Compiling Strategic Games with Complete Information into Stochastic CSPs

Frédéric Koriche, Sylvain Lagrue, Éric Piette, and **Sébastien Tabary**
CRIL Univ. Artois - CNRS UMR 8188, France-62307 Lens
{koriche,lagrue,epiette,tabary}@cril.fr

Abstract

Among the languages used for representing goals, actions and their consequences on the world for decision making and planning, GDL (Game Description Language) has the ability to represent complex actions in potentially uncertain and competitive environments.

The aim of this paper is to exploit stochastic constraint networks in order to provide compact representations of strategic games, and to identify optimal policies in those games with generic forward checking method. From this perspective, we develop a compiler allowing to translate games, described in GDL, into instances of the Stochastic Constraint Optimization Problem (SCSP). Our compiler is proved correct for the class GDL of games with complete information and oblivious environment. The interest of our approach is illustrated by solving several GDL games with a SCSP solver.

Introduction

The ability for a computer program to effectively play any strategic game, often referred to *General Game Playing* (GGP), is a key challenge in AI (Nau, Ghallab, and Traverso 2004). Actually, a competition in GGP is annually organized by AAAI. This competence to play at any game has led researchers to compare various approaches, including Monte Carlo methods, automatic constructions of evaluation functions (Clune 2008), logic programming (Thielscher 2005), and answer set programming (Möller et al. 2011). Of course, the GGP problem is not confined to games: it can model sequential decision problems with mono or multi-agent.

In the context GGP, games are described in a representation language, called GDL for *Game Description Language* (Love et al. 2008). Based on logic programming, the first version of this language was able to capture any game with complete and certain information. A new version of this language, GDLII (GDL *with Incomplete Information* (Thielscher 2010)), is expressive enough to handle games with uncertain and incomplete information. In this setting, uncertainty is modeled by the response of the environment, which is simply one of the players. The aim of the environment, called random, is not to pursue a specific policy, but to convey uncertainty in the form of a probability distribution over a set of possible actions. Thus, the environment can capture dice rolls, coin flips, or card shuffles.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

A promising approach for meeting the GGP challenge is the paradigm of constraint programming. Indeed, representing a game as a constraint network enables us to use general search algorithms and propagation techniques for inferring optimal policies. To this end, several formalisms have been proposed for encoding games into constraint networks; they include quantified constraint satisfaction problems (QCSP) (Genta et al. 2008), strategic constraint satisfaction problems (Bessiere and Verger 2006) and, more recently, constraint games (Nguyen, Lallouet, and Bordeaux 2013). Yet, these approaches cannot address random players and, more generally, the connection with GDLII. The aim of this article is to handle GGP with an original viewpoint, based on Stochastic Constraint Satisfaction Problems (SCSP) (Walsh 2002). This formalism is expressive enough to model and solve games with complete information, including those described in QCSP (Balafoutis and Stergiou 2006), but also to capture random players. We show how to rewrite GDL games in SCSP and to infer optimal strategies (in expectation) with a generic constraint optimization algorithm.

GDL Formalism

The purpose of GDL is to provide a generic language for representing any game, including collaborative games and games with simultaneous actions. The GDLII version (Thielscher 2010) handles games with partial observations and stochastic actions. Our present study focuses on a fragment of this version: games with complete information and oblivious environment. For such games, the current state is fully observable by all players, and the environment has a random behavior which cannot be influenced by other players. This fragment of GDLII covers a well-studied class in game theory (Cesa-Bianchi and Lugosi 2006); in practice, this class captures various strategy games where players have no effects on the actions of the environment. Characteristic examples are "dice games" in which players can choose the dice they throw, but cannot influence the dice behavior.

Language. GDL is derived from logic programming with negation and equality. Recall that the Herbrand universe of a logic program is the set of ground terms obtained from function symbols (including constants) of the program. In a GDL program, players and game objects are described by constants, while fluents (or percepts) and actions (moves) are described by terms. For example, in the tic-tac-toe game, the term $\mathtt{cell}(2,2,\mathtt{b})$ is a fluent indicating that the cell at position (2,2) on the board is marked with a black token.

By Σ, we denote the Hebrand universe of a GDL program; Σ_F and Σ_A respectively denote the subsets of Σ corresponding to fluent terms and action terms.

The atoms of a program GDL are constructed from a finite set of relation symbols and a countable set of variable symbols. Some symbols have a specific role in the program; they are described in Table 1.

Keywords	**Description**
`role(J)`	`J` is a player
`init(F)`	the fluent `F` is part of the initial state
`true(F)`	`F` is part of the current state
`legal(J, A)`	`J` can do the move `A`
`does(J, A)`	the move of `J` is `A`
`next(F)`	`F` is part of the next state
`terminal`	the current state is terminal
`goal(J, N)`	`J` has `N` on the current state
`random`	is the player environment

Table 1: GDL keywords

For example, $\mathtt{legal}(\mathtt{J},\mathtt{mark}(\mathtt{X},\mathtt{Y}))$ indicates that player J is allowed to mark the square $(\mathtt{X},\mathtt{Y})$ of the board.

The rules of a GDL program are first-order clauses composed of an atom for the consequent and a set of literals for the antecedent. For example, the rule:

$$\mathtt{legal}(\mathtt{random},\mathtt{noop}) \leftarrow \mathtt{true}(\mathtt{control}(\mathtt{bob}))$$

indicates that noop (do nothing) is a legal action of player random if it is the turn of player bob.

In order to be *valid*, a GDL program P must obey certain syntactic conditions. Specifically, P must be *stratified* (Apt, Blair, and Walker 1988) to admit a standard model, and *allowed* (Lloyd and Topor 1986) to ensure that only finitely many positive ground atoms are true in this model. Due to space reasons, we refer to (Love et al. 2008) for details. Furthermore, the GDL keywords must be used as follows:

(i) role only appears as facts;

(ii) true only appears in the body of rules;

(iii) init only appears in the head of rules and can not depend on keywords true, legal, Does, next, terminal and goal;

(iv) Does only appears in the body of rules and does not depend on legal, terminal and goal;

(v) next only appears in the head of rules.

Finally, in the context of games with complete information and oblivious environment, we add the conditions:

(vi) For each fluent F, there exists an instance f of F such that $\mathtt{init}(\mathtt{f})$ is a fact.

(vii) Atoms $\mathtt{legal}(\mathtt{random},\mathtt{A})$, where A is an action, only appear as facts.

Semantics. *Markov games* (a.k.a *Stochastic games*) (Shoham and Leyton-Brown 2009) is a common model for representing games with uncertain information. A Markov game with n players over the Herbrand universe Σ is a tuple $\langle N, S, \boldsymbol{A}, P, \boldsymbol{R}\rangle$ such that: [1]

- $N = \{1, \dots, n\}$ is the set of *players*,
- S is the set of *states*;
- $\boldsymbol{A} = A_1 \times ... \times A_n$, where A_i is the finite set of actions available to player i ;
- $P : S \times \boldsymbol{A} \times S \to [0,1]$ is the transition function; $P(s, \boldsymbol{a}, s')$ is the probability of moving from state s to state s' by applying action $\boldsymbol{a}$;
- $\boldsymbol{R} = \langle r_1, ..., r_n\rangle$, where $r_i : S \to \mathbb{R}$ is the reward function of player i.

Together with the above tuple, a Markov game includes an *init* state $s_0 \in S$ and a set $S_{ter} \subseteq S$ of *terminal* states.

Now, we show how to construct a Markov game from a valid GDL program P.

A game state is a set of ground fluent terms. Because the syntactic restrictions of GDL guarantee a finite derivation of ground terms, all states are finite subsets of Σ_F. The n instances of $\mathtt{role}(\mathtt{R})$ define the n players of the stochastic model, and the environment (random) is set to player $n+1$. The initial state is built from the terms of $\mathtt{init}(\mathtt{F})$.

Concerning the legal moves of players in a given state $s = \{f_1, \cdots, f_m\}$, we denote $s_{\mathtt{true}}$ the set of facts $\{\mathtt{true}(f_1), ..., \mathtt{true}(f_m)\}$. The ground terms of $\mathtt{legal}(\mathtt{J},\mathtt{A})$ derivable from $\mathtt{P} \cup s_{\mathtt{true}}$, define all legal moves of the player J in the state s. The final state S_{ter} and the vector of winning function $\boldsymbol{R}$ are constructed analogously, using the atoms terminal and $\mathtt{goal}(\mathtt{R},\mathtt{N})$. In a game with oblivious environment, the set of legal moves of $n+1$, denoted $L(n+1)$, is independent of the current state: it is built from the terms of $\mathtt{legal}(\mathtt{random},\mathtt{A})$.

For the transition function, we need the current state and the actions performed simultaneously in that state. For a vector $\boldsymbol{a} = \langle a_1, \cdots, a_n, a_{n+1}\rangle$ of actions performed by the n players and the environment $(n+1)$, we denote $\boldsymbol{a}_{\mathtt{does}}$ the set of facts :$\{\mathtt{does}(1, a_1), ..., \mathtt{does}(n+1, a_{n+1})\}$

The next state is constructed using the ground terms of $\mathtt{next}(\mathtt{F})$ derivable from $\mathtt{P} \cup s_{\mathtt{true}} \cup \boldsymbol{a}_{\mathtt{does}}$. This state is denoted $Q(s, \boldsymbol{a})$. In particular, if $\boldsymbol{a}$ is the concatenation $\boldsymbol{a}' \cdot a_{n+1}$, then $Q(s, \boldsymbol{a}' \cdot a_{n+1})$ describes the next state obtained from s when, simultaneously, the players performed $\boldsymbol{a}'$ and the environment played a_{n+1}. The probability distribution associated to the transition function is captured by the stochastic behavior of the environment: it defined by the uniform distribution over all legal moves that can be played by $n+1$ in the current state.

We note in passing that the resulting states are not necessarily equiprobable. For example, a loaded dice with a prob-

[1]For a set U, we denote 2^U the set of all *finite* subsets of U.

ability of 1/2 to give 6 can be modeled by ten actions of `random`, whose the first five have the same effect (i.e. 6).

Definition 1 *The semantic of a* GDL *program* P *(with oblivious environment) is a Markov game* $\langle N, S, \boldsymbol{A}, P, \boldsymbol{R} \rangle$ *such that:*

- $N = \{i : \mathtt{P} \models \mathtt{role(i)}$ *et* $\mathtt{i} \neq \mathtt{random}\}$ *;*
- $S = 2^{\Sigma_F}$ *;*
- $A_i = \{a : \mathtt{P} \cup s_{\mathtt{true}} \models \mathtt{legal(i,a)}, s \in S\}$ *;*
- $P(s, \boldsymbol{a}, s') = \frac{|\{a_{n+1} \in L(n+1) : s' = Q(s, \boldsymbol{a} \cdot a_{n+1})\}|}{|L(n+1)|}$
- $r_i(s) = \begin{cases} c & \textit{if } \mathtt{P} \cup s_{\mathtt{true}} \models \mathtt{goal(i,c)}, \\ 0 & \textit{otherwise}. \end{cases}$

The initial and terminal states are defined by:

$$s_0 = \{f \in \Sigma_F : G \models \mathtt{init}(f)\}$$
$$S_{ter} = \{s \in S : G \cup s_{\mathtt{true}} \models \mathtt{terminal}\}$$

An Example. The *Orchard* (*Obstgarten*) game is a cooperative board game involving up to four players and a random environment. The game consists in 4 trees, each containing 10 fruits, and a raven with 9 items. During each round of the game, each player rolls a six-sided dice whose 4 faces are associated to a specific tree where the player removes a fruit of this tree, 1 face is associated to the raven where the player removes an item of the raven and 1 basket face where the player removes two fruits of his choice.

The goal is to remove all fruits of each tree before removing all items of the raven. Interestingly, the only decision made by the player is to choose fruits to remove when the die lands on the basket face. The optimal policy (wins 68.4% of cases) is to take the fruits in the fullest tree.

A GDL description of a "restricted" Orchard game with one player (bob), 2 trees with 2 fruits and a raven of size 1, is reported in Figure 1. A four-sided dice is used: y (take one fruit in the yellow tree), g (take one fruit in the green tree), b (make a choice to remove two fruits) and r (remove an item of the raven).

```
% roles
role(bob)
role(random)
% integer operations
succ(0,0)
succ(0,1)
succ(1,2)
% colors of trees
tree(y)
tree(g)
% initialization of the state of the game
init(state(2,2,1))
init(control(random))
% definition of legals moves
legal(random,roll(D))
legal(bob,noop) ← true(control(random))
legal(bob,choice(C1,C2)) ← tree(C1),         tree(C2),
true(control(bob))
% change in game
next(control(bob)) ← does(random,roll(b)),
                                            true(control(random))
next(control(random)) ← true(control(bob))
next(control(random)) ← does(random,roll(D)),
                                 true(control(random)), D ≠ b
next(state(Y,G,R)) ← true(state(Y1,G,R)),
                              succ(Y,Y1),does(random,roll(y))
...
next(state(Y,G,R)) ← true(state(Y1,G1,R)),
              succ(Y,Y1), succ(G,G1), does(bob,choice(g,y))
% goals and rewards
goal(bob,100) ← true(state(0,0,R))
% end of the game
terminal ← true(state(Y,G,0))
terminal ← true(state(0,0,R))
```

Figure 1: GDL Program of "restricted" Orchard Game

Stochastic CSPs

The stochastic CSP model that we present in this study extends the original framework of Walsh (2002) to deal with soft constraints.

Formally, a *Stochastic Constraint Satisfaction Problem* (SCSP) is a 6-tuple $\langle X, Y, D, P, \mathcal{C}, \theta \rangle$ where X is a set of n variables, Y is a subset of X representing the stochastic variables, D represents the set of domains associated to variables of X, P is the set of probability distribution applied to domains of the stochastics variables, $\mathcal{C}$ is the set of constraints and θ is the threshold value in $\mathbb{R}$.

A SCSP is composed of decision variables and stochastic variables. The decision variables have the same meaning as those defined in the classical CSP framework. For each stochastic variable, we associate a probability distribution to the values of the domain. We denote $D(x)$ the domain associated to variable x. More generally, for a subset $Z = \{z_1, \cdots, z_k\} \subseteq X$, we denote $D(Z)$ the set of tuples of values $D(z_1) \times \cdots \times D(z_k)$.

Each constraint of the SCSP is a pair $c = \langle scp_c, val_c \rangle$, where scp_c is a subset of X, called *scope* of c, and val_c is a function that associates to each tuple $\boldsymbol{\tau} \in D(scp_c)$ a value (or utility) $val_c(\boldsymbol{\tau})$ in $\mathbb{R} \cup \{-\infty\}$. A constraint c is *hard* if val_c only returns $-\infty$ or 0. In this case, c can be represented in the usual manner by a relation, noted rel_c which contains the set of tuples allowed for scp_c (i.e. the tuples $\boldsymbol{\tau}$ for which $val_c(\boldsymbol{\tau}) \neq -\infty$). In a SCSP, the scope of any hard constraint is restricted to decision variables. We extend the SCSP framework defined in Walsh (Walsh 2002) to deal with soft constraints. Specifically, for any soft constraint c, the range of its value function val_c is $\mathbb{R}$.

An *instantiation* I of a set $Z = \{z_1, \ldots, z_k\}$ of k variables is a set $\{(z_1, a_1), \ldots, (z_k, a_k)\}$ such as $\forall z_i \in Z, a_i \in D(z_i)$ and $\forall z_i, z_j \in Z, z_i \neq z_j$. By $I_{|Z}$, we denote the projection of I onto the subset of variables $Z \subseteq X$. The utility of I is $val(I) = \sum_{c \in \mathcal{C}} val_c(I_{|scp_c})$

A *policy* π is represented by a tree in which each node is a variable of X. The nodes associated to decision variables have only one child, while the nodes associated to stochastic variables have a child for each possible value of the domain. Edges are labeled by the value assigned to the corresponding variable. Each path in the policy is a sequence of assignments of values to variables. The leaves are labeled by the utility associated to the sequence of assignments of the corresponding path. The expected utility of a policy π is defined as the sum of utilities of the leaves weighted by their probability. A *solution* of SCSP is a policy π whose expected utility is greater than or equal to the threshold θ. Since $\theta > -\infty$, this implies that all hard constraints must be satisfied by a solution.

For example, consider the SCSP presented on Figure 2 and the associated policy π. The decision variables are x and z (assigned to the value 0). The domain of the stochastic variable y contains tree values (0, 1 et 2), with same probability $\frac{1}{3}$. Since π does not violate the hard constraint c_h, and satisfies the soft constraint c_s with expected utility $2/3 \geq \theta$, this policy is a solution of the SCSP.

$X = \{x, y, z\}$	$D(x) = D(y) = D(z) = \{0, 1, 2\}$
$Y = \{y\}$	$P(0) = P(1) = P(2) = 1/3$
$\mathcal{C} = \{c_h, c_s\}$	$c_h : x = z$
	$c_s(x, y) = \begin{cases} 1 & \text{if } x + y \geq 1 \\ 0 & \text{else} \end{cases}$
$\theta = 1/2$	

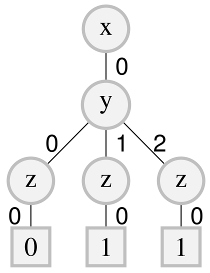

Figure 2: A SCSP and a policy π

From GDL to SCSP

In order to rewritte a GDL program into an SCSP, it is necessary to fix in advance a time horizon T, whose values $t \in \{1, \cdots, T\}$ capture the rounds of the game. In this section, we present a compilation method that takes as input a GDL program P and a time horizon T, and returns as output the set of variables X, the domains D, the constraints $\mathcal{C}$, and the probability distributions P of the corresponding SCSP. This compilation is divided into four steps detailled below.

Eliminating Functions. Given a GDL program P, the components of the associated SCSP are extracted from the terms, atoms and rules of P. In particular, the domains and constraints use ground terms derivable from P. Altough the syntactic restrictions of GDL allow us to derive all ground terms of P appearing as fluents and actions, the cost of such queries in a stratified logic program can be prohibitive (Dantsin et al. 2001).

To circumvent this issue, we transform the GDL program P into a program P′ without function symbols, using the method of Lifschitz et Yang (2011). This method includes two steps: *flattening* that reduces the nested terms, followed by *elimination* which replaces functions by predicates.

Flattening is specified as follows: for each function symbol f appearing in P, construct an equivalent program $\mathtt{P_f}$, called f-*flattened*, wherein each occurence of a term of the form $\mathtt{f(t_1, \cdots, t_k)}$ is transformed by the equality of the form $\mathtt{f(t_1, \cdots, t_k) = Z}$, where Z is a renaming variable.

For example, if the GDL program P contains the ground atom $\mathtt{legal(random, roll(c))}$, then in $\mathtt{P_f}$ the atom is replaced by the conjunction $\mathtt{legal(random, Z), roll(c) = Z}$. By flattening all function symbols f occurring in P, the resulting program does not contain any nested term. We note that flattening is polynomially bounded by the number of functions and the depth of terms.

The elimination step replaces each function symbol f of arity k with a relation symbol F of arity $k + 1$. In doing so, each equality of the form $\mathtt{f(t_1, \cdots, t_k) = Z}$ is replaced by the atom $\mathtt{F(t_1, \cdots, t_k, Z)}$. Finally, the rule[2]: $(\exists!\mathtt{Z})\mathtt{F(t_1, \cdots, t_k, Z)}$ is added to P′ for ensuring the equivalence of the stable models between P and P′.

Extracting Variables. The set X of the variables of the SCSP is obtained in the following way. To each round $t \in \{1, \cdots, T\}$, we associate the variables $role_t$, $control_t$, and $terminal_t$. Intuitively, $role_t$ captures the players at turn t, $control_t$ indicates who must play at turn t, and $terminal_t$ indicates whether the game is over, or not, at turn t.

To each instance i of role(J) and to each game round $t \in \{1, \cdots, T - 1\}$ we associate a variable $a_{i,t}$ indicating the action performed by player i at turn t. Similarly, the variables $a_{random,t}$ are associated to the action of the environment at turn t.

The SCSP variables associated with fluents are extracted from predicates of P′. Specifically, if $\mathtt{F(X_1, \cdots, X_k, Z)}$ is an atom such as the renaming variable appears in one of predicats init, true and next, then F is thus a fluent, and hence, the corresponding variable f_t is added to X.

Besides actions and fluents, a GDL program can include "static" relation symbols which are merely used to establish relationships between fluents (e.g. succ(I,J)). To each static relation and round t, a corresponding SCSP variable is associated.

Extracting Domains. Based on previous considerations, $terminal_t$ is boolean, and the domain of $role_t$ and $control_t$ is the set $\{1, \cdots, N + 1\}$ formed by the players and the environment (random).

The domain of every fluent variable f is given by all combinations of constants $\mathtt{c_1, \cdots, c_k}$ which can be instantiated by the atoms A of the form $\mathtt{F(t_1, \cdots, t_k, z)}$. These domains can be extracted in a simple way by filtering the following network. To each fluent F of arity $k + 1$, add the variables $f_1^v, \cdots, f_k^v$ and $f_1^c, \cdots, f_k^c$. The domains of $f_1^v, \cdots, f_k^v$ are initially formed by all constant symbols occurring in the program P. The domains of $f_1^c, \cdots, f_k^c$ are formed by the set of constant symbols occurring at position i of any atom prefixed by F in P. For each index i, add an edge (f_i^v, f_i^c), and

[2]In logic programming $\exists!$ means "there exists exactly one".

GDL	**Variable SCSP**	**Domain SCSP**
`role(J)`	$role_t$	$\{\mathtt{random}, \mathtt{bob}, \mathtt{undefined}\}$
`legal(bob,A)`	$a_{bob,t}$	$\{\{\mathtt{choice}\} \times \{\mathtt{r}, \mathtt{v}\} \times \{\mathtt{r}, \mathtt{v}\}\} \cup \{\mathtt{noop}, \mathtt{undefined}\}\}$
`legal(random,A)`	$a_{random,t}$	$\{\{\mathtt{roll}\} \times \{\mathtt{c}, \mathtt{p}, \mathtt{r}, \mathtt{v}\}\}$
`next(state(...))`	$state_t$	$\{(3,3,2), \dots, (0,0,0)\} \cup \{\mathtt{undefined}\}$
`next(control(...))`	$control_t$	$\{\mathtt{bob}, \mathtt{random}, \mathtt{undefined}\}$
`succ(...)`	$succ_t$	$\{(0,0), (0,1), (1,2), (2,3), \mathtt{undefined}\}$
`tree(...)`	$tree_t$	$\{\mathtt{r}, \mathtt{v}, \mathtt{undefined}\}$
`terminal(...)`	$terminal_t$	$\{\mathtt{true}, \mathtt{false}\}$

Table 2: Variables and domains at time t associated to the *Orchard* game

add an edge (f_i^v, g_j^v) whenever the same variable symbol appears at position i in an atom prefixed by F and in position j by a atom prefixed by G. By arc-consistency, we remove in every variable f_i^v the constants without support. Thus, the values of f_i^v are given by the union of the values of f_i^c and the values of neighboring variables g_j^v with a support. Based on this filtered network, the domain of the fluent variable f_t is $D(f_t) = D(f_1^v) \times \cdots \times D(f_k^v)$.

Domains of action variables are extracted analogously, by first identifying from the relation `legal(j,A)` the actions A involved in the construction of the domain of $a_{j,t}$.

Technically, we add the value `undefined` to every domain of decision variables, because if $terminal_t$ take the value `true`, all variables different from $terminal$ at $t' > t$ are instantiated to `undefined`.

Table 2 illustrates the extraction of SCSP variables and domains from the Orchard game specified in Figure 1.

Extracting Constraints. To each rule R of the GDL program P and each round t, we associate a constraint $C_{R,t}$. The cscopes are specified by the rewriting rules of Table 3.

Atom GDL	**Scope of the constraint**
init`(f(...))`	$\{f_0\} \in scp_{C_0}$
true`(f(...))`	$\{f_t\} \in scp_{C_t}$
does`(j,a(...))`	$\{a_{j,t}\} \in scp_{C_t}$
next`(f(...))`	$\{f_{t+1}\} \in scp_{C_t}$
legal`(j,a(...))`	$\{a_{j,t}\} \in scp_{C_t}$
goal`(j,N)`	$\{role_t\} \in scp_{score_t}$
terminal	$\{terminal_t\} \in scp_{terminal}$

Table 3: rewriting rules for the constraint scopes

In the SCSP model, all constraints are hard, except the constraint related to the game scores. Thus, for each rule R of the program P, the semantic of the corresponding constraint $C_{R,t}$ is a "relation" over the domain of its scope. This relation is constructed as follows. After eliminating function symbols, we know that all terms of R in P have been replaced with predicates in the corresponding rule R′ of P′. Without loss of generality, suppose that R′ is of the following form:

$$\mathtt{A}_1, \cdots, \mathtt{A}_k \leftarrow \mathtt{B}_1, \cdots, \mathtt{B}_{k'}$$

Note that, due to the flattening step, the consequent of R′ can include more than one atom. Since R′ expresses an implication, the relation of $C_{R,t}$ must capture this implication. From this perspective, let A (resp. B) denote the set $\{\mathtt{A}_1, \cdots, \mathtt{A}_k\}$ (resp. $\{\mathtt{B}_1, \cdots, \mathtt{B}_{k'}\}$). Let $\mathtt{U}(\mathtt{A}_i)$ denote all combinations of constants which are instances of $\mathtt{A}_i$, and U(A) (resp. U(B)) denote the join of this combination on A (resp. B). Furthermore, let C(B) (resp. C(AB)) denote all combinations of constants which are instances of the conjunction on B (resp. $\mathtt{A} \cup \mathtt{B}$). The combinations of constants which are instances of R′ are therefore:

$$\mathtt{C}(\mathtt{AB}) \cup (\mathtt{U}(\mathtt{A}) \bowtie (\mathtt{U}(\mathtt{B}) \setminus \mathtt{C}(\mathtt{B})))$$

These instances can be obtained using a conjunctive query on R′. The resulting projection on $scp_{C_{R,t}}$ gives $rel_{C_{R,t}}$.

By the syntactic restriction (vi) of GDL programs, the constraint associated to `init` allows exactly one value per fluent f_0. The soft constraint $score_t$ associates to each player j the value N given in **goal**`(j,N)` if the body of the associated rule for this atom is true, and the value 0 otherwise.

Finally, the component P of SCSP is formed by giving a uniform probability distribution over the domain of $random_t$ at each turn t.

Equivalence Between Models. In the resulting SCSP instance, we denote by $X_{f,0}$ the set of fluent variables occurring in the initial state, and by $X_{f,t}$ (resp. $X_{a,t}$) the set of fluent (resp. action) variables appearing at time t. By construction,

$$X = \left(\bigcup_{t=0}^{T} X_{f,t} \right) \cup \left(\bigcup_{t=1}^{T-1} X_{a,t} \right)$$

Based on these notations, our SCSP instance determines a directed acyclic graph (DAG), denoted $G_{\mathtt{SCSP},T}$, whose vertices S and edges A are built as follows:

- the only allowed tuple $s_0 \in D(X_{f,0})$ is a vertex of S;
- if $s_t \in D(X_{f,t})$ is a vertex of S and $s_{t+1} \in D(X_{f,t+1})$ given by the concatenation $s_t \cdot s_{t+1}$ is an allowed tuple, then s_{t+1} is a vertex of S and (s_t, s_{t+1}) is a edge of A.

Each edge $(\boldsymbol{s}_t, \boldsymbol{s}_{t+1})$ is labeled by the probability $\frac{k}{d}$ where d is the size of $D(a_{\mathtt{random},t})$ and k is the number of allowed tuples $(\boldsymbol{s}_t, \mathtt{a}, \boldsymbol{s}_{t+1})$ where $\mathtt{a} \in D(a_{\mathtt{random},t})$. Finally, each vertex s_t is labeled by the associated table of each player i in the domain of $role_t$, its utility given by $score_t(i)$.

Similarly, we associate to a GDL program P, the following DAG $G_{\mathtt{SCSP},T}$: each state s of the Markov game of P is rewritten by removing the function symbols (using flattening) and retaining the arguments (combination of constants). The set of vertices is the set S of states of the Markov game of $\mathtt{P}'$ reachable from the initial state by any path of size at most $T-1$.[3] An edge (s_t, s_t+1) is added between two vertices iff there is a vector of legal actions $\boldsymbol{a}$ for s_t, such as $s_{t+1} = Q(s_t, \boldsymbol{a})$. Each edge (s_t, s_t+1) is labeled by the proportion of actions $\mathtt{a}$ of `random` for which $s_{t+1} = Q(s_t, \boldsymbol{a}\cdot\mathtt{a})$. Finally, each vertex s_t is labeled by its reward vector $\boldsymbol{R}(s_t)$.

Theorem 1 *For all time horizons T, the graphs $G_{SCSP,T}$ et $G_{\mathtt{P},T}$ are identical.*

Proof (Sketch). Based on the rewritting rule associated to `init`, the root of $G_{\mathtt{SCSP},T}$ and the root of $G_{\mathtt{P},T}$ are identical, and given by the unique vertex s_0. Suppose by induction hypothesis that s_t belongs to both $G_{\mathtt{SCSP},T}$ and $G_{\mathtt{P},T}$. Consider a state $s_{t+1} \in D(X_{f,t+1})$ such as $s_t \cdot s_{t+1}$ is an allowed tuple of the `SCSP`. So, there is a tuple $\boldsymbol{a} \in D(X_{a,t})$ such as $s_t \cdot \boldsymbol{a} \cdot \boldsymbol{s}_{t+1}$ is allowed by the `SCSP`. This implies that $\boldsymbol{a}$ is a vector of legal actions, and so s_{t+1} belongs to both $G_{\mathtt{SCSP},T}$ and $G_{\mathtt{P},T}$. Similarly, the edge (s_t, s_{t+1}) belongs to both graphs. Since the environment is oblivious and its domain of actions is the same for `P` and `SCSP`, the edge labels are identical. Finally, using the rewritting of **`goal`**`(j,N)` in $score_t$, the vertex labels are identical.

Resolution and Experiments

Resolution. We use some preprocessing techniques to improve the efficiency of the resolution of the generated SCSP. The first technique is to merge hard constraints of same scope. Specifically, given an `SCSP` P, two hard constraints c_i and c_j of P (with associated relation rel_{c_i} and rel_{c_j}) such as $scp_{c_i} = scp_{c_j}$ are converted to a unique constraint c_k such that $rel_{c_k} = rel_{c_i} \cup rel_{c_j}$ and $scp_{c_k} = scp_{c_j} = scp_{c_i}$.

Second, we remove all unary constraints (e.g. constraints c such that $|scp_c| = 1$). The domain of the variable involved in the constraint is restricted to values allowed by the tuples of the associated relation.

The final preprocessing technique is to identify universal variables in each constraint. A variable is “universal” if whatever the value assigned, the constraint is always satisfied. In formal terms, given a constraint c, a variable $x \in scp_c$ is universal if $|rel_c|$ is equal to the product of the size of the domain of x with the number of tuples of the relation associated to the constraint c_i such as $scp_{c_i} = scp_c \setminus \{x\}$. Such variables are removed from the scope of constraints.

The algorithm used to solve `SCSP` instances is presented in Algorithm 1. Based on (Balafoutis and Stergiou 2006), this algorithm is a `Forward Checking` (FC) method adapted for `SCSP`. Since our algorithm is applied to game solving, our version returns a solution policy whereas the original algorithm in (Balafoutis and Stergiou 2006) returns a satisfaction threshold. Here, the idea is to iteratively build a solution policy of the `SCSP` by propagating choices in the search space of policies. Given an `SCSP` P, an ordering is defined on X and the algorithm 1 instantiates the variables

[3]The size of a path is given by the number of its arcs.

Algorithm 1: FC

```
   Data: threshold $\theta_h$ policy $\pi$, int $i$
   Result: policy $\pi$, satisfaction $\theta$
1  if $i > |X|$ then return $(\pi, 1)$
2  $\theta \leftarrow 0$
3  foreach $v_j \in D(x_i)$ do
4      if $prune[i,j] = 0$ then
5          if $check(x_i, v_j, \theta_h)$ then
6              $\pi \leftarrow \pi \cup \{(x_i, v_j)\}$
7              if $x_i \in S$ then
8                  $p \leftarrow prob(x_i, v_j)$
9                  $q_i \leftarrow q_i - p$
10                 $(\pi, \theta_t) \leftarrow FC(\frac{\theta_h - \theta}{p}, \pi, i+1)$
11                 $\theta \leftarrow \theta + p * \theta_t$
12                 $restore(i)$
13                 if $\theta \geq \theta_h$ then return $(\pi, \theta)$
14             else
15                 $(\pi, \theta_t) \leftarrow FC(\theta_h, \pi, i+1)$
16                 $\theta \leftarrow max(\theta, \theta_t)$
17                 $restore(i)$
18                 if $\theta \geq \theta_h$ then
19                     return $(\pi, \theta)$
20                 else
21                     $\pi \leftarrow \pi \setminus \{(x_i, v_j)\}$
22 return $(\pi, \theta)$
```

of P following this order. When a decision variable is encountered, the algorithm tries to assign successively each value in its domain. This value is added to the current policy and the maximum threshold is returned to the previous recursive call. When a stochastic variable is encountered, FC deals with each value in its domain, and returns the sum of all the answers to the subproblems weighted by the probabilities of their occurence. When instantiating a decision variable or a stochastic variable, the algorithm calls the function `check` (Algorithm 2). This function prunes values from the domains of unassigned variables and checks whether this does not lead to a domain wipeout or exceed the threshold. The bound θ_h is also used to prune the search space and we identify all policies satisfying $\theta \geq \theta_h$. If all variables are instantiated without breaking any constraint, FC returns a couple (π, θ) corresponding to a solution policy.

An array $prune[i,j]$ is used to record the depth at which the value $v_j \in D(x_i)$ has been removed by forward checking, 0 indicating that the value is still in the domain of the variable. Each stochastic variable x_i has an upper bound q_i (initially set to 1) capturing the probability that the remaining values in $D(x_i)$ can contribute to a solution.

In Algorithm 2, the function `allowed` takes as input two pairs (x_i, v_j) and (x_k, v_l) and attempts to find an allowed tuple in rel_c for each constraint involving x_i and x_k. Algorithm 2 fails if at least one decision variable has an empty domain or if a stochastic variable has so many values removed that we can not hope to satisfy the constraints. More precisely, when a value v_l is removed from a stochastic variable x_k, we reduce q_k to $prob(x_k \leftarrow v_l)$, the probability that

Algorithm 2: check

```
Data: Variable $x_i$,int $v_j$, threshold $\theta$
Result: boolean $check$
1  for $k \in i{+}1\ to\ |X|$ do
2      $DomainEmpty \leftarrow true$
3      foreach $v_l \in D(x_k)$ do
4          if $prune[k,l] = 0$ then
5              if $!allowed((x_i,v_j),(x_k,v_l))$ then
6                  $prune[k,l] \leftarrow i$
7                  if $x_k \notin S$ then
8                      $DomainEmpty \leftarrow false$
9                  else
10                     $q_k \leftarrow q_k - prob(x_k,v_l)$
11                     if $q_k < \theta$ then return $false$
12         if $DomainEmpty$ then return $false$
13 return $true$
```

x_k takes the value v_l. Once the algorithm has reduced q_k to a value which is less than θ, the function returns false and a backtrack occurs since it is impossible to instantiate x_k and while satisfying all constraints.

Procedure `restore` (not illustrated) is called to restore values that have been removed from decision variables and to reset q_i for stochastic variables when a backtrack occurs.

Once a solution policy π has been inferred by FC at time horizon T, we need to estimate the reward of π at final states. To this end, the UCB algorithm (Auer, Cesa-Bianchi, and Fischer 2002) is used with Monte-Carlo simulations. Specifically, after reaching the horizon T for resolution, UCB uses simulations on all next states and estimated reward of move a, n_a is the number of times move a has been played so far, and n is the total number of moves played so far.

Experiments. We applied our rewritting method and resolution algorithm to several GDLII games:

- *Backgammon* is a standard 2-player board game with 2 dices and 15 pieces by player. The playable pieces are moved according to the roll of dies, and a player wins by removing all of his pieces from the board.
- *Can't stop* is a board game with 4 dices and 3 buddhists by player, our version involving only two players. The board includes 9 climbs of different sizes, and the goal is to reach the top of three climbs with three buddhists.
- *Kaseklau* is a little 2-player board game with a mouse and a cat. The goal is to roll the 2 dices to advance mouse and cat on different squares where slices of cheese are placed.
- *Orchard* is a 2-player board game in cooperation with a die. This game is explained in figure 1.
- *Yathzee* is a 2-player game where the goal is to get the highest score by rolling five dices. At each round, the dices can be rolled up to three times in order to make one of the 13 scoring combinations.
- *Tic-tac-toe* is a well-known deterministic game with two players (X and O) who iteratively mark a 3x3 grid.

All games have been parsed and Table 4 presents the resulting SCSP parameters: number of variables (#vars), maximum domain size (maxDom) and number of constraints (#constraints).

Figure 3 shows running times for different horizons. Backgammon is the most difficult game, with the largest number of variables, each with a large domain, and an important number of constraints. Yathzee is also challenging even with relatively few variables, but with many constraints. The Orchard game is very specific: it includes only one variable with large domain, while other variables have a small domain. Thus, it can be quickly solved for some small horizons, but the complexity increases with horizon. Can't Stop is resource consuming due to the large scope of constraints. The others games (Kaseklau and Tic Tac Toe) are solved in a reasonnable amount of time because the generated SCSP are small.

Game	#vars	maxDom	#constraints
Backgammon	76	768	86
Can't Stop	16	1296	409
Kaseklau	18	7776	106
Orchard	9	146410	40
"restricted" Orchard	9	100	22
TicTacToe	19	18	37
Yathzee	12	30	8862

Table 4: Translated games in SCSP and characteristics.

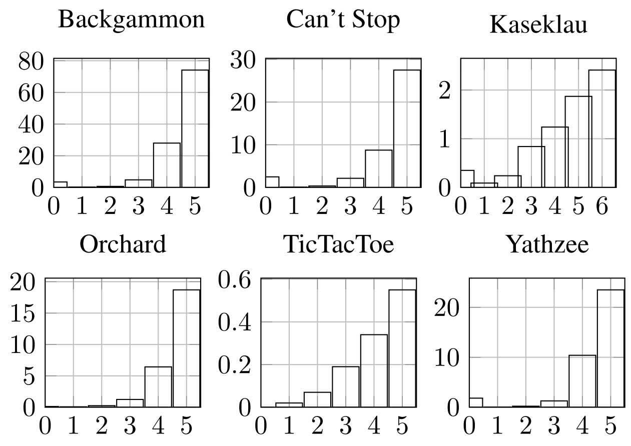

Figure 3: Running times for different horizons: The X axis plots the horizon and the Y axis plots the time in hundreds of seconds to solve the problem. Horizon 0 is the parsing time.

We present now results obtained for the "restricted" Orchard (the GDL program is described in Figure 1), and the winning policies for this game are given in Table 5. We focus on the values of variables $a_{bob,t}$ and $state_{0,t}$ in the policies satisfying the constraint network with horizon 4 and a threshold of 50%. The values of $state_{0,t}$ are triplets composed of the number of fruits in the trees r and v and the size of the raven c. The values of the variable $a_{bob,t}$ correspond to the 4 actions of fruit choice: (r, v), (v, r), (r, r), (v, v) and the action "noop". The player $random$ rolls the dice with the same probability.

The winning strategies are those involving the action $a_{bob,t}$. It is interesting to note that for $t = 1$ the amount of fruits in each tree is not modified. The computed strategy showed in the table for $t = 1$ is to select a fruit in each tree. For $t = 2$, the only feasible action is "noop" because it is the turn of `random`. The first four solutions capture the situation where the `random` takes the basket twice. Solutions 5 and 6 are those where the player has the choice at time 1 (so she chooses one fruit in each tree), then `random` takes a fruit in tree v followed by a fruit in tree r (solution 5) or the opposite (solution 6). Importantly, the number of solutions depends on the threshold. With a threshold value of 50%, the computed solutions correspond to the expected optimal strategy for the Orchard game.

Variable	t=1	t=2	t=3	t=4
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	choice v r	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice r v	noop	choice v r	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	choice r v	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice r v	noop	choice r v	undefined
$state_{0,t}$	2 2 1	1 1 1	1 0 1	0 0 1
$a_{bob,t}$	choice v r	noop	noop	undefined
$state_{0,t}$	2 2 1	1 1 1	0 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	noop	undefined

Table 5: Winning policies for the "restricted" Orchard game (in initial state, $state_{0,0}$ = (2 2 1) and $a_{bob,0}$ = noop)

Conclusion

In line with the paradigm of constraint programming for solving combinatorial optimization problems, we have proposed to use stochastic CSPs for solving `GDLII` games with full observation and oblivious environment. The semantics of this representaiton language is a Markov game. Our compiler `GDL`-to-`SCSP` is proved correct by the equivalence of models at each horizon. Using a variant of Forward Checking for solving `SCSP` instances, our first experiments on a series of `GDLII` games revealed that constraint programming is a promising approach for inferring optimal policies in strategic games with uncertainty.

This work calls for many perspectives. One of them is to extend our model to *non-oblivious* environments, whose probability distributions over actions may depend on the current state. Another perspective of research is to devise filtering methods for `SCSP` in order to accelerate the search of solution policies.

References

Apt, K. R.; Blair, H. A.; and Walker, A. 1988. Foundations of deductive databases and logic programming. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann. chapter Towards a Theory of Declarative Knowledge, 89–148.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.* 47(2-3):235–256.

Balafoutis, T., and Stergiou, K. 2006. Algorithms for stochastic CSPs. In *Proc. of CP'06*, 44–58.

Bessiere, C., and Verger, G. 2006. Strategic constraint satisfaction problems. In *Proc. of CP'06 Workshop on Modelling and Reformulation*, 17–29.

Cesa-Bianchi, N., and Lugosi, G. 2006. *Prediction, Learning, and Games*. Cambridge.

Clune, III, J. E. 2008. *Heuristic evaluation functions for general game playing*. Ph.D. Dissertation, University of California, Los Angeles, USA. Adviser-Korf, Richard E.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33(3):374–425.

Genta, I. P.; Nightingalea, P.; Rowleya, A.; and Stergiou, K. 2008. Solving quantified constraint satisfaction problems. *Artificial Intelligence* 172(6-7):73877.

Lifschitz, V., and 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.

Lloyd, I. W., and Topor, R. W. 1986. A basis for deductive database systems. ii. *J. Log. Program.* 30(1):55–67.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification. Technical report.

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

Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. Morgan Kaufmann.

Nguyen, T.-V.-A.; Lallouet, A.; and Bordeaux, L. 2013. Constraint games: Framework and local search solver. In *Proc. of ICTAI'13*, 8–12.

Shoham, Y., and Leyton-Brown, K. 2009. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

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

Thielscher, M. 2010. A general game description language for incomplete information games. In *Proc. of AAAI'10*, 994–999.

Walsh, T. 2002. Stochastic constraint programming. In *Proc. of ECAI'02*, 111–115.