WoodStock : un programme-joueur générique dirigé par les contraintes stochastiques

Frédéric Koriche, Sylvain Lagrue, Éric Piette, Sébastien Tabary

Université Lille-Nord de France CRIL - CNRS UMR 8188 Artois, F-62307 Lens
koriche@cril.fr lagrue@cril.fr epiette@cril.fr tabary@cril.fr

RÉSUMÉ. *Cet article décrit WoodStock, le premier programme-joueur générique modélisant chaque jeu issu du General Game Playing (GGP) par un réseau de contraintes stochastiques (SCSP). Chaque action jouée est décidée par la résolution de ce dernier par l'algorithme MAC-UCB. Après traduction d'une instance GDL (Game Description Language) en un réseau représentant l'état du jeu à tout temps, WoodStock résout chaque état par la maintenance d'arc-consistance (MAC) itérativement guidé par l'échantillonnage par bandit stochastique (UCB) des états suivants. À l'aide de cet algorithme, WoodStock est depuis mars 2016, le leader de la compétition continue de GGP organisée sur le serveur Tiltyard. De plus, dans sa dernière version exploitant les symétries de jeux déduites par la détection de symétries de contraintes, l'espace de recherche associé à un jeu est significativement réduit. Suite à cela, WoodStock est devenu champion lors de la compétition internationale de General Game Playing 2016 (IGGPC 2016).*

ABSTRACT. *This article describes WoodStock, the first general game player modeling each game from the General Game Playing (GGP) by a stochastic constraint network (SCSP). Each action played is decided by the resolution of this last one by the algorithm MAC-UCB. After the translation of an instance described in Game Description Language (GDL) in a network representative of the state of the game at any time, WoodStock solves each state by the maintening arc-consistency algorithm (MAC) iteratively guided by the bandit-based stochastic sampling (UCB) of the next states. Thanks to this algorithm, WoodStock is since march 2016, the leader of the GGP Tiltyard continuous tournament. Moreover, in its last version exploiting the game symmetries finding by the constraint symmetry detection, the search space associated with a game is significatively reduced. With that, WoodStock is now the GGP champion after its victory at the International General Game Playing Competition 2016 (IGGPC 2016).*

MOTS-CLÉS : *compétition internationale de* general game playing *(IGGPC), programmation par contraintes stochastiques (SCSP), échantillonnage par bandit stochastique (UCB).*

KEYWORDS: *international general game playing competition (IGGPC), stochastic constraint satisfaction problem (SCSP), bandit-based stochastic sampling (UCB).*

DOI:10.3166/RIA.31.281-310

1. Introduction

Contrairement à AlphaGo (Silver *et al.*, 2016) l'Intelligence Artificielle (IA) dédiée au *Go* remportant de nombreux succès, l'objectif du *General Game Playing* (GGP) est de concevoir des programmes-joueurs, non pas dédiés à un jeu de stratégie particulier, mais génériques de par leur capacité à jouer de manière pertinente à une grande variété de jeux. Il s'agit d'algorithmes de décisions capables de jouer à des jeux inconnus sans connaissance spécifique ne permettant donc pas l'utilisation d'heuristiques dédiées. Ils doivent plutôt être dotés de capacités cognitives de haut niveau issues du raisonnement abstrait ou de stratégies généralistes. De ce fait, le *General Game Playing* présente un défi pour l'IA englobant de nombreux thèmes comme la représentation de connaissance, la génération automatique d'heuristiques, la planification ou encore l'apprentissage automatique.

Dans le cadre GGP, les jeux sont décrits dans un langage de représentation, appelé GDL pour *Game Description Language* (Love *et al.*, 2006). Basé sur la programmation logique, la première version de ce langage capture tout jeu à information complète et parfaite. Récemment, une nouvelle version de ce langage, GDL-II (GDL *with Incomplete Information* (Thielscher, 2010)) englobe les jeux à information incomplète.

Depuis 2005, une compétition annuelle, dénommée *International General Game Playing Competition* (IGGPC (Genesereth, Björnsson, 2013)), est organisée par l'université de *Stanford* lors des conférences internationales AAAI (Genesereth, Love, 2005) ou IJCAI. Cette compétition réunie différents programmes-joueurs génériques se confrontant sur de nouvelles instances de jeux, où aucune intervention humaine n'est permise et où chaque action jouée doit être réalisée dans le temps imparti. De même, depuis 2009, *Tiltyard* organise une compétition GGP en ligne se déroulant en continue (Schreiber, 2014) et regroupant près d'un millier de programmes-joueurs génériques. Suite à ces compétitions différents programmes-joueurs génériques ont vu le jour utilisant diverses approches comme la programmation logique (Thielscher, 2005), l'*answer set programming* (Möller *et al.*, 2011), la construction automatique de fonctions d'évaluations (Clune, 2007) et ce qui fait à ce jour référence, les méthodes de type Monte Carlo (Finnsson, Björnsson, 2008 ; Cazenave, Mehat, 2010).

Dans (Koriche *et al.*, 2016), nous introduisons une nouvelle approche au *General Game Playing*, dénommée MAC-UCB, basée sur la programmation par contraintes stochastiques (SCSP) (Walsh, 2009). Expérimentalement, ce dernier s'est montré compétitif toutefois à ce jour aucun programme-joueur n'utilise actuellement cet algorithme en pratique. Ce papier présente WoodStock (*With Our Own Developper STOchastic Constraint toolKit*), notre programme-joueur générique implémentant MAC-UCB sous les contraintes de communication et de temps imposées par un tournoi GGP. WoodStock confirme les résultats de MAC-UCB sur les autres approches en devenant leader de la compétition continue depuis mars 2016. Enfin, en exploitant les symétries de jeux à l'aide de la structure graphique associée au réseau de contraintes généré pour chaque programme GDL que résout MAC-UCB, WoodStock est devenu

champion GGP 2016 en remportant la dernière compétition internationale de *General Game Playing*.

L'article est organisé de la manière suivante. Le formalisme GDL et un tour d'horizon des principales approches GGP sont réalisés en section 2 avant d'introduire la programmation par contraintes stochastiques en section 3. WoodStock est décrit en section 4 où la génération d'un SCSP équivalent à un jeu GDL est décrite avant d'introduire MAC−UCB et son implémentation. La section suivante illustre les résultats de WoodStock au cours de sa première participation à une compétition GGP mais également au cours de la compétition en continue que propose le serveur *Tiltyard*. La section 6 présente la méthode de réduction de l'espace de recherche utilisée par WoodStock à l'aide de la détection de symétries et les résultats obtenus au cours de la dernière compétition internationale de *General Game Playing* avant de conclure.

2. General Game Playing (GGP)

Le *General Game Playing* (ou souvent nommé GGP) a pris toute son ampleur en 2005 au travers d'un projet initié par le *Stanford Logic Group* [1] (Genesereth, Love, 2005). La grande variété de problèmes considérés partage une structure abstraite commune. Chaque jeu implique un nombre fini de joueurs et un nombre fini d'états, incluant un état distinct initial et un ou plusieurs états terminaux. À chaque tour de jeu, chaque joueur possède un nombre fini d'actions (aussi nommées actions légales). L'état courant du jeu est mis à jour par les conséquences simultanées de l'action de chaque joueur (qui peut être *noop* désignant l'action de ne rien faire). Le jeu commence avec un état initial et après un nombre fini de tours, se termine sur un état terminal au cours duquel un score compris entre 0 et 100 est attribué à chaque joueur.

2.1. Game Description Langage (GDL)

(Love *et al*., 2006) propose le langage *Game Description Language* (très souvent abrégé GDL) permettant d'encoder les jeux à information complète dans une forme plus compacte que leur représentation directe par un arbre de recherche. GDL propose de modéliser chaque état du jeu en utilisant la logique du premier ordre pour définir les différents composants du jeu à modéliser (actions légales, état initial et terminal, évolution du jeu, etc). (Thielscher, 2011b) montre que GDL est un langage suffisamment générique pour décrire des jeux de nombreux types.

GDL est un langage logique purement déclaratif du premier ordre où les entités (les objets) qui composent le jeu sont décrites par des termes et les propriétés sur ces entités sont décrites par des prédicats. Les atomes qui le composent sont souvent catégorisés en deux groupes distincts : la base de *Herbrand* représentant les composants booléens du jeu (les joueurs, les indices d'une ligne ou d'une colonne d'un plateau, la

1. Stanford Logic Group : http://games.stanford.edu/

valeur d'une case, etc) et les atomes d'actions représentant les actions réalisées par les joueurs. L'état d'un jeu est un sous-ensemble de la base de *Herbrand* du jeu. Tous les atomes inclus dans un état sont vrais et ceux exclus sont faux.

À chaque tour de jeu, chaque rôle peut réaliser une ou plusieurs actions légales. Une action dans un état du jeu correspond à une combinaison d'actions, une par rôle. Pour chaque action réalisée, quelques atomes de la base deviennent vrais et d'autres faux, permettant d'atteindre un nouvel état du jeu et par conséquent à de nouvelles actions légales. Dans tout jeu GDL utilisé en compétition, pour chaque état et chaque combinaison d'actions, il n'existe qu'un seul état suivant possible.

Un jeu GDL débute à l'état initial. Les différents joueurs réalisent leurs actions légales dans cet état afin d'atteindre un nouvel état. Ce processus est répété jusqu'à ce que le programme GDL atteigne un état terminal au cours duquel le jeu s'arrête et les joueurs obtiennent les scores adéquats.

GDL impose quelques mots-clés du langage pour définir une syntaxe identique quelque soit le jeu décrit. Le tableau 1 présente les dix mots-clés du langage [2].

Tableau 1. Mots-clés de GDL

Mot-clé	Description
role(j)	j est un joueur
input(j, a)	a est une action possible pour j
base(p)	p est un atome du jeu
init(p)	L'atome p est vrai à l'état initial
true(p)	L'atome p est vrai à l'état courant
does(j, a)	Le joueur j réalise l'action a à l'état courant
next(p)	L'atome p est vrai dans l'état suivant
legal(j, a)	L'action a est légale pour le joueur j à l'état courant
terminal	L'état courant est terminal
goal(j, n)	j reçoit un score de n dans l'état courant

Les différents prédicats utilisés comme mots-clés de GDL doivent répondre à certaines règles d'écriture :

– les mots-clés $role$, $base$, $input$ et $init$ utilisés dans un programme GDL sont obligatoires et doivent être décrits complètement afin de permettre la validité d'un programme GDL;

– les mots-clés $legal$, $goal$ et $terminal$ utilisés en tant que tête d'une règle, ne peuvent accepter que le mot-clé $true$ dans le corps de cette même règle ;

2. Notons que les mots-clés *input* et *base* ne sont pas présents dans l'article original présentant GDL. Ils ont été ajoutés en 2008, dans le but de rendre plus aisé une potentielle compilation de GDL vers d'autres modèles.

– le mot-clé $next$ utilisé en tant que tête d'une règle ne peut accepter que les mots-clés $true$ et $does$ dans le corps de cette même règle ;

– les mots-clés $does$ et $true$ ne peuvent pas être utilisés en tant que tête d'une règle ;

– le score possible d'un joueur défini par le mot-clé `goal` varie entre 0 et 100.

À cet ensemble de mots-clés, il est possible d'ajouter le mot-clé *not* permettant d'indiquer qu'un littéral est négatif et le mot-clé *distinct(d1,d2)* permettant d'indiquer que $d1 \neq d2$.

EXEMPLE 1. — *Le Matching pennies est un jeu simultané à deux joueurs. Chaque joueur possède une pièce. Au même moment, les deux joueurs (j_1 et j_2) retournent leur pièce sur $Pile$ ou $Face$. Une fois que les deux pièces sont retournées, le joueur j_1 remporte le jeu si les deux pièces présentent le même coté, sinon le second joueur remporte la partie. Au début du jeu, le coté de chaque pièce est assimilé à $inconnue$ (avant que la pièce ne soit dévoilée). La figure 1 correspond au programme `GDL` du Matching Pennies.*

```
% roles
role(j1).
role(j2).

% base de Herbrand
base(piece(J,C)) ← role(J), cote(C).
base(piece(J,inconnue)) ← role(J).

% actions possibles
input(J,retourne(C)) ← role(J), cote(C).

% coté d'une pièce
cote(pile).
cote(face).

% état initial
init(piece(j1,inconnue)).
init(piece(j2,inconnue)).

% actions légales
legal(J,retourne(C)) ← role(J), cote(C), true(piece(J,inconnue)).

% mis à jour de l'état du jeu
next(piece(P,C)) ← does(P,retourne(C)).

% états terminaux
terminal ← not(true(piece(P,inconnue))).
% scores
goal(J1,100) ← true(piece(J1,S)), true(piece(J2,S)).
goal(J1,0) ← true(piece(J1,S1)), true(piece(J2,S2)), distinct(S1,S2).
goal(J2,0) ← true(piece(J1,S)), true(piece(J2,S)).
goal(J2,100) ← true(piece(J1,S1)), true(piece(J2,S2)), distinct(S1,S2).
```

Figure 1. Le programme `GDL` correspondant au jeu « Matching Pennies »

Il commence par la description des joueurs $j1$ et $j2$ via l'utilisation du mot-clé role. Puis, la base de Herbrand est définie par l'utilisation du mot-clé base, ici nous définissons les différents atomes du seul prédicat ne représentant pas une action. Par la suite, les atomes d'actions sont caractérisés par l'utilisation du mot-clé input pour chaque joueur, ici seul l'atome $retourne(C)$ pour chaque joueur J est concerné.

Suite à cela, l'état initial est déclaré à l'aide du mot-clé init, ici il s'agit de le définir à l'aide des atomes $piece(j1, inconnue)$ *et* $piece(j2, inconnue)$.

Puis, par l'utilisation du mot-clé legal, nous définissons les règles logiques permettant de représenter les actions légales de chaque joueur en fonction de l'état courant défini par le mot-clé true.

Ensuite, la même chose est réalisée pour définir l'évolution du jeu par des règles logiques dont la tête est définie par le mot-clé next. Ici, on indique l'évolution des termes du prédicat $piece$ *en fonction des actions réalisées à l'état courant par chaque joueur identifié par l'utilisation du mot-clé does.*

Finalement, on définit un état comme terminal par l'utilisation du mot-clé terminal en fonction de l'état courant et les scores associés à chaque joueur par l'utilisation du mot-clé goal. Ici on associe un score de 100 *au joueur qui remporte la partie et un score de* 0 *au joueur qui la perd.*

Une extension du langage `GDL` est proposée par (Thielscher, 2010) dénommée `GDL-II` (pour *GDL with Incomplete Information*) (Thielscher, 2011a). Elle propose notamment le mot-clé *random* décrivant un joueur environnement représentant la notion de chance dans un jeu.

Le *General Game Playing* propose au travers de `GDL` un langage permettant de modéliser un grand nombre de jeux d'une grande variété. Afin de retrouver l'ensemble des jeux `GDL` valides existants, le lecteur pourra se référer à (Schreiber, 2014). Afin de motiver la recherche sur ce thème, une compétition annuelle est organisée au moment des conférences internationales $AAAI$ ou $IJCAI$, dénommée `IGGPC` (pour *International General Game Playing Competition*) (Genesereth, Björnsson, 2013).

2.2. Compétition internationale : `IGGPC`

Chaque compétition de *General Game Playing* se pratique selon le même processus au travers de matchs entre un à plusieurs joueurs selon le jeu `GDL` concerné. En début de match, chaque joueur reçoit le programme `GDL` et possède un nombre prédéfini de secondes ($startclock$), on parle de phase de pré-traitement, au cours duquel chaque joueur se prépare à jouer (analyse du jeu, développement d'une stratégie, etc). Une fois ce temps passé, le match commence. Au cours du match, chaque joueur possède un temps prédéfini ($playclock$), lui permettant de décider de sa prochaine action afin de l'envoyer. Le match se déroule tour par tour jusqu'à atteindre un état terminal du jeu `GDL` où les scores correspondants à cet état sont assignés à chaque joueur.

Au sein de la compétition, un protocole de communication est utilisé pour permettre la communication entre les différents protagonistes d'un match. Ce protocole porte le nom de `GCL` pour *Game Communication Language* au travers de connexions HTTP. Chaque joueur est à l'écoute des différents messages HTTP du *Game Manager* sur un port désigné pour la compétition. Nous détaillons brièvement les cinq types de

messages dans le tableau 2, toutefois, pour plus d'informations, le lecteur pourra se référer à (Love *et al*., 2006).

Tableau 2. GCL protocole

Message	Information
$info$	vérifie la disponibilité d'un programme-joueur
$start$	débute un match
$play$	indique les dernières actions de chaque joueur
$stop$	informe qu'un état terminal est atteint
$abort$	indique que le match est abandonné par le serveur

De plus, une compétition GGP en ligne se déroule en continue sur le serveur Tiltyard (Schreiber, 2014) où plus de $1,000$ compétiteurs s'affrontent sur plus de 150 jeux. Afin d'établir un classement entre les différents participants, le système *Agon* [3] est utilisé. Il s'agit d'une variante du classement Elo (couramment utilisé pour les échecs) au contexte GGP permettant de calculer la qualité (la compétence généraliste) de chaque joueur associée à la difficulté associée au rôle que représente le programme-joueur au cours d'un match dans chaque jeu sur la base de l'historique de l'ensemble des matchs réalisés sur le serveur. Ce système supporte les jeux à un joueur, multi-joueurs, les jeux asymétriques, les jeux à somme nulle ou non, etc.

Depuis la fondation de GDL en 2005, de nombreux programmes-joueurs ont vu le jour et certains ont été élus champions en remportant la compétition IGGPC. Le tableau 3 référence les champions annuels de GGP et les références si disponibles.

Tableau 3. Les champions GGP

Année	Programme-joueur	Référence disponible
2005	Cluneplayer	($Clune$,2007)
2006	Fluxplayer	($Schiffel, Thielscher$,2007)
2007	Cadiaplayer	($Finnsson, Bjrnsson$,2008)
2008	Cadiaplayer	($Finnsson, Bjrnsson$,2008)
2009	Ary	($Mehat, Cazenave$,2008)
2010	Ary	($Finnsson, Bjrnsson$,2011)
2011	TurboTurtle	–
2012	Cadiaplayer	($Finnsson, Bjrnsson$,2011)
2013	TurboTurtle	–
2014	Sancho	($Draper, Rose$,2014)
2015	Galvanise	($Emslie$,2015)

Le premier champion GGP est Cluneplayer, qui réalise une analyse automatique des règles du jeu dans le but de détecter différentes caractéristiques fondamen-

3. Agon : http://www.ggp.org/researchers/analysis_agonRating.html

tales d'un jeu : le gain espéré, le contrôle et la terminaison espérée d'un état donné. Chaque caractéristique identifiée est évaluée en fonction de sa corrélation avec les scores obtenus par chaque joueur. Les caractéristiques finalement sélectionnées sont combinées linéairement pour être utilisées comme fonction d'évaluations spécifique. Cette fonction associée à une recherche minimax dans l'arbre de recherche du jeu permet de décrire le mécanisme de sélection de la prochaine action à jouer.

`Fluxplayer`, le second champion emploie plusieurs éléments communs aux différents jeux `GDL` (plateau, pièces, ...) et les inclut dans une fonction d'évaluation appropriée. De plus, il utilise la logique floue pour déterminer la probabilité d'obtenir un score donné dans un état donné. Les caractéristiques atomiques sont évaluées directement alors que les structures plus complexes sont modélisées par des structures spécifiques.

Dès 2007, tous les champions suivants adoptent les méthodes de type Monte Carlo via l'utilisation de l'algorithme `UCT` (*Upper Confidence bounds applied to Trees*) (Sturtevant, 2008). Brièvement, ces méthodes se basent sur un ensemble de simulations pour choisir leurs actions, plutôt que sur la construction d'heuristiques par des fonctions d'évaluations. Chaque joueur choisit aléatoirement ses mouvements dans l'état courant jusqu'à atteindre un état terminal et enregistre les résultats obtenus sur chaque chemin parcouru. La part d'aléatoire est dirigée par la méthode `UCT` qui assure le suivi du rendement moyen de chaque combinaison état-action qui a été jouée et choisit l'action a^* à explorer dans l'état s selon :

$$a^* = argmax_{a \in A(s)} Q(s,a) + C\sqrt{\frac{ln(N(s)}{N(s,a)}}$$

où $Q(s,a)$ est la fonction d'évaluation d'état, $A(s)$ représente l'ensemble de toutes les actions possibles dans s, $N(s)$ est le nombre de fois où l'état s a été sélectionné dans les simulations précédentes, et $N(s,a)$ indique le nombre de fois où l'action a a été explorée dans l'état s.

`UCT` construit graduellement l'arbre de recherche du jeu et les paires d'état-action sont étiquetées par la moyenne des gains obtenus par simulation. Les séquences de jeux qui semblent mauvaises sont de moins en moins explorées au profit des séquences prometteuses. Ainsi, les séquences prometteuses sont développées de plus en plus vite et la profondeur de l'arbre exploré est de plus en plus importante. L'habituel dilemme exploration/exploitation de toute méthode de recherche dans un arbre est contenu par le choix du paramètre C dans la formule `UCT`.

A ce jour, les méthodes Monte Carlo restent l'état de l'art du `GGP` bien que à partir de 2011, le champion `TurboTurtle` introduit les réseaux de propositions (*propnet*) (Cox *et al.*, 2009) pour représenter les règles d'un jeu `GDL` et ainsi fortement accélérer les simulations. Avant ça, un jeu était représenté par une machine d'états où chaque état correspond à un ensemble d'atomes composés de termes et d'actions changeant d'un état à un autre. Un réseau de propositions est un graphe où les prédicats et les actions sont des nœuds plutôt que des états et où ces nœuds sont entrelacés de

nœuds représentant les connections logiques et les transitions. Un bénéfice important de cette représentation est sa densité face aux machines d'états permettant d'accroître le nombre de simulations réalisées.

(Koriche *et al.*, 2016) présente MAC−UCB, un nouvel algorithme capable d'utiliser la programmation par contraintes stochastiques (SCSP) pour déterminer ses actions. Toutefois avant d'introduire MAC−UCB, il est nécessaire de présenter le cadre SCSP.

3. Programmation par Contraintes Stochastiques (SCSP)

Le cadre CSP permet de modéliser et de résoudre un grand nombre de problèmes. Cependant, certains problèmes demandent plus de souplesse et ne sont pas modélisables par un CSP, c'est à dire uniquement sous la forme d'instanciations strictement autorisées ou strictement interdites par les contraintes. En effet, il existe un grand nombre de problèmes de décisions dont la notion d'incertitude est fondamentale.

Ainsi, dans le but de modéliser des problèmes de décision combinatoire permettant la prise en compte d'incertitudes et de probabilités, nous nous intéressons au cadre SCSP (pour *Stochastic Constraint Satisfaction Problem*) (Walsh, 2009) inspiré du problème de satisfaction stochastique (Littman *et al.*, 2001).

3.1. Définitions et Notations

Un réseau de contraintes stochastiques est un 6-uplet (X, Y, D, P, C, θ) tel que :

– X est l'ensemble fini et ordonné des n variables du problème ;

– Y est le sous-ensemble de X des variables stochastiques du problème ($Y \subset X$);

– D est l'ensemble des domaines de X ($\forall x \in X, dom(x) \in D$) ;

– P est l'ensemble des distributions de probabilités sur les domaines des variables stochastiques composant Y ($\forall x_s \in Y, P_{x_s} \in P$);

– C est l'ensemble des m contraintes du problème portant sur les variables composants X .

– θ est un seuil compris entre 0 et 1 inclus ($\theta \in [0, 1]$).

La formalisation de cette extension s'appuie sur un réseau de contraintes stochastiques où une distinction est réalisée entre deux types de variables: les variables de décisions et les variables stochastiques.

Une variable stochastique est une variable dont une distribution de probabilités est définie sur l'ensemble des valeurs définissant son domaine. Formellement, soit y une variable stochastique où $dom(y)$ représente le domaine de y. À chaque variable stochastique est associée une distribution de probabilités : il s'agit de l'ensemble des probabilités défini sur chaque valeur v du domaine de y ($v \in dom(y)$) tel que v soit affecté à y. On note $P(y = v)$ la probabilité que la valeur v soit affectée à la variable stochastique y et P_y la distribution de probabilité associée à y. Notons que

$\sum_{i=1}^{|dom(y)|} P(x = v_i) = 1$. Une variable de décision est l'appellation donnée à une variable non stochastique. C'est à dire une variable dont il n'existe pas de distribution de probabilité définie sur son domaine. Par la suite, nous notons V l'ensemble des variables de décisions d'un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ tel que $V = X \backslash Y$

La portée d'une contrainte c, nommée $scp(c)$, est définie sur l'ensemble de variables X. La satisfaction d'une contrainte est la même que celle du cadre `CSP`.

Soit un sous-ensemble $U = (x_{d1}, \ldots, x_{dm}) \subseteq V$, une instanciation de U est un assignement I de valeurs $v_1 \in dom(x_{d1}), \ldots, v_m \in dom(x_{dm})$ aux variables $x_{d1}, \ldots x_{dm}$. Une instanciation I sur U est complète si $U = V$. Les assignations des variables stochastiques dans une instanciation sont appelées scénario et une probabilité leurs est associée. Soit I une instanciation d'un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ où $Y = \{x_{s1}, x_{s2}, \ldots, x_{sn}\}$ ($|Y| = sn$). La probabilité associée au scénario de I notée $P(I)$ correspond au produit des probabilités d'assignation des valeurs aux variables stochastiques qui le compose. Formellement, elle est définie par : $\prod_{i=1}^{sn} P(x_{si} = v_i)$.

Une politique π pour le réseau $\mathcal{P}$ est un arbre où chaque nœud interne est étiqueté par une variable x et chaque arête est étiquetée par une valeur dans $dom(x)$. Spécifiquement, les nœuds sont étiquetés selon l'ordre X : le nœud racine est étiqueté par x_{d1}, et chaque fils d'un nœud x_{di} est étiqueté par $x_{d(i+1)}$. Les nœuds de décisions x_{di} ont un unique fils et les nœuds stochastiques x_{si} ont $|dom(x_{si})|$ enfants. Enfin, chaque feuille dans π est étiquetée par l'utilité du scénario spécifié par le chemin de la racine de π à chaque feuille. Notons qu'une politique π peut (comme une instanciation) être partielle, on note $vars(\pi)$ les variables instanciées de V du réseau de contraintes stochastiques. Une politique π couvre une contrainte c si et seulement si $scp(c) \in vars(\pi)$.

La satisfaction d'une politique est définie comme la somme de chaque valeur étiquetant chaque feuille qui la compose pondérée par la probabilité du scénario correspondant à la feuille. Une politique π d'un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ est dite politique solution si sa satisfaction est supérieure ou égale au seuil θ satisfaisant par la même occasion l'ensemble des contraintes C. L'ensemble des politiques solutions d'un réseau de contraintes stochastiques $\mathcal{P}$ est noté *sols($\mathcal{P}$)*.

EXEMPLE 2. — *Soit un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ avec :*

– $X = \{x_{d1}, x_{s1}, x_{d2}, x_{s2}\}$;

– $Y = \{x_{s1}, x_{s2}\}$;

– $D = \{dom(x_{d1}), dom(x_{s1}), dom(x_{d2}), dom(x_{s2})\}$ *avec* $dom(x_{d1}) = dom(x_{s1}) = dom(x_{d2}) = dom(x_{s2}) = \{0, 1, 2\}$;

– $P = \{P_{x_{s1}}, P_{x_{s2}}\}$ *avec* $P_{x_{s1}} = \{P(x_{s1} = 0) = \frac{1}{3}, P(x_{s1} = 1) = \frac{1}{3}, P(x_{s1} = 2) = \frac{1}{3}\}$ *et* $P_{x_{s2}} = \{P(x_{s2} = 0) = \frac{1}{3}, P(x_{s2} = 1) = \frac{1}{3}, P(x_{s2} = 2) = \frac{1}{3}\}$;

– $C = \{c_1, c_2, c_3\}$ *avec* $scp(c_1) = \{x_{d1}, x_{d2}\}$ *où* $c_1 : x_{d1} = x_{d2}$, $scp(c_2) = \{x_{d1}, x_{s1}\}$ *où* $c_2 : x_{d1} + x_{s1} > 1$ *et* $scp(c_3) = \{x_{d2}, x_{s2}\}$ *où* $c_2 : x_{d1} + x_{s2} > 1$ *;*
– $\theta = \frac{1}{3}$.

La figure 2 représente une politique pour le problème SCSP $\mathcal{P}$. *La politique représente la décision d'assigner la valeur* 1 *aux variables* x_{d1} *et* x_{d2}. *Dans cette politique, la contrainte* $c_1 : x_{d1} = x_{d2}$ *portant uniquement sur les variables de décisions est satisfaite. Ici, les neuf chemins entre chaque feuille et la racine représentent les neuf scénarios possibles. Par exemple, le scénario où la valeur* 0 *est assignée aux deux variables stochastiques* x_{s1} *et* x_{s2} *est le chemin entre la feuille la plus à gauche et la racine de chaque politique. Ce scénario ne satisfait pas les contraintes* $c_2 : x_{d1} + x_{s1} > 1$ *et* $c_3 : x_{d2} + x_{s2} > 1$, *la feuille correspondante est alors étiquetée par la valeur* 0. *Contrairement au scénario représenté par le chemin entre la feuille la plus à droite et la racine où la valeur* 2 *est assignée aux deux variables stochastiques qui est étiquetée par la valeur* 1.

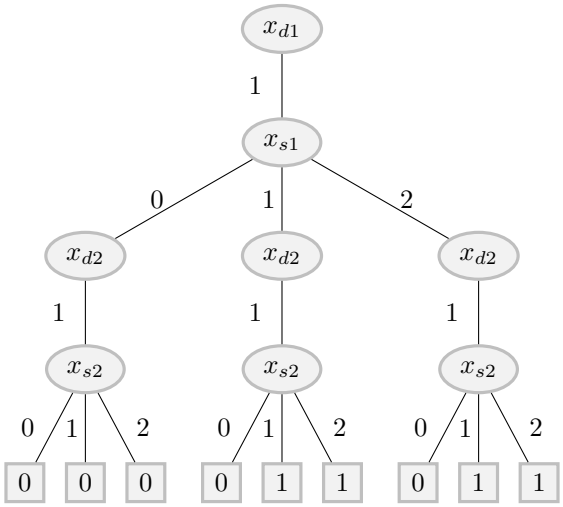

Figure 2. Une politique solution pour le SCSP $\mathcal{P}$

La satisfaction associée à chaque politique est égale à la somme de chacune des valeurs étiquetant les feuilles pondérées par la probabilité du scénario qui correspond. Dans cet exemple, la probabilité est uniforme pour chaque scénario, elle est correspond à $\frac{1}{3} \times \frac{1}{3} = \frac{1}{9}$. *Ainsi la satisfaction* s *de la politique est* $\frac{4}{9}$. *nous avons* $s > \theta$ *signifiant que la politique est une solution pour le problème* $\mathcal{P}$.

Il est possible de représenter un réseau de contraintes stochastiques par un m-SCSP (un SCSP à m niveaux) où chaque niveau représente un sous-ensemble de X nommé $X_i = V_i \cup Y_i$ tel que $X = \bigcup_{i=1}^{m} X_i$. Dit autrement $X = \{V_1, Y_1, V_2, Y_2, \ldots, V_m, Y_m\}$.

EXEMPLE 3. — *L'exemple 2 correspond à un* 2-SCSP *où* $X_1 = \{x_{d1}, x_{s1}\}$ *et* $X_2 = \{x_{d2}, x_{s2}\}$.

Notons que le SCSP à un seul niveau (1-SCSP) est nommé μSCSP (pour micro-SCSP) dans cet article. Un micro-SCSP est un SCSP $\mathcal{P} = (X, Y, D, C, P, \theta)$ dont l'ensemble X composant les n variables est ordonné strictement par l'ensemble des dn variables de décisions composant V puis par l'ensemble des sn variables stochastiques composant Y. Dit autrement, si $V = \{x_{d1}, x_{d2}, \ldots, x_{dn}\}$ et $Y = \{x_{s1}, x_{s2}, \ldots, x_{sn}\}$ alors $X = V \cup Y = \{x_{d1}, x_{d2}, \ldots, x_{dn}, x_{s1}, x_{s2}, \ldots, x_{sn}\}$.

Un μSCSP peut être satisfait si il existe un ensemble d'assignations pour les variables de décisions tel qu'il existe des scénarios possibles dont la somme des probabilités de chaque scénario est supérieure ou égale au seuil attendu.

Si on étend la résolution d'un μSCSP à un m-SCSP, on peut comprendre que la résolution d'un tel problème est de déterminer un ensemble d'assignations pour les variables de décisions composant le premier niveau (μSCSP_1) où soit un ensemble de valeurs aléatoires données pour les variables stochastiques, il est possible de déterminer un ensemble d'assignations pour les variables de décisions composant le second niveau (μSCSP_2), etc ... jusqu'au dernier niveau (μSCSP_m), tel qu'il existe des scénarios possibles dont la somme des probabilités de chaque scénario est supérieure ou égale au seuil θ attendu.

Il est important de préciser que les contraintes d'un SCSP porte sur l'ensemble des différents niveaux le composant, il n'est donc pas possible de décomposer l'ensemble des contraintes C comme il est possible de le faire pour l'ensemble X des variables.

4. WoodStock

Dans cette section, nous présentons WoodStock (*With Our Own Developper STOchastic Constraint toolKit*), le premier programme-joueur générique dirigé par les contraintes dans un tournoi GGP.

Comme vu en section 2.2, un programme-joueur doit pouvoir communiquer avec le *game manager* avant de s'intéresser au jeu lui-même. À chaque tour, pour obtenir l'état courant et communiquer les actions choisies de WoodStock, nous avons développé un joueur-réseau basé sur le protocole GCL dénommé *spy-ggp* [4]. Il utilise uniquement la librairie standard *Python* et n'a besoin que de l'interpréteur *Python3* pour fonctionner. Il est possible de lier à toute librairie dynamique (.so, .dll, .dylib, ... en accord avec le système utilisé) modélisant la partie stratégie du joueur. Cette librairie peut être écrite en tout autre langage (C, C++, C#, Go, OCaml, ...). Depuis sa première utilisation en compétition GGP, il s'est démontré robuste sans ne jamais provoquer d'erreur.

Le langage C++ a été choisi pour implémenter la partie stratégique de WoodStock afin de bénificier du paradigme proposé par la programmation orientée objet (POO) et de sa bonne capacité de calcul. L'utilisation de la POO permet d'obtenir une concep-

4. *spy-ggp*: https://github.com/syllag/spy-ggp disponible sous Licence GNULGPL3 / CeCILL-C.

tion élégante permettant de faciliter la maintenance et l'évolution du joueur au fil du temps.

La partie stratégique est composée de trois parties : (1) l'étape de traduction ; (2) la phase de résolution ; (3) la phase de simulation.

4.1. De GDL à SCSP

Au cours du temps de pré-traitement alloué (*start clock*), WoodStock réalise la traduction d'un programme GDL impliquant k joueurs en un SCSP dans le but que ce dernier puisse être utilisé le plus efficacement en compétition. Tout d'abord, suite à la réception du programme GDL, WoodStock génére un *template* dénommé μSCSP_t représentatif d'un tour quelconque du jeu GDL. Ce patron est une traduction du programme GDL sans y inclure les règles construites via le prédicat « init » modélisant l'état initial du jeu.

Dans un premier temps, nous appliquons la méthode de Lifschitz et Yang (Lifschitz, Yang, 2011) sur le programme GDL afin d'éliminer les fonctions et ainsi de simplifier la traduction. Cette méthode inclut deux étapes : l'*aplanissement* permettant de réduire les termes imbriqués, puis la phase d'*élimination* qui remplace ces fonctions par des prédicats. L'*aplanissement* est spécifiée comme suit : pour chaque symbole $\mathtt{f}$ qui apparaît dans P, nous construisons un équivalent $\mathtt{P_f}$, dénommé le $\mathtt{f}$-*aplanit*, dans lequel chaque occurrence d'un terme de la forme $\mathtt{f}(\mathtt{t_1}, \cdots, \mathtt{t_k})$ est transformé par l'égalité $\mathtt{f}(\mathtt{t_1}, \cdots, \mathtt{t_k}) = \mathtt{Z}$, où Z est une variable de renommage. Par exemple, si le programme GDL P contient $\mathtt{legal}(\mathtt{random}, \mathtt{roll}(\mathtt{c}))$, alors $\mathtt{P_f}$ est remplacé par la conjonction $\mathtt{legal}(\mathtt{random}, \mathtt{Z}), \mathtt{roll}(\mathtt{c}) = \mathtt{Z}$. Par l'aplanissement de toutes les fonctions f de P, plus aucun terme imbriqué n'apparaît. Notons que cette phase est polynomialement bornée par le nombre de fonctions et la profondeur des termes. L'étape d'élimination remplace chaque symbole de fonction f d'arité k par un symbole de relation F d'arité $k+1$. Ainsi, chaque égalité de la forme $\mathtt{f}(\mathtt{t_1}, \cdots, \mathtt{t_k}) = \mathtt{Z}$ est remplacée par l'atome $\mathtt{F}(\mathtt{t_1}, \cdots, \mathtt{t_k}, \mathtt{Z})$. Finalement, la règle $(\exists!\mathtt{Z})\mathtt{F}(\mathtt{t_1}, \cdots, \mathtt{t_k}, \mathtt{Z})$ est ajoutée au programme $\mathtt{P}'$ afin d'assurer l'équivalence des modèles entre P et $\mathtt{P}'$.

Par la suite, WoodStock génère chaque variable du programme SCSP: une variable $terminal$ modélisant si un état est terminal ou non, k variables $score_j$ afin de modéliser le score de chaque joueur j, k variables $action_j$ pour modéliser l'action de chaque joueur j et deux variables pour chaque fluent du jeu, respectivement une au temps courant t et une seconde au temps suivant $t+1$.

La prochaine étape est l'extraction des domaines. Le domaine de $terminal$ est booléen et le domaine de chaque variable $score_j$ est modélisé par la valeur entière apparaissant dans chaque règle « goal » correspondant à chaque joueur j. Finalement pour chaque variable $action_j$ et pour chaque variable modélisant un fluent au temps t ou $t+1$, WoodStock exploite les conditions d'une compétition GGP actuelle imposant l'utilisant des règles « input » et « base » composant un programme GDL. Ces derniers permettant de générer rapidement l'univers de Herbrand, il n'est pas néces-

saire de calculer l'ensemble des combinaisons des constantes pour obtenir le domaine de chaque variable « fluent ». Toutes les variables générées sont des variables de décisions sauf la variable correspondant à l'action de l'environnement *random* si celui-ci est utilisé. La distribution de probabilité associée à l'unique variable stochastique est uniforme sur l'ensemble des actions possibles de l'environnement composant son domaine. Enfin, le seuil θ est fixé à 1 pour que seuls les scénarios composés d'actions légales, soient conservés dans l'ensemble des solutions.

Tableau 4. Les règles de réécriture entre `GDL` *et les portées des contraintes*

Prédicat `GDL`	**Portée de la contrainte**
$true(f(\ldots))$	$\{f_t\} \in scp(C_t)$
$does(j, a(\ldots))$	$\{action_{j,t}\} \in scp(C_t)$
$legal(j, a(\ldots))$	$\{action_{j,t}\} \in scp(C_t)$
$next(f(\ldots))$	$\{f_{t+1}\} \in scp(C_t)$
$goal(j, N)$	$\{score_{j,t}\} \in scp(C_t)$
$terminal$	$\{terminal_t\} \in scp(C_t)$

La dernière étape est la génération de chaque contrainte. `WoodStock` commence par créer une contrainte pour chaque règle `GDL` en associant à chaque contrainte sa portée correspondante suite aux règles de réécriture détaillées dans le tableau 4. Quant aux relations des contraintes, elles sont restreintes uniquement aux termes présents dans chaque règle `GDL` correspondante. La présence des mot-clés « distinct » et « not » sont utilisés pour effectuer un premier filtrage de chaque relation. Les contraintes obtenues sont en extension et modélisées par des contraintes tables.

EXEMPLE 4. — *La figure 3 illustre un $\mu SCSP_t$ correspondant au programme* `GDL` *au temps t du « Matching Pennies », illustré dans l'exemple 1. Notons que les variables $action_{j1,t}$ et $action_{j2,t}$ sont assimilées respectivement à $retourne_{j1,t}$ et $retourne_{j2,t}$ pour des raisons de lisibilité afin de ne pas répéter le nom de l'action dans le domaine de chacune de ces variables. De même, la variable $terminal_t$ n'est pas exprimée dans les relations composant les contraintes goal afin de clarifier la lecture. Rappelons que si $terminal_t$ est faux alors quelque soit les valeurs des variables fluents $score_t$ est égale à 0. Les notations suivantes sont utilisées : P = $pile$; F = $face$; I = $inconnu$; V = $vrai$; W = $faux$.*

Premièrement, les trois variables $terminal_t$, $score_{j1,t}$ et $score_{j2,t}$ représentent respectivement la fin du jeu et les scores possibles de $j1$ et $j2$. Ces variables sont extraites des mots-clés $terminal$ et $goal(J, S)$. Le domaine associé à $terminal_t$ est booléen et celui des deux autres variables correspond aux différentes valeurs S possiblement impliquées dans $goal(J, S)$.

Les états du jeu au tour t et $t + 1$ sont représentés par l'ensemble des variables dérivées de par le mot-clé next. Le domaine de ces variables correspond aux valeurs possibles du fluent C utilisé dans l'exemple. Puis, on utilise le mot-clé legal pour générer les variables $retourne_{j1,t}$ et $retourne_{j2,t}$.

Une contrainte terminale relie la variable $terminal_t$ aux deux variables représentant les pièces. Le jeu se termine quand toutes les pièces ne sont plus assignées à la valeur « inconnue ». De la même manière, la contrainte goal relie les différents côtés de chacune des pièces aux scores associés à $j1$ et $j2$. Finalement les contraintes next permet au jeu de passer d'un état t à un autre état $t+1$ dépendant des actions choisies par les joueurs (variables $retourne_{j1,t}$ et $retourne_{j2_t}$).

Variable	Domaine
$\texttt{piece}_{j1,t}$	{F, P, I}
$\texttt{piece}_{j2,t}$	{F, P, I}
$\texttt{terminal}_t$	{V, W}
$\texttt{score}_{j1,t}$	{0, 50, 100}
$\texttt{score}_{j2,t}$	{0, 50, 100}
$\texttt{retourne}_{j1,t}$	{F, P}
$\texttt{retourne}_{j2,t}$	{F, P}
$\texttt{piece}_{j1,t+1}$	{F, P, I}
$\texttt{piece}_{j2,t+1}$	{F, P, I}

Variables et domaines

$\texttt{piece}_{\mathbf{j1},\mathbf{t}}$	$\texttt{pieces}_{\mathbf{j2},\mathbf{t}}$	$\texttt{terminal}_{\mathbf{t}}$
F	F	T
F	P	T
P	F	T
P	P	T
I	F	W
I	P	W
F	I	W
P	I	W

Contrainte *terminal*

$\texttt{piece}_{j1,t}$	$\texttt{retourne}_{j1,t}$
I	P
I	F
$\texttt{piece}_{j2,t}$	$\texttt{retourne}_{j2,t}$
I	P
I	F

Contraintes *legal*

$\texttt{piece}_{j1,t}$	$\texttt{retourne}_{j1,t}$	$\texttt{piece}_{j1,t+1}$
I	F	F
I	P	P
$\texttt{piece}_{j2,t}$	$\texttt{retourne}_{j2,t}$	$\texttt{piece}_{j2,t+1}$
I	F	F
I	P	P

Contraintes *next*

$\texttt{piece}_{j1,t}$	$\texttt{piece}_{j2,t}$	$\texttt{score}_{j1,t}$
F	F	100
P	P	100
F	P	0
P	F	0
$\texttt{piece}_{j1,t}$	$\texttt{piece}_{j2,t}$	$\texttt{score}_{j2,t}$
F	P	100
P	F	100
F	F	0
P	P	0

Contraintes *goal*

Figure 3. $\mu SCSP_t$ encodant le « Matching Pennies »

`WoodStock` utilise `XCSP 3.0`[5], un format basé sur `XML` pour représenter les instances obtenues dans le but de fournir de nouvelles instances `SCSP` et d'en extraire facilement la partie `CSP`.

5. `XCSP 3.0` : xcsp.org

4.2. MAC-UCB

Dans (Koriche *et al.*, 2016), la modélisation SCSP obtenue est identifiée dans un fragment SCSP où chaque contrainte ne porte que sur un seul et unique μSCSP composant le réseau. Ce fragment est exploité par WoodStock en utilisant la technique de résolution MAC-UCB.

Comme indiqué auparavant, le réseau de contraintes stochastiques d'un programme GDL est une séquence de μSCSPs, chacun associé à un tour de jeu. Pour chaque $\mu\text{SCSP}_t \in \{0, \ldots, T\}$, MAC-UCB recherche l'ensemble des politiques solutions en décomposant le problème en deux parties: un CSP classique et un μSCSP (plus petit que l'original). La première partie est résolue à l'aide de l'algorithme MAC (Sabin, Freuder, 1994) et la seconde partie grâce à l'algorithme SFC dédié au cadre SCSP (Walsh, 2009). Par la suite, une série d'échantillonnages avec borne de confiance est réalisée pour simuler l'utilité attendue de chaque politique solution de chaque μSCSP_t.

Suite à la génération du *template* μSCSP_t, quelques méthodes de pré-traitement ne demandant que très peu de temps y sont appliquées afin de rendre la phase de résolution plus efficace. Les contraintes possédant la même portée sont fusionnées en une seule contrainte en unifiant les relations. Puis, toutes les variables universelles, c'est à dire les variables dont toutes les valeurs sont comprises dans l'ensemble des solutions, sont supprimées.

Afin d'obtenir un μSCSP_t à un temps donné t, WoodStock utilise un injecteur. Au temps $t = 0$ représentant l'état initial, l'injecteur correspond à un ensemble de contraintes unaires générées à l'aide des règles « init ». À tout autre temps $t \neq 0$, l'injecteur est construit avec les solutions du micro SCSP correspondant à l'état à l'origine du nouvel état. Grâce à celui-ci, il est possible de projeter la relation de chaque contrainte unaire ajoutée sur le domaine de chaque variable apparaissant dans la portée sur l'ensemble du réseau de contrainte en le restreignant à la seule valeur autorisée par le tuple de la contrainte associée.

Le μSCSP obtenu est alors assez petit pour appliquer l'algorithme SAC (Debruyne, Bessière, 1997) sur ce dernier dans les délais imposées par une compétition GGP. Suite à cela, les valeurs inconsistantes sont supprimées du domaine des variables. C'est pourquoi, seules les actions légales représentées par les valeurs consistantes des variables $action_j$ de chaque joueur j sont conservées et garantissent la légalité des actions jouées par WoodStock au cours de l'ensemble du jeu.

4.2.1. Phase de résolution : MAC

Après la réalisation de ces techniques de pré-traitement, l'objectif de la phase de résolution est d'énumérer l'ensemble des politiques solutions du μSCSP, dont certaines peuvent mener à une solution optimale.

L'algorithme de *Forward Checking* (SFC) adapté à ce cadre (Walsh, 2009) utilisé seul n'est pas suffisant. En effet, dans le cadre d'un μSCSP impliquant un nombre

de contraintes important, SFC n'est pas assez efficace suite à sa faible capacité de filtrage. Par conséquent, l'algorithme MAC−UCB (Koriche *et al.*, 2015) une technique de résolution adaptée au SCSP à un niveau est utilisé par notre programme-joueur.

WoodStock sépare le μSCSP $\mathcal{P}$ en un CSP $\mathcal{P}'$ modélisant la partie dure composée uniquement des contraintes dures de $\mathcal{P}$ (voir algorithme 1) et en un μSCSP $\mathcal{P}''$ modélisation la partie chance contenant uniquement les contraintes de chances de $\mathcal{P}$ (voir algorithme 2). Les politiques solutions du μSCSP sont alors identifiées en combinant les solutions du CSP $\mathcal{P}'$ avec les solutions du μSCSP $\mathcal{P}''$.

Algorithme 1 : *partie_dure*

Données : μSCSP $\mathcal{P} = (X, Y, D, C, P, \theta)$
Résultat : CSP $\mathcal{P}' = (X', D', C')$
$\mathcal{P}' = (X' \leftarrow \emptyset, D' \leftarrow \emptyset, C' \leftarrow \emptyset)$
pour $c \in C$ **faire**
  si $\nexists x_s \in scp(c) \mid x_s \in Y$ **alors**
    $C' \leftarrow C' \cup \{c\}$
    pour $x_d \in scp(c)$ **faire**
      $X' \leftarrow X' \cup \{x_d\}$
      $D' \leftarrow D' \cup \{dom(x_d)\}$
    fin
  fin
fin
retourner $\mathcal{P}'$

Algorithme 2 : *partie_chance*

Données : μSCSP $\mathcal{P} = (X, Y, D, C, P, \theta)$
Résultat : μSCSP $\mathcal{P}'' = (X'', Y'', D'', C'', P'', \theta)$
μSCSP $\mathcal{P}'' = (X'' \leftarrow \emptyset, Y'' \leftarrow \emptyset, D'' \leftarrow \emptyset, C'' \leftarrow \emptyset, P'' \leftarrow \emptyset, \theta)$
pour $c \in C$ **faire**
  si $\exists x_s \in scp(c) \mid x_s \in Y$ **alors**
    $C'' \leftarrow C'' \cup \{c\}$
    pour $x \in scp(c)$ **faire**
      si $x \in Y$ **alors**
        $Y'' \leftarrow Y'' \cup \{x\}$
        $P'' \leftarrow P'' \cup \{P_x\}$
      fin
      sinon
        $X'' \leftarrow X'' \cup \{x\}$
      fin
      $D'' \leftarrow D'' \cup \{dom(x)\}$
    fin
  fin
fin
retourner $\mathcal{P}''$

Tout d'abord, examinons la résolution de $\mathcal{P}''$. Pour des instances GDL décrivant des jeux à information imparfaite, $\mathcal{P}''$ inclut au plus une unique contrainte (stochastique) capturant la règle de transition impliquée par le joueur environnement (*random*). Par

conséquent, $\mathcal{P}''$ peut être facilement résolu par SFC adapté au cadre des SCSP à un niveau. Par la suite, l'ensemble de politiques solutions obtenues par SFC sur $\mathcal{P}''$ est encodé par une contrainte dure c_f, dénommée contrainte de faisabilité, où $scp(c_f)$ contient l'ensemble des variables de décisions de $\mathcal{P}''$ et $rel(c_f)$ est l'ensemble des tuples correspondant aux assignations des variables de décisions qui font parties d'au moins une politique solution de $\mathcal{P}''$. Formellement, soit $sols(\mathcal{P}'')$ l'ensemble des politiques solutions de P'' alors :

- $scp(c_f) = vars(\pi)$ tel que $\pi \in sols(\mathcal{P}'')$;
- $rel(c_f) = (I_0, \ldots, I_i, \ldots, I_q)$ tel que $I_i \models \pi_i \in sols(\mathcal{P}'')$ avec $|sol(\mathcal{P}'')| = q$.

Désormais, intéressons nous à la résolution du CSP $\mathcal{P}'$. Ici, l'algorithme classique MAC (Sabin, Freuder, 1994) est utilisé pour énumérer toutes les solutions de $\mathcal{P}'$. Dans le but de tenir compte des solutions identifiées dans $\mathcal{P}''$, nous ajoutons simplement la contrainte de faisabilité c_f aux contraintes C' de $\mathcal{P}'$. Les solutions obtenues par MAC sur $\mathcal{P}' = (X', D', C' \cup \{c_f\})$ sont identiques à l'ensemble des solutions du μSCSP $\mathcal{P}$ original. MAC exploite la propriété de consistance d'arc avec pour objectif de filtrer efficacement les politiques non satisfaisantes du CSP $\mathcal{P}'$. La stratégie de recherche employée au travers de MAC par WoodStock est STR (*Simple Tabular Reduction*) (Ullmann, 2007) en associant à chaque contrainte, les quatre structures nécessaires à son implémentation. Enfin, l'heuristique *dom/ddeg* est utilisée afin de choisir les variables à résoudre par ordre croissant selon le ratio entre la taille du domaine courant et le degré dynamique des variables.

4.2.2. Phase de simulation : UCB

Rappelons dans un premier temps que tout programme GDL est un jeu séquentiel fini et que les utilités (les scores) de chaque joueur ne sont accessibles que dans un état terminal. De plus généralement, les instances de jeu utilisées en pratique impliquent un arbre de jeu possédant une profondeur et une largeur importante. MAC, à lui seul, ne peut donc pas résoudre l'ensemble de l'arbre de jeu en accord avec le temps délimité dans le cadre d'un match GGP.

C'est pourquoi, dans le but de choisir le plus judicieusement possible à chaque itération le prochain micro SCSP à résoudre, il est nécessaire de simuler les prochains états de l'arbre de jeu de tout état non-terminal afin d'estimer l'utilité des solutions qu'apporte MAC à chaque μSCSP_t. Dans ce but, nous utilisons une technique de bandits multi-bras dirigée par la propriété UCB (pour *Upper Confidence Bound*) (Browne *et al.*, 2012) en considérant chaque politique solution du μSCSP_t comme un « bras ». À partir d'une politique partielle du SCSP équivalent au jeu GDL associée à une politique solution du μSCSP_t, nous simulons uniformément et aléatoirement tous les coups possibles de $t+1$ à $T-1$.

Le principal problème est d'atteindre rapidement un état terminal sans résoudre chaque μSCSP rencontré. Heureusement, grâce à la propriété SAC que nous appliquons sur chaque μSCSP_t à chaque temps t, nous obtenons les coups légaux à chaque tour directement. Par conséquent, WoodStock peut choisir aléatoirement une sé-

quence d'actions jusqu'à atteindre un état terminal identifié par l'assignation de la valeur $true$ à la variable $terminal_t$.

La « meilleure » solution du μSCSP$_t$ est celle qui maximise $\bar{u}_i + \sqrt{\frac{2 \ln n}{n_i}}$, où $\bar{u}_i$ est le score moyen de la politique solution i, n_i est le nombre de fois où i a été simulé jusque là, et n est le nombre total de simulations réalisées. La résolution du prochain problème est décidée en instanciant μSCSP$_{t+1}$ par les valeurs de la meilleure politique solution estimée à partir de μSCSP$_t$.

Afin d'améliorer l'efficacité des techniques de filtrage de WoodStock une table de hachage de Zobrist (Zobrist, 1990) est utilisée pour stocker tous les états déjà explorés. En combinant les état terminaux déjà atteints avec le nombre de coups légaux de chaque état, il est possible de déterminer si un sous-arbre a été complètement exploré mais également d'éviter de simuler deux fois une même succession de coups légaux.

Finalement, le mouvement sélectionné par WoodStock correspond à l'action proposant la meilleure utilité espérée dans l'état courant. Notons que pour envisager tout problème de communication pouvant provoquer l'apparition d'un *timeout*, 2 secondes sont conservées pour sélectionner l'action et garantir son envoi au *game manager*.

L'algorithme 3 détaille le déroulement de MAC−UCB. Le *template* μSCSP$_t$ correspond au réseau de contraintes stochastiques à un niveau issue de la traduction du jeu GDL correspondant. Le second paramètre μSCSPs est une liste de réseaux stochastiques à un niveau (μSCSP$_i$) ordonnée de manière décroissante en fonction de l'utilité attendue (v_i) de chaque réseau. À l'état initial ($t = 0$), μSCSPs n'est composée que d'un seul réseau de contraintes stochastiques à un niveau correspondant au *template* μSCSP$_t$ et où les contraintes modélisant les règles « init » du jeu GDL correspondant sont ajoutées. Toutefois lors de l'appel de MAC−UCB à d'autres temps $t \neq 0$, μSCSPs ne contient que les réseaux de contraintes stochastiques à un niveau de temps supérieur à t associés à leurs utilités calculées précédemment.

MAC−UCB résout itérativement chaque μSCSP de la liste μSCSPs tant qu'il reste du temps (ligne 1). À chaque itération, le réseau $\mathcal{P}$ choisi pour être résolu est le premier élément de la liste μSCSPs correspondant au réseau associé à la plus grande utilité attendue et il est supprimé de la liste μSCSPs (ligne 2).

La résolution de $\mathcal{P}$ commence par la génération du CSP $\mathcal{P}'$ correspondant à la partie dure (ligne 3) et par la génération du μSCSP $\mathcal{P}''$ représentant la partie chance (ligne 4) à l'aide des algorithmes 1 et 2. $\mathcal{P}''$ est résolu par l'algorithme de *Forward Checking* adapté au cadre SCSP (ligne 5). Les lignes 6 à 13 illustrent la génération de la contrainte de faisabilité c_f à l'aide des solutions ($sols(\mathcal{P}'')$) du réseau $\mathcal{P}''$. c_f est ajoutée aux contraintes du réseau P' (ligne 14) et il est résolu par l'algorithme MAC (ligne 15).

Les lignes 16 à 25 permettent de générer et d'ajouter les réseaux $muSCSP_{t+1}$ à la liste μSCSPs afin de les résoudre aux prochaines itérations. La génération de chaque réseau au temps $t + 1$ est effectuée à partir de toute solution obtenue qui ne

correspond pas à un état terminal (ligne 17) détecté à l'aide de la variable $terminal$. Le réseau $\mu\texttt{SCSP}_i$ généré pour chaque solution τ_i est initialisé à l'aide du *template* $\mu\texttt{SCSP}_t$ (ligne 18). Puis pour chaque instanciation d'une variable f_{t+1} (correspondant au fluent au temps suivant $t+1$) du tuple solution τ_i, nous ajoutons une contrainte unaire restreignant le domaine de chaque variable f_t (correspondant au même fluent au temps t) du nouveau réseau $\mu\texttt{SCSP}_i$ au singleton $\{\tau_i[f_{t+1}]\}$ (lignes 19 et 20). L'utilité attendue v_i correspondant au réseau $\mu\texttt{SCSP}_i$ est générée à l'aide de la technique UCB (ligne 22). Finalement, le couple $(\mu\texttt{SCSP}_i, v_i)$ est inséré correctement dans la liste $\mu\texttt{SCSP}s$ en fonction de son utilité v_i (ligne 23).

Algorithme 3 : MAC−UCB

Données : Réseau *template* $\mu\texttt{SCSP}_t = (X_t, Y_t, D_t, C_t, P_t, 1)$, Liste ordonnée de réseaux stochastiques $\mu\texttt{SCSP}s = \{(\mu\texttt{SCSP}_0, v_0), \ldots, (\mu\texttt{SCSP}_n, v_n)\}$ avec $v_0 \geq \ldots \geq v_n$

1 **tant que** ¬ *timeout* **faire**
2 Sélectionner et supprimer le premier réseau $\mathcal{P}$ de $\mu\texttt{SCSP}s$
3 CSP $\mathcal{P}' = (X', D', C') \leftarrow partie\_dure(\mathcal{P})$
4 μSCSP $\mathcal{P}'' \leftarrow partie\_chance(\mathcal{P})$
5 $sols(\mathcal{P}'') \leftarrow \texttt{SFC}(\mathcal{P}'')$
6 Contrainte c_f = $(scp(c_f) \leftarrow \emptyset, rel(c_f) \leftarrow \emptyset)$
7 $scp(c_f) \leftarrow vars(\pi) \mid \pi \in sols(\mathcal{P}'')$
8 **pour chaque** $\pi_k \in sols(\mathcal{P}'')$ **faire**
9 **pour chaque** $(x = v) \in \pi_k \mid x \in V_t''$ **faire**
10 $\tau_k[x] \leftarrow v$
11 **fin**
12 $rel(c_f) \leftarrow rel(c_f) \cup \tau_k$
13 **fin**
14 $\mathcal{P}' = (X', D', C' \cup \{c_f\})$
15 $sols(\mathcal{P}') \leftarrow \texttt{MAC}(\mathcal{P}')$
16 **pour chaque** $\tau_i \in sols(\mathcal{P}')$ **faire**
17 **si** $\tau_i[terminal] = 0$ **alors**
18 $\mu\texttt{SCSP}_i = (X_i \leftarrow X_t, Y_i \leftarrow Y_t, D_i \leftarrow D_t, C_i \leftarrow C_t, P_i \leftarrow P_t, 1)$
19 **pour chaque** $\tau_i[f_{t+1}] \mid f_{t+1} \in X'$ **faire**
20 $C_i \leftarrow C_i \cup \{(f_t = \tau_i[f_{t+1}])\} \mid f_t \in X_i$
21 **fin**
22 $v_i \leftarrow \texttt{UCB}(\mu\texttt{SCSP}_i)$
23 $\mu\texttt{SCSP}s \leftarrow \mu\texttt{SCSP}s \cup \{(\mu\texttt{SCSP}_i, v_i)\}$ // dans l'ordre de $\mu\texttt{SCSP}s$
24 **fin**
25 **fin**
26 **fin**

Au cours de la section suivante, nous présentons les résultats de WoodStock au cours de sa première participation à une compétition GGP mais également au cours de la compétition en continue que propose le serveur *Tiltyard*.

5. Résultats compétitifs

Actuellement en compétition, `WoodStock` fonctionne sur un Intel Core i7-4770L CPU 3.50 Ghz associé à 32 Gb de RAM sous Linux. La première participation de `WoodStock` fut lors de la compétition `GGP` *Tiltyard Open* en décembre 2015 organisée par *Sam Schreiber* et *Alex Landau* sur le site Tiltyard [6].

5.1. Tiltyard Open 2015

5.1.1. Contexte

Cette compétition est réalisée en deux étapes. La première représente la phase de qualification où le temps de pré-traitement est fixé à 120 secondes et la temps de délibération à 15 secondes, la seconde étape représentant la finale propose un temps de pré-traitement de 180 secondes et le même temps de délibération.

Les participants de la phase de qualification jouent sur 10 nouveaux jeux au cours d'un tournoi suisse; la plupart d'entre eux sont réalisés plusieurs fois. Ils incluent des jeux à 1, 2, ou 4 joueurs et plusieurs d'entre eux sont simultanés.

Les programmes-joueurs participent automatiquement à chaque match sur la base des résultats de ces précédents matchs. Premièrement, les matchs sont réalisés entre joueurs possédant un score courant similaire sur le jeu courant et deuxièmement entre joueurs possédant un score courant similaire sur l'ensemble de la compétition. À cela s'ajoute un facteur permettant de réduire les matchs répétés entre les mêmes joueurs.

La table 5 indique les informations disponibles sur les 10 jeux proposés lors de la phase de qualification associée à un identifiant unique afin de conserver leurs anonymats. 1 à 4 joueurs sont impliqués dans le nombre de matchs prédéterminés pour chaque jeu. Chaque programme-joueur victorieux obtient un nombre de points égal au score qu'il a obtenu à l'issu du match pondéré par différents poids associés à chaque jeu.

Nous pouvons noter que la phase de qualification avantage les jeux à deux joueurs et que le score maximal pouvant être atteint est de $2,500$.

La finale inclut les quatre programmes-joueurs obtenant les meilleurs scores lors des qualifications et propose un système d'élimination sur plusieurs matchs réalisés par tour.

5.1.2. Résultats de `WoodStock` à l'issue de la compétition

Neuf programmes listés dans la table 6 ont participé à la *Tiltyard* Open 2015. Trois d'entre eux, fonctionnant avec un réseau de propositions (*propnet*) et des méthodes Monte Carlo, sont les derniers champions `GGP` de ces dernières années. La même

6. www.ggp.org/view/tiltyard/matches/\#tiltyard_open_20151204_2

Tableau 5. Les informations disponibles pour chaque jeu de la phase de qualification

Id	#Joueurs	#Matchs	Poids
1	1	1	2
2	2	6	0.5
3	1	2	1
4	2	6	0.5
5	1	1	2
6	2	6	0.5
7	4	4	0.5
8	2	3	1
9	4	4	0.5
10	2	3	1

table montre le score obtenu par chaque programme-joueur générique à l'issue de la phase de qualification.

Au cours de cette étape, `WoodStock` finit second avec un score (1366.250 points) plus élevé que les derniers champions `GGP`. Notons que `Galvanise` et `Sancho`, les deux derniers champions ne se sont pas qualifiés car ils ont perdu de nombreux points contre `WoodStock` et `LeJoueur`. `LeJoueur` (Méhat, Cazenave, 2011) est une implémentation en parallèle de l'ancien champion `GGP Ary` permettant de réaliser de nombreuses simulations. Ce dernier a obtenu la première place de la qualification en utilisant $3,000$ simulateurs et fonctionnant sur un cluster composé de 48 threads. Notons que pour le moment, `WoodStock` fonctionne sur un seul thread, par conséquent, les deux infrastructures ne sont pas similaires et explique pourquoi `LeJoueur` a obtenu un meilleur score.

Tableau 6. Les scores finaux suite à la phase de qualification

Rang	GGP player	Score final
1	LeJoueur	1783.750
2	WoodStock	1366.250
3	GreenShell [7]	1351.000
4	QFWFQ	1245.875
5	Sancho	1213.500
6	SteadyEddie	1201.875
7	Galvanise	1185.250
8	Modest	1129.000
9	MonkSaki	1037.813

Pour sa première participation, WoodStock s'est qualifié pour la finale au côté de LeJoueur, GreenShell et QFWFQ. Malheureusement, au cours de cette phase, WoodStock a rencontré une erreur sur le premier jeu GDL et il était dans l'impossibilité d'envoyer une action au *game manager*. Cette erreur était due à un problème dans son implémentation. Par conséquent, ces mouvements ont été remplacés par des actions aléatoires et il a perdu les deux premiers matchs. Sur les trois matchs suivants, WoodStock n'a rencontré aucun problème et il les a remportés pour finalement obtenir la troisième position de la compétition GGP. Le tableau 7 indique les rangs finaux.

Tableau 7. Le classement final de la Tiltyard Open 2015

Rang	GGP player
1	LeJoueur
2	GreenShell
3	WoodStock
4	QFWFQ

5.2. Tournoi continu GGP

Tableau 8. Le classement au 15 juillet 2016 du tournoi continu GGP

Rang	GGP player	Score Agon
1	WoodStock	225.4
2	Sancho	220.63
3	Galvanise	214.4
4	LabThree	191.71
5	Coursera Quixote	169.77
6	SgianDubh	163.92
7	CloudKingdom	162.23
8	SteadyEddie	159.53
9	Alloy	133.5
10	LeJoueur	120.26

Suite à sa première participation en compétition GGP, l'erreur d'implémentation rencontrée par WoodStock a été corrigée et il a pu participer à la compétition continue de *General Game Playing* où l'ensemble des programmes-joueurs sont représentés (à ce jour, plus de $1,000$ programmes) jouant sur l'ensemble des jeux que propose le serveur Tiltyard. Ce type de compétition est naturellement plus complexe car le temps de pré-traitement et le temps de délibération sont choisis aléatoirement entre

7. Il est nécessaire de mentionner que GreenShell est le nouveau pseudonyme utilisé par TurboTurtle, le champion 2011 et 2013 de GGP.

deux matchs et aucun programme-joueur ne peut donc se calibrer en fonction de ce paramètre.

Depuis mars 2016 et après près de $2,000$ matchs, `WoodStock` est le leader du tournoi et a détrôné `Sancho` qui était leader de cette compétition depuis plus de trois ans. À ce jour, `WoodStock` obtient un score moyen supérieur à 76 et le tableau 8 indique le classement du 15 juillet 2016 du tournoi continu `GGP`.

Le classement actuel et les matchs de `WoodStock` sont accessibles en direct à l'adresse suivante : www.ggp.org/view/tiltyard/players/WoodStock. Il est notamment possible de retrouver le score moyen obtenu par `WoodStock` sur chaque jeu `GDL`.

6. Optimisation

Conceptuellement, tout jeu (déterministe ou stochastique) à horizon fini impliquant deux joueurs et à somme nulle possède une fonction optimale spécifiant le résultat attendu d'un jeu, pour tout état possible, sous la condition que l'ensemble des joueurs jouent parfaitement.

En théorie de tels jeux peuvent être résolus en calculant récursivement la fonction d'utilité dans un arbre de recherche de taille l^d, où l est la largeur du jeu (le nombre d'actions légales par état) et d est la profondeur du jeu (le nombre de mouvements nécessaires pour atteindre un état terminal). Pourtant, même pour des jeux d'une taille modérée, une recherche exhaustive n'est pas envisageable en pratique lors des tournois de *General Game Playing*, dû au temps de délibération très court imposé à chaque joueur pour déterminer sa prochaine action.

C'est pour cette raison, que l'un des challenges de `GGP` est de concevoir des techniques génériques d'inférence et d'apprentissage pour réduire efficacement l'espace de recherche des jeux, dont les règles ne sont fournies qu'au début du jeu.

À cette fin, la détection de symétries est une approche d'inférence bien connue en Intelligence Artificielle pour accroître la résolution de tels problèmes combinatoires, en transférant les connaissances apprises en des régions équivalentes de l'espace de recherche. Les jeux représentent un exemple notable car ils impliquent typiquement de nombreux états équivalents et des actions équivalentes. De telles similarités peuvent être exploitées pour transférer la fonction d'utilité entre plusieurs nœuds de l'arbre de recherche : la largeur l de l'arbre peut être réduite en exploitant les actions menant à des résultats attendus similaires et la profondeur d de l'arbre peut également être réduite en reconnaissant des états associés à des valeurs similaires.

Ainsi la plupart des méthodes de détections de symétries sont réalisées en utilisant des algorithmes basés sur des automorphismes de graphes (Darga *et al.*, 2008 ; McKay, Piperno, 2014). Le composant principal pour mettre en évidence les symétries de jeux est une structure graphique sur laquelle un groupe de permutation est induit.

Rappelons qu'une symétrie dans un ensemble D est une permutation sur D ou, de manière équivalente une bijection σ de D vers D. De nombreuses types de symétries ont été proposées dans la littérature de la programmation par contraintes, notamment les symétries de solutions qui préservent l'ensemble des solutions et les symétries de contraintes qui préservent l'ensemble des contraintes. (Cohen *et al.*, 2006) montre que chaque symétrie de contrainte correspond à un automorphisme de la micro-structure complémentaire du réseau de contraintes et que chaque symétrie de solution correspond à un automorphisme de la micro-structure complémentaire enrichie par l'ensemble des instanciations globalement incohérentes.

Afin de détecter les symétries, `WoodStock` doit générer la micro-structure complémentaire $\mathcal{H}_t$ de chaque $\mu\texttt{SCSP}_t$. Pour cela, il commence par générer la micro-structure complémentaire commune à chaque $\mu\texttt{SCSP}_t$ issue de la traduction du programme `GDL` à tout temps t (sans les règles *init*). De ce fait pour obtenir $\mathcal{H}_t$ à un temps t donné, il suffit pour le temps $t = 0$ d'ajouter les arêtes correspondantes aux contraintes issues des règles *init* et pour $t \neq 0$ les arêtes correspondantes aux contraintes générées à partir des solutions de $\mu\texttt{SCSP}_{t-1}$. Les micro-structures obtenues permettent de capturer graphiquement les règles d'un jeu `GDL` à chaque temps t. `WoodStock` représente chaque micro-structure complémentaire par un hypergraphe pouvant aussi être considéré comme un graphe bipartie : La première partie représentant l'ensemble des littéraux (couples variables-valeurs) et la seconde partie modélisant l'ensemble des tuples interdits composants toutes les contraintes de $\mathcal{P}$. Une arête (l, τ) de ce graphe correspond à l'occurrence d'un littéral l dans un tuple τ d'une contrainte.

Notons que pour chaque $\mu\texttt{SCSP}_t$, l'ensemble des contraintes est enrichi de nouvelles contraintes modélisant les stratégies. Ainsi, chaque $\mu\texttt{SCSP}_t$ $\mathcal{P}$ est enrichi d'une contrainte de portée $(\{F_t\}, \{A_t\}, A_{k+1,t})$ (où F_t est l'ensemble des fluents à l'instant t, A_t l'ensemble des actions des k joueurs et $A_{k+1,t}$ les actions du joueur environnement $k+1$) et de relation formée par l'ensemble des tuples de la forme $(s, \boldsymbol{a})$, s étant défini sur $\{F_t\}$ et $\boldsymbol{a}$ représentant une combinaisons d'actions sur $\{A_t\}$ et $A_{k+1,t}$. Chaque tuple $(s, \boldsymbol{a})$ est vu comme une instanciation globalement incohérente si toutes les politiques débutant à l'état s avec le vecteur d'action $\boldsymbol{a}$ est prédite par `UCB` comme une politique non solution. Dit autrement, l'utilité attendue simulée par `UCB` est inférieure au seuil θ.

Sur la base de chaque $\mu\texttt{SCSP}$ $\mathcal{P}$ enrichi de toute instanciation globalement incohérente détectée au fil de la résolution du problème, `WoodStock` déduit les automorphismes de la micro-structure complémentaire de $\mathcal{P}$ à l'aide de `NAUTY` (McKay, Piperno, 2014) et génère les $\mu\texttt{SCSP}$s symétriques. Chacun de ces $\mu\texttt{SCSP}$ est stocké et associé à leur utilité attendue dans une table de hachage de *Zobrist* (Zobrist, 1990) où chaque valeur assignée dans chaque $\mu\texttt{SCSP}$ correspond à une valeur de hachage générée aléatoirement. Le nombre de *Zobrist* alors utilisé pour retrouver un $\mu\texttt{SCSP}$ dans cette table est alors le `XOR` des nombres aléatoires associés à chaque couple variable-valeur assignée garantissant une probabilité de collision très faible. Ainsi avant de résoudre un $\mu\texttt{SCSP}$, `WoodStock` vérifie si il ne correspond pas à un $\mu\texttt{SCSP}$ symé-

trique déjà rencontré. Si c'est le cas, l'utilité attendue associée est directement obtenue sans avoir besoin de le résoudre ou de simuler les état suivants permettant ainsi un gain de temps substantiel.

6.1. WoodStock, Champion GGP 2016

Lors de la dernière compétition internationale organisée par l'université de *Stanford* (IGGPC'16), nous avons pu mettre en pratique WoodStock optimisé par la détection de symétries. La compétition a regroupé 10 concurrents comprenant les champions GGP 2011, 2013, 2014 et 2015 : Alloy, DROP-TABLE-TEAMS, Galvanise, General, MaastPlayer, QFWFQ, Sancho, SteadyEddie, TurboTurtle et WoodStock. Ils se sont affrontés au travers de nombreux jeux à un ou deux joueurs au cours de deux journées. Tous les jeux à deux joueurs étaient garantis à somme nulle. La plupart des jeux GDL étaient nouveaux et mis en pratique pour la première fois en compétition. 120 secondes de temps de pré-traitement (*StartClock*) et 30 secondes de temps de délibération (*PlayClock*) ont été allouées.

6.1.1. Jour 1

Lors de la première journée, deux groupes regroupant chacun 5 joueurs ont été réalisés arbitrairement. Le classement final de chaque groupe est obtenu à la suite du calcul de la somme pondérée des scores de chaque joueur. Les deux joueurs obtenant les scores maximaux sont qualifiés au sein de la « poule des vainqueurs » et les deux suivants au sein de la « poule des vaincus » pour la seconde journée. Les scores obtenus par WoodStock sur chaque jeu à l'issue de la première journée sont présentés dans le tableau 9. WoodStock a obtenu 1.200 points suite à cette première journée et il fut le seul programme-joueur à obtenir le score maximum sur l'ensemble des puzzles. Ainsi, il fut qualifié au sein de la « poule des vainqueurs » pour la seconde journée au coté de DROP-TABLE-TEAMS, Galvanise et TurboTurtle.

6.1.2. Jour 2

La seconde journée s'est déroulée sous la forme de duels remportés par le premier programme obtenant 3 victoires (excepté pour la finale où 4 victoires sont nécessaires). Le tableau 10 répertorie l'ensemble des matchs joués par WoodStock. Au cours des quarts de finale, WoodStock a affronté DROP-TABLE-TEAMS et a obtenu 3 victoires et 2 défaites lui permettant d'atteindre les demi-finales face à TurboTurtle. Au cours de ces dernières, WoodStock n'a rencontré aucune défaite et a obtenu 3 victoires. Finalement, lors de la finale l'opposant à Galvanise, WoodStock est devenu Champion GGP à la suite de 2 défaites et de 4 victoires.

Un point important à noter est que la détection de symétries a été primordiale au cours de la compétition, en particulier lors du dernier jeu pratiqué par WoodStock. Lors des deux matchs de Jointconnectfour, un grand nombre de symétries détectées ont permis de réduire fortement l'espace de recherche et d'explorer complète-

ment l'arbre de recherche au bout d'une trentaine de coups joués contrairement à son opposant.

Tableau 9. `WoodStock`*-* `IGGPC` *2016 - Jour 1.*

Jeu	#Joueurs	`WoodStock`	Poids
`Vectorracer6`	1	100	1
`EightPuzzle`	1	100	1
`HunterBig`	1	100	1
`UntwistyComplex`	1	100	0.5
`Jointbuttonsandlights25`	1	100	0.5
`Sudoku`	1	100	0.5
`Bandl3`	1	100	0.33
`bandl7`	1	100	0.33
`bandl10`	1	100	0.33
`Majorities`	2	100 / 300	1
`Battleofnumbers`	2	150 / 300	1
`Jointconnectfour77`	2	100 / 300	1
`Breakthrough`	2	300 / 300	1

Tableau 10. `WoodStock`*-* `IGGPC` *2016 - Jour 2.*

Jeu	`WoodStock`	Adversaire
Quart de finale contre `DROP-TABLE-TEAMS`		
`Hexawesome`	100	0
`Reflectconnect6`	30	70
`Connectfour77`	100	0
`Connectfour77`	0	100
`Majorities`	100	0
Demi finale contre `TurboTurtle`		
`Battleofnumbersbig`	100	0
`Platformjumpers`	100	0
`Jointconnectfour77`	100	0
Finale contre `Galvanise`		
`Reversi`	0	100
`Reversi`	0	100
`Skirmish`	100	0
`Skirmish`	100	0
`Jointconnectfour`	100	0
`Jointconnectfour`	100	0

7. Conclusion

Dans cet article, nous décrivons un nouveau type de programme-joueur générique basé sur la programmation par contraintes stochastiques illustrée par WoodStock au travers d'un programme orienté objet. Pour réaliser la communication entre un programme-joueur GGP et le serveur de jeux, nous fournissons un joueur réseau GGP dénommé spy-ggp adaptable à différents langages de programmation. WoodStock permet de mettre en pratique MAC-UCB dans le contexte d'une compétition GGP, tout d'abord en implémentant le processus de traduction d'un programme GDL vers un SCSP puis en réalisation les phases de résolution et de simulation nécessaires.

À l'aide de cet algorithme, notre programme-joueur mono-thread, WoodStock est actuellement leader de la compétition continue sur le serveur *Tiltyard* et suite à la compétition GGP dénommée *Open Tiltyard* 2015, il s'est classé second lors des qualifications avec un score plus élevé que trois des anciens champions GGP et troisième en finale derrière LeJoueur, un programme-joueur multi-threads.

Par la suite, nous avons optimisé WoodStock en ajoutant à son implémentation la détection de symétries issue de la programmation par contraintes et permettant de réduire l'espace de recherche en évitant d'explorer des actions et des états dont l'utilité attendue est directement déductible des symétries détectées. Ainsi, avec cette optimisation, WoodStock a participé pour la première fois à la compétition internationale de *General Game Playing* 2016 organisée par l'université de *Stanford*. À la suite de celle-ci, WoodStock a remporté la compétition et il est devenu le champion GGP 2016.

Améliorations et perspectives

L'objectif de ce papier est de confirmer l'efficacité d'un programme-joueur générique dirigé par les contraintes. Cependant, WoodStock implémente ce formalisme au travers d'un programme orienté objet permettant de l'améliorer continuellement en ajoutant différents modules au programme.

Par exemple, WoodStock utilise STR pour réaliser la stratégie de recherche de l'algorithme mais (Lecoutre *et al*., 2015) fournit un nouvel algorithme beaucoup plus performant pour la résolution d'instances de grande taille, pouvant ainsi être intéressant à implémenter dans le cadre des compétitions GGP. De même, l'utilisation d'UCB pour réaliser les simulations pourrait être remplacée par une technique de bandits plus adaptée au cadre GGP. Une autre voie naturelle serait de paralléliser les phases de résolution et de simulation afin d'être compétitif avec les programmes-joueurs multi-threads.

Finalement, le modèle stochastique utilisé pourrait être étendu au cadre des jeux à information incomplète décrit en GDL-II en intégrant la traduction du mot-clé *sees* permettant de décrire les observations de chaque joueur en fonction de l'état courant. Une étude en ce sens est en cours ainsi que l'implémentation d'un algorithme adapté à cette classe de jeux.

Bibliographie

Browne C., Powley E., Tavener S. (2012). A survey of monte carlo tree search methods. *Intelligence and AI*, vol. 4, n° 1, p. 1–49.

Cazenave T., Mehat J. (2010). Ary - a general game playing program. In *Proc. of board games studies colloquium.*

Clune J. (2007). Heuristic evaluation functions for general game playing. In *Proc. of AAAI'07*, p. 1134–1139.

Cohen D. A., Jeavons P., Jefferson C., Petrie K. E., Smith B. M. (2006). Symmetry definitions for constraint satisfaction problems. *Constraints*, vol. 11, n° 2-3, p. 115–137.

Cox E., Schkufza E., Madsen R., Genesereth M. (2009). Factoring general games using propositional automata. In *Ijcai'09 workshop on general game playing (giga'09)*, p. 13-20.

Darga P. T., Sakallah K. A., Markov I. L. (2008). Faster symmetry discovery using sparsity of symmetries. In *Dac'08*, p. 149–154. ACM.

Debruyne R., Bessière C. (1997). Some practicable filtering techniques for the constraint satisfaction problem. In *Ijcai'97*, p. 412–417.

Draper S., Rose A. (2014). *Sancho.* http://sanchoggp.blogspot.fr/2014/07/sancho-is-ggp-champion-2014.html.

Emslie R. (2015). *Galvanise.* https://bitbucket.org/rxe/galvanise_v2.

Finnsson H., Björnsson Y. (2008). Simulation-based approach to General Game Playing. In *Aaai'08*, p. 259–264. AAAI Press.

Finnsson H., Björnsson Y. (2011). Cadiaplayer: Search-control techniques. *KI - Künstliche Intelligenz*, vol. 25, n° 1, p. 9–16.

Genesereth M., Björnsson Y. (2013). The international general game playing competition. *AI Magazine*, vol. 34, n° 2, p. 107-111.

Genesereth M., Love N. (2005). General game playing: Overview of the aaai competition. *AI Magazine*, vol. 26, p. 62–72.

Koriche F., Lagrue S., Piette É., Tabary S. (2015). Résolution de scsp avec borne de confiance pour les jeux de stratégie. In *Jfpc'15*, p. 160-169.

Koriche F., Lagrue S., Piette É., Tabary S. (2016). General game playing with stochastic csp. *Constraints*, vol. 21, n° 1, p. 95–114.

Lecoutre C., Likitvivatanavong C., Yap R. H. C. (2015). STR3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, vol. 220, p. 1–27.

Lifschitz V., Yang F. (2011). Eliminating function symbols from a nonmonotonic causal theory. In *Knowing, reasoning, and acting: Essays in honour of hector j. levesque.* College Publicaions.

Littman M. L., Majercik S. M., Pitassi T. (2001). Stochastic boolean satisfiability. *Journal of Automated Reasoning*, vol. 27, n° 3, p. 251–296.

Love N., Genesereth M., Hinrichs T. (2006). *General game playing: Game description language specification*. Rapport technique n° LG-2006-01. Stanford University.

McKay B. D., Piperno A. (2014). Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, vol. 60, n° 0, p. 94 - 112.

Mehat J., Cazenave T. (2008). *An account of a participation to the 2007 general game playing competition.* http://www.ai.univ-paris8.fr/~jm/ggp/ggp2008-2.pdf. (unpublished)

Möller M., Schneider M. T., Wegner M., Schaub T. (2011). Centurio, a general game player: Parallel, Java- and ASP-based. *Künstliche Intelligenz*, vol. 25, n° 1, p. 17-24.

Méhat J., Cazenave T. (2011). A parallel general game player. *KI - Künstliche Intelligenz*, vol. 25, n° 1, p. 43-47.

Sabin D., Freuder E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *ECAI'94*, p. 125–129.

Schiffel S., Thielscher M. (2007). Fluxplayer: A successful general game player. In *Aaai'07*, p. 1191–1196. AAAI Press.

Schreiber S. (2014). *Ggp.org - tiltyard gaming server.* http://tiltyard.ggp.org/. (CS227b class at Stanford University)

Silver D., Huang A., Maddison C. J., Guez A., Sifre L., Driessche D. v. d. *et al.* (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, vol. 529, p. 484–503.

Sturtevant N. R. (2008). An analysis of UCT in multi-player games. *International Computer Games Association Journal*, vol. 31, n° 4, p. 195–208.

Thielscher M. (2005). Flux: A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, vol. 5, n° 4-5, p. 533-565.

Thielscher M. (2010). A general game description language for incomplete information games. In *Aaai'10.*

Thielscher M. (2011a). GDL-II. *KI - Künstliche Intelligenz*, vol. 25, n° 1, p. 63–66.

Thielscher M. (2011b). The general game playing description language is universal. In *Ijcai'11*, p. 1107–1112. AAAI Press.

Ullmann J. R. (2007). Partition search for non-binary constraint satisfaction. *Information Sciences*, vol. 177, n° 18, p. 3639–3678.

Walsh T. (2009). Stochastic constraint programming. *Computing Research Repository*, vol. abs/0903.1152.

Zobrist A. (1990). A new hashing method with application for game playing. *International Computer Games Association Journal*, vol. 13, n° 2, p. 69–73.