Constraints (2016) 21:95–114
DOI 10.1007/s10601-015-9199-5

General game playing with stochastic CSP

Frédéric Koriche[1] · Sylvain Lagrue[1] · Éric Piette[1] ·
Sébastien Tabary[1]

Published online: 26 August 2015

Abstract The challenge of General Game Playing (GGP) is to devise game playing programs that take as input the rules of any strategic game, described in the Game Description Language (GDL), and that effectively play without human intervention. The aim of this paper is to address the GGP challenge by casting GDL games (potentially with chance events) into the Stochastic Constraint Satisfaction Problem (SCSP). The stochastic constraint network of a game is decomposed into a sequence of μSCSPs (also know as one-stage SCSP), each associated with a game round. Winning strategies are searched by coupling the MAC (Maintaining Arc Consistency) algorithm, used to solve each μSCSP in turn, together with the UCB (Upper Confidence Bound) policy for approximating the values of those strategies obtained by the last μSCSP in the sequence. Extensive experiments conducted on various GDL games with different deliberation times per round, demonstrate that the MAC-UCB algorithm significantly outperforms the state-of-the-art UCT (Upper Confidence bounds for Trees) algorithm.

Keywords Stochastic constraint satisfaction problem · Stochastic games ·
Game description language · General game playing

✉ Éric Piette
epiette@cril.fr

Frédéric Koriche
koriche@cril.fr

Sylvain Lagrue
lagrue@cril.fr

Sébastien Tabary
tabary@cril.fr

[1] Université Lille-Nord de France CRIL - CNRS UMR 8188, Artois, 62307 Lens, France

1 Introduction

From early on [23], the development of game-playing programs has become a major research area in the field of Artificial Intelligence. Nowadays, computer players for classical board games like chess [5] and checkers [22] are able to defeat human players on grandmaster level. Even for games of "chance" like Backgammon [27], and games with incomplete information like (heads-up limit hold'em) Poker [4], computer players have reached an excellent level. Yet, one point of objection to the success of these game-playing programs is that they are highly tailored to the game at hand, relying on the game specific knowledge and expertise of their developers.

The aim of *General Game Playing* (GGP) is to devise game-playing algorithms which are not dedicated to a specific game, but are general enough to effectively play a wide variety of games. A tournament is held every year by AAAI [10], in which computer players are supplied the rules of arbitrary new games and, without human intervention, have to play those games optimally. The rules of each game are described in a declarative representation language, called GDL for *Game Description Language* [14]. The latest version of this language, GDLII, is an extension of GDL which is expressive enough to describe finite multi-player games with uncertain and incomplete information [29]. GGP algorithms include, among others, logic programming [28], answer set programming [17], automatic construction of evaluation functions [7], and Monte Carlo methods [6, 9]. Beyond its play value, GGP offers a rigorous setting for modeling and analyzing sequential decision-making algorithms in multi-agent environments.

By providing a declarative approach for representing and solving combinatorial problems, Constraint Programming appears as a promising technology to address the GGP challenge. Actually, several constraint-based formalisms have already been proposed to model and solve games; they notably include *Quantified* CSP[11], *Strategic* CSP[3] and *Constraint Games* [19]. Most of these formalisms are, however, restricted to deterministic, perfect information games: during each round of the game, players have full access to the current state and their actions have deterministic effects. This paper focuses on *stochastic* games, with chance events, using the framework of stochastic constraint networks [12, 26, 30].

From this perspective, we study a fragment of the *Stochastic Constraint Satisfaction Problem* (SCSP), that captures GDLgames with uncertain (but complete) information. Interestingly, the SCSP for this class of games can be decomposed into a sequence of μSCSPs (also known as *one-stage* stochastic constraint satisfaction problems [30]). Based on this decomposition, we propose a sequential decision-making algorithm, MAC-UCB, that combines the MAC (*Maintaining Arc Consistency*) technique for solving each μSCSP in the sequence, and the multi-armed bandits *Upper Confidence Bound* (UCB) method [1] for approximating the expected utility of strategies. We show that, in practice, MAC-UCB significantly outperforms UCT (*Upper Confidence bound for Trees*), which is the state-of-the-art GGP algorithm for stochastic games [25]. MAC-UCB also dominates FC-UCB, a variant where the MAC algorithm is replaced with the classical Forward Checking (FC) method. Such conclusions are drawn from comparing the performance of these algorithms, using extensive experiments (1,800,000 face-offs) over a wide range of GDL games.

The paper is organized as follows. The formal setting of GDL games is introduced in Section 2, and the SCSP framework for encoding GDL games is detailed in Section 3. Our algorithm MAC-UCB is examined in Section 4. The experimental setup and empirical results are discussed in Section 5. Finally, Section 6 concludes with several perspectives of further research.

2 GDLgames

The problems under consideration in GGP are *finite sequential* games. Each game involves a finite number of players, and a finite number of states, including one distinguished initial state, and one or several terminal states. On each round of the game, each player has at her disposal a finite number of actions (called "legal moves"); the current state of the game is updated by the simultaneous application of each player's action (which can be "noop" or do nothing). The game starts at the initial state and, after a finite number of rounds, ends at some terminal state, in which a reward is given to each player. In a *stochastic* game, a distinguished player, often referred to as "chance", can choose its actions at random according to a probability distribution defined over its legal moves. In this study, we shall focus on *fully observable stochastic* games in which, at each round, the current game state, the players' legal moves, and the probability distribution over chance events, are accessible to all agents.

2.1 GDLsyntax

GDL is a declarative language for representing, in a compact and intuitive way, finite games. Basically, a GDL program is a set of rules described in first-order logic. Players and game objects (coins, dices, locations, etc.) are described by constants, while fluents and actions are described by first- order terms. The atoms of a GDL program are constructed over a finite set of relation symbols and variable symbols. Some symbols have a specific meaning in the program, and are described in Table 1. For example, in the tic-tac-toe game, legal(alice, mark(X, Y)) indicates that player alice is allowed to mark the square (X, Y) of the board. In GDLII, the last two keywords of the table (random and sees) are added to represent stochastic games (ex: Backgammon), and partially observable games (ex: Battleship). In light of the games considered in this study, we will focus on GDLII programs *without* the sees keyword.

The rules of a GDL program are first-order Horn clauses. For example, the rule:

$$\texttt{legal(bob, noop)} \leftarrow \texttt{true(control(alice))}$$

Table 1 GDLII keywords

Keywords	Description
role(P)	P is a player
init(F)	the fluent F holds in the initial state
true(F)	the fluent F holds in the current state
legal(P, A)	the player P can take action A in the current state
does(P, A)	the player P performs action A
next(F)	the fluent F holds in the next state
terminal	the current state is terminal
goal(P, R)	the player P gets reward R in the current state
random	the "chance" player
sees(P, F)	the player P perceives F in the next state

indicates that `noop` is a legal action of `bob` if it is `alice`'s turn to move. In order to represent a finite sequential game, a `GDL` program must obey to syntactic conditions, defined over the terms and relations occurring in rules, and the structure of its rule set. We refer the reader to [14, 29] for a detailed analysis of these conditions.

Example 1 ”Matching Pennies” is a well-known game involving two players, who simultaneously place a penny (or coin) on the table, with the payoff depending on whether pennies match. We consider here a variant in which `alice` and `bob` cooperatively play against the chance player (`random`). During the first round, `alice` and `bob` simultaneously place a coin on the table and, during the second round, `random` flips a coin; `alice` and `bob` win only if all the three sides are heads or tails. The corresponding `GDL` program is described in Fig. 1. Notably, the built-in predicate `alleq(X, Y, Z)` is true if and only if `X` = `Y` = `Z`.

2.2 GDLsemantics

The semantics of any `GDL` program with chance events can be captured by a “stochastic game” [18, 24] which is essentially a multi-player Markov decision process. For the sake of clarity, we present here a slight variant of this model which uses an explicit representation of the chance player.

Definition 1 A k-player stochastic game consists of a set $\{0, 1, \cdots, k\}$ of *players*, with 0 referring to as the chance player, a set $\mathcal{A}$ of *actions*, a set $\mathcal{F}$ of *fluents*, and a labeled directed graph $G = \langle S, E\rangle$ where S is the set of nodes and E the set of directed edges (or arcs).

```
role(alice).
role(bob).
role(random).

side(tails).
side(heads).

init(coin(alice,unset)).
init(coin(bob,unset)).
init(coin(random,unset)).
init(control(alice)).
init(control(bob)).

legal(P,choose(X)) ← true(control(P)), side(X).
legal(P,noop) ← not(true(control(P))).

next(coin(P,X)) ← does(P,choose(X)), side(X).
next(coin(P,X)) ← does(P,noop), true(coin(P,X)).
next(control(random)) ← true(control(alice)).

terminal ← not(true(coin(alice,unset))), not(true(coin(random,unset))),
           not(true(coin(random,unset))).

has_won ← true(coin(alice,X)), true(coin(bob,Y)), true(coin(random,Z)),
          alleq(X,Y,Z).

goal(alice,1) ← has_won.
goal(alice,0) ← not(has_won).
goal(bob,1) ← has_won.
goal(bob,0) ← not(has_won).
```

Fig. 1 GDL program of the cooperative Matching Pennies game

$S_{goal} \subseteq S$ is the set of terminal nodes and $s_{init} \in S$ is the initial node. With G is associated a tuple of labeling functions $\langle A, F, P, r\rangle$ such that:

– $A = (A_0, A_1, \cdots, A_k)$, where A_p maps each non-terminal node $s \in S \setminus S_{\text{goal}}$ to a finite subset of actions $A_p(s) \in 2^{\mathcal{A}}$,
– F maps each node $s \in S$ to a finite subset of fluents $F(s) \in 2^{\mathcal{F}}$,
– P maps each non-terminal node $s \in S \setminus S_{\text{goal}}$ to a probability distribution over the action set of the chance player $A_0(s)$,
– $r = (r_1, \cdots, r_k)$, where each r_p maps each terminal node $s \in S_{\text{goal}}$ to a value $r_p(s) \in [0, 1]$.

With each non-terminal node $s \in S \setminus S_{\text{goal}}$ is associated one edge per tuple $a = (a_0, a_1, \cdots, a_k) \in A(s)$, where $A(s) = (A_0(s), A_1(s), \cdots, A_k(s))$. The successor of s with respect to a is denoted $Q(s, a)$. A *tree-like* stochastic game is a stochastic game for which the undirected graph of G is a tree.

Intuitively, $F(s)$ captures the state description of s, $A(s)$ defines the set of joint legal moves at s, $P(s)$ captures the likeliness of chance events, and $r(s)$ specifies the players' rewards at s. Any tuple $a = (a_0, a_1, \cdots, a_k) \in A(s)$, is called an *action profile*; a_p is the action of player p at s, and $a_{-p} = (a_0, a_1, \cdots, a_{p-1}, a_{p+1}, \cdots, a_k)$ is the action profile of the other players. Notably, a_{-0} induces a probability distribution over the set of states $\{Q(s, a) : a = (a_0, a_{-0}), a_0 \in A_0(s)\}$, where the probability of $Q(s, a)$ is given by $P(s)(a_0)$, the likeliness that event a_0 occurs at s.

Based on these notions, the stochastic game G associated with a GDL program G is defined as follows. Let B denote the Herbrand base (i.e. the set of all ground terms) of G. Then $\mathcal{A}$ (resp. $\mathcal{F}$) is the set of all ground action terms (resp. fluent terms) occurring in B. The number k of ground terms p such that role(p) $\in$ G, determines the set of players $\{0, 1, \cdots, k\}$. The node set S is constructed inductively from the source node s_{init} and the GDL keywords. Namely, the state description $F(s_{\text{init}})$ is given by the set of ground fluents f occurring in any atom init(f) of G. By induction hypothesis, suppose that $s \in S$, and let fluents(s) denote the set of ground atoms {true(f) : f $\in F(s)$}. For any player $p \in \{0, 1, \cdots, k\}$, $A_p(s)$ is the set of all ground actions a, for which the fact legal(p, a) is derivable from the program G $\cup$ fluents(s). Any action profile $a = (\mathtt{a}_0, \mathtt{a}_1, \cdots, \mathtt{a}_k) \in A(s)$ determines a successor $s' = Q(s, a)$ of s in S, and $F(s')$ is the set of ground all fluents f, for which the fact next(f) is derivable from the program G $\cup$ fluents(s) $\cup$ {does(p, $\mathtt{a}_\mathtt{p}$) : $\mathtt{a}_\mathtt{p} \in a$}. According to the specifications of GDLII, chance events are uniformly distributed,[1] which implies that $P(s)$ is the uniform distribution over $A_0(s)$. Finally, S_{goal} is the set of states s for which terminal is derivable from G$\cup$fluents(s). In this case, the reward of player p at s is given by the constant r occurring in the fact goal(p, r) derived from the program G $\cup$ fluents(s).

Recall that in General Game Playing, any game starting at the initial state s_{init} must reach a terminal state in S_{goal} in finite time (i.e. using a finite number of moves). Thus, the stochastic game G of a "valid" GDL program G must be acyclic. This, together with the fact that any node $s \in S$ is, by inductive construction, connected to s_{init} implies that G is a tree-like stochastic game.

[1]The actions of the chance player are not necessarily equiprobable. For example, a loaded dice with a probability 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).

3 A fragment of SCSP for GDL

From a game-theoretic viewpoint, the stochastic constraint networks investigated in [12, 26, 30] capture one-player stochastic games, in which the chance player (defined over stochastic variables) is "oblivious": at each round of the game, the probability distribution over the chance player's moves is independent from the description of the current state. In order to encode GDL programs into stochastic constraint programs, we shall examine in this section a slight generalization of the original SCSP model that captures multiplayer and non- oblivious stochastic games.

3.1 Stochastic CSPs

Recall that a (valued) constraint network consists in a finite tuple $V = (v_1, \cdots, v_n)$ of *variables*, a function D that associates with each variable $v_i \in V$ a finite *domain* $D(v_i)$ capturing the set of values that v_i can take, and a set C of constraints, which can be divided into "hard" constraints expressing restrictions on possible variable assignments, and "soft" constraints, assigning utilities to variable assignments. Given a subset of variables $U = (v_1, \cdots, v_m) \subseteq V$, we denote by $D(U)$ the relation $D(v_1) \times \cdots \times D(v_m)$.

Definition 2 A k-player *Stochastic Constraint Satisfaction Problem (SCSP)* is a 6-tuple $N = \langle V, Y, D, C, P, \theta \rangle$, such that $V = (v_1, \cdots, v_n)$ is a finite tuple of variables, $Y \subseteq V$ is the set of *stochastic variables*, D is a mapping from V to finite *domains*, C is a set of *constraints*, P is a set of *conditional probability tables*, and $\theta \in [0, 1]^k$ is a *threshold.*

– Each constraint in C is a pair $c = (scp_c, val_c)$, such that scp_c is a subset of V, called the *scope* of c, and val_c is a map from $D(scp_c)$ to $([0, 1] \cup \{-\infty\})^k$.
– Each conditional probability table in P is a triplet $(y, scp_y, prob_y)$, where $y \in Y$ is a stochastic variable, scp_y is a subset of variables occurring before y in V, and $prob_y$ is a map from $D(scp_y)$ to a probability distribution over $D(y)$.

In what follows, we shall often adopt standard notations from probabilistic models. Notably, if $y \in Y$ is a stochastic variable and $\tau \in D(scp_y)$ is a tuple of values in the conditional probability table of y, then we denote by $P(y \mid \tau)$ the probability distribution $prob_y(\tau)$. In particular, if $d \in D(y)$, then $P(y = d \mid \tau)$ indicates the probability that y takes value d given τ.

By X, we denote the set $V \setminus Y$ of *decision variables*. A constraint $c \in C$ is called a *decision constraint* iff its scope is restricted to decision variables, that is, $scp_c \subseteq X$; otherwise c is called a *stochastic constraint*. c is called a *hard constraint* if the range of val_c is $\{0, -\infty\}^k$, and c is called a *soft constraint* if the range val_c is $[0, 1]^k$. With the intuitive meaning that $-\infty$ is the "forbidden" value, every hard constraint c can be represented in the usual way by a relation, denoted rel_c, which lists the set of allowed tuples for scp_c, that is, $rel_c = \{\tau \in D(scp_c), val_c(\tau) = (0, \cdots, 0)\}$.

Given a subset $U = (v_1, \cdots, v_m) \subseteq V$, an *instantiation* on U is an assignment I of values $d_1 \in D(v_1), \cdots, d_m \in D(v_m)$ to the variables $v_1, \cdots, v_m$, that is, I is a tuple of $D(U)$. We use the notation $I = \{(v_1, d_1), \ldots, (v_m, d_m)\}$, to indicate that I is an instantiation on $\{v_1, \cdots, v_m\}$, that associates the value $d_i \in D(v_i)$ with the variable v_i. An instantiation I on U is *complete* if $U = V$. Given a subset $U' \subseteq U$, we denote by $I_{|U'}$ the restriction of

I to U', that is, $I_{|U'} = \{(v_i, d_i) \in I : v_i \in U'\}$. The *probability* of an instantiation I on U is given by

$$P(I) = \prod_{y \in Y : scp_y \subseteq U} P(y = I_{|y} \mid I_{|scp_y})$$

Correspondingly, the *utility* of an instantiation I on U is given by

$$val(I) = \sum_{c \in C : scp_c \subseteq U} val(I_{|scp_c})$$

Note that $val(I)$ is a tuple $(val_1(I), \cdots, val_k(I))$ assigning a utility to each player. An instantation I is *locally consistent* if $val_p(I) \neq -\infty$ for every player p, that is, I satisfies every hard constraint in C. I is *globally consistent* (or *consistent*) if it can be extended to a complete instantiation I' which is locally consistent.

A *policy* π for the network N is a labeled tree inductively defined as follows. The root of π is labeled by v_1. For each internal node s of π, if s is labeled by a decision variable $v_i = x_i$, then s has a unique successor s' labeled by v_{i+1}, and the edge (s, s') is labeled by a value $d_i \in D(x_i)$. Alternatively, if s is labeled by a stochastic variable $v_i = y_i$ with domain $D(y_i) = \{d_1, \cdots, d_m\}$, then s has m successors $\{s_1, \cdots, s_m\}$; each s_i is labeled by v_{i+1}, and its incident edge (s, s_i) is labeled by the corresponding value d_i. Finally, each leaf s in π is labeled by the utility $val(I)$, where I is the complete instantiation specified by the path from the root of π to the leaf s. Let $\mathcal{L}(\pi)$ be the set of all complete instantiations induced by π, i.e. $I \in \mathcal{L}(\pi)$ iff there is a leaf s of π such that I is the path from the root to s. The *expected utility* of π is the sum of its leaf utilities weighted by their probabilities. Formally,

$$val(\pi) = \sum_{I \in \mathcal{L}(\pi)} P(I) val(I)$$

A *policy* π is *feasible* iff $val(\pi) \geq (0, \cdots, 0)$. In other words, π is feasible iff all paths in π are globally consistent. Finally, π is a *solution* of the stochastic constraint network N if its expected utility is greater than or equal to the threshold $\theta = (\theta_1, \cdots, \theta_k)$. This implies that $val_p(\pi) \geq \theta_p$ for all players p. Clearly, any solution policy is feasible, but the converse is not necessarily true. A network N is *satisfiable* if it admits at least one solution policy.

It is important to keep in mind that a solution policy π for a stochastic constraint network N is *not* guaranteed to be a "dominant strategy" for some of the players [24]. In fact, the notion of "policy" investigated in this study is not equivalent to the definition of "game tree", in which the set of decision nodes would have been partitioned into k subsets, each associated with a specific player. Instead, the overall goal of stochastic constraint satisfaction is to find an assignment of decision nodes for which the resulting expected utility matches the threshold criterion.

Example 2 We consider here a conceptually simple `SCSP` that captures the semantics of the cooperative "Matching Pennies" game, specified in Example 1. The network is defined over the tuple of variables $\langle x_{1,\mathtt{a}}, x_{1,\mathtt{b}}, y_1, x_{2,\mathtt{a}}, x_{2,\mathtt{b}}, y_2 \rangle$, where the decision variables $x_{1,\mathtt{a}}$ and $x_{2,\mathtt{a}}$ (resp. $x_{1,\mathtt{b}}$ and $x_{2,\mathtt{b}}$) specify the choices of `alice` (resp. `bob`) at rounds 1 and 2, and the stochastic variables y_1 and y_2 describe the behavior of the chance player during both rounds. Using the values `U` (unset), `H` (heads), and `T` (tails), the domains of $x_{t,p}$ are

$\{\mathtt{H}, \mathtt{T}\}$ for $t \in \{1, 2\}$ and $p \in \{\mathtt{a}, \mathtt{b}\}$, the domain of y_1 is $\{\mathtt{U}\}$, and the domain of y_2 is $\{\mathtt{H}, \mathtt{T}\}$, equipped with the uniform distribution ($P(y_2 = \mathtt{H}) = P(y_2 = \mathtt{T}) = 1/2$). The decisions of `alice` and `bob` cannot be changed during the second round, which is captured by the hard constraints c_p ($p \in \{\mathtt{a}, \mathtt{b}\}$):

$$c_p(x_{1,p}, x_{2,p}) = \begin{cases} 0 & \text{if} x_{1,p} = x_{2,p} \\ -\infty & \text{otherwise} \end{cases}$$

The game scores are encoded by the soft constraint c_s specified in Fig. 2a. Using $\theta = 1/2$, the policy π depicted in Fig. 2b is a solution: it is consistent with both $c_{\mathtt{a}}$ and $c_{\mathtt{b}}$, and satisfies c_s with an expected utility of $1/2 = \theta$.

Borrowing the terminology of [12], a *(decision) stage* in a SCSP is a tuple of variables $\langle X_t, Y_t \rangle$, where X_t is a subset of decision variables, Y_t is a subset of stochastic variables, and decision variable occurs before any stochastic variable.

Definition 3 A T-stage k-player SCSP is a k-player SCSP$N = \langle V, Y, D, C, P, \theta \rangle$, in which V can be partitioned into T *stages*, i.e. $V = (\langle X_1, Y_1 \rangle, \cdots, \langle X_T, Y_T \rangle)$, where $\{X_t\}_{t=1}^{T}$ is a partition of $V \setminus Y$, $\{Y_t\}_{i=1}^{T}$ is a partition of Y, and $scp_{y_i} \subseteq X_t$ for each $t \in \{1, \cdots, T\}$ and each $y_t \in Y_t$. If $T = 1$, N is called a *one-stage* SCSP, and denoted μSCSP.

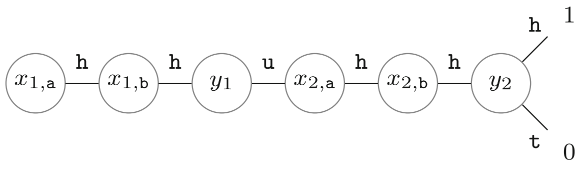
From a computational viewpoint, the satisfiability problem for any T-stage k-player SCSP is PSPACE-hard, since it includes as a particular case the T-state one-player SCSP[30]. On the other hand, the complexity of a k-player μSCSP is only NP$^{\text{PP}}$-complete. This follows from the NP$^{\text{PP}}$-hardness of 1-player μSCSP[30], and the fact that, for a one-stage SCSP, the decision nodes of a solution policy π can be non-deterministically guessed in polynomial time (NP), and the expected reward of π can be checked in probabilistic polynomial time (PP).

3.2 From GDL to SCSP

In [13], we developed a procedure for encoding GDL games in SCSPs. The procedure takes as input a GDL program G and a horizon T, and returns as output a T-stage SCSPN, each decision stage $\langle X_t, Y_t \rangle$ capturing a "round" of the sequential game.

$x_{2,\mathtt{a}}$	$x_{2,\mathtt{b}}$	y_2	c_s
H	H	H	1
T	T	T	1
H	H	T	0
H	T	H	0
⋮	⋮	⋮	⋮
T	H	H	0

(a)



(b)

Fig. 2 The utility function of the soft constraint c_s (**a**) and a policy (**b**) of the SCSPin Example 2

Specifically, each decision stage of N is a tuple of the form $\langle g_t, \{f_t\}, \{a_t\}, y_t\rangle$, where g_t is a Boolean variable indicating whether the game has reached a terminal (goal) state; $\{f_t\}$ is a set of fluent variables describing the game state at round t; $\{a_t\} = \{a_{t,1}, \cdots, a_{t,k}\}$ is a set of action variables, each $a_{t,p}$ describing the set of legal moves of player p, and $y_t = a_{t,0}$ is the unique stochastic variable describing the set of legal moves of the chance player. Additional decision variables (occurring before y_t) are used to define the conditional probability table of y_t (see below), and to express relationships between game elements (ex: `alleq` in Example 1). As detailed in [13], these variables and their domains are extracted by first eliminating function symbols from `G`, next identifying a variable per atom name, and then filling the domain of the variable by collecting all ground instances of the atom in the program.

The Horn clauses of a `GDL` program `G` can naturally be partitioned into `init` rules describing the initial state, `legal` rules specifying the legal moves at the current state, `next` rules capturing the effects of actions, and `goal` rules defining the players' rewards at a terminal state. `init`, `legal` and `next` rules are encoded into hard constraints in the network N. The constraint relation is extracted in the same way as the domains of variables, by identifying all allowed combinations of constants. Similarly, `goal` rules are encoded by a soft constraint in N; based on their semantics, the players' rewards are set to 0 for any nonterminal state, and to a value in [0, 1] for any terminal state.

Based on the stochastic game of `G`, the conditional probability table of y_t specifies a uniform distribution over the set of action values $D(y_t) = \{d_1, \cdots, d_m\}$ which have a consistent support in the `legal` constraint. To this end, we use a set $\{b_t\} = \{b_{t,1}, \cdots, b_{t,m}\}$ of Boolean variables, each $b_{t,i}$ indicating whether d_i is a legal move, or not. A standard channelling constraint is used to express the correspondence between $\{b_t\}$ and y_t. Based on this encoding, the table is intensionally defined over the scope $scp_{y_t} = \{b_t\}$ by the function $P(y_t = d_i \mid I_{|\{b_t\}}) = I_{|b_{t,i}} / \sum_i I_{|b_{t,i}}$.

The threshold θ can be adjusted according to the desired strategy: starting from the value $\theta = (0, \cdots, 0)$ that allows all feasible policies, one can use the expected value of a solution of the current `SCSP` as a new threshold, which determines a more constrained `SCSP` defined over the same constraints.

3.3 From `SCSP` to μ`SCSP`

As a key point of our framework, the T-stage stochastic constraint network encoding a `GDL` game can be decomposed into a sequence $\langle \mu\mathtt{SCSP}_1, \cdots, \mu\mathtt{SCSP}_T\rangle$ of one-stage stochastic constraint networks. Specifically, let $N = \langle V, Y, D, C, P, \theta\rangle$ be the `SCSP`associated with a `GDL` game, where the variable ordering V is partitioned into T stages $\langle V_1, \cdots, V_T\rangle$, with $V_t = \langle g_t, \{f_t\}, \{a_t\}, y_t\rangle$. Then, each $\mu\mathtt{SCSP}_t$ in the sequence is a tuple $\langle V_t, Y_t, D_t, C_t, P_t, \theta\rangle$, where $V_t = \langle g_t, \{f_t\}, \{a_t\}, y_t, \{f_{t+1}\}\rangle$, Y_t is the restriction of Y to the stochastic variable y_t, D_t and C_t are the restrictions of D and C to the variables in V_t, and P_t is the restriction of P to the conditional probability table of y_t. Though y_t is followed by the set of decision variables $\{f_{t+1}\}$ in the ordering V_t, such variables express fluents for which the value is propagated by the `next` constraint, once the previous action variables $\{a_{t,1}, \cdots, a_{t,k}, y_t\}$ have been instantiated. By construction, the above decomposition induces a partition of the constraint set C of N into T constraint sets C_t, each associated with its $\mu\mathtt{SCSP}_t$. Furthermore, because any "valid" `GDL` program `G` represents a tree-structured stochastic game, every game state can reach a terminal state. Assuming that

N is a correct encoding of G, this implies that any instantiation I which is consistent with the subsequence $\langle \mu\mathtt{SCSP}_1, \cdots, \mu\mathtt{SCSP}_t\rangle$ is guaranteed to be globally consistent for N.

In a nutshell, the stochastic constraint network of a GDL program at horizon T can be decomposed into simpler μSCSP*s*, each associated with a distinct subset of hard constraints. Such a decomposition naturally encourages to solve the GDL game in a sequential way, by iteratively solving each $\mu\mathtt{SCSP}_t$ in the sequence.

To conclude this section by an illustrative example, Fig. 3 describes the *t*th μSCSPreturned by our encoding procedure on the GDL program of Example 1. For the sake of clarity, the identifiers representing variables and domains were renamed. Notably, a, b, r denote the players alice, bob and random, respectively. The (soft) goal constraint ommitted in the figure is simply a reformulation of the constraint c_s in Fig. 2a, using the scope $\{\mathtt{coin}_{t,\mathtt{a}}, \mathtt{coin}_{t,\mathtt{b}}, \mathtt{coin}_{t,\mathtt{r}}\}$.

4 **MAC-UCB**

Based on a fragment of SCSPfor GDL games, we now present our resolution technique called MAC-UCB. As indicated above, the stochastic constraint network of a GDL program is a sequence of μSCSPs, each associated with a game round. For each $\mu\mathtt{SCSP}_t$ in $\{1, \cdots, T\}$, MAC-UCB searches the set of feasible policies by splitting the problem into two parts: a CSPand a μSCSP (smaller than the original one). The first part is solved using the

Variable	Domain
$\mathtt{terminal}_t$	$\{\mathtt{true}, \mathtt{false}\}$
$\mathtt{coin}_{t,p}$	$\{\mathtt{H}, \mathtt{T}, \mathtt{U}\}$
$\mathtt{control}_{t,p}$	$\{\mathtt{true}, \mathtt{false}\}$
$\mathtt{choose}_{t,p}$	$\{\mathtt{heads}, \mathtt{tails}, \mathtt{noop}\}$
$\mathtt{coin}_{t+1,p}$	$\{\mathtt{H}, \mathtt{T}, \mathtt{U}\}$
$\mathtt{control}_{t+1,p}$	$\{\mathtt{true}, \mathtt{false}\}$

Variables and domains ($p \in \{\mathtt{a}, \mathtt{b}, \mathtt{r}\}$)

$\mathtt{control}_{t,\mathtt{r}}$	heads	tails	noop
true	1/2	1/2	0
false	0	0	1

Conditional probability table of $\mathtt{choose}_{t,\mathtt{r}}$

$\mathtt{control}_{t,p}$	$\mathtt{choose}_{t,p}$
true	heads
true	tails
false	noop

legal constraints ($p \in \{\mathtt{a}, \mathtt{b}, \mathtt{r}\}$)

$\mathtt{coin}_{t,\mathtt{a}}$	$\mathtt{coin}_{t,\mathtt{b}}$	$\mathtt{coin}_{t,\mathtt{r}}$	$\mathtt{terminal}_t$
H	H	H	true
H	H	T	true
⋮	⋮	⋮	⋮
T	T	T	true
H	H	U	false
H	U	H	false
⋮	⋮	⋮	⋮
U	U	U	false

terminal constraint

$\mathtt{control}_{t,p}$	$\mathtt{control}_{t+1,p}$
false	true
true	false

$\mathtt{coin}_{t,p}$	$\mathtt{choose}_{t,p}$	$\mathtt{coin}_{t+1,p}$
H	noop	H
T	noop	T
U	noop	U
U	heads	H
U	tails	T

next constraints ($p \in \{\mathtt{a}, \mathtt{b}, \mathtt{r}\}$)

Fig. 3 A μSCSP encoding the GDL program of cooperative Matching Pennies

MAC algorithm and the second part with the FC algorithm dedicated to SCSP. Then, a sampling with confidence bound is performed to estimate the expected utility of each feasible solution of $\mu\mathtt{SCSP}_t$.

4.1 Preprocessing step

Before examining the resolution of the μSCSP, we use some classical preprocessing techniques to improve the efficiency of the resolution step. First, hard constraints with the same scope are merged. Given a μSCSPN, two hard constraints c_i and c_j of N such as $scp_{c_i} = scp_{c_j}$ are converted into a unique constraint c_k such that $rel_{c_k} = rel_{c_i} \cap rel_{c_j}$ and $scp_{c_k} = scp_{c_j} = scp_{c_i}$. Next, we remove all unary constraints (e.g. constraints c such that $|scp_c| = 1$), by projecting their relation onto the domain of the single variable occurring in scp_c, restricted to values allowed by the tuples of the associated relation. We also remove the so-called *universal* variables. Recall that a variable is universal in c if whatever the value assigned, c is always satisfied. Formally, 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 with the constraint c_i, where $scp_{c_i} = scp_c \setminus \{x\}$. Such variables (induced by the encoding of a GDL game in SCSP) are removed from the scope of the constraints. The last preprocessing technique is to exploit the Singleton Arc Consistency (SAC) [8] property. A constraint network N is singleton arc- consistent if each value (of each variable) of N is singleton arc- consistent. A value is singleton arc-consistent if when assigned to its variable it does not lead to an arc-inconsistent network. By application of this property, inconsistent values are removed from the domain of variables. These preprocessing techniques are performed on all μSCSPs, except the last one (in which the rewards are revealed). SAC is performed on a CSP extracted from the μSCSP; the model of this CSP is detailed in the next section.

4.2 Resolution step

After performing the preprocessing step, the aim of the resolution step is to enumerate the feasible policies of the μSCSP, some of which can lead to an optimal solution. To the best of our knowledge, the best method is Forward Checking (FC) presented in [2]. Unfortunately for a μSCSP with many constraints, FCis not efficient enough, due to its low pruning capabilities. Instead, we split the μSCSPN into a CSPN' including all decision constraints of N, and a μSCSPN'' containing only stochastic constraints of N. The feasible solutions of the μSCSP are then identified by merging the solutions the CSPN' with the solutions of the μSCSPN''.

We first examine the resolution of N''. For GDL programs, N'' includes a single constraint capturing the transition rule for the chance (random) player. Thus, N'' can be solved using Forward Checking (FC) adapted to one side SCSPs [2].[2] The set of the solution policies obtained by FCon N'' is encoded into a hard constraint c_s, called *feasibility constraint*, where scp_{c_s} is the set of the decision variables of N'', and rel_{c_s} is the set of tuples corresponding to assignments of decision variables that are part of a feasible policy.

We now turn to the resolution of the CSPN'. Here, the classical MAC algorithm [20, 21] is applied to enumerate solutions of N'. Recall that MAC interleaves inference and search,

[2]Specifically, our version of FC returns all solution policies of N'', whereas the original algorithm returns a satisfaction threshold.

since at each step of a depth-first search with backtracking, the Arc Consistency (AC) [15, 16] property is maintained. In order to take into account the solutions identified in N'', we simply add the corresponding feasibility constraint c_s to N'. The MAC algorithm is then applied on N', and returns a set of solutions which is guaranteed to coincide with the set of solutions of the original μSCSP (when adding the stochastic variable). MACexploits the arc consistency property in order to effectively prune infeasible solutions of N'. We note in passing that it is also possible to first solve the CSPN' (without the feasibility constraint) and next to process the μSCSPN''. However, due to the small size of N'', it is more effective in practice to first solve this problem, before proceeding to the larger problem N'.

Example 3 We illustrate the resolution of a μSCSPN defined by the decision variables x_1 and x_2, and the stochastic variable y. The domains are $D(x_1) = \{1, 2, 3\}$ and $D(y) = D(x_2) = \{0, 1, 2\}$. The probability distribution N over $D(y)$ is uniform, and the threshold θ is set to 3/4. The network includes three constraints specified in Fig. 4a. The μSCSPN'' is restricted to the constraints $\{c_1, c_2\}$. For this problem, FCreturns the policies π_1 and π_2 described in Fig. 4. The feasibility constraint c_s is added to N' with $scp_{c_s} = \{x_1, x_2\}$ and $rel_{c_s} = \{(2, 1), (2, 2)\}$. The problem N', associated with the CSP part of N, is thus defined by the set of (decision) variables $\{x_1, x_2\}$ (with their associated domain) and the constraints $\{c_3, c_s\}$. The MAC algorithm returns for this problem the unique solution: $(x_1 = 2\,; x_2 = 2)$.

Thus, by combining the solutions obtained from N' and N'', it follows that the unique solution policy of N is π_1.

4.3 UCB

Though any GDLprogram represents a finite sequential game, the players' rewards are only accessible at a terminal state. So, after each resolution of a μSCSP$_t$, we need to simulate the next states of the game in order to estimate the utility of solutions found in μSCSP$_t$. To this end, we use the multi-armed bandits UCB (Upper Confidence Bound) technique [1], by considering each feasible solution of μSCSP$_t$ as an "arm". Starting from the partial policy associated with a feasible solution of μSCSP$_t$, we sample uniformly at random all possible moves from $t+1$ to $T-1$. The "best" feasible solution of μSCSP$_t$ is the one that maximizes $\bar{u}_i + \sqrt{\frac{2 \ln n}{n_i}}$, where $\bar{u}_i$ is the averaged score of the feasible solution i, n_i is the number of times i has been sampled so far, and n is the overall number of samples.[3] The resolution of the next problem in the sequence is performed by instantiating μSCSP$_{t+1}$ with the values of the best feasible solution estimated from μSCSP$_t$.

4.4 Pruning improvements

Recall that the task of sequential decision making associated with a strategic game is an optimization problem. Classically, this problem is addressed by solving a sequence of stochastic satisfaction problems whose threshold is gradually increased. Starting from the threshold $\theta = (0, \cdots, 0)$, if $r = (r_1, \cdots, r_k)$ is the tuple of expected rewards of the best policy estimated by UCBover the sequence $\langle \mu\text{SCSP}_1, \cdots, \mu\text{SCSP}_t \rangle$, then θ is reset to $(r_{\min}, \cdots, r_{\min})$, where $r_{\min} = \min\{r_i\}$.

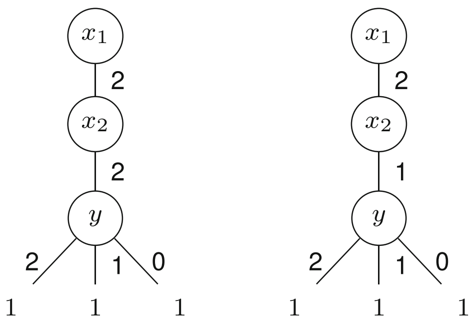
[3]For our experiments, 10000 simulations were performed.

$$c_1(x_1, y) = \begin{cases} 1/2 & \text{if } x_1 + y > 1 \\ 0 & \text{otherwise} \end{cases}$$

$$c_2(y, x_2) = \begin{cases} 1/2 & \text{if } y + x_2 \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

$$c_3(x_1, x_2) = \begin{cases} 0 & \text{if } x_1 = x_2 \\ -\infty & \text{otherwise} \end{cases}$$

(a) constraints



(b) π_1 (c) π_2

Fig. 4 Constraints of the μSCSPN, and solutions of the μSCSPN''

The gradual increase of the threshold is exploited to prune the search space. To this end, our UCB implementation uses a caching technique which stores the leaves already explored.[4] By combining the leaves already encountered with the number of legal moves at each state, one can determine whether a subtree is completely explored. If this is indeed the case, any solution of the μSCSP whose expected value is less than θ can be safely removed. Based on this cutting scheme, UCB can exploit its cache to remove in each μSCSP the suboptimal solutions for θ. We also take advantage of the confidence values in UCB, which estimate the quality of each sampled solution. When a sufficient number of domain values of the stochastic variable are below the confidence value, the corresponding subtree can be safely removed from the solution set of the stochastic subproblem N''. Correspondingly, the number of tuples in the relation of the feasibility constraint c_s is reduced, which simplifies the resolution of the CSP. In other words, the higher the threshold θ is, the more efficient is the resolution of the μSCSP.

5 Experimental results

We now present some experimental results conducted on a cluster of Intel Pentium CPU 3.4 GHz with 32 GB of RAM under Linux. Our framework was implemented in C++ and we did not use any other tools.

We selected 15 games described in GDLII, including both deterministic games and stochastic games. Game descriptions differ in the number of players, the number of fluents and moves, the number and size of rules, and the scoring function (goal).

- *Awale/Oware* is an abstract strategy (board) game with 48 seeds and 2 straight rows of 6 pits called ”house”. Each player controls the 6 houses on their side of the board. The goal of is to capture a maximum of seeds before the opponent.
- *Backgammon* is a board game with 2 dices and 15 pieces by player. The playable pieces are moved according to the roll of the dice, and a player wins by removing all of her pieces from the board.

[4]In our experiments, 32 GB were allowed for caching and this limit was never reached.

- *Bomberman* is a strategy maze-based video game where the goal is to complete levels by placing bombs in order to kill enemies and destroy obstacles.
- *Can't stop* is a board game with 4 dices. The board includes 9 columns of different sizes, and the goal is to reach the top of three of them with the right combinations of dices.
- *Checkers* is a strategy 8×8 board game involving in diagonal moves of uniform game pieces and mandatory captures by jumping over opponent pieces.
- *Chess* is a strategy board game with 64 squares where two players move 16 different pieces in order to "checkmate" the opponent's king by placing it under an inescapable threat of capture.
- *Chinese Checkers* is a strategy board game where the objective is to be the first to race one's pieces across the hexagram-shaped board into the corner of the star opposite one's starting corner. Our version has 3 players.
- *Hex* is an alternating move game played on a 9×9 board. On each step, one of the players places a colored marker on an open hexagon. The goal is for the player to form a path of markers of its color connecting one side of the board to the other.
- *Kaseklau* is a small board game involving a mouse and a cat. The goal is to roll the 2-dice to move the mouse and the cat on different squares where slices of cheese are placed.
- *Orchard* is a cooperative board game. During each round, each player rolls a 6-dice, which includes 4 faces associated with a specific fruit tree, one face associated with a set of pieces composing a raven, and one face (the "basket") allowing the player to remove two fruits of her choice. The goal is to pick all fruits from the trees before removing all pieces of the raven.
- *Othello/Reversi* is a strategy board game, played on an 8×8 uncheckered board. There are 64 disks with one black face and one white face. Every player is assigned to a color. During a move, all disks of the opponent which are flanked by the disks of the current player are flipped, and hence, assigned to the current player. The winner is the player who has more discs of his colour than his opponent when the last playable empty square is filled.
- *Pacman* is an arcade game where the player controls Pac-Man through a maze, eating pac-dots and fruits, and avoiding 4 roaming ghosts. The player loses if a ghost touches Pac-Man before all pac-dots are eaten, and wins otherwise.
- *Pickomino* is a dice game with 8 dices and 16 numbered tiles including 1 at 4 worms. At each round, the players obtain a score by rolling dices. The goal is to get the maximum of worms.
- *Tic-tac-toe* is a well-known deterministic game with two players (X and O) who iteratively mark a 3×3 grid.
- *Yathzee* is a game where the goal is to get the highest score by rolling five dices. At each round, the dice can be rolled up to 3 times in order to make one of the 13 scoring combinations.

The translation of these GDL games into SCSP is summarized in Table 2, which indicates the number of variables (#vars), the maximum domain size (maxDom), and the number of constraints (#const) of the resulting SCSP, together the parsing time in seconds (time) for constructing this SCSP. The most difficult game to is Backgammon, involving a large number of variables, each with a large domain and an important number of constraints with

Table 2 The games translated into SCSPand theirs parameters

Game	#vars	maxDom	#const	time
Awale	19	37	73	94
Backgammon	76	768	86	347
Bomberman	145	64	42	31
Can't Stop	16	1296	409	248
Checkers	86	262144	52	43
Chess	71	4096	50	76
Chinese Checkers	103	192	87	54
Hex	22	6561	27	91
Kaseklau	18	7776	106	35
Orchard	9	146410	40	12
Othello	81	65	29	51
Pacman	93	64	22	36
Pickomino	29	1679616	223	172
TicTacToe	19	18	37	0
Yathzee	12	30	8862	182

large scope. Awale, Can't Stop, Chess, Hex, Pickomino and Yathzee are also challenging due to the size of their domains or the number of their constraints.

5.1 Setup

Game competitions were organized between three players. The first player is the state-of-the-art UCT algorithm. The second player is the FC-UCB algorithm which solves μSCSPs using only the FCalgorithm. The last player is the MAC-UCB algorithm, which solves μSCSPs by decomposing them into two parts, and running MAC on the deterministic part, as indicated in Section 4.2. We have implemented UCT, following the specification of the multi-player version [25]. For the sake of fairness, we also added a cache to UCT, allowing it to know in advance the subtrees already explored. We realized 1,800,000 instances of duels between UCT, MAC-UCB and FC-UCB. For each game, a player follows the strategy UCT, FC-UCB or MAC-UCB. 5000 match plays are realized with different deliberation times per round (1s, 5s, 10s, 20s, 30s, 40s, 50s, 60s).

The horizon T was fixed to the maximum number of turns that can be sampled by UCB during a given deliberation time. If a goal state is reached before T turns, this state and all subsequent states are considered as terminal. If no goal state is reached at (or before) T, the state at T is considered irrelevant (with a reward of 0 to all players).

5.2 Results

In Table 3 are reported the percentage of wins obtained by MAC-UCB (or FC-UCB when MAC-UCB is not used) for each game, with 30 seconds per move. The standard deviation (σ) is also indicated. For all games, MAC-UCB statistically outperforms both UCT and FC-UCB, and this phenomenon increases with deliberation time.

Table 3 Results for several GDLgames with 30s by move

Game	MAC-UCB vs. UCT	σ	MAC-UCB vs. FC-UCB	σ	FC-UCB vs. UCT	σ
Awale	56.7 %	1.63 %	77.7 %	1.92 %	43.2 %	2.34 %
Backgammon	68.2 %	5.49 %	84.8 %	6.36 %	47.3 %	5.87 %
Bomberman	65.4 %	6.32 %	75.7 %	5.34 %	58.4 %	5.46 %
Can't Stop	71.7 %	6.43 %	65.9 %	4.87 %	54.7 %	5.34 %
Checkers	61.2 %	2.12 %	76.9 %	1.61 %	57.5 %	1.43 %
Chess	53.8 %	1.75 %	68.5 %	1.76 %	39.4 %	1.75 %
Chinese checkers[a]	55.4 %	8.24 %	78.1 %	7.23 %	32.7 %	6.51 %
Hex	69.7 %	2.45 %	73.2 %	3.24 %	55.3 %	3.12 %
Kaseklau	71.4 %	6.56 %	58.9 %	7.87 %	68.3 %	7.34 %
Orchard[b]	70.2 %	3.41 %	70.2 %	3.45 %	70.0 %	2.45 %
Othello	79.3 %	1.41 %	75.1 %	0.89 %	61.3 %	1.14 %
Pacman	69.1 %	2.73 %	74.1 %	3.12 %	61.8 %	3.78 %
Pickomino	63.4 %	4.89 %	65.9 %	6.10 %	52.1 %	5.34 %
Tictactoe	65.7 %	0.89 %	51.8 %	0.76 %	64.9 %	0.73 %
Yathzee	71.2 %	5.48 %	74.0 %	5.12 %	58.6 %	5.34 %

[a]This game involves three players : one controls by MAC-UCB or FC-UCB and the two others by two UCT or FC-UCB

[b]Since this game is cooperative, the cooresponding row indicates the percentage of victories using the same algorithm for all players

The leftmost part of the table (MAC-UCB vs. UCT) indicates that MAC-UCB is particularly efficient for handling stochastic games. Indeed, for *Bomberman*, *Can't Stop*, *Kaseklau*, *Pacman*, *Pickomino*, *Yathzee*, and *Backgammon*, MAC-UCB wins more than 70% of match plays. For deterministic games, the standard deviation is smaller, because MAC-UCB cannot benefit from stochastic pruning.

It is important to emphasis the specificity of two games: Orchard and Chinese Checkers. The former is a cooperative game, and the performance of both algorithms (MAC-UCB and UCT) is around 70%. We note in passing that this score is maximal for the optimal

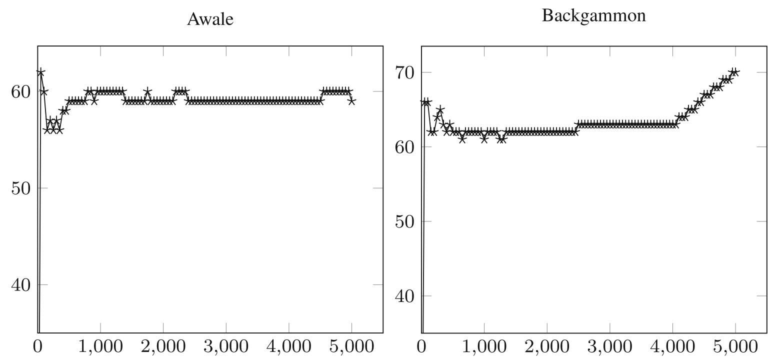


Fig. 5 Number of match plays (*horizontal axis*) versus ratio of victories (*vertical axis*) for MAC-UCB against UCT, using 30 seconds per move

Table 4 Ratio of victories for MAC-UCB vs. UCTwith different deliberation time per round on 5000 plays

Game	1 s	5 s	10 s	20 s	30 s	40 s	50 s	60 s
Awale	38.6 %	43.3 %	47.4 %	**51.6** %	56.7 %	57.9 %	61.3 %	63.0 %
Backgammon	**53**.1 %	58.1 %	61.4 %	65.8 %	68.2 %	71.3 %	77.0 %	79.6 %
Bomberman	**58.3** %	60.0 %	60.2 %	64.7 %	65.4 %	70.1 %	72.4 %	75.6 %
Can't Stop	**50.3** %	54.7 %	59.9 %	62.2 %	71.7 %	74.9 %	76.3 %	78.9 %
Checkers	43.4 %	49.6 %	**51.4** %	56.8 %	61.2 %	67.2 %	71.3 %	75.7 %
Chess	34.0 %	39.3 %	46.1 %	49.9 %	**53.8** %	56.4 %	57.6 %	60.8 %
Chinese Checkers	27.4 %	35.5 %	43.7 %	**50.7** %	55.4 %	59.1 %	63.2 %	65.4 %
Hex	**54.7** %	56.2 %	58.5 %	67.4 %	69.7 %	71.4 %	71.9 %	72.5 %
Kaseklau	**63.6** %	65.2 %	68.3 %	70.2 %	71.4 %	73.2 %	74.7 %	75.1 %
Orchard	**65.3** %	68.5 %	69.9 %	70.2 %	70.2 %	70.1 %	70.2 %	70.2 %
Othello	**61.9** %	64.0 %	70.6 %	75.8 %	79.3 %	82.0 %	84.2 %	84.9 %
Pacman	**64.4** %	66.2 %	67.1 %	67.9 %	69.1 %	69.4 %	69.8 %	70.5 %
Pickomino	**52.4** %	55.3 %	58.0 %	61.1 %	63.4 %	65.8 %	66.1 %	68.6 %
TicTacToe	**63.6** %	64.4 %	64.9 %	65.4 %	65.7 %	65.9 %	65.8 %	65.4 %
Yathzee	43.5 %	**50.1** %	53.7 %	64.3 %	71.2 %	75.2 %	77.1 %	78.9 %

The significance of bold corresponds to the first necessary time to win by average with MAC-UCB

strategy. For Chinese Checkers, a three player game, we observe that even if MAC-UCB is the winner in 55 % of plays against "two" UCT, the standard deviation is important ($> 8\%$). So, we cannot statistically confirm that MAC-UCB is efficient enough to outperform two UCT adversaries.

For the middle part of the table (MAC-UCBvs. FC-UCB), it is clear that MAC-UCB dominates FC-UCB. Thus, the more aggressive pruning technique used by MAC-UCB is paying off: the arc consistency property maintained by the algorithm has a significant impact for efficiently solving μSCSPs. The rightmost part of the table (FC-UCBvs. UCT) indicates that even if FC-UCB outperforms UCT for some games with few constraints or variables, UCT is the winner in the majority of cases. Thus, in light of the three columns of the table, the effectiveness of MAC (coupled with UCB) is crucial for quickly finding winning policies.

Figure 5 reports the evolution of the victories for MAC-UCB against UCT with 30s per move when the number of plays increases up to 5000. For the deterministic *Awale* game, the evolution is almost constant with a standard deviation of 1,41% for 1000 plays. In such a deterministic case, pruning is only realized by MAC on the CSP part. For the stochastic *Backgammon* game, the performance of MAC-UCB is better and significantly increases with the number of plays. This can be explained by the increasing number of prunings induced by UCB, whose cache is more and more exploited. The same phenomenon can be observed for other stochastic games.

Table 4 describes the percentage of victories of MAC-UCB against UCT with different deliberation times per round, ranging from 1 second to 60 seconds on 5000 plays. We can observe that the performance gap between MAC-UCB and UCT increases with deliberation time. This gap is explained by the ability of MAC to solve a more important number of μSCSPs in the sequence, which makes easier the exploration task of UCB. For the smallest game *TicTacToe*, all the search tree is explored using only 5s per move. Moreover, for small games like *Bomberman* or *Pacman*, MAC-UCB can win using only 1s per move. Contrastingly, for some larger games like *Awale*, *Chess* or *Chinese Checkers*, MAC-UCB is

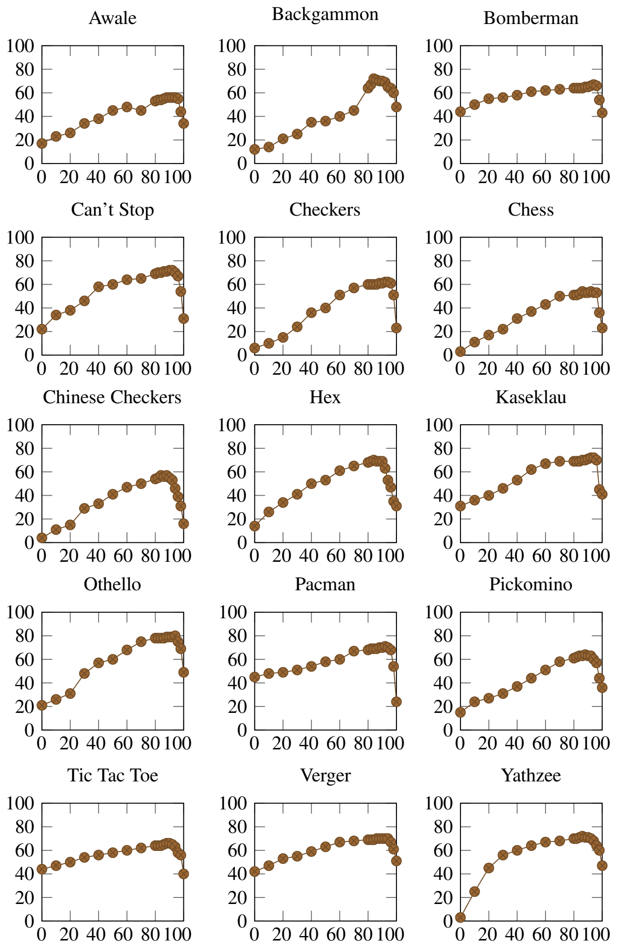


Fig. 6 Sensitivity analysis of MAC-UCB:. On the *vertical axis*, the percentage of victories of MAC-UCB vs. UCT, and on the *horizontal axis*, the percentage of resolution during the deliberation time (30s)

defeated by UCTwhen the deliberation time is too small (typically less than 10s). However, MAC-UCB obtains better results for stochastic games by exploiting stochastic pruning. Notably, for the *Orchard* game, the optimal strategy is discovered using only 20s per move.

We conclude the experimental analysis by shedding a light on the dilemma for MAC-UCB between exploitation (solving) and exploration (sampling). In our experiments, 90 % of deliberation time was dedicated to exploitation and 10 % to exploration. In order to justify this ratio, Fig. 6 shows a sensitivity analysis of MAC-UCB for the different games, using 30s per move.[5] The plots report the percentage of victories of MAC-UCB for ratios ranging from 0 % to 100 % for the solving part. The optimum is reached between 86 and 94 %, which stresses the importance of focusing mainly on the structure of the games, captured by the hard constraints.

[5]A similar phenomenon was observed using 10s and 50s per move.

6 Conclusion

In this paper, we identified a fragment of SCSP for representing GDL games with uncertain and complete information. Based on this fragment, we proposed an algorithm, MAC-UCB, that searches solution policies by combining a Constraint Programming method (MAC) with a multi-armed bandit method (UCB). Extensive experiments on various games, with different deliberation times per round, highlight the ability of MAC-UCB to address the GGP challenge. In most cases, MAC-UCB outperforms UCT, the reference in the field of games with uncertain (but complete) information.

This work paves the way for many research opportunities. From an algorithmic viewpoint, the resolution step could be improved by exploiting symmetries, and arc consistency methods dedicated to SCSPs (ex: [2]). Another natural perspective of research is to extend the approach to games with incomplete information.

References

1. Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*(2–3), 235–256.
2. Balafoutis, T., & Stergiou, K. (2006). Algorithms for stochastic CSPs. In Proc. of CP'06 (pp. 44–58).
3. Bessiere, C., & Verger, G. (2006). Strategic constraint satisfaction problems. In Proc. of CP'06 workshop on modelling and reformulation (pp 17–29).
4. Bowling, M., Burch, N., Johanson, M., & Tammelin, O. (2015). Heads-up limit hold'em poker is solved. *Science*, *347*(6218), 145–149.
5. Campbell, M., Hoane Jr., A.J., & Hsu, F. (2002). Deep blue. *Artificial Intelligence*, *134*(1–2), 57–83.
6. Cazenave, T., & Mehat, J. (2010). Ary, a general game playing program. In Proc. of board games studies colloquium.
7. Clune III, J.E. (2008). Heuristic evaluation functions for general game playing. PhD thesis, University of California, Los Angeles, USA. Adviser-Korf, Richard E.
8. Debruyne, R., & Bessiere, C. (1997). Some practical filtering techniques for the constraint satisfaction problem. In Proc. of IJCAI'97 (pp. 412–417) Springer.
9. Finnsson, H., & Björnsson, Y. (2008). Simulation-based approach to general game playing. In Proc. of AAAI'08 (pp. 259–264).
10. Genesereth, M., Love, N., & Pell, B. (2005). General game playing: overview of the AAAI competition. *AAAI Magazine*, *26*(2), 62–72.
11. Gent, I.P., Nightingale, P., Rowley, A., & Stergiou, K. (2008). Solving quantified constraint satisfaction problems. *Artificial Intelligence*, *172*(6-7), 738–77.
12. Hnich, B., Rossi, R., Tarim, S.A., & Prestwich, S.D. (2012). Filtering algorithms for global chance constraints. *Artificial Intelligence*, *189*, 69–94.
13. Koriche, F., Lagrue, S., Piette, É., & Tabary, S. (2015). Compiling strategic games with complete information into stochastic csps. In AAAI workshop on planning, search, and optimization (PlanSOpt-15).
14. Love, N., Hinrichs, T., Haley, D., Schkufza, E., & Genesereth, M. (2008). General game playing: game description language specification. Technical report.
15. Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, *8*, 99–118.
16. Mackworth, A. (1977). On reading sketch maps. In Proc. of IJCAI'77 (pp. 598–606). Springer.
17. Möller, M., Schneider, M.T., Wegner, M., & Schaub, T. (2011). Centurio, a general game player: parallel, Java- and ASP-based. *Künstliche Intelligenz*, *25*(1), 17–24.
18. Neyman, A., & Sorin, S. (Eds.) (2003). *Stochastic games and applications.* Springer.
19. Nguyen, T.-V.-A., Lallouet, A., & Bordeaux, L. (2013). Constraint games: framework and local search solver. In Proc. of ICTAI'13, pp. 8–12.
20. Sabin, D., & Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In Proc. of CP'94 (pp. 10–20). Springer.
21. Sabin, D., & Freuder, E. (1997). Understanding and improving the mac algorithm. In Proc. of CP'97 (pp. 167–181). Springer.

22. Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., & Sutphen, S. (2007). Checkers is solved. *Science*, *317*(5844), 1518–1522.
23. Shannon, C. (1950). Programming a computer for playing chess. *Philosophical Magazine*, *41*, 256–275.
24. Shoham, Y., & Leyton-Brown, K. (2009). Multiagent systems: algorithmic, game-theoretic, and logical foundations. Cambridge University Press.
25. Sturtevant, N.R. (2008). An analysis of uct in multi-player games. *ICGA Journal*, *31*(4), 195–208.
26. Tarim, A., Manandhar, S., & Walsh, T. (2006). Stochastic constraint programming: a scenario-based approach. *Constraints*, *11*(1), 53–80.
27. Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, *134*(1–2), 181–199.
28. Thielscher, M. (2005). Flux: a logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, *5*(4–5), 533–565.
29. Thielscher, M. (2010). A general game description language for incomplete information games. In Proc. of AAAI'10 (pp. 994–999).
30. Walsh, T. (2002). Stochastic constraint programming. In Proc. of ECAI'02 (pp. 111–115).