des actions. Nous décrivons également quelques approches mixtes *online* et *offline* avant de présenter l'usage actuel des heuristiques associé à MCTS.

3.4.1 Recherche de caractéristiques dans les règles

Le joueur *UTexas LARG* [Kuhlmann *et al.*, 2006] utilise une technique de *pattern matching* se reposant sur la syntaxe des règles GDL pour émettre des suppositions sur la signification de certains fluents : présence d'un plateau de jeu, de pièces, relation de successeur, présence se coéquipiers ou adversaires dans les jeux multijoueurs. Ensuite des simulations de parties sont effectuées pour vérifier ou infirmer ces hypothèses. Ces caractéristiques permettent de créer des heuristiques en fonction par exemple de la position, du nombre ou de la distance entre les pièces. Ces heuristiques consistent à voir si la maximisation ou

31. A contrario de ce courant d'utilisation d'heuristiques dans UCT, les concepteurs de *Ary* [Méhat et Cazenave, 2008, 2010] ont utilisé NMCS (cf. §3.2), combiné à l'exploitation des transpositions [Greenblatt *et al.*, 1967; Slate et Atkin, 2012; Breuker, 1998], et ont obtenu la victoire en 2009 à l'IGGPC. Sans diminuer l'intérêt des heuristiques, cet événement souligne l'influence des transpositions dans l'évaluation des positions d'un jeu.

minimisation d'une caractéristique est liée au score obtenu. Les heuristiques collectées dans les premiers 10% du *startclock* sont utilisées pour guider une recherche Alpha-Beta par approfondissement itératif, étendue au jeux multijoueurs associée à une table de transposition et une heuristique historique [32].

UTexas LARG a obtenu la 3ème place au concours de GGP en 2006 ; il n'a jamais gagné la compétition, mais la démarche de Kuhlmann et al. est néanmoins remarquable par son originalité. Kuhlmann et Stone proposent également une autre idée intéressante : mémoriser les jeux déjà rencontrés et les heuristiques utilisées en codant les règles sous forme de graphe pour pouvoir les comparer ultérieurement à d'autres jeux rencontrés dans le but de réutiliser les connaissances déjà acquises [Kuhlmann et Stone, 2007]. Cette idée a été reprise récemment par Zhang *et al.* [2015] qui proposent une approche pour déterminer l'équivalence entre deux jeux en GDL. Cependant, il ne semble pas que le transfert de connaissances entre jeux similaires ait été appliqué en compétition.

Le joueur *OGRE* [Kaiser, 2007] propose une approche statistique pour identifier les caractéristiques. Les arguments des termes GDL sont classés par variance de manière à identifier par exemple des pièces mobiles (grande variance) ou des coordonnées (faible variance). Il extrait différentes caractéristiques : distance d'un élément mobile par rapport à la position initiale et terminale, nombre de pièces total ou par coordonnée (par ligne ou colonne), mobilité, nombre de *steps*, taux d'accomplissement des buts (nombre de fluents présents en prémisse des goals dans la position), valeur de goal, état des compteurs. Les caractéristiques évaluées comme désirables par des simulations contre un joueur jouant aléatoirement sont sélectionnées et leur moyenne constitue une fonction d'évaluation utilisée pour une recherche Alpha-Beta. *Ogre* n'obtiendra que la 4ème place lors de la compétition de GGP de 2006.

FluxPlayer [Schiffel et Thielscher, 2007a,b; Schiffel, 2011] opère lui aussi une détection de caractéristiques (compteurs, relation d'ordre, marques, pièces, plateau de jeu) en analysant la structure sémantique et non syntaxique des prédicats. Il utilise la logique floue [33] pour estimer la proximité de la fin du jeu et le degré de satisfaction des buts. Ainsi, le joueur évite d'atteindre un état terminal tant que le ou les buts permettant de maximiser le score ne sont pas atteints. La maximisation du score est utilisée pour les jeux en coups alternés tandis qu'un comportement paranoïaque est utilisé dans le cas de coups simultanés (on considère que le ou les adversaires jouent le coup le plus défavorable) ce qui n'aboutit pas forcément à un comportement optimal. Malgré tout, *FluxPlayer* a gagné la compétition de GGP de 2006 et a su se placer parmi les 4 premiers les années suivantes.

Michulke et Schiffel [2012] proposent une amélioration de la technique de Schiffel et de sa notion de distance obtenue cette fois grâce à un graphe de dépendances entre les fluents. Cependant, ils indiquent que l'heuristique de distance est inadaptée pour certains jeux comme *Breakthrough suicide*, son amélioration augmente alors la contre-performance du joueur.

3.4.2 Génération automatique de caractéristiques

Clune [2007, 2008] (*Cluneplayer*) propose l'idée que les prédicats auxiliaires utilisés pour décrire un jeu correspondent à des caractéristiques importantes de ce jeu. Il extrait ces expressions des règles en

32. L'heuristique historique consiste à réordonner les nœuds enfants en fonction de leurs valeurs trouvées dans les recherches à profondeur moindre

33. *FluxPlayer* tient son nom du système Prolog Flux développé par Thielscher en 2005, fondé sur le *Fluent Calculus*, qui permet d'analyser les règles GDL

GDL et propose différents traitements pour en déduire des *interprétations*. Ces interprétations peuvent par exemple être la *cardinalité* i.e. dénombrer les fluents correspondant à un modèle (nombre de pièces d'un type donné sur le plateau), ou la *distance* i.e. une relation de succession ou d'adjacence (entre des lignes ou des colonnes du plateau de jeu). Des caractéristiques supplémentaires sont créées en détectant les relations *symétriques*. Le nombre de pièces des joueurs, par exemple, est détecté comme *symétrique* : une caractéristique correspondant à la différence entre les nombres de pièces est alors ajoutée. Clune évalue également le degré de satisfaction des buts et fonde ses heuristiques sur 3 aspects : la *récompense*, le *contrôle* et la *proximité d'un état terminal*. Pour le degré de satisfaction, il détecte les éléments stables et utilise la variance pour établir une prévision de l'évolution du jeu à partir d'une position donnée : les éléments qui varient trop ne sont pas fiables pour évaluer une position. Les éléments stables (dont la valeur évolue régulièrement sans fluctuation) ayant une influence sur l'évolution de la récompense (positive ou négative) sont utilisés pour les heuristiques. Ceux dont la valeur n'est pas liée à celle de la récompense sont jugés insignifiants. Pour le *contrôle*, il sélectionne les éléments qui ont une influence sur le nombre de coups légaux pour réduire la mobilité de l'adversaire et augmenter celle du joueur. La *proximité d'un état terminal* est utilisée pour pondérer les fonctions de *récompense* et de *contrôle*, l'augmentation de la récompense étant d'autant plus recherchée que l'issue du jeu approche. Les heuristiques sont combinées linéairement pour créer une fonction d'évaluation utilisée avec une recherche *Minimax*.

Günther [2008] développe cette idée d'utiliser les prédicats auxiliaires du jeu. Il génère différentes expressions composées de prédicats non instanciés utilisés dans la description GDL. Une caractéristique utilisée pour l'évaluation d'un état du jeu correspond à une liste de N variables associée à une expression. Le nombre d'instanciations possibles de cette liste de variables en accord avec une position du jeu, correspond à l'évaluation de cette caractéristique. Pour générer les caractéristiques, il part de l'ensemble des clauses utilisées dans les règles de conclusion `terminal` ou `goal`. Il propose de générer d'autres caractéristiques à travers une généralisation ou spécialisation en instanciant plus ou moins les variables, en découpant les disjonctions ou en supprimant les négations. Pour limiter le nombre de caractéristiques générées, il propose d'établir un graphe exprimant leur filiation et de n'en sélectionner que certaines, des plus générales aux plus spécifiques, en fonction d'un temps limite accordé pour le calcul de la fonction d'évaluation. Les caractéristiques qui correspondent à la négation ou la disjonction d'autres caractéristiques, qui sont présentes trop rarement ou au contraire trop souvent dans les positions ou qui sont trop lourdes à évaluer, sont écartées. Il utilise enfin l'apprentissage par renforcement pour estimer le poids de chaque caractéristique et créer une fonction d'évaluation.

Waledzik et Mandziuk [2014] génèrent également des caractéristiques à partir des prédicats utilisés dans la description d'un jeu. Comme Günther [2008] et contrairement à Clune [2007], ils n'essayent pas d'imposer des interprétations prédéfinies aux expressions. Leur approche est cependant un peu différente de la précédente. Ils n'utilisent que des termes uniques plus ou moins instanciés extraits de toutes les règles du jeu et proposent de créer des combinaisons limitées à un couple de termes qui varient selon un seul argument, par exemple `(cell ?m ?n x)` et `(cell ?m ?n o)`, associées à une comparaison de leur dénombrement. Le nombre total de caractéristiques est limité par un seuil. L'évaluation et la pondération de chaque caractéristique est établie à travers des simulations en fonction de sa corrélation avec le score et de sa stabilité (calculée différemment de Clune [2007]) i.e. le nombre de fluents, présents d'une position à l'autre et correspondant au modèle, doit avoir une faible variance. La fonction d'évaluation est une

combinaison linéaire des caractéristiques sélectionnées, construite dans un temps compatible avec le *startclock*. Cette fonction est utilisée avec UCT (*Guided UCT*) ou avec l'algorithme MTD(f), membre de la famille des algorithmes *Memory Enhanced Test Driver*, qui est l'un des plus populaires algorithmes fondé sur l'élagage Alpha-Beta.

MaLigne [Kirci *et al.*, 2011], arrivé troisième en 2009 et second en 2010 à l'IGGPC, propose une démarche remarquable par son originalité. Les caractéristiques utilisées pour la génération d'heuristiques sont constituées de groupes de fluents qui, associés à une action, permettent d'obtenir une victoire ou de prévenir une défaite. Ainsi, des caractéristiques agressives ou défensives sont collectées avec l'algorithme *Game Independent Feature Learning* (GIFL) qui consiste à extraire les prédicats vrais d'un état terminal qui ont une influence sur le résultat obtenu en conjonction avec la dernière action effectuée. Le programme de Kirci détermine les faits qui ont une influence en supprimant tour à tour chaque fait et en vérifiant si cette ablation a eu un impact sur le score obtenu. Cependant certaines combinaisons plus complexes de prédicats ne sont pas détectées par cette technique, ce qui en limite l'efficacité pour certains jeux et provoque même dans certains cas une contre-performance.

3.4.3 Heuristiques évaluant les actions

Cadiaplayer utilise les heuristiques *online* MAST [Finnsson et Björnsson, 2008] et RAVE [Finnsson et Björnsson, 2009] pour guider une recherche UCT. Tandis que RAVE guide la descente dans l'arbre, MAST est utilisé pour guider le choix des actions pendant les *playouts*. Dans les versions suivantes du joueur [Finnsson et Björnsson, 2011; Finnsson, 2012], une nouvelle technique, *offline* cette fois, est proposée : la *Features-to-Action Sampling Technique* (FAST). Elle est utilisée conjointement à MAST pour classer les actions pendant les *playouts*. Au début du jeu, des caractéristiques sont détectées dans les règles d'après des modèles (types de pièces, cases d'un plateau de jeu) et sont pondérées grâce au *Temporal Difference learning* pendant le *startclock* mais aussi durant les *playclocks*. L'évaluation des actions apportée par FAST est associée à celles de MAST quand des caractéristiques recherchées sont présentes dans une position, sinon l'évaluation retournée par MAST est utilisée seule.

Une autre approche proposée par Trutman et Schiffel [2015] permet de générer une heuristique estimant l'utilité des actions par l'analyse des règles plutôt que par la simulation. Leur approche est une version, fondée sur les actions, de la fonction d'évaluation des positions proposée par Schiffel et Thielscher [2007a] utilisant la logique floue pour évaluer le degré de satisfaction des conditions de victoires énoncées dans les règles `goal`. Pour cela, il analyse les dépendances entre les fluents et les actions avant le jeu et utilise la logique floue pour évaluer les actions d'après d'une régression des prédicats `goal` décrivant les conditions nécessaires à l'obtention du score maximum. Cette heuristique indique dans quelle proportion une action rapproche le joueur de la victoire.

3.4.4 Approches mixtes

En dehors de l'algorithme FAST qui associe une recherche de caractéristiques *offline* avec une pondération calculée en partie pendant le jeu, nous pouvons noter une autre approche mixte qui améliore le résultat retourné par une fonction d'évaluation *offline* par l'expérience acquise *online*.

Michulke et Thielscher [2009] proposent une fonction d'évaluation fondée sur l'utilisation d'un réseau de neurones. Les règles `goal` sont utilisées pour générer un réseau de neurones dont les entrées sont les fluents présents en prémisse de ces règles et les sorties sont les scores possibles. Les neurones des couches cachées correspondent aux expressions logiques (line, col, row, diagonal) formées à partir des fluents et les connections sont pondérées positivement ou négativement en fonction de la présence ou non de négations. Le réseau est modifié pour obtenir une connexion totale de chaque strate, et un biais est ajouté. Le réseau initial encodant la logique des règles `goal` peut être utilisé sans entraînement préalable. Tout au long du jeu, les pondérations internes du circuit sont ajustées en fonction du résultat des simulations de parties.

Sharma *et al.* [2008a,b, 2009] proposent d'autres approches : une fondée sur la technique de neuro-évolution NEAT permettant de faire évoluer la topologie du réseau tout en ajustant les pondérations, une autre découvrant ces pondérations grâce aux algorithmes de colonies de fourmis. Ces approches sont cependant peu convaincantes : le temps de calcul nécessaire est très important et la fonction d'évaluation générée qui associe les caractéristiques de manière linéaire, n'apporte pas un bon niveau de jeu.

D'autres approches utilisant les algorithmes génétiques souffrent du même problème de lourdeur des calculs. WAR [Kaiser, 2000] réalise l'apprentissage pour sélectionner des heuristiques spécialisées chacune dans un type de jeu avant même le *startclock*. Lancucki [2014] s'affranchit également des contraintes des compétitions en accordant plusieurs heures à son programme pour collecter des informations sur le plateau de jeu.

3.4.5 Usage actuel des heuristiques

Les heuristiques sont nécessaires pour évaluer les positions dans le cas d'une recherche limitée en profondeur. Cependant, même si elles sont préférables à un jeu au hasard, elles ne sont pas efficaces dans tous les jeux. Malgré les tentatives précédentes pour créer un système capable d'apprendre des heuristiques générales indépendantes d'un type de jeu particulier, les heuristiques *offline* souffrent généralement d'un manque de robustesse.

Certaines approches assignent une pondération aux heuristiques, mais ceci est aussi susceptible de conduire à des erreurs. Pour mémoire, dans la partie finale de la seconde compétition en 2006, *Clune-player* avait pondéré fortement la mobilité inverse de l'adversaire et n'avait pas trouvé de meilleur moyen pour limiter cette mobilité que de sacrifier ses propres pièces une à une perdant ainsi la partie [Genesereth et Björnsson, 2013]. Les heuristiques sont liées à des caractéristiques qui ne sont pas forcement présentes dans tous les jeux (plateau de jeu, marques) ou qui n'ont pas le même impact selon les jeux (la distance dans *Breakthrough* ou sa version suicide). Un biais heuristique trop fortement marqué est donc défavorable pour jouer correctement à une variété de jeux suffisamment large.

Depuis 2007, les joueurs utilisent principalement les méthodes de Monte-Carlo et plus précisément UCT qui a l'avantage de donner une évaluation correcte sur la majorité des jeux à condition de pouvoir effectuer une grande quantité de *playouts*. Les heuristiques sont désormais utilisées comme un moyen de guider UCT lorsque trop peu d'information est disponible.

Par exemple Chaslot *et al.* [2007] proposent d'appliquer une heuristique pour guider les *playouts* et pour éliminer certains coups légaux dans la phase de sélection (prunning) quand il n'y a pas assez

d'évaluations. À mesure que plus de *playouts* sont effectués, la valeur UCT est pondérée plus fortement jusqu'à être utilisée seule pour la sélection. Lanctot *et al.* [2014] montrent comment (dans le cadre de jeux particuliers) une fonction d'évaluation fondée sur des heuristiques peut être utilisée avec MCTS pour améliorer les performances d'un joueur en maintenant séparées les valeurs MCTS et une valeur de type min/max de l'heuristique.

Sancho [Draper et Rose, 2015], le vainqueur de l'IGGPC de 2014, utilise un ensemble d'heuristiques en plus d'UCT, d'un *propnet* optimisé (factorisation, sous-*propnet* pour les goals, identification des *latches*) et de structures de données optimisées pour améliorer l'usage de la mémoire. Il détecte les jeux dont chaque position gagnante est créée par une seule action et les jeux où la mobilité des pièces ne dépasse pas un certain seuil, ce qui permet de créer une métrique de distance et de détecter une victoire forcée en N coups sans explosion du temps de calcul. Les actions bonnes dans une position sont favorisées dans les positions similaires. Les heuristiques sont utilisées uniquement pour donner une évaluation initiale aux nœuds de l'arbre à leur création, UCT permettant de d'ajuster progressivement l'évaluation des nœuds et de corriger cette valeur initiale en cas de besoin. Certaines heuristiques (mobilité, nombre de pièces) sont limitées aux simulations effectuées pendant le *startclock*.

Chapitre 4
Conclusion

Dans cette partie nous avons présenté le GGP, un champ de recherche de l'intelligence artificielle qui vise à produire des programmes pouvant jouer à n'importe quel jeu sans intervention humaine. Les règles du jeu décrites en GDL sont découvertes par le joueur au dernier moment ce qui exclue l'utilisation de tout savoir à priori.

Nous avons vu que depuis 2005, les compétitions de GGP permettent d'évaluer les joueurs et les nouvelles approches qu'ils mettent en œuvre. Les premiers joueurs généraux, reprenant les approches utilisées par des joueurs plus spécialisés, utilisaient des heuristiques fondées sur la détection de structures comme un plateau de jeu ou sur des caractéristiques comme la mobilité des pièces pour guider une exploration de type *Minimax* de l'arbre du jeu. Malgré de multiples tentatives pour extraire des heuristiques plus générales et moins axées sur des types de jeux particuliers, ces dernières ce sont avérées trop peu robustes pour être utilisées comme fonction d'évaluation principale.

Depuis 2007 les joueurs utilisent en majorité les algorithmes de la famille Monte-Carlo. Ces algorithmes ont apporté une amélioration significative du niveau de jeu. L'attention des chercheurs s'est alors portée sur l'optimisation de ces techniques. Les interprètes logiques de type Prolog ont été remplacés par une transformation des règles en *propnet* ou circuit logique qui permettent une interprétation des règles bien plus rapide, l'exécution d'un plus grand nombre de *playouts* par secondes et améliore l'évaluation des positions produites par UCT.

Actuellement, les joueurs les plus performants utilisent les heuristiques uniquement pour guider les algorithmes de la famille MCTS en début de jeu, lorsqu'un faible nombre de *playouts* ne permet pas une évaluation fiable.

Le développement de la performance de MCTS et l'exécution d'un plus grand nombre de *playouts* par seconde passe également par la parallélisation : nous présentons cet aspect dans la partie suivante.

Deuxième partie

Parallélisation

Dans cette partie nous dressons un état de l'art des différentes techniques de parallélisation de MCTS décrites dans la littérature et d'une technique de parallélisation proposée pour l'algorithme Nested Monte-Carlo. Nous présentons ensuite notre contribution : une étude des possibilités de parallélisation de MCTS, à travers *LeJoueur* de Jean-Noël Vittaut, sur une nouvelle architecture, le *Multi Purpose Processor Array* de la société *Kalray*.

Chapitre 5
Techniques de parallélisation de MCTS

Sommaire

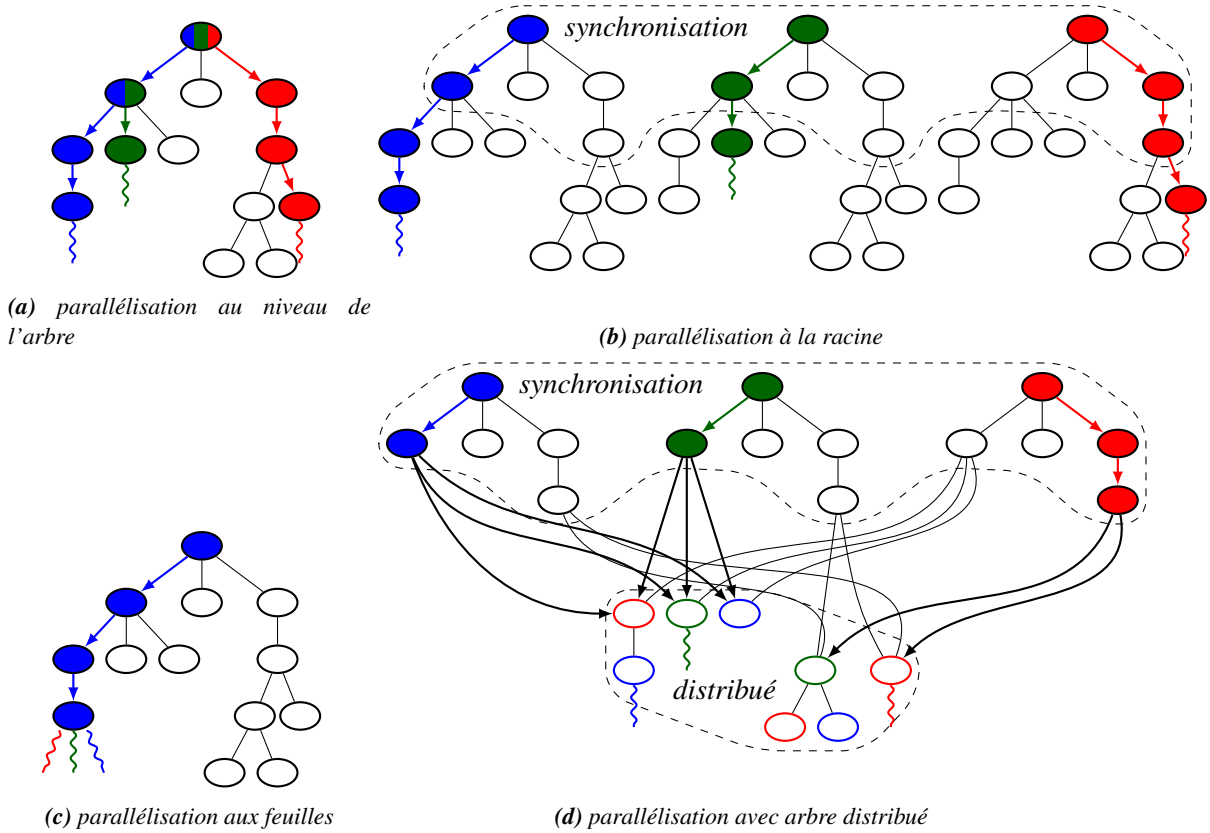
L'évaluation d'une position fournie par une recherche MCTS (ou NMC/SHOT) peut être améliorée en augmentant la taille de l'arbre construit[34], ce qui permet d'explorer une plus grande partie de la branche prometteuse de l'espace de recherche. Elle peut aussi être améliorée en augmentant le nombre de *playouts* pour un même nœud. Kocsis et Szepesvári [2006] indiquent qu'en théorie la probabilité de choisir le meilleur coup avec l'algorithme UCT tend vers 1 quand le nombre de *playouts* tend vers l'infini. Pour augmenter le nombre de simulations lors de l'exploration de l'arbre du jeu, différentes techniques de parallélisation ont été tentées dont certaines permettent aussi la construction d'un arbre plus grand. Nous nous intéressons ici aux parallélisations de MCTS utilisant une politique par défaut avec des *playouts*.

5.1 Au niveau de l'arbre

Pour augmenter le nombre de simulations lors de l'exploration de l'arbre du jeu, Gelly *et al.* [2006] sont les premiers à proposer une parallélisation de MCTS appliquée au jeu de Go (Mogo). Un arbre unique est construit dans une mémoire partagée par plusieurs *threads* réalisant des *playouts* (fig.5.1a). L'accès à l'arbre est protégé par un *mutex* et l'algorithme passe mal à l'échelle au delà de 4 *threads* exécutés sur 4 processeurs distincts.

Pour remédier à cela, Chaslot *et al.* [2008] proposent de réaliser cette *parallélisation de l'arbre* avec des *mutex* locaux, un pour chaque nœud. Ces *mutex* sont fondés sur une attente active (*spinlock*), pour limiter le surcoût dû aux fréquents verrouillages et déverrouillages. Cependant, cette amélioration est au prix d'une augmentation de la taille de la structure de données représentant chaque nœud [Enzenberger

34. La taille de l'arbre construit dépend directement du nombre de *playouts* exécutés dans un temps donné, chaque *playout* donnant lieu à la création d'un nouveau nœud.



(a) *parallélisation au niveau de l'arbre*

(b) *parallélisation à la racine*

(c) *parallélisation aux feuilles*

(d) *parallélisation avec arbre distribué*

FIGURE 5.1 – Aperçu des différentes techniques de parallélisation de MCTS [Graf *et al.*, 2011]

et Müller, 2010].

Pour éviter que les *threads* suivent un chemin déjà parcouru par un autre *thread* lors de la phase de sélection et attendent le déblocage du même *mutex* local, Chaslot *et al.* [2008] et Chaslot [2010] proposent d'utiliser une perte virtuelle (*virtual loss*). Elle consiste à faire comme si le *playout* était perdant pour les différents joueurs et à baisser l'évaluation des nœuds visités pendant la phase de sélection. Cette perte virtuelle est supprimée lorsque l'évaluation est remontée à la fin du *playout*.

Enzenberger et Müller [2010] proposent un algorithme sans *mutex* [35] qui permet de limiter les erreurs provoquées par des accès concurrents à l'arbre du jeu. En limitant la perte de temps due aux *mutex*, Enzenberger et Müller [2010] montrent que, sur le jeu de Go, la recherche effectuée avec N *threads* est équivalente en terme de parties gagnées à la version séquentielle avec N fois plus de temps.

5.2 A la racine

Une méthode de *parallélisation à la racine* [Chaslot *et al.*, 2008; Chaslot, 2010] est proposée par Cazenave et Jouandeau [2007] (fig.5.1b). Elle consiste à développer, avec une politique UCT, N arbres distincts à partir de la racine. À la fin du temps de réflexion, les valeurs des nœuds enfants de la racine calculées par les différents processus sont additionnées par un processus maître et le meilleur nœud est choisi.

35. Ces *mutex* ne sont pas remplacés, l'algorithme n'utilise aucune protection contre les accès concurrents.

En plus de cette procédure *single-run*, Cazenave et Jouandeau [2007] testent une approche *multiple-run* où le temps de réflexion est découpé en plusieurs phases à la fin de chacune desquelles la valeur des nœuds est actualisée. Cette approche implique une augmentation des temps de communication mais apporte un partage des connaissances qui permet de mieux orienter la recherche. Gelly *et al.* [2008] étendent cette technique en synchronisant à intervalles de temps réguliers les évaluations des nœuds enfants de la racine qui ont été suffisamment visités sur k niveaux de profondeur.

Méhat et Cazenave [2011a] proposent l'application de la *parallélisation à la racine* au domaine du GGP. Ils utilisent la parallélisation de type *single-run* proposée par Cazenave et Jouandeau [2007] mais expérimentent différentes approches (*Best*, *Sum*, *Sum10*, *Raw*) pour fusionner les résultats des différentes recherches UCT effectuées par les esclaves. Comparé à une recherche séquentielle effectuée sur un même laps de temps, le gain apporté par la parallélisation sur N esclaves, en terme de niveau de jeu pour les différents jeux testés [36], est positif bien que parfois inférieur ou identique à une recherche séquentielle bénéficiant de N fois plus de temps. Contrairement à cette *parallélisation à la racine* qui ne fonctionne pas pour tous les jeux, la *parallélisation au niveau de l'arbre* améliore les résultats pour tous les jeux testés par rapport à un joueur séquentiel bénéficiant de *N* fois plus de temps [Méhat et Cazenave, 2011b]. Cependant, la performance de cette *parallélisation au niveau de l'arbre* pour le GGP dépend principalement de la capacité du joueur à garder les processus esclaves occupés.

Soejima *et al.* [2010] comparent la stratégie de sélection par moyenne (*Raw*) avec un vote à la majorité : chaque recherche UCT par un esclave permet la sélection d'une action qui constitue un vote, le processus maître collecte alors ces votes et l'action remportant la majorité est choisie. Ils montrent que cette stratégie permet un meilleur niveau de jeu pour les jeux de Go 9×9 et 19×19. Ils remarquent que la *parallélisation au niveau de l'arbre* est considérée plus efficace que la *parallélisation à la racine* grâce au partage de connaissances qu'elle permet et le fait que l'arbre construit est potentiellement plus grand. Cependant la *parallélisation au niveau de l'arbre* n'est pas applicable telle quelle dans le cas d'un système distribué.

5.3 Au niveau des feuilles

Une technique de *parallélisation aux feuilles* (fig.5.1c) est proposée pour la première fois par Cazenave et Jouandeau [2007] pour une architecture avec une mémoire distribuée. L'arbre est construit par le processus maître et seuls les *playouts* sont exécutés par les processus esclaves.

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

36. Les jeux testés sont *Blocker*, *Breakthrough*, *Checkers*, *Connect4*, *Othello*, *Pawn Whopping*, *Pentago* et *Skirmish*.

Pour améliorer la performance de la *parallélisation aux feuilles*, Chaslot *et al.* [2008] tentent d'arrêter les *threads* restants si les résultats des premiers indiquent un coup peu intéressant, ce qui permet de relancer une recherche sur un autre nœud. Cependant cette modification ne permet pas un bon passage à l'échelle, beaucoup de temps étant perdu pour des *playouts* non menés à terme. *Cadiaplayer* [Finnsson, 2012] utilise une *parallélisation aux feuilles* où chaque processus esclave lance plusieurs *playouts* à partir d'une même position pour améliorer l'équilibre entre le poids des communications et les temps de calcul.

L'utilisation de communications asynchrones [Cazenave et Jouandeau, 2008] permet au processus maître de relancer un *playout* sans attendre le résultat de tous les processus esclaves. Cette technique testée sur le jeu de Go permet un bon passage à l'échelle jusqu'à 16 esclaves. Kato et Takeuchi [2010] utilisent la même approche et proposent un algorithme adapté à une architecture de type client-serveur permettant à des clients hétérogènes de se connecter à la volée [37] au serveur pour participer au calcul des simulations.

Malgré ces améliorations, la technique de *parallélisation aux feuilles* reste moins performante que la *parallélisation au niveau de l'arbre* ou *à la racine* car elle souffre de limitations provoquées par le processus maître et par les communications [Soejima *et al.*, 2010]. C'est pourtant cette technique de parallélisation qui est utilisée par Sancho [Draper et Rose, 2015], vainqueur de l'IGGPC en 2014 [38].

5.4 Avec arbre distribué

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

Graf *et al.* [2011] et Schaefers *et al.* [2011] proposent une solution en distribuant le stockage de l'arbre sur différents clusters (fig.5.1d). MCTS peut être associé à l'utilisation d'une table de transposition dans laquelle chaque nœud est stocké en l'associant à une clef de hachage pour y accéder plus rapidement. Ils proposent d'utiliser cette clef de hachage pour déterminer le cluster dans lequel le nœud sera stocké en mémoire.

La descente dans l'arbre jusqu'à la feuille la plus prometteuse se fait par échange de messages entre les clusters. Le cluster devant stocker la feuille, crée ce nouveau nœud et lance un *playout*. Lorsqu'un *playout* retourne une évaluation pour un nœud, un message de mise à jour est envoyé à tous les clusters stockant les nœuds parents.

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

38. *LeJoueur*, vainqueur de la compétition en ligne de Tiltyard en 2015 utilise une *parallélisation aux feuilles* particulièrement efficace qui consiste à réaliser *n playouts* simultanés sur un seul *thread* en utilisant une représentation *bytecode* du jeu [Vittaut, 2017]

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

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

5.5 De Nested Monte-Carlo

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

Un processus *répartiteur* utilise un algorithme *Round-Robin* pour classer les travaux en fonction de leur taille estimée et indique aux processus *médians* les esclaves disponibles. Dans une autre version *LastMinute*, un esclave informe le *répartiteur* quand il n'a plus de travail à faire, le travail le plus long de la file d'attente lui est alors donné, sinon, si aucune tâche n'est en attente, l'identifiant de l'esclave est enregistré dans la liste des esclaves disponibles.

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

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

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

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

Chapitre 6
Parallélisation sur une architecture MPPA

Sommaire

La société Kalray développe depuis 2013 une famille de processeurs avec un grand nombre de cœurs nommés *Multi-Purpose Processor Array* (MPPA). Un partenariat entre le laboratoire LIASD de l'université Paris 8 et la société Kalray, nous a permis de disposer dune machine dotée du tout premier modèle de carte MPPA, la *MPPA-256* proposant 256 processeurs de calcul regroupés en 16 clusters.

Les outils logiciels permettant d'exploiter cette architecture étaient en cours de développement au moment de nos travaux, toutes les fonctionnalités n'étaient pas encore exploitées et certains éléments logiciels n'étaient pas totalement opérationnels. Au même moment, en 2013, *LeJoueur* de Jean-Noël Vittaut était en cours de programmation. Jean-Noël Vittaut envisageait différentes parallélisations au niveau de la construction du circuit logique permettant d'interpréter les règles du jeu, au niveau du calcul des *playouts* et de la construction de l'arbre de jeu [Vittaut, 2017]. Nous avons évalué les capacités de la carte MPPA et les contraintes techniques qu'elle impose tout en étudiant les possibilités de parallélisation de *LeJoueur* de Jean-Noël Vittaut sur cette architecture particulière [Hufschmitt, 2014; Hufschmitt *et al.*, 2015b,a].

Nous commençons par présenter l'architecture de la machine MPPA-256. Nous indiquons ensuite la technique de parallélisation réalisable sur le MPPA et la structure de notre programme. Nous étudions la performance du MPPA pour la transmission de messages de taille variable. Nous examinons les meilleures approches pour le calcul des *playouts*. Enfin, nous comparons différentes approches pour la synchronisation des *threads* à l'intérieur des clusters.

FIGURE 6.1 – Carte MPPA, image tirée de Kalray [2013a].

6.1 L'architecture MPPA-256

La machine *MPPA-256* est un serveur équipé d'un processeur Intel Core i7 à 3.6GHz tournant sous Linux et d'une carte PCIe Application Board AB01 équipée d'une puce MPPA-256 (voir figure 6.1) [Kalray, 2013a]. Cette machine développée par la société Kalray est commercialisée depuis 2013. La société Kalray (Essonne, France), créée en 2008, conçoit des semi-conducteurs et des logiciels. Elle développe et vend cette nouvelle génération de processeurs multi-cœurs pour l'imagerie, les infrastructures de télécommunication, les applications de calcul intensif et les applications embarquées de sécurité des données et réseaux.

Le Multi Purpose Processor Array (MPPA-256) est un processeur multi-cœurs, premier membre de la famille *MPPA MANYCORE* ; il comprend 256 processeurs de calcul, les autres MPPA pouvant aller jusqu'à 1024 processeurs en une seule puce silicium. Le MPPA-256 [39] est composé d'une grille de 16 clusters connectés à travers un réseau haute vitesse (Network-on-Chip, NoC). Chaque cluster (voir figure 6.2) contient 16 processeurs élémentaires utilisant une architecture VLIW (Very Long Instruction Word) [40] et un processeur système exécutant un système d'exploitation spécifique (NodeOS). Ce processeur système a pour rôle de lancer le programme dédié au cluster et de gérer les communications, ce qui laisse chaque processeur élémentaire totalement disponible pour exécuter les calculs eux-mêmes (voir figure 6.3). Chaque cluster dispose d'une mémoire partagée de 2Mo et chaque processeur élémentaire peut exécuter son propre code stocké dans cette mémoire partagée. Le MPPA-256 est une architecture MIMD (Multiple Instruction streams, Multiple Data) avec une mémoire distribuée accessible uniquement localement, i.e. chaque cluster est encapsulé, il n'accède qu'à sa propre mémoire et peut exécuter un code distinct des autres clusters.

La limitation de la mémoire à 2Mo est une contrainte majeure dans le développement d'une appli-

39. Le discours commercial de la société Kalray met en avant la faible consommation énergétique du MPPA-256 ainsi que la facilité de développement d'applications C/C++ dédiées à cette architecture *MPPA MANYCORE* grâce à la chaîne de compilation *MPPA ACCESSCORE*. Selon l'affirmation de Joël Monier, PDG de Kalray, dans une interview donnée à silicon.fr le 26 août 2013 (`https://www.silicon.fr/joel-monnier-kalray-a-decide-de-reinventer-le-processeur-88705.html/`), le MPPA-256 consomme moins de 10 W contre 130 W typiquement pour un Intel Core i7.

40. Dans une architecture VLIW, les instructions en assembleur (parfois nommées *bundle*) sont composées de différents champs (ou *slots*). Chaque champ commande une opération sur une unité de la machine. Certains champs sont réservés pour des types d'opérations spécifiques (calcul sur des entiers, calcul sur des flottants, accès mémoire, branchements). Le compilateur a la charge de générer un code qui prend en compte la disponibilité des ressources et préserve la synchronisation des opérations. L'intérêt de cette architecture est de permettre l'exécution de plusieurs opérations à chaque cycle d'horloge.

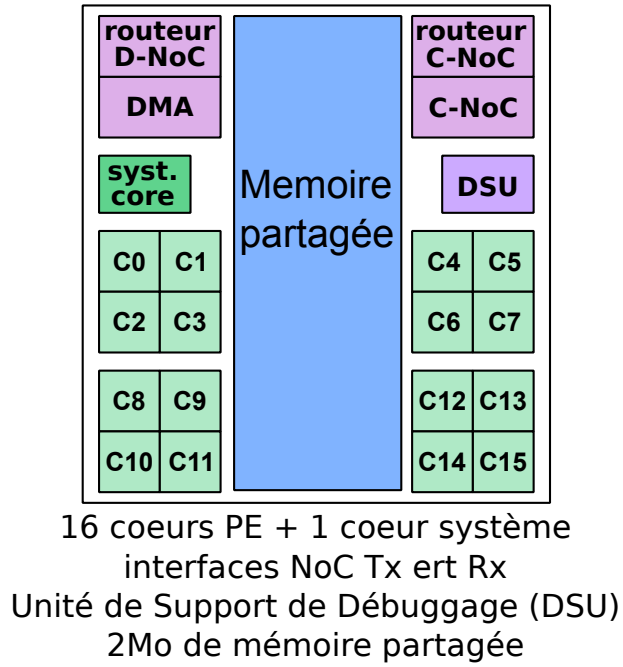


FIGURE 6.2 – Un cluster de la carte MPPA-256 [Kalray, 2013a].

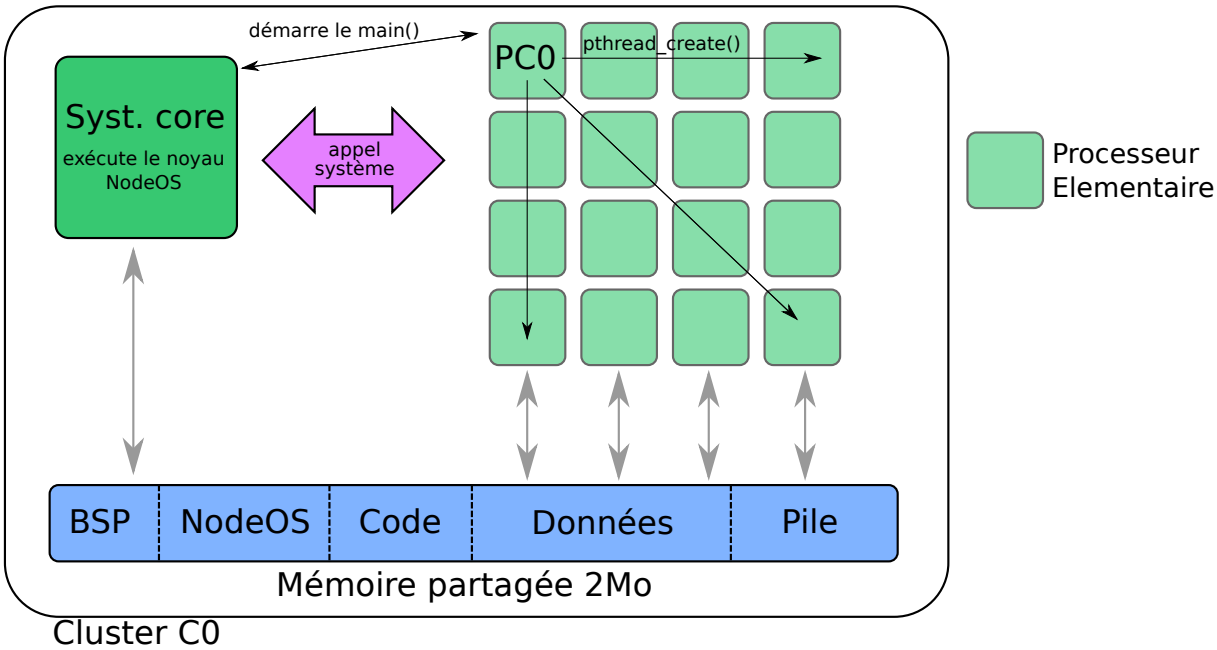


FIGURE 6.3 – Exécution d'un programme sur un cluster [Kalray, 2013a].

cation pour le MPPA : cette mémoire doit contenir l'OS du processeur système, le code et les données des 16 processeurs élémentaires. Nous verrons dans la section suivante, que cette limitation diminue radicalement les choix possibles de parallélisation.

Quatre interfaces d'entrées/sorties (I/O) permettent de gérer les communications entre les clusters et la machine hôte pour deux d'entre eux, entre les clusters et le réseau Ethernet pour les deux autres. Cependant la version 1.01.[41] des outils logiciels *MPPA ACCESSCORE* permettant l'exploitation de la carte ne permet d'utiliser qu'une seule interface d'I/O comme intermédiaire entre la machine hôte et les clusters de calcul (voir figure 6.4). Cette interface d'entrée/sortie possède un processeur quadri-cœur SMP avec une mémoire DDR3 de 4Go et une interface PCIe Gen3 pour les échanges de données avec la machine hôte. Différents connecteurs logiciels permettent d'implémenter des communications synchrones ou asynchrones utilisant un simple tampon (*buffer*) ou un système de file de message (*queues*).

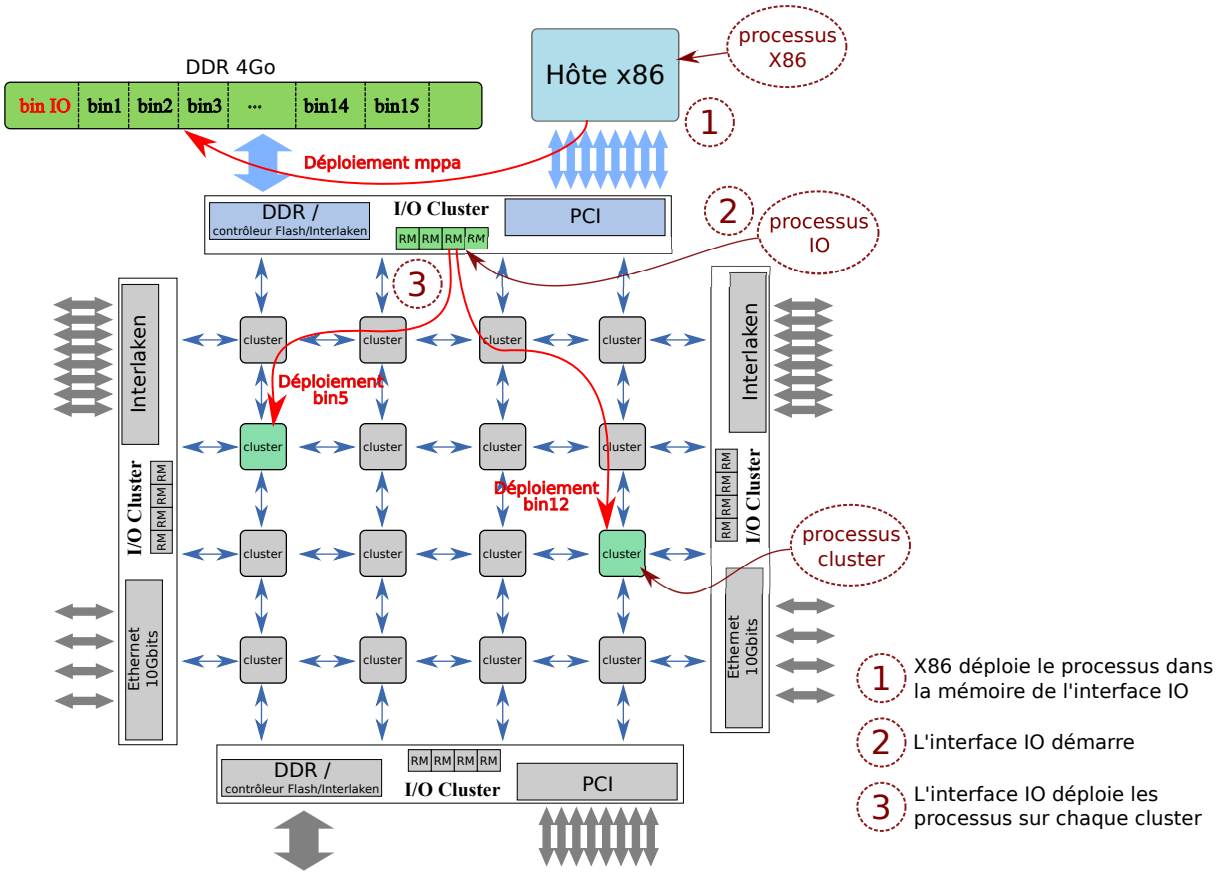


Figure 6.4 – Schéma du serveur équipé d'une carte MPPA-256 [Kalray, 2013b], trois programmes distincts sont déployés sur l'hôte, l'IO et les clusters.

Les premiers tests effectués sur le MPPA-256 [Jouandeau, 2013] montrent que la puissance de calcul du MPPA sur 256 cœurs est proche de celle de la machine hôte équipée d'un processeur à 8 cœurs. Cette estimation est fondée sur la performance observée pour un des cœurs dans la résolution du problème du verre de Spin et multipliée par le nombre de cœurs ce qui implique une parallélisation avec un temps de

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

communication nul. Les performances de cette carte MPPA-256 semblent limitées, mais il s'agit du tout premier membre de la famille *MPPA MANYCORE*. Le MPPA est encore une machine expérimentale et ses outils logiciels sont en cours de développement. Plusieurs fonctionnalités prévues par l'architecture n'étaient pas encore disponibles avec les outils de MPPA ACCESSCORE que nous avons utilisés.

6.2 Parallélisation de *playouts* sur le MPPA

La taille de la mémoire disponible dans les clusters (qui est de 2Mo [42]) ne permet pas la construction d'un arbre de jeu lorsque celui-ci n'est pas trivial. Il est donc impossible de procéder à une *parallélisation au niveau de l'arbre*, *à la racine* ou *avec arbre distribué* sur le MPPA. Nous avons en revanche pu envisager une *parallélisation aux feuilles*

Notre programme est fondé sur *LeJoueur* de Jean-Noël Vittaut, écrit en C++. Ce joueur utilise Yap Prolog pour réaliser une instanciation rapide des règles [Vittaut et Méhat, 2014] et crée un graphe de règles. Ce graphe est factorisé pour réduire sa taille et chaque strate du circuit est transformée en un ensemble de règles sous forme de *bytecode* qui est évalué très rapidement avec des opérateurs binaires. Cet ensemble de règles est transmit au MPPA pour réaliser les simulations. La mémoire limitée à l'intérieur des clusters implique un codage aussi concis que possible pour permettre au circuit d'y être stocké. Malheureusement, pour les jeux les plus complexes, comme *Hex*, la taille du circuit excède celle de la mémoire disponible.

Pour toutes nos expérimentations nous avons utilisé des communications synchrones de manière à pouvoir, par la suite, comparer nos résultats avec ceux obtenus pour les communications asynchrones. La machine hôte envoie des ensembles de positions du jeu à l'interface I/O du MPPA qui les distribue aux différents clusters. Le premier processeur élémentaire (PC0) de chaque cluster est responsable des communications avec l'I/O et du lancement des *threads* sur les autres processeurs élémentaires via le processeur système. Un *thread* est exécuté par chacun des 15 processeurs restants. Le PC0 envoie les positions reçues aux différents *threads* et attend tous les résultats avant de les retourner tous ensemble à l'I/O.

L'interface I/O collecte les résultats de tous les clusters avant de les envoyer tous ensemble vers l'hôte. Nous ferons référence à la transmission d'un ensemble de positions, le calcul des playouts et la récupération des résultats (les scores) sous le terme *run*.

6.3 Passage à l'échelle des communications

Selon les jeux, la taille des positions à transmettre aux clusters pour effectuer les *playouts* est variable. Nous avons donc examiné premièrement la capacité du réseau de communication à transmettre des messages de taille variable de la machine hôte jusqu'aux clusters sans une augmentation pénalisante du temps de communication. La documentation du MPPA indique qu'aucune latence supplémentaire n'est induite par l'augmentation de la taille des transmissions. Afin de nous en assurer nous avons pro-

42. Dans ces 2Mo, on a le système NodeOS, la pile, le code du programme et ses données qui comprennent la description du jeu. Pour exemple, on constate que la description du jeu *Breakthrough* utilise tout l'espace disponible pour les données.

cédé à une expérimentation.

La taille de la description d'une position peut varier significativement d'un jeu à l'autre dans un rapport de 1 à 10 : le codage d'une position nécessite de 200 à 2000 octets [43] pour les jeux courants. Dans le cas de communications synchrones, si l'on souhaite effectuer un *playout* par *thread*, la taille du message à envoyer correspond à la taille d'une position multipliée par le nombre de *threads*. Pour évaluer le passage à l'échelle des communications sur le MPPA, nous avons fait varier la taille des positions envoyées entre l'hôte et l'I/O puis de l'I/O aux clusters de 200 à 2000 octets. Pour éviter que la variation du temps nécessaire au calcul des *playouts* ne perturbe les mesures, aucun *playout* n'est calculé et un résultat vide est retourné immédiatement. Chaque exécution du programme réalise 1000 *runs*. Nous avons mesuré la performance comme le temps nécessaire pour exécuter 1000 *runs* avec de 1 à 16 clusters et de 1 à 16 *threads* par cluster.

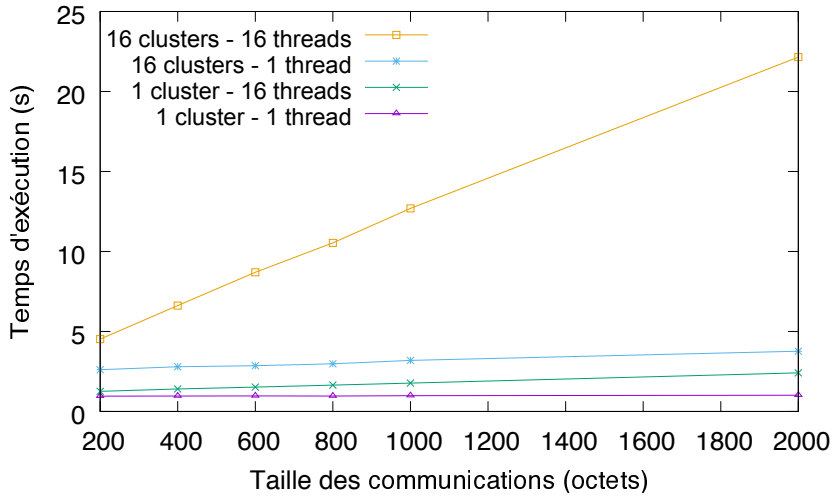


Figure 6.5 – Évolution du temps d'exécution en fonction de la taille des communications (1000 *runs*).

La figure 6.5 présente les résultats. Avec un *thread* par cluster, le temps d'exécution est à peu près constant tandis que la taille des communications est multipliée par 10. Avec 16 *threads* pour 1 cluster, le temps additionnel correspond à l'initialisation des *threads* et la distribution des données entre les *threads*. L'utilisation d'un unique *thread* sur chacun des 16 clusters montre qu'un temps additionnel est nécessaire à l'I/O pour initialiser le connecteur de communication vers le cluster de destination. Le passage à l'échelle est donc contraint par deux aspects : le temps nécessaire à l'I/O pour distribuer les données entre les clusters et le temps de démarrage des *threads*.

Le passage à l'échelle n'est que peu affecté par ces deux aspects pris séparément, mais ce n'est pas le cas lorsqu'ils sont combinés. La courbe obtenue pour 16 clusters et 16 *threads* montre une dégradation significative du passage à l'échelle. Cependant, nous voyons que le temps d'exécution reste acceptable puisqu'il n'est multiplié que par 5 quand la taille des messages est décuplée.

43. Ce codage pourrait être optimisé par un vecteur de bits, tout en restant de taille variable à cause de la diversité des descriptions de jeu en GGP.

6.4 Différentes approches pour le calcul des *playouts*

Même si une *parallélisation aux feuilles* est la seule que nous ayons envisagée sur le MPPA, plusieurs approches sont possibles pour le calcul des *playouts* : chaque processeur élémentaire peut calculer un ou plusieurs *playouts* à partir d'une position donnée ou bien, à l'intérieur d'un cluster les différents processeurs élémentaires peuvent collaborer au calcul d'un seul et même *playout* en se partageant l'évaluation du circuit.

Nous avons conduit les expérimentations suivantes sur trois jeux : *Tictactoe* qui a des playouts courts (entre 5 et 9 coups) et un circuit de taille réduite rapide à évaluer, *EightPuzzle* dont le circuit est également petit mais les playouts s'étendent jusqu'à 60 coups, et *Breakthrough* dont le circuit est plus long à évaluer. Chaque exécution de notre programme réalise 10000 *runs*. Nous avons mesuré la performance sur la base du nombre total de *playouts* exécutés par seconde.

6.4.1 Parallélisation de l'évaluation du circuit

Nous avons tout d'abord envisagé la distribution du calcul de chaque strate du circuit sur les différents processeurs élémentaires d'un cluster. Les règles de chaque strate peuvent être évaluées dans n'importe quel ordre. Cependant, chaque strate dépend de l'évaluation de la précédente, une synchronisation est donc nécessaire.

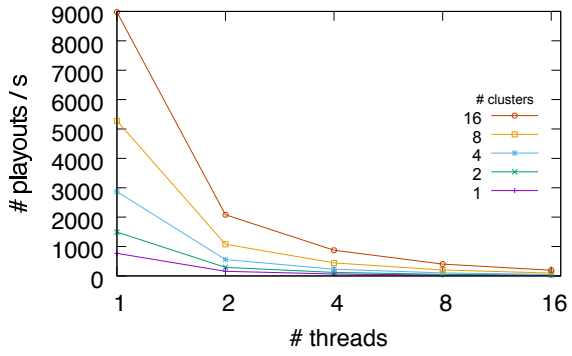


Figure 6.6 – Évolution du nombre moyen de playouts par seconde pour 10000 *runs* pour le jeu *Tictactoe* lorsque le calcul des *playouts* est distribué entre les *threads*.

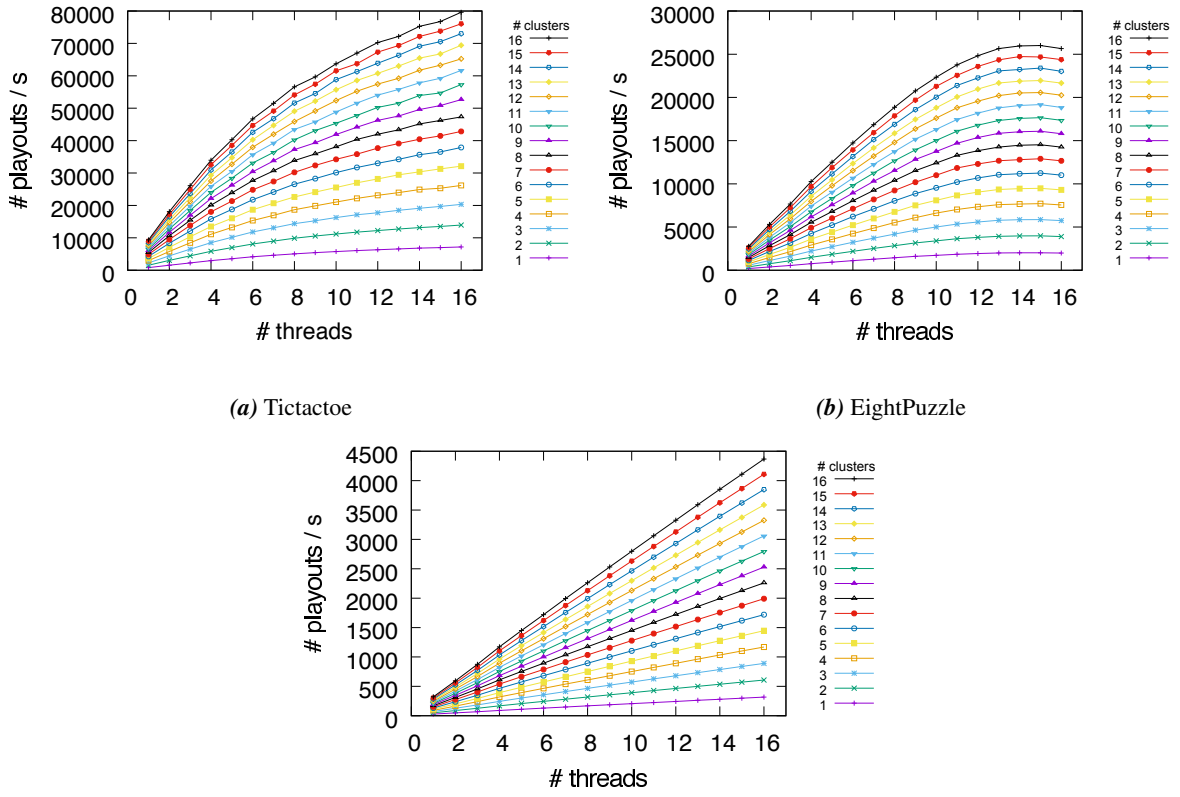
La figure 6.6 présente la performance obtenue sur le jeu *Tictactoe* avec de 1 à 16 clusters et de 1 à 16 *threads* par cluster. Les différentes courbes confirment que l'augmentation du nombre de clusters actifs permet d'augmenter le nombre de *playouts* exécutés par seconde et n'a pas d'impact sur la variation de la performance. Cette dernière dépend du nombre de *threads* utilisés sur chaque cluster. Chaque courbe montre une nette dégradation de la performance à mesure que l'évaluation est distribuée entre les *threads*. Cela indique combien le coût de la synchronisation généré par la division de l'évaluation des *playouts* est significatif.

Nous pouvons en conclure que notre implémentation du calcul distribué des *playouts* ne permet pas d'obtenir un quelconque gain de performance étant donné l'important surcoût induit par la barrière de synchronisation entre les strates.

6.4.2 Parallélisation de l'évaluation des *playouts*

Pour éviter le coût généré par une synchronisation entre chaque strate, il est possible d'utiliser chaque processeur élémentaire pour exécuter un *playout* distinct, chaque *thread* évaluant l'ensemble des strates à chaque étape du playout. La synchronisation est uniquement nécessaire pour lancer le calcul lorsque la position initiale est définie et pour collecter les résultats.

La figure 6.7 présente les résultats pour les trois jeux testés en utilisant de 1 à 16 clusters et de 1 à 16 *threads*. Les résultats montrent un bon passage à l'échelle.



(a) Tictactoe

(b) EightPuzzle

(c) Breakthrough

FIGURE 6.7 – Évolution du nombre de playtous par seconde (moyenne sur 10000 runs) en calculant un playout par *thread*.

Les résultats pour le jeu *Tictactoe* (fig.6.7a) montre une progression de ≈450 *playouts/s* pour un cluster et un *thread* jusqu'à ≈77700 *playouts/s* pour 16 clusters et 16 *threads* i.e. une multiplication par 170 du nombre de *playouts* par seconde pour une multiplication par 256 de la puissance de calcul.

Pour le jeu *EightPuzzle*, le nombre de *playouts* est multiplié, au mieux, par 133 (pour 14 *threads* par cluster) en utilisant 224 cœurs. La performance ne passe pas bien à l'échelle après 14 *threads*. Nous expliquons ce résultat par la longueur constante des *playouts* : en jouant au hasard, la solution a peu de chance d'être trouvée et les *playouts* sont stoppés après 60 coups par le *stepper*. Tous les scores sont alors envoyés à l'interface I/O au même moment par tous les clusters, provoquant une congestion du réseau de communication. Un ralentissement similaire entravant le passage à l'échelle est observé pour *Tictactoe* en forçant les joueurs à complètement remplir la grille à chaque partie : la longueur des *playouts* est alors

systématiquement de 9 coups.

Le passage à l'échelle est meilleur pour *Breakthrough*. Le temps de calcul est plus long, le poids des communications et synchronisations est donc moins important en comparaison. Le nombre de *playouts* est multiplié par 155 pour une multiplication par 256 de la puissance de calcul.

6.5 Stratégies de gestion des threads

Dans les expériences précédentes, les *threads* étaient relancés pour chaque lot de *playouts*. Une autre stratégie consiste à garder les *threads* en activité en attendant une requête de calcul. Pour tester différentes stratégies de synchronisation, nous avons changé la gestion des *threads* par rapport à la section 6.4.2 : sur chaque processeur élémentaire de chaque cluster, un *thread* est lancé en attente d'une position à traiter.

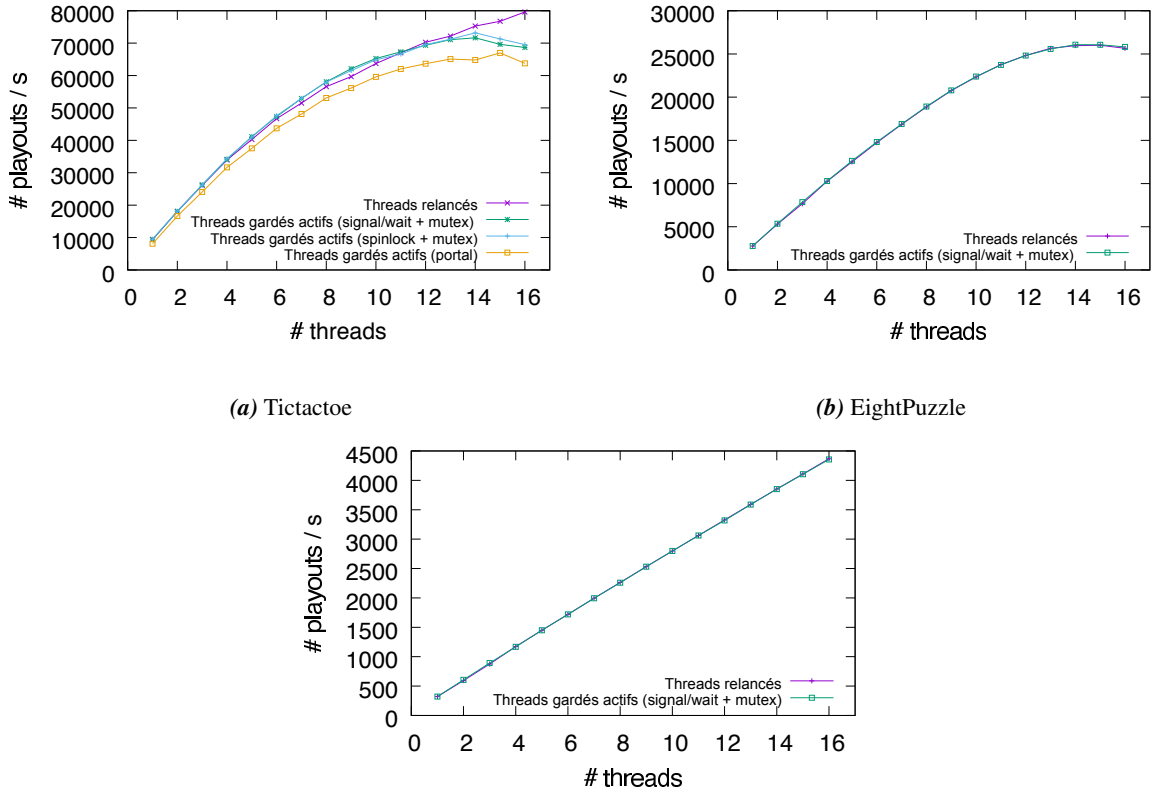
Toutes les positions envoyées correspondent à la position initiale du jeu et l'identifiant de chaque processeur élémentaire est utilisé comme graine du générateur pseudo-aléatoire qui détermine l'exécution des *playouts*. Les tests ont été menés sur différents jeux en faisant varier le nombre de clusters et de *threads* par cluster ; 10000 *runs* sont exécutés chaque fois.

Dans la première partie de l'expérimentation, nous avons utilisé les fonctions POSIX `pthread_cond_wait` et `pthread_cond_signal` pour communiquer les requêtes de *playouts* aux *threads*. Dans la seconde partie, nous avons remplacé ces fonctions POSIX par une attente active. Dans la troisième partie, nous avons utilisé le connecteur de type *portal* qui fait partie des outils *MPPA ACCESCORE* et utilisé le *Network on Chip* (NoC) pour créer des canaux de communication entre le *thread* exécuté sur le premier processeur élémentaire PC0 et les autres *threads* du cluster. Cette troisième approche permet d'éviter d'utiliser des *mutex*.

Les résultats sont regroupés sur la figure 6.8 pour 16 clusters et de 1 à 16 *threads*. Ils sont comparés aux résultats obtenus avec des *threads* redémarrés à chaque *run*. Pour les jeux *EightPuzzle* et *Breakthrough*, au regard des résultats obtenus avec *Tictactoe*, nous avons limité les expérimentations à la comparaison du redémarrage des *threads* avec l'utilisation des fonctions POSIX `pthread_cond_wait` et `pthread_cond_signal`.

Avec le jeu *Tictactoe*, l'utilisation des *portals* est moins efficace quel que soit le nombre de *threads* utilisés et les différentes approches conservant les *threads* actifs passent mal à l'échelle au dessus de 14 *threads* par cluster. Nous notons que le redémarrage des *threads* pour chaque requête de calcul fournit le meilleur passage à l'échelle et donne les meilleurs résultats, ce qui contredit l'hypothèse que nous avions formulée pour cette expérience. Les résultats obtenus pour les jeux *EightPuzzle* et *Breakthrough* ne montrent pas une différence aussi marquée ; les courbes sont quasiment identiques. Ce qui confirme que garder les *threads* actifs n'apporte aucun bénéfice.

Ceci peut s'expliquer par l'utilisation d'une mémoire partagée entre les différents processeurs élémentaires d'un cluster et le fait que les primitives de synchronisation sont implémentées par une attente active (*spinlock*) ou autre processus similaire. Quand un *thread* a achevé son calcul, il attend un signal en interrogeant régulièrement la mémoire partagée (*polling*). En conséquence, si les autres *threads* n'ont pas achevé le calcul de leur *playout*, ils sont ralentis car ils essayent eux aussi d'accéder à la mémoire partagée. Le processus principal lancé sur le PC0, responsable des communications avec l'interface I/O,



(a) Tictactoe

(b) EightPuzzle

(c) Breakthrough

Figure 6.8 – Évolution de la performance i.e. du nombre de playouts par secdonde (10000 *runs*, 16 clusters) pour différentes stratégies de gestion des *threads*.

est également ralenti. Plus le nombre de *threads* ayant fini leur tâche est important, plus les *threads* restants sont ralentis. Au contraire, quand un *thread* fait un appel à `pthread_exit`, le processeur élémentaire correspondant est placé dans un état de repos (*idle state*) et ne perturbe pas le travail des autres processeurs élémentaires. Stopper et relancer les *threads* est donc plus efficace dans notre cas.

Chapitre 7
Conclusion

Nous avons présenté les différentes techniques de parallélisation de MCTS existant dans la littérature. La *parallélisation de l'arbre* consiste à utiliser plusieurs *threads* pour construire un arbre unique dans une mémoire partagée. Dans le cas de la *parallélisation aux feuilles*, un processus maître construit l'arbre et utilise des processus esclaves pour réaliser les *playouts*. La parallélisation de NMC reprend le principe de la *parallélisation aux feuilles*, mais utilise en plus un processus *répartiteur* qui sert d'intermédiaire entre le maître et les esclaves. La *parallélisation à la racine* consiste à construire plusieurs arbres dans une mémoire distribuée et à synchroniser les résultats à intervalles réguliers. Enfin, la *parallélisation distribuée* consiste à construire un arbre unique dont les nœuds sont stockés sur différentes machines.

Nous avons examiné les possibilités offertes par l'architecture *Multi Purpose Processor Array* (MPPA) pour la parallélisation de *LeJoueur* de Jean-Noël Vittaut. La limitation de la mémoire à l'intérieur des clusters à une taille de 2Mo est une contrainte majeure. Parmi les différentes options de parallélisation décrites dans la littérature, la seule applicable dans une mémoire aussi limitée est la *parallélisation aux feuilles*.

Nous avons montré que le MPPA permet un bon passage à l'échelle quand la taille des communications augmente, ce qui permet un bon résultat en utilisant des communications synchrones et en transmettant des ensembles importants de positions initiales de jeux pour le calcul d'un lot de *playouts*.

Nous avons considéré deux approches pour le calcul des *playouts*. Le calcul distribué d'un playout sur différents processeurs élémentaires provoque un important surcoût de synchronisation. Le calcul d'un *playout* complet sur chaque processeur élémentaire donne de meilleurs résultats.

Nous avons constaté que, pour notre tâche, il est plus efficace de relancer les *threads* pour chaque requête de calcul. Toutes les primitives de synchronisation du MPPA utilisent une attente active (*spinlock*). Les *threads* ayant achevé leurs calculs provoquent donc un ralentissement des autres *threads* s'ils sont gardés en attente à cause des accès concurrents à la mémoire partagée.

Il serait possible de réduire le poids des communications en réalisant plusieurs *playouts* par position. Cependant, la réalisation de plusieurs *playouts* à partir de la même position peut être moins efficace qu'à partir de positions distinctes [Chaslot *et al.*, 2008].

Jean-Noël Vittaut a testé la parallélisation de *LeJoueur* sur GPU. Le calcul d'un *playout* est distribué sur les différentes unités de calcul du GPU comme dans notre approche de la section 6.4.1. Malgré le

fait qu'aucune barrière de synchronisation n'est nécessaire pour synchroniser le calcul des strates et que la mémoire est bien moins limitée, Vittaut [2017] indique que l'utilisation du GPU n'apporte aucun gain en temps comparé au CPU. Ceci est dû au temps de transfert des données de la RAM vers la DRAM qui est extrêmement pénalisant.

Nous avons envisagé l'utilisation de communications asynchrones pour permettre la réalisation d'un nombre plus important de *playouts* par seconde. Cependant, les premières expérimentations que nous avons menées ont été perturbées par différents dysfonctionnements du *middleware* version 1.2.1. Ces problèmes seront sûrement résolus dans les versions futures du *MPPA ACCESSCORE* de Kalray. L'utilisation de communications asynchrones devra donc faire l'objet de futures expérimentations.

Sur les nouvelles versions du MPPA il serait également intéressant de tester de nouvelles techniques de parallélisation comme la création de mini-arbres dans la mémoire limitée des clusters pour économiser les temps de communication. L'algorithme SHOT [Cazenave, 2015b] peut également être une alternative intéressante pour utiliser moins de mémoire : cet algorithme peut être efficacement parallélisé.

Troisième partie

Décomposition

Entre 2005 et 2007, une nouvelle classe de jeux apparaît : les jeux composés. Ces jeux sont composés de différents sous-jeux connus (*Tictactoe*, *Maze*, *Blocks World*). Par exemple *Chinook* est un jeu constitué de deux parties de dames Anglaises jouées simultanément sur les cases blanches et les cases noires du damier [44].

Bien que ces sous-jeux soient très simples pris séparément, certains joueurs se montrent peu performants quand ils sont combinés car l'espace de recherche croît alors de manière exponentielle. Les organisateurs de la compétition IGGPC ne cachent pas leur objectif qui est de stimuler la recherche d'un raisonnement permettant un réel changement d'échelle dans la performance des joueurs. Ils espèrent ainsi décourager la recherche de performance réduite à des optimisations et à de la parallélisation. Genesereth et Björnsson [2013] indiquent que la compétition commence à mettre l'accent sur les jeux composés, présentant une importante difficulté et une forte combinatoire, dans le but de relancer l'intérêt pour les heuristiques et d'encourager la recherche et l'identification de structures de haut niveau dans les jeux. Ils indiquent cependant que les quelques algorithmes proposés entre 2009 et 2013 [Cox *et al.*, 2009; Günther, 2007; Günther *et al.*, 2009; Schiffel, 2011] ne sont pas utilisés en compétition.

Lors du workshop GIGA (conférence IJCAI) de 2016, les représentants de l'IGGPC de Stanford ont réitéré leur souhait de voir des joueurs analyser les règles des jeux et en extraire des connaissances de haut niveau pour diminuer la complexité de leur exploration. Les travaux se focalisant sur la parallélisation ou l'amélioration de techniques comme celle des *propnets* ne pouvant, d'après eux, que repousser la limite de ce qui est calculable sans apporter de véritable progrès au domaine de l'intelligence artificielle. Dans le but d'encourager ce type de recherche, de plus en plus de jeux suffisamment complexes pour mettre en échec une recherche de type UCT sont proposés. C'est dans cette optique de produire des jeux complexes avec un fort facteur de branchement ou une profondeur importante de l'arbre du jeu que les jeux composés sont de plus en plus mis en avant.

Dans cette partie, nous commençons par présenter un *bestiaire* des jeux composés pour en expliquer la nature et la diversité. Nous faisons ensuite un état de l'art des méthodes de décomposition dans le domaine du GGP, mais aussi dans les domaines de la planification, de la satisfiabilité booléenne (SAT), de la satisfaction de contraintes (CSP) et de la vérification de modèles (*model checking*) qui sont connexes au GGP. Notre état de l'art couvre également le problème de la détection de symétries qui permet de décomposer un problème en sous-problèmes identiques, ainsi que les techniques de détection d'invariants, prérequis pour la détection de symétrie et la décomposition. Nous présentons enfin la contribution majeure de cette thèse : deux méthodes générales de décomposition des jeux GGP, l'une fondée sur une analyse des règles, l'autre sur une analyse statistique de simulations de parties.

44. Voir l'annexe A pour une description détaillée de tous jeux cités dans la suite de ce mémoire

Chapitre 8
Bestiaire des jeux composés

Sommaire

Les jeux composés présentent une grande diversité. Différentes classes de jeux composés peuvent néanmoins être identifiées. Chaque type de composition présente des difficultés particulières, soit au niveau de l'identification des sous-jeux, soit au niveau de l'exploitation de cette information pour pouvoir jouer efficacement et constituer une stratégie à partir de l'analyse de sous-jeux distincts. Ainsi la décomposition des jeux ne constitue pas un problème, mais bien un ensemble de sous-problèmes distincts à résoudre simultanément. Il est donc utile d'établir un classement de ses jeux mettant en évidence les caractéristiques à prendre en compte dans la réalisation d'une méthode générale de décomposition.

Nous proposons ici une classification des différents types de compositions présents parmi les jeux du serveur `http://games.ggp.org/` [Schreiber, 2017] qui tient compte de ces deux critères majeurs : les caractéristiques particulières qui influencent l'identification des sous-jeux (jeux présentant un *stepper*, jeux multiples, jeux à actions composées, jeux en série, jeux parallèles), et celles qui engendrent des difficultés particulières au moment de réunir les informations obtenues pour chaque sous-jeu en une stratégie globale (jeux synchrones, asynchrones, en série, jeux impartiaux, sous-jeux terminaux ou influençant le score).

Notre classification reprend les notions de jeux *séquentiels*, *synchrones* et *asynchrones* proposées

Synchronisation	Nombre de joueurs	Alternance des actions
séquentielle	quelconque	sous-jeux à coups alternés ou simultanés
parallèle asynchrone	quelconque	actions simples ou composées
parallèle synchrone	jeu solitaire	actions composées
	jeu multijoueur	sous-jeux solitaires
		sous-jeux à coups alternés
		actions composées alternées ou simultanées

Figure 8.1 – Classification des jeux composés en fonction du mode de synchronisation des sous-jeux, du nombre de joueurs et de l'alternance des actions des joueurs dans les sous-jeux. Tous ces types de jeux peuvent également présenter des sous-jeux multiples et un ou plusieurs *steppers*.

par Cerexhe *et al.* [2014] qui sont focalisées sur la résolution des jeux décomposés[45]. Nous les intégrons dans une classification plus détaillée qui tient compte des particularités des jeux influençant la décomposition. La figure 8.1 présente cette classification. Les jeux sont organisés en fonction du type de synchronisation des sous-jeux et de l'alternance des actions des joueurs dans les différents sous-jeux. À ces caractéristiques s'ajoutent également deux particularités qui peuvent être présentes dans toutes les classes de jeux : les jeux peuvent être multiples et/ou utiliser un *stepper*.

Dans ce chapitre, nous commençons par présenter les particularités des jeux utilisant un *stepper*, des jeux multiples et expliquons la difficulté engendrée par la présence d'actions composées dans certains jeux. Nous détaillons ensuite chacune des classes de jeux composés avec sous-jeux *séquentiels*, parallèles *synchrones* ou *asynchrones*. Enfin nous présentons le cas particulier des jeux impartiaux décomposables, nous définissons les notions de *sous-jeux terminaux* et/ou *influençant le score*, et nous évoquons le cas des sous-jeux qui deviennent décomposables en cours de partie.

8.1 Jeux avec un *stepper*

Certains jeux composés ne sont pas le résultat de la juxtaposition de plusieurs jeux, cependant ils peuvent quand même être considérés comme des jeux composés, ce sont les jeux possédant un *stepper* (représenté par le prédicat `(step ?x)` dans les jeux non obfusqués) i.e. un compteur qui numérote des tours du jeu et permet de mettre fin au jeu au bout d'un nombre fini de tours (fig.8.2). Les compteurs sont utilisés dans les jeux qui sont susceptibles de contenir des cycles comme *Asteroids*, *Eight Puzzle*, *Blocks* pour s'assurer qu'il se terminent en un nombre fini de tours. Dans le jeu *Roshambo2* (jeu de Pierre, Papier, Ciseau, Puits) un *stepper* est également utilisé, cette fois pour numéroter les manches.

Dans les jeux utilisant un *stepper*, différentes descriptions d'une position ne diffèrent que par la valeur du *stepper*. En opérant une décomposition pour séparer le *stepper* du reste du jeu, il est possible ensuite de tirer parti des transpositions pour simplifier la résolution du jeu. Le *stepper* constitue alors un

45. Si un jeu est composé de N sous-jeux et que l'arbre de chaque sous-jeu i présente un facteur de branchement B_i et une profondeur P_i, la complexité du jeu composé est de l'ordre de :

- $\prod_{i=1}^{N} B_i^{P_i}$ pour N sous-jeux séquentiels,
- $(\prod_{i=1}^{N} B_i)^{P}$ pour N sous-jeux parallèles synchrones avec $P = P_i$ pour $1 \leq i \leq N$, et
- $(\sum_{i=1}^{N} B_i)^{\sum_{i=1}^{N} P_i}$ pour N sous-jeux parallèles asynchrones.

La complexité du jeu décomposé est en revanche de l'ordre de $\sum_{i=1}^{N} B_i^{P_i}$ et peut descendre à $B_i^{P_i}$ dans le cas d'un jeu multijoueur à sous-jeux solitaires parallèles synchrones : l'exploration peut se concentrer sur le sous-jeu i dans lequel le joueur peut agir.

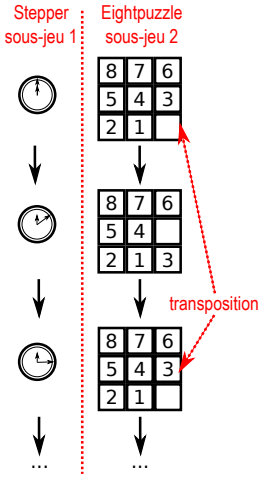


FIGURE 8.2 – Représentation schématique du jeu *Eightpuzzle* décomposé. Le jeu principal est séparé de son *stepper*. Des positions identiques sont mises en évidence et constituent une transposition.

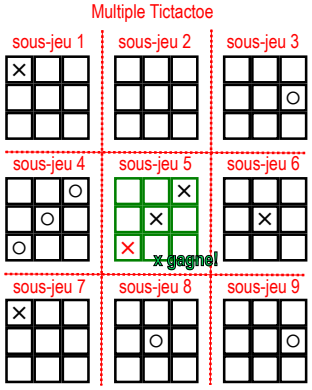


FIGURE 8.3 – Exemple de jeu multiple : représentation schématique du jeu *Multiple Tictactoe* décomposé. La grille centrale est la seule utilisée pour déterminer le score. La victoire de *O* dans le sous-jeu numéro 4 est totalement ignorée.

sous-jeu indépendant des actions.

Plus de 50% des descriptions de jeux présentes sur le serveur `http://games.ggp.org/` [Schreiber, 2017] contiennent un *stepper*. Tirer parti de la décomposition d'un *stepper* constituerait donc un avantage notable pour un joueur.

8.2 Jeux multiples

Certains jeux composés multiplient les sous-jeux pour augmenter le nombre d'actions à évaluer à chaque position du jeu et augmenter ainsi le facteur de branchement dans l'arbre du jeu. Cependant, dans ces *jeux multiples*, un seul sous-jeu a une influence sur le score, une victoire ou une défaite dans les autres sous-jeux n'ayant absolument aucune influence sur ce dernier (fig.8.3).

Dans cette catégorie, on peut trouver par exemple les jeux *Multiple Button And Lights*, *Multiple Hamilton* ou *Multiple Tictactoe*. Ces jeux regroupent des sous-jeux indépendants qui ne posent donc pas de difficulté particulière pour la décomposition. Une fois le jeu décomposé, une vérification assez élémentaire permet de détecter les *sous-jeux inutiles* qui peuvent donc être purement et simplement

ignorés lors de la résolution du jeu.

Dans le jeu *Incredible*, un ensemble d'actions `(contemplate ?x ?y)` sont proposées au joueur juste dans le but d'augmenter le facteur de branchement dans l'arbre du jeu. On peut, soit considérer ces actions comme constituant un sous-jeux inutile, soit les éliminer directement en considérant qu'elles constituent un ensemble d'action *noop* [46].

8.3 Jeux à actions composées

La présence d'*actions composées* dans un jeu, représente une difficulté particulière pour la décomposition. Ces actions ont la particularité d'agir sur plusieurs sous-jeux en même temps i.e. de modifier la valeur des fluents de ces sous-jeux, ce qui crée un lien fort entre ces derniers et rend la décomposition complexe. Les sous-jeux affectés par les actions composées sont parallèles et si ces actions agissent sur tous les sous-jeux, ceux-ci sont nécessairement *synchrones*. Dans certains cas, les actions composées peuvent agir seulement sur une partie des sous-jeux. Par exemple, on trouve ce cas de figure dans le jeu *Lights On Simultaneous*.

Les jeux parallèles à *actions composées* peuvent être mono- ou multijoueurs. Par exemple dans *Asteroids Parallel* et *Blocks World Parallel* un seul joueur joue simultanément deux parties de *Asteroids* ou de *Blocks* (fig.8.5a). Tandis que dans les jeux *Tictactoe Parallel* et *Blocker Parallel* deux joueurs s'affrontent dans deux parties de *Tictactoe* ou de *Blocker* jouées simultanément (fig.8.7a et 8.7b).
Remarquons que le jeu *Blocker* présente la particularité d'être un jeu à action simultanées : les deux joueurs jouent une action différente de *noop* à chaque tour. Le nombre de combinaisons possibles des actions des deux joueurs, associé aux actions composées, fait de *Blocker Parallel* un jeu particulièrement lourd à instancier, rendant également sa décomposition et sa résolution difficiles.

Les actions composées sont généralement constituées de deux parties, mais il est possible d'augmenter encore la combinatoire avec des actions composées de plus de 2 options des joueurs. Dans le jeu *Lights On Simul 4*, par exemple, les actions sont composées de 4 options i.e. chaque action équivaut à quatre actions consistant à allumer une des lampes.

8.4 Sous-jeux séquentiels (jeux en série)

Les jeux en série associent différents sous-jeux joués l'un après l'autre, de manière *séquentielle* (fig.8.4), tel que défini par Cerexhe *et al.* [2014]. Dans ces jeux, la position terminale d'un sous-jeu est souvent une condition nécessaire à la légalité des actions du sous-jeu suivant.

La présence d'une expression définie à partir des fluents du premier sous-jeu et pré-conditionnant la légalité des actions du second sous-jeu rend la décomposition des jeux en série difficile. À cause d'elle les sous-jeux ne sont pas indépendants, la détection de cette expression est donc un élément critique pour la décomposition. Pour mettre au point une méthode générale de décomposition, il faut prendre en compte le fait que l'expression séparant les sous-jeux peut être utilisée non pas pour pré-conditionner la

46. Une action *noop* parmi toutes celles détectées peut néanmoins être explorée pour éviter un *Zugzwang* i.e. un coup forcé défavorable.

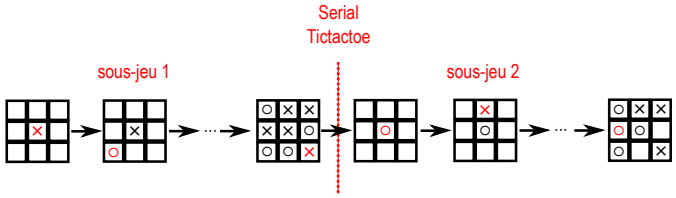


FIGURE 8.4 – Exemple de sous-jeux séquentiels : représentation schématique du jeu *Serial Tictactoe* décomposé. Le second sous-jeu démarre quand le premier est achevé. La limite entre les sous-jeux n'est pas explicitement décrite dans les règles

légalité des actions, mais pour pré-conditionner leurs effets. Toutes les actions des différents sous-jeux sont alors légales, mais les actions du second sous-jeu sont sans effet tant que la première partie du jeu n'est pas achevée.

Les jeux en série présents dans les dépôts, comme *Asteroids Serial*, *Blocker Serial*, *Blocks World Serial* ou *Tictactoe Serial* associent deux parties d'un même jeu jouées l'une après l'autre. Cependant il existe également des jeux en série enchaînant plus de deux parties. Par exemple *Factoring Easy Turtle Brain*, *Factoring Medium Turtle Brain* et *Factoring Impossible Turtle Brain* enchaînent respectivement 3, 19 et 21 parties de *Lights On* en série. Une particularité de *Factoring Impossible Turtle Brain* est que les 11 premières parties sont inutiles et n'ont aucune influence sur le score. Ce jeu peut donc être assimilé à un jeu multiple bien que les sous-jeux ne puissent pas être purement ignorés : il est nécessaire de jouer ces sous-jeux pour accéder aux suivants.

Quelques jeux en série présentent des caractéristiques particulières qui rendent leur décomposition problématique :

Brain Teaser Extended est constitué de deux parties de *Lights Out* jouées successivement sur un plateau de 3×3 et 4×4 cases. Le premier puzzle doit être résolu pour donner accès à la seconde partie du jeu. L'expression indiquant la résolution du premier sous-jeu ne conditionne pas directement la légalité des actions du second. À la place elle est utilisée pour initialiser le second plateau de jeu, ce qui rend indirectement les actions du second sous-jeu légales.

Tic Block est composé d'une partie de *Tictactoe* suivie d'une partie de *Blocker*. Bien que les plateaux de jeu soient distincts, les mêmes actions sont réutilisées d'une phase du jeu à l'autre pour marquer les cases de même index des deux plateaux.

Dans le jeu *Factoring George Forman* les quatre mêmes lampes (décrites par un ensemble de fluents) sont réutilisées pour jouer 4 parties de *Lights On* en série [47].

Platform Jumper est composé de deux phases distinctes. Dans une première phase de jeu, les joueurs doivent choisir tour à tour de colorer les cases d'une colonne ou d'une ligne du plateau de jeu, les cases déjà colorées ne pouvant plus changer par la suite. Dans une seconde phase, les joueurs doivent déplacer leurs pions sur les cases de leur couleur pour traverser le plateau de jeu jusqu'au bord opposé. Dans ce jeu, les deux phases sont fortement liées. Bien que nettement identifiables, il convient de s'interroger sur l'utilité de séparer ces deux phases du jeu.

47. La description de ce jeu n'est pas bien formée, sa terminaison n'est pas assurée. L'ajout d'un *stepper* suffirait à corriger cette description.

8.5 Sous-jeux parallèles synchrones

Les sous-jeux parallèles se déroulent simultanément. Les jeux composés de sous-jeux parallèles proposés sur les serveurs de GGP sont majoritairement constitués de deux sous-jeux, mais il est tout à fait envisageable de créer des jeux parallèles associant N sous-jeux.

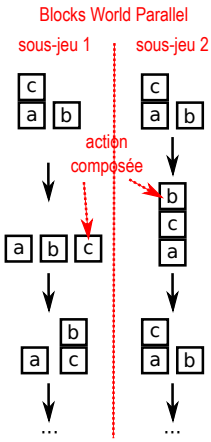
Dans un jeu possédant des *sous-jeux parallèles synchrones*, à chaque tour, dans chaque sous-jeu, au moins une action d'un des joueurs modifie la position dans ce sous-jeu. Ainsi, si deux sous-jeux de durée identique (même nombre de *steps*) se déroulent en parallèle de manière synchrone, ils sont assurés de se terminer en même temps.

Il n'existe pas de jeu *multiple* à sous-jeux synchrones pour le moment. Le caractère synchrone des sous-jeux ne permettrait pas d'éviter de jouer dans les *sous-jeux inutiles*. L'identification de tels sous-jeux permettrait néanmoins de gagner du temps lors de l'exploration en les ignorant.

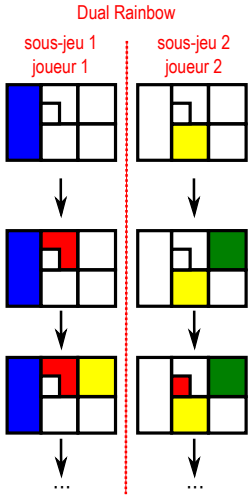
Nous répartissons les sous-jeux parallèles synchrones en différentes classes en fonction du nombre de joueurs impliqués dans chaque sous-jeu et de l'alternance des actions des joueurs.

8.5.1 Sous-jeux solitaires

Les sous-jeux parallèles synchrones peuvent être une combinaison de sous-jeux solitaires. Le jeu global peut dans ce cas être mono- ou multijoueurs.



(a) *Représentation schématique du jeu* Blocks World Parallel *décomposé. Le joueur dispose d'actions composées qui lui permettent d'agir sur les deux sous-jeux simultanément.*



(b) *Représentation schématique du jeu* Dual Rainbow *décomposé. Chaque joueur joue dans son propre sous-jeu. Chaque sous-jeu est assimilable à un solitaire.*

FIGURE 8.5 – Exemple de *sous-jeux parallèles synchrones* solitaires.

Si le jeu complet est un solitaire, il comporte nécessairement des *actions composées* ce qui le rend difficile à décomposer (fig.8.5a). C'est le cas par exemple des jeux *Asteroids Parallel*, *Blocks World Parallel* ou *Joint Buttons And Lights* qui combinent respectivement deux parties de *Asteroids*, deux parties de *Blocks World* et trois parties de *Buttons And Lights*.

En revanche, dans le cas d'un jeu global multijoueurs, Chaque joueur joue dans son propre sous-jeu, sans aucune interaction entre les différents sous-jeux (fig.8.5b). C'est le cas par exemple, des jeux

Dual Hamilton, *Dual Hunter* et *Dual Rainbow* qui associent deux jeux solitaires de *Hamilton*, *Hunter* ou *Rainbow*. La décomposition et la résolution de ces sous-jeux ne présente aucune difficulté particulière grâce à la parfaite séparation des actions associées aux sous-jeux.

Les sous-jeux solitaires synchrones sont généralement identiques. Cependant, il est tout à fait envisageable de créer ce type de jeu composé avec des sous-jeux distincts. Il suffit de s'assurer que les différents sous-jeux ont la même durée, éventuellement grâce à l'ajout d'un *stepper*. Le jeu *Hodge Podge* par exemple, possède des sous-jeux *parallèles synchrones asymétriques*. Il regroupe, comme *Incredible*, un jeu de *Maze* et un jeu de *Blocks World*, à la différence que chaque sous-jeu est joué par un joueur différent.

8.5.2 Sous-jeux multijoueurs

Les jeux à sous-jeux multijoueurs synchrones, proposés dans les dépôts, sont généralement des jeux *doubles* composés de deux sous-jeux joués chacun par deux joueurs. Bien que les dépôts ne proposent pas de jeux *triples* ou plus, il est envisageable de construire des jeux composés de N sous-jeux (identiques ou différents) avec un nombre variable de joueurs, sur le même principe. L'alternance des actions dans les jeux comportant des sous-jeux multijoueurs synchrones peut se faire de différentes manières. Les joueurs peuvent changer de sous-jeu à chaque tour (sous-jeux à coups alternés) ou bien utiliser des *actions composées* (alternées ou simultanées).

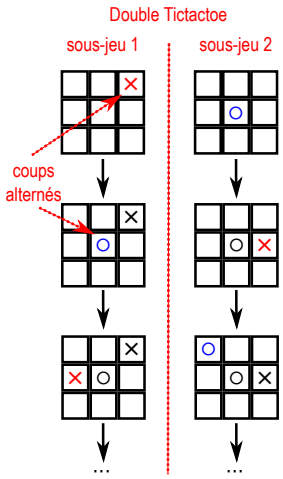
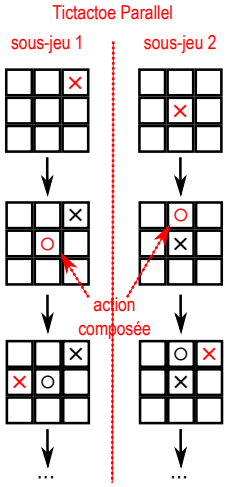
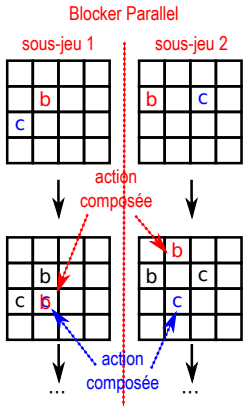


FIGURE 8.6 – Exemple de *sous-jeux parallèles synchrones à coups alternés* (sans action composée) : représentation schématique du jeu *Double Tictactoe* décomposé. Chaque joueur commence à jouer dans un sous-jeu distinct puis réplique dans l'autre sous-jeu. Chaque sous-jeu est assimilable à un jeu à coups alternés.

Dans les jeux *Double Tictactoe* ou *Dual Connect Four* chaque joueur choisi sa première action dans un sous-jeu distinct (fig.8.6). Au tour suivant, chaque joueur réplique dans l'autre sous-jeu et ainsi de suite. Dans un sous-jeu donné les actions des joueurs alternent régulièrement. Il faut noter cependant que chaque mouvement joint dans le jeu global ne contient qu'une seule action appartenant au sous-jeu. Comme l'action de l'autre joueur ne concerne pas ce sous-jeu et n'a pas d'effet sur les fluents le constituant, elle peut être considérée comme une action *noop* vis à vis de ce sous-jeu. Remarquons que *Dual Connect Four* présente la particularité d'associer un jeu de *Connect Four* classique avec sa version *suicide*. Les deux sous-jeux ne diffèrent que par le but du jeu qui est inversé : dans la version *suicide*,



(a) *Représentation schématique du jeu* Tictactoe Parallel *décomposé. À chaque tour, le joueur qui a la main joue une action composée qui a pour effet de placer une marque dans chacune des deux grilles.*



(b) *Représentation schématique du jeu* Blocker Parallel *décomposé. À chaque tour, les deux joueurs jouent deux actions composées simultanées. Les marques du joueur* blocker *sont représentées par les* b *bleus, et les marques de* crosser *sont représentées par des* c *rouges.*

Figure 8.7 – Exemple de jeu multijoueurs à sous-jeux asynchrones.

celui qui aligne 4 pions a perdu. Cette différence implique deux stratégies distinctes pour la résolution des sous-jeux.

Dans les jeux *Tictactoe Parallel* (fig.8.7a) et *Blocker Parallel* (fig.8.7b), les joueurs agissent sur les deux sous-jeux en même temps, il utilisent donc des *actions composées*. Dans *Tictactoe Parallel* les *actions composées* sont alternées pour permettre de jouer simultanément les deux jeux de *Tictactoe* qui sont à coups alternés. Dans *Blocker Parallel*, les *actions composées* sont simultanées pour respecter la simultanéité des coups dans chaque jeu de *Blocker*.

Une fois décomposés, tous les sous-jeux *parallèles synchrones* se réduisent à des types de jeux connus : jeux solitaires, jeux à coups alternés ou simultanés. Leur résolution ne présente donc pas de difficulté particulière. L'exploration séparée des sous-jeux simplifie significativement la résolution du jeu global.

8.6 Sous-jeux parallèles asynchrones

Dans un jeu à *sous-jeux parallèles asynchrones*, à chaque tour un des joueurs choisit dans quel sous-jeu il souhaite agir, pendant que l'autre joueur joue une action *noop* (*Double Tictactoe Dengji*, *Snake Parallel*). Ainsi, la situation peut évoluer dans l'un des sous-jeux tandis qu'elle reste identique dans l'autre sous-jeu : les deux sous-jeux n'évoluent pas au même rythme (fig.8.8). Ces jeux peuvent posséder des *actions composées*. Celles-ci n'agissent pas sur la totalité des sous-jeux ce qui permet à ces derniers d'évoluer de manière asynchrone (*Lights On Simultaneous*). Ce type de jeu peut également être à actions simultanées. Il n'existe pas actuellement de description correspondant à ce dernier cas mais on peut imaginer, par exemple, un jeu de *Double Blocker*, les deux joueurs jouant simultanément dans le sous-jeu de leur choix.

Bien qu'il existe des jeux en série assimilables à des *jeux multiples*, ces derniers sont en majorité composés de sous-jeux asynchrones (*Multiple Hamilton*, *Multiple Tictactoe*, ...). Dans les jeux *multiples asynchrones*, le joueur programmé devant choisir une action parmi celles de tous les sous-jeux, l'identification des *sous-jeux inutiles* simplifie significativement la recherche.

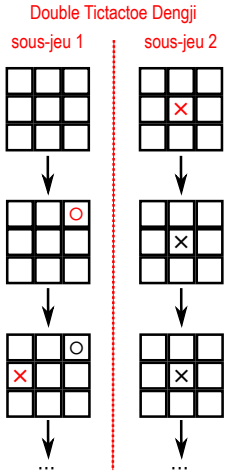


FIGURE 8.8 – Exemple de jeu multijoueur à sous-jeux asynchrones : représentation schématique du jeu *Double Tictactoe Dengji* décomposé. À chaque tour, le joueur qui a la main, choisit une action parmi toutes les actions légales des deux sous-jeux. Il joue donc dans un seul des sous-jeux.

Les sous-jeux solitaires asynchrones ne sont pas systématiquement identiques. Par exemple, *Incredible* regroupe une partie de *Maze* et une partie de *Blocks*, ainsi qu'un ensemble d'actions de contemplation. Ce jeu présente une difficulté particulière due au fait que le joueur doit sortir du labyrinthe (*Maze*) en ayant résolu le sous-jeu *Blocks* pour obtenir la totalité des points. Dans le cas contraire, la victoire dans le sous-jeu *Maze* met fin prématurément au jeu avec seulement une partie des points. La résolution du jeu décomposé nécessite de gérer ce problème pour ne pas bloquer la recherche dans un maximum local.

Les jeux multijoueurs à sous-jeux parallèles asynchrones proposés dans les dépôts sont eux composés de sous-jeux identiques. Il est cependant envisageable de créer des jeux de ce type avec des sous-jeux différents. Comme les différents joueurs peuvent jouer indifféremment dans un sous-jeu ou un autre, à l'intérieur d'un des sous-jeux, il n'est pas possible de prévoir l'alternance des joueurs. Tous les enchaînements possibles d'actions d'un ou plusieurs joueurs doivent être envisagés ce qui rend l'exploration de chaque sous-jeu plus complexe. Réunir les résultats obtenus pour chaque sous-jeu pour obtenir une évaluation du jeu global est également un problème particulièrement difficile.

8.7 Jeux impartiaux décomposables

Certains jeux impartiaux débutant, comme *Nim*, avec plusieurs piles ou groupes d'objets peuvent également être considérés comme des jeux composés (*asynchrones*) puisqu'ils peuvent être décomposés en plusieurs sous-jeux, un pour chaque pile/groupe, chacun d'eux étant un jeu impartial [Zhao, 2009].

La décomposition de ces jeux ne présente pas de difficulté particulière, les sous-jeux étant indépendants. En revanche, il convient d'identifier que ces sous-jeux sont impartiaux pour permettre ensuite

d'utiliser les techniques connues [Berlekamp *et al.*, 1982; Conway, 2000] pour la résolution du jeu global. Remarquons que tous les jeux impartiaux ne peuvent pas être décomposés, par exemple, *Chomp* qui n'est constitué que d'un seul groupe d'objets n'est pas considéré comme un jeu décomposable.

8.8 Sous-jeux terminaux et/ou influençant le score

Les jeux décrits en GDL disposent de règles logiques distinctes pour décrire les conditions mettant fin à la partie et les conditions d'obtention de chaque score. La fin d'un jeu ne correspond pas forcément à l'obtention du score maximum par l'un des joueurs. Par exemple, dans le jeu solitaire *Incredible*, l'achèvement du sous-jeu *Maze* met fin à la partie même si le joueur n'a pas obtenu le score maximal dans le reste du jeu. Cette particularité représente une difficulté propre au GDL pour la résolution des jeux décomposés.

La règle de conclusion `terminal` indique si l'état courant du jeu est une fin de partie. Les fluents présents dans le corps de cette règle sous forme normale disjonctive, constituent les prémisses nécessaires à cette terminaison. Si un de ces fluents fait partie de l'ensemble des fluents décrivant l'état d'un sous-jeu, alors ce sous-jeu est impliqué dans l'évaluation de cette terminaison. Nous dirons que *ce sous-jeu est terminal*. Au contraire, si aucun des fluents décrivant l'état d'un sous-jeu est une prémisse de `terminal` alors le sous-jeu est *non-terminal*. Similairement, s'il existe un fluent décrivant l'état d'un sous-jeu, tel que ce fluent est une prémisse d'une des règles `goal` permettant d'évaluer le score, alors ce sous-jeu est considéré comme *influençant le score*. Dans le cas contraire il ne l'influence pas.

Suite à la décomposition, un sous-jeu peut ne pas être *terminal*. Il convient alors de mettre fin à une partie au bout d'un certain nombre de *steps* de manière arbitraire, ou en évaluant la durée maximale d'une partie grâce aux autres sous-jeux.

Un sous-jeu *terminal* peut ne pas *influencer le score*, dans ce cas, une exploration d'un tel sous-jeu peut se résumer à estimer la durée d'une partie. En revanche s'il influence le score, il faut veiller à ne pas atteindre une position terminale avant d'avoir obtenu le score maximal dans les autres sous-jeux.

Un sous-jeu non terminal qui n'influence pas le score est considéré inutile. Dans le cas de sous-jeux séquentiels, si ce sous-jeu précède un autre *sous-jeu utile*, il est néanmoins nécessaire d'y jouer. Dans ce cas les actions peuvent être choisies aléatoirement puisqu'elles n'ont pas d'influence sur le score ou la terminaison du jeu.

La décomposition peut donc faire apparaître des sous-jeux terminaux ou non, influençant le score ou non, chaque combinaison nécessitant une approche spécifique pour l'exploration des sous-jeux.

8.9 Jeux décomposables en cours de partie

Certains jeux non-décomposables peuvent le devenir en cours de partie. Cox *et al.* [2009] proposent comme exemple, deux jeux de *Tictactoe* reliés par une case centrale (fig.8.9) permettant d'effectuer des alignements supplémentaires de trois marques, horizontaux uniquement. Ce *Joined Tictactoe* devient décomposable en deux jeux distincts à partir du moment où la case centrale est marquée par un joueur et les cases adjacentes sont marquées par l'autre joueur, interdisant ainsi tout alignement horizontal

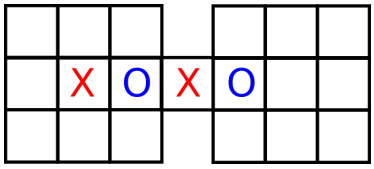


FIGURE 8.9 – Joined Tictactoe

utilisant la case centrale. Le jeu est alors coupé en deux et il est possible d'explorer chacune des parties indépendamment.

On peut se trouver dans une situation similaire sans que le jeu soit nécessairement constitué de deux jeux connus reliés entre eux. C'est le cas pour *Domineering* par exemple. Les dominos placés peuvent former une frontière séparant le plateau de jeu en différentes zones. Pour détecter cette situation et profiter de la décomposition pour jouer plus efficacement, il convient d'identifier les *latches* i.e. les fluents qui ne changent qu'une fois de valeur et gardent ensuite leur nouvelle valeur pour le reste de la partie. Ces fluents n'étant plus jamais affectés par les actions, ils peuvent être ignorés, coupant ainsi le jeu en sous-jeux distincts.

Chapitre 9
Invariants, symétries et décomposition

Sommaire

Bien que l'importance de ce problème ait été soulignée par Genesereth *et al.* [2005], peu de recherches concernent spécifiquement la décomposition des jeux GGP. La simplification d'un problème

implique la détection de structures, ce qui peut passer par la décomposition du problème en sous-parties, mais aussi par la recherche d'éléments symétriques qui peuvent être factorisés. La recherche de symétries peut permettre de détecter deux sous-jeux identiques. Décomposition et identification de symétries sont donc liés.

L'identification de symétries et la décomposition sont deux problèmes qui ont fait l'objet de recherches dans le domaine de la planification (*hierarchical*, *factored* et *localized planning*), de la satisfiabilité booléenne (SAT), de la satisfaction de contraintes (CSP) et de la vérification de modèles (*model checking*).

En planification, un problème est représenté sous forme d'un système de transitions comprenant un état initial défini par un ensemble de propositions vraies, un ensemble d'opérateurs (actions) permettant de modifier l'état courant et un ensemble d'états finaux (partiellement décrits) indiquant le but à atteindre. Les opérateurs sont définis par un ensemble de préconditions, propositions qui doivent être vraies pour que l'opérateur puisse être utilisé, et un ensemble d'effets positifs et/ou négatifs obtenus par l'application de l'opérateur. Ces effets sont présentés sous la forme d'une liste de fluents vrais à ajouter ou retrancher de la position courante. La résolution d'un problème de planification consiste à trouver la séquence minimale d'opérateurs à activer pour passer de l'état initial à un état final. Un problème de planification est donc très similaire à un jeu à un seul joueur.

Un problème SAT est représenté par une formule constituée d'un ensemble de clauses sous forme normale conjonctive (CNF) représentant un ensemble de contraintes à respecter. Chaque clause comprend un ensemble de littéraux positifs ou négatifs. La résolution d'un problème SAT consiste à trouver tous les ensembles de valeurs binaires associées aux littéraux qui permettent de satisfaire toutes les contraintes i.e. l'ensemble des clauses. Dans un CSP, les contraintes sont représentées par des formules plus variées (opérateurs binaires, opérateurs mathématiques, implications logiques, quantificateurs existentiels, etc.) en fonction de variables qui peuvent être instanciées dans un domaine de N valeurs. La résolution du CSP consiste également à trouver les instanciations des variables permettant de satisfaire les contraintes. Les problèmes SAT et CSP sont des problèmes logiques sans notion de temps ou de transition [48]. Cependant, la procédure de résolution SAT peut être représentée par un arbre de recherche, chaque nœud représente une instanciation partielle, un état intermédiaire dans une résolution progressive du problème. Il est également possible de représenter un système de transitions sous la forme d'un problème SAT ou CSP en dupliquant l'ensemble des règles décrivant le problème pour chaque temps T et en ajoutant une composante temporelle à chaque proposition. Cette approche est utilisée en planification pour profiter de la performance des solveurs SAT. Piette [2016] montre qu'une représentation sous forme de CSP peut être utilisé en GGP pour interpréter les règles. La recherche de symétries et la décomposition peuvent être opérées par identification de structures dans ces représentations SAT et CSP.

Dans le domaine de la vérification de modèles, le modèle est la représentation d'un système ou d'un programme dont on souhaite vérifier certaines caractéristiques. Ce modèle à vérifier est un automate à

48. Un problème SAT ou CSP est un problème statique qui consiste à attribuer les bonnes valeurs à un ensemble de variables, de manière à satisfaire un ensemble de contraintes. L'état du problème n'est pas modifié par l'application d'opérateurs ou d'actions. Par opposition, un jeu GGP ou un problème de planification est un problème dynamique assimilable à un système de transitions en temps discret. La résolution de ce problème consiste à trouver la séquence de transitions permettant d'atteindre un état final donné.

états finis $M = (S, R, L)$ ou S est un ensemble d'états, R un ensemble de transitions d'un état à un autre et L une fonction qui associe à chaque état un ensemble de propositions. Ce modèle est généralement nommé *structure de Kripke*[49]. Chaque règle est constituée d'une expression indiquant les prémisses et d'une expression indiquant le nouvel état. Les règles peuvent contenir des propositions et des variables associées à une valeur donnée. Le nouvel état peut être non-déterministe et, dans ce cas, un ensemble de valeurs possibles est indiqué pour les variables. Chaque état est décrit par l'ensemble des propriétés vraies pour les différents processus et l'ensemble des variables associées à une valeur. La caractéristique à vérifier est énoncée sous forme d'une formule. La formule peut indiquer un état que l'on souhaite éviter (*safety property*) comme un interblocage ou un plantage du système, ou bien indiquer une caractéristique souhaitée, par exemple vérifier qu'un message arrive toujours à destination (*liveless property*). Comme pour le GGP, l'exploration de tous les états possibles du modèle n'est pas envisageable et la détection de structures permettant de simplifier le problème, comme des symétries, est souhaitable.

Dans tous ces domaines, décomposition et identification de symétries amènent deux questions : comment identifier automatiquement les sous-structures et les éléments symétriques ? Comment exploiter décomposition et symétries pour résoudre le problème plus efficacement ? Dans ce chapitre, nous nous intéressons tout particulièrement aux travaux portant sur la détection automatique de sous-structures et d'éléments symétriques[50].

Nous présentons tout d'abord différentes techniques de détection des invariants dans le domaine de la planification et du GGP qui permettent d'inférer des types d'objets nécessaires à la détection de symétries. Nous abordons ensuite différentes méthodes utilisées pour la détection de symétries dans les domaines de la planification, SAT, CSP, vérification de modèles et GGP. Nous examinons les approches utilisées pour la décomposition appliquée aux SAT, CSP et à différents domaines de la planification. Nous détaillons les rares approches proposées pour la décomposition des jeux. Nous indiquons comment les décompositions obtenues peuvent être exploitées pendant l'exploration des jeux. Finalement, nous exposons brièvement une méthode de simplification des règles qui peut s'apparenter à une décomposition.

9.1 Détection d'invariants et types d'objets

En planification comme en GGP, les invariants sont des contraintes, exprimées sous forme de formules logiques, qui sont respectées dans tous les états qu'il est possible d'atteindre à partir de l'état initial d'un problème. La détection d'invariants est une étape préliminaire pour certaines approches de détection de symétrie ou de décomposition et constitue un problème en soit.

Les invariants détectés permettent d'inférer des structures de haut niveau comme des types d'objets en planification, ou bien des pièces mobiles, des coordonnées de positions et le statut de cases en GGP. Les invariants de type *exclusion mutuelle* servent à éliminer de l'espace de recherche des états qui ne

49. La structure de Kripke est un modèle de calcul, proche d'un automate fini non déterministe, inventé par le philosophe et logicien américain Saul Kripke en 1960. Il est utilisé pour représenter des systèmes de logique temporelle et plus précisément de logique modale.

50. Dans le domaine du GGP, une autre manière de faire apparaître des sous-structures passe par la factorisation logique des règles ; cette méthode de simplification des règles peut s'apparenter à une décomposition.

sont pas le résultat de l'application des opérateurs (*resp.* actions) en planification (*resp.* en GGP).

Nous commençons par exposer les différentes méthodes de détection des invariants en planification. Nous présentons ensuite un outils permettant d'inférer des types d'objets à partir de ces invariants. Enfin, nous détaillons une technique de recherche d'invariants utilisée en GGP.

9.1.1 Recherche d'invariants en planification

Le planificateur *GraphPlan* [Blum et Furst, 1997] utilisent des invariants sous la forme de relations d'exclusion mutuelle (*mutex*) entre les propositions : une seule proposition parmi celles d'un ensemble *mutex* est vraie dans un état du problème. Les opérateurs sont signalés comme ayant des *besoins concurrents* si leurs préconditions sont mutuellement exclusives, ou sont signalés comme étant en *interférence* si l'un d'eux supprime une précondition nécessaire à l'autre. Les relations sont propagées en fonction des opérateurs dans un graphe de planification à partir de la position initiale. Les informations sur les propositions mutuellement exclusives permettent d'éliminer les positions précédentes impossibles lors de la recherche arrière effectuée par *GraphPlan*. La détection est néanmoins limitée à une partie des relations d'exclusion mutuelles. Ils indiquent que déterminer l'intégralité de ces relations est aussi complexe que la recherche d'une solution au problème de planification.

Gerevini et Schubert [1998] proposent le système *DISCOPLAN* (*DIScovering COnstraints for PLANning*) pour détecter des invariants et améliorer l'efficacité des logiciels de planification fondés sur les solveurs SAT. Ils distinguent trois types de contraintes : des relations d'implication i.e. un littéral vrai implique qu'un autre soit vrai dans le même état, des contraintes de valeur unique assimilables à des *mutex* et des contraintes sur le domaine de valeurs des prédicats. Les invariants potentiels des deux premiers types sont extraits de la description de chaque opérateur et leur exactitude est vérifiée par l'analyse de ces derniers. Les contraintes de domaines sont extraites d'un graphe de planification, tel qu'utilisé par *GraphPlan*, sans les relations d'exclusion mutuelle (*forward action graph*). À chaque strate correspondant à un temps t dans le graphe, les valeurs possibles des propositions permettent d'inférer le domaine de valeurs des prédicats présents dans les préconditions ou effets des opérateurs. Leur méthode permet d'extraire ces invariants dans des problèmes de planification de type STRIPS en temps polynomial. Des clauses dérivées de l'instanciation des contraintes extraites sont utilisés pour enrichir la description de problèmes difficiles pour les solveurs *Walksat* et *Relsat* : les problèmes ainsi enrichis sont résolu significativement plus vite.

Bonet et Geffner [2001], toujours dans le contexte de la planification STRIPS, proposent de limiter la recherche d'invariant aux *mutex*. Ces derniers sont utilisés pour éliminer les états *fallacieux* générées lors d'une recherche par régression i.e. une recherche inférant des états précédents possibles à partir d'un état donné et des opérateurs ayant pour effet une des propositions décrivant cet état. Les *mutex* potentiels sont inférés d'après les préconditions et effets des opérateurs, puis vérifiés au regard de tous les opérateurs. Les *mutex* contredits sont itérativement éliminés. Leur approche permet de détecter des *mutex structurels*, par opposition aux *mutex* issus de relations temporelles détectés par *GraphPlan*. Cette approche est complémentaire à *GraphPlan*. L'information concernant les *mutex* permet au logiciel de planification par régression *HSPr*, de surpasser le logiciel *HSP2* de planification en avant sur certains problèmes. Cependant, le mécanisme de détection des *mutex* ne les détecte pas tous ; des états *fallacieux*

ne sont pas reconnus comme tels ce qui provoque dans certains cas surcoût en recherche inutiles.

Helmert [2009], dans le cadre de la transformation d'un problème PDDL en une représentation instanciée plus concise, propose de détecter des *invariants numériquement monotones* plutôt que des *mutex*. L'invariant est une proposition sous forme d'un prédicat dont un des arguments est fixé : pour une valeur donnée de l'argument, le nombre d'instances vraies de la proposition dans un état est constant. En limitant ce nombre à 1, cet invariant est assimilable à un *mutex*. Cependant, cette définition permet de faire abstraction de l'état initial pendant la détection. Les candidats invariants potentiels sont extraits et vérifiés par l'analyse des opérateurs. Les invariants non vérifiés i.e. dont nombre d'instance varie, sont transformés par l'ajout d'atomes, extraits de l'analyse de ces opérateurs, pouvant restaurer l'équilibre du nombre d'instances. Par exemple, dans un puzzle comme *Blocks World* si le nombre d'instances de `X` est variable pour le prédicat *table(X)* d'un état à l'autre, on peut lui associer *on(X, Y)* et prouver que le nombre d'instances d'objets posés sur la table ou sur un autre objet est fixe. Cette approche permet d'identifier rapidement des invariants utiles pour opérer la transformation souhaitée du problème PDDL, mais les relations d'exclusion mutuelle détectées se limitent aux propositions ayant la syntaxe particulière recherchée. Helmert montre que les invariants extraits présentent d'intéressantes propriétés de monotonicité qui ne sont pas couvertes par les *mutex*. Il indique que son approche présente l'avantage de ne pas être limitée au domaine de planification STRIPS. Contrairement à DISCOPLAN qui corrige tous les invariants contredits en même temps, cette approche les corrige dès qu'un opérateur les contredit, ceci permet à l'algorithme de terminer en moins d'une seconde sa recherche sur toutes les tâche du benchmark IPC tandis que DISCOPLAN dépasse les 30 minutes sur certaines tâches difficiles.

Rintanen [1998] propose une méthode de détection d'invariants plus génériques exprimés sous forme de disjonctions de 2 littéraux. Toutes les clauses de 2 littéraux, vraies dans l'état initial, sont générées et vérifiées par application itérative des opérateurs à partir de la position initiale. Les invariants contredits sont éliminés progressivement. Une généralisation de cette approche [Rintanen, 2000] consiste à remplacer les invariants contredits par des invariants plus faibles i.e. satisfaits dans un plus grand nombre de positions. Les invariants produits sont limités à n littéraux pour limiter le temps de calcul. Rintanen [2008] propose un algorithme plus simple utilisant le principe de régression. Une modélisation SAT est utilisée pour générer une position satisfaisant tous les invariants sauf un que l'on souhaite vérifier. Si une des positions précédentes, possible en fonction des opérateurs, satisfait tous les invariants, l'invariant non respecté dans la position courante est remplacé par un invariant plus faible. En cas d'interaction entre eux, il n'est pas possible d'inférer un invariant puis de l'utiliser pour en inférer d'autres. La vérification proposée de tous les invariants en parallèle permet d'éviter ce problème. Rintanen indique que, contrairement aux approches précédentes, cette approche n'est pas limité aux invariants de formes syntaxiques particulières et fonctionne uniformément pour tous les invariants. Cependant, le temps de calcul s'accroît rapidement quand la taille des invariants augmente, la taille de ceux-ci doit donc être limitée à 2 ou 3 littéraux. Les techniques antérieures établissent chaque invariant séparément et échouent à produire les invariants que cet algorithme produit.

9.1.2 Inférence de types d'objets

Afin de permettre la détection de symétries dans son logiciel de planification *STAN*, Fox et Long [1998] proposent un module d'inférence automatique des invariants d'état (*State space invariants*) nommé *TIM* (*Type Inference Module*) qui est conceptuellement très différent des approches précédentes. A partir des opérateurs, un ensemble de règles de transition sont inférées. L'analyse de ces règles permet d'identifier des propriétés i.e. des invariants représentant le statut d'un objet dans un état donné, et des attributs i.e. différentes caractéristiques d'un objet pouvant coexister dans un même état. Les objets utilisés dans le problème de planification sont alors associés à un type en fonction des propriétés et attributs qu'ils peuvent posséder dans un des états. *TIM* est alors en mesure de produire une description PDDL d'un problème où tous les objets sont étiquetés de manière à identifier des entités du même type. Par exemple dans un problème de fret (Rocket), les types peuvent regrouper les objets en trois groupes : les lieux, les colis, les fusées. Les résultats montrent que le temps de résolution de *STAN* avec *TIM* sur un problème de fret comportant de plus en plus de lieux, de colis et de fusées, augmente linéairement tandis que *STAN* sans *TIM* suit une courbe cubique.

9.1.3 Recherche d'invariants en GGP

Haufe *et al.* [2012] proposent une transposition de la recherche d'invariants pour le GGP utilisant ASP pour effectuer des déductions. Ces invariants permettent d'inférer des connaissances de haut niveau sur le jeu comme la présence d'un plateau de jeu, de pièces, de marques d'un compteur (score, temps) évoluant de manière monotonique, etc. Ces connaissances sont prouvées de manière logique plutôt que testées par simulation comme dans les méthodes heuristiques (cf. §3.4).

Leur approche consiste à tirer parti de la syntaxe des fluents pour poser des hypothèses du type : "Dans les fluents du type `(cell ?x ?y ?s)` du jeu *Tictactoe*, `?s` représente un statut". Cette hypothèse peut être formulée sous forme d'un invariant : "dans une position, il n'existe qu'une seule instance de `(cell ?x ?y ?s)` pour une valeur donnée de `?x` et `?y`".

Pour prouver un invariant via un raisonnement par récurrence (si l'hypothèse est vrai au temps T, elle sera vraie à $T + 1$), ils proposent d'effectuer une réécriture des règles pour ASP en les enrichissant d'une composante temporelle [Thielscher, 2009]. Une règle GDL comme

```
(<= (next (cell ?x ?y o)) (does oplayer (mark ?x ?y o))
                          (true (cell ?x ?y b)))
```

devient en ASP

```
true1(cell(X, Y, o)) :- does0(oplayer, mark(X, Y)),
                        true0(cell(X, Y, b)).
```

Les règles peuvent être dupliquées pour chaque temps T de 0 à N, pour prouver une hypothèse portant sur une séquence de N transitions. Afin de permettre l'instanciation des règles par ASP, les instances possibles des fluents et actions (`input` et `base`) sont ajoutées à la description du jeu.

Un générateur d'actions permet à ASP de tester les différentes transitions possibles. Pour chaque joueur, ce générateur indique qu'une et une seule action `does`, instanciée d'après les `input`, doit être

vraie au temps T (est choisie par le joueur) à condition que la position ne soit pas terminale au temps T. Une contrainte est ensuite ajoutée pour indiquer qu'une position contenant une action non légale est à exclure. Voici par exemple le générateur d'actions pour la première transition :

```
terminated0 :- terminal0.
1{ does0(R,A) : input(R,A) }1 :- role(R), not terminated0.
:- does0(R,A), not legal0(R,A).
```

Pour prouver qu'un invariant est vrai pour toute une séquence de N transitions depuis la position initiale, la position T_0 est définie comme l'ensemble des fluents `true0` correspondant aux instances du prédicat `init` :

```
true0(X) :- init(X).
```

Pour prouver cet invariant depuis n'importe qu'elle position du jeu, ASP doit pouvoir générer toutes les positions T_0 possibles. La règle précédente est alors remplacée par un générateur de positions, déclarant comme vrai un ensemble aléatoire de fluents `true0`. Ces fluents `true0` sont choisis d'après les instances du prédicat `base`. Cet ensemble aléatoire peut être l'ensemble vide. Des contraintes sont ajoutées pour exclure les positions violant l'invariant considéré. Par exemple, pour prouver que le paramètre `?s` représente le statut d'une case, on peut générer toutes les positions et exclure celles où il n'y a pas une et une seule instance du fluent :

```
0{ true0(F) : base(F) }.
:- not 1{ true0(cell(X,Y,_)) }1, base(cell(X,Y,_)).
```

Enfin l'hypothèse elle même est formulée sous forme d'une affirmation décrivant l'invariant. Par exemple, on peut affirmer que pour une valeur de `?x` et `?y` il n'y a qu'une seule instance de `(cell ?x ?y ?s)` au temps T_0 et T_1 :

```
hypothese :- 1{ true0(cell(X,Y,_)) }1,
             1{ true1(cell(X,Y,_)) }1, base(cell(X,Y,_)).
```

Pour limiter le temps de calcul, plusieurs invariants sont encodés dans le problème en ASP et vérifiés simultanément [51]. Différents prétraitements du GDL, notamment l'élimination des variables existentielles [Schiffel, 2016], permet de limiter le nombre de règles lors de l'instanciation.

Pour conserver un temps de calcul raisonnable, seules des hypothèses formulables sous forme d'invariants et limitées en nombre peuvent donc être prouvées. Les hypothèses formulables sont dépendantes de la syntaxe des règles GDL. Si le statut des cases de *Tictactoe* est encodé sous forme de différents prédicats comme `(cellx 1 1)`, `(cellb 1 1)`, `(cello 1 1)`, la recherche se trouve entravée. Le cas extrême serait celui d'un jeu formulé uniquement avec des fluents sous forme d'atomes : `cellx-1-1`, `cello-1-1`, `cellb-1-1`. La recherche d'invariants reviendrait à chercher tous les fluents mutuellement exclusifs ce qui, comme l'indiquent Blum et Furst [1997], serait aussi complexe que la résolution du problème lui-même.

51. Chaque hypothèse à prouver étant un invariant i.e. un prédicat vérifié dans tous les ensembles solution, il est possible avec un solveur ASP comme *Clingo* [Gebser *et al.*, 2010] de faire l'intersection de toutes les solutions trouvées ce qui, en pratique, évite une explosion du temps de recherche. Si un invariant n'est pas dans une des réponses, il sera éliminé des réponses suivantes et si tous les invariants à prouver sont éliminés, la recherche s'arrête.

Pour finir, les invariants à prouver ne doivent pas dépendre d'autres invariants qui eux ne font pas l'objet d'une hypothèse. Les positions générées par ASP pour établir la preuve par récurrence étant aléatoires, il est possible qu'elles n'appartiennent pas aux positions accessibles depuis l'état initial du jeu. Par exemple, pour le jeu *Tictactoe*, on ne peut pas prouver qu'il est impossible d'avoir une case à la fois marquée d'un `x` et d'un `o` si la position générée par ASP indique que la case peut être à la fois blanche (`b`) et marquée d'un `o`.

Dans le jeu *Eight Puzzle*, certaines instances des règles ne sont jamais satisfiables i.e. leur conclusion est toujours fausse dans toutes les positions accessibles depuis la position initiale du jeu. La règle suivante :

```
(<= (next (cell ?u ?v ?z))
        (true (cell ?u ?v ?z))
        (does player (move ?x ?y))
        (or (distinct ?x ?u) (distinct ?y ?v))
        (true (cell ?x1 ?y1 b))
        (or (distinct ?x1 ?u) (distinct ?y1 ?v)))
```

correspond, entre autre, à l'instance suivante :

```
(<= (next (cell 1 1 b)) (true (cell 1 1 b))
    (does player (move 3 3)) (true (cell 3 3 b)))
```

Cette instance de la règle indique que la case $(1, 1)$ restera blanche s'il y a une autre case blanche en $(3, 3)$ et que le joueur joue à cet endroit. Ceci est parfaitement impossible puisqu'il n'y a qu'une seule case blanche dans le jeu dans la position initiale. Si tous les invariants concernant l'état `b` des cases de ce jeu ne font pas l'objet d'une hypothèse lors de la recherche, alors l'interprétation de ces règles fera disparaître l'ensemble des hypothèses d'invariance en laissant supposer la coexistence possible de plusieurs cases blanches et donc, ensuite, d'un déplacement possible des tuiles numérotées vers deux emplacements distinct au même moment.

9.2 Détection de symétries

La recherche de méthodes automatiques pour la détection de symétries est plus développée dans le domaine de la planification, qui nourrit un intérêt particulier pour les solutions générales indépendantes des problèmes, que dans les domaines de la vérification de modèles, de la satisfaction de contraintes (CSP) ou de la résolution SAT. Le domaine des CSP, par exemple, est plus concerné par la recherche de la meilleure manière d'encoder un problème pour mieux le résoudre. De fait, moins de travaux sont effectuées dans ces domaines sur la découverte automatique de symétries [Long et Fox, 2003]. S'il existe quelques travaux sur la détection automatique de symétrie en SAT, beaucoup de travaux sont fondés sur des symétries signalées *a priori* [Saïs, 2008].

On peut distinguer deux types de symétries traitées dans la littérature : les *symétries globales*, identifiées de manière statique d'après l'état initial d'un problème, et les *symétries locales*, identifiées dynamiquement en cours de résolution.

Long et Fox [2003] proposent un état de l'art de la recherche et de l'exploitation des symétries dans

le domaine de la planification et ils distinguent différents types de symétries. Premièrement, des travaux s'intéressent à l'identification de symétries résultant d'objets interchangeables, aussi nommée *équivalence fonctionnelle* [Fox et Long, 1999]. Ces objets sont utilisés pour identifier des plans équivalents [Fox et Long, 1999, 2002] ou pour éliminer des impasses équivalentes [Miguel, 2001]. Deuxièmement, des travaux portent sur l'identification de symétries plus complexes concernant des groupes d'objets [Joslin et Roy, 1997]. Troisièmement, il est possible de détecter dans les systèmes de transitions des *symétries de séquences d'actions* i.e. des actions dont l'ordre est interchangeable sans modification de la solution [Emerson et Sistla, 1996]. D'après Long et Fox [2003] les symétries de séquences d'actions exploitées par Rintanen [2003] relèvent de l'équivalence fonctionnelle. Selon Rintanen, son approche va au delà de l'équivalence fonctionnelle. Son approche est plus générale et permet permet une détection plus robuste des symétries locales par exploitation des *symétries de séquences de transitions*.

Dans les domaines de la planification, de SAT, de CSP et de la vérification de modèles, il est possible d'identifier globalement deux approches pour la détection des symétries. La première passe par une détection d'objets interchangeables fondée sur l'étiquetage des objets équivalents. La seconde permet de détecter davantage de symétries et des symétries plus complexes : le problème est dans un premier temps représenté sous forme d'un graphe et, dans un second temps, un algorithme de recherche d'isomorphisme [Darga *et al.*, 2008, 2004; McKay et Piperno, 2013] permet de détecter les symétries dans le graphe précédent. Cette idée de recherche d'isomorphismes sur un graphe a été transposée au GGP [Schiffel, 2010]. Transposé à un problème SAT, ce type de recherche peut être appliqué sur des expressions logiques [Saïs, 2008].

La recherche d'isomorphismes dans un graphe est un problème difficile dont la complexité est estimée entre P et NP [52] et la détection de toutes les symétries dans un problème donné est polynomialement équivalente à cette recherche d'isomorphisme [Crawford, 1992]. Cependant, différentes approches ont été proposées pour détecter une partie des symétries.

Nous présentons premièrement les méthodes fondées sur la détection d'objets équivalents, i.e. d'objets symétriques, en planification. Nous détaillons ensuite différentes approches pour la détection de symétries dans les problèmes SAT ainsi que les problèmes de planification encodés en SAT. Nous exposons les méthodes de recherche de symétries de variables et de valeurs dans le domaine des CSP. Nous indiquons comment les approches fondées sur les isomorphismes de graphes sont également utilisées dans le domaine de la vérification de modèles. Enfin, nous présentons les techniques de détection de symétries utilisées dans le domaine du GGP.

9.2.1 Détection d'objets équivalents

A partir de l'identification des types d'objets réalisée avec *TIM* [Fox et Long, 1998], des objets équivalents et donc symétriques peuvent être identifiés [Fox et Long, 1999]. Un problème encodé avec le

52. P et NP considère les problèmes de complexité temporelle polynomiale. P représente la classe des problèmes qui peuvent être résolus en temps polynomial par une machine déterministe. Un problème polynomial est considéré comme pouvant être résolu par un algorithme efficace. NP représente la classe des problèmes qui peuvent être résolus en temps polynomial par une machine non-déterministe i.e. une solution existante peut être vérifiée en temps polynomial mais on considère qu'il n'existe pas d'algorithme efficace pour trouver l'ensemble des solutions. (on nomme co-NP la classe des problèmes complémentaires). Les problèmes de complexité exponentielle en temps sont placés dans la classe EXPTIME s'ils peuvent être résolus par une machine déterministe et NEXPTIME s'ils peuvent être résolus par une machine non-déterministe.

langage STRIPS est préalablement étiqueté, ensuite les objets de même type avec le même état initial et final sont examinés. Si pour chaque opérateur utilisant un objet comme variable, il existe un autre opérateur symétrique, alors les deux objets sont considérés comme interchangeables. Cette approche, procédant par examen des règles, ne détecte que certains types de symétries particuliers. De plus, la détection est sensible à la formulation des règles et ne peut détecter que des objets représentés par des variables à l'intérieur des propositions décrivant l'état du problème. Les symétries sont utilisées pour élaguer les branches équivalentes pendant la recherche d'un plan. La recherche est effectuée avec le planificateur *STAN* (fondé sur *GraphPlan* [Blum et Furst, 1997]). Les opérateurs symétriques sont éliminés lors du choix d'une action. Cette approche à l'avantage de réduire l'examen des symétries à celles utiles au moment de leur exploitation. Cependant, tout objet modifié par un opérateur perd sa symétrie avec les autres : les opérateurs et objets ayant perdu leur symétrie sont stockés dans deux structures de données dédiées. Cette symétrie n'est restaurée qu'en cas de retour arrière pour explorer une autre branche. Il s'agit d'une symétrie des objets et non des états : à mesure que le plan est construit, de plus en plus de symétries sont éliminées et le gain d'une recherche fondée sur les symétries devient nul. De plus, le fait de chercher un plan par régression, i.e. en partant de la fin en remontant progressivement vers l'état initial, modifie le contexte dans lequel un groupe d'actions symétriques est évalué : en ne remettant pas en cause les coupes effectuées dans l'arbre, la solution peut-être sous-optimale.

Fox et Long [2002] proposent une amélioration de l'approche pour la détection dynamique des symétries : les structures de données précédentes sont utilisées différemment pour identifier les groupes d'objets symétriques et leur évolution. En déplaçant un objet d'un groupe de symétrie à un autre, il devient possible d'exploiter de nouvelles symétries apparaissant en cours de jeu.

Miguel [2001] propose une autre manière d'exploiter les objets symétriques identifiés. Lors d'un échec, un *no-good*[53] schématique est généré dans lequel chaque objet est remplacé par l'ensemble des objets symétriques. À chaque nouvel état atteint par régression avec *GraphPlan*, les *no-goods* pouvant potentiellement correspondre sont générés et comparés. Les propositions schématiques sont organisées par ordre lexicographique pour ne pas avoir à générer tous les *no-goods* symétriques. Si un *no-good* correspond, la branche est coupée ce qui diminue l'espace de recherche.

9.2.2 Symétries en SAT

Dans le domaine des solveurs SAT, la détection de symétries est majoritairement fondée sur l'identification d'automorphismes [Darga *et al.*, 2008, 2004; McKay et Piperno, 2013] d'un graphe modélisant le problème. Différents travaux proposent des approches pour la recherche de symétries au niveau global i.e. au départ de la recherche, ou local i.e. sur des états intermédiaires correspondant à des affectations partielles des variables.

L'idée de réduire la recherche de symétries à la recherche d'isomorphismes de graphe est suggérée par Crawford [1992]. Cette idée est développée par Aloul *et al.* [2002] qui proposent une approche pour détecter les symétries globales prenant également en compte les symétries par cycle de permutations.

53. Un *no-good* est un ensemble de propositions qui ne peuvent être vraies au temps T. Ce *no-good* caractérise un état qui ne peut être atteint et donne lieu à un élagage de l'arbre de recherche. Par exemple, si le problème consiste à transporter les objets X, Y et Z, un à la fois, de A à B, que tous les objets sont en A au *step* 0 et que `at(i,j)` est une proposition indiquant que i est à la position j, {at(X,B), at(Y,B), at(Z,B)} est un *no-good* au *step* 2.

Pour détecter les symétries, un graphe coloré non orienté représentant le problème SAT est construit de la manière suivante : un sommet est ajouté pour chaque littéral positif ou négatif et pour chaque clause non binaire sous forme normale conjonctive (CNF). Les littéraux et les clauses sont colorés différemment pour qu'une recherche d'automorphisme ne puisse faire correspondre les uns aux autres. Une arête est ajoutée entre les versions positive et négative d'un même littéral, ainsi qu'entre chaque clause et les littéraux qui la composent. Une arête double est ajoutée entre les littéraux d'une clause binaire. L'algorithme *NAUTY* [Darga *et al.*, 2004] est utilisé pour identifier les automorphismes de ce graphe.

Les symétries sont brisées par l'ajout de contraintes dans le problème initial. Ajouter une contrainte $(\neg x \vee y)$ permet de limiter l'exploration à une seule des affectations de x et y si ces deux variables sont symétriques. Ceci revient à imposer un ordre lexicographique sur les valeurs prises par les variables x et y [Crawford *et al.*, 1996]. Cependant, le nombre de contraintes ajoutées peut faire grossir la formule CNF de telle manière que l'effort nécessaire au solveur pour traiter cette formule surpasse le gain obtenu par l'élimination des symétries. Pour éviter cela, seule une partie des symétries est traitée. Aloul *et al.* [2002] limitent l'ajout de contraintes aux dix premiers couples de variables échangées de chaque symétrie. Ils proposent aussi, pour ajouter moins de contraintes, de ne considérer que les générateurs d'un groupe de symétries i.e. un ensemble minimal de symétries élémentaires permettant d'obtenir toutes les autres symétries par composition. Il est possible également d'éliminer les contraintes tautologiques ou redondantes [Aloul *et al.*, 2006].

Benhamou *et al.* [2010] proposent d'étendre la recherche automatique de symétries en détectant dynamiquement les symétries locales (en cours de résolution SAT). Un graphe identique au précédent est maintenu pendant la recherche en supprimant les littéraux assignés. Les automorphismes du graphe sont recalculés pour détecter de nouvelles symétries. Les symétries détectées sont, là aussi, utilisées pour couper des branches de l'arbre de recherche, mais via la détection de *no-good* symétriques. Si une branche qui affecte une valeur à un littéral mène à un échec (*no-good*), il est possible de couper toutes les branches qui affectent la même valeur aux littéraux symétriques. Cela revient, si un littéral l vrai rend la formule inconsistante, à revenir en arrière et mettre l à faux ainsi que tous les autres littéraux symétriques.

Rintanen [2003] propose une approche pour détecter et exploiter les symétries dans les systèmes de transitions encodés en logique propositionnelle et résolus avec les algorithmes SAT. L'approche qui consiste à imposer un ordre lexical sur les propositions décrivant les états est incompatible avec les systèmes de transitions, car les relations de transition ne préservent pas l'ordre lexicographique imposé. Il propose donc de ne pas ordonner les états mais les séquences de transitions. Les contraintes ajoutées pour briser la symétrie portent donc sur des propositions encodant les transitions du système. L'encodage d'un système de transitions en SAT se fait par l'ajout d'une composante temporelle dans les propositions décrivant un état du système. Certaines transitions sans aucune interaction entre-elles peuvent être testées en parallèle dans le problème SAT, indiquant que leur ordre dans le système de transitions d'origine n'est pas signifiant. L'élimination des transitions symétriques peut limiter ce parallélisme. Pour éviter cela une simple extension des contraintes permet d'exclure uniquement les transitions de plus haute priorité qui pourraient interférer avec elles. Le nombre de contraintes augmente de manière quadratique par rapport au nombre transitions, cependant l'extension des contraintes pour éviter de limiter le parallélisme, peut

rendre l'augmentation cubique dans le pire des cas. De plus, cette approche n'élimine pas forcement toutes les symétries. En outre, les expérimentations montrent que l'approche n'apporte un gain de temps de résolution que si les contraintes sont limitées aux paires d'opérateurs symétriques par échange d'un seul paramètre : l'extension des contraintes permet d'obtenir des solutions plus courtes, mais le temps de calcul est significativement plus long.

9.2.3 Symétries en CSP

Un système de transitions, comme celui d'un problème de planification peut également être encodé sous forme d'un problème de satisfaction de contraintes (CSP) par l'ajout d'une composante temporelle dans les propositions. Les symétries d'objets ou de groupes d'objets, représentées respectivement par des variables ou des propositions en planification, se transposent en symétrie de valeurs ou de variables en CSP.

Joslin et Roy [1997] proposent une approche pour détecter les symétries globales (éliminées à l'état initial) fondée sur la détection automorphismes dans un graphe coloré avec *NAUTY* [Darga *et al.*, 2004]. Le problème CSP est encodé sous forme d'axiomes quantifiés par rapport aux objets du domaine. Cette représentation de haut niveau présente déjà une forme d'abstraction qui rend la présence des symétries plus évidente et correspond à la représentation des problèmes de planification transposés en CSP. Des littéraux, sous forme de prédicats, expriment l'état des objets du problème et les valeurs des arguments utilisés dans ces prédicats correspondent aux objets. La symétrie détectée ici est une symétrie des valeurs qui correspond à l'interchangeabilité des objets. A partir des axiomes, un graphe coloré est constitué. Un sommet est ajouté pour chaque objet (valeur) avec une coloration correspondant à son type. Les objets instanciés dans la description de l'état initial sont distingués des autres et sont colorés différemment. Chaque prédicat (littéral) instancié est aussi représenté par un sommet avec des arêtes le reliant vers chaque valeur utilisée comme argument. Les littéraux de même prédicat ont la même couleur. Le programme *NAUTY* est utilisé pour la recherche d'isomorphismes dans ce graphe. Pour chaque *générateur* de symétrie identifié, une contrainte est ajoutée pour briser les symétries. Les contraintes sont exprimées de manière compacte pour limiter l'explosion de leur nombre. Joslin et Roy suggèrent la détection de symétries locales pour améliorer leur approche.

Roy et Pachet [1998] recherchent une symétrie des variables qui correspond à la situation où différentes variables remplissent le même rôle. Ils donnent l'exemple du problème des pigeons et pigeonniers : chaque pigeon est représenté par une variable parmi N et partage le même domaine de valeurs i.e. un des $N-1$ pigeonniers. Les variables sont ici symétriques, si une affectation de pigeonniers ne permet par de loger tous les pigeons, toutes les affectations symétriques peuvent être éliminées également. Ils proposent de détecter une classe particulière de symétries nommées *permutabilités en intention* à partir des contraintes non instanciées du CSP. Deux variables permutables doivent avoir le même domaine. Ces variables de même domaines sont placées dans un même ensemble. Ensuite, pour chaque contrainte, les opérateurs commutatifs permettent de déduire des ensembles de variables potentiellement *permutables en intention*. Les ensembles de variables symétriques sont identifiés en comparant les ensembles obtenus pour tous les opérateurs. Les symétries sont ensuite utilisées lors de la détection des *no-goods* pour élaguer la recherche : si une affectation partielle des variables contredit une contrainte, les affec-

tations symétriques peuvent être ignorées. Une généralisation de l'approche à des groupes de variables symétriques permet de détecter des structures permutables.

Puget [2005] propose une approche qui permet la découverte des symétries à la fois des variables et des valeurs d'après les isomorphismes d'un graphe coloré. Son graphe coloré comprend un sommet pour chaque variable associée à une valeur de son domaine. Ces sommets ont tous la même couleur ce qui permet la découverte des deux types de symétries. Des sommets sont également ajoutés pour chaque contrainte et chaque instance de cette contrainte. Les contraintes de même type ont la même coloration ; leurs instances ont une couleur propre, mais liée à celle de la contrainte correspondante. Des arêtes relient chaque contrainte à ses instances et chaque instance aux couples variables/valeurs utilisés. Différentes réécritures des contraintes permettent de limiter la taille du graphe généré et de mettre en évidence la symétrie de certains opérateurs, masquée par la formulation originale des contraintes. Malgré tout, seule des expressions simples sont prises en charge dans la recherche de toutes les symétries pour éviter un temps de calcul trop long. Les automorphismes du graphe sont détectés avec l'algorithme *AUTOM* fondé sur *NAUTY* et tirant parti de la faible connexité des graphes. L'approche détecte des symétries syntaxiques qui ne sont pas la combinaison de symétries de variables et de valeurs.

Mears *et al.* [2009] proposent une amélioration de l'approche permettant de prendre en charge tous les types d'expressions. Plutôt que d'utiliser une représentation en intention (non instanciée) qui nécessite d'utiliser beaucoup de variables intermédiaires et amène à la perte de certaines symétries, ils proposent d'utiliser une représentation en extension. Pour simplifier le graphe résultant, ils proposent d'éliminer du graphe toutes les sommets correspondant à des affectations incohérentes de variables et de propager ces simplifications de manière itérative. Ces simplifications ont en outre l'avantage de faire apparaître de nouvelles symétries. Cependant, elles ne sont pas assez importantes pour empêcher le graphe d'atteindre une taille trop importante. Cette approche permet de détecter plus de symétries et d'accélérer la résolution des CSP, mais elle reste impraticable pour les instances difficiles de CSP.

Ces différentes approches permettent la détection de symétries à partir de l'état initial du problème. Trouver des symétries dynamiquement pendant la résolution d'un CSP nécessite de ré-examiner les contraintes à chaque unification partielle des variables ce qui peut s'avérer très lourd. Saïs [2008] indique que la détection et l'exploitation des symétries locales d'un CSP est un problème difficile.

Cohen *et al.* [2005] proposent une autre classification séparant les symétries en deux groupes : les *symétries de solutions* i.e. les transformations qui préservent l'ensemble des solutions, et les *symétries de contraintes* i.e. des transformations qui préservent les contraintes et indirectement, donc, les solutions. Ils proposent un tour d'horizon des différentes définitions de la notion de symétrie. Pour eux, ces définitions diffèrent en fonction des transformations envisagées (permutation de valeurs, variables ou de paires variables/valeurs) et des éléments préservés par la symétrie (les contraintes ou l'ensemble des solutions). Ils insistent sur le fait que les symétries qui peuvent être détectées dépendent fortement du point de vue considéré : rechercher toute transformation qui préserve les solutions ou bien rechercher toute transformation qui préserve les contraintes, la préservation des solutions étant une conséquence indirecte. Les symétries de contraintes sont donc un sous-ensemble des symétries de solutions.

Pour détecter les symétries de contraintes, ils proposent d'utiliser la *microstructure complémentaire* du CSP, un hypergraphe dont chaque sommet représente un couple variable/valeur et chaque hyperlien

indique les affectations en conflit d'après une des contraintes du CSP. L'application d'un algorithme de recherche des isomorphismes dans ce graphe permet de mettre en évidence les symétries de contraintes. La microstructure est trop lourde à calculer pour des CSP importants avec des contraintes non-binaires. Ils indiquent que des solutions existent néanmoins pour représenter les contraintes de manière plus compacte.

Pour détecter les symétries de solutions, ils proposent d'utiliser un *hypergraphe k-nogood* comprenant tous les sommets et hyperliens de la *microstructure complémentaire* du CSP avec en supplément des hyperliens indiquant tous les *no-good* de taille $m \leq k$ i.e. toutes les instanciations de m variables qui ne peuvent être étendues à une solution. Les isomorphismes de cet *hypergraphe k-nogood* correspondent aux symétries de solutions. Ils indiquent que pour trouver toutes les symétries, il suffit de considérer les *no-goods* de taille 1 et 2. À partir d'un hypergraphe réduit à la *microstructure complémentaire* du CSP, ils proposent de construire dynamiquement l'*hypergraphe k-nogood* pendant la recherche. À chaque *no-good* rencontré, ce *no-good* et tous les *no-goods* symétriques (d'après les symétries déjà connues) sont ajoutés dans l'hypergraphe, ce qui permet de détecter de nouvelles symétries en recalculant les isomorphismes du graphe.

9.2.4 Symétries en Model Checking

Les automorphismes de graphe sont également utilisés dans le domaine de la vérification de modèles en logique temporelle (*Temporal Logic Model Checking*). En théorie, la découverte des automorphismes dans la *structure de Kripke* représentant le modèle à vérifier permet de réduire ce modèle à une structure plus simple (*quotient Kripke structure*). En pratique, la découverte des automorphismes sur un modèle n'est pas réalisable : la recherche d'automorphismes est trop complexe sur un modèle de grande taille et le modèle lui-même est de toute manière trop lourd à construire [Miller *et al.*, 2006].

Un sous-groupe d'automorphismes peut être calculé à partir des relations de communication. Emerson et Sistla [1996] proposent de construire un graphe dont les sommets représentent les processus décrits par le modèle et les liens indiquent la présence de variables partagées entre les processus. Le nombre de symétries détectées par la recherche d'isomorphismes dans ce graphe est inférieur ou égal au nombre de symétries du modèle. Cette approche est limitée à des processus de même type et des variables partagées au plus entre deux processus. Clarke *et al.* [1998] proposent de généraliser cette approche en utilisant un hypergraphe coloré. Les processus intervenant dans le modèle sont regroupés par type. Chaque processus est représenté par un sommet dans le graphe et reçoit une couleur correspondant à son type. Un hyperlien relie les processus partageant une même variable.

Un ensemble d'états symétriques forme une *orbite*. Pour exploiter les symétries obtenues pendant la vérification du modèle, il est nécessaire de pouvoir identifier que deux états sont symétriques : ce problème est nommé *problème de l'orbite* (*orbit problem*). Clarke *et al.* [1998] indiquent que ce problème est au moins aussi complexe que la recherche d'isomorphisme dans un graphe. Miller *et al.* [2006] présentent les différentes approches pour résoudre ce problème dans le domaine de la vérification de modèles. Ces solutions s'appuient sur le calcul d'états canoniques. L'exploitation des symétries peut être réalisée également via l'usage d'une structure de données spécifique, le *scalarset*, cependant les symétries sont alors manuellement spécifiées par l'utilisateur.

*

* *

Dans toutes les approches présentées, un compromis est nécessaire entre le nombre de symétries identifiées, le coût en calcul de leur exploitation (quantité de contraintes ajoutées en SAT ou CSP, problème de l'*orbite* en vérification de modèles) et le gain en temps obtenu pour la recherche. L'identification d'objets interchangeables en planification permet une élimination rapide et peu coûteuse de certaines symétries spécifiques. Les approches fondées sur les isomorphismes de graphe permettent de détecter plus de symétries, mais représentent un coût plus important en calcul.

Les différentes approches sont toutes tributaires de la formulation du problème : une asymétrie superflue peut entraver la détection. Les différents problèmes traités en planification, SAT, CSP et vérification de modèles sont décrits sous une forme abstraite qui restreint le domaine à un ensemble d'objets limité. C'est cette absence de détails qui permet l'identification des symétries. Long et Fox [2003] suggèrent d'utiliser l'heuristique de Nebel *et al.* [1997] pour éliminer les propositions et opérateur non pertinents dans la description d'un problème i.e. ceux qui ne sont pas indispensables pour la réalisation des objectifs du plan, et ainsi faire apparaître éventuellement plus de symétries.

Long et Fox [2003] présentent deux aspects intéressants de la détection des symétries qui restent cependant des problèmes ouverts. Premièrement, les symétries détectées n'ont pas toutes le même impact sur la résolution d'un problème. Les approches existantes ne permettent pas de faire la distinction entre celles qui sont vraiment utilises pour réduire l'espace de recherche et les autres. Deuxièmement, la détection d'éléments *presque* symétriques permettrait d'exploiter une solution partielle existante pour inférer une autre partie presque identique de la solution. De telles presque symétries seraient cependant difficile à détecter et poseraient la question de déterminer quel élément peut être ignoré ou non.

9.2.5 Symétries en GGP

Banerjee *et al.* [2006] proposent une détection des symétries dans le cadre de l'élaboration d'une technique de transfert de fonction d'évaluation entre jeux similaires. Leur approche repose sur l'identification d'un plateau de jeu. Un prédicat ternaire, dont deux arguments représentent des coordonnées (*valeurs d'entrée*) et un troisième représente un statut (*valeur de sortie*) est recherché [Kuhlmann *et al.*, 2006]. Toutes les instances de ce prédicat représentent l'état des cases du plateau. À partir de ce plateau tous les types possibles de symétries sont testés. Une position terminale et ses symétriques doivent recevoir la même évaluation et toutes les transformations opérées par les actions doivent avoir leur symétrique. Leur approche nécessite l'exploration de tous les états d'un jeu : ils l'utilisent sur des versions réduites des jeux afin de transférer ensuite les informations acquises sur des jeu plus grands. Cette approche est néanmoins très limitée, par la lourdeur des calculs nécessaires, mais également par son manque de robustesse face aux moindres modifications dans la syntaxe du GDL. Comme évoqué précédemment (cf. §9.1.3), il est possible de ne pas utiliser de prédicats ternaires pour décrire une case : le nombre d'arguments peut varier, plusieurs arguments peuvent être fusionnés, plusieurs prédicats différents peuvent être utilisés pour décrire différentes zones du plateau, etc. Dans le meilleur des cas, les symétries détectées par cette approche se limitent à celles du plateau de jeu.

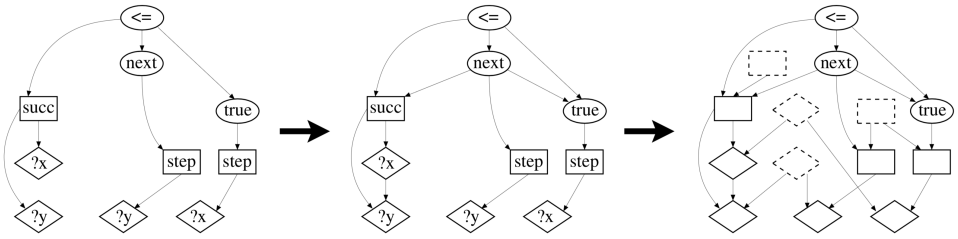


Figure 9.1 – Graphe de règles [Kuhlmann et Stone, 2007] représentant la règle `(<= (next (step ?y)) (true (step ?x)) (succ ?x ?y))`

Toujours dans l'optique de transférer des connaissances d'un jeu à un autre, Kuhlmann et Stone [2007] proposent de représenter les règles d'un jeu sous forme d'un graphe de règles (*rule graph*), une structure canonique qui capture les caractéristiques importantes du jeu en faisant abstraction des détails concernant le nom des prédicats, arguments, atomes ou l'ordre des règles (fig.9.1). Ce graphe contient un sommet pour chaque règle logique (relation d'implication), prédicat, variable et constante. Ces types de sommets sont étiquetés différemment pour les distinguer. Les prédicats qui sont des mots clefs du langage sont distingués des autres. Chaque sommet est coloré en fonction du nom du prédicat, variable ou atome auquel il correspond. Une arête relie chaque terme à ses arguments. Des arêtes sont ajoutées pour forcer l'ordre entre la conclusion d'une règle et ses prémisses. Finalement, pour faire abstraction des noms des prédicats, variables et termes, un sommet est ajouté pour chaque nom qui n'est pas un mot clef et est relié aux différents sommets correspondants. Deux graphes représentant des jeux distincts peuvent être comparés par recherche d'isomorphismes entre leurs graphes.

Cette approche n'est pas utilisée ici pour la recherche de symétries mais l'idée est reprise dans ce but par Schiffel [2010, 2011]. Ce dernier propose un graphe de règles amélioré (*enhanced rule graph*) qui permet de trouver également des symétries par permutation de l'ordre des arguments : une rotation du plateau de jeu par exemple. Des sommets supplémentaires représentent la position des arguments dans chaque prédicat et sont reliés aux prédicats et aux variables (ou constantes) correspondantes (fig.9.2). Pour pouvoir identifier les symétries correspondant à n'importe quel état du jeu et donc les détecter dynamiquement pendant le jeu, l'état initial du jeu n'est pas représenté dans le graphe. Pour exploiter les symétries détectées, il utilise une table de transposition avec un hachage à la Zobrist permettant un calcul rapide des clefs. À chaque position rencontrée pendant le jeu, les positions symétriques sont générées. Si une des positions symétrique est déjà présente dans la table de transposition, son évaluation est utilisée directement et aucun nouveau nœud n'est créé dans l'arbre du jeu.

Si cette approche permet de détecter toutes les symétries dans les jeux présentant une description GDL standard, une légère modification de la description comme l'ajout d'une règle tautologique suffit à entraver la détection.

Schkufza [2007] et Schkufza *et al.* [2008] présentent le concept d'automate propositionnel (*propositional automata*) qui sera ensuite utilisé dans le domaine du GGP sous le nom de *propnet* (*propositional network*). Ils concluent sur le potentiel des automates propositionnels dans la détection de structures dans un système de transitions et sur la possibilité de les utiliser pour identifier des sous-jeux indépendants. Cox *et al.* [2009] proposent une étude théorique sur l'utilisation de l'homomorphisme de deux parties

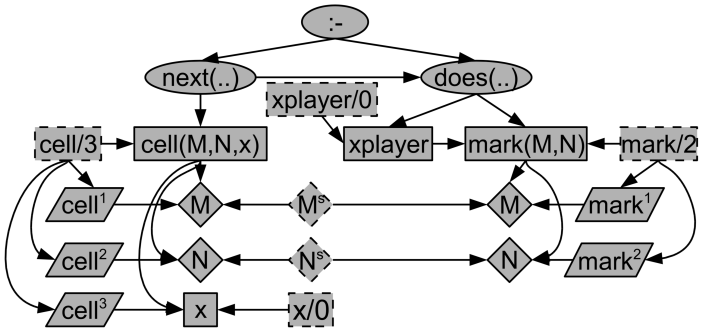


FIGURE 9.2 – Graphe de règles amélioré [Schiffel, 2010] représentant la règle `(<= (next (cell ?m ?n x)) (does xplayer (mark ?m ?n)))`

d'un *propnet* pour la découverte d'une symétrie du jeu. L'indépendance entre deux parties du *propnet* est d'abord vérifiée puis l'existence d'un homomorphisme entre ces deux parties est recherchée. Si les deux parties du jeu correspondent, une seule nécessite d'être examinée, permettant ainsi une importante économie durant l'exploration du jeu. Pour généraliser cette approche, ils indiquent que la factorisation d'une conjonction de sous-jeux peut être également dépendante d'une situation particulière du jeu (*Contingent Conjunctive Factorability*). Après un certain nombre de coups des joueurs, deux zones du plateau de jeu peuvent se trouver isolées et constituer deux sous-jeux totalement indépendants l'un de l'autre. La découverte de *latches* permet d'identifier la zone du plateau qui n'est plus modifiable et peut être ignorée. En éliminant du *propnet* les informations concernant cette zone du plateau, il est possible d'identifier deux parties indépendantes du jeu pouvant être explorées séparément. Cette approche pourrait également être étendue à la détection de N sous-jeux éventuellement non symétriques. Cependant, ici, la détection présentée est limitée à deux sous-jeux identiques.

Zhang *et al.* [2015] proposent une approche utilisant un graphe de dépendances pour le transfert de connaissances d'un jeu à l'autre. Cette approche peut également être utilisée pour la détection de symétries. Les règles du jeu sont instanciées (*grounding*) et transformées en graphe. Chaque sommet représente une proposition instanciée dont le prédicat est un mot clef du language (`init`, `role`, `does`, `true`, `next`, `legal`, `goal`, `terminal`) ou une des constantes numériques utilisées pour représenter le score. Les sommets sont étiquetés d'après le nom du prédicat correspondant. Une arête est ajoutée entre les termes liés logiquement. La recherche d'isomorphismes entre les graphes de deux jeux permet d'identifier s'ils utilisent des propositions équivalente. Une dernière phase consiste à vérifier si la logique des deux jeux est la même. La correspondance possible entre les propositions des deux jeux étant connue, les règles logiques peuvent être encodée sous forme d'un problème SAT. L'équivalence logique entre les deux jeux est alors vérifiée par un solveur SAT. Pour la recherche de symétries, les automorphismes du graphe sont recherchés et SAT est utilisé pour vérifier l'équivalence logique du jeu et de son symétrique. L'algorithme proposé est assez complexe, mais l'efficacité des algorithmes de recherche d'isomorphismes et des solveurs SAT le rend utilisable en pratique sur des jeux moyennement difficiles. Les temps de calcul annoncés laissent supposer que l'algorithme n'est pas applicable sur des jeux difficiles.

Une nouvelle approche dirigée par les contraintes à été proposée récemment par Piette [2016] et Koriche *et al.* [2017a] pour l'élaboration d'un joueur général nommé *Woodstock*. Ce joueur transforme les

règles d'un jeu en GDL à information complète ou incomplète en problème de satisfaction de contraintes stochastique (SCSP). Suivant l'approche proposée par Cohen *et al.* [2005], ils utilisent la *microstructure complémentaire* du SCSP, un hypergraphe représentant les conflits entre les différentes affectations potentielles des variables, pour identifier les symétries de contraintes [Koriche *et al.*, 2017b]. Cette approche, bien que limitée à la détection des symétries de contraintes (et non de solutions) présente l'avantage, par rapport aux méthodes précédentes, de ne pas être sensible à la syntaxe des règles.

9.3 Décomposition en planification, SAT et CSP

La décomposition d'un problème en sous-problèmes solubles séparément permet de réduire significativement l'espace de recherche à explorer pour sa résolution. Le problème de la décomposition a donc fait l'objet de nombreuses recherches en planification. Ces recherches peuvent être source d'idées pour l'élaboration d'une méthode de décomposition applicable aux jeux généraux. Dans cette section nous évoquons premièrement les quelques travaux concernant la décomposition dans les domaines de la résolution SAT et CSP. Nous détaillons ensuite les travaux effectués dans les domaines de la planification factorisée, hiérarchique et localisée. Nous présentons également deux approches originales, l'une concernant la détection et l'utilisation de *macro* en planification, l'autre concernant la décomposition des actions. Enfin nous discutons de la possibilité de transposer ces approches au domaine du GGP et des difficultés qui peuvent en découler.

9.3.1 Décomposition en SAT et CSP

L'heuristique *MINCE* (*MIN Cut Etc.*) [Aloul *et al.*, 2001] consiste a réordonner les variables d'un problème SAT pour tirer partie de la structure du problème et est assimilable à une décomposition. Un hypergraphe est construit : les sommets sont les variables du problème et un hyperlien relie chaque couple de variables utilisé dans une même clause (sous forme normale conjonctive). La coupure minimale du graphe [Karger et Stein, 1996] permet de séparer les variables en deux groupes qui seront examinés l'un après l'autre. En partitionnant récursivement le graphe toutes les variables sont ordonnées : les variables d'avantage liées sont examinées à la suite ce qui permet de détecter plus vite une affectation des variables ne satisfaisant pas les contraintes et de diminuer la zone de l'espace de recherche à explorer.

Dans le domaine des CSP, Jégou [1993] propose une approche pour la décomposition d'un problème de satisfaction de contraintes en sous-problèmes solubles séparément. Son approche est fondée sur l'analyse d'un hypergraphe représentant la *miscrostructure* du CSP : un graphe dont chaque sommet est un couple variable/valeur et chaque arête entre deux sommets représente une instanciation partielle compatible avec les contraintes. Après avoir calculé la triangulation du graphe, il identifie les cliques maximales correspondant à des sous-problèmes dont le domaine des variables est réduit. Plus le domaine des variables est réduit, plus la décomposition apporte un gain pour la résolution du CSP ; des domaines réduits à 1 ou 2 valeurs impliquent des sous-problèmes de complexité polynomiale. Les solutions du CSP sont comprises dans l'union des solutions des sous-problèmes.

9.3.2 Planification factorisée

Amir et Engelhardt [2003] proposent de décomposer un problème de planification en passant par sa représentation sous forme d'un graphe non orienté. Un sommet est ajouté pour chaque fluent propositionnel intervenant dans la description du problème. Une arête relie chaque couple de sommets intervenant dans la description d'une action comme précondition ou comme effet. La *décomposition arborescente* du graphe [54] permet d'obtenir une représentation sous forme d'arbre dont chaque nœud associe un groupe de fluents propositionnels aux actions exprimées en fonction de ces fluents, et représente une partie du problème initial faiblement liée aux autres. Les liens dans l'arbre sont étiquetés en fonction des fluents partagés d'un sous-domaine à l'autre. Le problème global est résolu par la procédure *PartPlan* en partant des feuilles. Tous les plans d'un sous-domaine, permettant d'aller d'un état initial des fluents partagés avec le sous-domaine parent à un état final différent, sont calculés. Chaque sous-plan est représenté par une action abstraite utilisable dans le domaine parent. Pour pouvoir gérer les cas où des interactions complexes sont nécessaires avec le sous-domaine parent, les plans du sous-domaine enfant peuvent être constitués de N étapes entrecoupées d'interactions avec le domaine parent. Le nombre d'interactions est augmenté par approfondissement itératif en cas d'échec de la planification et les sous-plans sont recalculés. La complexité de la méthode de décomposition est dominée par la complexité de la décomposition arborescente du graphe qui est en $O(n^2)$ pour n sommets. Cette complexité annoncée montre que l'approche est difficilement utilisable pour des problèmes décrits avec un grand nombre de fluents.

Brafman et Domshlak [2006] proposent une approche similaire fondée sur la décomposition arborescente d'un graphe de causalité (*causal graph*). L'approche est théorique et ne fait pas l'objet d'expérimentations. La complexité de la résolution du problème dépend de la largeur arborescente du graphe et de la taille des sous-plans dans chaque sous-domaine. Ils démontrent que leur approche fondée sur les graphes de causalité est strictement plus efficace (jusqu'à un facteur exponentiel) que l'approche de Amir et Engelhardt. Dans le graphe de causalité, chaque sommet représente une variable d'état. Un arc orienté est ajouté d'une précondition vers un effet d'une action et un arc non orienté entre deux effets d'une même action, alors que le graphe de Amir et Engelhardt contient une clique entre toutes les préconditions et effets d'une action. La largeur arborescente pour le graphe de causalité est donc inférieure. La procédure *LID* (*Local Iterative Deepening*), peut-être utilisée directement sur le graphe de causalité. Chaque sommet représente un sous-problème réduit à une variable et la liste des sous-plans de longueur L pour changer la valeur d'une variable est fournie à priori. La recherche d'un plan global à partir des sous-plans est codée sous la forme d'un SeqCSP, un problème de satisfaction de contraintes fondé sur des séquences : chaque variable du CSP est un ensemble de sous-plans et chaque contrainte est une arête dans le graphe de causalité. La recherche est effectuée par approfondissement itératif en augmentant la longueur L des séquences si aucun plan n'est trouvé. La procédure *LID-GF* (*Local Iterative Deepening, Generalized Factoring*) procède de manière similaire sur la décomposition arborescente du graphe de

54. Des algorithmes permettent de produire des décompositions arborescentes proches de l'optimum [Becker et Geiger, 1996; Bodlaender, 1997; Amir, 2001]. La décomposition arborescente (*tree decomposition*) identifie un ensemble de séparateurs, sous-ensembles de sommets (de taille minimale) du graphe original, dont la suppression rend le graphe non connexe. Ces séparateurs sont connectés dans un arbre aussi nommé arbre de jonction (*junction tree*). Il existe une arête entre deux séparateurs s'ils partagent un même sommet du graphe original. La largeur arborescente du graphe (*treewidth*) correspond à la taille du plus grand séparateur moins 1 dans une décomposition arborescente optimale. Intuitivement plus le graphe est proche d'un arbre plus la largeur arborescente se rapproche de 1.

causalité, dont la largeur arborescente est par définition de 1 ce qui réduit la complexité de la recherche. Le graphe étant moins connecté que celui de Amir et Engelhardt, la décomposition obtenue est également plus optimale. La recherche peut aussi être optimisée en favorisant le regroupement des variables de même variance (sous-plans de longueur semblable) dans les mêmes nœuds de l'arbre.

Ces deux approches évitent le *backtracking* en calculant tous les sous-plans possibles au niveau n avant de chercher au niveau $n + 1$. Ce qui implique une grosse consommation de mémoire et des plans optimum au niveau local mais pas forcement de longueur minimum au niveau global.

Kelareva *et al.* [2007] proposent une autre approche de planification factorisée inspirée des solutions proposées en planification hiérarchique ou localisée. Leur algorithme utilise le *backtracking*, mais permet d'éviter le calcul de tous les sous-plans. Ils construisent un hypergraphe avec un sommet pour chaque opérateur et un hyperlien pour chaque variable connectant tous les sommets d'opérateurs dans lesquels elle apparaît. Le graphe est alors transformé en *arbre de décomposition* [55] (*dTree*) [Darwiche et Hopkins, 2001]. Les nœuds représentent des groupes d'actions à utiliser dans les sous-plans, ils peuvent être fusionnés pour réduire la profondeur de l'arbre et sont ordonnés pour examiner en premier les actions susceptibles de modifier d'avantage de variables. La résolution est effectuée en partant de la racine. Chaque sous-plan utilise les actions du nœuds, associées à des actions abstraites correspondant à des trous dans le plan. À la racine, les buts du problème original sont utilisés. Aux autres niveaux, un ensemble de trous à combler est transmis par le niveau précédent. Un plan est avorté et donne lieu à un retour arrière dès qu'un trou ne peut être comblé. Le problème est encodé sous forme de problème SAT et résolu avec *zChaff*. L'approche évite les retours arrières mais elle est limitée aux problèmes encodés avec STRIPS : les buts doivent être une conjonction de littéraux positifs.

9.3.3 Planification hiérarchique

Le système *ALPINE* permet de décomposer un problème en plusieurs niveaux d'abstraction [Knoblock, 1990, 1994]. Le problème est représenté sous forme d'un graphe orienté dont les sommets sont des littéraux et les arcs sont des contraintes. Le graphe est construit en partant des littéraux buts et en ajoutant un arc entre préconditions et effets pour chaque action amenant une proposition but. Les différents effets d'un opérateur sont liés pour les placer au même niveau d'abstraction. Le processus est itéré sur chaque précondition qui peut être considérée comme un sous-but [Knoblock, 1991] jusqu'à ce que toutes les contraintes de chaque littéral soient ajoutées au graphe. Les composantes fortement connexes du graphe constituent les niveaux d'abstraction. Ces niveaux sont ordonnés par tri topologique du graphe. La résolution du problème hiérarchique est implémentée dans l'architecture *PRODIGY*. La conception du plan démarre au niveau d'abstraction le plus haut en raffinant progressivement le plan. Les actions abstraites sont des actions simplifiées en retirant des préconditions et des effets les fluents appartenant aux autres niveaux d'abstraction. La hiérarchie des niveaux d'abstraction garantit que les effets d'une action ne seront pas remis en cause aux niveaux inférieurs. Aucun retour en arrière n'est donc nécessaire pour le raffinement d'un plan, uniquement pour une modification du plan global.

55. Il s'agit de *branch decomposition* produisant un arbre binaire sans racine dont chaque feuille est un sommet du graphe original. Enlever une arête dans l'arbre correspond à partitionner le graphe en deux parties. À ne pas confondre avec la décomposition arborescente.

9.3.4 Planification localisée

Lansky et Getoor [1995] proposent la décomposition d'un problème formulé en terme d'actions et de contraintes sur ces actions (par exemple quelle action doit en précéder une autre). Le résultat de la décomposition est un ensemble de *régions* et *sous-régions* avec de possibles chevauchements. Son algorithme *LOC* (*Localisation generator*) part d'une liste de régions composées d'un seul type d'action, et de régions composées d'une seule contrainte. Chaque région de contrainte est liée à l'ensemble des régions composées d'un type d'action concerné par la contrainte. Il applique ensuite des transformations itérativement pour regrouper, hiérarchiser les régions. Les résultats expérimentaux montrent que la localisation obtenue est semblable à une décomposition experte. Chaque région est un ensemble de types d'actions (actions non instanciées) et de contraintes sur ces actions. Régions et sous-régions sont organisées en graphe orienté acyclique. La résolution du problème localisé est effectuée par l'algorithme *COLLAGE*. Une contrainte d'une région est activée et le plan est affiné pour la résoudre. Les contraintes concernées par les modifications du plan se trouvent activées et sont résolues à leur tour. Une heuristique *agenda* est utilisée pour ordonner la sélection et la résolution des contraintes activées, en favorisant par exemple les contraintes d'une même région ou d'une région voisine. La procédure est itérée jusqu'à satisfaction de toutes les contraintes et obtention d'un plan global.

9.3.5 Génération de *Macros*

Asai et Fukunaga [2015] proposent une approche pour décomposer des problèmes de planification à grande échelle i.e. impliquant un grand nombre d'opérations semblables. Des *composants* sont identifiés à partir d'un graphe (*static graph*). Les sommets sont les types objets utilisés dans la description du problème, étiquetés avec l'algorithme *TIM* [Fox et Long, 1998]. Les arêtes relient les objets utilisés en argument d'un même prédicat statique. Les *composants*, groupes d'objets connexes sans chevauchement, sont identifiés en partant des objets présents dans les buts. Ils sont ensuite comparés par isomorphisme de graphe pour dégager des types de composants. Des *composants de tâche* associent les objets d'un *composant* aux littéraux de l'état initial et aux buts qui concernent ces objets. Les sous-plans permettant de passer les objets de leur état initial à leur état but sont générés et constituent des *macro* utilisables pour la planification globale. Le problème est reformulé pour inclure ces *macro-actions* et il est résolu par un solveur quelconque. Le plan global est obtenu en re-développant les *macro*.

9.3.6 Décomposition des actions

Areces *et al.* [2014] proposent une approche originale pour simplifier le travail des solveurs qui instancient une description PDDL d'un problème. La décomposition des actions permet de limiter le nombre d'instances des actions à examiner, en réduisant le nombre de variables dans chaque sous-actions. Le but est d'obtenir une décomposition des actions assurant que les sous-actions seront exécutées en groupe et que l'instanciation des variables sera équivalente à celles de l'action d'origine. Pour cela, les littéraux d'une action sont annotés et ordonnés en fonction de leur rôle (précondition, effet positif ou négatif). Ces littéraux annotés sont les sommets d'un graphe orienté (*quotient graph*). Une arête est ajoutée entre les littéraux de même prédicat mais de rôle différent et est orientée en fonction de leur

ordre. Les littéraux sont partitionnés de manière à respecter l'ordre topologique du graphe et à minimiser les variables dans chaque sous-ensemble. Les sous-actions sont générées à partir de ces sous-ensembles. Le lien entre les sous-actions est assuré par l'ajout d'atomes dans les effets d'une action et dans les préconditions de la suivante.

9.3.7 Pistes pour le GGP et limitations

Les différentes approches présentées dans le domaine de la planification, de la vérification de modèles et de la résolution SAT ou CSP, apportent des idées pour la décomposition des jeux, cependant la transposition de ces approches présente différentes difficultés.

L'idée majeure qui ressort de ces approches est l'utilisation d'un graphe pour représenter la structure du problème. Même si ce graphe prend différentes formes, *graphe causal* [Brafman et Domshlak, 2006], *hypergraphe* représentant les dépendances entre fluents et actions [Kelareva *et al.*, 2007] ou *graphe de dépendances* entre prédicats statiques [Asai et Fukunaga, 2015], l'identification de parties connexes indépendantes ou faiblement liées permet chaque fois d'identifier des sous-problèmes. Cette idée présente toutefois différentes difficultés pour être adaptée au GGP.

Les prédicats statiques ne sont pas toujours présents en GDL et ne concernent pas toujours plusieurs fluents (objets). Les techniques présentées dans le domaine de la planification identifient également les objets d'après les variables d'état. Ceci n'a pas vraiment d'équivalent en GDL. Un fluent non instancié peut représenter un ensemble d'objets et chaque variable ne décrit qu'une caractéristique de l'objet en question. Par exemple le fluent `(cell ?x ?y ?s)` du jeu *Tictactoe* décrit les cases de coordonnées (x, y) (de domaine 1,2,3) et de statut s (de domaine b,x,o). Cependant, comme signalé auparavant (cf. §9.1.3), l'expressivité du GDL permet aussi d'utiliser un ensemble de fluents pour décrire les différents états des objets. Un cas extrême serait un jeu totalement instancié ou chaque fluent est représenté par un atome distinct. La relation existant entre ces fluents n'est pas donnée explicitement, il est alors nécessaire de la découvrir pour identifier les objets. Identifier les objets (cases, colonnes, pions) repose sur la détection des invariants [Haufe *et al.*, 2012] ce qui peut être assez lourd à calculer pour des jeux complexes. Ceci est inenvisageable dans le cas d'un jeu totalement instancié où la détection des objets se ramène à une identification systématique de tous les invariants de type *mutex*.

Réaliser un graphe entre des propositions et des opérateurs comme en planification nécessite de disposer de la liste des préconditions et des effets des opérateurs. Malheureusement le GDL ne décrit pas explicitement les effets des actions. Les règles `next` indiquent si un fluent est vrai dans l'état suivant en fonction de certaines caractéristiques de la position courante et de certaines actions des joueurs. Les règles ne précisent pas toujours l'état du fluent dans la position courante et n'indiquent donc pas clairement si les actions citées ont modifié quelque chose dans cette position. Les préconditions des actions sont réparties entre les règles `legal` et `next`. Cette distinction entre les propositions qui conditionnent la légalité et l'effet des actions n'a pas d'équivalent dans les domaines de la planification et de la vérification de modèles.

A cause de cette difficulté, nous considérons que l'approche de Areces *et al.* [2014], qui pourrait sembler intéressante pour traiter le problème des actions composées, n'est pas transposable au GGP.

Les différentes approches proposées en planification présentent également un problème de com-

plexité. La complexité de l'approche proposée par Amir et Engelhardt [2003] indique qu'elle ne peut vraisemblablement pas être transposée à des jeux complexes pouvant être décrits par un très grand nombre de fluents.

Les problèmes traités en planification, vérification de modèles, CSP et SAT, sont assimilables à des puzzles ou jeux solitaires. Ces problèmes ne sont pas comparables aux jeux multijoueurs où chaque joueur est en compétition pour maximiser son propre score. De plus, les jeux décrits en GDL présentent une séparation entre les conditions de terminaison (`terminal`) et l'achèvement des buts déterminant le score (`goal`) qui n'a pas d'équivalent dans ces autres domaines. On peut donc considérer que le GGP représente une classe de problèmes plus généraux.

L'offuscation des règles en compétition prive également le joueur de la possibilité de faire une analyse sémantique de la description du jeu. Les joueurs humains utilisent leur expérience de jeux connus et de la signification de termes comme "plateau", "case", "pion", etc. pour comprendre les règles d'un nouveau jeu, ce qu'un joueur programmé de GGP ne peut faire. Le GDL apporte donc une difficulté supplémentaire artificielle qui coupe de GGP d'autres domaines à partir desquels on pourrait transférer les approches utilisées.

Les jeux composés proposés dans domaine du GGP sont souvent constitués de sous-jeux indépendants i.e. ce sont souvent des combinaisons de jeux simples connus. Les sous-jeux ne partagent généralement pas d'actions (si on considère que les actions composées sont décomposables en actions distinctes) ni de fluents [56] contrairement aux sous-problèmes en planification qui peuvent être faiblement liés voire se chevaucher [Jégou, 1993]. Les approches utilisées en planification apportent différentes idées pour la décomposition dans le domaine des jeux. On peut envisager la décomposition des jeux en sous-jeux faiblement liés : les fluents et les actions sont partitionnés en groupes distincts, mais les fluents d'un sous-jeu peuvent pré-conditionner les actions d'un autre sous-jeu, ou bien les actions de l'un peuvent influencer les fluents d'un autre. On peut aussi envisager la décomposition d'un jeu en sous-jeux dont les objectifs se superposent, chaque sous-jeu partage alors des fluents et des actions avec d'autres sous-jeux.

Finalement l'approche de Asai et Fukunaga [2015] suggère l'idée de décomposer les jeux puis d'appliquer les techniques de détection de symétries proposées en planification pour identifier des sous-jeux isomorphes. L'application de la détection de symétrie sur les sous-jeux permettrait également de détecter des symétries supplémentaires non détectables dans le jeu complet.

9.4 Décomposition des jeux en GGP

Bien que l'importance de la décomposition dans le domaine du GGP soit reconnue [Genesereth et Björnsson, 2013], très peu de recherches ont été effectuées dans ce domaine. Comme nous l'avons vu (cf. §9.2.5), Cox *et al.* [2009] proposent une étude théorique sur l'utilisation d'un *propnet* pour l'identification de structures comme des sous-jeux. Le support de cours en ligne de l'université de Stanford sur le GGP [Genesereth, 2017] reprend ces idées dans un chapitre consacré à la factorisation des *propnets*. Ce cours propose d'opérer l'identification d'autre classes de jeux composés (jeux multiples, jeux avec une disjonction de sous-buts, jeux à actions-composées) via la structure du *propnet*. Cependant, aucune

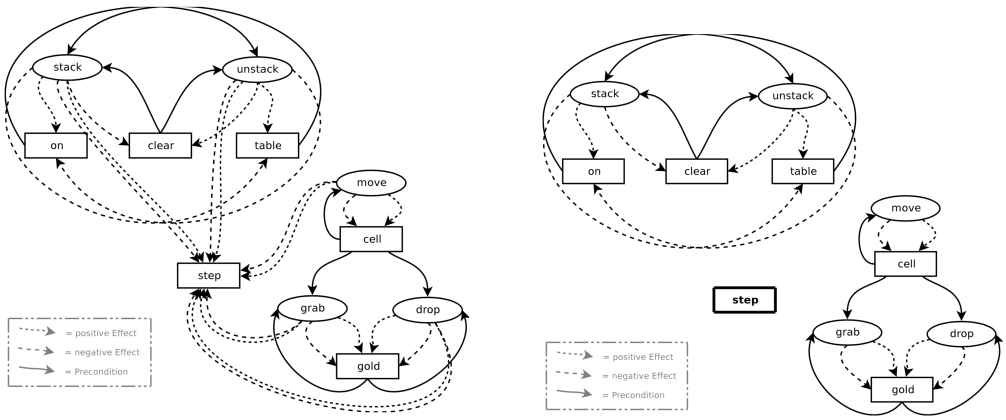
56. Il existe des exception comme nous l'avons vu à la section 8.4.

implémentation et évaluation de cette approche n'est proposée. Seuls deux articles traitent de la décomposition des jeux d'un point de vue pratique et ils proposent une approche similaire : Günther *et al.* [2009] présentent une méthode de décomposition pour les jeux solitaires et Zhao *et al.* [2009] étendent cette approche pour les jeux multijoueurs.

9.4.1 Jeux à un seul joueur

Günther [2007] (également Günther *et al.* [2009]) propose une approche pour la décomposition des jeux à un seul joueur. Les jeux considérés sont des jeux composés de sous-jeux indépendants. Deux sous-jeux sont considérés indépendants si une action d'un sous-jeu n'a aucune influence sur les fluents d'un autre sous-jeu. De plus, utiliser cette action ne doit aucunement modifier la légalité des actions de l'autre sous-jeu ni leurs effets.

Pour détecter les sous-jeux, il utilise un *graphe de dépendances*. Chaque sommet représente un symbole d'action (un prédicat N-aire `a` utilisé dans une action du type `(does r (a` b_1 `...` b_N`))`) ou symbole de fluent (prédicat N-aire `f` utilisé dans un fluent du type `(true (f` c_1 `...` c_N`))`. Pour chaque effet positif ou négatif potentiel une arête est ajoutée du symbole d'action vers le symbole du fluent concerné. De même, pour chaque précondition, une arête est ajoutée entre les symboles du fluent et de l'action correspondante. Les composantes connexes du graphe correspondent aux sous-jeux identifiés.



(a) *sans les informations sur les fluents indépendants des actions*

(b) *avec le prédicat* `step` *détecté comme étant indépendant des actions*

FIGURE 9.3 – Graphe de dépendances pour le jeu *Incredible* [Günther, 2007]

A cause de l'hypothèse d'un monde clos et de l'emploi d'axiomes définissant le cadre, les règles en GDL ne contiennent pas une description explicite de l'effet des actions. Il propose donc d'identifier des effets *potentiels*. Le symbole d'un fluent est un *effet positif potentiel* d'une action s'il existe une règle `next` indiquant que ce fluent sera vrai dans l'état suivant, que les prémisses de cette règle n'impliquent pas que le fluent soit déjà vrai et qu'ils soient compatibles avec l'action (l'action est explicitement citée ou non). Le symbole d'un fluent est un *effet négatif potentiel* d'une action si aucun *axiome du cadre*, indiquant les conditions dans lesquelles ce fluent reste vrai, n'implique cette action. Les préconditions d'un symbole d'action sont les symboles de fluents intervenant dans les prémisses de la règle `legal` concernant l'action et dans les prémisses des règles `next` correspondant à un effet potentiel positif ou

négatif de l'action.

Cette définition des *effets potentiels* implique que les fluents indépendants des actions seront identifiés comme des effets positifs et négatifs de toutes les actions (fig.9.3a). Pour éviter cela, il procède à une détection spécifique des fluents indépendant des actions : un symbole de fluent est considéré indépendant des actions si aucune action n'intervient dans les prémisses d'une règle `next` le concernant. De plus ce fluent ne doit jamais intervenir dans les prémisses d'une autre règle `next` ou dans les prémisses d'une règle `legal`. Cette détection permet d'isoler ces symboles de fluents et de mettre en évidence les sous-jeux dans le graphe de dépendances (fig.9.3b).

En plus d'être limitée aux jeux à un seul joueur, l'approche de Günther [2007] ne prend pas en compte les jeux en séries ou à action composées. Seuls les jeux totalement indépendants sont décomposés. L'utilisation des effets *potentiel* des actions peut amener à créer plus de liens dans le graphe qu'il n'y a d'effets réels, certains sous-jeux indépendants peuvent donc ne pas être identifiés. La performance limitée des machines et l'absence de techniques d'instanciation rapide des règles GDL en 2007 expliquent qu'il ait fait le choix d'établir le *graphe de dépendances* à partir des symboles de fluents et d'actions. Cette approche permet de limiter efficacement le temps de calcul mais l'approche est de fait sensible à la syntaxe des règles : l'utilisation, par exemple, d'un même symbole de fluent pour décrire le plateau de deux sous-jeux peut mettre la détection en échec. Malgré toutes ces limitations, l'utilisation d'un graphe, inspirée du domaine de la planification, représente une approche efficace pour l'identification des sous-jeux.

9.4.2 Jeux multijoueurs

L'idée d'utiliser un graphe de dépendances est reprise par Zhao *et al.* [2009] pour décomposer les jeux multijoueurs. Cependant deux modifications importantes sont nécessaires.

Premièrement, ils réalisent une instanciation partielle des fluents et des actions. Ceci pour éviter le problème évoqué précédemment lorsqu'un même symbole de fluent est utilisé dans deux sous-jeux, tout en limitant le temps de calcul. Un argument c_i d'un fluent `(f` c_1 `...` c_N`)` est instancié si sa valeur ne change pas d'un état à un autre et si les fluents qui différent par la valeur de cet argument n'interagissent pas ensemble. Autrement dit, si l'argument est représenté toujours par la même variable ou la même constante dans la conclusion et les prémisses d'une règle, il est instancié. Un argument b_j d'une action `(a` b_1 `...` b_N`)` est instancié si l'argument c_i d'un fluent est instancié et que l'action intervient comme prémisse dans une règle `(next (f` c_1 `...` c_N`))` avec $c_i = b_j$. Par exemple, dans le jeu *Nim*, cette instanciation partielle permet de distinguer les différentes piles d'allumettes représentées par les fluents `(heap a _)`, `(heap b _)`, etc.

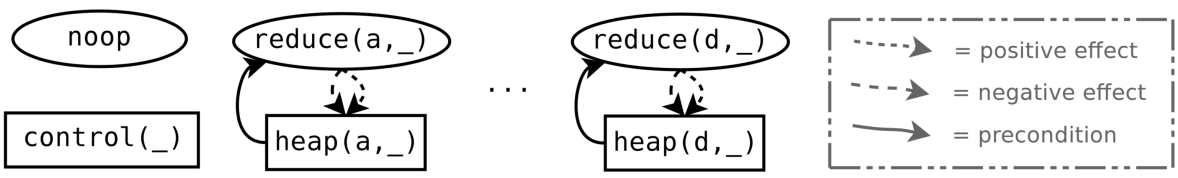


Figure 9.4 – Graphe de dépendances pour le jeu *Nim* [Zhao *et al.*, 2009]

Deuxièmement, les différents joueurs agissant simultanément, il devient nécessaire de pouvoir identifier quelle action composant un mouvement joint est responsable d'un effet positif ou négatif. Ils définissent donc la notion de *rôle affectant un fluent* : un rôle affecte un fluent si une action pour ce rôle est présente dans les prémisses d'une règle `next` décrivant l'état suivant de ce fluent. La présence possible de coups alternés des joueurs rend également nécessaire la détection du fluent désignant un joueur qui à la main (*control*) ainsi que de l'action indiquant qu'un joueur passe son tour (*noop*). Ils définissent l'action *noop* comme étant la seule action légale pour un joueur qui n'a pas le *control*, ayant en plus la caractéristique de n'intervenir dans les prémisses d'aucune règle `next`. Les effets positifs et négatifs potentiels sont définis comme pour les jeux à un seul joueur à la restriction près que l'action doit être différente de *noop* et que le rôle exécutant l'action doit affecter le fluent. Les préconditions potentielles sont définies comme précédemment.

Enfin pour éviter que le fluent représentant le *control*, qui intervient dans les prémisses de toutes les règles `legal`, n'unifie les sous-jeux, il est mit de côté et ne fait l'objet d'aucune arête dans le graphe de dépendances (fig.9.4).

Une fois encore, l'instanciation partielle des fluents rend l'approche très sensible à la syntaxe. La réécriture de Nim avec des fluents comme `(heap a1)`, ..., `(heap a3)`, `(heap b1)`, ..., `(heap b5)`, rendrait toute décomposition impossible. La détection des actions responsables d'un effet ou des *noop* est également fragile : dans certains jeux, une action *noop* est intentionnellement placée dans les permisses de règles `next` pour brouiller les pistes. Par exemple dans le jeu *Connect 5*, une action *noop* est utilisée dans le corps d'une règle `next` indiquant l'état suivant du fluent *control* : l'absence d'effet de l'action *noop* et la nature du fluent *control* sont alors mal détectées.

9.4.3 Jeux à actions composées

Zhao [2009] propose un traitement spécial pour les jeux comprenant des actions composées. Il part du principe qu'une action composée est constituée d'un prédicat avec au moins deux arguments représentant les effets de l'action dans chacun des sous-jeux. Ces arguments peuvent être identifiés du fait qu'ils ne sont pas utilisés dans toutes les règles `next` ou l'action intervient. Par exemple l'action `(mark ?x1 ?y1 ?x2 ?y2)` intervient dans la règle suivante du jeu *Tictactoe Parallel* :

```
(<= (next (cell1 ?x ?y ?mark))
    (true (cell1 ?x ?y ?mark))
    (does xplayer (mark ?x1 ?y1 ?x2 ?y2))
    (distinctcell ?x ?y ?x1 ?y1))
```

On peut constater que les deux premiers arguments sont utilisés dans cette règle qui concerne un fluent du premier sous-jeu i.e. ils apparaissent plus d'une fois, tandis que les deux autres arguments n'ont aucun rôle à jouer. Le rôle des arguments est examiné dans toutes les règles `next` et les groupes d'arguments identifiés sont vérifiés au regard des règles `legal`. Si des arguments placés dans des groupes distincts sont utilisés dans un même fluent, leur indépendance est remise en cause. Sinon, l'action est considérée comme composée. Voici par exemple la règle `legal` correspondant à l'action précédente :

```
(<= (legal ?player (mark ?x1 ?y1 ?x2 ?y2))
    (true (control ?player))
```

```
        (true (cell1 ?x1 ?y1 b))
        (true (cell2 ?x2 ?y2 b)))
```

Cette détection est présentée uniquement en théorie et ne semble pas avoir été appliquée en pratique. L'hypothèse de départ est malheureusement assez faible. De nombreuses actions composées ne sont pas constituées de plusieurs arguments. Par exemple le jeu *Asteroid Parallel* utilise les actions suivantes qui combinent les actions `thrust`, `clock` ou `counter` pour chacun des deux sous-jeux :

```
(does ship (do thrustThrust))
(does ship (do thrustClock))
(does ship (do thrustCounter))

(does ship (do clockThrust))
(does ship (do clockClock))
(does ship (do clockCounter))

(does ship (do counterThrust))
(does ship (do counterClock))
(does ship (do counterCounter))
```

Même dans le cas ou des arguments séparés sont présents, la manière d'écrire les règles peut entraver la détection. Par exemple, nous pouvons réécrire l'axiome du cadre précédent en utilisant la négation de l'action ayant un effet négatif, plutôt que le modèle d'une action sans effet sur le fluent. Dans ce cas, il est nécessaire d'ajouter un littéral pour s'assurer que les variables `?x2` et `?y2` pourront bien être instanciées, ce qui masque leur inutilité :

```
(<= (next (cell1 ?x ?y ?mark))
    (true (cell1 ?x ?y ?mark))
    (true (cell2 ?x2 ?y2 ?mark2))
    (not (does xplayer (mark ?x ?y ?x2 ?y2))))
```

9.4.4 Jeux en série

Pour l'identification de sous-jeux en série, Zhao [2009] propose également un traitement spécifique. Dans un jeu en série, la légalité des actions d'un sous-jeu intervenant après un autre est conditionnée par une situation spécifique du sous-jeu précédent. Son idée est de comparer les prémisses des règles `legal` pour identifier les actions des différents sous-jeux.

Si un même fluent intervient avec la même valeur (positif ou négatif) dans les prémisse des règles `legal` de deux actions, ces actions sont potentiellement légales en même temps et font partie du même sous-jeu. Si au contraire un fluent tantôt positif, tantôt négatif intervient dans les prémisses des règles `legal` de deux actions, ces actions ne sont jamais légales en même temps et appartiennent à deux sous-jeux distincts. Dans le cas où deux actions n'ont aucune prémisse en commun dans leur règle `legal`, il propose de les considérer comme appartenant à deux sous-jeux distincts jusqu'à ce que la détection d'un effet commun de ces actions sur un même fluent partiellement instancié crée un lien entre elles. Les différents groupes détectés indiquent les différents sous-jeux en série.

Dans le cas où différents groupes d'actions sont détectés, en identifiant un des prémisses condition-

nant la légalité d'un des groupes comme étant un effet des actions d'un autre groupe, il est possible d'ordonner les groupes d'actions pour savoir quel sous-jeu est joué après l'autre.

9.4.5 Jeux impartiaux

L'identification des jeux impartiaux permet d'utiliser les techniques connues pour leur résolution [Berlekamp *et al.*, 1982]. Un jeu est impartial, s'il est à coups alternés, si pour toute position du jeu chaque joueur dispose des mêmes options (s'il a la main) et si une action a toujours le même effet quelque soit le joueur qui la joue. Zhao [2009] propose d'identifier les jeux impartiaux par une analyse syntaxique des règles. Si pour toutes les règles `legal` et `next` concernant les actions d'un joueur il existe une règle équivalente pour le joueur adverse alors le jeu est considéré impartial. La méthode est bien évidemment sensible à la syntaxe, mais elle permet rapidement d'identifier les jeux impartiaux les plus évidents.

9.5 Exploitation des décompositions en GGP

Bien qu'il ne propose pas de méthode pour obtenir la décomposition d'un jeu, Müller [1999] présente une méthode de recherche pour des jeux à deux joueurs déjà décomposés et dont les sous-jeux ne contiennent pas de cycles. Son approche est fondée sur la théorie combinatoire des jeux [Conway, 2000] et présuppose que le jeu puisse être découpé en sous-jeux correspondant au modèle des jeux combinatoires i.e. que l'évaluation globale d'une position corresponde à la somme des valeurs des sous-jeux. Il adapte une recherche de type *Minimax* pour prendre en compte une alternance quelconque des joueurs dans un sous-jeu. Les sous-jeux sont entièrement explorés. Au niveau global les évaluations des meilleurs coups locaux sont comparées pour choisir la prochaine action. Si aucune action ne domine les autres, une comparaison plus coûteuse utilisant la somme combinatoire des jeux est utilisée. Cette approche est appliqué au *Go* pour le traitement des fins de parties. Une base de données permet de stocker les meilleures stratégies locales pour un ensemble de motifs particuliers réduits à quelques cases.

Dans le domaine du GGP, Günther et Zhao s'inspirent des outils de planification hiérarchique *PartPlan* [Amir et Engelhardt, 2003] et *LID-GF* [Brafman et Domshlak, 2006] pour exploiter les sous-jeux identifiés. Leurs approches pour l'exploration des jeux décomposés à un seul ou plusieurs joueurs consistent à rechercher l'ensemble des sous-plans locaux dans chaque sous-jeu puis à recombiner ces sous-plans en un plan global. L'approche *dTreePlan* [Kelareva *et al.*, 2007] permet d'éviter un calcul exhaustif des sous-plans. Cependant, elle impose que les buts soient exprimés sous forme de conjonction de fluents positifs ce qui n'est pas toujours le cas des jeux décrits en GDL. De fait, elle n'est pas facilement transposable au GGP.

Un premier problème se pose concernant l'évaluation des positions dans les sous-jeux puisque celles-ci correspondent à des positions partielles du jeu global. Pour évaluer les sous-plans, Günther [2007] propose une notion de *sous-but* et *sous-terminal* qu'il nomme *concepts locaux* (*local concepts*). Pour identifier ces derniers, il crée un arbre des appels pour les prédicats `goal` et `terminal`, puis, en partant de la racine, la succession des appels est suivie jusqu'à ce qu'un prédicat rencontré n'ait comme enfants que des prédicats appartenant à un même sous-jeu. Les prédicats de cet arbre doivent être instanciés et

un *concept local* ne doit pas être l'objet d'une récursion i.e. il ne doit pas être impliqué dans un cycle.

A partir de ces *concepts locaux*, Günther [2007] propose une recherche optimiste pour les jeux à un seul joueur (*greedy search*) qui consiste à simplement identifier un sous-plan amenant à une évaluation maximum dans chaque sous-jeu i.e. un maximum de *sous-buts* sont remplis. Au niveau global, ces sous-plans sont alors exécutés l'un après l'autre.

Cette approche pose deux autres problèmes. Les scores et la fin du jeu étant décrits en GDL par deux types de règles distincts (`goal` et `terminal`), une combinaison particulière de positions partielles des sous-jeux peut mettre fin à la partie avant que les sous-plans soient entièrement exécutés. En plus de ce risque de *mort subite*, il est possible qu'un des sous-jeux ne soit pas impliqué dans le calcul du score, ainsi aucun *sous-but* ne permet de sélectionner un sous-plan plutôt qu'un autre.

9.5.1 Jeux à un seul joueur

Pour explorer les jeux à un seul joueur en évitant le risque de *mort subite*, Günther [2007] et Günther *et al.* [2009] proposent une recherche à profondeur itérative (*Concept Decomposition Search*) fondée sur l'ajout d'une signature sur chaque séquence d'actions constituant un sous-plans. La signature est constituée de *concepts locaux* ; elle comprend la liste des *sous-buts* remplis à la fin de la séquence d'actions et la liste des *sous-terminaux* remplis à chaque tour. Une séquence d'actions s'achève quand il n'y a plus de coup légal dans le sous-jeu, si la position est terminale globalement ou si la profondeur maximum d'exploration est atteinte.

A chaque itération, l'ensemble des sous-plans à profondeur N est calculé pour chaque sous-jeu. Un seul sous-plan est conservé parmi les sous-plans de même signature. Finalement, la recherche globale examine toutes les combinaisons possibles des séquences d'actions pour obtenir une séquence globale qui ne soit pas terminale prématurément et menant à l'évaluation maximale. Cette approche permet de trouver le plan optimal, mais elle est évidement très coûteuse en temps et en espace.

Cerexhe *et al.* [2014] apportent un fondement théorique à cet algorithme et proposent une méthode de résolution fondée sur la vérification de modèles avec ASP. Ils proposent de considérer la séquence d'actions recherchée au niveau global comme une *condition de stabilité*, une formule qui doit être vérifiée par le modèle i.e. le jeu. En partant des règles `terminal` et `goal`, ils proposent de constituer une formule qui est une séquence de N conditions : les $N-1$ première conditions indiquent que les $N-1$ positions ne doivent pas être terminales, la dernière étant que la position doit être terminale et vérifier une des clauses définissant l'obtention du score maximum.

Ils définissent alors trois opérateurs de composition : les sous-jeux peuvent être *synchrones* i.e. les sous-jeux progressent à chaque tour, *asynchrones* i.e. le joueur joue tantôt dans un sous-jeu tantôt dans un autre, ou *séquentiels* i.e. les sous-jeux se déroulent l'un après l'autre. Dans le cas des jeux *synchrones*, les sous-formules ont la même longueur que la formule globale. Pour obtenir les sous-formules, il suffit de découper chaque condition en attribuant les sous-terminaux et les sous-buts à la sous-formule correspondante. Dans le cas des jeux *asynchrones*, pour obtenir les conditions de non-terminaison, il faut trouver toutes les combinaisons possibles entre les conditions de non-terminaison des sous-jeux. La dernière condition de la séquence est de remplir les sous-buts et les sous-terminaux pour chaque sous-jeu. La somme des longueurs des sous-formules *asynchrones* doit correspondre à celle de la formule globale.

Dans le cas de deux sous-jeux *séquentiels*, la formule globale est simplement coupée en deux dans sa longueur.

En pratique, pour résoudre le jeu, ils proposent de l'encoder en ASP selon la méthode proposée par Thielscher [2009] (cf. §9.1.3). La formule de chaque sous-jeu est également encodée et un solveur ASP est utilisé pour trouver une séquence d'actions respectant la formule. La méthode ramène un problème de vérification d'un modèle à la vérification de ses parties, ce qui réduit énormément la difficulté de résolution. Cependant, les complexités annoncées pour la résolution des différentes compositions de sous-jeux sont relativement importantes et semblent inadaptées pour des sous-jeux complexes avec un grand nombre d'états. La manière dont cette méthode de réduction d'un jeu à un problème de vérification de modèles peut être étendue à un jeu multijoueur n'est pas définie.

9.5.2 Jeux multijoueurs

Pour adapter la méthode de Günther aux jeux multijoueurs, Zhao [2009] indique qu'il est nécessaire d'étendre la notion de *concept local* à des propositions (fluents ou prédicats auxiliaires) associées à une valeur. En effet, un joueur doit souvent remplir certaines conditions (propositions vraies) et empêcher son adversaire de faire de même (propositions fausses) pour obtenir le score maximum.

L'algorithme détaillé [Zhao *et al.*, 2009] concerne le cas des jeux à coups alternés i.e. jeux composés asynchrones. À chaque tour, un joueur doit choisir le coup qu'il veut jouer, et le sous-jeu dans lequel il désire jouer, pendant que l'autre joueur joue une action *noop* sans effet. Pour établir les séquences d'actions au niveau local, Zhao ne tient compte, à chaque tour, que de l'action du joueur qui a la main. La signature des séquences est enrichie par l'ajout d'une liste indiquant l'alternance des joueurs. Toutes les séquences sont recherchées à profondeur N comme dans l'approche de Günther. Une fois toutes les séquences locales évaluées, certaines peuvent être supprimées car elles correspondent à la même signature ou quasiment la même avec une évaluation plus faible i.e. moins de sous-buts remplis. Pour combiner les séquences au niveau global une nouvelle difficulté apparaît, celle d'avoir une alternance cohérente des joueurs.

Deux solutions sont proposées. La première consiste à utiliser un préfixe indiquant l'alternance des joueurs dans un sous-plan pour trouver rapidement les sous-plans cohérents à partir d'une position donnée. Quand un coup est choisi, le joueur est ajouté au préfixe du sous-jeu concerné. À partir des différentes séquences, il propose un examen des séquences deux à deux à rebours pour sélectionner celle qui permet d'obtenir le meilleur score alors que l'autre joueur tente de maximiser le sien. La seconde approche consiste à utiliser une recherche classique, mais dans l'espace des séquences plutôt que dans l'espace des positions, et à restreindre le choix des actions d'après les séquences possibles plutôt que d'après l'ensemble des actions légales. La recherche locale permet alors juste de filtrer les actions les plus prometteuses pour guider la recherche globale.

Dans les jeux à coups simultanés, il faut tenir compte des mouvements joints. Zhao propose d'adapter son approche précédente et d'utiliser des alternances de tours combinés dans la signature des séquences. Une recherche classique à partir des séquences plutôt qu'à partir des actions légales est là aussi applicable.

9.5.3 Jeux à actions composées, en séries et jeux impartiaux

Les méthodes proposées par Zhao [2009] pour la détection des actions composées et des jeux en série ne font pas l'objet d'expérimentations.

Concernant le cas particulier des jeux impartiaux, Zhao indique que chaque sous-jeu obtenu est lui-même impartial. Il propose de calculer le *Nimber* (le nombre de *steps*) de chaque sous-jeu puis de calculer le meilleur coup parmi les sous-jeux en fonction des conditions de victoire (jeu normal ou en version *misère* [57]).

9.6 Décomposition des règles GDL

Schiffel propose une simplification des règles GDL qui consiste à éliminer les variables existentielles [Haufe *et al.*, 2012; Schiffel, 2016]. Elle a comme avantage direct de réduire considérablement le nombre de règles générées lors de leur instanciation, mais elle a aussi l'effet indirect de créer une forme de factorisation des règles. De fait, cette transformation peut être assimilée à une décomposition des règles.

Une variable est quantifiée existentiellement si elle n'apparaît pas dans la conclusion de la règle. Schiffel donne l'exemple de la règle suivante :

```
(<= (p ?x ?z) (q ?x ?y) (r ?y) (s ?z))
```

Si X, Z et Y sont les tailles des domaines de valeurs respectifs de ces variables alors le nombre d'instances de la règle est égal à $X \times Y \times Z$. Cependant si la règle est scindée en deux règles contenant chacune moins de variables le nombre d'instances des deux règles n'est plus que de $X \times Z + X \times Y$:

```
(<= (p ?x ?z) (qr ?x) (s ?z))
(<= (qr ?x) (q ?x ?y) (r ?y))
```

Cela permet de réduire significativement le nombre de règles instanciées dès que le domaine des variables est supérieur à deux valeurs.

57. La version *misère* d'un jeu consiste à inverser son but. Par exemple, dans la version du jeu de *Nim* proposée par Conway, le joueur qui n'est plus en mesure de prendre une allumette est perdant. Dans la version *misère* de ce jeu, le perdant est celui qui saisit la dernière allumette.

Chapitre 10
Méthodes générales de décomposition

Sommaire

La diversité des jeux et de leurs descriptions en GDL rend la mise au point d'une méthode de décomposition générale et robuste extrêmement complexe. Pour toute méthode de décomposition partiellement ad-hoc, il est envisageable de produire une description de jeu pouvant mettre la décomposition en échec. Notre but est donc la mise au point d'une méthode la plus générale et la plus robuste possible pour décomposer l'ensemble des descriptions de jeu existantes dans les limites de ce qui est calculable dans un

temps restreint donné. L'objectif final étant de pouvoir fournir à un joueur GGP des décompositions de jeu utilisables ensuite pour une résolution facilitée du jeu global.

Dans ce chapitre, nous présentons deux méthodes de décomposition distinctes. La première [Hufschmitt *et al.*, 2016, 2017], s'appuyant sur les travaux précédents [Günther, 2007; Zhao, 2009], se fonde sur une analyse des règles GDL. Nous la désignerons sous le nom de *décomposition générale fondée sur les règles* (DGFR). Cette approche, bien que plus générale que les précédentes, présente différents défauts qui en limitent la performance et la robustesse. La seconde [Hufschmitt *et al.*, 2018a,b], fondée sur une analyse statistique d'informations collectées pendant des simulations permet de résoudre les problèmes soulevés par l'approche précédente et constitue une méthode générale et robuste pour décomposer une majorité de jeux. Nous la désignerons sous le nom de *décomposition générale fondée sur les simulations* (DGFS).

10.1 Préliminaires

Avant de présenter le détail de ces deux méthodes de décomposition dans la suite de ce chapitre, nous commençons par définir les notions de jeu, de sous-jeu et de décomposition correcte. Les jeux composés décrits en GDL étant généralement constitués de sous-jeux disjoints, nous envisageons uniquement le cas des sous-jeux parfaitement séparés et non celui de sous-jeux se recoupant partiellement. Nous présentons ensuite différents choix que nous avons effectués : l'utilisation d'un graphe de dépendances, la décomposition des jeux à partir de règles instanciées et l'utilisation d'une représentation des règles sous forme de circuit logique. Nous expliquons enfin une des difficultés majeures à résoudre pour la mise au point d'une méthode de décomposition : l'identification des liens de causalité entre les actions et les fluents d'un jeu.

10.1.1 Jeux, sous-jeux et décomposition correcte

Un jeu général peut être considéré comme un automate à états finis dont la structure peut être représentée par un graphe orienté acyclique. Soit F l'ensemble des fluents du jeu. Un état du jeu est un ensemble $s \subset F$ et une transition un tuple $(s, s') \subset F^2$. Une décomposition est une partition $(F_1, ..., F_n)$ telle que $\bigcup_{i=1}^{n} F_i = F$ et $\forall i, \forall j, i \neq j \ : F_i \cap F_j = \emptyset$. Dans un sous-jeu i un état est un ensemble $s_i \subset F_i$ et une transition est un couple $(s_i, s'_i) \subset F_i^2$ tel qu'il existe une transition $(s, s') \subset F^2$ où $s_i = s \cap F_i$ et $s'_i = s' \cap F_i$.

Definition 10.1 (choix libre d'une transition)**.** *Considérons un état $s = s_1 \cup s_2$ d'un jeu, avec s_1 un état du premier sous-jeu et s_2 un état du second. Nous pouvons choisir* librement *une transition (s_1, s'_1) dans le premier sous-jeu et (s_2, s'_2) dans le second si, dans le jeu global, il existe une transition $(s_1 \cup s_2, s'_1 \cup s'_2)$ ou une séquence de transitions $(s_1 \cup s_2, s'_1 \cup s_2), (s'_1 \cup s_2, s'_1 \cup s'_2)$.*

Definition 10.2 (décomposition compatible)**.** *La décomposition d'un jeu en deux sous-jeux est* compatible *avec le jeu global si dans chaque sous-jeu il est possible de se déplacer librement d'un état à un autre état distinct.*

La généralisation des définitions 10.1 et 10.2 à plus de deux sous-jeux est évidente.

Definition 10.3 (décomposition correcte)**.** *Une décomposition est* correcte *si elle est compatible avec le jeu global et s'il existe des conditions de victoire indépendantes (fonction d'évaluation) dans les sous-jeux telles que gagner dans tous les sous-jeux implique de gagner le jeu global.*

```
(role r)
(light a) (light b) (light c) (light d)
(<= (legal r (push ?x)) (not (true (on ?x))) (light ?x))
(<= (next (on ?x)) (does r (push ?x)))
(<= (next (on ?x)) (true (on ?x)))
(<= terminal (true (on a)))
(<= (goal r 0) (not (true (on b))) (not (true (on c))))
(<= (goal r 40) (true (on b)) (not (true (on c))))
(<= (goal r 60) (not (true (on b))) (true (on c)))
(<= (goal r 100) (true (on b)) (true (on c)))
```

FIGURE 10.1 – Jeu composé minimaliste représenté en GDL. Le joueur peut allumer 4 lampes. Allumer a met fin au jeu. Il n'obtient le score maximal que si b et c sont allumés à la fin.

Dans l'exemple de la figure 10.1, chaque lampe peut être isolée dans un sous-jeu distinct : il est possible de choisir librement une transition pour allumer une des lampes et des fonctions d'évaluation indépendantes sont identifiables car chaque lampe impliquée dans le calcul du score contribue à l'obtention d'une partie des points. Dans le cas d'un jeu décomposable à score binaire, une fonction d'évaluation peut être néanmoins identifiée quand la condition de victoire est composée de sous-buts distincts pour chaque sous-jeu.

Certains jeux sont intentionnellement composés, mais il existe également d'autres jeux comme les *Nonogrammes*, que l'on peut considérer non-composés à priori et qui peuvent pourtant faire l'objet d'une décomposition correcte.

D'après les définitions précédentes, nous considérons en revanche qu'un jeu comme *Nine Board Tictactoe* n'est pas décomposable. Dans ce jeu regroupant 9 grilles de *Tictactoe* disposées en 3 lignes par 3 colonnes, chaque coup d'un joueur dans la case d'indice i d'une grille détermine la grille i suivante où l'adversaire devra répliquer, le but étant d'aligner 3 marques dans l'une des grilles. Les transitions dépendent du coup précédent et ne peuvent donc pas être choisies librement si chaque grille constitue un sous-jeu. Un jeu comme *Blocker* est également non décomposable. Dans ce jeu, disputé sur un plateau de 4×4 cases, *Crosser* doit placer sa marque dans les cases pour bâtir un pont d'un côté à l'autre du plateau tandis que *Blocker* essaye de lui barrer la route en déposant ses propres marques. Même s'il est possible de choisir librement les transitions dans les 16 sous-jeux constitués chacun d'une seule case du jeu global, il n'existe pas de condition de victoire indépendante permettant d'évaluer le score dans chaque sous-jeu, une case isolée pouvant, quelque soit le joueur, tout aussi bien appartenir à une position globale gagnante que perdante.

Certains jeux, pouvant être considérés comme composés, possèdent des fluents ou actions réutilisés dans plusieurs sous-jeux. C'est le cas par exemple des jeux en série *Ticblock*, *Brain Teaser Extended* ou *Factoring George Forman* (cf. §8.4). D'après les définitions précédentes, ces jeux sont non décomposables.

10.1.2 Graphe de dépendances

Nos deux méthodes de décomposition (DGFR et DGFS) utilisent toutes deux l'idée de Günther [2007] (cf. §9.4.1) qui consiste à construire un *graphe de dépendances* entre les actions et les fluents.

Notre *graphe de dépendances* est un graphe non orienté $\{V, E\}$ dans lequel nous ajoutons un sommet dans V pour chaque fluent et chaque action du jeu. Nous identifions ensuite des liens de dépendance entre eux (préconditions et effets) sans tenir compte de leur nom de prédicat ni de leurs arguments pour limiter au maximum la dépendance de notre approche à la syntaxe des termes GDL. Nous ajoutons des arêtes entre les sommets du graphe pour chaque relation logique existant entre les fluents et les actions. Les parties connexes du graphe ainsi obtenu désignent les sous-jeux.

Le graphe utilisé par Günther contient des fluents et des actions non instanciés, tandis que Zhao [2009] indique qu'une instanciation partielle est indispensable pour pouvoir décomposer certains jeux qui utilisent un fluent de même prédicat pour représenter différentes structures (cf. §9.4.2). Par exemple `(heap ?x ?n)` dans *Nim* représente différentes piles `x` de `n` allumettes. L'argument `x` doit être instancié pour permettre de distinguer les piles d'allumettes les unes des autres. Leur réticence à instancier complètement les règles s'explique facilement par l'absence de technique d'instanciation rapide et par les limites matérielles des machines au moment où ils ont réalisé leurs travaux. Cette contrainte étant moins forte actuellement, il convient de s'interroger sur les avantages et les inconvénients que peuvent apporter une instanciation des règles préalable à la décomposition. Nous examinons cette question dans la section suivante.

Pour ajouter les liens dans le graphe de dépendances, il est ensuite nécessaire de détecter les relations de causalité entre les actions et fluents. Nous verrons (cf. §10.1.4) que ces relations ne sont pas simples à identifier dans la description des jeux en GDL.

10.1.3 Décomposition avant ou après instanciation

Les règles instanciées peuvent être exponentiellement plus nombreuses que les règles originales en fonction du nombre de variables utilisées et de la taille de leur domaine de valeurs. L'instanciation de ces règles est une opération également coûteuse en temps. L'économie de temps et de mémoire est donc un argument plaidant en faveur d'une décomposition réalisée à partir de règles non instanciées.

Dans une description GDL, des prédicats statiques sont utilisés afin de permettre l'unification des variables pendant leur interprétation. Ces prédicats statiques deviennent inutiles une fois l'instanciation réalisée et disparaissent des règles. Schiffel et Thielscher [2007a] montrent que les règles non instanciées associées à ces prédicats statiques permettent d'identifier des structures sémantiques comme un *stepper* (fig.10.2) de manière économique. L'approche proposée par Zhao [2009] montre également que les variables utilisées dans les règles (dans la conclusion et les différents prémisses) apportent des informations utiles pour une décomposition, pour le traitement les *actions composées* notamment.

Cependant, ce type de détection, structurelle ou syntaxique, s'appuie sur des habitudes d'écriture du GDL et manque de robustesse. Dans un jeu à coups alternés, on pourrait indiquer que le compteur évolue chaque fois qu'un joueur fait l'action *noop* qui consiste à attendre son tour :

```
(<= (next (step ?x)) (true (step ?y)) (succ ?y ?x) (does ?r noop))
```

```
(succ 0 1) (succ 1 2) (succ 2 3)
(init (step 0))
(<= (next (step ?x)) (true (step ?y)) (succ ?y ?x))
```

FIGURE 10.2 – Structure sémantique d'un *stepper*. Une règle *init* indique l'état initial du compteur. Les prédicats statiques `(succ ?x ?y)` indiquent une relation d'ordre entre les *steps*. La règle `(next (step ?x))` indique l'évolution du compteur indépendamment de toute action.

La structure du *stepper* serait alors différente de celle habituellement utilisée et décrite par Schiffel. Les règles peuvent être offusquées de différentes manières ou bien pré-instanciées ce qui exclue dans ce cas la possibilité d'en extraire ces informations. Comme pour le jeu de *Nim*, différents fluents de même prédicat peuvent être utilisés pour décrire des éléments appartenant à différents sous-jeux. Ainsi, il convient de considérer chaque fluent comme un atome. Pour obtenir une décomposition optimale, comme l'indique Zhao [2009], une instanciation complète est souhaitable :

« Considering fully instantiated (i.e., ground) fluents and actions would yield the best results for subgame detection, but this is practically infeasible except for very simple games. »

Dans le cas des actions composées et/ou simultanées, une décomposition préalable peut permettre de réduire significativement le nombre de règles générées pendant l'instanciation. Par exemple, dans le jeu *Blocker Parallel*, chaque action du type `(mark ?a ?b ?c ?d)` consiste à marquer une case (a, b) dans un sous-jeu et (c, d) dans l'autre, avec a, b, c et d dans l'intervalle $\{1, 4\}$. Après instanciation, on obtient 57600 instances de la règle suivante pour une valeur donnée de `?x`, `?y` et `?mark` :

```
(<= (next (cell1 ?x ?y ?mark))
    (true (cell1 ?x ?y ?mark))
    (does blocker (mark ?xB1 ?yB1 ?xB2 ?yB2))
    (does crosser (mark ?xC1 ?yC1 ?xC2 ?yC2))
    (distinctCell ?x ?y ?xB1 ?yB1)
    (distinctCell ?x ?y ?xC1 ?yC1))
```

En décomposant les actions de *blocker* et de *crosser* avant l'instanciation i.e. en détectant les arguments sans influence sur la conclusion de la règle comme proposé par Zhao [2009], ce nombre pourrait être significativement réduit. Une approche possible serait de remplacer ces arguments par des *jokers* pendant l'instanciation pour créer des modèles d'actions instanciés une seule fois. Cependant, une telle transformation des actions nécessiterait une étape intermédiaire pour faire correspondre les modèles d'actions contenant des *jokers* avec les différentes actions (*input*) soit pour construire un *propnet* ou circuit logique tel qu'utilisé par *LeJoueur*, soit pour communiquer avec le *Game Manager* si cette représentation des actions est directement utilisée pendant le jeu.

Une approche plus simple et plus systématique consiste à éliminer les variables existentielles (cf. §9.6) comme suggéré par Schiffel [2016]. À partir de la règle précédente, on obtient 3 règles plus simples par élimination de ces variables :

```
(<= (next (cell1 ?x ?y ?mark))
    (true (cell1 ?x ?y ?mark))
    (foo ?x ?y)
    (bar ?x ?y))
```

```
(<= (foo ?x ?y)
    (does blocker (mark ?xB1 ?yB1 ?xB2 ?yB2))
    (distinctCell ?x ?y ?xB1 ?yB1))

(<= (bar ?x ?y)
    (does crosser (mark ?xC1 ?yC1 ?xC2 ?yC2))
    (distinctCell ?x ?y ?xC1 ?yC1))
```

Pour une valeur donnée de `?x`, `?y` et `?mark`, une seule instance de la première règle sera obtenue et 240 pour chacune des deux autres, soit un total de 481 règles au lieu de 57600. L'élimination des variables existentielles avant l'instanciation permet donc d'optimiser l'instanciation des règles sans passer par une décomposition préalable des actions composées [58].

Dans *LeJoueur*, l'instanciation des règles est de toute manière réalisée pour générer un circuit logique utilisé pour accélérer l'interprétation des règles pendant l'exploration d'un jeu. Ce circuit peut également être utilisé pour collecter des informations utiles à la décomposition : en remplaçant l'analyse des règles par la propagation d'un signal pour la collecte des prémisses d'un terme particulier par exemple, ou bien en réalisant rapidement des simulations pour identifier des fluents qui sont affectés simultanément par l'exécution d'une action.

Comme rien n'interdit qu'une description GDL utilise des règles totalement instanciées ou des fluents réduits à de simples atomes, que les limites techniques ont évoluées et que, de toute manière, les règles sont instanciées pour créer d'un circuit permettant de faire des simulations rapidement pendant l'exploration du jeu, nous avons fait le choix d'établir notre décomposition, i.e. notre *graphe de dépendances*, à partir de fluents et d'actions totalement instanciés.

Pour l'instanciation des règles GDL et leur transformation en circuit logique, nous nous sommes appuyés sur les travaux de Jean-Noël Vittaut. Nous avons ré-implémenté la technique d'instanciation rapide des règles avec Yap Prolog et le *tabling* proposée par Vittaut et Méhat [2014] et nous nous sommes inspirés de la version de 2015 de *LeJoueur* pour implémenter la création d'un circuit logique à partir des règles instanciées.

10.1.4 Identification des liens de causalité en GDL

Pour décomposer un jeu, l'identification des liens logiques unissant les actions et les fluents est une étape essentielle. Cependant, la formulation du GDL rend les effets des actions particulièrement difficiles à identifier. Nous présentons ici certaines spécificités du GDL et de la description des jeux qui représentent un défi pour la décomposition. Nous expliquons tout d'abord les conséquences de l'*hypothèse d'un monde clos* et du *problème du cadre*. Ensuite nous montrons que les règles décrivant un état suivant sont difficiles à analyser concernant l'effet des actions. Enfin nous indiquons pourquoi une identification exacte de l'effet des actions n'est pas envisageable.

58. Malgré le gain apporté par l'élimination des variables existentielles, celle-ci n'est pas encore implémentée dans *LeJoueur*. Cette optimisation fait partie des futures améliorations prévues.

Hypothèse d'un monde clos et problème du cadre

Le GDL ne décrit pas explicitement les effets des actions contrairement aux langages STRIPS, PDDL ou SAS+ utilisés dans le domaine de la planification. Dans ces langages, les actions sont des opérateurs définis par un ensemble de préconditions et d'effets. Les préconditions sont exprimées sous forme d'une expression fonction des fluents, les effets sont exprimés généralement sous forme d'une liste de fluents positifs et négatifs : les fluents à ajouter ou à retrancher de l'état courant pour obtenir le nouvel état. En GDL, l'état suivant est décrit par les règles `next`. Pour chaque fluent `f`, une règle de conclusion `(next f)` indique si ce fluent sera vrai dans l'état suivant en fonction de la valeur de certains fluents dans l'état courant et en fonction de certaines actions. Les effets et les préconditions de chaque action ne sont pas explicitement indiqués. Les effets négatifs (fluents retranchés à la position courante) ne sont pas non plus indiqués. Le GDL fait l'hypothèse d'un monde clos (*closed world asumption*) i.e. tout fluent `f` pour lequel `(next f)` n'est pas vrai est considéré faux par défaut dans la position suivante.

A chaque tour du jeu il est donc nécessaire de ré-évaluer la liste de tous les fluents vrais. Il n'est pas possible de connaître uniquement la liste des fluents ajoutés ou retranchés de la position courante (effets positifs et négatifs). Ceci est désigné comme le *problème du cadre* (*frame problem*). Romero *et al.* [2014] proposent une solution à ce problème passant par une réécriture des règles `next` en un ensemble de règles `next` et `fnext` décrivant respectivement les effets positifs et négatifs. Hélas, un examen attentif de leur travail montre que leur approche présente différentes limitations et n'est pas applicable à toutes les descriptions de jeux [59]. Abstraction faite de ces limitations, une réécriture des règles ne peut en aucun cas permettre de restaurer une information qui n'est pas présente i.e. la description explicite de l'effet des actions.

Effets implicites des actions

Quand bien même le *problème du cadre* pourrait-être résolu par une réécriture des règles, le lien entre les actions et les fluents ne serait toujours pas explicitement décrit. Une action présente dans les prémisses d'une règle `next` peut, selon les interprétations, avoir ou ne pas avoir de lien avec la conclusion.

Par exemple, considérons les actions légales `(does r a)`, `(does r b)` et `(does r c)` dans la règle

59. Premièrement, un pré-requis de cette approche est qu'aucune action ne doit intervenir à l'intérieur d'une négation ce qui exclue son application sur un nombre significatif de descriptions de jeux.

Deuxièmement, cette approche propose une transformation pour les *règles du cadre* i.e. les règles qui indiquent quels fluents restent non modifiés d'un état à l'autre. Ces règles sont de la forme $next(f) \leftarrow true(f) \wedge does_r \wedge cond_r$ où $does_r$ représente un ensemble d'actions positives et $cond_r$ un ensemble de conditions limitant l'instanciation des termes précédents. Ils proposent de remplacer les règles du cadre par des règles indiquant l'effet négatif survenant en cas de non conservation de l'état des fluents par la réécriture suivante : $neg(f) \leftarrow true(f) \wedge does(p,m) \wedge \bigwedge_{r \in Fr(f,m)} \neg cond_r$ où $Fr(f,m)$ est l'ensemble des règles du cadre de conclusion $next(f)$ et $does_r \subseteq \{does(p,m)\}$. La méthode n'indique pas comment transformer les règles du type `(<= (next f) (true f) (does r m))` où $cond_r = \emptyset$.

Troisièmement, la présence de prédicats auxiliaires, pourtant fréquente en GDL, n'est pas prise en compte. Les règles du cadre détectées impliquent la présence de `(true f)` dans le corps d'une clause de conclusion `(next f)`. De fait, une règle formulée de la manière suivante ne sera pas détectée comme règle du cadre : `(<= (next (f a)) (does r m) (aux a))` `(<= (aux ?x) (true (f ?x)))` La détection de la présence de `(true (f a))` dans les prémisses ne peut se faire qu'au prix d'une instanciation (au moins partielle) des règles.

Quatrièmement, cette approche présuppose qu'un fluent `f` est faux quand il n'est pas précisé dans les prémisses d'une clause de conclusion `(next f)`. Une clause du type `(<= (next f) (does r o))` n'est donc pas considérée comme une règle du cadre alors qu'elle sous-entend deux clauses : `(<= (next f) (does r o) (true f))` et `(<= (next f) (does r o) (not (true f))`.

n°	règle GDL	influence de `o`	influence de `o'`
1	`(<= (next f) (does r o))`	pos. ou conserv.	inhib. ou neg.
2	`(<= (next f) (does r o) (true f))`	conserv.	neg.
3	`(<= (next f) (does r o) (not (true f)))`	pos.	inhib.
4	`(<= (next f) (not (does r o)))`	neg. ou inhib.	conserv. ou pos.
5	`(<= (next f) (not (does r o)) (true f))`	neg.	conserv.
6	`(<= (next f) (not (does r o)) (not (true f)))`	inhib.	pos.

FIGURE 10.3 – Les 6 modèles possibles de règle `(next f)` en fonction d'une action `o` et d'un fluent `f` en prémisse. L'action `o'` est légale mais non présente dans ces règles. Pour chaque règle les deux colonnes indiquent les influences possibles de `o` et `o'` sur `f` en fonction des interprétations possibles de la règle : pos. (effet positif), neg. (effet négatif), conserv. (conservation de `f`), inhib. (inhibition de `f`)

`(<= (next f) (not (true f)) (or (does r a) (does r b))`. Les actions `a` et `b` ont un effet si la règles signifie *"La case contiendra un pion si r fait un déplacement a ou b vers cette case"*. Mais la même règle peut tout aussi bien avoir un autre sens dans un autre jeu : *"Le bateau aura coulé si le joueur fait une autre action que (does r c) qui consiste à écoper"*. Un exemple similaire peut être trouvé pour toute règle `next` contenant une action (à l'intérieur d'une négation ou non) dans ses prémisses, quelque soit l'état du fluent `f` (vrai ou faux) et sa présence ou non dans les prémisses de la règle (fig.10.3).

Il est donc possible de produire des descriptions GDL dans lesquelles les actions présentes dans le corps d'une règle `next` appartiennent à un autre sous-jeu que le fluent en conclusion ce qui constitue une grande difficulté pour la décomposition.

Si on choisit de lier systématiquement une action aux fluents qui sont modifiés par son exécution sans tenir compte des cas ou l'effet recherché est le maintien ou l'inhibition d'un fait (empêcher le bateau de couler), il reste néanmoins des règles dont la formulation empêche de faire ce lien (règles 1 et 4 de la figure 10.3). Dans une règle `(next f)`, si l'état du fluent `f` n'est pas précisé dans les prémisses, il n'est pas possible de savoir si la règle décrit les conditions dans lesquelles le fluent ne change pas (i.e. il est et reste vrai), les conditions dans lesquelles le fluent devient vrai ou bien les deux.

Identification des effets par les invariants (*mutex*)

Dans le cas ou une action présente dans une règle n'a pas d'effet, il faut chercher la ou les actions susceptibles de modifier le fluent parmi les actions non présentes dans la règle. Cependant parmi ces actions absentes de la règle, certaines ne sont pas légales au moment où la règle s'applique. Une solution pour filtrer ces actions non légales est d'identifier les fluents *mutex* du jeu et d'éliminer les actions non légales en comparant leurs prémisses et ceux de la règle `next`. Si un fluent en prémisse de la règle `(next f)` ne peut jamais être vrai en même temps qu'un fluent nécessaire à la légalité d'une action, on peut en déduire que cette action ne peut pas avoir d'influence sur le fluent `f`. Ces *mutex* pourraient également être utilisés pour identifier les hypothèses incorrectes inférées à partir d'une règle décrivant différents effets potentiels.

Différentes approches pour la détection des invariants et plus particulièrement des propositions *mutex* ont été proposées en planification (cf. §9.1.1). Une adaptation spécifique pour la recherche d'invariants dans une description de jeu en GDL a été proposée par Haufe *et al.* [2012] (cf. §9.1.3).

Pour rechercher tous les *mutex* d'un jeu multijoueur, il serait nécessaire d'ajouter une composante temporelle pour encoder $2N$ tours du jeu pour N joueurs. En effet il faudrait pouvoir vérifier que chaque couple de deux fluents résultats de deux actions distinctes d'un même joueur sont *mutex* ou non l'un de l'autre même dans les jeux à coups alternés. En posant au départ l'hypothèse que tous les fluents qui ne sont pas vrais dans la position initiale sont incompatibles deux à deux, on pourrait alors lancer un solveur comme *Clingo* [Gebser *et al.*, 2010] pour identifier les fluents qui ne sont pas *mutex* à partir de la position initiale et les éliminer de l'hypothèse. Puis, à partir d'une position aléatoire compatible avec l'hypothèse, on pourrait renouveler la recherche, éliminer les invariants contredits et itérer ainsi jusqu'à ce que plus aucun invariant ne puisse être éliminé.

Il est facile d'imaginer un jeu où l'élimination des faux invariants nécessiterait autant d'itérations qu'il y a de tours dans le jeu. Une telle recherche reviendrait à explorer tous les états du jeu. Une recherche systématique de tous les invariants serait donc bien trop coûteuse, même pour des jeux relativement simples, pour être applicable au GGP avec les contraintes de temps limité des compétitions.

Identification des liens de causalité

Puisque les règles GDL ne décrivent pas explicitement l'effet des actions et que les règles `next` décrivant l'état suivant des fluents ne permettent pas d'inférer ces effets dans un temps raisonnable, l'identification des liens de causalité entre les actions et les fluents ne peut être réalisée de manière exacte.

Les travaux précédents sur la décomposition des jeux se reposent sur les habitudes d'écriture du GDL et utilisent différentes heuristiques pour l'identification des liens de causalité entre actions et fluents. Les méthodes de décomposition que nous présentons dans cette thèse réalisent également une détection approximative de ces liens de causalité. Dans notre première approche DGFR, comme dans ces précédents travaux, nous utilisons des heuristiques pour identifier les liens de causalité d'après les règles. Notre seconde approche DGFS utilise une méthode statistique, plus fiable, fondée sur des simulations de parties.

10.1.5 Étapes du processus de décomposition

Nous avons défini ce qu'est un sous-jeu et une décomposition correcte. Nous avons expliqué notre choix d'élaborer un *graphe de dépendances* à partir de fluents et actions totalement instanciés pour l'identification des sous-jeux. Nous avons également expliqué pourquoi l'identification des liens de causalité entre actions et fluents, nécessaire pour l'ajout des arêtes dans ce graphe, ne peut être effectuée de manière exacte.

Plusieurs autres questions se posent pour l'ajout des arêtes dans notre graphe de dépendances. Comment gérer le problème posé par les actions composées i.e. comme éviter que la présence d'actions composées dans certains jeux ne mène à la création d'arêtes unissant ces actions à des fluents de différents sous-jeux ? Et comment éviter que des sous-jeux en série soient liés alors que le démarrage d'un sous-jeu est dépendant du sous-jeu précédent i.e. les actions d'un sous-jeu sont pré-conditionnées par les fluents d'un autre ?

Un autre problème que nous devons gérer, non traité dans les travaux précédents [Günther, 2007;

Zhao, 2009] car non présent dans les jeux qu'ils ont testés, est celui de la sur-décomposition qui peut survenir quand les actions ont une portée de dépendance limitée. Par exemple, dans les jeux *Tictactoe*, *Connect Four* ou *Rainbow*, les actions ne sont pré-conditionnées et n'ont d'effet que sur une seule case, colonne ou région de couleur. Les liens de dépendance entre les actions et les fluents représentés par des arêtes dans le *graphe de dépendances* ne sont donc pas suffisants pour unifier les sous-jeux.

Dans la suite de ce chapitre, nous détaillons chaque étape du processus d'identification des liens logiques permettant de construire notre *graphe de dépendances*. Pour chaque étape nous détaillons les différences entre notre approche de décomposition fondée sur les règles (DGFR) et notre approche fondée sur les simulations (DGFS), cette dernière représentant un perfectionnement de la première.

Ainsi, nous présentons deux approches pour la détection de l'effet des actions fondées respectivement sur l'analyse de la forme normale disjonctive des règles et sur des probabilités collectées pendant des simulations réalisées avec le circuit (cf. §10.2). Nous indiquons comment nous détectons les liens de préconditions en fonction de ces deux approches (cf. §10.3). Nous expliquons ensuite comment nous ajoutons un traitement des actions composées et des jeux en séries à ces deux approches (cf. §10.4). Enfin vous présentons pour chacune la phase finale d'identification des sous-jeux et un post-traitement permettant d'éviter l'éclatement des jeux dont les actions ont une portée de dépendance limitée (cf. §10.5).

10.2 Détection des effets des actions

Comme nous l'avons indiqué dans la section précédente, l'identification des liens de causalité entre l'exécution des actions et le changement des fluents est primordial pour pouvoir établir un *graphe de dépendances* afin d'identifier les sous-jeux. Cette identification ne pouvant être réalisée de manière exacte nous avons testé deux approches, l'une fondée sur l'analyse des règles, l'autre sur une analyse probabiliste des changements observés. Dans ce chapitre nous détaillons ces deux approches.

Les deux approches que nous avons testées nécessite l'instanciation complète des règles (grounding). Pour cela nous avons utilisé la technique d'instanciation rapide proposée par Jean-Noël Vittaut utilisant Yap Prolog et le principe du *tabling* (cf. §2.2). Nous nous sommes également inspirés des travaux de Jean-Noël Vittaut sur *LeJoueur* (version 2015) pour construire un graphe des règles semblable à un *propnet* à partir de ces règles instanciées. Ce graphe est ensuite simplifié puis transformé en circuit (cf. §2.4).

Dans le circuit ainsi créé, la conclusion des règles `legal`, `next`, `goal` et `terminal` constituent les sorties, et ne dépendent que des fluents (`true`) et actions (`does`) en entrée. Les expressions logiques ou *prédicats auxiliaires* utilisés dans le corps des règles sont représentés par les portes internes du circuit.

10.2.1 Par analyse des règles

Notre première approche DGFR procède à l'identification des relations de causalité entre actions et fluents en se fondant sur l'analyse des règles.

Pour l'analyse de ces relations nous utilisons les définitions suivantes :

Definition 10.4. *Soit F l'ensemble des fluents instanciés f apparaissant dans les expressions* `true`(f) *ou* $\neg$`true`(f).

Definition 10.5. *Soit R l'ensemble de tous les rôles r et O l'ensemble de toutes les options o de ces rôles,* $A \subset R \times O$ *est l'ensemble de toutes les actions instanciées des joueurs* $a = \texttt{does}(r, o)$.
O_r *est l'ensemble de toutes les options possibles pour un rôle r.*

Definition 10.6. *Soit C l'ensemble de toutes les conjonctions possibles d'atomes de la forme* $\texttt{true}(f)$, $\neg\texttt{true}(f)$, $\texttt{does}(r, o)$ *ou* $\neg\texttt{does}(r, o)$.

Après une étape de pré-traitement, réalisée pendant la construction du circuit, visant à calculer la forme normale disjonctive des règles et identifier les prédicats auxiliaires, nous utilisons une heuristique pour la détection de l'effet des actions. Une analyse des mouvements joints des joueurs permet ensuite d'identifier quelle action, quand plusieurs sont présentes dans une même règle, est responsable d'un effet décrit sur un fluent. Nous verrons que l'efficacité de cette approche dépend fortement de la qualité de l'heuristique utilisée.

Pré-traitement du circuit

Notre approche DGFR nécessite de disposer de la forme normale disjonctive des règles et d'identifier d'éventuels prédicats auxiliaires dans la description du jeu. Nous réalisons ces deux étapes durant la création du circuit logique représentant le jeu.

Afin d'obtenir les règles sous une forme canonique, pendant la construction du circuit nous parcourons le graphe des règles des entrées vers les sorties en remplaçant systématiquement chaque porte logique par l'expression correspondante sous forme normale disjonctive exprimée uniquement en fonction des fluents et actions en entrée. Par la suite nous dirons que ces expressions sont sous forme normale disjonctive développée (DNFD) [60].

Le calcul de la DNFD de toutes les règles pouvant être très lourds, nous avons également testé une procédure légèrement simplifiée qui consiste à ne pas remplacer les prédicats auxiliaires utilisés dans la description d'un jeu par l'expression sous DNFD correspondante, mais à conserver ces prédicats auxiliaires sous forme d'atomes dans les expressions logiques. Nous obtenons alors des règles sous forme normale disjonctive pouvant contenir des prédicats auxiliaires. Par la suite nous dirons que ces expressions sont sous forme normale disjonctive simple (DNF) [61].

Les prédicats auxiliaires utilisés pour décrire un jeu sont un indice fort de sa structure : *« a game that is succinctly described will necessarily be described in terms of its most salient features. »* [Clune, 2007]. On peut cependant décrire un jeu de manière à ne pas les utiliser, même si la description peut être exponentiellement plus volumineuse. Rien dans les spécifications GDL n'interdit de produire une description de jeu utilisant un nombre réduit ou aucun prédicat auxiliaire. Ils sont alors remplacés par des expressions logiques utilisant les prédicats `true` et `does` dans les prémisses des règles.

Pour limiter le développement des DNFD et obtenir des expressions sous DNF, nous devons pourtant nous assurer d'identifier ces prédicats auxiliaires en les reconstituant le cas échéant. Pour s'assurer de reconstituer ces prédicats auxiliaires s'ils ne sont pas présents dans la description du jeu, nous avons, en plus de la factorisation des conjonctions et des disjonctions opérée pour la simplification du graphe de

60. Disjunctive Normal Form, fully Developed (DNFD).
61. Disjunctive Normal Form (DNF).

règles, utilisé les lois de De Morgan pour réduire le nombre de négations dans le circuit. Comme une factorisation parfaite est un problème NP-hard, notre programme utilise une approche gloutonne où le premier facteur commun est utilisé. Les phases de factorisation et d'application des lois de De Morgan sont itérées jusqu'à ce que le circuit atteigne une taille minimale.

Les prédicats auxiliaires, représentant des expressions importantes dans la logique du jeu, correspondent à des portes logiques internes du circuit reconstituées le cas échéant par cette procédure. Nous identifions ces portes logiques internes au fait qu'elles ne dépendent directement ou indirectement en entrée que de fluents (`true`) et non d'actions (`does`) et sont utilisées plusieurs fois i.e. plusieurs portes du circuit utilisent leur valeur de sortie.

Notre première approche utilise également les prédicats auxiliaires comme `line`, `column`, `open` ou `gameover` pour identifier les objectifs des joueurs ou la séparation entre des sous-jeux en série. Cette phase de factorisation supplémentaire est donc utilisée même dans le cas de la procédure utilisant les DNFD.

Utilisation d'une heuristique

Identifier un potentiel lien entre une action et le changement d'un fluent alors que les effets ne sont pas explicitement décrits ne peut être fait de manière exacte. La première approche que nous avons expérimentée pour traiter ce problème consiste à utiliser des heuristiques similaires à celles proposées par Günther *et al.* [2009] pour l'analyse des règles. Nous présentons ici notre heuristique. Dans la section suivante nous indiquons comment nous identifions une action potentiellement responsable d'un effet en présence d'actions de joueurs distincts dans une règle.

Günther *et al.* [2009] proposent de considérer qu'une action a a un effet négatif sur un fluent f si aucune règle n'indique que cette action préserve la valeur de ce fluent i.e. si $\texttt{next}(f)$ ne contient pas $\texttt{true}(f) \wedge \texttt{does}(a)$ dans ses prémisses. Cependant, parmi les actions non présentes dans ce type de règle certaines peuvent appartenir à un autre sous-jeu que le fluent et peuvent être illégales quand la règle de conservation s'applique.

Par exemple, dans un jeu parallèle synchrone comme *Double Tictactoe*, les axiomes du cadre indiquent que l'état d'une case est conservé uniquement si l'on joue une action du même sous-jeu que cette case et que cette action ne concerne pas la case en question. L'état de la case est également conservé si le joueur joue une action sans effet alors que c'est à son tour de jouer :

```
(<= (next (blank ?m ?board)) (does ?w (mark ?n ?board))
    (true (blank ?m ?board)) (distinct ?m ?n))

(<= (next (blank ?m ?board)) (does ?w noop)
    (true (blank ?m ?board)) (true (control ?board ?w)))
```

Si l'on considère que les actions non présentes dans ces axiomes du cadre ont un effet négatif sur la conclusion de la règle, on crée un lien entre les fluents d'un sous-jeu et les actions de l'autre sous-jeu. Günther *et al.* [2009] ne considèrent que les actions de même prédicat pour éviter cet écueil ce qui rend leur approche très dépendante de la syntaxe de la description. En effet, dans cette version de *Double*

Tictactoe [62], le même prédicat `mark` est utilisé pour les actions des deux sous-jeux et c'est l'argument `?board` qui permet de distinguer les actions agissant sur chacun des deux plateaux de jeu. Ce jeu met donc leur approche en échec.

Dans notre approche DGFR, nous utilisons une heuristique légèrement différente, permettant de ne pas s'appuyer sur la syntaxe, dont nous proposons la définition suivante :

Definition 10.7. *Le fluent f est un* ***effet négatif potentiel*** *de l'action $a = (r, o)$ si la règle* `next`(f) *sous forme DNFD a une clause ou* $\neg$`does`(r, o) *apparaît.*
Le fluent f est un ***effet positif potentiel*** *de l'action $a = (r, o)$ si la règle* `next`(f) *sous forme DNFD a une clause contenant le littéral* `does`(r, o) *et ne contenant pas le littéral* `true`(f).

Analyse des mouvements joints

A la difficulté précédente s'ajoute celle provoquée par l'action simultanée de tous les joueurs. À chaque tour, tous les joueurs choisissent une action. Même quand un effet est clairement décrit (i.e. la règle de conclusion `(next f)` possède `(not (true f))` parmi ses prémisses), si des actions de différents joueurs interviennent dans les prémisses, un doute persiste sur l'effet de chacune. Par exemple dans la règle suivante :

```
(<= (next f) (does r1 o1) (does r2 o2) (does r3 o3) (not (true f)))
```

Quelle action est responsable du basculement de `f` à vrai ? Une seule, deux, toutes ?

Il est nécessaire d'identifier la ou les actions responsables d'un effet dans le cas où une clause d'une règle `(next f)` contient un mouvement joint de plusieurs joueurs i.e. plusieurs actions (`does`) afin d'éviter de lier par erreur une action au changement d'un fluent appartenant à un autre sous-jeu.

Pour résoudre ce problème, Zhao *et al.* [2009] proposent de comparer les arguments utilisés dans la conclusion d'une règle `next` avec ceux utilisés dans les options des différents joueurs. Par exemple, dans la règle suivante du jeu *Blocker Serial*, nous pouvons voir que l'option de *crosser* est la seule susceptible d'affecter la conclusion :

```
(<= (next (cell2 ?xc ?yc crosser))
    (does crosser (mark2 ?xc ?yc))
    (does blocker (mark2 ?xb ?yb))
    (distinctCell ?xc ?yc ?xb ?yb))
```

Cependant, il est possible de s'affranchir des arguments, d'utiliser des règles complètement instanciées et de simples atomes pour représenter les fluents et les actions. Par exemple, nous pouvons remplacer les règles précédentes par un ensemble de règles instanciées :

```
(<= (next f) (does crosser o1) (does blocker o2))
(<= (next f) (does crosser o1) (does blocker o3))
(<= (next f) (does crosser o1) ...
```

Avec des fluents comme `f` et des actions comme `(does r o)`, leur approche ne permet plus de gérer le problème posé par les mouvements joints. Pour identifier quelle action est responsable d'un effet

62. Cette version de *Double Tictactoe* provient du dépôt *Base* de Schreiber [2017] : `http://games.ggp.org/base/`

sans s'appuyer sur la syntaxe, nous proposons de comparer, pour chaque joueur, les différentes options utilisées en conjonction avec les mêmes fluents et actions des autres joueurs dans les clauses d'une règle `next`.

Supposons que $\texttt{next}(f) \leftarrow C_f$ est sous forme DNFD et considérons une option spécifique o' du joueur r'. Nommons $E(o')$ l'ensemble des différentes options du rôle r quand r' choisi l'option o' :

$$E(o') = \{\ o \in O_r \mid \exists c \in C_f, \exists b \in C, \\ c = \texttt{does}(r, o) \wedge \texttt{does}(r', o') \wedge b\ \}$$

Nous définissons $E(o)$ de la même manière en échangeant les rôles de (r, o) avec (r', o').

Si toutes les options de r sont présentes en conjonction avec les mêmes actions de r', ces options ont probablement pas d'effet i.e. le résultat est le même quelque soit l'action choisie. Au contraire, si une unique option de r est présente, elle est probablement responsable de l'effet observé. Nous utilisons donc les heuristiques suivantes :

Definition 10.8. *L'action $a = (r, o) \in A$ est **potentiellement responsable d'un effet** sur f si :*

- $card(E(o')) = 1$, *ou*
- $E(o') \subsetneq O_r$ *et* $card(E(o)) \neq 1$

Prenons par exemple cet ensemble de règles instanciées tirées du jeu *BlockerSerial* :

```
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 1 1)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 1 2)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 1 3)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 1 4)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 2 1)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 2 2)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 2 4)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 3 1)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 3 2)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 3 3)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 3 4)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 4 1)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 4 2)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 4 3)))
(<= (next (cell1 2 3 crosser)) (does crosser (mark1 2 3)) (does blocker (mark1 4 4)))
```

Le terme `(next (cell1 2 3 crosser))` est vrai si *blocker* choisit n'importe quelle option sauf `(mark1 2 3)` et *crosser* choisit l'option `(mark1 2 3)`. Toutes les options de *blocker* ne sont pas représentées, mais comme *crosser* n'a qu'une seule option possible, cette action est considérée responsable de l'effet tandis que les actions de *blocker* ne sont pas liées au fluent `(cell1 2 3 crosser)`.

Même si cette approche écarte parfois des actions qui sont vraiment liées à la conclusion de la règle `next`, cela ne provoque pas de sur-décomposition sur l'ensemble des jeux testés. En effet, au moins une action est liée à la conclusion et les liens entre les fluents et les actions qui sont ajoutés dans le graphe de dépendances par la suite pour représenter les relations de précondition sont redondants avec ceux représentant les relations d'effet.

Dans notre approche simplifiée utilisant les DNF des règles, nous comparons de même, pour chaque

joueur, les différentes options utilisées en conjonction avec les mêmes prémisses dans les clauses d'une règle `next`, à la différence que ces prémisses peuvent être des fluents, actions ou des prédicats auxiliaires.

Ainsi, un fluent est un *effet potentiel* d'une action si cette action a un *potentiel effet positif ou négatif* sur ce fluent et qu'elle est *potentiellement responsable de cet effet* dans le cas de mouvements joints. À partir de ces effets potentiels des actions, nous pouvons déduire les fluents qui sont indépendants des actions comme les fluents représentant les *steps* ou le *control* et les actions qui sont indépendantes des fluents comme les actions *noop*.

Definition 10.9. *Un fluent f est* ***indépendant des actions*** *s'il n'est l'effet potentiel d'aucune action a.*

Une action a est ***indépendante des fluents*** *si aucun fluent f n'est l'effet potentiel de cette action.*

Limite de l'heuristique

L'heuristique utilisée est limitée à la détection des effets explicites des actions pour éviter au maximum toute erreur d'analyse qui entraverait la décomposition. Dans la plupart des descriptions de jeux, l'évolution de l'état du jeu est décrit en fonction des actions qui modifient une partie des fluents et non en fonction des actions qui gardent les autres inchangés. Notre heuristique est donc efficace pour une grande partie des jeux décrits en GDL.

Cependant, s'appuyer sur cette habitude d'écriture rend l'heuristique fragile. Ignorer les effets implicites des actions est problématique pour certains jeux. Par exemple, dans le jeu *Chomp*, chaque action d'un des joueurs consiste à croquer dans une tablette de chocolat : l'état du jeu est décrit en fonction des carrés de chocolat non affectés par l'action du joueur. Dans ce jeu, presque qu'aucun effet explicite des actions n'est décrit en dehors de l'empoisonnement du joueur qui aura le malheur de croquer le dernier carré de la plaque. Tous ces liens de causalité non détectés amènent à créer un graphe de dépendances totalement éclaté. Heureusement, la faible connectivité du graphe de dépendances permet, dans ce type de jeu, de détecter un problème. La décomposition du jeu est considérée non fiable et est écartée.

Dans le but d'obtenir des informations plus robustes sur les liens de causalité entre actions et fluents, prenant également en compte les effets implicites des actions, nous avons exploré une seconde approche fondée sur une analyse statistique d'informations collectées pendant des *playouts*.

10.2.2 Par analyse statistique

Notre seconde approche DGFS procède à l'identification des relations de causalité entre actions et fluents, en se fondant sur une analyse statistique des changements observés dans l'état du jeu, en fonction des actions jouées lors de simulations aléatoires de parties (*playouts*).

Nous définissons les ensembles F des fluents et A des actions des joueurs comme précédemment (def.10.4 et def.10.5). L'effet des actions n'étant pas explicitement décrit dans les règles GDL, nous définissons un *effet* comme étant une observation constatée lors d'une transition dont la cause n'est pas connue *à priori* :

Definition 10.10 (Effet positif ou négatif)**.** *Un effet positif f^+ (resp. négatif f^-) est le changement de valeur d'un fluent de* faux *(resp.* vrai*) à* vrai *(resp.* faux*) observé durant une transition d'un état à un autre. Un effet quelconque (positif ou négatif) observé pour un fluent $f \in F$ est représenté par f^*.*

Certains effets surviennent simultanément. Nous distinguons deux types d'effets cooccurrents :

Definition 10.11 (Effets Globalement Cooccurrents (GCE[63])). *L'ensemble des effets globalement co-occurrents avec f^* notés $GCE(f^*)$ est composé des effets qui surviennent simultanément avec f^* dans toutes les transitions explorées du jeu. Notons $|GCE(f^*)|$ le nombre de fois que cette cooccurrence a été observée durant les playouts.*

Definition 10.12 (Effets Cooccurrents par Actions (ACE[64])). *L'ensemble des effets cooccurrents par action avec f^* notés $ACE(a, f^*)$ est composé des effets qui surviennent simultanément avec f^* dans les transitions explorées du jeu où l'action $a \in A$ est jouée. Notons $|ACE(a, f^*)|$ le nombre de fois que cette cooccurrence a été observée durant les playouts.*

Après avoir évalué la probabilité d'observer un changement suite à une action donnée, nous effectuons un filtrage de ces corrélations pour en déduire les liens de causalité entre actions et fluents. Un premier filtrage est effectué d'après les règles GDL, puis un second filtrage est effectué en utilisant les *effets cooccurrents*.

Probabilité d'effet des actions

Pour identifier le lien de causalité entre les actions et les fluents, nous réalisons des *playouts* et collectons des informations sur les effets observés. À chaque tour, chaque joueur doit choisir une action et une seule. L'ensemble des actions des différents joueurs constitue un mouvement joint. Dans les jeux à coups alternés, à un tour donné du jeu, un seul joueur a le choix entre plusieurs actions légales tandis que les autres ne peuvent jouer qu'une seule action sans effet (souvent nommée *noop*). Pour chaque transition d'un *playout*, nous collectons le mouvement joint associé aux effets observés. Après un nombre donné de simulations, pour chaque action a qui apparaît dans $|J(a)|$ mouvements joints distincts, nous calculons la probabilité $\mathcal{P}(a, f^+)$ qu'une action a soit suivie d'un effet positif sur le fluent f :

$$\mathcal{P}(a, f^+) = (\sum_{0<i<|J(a)|} \frac{E(J(a)_i, f^+)}{O(J(a)_i)})/|J(a)|$$

avec $|J(a)$ l'ensemble des mouvements joints incluant l'action a, $O(J(a)_i)$ le nombre d'occurrences d'un mouvement joint de cet ensemble, $E(J(a)_i, f^+)$ le nombre de fois où un effet f^+ a suivi l'application de ce mouvement joint. La probabilité qu'une action a soit suivie d'un effet négatif f^- est calculée de manière similaire.

$\mathcal{P}$ indique la probabilité d'observer un effet suite à une action. Cependant, certains fluents utilisés pour définir l'alternance des joueurs (*control*) ou pour définir un compteur de coups (*step*) changent à chaque tour du jeu quelque soit l'action jouée : ils sont *indépendants des actions*. La formulation des règles ne permet pas toujours de les détecter. Par exemple, la présence d'une action *noop* parmi les prémisses d'une règle `next` décrivant l'état suivant d'un fluent de type *control* ou *step*, peut laisser penser que l'action *noop* a un effet et que *control* ou *step* sont dépendants d'une action[65].

63. Globally co-occurring effects (GCE).
64. Action co-occurring effects (ACE).
65. *noop*, *step* et *control* ne sont pas des mots clefs du GDL. En compétition, les règles sont obfusquées ce qui ne permet pas d'identifier ces éléments par leur nom.

Filtrage des effets d'après les règles

Une valeur positive de $\mathcal{P}(a, f^*)$ n'indique pas si l'action a est responsable d'un effet sur f ou s'il s'agit d'une simple corrélation. L'étape suivante pour l'identification des liens de causalité consiste à examiner les relations potentielles suggérées par une valeur positive de $\mathcal{P}$ pour filtrer les liens de causalité et éliminer les corrélations.

Nous commençons par détecter la présence de coups alternés : notre but est d'identifier les actions *noop* sans effet et les fluents *control* indépendants des actions. Un jeu à n joueurs est un jeu à coups alternés si à chaque tour $n-1$ joueurs n'ont qu'une seule action possible qui peut alors être considérée comme une action *noop*. Le fluent qui a la plus grande probabilité de changer quand une action *noop* est jouée est le fluent *control* correspondant. Ceci permet de détecter les actions *noop* et les fluents *control*. Si `(next f)`, indiquant qu'un fluent `f` est vrai à l'état suivant, ne compte que des actions *noop* parmi ses prémisses, alors ce fluent est considéré *indépendant des actions*.

Pour chaque action a nous vérifions toutes les relations potentielles suggérées par $\mathcal{P}(a, f^*) > 0$ afin de confirmer ou infirmer une relation de causalité entre a et f^*. En observant les règles du jeu, il est possible de déterminer si une règle décrit un lien explicite ou sous-entend un lien implicite de causalité entre une action et un effet. Si aucune règle de ce type n'existe dans la description GDL, on peut conclure qu'aucun lien de causalité ne peut exister entre l'action et l'effet : la probabilité $\mathcal{P}$ est alors remise à 0.

Par exemple, une règle `(<= (next f) (does r o) (not (true f)))` indique explicitement que l'action a `=(does r o)` est responsable d'un effet positif sur `f`. Dans une règle, `(<= (next f) (does r o) (true f))` un effet négatif potentiel provoqué par toute autre action que `(does r o)` est suggéré. Enfin, la règle `(<= (next f) (does r o))` laisse supposer une relation possible de causalité avec toute action selon la valeur courante de `f`.

Filtrage par comparaison des effets cooccurrents

En examinant les effets cooccurrents, il est possible de filtrer les liens de causalité non cohérents. Pour chaque lien de causalité éliminé, les différents effets cooccurrents sont reconsidérés jusqu'à ce qu'il ne soit plus possible d'en éliminer de nouveaux. Une action a a une probabilité nulle d'être la cause d'un effet f^* si $\mathcal{P}(a, g^*) = 0$ avec $g^* \in GCE(f^*)$. Au contraire le lien entre a et f^* est confirmé s'il existe un g^* avec $\mathcal{P}(a, g^*) > 0$ et soit $g^* \in GCE(f^*)$ avec $|GCE(f^*)| > \Psi$ ou $g^* \in ACE(a, f^*)$ avec $|ACE(a, f^*)| > \Theta$, ou Ψ et Θ sont des seuils nécessaires pour ne considérer que les cooccurrences réelles et écarter les coïncidences.

Pour les jeux multijoueurs, nous comparons également les probabilités attribuées aux différentes actions d'un mouvement joint pour un même effet f^*. Si une action a d'un mouvement joint a une probabilité Φ fois supérieure à celle d'une action b d'être suivie de l'effet f^* alors a est considéré comme la cause de cet effet et $\mathcal{P}(b, f^*) = 0$.

Si un effet intervient systématiquement dans toute transition depuis la position initiale et jamais dans une autre position du jeu, il est considéré *indépendant des actions*. Par exemple, dans le jeu *Dual Hamilton*, chaque sous-jeu démarre par le même changement. Le joueur quitte la position initiale vers une autre destination : toutes les actions ont donc le même effet négatif sur les fluents de la position

initiale. Cependant, un effet négatif est également détecté entre chacune de ces actions et la position initiale de l'autre sous-jeu puisque ce changement intervient systématiquement. Il est donc nécessaire d'ignorer ces effets qui interviennent systématiquement et uniquement dans la première transition du jeu pour éviter de lier les actions aux fluents d'un autre sous-jeu.

Quand tous les liens de causalité ont été vérifiés, si aucune action n'a été identifiée comme responsable d'un effet, le fluent correspondant est étiqueté comme *indépendant des actions*.

10.2.3 Bilan de la détection des liens de causalité

Nous avons présenté deux approches pour l'identification de l'effet des actions i.e. des liens de causalité entre les actions et le changement de valeur des fluents dans les descriptions GDL des jeux. La première approche fondée sur une analyse des règles nécessite l'utilisation d'une heuristique. Cette heuristique ne prenant en compte que les effets explicitement décrits dans les règles GDL, l'approche manque de robustesse et ne détecte qu'un nombre limité d'effets dans certains jeux.

La seconde approche fondée sur une analyse statistique du changement des fluents en fonction des actions jouées durant des *playouts* permet une détection plus robuste. Les informations collectées pendant les *playouts* permettent de ne pas faire de distinction entre les effets négatifs ou positifs ou entre les effets explicitement ou implicitement décrits dans les règles. Cette approche permet d'affiner la détection de l'effet des actions progressivement en fonction du nombre de simulations et de la quantité d'information collectée. Elle pourrait également permettre une détection progressive des effets en fonction de la profondeur des simulations. Les actions inexplorées sont temporairement mises de côté et ne sont rattachées à aucun sous-jeu en particulier en attendant que des informations les concernant soient collectées.

10.3 Détection des liens de précondition

Dans l'objectif de construire un *graphe de dépendances* entre les actions et les fluents, une fois les effets des actions identifiés, il convient d'identifier les fluents préconditions de ces actions.

Le GDL fait la distinction entre la description des actions légales, définies par le prédicat `legal`, et la description de l'état suivant de chaque fluent du jeu, défini par le prédicat `next`, en fonction des actions choisies par les joueurs. Ainsi une action possède deux types de préconditions : les préconditions conditionnant sa légalité et les préconditions conditionnant son influence sur l'état suivant du jeu. Il est donc nécessaire d'analyser ces deux types de règles pour en déduire les différents fluents-préconditions des actions.

Comme dans les approches de Günther *et al.* [2009] et Zhao *et al.* [2009], les fluents indépendants des actions font l'objet d'un traitement spécifique : les liens entre ces fluents et les actions sont évités pour empêcher qu'ils ne relient tous les sous-jeux.

Nous avons utilisé deux approches distinctes pour la détection des liens de précondition en fonction de la méthode de décomposition utilisée. L'approche DGFR nécessitant le calcul de la forme normale disjonctive des règles (DNF/DNFD), nous avons utilisé cette information déjà disponible pour identifier les préconditions. Dans la seconde procédure DGFS, cette information n'étant pas disponible, nous avons

utilisé une autre méthode s'appuyant sur la propagation et la rétro-propagation de différents signaux dans le circuit.

10.3.1 A partir des DNF/DNFD

Une fois les effets potentiels des actions identifiés, ainsi que les actions indépendantes des fluents et les fluents indépendants des actions, nous pouvons identifier les préconditions des actions appartenant au même sous-jeu :

Definition 10.13. *Le fluent f est une* ***précondition potentielle dans le même sous-jeu*** *qu'une action $a = (r, o)$ si :*

- *a n'est pas indépendant des fluents et*
- *f n'est pas indépendant des actions et*
- *une des conditions suivantes est respectée :*
  - `legal`(r, o) *sous DNFD possède une clause où* `true`(f) *ou* $\neg$`true`(f) *apparaît, ou*
  - *il existe f', potentiel effet de a, tel que* `next`(f') *sous DNFD possède une clause contenant* `does`$(r, o) \wedge$ `true`(f) *ou* `does`$(r, o) \wedge \neg$`true`(f).

Un fluent indépendant des actions peut être présent dans les prémisses de toutes les règles `legal`, il constitue alors une précondition de toutes les actions, mais il appartient à un autre sous-jeu qui est indépendant des actions.

Dans la version de notre programme utilisant les DNFD des règles, les préconditions d'une action peuvent inclure des prédicats auxiliaires. Dans ce cas, tous les fluents composant l'expression définie par le prédicat auxiliaire sont des préconditions de l'action à l'exception des fluents indépendants des actions.

Pour identifier les sous-jeux indépendants des actions, il convient d'identifier les liens de préconditions existant entre leurs fluents. La valeur des fluents indépendants des actions dans un état du jeu ne dépend que des fluents présent dans l'état précédent. Pour identifier ces préconditions, il suffit d'examiner les fluents présents dans les prémisses des règles `next` pour chacun des fluents indépendants des actions.

Definition 10.14. *Le fluent f est une* ***précondition du fluent g indépendant des actions****, si f est indépendant des actions et* `next`(g) *sous DNFD possède une clause où* `true(f)` *ou* $\neg$`true`(f) *apparaît.*

10.3.2 A partir du circuit

En l'absence d'une forme canonique des règles, l'identification des préconditions des actions dans les règles `legal` et `next` nécessiterait d'examiner récursivement chaque prédicat auxiliaire utilisé dans le corps de ces règles pour collecter l'ensemble des prémisses. La diffusion d'un signal dans le circuit permet d'effectuer cette identification plus rapidement. Pour identifier les liens existants entre les fluents indépendants des actions et les préconditions, il est également possible d'utiliser le circuit logique.

Cette approche présente un avantage par rapport à celle utilisant les DNF/DNFD : l'utilisation du circuit permet d'évaluer directement les interactions logiques complexes existant entre les différentes prémisses d'une règle et d'identifier des *prémisses contradictoires ou exclusives*[66].

Definition 10.15 (Prémisse)**.** *Soit h la conclusion d'une règle GDL totalement instanciée. $g \in F \cup A$ est une prémisse de h si :*

- *g est dans le corps de cette règle, ou*
- *g est dans le corps d'une règle instanciée de conclusion i avec i une prémisse de h.*

*g est une **prémisse non-contradictoire** de h si aucune contradiction n'existe entre g et une autre prémisse de h, i.e. si h peut être vraie quand g est vraie. g est une **prémisse exclusive** de h si h est vraie quand g est vraie quelque soit la valeur des autres prémisses.*

Par exemple, dans le jeu *TicTacHeaven*, les expressions `(true (mark_large 1 1 1 1 x))` et `(not (true (mark_large 1 1 1 1 x)))` sont présentes parmi les prémisses de `(legal xplayer (play_large 1 1 1 1 x))`[67]. En effet, la description GDL instanciée du jeu contient la règle suivante décrivant les conditions sous lesquelles l'action est légale :

```
(<= (legal xplayer (play_large 1 1 1 1 x))
    cur_board_closed_large (empty_cell_large 1 1 1 1)
    (true (control_large xplayer)) )
```

Dans cette règle, `cur_board_closed_large` indique que si le plateau de jeu courant est plein, il est possible de jouer sur un autre plateau. Le plateau de jeu courant pourrait être `(current_board 1 1)`, mais cela doit être faux dans le cas présent car cela serait en contradiction avec `(empty_cell_large 1 1 1 1)` qui indique que la case doit être vide pour que l'action soit légale. Nous pouvons donc trouver `(true (mark_large 1 1 1 1 x))` parmi les prémisses de `cur_board_closed_large`, mais ce fluent ne permet pas à `(legal xplayer (play_large 1 1 1 1 x))` d'être vrai.

Pour identifier les préconditions à partir du circuit, nous procédons donc de la manière suivante. Pour chaque fluent `f` indépendant des actions, la sortie `(next f)` est marquée *vraie* puis le signal est rétro-propagé dans le circuit avec 4 états possibles (*indéfini*, *vrai*, *faux*, *vrai-et-faux*). Les fluents activés en entrée indiquent les prémisses de `(next f)` et leur valeur : prémisse utilisée telle quelle, dans une négation ou les deux.

Chaque prémisse `g` est ensuite vérifiée avec une logique à 3 états : si elle permet de changer la valeur de `(next f)` de *faux* à *vrai*, il s'agit bien d'une précondition de `f` et non d'une précondition contradictoire, sinon, si elle force `f` à rester *vrai*, le fluent inverse `(not g)` est responsable d'un effet négatif sur `f`.

L'identification des préconditions des fluents indépendants des actions est suffisante pour identifier les sous-jeux indépendants des actions.

66. Ces notions de *prémisses contradictoires ou exclusives* seront également utiles pour la détection des actions composées (cf. 10.4.1)

67. Les deux premiers arguments de `mark_large` représentent les coordonnées du plateau de jeu, les deux suivants représentent les coordonnées de la case à marquer sur ce plateau.

10.4 Jeux parallèles à actions composées et jeux en série

Les jeux à actions composées et les jeux en série représentent des cas particuliers difficiles à décomposer. Les actions composées ont la propriété d'avoir un effet simultané sur l'état de sous-jeux distincts : séparer ces effets est un prérequis pour pouvoir séparer les sous-jeux. Dans les jeux en série, le début d'un sous-jeu est conditionné par l'achèvement du précédent, ceci créant un lien logique fort entre les fluents décrivant l'état des sous-jeux.

Dans ce chapitre nous présentons différentes approches pour traiter ces deux cas particuliers.

10.4.1 Actions composées et méta-actions

Une action composée est une action regroupant plusieurs sous-actions appartenant chacune à un sous-jeu distinct. Par exemple, dans le jeu *Asteroid Parallel* l'action composée `(legal ship (do clockcounter))` correspond à l'action `clock` dans un premier sous-jeu et à l'action `counter` dans un second. Dans notre graphe de dépendances, une telle action est représentée par un seul sommet. Ce sommet est alors relié par des liens de précondition et d'effet à différents fluents appartenant aux deux sous-jeux, créant ainsi un seul groupe connexe, ce qui fait échouer la décomposition.

Zhao *et al.* [2009] utilisent une analyse syntaxique des règles pour détecter les actions composées. Par exemple, dans la règle suivante de *Tictactoe Parallel*, nous pouvons voir que les deux premiers arguments de l'action sont reliés à la conclusion de la règle alors que les deux arguments suivants sont inutiles :

```
(<= (next (cell1 ?x1 ?y1 o))
    (does oplayer mark(?x1 ?y1 ?x2 ?y2)))
```

Cependant, comme nous l'avons vu (cf. §9.4.3), cette détection totalement tributaire de la syntaxe ne fonctionne pas pour *Asteroid Parallel* ou tout autre jeu dans la description duquel les actions composées ne présentent pas des arguments séparés.

Pour réaliser une détection indépendante de la syntaxe, nous nous appuyons sur la constatation suivante : dans les jeux à actions composées, l'ensemble de toutes les actions est une combinaison des ensembles des actions de chaque sous-jeu. Par exemple, dans un jeu comprenant deux sous-jeux, pour chaque action du premier sous-jeu, il y a N actions composées correspondant à la combinaison de cette action avec une des N actions possibles de l'autre sous-jeu. Pour isoler les différentes composantes des actions composées, il est possible de regrouper les actions dans des méta-actions i.e. des ensembles d'actions ayant un même effet dans les mêmes circonstances dans l'un des sous-jeux. Ces circonstances correspondent aux préconditions des actions dans les règles `legal` et `next`.

Par exemple dans le jeu *Tictactoe Parallel*, on peut identifier les actions suivantes qui ont toutes un effet positif sur le fluent `(cell1 1 3 o)` et qui interviennent toutes avec un ensemble vide de préconditions dans la règle de conclusion `(next (cell1 1 3 o))` :

```
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 1 1)))
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 1 2)))
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 1 3)))
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 2 1)))
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 2 2)))
```

```
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 2 3)))
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 3 1)))
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 3 2)))
(<= (next (cell1 1 3 o)) (does oplayer (mark 1 3 3 3)))
```

Ces actions sont pré-conditionnées par différents fluents dans les règles `legal` :

```
(<= (legal oplayer (mark 1 3 1 1)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 1 1 b)))
(<= (legal oplayer (mark 1 3 1 2)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 1 2 b)))
(<= (legal oplayer (mark 1 3 1 3)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 1 3 b)))
(<= (legal oplayer (mark 1 3 2 1)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 2 1 b)))
(<= (legal oplayer (mark 1 3 2 2)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 2 2 b)))
(<= (legal oplayer (mark 1 3 2 3)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 2 3 b)))
(<= (legal oplayer (mark 1 3 3 1)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 3 1 b)))
(<= (legal oplayer (mark 1 3 3 2)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 3 2 b)))
(<= (legal oplayer (mark 1 3 3 3)) (true (control oplayer))
    (true (cell1 1 3 b)) (true (cell2 3 3 b)))
```

Comme une action composée réunit différentes actions en une, les préconditions présentes dans ces règles `legal` correspondent aux conditions nécessaires pour que les différentes sous-actions soit légales. Pour identifier les méta-actions, il convient donc considérer ces différentes préconditions indépendamment les unes des autres. Une méta-action regroupe donc des actions qui ont au moins une précondition en commun dans leurs règles `legal` (ici `(true (cell1 1 3 b))`) ou un ensemble vide de préconditions. Les préconditions indépendantes des actions, comme ici le fluent `(control oplayer)`, doivent être écartées pour éviter qu'elles ne relient tous les actions entre elles.

Comme nous utilisons les effets détectés et les préconditions pour l'identification des méta-actions, nous avons utilisé deux approches distinctes en fonction de la procédure de décomposition utilisée. La première (DGFR) tire parti des effets et des préconditions détectées à partir de la forme normale disjonctive des règles (DNF/DNFD), l'autre (DGFS) utilise l'identification statistique des effets et la propagation ou rétro-propagation de différents signaux dans le circuit pour identifier les préconditions.

A partir des règles

La détection des méta-actions peut être effectuée à partir des règles sous DNFD après une détection préalable des effets potentiels des actions et des fluents indépendants des actions. Une action peut appartenir à une ou plusieurs méta-actions qui dépendent uniquement du rôle r d'un des joueurs, d'un fluent $f \in F$ et de deux clauses $c \in C$ et $c' \in C$.

Definition 10.16 (Méta-action)**.** *Une action* $a = \texttt{does}(r, o)$ *appartient à la* ***méta-action*** $P(r, f, c, c')$ *si :*

- *f est un effet potentiel de a, et*
- `next`(f) *sous DNFD possède une clause* (`does`$(r, o) \wedge c$)*, et*
- *si c' est vide,* `legal`(r, o) *doit être toujours vrai, ou*
  si c' n'est pas vide, il contient uniquement des littéraux dépendants des actions et apparaît dans au moins une clause de `legal`(r, o) *sous DNFD.*

Une méta-action est donc un groupe d'actions avec un effet identique sur un fluent d'un sous-jeu particulier, des préconditions identiques dans les règles `next` correspondantes et au moins une précondition en commun dans les règles `legal`.

L'utilisation des DNF permet de gagner du temps en ne développant pas toutes les expressions. Dans ce cas, les préconditions utilisées dans les règles `next` et `legal` pour la détection des méta-actions peuvent être des prédicats auxiliaires. L'utilisation de DNF présente un inconvénient : si une des règles contenant des variables est déjà instanciée dans la description GDL originale et si certaines de ces instances seulement sont exprimées en fonction de prédicats auxiliaires, les actions peuvent apparaître en conjonction avec des prémisses différentes mais équivalentes. La factorisation du circuit doit permettre de restaurer ces prédicats auxiliaires dans toutes les règles. Cependant, comme nous utilisons une factorisation imparfaite utilisant le premier facteur commun, ceci n'est pas garanti. Comme conséquence, la détection des méta-actions peut être entravée. Néanmoins, ce cas est suffisamment spécifique pour que les prédicats auxiliaires puissent être utilisés pendant la détection des méta-actions dans la majorité des cas.

A partir du circuit

Une méta-action est l'ensemble des actions a responsables d'un même effet f^* dans les mêmes circonstances. Ces circonstances correspondent aux fluents conditionnant la légalité des actions i.e. les prémisses des règles de conclusion `legal`(a), mais également aux fluents intervenant en conjonction avec les actions dans les prémisses des règles de conclusion `next`(f). Ces règles indiquent si les circonstances sont réunies pour que l'action ait bien un effet. Identifier les actions intervenant en conjonction avec les mêmes fluents dans les différentes clauses d'une règle `next` implique de disposer de la forme normale disjonctive de cette règle. Cependant, par une comparaison des ensembles d'actions intervenant en conjonction avec chaque fluent séparément il est également possible de les identifier.

Les actions non responsables d'un effet f^*, mais utilisées en conjonction avec un même fluent g dans les prémisses d'une règle de conclusion `next`(f), correspondent à un ensemble $A' = A - (M \cup I)$ avec A l'ensemble des actions (def.10.5), M la méta-action responsable de l'effet f^* et I un ensemble d'actions illégales dans les mêmes conditions. Ces actions illégales peuvent appartenir à un autre sous-jeu que f : dans ce cas, chaque méta-action responsable d'un effet sur les fluents de cet autre sous-jeu est incluse dans I. La comparaison des ensembles des actions associées à un même effet, avec les ensembles des actions utilisées en conjonction avec une même prémisse dans une règle `next`, et avec les ensembles des actions ayant la même prémisse dans leur règle `legal` permet donc d'identifier les méta-actions, même si les liens de causalité ne sont pas explicitement décrits dans les règles GDL. Nous proposons donc la définition suivante d'une méta-action ne nécessitant pas de disposer de la forme normale disjonctive des

règles :

Definition 10.17 (méta-action)**.** *Une méta-action $M(r, \mathcal{E}, \mathcal{N}, \mathcal{L})$ est un ensemble d'actions $a = \texttt{does}(r, o)$ tel que les conditions suivantes sont vérifiées :*

- $\mathcal{E} \neq \emptyset$ *et pour chaque* $f^* \in \mathcal{E}$*, a est responsable de l'effet* f^**.*
- *pour chaque fluent* $g \in \mathcal{N}$*, g est utilisé en conjonction avec a dans les prémisses de* $\texttt{next}(f)$*, a est responsable de l'effet* f^* *et a n'est pas une* précondition exclusive *de* $\texttt{next}(f)$*.*
- *pour chaque fluent* $h \in \mathcal{L}$*, h est une précondition non-contradictoire de* $\texttt{legal}(r, o)$*. Si* $\texttt{legal}(r, o)$ *est toujours vrai,* $\mathcal{L} = \emptyset$*.*
- *il n'existe pas* $A' \subsetneq M(r, \mathcal{E}, \mathcal{N}, \mathcal{L})$ *tel que pour chaque* $a' \in A'$*,* g' *est utilisé en conjonction avec* a' *dans les prémisses de next(*f'*),* a' *n'est pas une précondition exclusive de* $\texttt{next}(f')$ *et* a' *n'a aucun effet sur* f'*.*

Pour identifier les méta-actions d'après cette définition, nous utilisons les résultats obtenus après filtrage des probabilités d'effet des actions (cf. §10.2.2) et considérons que pour tout $\mathcal{P}(a, f^*) > 0$, $a = \texttt{does}(r, o)$ est responsable de l'effet f^*.

Pour identifier chaque fluent h prémisse de $\texttt{legal}(r, o)$, nous utilisons le circuit logique. La sortie $\texttt{legal}(r, o)$ est marquée *vraie* puis le signal est rétro-propagé dans le circuit avec 4 états possibles comme pour la détection des préconditions (cf. §10.3.2). Chaque prémisse est ensuite vérifiée avec une logique à 3 états pour s'assurer qu'elle permet réellement d'obtenir $\texttt{legal}(r, o)$ i.e. c'est une prémisse non-contradictoire.

Pour identifier chaque fluent g utilisé en conjonction avec une action a dans les prémisses de $\texttt{next}(f)$, nous marquons l'entrée $\texttt{does}(r, o)$ à *vrai* et propageons le signal à travers le circuit sans tenir compte des portes logiques de manière à marquer chaque porte dépendant de cette entrée. À partir d'une des sorties $\texttt{next}(f)$ activées, nous rétro-propageons le signal en utilisant différents marqueurs, pour étiqueter les portes représentant une conjonction entre l'action et une autre entrée (ce peut être une porte OU en amont d'une négation), et pour étiqueter les fluents en amont de ces conjonctions.

Nous effectuons finalement une comparaison des différents ensembles d'actions obtenus pour en déduire les méta-actions.

Par exemple, pour le jeu *Asteroid Parallel*, toutes les actions ayant le même ensemble vide de préconditions définissant leur légalité, elles constituent un groupe unique L1 :

```
L1 : (does ship (do thrustthrust)) (does ship (do thrustcounter))
     (does ship (do thrustclock))  (does ship (do counterthrust))
     (does ship (do clockthrust))  (does ship (do counterclock))
     (does ship (do clockclock))   (does ship (do clockcounter))
     (does ship (do countercounter))
```

En prenant en compte les effets détectés, quatre groupes peuvent être identifiés, les deux premiers (E1 et E2) correspondent à un changement d'orientation des vaisseaux pour chacun des sous-jeux, les deux autres (E3 et E4) à une modification de la propulsion :

```
E1 : (does ship (do clockthrust))  (does ship (do counterthrust))
     (does ship (do clockcounter)) (does ship (do countercounter))
```

```
      (does ship (do clockclock))   (does ship (do counterclock))

E2 : (does ship (do thrustclock))  (does ship (do thrustcounter))
     (does ship (do counterclock)) (does ship (do countercounter))
     (does ship (do clockclock))   (does ship (do clockcounter))

E3 : (does ship (do thrustthrust))
     (does ship (do thrustcounter))
     (does ship (do thrustclock))

E4 : (does ship (do thrustthrust))
     (does ship (do counterthrust))
     (does ship (do clockthrust))
```

Enfin, en prenant en compte les préconditions dans les règles `next`, il est possible, entre autre, de détecter que la nouvelle orientation du vaisseau est pré-conditionnée par la position précédente et de distinguer les rotations horaires et anti-horaires, ce qui permet d'identifier 6 méta-actions correspondant respectivement aux actions *thrust*, *clock* et *counter* dans chacun des deux sous-jeux :

```
M1 : (does ship (do thrustthrust)) (does ship (do thrustcounter))
     (does ship (do thrustclock))
M2 : (does ship (do clockthrust)) (does ship (do clockclock))
     (does ship (do clockcounter))
M3 : (does ship (do counterthrust)) (does ship (do counterclock))
     (does ship (do countercounter))
M4 : (does ship (do thrustthrust)) (does ship (do clockthrust))
     (does ship (do counterthrust))
M5 : (does ship (do clockclock)) (does ship (do counterclock))
     (does ship (do thrustclock))
M6 : (does ship (do thrustcounter)) (does ship (do clockcounter))
     (does ship (do countercounter))
```

10.4.2 Détection des jeux en série

Un jeu en série est un jeu dans lequel plusieurs sous-jeux s'enchaînent l'un après l'autre i.e. quand un des sous-jeux s'achève, un autre commence. Ce type de jeu nécessite une détection spécifique pour séparer les sous-jeux. Quand le début d'un sous-jeu dépend de l'état du précédent, les actions de ce sous-jeu sont liées aux fluents du sous-jeu précédent qui constituent des préconditions à leur légalité ou conditionnent leur influence dans les règles `next`. Ces liens réunissent les sous-jeux dans le graphe de dépendances, il convient donc de détecter les sous-jeux en série pour briser ces liens et séparer ces sous-jeux.

Nous avons testé deux approches pour l'identification des sous-jeux en série en fonction de la procédure de décomposition utilisée. La première s'inspire des travaux précédents et est fondée sur un partitionnement des actions ; elle utilise les prédicats auxiliaires identifiés pendant la création du circuit. La seconde est une approche plus systématique fondée sur la recherche de *points de croisement* dans le jeu.

Partitionnement des actions

Dans les jeux en série constitués de deux sous-jeux, comme *Tictactoe Serial*, *Asteroid Serial*, *Blocker Serial* ou *Blocks World Serial* un prédicat auxiliaire décrivant l'état terminal du premier sous-jeu détermine la légalité des actions du second sous-jeu. Zhao [2009] utilise une analyse spécifique des règles pour le détecter (cf. §9.4.4) : ce prédicat auxiliaire doit être *faux* pour autoriser les actions du premier sous-jeu et *vrai* pour autoriser les actions du second. Par exemple dans le jeu *Tictactoe Serial*, ce prédicat auxiliaire est *game1over* dont l'utilisation est :

```
(<= (legal ?player (mark1 ?x ?y)) (not game1Over)
    (true (control1 ?player)) (true (cell1 ?x ?y b)))
(<= (legal ?player (mark2 ?x ?y)) game1Over
    (true (control2 ?player)) (true (cell2 ?x ?y b)))
```

Dans sa définition, le prédicat `game1over` dépend de l'expression $\texttt{line1}(x) \vee \texttt{line1}(o) \vee \neg\texttt{open1}$ dont les termes indiquent que l'état est terminal si un des joueurs a aligné 3 marques ou que le plateau de jeu est plein. Notons que game1Over est un prédicat auxiliaire unique et que l'utilisation d'un tel prédicat pour conditionner la légalité des actions des deux sous-jeux est une habitude d'écriture qui peut être facilement changée. On pourrait par exemple utiliser deux expressions `ongoing1` dans les règles `legal` du premier sous-jeu et `game1over` dans celles du second, avec `ongoing1` dépendant de $\neg\texttt{line1}(x) \wedge \neg\texttt{line1}(o) \wedge \texttt{open1}$. Ce qui mettrait en échec l'approche proposée par Zhao.

Pour généraliser l'approche de Zhao [2009], nous considérons qu'un *pivot* entre deux sous-jeux est composé de deux prédicats auxiliaires qui peuvent être la négation l'un de l'autre ou bien deux prédicats différents. Nous utilisons la propagation d'un signal dans le circuit pour tester l'influence de chaque prédicat auxiliaire, détecté pendant la création du circuit, sur la légalité des actions et nous recherchons un couple de prédicats qui partitionnent les actions dépendantes des fluents en deux groupes. Si un tel couple est découvert, alors il s'agit d'un *pivot*.

Cette approche est efficace pour séparer les sous-jeux en série composés de deux sous-jeux. Malheureusement, il n'est pas possible de généraliser cette approche pour identifier un *pivot* dans le cas des jeux avec $N > 2$ sous-jeux en série sans risquer une sur-décomposition des jeux à pièces mobiles. Dans un pivot, chaque prédicat auxiliaire rend les actions d'un premier sous-jeu légales et interdit les actions du second, ou l'inverse. Si un troisième sous-jeu est présent, ses actions ne sont pas affectées par ces prédicats. Dans un jeu avec des pièces mobiles, un prédicat auxiliaire peut être utilisé pour décrire l'état d'une case ; ce prédicat peut autoriser certaines actions de déplacement à partir de cette case, interdire certaines actions de déplacement vers cette case et n'affecte pas les autres actions du jeu. Par conséquent, ce prédicat auxiliaire pourrait être confondu avec un élément d'un *pivot*. Si nous tentons d'identifier les *pivots* dans des jeux en série composés de plus de deux sous-jeux par une généralisation de cette approche, nous risquons donc une sur-décomposition des jeux à pièces mobiles, chaque case étant identifiée comme étant un petit sous-jeu en série amenant au suivant.

Cette approche présente d'autres limitations. La détection des pivots est fondée sur l'*a priori* qu'un prédicat auxiliaire conditionne la légalité des actions. Cependant, il est tout à fait envisageable de créer un jeu en série dont toutes les actions sont toujours légales mais dont l'influence sur l'état du jeu, décrite par les règles `next`, est conditionnée en fonction du sous-jeu en cours. Comme nous l'avons indiqué

à la section 10.2.1, l'auteur d'une description de jeu peut ne pas utiliser de prédicats auxiliaires. Nous effectuons une factorisation du graphe de règles avant la création du circuit pour s'assurer de reconstituer ces prédicats auxiliaires le cas échéant.

Recherche systématique de points de croisement

Dans notre seconde méthode de décomposition DGFS, afin d'obtenir une détection plus robuste des jeux en série, nous avons écarté l'utilisation de ces prédicats auxiliaires. Pour que notre approche soit applicable à un nombre quelconque de sous-jeux nous avons fondé notre détection sur une caractéristique plus générale des jeux séquentiels.

Les jeux en série, et plus généralement les jeux composés de différentes parties jouées séquentiellement, sont caractérisés par la présence d'un état ou d'un ensemble d'états nécessairement visités pendant une partie et correspondant à l'achèvement d'un sous-jeu et le début d'un autre. Aucune séquence d'action ne permet d'atteindre le reste du jeu sans passer par un de ces états. Ces états sont caractérisés par l'apparition d'un fluent ou d'un ensemble de fluents vrais que nous nommons *point de croisement*. Par exemple, dans *Blocker Serial*, un jeu constitué de deux parties de *Blocker* jouées séquentiellement, l'apparition du fluent `game1overlock` [68] signale la fin du premier sous-jeu et conditionne la légalité des actions du second sous-jeu. Dans *Asteroids Serial*, un jeu constitué de deux parties de *Asteroid*, le premier sous-jeu prend fin quand le vaisseau s'arrête, ce qui est représenté par la conjonction de deux faits (`north-speed1(0)`$\wedge$`east-speed1(0)`) indiquant une vitesse nulle sur les 2 axes cardinaux. Ces *points de croisement* peuvent être identifiés à partir d'un *graphe causal* représentant les relations de causalité entre actions et fluents. Chaque fluent composant un *point de croisement* est un *fluent charnière* :

Definition 10.18 (fluent charnière)**.** *Soit G un graphe orienté représentant les relations de causalité entre les actions et les fluents (positifs ou négatifs) d'un jeu. Soit C(G) la fermeture transitive de ce graphe. Un fluent x parmi les sommets du graphe est un fluent charnière si dans C(G) :*

- *il existe au moins un arc $y \rightarrow x$,*
- *il existe au moins un arc $x \rightarrow a$ avec a une action,*
- *x n'est pas dans l'état initial du jeu.*

Un *point de croisement X* est un ensemble d'au moins un *fluent charnière*.

Pour identifier les *points de croisement* à partir des *fluents charnière*, nous construisons un *graphe causal G* dont les sommets sont des actions et des fluents (positifs ou négatifs). Les relations logiques entre les sommets sont représentées par des arcs orientés. Nous ajoutons un arc y $\rightarrow z$ dans G si :

- y est une action et z le résultat d'un effet dont y est responsable ;
- y est un fluent (peut-être indépendant des actions), prémisse de `legal`(r, p) avec $z =$ `does`(r, p) ;
- y est un fluent, z est une action et $y \wedge z$ est une prémisse de `next`(f) ou z est responsable de l'effet f^* ;
- y est une prémisse de `next`(z) ou z est un fluent indépendant des actions.

68. Une position est décrite par l'ensemble des fluents vrais dans cet état du jeu. L'apparition d'un fluent correspond donc à son passage de la valeur *faux* à *vrai*.

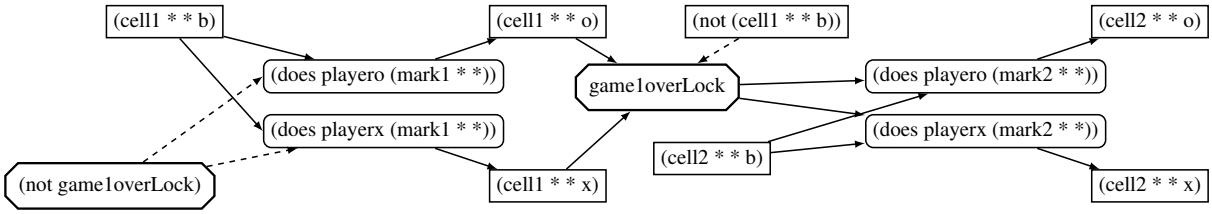


Figure 10.4 – Graphe causal extrêmement simplifié pour le jeu *Tictactoe Serial*. Un ensemble de termes avec des arguments variables est représenté avec des jokers *. Une partie des liens est omise pour clarifier le graphe. *Game1overLock* est identifié comme *point de croisement*.

Un *fluent charnière* unique est un *point de croisement* s'il n'existe pas deux arc $x \rightarrow y$ et $y \rightarrow x$ dans C(G). La figure 10.4 représente une version très simplifiée du graphe causal obtenu pour le jeu *Tictactoe Serial*. Le fluent *game1overLock* marque une limite nette entre deux parties distinctes du jeu et constitue un *point de croisement*.

Soit X_c l'ensemble des *fluents charnière*. Un ensemble $X_i \subset X_c$ est un *point de croisement potentiel*, s'il existe dans C(G) un ensemble de sommets Q tel que pour tout $x \in X_i$ et pour tout $q \in Q$ il existe un arc $x \rightarrow q$. Pour chaque X_i qui est potentiellement un *point de croisement* nous ajoutons un sommet o_i dans G. Nous utilisons alors le circuit pour identifier les relations logiques entre chaque sommet o_i et les autres.

Dans G, nous ajoutons un arc $y \rightarrow o_i$ pour chaque arc $y \rightarrow x$ avec $x \in X_i$ et nous ajoutons un arc $o_i \rightarrow y$ si :

- une action $y = \texttt{does}(r, p)$ est légale quand tous les $x \in X_i$ sont *vrais* et il existe un $x \in X_i$ prémisse de $\texttt{legal}(r, p)$;
- $\texttt{next}(f)$ peut être *vrai* quand tous les $x \in X_i$ sont *vrais* et il existe $x \wedge y$ prémisse de $\texttt{next}(f)$ où f^* est un effet de l'action y ;
- y est un fluent indépendant des actions et y^* est observé quand tous les fluents $x \in X_i$ sont *vrais*.

X_i est un *point de croisement* s'il n'existe pas deux arcs $o_i \rightarrow y$ et $y \rightarrow o_i$ dans $C(G)$.

Un *point de croisement* est écarté s'il a pour unique précondition un autre *point de croisement* ou un fluent de l'état initial ou s'il inclut un autre *point de croisement*.

10.4.3 Bilan de la détection des actions composées et jeux en série

Nous avons proposé différentes approches pour le traitement des *actions composées* et des *jeux en série*.

Pour identifier les actions composées et éviter qu'elles ne relient différents sous-jeux, nous avons proposé le concept de *méta-action*. Une *méta-action* représente une composante d'une action ayant un effet sur un seul des sous-jeux et regroupe toutes les actions composées contenant la même composante. Dans un jeu comprenant des actions composées, chaque action est placée dans M méta-actions correspondant à ces M effets. Si un jeu ne contient pas d'actions composées, chaque action constitue un *méta-action-singleton*. Ainsi, l'utilisation des méta-actions est compatible avec tous les jeux.

Pour les jeux en série, nous avons présenté deux approches. La première, fondée sur le partitionnement des actions en deux groupes pour identifier un *pivot*, est limitée à la détection de jeux en série

composés de seulement deux sous-jeux. La seconde, utilisant le concept de *point de croisement* est plus générale et permet d'identifier n'importe quel nombre de sous-jeux en série à partir d'un *graphe causal*.

Dans notre *graphe de dépendances*, la prise en compte des *méta-actions* et des *pivots* ou *points de croisement* se traduit par différentes modifications. Dans la section suivante, nous indiquons comment ce graphe est construit et comment les sous-jeux sont identifiés.

10.5 Identification des sous-jeux

Nous expliquons maintenant comment le *graphe de dépendances* est construit à partir de toutes les informations collectées précédemment : effets des actions, préconditions des actions et des fluents indépendants des actions, identification des actions composées et des sous-jeux en série. Nous présentons ensuite un post-traitement permettant de corriger certains cas de sous- ou sur-décomposition. Enfin nous montrons comment les sous-jeux obtenus sont étiquetés afin d'en identifier le rôle et comment certaines informations complémentaires peuvent être extraites.

10.5.1 Finalisation du graphe de dépendances

Nous utilisons toutes les informations collectées précédemment (effets des actions, préconditions des actions et des fluents indépendants des actions, identification des actions composées et des sous-jeux en série) pour construire le *graphe de dépendances* nécessaire à l'identification des sous-jeux. La prise en compte des *méta-actions* nécessaires à la séparation des actions composées et des *pivots* ou *points de croisement* nécessaires pour la séparations des sous-jeux en série, se traduit par différentes modifications du graphe initial.

Le graphe de dépendances modifié est un graphe non orienté $\{V, E\}$ dans lequel nous ajoutons un sommet dans V pour chaque fluent f et pour chaque méta-action M (et non chaque action) du jeu. Les liens entre les actions et les fluents sont remplacés par des liens entre les méta-actions et les fluents.

Dans le cas d'une détection des méta-actions à partir des règles, nous ajoutons une arête entre chaque méta-action et son effet f, et ses préconditions $f' \in c \cup c'$.

Dans le cas d'une détection des méta-actions à l'aide du circuit, nous ajoutons une arête entre une méta-action $M(r, \mathcal{E}, \mathcal{N}, \mathcal{L})$ et un fluent f si :

(1) $f \in \mathcal{L} \backslash C$ avec C l'ensemble des fluents *control* détecté (pour éviter de lier f aux méta-actions de différents sous-jeux).

(2) $f \in \mathcal{N} \backslash C'$ avec C' l'ensemble des fluents *control* détectés ou, à défaut, l'ensemble des fluents indépendants des actions.

(3) $f* \in \mathcal{E}$

Les arcs (1) et (2) sont associés à une pondération de 1. La pondération W des arcs (3) est la moyenne des probabilités d'effet associées à chaque action : $W = \sum_{0<i<N} \frac{\mathcal{P}(a_i, f^*)}{N}$ avec $a_i \in M(r, \mathcal{E}, \mathcal{N}, \mathcal{L})$. Nous utilisons cette pondération pour effectuer le post-traitement nécessaire au traitement de certains cas de sur- ou sous-décomposition.

Dans le cas où des sous-jeux en série sont identifiés, l'étape suivante de la construction du graphe consiste à éliminer certaines arêtes reliant ceux-ci.

Dans le cas de l'identification d'un *pivot*, pour chacun des prédicats composant le *pivot*, nous ajoutons un sommet dans le graphe de dépendances. Ces prédicats sont utilisés directement comme préconditions des méta-actions à la place des fluents composant les expressions logiques correspondantes. Dans le graphe de dépendances, les liens de préconditions relient donc les sommets représentant les prédicats auxiliaires aux méta-actions du second sous-jeu. Les fluents du premier sous-jeu se retrouvent encapsulés dans ces prédicats auxiliaires ce qui permet de s'assurer qu'ils ne relieront pas les différents sous-jeux avec des liens directs vers les méta-actions du second sous-jeu.

Dans le cas de l'identification de *points de croisement*, pour séparer les sous-jeux en série dans le graphe de dépendances, nous supprimons les arcs entre chaque fluent d'un *point de croisement* et les fluents ou méta-actions que ce *point de croisement* pré-conditionne.

Pour réunir les fluents des sous-jeux indépendants des actions, nous ajoutons une arête entre les fluents indépendants des actions et leurs préconditions. Certains fluents indépendants des actions peuvent être pré-conditionnés par des fluents appartenant à un sous-jeu dépendant des actions. C'est le cas par exemple des instances des fluents `(x1 ?n)`, `(y1 ?n)`, `(x2 ?n)` et `(y2 ?n)` représentant les coordonnées des vaisseaux dans le jeu *Asteroid Parallel*. Dans ce cas, ces fluents sont reliés au sous-jeu associé et ne constituent pas un sous-jeu indépendant des actions.

Dans le cas de notre approche fondée sur une analyse probabiliste de l'effet des actions, certaines actions peuvent ne pas avoir été testées durant les *playouts* et certains effets peuvent ne pas avoir été observés. Les actions et fluents correspondants sont alors considérés comme appartenant à tous les sous-jeux jusqu'à ce que plus de *playouts* apportent d'avantage d'informations.

10.5.2 Séparation/re-composition des sous-jeux

Dans certains jeux, il n'existe pas de relation entre les différentes structures internes en dehors de l'évaluation du score. C'est le cas pour les cases de *Tictactoe*, les colonnes de *Connect Four*, ou les régions colorées de *Rainbow*. Une étape de post-traitement est donc nécessaire pour identifier les différents éléments connexes du graphe de dépendances qui doivent être réunis en fonction du but du jeu.

Dans notre première approche de décomposition (DGFR), nous avons utilisé les prédicats auxiliaires détectés durant la création du circuit pour identifier des sous-buts. Ces prédicats auxiliaires intervenant dans les prémisses des règles de conclusion `goal`, décrivent différentes conditions nécessaires à l'obtention de points. Par exemple dans le jeu *Tictactoe Parallel*, les prédicats auxiliaires `line1` et `line2` – qui dépendent respectivement des prédicats `diagonal1`, `column1`, `row1` et `diagonal2`, `column2`, `row2` – décrivent les différents alignements de trois marques possibles dans chacun des deux sous-jeux. Ces prédicats décrivent un lien logique existant entre les différentes cases des sous-jeux.

Dans notre seconde approche de décomposition (DGFS), la prise en compte des effets implicites des actions amène un problème inverse pour certains jeux. Dans un jeu comme *Lights On Parallel*, toutes les lampes sont réunies en un seul sous-jeu car chaque action d'allumer une lampe a comme effet implicite collatéral de laisser s'éteindre les autres. Cependant, nous souhaitons séparer le jeu en groupes de 4 lampes car allumer toutes les lampes d'un des groupes est suffisant pour obtenir la victoire. Les effets

collatéraux implicites ne se produisent pas systématiquement lors des transitions. En effet, une autre lampe doit être allumée pour qu'elle puisse ensuite s'éteindre lentement. La probabilité associée à ces effets est donc nettement plus faible que celle associée aux effets directs des actions. Les pondérations que nous avons associées aux liens d'effet dans le graphe permettent d'ignorer ces effets collatéraux en brisant les liens de faible pondération correspondants. L'examen des conditions de victoire permet alors de restituer, le cas échéant, le lien entre les fluents d'un même sous-jeu.

Dans un premier temps nous détaillons la première approche utilisant les prédicats auxiliaires pour effectuer un post-traitement des jeux sur-décomposés. Dans un second temps, nous présentons une autre approche utilisant le circuit pour identifier les buts du jeux et effectuer un post-traitement des jeux sur- ou sous-décomposés.

10.5.3 Utilisation des prédicats auxiliaires

Dans le jeu *Double Tictactoe* utilisé comme exemple par Zhao *et al.* [2009], les prédicats auxiliaires `line1` et `line2` sont présents dans les prémisses de certaines règles `legal`. Un lien de précondition entre les différents fluents des sous-jeux et les actions est donc ajouté au graphe de dépendances, ce qui évite le problème de sur-décomposition. Les travaux précédents sur la décomposition ne se préoccupent donc pas de ce problème survenant pourtant dans de nombreux jeux ou le joueur doit marquer des cases ou zones pour atteindre un but i.e. aligner ou créer un motif donné avec des marques ou des couleurs.

Pour identifier le lien logique existant entre les fluents d'un sous-jeu, il est nécessaire de pouvoir distinguer les prédicats auxiliaires décrivant un seul sous-but dans l'un des sous-jeux, de ceux décrivant un ensemble de sous-buts correspondant à différents sous-jeux ; les seconds risquant de créer un lien entre tous les fluents et d'annuler la décomposition. Pour traiter ce problème, nous utilisons l'heuristique suivante pour identifier les potentiels prédicats sous-buts correspondant à un seul sous-jeu :

Definition 10.19. *Soit g le score maximum qui peut être atteint par le rôle r. Un prédicat auxiliaire b est un* ***prédicat sous-but potentiel*** *si :*

- `terminal` *dépend de la valeur logique de b, et*
- `goal`(r, g) *sous DNF possède une clause où b apparaît.*

ou

- *Tous les joueurs jouent dans des sous-jeux différents, et*
- `goal`(r, g) *sous DNF possède une clause où b apparaît, et pour tous les rôles $r' \neq r$,* `goal`(r', g') *sous DNF n'a pas de clause ou b apparaît.*

Cette heuristique se fonde sur l'idée qu'un prédicat auxiliaire intervenant à la fois dans `goal` et `terminal` a peu de chance de concerner plusieurs sous-jeux. Il ne serait en effet pas très intéressant de devoir gagner dans plusieurs sous-jeux en même temps pour pouvoir mettre fin à la partie. Il n'y aurait pas vraiment de séparation entre ces sous-jeux s'il n'y avait pas la possibilité de perdre dans l'un et de gagner dans l'autre. La première partie de cette définition est donc valable pour les jeux ou la victoire dans l'un des sous-jeu met fin à la partie, comme cela est souvent le cas dans les jeux composés. Dans le cas contraire, un prédicat identifié à tort comme sous-but connecterait les différents sous-jeux.

Dans des jeux comme *Dual Rainbow* ou *Dual Hamilton*, correspondant à différents sous-jeux solitaires joués en parallèle, les prédicats sous-buts apparaissent uniquement dans les règles `goal`. Cependant, comme chaque sous-jeu est un solitaire, un prédicat auxiliaire présent dans les règles `goal` d'un seul joueur ne concerne forcément qu'un seul sous-jeu. Ce cas est couvert par la seconde partie de la définition.

Les prédicats sous-buts identifiés sont utilisés pour ajouter des liens dans le graphe de dépendances. Les sommets correspondant aux fluents qui apparaissent dans une même clause de ces prédicats sous DNFD, sont reliés entre eux.

Cette approche fonctionne sur un bon nombre de jeux composés mais est néanmoins fragile. L'heuristique est sensible à certaines réécritures des règles car elle dépend de l'identification de prédicats auxiliaires qui peuvent être absents des règles. Ces prédicats peuvent ne pas être correctement reconstitués par notre procédure de factorisation du graphe de règles préalable à la création du circuit.

10.5.4 Détection de sous-buts et conditions de victoire

Pour remplacer l'utilisation de ces prédicats auxiliaires, dans notre seconde méthode de décomposition (DGFS), nous avons utilisé le circuit pour identifier les *sous-buts*. Chaque sommet (ou porte logique) interne du circuit représentant une expression logique de plus en plus complexe à mesure que l'on se rapproche des sorties, nous examinons dans l'ordre chacune de ces expressions intervenant en amont des sorties `goal` indiquant si un score donné est atteint pour l'un des joueurs, pour identifier leur influence sur le score. Nous recherchons les expressions minimales représentant un *sous-but* i.e. un état partiel apportant des points à l'un des joueurs ou bien apportant le score maximal. Dans ce dernier cas, l'expression représente une *condition de victoire*.

Le sous-circuit permettant de calculer les sorties `goal` (évaluation du score) fait l'objet d'un tri topologique : chaque porte logique dont la sortie est utilisée comme entrée d'une autre porte est examinée en premier. Chaque porte p du sous-circuit est activée indépendamment des autres avec une valeur $o \in \{vrai, faux\}$ et, en utilisant une logique à 3 états pour diffuser le signal, l'état des sorties `goal` est examiné. Si une sortie représentant le score maximum pour un des joueurs est activée, alors la valeur o pour la porte p représente une *condition de victoire*. Sinon, si un score non nul peut néanmoins être obtenu (le score 0 est *faux*, un score supérieur à 0 est *vrai* ou bien la valeur $\neg o$ rend le score maximum impossible à atteindre), alors la valeur o pour la porte p est un sous-but. Chaque porte utilisant la sortie d'une porte p détectée comme *sous-but* ou *condition de victoire* est exclue de la recherche : ceci permet de collecter uniquement les conditions minimales pour obtenir un score non nul.

Si plusieurs sous-jeux dépendants des actions sont détectés, i.e. le *graphe de dépendances* présente plusieurs zones connexes comprenant des sommets méta-action, nous vérifions si le jeu n'est pas surdécomposé. Nous vérifions que chaque *sous-but* ou *condition de victoire* ne dépend pas de fluents placés dans différents sous-jeux. Dans le cas contraire, les sous-jeux sont regroupés en un. Cela permet de s'assurer que chaque sous-jeu, si ses fluents sont impliqués dans le calcul du score, apporte au moins une partie des points : il existe une fonction d'évaluation.

Si un seul sous-jeu dépendant des actions est détecté, il regroupe peut-être des sous-jeux permettant chacun d'obtenir le score maximum, mais liés par des effets collatéraux des actions. Pour vérifier si de

tels sous-jeux existent, les liens dont le poids est inférieur au seuil Ω sont supprimés du graphe puis les *conditions de victoire* sont examinées pour vérifier si chaque sous-jeu identifié peut apporter la victoire à lui seul.

10.5.5 Étiquetage des types de sous-jeux

Les différentes composantes connexes du graphe de dépendances obtenues après le post-traitement précédent correspondent aux sous-jeux. La séparation des sous-jeux correspond à une partition des fluents. Les liens logiques existants entre actions et fluents nous ayant permis d'identifier ces sous-jeux, l'ensemble des actions associées à un sous-jeu peut également être obtenu depuis le graphe.

A partir de ces ensembles de fluents et d'actions représentant les sous-jeux, nous pouvons extraire des informations complémentaires : la nature des sous-jeux (dépendants ou indépendants des actions), l'ordre dans lequel les sous-jeux se succèdent dans le cas d'un jeu en série, l'influence d'un sous-jeu sur le score ou la terminaison de la partie, et l'ensemble des fluents, actions ou *sous-jeux inutiles*.

Sous-jeux dépendants ou indépendants des actions

La présence de sommets représentant des méta-actions dans une composante connexe du graphe de dépendances permet d'identifier directement un sous-jeu comme dépendant des actions. Les jeux décrits en GDL ne présentent que deux types de sous-jeux indépendants des actions : les sous-jeux représentant des compteurs (*stepper*) et les sous-jeux regroupant les fluents *control* dont le rôle est de définir l'alternance des joueurs dans les jeux à coups alternés.

Il est possible de distinguer ces deux types de sous-jeux indépendants des actions grâce à certaines de leurs propriétés. Les fluents d'un sous-jeu définissant le *control* interviennent pour déterminer la légalité des actions d'un joueur donné, ce sous-jeu contient donc généralement autant (ou au moins autant) de fluents qu'il y a de joueurs, et la succession des fluents vrais à chaque tour du jeu forme un cycle $\texttt{control}(p_1), \texttt{control}(p_2), ..., \texttt{control}(p_N), \texttt{control}(p_1), ...$. Ce type de sous-jeu est généralement détecté comme inutile (cf. §10.5.5), sauf dans le cas des jeux impartiaux où le fait d'avoir le *control* en fin de jeu détermine la victoire ou la défaite d'un joueur. L'indication de l'alternance des joueurs étant indispensable pour jouer, un sous-jeu de type *control* détecté inutile n'est pas ignoré pendant le jeu mais au contraire associé à chaque sous-jeu dépendant des actions. Un sous-jeu de type *stepper* est utilisé pour mettre fin au jeu en un nombre fini de tours, au moins un des fluents qui le composent est donc une prémisse de `terminal` ou bien pré-conditionne le démarrage d'un autre sous-jeu dans le cas d'un jeux en série.

Ordre des jeux en série

Dans notre première méthode de décomposition (DGFR), les *pivots* séparant les jeux en série sont identifiés par leur propriété de partitionner les actions légales en deux groupes. Cependant, cette information ne permet pas d'inférer directement l'ordre dans lequel les sous-jeux se succèdent.

Pour identifier le second sous-jeu en série et, de manière plus générale, pour tout sous-jeu qui ne démarre pas en même temps que le jeu global, nous identifions tous les sous-jeux qui n'ont pas d'action

légale ou pas d'état suivant à la position initiale du jeu.

Si un tel sous-jeu est trouvé, il est possible d'en identifier la position initiale. Nous recherchons toutes les conjonctions de faits dans les règles `legal` et `next` sous DNFD ou DNF permettant de faire apparaître un coup légal ou un état suivant. Ces conjonctions de fluents sont considérées comme les états initiaux possibles de ce sous-jeu et une relation de dépendance est identifiée entre celui-ci et les sous-jeux susceptibles de faire apparaître ces fluents.

Dans certains jeux en série, l'achèvement d'un sous-jeu A, démarrant avec le jeu global, conditionne le démarrage de deux autres sous-jeux B et C. Cependant, différentes conjonctions de fluents du sous-jeu B peuvent-être détectées comme nécessaires pour le démarrage du sous-jeu C. Une telle situation est présente dans le jeu *Tictactoe Serial* : un second sous-jeu indépendant des actions regroupant des fluents *control* est détecté comme nécessaire pour rendre un coup légal dans la seconde grille de *Tictactoe*. Deux états initiaux sont détectés pour ce second *Tictactoe* en fonction du joueur ayant le *control*. Dans un tel cas, parmi les différentes conjonctions de fluents du sous-jeu B conditionnant le démarrage du sous-jeu C, seule la conjonction de fluents présente dans l'état initial global peut-être considérée comme faisant partie de l'état initial du sous-jeu C et ce sous-jeu B ne doit pas être considéré nécessaire au démarrage du sous-jeu C.

Dans notre seconde méthode de décomposition (DGFS), le graphe causal établi pour la détection des *points de croisement*, permet de déduire directement les dépendances entre les différents sous-jeux en série et l'ordre dans lequel ses sous-jeux s'enchaînent.

Sous-jeux terminaux ou influençant le score

Chaque sous-jeu identifié, qu'il soit dépendant ou indépendant des actions est étiqueté en fonction de son rôle dans l'évaluation du score ou de la fin de partie. Un sous-jeu est *terminal* si certains de ses fluents apparaissent parmi les prémisses de la règle de conclusion `terminal`. De même les fluents intervenant parmi les prémisses des règles de conclusion `goal` permettent d'identifier les *sous-jeu influençant le score*. Ces fluents prémisses de `terminal` ou `goal` peuvent être identifiés rapidement en rétro-propageant un signal dans le circuit, sans tenir compte des portes logiques, à partir des sorties correspondantes.

Fluents, actions et sous-jeux utiles/inutiles

Dans notre première approche de décomposition (DGFR), les fluents et actions représentés par des sommets isolés dans le graphe de dépendances peuvent être considérés comme inutiles i.e. des fluents non utilisés et des actions sans effet. Cependant, comme nous l'avons expliqué dans le cas de *Chomp* (cf. §10.2.1) un très grand nombre de sommets isolés peut permettre d'identifier un problème de décomposition dû à la présence d'actions aux effets essentiellement implicites et non détectés par notre heuristique.

Dans notre seconde approche de décomposition (DGFS), le changement de certains fluents et l'application de certaines actions peut ne pas avoir été testé durant les simulations. Aucune information étant disponible sur les relations existant entre ces fluents et actions et les autres, ceux-ci correspondent à des

sommets isolés dans le graphe de dépendances. Ces fluents dont le changement n'est jamais observé sont temporairement considérés comme *inutiles* dans le jeu, mais ne peuvent être écartés définitivement. Ils sont donc associés à chaque sous-jeu en attendant que plus de simulations apportent un complément d'information. Les actions non testées doivent être distinguées des autres actions représentées par des sommets isolés qui elles ont été détectées sans effet (actions *noop*) durant la décomposition. Ces actions non testées, ne sont associées à aucun sous-jeu en particulier, mais ne sont pas considérées comme des actions *noop* en attendant que plus de simulations apportent un complément d'information. Si des fluents ou/et des actions n'ont pas été testés durant les simulations, la décomposition obtenue est considérée *imparfaite*.

Un sous-jeu est considéré utile s'il est étiqueté *terminal*, *influençant le score* ou s'il est nécessaire pour démarrer un autre *sous-jeu utile* dans le cas des jeux en série :

Definition 10.20. *Soit V_S l'ensemble des sommets d'une partie connexe du graphe de dépendances représentant un sous-jeu S. S est un* ***sous-jeu utile*** *si :*

- *S est joué avant un autre sous-jeu dans un jeu en série et est nécessaire pour le démarrer, ou*
- *il existe $f \in F \cap V_S$ tel que* `terminal` *dépend de la valeur logique de* `true`(f)*, ou*
- *il existe $f \in F \cap V_S$ tel que* `goal`(r, g) *dépend de la valeur logique de* `true`(f)*.*

Dans les jeux *multiples*, tous les sous-jeux qui ne sont pas détectés comme *utiles* peuvent être ignorés pour la résolution du jeu global. Les actions de ces sous-jeux peuvent néanmoins être considérées comme des actions *noop* sans effet sur les *sous-jeux utiles*. Ces actions *noop* peuvent être stratégiquement utiles dans certains cas pour éviter un *zugzwang*. Elles peuvent être considérées équivalentes et une seule d'entre elles nécessite d'être explorée (si légale) pour chaque position du jeu.

10.6 Expérimentations

Toutes nos expérimentations pour les deux méthodes de décomposition (DGFR et DGFS) ont été réalisées sur un ordinateur doté d'un processeur Intel Core i7 2,7GHz avec 8Go de DDR3 à 1600MHz.

Dans ce chapitre, nous présentons le résultat de nos expérimentations pour les deux méthodes de décomposition présentées.

10.6.1 Méthode fondée sur les règles

Voici un résumé des étapes du processus de décomposition pour la première méthode que nous avons testée :

(1) instanciation des règles (grounding)
(2) construction d'un graphe de règles
(3) factorisation des conjonctions, des disjonctions et application des lois de De Morgan pour réduire la taille du graphe et reconstituer les prédicats auxiliaires
(4) construction du circuit et calcul des DNF/DNFD
(5) détection de l'effet potentiel des actions par l'analyse des DNF/DNFD

(6) détection des préconditions des actions et des fluents indépendants des actions
(7) détection des méta-actions
(8) détection d'un éventuel pivot (jeu en série)
(9) construction d'un graphe de dépendances à partir des informations collectées en (5), (6), (7) et (8)
(10) re-composition des sous-jeux sur-décomposés à partir des prédicats auxiliaires décrivant des sous-buts
(11) étiquetage des sous-jeux dépendants ou indépendants des actions, filtrage des *sous-jeux inutiles*.

Nous avons évalué notre premier programme de décomposition sur un panel de 40 descriptions de jeux, composés ou non, extraits des serveurs de Dresde, Stanford et Tiltyard. Nous avons collecté toutes les descriptions de jeux composés [69] à l'exception des descriptions redondantes. Nous avons ajouté à notre panel les descriptions de jeux fréquemment utilisés comme sous-jeux et un panel représentatif de jeux non-composés avec différentes caractéristiques (pièces mobiles, *stepper*, rôles asymétriques, jeux impartiaux) et différents niveaux de complexité.

Pour chaque jeu, nous avons mesuré le temps moyen nécessaire pour chaque étape de la décomposition sur un échantillon de 100 tests. Pour limiter la durée du banc de test, une tentative de décomposition est avortée après 60 minutes. Les étapes les plus longues pour la décomposition sont l'instanciation des règles (1), la factorisation du graphe de règles (2)/(3) et le calcul des DNF/DNFD (4). Les figures 10.5 et 10.6 présentent le temps total de décomposition et le temps pris par chacune de ces étapes [70]. La proximité entre les courbes indiquant le temps total de décomposition et de temps nécessaire pour le calcul des DNF/DNFD montre que les temps cumulés des étapes (5) à (11) restantes n'est pas significatif en comparaison.

Il faut noter que la version très optimisée de *LeJoueur* de Jean-Noël Vittaut qui a gagné le *Tiltyard Open* de 2015 est en moyenne 8,5 fois plus rapide pour réaliser l'instanciation et factoriser le graphe de règles pour les trois jeux les plus complexes (*Breakthrough*, *Hex* et *Blocker Parallel*). Sur la figure 10.5, les temps obtenus par *LeJoueur* sont représentés par deux petites courbes additionnelles. Ceci indique la marge potentielle d'amélioration pour ces étapes de la décomposition.

La figure 10.5 présente le temps de chaque étape de la décomposition en utilisant les DNFD [71]. La figure 10.6 présente le temps de chaque étape de la décomposition en utilisant les DNF et permet de constater le temps économisé en ne développant pas les prédicats auxiliaires. Ces temps sont reportés dans les colonnes 5 et 6 de la figure 10.7. Le gain obtenu avec les DNF est assez conséquent pour permettre une décomposition correcte de 32 jeux sur les 40 en moins de 5 secondes.

Pour *Hex* et *Blocker Parallel*, le temps nécessaire pour l'instanciation des règles, la factorisation et le calcul des DNF reste trop important pour obtenir le résultat de la décomposition en moins d'une heure. La factorisation ne permet pas de réduire suffisamment la complexité de *Hex* et dans *Blocker Parallel*, la présence d'actions composées combinées à des mouvements simultanés pour les deux joueurs apporte un nombre très important de combinaisons. Les prédicats auxiliaires non développés avec l'usage des DNF

69. Nous avons réalisé ces expérimentations en 2015. Les descriptions de jeux composés collectées correspondent donc aux descriptions disponibles à cette date. De nouvelles descriptions de jeux sont proposées régulièrement, ce qui explique que nos expérimentations avec notre seconde méthode de décomposition (DGFS) ont été menées également sur d'autres jeux composés.
70. L'écart type n'est pas représenté car trop faible pour être significatif.
71. Les jeux sont ordonnés en fonction du temps total de décomposition et du temps de factorisation pour faciliter la lecture du graphe.

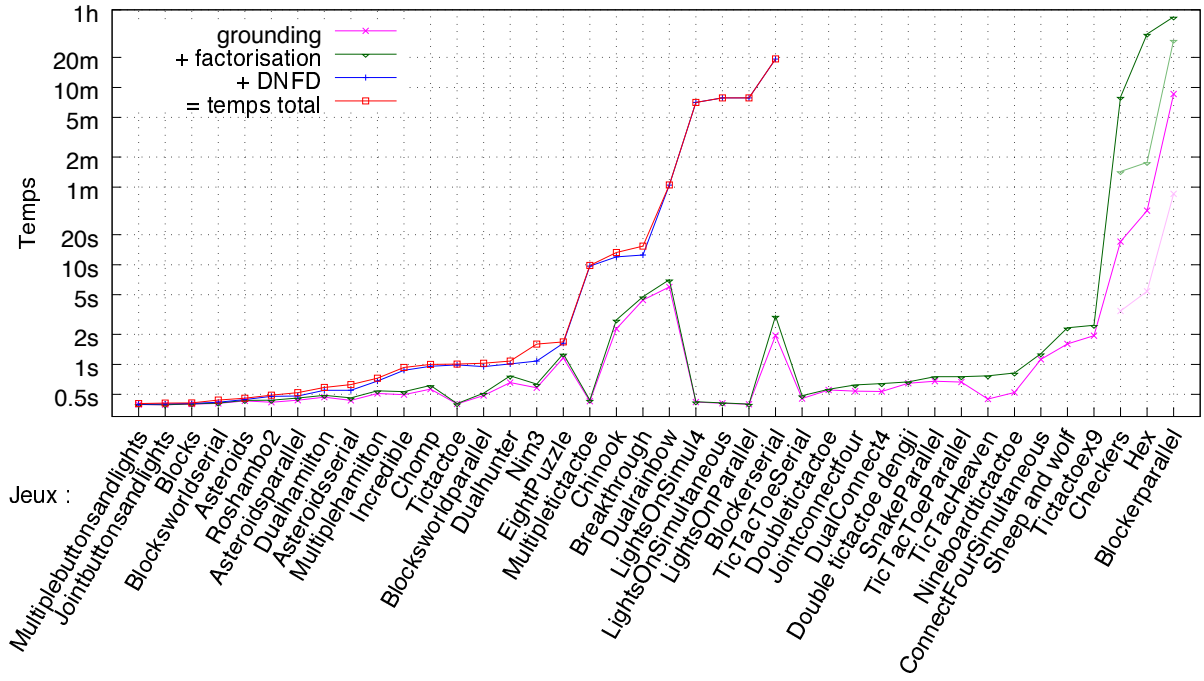


FIGURE 10.5 – Décomposition utilisant les formes normales disjonctives complètement développées (DNFD)

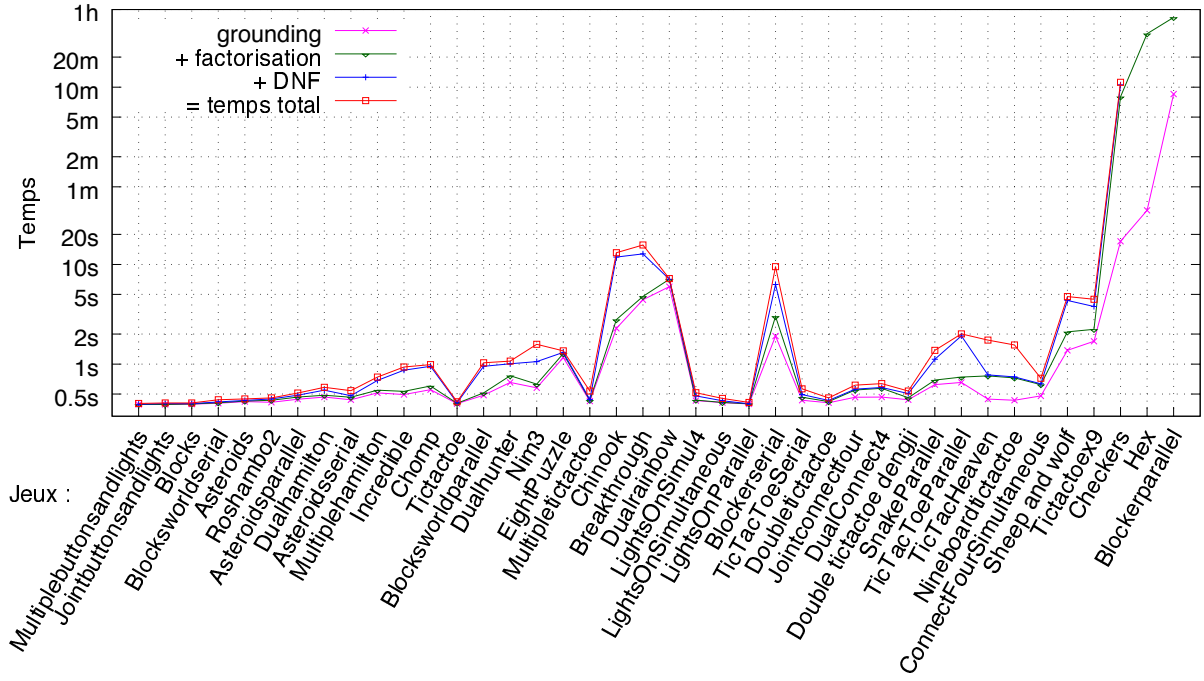


FIGURE 10.6 – Décomposition utilisant les prédicats auxiliaires avec des formes normales disjonctives partiellement développées (DNF)

(fig.10.6) sont définis uniquement en terme de fluents. Ne pas les développer n'apporte donc aucune amélioration : le nombre important d'expressions combinant des actions composées jointes des deux joueurs sont toujours développées.

Ce problème pourrait être résolu par un pré-traitement des règles pour éliminer les variables existentielles, ce qui permettrait un gain conséquent de temps (cf : §9.6 et §10.1.3).

La figure 10.7 présente le résultat de la décomposition pour les 40 jeux avec le nombre total de sous-jeux détectés. Le nombre de sous-jeux dépendants ou indépendants des actions présent parmi eux est indiqué dans les colonnes suivantes ainsi que, entre parenthèses, le nombre de sous-jeux considérés inutiles parmi ceux-ci.

Les premiers jeux du tableau sont des jeux *simples* constitués d'un seul sous-jeu dépendant des actions, parfois accompagné d'un *stepper* détecté comme *sous-jeu utile* indépendant des actions. Le sous-jeu inutiles indépendant des actions, détecté pour des jeux comme *Breakthrough* ou *Sheep and Wolf*, correspond à l'ensemble des fluents décrivant le *control* qui ne constitue pas un sous-jeu jouable en lui-même.

Les *sous-jeux inutiles* dans les jeux *multiples* sont correctement identifiés. Nous remarquons que pour *Multiple Tictactoe* le nombre de *sous-jeux inutiles* est particulièrement important. Cela s'explique par le fait que ces *sous-jeux inutiles* ont été sur-décomposés car aucun prédicat auxiliaire sous-but ne crée de lien entre leurs cases.

Pour le jeu *Nim*, notre programme détecte un sous-jeu indépendant des actions non impliqué dans la terminaison du jeu (ce n'est pas un *stepper*). Cependant, ce sous-jeu est le seul impliqué dans le calcul du score : ceci est un indice fort indiquant que ce jeu est impartial. Cette information peut permettre l'application de techniques connues [Berlekamp *et al.*, 1982] pour la résolution du jeu.

Excepté pour le jeu *Chomp*, tous les sous-jeux détectés sont corrects et correspondent à ce qui aurait été obtenu par une décomposition manuelle experte.

Chomp est, comme nous l'avons expliqué (cf. §10.2.1), un exemple de jeu pour lequel notre heuristique utilisée pour détecter l'effet des actions ne fonctionne pas correctement. La présence de nombreux sommets isolés dans le graphe de dépendances permet de détecter ce problème et d'éviter qu'il n'affecte la résolution du jeu.

10.6.2 Méthode statistique

Comme nous l'avons vu dans la section précédente, le calcul de la DNFD, utilisée pour notre première méthode de décomposition, s'avère très long pour certains jeux. L'heuristique utilisée pour la détection des effets des actions amène des cas de sur-décomposition pour des jeux non décomposables. La conservation et l'utilisation des prédicats auxiliaires dans la DNF amène également le risque d'obtenir une décomposition erronée.

Notre seconde méthode de décomposition a pour but d'éliminer toutes ces faiblesses. Voici un résumé des différentes étapes de cette décomposition :

(a) instanciation des règles (grounding)
(b) construction d'un graphe de règles

jeu	# SJ total	# SJ avec actions	(# inutile)	# SJ action-indep.	(# inutile)	temps (DNFD)	temps (DNF)	commentaires
Hex (T)	*non décomposé après 1 heure*							
Blockerparallel (D)	*non décomposé après 1 heure*							
Asteroids (D)	2	1		1		<1sec	<1sec	
Blocks (D)	2	1		1		<1sec	<1sec	
EightPuzzle (T)	2	1		1		<2sec	<2sec	
Roshambo2 (D)	2	1		1		<1sec	<1sec	
Checkers (D)	3	1		1	(1)	>**1hr**	<12min	
Breakthrough (T)	2	1			(1)	<16min	<16min	
Sheep and wolf (D)	2	1			(1)	>**1hr**	<5sec	
Tictactoe (S)	2	1			(1)	≈1sec	<1sec	
Nineboardtictactoe (S)	2	1			(1)	>**1hr**	<2sec	SG = 9 Tictactoe ensemble
Tictactoex9 (D)	2	1			(1)	>**1hr**	<5sec	SG = 9 Tictactoe ensemble
Chomp (D)	2	1			(1)	<1sec	<1sec	⚠**Mauvaise décomposition**
Multiplehamilton (S)	3	1	(1)	1		<1sec	<1sec	SJ = Hamilton (seul utile : right)
Multiplebuttonsandlights (S)	10	1	(9)	1		<1sec	<1sec	SJ = groupes de boutons (seul utile : n^{o}5)
Multipletictactoe (S)	75	1	(72)	1	(1)	<10sec	<1sec	SJ = Tictactoe n^{o}5 (+ SJ inutiles = cases isolées)
Blockerserial (D)	2	2				<20min	<10min	SG = Blocker ×2
Dualrainbow (S)	2	2				≈1min	<8sec	SG = Rainbow ×2
Asteroidsparallel (D)	3	2		1		<1sec	<1sec	SJ = Asteroid ×2
Blocksworldparallel (D)	3	2		1		≈1sec	≈1sec	SJ = Blocks ×2
Dualhamilton (S)	3	2		1		<1sec	<1sec	SJ = Hamilton ×2
Dualhunter (S)	3	2		1		<2sec	<2sec	SJ = Hunter ×2
Incredible (D)	3	2		1		<1sec	<1sec	SJ = Maze et Block
Asteroidsserial (D)	4	2		2		<1sec	<1sec	SJ = Asteroid ×2
Blocksworldserial (D)	4	2		2		<1sec	<1sec	SJ = Blocks ×2
Jointbuttonsandlights (S)	4	3		1		<1sec	<1sec	SJ = 3 groupes de boutons
LightsOnParallel (T)	5	4		1		<8min	<1sec	SJ = 4 groupes de lampes
LightsOnSimul4 (T)	5	4		1		<8min	<1sec	SJ = 4 groupes de lampes
LightsOnSimultaneous (T)	5	4		1		<8min	<1sec	SJ = 4 groupes de lampes
Nim3 (D)	5	4		1		<2sec	<2sec	SJ = 4 piles + *control* utile pour `goal`
Chinook (S)	6	2		2	(2)	<14sec	<14sec	SJ = Checkers ×2
Double tictactoe dengji (D)	3	2			(1)	>**1hr**	<1sec	SJ = Tictactoe ×2
SnakeParallel (T)	3	2			(1)	>**1hr**	<2sec	SJ = Snake ×2
TicTacToeParallel (T)	3	2			(1)	>**1hr**	≈2sec	SJ = Tictactoe ×2
Doubletictactoe (D)	4	2			(2)	>**1hr**	<1sec	SJ = Tictactoe ×2
TicTacHeaven (T)	4	2			(2)	>**1hr**	<2sec	SJ = 9 Tictactoe ensemble + 1 isolé
TicTacToeSerial (T)	4	2			(2)	>**1hr**	<1sec	SJ = Tictactoe ×2
ConnectFourSimultaneous (T)	4	2			(2)	>**1hr**	<1sec	SJ = Connect4 ×2
DualConnect4 (T)	4	2			(2)	>**1hr**	<1sec	SJ = Connect4 ×2
Jointconnectfour (S)	4	2			(2)	>**1hr**	<1sec	SJ = Connect4 ×2

FIGURE 10.7 – Résultat de la décomposition pour un panel de 40 descriptions de jeux provenant des serveurs de Dresde (D), Stanford (S) et Tiltyard (T) avec un commentaire décrivant les sous-jeux (SJ) identifiés.

(c) factorisation des conjonctions et des disjonctions
(d) construction du circuit
(e) collecte d'informations à partir du circuit : fluents explicitement indépendants des actions, préconditions, *sous-buts* et *conditions de victoire*
(f) collecte d'informations pendant des playouts
(g) analyse statistique des informations collectées : détection de l'effet potentiel des actions, des fluents de type *control* et actions *noop* correspondantes, des fluents indépendants des actions et des actions sans effets
(h) identification des *méta-actions*
(i) construction d'un *graphe causal* et identification des *points de croisement*
(j) construction du *graphe de dépendances* et identification des sous-jeux à partir des informations collectées en (e), (g), (h) et (i)
(k) vérification des sous-jeux à partir des *sous-buts* et des *conditions de victoire*
(l) étiquetage des sous-jeux dépendants ou indépendants des actions, filtrage des *sous-jeux inutiles*.
(m) retour à la phase (f) pour une autre tentative de décomposition avec plus d'informations le cas échéant (fig.10.9).

Pour vérifier la robustesse de cette seconde approche, nous l'avons testée sur significativement plus de jeux. Nous avons donc collecté et utilisé pour nos tests, 597 jeux [72] trouvés dans les dépôts *Base*, *Stanford* et *Dresden* de `http://games.ggp.org/` [Schreiber, 2017].

Il n'existe pas de données expertes indiquant ce qu'est la décomposition attendue pour chacune des descriptions de jeu présentes dans les dépôts. D'après les définitions de la section 10.1.1, nous avons donc établi, pour les 597 jeux trouvés dans les dépôts, le nombre de sous-jeux attendus avec séparément le nombre de sous-jeux dépendants des actions, indépendants des actions et pour chacun le nombre de *sous-jeux inutiles* (cf. Annexe B). Cependant, comme il est possible de partitionner les fluents d'un jeu en N sous-jeux de différentes manières, dans chacune de nos expériences, nous avons vérifié manuellement chaque décomposition signalant la détection de plus d'un sous-jeu pour nous assurer qu'elle soit en accord avec les définitions 10.1 à 10.3 [73].

Les valeurs des seuils et des ratios ont été ajustées empiriquement aux valeurs suivantes : $\Psi = 3$, $\Theta = 5$, $\Phi = 1.5$ et $\Omega = 0.1$. Ces paramètres sont peu sensibles à de légères variations. Par exemple la valeur de Ψ amène à ignorer les effets cooccurrents pendant les Ψ premières observations : la valeur de Ψ doit être assez haute pour écarter les coïncidences. Une valeur paranoïaque nécessitera juste un nombre plus important de *playouts* pour obtenir le même résultat.

Pour chaque jeu nous avons mesuré le temps nécessaire pour la construction du circuit, la réalisation de 5k *playouts* et le processus de décomposition. Les résultats présentés sur la figure 10.8 montrent que 70% des jeux peuvent être ainsi analysés en moins de 1 minute, temps compatible avec les contraintes des compétitions de GGP. La majorité du temps est utilisée pour la création du circuit. Par exemple, pour le jeu *Blocker Parallel*, une fois le circuit créé, 35 secondes sont suffisantes pour réaliser les 5k simulations

72. Nous avons exclu les descriptions GDL-II et certaines descriptions mal formées et injouables.
73. Une décomposition experte peut différer d'une decomposition *correcte* (def.10.3). Par exemple, un expert humain pourrait souhaiter décomposer *Nine Board Tictactoe*. Cependant, cette décomposition serait difficilement exploitable pour résoudre le jeu.

5 kp	<1s	<10s	<30s	<1min	<3min	<10min	<30min	<1h
total	72	307	391	434	491	527	540	557

FIGURE 10.8 – Nombre de jeux pour lesquels le processus de décomposition est achevé dans le temps donné (pour 5k *playouts*).

et la décomposition. Pour 40 jeux la décomposition n'a pas été obtenue en moins d'une heure, parmi eux 32 sont décomposables.

Le processus de décomposition peut être réitéré lorsque plus de simulations ont été effectuées. Les phases (a) à (e) ne sont effectuées qu'une seule fois. Les informations sont ensuite réutilisées tandis que les phases (f) à (l) sont répétées pour chaque tentative de décomposition : les informations collectées pendant les *playouts* supplémentaires sont cumulées avec les précédentes. Sur la figure 10.9 nous évaluons le nombre de décompositions correctes obtenues après différents lots de 1k *playouts* (10 au total). Nous constatons que 1k playouts sont suffisants pour décomposer correctement 87% des jeux. Le seul jeu sous-décomposé au bout de 10k playouts est *Simultaneous Win 2* pour lequel aucun sous-but (fonction d'évaluation) n'a pu être identifié pour les sous-jeux.

	1kp	2kp	3kp	4-6kp	7kp	8kp	9-10kp
under-decomp.	9	5	4	2	2	2	1
decomposed	521	525	527	529	530	532	533
over-decomp.	26	26	25	25	24	22	22

FIGURE 10.9 – Nombre de jeux correctement décomposés après chaque phase de 1k à 10k *playouts*.

Parmi les 597 jeux testés, 195 sont des jeux non décomposables. Cinq d'entre eux sont sur-décomposés au bout de 10k playouts. Les cas de sur-décomposition sont en majorité dus au manque d'information. Par exemple, pour les jeux *Snake* ou *Tron* certaines parties du plateau de jeu ne sont pas explorées pendant les *playouts* et constituent un sous-jeu séparé. D'autres cas de sur-décomposition sont observés pour les jeux qui consistent à survivre pendant N tours du jeu (*Queens*, *Max-Knights*) : une mauvaise action met fin au jeu. Dans ce cas, le rôle du sous-jeu principal est mal détecté : le compteur de coups est le seul sous-jeu jugé utile. Dans les jeux à coups simultanés comme *Point Grab* ou *Smallest*, chaque joueur est placé dans un sous-jeu distinct : les sous-buts identifiés ne font pas le lien entre les actions des deux joueurs.

Tous les jeux décomposables pour lesquels la décomposition n'a pu être obtenue en moins d'une heure, sont constitués d'un compteur de coups (*stepper*) associé à un sous-jeu dépendant des actions. Une détection ad-hoc des compteurs de coups permettrait de gagner du temps sur les jeux présentant cette structure particulière.

Des cas intéressants de décomposition ont été obtenus pour certains jeux qui ne semblent pas décomposables *à priori*. Par exemple le jeu *Snake Assemblit* présente la particularité d'avoir un plateau de jeu inutilement grand : la description du jeu prévoit un plateau de 50×50 cases quand seulement

6×6 sont utiles. Les cases inutiles sont détectées et placées dans un sous-jeu étiqueté comme inutile. [74] Dans les jeux *Nonogram* (5×5 et 10×10) le statut de certaines cases est décidable indépendamment des autres : ces cases sont donc isolées dans des sous-jeux distincts (fig.10.10). Le reste du plateau est découpé en différents sous-jeux dont le statut des cases peut être décidé indépendamment (fig.10.11). Cette décomposition peut permettre de résoudre le jeu plus rapidement.

		3	1	1	1	2
				1	1	
			1	1	1	
	3	0	0	11	6	0
	1	0	0	10	5	0
1	3	13	12	9	4	1
1	1	0	0	8	3	0
	3	0	0	7	2	0

Figure 10.10 – Illustration de la décomposition obtenue pour le jeu *Nonogram* 5×5. Le numéro dans chaque case représente le sous-jeu auquel elle appartient.

		3	5	3	5	1	7	7	4	2	4
				2		5	1	1	1	3	
	1	1	11	11	11	11	11	11	11	11	11	11
		5	11	11	11	11	11	11	11	11	11	11
1	2	1	11	11	11	11	11	11	11	11	11	11
	6	1	11	11	11	2	3	16	7	11	11	11
	7	1	11	11	11	10	1	13	4	11	11	11
4	2	1	11	11	11	2	2	18	9	11	11	11
2	3	1	11	11	11	2	2	15	6	11	11	11
	3	1	11	11	11	2	2	17	8	11	11	11
		5	11	11	11	2	2	14	5	11	11	11
		6	11	11	11	2	0	12	2	11	11	11

Figure 10.11 – Illustration de la décomposition obtenue pour le jeu *Nonogram* 10×10. Le numéro dans chaque case représente le sous-jeu auquel elle appartient.

Nous avons comparé le résultat de notre décomposition avec ceux des travaux de Günther *et al.* [2009] et Zhao *et al.* [2009] : nous obtenons une décomposition correcte pour tous les jeux testés dans ces documents [75]. Aucune mesure du temps de décomposition n'est indiquée par Günther *et al.* [2009] et Zhao *et al.* [2009]. La figure 10.12 met en parallèle les résultats obtenus avec nos deux méthodes de décomposition. Nous constatons que la seconde approche permet d'obtenir des temps proches de ceux obtenus avec les DNF, mais l'approche est bien plus robuste et apporte une décomposition plus fiable. Cette seconde approche permet d'obtenir une décomposition correcte pour *Chomp* et une décomposition de *Blocker Parallel* dans un délai inférieur à 1 heure bien que l'élimination des variables existentielles

74. Lors d'une discussion avec Abdallah Saffidine, nous avons établit qu'en pratique cette information est difficilement utilisable pour améliorer le niveau de jeu d'un joueur programmé. Les règles concernant les cases inutiles peuvent être éliminées pour simplifier le *circuit* et accélérer son évaluation. Cependant, dans le cas d'un jeu complexe, l'utilité de certains fluents peut ne pas être détectée par manque d'information suite à un nombre limité de simulations.

75. Excepté *Double Crisscross2* que nous n'avons pas pu tester car sa description n'est pas disponible dans les dépôts.

ne soit pas encore implémentée dans notre programme.

Les décompositions obtenues ont été calculées à partir de l'état initial du jeu. Quand plus d'informations sont disponibles suite à un lot supplémentaire de *playouts*, la décomposition est recalculée en réutilisant une grande partie des résultats, les tentatives ultérieures de décomposition sont donc plus rapides à calculer. À chaque tour du jeu, il est envisageable de détecter les fluents dont la valeur restera fixée pour le restant de la partie, de les éliminer du graphe et de détecter ainsi de nouvelles décompositions.

Jeu	DNFD	DNF	proba
Blocker Parallel	>1hr	>1hr	≈48min
Asteroids	<1sec	<1sec	<2sec
EightPuzzle	<2sec	<2sec	≈3sec
Checkers	>1hr	<12min	≈28min
Breakthrough	<16min	<16min	≈14sec
Sheep and wolf	>1hr	<5sec	≈6sec
Tictactoe	≈1sec	<1sec	<1sec
Nineboardtictactoe	>1hr	<2sec	<17sec
Tictactoex9	>1hr	<5sec	≈11sec
Chomp	<1sec	<1sec	<2sec
Multiplehamilton	<1sec	<1sec	<2sec
Multipletictactoe	<10sec	<1sec	≈1sec
Blockerserial	<20min	<10min	≈3sec
Dualrainbow	≈1min	<8sec	≈6sec
Asteroidsparallel	<1sec	<1sec	≈2sec
Dualhamilton	<1sec	<1sec	<2sec
Dualhunter	<2sec	<2sec	<3sec
Asteroidsserial	<1sec	<1sec	<4sec
LightsOnParallel	<8min	<1sec	<1sec
LightsOnSimul4	<8min	<1sec	≈3sec
LightsOnSimultaneous	<8min	<1sec	≈3sec
Nim3	<2sec	<2sec	<2sec
Chinook	<14sec	<14sec	≈21sec
Double tictactoe dengji	>1hr	<1sec	<1sec
SnakeParallel	>1hr	<2sec	<7sec
TicTacToeParallel	>1hr	≈2sec	<2sec
Doubletictactoe	>1hr	<1sec	<1sec
TicTacHeaven	>1hr	<2sec	≈17sec
TicTacToeSerial	>1hr	<1sec	<1sec
ConnectFourSimultaneous	>1hr	<1sec	<2sec
DualConnect4	>1hr	<1sec	<2sec
Jointconnectfour	>1hr	<1sec	<2sec

Figure 10.12 – Comparaison des résultats de notre première méthode de décomposition fondée sur l'analyse logique des règles avec le calcul de la forme normale disjonctive partiellement ou complètement développée (DNF, DNFD) avec la seconde pour 32 des 40 jeux testés. Les résultats sont identiques pour les 8 jeux restants.

Chapitre 11
Jouer avec les jeux décomposés

Sommaire

Dans ce chapitre nous commençons par examiner les différentes informations, disponibles dans un jeu global, qui sont perdues ou inaccessibles dans chaque sous-jeu pris séparément et les problèmes qui en résultent. À partir de ces contraintes, nous proposons une approche permettant de jouer avec les jeux décomposés et d'exploiter une décomposition pour résoudre plus facilement le jeu global. Cette approche, en cours de développement est testée sur quelques jeux solitaires.

11.1 Problèmes résultants des décompositions

La décomposition d'un jeu s'accompagne de la perte de certaines informations à l'intérieur des sous-jeux. Comme nous l'avons expliqué (cf. §8.8), tous les sous-jeux ne sont pas impliqués dans l'évaluation du score ou la terminaison du jeu.

Dans ce chapitre nous examinons les différentes informations perdues lors de la décomposition et les problèmes que cela engendre pour explorer les sous-jeux et exploiter cette décomposition dans un joueur. Nous examinons le problème de la terminaison, de l'évaluation du score et de l'identification des actions légales.

11.1.1 Progression et terminaison

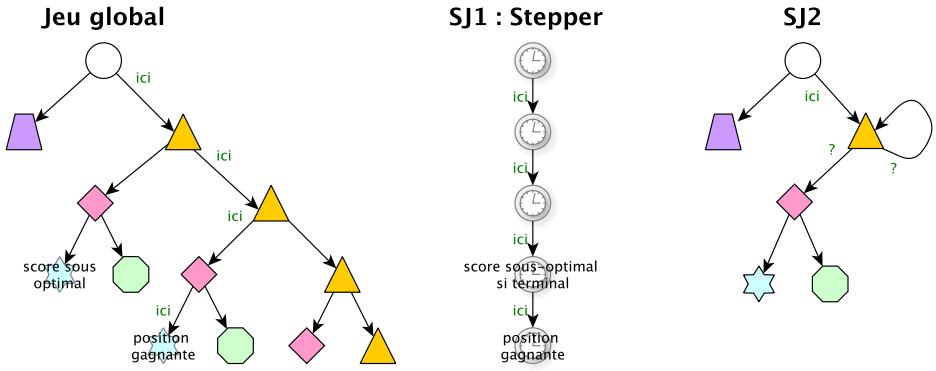
La décomposition d'un jeu en plusieurs sous-jeux dépendants des actions pose un problème pour l'identification des positions terminales. Une sous-position peut être terminale ou non en fonction du reste de la position globale. Dans un jeu comme *Incredible*, par exemple, une position du sous-jeu *Blocks* peut être terminale si le sous-jeu *Maze* est achevé, mais elle peut également être non terminale si ce n'est pas le cas. Il est possible d'identifier qu'une telle sous-position est *localement* terminale dans le sous-jeu *Maze* mais pas dans *Blocks*. On peut aussi imaginer un jeu dont deux sous-positions sont localement non terminales, mais où la conjonction des deux sous-positions provoque la fin du jeu. Pour exploiter les décompositions, il est donc nécessaire d'identifier la terminaison du jeu global pour éviter ces cas de *mort subite*.

Dans le cas des jeux comportant un *stepper*, comme *Eight Puzzle*, après la décomposition nous nous retrouvons avec au moins deux sous-jeux : un sous-jeu pour le *stepper* et un sous-jeu pour le reste ne respectant plus la contrainte de terminaison (def.1.1). Ce dernier contient des cycles qui étaient évités par la présence du *stepper*. Avec cette décomposition, nous espérerons exploiter les transpositions identifiables dans ce sous-jeu pour en accélérer considérablement l'exploration. Cependant, l'évaluation d'une position peut dépendre fortement de la proximité de l'état terminal. Dans un jeu décrit en GDL, la séquence de transitions la plus courte pour arriver à un état but n'est pas toujours la meilleure. Pour obtenir le score maximum, il peut parfois être nécessaire de perdre volontairement du temps pour arriver à la solution en un nombre de *steps* spécifique. Ainsi selon le *step* auquel une position du jeu est atteinte, la meilleure action n'est pas forcément la même. Un jeu contenant des cycles est représenté par un graphe. Si chaque position est représentée par un sommet unique dans ce graphe et si chaque action légale est représentée par un arc associé à une évaluation unique, alors il n'est pas possible de représenter la stratégie gagnante d'un jeu dont la solution la plus courte n'est pas la meilleure (fig.11.1). L'usage de transpositions peut donc amener à une évaluation erronée des positions en présence de cycles. Ce problème, est connu sous le nom de *graph history interaction problem* [Palay, 1985]. Kishimoto et Müller [2004] proposent une solution générale à ce problème cependant elle nécessite une procédure de vérification de l'évaluation des positions stockées dans la table de transposition pour identifier les positions qui ne constituent pas des transpositions exactes. Cette approche ne semble pas transposable dans le cas d'une recherche de type MCTS avec un score non binaire en fin de partie.

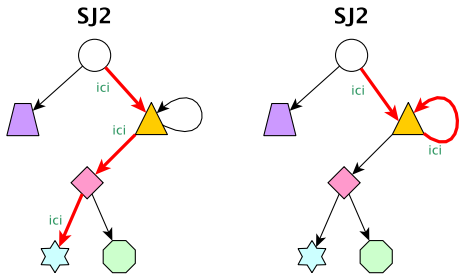
Nous constatons donc que pour exploiter les décompositions et les nouvelles transpositions qu'elles permettent d'identifier, il est nécessaire dans certains cas de reconstituer l'information manquante i.e. la progression et/ou la terminaison du jeu.

11.1.2 Évaluation du score

Notre méthode de décomposition garantit la présence d'une fonction d'évaluation pour chaque sous-jeu qui intervient dans le calcul du score. Pour évaluer une sous-position nous disposons donc des sous-buts et conditions de victoire identifiés, dont nous pouvons évaluer la valeur. Cependant, le score obtenu par l'évaluation des `goal` n'est fiable que dans le cas d'une position terminale. Nous avons vu dans la section précédente, qu'une sous-position peut être ou non terminale en fonction du reste de la position



(a) *Arbre du jeu global et des sous-jeux avec une indication des transitions à suivre pour atteindre la position gagnante.*



(b) *En fonction de la transition évaluée meilleure, soit la sous-position terminale gagnante est atteinte trop tôt (à gauche), soit la descente boucle sur la même transition jusqu'à ce que la position globale soit terminale (à droite).*

FIGURE 11.1 – Exemple de jeu décomposé en un *stepper* et un autre sous-jeu contenant un cycle. Les positions représentées par une même forme sont des transpositions presque parfaites au *stepper* près. La stratégie gagnante n'est pas indiquée pour le sous-jeu SJ2 car la meilleure transition dépend du *step* courant.

globale. Il convient donc de restaurer cette information.

Les précédents travaux [Günther *et al.*, 2009; Zhao *et al.*, 2009; Cerexhe *et al.*, 2014] suggèrent l'identification et l'utilisation d'expressions *sous-terminales* pour identifier les sous-positions potentiellement terminales dans lesquelles les sous-buts peuvent-être évalués. On peut aussi envisager de n'évaluer le score que pour des ensembles de sous-positions correspondant aux différentes parties d'une position globale. Dans ce cas c'est le score de cette position globale qui est évalué. Pour les jeux à score binaire, l'utilisation des sous-buts et des conditions de victoire est néanmoins indispensable pour évaluer les positions des sous-jeux. Le score global étant nul pour la plupart des positions terminales, il ne serait pas possible de guider efficacement la recherche vers la combinaison de sous-positions amenant à la victoire.

11.1.3 Choix libre d'une transition

Pour explorer les sous-jeux, il est nécessaire d'identifier les actions légales dans une sous-position donnée. Si le jeu est correctement décomposé, dans un sous-jeu donné, le calcul des actions légales ne dépend absolument pas des autres sous-jeux et une transition peut être librement choisie. Dans le cas des jeux à *actions composés* il est cependant nécessaire d'identifier les méta-actions légales. Ceci peut-être fait en évaluant le sous-circuit `legal` avec une logique tri-valuée permettant d'identifier les actions

potentiellement légales et d'en déduire les méta-actions correspondantes. Cependant, l'utilisation d'une logique tri-valuée implique une évaluation du *circuit* plus difficile qui nous laisse supposer une réduction du gain potentiel apporté par la décomposition. Il faut également envisager la possibilité d'une erreur de décomposition [76] : une erreur dans l'identification des méta-actions amène le risque de jouer des actions illégales ou, au contraire, de ne trouver aucune action légale dans une position.

Pour une action donnée, il faut également s'interroger sur la manière de calculer la sous-position suivante. Encore une fois, le *circuit* peut être évalué avec une logique à 3 états en indiquant que les fluents des autres sous-jeux ont une valeur indéfinie. Si la décomposition n'est pas correcte ou si une partie des fluents n'est pas encore identifiée comme appartenant à un sous-jeu ou un autre, l'état suivant peut être incohérent [77], ce qui fausse l'exploration du jeu et la décision résultante.

Le *circuit* peut aussi être décomposé pour évaluer plus rapidement les `legal`, `goal`, `next` et `terminal` de chaque sous-jeu et éviter l'usage d'une logique tri-valuée. Cependant, la création du circuit de chaque sous-jeu nécessite un traitement long qui peut lui aussi diminuer la performance de l'exploitation de la décomposition. Au pire, le jeu peut être totalement injouable si la décomposition est erronée ou *imparfaite* [78].

11.2 Exploration/exploitation des jeux décomposés

Dans l'optique de mettre au point une technique d'exploration d'un jeu décomposé – suffisamment robuste pour supporter une décomposition *imparfaite* et suffisamment performante pour compenser le surcoût de la décomposition – nous avons posé un certain nombre d'hypothèses concernant les propriétés que doit posséder un algorithme d'exploration de jeux décomposés (AEJD) efficace, et étudié différentes possibilités pour les mettre en œuvre et les vérifier.

Les approches précédentes de Günther [2007] et Zhao [2009] s'inspirent de la planification hiérarchique : ils recherchent l'ensemble des sous-plans locaux dans chaque sous-jeu avant de les recombiner en un plan global. Cette approche n'est pas envisageable pour des jeux complexes : en temps limité, il faut être en mesure de déterminer rapidement les premiers coups à jouer durant le *startclock* et affiner la stratégie pendant les *playclocks*. Il nous semble donc indispensable qu'un AEJD ne sépare pas les phases d'exploration locale et globale. Ces deux aspects doivent être opérés en parallèle. Il ne doit pas être nécessaire d'explorer totalement les sous-jeux avant d'utiliser l'information globalement.

Cerexhe *et al.* [2014] proposent une méthode de résolution fondée sur la vérification de modèles avec

76. Une sous-décomposition n'a pas de conséquence en dehors du fait qu'il n'est pas possible de tirer parti de la résolution de sous-jeux pour accélérer la résolution du jeu global. En revanche, dans un jeu sur-décomposé, des fluents normalement liés par une relation logique se retrouvent partitionnés en différents groupes. Ceci peut résulter du fait qu'un des effets d'une action n'a pas été testé durant les simulations. L'effet, ou la légalité, de cette action est lié aux fluents des différents groupes. L'évaluation des actions légales ou de l'état suivant risque donc d'être erronée dans les sous-jeux correspondants. Une mauvaise détection des effets des actions peut également amener à regrouper dans une même méta-action des actions composées qui auraient dû être séparées (une des actions à un deuxième effet qui la distingue des autres, mais il n'a pas été détecté), ou inversement à séparer ces méta-actions (le deuxième effet d'une des actions n'a pas été détecté et elle a été jugée différente des autres).

77. Admettons qu'une décomposition incorrecte a séparé un jeu en deux sous-jeux A et B. L'état suivant des fluents du sous-jeu A dépend de l'état courant de B. Pour estimer l'état suivant de A avec une logique tri-valuée, les fluents de B sont associés à une valeur indéfinie. Alors l'état suivant de A peut être incohérent et comporter des fluents dont la valeur est indéfinie.

78. Nous considérons qu'une décomposition est *imparfaite* si certaines actions n'ont jamais été testées durant les simulations destinées à collecter de l'information ou si le changement de valeur de certains fluents n'a jamais été observé (cf. §10.5.5).

ASP. Là aussi, l'approche n'est pas utilisable dans des jeux complexes pour obtenir un début de séquence d'action rapidement et l'affiner par la suite. De plus, cette approche est limitée aux jeux à un seul joueur. Il nous semble plus prometteur de chercher un AEJD fondé sur les techniques MCTS qui constituent l'état de l'art pour explorer progressivement les sous-jeux en parallèle

Puisque l'exploration des sous-jeux soulève différents problèmes pour déterminer les actions légales, identifier un état terminal et évaluer le score, nous proposons de faire des simulations globalement, mais construire des arbres (ou graphes) localement. Dans un premier temps, nous présentons une approche pour l'exploration des sous-jeux exploitant cette idée. Nous expliquons ensuite trois aspects de cet AEJD qui nécessitent de définir une politique spécifique : le marquage des branches totalement explorées nécessaire pour exploiter au mieux les transpositions, la sélection d'une action au niveau global pour prendre en compte le cas des jeux asynchrones, et la sélection locale d'un nœud dans un sous-jeu. Enfin nous présentons quelques résultats préliminaires sur des jeux solitaires et les perspectives qui s'en dégagent.

11.2.1 Simulations globales et construction d'arbres locale

Dans le but de simplifier l'application de notre idée de simulations globales et construction d'arbres locale, nous ignorons pour le moment les transpositions intervenant à des profondeurs différentes (*step* différents) dans l'arbre des sous-jeux. En revanche, deux sous-positions identiques à même profondeur sont représentées par un seul nœud. Ceci revient à associer un *stepper* à chaque sous-jeu. Si deux actions provoquent la même transition (même état de départ et d'arrivée), elles sont associées au même arc dans l'arbre et leur évaluation est partagée. La figure 11.2 illustre la manière dont nous proposons d'effectuer la construction des sous-arbres, l'évaluation de leurs positions et l'exploration du jeu. Prenons l'arbre partiel du jeu global décrit en haut de la figure. Il s'agit d'un jeu solitaire. Admettons que la décomposition de ce jeu indique qu'il existe 2 sous-jeux (asynchrones). Ce jeu ne contient pas d'actions composées et n'est pas un jeu en série. Les actions *a* et *d* sont liées au premier sous-jeu (SJ1) tandis que les actions *b*, *c* et *e* sont liées au second (SJ2).

A la phase 1, les actions *a*, *b* et *c* ont déjà été testées lors de simulations précédentes. Les sous-arbres de SJ1 et SJ2 possèdent déjà des arcs correspondant à ces actions associés à une évaluation [79]. Ils possèdent aussi des nœuds correspondant aux nouvelles sous-positions atteintes suite à l'exécution de ces actions. Remarquons que les actions *b* et *c* qui n'appartiennent pas à SJ1 laissent toutes deux l'état de ce sous-jeu inchangé. Ces deux actions sont donc associées à la même transition, représentée par le même arc dans le sous-arbre. Faire apparaître dans le sous-arbre une transition de ce type, qui laisse l'état du sous-jeu inchangé, permet d'évaluer l'intérêt de *jouer ailleurs* que dans le sous-jeu.

Dans la phase 1, nous sommes à l'état initial du jeu global qui est donné dans la description GDL. À partir de cette position il est possible au niveau global d'évaluer les actions légales. Les sous-jeux sont alors interrogés sur l'action à sélectionner pour effectuer une descente dans les sous-arbres.

Dans la phase 2, chaque sous-jeu procède à l'évaluation des actions indiquées comme légales et retourne la liste des meilleures actions trouvées i.e. les meilleures actions de même évaluation. Au niveau global une action est choisie en fonction des propositions des sous-jeux. Ici, c'est l'action *c* qui est sélec-

79. En pratique et en conséquence de l'utilisation d'une table de transpositions, le cumul des scores obtenus pendant les *playouts* et le nombre de visites des nœuds enfants sont stockés dans les arcs plutôt que les nœuds de l'arbre.

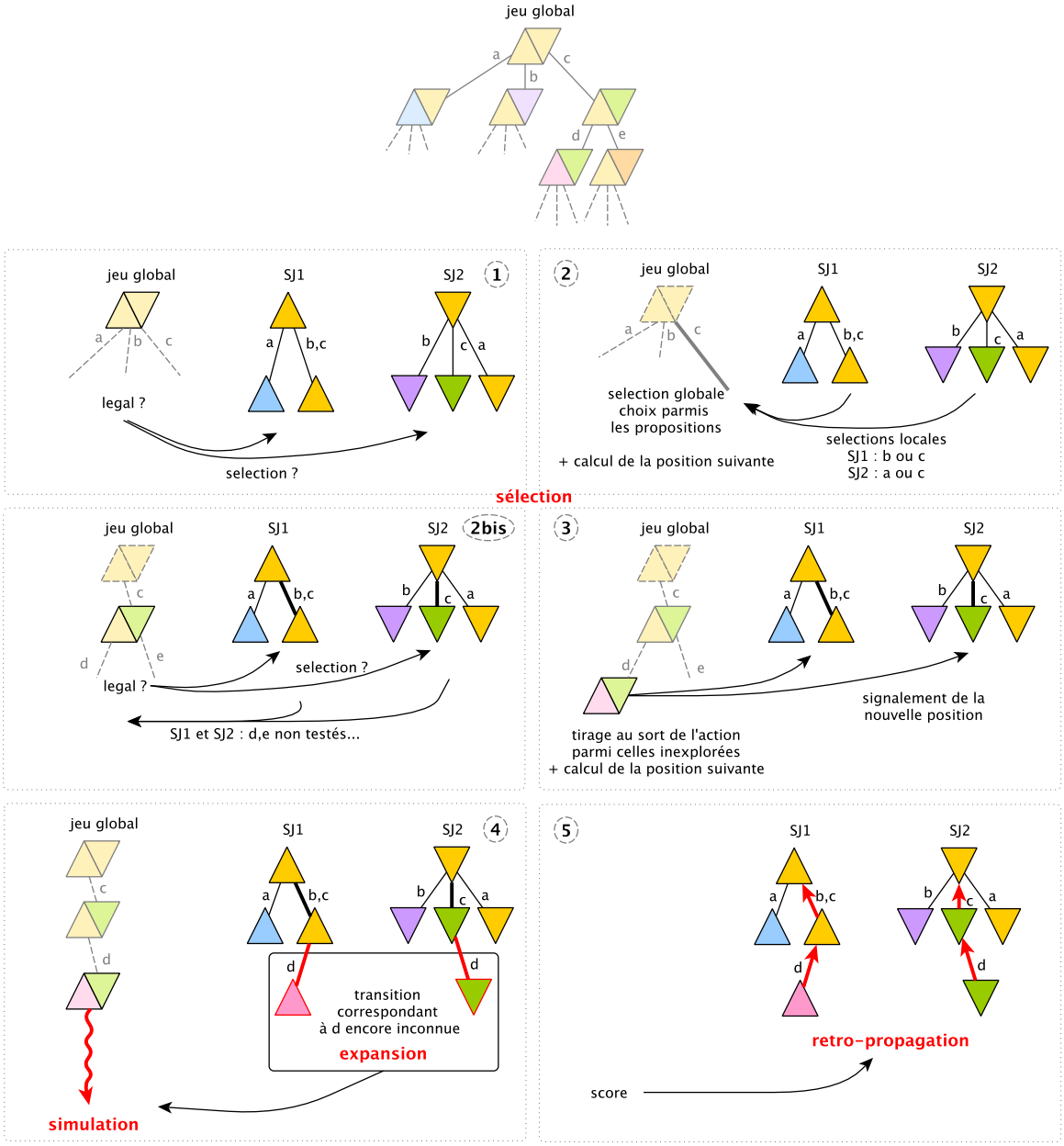


Figure 11.2 – Les 4 étapes de MCTS avec un jeu décomposé en utilisant des simulations globales et la construction d'arbres locaux. Les sous-positions identiques possèdent la même couleur dans un même sous-arbre.
(1) Les coups légaux sont évalués dans la position globale. Les sous-jeux sont interrogés pour connaître les actions légales qu'ils recommandent.
(2) Une action est sélectionnée en fonction des recommandations des sous-jeux. La position suivante est évaluée au niveau global. Les étapes 1 et 2 sont répétées pour faire la descente dans les sous-arbres.
(2bis) Si parmi les actions légales certaines n'ont pas encore été testées dans l'un des sous-jeux, celui-ci l'indique. La descente prend fin et on passe à l'étape (3)
(3) Une action est tirée au sort au niveau global, la position suivante est calculée et les sous-jeux sont informés de cette nouvelle position. Si la transition est déjà connue, l'action est ajoutée à cette transition et l'étape (3) est réitérée. S'il ne reste aucune action non testée, on retourne à l'étape (1). Sinon on passe à l'étape (4).
(4) Les sous-jeux réalisent une expansion : les sous-positions correspondantes à la nouvelle position sont ajoutées dans les sous-arbres puis une simulation est effectuée au niveau global.
(5) Le score obtenu à la fin de la simulation est remonté dans les sous-arbres pour évaluer les positions visitées pendant la descente.

tionnée. L'action choisie est jouée et la position suivante est calculée au niveau global. Les phases 1 et 2 peuvent être répétées pour effectuer la descente dans les sous-arbres. La position suivante et l'évaluation des actions légales étant calculés au niveau global, nous avons la garantie de jouer des actions légales et de préserver la cohérence de l'état du jeu.

La descente prend fin quand une action légale n'a pas encore été testée dans l'un des sous-jeux (phase 2bis). Les sous-jeux retournent alors la liste des actions non testées.

A la phase 3, une action non explorée est tirée au hasard, la position suivante correspondante est calculée et les sous-jeux sont interrogés pour savoir si cette position suivante correspond à une transition déjà connue. Si les nouvelles sous-positions sont connues de tous les sous-jeux, l'action est associée aux transitions correspondantes et la phase 3 est réitérée en tirant au sort une autre action non explorée. S'il ne reste plus aucune action non explorée, la descente se poursuit en retournant à la phase 1.

Si la sous-position atteinte est nouvelle pour au moins un des sous-jeux, une expansion est réalisée à la phase 4 pour ajouter, le cas échéant, la nouvelle transition et la nouvelle position dans chaque sous-arbre. Si une transposition existait, la position correspondante serait associée à la transition. Un *playout* est exécuté au niveau global, à partir de la nouvelle position.

A la phase 5, le score obtenu dans la position terminale atteinte à la fin du *playout* est transmis aux sous-jeux qui rétro-propagent l'information dans chaque sous-arbre pour évaluer les transitions. Au niveau global aucun arbre n'est construit. Seul l'état courant du jeu est maintenu pour permettre, à chaque étape de la sélection, de calculer les coups légaux et l'état suivant, puis d'effectuer un *playout*.

Une question importante est celle de l'application de cet AEJD sur toutes les classes de jeux composés, mono- ou multijoueurs. Le tableau de la figure 11.3 résume les principaux types de jeux composés et la structure des sous-arbres correspondants qui seraient générés par notre AEJD. L'examen de la structure de ces sous-arbres montre que l'AEJD est applicable sur les différentes classes de jeux sans nécessiter de modifier ses différentes phases.

Synchronisation	Nombre de joueurs	Alternance des actions	Structure
séquentielle	quelconque	coups alternés ou simultanés	(fig.11.4)
parallèle asynchrone	quelconque	actions simples ou composées	(fig.11.5)
parallèle synchrone	jeu solitaire	actions composées	(fig.11.6)
	jeu multijoueur	sous-jeux à coups alternés	
		actions composées	
		sous-jeux solitaires	

Figure 11.3 – Principales classes de jeux composés et la structure des sous-arbres correspondante.

Deux étapes clefs de cet AEJD doivent être examinées en détail :

- *Sélection locale* : comment choisir les meilleures actions d'après une sous-position ?
- *Sélection globale* : comment choisir l'action à jouer à partir des différentes *sélections locales* ?

Un problème particulier découle de l'usage de transpositions. Dans l'arbre d'un jeu, une position à la racine d'un sous-arbre peut avoir été visitée de nombreuses fois à la suite de différentes séquences d'actions. Pour éviter de redescendre inutilement dans un sous-arbre totalement exploré, il est courant de marquer les branches totalement explorées pour stopper la descente et/ou encourager la visite des branches voisines [Mehat et Cazenave, 2011; Winands *et al.*, 2008]. Cependant, dans le cas des sous-jeux,

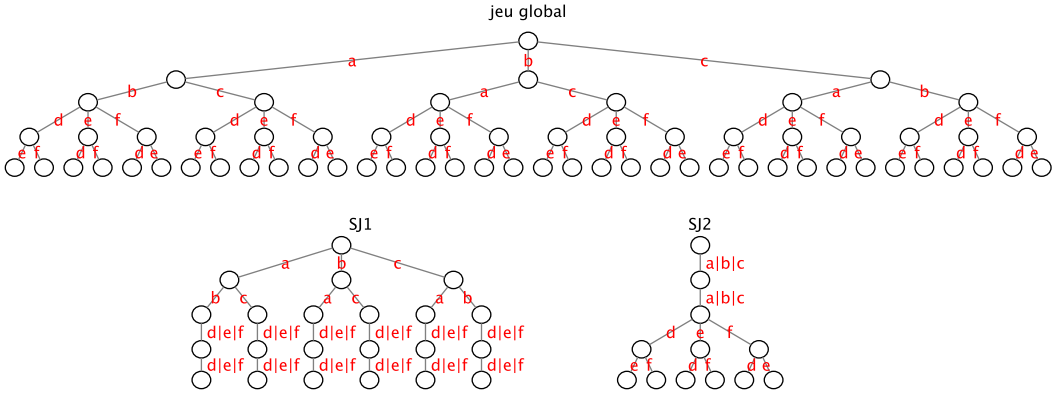


Figure 11.4 – Structure des sous-arbres pour un jeu solitaire ou multijoueur séquentiel (jeu en série). Dans le cas d'un jeu multijoueur, chaque niveau de l'arbre correspond aux actions du joueur qui a la main (les autres joueurs jouent *noop*).

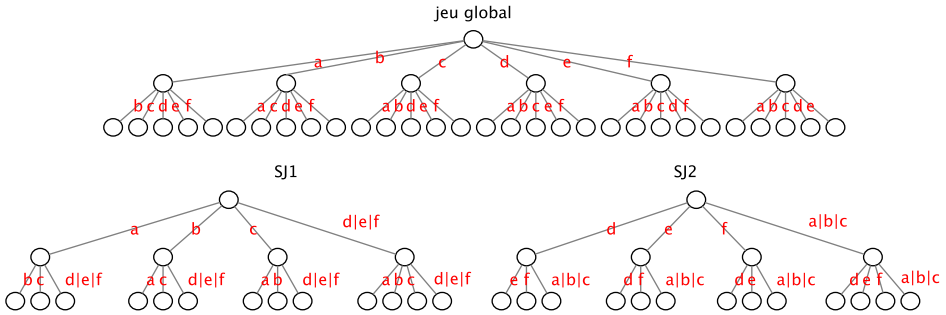


Figure 11.5 – Structure des sous-arbres pour un jeu solitaire ou multijoueur asynchrone. Dans le cas d'un jeu multijoueur, chaque niveau de l'arbre correspond aux actions du joueur qui a la main (les autres joueurs jouent *noop*).

une position n'est pas toujours terminale en fonction du reste de la position globale. Le marquage des branches totalement explorées dans les sous-arbres nécessite la mise au point d'une approche spécifique.

11.2.2 Marquage des branches totalement explorées

Le marquage des sous-positions terminales et l'utilisation de cette information pour marquer progressivement les sous-branches totalement explorées pose un problème illustré par la figure 11.7. Lors d'une première visite, une sous-position peut être marquée comme terminale. Cependant, lors d'une nouvelle descente dans l'arbre, la même transition peut être le résultat d'une autre action et ne pas mener à une sous-position terminale. Nous proposons de résoudre ce problème en révisant simplement le marquage lors des descentes successives.

Pendant la phase de sélection, la liste des actions légales de la position globale courante est calculée. Une action inconnue des sous-jeux est examinée pour savoir si elle correspond à une transition connue. Nous vérifions au même moment si la position suivante d'une transition déjà connue est bien terminale. Dans le cas contraire, le marquage est révisé. Lorsque toutes les transitions depuis la position courante sont signalées comme visitées au moins une fois par chacun des sous-jeux, nous vérifions dans chacun

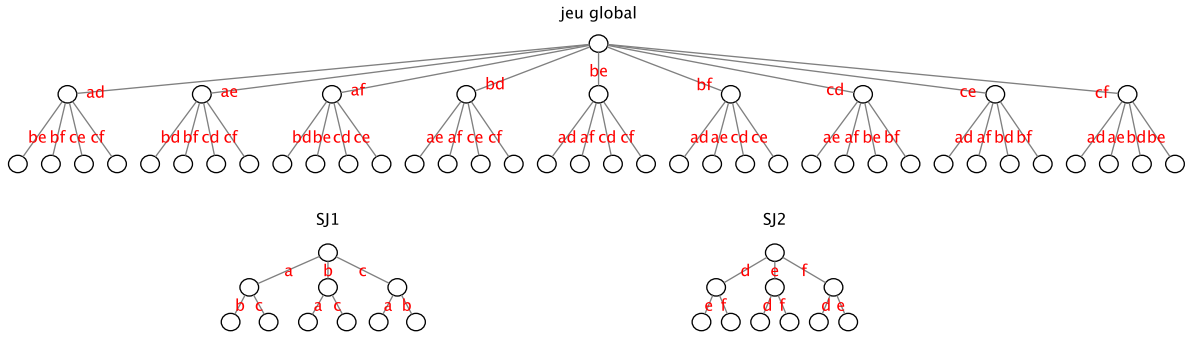


FIGURE 11.6 – Structure des sous-arbres pour un jeu solitaire ou multijoueur synchrone. Dans le sous-jeu SJ1, l'action *a* représente l'ensemble des actions *ad*, *ae* et *af*, de même pour les autres actions. Dans le cas d'un jeu solitaire, chaque couple de lettres représente une action composée. Dans le cas d'un jeu multijoueur synchrone, l'exemple illustré représente un jeu à deux joueurs et chaque couple de lettre représente un mouvement joint. Dans le cas d'un jeu avec des sous-jeux solitaires, la première lettre représente toujours l'action du premier joueur. Dans le cas d'un jeu avec des sous-jeux à coups alternés, la première lettre représente tantôt l'action du premier joueur, tantôt celle du second. Dans le cas d'un jeu avec des actions composées, les deux lettres représentent soit l'action composée d'un des joueurs (différent à chaque tour), l'autre joueur jouant une action *noop*, soit la combinaison des actions composées simultanées des deux joueurs.

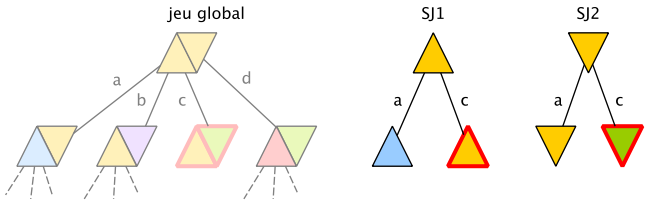


FIGURE 11.7 – Illustration du problème posé par le marquage des sous-positions terminales. À partir de la position initiale globale, les actions *a* et *c* ont été testées. L'action *c* mène à un état globalement terminal. Les sous-positions correspondantes sont marquées terminales. Cependant, dans SJ1 l'action *b* (non encore testée) peut mener à la même sous-position que *c*, mais cette fois elle ne sera pas terminale. De même dans SJ2, l'action *d* mènera à la même sous-position que *c*, mais non terminale.

des sous-jeux si ces transitions sont totalement explorées. Si une sous-position est la racine d'un sous-arbre totalement exploré, nous remontons l'information dans la transition précédente du sous-arbre.

Nous avons envisagé de placer le marquage des branches totalement explorées sur les sous-positions et/ou les transitions. Cependant cette approche s'avère inexacte (fig.11.8). Même en utilisant une approche gourmande qui consisterait à réviser le marquage en remontant récursivement l'information dans les nœuds parents, certaines branches pourraient être signalées totalement explorées de manière erronée. On peut vite imaginer les conséquences qu'aurait le choix de l'action *c* au lieu de *b* dans la phase (2) décrite à la figure 11.8. La deuxième branche sous la racine de chaque sous-arbre serait marquée comme totalement explorée et l'exploration des deux sous-jeux prendrait fin prématurément.

Nous procédons donc en marquant chaque action séparément. Cette approche, illustrée à la figure 11.9, nous dispense de réviser le marquage des branches totalement explorées. Dans cet exemple, si la séquence d'actions *b-d* est testée en premier, l'action *a* n'est pas marquée comme totalement explorée. Pour éviter cela, quand toutes les actions légales correspondent à des transitions déjà visitées dans

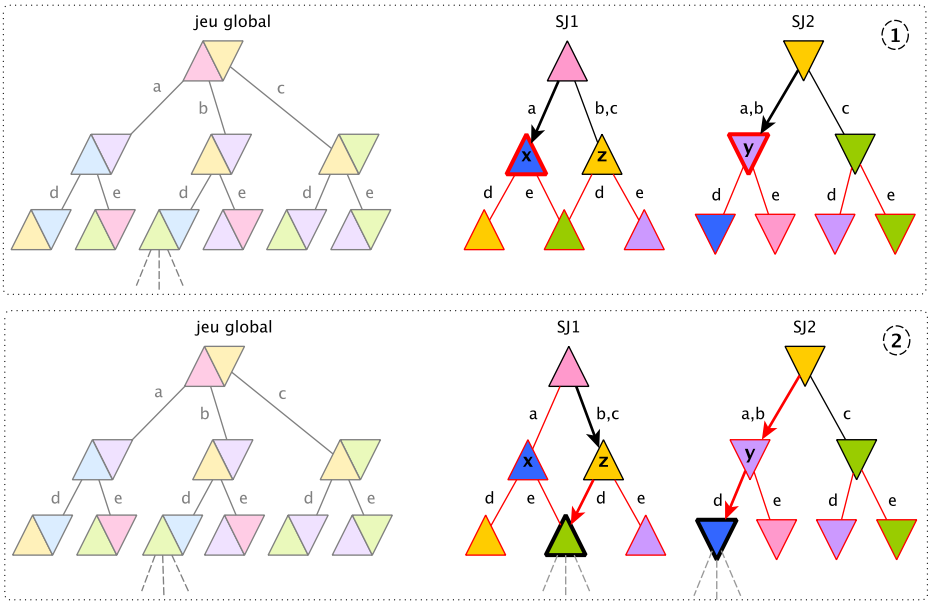


FIGURE 11.8 – Illustration du problème posé par le marquage des sous-branches totalement explorées si le marquage est placé sur les transitions. Les sous-arbres sont construits de manière incrémentale lors des descentes successives. À partir de la racine, les 3 actions *a*, *b*, et *c* sont testées successivement et évaluées par un *playout*. Les transitions et nœuds atteints sont ajoutés dans les sous-arbres. Lors des descentes suivantes, les séquences *a-d*, *b-e*, *c-d*, *c-e* et *a-e* sont testées successivement et aboutissent toutes à une position terminale. Les arcs et nœuds ajoutés dans les sous-arbres sont signalés terminaux.
(1) L'action *a* est à nouveau sélectionnée pour la descente suivante. Lors de la vérification des coups légaux au départ des nœuds *x* et *y*, les deux sous-jeux signalent que toutes les transitions possibles sont terminales. Les nœuds *x* et *y* et les transitions y menant sont marquées totalement explorées. La descente est alors stoppée.
(2) L'action *b* est testée à son tour (la transition correspondante dans SJ1 n'est pas totalement explorée). Au départ des nœuds *z* et *y* les deux sous-jeux signalent que l'action *e* est totalement explorée. L'action *d* est alors testée et les sous-jeux réalisent que la position suivante n'est finalement pas terminale : le marquage est supprimé mais les nœuds et transitions en amont sont toujours signalés comme totalement explorés.

les sous-jeux, la liste de celles provoquant la terminaison du jeu est calculée pour la position globale courante. Dans un sous-jeu l'action ayant amené à la position globale courante est marquée totalement explorée si (dans le sous-jeu) toutes les actions légales sont soit marquées totalement explorées, soit provoquent la terminaison du jeu.

Nos expérimentations montrent que ce marquage permet de détecter efficacement les branches totalement explorées. Dans le cadre du domaine de la complexité et de la combinatoire des problèmes, il serait intéressant de vérifier et d'établir une preuve théorique de l'exactitude de ce marquage pour tous les types d'arbres de sous-jeux.

11.2.3 Sélection globale

La *sélection globale* de la meilleure action est réalisée à partir des actions recommandées par les sous-jeux lors d'une *sélection locale*. Pour cette *sélection locale*, chaque sous-jeu évalue les actions signalées comme légales. Ces actions peuvent correspondre à une même transition locale, elles reçoivent alors la même évaluation. Chaque sous-jeu retourne donc un ensemble de meilleures actions.

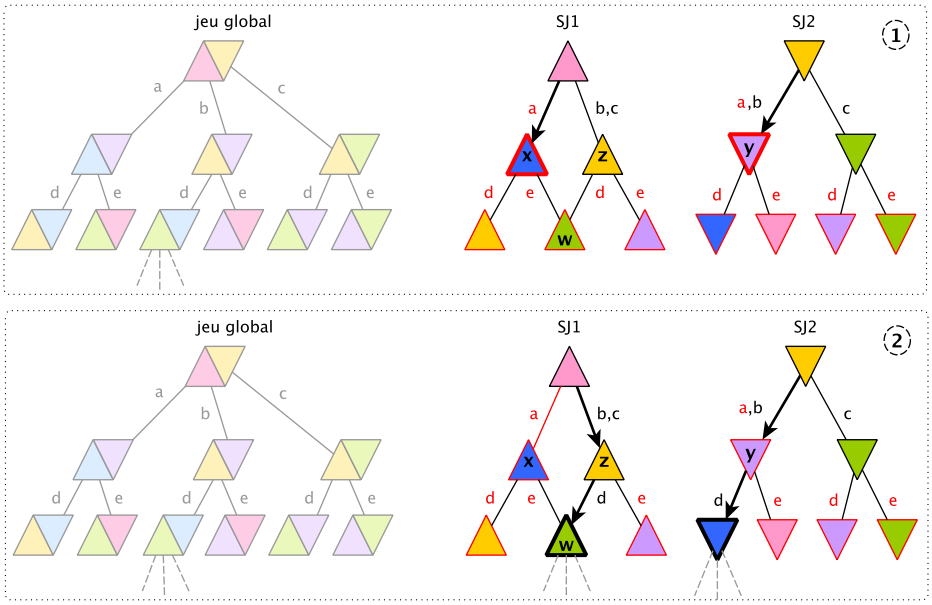


FIGURE 11.9 – Marquage des branches totalement explorées à travers le marquage des actions. À partir de la racine, les 3 actions *a*, *b*, et *c* sont testées successivement et évaluées par un *playout*. Les transitions et nœuds atteints sont ajoutés dans les sous-arbres. Lors les descentes suivantes, les séquences *a-d*, *b-e*, *c-d*, *c-e* et *a-e* sont testées successivement et aboutissent toutes à une position terminale. Les actions d et e aboutissant à des positions terminales sont marquées totalement explorées.
(1) L'action *a* est sélectionnée comme départ de la descente suivante. Lors de la vérification des coups légaux au départ des nœuds *x* et *y*, les deux sous-jeux signalent que toutes les transitions possibles sont terminales. L'action a est marquée totalement explorée et la descente est stoppée.
(2) L'action *b* est testée à son tour. Au départ du nœud *z* et *y* les deux sous-jeux signalent que l'action *e* est totalement explorée. L'action *d* est alors testée et les sous-jeux réalisent que la position suivante n'est finalement pas terminale : le marquage est supprimé. Le nœud w est signalé potentiellement non terminal, l'action d est signalée à explorer. Le marquage de l'action a est conservé.

Dans les jeux en série (cf. §11.4) et les jeux synchrones (cf. §11.6), si chaque sous-jeu soumet la liste des meilleures actions qu'il a sélectionnées, l'intersection de ces ensembles est forcement non vide. Le choix d'une action au niveau global est alors simple : une action peut être sélectionnée au hasard dans l'intersection de ces ensembles. En revanche dans un jeu asynchrone (cf. §11.5), les différents sous-jeux peuvent proposer des ensembles d'actions disjoints, il est alors nécessaire de définir une politique pour le choix d'une action au niveau global.

Nous proposons de définir une politique globale applicable aussi bien aux jeux en série et synchrones que asynchrones. Chaque action légale est associée à une évaluation : chaque sous-jeu recommandant une action apporte un point. Ainsi, si les ensembles d'actions retournés par les sous-jeux possèdent une intersection, les actions dans cette intersection obtiennent autant de points qu'il y a de sous-jeux. Dans le cas d'ensembles totalement disjoints, chaque action remporte au maximum un seul point. Dans le cas d'un jeu décomposé en N sous-jeux, ce principe évite de devoir évaluer l'intersection des différents ensembles d'actions recommandées par les sous-jeux.

Dans un jeu comme *Incredible*, le sous-jeu *Maze* est rapidement totalement exploré. Le sous-jeu propose alors systématiquement la séquence d'action permettant d'obtenir le gain maximum permis par

ce sous-jeu. Cependant, terminer ce sous-jeu met fin prématurément au jeu global. De nombreuses descentes dans les sous-arbres se terminent donc par la fin prématurée du jeu. Il faut un grand nombre de visites de l'action mettant fin à *Maze* pour qu'un algorithme de *sélection locale*, fondé sur un équilibre entre exploration et exploitation, dirige la recherche vers l'exploration des autres actions possibles (jouer dans le sous-jeu *Block*).

Afin d'éviter ce problème, les actions légales sont étiquetées en fonction de leur statut, terminal ou non. Une action est terminale si la transition associée mène à une position globalement terminale. S'il existe des actions terminales, le score pouvant être obtenu en les jouant est directement évalué au niveau global. Si une action terminale rapporte le score maximal possible pour le joueur courant (généralement 100 points), elle est directement sélectionnée. Sinon les actions légales non terminales sont soumises à la *sélection locale* des sous-jeux. Parmi les actions ayant remporté le plus de points, certaines peuvent être totalement explorées. Si toutes ces actions les mieux notées ne sont pas totalement explorées, celles totalement explorées sont écartées. Parmi les actions restantes celles offrant la plus grande espérance de gain sont sélectionnées. Cette espérance de gain correspond au produit de la probabilité de gain de chaque sous-jeu s :

$$\prod_{s=1}^{n} \frac{W_i^s}{N_i^s} / 100 \tag{11.1}$$

W_i^s correspond à la somme cumulée des gains obtenus pendant les *playouts* et N_i^s au nombre de visites de la transition étiquetée avec cette action. La moyenne des gains est divisée par cent pour obtenir une probabilité de gain comprise entre 0 et 1. Si plusieurs actions offrent la probabilité de gain la plus grande, une d'entre elles est sélectionnée au hasard.

Nous avons testé cette approche pour quelques jeux solitaires. Nous envisageons dans le futur d'étendre nos tests aux jeux multijoueurs et de valider cette méthode de *sélection globale* sur un large panel de jeux.

11.2.4 Sélection locale

A chaque étape de la sélection, un sous-jeu reçoit la liste des actions légales qu'il doit évaluer. La position courante d'un sous-jeu est associée à un nombre total de visites N. Chaque transition au départ de cette position est évaluée par un nombre de visites N_i et un cumul W_i des évaluations obtenues en jouant une des actions associées. Comme plusieurs actions peuvent être associées à la même transition, l'évaluation de ces actions peut retourner plusieurs meilleures actions.

Nous avons envisagé plusieurs manières de faire la *sélection locale*. La première est une application classique de la formule UCT (cf. §3.1.2) en cumulant dans W_i le score global obtenu à chaque *playout*. Cette approche n'est pas satisfaisante car une transition dans un sous-jeu peut alors recevoir ponctuellement une bonne évaluation sans y avoir participé : les points ont été obtenus grâce à une séquence d'action amenant à un score positif dans un autre sous-jeu. L'évaluation des positions du sous-jeu se trouve alors faussée. Un autre problème plus gênant encore survient dans le cas des jeux à score binaire comme les puzzles : le score est toujours nul jusqu'à ce que la solution soit trouvée. Le premier terme de la formule UCT s'annule et la recherche se ramène à une exploration en largeur d'abord. Avec une

recherche dans des sous-jeux, un problème d'oscillation peut alors apparaître (fig.11.10). Une évaluation prenant en compte la contribution d'un sous-jeu en particulier est nécessaire.

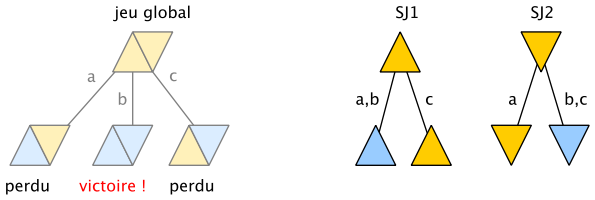


FIGURE 11.10 – Dans ce jeu composé minimaliste, le joueur a le choix entre 3 actions. Seule l'action *b* mène à la victoire. Admettons qu'UCT soit utilisé pour la *sélection locale* des actions et que le score global soit remonté à la fin des *playouts* pour évaluer les transitions. Si l'action *a* et *c* sont testées en premier et reçoivent toutes deux un score nul. À la troisième descente, les sous-jeux considèrent que toutes les actions ont déjà été testées puisque l'action *b* correspond à une transition déjà connue de chaque sous-jeu. Lors de cette troisième descente, toutes les actions reçoivent donc la même évaluation par UCT et une action est tirée au hasard. Si *a* est choisi, à la descente suivante, SJ1 recommandera de jouer *c*. Dans ce scénario pessimiste, la recherche peut continuer à osciller ainsi entre a et c très longtemps avant que *b* ne soit choisi par hasard.

Dans une position terminale globale, il est possible de garder uniquement la partie de la position correspondant à un sous-jeu et d'estimer une espérance de gain. En utilisant le circuit logique représentant la logique du jeu et en indiquant une valeur indéfinie pour les fluents n'appartenant pas au sous-jeu, il est possible de diffuser le signal dans le circuit avec une logique tri-valuée pour examiner la valeur des sorties `goal`. Ces dernières indiquent les scores qui peuvent être potentiellement obtenus en fonction de la valeur des autres fluents. Le `goal` de score maximum (*lmax*), indique le score maximum potentiel pouvant être obtenu si le reste de la position globale correspond à la meilleure configuration possible dans les autres sous-jeux. Le `goal` de score minimum (*lmin*), indique le score local minimum (*lmin*) possible si le reste de la position correspond à la pire configuration possible dans les autres sous-jeux. Plus précisément, *lmax* indique l'espoir de gain correspondant à la séquence de transitions menant à cette sous-position et *lmin* indique la participation du sous-jeu dans le score obtenu pour une position terminale globale contenant cette sous-position.

Logiquement $100-lmax+lmin$ indique le nombre de points maximum que le sous-jeu peut rapporter, mais il ne peut s'agir que d'une estimation minimale. En effet, ces scores *lmin* et *lmax* sont des valeurs indicatives. Les vrais *lmin* et *lmax* sont inclus dans cet intervalle, mais ne correspondent peut-être pas exactement. Ceci parce que l'utilisation d'une logique tri-valuée ne garantit pas l'information la plus précise possible [Reps *et al.*, 2002]. Les sorties `goal` du circuit avec la valeur *indéfinie* pourraient être évaluées à *faux* avec un circuit représentant une formule logique optimisée.

Ces scores *lmin* et *lmax* sont néanmoins une indication précieuse de la valeur d'une sous-position. On peut remonter ces espérances de gain pendant la phase de rétro-propagation et en faire le cumul W_i^{min} et W_i^{max} dans le but de les utiliser dans une formule UCT modifiée.

Lors de nos tests préliminaires, nous avons testé empiriquement différentes modifications de la formule UCT pour l'évaluation des sous-positions en remplaçant W_i par différentes expressions :

(1) W_i^{min}
(2) W_i^{max}

(3) $(W_i^{min} + W_i^{max})/2$
(4) $(W_i + W_i^{max})/2$
(5) $(3 \times W_i + W_i^{max})/4$
(6) $(W_i + 3 \times W_i^{max})/4$

Un examen plus systématique de l'influence de chaque évaluation (score global, *lmin* et *lmax*) dans différents jeux devra être effectué pour la mise au point d'une variante optimale de UCT pour les jeux décomposés. Il conviendra d'examiner d'autres alternatives pour l'évaluation des sous-positions et de vérifier l'application de cette *sélection locale* sur les sous-jeux multijoueurs.

11.2.5 Résultats préliminaires

Nous avons implémenté une version *beta* de notre AEJD limitée aux jeux à un seul joueur et testé notre programme sur une machine équipée d'un processeur Intel Core i7 à 3,1GHz et 16 Go de RAM. Pour chaque jeu, nous avons réalisé 10 tests avec chaque configuration. Le jeu est considéré résolu quand une feuille de score 100 est découverte.

Incredible			
10^5 *playouts* max	# playouts	temps (temps décomp.)	σ
W_i	10433	27.48s (2.73s)	9.3s
W_i^{min}	18113	51.52s (2.65s)	20.7s
W_i^{max}	16328	46.33s (2.73s)	13.7s
$(W_i^{min} + W_i^{max})/2$	16907	48.43s (2.91s)	19.4s
$(W_i + W_i^{max})/2$	13199	32.86s (2.50s)	4.4s
$(W_i + 3 * W_i^{max})/4$	16037	42.41s (2.60s)	14.8s
$(3 * W_i + W_i^{max})/4$	10607	25.71s (2.63s)	3.5s

Nonogramme 5×5 - *« damier »*					
10^5 *playouts* max	# playouts	temps (temps décomp.)	σ	# échecs	temps
W_i	-	-	-	10	7m19s
W_i^{min}	-	-	-	10	7m23s
W_i^{max}	647	4.59s (0.70s)	0.3s	-	-
$(W_i^{min} + W_i^{max})/2$	820	5.84s (0.75s)	0.4s	-	-
$(W_i + W_i^{max})/2$	776	4.81s (0.69s)	0.5s	-	-
$(W_i + 3 * W_i^{max})/4$	635	4.37s (0.73s)	0.6s	-	-
$(3 * W_i + W_i^{max})/4$	1296	7.44s (0.66s)	1.1s	-	-

Figure 11.11 – Moyenne sur 10 tests de résolution de *Incredible* et *Nonogramme* 5×5 - *« damier »* avec différentes variantes d'UCT pour la *sélection locale*. Les deux premières colonnes indiquent le nombre moyen de *playouts* effectués pour résoudre le puzzle et le temps moyen nécessaire pour cette résolution. Entre parenthèses figure la part du temps utilisée pour la décomposition. La colonne *échec* indique combien de recherches ont été stoppées sans que la solution ne soit trouvée. Le cas échéant, le temps moyen utilisé pour faire les 10^5 *playouts* est indiqué dans la dernière colonne. La colonne σ indique les écarts types. Globalement ceux-ci sont grands, du même ordre de grandeur que les temps de résolution ou un ordre en dessous. Ce qui signifie que les temps de résolution sont similaires pour toutes les politiques. L'association de W_i^{max} et W_i apparaît néanmoins souhaitable pour constituer une politique polyvalente.

Nous avons premièrement comparé différentes variantes d'UCT pour la *sélection locale* en remplaçant W_i par différentes expressions (fig.11.11). Nous avons testé ces variantes sur deux jeux : *Incredible* dont le score est progressif et monotone et *Nonogramme* 5×5 - *« damier »* (fig.11.14a) dont le score est binaire. Sans surprise l'utilisation de W_i ou W_i^{min} ne permet pas de résoudre un puzzle au score binaire. L'utilisation de W_i provoque le problème d'oscillation évoqué à la section 11.2.4 et, dans un jeu à score

binaire, le cumul W_i^{min} est toujours égal à 0 puisque le sous-jeu seul ne rapporte aucun point. Étonnamment, W_i^{min} qui indique la contribution d'un sous-jeu dans le score global ne donne pas non plus de bon résultat sur *Incredible*. Plus d'expérimentations nous semblent nécessaires pour expliquer ce résultat. La combinaison de W_i^{min} et W_i^{max} n'apporte pas non plus de bons résultats ce qui semble normal au vu des résultats précédents.

L'utilisation du score global W_i donne, dans l'ensemble, de meilleurs résultats sur *Incredible* tandis que l'espérance de gain W_i^{max} est meilleure pour *Nonogramme*. La combinaison des deux $(W_i + W_i^{max})/2$ permet d'obtenir un résultat satisfaisant pour les deux jeux. Les deux tests restants $(W_i + 3 * W_i^{max})/4$ et $(3 * W_i + W_i^{max})/4$ ont pour but d'identifier si la performance peut être améliorée pour *Incredible* en associant W_i avec une faible pondération de W_i^{max}, et si la pondération inverse est efficace pour *Nonogramme*. L'importance des écarts types, du même ordre de grandeur que les temps de résolution ou un ordre en dessous, ne permet pas d'identifier un effet significatif de la variation de cette pondération. En considérant ces écarts, les temps de résolution sont similaires pour toutes les politiques. Plus d'expérimentations seront nécessaires pour vérifier l'influence d'une pondération inégale de ces deux informations.

Pour avoir des résultats de référence auxquels comparer ceux obtenus avec notre AEJD, nous avons implémenté deux versions d'une recherche UCT dans un arbre unique avec usage de transpositions.

La première version est un UCT classique réalisé sur le jeu global (UCT-JG). Chaque nœud créé est associé à une structure de données de taille fixe prévue pour stocker l'évaluation de chaque transition au départ du nœud et l'identifiant de chaque nœud suivant atteint par chaque transition. Ces identifiants sont des clefs de hachage à la Zobrist calculées d'après la liste des fluents vrais dans la position courante. Les nœuds sont stockés dans une table de transpositions. Avant de créer un nouveau nœud, son existence préalable est vérifiée dans la table et il est réutilisé le cas échéant.

La seconde version est une version d'UCT modifiée de manière à construire l'arbre comme celui des sous-jeux i.e. l'arbre d'un unique sous-jeu correspondant au jeu global (UCT-1SJ). Un nouveau nœud est associé à une structure de données vide. Les transitions possibles à partir d'un nœud sont ajoutées progressivement dans cette structure et les actions testées sont associées à ces transitions. Comme pour la première version, les nœuds sont stockés dans une table de transpositions. Les clefs de hachage à la Zobrist sont calculées d'après la liste des fluents vrais, mais aussi d'après la profondeur du nœud correspondant dans l'arbre.

La différence majeure entre ces deux manières de construire l'arbre du jeu est que dans le second cas, plusieurs actions peuvent correspondre à la même transition. Dans ce cas, les scores cumulés et les nombres de visite sont partagés. Si deux actions provoquent la même transition, chacune d'elle sera deux fois moins visitée.

Dans les expérimentations suivantes, nous avons conservé la variante d'UCT utilisant $(W_i + W_i^{max})/2$ pour la *sélection locale* puisqu'elle donne globalement le meilleur résultat. Nous avons comparé notre AEJD avec les deux versions d'une recherche UCT dans un arbre unique avec usage de transpositions (UCT-JG et UCT-1SJ).

Pour le jeu *Incredible* (fig.11.12), le gain en terme de *playouts* nécessaires pour résoudre le jeu est significatif : notre AEJD nécessite 21 fois moins de *playouts* qu'un UCT classique. Cependant le gain en temps n'est que de 50% ce qui indique à quel point les simulations sont ralenties par la gestion des

Incredible			
	# playouts	temps (temps décomp.)	σ
UCT-JG	280158	1m	14.3s
UCT-1SJ	34660	7.45s	1.4s
AEJD	13199	32.86s (2.50s)	4.4s

Figure 11.12 – Moyenne sur 10 tests de résolution de *Incredible* avec un UCT classique réalisé sur le jeu global (UCT-JG), une version d'UCT modifiée de manière à construire l'arbre comme celui des sous-jeux (UCT-1SJ) et notre algorithme d'exploration des jeux décomposés (AEJD). Les premières colonnes indiquent le nombre moyen de *playouts* effectués pour résoudre le puzzle et le temps moyen nécessaire pour cette résolution. Entre parenthèses figure la part du temps utilisée pour la décomposition. La colonne σ indique les écarts types.

différents sous-jeux. Ceci est encore plus visible si on examine les résultats obtenus en construisant l'arbre global comme celui d'un sous-jeu (UCT-1SJ) : un gain énorme est apporté par l'utilisation d'une seule transition pour toutes les actions similaires (*contemplate*). Les nombreuses actions *contemplate* reçoivent la même évaluation et beaucoup de temps est économisé en évitant d'explorer chacune d'elles. La décomposition du jeu permet de le résoudre en moins de *playouts*, mais leur coût réduit beaucoup le gain apporté par la réunion des *contemplate*. Ceci est propre au jeu *Incredible* qui contient de nombreuses actions sans effet qui se trouvent donc réunies. Dans les expérimentations suivantes, nous constatons que la construction de l'arbre comme celui d'un sous-jeu provoque plutôt une dégradation de la performance. Vérifier si deux actions provoquent la même transition implique un surcoût de calcul. Cependant, pour certain jeu cette approche permet de le résoudre en moins de *playouts*. Le gain est significatif pour le *Nonogramme* 5×5 *« iG »* (fig.11.15). Plus d'investigations seront nécessaires pour expliquer ce résultat.

La comparaison de nos résultats avec ceux de Günther *et al.* [2009] (*Fluxplayer*) et Cerexhe *et al.* [2014] (ASP) est difficile étant donné qu'ils n'utilisent pas UCT pour résoudre les jeux. *Fluxplayer* met environ 2 heures pour résoudre *Incredible* en calculant plus de 41 millions d'états (à comparer aux 280 mille *playouts* pour UCT [80]). Leur méthode de décomposition réduit ce temps à 45 secondes et 3212 états calculés. Ce temps de résolution est supérieur au notre bien que moins d'états soient calculés par leur approche. Le codage de *Incredible* en ASP permet une résolution du jeu en 6,11 secondes. Ce temps est réduit à 1,94 secondes grâce à la décomposition soit un facteur de 3. Il faut noter que ces deux approches sont optimisées pour des jeux solitaires. Étant donné que notre approche nécessite 21 fois moins de *playouts* pour la résolution du jeu décomposé, en optimisant la vitesse d'exécution des simulations avec les sous-arbres, nous espérons obtenir un ratio similaire voire meilleur.

Pour les jeux *Queens08lg*, *Futoshiki4* et *Selective Sukoshi* (fig.11.13), nous constatons que le temps de décomposition surpasse le temps nécessaire pour la résolution du jeu par UCT-JG. Dans ces jeux très simples, la décomposition n'apporte pas de gain de temps. Cependant, on peut constater que pour *Futoshiki4* et *Queens08lg* la résolution du jeu décomposé nécessite de 3 à 5 fois moins de simulations. Pour un jeu comme *Selective Sukoshi*, où il suffit de gagner dans l'un des sous-jeux [81] pour remporter la victoire, on pourrait envisager de guider la recherche en ordonnant les sous-jeux de manière à toujours concentrer l'attention de la recherche sur un sous-jeu en priorité.

80. Chaque playout donne lieu à une expansion de l'arbre, on peut donc comparer le nombre de *playouts* au nombre d'états calculés.

81. On peut détecter qu'un sous-jeu rapporte la victoire à lui seul en examinant les sous-buts i.e. en identifiant qu'un sous-but rapporte 100 points.

Queens08lg			
	# playouts	temps (temps décomp.)	σ
UCT-JG	67	0.01s	<0.1s
UCT-1SJ	109	0.01s	<0.1s
AEJD	22	1.30s (1.29s)	<0.1s

Futoshiki4			
	# playouts	temps (temps décomp.)	σ
UCT-JG	13988	1.13s	0.9s
UCT-1SJ	19173	2.33s	2.0s
AEJD	6618	6.86s (1.44s)	8.7s

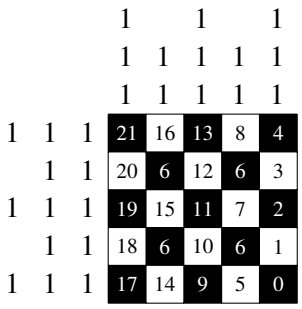
Selective Sukoshi			
	# playouts	temps (temps décomp.)	σ
UCT-JG	34	0.02s	<0.1s
UCT-1SJ	31	0.02s	<0.1s
AEJD	40	9.06s (8.98s)	0.3s

FIGURE 11.13 – Moyenne sur 10 tests de résolution de *Queens08lg*, *Futoshiki4* et *Selective Sukoshi* avec un UCT classique réalisé sur le jeu global (UCT-JG), une version d'UCT modifiée de manière à construire l'arbre comme celui des sous-jeux (UCT-1SJ) et notre algorithme d'exploration des jeux décomposés (AEJD). Les premières colonnes indiquent le nombre moyen de *playouts* effectués pour résoudre le puzzle et le temps moyen nécessaire pour cette résolution. Entre parenthèses figure la part du temps utilisée pour la décomposition. La colonne σ indique les écarts types.

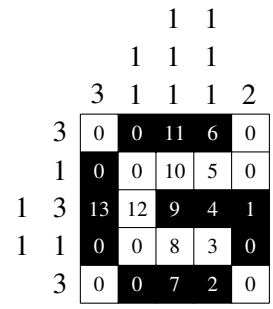
Pour les jeux de *Nonogramme* 5×5 et 6×6 (fig.11.15), nous constatons que la décomposition permet une résolution accélérée du puzzle. La résolution est 25 fois plus rapide pour le motif *« tetris »*. Ce gain n'est pas directement lié au nombre de sous-jeux identifiés puisque pour *« damier »*, qui présente le plus grand nombre de sous-jeux, la résolution n'est que 6 fois plus rapide. Il conviendra d'examiner par la suite la balance entre le surcoût engendré par l'exploration de plus de sous-jeux et le gain apporté par leur résolution accélérée. Le gain obtenu pour la vitesse de résolution est directement lié au nombre moindre de simulations nécessaires : de 300 fois moins pour *« iG »* jusqu'à 2400 fois moins pour *« damier clos »* (sans compter les 4 tests où UCT-JG a été interrompu après 10^7 *playouts* sans avoir trouvé la solution). La phase de sélection plus complexe rend chaque simulation beaucoup plus coûteuse, mais l'important gain dans le nombre de simulations permet une résolution nettement plus rapide. Nous constatons que le temps nécessaire à la résolution de *Nonogramme* 6×6 - *« cube »* avec notre AEJD est en moyenne d'une heure. La résolution avec UCT-JG est parfois couronnée de succès en moins d'une heure, mais la majorité des tests (8/10) a échoué à trouver la solution au bout de 5×10^7 playouts (3 heures en moyenne).

Pour un jeu comme *Nonogramme*, une décomposition en cours de jeu devrait apporter un gain encore plus significatif dans le temps de résolution : les cases isolées devant être marquées sont très vite identifiées. Une fois fixées, ces cases devraient permettre une décomposition plus importante du jeu mettant en évidence de nouveaux sous-jeux réduits à une case, etc.

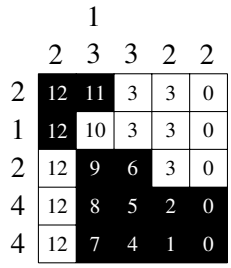
Nos premiers tests portant sur des jeux solitaires présentent des résultats prometteurs. Notre approche semble pouvoir être étendue directement aux jeux multijoueurs en réalisant une *sélection locale* indépendante pour chaque joueur, puis en choisissant l'action de chaque joueur par la *sélection globale*. Nos recherches futures nous permettrons de développer cette piste. La phase de sélection et la construction des sous arbres peuvent être optimisées pour minimiser le surcoût de ces calculs. Il conviendra de valider la performance de cette approche sur un large panel de jeux.



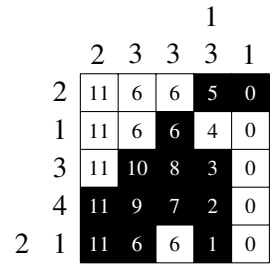
(a) Nonogramme 5×5 - « damier » *(22 sous-jeux)*



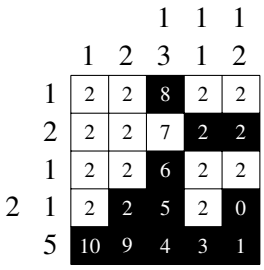
(b) Nonogramme 5×5 - « G » *(14 sous-jeux)*



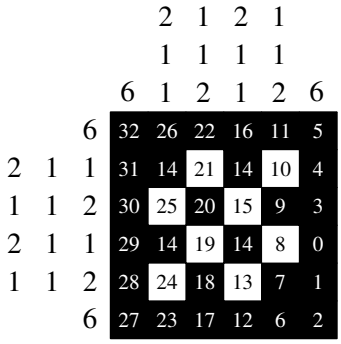
(c) Nonogramme 5×5 - « tetris » *(13 sous-jeux)*



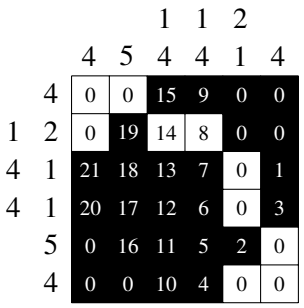
(d) Nonogramme 5×5 - « chien assis » *(12 sous-jeux)*



(e) Nonogramme 5×5 - « iG » *(11 sous-jeux)*



(f) Nonogramme 6×6 - « damier clos » *(33 sous-jeux)*



(g) Nonogramme 6×6 - « cube » *(22 sous-jeux)*

Figure 11.14 – Différentes grilles de *Nonogramme* 5×5 et 6×6. Les numéros à l'intérieur des cases sont les identifiants des différents sous-jeux détectés. Dans chaque ligne ou colonne prise séparément, s'il n'existe qu'une seule configuration permettant de respecter la contrainte indiquée, le statut de chaque case est décidable indépendamment des autres et ces cases constituent chacune un sous-jeu. Plus le nombre de sous-jeux détecté est important, plus la résolution du problème est facilitée (fig.11.15). La grille 11.14b représente cependant une exception. La performance moindre obtenue pour ce jeu semble s'expliquer par la structure du sous-jeu 0 qui englobe les 4 angles de la grille. Fixer le statut des cases constituant des sous-jeux à elles-seules ne permet pas de réduire le choix à une seule configuration possible dans chaque ligne et chaque colonne pour le sous-jeu 0.

Nonogramme 5×5 - *« damier »* (22 sous-jeux)			
	# playouts	temps (temps décomp.)	σ
UCT-JG	432019	31.31s	16.5s
UCT-1SJ	704452	1m	55.7s
AEJD	776	4.81s (0.69s)	0.5s

Nonogramme 5×5 - *« G »* (14 sous-jeux)			
	# playouts	temps (temps décomp.)	σ
UCT-JG	2988921	4m24s	4m8s
UCT-1SJ	2894537	7m5s	6m11s
AEJD	9344	36.61s (0.77s)	16.45s

Nonogramme 5×5 - *« tetris »* (13 sous-jeux)					
10^7 play. max	# playouts	temps (temps décomp.)	σ	# échecs	temps
UCT-JG	4969111	7m20s	3m53s	1	15m17s
UCT-1SJ	3828718	9m3s	3m28s	3	25m45s
AEJD	3767	16.36s (1.01s)	6.5s	-	-

Nonogramme 5×5 - *« chien assis »* (12 sous-jeux)					
10^7 play. max	# playouts	temps (temps décomp.)	σ	# échecs	temps
UCT-JG	2552082	3m43s	2m50s	-	-
UCT-1SJ	3573131	9m5s	7m9s	1	28m38s
AEJD	5476	26.27s (0.98s)	14.92s	-	-

Nonogramme 5×5 - *« iG »* (11 sous-jeux)			
	# playouts	temps (temps décomp.)	σ
UCT-JG	3086172	4m34s	3m9s
UCT-1SJ	1961394	4m19s	2m17s
AEJD	10232	28.60s (0.85s)	12.40s

Nonogramme 6×6 - *« damier clos »* (33 sous-jeux)					
10^7 play. max	# playouts	temps (temps décomp.)	σ	# échecs	temps
UCT-JG	4284872	13m7s	6m41s	4	31m9s
UCT-1SJ	6880213	28m35s	7m12s	6	42m50s
AEJD	1762	49.26s (2.40s)	3.72s	-	-

Nonogramme 6×6 - *« cube »* (22 sous-jeux)					
5×10^7 play. max	# playouts	temps (temps décomp.)	σ	# échecs	temps
UCT-JG	21438731	1h17m23s	19m16s	8	3h4m
UCT-1SJ	-	-	-	10	4h36m
AEJD	358608	1h53m8s (2.75s)	45m7s	-	-

FIGURE 11.15 – Moyenne sur 10 tests de résolution de différents *Nonogrammes* avec un UCT classique réalisé sur le jeu global (UCT-JG), une version d'UCT modifiée de manière à construire l'arbre comme celui des sous-jeux (UCT-1SJ) et notre algorithme d'exploration des jeux décomposés (AEJD). Les deux premières colonnes indiquent le nombre moyen de *playouts* effectués pour résoudre le puzzle et le temps moyen nécessaire pour cette résolution. Entre parenthèses figure la part du temps utilisée pour la décomposition. La colonne σ indique les écarts types. La colonne *échec* indique combien de recherches ont été stoppées sans que la solution ne soit trouvée. Le cas échéant, le temps moyen utilisé pour faire le maximum de *playouts* fixé est indiqué dans la dernière colonne.

11.2.6 Graphes vs. sous-arbres

Dans l'approche que nous avons proposée, nous avons écarté le problème posé par la décomposition d'un *stepper* en ne considérant les transpositions qu'à une profondeur donnée. Ceci revient à associer un *stepper* à chaque sous-jeu.

Dans un jeu comme *Eightpuzzle*, considérer toutes les transpositions diminue considérablement le nombre de nœuds explorés. Nous aimerions donc pouvoir remplacer les sous-arbres par des graphes pouvant contenir des cycles. Pour résoudre le problème de l'évaluation des transitions dans un graphe (fig.11.1a et 11.1b), nous proposons de stocker plusieurs évaluations dans les nœuds faisant l'objet d'un cycle. La structure stockant le cumul des scores et le nombre de visites de chaque transition sera transformée en matrice, chaque ligne correspondant à un passage distinct dans le nœud. À sa création, un nœud sera associé à une matrice vide d'une seule ligne correspondant à la structure que nous avons proposé d'utiliser pour stocker progressivement l'évaluation des transitions. Pendant la phase de sélection, la matrice d'un nœud sera agrandie si nécessaire de manière à pouvoir stocker l'évaluation d'une transition pour le $N^{\text{ème}}$ passage. Si aucune évaluation n'est encore stockée dans la ligne correspondant au passage courant, l'action correspondante sera signalée comme jamais testée. Une fois le score obtenu à la fin du *playout*, l'évaluation sera remontée dans chaque nœud. Si la même transition au départ d'un nœud a été visitée N fois, le cumul des scores et le nombre de visites seront mis à jour pour les N premières lignes de la matrice. De même, le nombre total de visites du nœud sera remplacé par une table indiquant le nombre total de visites pour chaque passage dans le nœud. Cette approche entraînera un surcoût en mémoire uniquement pour les nœuds impliqués dans un cycle.

Nous espérons dans le futur, implémenter cette idée et la tester sur les différents jeux disponibles en GDL. Il conviendra également d'étudier la possibilité d'utiliser les évaluations stockées lors des différents passages dans un nœud pour, par exemple, éviter une transition amenant à un cycle contre productif.

Chapitre 12
Conclusion

Les jeux composés

Nous avons présenté une classification des jeux composés. Nous avons vu que chaque classe de jeux composés présente des difficultés spécifiques, soit pour la décomposition, soit pour l'exploitation des jeux décomposés. Parmi ces types de jeux composés, ceux qui présentent les plus grandes difficultés pour la décomposition, indépendamment du nombre d'états et du nombre de sous-jeux, sont les jeux comportant des actions composées ou les jeux en série. Pour jouer, les jeux asynchrones représentent à priori le plus grand défi car il est nécessaire d'envisager à l'intérieur des sous-jeux toutes les alternances possibles d'actions des différents joueurs. Une autre difficulté à prendre en compte est la variété des sous-jeux identifiés : sous-jeux dépendants ou indépendants des actions (*stepper*), avec dans certains cas aucune influence sur le score ou la terminaison de la partie.

Les travaux précédents

Dans un second temps, nous avons présenté les différentes techniques utilisées pour la détection d'*invariants*, pour la détection de symétries et pour la décomposition dans les domaines de la planification, des CSP, des problèmes SAT, de la vérification de modèles et du GGP. Nous avons vu que les *invariants* permettent d'identifier des objets équivalents en planification ou des éléments de la structure d'un jeu (cases, coordonnées, pièces, marques) dans le domaine du GGP. La détection des *invariants* est une étape préliminaire pour certaines techniques de détection de symétrie. D'autres approches de détection de symétries sont fondées sur la détection d'isomorphismes dans un graphe représentant la structure du problème. Dans le domaine du GGP, l'identification d'une symétrie permet de décomposer un jeu en sous-jeux identiques. Dans les domaines autre que le GGP, les méthodes de décomposition permettent d'identifier des sous-problèmes faiblement liés. Les méthodes proposées ne sont pas directement applicables au GGP car celui-ci représente une classe de problèmes plus généraux.

Peu de travaux existent concernant le calcul de la décomposition des jeux en GGP. Les travaux de Günther [2007] et Zhao [2009] sont les seuls à proposer une approche pratique pour la décomposition des jeux décrits en GDL. Pourtant, les approches de Günther et Zhao sont loin d'apporter une solution suffisamment générale, robuste et efficace pour être applicable en compétition. Bien qu'il propose

des approches pour la décomposition des différents types de jeux (jeux impartiaux, à coups alternés, à coups simultanés, à actions composées ou en série), Zhao n'indique pas comment détecter le type de jeu rencontré pour y appliquer la méthode de décomposition et de résolution appropriée.

A la variété des jeux composés que nous avons présentée dans le chapitre 8, s'ajoute la multiplicité des manières de les décrire. Il n'existe pas de représentation canonique du GDL et différentes descriptions peuvent rendre un jeu plus ou moins facile à décomposer. Les approches de Günther et Zhao sont très tributaires de la syntaxe. Un jeu comme *Asteroid Parallel* résiste à la détection des actions composées proposée par Zhao. Toute méthode de décomposition s'appuyant sur la syntaxe peut être mise aisément en échec par une réécriture différente des règles. Le fait est que seuls quelques jeux simples peuvent être efficacement décomposés par leurs approches.

L'expressivité du GDL amène beaucoup d'écueils pour extraire des connaissances de haut niveau concernant les jeux. Durant les compétitions les règles sont offusquées, on ne peut donc pas compter sur les noms des fluents, actions ou variables pour identifier leur rôle. Certaines descriptions de jeu peuvent contenir des instructions mettant en échec la détection de caractéristiques fondées sur la structure des règles comme dans le jeu *Connect 5*. Certaines règles peuvent également n'être jamais satisfaites comme dans *Eight Puzzle* ou *Dual Connect Four* (version de Stanford).

De manière générale, le GDL ne contient aucune indication de ce que les prédicats signifient [Swiechowski et Mandziuk, 2014] et il n'y a pas de relation directe entre les fluents et les objets du jeu. C'est cette caractéristique qui permet au GDL d'être suffisamment expressif pour décrire n'importe quel jeu, et c'est elle qui rend extrêmement difficile la mise au point d'une méthode de décomposition robuste et générale. Tous ces écueils peuvent expliquer le faible nombre de travaux sur ce sujet.

Au niveau de l'utilisation des décompositions obtenues, les approches de Günther et Zhao pour la résolution des jeux décomposés impliquent une exploration exhaustive des sous-jeux ce qui est inapplicable dans le cas de sous-jeux complexes. Face aux joueurs les plus récents utilisant des *propnets*, circuits logiques ou SCSP pour effectuer un très grand nombre de *playouts*, on constate un surcoût de cette recherche par rapport gain apporté par la décomposition. Pourtant, il ne fait pas de doute qu'une exploitation efficace des jeux décomposés permettrait à un joueur d'améliorer significativement son niveau de jeu.

Notre méthode générale de décomposition

Nous avons proposé deux méthodes générales de décomposition permettant de traiter un jeu décrit en GDL sans avoir à détecter au préalable le type de composition dont il est potentiellement l'objet. Ces deux méthodes sont fondées sur la création d'un *graphe de dépendances* représentant les liens logiques existant entre les fluents et les actions du jeu. Les zones connexes de ce graphe représentent les différents sous-jeux. Les règles du jeu sont au préalable instanciées et transformées en *circuit* logique rapide à évaluer.

La première approche se fonde sur une analyse des règles GDL. Une heuristique est utilisée pour analyser les effets potentiels des actions. Afin de traiter le problème des actions composées, les règles, reformulées sous forme canonique (forme normale disjonctive), sont analysées pour identifier des méta-

actions. Ces méta-actions sont des ensembles d'actions ayant un même effet dans l'un des sous-jeux. Elles encapsulent les actions composées et permettent d'en isoler les différentes composantes pour séparer les sous-jeux. Les jeux en séries, composés de deux sous-jeux, sont séparés grâce à l'identification d'un *pivot* i.e. un couple de prédicats auxiliaires dont la valeur logique partitionne les actions légales en deux groupes. Les prédicats auxiliaires représentant des sous-buts sont utilisés pour regrouper les différentes parties des sous-jeux quand aucun autre lien logique ne les relie. Nous avons testé cette approche sur 40 jeux, composés et non-composés ; 32 sont décomposés en moins de 5 secondes ce qui représente un temps compatible avec les conditions des compétitions de GGP.

La seconde approche, fondée sur une analyse statistique d'informations collectées pendant des simulations, permet de résoudre différents problèmes posés par l'approche précédente. Les effets des actions sont déduits de l'analyse statistique des changements des fluents survenant pour chaque mouvement joint des joueurs, dans chaque transition des simulations. Cette approche est plus robuste, elle permet de détecter les effets, aussi bien positifs que négatifs, sans distinction entre ceux qui sont explicitement ou implicitement décrits dans les règles du jeu. Elle permet aussi d'identifier des *méta-actions* sans recourir à la forme canonique des règles, coûteuse à calculer. Les jeux en série composés de deux sous-jeux ou plus sont traités grâce à un *graphe causal* qui permet l'identification de *points de croisements*. Ces *points de croisements* sont des expressions caractérisant des positions nécessairement visitées pendant le jeu i.e. des positions correspondant à la fin d'un sous-jeu et au début d'un autre. Les sous-buts utilisés pour vérifier la décomposition et résoudre les cas de sur- ou sous- décomposition sont identifiés par l'analyse du *circuit*, ce qui évite le recourt aux prédicats auxiliaires pas toujours présents dans la description d'un jeu. Nous avons testé cette approche sur 597 jeux provenant des dépôts *Tiltyard*, *Stanford* et *Dresden*. Nos résultats montrent qu'il est possible d'obtenir une première décomposition rapidement : 87% des jeux sont correctement décomposés avec les informations collectées pendant seulement 1k playouts et 70% des jeux sont décomposés en moins de 1 minute en utilisant 5k *playouts*.

Nous avons ici envisagé la décomposition de jeux parfaitement séparables. Cependant, dans certains jeux comme *Tic-Block* [82] ou *Factoring George Forman* [83] des actions ou fluents sont réutilisés entre différentes parties du jeu. Une piste à développer serait la généralisation de notre méthode pour pouvoir décomposer les jeux dont les fluents ou les actions changent de rôle en court de partie.

Nous avons envisagé une décomposition des jeux en N parties disjointes. Certains jeux présentent une structure composée de plusieurs niveaux. Une autre piste à développer serait l'identification de ces différents niveaux de structure pour décomposer des jeux en sous-jeux se chevauchant. Les approches utilisées dans les domaines de la *planification hiérarchique* et *planification localisée* peuvent en être une source d'inspiration.

Nous avons également établi une nette séparation entre fluents dépendants et indépendants des actions. Hors, dans de rare cas, certains fluents peuvent selon la phase du jeu passer d'un statut à l'autre. De nombreuses exceptions ou particularités de ce genre peuvent être observées dans les descriptions en GDL qui rendent difficile la mise au point d'une analyse des règles de haut niveau.

Les approches précédentes, synthétisant les solutions de sous-jeux pour mieux résoudre un jeu global,

82. *Tic-Block* est constitué d'une partie de *Tictactoe* suivie d'une partie de *Blocker*. Les mêmes actions sont réutilisées d'une phase du jeu à l'autre pour marquer les cases de même index des deux plateaux.

83. Dans *Factoring George Forman* les 4 mêmes lampes sont réutilisées pour jouer 4 parties de *Lights On* en série.

se restreignent à certains types de jeux facilement décomposables de manière ad-hoc (puzzles ou jeux parallèles synchrones à 2 joueurs). Comme notre approche permet d'obtenir des décompositions robustes sur un large panel de jeux dans un temps compatible avec les délais imposés en compétition de GGP, il est désormais envisageable d'avoir un joueur en compétition exploitant les décompositions.

Tests préliminaires pour jouer avec les décompositions

Nous avons proposé une approche originale, fondée sur les méthodes MCTS, pour exploiter les décompositions : réaliser des simulations au niveau global et construire un ensemble de sous-arbres. L'évaluation des actions légales et le calcul de la position suivante au niveau global permet de résoudre les problèmes liés à la présence d'une information incomplète au niveau des sous-jeux. Les sous-jeux sont interrogés lors de la phase de sélection pour déterminer la meilleure action à chaque étape de la descente. L'expansion est réalisée dans l'arbre de chaque sous-jeu en fonction de la position atteinte au niveau global. Un *playout* est réalisé au niveau global et le score est remonté dans chaque sous-arbre pour évaluer ses nœuds. Les tests préliminaires que nous avons effectués sur quelques jeux solitaires sont prometteurs. Le jeu *Incredible* décomposé est résolu deux fois plus vite. Les grilles de *Nonogramme* testées sont toutes résolues plus rapidement : la solution est trouvée 25 fois plus vite dans le meilleur des cas.

Conclusion et perspectives

Ce manuscrit présente nos différentes contributions dans le domaine du *General Game Playing* (GGP) : une étude du potentiel d'une nouvelle architecture *manycore*, une méthode de décomposition des jeux fondée sur l'analyse des règles, une méthode statistique de décomposition et une méthode pour l'exploitation des décompositions dans un joueur programmé.

Notre première contribution est une étude de l'architecture *Multi Purpose Processor Array* (MPPA) de Kalray pour la parallélisation d'une recherche de type MCTS. La limitation de la mémoire à l'intérieur des clusters à une taille de 2Mo constitue la contrainte majeure de ce MPPA. Dans ce contexte, nous avons appliqué une *parallélisation aux feuilles* et utilisé le MPPA pour le calcul parallèle de *playouts*. Le MPPA permet un bon passage à l'échelle quand la taille des communications augmente, mais la répartition des messages vers les différents clusters présente un coût significatif. À l'intérieur des clusters, l'implémentation des primitives de synchronisation par des *spinlocks* implique qu'il est plus performant de relancer les *threads* pour chaque calcul. Le MPPA est une architecture intéressante car elle permet l'exécution de code et le traitement de données distinctes sur chaque cœur. Cependant, la puissance de calcul du MPPA sur 256 cœurs n'excède pas en pratique celle de la machine hôte équipée d'un processeur à 8 cœurs [84]. Il faudra attendre le complet développement de toutes ses fonctionnalités pour en estimer pleinement la performance.

Notre deuxième et troisième contributions sont deux méthodes générales de décomposition des jeux décrits en *Game Description Language* pour le GGP.

Notre première méthode de décomposition est fondée sur l'analyse logique des règles GDL développées sous forme normale disjonctive. Les effets des actions sont analysés par le biais d'une heuristique. La détection de *méta-actions*, des ensembles d'actions de même effet dans les mêmes circonstances, permet d'encapsuler les *actions composées* et d'en isoler les différentes composantes pour séparer les sous-jeux. L'identification d'un *pivot*, un couple de prédicats auxiliaires dont la valeur logique partitionne les actions légales en deux groupes, permet de séparer les *sous-jeux en série*. Des *prédicats auxiliaires* décrivant des *sous-buts* sont détectés pour assurer l'évaluation des sous-jeux. Cette approche a été testée

84. Le *Kalray MPPA-256 Andey* possède une puissance de calcul de 25 Gflop/W selon les spécifications, ce qui est légèrement supérieur à la puissance d'un processeur Intel i7 3820 à 3,6GHz à 8 cœurs. Notons qu'Intel propose depuis 2009 son processeur à 48 cœurs appelé le *SCC* et prépare son successeur le *D1X*. Les contraintes imposées par les architectures de type *MPPA* réduisent la potentielle efficacité de ces processeurs. L'augmentation du nombre de cœurs est une perspective intéressante mise en avant par les constructeurs.

sur un panel de 40 jeux comprenant 6 jeux simples et un ensemble représentatif des différentes classes de jeux composés ; 32 sont décomposés en moins de 5 secondes ce qui représente un temps compatible avec les conditions des compétitions de GGP.

Notre deuxième méthode de décomposition est fondée sur l'analyse statistique d'informations collectées pendant des *playouts*. Ces *playouts* sont réalisées grâce à un *circuit* logique similaire à un *propnet*. L'utilisation de données statistiques permet de s'affranchir des heuristiques, moins robustes pour résoudre le problème de l'identification des effets des actions. L'approche permet de détecter les effets aussi bien positifs que négatifs des actions, qu'ils soient explicitement ou implicitement décrits dans les règles GDL. Le *circuit* logique, construit à partir des règles et permettant d'effectuer rapidement les *playouts*, est également utilisé pour détecter les relations de précondition entre les fluents et les actions en rétro-propageant différents signaux. La détection de *méta-actions* est réalisée à partir des relations d'effet et de précondition préalablement détectées, ce qui permet d'éviter le coûteux calcul de la forme normale disjonctive des règles. Les transitions entre les sous-jeux d'un jeu en série sont identifiées grâce au concept de *points de croisements* mis en évidence dans un *graphe causal*. Toutes ces informations (effets, préconditions, *méta-actions*, *points de croisement*) sont utilisées pour construire un *graphe de dépendances* dont les parties connexes correspondent aux sous-jeux. Un post-traitement permet, à partir de *sous-buts* et de *conditions de victoires*, identifiés grâce au *circuit*, de valider les sous-jeux obtenus. Nous avons validé cette méthode de décomposition sur 597 jeux provenant des dépôts *Tiltyard*, *Stanford* et *Dresden*. Nos résultats montrent qu'il est possible d'obtenir une première décomposition rapidement : 87% des jeux sont correctement décomposés avec les informations collectées pendant seulement 1k playouts et 70% des jeux sont décomposés en moins d'une minute en utilisant 5k *playouts*.

Nous avons envisagé une décomposition en sous-jeux disjoints. Une autre piste à développer est l'identification de différents niveaux de structure pour décomposer des jeux en sous-jeux connectés, imbriqués ou se chevauchant comme dans les domaines de la *planification hiérarchique* et *planification localisée*. Disposer d'une méthode générale de décomposition offre des perspectives de recherche sur l'exploitation de ces jeux décomposés, d'autant plus que notre approche permet d'envisager la découverte de nouveaux sous-jeux en cours de partie.

Notre quatrième et dernière contribution est une méthode pour exploiter ces décompositions dans un joueur programmé. Pour jouer avec ces jeux décomposés, nous avons proposé une approche fondée sur les méthodes MCTS. Les simulations du jeu sont opérées sur le jeu global ce qui permet de calculer les transitions et les états du jeu de manière exacte. Des arbres, correspondant à chaque sous-jeu, sont construits à partir des positions partielles extraites des positions globales explorées. Ces sous-positions sont évaluées en rétro-propageant le résultat des *playouts*. Nous avons effectué des tests préliminaires sur *Incredible*, *Nonogramme* 5×5, *Nonogram* 6×6, *Queens08lg*, *Futoshiki4* et *Selective Sukoshi*. Les résultats sont prometteurs et ouvrent la possibilité d'avoir un joueur en compétition exploitant efficacement les décompositions.

Contrairement aux approches précédentes qui explorent totalement les sous-jeux pour ensuite synthétiser les résultats en une solution globale, notre approche permet d'améliorer progressivement l'évaluation des positions des sous-jeux en cours de partie. Une recherche de type MCTS pour l'exploration parallèle et simultanée de sous-problèmes ouvre également des perspectives de recherche pour l'applica-

tion de cette approche à d'autres domaines comme ceux de la planification, de la satisfiabilité booléenne, de la satisfaction de contraintes et de la vérification de modèles [85].

85. Günther [2007] propose des extensions dans le domaine de la programmation d'agents par application d'une stratégie de type "diviser pour régner"

Annexe A
Règles des jeux évoqués

Sommaire

A.1 Asteroids

Un joueur unique dirige un vaisseau dans un espace 2D, de 20 cases de coté, bouclé sur lui-même : le vaisseau disparaît au nord (resp. à l'est) pour réapparaître au sud (resp. à l'ouest) et inversement. À chaque tour du jeu, le joueur peut tourner le vaisseau d'un quart de tour sur sa gauche ou sa droite ou actionner la propulsion du vaisseau : dans ce cas la vitesse de déplacement du vaisseau augmente dans la direction à laquelle il fait face. Toute poussée appliquée au vaisseau continue à le déplacer par la suite (inertie) et ne peut être annulée que par une poussée dans la direction inverse. Les vitesses de déplacement selon l'axe nord-sud et est-ouest sont limitées entre -3 et 3 cases par tour. Le vaisseau est positionné initialement aux coordonnées $(10, 10)$, il fait face au nord, se déplace avec une vitesse de 3 en direction du nord et de 2 en direction de l'est (fig.A.1). Le joueur gagne 100 points s'il s'arrête à la position d'une planète aux coordonnées $(15, 5)$. S'il s'arrête à une autre position il met fin au jeu et ne gagne que 50 points. S'il est encore en mouvement au *step* 50, il perd. Il existe une solution optimale en 7 tours. Une estimation en fonction du facteur de branchement (3) et de la profondeur (51) indique que la complexité de l'arbre de ce jeu est au maximum de l'ordre de 10^{22}.

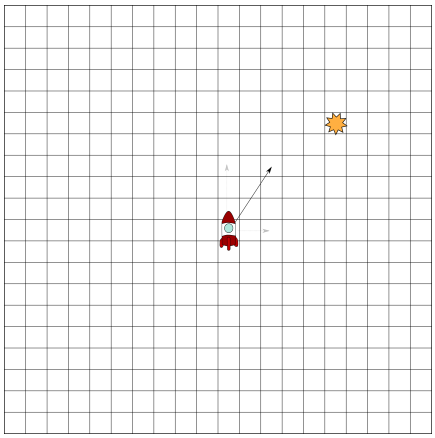


Figure A.1 – État initial pour le jeu *Asteroids*

Asteroids Parallel : Deux jeux de *Asteroids* sont joués de manière synchrone par deux joueurs solitaires. Les deux sous-jeux sont parfaitement indépendants. Le premier joueur qui arrête son vaisseau met fin à la partie. Le score de chaque joueur est calculé comme dans le jeu global.

Asteroids Serial : Deux jeux de *Asteroids* sont joués successivement par le même joueur. Le score global est la moyenne des scores des sous-jeux [86].

86. Une victoire dans un sous-jeu (100 points) et une défaire dans l'autre (0 points) permet d'obtenir 50 points au jeu global.

A.2 Blocker

Sur un damier 4×4 cases, deux joueurs *blocker* et *crosser* choisissent simultanément dans quelle case ils souhaitent poser leur marque. Si le joueur *blocker* a choisi la même case que *crosser*, alors c'est *blocker* qui pose sa marque dans la case et le coup de *crosser* est sans effet. Deux marques sont considérées contiguës si elles sont voisines sur la même ligne ou en diagonale. Si *crosser* arrive à rejoindre les bords droit et gauche du damier avec une série contiguë de marques (sans retour arrière), il met fin à la partie et gagne 100 points (voir image ci-dessous). Si aucune case vide ne subsiste, le jeu prend fin et *blocker* remporte 100 points. Il s'agit d'un jeu asymétrique qui, pour deux joueurs de force égale, offre beaucoup plus de chances de victoire à *blocker* : à mesure que le nombre de cases libres diminue, les chances de *blocker* d'anticiper l'action de *crosser*, de choisir la même case et de le contrer augmentent. La présence d'actions simultanées implique un important facteur de branchement. Par exemple, depuis la position initiale, 256 transitions sont possibles.

	B		
C	B		C
C	B	C	B
	C	B	

FIGURE A.2 – Position gagnante pour *Crosser* dans le jeu *Blocker*

Blocker Parallel : Deux jeux de *Blocker* sont joués de manière synchrone par deux joueurs. *Blocker* étant un jeu à actions simultanées, les deux joueurs ont des actions composées qui agissent dans les deux sous-jeux à la fois. Ce jeu est particulièrement lourd à instancier : les combinaisons possibles à l'intérieur des actions composées et entre les actions simultanées des deux joueurs amènent une importante combinatoire (16 actions par joueur, 16^2 actions composées par joueur, 16^4 actions composées simultanées). Le score global est la moyenne des scores des sous-jeux.

Blocker Serial : Deux jeux de *Blocker* sont joués successivement. Le score global est la moyenne des scores des sous-jeux.

A.3 Blocks ou Blocks World

Ce jeu pour un seul joueur consiste à obtenir un empilement de cubes (A sur B sur C). Le jeu commence avec la configuration représentée sur la figure A.3. À chaque tour, le joueur peut prendre un cube au sommet d'une pile et le poser sur la table ou l'inverse. S'il obtient l'empilement en 3 tours, ce qui correspond à la solution optimale, il gagne 100 points, sinon 0.

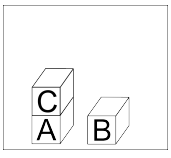


FIGURE A.3 – État initial pour le jeu *Blocks*

Blocks World Parallel : Deux jeux de *Blocks* sont joués de manière synchrone par le même joueur qui dispose d'actions composées. Le score global est la moyenne des scores des sous-jeux.

Blocks World Serial : Deux jeux de *Blocks* sont joués successivement par le même joueur. Le score global est la moyenne des scores des sous-jeux.

A.4 Breakthrough

Chaque joueur dispose de deux rangées de pions à chaque extrémité d'un damier de 8×8 cases (fig.A.4). À chaque tour l'un des joueurs peut avancer un pion d'une case en avant ou en diagonale. Une case ne peut contenir qu'une seule pièce à la fois. La capture d'une pièce adverse ne peut être effectuée qu'en se déplaçant en diagonale, le pion occupe alors la place devenue vacante. Le premier joueur à atteindre l'autre côté du damier met fin au jeu [87] et remporte 100 points, l'autre 0. *Breakthrough* est un jeu difficile pour UCT à cause de la profondeur et de l'important facteur de branchement de l'arbre du jeu.

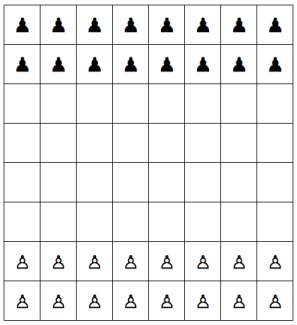


FIGURE A.4 – État initial pour le jeu *Breakthrough*

A.5 Buttons And Lights

Ce jeu solitaire est composé de 3 boutons et 3 lampes. Le premier bouton allume la première lampe, le second échange la première et la seconde lumière, le troisième échange la seconde et la troisième lumière. Toutes les lampes sont initialement éteintes, le joueur gagne 100 points et met fin au jeu s'il les allume toutes en 6 tours, ce qui correspond à la solution optimale, sinon il n'obtient aucun point. Ce jeu est extrêmement simple. L'arbre du jeu contient 1093 états sans tenir compte des transpositions. Ce jeu est pratique comme terrain de test : l'augmentation de la complexité de l'arbre de recherche dans les versions composées (de l'ordre de 10^{10} pour *Best Buttons And Lights Big*) encourage à l'analyse du jeu.

Multiple Buttons And Lights : 9 jeux de *Buttons And Lights* sont joués en parallèle par le même joueur. Seul le cinquième sous-jeu influence le score, les autres sont inutiles.

Joint Buttons And Lights : 3 jeux de *Buttons And Lights* sont joués en parallèle de manière synchrone par le même joueur qui dispose d'actions composées. Il suffit de gagner dans l'un des sous-jeux pour remporter la victoire.

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

Best Buttons And Lights : 9 jeux de *Buttons And Lights* sont joués en parallèle par le même joueur. Le joueur obtient 100 points s'il allume un des groupes de lampes, n'importe lequel.

Best Buttons And Lights Big : Ce jeu est une variante de *Best Buttons And Lights* avec 15 groupes de lampes.

A.6 Checkers Tiny

Il s'agit d'un jeu de *Dames Anglaises* où chaque joueur commence avec seulement 6 pions (fig.A.5). Le jeu prend fin au bout de 102 coups ou bien lorsque l'un des joueurs n'a plus de coup légal. Si l'un des joueurs possède plus de pièces que l'autre en fin de jeu, il gagne 100 points et l'autre 0. Si les joueurs sont ex-æquo ils gagnent 50 points chacun.

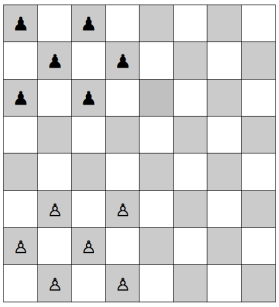


Figure A.5 – État initial pour le jeu *Checkers Tiny*

A.7 Chinese Checkers

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

A.8 Chinook

Ce jeu est composé de deux parties d'une variante des dames anglaises disputées simultanément sur les cases blanches et les cases noires du damier. Ce damier a la particularité d'être replié en forme de cylindre. Il n'y a pas de promotion des pions en reines. Le jeu prend fin au bout de 40 *steps* maximum.

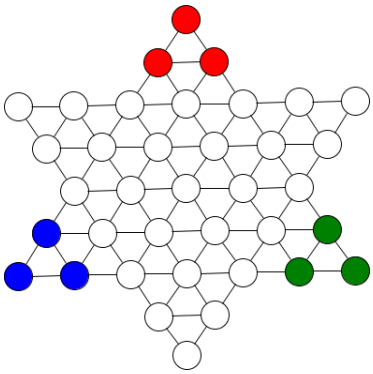


Figure A.6 – État initial pour le jeu *Chinese Checkers*

Dans la version de *Tiltyard*, les captures sont optionnelles et le premier joueur à mener une de ses pièces du côté opposé est le gagnant.

Dans la version de *Stanford*, le score est fonction du nombre de pièces capturées. Le joueur noir obtient le score atteint par la capture des pièces adverses sur les cases paires et le joueur blanc obtient le score correspondant aux captures sur les cases impaires. Le score maximum de 100 points est atteint après 8 captures.

A.9 Chomp

Deux joueurs doivent tour à tour manger une zone de $N \times M$ carrés de chocolat sur une plaque, en commençant par le coin supérieur gauche (fig.A.7). Le dernier carré en bas à droite étant empoisonné, le but du jeu est de ne pas être le dernier à mordre dans la tablette. Le vainqueur obtient 100 points et l'autre 0. Ce jeu est similaire à un jeu de *Nim* version misère avec une seule pile d'allumettes pour lequel il existe une stratégie gagnante (théorème de Sprague-Grundy).

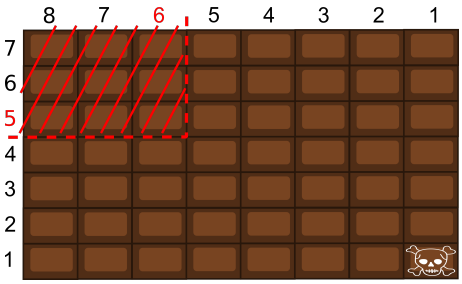


Figure A.7 – Exemple de premier coup au jeu *Chomp* : le joueur a choisi le carré $(5,6)$, tous les carrés d'index égal ou supérieur seront mangés. Le carré $(1,1)$ est empoisonné.

A.10 Connect Four

Il s'agit du jeu connu en France sous le nom de *Puissance 4* qui consiste à empiler des pions sur 7 (*Connect 4*) ou 8 (*Connect Four*) colonnes jusqu'à former un alignement de 4 pions (fig.A.8). Les joueurs jouent tour à tour et placent un pion de leur couleur (noir ou rouge) sur une des piles. Le premier joueur

qui arrive à aligner 4 pions à l'horizontale, à la verticale ou en diagonale gagne 100 points et l'autre 0. En cas de match nul chacun reçoit 50 points. La complexité de l'arbre du jeu est de l'ordre de 10^{21} [Bonnet et Saffidine, 2013].

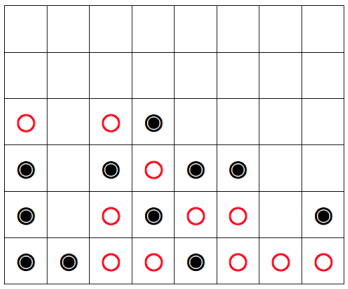


Figure A.8 – Exemple de position gagnante pour rouge dans le jeu *Connect Four*.

Dual Connect Four (ou Joint connect Four) : Deux jeux de *Connect Four* sont joués en parallèle de manière synchrone. Chaque joueur commence dans un sous-jeu et réplique dans l'autre sous-jeu au tour suivant. Une particularité de ce jeu est que l'un des sous-jeux est une version normale, l'autre une version suicide. Une victoire ou un match nul dans l'un des sous-jeux met fin au jeu. La victoire dans un des sous-jeux apporte 100 points au joueur qui a aligné 4 pions ou à son adversaire dans la version suicide. Si les deux sous-jeux se terminent en même temps par une victoire pour chaque joueur, les deux joueurs obtiennent 100 points. Les joueurs obtiennent 50 points si aucun d'eux n'a gagné dans un sous-jeu. Dans *Joint connect Four*, les règles ne sont pas formulées de la même manière, mais les deux jeux sont équivalents.

A.11 Domineering

Domineering est un jeu combinatoire [Berlekamp *et al.*, 1982] à deux joueurs, à coups alternés, disputé sur une grille de dimensions variables. À chaque tour, un joueur doit placer un domino (sans marque) couvrant 2 cases encore libres, horizontalement pour le premier joueur et verticalement pour le second. Le premier joueur qui ne peut pas poser de domino a perdu. Ce jeu a la propriété de devenir décomposable en court de partie quand le plateau est séparé en zones distinctes par les dominos déjà présents (fig.A.9).

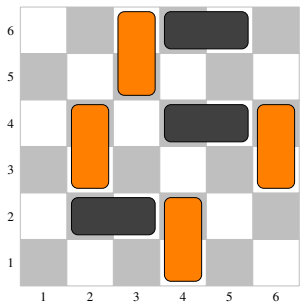


Figure A.9 – Exemple de position du jeu *Domineering* où le plateau de jeu se retrouve scindé en zones distinctes et où le jeu devient décomposable (ici en 3 parties).

A.12 Eight-Puzzle

Il s'agit d'un jeu à un seul joueur connu en France sous les noms de *pousse-pousse* 3×3 et de *taquin* 3×3. Il est constitué d'une grille de 3×3 cases contenant 8 tuiles numérotées de 1 à 8 et un espace vide. Au début du jeu, les tuiles sont dans le désordre. À chaque coup, le joueur peut faire glisser une tuile située à côté de l'emplacement vide vers celui-ci, le but étant de replacer les tuiles dans l'ordre (fig.A.10). Si le joueur y parvient en exactement 30 coups, il obtient 100 points. Sinon, s'il y parvient en moins de 60 coups il obtient 99 points. Le jeu prend fin quand le joueur a réussi, sinon il est stoppé au bout de 60 coups et il obtient un score de 0. Le jeu possède $9!/2$ états possibles mais, en GDL, l'ajout d'un *stepper* pour assurer la terminaison du jeu ne permet pas de détecter l'ensemble des transpositions.

1	2	3
4	5	6
7	8	

Figure A.10 – Position gagnante pour le jeu *Eight-Puzzle*.

A.13 Futoshiki 4

Une grille de 4×4 cases est partiellement remplie de chiffres. Entre certaines cases un symbole indique une inégalité (plus grand ou plus petit) (fig.A.11). Le but du jeu est de remplir la grille avec des entiers de 1 à 4 de manière à ce qu'il n'y ait aucun entier dupliqué sur une ligne ou sur un colonne et de manière à respecter toutes les inégalités. Le jeu prend fin quand il n'est plus possible de placer un chiffre sans violer une des règles. Le joueur obtient 100 points s'il a rempli la grille, 0 sinon. Des versions à 5×5 et 6×6 cases sont également présentes dans les dépôts.

-	<	-	<	-	-
-		2		-	-
					∧
-		-		-	-
-		-	<	-	-

***(a)** Grille vierge : état initial du jeu*

1	<	3	<	4	2
4		2		3	1
					∧
2		4		1	3
3		1	<	2	4

***(b)** Grille complétée : unique position gagnante possible*

Figure A.11 – Jeu de *Futoshiki 4*.

A.14 Hamilton

Hamilton est un jeu à un seul joueur se déroulant sur un graphe (fig.A.12) ou chaque sommet est connecté à trois autres sommets. Le joueur démarre sur l'un des 20 sommets du graphe et à chaque tour se

déplace vers un sommet connexe. L'objectif du jeu est de visiter le maximum possible de sommets en 20 tours. La récompense est proportionnelle au nombre de sommets visités. Ce jeu incarne le problème d'une recherche de chemin hamiltonien dans un graphe avec des ressources limitées. Une solution optimale consiste à visiter les sommets dans l'ordre alphabétique.

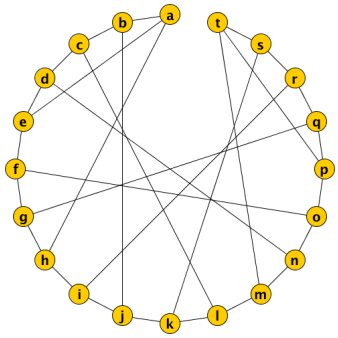


FIGURE A.12 – Graphe utilisé dans le jeu *Hamilton*.

Dual Hamilton : Deux jeux de *Hamilton* sont joués simultanément par deux joueurs solitaires. La récompense du joueur dans chaque sous-jeu est calculée comme dans le jeu de *Hamilton*. Le jeu est à somme non-nulle.

Multiple Hamilton : Un joueur joue à deux jeux de *Hamilton* simultanément. À chaque tour, il choisi dans quel sous-jeu il souhaite jouer. Seul le jeu de droite a une influence sur le score, l'autre est inutile.

A.15 Hex

Ce jeu pour deux joueurs se joue sur un plateau en forme de losange dont les cases sont hexagonales (fig.A.13). Chaque joueur à son tour place sa couleur dans une des cases. Le but du jeu est de créer une ligne traversant le plateau et joignant les deux côtés opposés de la couleur du joueur. La version GDL est décrite pour un plateau de 9×9 cases. Pour information, la complexité de l'arbre du jeu pour un plateau de 11×11 cases est de l'ordre de 10^{98} [Bonnet et Saffidine, 2013].

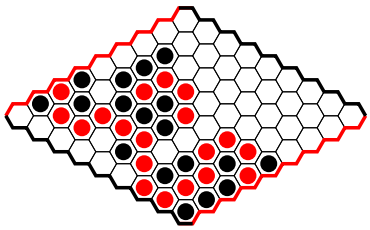


FIGURE A.13 – Exemple de position gagnante pour rouge au jeu de *Hex*.

A.16 Hunter

Hunter est un jeu solitaire se déroulant sur un plateau de 5×3 cases. Le jeu démarre avec un cavalier dans un angle du plateau et des pions dans toutes les autres cases. À chaque tour, le cavalier se déplace

comme aux échecs capturant le pion éventuellement présent dans la case ou il se déplace (fig.A.14). Le but du jeu est de capturer le maximum de pions possible en 14 tours. Remarque : on ne peut capturer que 13 pions en 14 déplacements, il est donc impossible d'atteindre le score maximum (100) pour ce jeu [88].

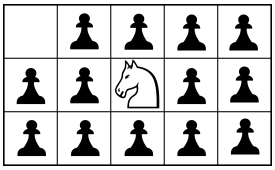


Figure A.14 – Exemple de premier coup au jeu *Hunter*. Le cavalier, positionné dans l'angle en haut à gauche à l'état initial, vient de capturer un pion au centre du plateau.

Dual Hunter : Deux jeux de *Hunter* sont joués de manière synchrone par deux joueurs solitaires. La récompense du joueur dans chaque sous-jeu est calculée comme dans le jeu de *Hunter*. Le jeu est à somme non-nulle.

A.17 Incredible

Ce jeu solitaire associe un jeu de *Maze* avec un jeu de *Blocks* à 6 cubes et un groupe d'actions inutiles de contemplation. Le joueur obtient le score maximum s'il crée deux tours (b sur d sur f et c sur a sur e) dans le jeu de *Blocks* et si le robot ramène l'or dans la première case du jeu de *Maze*. Le jeu prend fin au bout de 20 tours. Le joueur obtient un score proportionnel au nombre de sous-buts atteints (pour chacune des deux tours et pour l'or). La difficulté de ce jeu tient au fait que le jeu prend également fin au moment ou le robot dépose l'or dans la première case de *Maze* : si les autres sous-buts ne sont pas remplis à ce moment là, le joueur peut mettre fin au jeu sans avoir atteint le score maximum.

A.18 Lights On

Ce jeu solitaire trivial est composé de 4 minuteurs et 4 lampes. Le joueur peut, à chaque tour, actionner un minuteur qui allumera alors sa lampe pour une durée de 4 tours. Le but du jeu est d'allumer les 4 lampes en même temps en moins de 20 tours : le joueur obtient 100 points s'il atteint ce but, 0 sinon. La solution optimale est obtenue en 4 tours en allumant les lampes l'une après l'autre. Le seul intérêt de ce jeu réside dans la simplicité de ces règles qui contraste avec la complexité de son arbre de recherche dans les versions composées : le but est d'encourager l'analyse du jeu pour éviter une exploration aveugle très coûteuse.

Lights On Parallel : 4 jeux de *Lights On* sont joués en parallèle par un seul joueur qui choisit dans quel sous-jeu il souhaite agir à chaque tour. Le joueur obtient 100 points s'il gagne n'importe lequel des 4 sous-jeux.

Lights On Simultaneous : 4 jeux de *Lights On* sont joués en parallèle par un seul joueur qui choisit deux lampes sur lesquelles il souhaite agir à chaque tour (ce peut être deux fois la même) par le biais

88. Il existe une solution en 19 tours.

d'actions composées (16 lampes peuvent être allumées : 16^2 actions composées). Le joueur obtient 100 points s'il gagne n'importe lequel des 4 sous-jeux.

Lights On Simul 4 : Le jeu est identique au précédent, sauf que cette fois les actions composées permettent d'actionner 4 lampes (possiblement identiques) à la fois (16 lampes peuvent être allumées : 16^4 actions composées). Le joueur obtient 100 points s'il gagne n'importe lequel des 4 sous-jeux.

Factoring Easy Turtle Brain : 3 jeux de *Lights On* sont joués simultanément, mais le joueur ne peut agir que dans le jeu actif à un moment donné. Le jeu actif est désigné par un *stepper*. Le jeu actif change tous les 20 *steps*, le jeu global durant 60 *steps* en tout. Ce jeu peut donc être considéré comme un jeu en série. Le joueur doit allumer les 4 lampes d'un des sous-jeux pour gagner.

Factoring Medium Turtle Brain : Ce jeu est une variante du précédent avec 21 sous-jeux en série. Le joueur doit gagner un des jeux en série entre le $11^{\text{ème}}$ et le $20^{\text{ème}}$.

Factoring Impossible Turtle Brain : Ce jeu est une variante du précédent avec un ensemble de 7225 fluents inutiles ajoutés au jeu pour brouiller les pistes.

Factoring George Forman : Les 4 mêmes lampes (décrites par un ensemble de fluents) sont réutilisées pour jouer 4 parties de *Lights On* en série. La description de ce jeu a la particularité de ne pas respecter la contrainte de terminaison (def.1.1). La victoire dans un sous-jeu permet de démarrer le sous-jeu suivant et le jeu ne prend fin que si le joueur à réussi les 4 parties et obtenu 100 points. L'ajout d'un *stepper* permettrait de corriger la description du jeu.

Factoring Mutually Assured Destruction : 5 parties de *Lights On* sont jouées en série. Un *stepper* permet de s'assurer de la terminaison du jeu. Chaque partie de *Lights On* gagnée permet de passer à la suivante. Le joueur obtient 100 points s'il gagne les 5 parties en 20 coups, aucun point dans le cas contraire. Cependant, ce jeu est gravement mal formé et n'est pas gagnable (def.1.3). Le prédicat `lightson` utilisé pour comptabiliser les parties gagnées n'est pas un fluent mais un prédicat auxiliaire : le cumul des victoires n'est donc pas transmis d'un état à l'autre du jeu [89].

Simultaneous Win 2 : Ce jeu est composé de 4 jeux de *Lights On*. Le joueur dispose d'actions composées permettant d'actionner deux lampes quelconques à chaque tour. Le but est de gagner dans deux des sous-jeux simultanément.

A.19 Lights Out

Ce jeu solitaire est composé d'une grille de 5×5 lampes dont certaines sont allumées (fig.A.15). À chaque tour, le joueur peut choisir une lampe de la grille (allumée ou non) et changer son état. Cependant, les lampes adjacentes orthogonalement sont également affectées : une lampe allumée s'éteint ou inversement. Le but est d'éteindre toutes les lampes en 20 coups ou moins. Il existe une solution optimale en 9 coups.

Brain Teaser Extended : Ce jeu est constitué de deux jeux de *Lights Out* joués successivement sur un plateau de 3×3 et 4×4 cases. Le premier puzzle doit être résolu pour donner accès au second puzzle.

89. Le défaut présent dans cette description fait échouer notre programme de décomposition. Le *stepper* est correctement identifié, mais la victoire est considérée uniquement possible si toutes les parties de *Lights On* sont gagnées simultanément. Ces dernières ne sont donc pas séparées. Ce défaut générant une sous-décomposition, nous avons écarté ce jeu de nos bancs de tests.

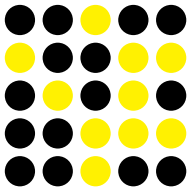


FIGURE A.15 – État initial du jeu *Lights Out* décris en GDL.

Le jeu prend fin bout de 14 tours. Le joueur reçoit 100 points s'il a résolu les deux puzzles en exactement 9 tours, ce qui correspond à la solution optimale, 80 points s'il a mis plus de temps pour résoudre les deux, 40 points s'il n'a résolu que le premier puzzle et aucun point en cas d'échec.

A.20 Max-Knights

Ce jeu solitaire consiste à placer sur un échiquier de 8×8 cases autant de cavaliers que possible sans qu'aucun d'eux ne menace les autres. Le score maximum est obtenu pour 32 cavaliers placés. Placer sur l'échiquier un cavalier qui met en échec un autre provoque immédiatement la fin du jeu. Le score optimal est obtenu en plaçant les cavaliers sur les cases noires, les cases blanches étant menacées (ou inversement).

A.21 Maze

Ce jeu solitaire est composé de 4 cases connectées par un chemin en sens unique (fig.A.16). Le joueur peut déplacer un robot vers la case suivante, tenter de ramasser ou déposer ce qu'il a en main. Le but est de se déplacer jusqu'à un trésor situé dans la troisième case, de le ramasser et de revenir le déposer dans la case de départ. Le joueur obtient 100 points s'il ramène le trésor en 9 tours, 0 sinon. La solution optimale nécessite 6 tours.

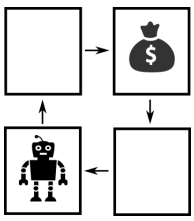


FIGURE A.16 – État initial pour le jeu *Maze*

A.22 Nim

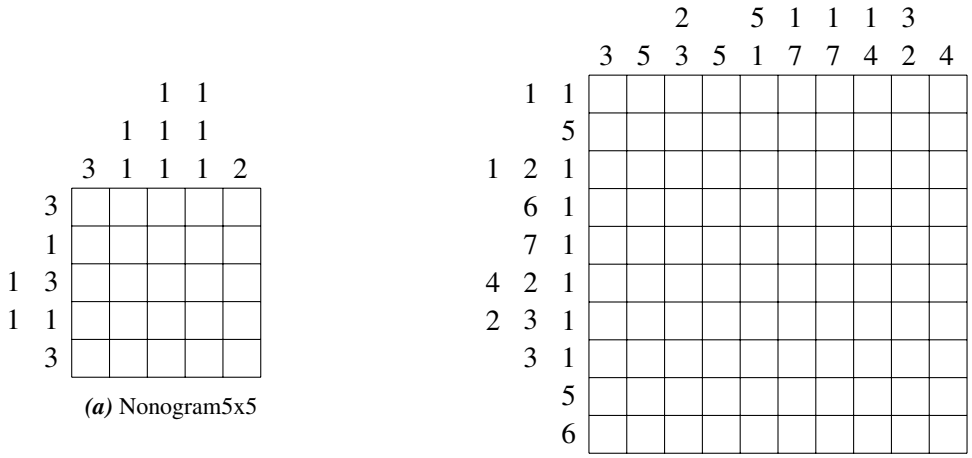
Il s'agit du classique jeu impartial de *Nim* pour lequel il existe une stratégie gagnante (théorème de Sprague-Grundy), mais en version misère. Le jeu est composé de 4 piles d'allumettes, de tailles variables selon les versions présentes dans les dépôts [90]. Chaque joueur prend tour à tour autant d'allumettes qu'il

90. Les quatre versions présentes dans les dépôts présentent des piles avec les tailles suivantes : (1,5,4,2), (2,2,10,10), (11,12,15,25) et (12,12,20,20).

souhaite dans l'une des piles. Celui qui saisit la dernière allumette a perdu.

A.23 Nonogramme

Ce jeu solitaire est constitué d'une grille avec des nombres indiqués à gauche de chaque ligne et en haut de chaque colonne (fig.A.17). Chaque nombre indique le nombre de cases adjacentes qui doivent être colorées dans la ligne ou la colonne correspondante. Si plusieurs nombres sont indiqués, plusieurs groupes de cases colorées doivent être présents et séparés par au moins une case vide. Le joueur doit colorer les cases de manière à respecter les contraintes imposées par ces nombres. Le joueur perd si la case qu'il vient de colorer viole une des contraintes i.e. si un groupe de cases adjacentes supérieur au plus grand nombre indiqué sur la ligne ou colonne correspondante est créé. Le joueur gagne si toutes les contraintes sont respectées. L'ensemble des cases colorées forme alors un motif (fig. 10.10 et 10.11 page 152). Le statut de certaines cases peut-être déterminé indépendamment des autres : elles constituent alors un sous-jeu.



(a) Nonogram5x5

(b) Nonogram10x10

FIGURE A.17 – Deux jeux de *Nonogramme* dont la description GDL est proposée dans les dépôts.

A.24 Platform Jumper

Ce jeu comporte deux phases. Durant la première phase, les deux joueurs préparent le plateau de jeu en colorant chacun leur tour toute une ligne ou toute une colonne avec leur couleur. Au départ, deux lignes sont déjà colorées (fig.A.18). Les cases colorées ne peuvent plus changer de couleur par la suite. Pendant la seconde phase du jeu, les pièces de chaque joueur ne peuvent se déplacer que sur les cases portant la même couleur que leur case de départ. La seconde phase du jeu démarre quand toutes les cases ont été colorées. Les pièces peuvent se déplacer comme des pions ou des cavaliers mais sans jamais reculer. Un joueur gagne quand il a amené une de ses pièces sur le bord opposé du plateau de jeu ou si l'autre joueur ne peut plus bouger.

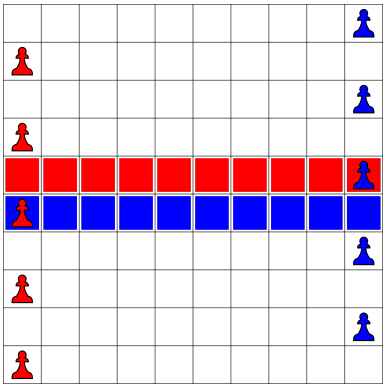


FIGURE A.18 – État initial pour le jeu *Platform Jumper*

A.25 Point Grab

Dans ce jeu à deux joueurs et à coups simultanés, chaque joueur a le choix entre 10 actions : deux actions A et B consistent à réclamer des points, les 8 autres sont inutiles. Si les deux joueurs choisissent la même action ils ne gagnent rien. Si un joueur est seul à choisir l'action A ou B, il gagne 5 points. Le jeu prend fin au bout de 30 tours maximum ou si un des joueurs atteint les 100 points. Ce jeu est un exemple type de *jeu répété* tel qu'étudié en théorie des jeux : les joueurs sont amenés à collaborer pour choisir chacun une des deux actions A et B.

A.26 Queens

Ce jeu solitaire consiste à placer N reines sur un échiquier de taille $N \times N$ sans qu'aucune ne menace les autres. Il existe plusieurs descriptions GDL de ce jeu pour différentes tailles N (8,12,16,31). La version avec $N = 8$ possède 92 solutions distinctes, 12 si on tient compte des symétries et rotations. Il existe deux versions différentes. Dans la première (*Queens* avec $N = 8$), le joueur place ses N reines et obtient un score qui est inversement proportionnel au nombre de reines menaçant les autres [91]. Dans la seconde, Le joueur obtient 100 points si toutes les reines sont correctement placées et aucun point sinon. Il existe deux variantes : *legal guided* (suffixe *lg*, par exemple *Queens12lg*) ou *unguided* (suffixe *ug* par exemple *Queens12ug*). Dans la version guidée, il est illégal de placer une reine qui en menace une autre. Une reine placée ne peut pas être modifiée : un mauvais coup entraîne donc la fin du jeu puisqu'il n'est plus possible de placer les reines restantes. Dans la version non guidée, toutes les reines sont placées avant d'évaluer le score, l'arbre du jeu a alors une complexité de l'ordre de 10^{71} pour un plateau de 8×8 cases.

91. Le joueur obtient 100 points si aucune reine n'en menace une autre, il perd 10 points pour chacune des 6 premières menaces et son score chute à 0 si le nombre de menaces est supérieur à 6.

A.27 Rainbow

Rainbow est un jeu solitaire de coloriage. Le joueur reçoit une carte et 4 couleurs. Le but du jeu est de colorer les régions de la carte de telle manière que deux régions adjacentes soient de couleurs différentes, y compris les régions qui se rejoignent à un angle (fig.A.19). La carte est entièrement colorée avant évaluation du score.

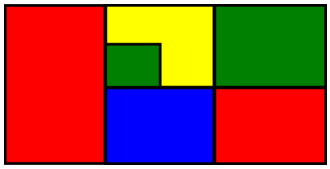


FIGURE A.19 – Position gagnante pour le jeu *Rainbow*

Dual Rainbow : Deux jeux de *Dual Rainbow* sont joués de manière synchrone par deux joueurs solitaires. Le score de chaque joueur est évalué comme dans le jeu global.

A.28 Roshambo 2

Il s'agit d'un jeu à deux joueurs composé de 10 manches de *Pierre Papier Ciseau Puits*. Les deux joueurs choisissent simultanément un de ces objets (potentiellement le même). Si les deux objets sont identiques, c'est un match nul, sinon un objet bat l'autre selon le schéma suivant : la *pierre* brise les *ciseaux*, le *papier* étouffe la *pierre*, le *papier* bouche le *puits*, les *ciseaux* coupent le *papier*, le *puits* engloutit les *ciseaux* ou la *pierre*. Contrairement au jeu *Pierre Papier Ciseau*, la victoire n'est pas totalement soumise au hasard puisque les objets *papier* et *puits* ont deux fois plus de chance de battre un autre objet que les objets *pierre* et *ciseaux* (fig.A.20). Le joueur qui gagne le plus de manches obtient 100 points, l'autre 0. En cas de match nul, les deux obtiennent 50 points.

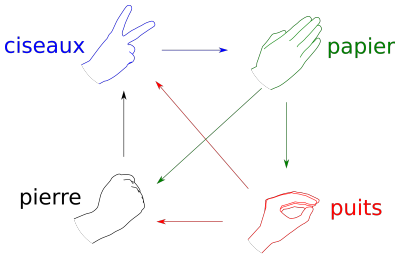


FIGURE A.20 – Graphe représentant les menaces entre les différents objets de *Roshambo 2*

A.29 Sheep And Wolf

Il s'agit d'un jeu asymétrique. Sur un damier de 8×8 cases, un joueur dirige un loup (W) et l'autre joueur 4 moutons (S). Au début du jeu les pions sont aux deux extrémités du damier (fig.A.21). Les moutons peuvent se déplacer d'une case à la fois, en diagonale et vers l'avant uniquement. Le loup peut se déplacer d'une case en diagonale dans toutes les directions. Pour un joueur comme pour l'autre, la

case de destination doit être vide pour pouvoir s'y déplacer. Le but des moutons est d'encercler le loup pour l'empêcher de bouger. Le jeu prend fin quand le loup ne peut plus bouger : les moutons gagnent alors 100 points et le loup 0. Si le loup dépasse les moutons et qu'ils n'ont donc plus aucun espoir de l'encercler, le jeu prend fin, le loup remporte 100 points et les moutons 0.

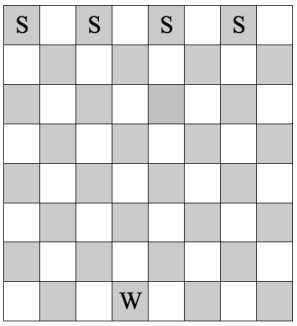


FIGURE A.21 – État initial du jeu *Sheep And Wolf*

A.30 Smallest

Quatre joueurs s'affrontent pendant 25 manches. À chaque manche, chaque joueur choisi un nombre entre 1 et 10. Un joueur gagne si son nombre est unique (aucun autre joueur n'a choisi le même) et s'il est plus petit que les autres nombres. Si le nombre le plus petit n'est pas unique, alors c'est le joueur ayant tiré le second plus petit nombre qui gagne si ce dernier est unique. Chaque manche gagnée rapporte 10 points. Ce jeu est un exemple de *jeu répété* tel que ceux étudiés en théorie des jeux.

A.31 Snake

Il s'agit d'un jeu solitaire se déroulant sur une grille de 9×7 cases entourée de murs. Le joueur dirige un serpent. Ce serpent grandit de 1 case un tour sur deux. Le serpent doit ramasser différents points placés sur la grille, éviter de percuter des obstacles présents ça et là sur la grille, éviter de croiser sa propre queue et atteindre la sortie en moins de 50 tours. Chaque point collecté permet d'augmenter le score final. Si la partie prend fin avant que le serpent n'ait atteint la sortie, le score est de 0.

Snake Parallel : Deux jeux de *Snake* sont joués par le même joueur en parallèle de manière asynchrone. Chaque grille a 4×4 cases. La grille est encerclée de murs. Dans cette version, il n'y a pas d'obstacle ni de point sur la grille. Le but du jeu est de remplir toute la grille avec le corps du serpent qui grandit à chaque mouvement. À chaque *step*, le serpent qui n'est pas déplacé par le joueur ne grandit pas. Le joueur remporte 100 points s'il remplit au moins une des grilles, aucun point sinon. Le jeu prend fin au bout de 20 tours ou si un des serpents heurte un mur ou sa propre queue.

Snake Assemblit : Un jeu de *Snake* et un jeu d'*Assemblit* sont joués en parallèle de manière synchrone par 2 joueurs (chaque sous-jeu est à coups alternés). Les joueurs obtiennent le score du premier sous-jeu qui s'achève. Le jeu de *Snake* consiste à survivre plus longtemps que l'adversaire sans percuter un mur, le corps de son serpent ou de celui de l'adversaire. Le joueur qui percute un obstacle met fin

à la partie. Il n'obtient aucun point et en donne 100 à son adversaire. Si les deux joueurs se percutent simultanément, ils obtiennent chacun 50 Points. Dans *Assemblit*, le joueur *colonne* et le joueur *ligne* disposent chacun de 8 marques. Ils démarrent respectivement aux coordonnées $(3,4)$ et $(4,3)$ d'un plateau de 6×6 cases. Quand vient son tour, un joueur peut déposer une marque là où il se trouve ou décaler toute une ligne d'un cran vers la droite ou vers la gauche, ou toute une colonne d'un cran vers le haut ou vers le bas. Les marques et les joueurs présents sur la ligne ou sur la colonne sont déplacés avec elle (un élément qui déborde du plateau de jeu, réapparaît de l'autre côté). Le joueur *colonne* (resp. *ligne*) gagne un point s'il se trouve sur la même ligne ou la même colonne qu'une de ses marques (sans être superposé à elle) et si cette marque est dans une des trois première *colonnes* (resp. *lignes*) du plateau. S'il existe un alignement sur les deux axes, le joueur obtient 2 points d'un coup. Les joueurs ne peuvent dépasser le score de 50 points dans *Assemblit*. Le jeu global prend fin au bout de 50 tours.

A.32 Sukoshi

Sukoshi est un jeu de *Sudoku* réduit à une grille de 3×3 cases. Les entiers de 1 à 3 doivent être placés sur la grille de manière à ce qu'aucune ligne ou colonne contienne plus d'une occurrence du même chiffre. Le jeu est terminé quand il n'est plus possible de placer aucun chiffre dans la grille. Le score est de 100 points si la grille est correctement complétée, 0 sinon.

Selective Sukoshi : Ce jeu regroupe 9 grilles de *Sukoshi*. Les sous-jeux sont parallèles asynchrones. Le but est de compléter une seule des neufs grilles. Le jeu s'achève lorsqu'il n'est plus possible de jouer dans aucune des grilles.

A.33 TicBlock

Ce jeu est constitué d'un jeu de *Tictactoe* suivi d'un jeu de *Blocker*. Dans la description du jeu, une même action (même instance du prédicat `does`) est utilisée dans les deux phases du jeu pour marquer une case (de mêmes coordonnées) du plateau de *Tictactoe* et du plateau de *Blocker*. Le score obtenu est la moyenne des scores des deux sous-jeux.

A.34 Tictactoe

Il s'agit du jeu également connu en France sous le nom de *Morpion* qui consiste à aligner trois croix ou trois cercles sur une grille de 3×3 cases (fig.A.22). Par convention X est considéré comme étant le joueur 1 et commence. Les joueurs jouent chacun à leur tour et placent leur symbole dans une des cases encore vide. Le premier joueur qui aligne trois symboles dans une ligne, une colonne ou une diagonale remporte 100 points et l'autre 0. En cas de match nul chacun remporte 50 points. Si les deux joueurs jouent parfaitement, l'issue du match est toujours nulle [92] ce qui limite l'intérêt du jeu. Cependant, la petite taille de l'arbre du jeu le rend pratique comme terrain de tests [93]. La complexité de l'arbre du jeu

92. Ce fait constitue un élément majeur de l'intrigue du film *WarGames*.
93. La description GDL de ce jeu utilise le *(or ...)* pourtant interdit dans les versions récentes des spécifications GDL

est de l'ordre de 10^5 [Bonnet et Saffidine, 2013].

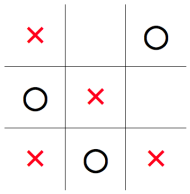


FIGURE A.22 – **Exemple d'état final pour le jeu *Tictactoe***

Double Tictactoe : Deux parties de *Tictactoe* sont jouées en parallèle de manière synchrone. Chaque joueur commence dans un sous-jeu et réplique dans l'autre sous-jeu au tour suivant. Le jeu se termine quand les deux parties de *Tictactoe* sont achevées : si une partie se termine avant l'autre, les joueurs ne peuvent plus y jouer que des actions *noop* quand vient leur tour. Le score est la moyenne des scores obtenus dans les sous-jeux.

Double Tictactoe Dengji : Deux parties de *Tictactoe* sont jouées en parallèle de manière asynchrone. Le joueur actif choisi dans quel sous-jeu il souhaite ajouter une marque pendant que l'autre exécute une action *noop*. Le jeu se termine quand les deux parties de *Tictactoe* sont achevées. Le score est la moyenne des scores obtenus dans les différents sous-jeux.

Tictactoe Serial : Deux parties de *Tictactoe* sont jouées de manière séquentielle. Le score est la moyenne des scores obtenus dans les différents sous-jeux.

Multiple Tictactoe : Neuf parties de *Tictactoe* sont jouées en parallèle de manière asynchrone. Le joueur actif choisit dans quel sous-jeu il souhaite ajouter une marque pendant que l'autre exécute une action *noop*. Seul le sous-jeu numéro 5 influence le score, les autres sont inutiles. Le jeu prend fin au bout de 19 tours maximum ou si le sous-jeu numéro 5 se termine.

Nine Board Tictactoe : Neuf jeux de *Tictactoe* disposés selon une grille de 3×3 sont disputés en parallèle de manière asynchrone. Le premier joueur joue son premier coup dans la grille de son choix (pendant que l'autre joue *noop*). La position de la case jouée détermine la position du sous-jeu sans lequel l'autre joueur jouera au prochain tour. Par exemple si le joueur X marque la case $(1, 3)$ du sous-jeu central, le joueur O devra jouer le coup suivant dans le sous-jeu $(1, 3)$. Si ce sous-jeu ne comporte aucune case libre, alors le joueur est libre de choisir le sous-jeu dans lequel il souhaite agir. Le jeu prend fin quand il n'est plus possible de jouer dans aucun sous-jeu ou quand un des joueurs a aligné trois marques dans l'un des sous-jeux. Le lien fort existant entre la position d'une marque et le sous-jeu où le joueur suivant doit jouer rend ce jeu non-décomposable.

TicTacHeaven : Ce jeu rassemble un jeu de *Nine Board Tictactoe* avec un jeu de *Tictactoe* simple. Les jeux sont parallèles synchrones. Un joueur peut passer son tour. Le premier sous-jeu qui se termine détermine le score.

Joined Tictactoe : Ce jeu rassemble deux jeux de *Tictactoe* avec une case supplémentaire permettant de faire des alignements supplémentaires (horizontaux uniquement) de trois marques. Ce jeu devient décomposable en cours de partie quand il n'est plus possible d'utiliser la case centrale pour réussir un alignement (fig.8.9 page 77).

A.35 Tron

Il s'agit d'un jeu à 2 joueurs inspiré du film *Tron* (1982). Les joueurs jouent simultanément et avancent d'une case à chaque tour. Ils laissent derrière eux une traînée de lumière. Le jeu prend fin si un joueur heurte les murs qui entourent la grille de 10×10 cases du jeu, une traînée de lumière ou si les deux joueurs se percutent. Un joueur mort ne remporte aucun point ; le survivant 100 points. Si les deux joueurs perdent simultanément ils remportent chacun 50 points.

Annexe B
Décompositions attendues

Le tableau suivant présente le résultat attendu pour la décomposition des jeux provenant du dépôt `http://games.ggp.org/`.

La colonne **NAME** indique si le jeu provient du dépôt *Base*, *Dresden GGPServer* ou *Stanford Gamemaster*, le nom du jeu et la version de la description.

Les autres colonnes présentent les instructions suivantes :

#AD le nombre de sous-jeux dépendants des actions

#UAD le nombre de *sous-jeux inutiles* dépendants des actions

#AI le nombre de sous-jeux indépendants des actions

#UAI le nombre de *sous-jeux inutiles* indépendants des actions

La liste contient 604 jeux, 7 jeux contiennent des fluents/actions partagés entre plusieurs sous-jeux ou sont gravement mal formés. Les résultats obtenus pour ces jeux n'ont pas été pris en compte dans nos résultats publiés. Dans certains jeux, les fluents *control* constituent un sous-jeu indépendant des actions détecté comme inutile.

#AD	#UAD	#AI	#UAI	NAME
1	0	0	0	/base/ad_game_2x2_v0.kif
1	0	0	0	/base/beatMania_v0.kif
1	0	0	0	/base/biddingTicTacToe_10coins_v0.kif
1	0	0	0	/base/biddingTicTacToe_v0.kif
1	0	0	0	/base/blocker_v0.kif
1	0	0	0	/base/chickentictactoe_v0.kif
1	0	0	0	/base/chickentoetictac_v0.kif
1	0	0	0	/base/circlesolitaire_v0.kif
1	0	0	0	/base/colonelBlottoVariant2_v0.kif
1	0	0	0	/base/copolymer_4_pie_v0.kif
1	0	0	0	/base/copolymer_4_v0.kif
1	0	0	0	/base/cubicup_3player_v0.kif
1	0	0	0	/base/dotsAndBoxes_v0.kif
1	0	0	0	/base/dotsAndBoxesSuicide_v0.kif
1	0	0	0	/base/duplicateStateLarge_v0.kif
1	0	0	0	/base/duplicateStateMedium_v0.kif
1	0	0	0	/base/duplicateStateSmall_v0.kif
1	0	0	0	/base/haystack_v0.kif
1	0	0	0	/base/kalaha5x2x3_v0.kif
1	0	0	0	/base/kalaha6x2x4_v0.kif
1	0	0	0	/base/knightwar_v0.kif
1	0	0	0	/base/mineClearingSmall_v0.kif
1	0	0	0	/base/onestep_v0.kif
1	0	0	0	/base/pearls_v0.kif
1	0	0	0	/base/peg_v0.kif
1	0	0	0	/base/pegEuro_v0.kif
1	0	0	0	/base/pegEuro_v1.kif
1	0	0	0	/base/quad_5x5_v0.kif
1	0	0	0	/base/snake2p_v0.kif
1	0	0	0	/base/stateSpaceLarge_v0.kif
1	0	0	0	/base/stateSpaceMedium_v0.kif
1	0	0	0	/base/stateSpaceSmall_v0.kif
1	0	0	0	/base/sudokuGrade1_v0.kif
1	0	0	0	/base/sudokuGrade2_v0.kif
1	0	0	0	/base/sudokuGrade3_v0.kif
1	0	0	0	/base/sudokuGrade4_v0.kif
1	0	0	0	/base/sudokuGrade5_v0.kif
1	0	0	0	/base/sudokuGrade6E_v0.kif
1	0	0	0	/base/sudokuGrade6H_v0.kif
1	0	0	0	/base/survival_v0.kif
1	0	0	0	/base/ticblock_v0.kif
1	0	0	0	/base/tictactoe_3d_6player_v0.kif
1	0	0	0	/base/tictactoe_3d_small_6player_v0.kif
1	0	0	0	/base/tpeg_v0.kif
1	0	0	0	/base/wallmaze_v0.kif
1	0	0	0	/dresden/2player_normal_form_2010_v0.kif
1	0	0	0	/dresden/ad_game_2x2_v0.kif
1	0	0	0	/dresden/beatmania_v0.kif
1	0	0	0	/dresden/bidding-tictactoe_10coins_v0.kif
1	0	0	0	/dresden/bidding-tictactoe_v0.kif
1	0	0	0	/dresden/blocker_v0.kif
1	0	0	0	/dresden/catch_me_v0.kif
1	0	0	0	/dresden/Catch-Me-If-You-Can_v0.kif
1	0	0	0	/dresden/CatchMeIfYouCanTest_v0.kif
1	0	0	0	/dresden/chickentictactoe_v0.kif
1	0	0	0	/dresden/chickentoetictac_v0.kif
1	0	0	0	/dresden/circlesolitaire_v0.kif
1	0	0	0	/dresden/connect5_v0.kif
1	0	0	0	/dresden/cubicup_3player_v0.kif
1	0	0	0	/dresden/duplicatestatelarge_v0.kif
1	0	0	0	/dresden/duplicatestatemedium_v0.kif
1	0	0	0	/dresden/duplicatestatesmall_v0.kif
1	0	0	0	/dresden/firesheep_v0.kif
1	0	0	0	/dresden/fizzbuzz_v0.kif
1	0	0	0	/dresden/grid_game_v0.kif
1	0	0	0	/dresden/knightwar_v0.kif
1	0	0	0	/dresden/Nine_Mens_Morris_0.11_2p_v0.kif
1	0	0	0	/dresden/peg_bugfixed_v0.kif

#AD	#UAD	#AI	#UAI	NAME
1	0	0	0	/dresden/peg_v0.kif
1	0	0	0	/dresden/Pilgrimage_v0.kif
1	0	0	0	/dresden/quad_5x5_v0.kif
1	0	0	0	/dresden/RobinsonRoulette_no_white_peg_v0.kif
1	0	0	0	/dresden/statespacelarge_v0.kif
1	0	0	0	/dresden/statespacemedium_v0.kif
1	0	0	0	/dresden/statespacesmall_v0.kif
1	0	0	0	/dresden/sudoku_simple_v0.kif
1	0	0	0	/dresden/Thief_Police_v0.kif
1	0	0	0	/dresden/tictactoe_3d_6player_v0.kif
1	0	0	0	/dresden/tictactoe_3d_small_6player_v0.kif
1	0	0	0	/dresden/tpeg_v0.kif
1	0	0	0	/dresden/wallmaze_v0.kif
1	0	0	0	/stanford/firesheep_v0.kif
1	0	0	0	/stanford/pilgrimage_v0.kif
1	0	0	0	/stanford/rainbow_v0.kif
1	0	0	0	/stanford/sudoku_v0.kif
1	0	0	0	/stanford/sukoshi_v0.kif
1	0	0	0	/stanford/untwistycomplex_v0.kif
1	0	1	0	/base/4pffa_v1.kif
1	0	1	0	/base/4pttc_v2.kif
1	0	1	0	/base/aipsrovers01_v0.kif
1	0	1	0	/base/asteroids_v0.kif
1	0	1	0	/base/atariGo_7x7_v0.kif
1	0	1	0	/base/atariGoVariant_7x7_v0.kif
1	0	1	0	/base/battle_v0.kif
1	0	1	0	/base/battlebrushes_v0.kif
1	0	1	0	/base/blobwars_v0.kif
1	0	1	0	/base/blocks_v0.kif
1	0	1	0	/base/blocks2player_v0.kif
1	0	1	0	/base/blocksWorld_v0.kif
1	0	1	0	/base/blokbox_simple_v0.kif
1	0	1	0	/base/bomberman2p_v0.kif
1	0	1	0	/base/breakthroughSmallHoles_v1.kif
1	0	1	0	/base/buttons_v0.kif
1	0	1	0	/base/catcha_mouse_v0.kif
1	0	1	0	/base/chineseCheckers1_v0.kif
1	0	1	0	/base/chineseCheckers1_v1.kif
1	0	1	0	/base/chineseCheckers4_v0.kif
1	0	1	0	/base/chineseCheckers4_v1.kif
1	0	1	0	/base/chineseCheckers6_v0.kif
1	0	1	0	/base/chineseCheckers6_v1.kif
1	0	1	0	/base/chineseCheckers6_v2.kif
1	0	1	0	/base/coins_atomic_v0.kif
1	0	1	0	/base/coins_v0.kif
1	0	1	0	/base/colonelBlotto_v0.kif
1	0	1	0	/base/colonelBlottoVariant_v0.kif
1	0	1	0	/base/crissrace_v0.kif
1	0	1	0	/base/crossers3_v0.kif
1	0	1	0	/dresden/racetrackcorridor_v0.kif
1	0	1	0	/base/eightPuzzle_v0.kif
1	0	1	0	/base/eotcitcit_v0.kif
1	0	1	0	/base/four_way_battle_v0.kif
1	0	1	0	/base/god_v0.kif
1	0	1	0	/base/gt_chicken_v0.kif
1	0	1	0	/base/gt_coordination_v0.kif
1	0	1	0	/base/gt_prisoner_v0.kif
1	0	1	0	/base/gt_staghunt_v0.kif
1	0	1	0	/base/gt_tinfoil_v0.kif
1	0	1	0	/base/gt_ultimatum_v0.kif
1	0	1	0	/base/hanoi_v0.kif
1	0	1	0	/base/hanoi7_bugfix_v0.kif
1	0	1	0	/base/hitori_v0.kif
1	0	1	0	/base/kalaha_2009_v0.kif
1	0	1	0	/base/kitten_escapes_from_fire_v0.kif
1	0	2	0	/base/knightmove_v0.kif
1	0	1	0	/base/knightsTour_v0.kif
1	0	1	0	/base/knightsTourLarge_v0.kif
1	0	1	0	/base/lightsOn_v0.kif

#AD	#UAD	#AI	#UAI	NAME
1	0	1	0	/base/lightsOut_v0.kif
1	0	1	0	/base/max_knights_v0.kif
1	0	1	0	/base/maze_v0.kif
1	0	1	0	/base/merrills_v0.kif
1	0	1	0	/base/numbertictactoe_v0.kif
1	0	1	0	/base/pancakes_v0.kif
1	0	1	0	/base/pancakes6_v0.kif
1	0	1	0	/base/pancakes88_v0.kif
1	0	1	0	/base/pawnToQueen_v0.kif
1	0	1	0	/base/point_grab_v0.kif
1	0	1	0	/base/point_grab_v1.kif
1	0	1	0	/base/quarto_v0.kif
1	0	1	0	/base/quartoSuicide_v0.kif
1	0	1	0	/base/queens_v0.kif
1	0	1	0	/base/queens06ug_v0.kif
1	0	1	0	/base/queens08ug_v0.kif
1	0	1	0	/base/queens12ug_v0.kif
1	0	1	0	/base/queens16ug_v0.kif
1	0	1	0	/base/qyshinsu_v0.kif
1	0	1	0	/base/qyshinsu_v1.kif
1	0	1	0	/base/racetrackcorridor_v0.kif
1	0	1	0	/base/roshambo2_v0.kif
1	0	1	0	/base/rubiksCube_v0.kif
1	0	1	0	/base/rubiksCubeSuperflip_v0.kif
1	0	1	0	/base/ruleDepthLinear_v0.kif
1	0	1	0	/base/slidingpieces_v0.kif
1	0	1	0	/base/smallest_4player_v0.kif
1	0	1	0	/base/snake_2008_tweaked_v0.kif
1	0	1	0	/base/snake_2008_v0.kif
1	0	1	0	/base/snake_2009_big_v0.kif
1	0	1	0	/base/snake_2009_v0.kif
1	0	1	0	/base/ticTicToe_v0.kif
1	0	1	0	/base/twisty-passages_v0.kif
1	0	1	0	/base/untwistycomplex2_v0.kif
1	0	1	0	/base/wargame01_v0.kif
1	0	1	0	/base/wargame01_v1.kif
1	0	1	0	/base/wargame02_v0.kif
1	0	1	0	/base/wargame02_v1.kif
1	0	1	0	/base/wargame03_v0.kif
1	0	1	0	/base/wargame03_v1.kif
1	0	1	0	/base/wargame03_v2.kif
1	0	1	0	/base/zombieAttack1PL6_v0.kif
1	0	1	0	/dresden/4pttc_v0.kif
1	0	1	0	/dresden/8puzzle_v0.kif
1	0	1	0	/dresden/aipsrovers01_v0.kif
1	0	1	0	/dresden/asteroids_v0.kif
1	0	1	0	/dresden/babel_v0.kif
1	0	1	0	/dresden/battle_v0.kif
1	0	1	0	/dresden/blobwars_v0.kif
1	0	1	0	/dresden/blocks_v0.kif
1	0	1	0	/dresden/blocks2player_v0.kif
1	0	1	0	/dresden/bomberman2p_v0.kif
1	0	1	0	/dresden/buttons_v0.kif
1	0	1	0	/dresden/catcha_mouse_v0.kif
1	0	1	0	/dresden/Checkers-BreakThrough_v0.kif
1	0	1	0	/dresden/chinesecheckers1_v0.kif
1	0	1	0	/dresden/chinesecheckers4_v0.kif
1	0	1	0	/dresden/chinesecheckers6_v0.kif
1	0	1	0	/dresden/chinesecheckers6-simultaneous_v0.kif
1	0	1	0	/dresden/clobber_v0.kif
1	0	1	0	/dresden/coins_v0.kif
1	0	1	0	/dresden/crissrace_v0.kif
1	0	1	0	/dresden/crossers3_v0.kif
1	0	1	0	/dresden/eotcitcit_v0.kif
1	0	1	0	/dresden/farmingquandries_v0.kif
1	0	1	0	/dresden/fighter_v0.kif
1	0	1	0	/dresden/four_way_battle_v0.kif
1	0	1	0	/dresden/ggp-course2013_a1_v0.kif
1	0	1	0	/dresden/ggp-course2013_jordi_v0.kif

#AD	#UAD	#AI	#UAI	NAME
1	0	1	0	/dresden/ggp-course2013_michal13_v0.kif
1	0	1	0	/dresden/god_v0.kif
1	0	1	0	/dresden/Goldrush_v0.kif
1	0	1	0	/dresden/gt_chicken_v0.kif
1	0	1	0	/dresden/gt_prisoner_v0.kif
1	0	1	0	/dresden/gt_ultimatum_v0.kif
1	0	1	0	/dresden/hanoi_v0.kif
1	0	1	0	/dresden/hanoi7_bugfix_v0.kif
1	0	1	0	/dresden/hanoi7_v0.kif
1	0	1	0	/dresden/hitori_v0.kif
1	0	1	0	/dresden/kitten_escapes_from_fire_v0.kif
1	0	1	0	/dresden/knightstour_v0.kif
1	0	1	0	/dresden/lightsout_v0.kif
1	0	1	0	/dresden/lightsout2_v0.kif
1	0	1	0	/dresden/max_knights_v0.kif
1	0	1	0	/dresden/maze_v0.kif
1	0	1	0	/dresden/merrills_v0.kif
1	0	1	0	/dresden/mimikry_v0.kif
1	0	1	0	/dresden/numbertictactoe_v0.kif
1	0	1	0	/dresden/oisters_farm_v0.kif
1	0	1	0	/dresden/pancakes_v0.kif
1	0	1	0	/dresden/pancakes6_v0.kif
1	0	1	0	/dresden/pancakes88_v0.kif
1	0	1	0	/dresden/pawn_whopping_simultaneous_v0.kif
1	0	1	0	/dresden/pawntoqueen_v0.kif
1	0	1	0	/dresden/point_grab_v0.kif
1	0	1	0	/dresden/pointgrab_state_v0.kif
1	0	1	0	/dresden/quarto_v0.kif
1	0	1	0	/dresden/quartosuicide_v0.kif
1	0	1	0	/dresden/queens_v0.kif
1	0	1	0	/dresden/Qyshinsu_v0.kif
1	0	1	0	/dresden/rendezvous_asteroids_v0.kif
1	0	1	0	/dresden/roshambo2_v0.kif
1	0	1	0	/dresden/ruledepthlinear_v0.kif
1	0	1	0	/dresden/Runners_v0.kif
1	0	1	0	/dresden/SC_TestOnly_enabled_v0.kif
1	0	1	0	/dresden/slidingpieces_v0.kif
1	0	1	0	/dresden/smallest_4player_v0.kif
1	0	1	0	/dresden/snake_2008_v0.kif
1	0	1	0	/dresden/snake_2009_big_v0.kif
1	0	1	0	/dresden/snake_2009_v0.kif
1	0	1	0	/dresden/tictictoe_v0.kif
1	0	1	0	/dresden/towerworld_v0.kif
1	0	1	0	/dresden/Travelers-Dilemma_v0.kif
1	0	1	0	/dresden/twisty-passages_v0.kif
1	0	1	0	/dresden/walkingman_v0.kif
1	0	1	0	/dresden/wargame01_v0.kif
1	0	1	0	/dresden/wargame02_v0.kif
1	0	1	0	/stanford/3puzzle_v0.kif
1	0	1	0	/stanford/8puzzle_v0.kif
1	0	1	0	/stanford/buttonsandlights_v0.kif
1	0	1	0	/stanford/chinesecheckers_v0.kif
1	0	1	0	/stanford/chinesecheckers4_v0.kif
1	0	1	0	/stanford/duidoku_v0.kif
1	0	1	0	/stanford/duikoshi_v0.kif
1	0	1	0	/stanford/eightpuzzle_v0.kif
1	0	1	0	/stanford/farmingquandries_v0.kif
1	0	1	0	/stanford/hamilton_v0.kif
1	0	1	0	/stanford/hunter_v0.kif
1	0	1	0	/stanford/knightstour_v0.kif
1	0	1	0	/stanford/knightstourbig_v0.kif
1	0	1	0	/stanford/knightstourmedium_v0.kif
1	0	1	0	/stanford/threepuzzle_v0.kif
1	0	1	0	/stanford/tictictoe_v0.kif
1	0	1	0	/stanford/untwistycorridor_v0.kif
1	0	1	0	/stanford/vectorracer4_v0.kif
1	0	1	0	/stanford/vectorracer6_v0.kif
1	0	1	0	/stanford/vectorracerbig_v0.kif
1	0	1	1	/base/3pConnectFour_v0.kif

#AD	#UAD	#AI	#UAI	NAME
1	0	1	1	/base/breakthrough_v0.kif
1	0	1	1	/base/breakthroughHoles_v0.kif
1	0	1	1	/base/breakthroughSmall_v0.kif
1	0	1	1	/base/breakthroughSmallHoles_v0.kif
1	0	1	1	/base/breakthroughSuicide_v0.kif
1	0	1	1	/base/breakthroughSuicideSmall_v0.kif
1	0	1	1	/base/cephalopodMicro_v0.kif
1	0	1	1	/base/chomp_v0.kif
1	0	1	1	/base/cittaceot_v0.kif
1	0	1	1	/base/conn4_v0.kif
1	0	1	1	/base/connect4_v0.kif
1	0	1	1	/base/connect5_v0.kif
1	0	1	1	/base/connect5_v1.kif
1	0	1	1	/base/connectFour_9x6_v0.kif
1	0	1	1	/base/connectFour_v0.kif
1	0	1	1	/base/connectFourLarge_v0.kif
1	0	1	1	/base/connectFourLarger_v0.kif
1	0	1	1	/base/connectFourSuicide_7x7_v0.kif
1	0	1	1	/base/connectFourSuicide_v0.kif
1	0	1	1	/base/eotcatcit_v0.kif
1	0	1	1	/base/golden_rectangle_v0.kif
1	0	1	1	/base/golden_rectangle_v1.kif
1	0	1	1	/base/gomoku_11x11_v0.kif
1	0	1	1	/base/gomoku_15x15_v0.kif
1	0	1	1	/base/gomoku_swap2_11x11_v0.kif
1	0	1	1	/base/gomoku_swap2_15x15_v0.kif
1	0	1	1	/base/gt_dollar_v0.kif
1	0	1	1	/base/knightazons_v0.kif
1	0	1	1	/base/knightfight_v0.kif
1	0	1	1	/base/knightThrough_v0.kif
1	0	1	1	/base/madBishops_v0.kif
1	0	1	1	/base/majorities_v0.kif
1	0	1	1	/base/nineBoardTicTacToe_v0.kif
1	0	1	1	/base/nineBoardTicTacToePie_v0.kif
1	0	1	1	/base/nineBoardTicTacToePie_v1.kif
1	0	1	1	/base/othello-comp2007_v0.kif
1	0	1	1	/base/othelloHoles_v0.kif
1	0	1	1	/base/othelloSuicide_v0.kif
1	0	1	1	/base/pawnWhopping_v0.kif
1	0	1	1	/base/pawnWhopping_v1.kif
1	0	1	1	/base/pentago_v0.kif
1	0	1	1	/base/pentago_v1.kif
1	0	1	1	/base/pentagoSuicide_v0.kif
1	0	1	1	/base/pentagoSuicide_v1.kif
1	0	1	1	/base/quad_7x7_v0.kif
1	0	1	1	/base/sheepAndWolf_v0.kif
1	0	1	1	/base/shmup_v0.kif
1	0	1	1	/base/sum15_v0.kif
1	0	1	1	/base/tictactoe_3d_2player_v0.kif
1	0	1	1	/base/tictactoe_3d_small_2player_v0.kif
1	0	1	1	/base/tictactoe_3player_v0.kif
1	0	1	1	/base/tictactoe_orthogonal_v0.kif
1	0	1	1	/base/ticTacToe_v0.kif
1	0	3	1	/base/tictactoe-init1_v0.kif
1	0	3	1	/base/tictactoe-init1_v1.kif
1	0	1	1	/base/tictactoe2_v0.kif
1	0	1	1	/base/ticTacToeLarge_v0.kif
1	0	1	1	/base/ticTacToeLargeSuicide_v0.kif
1	0	1	1	/base/ticTacToeNoVars_v0.kif
1	0	1	1	/base/tictactoex9_v0.kif
1	0	1	1	/base/toetictac_v0.kif
1	0	1	1	/base/withConviction_v0.kif
1	0	1	1	/dresden/3conn3_v0.kif
1	0	1	1	/dresden/breakthrough_v0.kif
1	0	1	1	/dresden/breakthroughsuicide_v0.kif
1	0	1	1	/dresden/breakthroughsuicide_v2_v0.kif
1	0	1	1	/dresden/CephalopodMicro_v0.kif
1	0	1	1	/dresden/chomp_v0.kif
1	0	1	1	/dresden/conn4_v0.kif

#AD	#UAD	#AI	#UAI	NAME
1	0	1	1	/dresden/connect4_v0.kif
1	0	1	1	/dresden/connectfour_v0.kif
1	0	1	1	/dresden/connectfoursuicide_v0.kif
1	0	1	1	/dresden/coopconn4_v0.kif
1	0	1	1	/dresden/eotcatcit_v0.kif
1	0	1	1	/dresden/golden_rectangle_v0.kif
1	0	1	1	/dresden/gt_attrition_v0.kif
1	0	1	1	/dresden/knightazons_v0.kif
1	0	1	1	/dresden/knightfight_v0.kif
1	0	1	1	/dresden/knightthrough_v0.kif
1	0	1	1	/dresden/othello-comp2007_v0.kif
1	0	1	1	/dresden/othellosuicide_v0.kif
1	0	1	1	/dresden/pawn_whopping_corrected_v0.kif
1	0	1	1	/dresden/pawn_whopping_v0.kif
1	0	1	1	/dresden/pentago_2008_v0.kif
1	0	1	1	/dresden/quad_5x5_8_2_v0.kif
1	0	1	1	/dresden/quad_7x7_v0.kif
1	0	1	1	/dresden/sheep_and_wolf_v0.kif
1	0	1	1	/dresden/sum15_v0.kif
1	0	1	1	/dresden/tictactoe_3d_2player_v0.kif
1	0	1	1	/dresden/tictactoe_3d_small_2player_v0.kif
1	0	1	1	/dresden/tictactoe_3player_v0.kif
1	0	1	1	/dresden/tictactoe_orthogonal_v0.kif
1	0	1	1	/dresden/tictactoe_v0.kif
1	0	3	1	/dresden/tictactoe-init1_v0.kif
1	0	1	1	/dresden/tictactoelarge_v0.kif
1	0	1	1	/dresden/tictactoelargesuicide_v0.kif
1	0	1	1	/dresden/tictactoex9_v0.kif
1	0	1	1	/dresden/toetictac_v0.kif
1	0	1	1	/dresden/tritactoe_v0.kif
1	0	1	1	/stanford/breakthrough_v0.kif
1	0	1	1	/stanford/breakthroughsmall_v0.kif
1	0	1	1	/stanford/connectfour_v0.kif
1	0	1	1	/stanford/nineboardtictactoe_v0.kif
1	0	1	1	/stanford/pentago_v0.kif
1	0	1	1	/stanford/tictactoe_v0.kif
1	0	1	1	/stanford/tictactoe3_v0.kif
1	0	1	1	/stanford/tictactoe5_v0.kif
1	0	1	1	/stanford/tictactoe7_v0.kif
1	0	1	1	/stanford/trifecta_v0.kif
1	0	2	0	/base/amazons_10x10_v0.kif
1	0	2	0	/base/amazons_8x8_v0.kif
1	0	2	0	/base/amazonsSuicide_10x10_v0.kif
1	0	2	0	/base/amazonsTorus_10x10_v0.kif
1	0	2	0	/base/checkers_v0.kif
1	0	2	0	/base/checkers_v1.kif
1	0	2	0	/base/checkers-mustjump_v0.kif
1	0	2	0	/base/checkers-mustjump_v1.kif
1	0	2	0	/base/checkers-newgoals_v0.kif
1	0	2	0	/base/checkers-newgoals_v1.kif
1	0	2	0	/base/checkersBarrelNoKings_v1.kif
1	0	2	0	/base/checkersSmall_v0.kif
1	0	2	0	/base/checkersSmall_v1.kif
1	0	2	0	/base/checkersTiny_v0.kif
1	0	2	0	/base/checkersTiny_v1.kif
1	0	2	0	/base/checkersTorus_v1.kif
1	0	2	0	/base/checkersTorusNoKings_v0.kif
1	0	2	0	/base/speedChess_v0.kif
1	0	2	0	/base/zhadu_v0.kif
1	0	2	0	/dresden/checkers_v0.kif
1	0	2	0	/dresden/checkers-mustjump_v0.kif
1	0	2	0	/dresden/checkers-newgoals_v0.kif
1	0	2	0	/dresden/dobutsushogi_v0.kif
1	0	2	0	/dresden/knightmove_v0.kif
1	0	2	0	/dresden/skirmish2_v0.kif
1	0	2	0	/dresden/Zhadu_v0.kif
1	0	2	0	/stanford/battleofnumbers_v0.kif
1	0	2	0	/stanford/checkersonabarrelnokings_v0.kif
1	0	2	0	/stanford/horseshoe_v0.kif

#AD	#UAD	#AI	#UAI	NAME
1	0	2	1	/dresden/ghostmaze2p_v0.kif
1	0	2	1	/base/2pffa_v0.kif
1	0	2	1	/base/2pffa_v1.kif
1	0	2	1	/base/2pffa_zerosum_v0.kif
1	0	2	1	/base/2pttc_v0.kif
1	0	2	1	/base/2pttc_v1.kif
1	0	2	1	/base/3pffa_v0.kif
1	0	2	1	/base/3pffa_v1.kif
1	0	2	1	/base/3pttc_v0.kif
1	0	2	1	/base/3pttc_v1.kif
1	0	2	1	/base/4pffa_v0.kif
1	0	2	1	/base/4pttc_v0.kif
1	0	2	1	/base/4pttc_v1.kif
1	0	2	1	/base/brawl_v0.kif
1	0	2	1	/base/checkers-cylinder-mustjump_v0.kif
1	0	2	1	/base/checkers-cylinder-mustjump_v1.kif
1	0	2	1	/base/checkers-mustjump-torus_v0.kif
1	0	2	1	/base/checkers-mustjump-torus_v1.kif
1	0	2	1	/base/checkers-suicide-cylinder-mustjump_v0.kif
1	0	2	1	/base/checkers-suicide-cylinder-mustjump_v1.kif
1	0	2	1	/base/checkersBarrelNoKings_v0.kif
1	0	2	1	/base/checkersTorus_v0.kif
1	0	2	1	/base/chineseCheckers2_v0.kif
1	0	2	1	/base/chineseCheckers2_v1.kif
1	0	2	1	/base/chineseCheckers3_v0.kif
1	0	2	1	/base/chineseCheckers3_v1.kif
1	0	2	1	/base/choicethroughalt_v0.kif
1	0	2	1	/base/coloredtrails_v0.kif
1	0	2	1	/base/crisscross_v0.kif
1	0	2	1	/base/cylinder-checkers_v0.kif
1	0	2	1	/base/endgame_v0.kif
1	0	2	1	/base/escortLatch_v0.kif
1	0	2	1	/base/ghostMaze2p_v0.kif
1	0	2	1	/base/guess_v0.kif
1	0	2	1	/base/hallway_v0.kif
1	0	2	1	/base/hallway_v1.kif
1	0	2	1	/base/hex_v0.kif
1	0	2	1	/base/hexPie_v0.kif
1	0	2	1	/base/minichess_v0.kif
1	0	2	1	/base/minichess-evilconjuncts_v0.kif
1	0	2	1	/base/mummyMaze2p_2007_v0.kif
1	0	2	1	/base/mummymaze2p_v0.kif
1	0	2	1	/base/pacman2p_v0.kif
1	0	2	1	/base/pacman3p_v0.kif
1	0	2	1	/base/shogi_ctk_v0.kif
1	0	2	1	/base/skirmishNew_v0.kif
1	0	2	1	/base/skirmishZeroSum_v0.kif
1	0	2	1	/base/solitaireChineseCheckers_v0.kif
1	0	2	1	/base/speedChess_v1.kif
1	0	2	1	/base/ttcc4_2player_alt_v0.kif
1	0	2	1	/base/ttcc4_2player_small_v0.kif
1	0	2	1	/base/ttcc4_2player_v0.kif
1	0	2	1	/base/ttcc4_v0.kif
1	0	2	1	/base/tttcc4_v0.kif
1	0	2	1	/dresden/3pffa_v0.kif
1	0	2	1	/dresden/3pttc_v0.kif
1	0	2	1	/dresden/brawl_v0.kif
1	0	2	1	/dresden/checkers-cylinder-mustjump_v0.kif
1	0	2	1	/dresden/checkers-mustjump-torus_v0.kif
1	0	2	1	/dresden/checkers-suicide-cylinder-mustjump_v0.kif
1	0	2	1	/dresden/chinesecheckers2_v0.kif
1	0	2	1	/dresden/chinesecheckers3_v0.kif
1	0	2	1	/dresden/crisscross_v0.kif
1	0	2	1	/dresden/cylinder-checkers_v0.kif
1	0	2	1	/dresden/endgame_v0.kif
1	0	2	1	/dresden/grid_game2_v0.kif
1	0	2	1	/dresden/guess_v0.kif
1	0	2	1	/dresden/hallway_v0.kif
1	0	2	1	/dresden/kalaha_2009_v0.kif

#AD	#UAD	#AI	#UAI	NAME
1	0	2	1	/dresden/logistics_v0.kif
1	0	2	1	/dresden/minichess_v0.kif
1	0	2	1	/dresden/minichess-evilconjuncts_v0.kif
1	0	2	1	/dresden/mummymaze2p_v0.kif
1	0	2	1	/dresden/mummymaze2p-comp2007_v0.kif
1	0	2	1	/dresden/pacman3p_v0.kif
1	0	2	1	/dresden/PEG_3v1_turn_v0.kif
1	0	2	1	/dresden/skirmish_v0.kif
1	0	2	1	/dresden/skirmish3_v0.kif
1	0	2	1	/dresden/tttcc4_v0.kif
1	0	2	1	/stanford/alquerque_v0.kif
1	0	2	1	/stanford/alquerquezero_v0.kif
1	0	2	1	/stanford/freeforall_v0.kif
1	0	2	1	/stanford/hex_v0.kif
1	0	2	1	/stanford/kono_v0.kif
1	0	2	1	/stanford/madness_v0.kif
1	0	2	1	/stanford/skirmish_v0.kif
1	0	2	1	/stanford/tictoctoe_v0.kif
1	0	2	2	/base/ticTacHeaven_v0.kif
1	0	3	0	/base/checkLines_v0.kif
2	0	0	0	/base/blockerSerial_v0.kif
2	0	0	0	/base/blockerSerial_v1.kif
2	0	0	0	/base/firefighter_v0.kif
2	0	0	0	/base/futoshiki4_v0.kif
2	0	0	0	/base/futoshiki5_v0.kif
2	0	0	0	/base/futoshiki6_v0.kif
2	0	0	0	/base/hidato19_v0.kif
2	0	0	0	/base/hidato37_v0.kif
1	0	0	0	/base/tron_10x10_v0.kif
2	0	0	0	/base/troublemaker01_v0.kif
2	0	0	0	/base/troublemaker02_v0.kif
2	0	0	0	/dresden/blockerparallel_v0.kif
2	0	0	0	/dresden/blockerserial_v0.kif
2	0	0	0	/dresden/firefighter_v0.kif
2	0	0	0	/dresden/ticblock_v0.kif
2	0	0	0	/dresden/troublemaker01_v0.kif
2	0	0	0	/dresden/troublemaker02_v0.kif
2	0	0	0	/stanford/dualrainbow_v0.kif
2	0	1	0	/base/asteroidsParallel_v0.kif
2	0	1	0	/base/asteroidsParallel_v1.kif
2	0	1	0	/base/blocksWorldParallel_v0.kif
2	0	1	0	/base/brain_teaser_extended_v0.kif
2	0	1	0	/base/hanoi_6_disks_v0.kif
2	0	1	0	/base/hodgepodge_v0.kif
2	0	1	0	/base/incredible_v0.kif
2	0	1	0	/base/queens08lg_v0.kif
2	0	1	0	/base/queens31lg_v0.kif
1	0	1	0	/base/racetrackcorridor_v1.kif
2	0	1	0	/base/snakeParallel_v0.kif
2	0	1	0	/dresden/asteroidsparallel_v0.kif
2	0	1	0	/dresden/blocksworldparallel_v0.kif
2	0	1	0	/dresden/brain_teaser_extended_v0.kif
2	0	1	0	/dresden/hanoi_6_disks_v0.kif
2	0	1	0	/dresden/incredible_v0.kif
2	0	1	0	/dresden/platformjumpers_v0.kif
2	0	1	0	/stanford/dualhamilton_v0.kif
2	0	1	0	/stanford/dualhunter_v0.kif
2	0	1	0	/stanford/platformjumpers_v0.kif
2	0	1	1	/base/breakthroughWalls_v0.kif
2	0	1	1	/base/double_tictactoe_dengji_v0.kif
2	0	1	1	/base/gt_centipede_v0.kif
2	0	1	1	/base/ticTacToeParallel_v0.kif
2	0	1	1	/dresden/double_tictactoe_dengji_v0.kif
2	0	1	1	/dresden/gt_centipede_v0.kif
2	0	1	1	/dresden/tictactoeparallel_v0.kif
2	0	2	0	/base/asteroidsSerial_v0.kif
2	0	2	0	/base/asteroidsSerial_v1.kif
2	0	2	0	/base/blocksWorldSerial_v0.kif
2	0	2	0	/base/blocksWorldSerial_v1.kif

#AD	#UAD	#AI	#UAI	NAME
2	0	2	0	/dresden/asteroidsserial_v0.kif
2	0	2	0	/dresden/blocksworldserial_v0.kif
2	0	2	2	/base/bunk_t_v0.kif
2	0	2	2	/base/connectFourSimultaneous_v0.kif
2	0	2	2	/base/doubletictactoe_v0.kif
2	0	2	2	/base/doubletoetictac_v0.kif
2	0	2	2	/base/dualConnect4_v0.kif
2	0	2	2	/base/dualConnect4_v1.kif
2	0	2	2	/base/ticTacToeSerial_v0.kif
2	0	2	2	/dresden/bunk_t_v0.kif
2	0	2	2	/dresden/doubletictactoe_v0.kif
2	0	2	2	/dresden/doubletoetictac_v0.kif
2	0	2	2	/dresden/dualconnect4_v0.kif
2	0	2	2	/dresden/tictactoeserial_v0.kif
2	0	2	2	/stanford/dualconnectfour_v0.kif
2	0	2	2	/stanford/jointconnectfour_v0.kif
2	0	3	1	/base/snakeAssemblit_v0.kif
2	0	3	1	/base/snakeAssemblit_v1.kif
2	0	4	2	/base/chinook_v0.kif
2	0	4	2	/stanford/chinook_v0.kif
2	1	1	0	/stanford/multiplehamilton_v0.kif
3	0	0	0	/stanford/triplesukoshi_v0.kif
3	0	1	0	/base/factoringEasyTurtleBrain_v0.kif
3	0	1	0	/base/switches_v0.kif
3	0	1	0	/stanford/jointbuttonsandlights_v0.kif
3	2	2	2	/base/ticTacHeavenFC_v0.kif
4	0	0	0	/base/factoringGeorgeForman_v0.kif
4	0	0	0	/base/simultaneousWin2_v0.kif
4	0	1	0	/base/lightsOnParallel_v0.kif
4	0	1	0	/base/lightsOnSimul4_v0.kif
4	0	1	0	/base/lightsOnSimultaneous_v0.kif
4	0	1	0	/base/nim1_v0.kif
4	0	1	0	/base/nim2_v0.kif
4	0	1	0	/base/nim3_v0.kif
4	0	1	0	/base/nim4_v0.kif
4	0	1	0	/dresden/nim1_v0.kif
4	0	1	0	/dresden/nim2_v0.kif
4	0	1	0	/dresden/nim3_v0.kif
4	0	1	0	/dresden/nim4_v0.kif
5	0	1	0	/base/factoringMutuallyAssuredDestruction_v0.kif
9	0	0	0	/stanford/selectivesukoshi_v0.kif
9	0	1	0	/stanford/bestbuttonsandlights_v0.kif
9	8	0	0	/stanford/multiplesukoshi_v0.kif
9	8	1	0	/stanford/multiplebuttonsandlights_9_v0.kif
9	8	1	0	/stanford/multiplebuttonsandlights_v0.kif
9	8	2	1	/stanford/multipletictactoe_v0.kif
14	0	0	0	/base/nonogram_5x5_1_v0.kif
15	0	1	0	/stanford/bestbuttonsandlightsbig_v0.kif
16	0	1	0	/dresden/lightson2x2_v0.kif
19	0	0	0	/base/nonogram_10x10_1_v0.kif
21	11	1	0	/base/factoringImpossibleTurtleBrain_v0.kif
21	11	1	0	/base/factoringMediumTurtleBrain_v0.kif
25	24	1	0	/stanford/multiplebuttonsandlights25_v0.kif

Bibliographie

Abramson, B. et Korf, R. E. (1987). A Model of Two-Player Evaluation Functions. *In Proceedings of the 6th National Conference on Artificial Intelligence. Seattle, WA, July 1987.*, pages 90–94.

Aloul, F. A., Markov, I. L. et Sakallah, K. A. (2001). Faster sat and smaller bdds via common function structure. *In Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 443–448, Piscataway, NJ, USA. IEEE Press.

Aloul, F. A., Ramani, A., Markov, I. L. et Sakallah, K. A. (2002). Solving difficult sat instances in the presence of symmetry. *In Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 731–736, New York, NY, USA. ACM.

Aloul, F. A., Sakallah, K. A. et Markov, I. L. (2006). Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558.

Amir, E. (2001). Efficient approximation for triangulation of minimum treewidth. *In IN PROCEEDINGS OF THE 17TH CONFERENCE ON UNCERTAINTY IN ARTIFICIAL INTELLIGENCE*, pages 7–15. Morgan Kaufmann Publishers.

Amir, E. et Engelhardt, B. (2003). Factored planning. *In IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 929–935.

Areces, C., Bustos, F., Dominguez, M. et Hoffmann, J. (2014). Optimizing planning domains by automatic action schema splitting. *In Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*, Portsmouth, NH, USA.

Asai, M. et Fukunaga, A. (2015). Solving large-scale planning problems by decomposition and macro generation. *In Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, pages 16–24.

Banerjee, B., Kuhlmann, G. et Stone, P. (2006). Value function transfer for general game playing. *In ICML workshop on Structural Knowledge Transfer for Machine Learning*.

Becker, A. et Geiger, D. (1996). A sufficiently fast algorithm for finding close to optimal junction trees. *In Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence*, UAI'96, pages 81–89, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Benhamou, B., Nabhani, T., Ostrowski, R. et Saïdi, M. R. (2010). Détection et élimination dynamique de la symétrie dans le problème de satisfiabilité. *In Actes JFPC 2010*, pages 81–90.

Berlekamp, E., Conway, J. et Guy, R. (1982). *Winning Ways for your Mathematical Plays*, volume 2. Academic.

Björnsson, Y. et Schiffel, S. (2013). Comparison of gdl reasoners. *In Proceedings of the IJCAI-13 Workshop on General Game Playing (GIGA'13)*, pages 55–62.

Blum, A. et Furst, M. L. (1997). Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300.

Bodlaender, H. L. (1997). Treewidth : Algorithmic techniques and results. *In* Prívara, I. et Ružička, P., éditeurs : *Mathematical Foundations of Computer Science 1997*, pages 19–36, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bonet, B. et Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1):5 – 33.

Bonnet, E. et Saffidine, A. (2013). Complexité des Jeux. Article invité dans Le bulletin de la ROADEF nn^{o} 30 - Printemps Été 2013.

Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hoock, J.-B., Hérault, T., Rimmel, A., Teytaud, F., Teytaud, O., Vayssière, P. et Yut, Z. (2010). Scalability and parallelization of monte-carlo tree search. *In Computers and Games*, pages 48–58.

Brafman, R. I. et Domshlak, C. (2006). Factored planning : How, when, and when not. *In Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 809–814.

Breuker, D. (1998). *Memory versus search in games*. Thèse de doctorat, Universiteit Maastricht, Maastricht.

Browne, C. B. (2016). A class grammar for general games. *In Advances in Computer Games*, Leiden.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. et Colton, S. (2012). A survey of Monte Carlo Tree Search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.

Brügmann, B. (1993). Monte carlo go.

Cazenave, T. (2009). Nested Monte-Carlo Search. *In* Boutilier, C., éditeur : *IJCAI*, pages 456–461.

Cazenave, T. (2015a). Generalized rapid action value estimation. *In* Qiang Yang, M. W., éditeur : *IJCAI'15 Proceedings of the 24th International Conference on Artificial Intelligence*, pages 754–760, Palo Alto (USA) -. IJCAI'15 Proceedings of the 24th International Conference on Artificial Intelligence, AAAI Press.

Cazenave, T. (2015b). Sequential Halving Applied to Trees. *In IEEE Trans. Comput. Intellig. and AI in Games*, volume 7, pages 102–105.

Cazenave, T. et Jouandeau, N. (2007). On the parallelization of UCT. *In Proceedings of the Computer Games Workshop*, pages 93–101.

Cazenave, T. et Jouandeau, N. (2008). A Parallel Monte-Carlo Tree Search Algorithm. *In* van den Herik, H. J., Xu, X., Ma, Z. et Winands, M. H. M., éditeurs : *Computers and Games*, volume 5131 de *Lecture Notes in Computer Science*, pages 72–80. Springer.

Cazenave, T. et Jouandeau, N. (2009). Parallel nested monte-carlo search. *In IPDPS*, pages 1–6.

Cazenave, T. et Méhat, J. (2010). Ary, a general game playing program.

Cazenave, T. et Saffidine, A. (2011). A forward chaining based game description language compiler.

Cerexhe, T., Rajaratnam, D., Saffidine, A. et Thielscher, M. (2014). A systematic solution to the (de-)composition problem in general game playing. *In Proceedings of the European Conference on Artificial Intelligence (ECAI)*.

Chaslot, G. (2010). *Monte-Carlo Tree Search*. Thèse de doctorat, Universiteit Maastricht.

Chaslot, G., Winands, M., Uiterwijk, J., Van Den Herik, H. et Bouzy, B. (2007). Progressive strategies for monte-carlo tree search. *In Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661.

Chaslot, G., Winands, M. H. M. et van den Herik, H. J. (2008). Parallel Monte-Carlo Tree Search. *In Computers and Games*, pages 60–71.

Clarke, E. M., Emerson, E. A., Jha, S. et Sistla, A. P. (1998). Symmetry reductions in model checking. *In* Hu, A. J. et Vardi, M. Y., éditeurs : *Computer Aided Verification*, pages 147–158, Berlin, Heidelberg. Springer Berlin Heidelberg.

Clune, J. (2007). Heuristic evaluation functions for general game playing. *In Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, AAAI'07, pages 1134–1139. AAAI Press.

Clune, III, J. E. (2008). *Heuristic evaluation functions for general game playing*. Thèse de doctorat, University of California, Los Angeles, Los Angeles, CA, USA. AAI3346923.

Cohen, D., Jeavons, P., Jefferson, C., Petrie, K. E. et Smith, B. M. (2005). Symmetry definitions for constraint satisfaction problems. *In* van Beek, P., éditeur : *Principles and Practice of Constraint Programming - CP 2005*, pages 17–31, Berlin, Heidelberg. Springer Berlin Heidelberg.

Conway, J. (2000). *On Numbers and Games*. Ak Peters Series. Taylor & Francis.

Cox, E., Schkufza, E., Madsen, R. et Genesereth, M. (2009). Factoring general games using propositional automata. *In Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 13–20.

Crawford, J. M. (1992). A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). *In In AAAI Workshop on Tractable Reasoning*, pages 17–22.

Crawford, J. M., Ginsberg, M. L., Luks, E. M. et Roy, A. (1996). Symmetry-breaking predicates for search problems. *In Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, KR'96, pages 148–159, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Darga, P. T., Liffiton, M. H., Sakallah, K. A. et Markov, I. L. (2004). Exploiting structure in symmetry detection for cnf. *In Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 530–534, New York, NY, USA. ACM.

Darga, P. T., Sakallah, K. A. et Markov, I. L. (2008). Faster symmetry discovery using sparsity of symmetries. *In Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 149–154, New York, NY, USA. ACM.

Darwiche, A. et Hopkins, M. (2001). Using recursive decomposition to construct elimination orders, jointrees, and dtrees. *In Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 6th European Conference, ECSQARU 2001, Toulouse, France, September 19-21, 2001, Proceedings*, pages 180–191.

Draper, S. et Rose, A. (2015). Sancho GGP Player Blog.

Ebner, M., Levine, J., Lucas, S. M., Schaul, T., Thompson, T. et Togelius, J. (2013). Towards a Video Game Description Language. *In* Lucas, S. M., Mateas, M., Preuss, M., Spronck, P. et Togelius, J., éditeurs : *Artificial and Computational Intelligence in Games*, volume 6 de *Dagstuhl Follow-Ups*, pages 85–100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Emerson, E. A. et Sistla, A. P. (1996). Symmetry and model checking. *Formal Methods in System Design*, 9(1):105–131.

Enzenberger, M. et Müller, M. (2010). A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm. *In Proceedings of the 12th International Conference on Advances in Computer Games*, ACG'09, pages 14–20, Berlin, Heidelberg. Springer-Verlag.

Finnsson, H. (2007). Cadia-player : A general game playing agent. Mémoire de D.E.A., Reykjavík University, School of Computer Science, Reykjavík.

Finnsson, H. (2012). *Simulation-Based General Game Playing*. Doctor of philosophy, School of Computer Science, Reykjavík University.

Finnsson, H. et Björnsson, Y. (2008). Simulation-based approach to general game playing. *In Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, AAAI'08, pages 259–264. AAAI Press.

Finnsson, H. et Björnsson, Y. (2009). Simulation Control in General Game Playing Agents. *In IJCAI'09 Workshop on General Intelligence in Game Playing Agents*.

Finnsson, H. et Björnsson, Y. (2011). CadiaPlayer: Search Control Techniques. *KI Journal*, 25(1):9–16.

Finnsson, H. et Björnsson, Y. (2010). Learning simulation control in general game playing agents. *In Proceedings of the 24rd national conference on Artificial intelligence*, pages 954–959. AAAI Press.

Fox, M. et Long, D. (1998). The automatic inference of state invariants in TIM. *J. Artif. Intell. Res.*, 9:367–421.

Fox, M. et Long, D. (1999). The detection and exploitation of symmetry in planning problems. *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 956–961.

Fox, M. et Long, D. (2002). Extending the exploitation of symmetries in planning. *In Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23-27, 2002, Toulouse, France*, pages 83–91.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. et Thiele, S. (2010). A user's guide to gringo, clasp, clingo, and iclingo.

Gelly, S., Hoock, J.-B., Rimmel, A., Teytaud, O. et Kalemkarian, Y. (2008). The Parallelization of Monte-Carlo Planning - Parallelization of MC-Planning. *In* Filipe, J., Andrade-Cetto, J. et Ferrier, J.-L., éditeurs : *ICINCO-ICSO*, pages 244–249. INSTICC Press.

Gelly, S. et Silver, D. (2007). Combining online and offline knowledge in UCT. *In Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007*, pages 273–280.

Gelly, S. et Silver, D. (2011). Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875.

Gelly, S., Wang, Y., Munos, R. et Teytaud, O. (2006). Modification of UCT with patterns in Monte-Carlo go. Rapport technique 6062, Inria.

Genesereth, M. (2017). General Game Playing - Home Page. Stanford Logic Group.

Genesereth, M. (2018). Gamemaster - general game playing, `http://ggp.stanford.edu/gamemaster/showgames.php`. Stanford Logic Group.

Genesereth, M. et Björnsson, Y. (2013). The international general game playing competition. *AI Magazine*, 34(2):107–111.

GENESERETH, M. R., LOVE, N. et PELL, B. (2005). General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26(2):62–72.

GEREVINI, A. et SCHUBERT, L. (1998). Inferring state constraints for domain-independent planning. *In Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 905–912, Menlo Park, CA, USA. American Association for Artificial Intelligence.

GRAF, T., LORENZ, U., PLATZNER, M. et SCHAEFERS, L. (2011). Parallel Monte-Carlo Tree Search for HPC Systems. *In Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 365–376, Berlin, Heidelberg. Springer-Verlag.

GREENBLATT, R. D., EASTLAKE, III, D. E. et CROCKER, S. D. (1967). The Greenblatt Chess Program. *In Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 801–810, New York, NY, USA. ACM.

GÜNTHER, M. (2007). Decomposition of single player games. Mémoire de D.E.A., TU-Dresden.

GÜNTHER, M. (2008). Automatic feature construction for general game playing. Mémoire de D.E.A., TU-Dresden.

GÜNTHER, M. et SCHIFFEL, S. (2013). General Game Playing, http://ggpserver.general-game-playing.de/ggpserver/. Computational Logic group, TU Dresden.

GÜNTHER, M., SCHIFFEL, S. et THIELSCHER, M. (2009). Factoring general games. *In Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 27–33.

HAUFE, S., SCHIFFEL, S. et THIELSCHER, M. (2012). Automated verification of state sequence invariants in general game playing. *Artif. Intell.*, 187-188:1–30.

HELMERT, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artif. Intell.*, 173(5-6):503–535.

HUFSCHMITT, A. (2014). Parallélisation d'un general game player sur une architecture mppa. Mémoire de D.E.A., Université Paris 8 de Vincennes à Saint-Denis.

HUFSCHMITT, A., MEHAT, J. et VITTAUT, J.-N. (2015a). MCTS Playouts Parallelization with a MPPA Architecture. *In Proceedings of the IJCAI-15 Workshop on General Game Playing (GIGA'15)*, pages 63–69.

HUFSCHMITT, A., MEHAT, J. et VITTAUT, J.-N. (2015b). Using the MPPA architecture for UCT parallelization. *In* PRESS, I., éditeur : *IADIS International Conference Game and Entertainment Technologies (GET '15)*, pages 109–115.

HUFSCHMITT, A., MEHAT, J. et VITTAUT, J.-N. (2016). A General Approach of Game Description Decomposition for General Game Playing. *In Proceedings of the IJCAI-16 Workshop on General Game Playing (GIGA'16)*, pages 23–29.

Hufschmitt, A., Vittaut, J.-N. et Jouandeau, N. (2018a). Décomposition des jeux pour le general game playing. *In 12èmes Journées de l'Intelligence Artificielle Fondamentale (JIAF'18)*, page 10p.

Hufschmitt, A., Vittaut, J.-N. et Jouandeau, N. (2018b). Statistical ggp games decomposition. *In Proceedings of the IJCAI-18 Workshop on Computer Games (CGW 2018)*, page 19p.

Hufschmitt, A., Vittaut, J.-N. et Méhat, J. (2017). *A General Approach of Game Description Decomposition for General Game Playing*, pages 165–177. Springer International Publishing, Cham.

Jégou, P. (1993). Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. *In AAAI*, pages 731–736. AAAI Press / The MIT Press.

Joslin, D. et Roy, A. (1997). Exploiting symmetry in lifted csps. *In Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 197–202. AAAI Press.

Jouandeau, N. (2013). Intel versus MPPA. Rapport technique, LIASD Université Paris8.

Kaiser, D. M. (2000). A Generic Game Playing Machine. Mémoire de D.E.A., Master's thesis, Florida International University, Miami, FL.

Kaiser, D. M. (2005). The structure of games. *In Proceedings of the 43rd annual Southeast regional conference - Volume 1*, ACM-SE 43, pages 61–62, New York, NY, USA. ACM.

Kaiser, D. M. (2007). Automatic feature extraction for autonomous general game playing agents. *In Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 93 :1–93 :7, New York, NY, USA. ACM.

Kalray (2013a). *Getting started reviewed 1.01b*. Kalray SA.

Kalray (2013b). *MPPA® ACCESSCORE 1.0.1 - POSIX Programming Reference Manual - KETD-325 W08*. Kalray SA. 142 pages.

Kalray (2014). *MPPA® ACCESSCORE 1.2.1 - POSIX Programming Reference Manual - KETD-325 W08*. Kalray SA. 166 pages.

Karger, D. R. et Stein, C. (1996). A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640.

Karnin, Z., Koren, T. et Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. *In* Dasgupta, S. et McAllester, D., éditeurs : *Proceedings of the 30th International Conference on Machine Learning*, volume 28 de *Proceedings of Machine Learning Research*, pages 1238–1246, Atlanta, Georgia, USA. PMLR.

Kato, H. et Takeuchi, I. (2010). Parallel monte-carlo tree search with simulation servers. *In Proceedings of the 2010 International Conference on Technologies and Applications of Artificial Intelligence*, TAAI '10, pages 491–498, Washington, DC, USA. IEEE Computer Society.

Kelareva, E., Buffet, O., Huang, J. et Thiébaux, S. (2007). Factored planning using decomposition trees. *In IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 1942–1947.

Kirci, M., Sturtevant, N. et Schaeffer, J. (2011). A GGP Feature Learning Algorithm. *KI - Künstliche Intelligenz*, 25(1):1–8.

Kishimoto, A. et Müller, M. (2004). A general solution to the graph history interaction problem. *In Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 644–649.

Kissmann, P. et Edelkamp, S. (2010). Instantiating general games using prolog or dependency graphs. *In Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*, KI'10, pages 255–262, Berlin, Heidelberg. Springer-Verlag.

Knoblock, C. A. (1990). Learning abstraction hierarchies for problem solving. *In Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, July 29 - August 3, 1990, 2 Volumes.*, pages 923–928.

Knoblock, C. A. (1991). *Automatically Generating Abstractions for Problem Solving*. Thèse de doctorat, School of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-91-120.

Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2).

Kocsis, L. et Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *In Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg. Springer-Verlag.

Koriche, F., Lagrue, S., Piette, E. et Tabary, S. (2016). Stochastic constraint programming for general game playing with imperfect information. *In General Intelligence in Game-Playing Agents (GIGA'16) at the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages –.

Koriche, F., Lagrue, S., Piette, E. et Tabary, S. (2017a). Woodstock : Un programme-joueur générique dirigé par les contraintes stochastiques. *RIA, numéro spécial "IA des jeux informatisés"*, 31(3):281–310.

Koriche, F., Lagrue, S., Éric Piette et Tabary, S. (2017b). Constraint-based symmetry detection in general game playing. *In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 280–287.

Kowalski, J. et Kisielewicz, A. (2016). Towards a real-time game description language. *In Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016), Volume 2, Rome, Italy, February 24-26, 2016.*, pages 494–499.

Kowalski, J. et Szykuła, M. (2013). Game description language compiler construction. *In Proceedings of the Australasian Joint Conference on Artificial Intelligence*. Springer.

Kuhlmann, G., Dresner, K. et Stone, P. (2006). Automatic Heuristic Construction in a Complete General Game Player. *In Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1457–62.

Kuhlmann, G. et Stone, P. (2007). Graph-based domain mapping for transfer learning in general games. *In Proceedings of the 18th European Conference on Machine Learning*.

Lanctot, M., Winands, M. H. M., Pepels, T. et Sturtevant, N. R. (2014). Monte carlo tree search with heuristic evaluations using implicit minimax backups. *CoRR*, abs/1406.0486.

Lancucki, A. (2014). GGP with advanced reasoning and board knowledge discovery. *CoRR*, abs/1401.5813.

Lansky, A. L. et Getoor, L. (1995). Scope and abstraction : Two criteria for localized planning. *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, 14:1612 – 1619.

Long, D. et Fox, M. (2003). Symmetries in planning problems. *In Proceedings of SymCon'03 (CP Workshop)*, pages 142–152.

Love, N., Hinrichs, T., Haley, D., Schkufza, E. et Genesereth, M. (2006). General Game Playing : Game Description Language Specification. Rapport technique LG-2006-01, Stanford University.

Marcolino, L. S. et Matsubara, H. (2011). Multi-agent Monte Carlo Go. *In The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, pages 21–28, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

McKay, B. D. et Piperno, A. (2013). Practical graph isomorphism, II. *CoRR*, abs/1301.1493.

Mears, C., de la Banda, M. G. et Wallace, M. (2009). On implementing symmetry detection. *Constraints*, 14(4):443–477.

Mehat, J. et Cazenave, T. (2011). Combining uct and nested monte carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277.

Méhat, J. et Cazenave, T. (2011a). A Parallel General Game Player. *KI*, 25(1):43–47.

Méhat, J. et Cazenave, T. (2011b). Tree parallelization of Ary on a cluster. *In GIGA 2011, IJCAI 2011*, Barcelona.

Michulke, D. et Schiffel, S. (2012). Distance features for general game playing agents. *In Proceedings of the 4th International Conference on Agents and Artificial Intelligence (ICAART)*.

Michulke, D. et Thielscher, M. (2009). Neural networks for state evaluation in general game playing. *In Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2009, Bled, Slovenia, September 7-11, 2009, Proceedings, Part II*, pages 95–110.

Miguel, I. (2001). Symmetry-breaking in planning : Schematic constraints. *In Proceedings of the CP'OI Workshop on Symmetry in Constraints*, pages 17–24.

Miller, A., Donaldson, A. et Calder, M. (2006). Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3).

Müller, M. (1999). Decomposition search : A combinatorial games approach to game tree search, with applications to solving go endgames. *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 578–583.

Méhat, J. (2013). Rapport sur la compétition de general game playing de 2013 première partie : déroulement de la compétition. Rapport technique, LIASD, Université de Paris 8.

Méhat, J. et Cazenave, T. (2008). An Account of a Participation to the 2007 General Game Playing Competition.

Méhat, J. et Cazenave, T. (2010). Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):271–277.

Nebel, B., Dimopoulos, Y. et Koehler, J. (1997). Ignoring irrelevant facts and operators in plan generation. *In* Steel, S. et Alami, R., éditeurs : *Recent Advances in AI Planning*, pages 338–350, Berlin, Heidelberg. Springer Berlin Heidelberg.

Palay, A. (1985). *Searching With Probabilities*. Research Notes in Artificial Intelligence Series. Pitman Advanced Publishing Program.

Pell, B. D. (1993). *Strategy generation and evaluation for meta-game playing*. Thèse de doctorat, Trinity College, University of Cambridge.

Piette, E. (2016). *Une nouvelle approche au General Game Playing dirigée par les contraintes*. Thèse de doctorat, Universitée d'Artois.

Pitrat, J. (1968). Realization of a general game-playing program. *In IFIP Congress*, pages 1570–1574.

Pitrat, J. (1971). A general game playing program. *In* Findler et Meltzer, éditeurs : *Artificial Intelligence and Heuristic Programming*, pages 125–155. Edinburgh University Press.

Pitrat, J. (1976). A program to learn to play chess. *In Pattern Recognition and Artificial Intelligence*, pages 399–419. Academic Press Ltd. London, UK.

Puget, J. (2005). Automatic detection of variable and value symmetries. *In Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, pages 475–489.

Reps, T. W., Loginov, A. et Sagiv, S. (2002). Semantic minimization of 3-valued propositional formulae. *In 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, page 40.

Rintanen, J. (1998). A planning algorithm not based on directional search. *In PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING : PROCEEDINGS OF THE SIXTH INTERNATIONAL CONFERENCE (KR '98*, pages 617–624. Morgan Kaufmann Publishers.

Rintanen, J. (2000). An iterative algorithm for synthesizing invariants. *In Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 806–811. AAAI Press.

Rintanen, J. (2003). Symmetry reduction for sat representations of transition systems. *In Proceedings of the Thirteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'03, pages 32–40. AAAI Press.

Rintanen, J. (2008). Regression for classical and nondeterministic planning. *In In Proc. of the European Conf. on Artificial Intelligence (ECAI*, pages 568–572.

Romero, J., Saffidine, A. et Thielscher, M. (2014). Solving the inferential frame problem in the general game description language. *In AAAI*, pages 515–521. AAAI Press.

Rosin, C. (2011). Nested rollout policy adaptation for monte carlo tree search. *In Proc. 22nd Int. Joint Conf. Artif. Intell.*, pages 649–654, Barcelona, Spain.

Roy, P. et Pachet, F. (1998). Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. *In in Workshop on Non Binary Constraints, ECAI-98*, pages 27–33.

Saffidine, A. (2014). The game description language is turing complete. *IEEE Trans. Comput. Intellig. and AI in Games*, 6(4):320–324.

Saïs, L. (2008). *Problème SAT : progrès et défis*. Collection Programmation par contraintes. Hermes Science Publications.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II: recent progress. *IBM J. Res. Dev.*, 11(6):601–617.

Schaefers, L., Platzner, M. et Lorenz, U. (2011). UCT-Treesplit - Parallel MCTS on Distributed Memory. *In MCTS Workshop*, Freiburg, Germany.

Schiffel, S. (2010). Symmetry detection in general game playing. *In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*.

Schiffel, S. (2011). *Knowledge-Based General Game Playing*. Thèse de doctorat, Technische Universität Dresden.

Schiffel, S. (2016). Grounding GDL Game Descriptions. *In Proceedings of the IJCAI-16 Workshop on General Game Playing (GIGA'16)*, pages 15–22.

Schiffel, S. et Björnsson, Y. (2014). Efficiency of GDL Reasoners. *In IEEE Trans. Comput. Intellig. and AI in Games*, volume 6, pages 343–354.

Schiffel, S. et Thielscher, M. (2007a). Fluxplayer: A successful general game player. *In In : Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 1191–1196. AAAI Press.

Schiffel, S. et Thielscher, M. (2007b). *Automatic Construction of a Heuristic Search Function for General Game Playing*. Hyderabad, India.

Schkufza, E. (2007). Decomposition of games for efficient reasoning. *In Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings*, pages 409–410.

Schkufza, E., Love, N. et Genesereth, M. R. (2008). Propositional Automata and Cell Automata : Representational Frameworks for Discrete Dynamic Systems. *In AI 2008 : Advances in Artificial Intelligence, 21st Australasian Joint Conference on Artificial Intelligence, Auckland, New Zealand, December 1-5, 2008. Proceedings*, pages 56–66.

Schreiber, S. (2014). GGP.org - Tiltyard Gaming Server, http://tiltyard.ggp.org/. CS227b class at Stanford University.

Schreiber, S. (2017). GGP.org public game repositories. Stanford Logic Group.

Sharma, S., Kobti, Z. et Goodwin, S. (2008a). Learning and Knowledge Generation in General Games. *In IEEE Symposium On Computational Intelligence and Games*, CIG '08, pages 329–335, Sch. of Comput. Sci., Univ. of Windsor.

Sharma, S., Kobti, Z. et Goodwin, S. (2008b). Knowledge Generation for Improving Simulations in UCT for General Game Playing. *In Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence : Advances in Artificial Intelligence*, AI '08, pages 49–55, Berlin, Heidelberg. Springer-Verlag.

Sharma, S., Kobti, Z. et Goodwin, S. G. (2009). Coevolving intelligent game players in a cultural framework. *In Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, CEC'09, pages 638–645, Piscataway, NJ, USA. IEEE Press.

Sironi, C. F. et Winands, M. H. M. (2016). Optimizing Propositional Networks . *In Proceedings of the IJCAI-16 Workshop on General Game Playing (GIGA'16)*, pages 23–29.

Slate, J. et Atkin, L. (2012). *Chess Skill in Man and Machine*, chapitre CHESS 4.5 : The Northwestern University Chess Program, pages 82–118. Springer New York.

Soejima, Y., Kishimoto, A. et Watanabe, O. (2010). Evaluating Root Parallelization in Go. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):278–287.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Mach. Learn.*, 3(1):9–44.

Swiechowski, M. et Mandziuk, J. (2014). Specialized vs. multi-game approaches to AI in games. *In Intelligent Systems'2014 - Proceedings of the 7th International Conference Intelligent Systems IEEE IS'2014, September 24-26, 2014, Warsaw, Poland, Volume 1 : Mathematical Foundations, Theory, Analyses*, pages 243–254.

Tak, M. J. W., Winands, M. H. M. et Björnsson, Y. (2012). N-grams and the last-good-reply policy applied in general game playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(2):73–83.

Thielscher, M. (2009). *Answer Set Programming for Single-Player Games in General Game Playing*, pages 327–341. Springer Berlin Heidelberg, Berlin, Heidelberg.

Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. *In AAAI*.

Thielscher, M. (2011). The general game playing description language is universal. *In Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1107–1112. AAAI Press.

Thielscher, M. (2017). Gdl-iii : A description language for epistemic general game playing. *In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1276–1282.

Trutman, M. et Schiffel, S. (2015). Creating action heuristics for general game playing agents. *In Computer Games - Fourth Workshop on Computer Games, CGW 2015, and the Fourth Workshop on General Intelligence in Game-Playing Agents, GIGA 2015, Held in Conjunction with the 24th International Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 26-27, 2015, Revised Selected Papers*, pages 149–164.

Utgoff, P. E. (2001). *Machines that Learn to Play Games*, chapitre 7, Feature Construction for Game Playing, pages 131—-152. Nova Science Publishers, Huntington, NY.

Vittaut, J. et Méhat, J. (2014). 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 - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pages 1121–1122.

Vittaut, J. N. (2017). *LeJoueur : un programme de General Game Playing pour les jeux à information incomplète et/ou imparfaite*. Thèse de doctorat, Université Paris 8.

Waledzik, K. et Mandziuk, J. (2014). An automatically generated evaluation function in general game playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 6(3):258–270.

Winands, M. H., Björnsson, Y. et Saito, J.-T. (2008). Monte-carlo tree search solver. *In Proceedings of the 6th International Conference on Computers and Games*, CG '08, pages 25–36, Berlin, Heidelberg. Springer-Verlag.

Yoshizoe, K., Kishimoto, A., Kaneko, T., Yoshimoto, H. et Ishikawa, Y. (2011). Scalable distributed Monte-Carlo Tree Search. *In SOCS*.

Zhang, H., Liu, D. et Li, W. (2015). Space-consistent game equivalence detection in general game playing. *In Computer Games - Fourth Workshop on Computer Games, CGW 2015, and the Fourth Workshop on General Intelligence in Game-Playing Agents, GIGA 2015, Held in Conjunction with the 24th International Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 26-27, 2015, Revised Selected Papers*, pages 165–177.

Zhao, D. (2009). Decomposition of multi-player games. Mémoire de D.E.A., TU-Dresden.

Zhao, D., Schiffel, S. et Thielscher, M. (2009). Decomposition of multi-player games. *In Proceedings of the Australasian Joint Conference onArtificial Intelligence*, volume 5866, pages 475–484. Springer.

Résumé

Dans cette thèse nous présentons une approche générale et robuste pour la décomposition des jeux décrits en *Game Description Language* (GDL). Dans le domaine du *General Game Playing* (GGP), les joueurs peuvent significativement diminuer le coût de l'exploration d'un jeu s'ils disposent d'une version décomposée de celui-ci. Les travaux existants sur la décomposition des jeux s'appuient sur la structure syntaxique des règles et sur des habitudes d'écriture du GDL. Nous proposons une méthode plus générale pour décomposer les jeux solitaires ou multijoueurs. Dans un premier temps nous envisageons une approche fondée sur l'analyse logique des règles. Celle-ci nécessite l'utilisation d'heuristiques, qui en limitent la robustesse, et le coûteux calcul de la forme normale disjonctive des règles. Une seconde approche plus efficace est fondée sur la collecte d'informations durant des simulations (*playouts*). Cette dernière permet la détection des liens de causalité entre les actions et les fluents d'un jeu. Elle est capable de traiter les différents types de jeux composés et de prendre en charge certains cas difficiles comme les jeux à actions composées et les jeux en série. Nous avons testé notre approche sur un panel de 597 jeux GGP. Pour 70% des jeux, la décomposition nécessite moins d'une minute en faisant 5k *playouts*. Nous montrons de 87% d'entre eux peuvent être correctement décomposés après seulement 1k *playouts*. Nous ébauchons également une approche originale pour jouer avec ces jeux décomposés. Les tests préliminaires sur quelques jeux solitaires sont prometteurs.

Une autre contribution de cette thèse est l'évaluation d'une architecture MPPA pour la parallélisation d'un joueur GGP (*LeJoueur* de Jean-Noël Vittaut).

Mots-clefs : General Game Playing, Game Description Langage, décomposition, causalité, actions composées, jeux en série, parallélisation, MPPA.

Abstract

This PhD thesis presents a robust and general approach for the decomposition of games described in *Game Description Language* (GDL). In the *General Game Playing* framework (GGP), players can drastically decrease game search cost if they hold a decomposed version of the game. Previous works on decomposition rely on syntactical structures and writing habits of the GDL. We offer an approach to decompose single or multi-player games. First, we consider an approach based on logical rule analysis. This requires the use of heuristics, which limit its robustness, and the costly calculation of the disjunctive normal form of the rules. A second more efficient approach is based on information gathering during simulations of the game (*playouts*). The latter allows the detection of causal links between actions. It can handle the different classes of compound games and can process some difficult cases like synchrounous parallel games with compound moves and serial games. We tested our program on 597 games. Given 5k *playouts*, 70% of the games are decomposed in less than one minute. We demonstrate that for 87% of the games, 1k *playouts* are sufficient to obtain a correct decomposition. We also sketch an original approach to play with these decomposed games. Preliminary tests on some one-player games are promising.

Another contribution of this thesis is the evaluation of the MPPA architecture for the parallelization of a GGP player (*LeJoueur* of Jean-Noël Vittaut).

Keywords : General Game Playing, Game Description Langage, decomposition, causality, compound moves, serial games, parallelization, MPPA.