Beyond QCSP for Solving Control Problems

Cédric Pralet, Gérard Verfaillie

ONERA – The French Aerospace Lab, F-31055, Toulouse, France
{cedric.pralet,gerard.verfaillie}@onera.fr

Abstract. Quantified Constraint Satisfaction Problems (QCSP) are often claimed to be adapted to model and solve problems such as two-player games, planning under uncertainty, and more generally problems in which the goal is to control a dynamic system subject to uncontrolled events. This paper shows that for a quite large class of such problems, using standard QCSP or QCSP+ is not the best approach. The main reasons are that in QCSP/QCSP+, (1) the underlying notion of system state is not explicitly taken into account, (2) problems are modeled over a bounded number of steps, and (3) algorithms search for winning strategies defined as "memoryfull" policy trees instead of winning strategies defined as "memoryless" mappings from states to decisions. This paper proposes a new constraint-based framework which does not suffer from these drawbacks. Experiments show orders of magnitude improvements when compared with QCSP/QCSP+ solvers.

1 Introduction

Quantified Constraint Satisfaction Problems (QCSP [1]) were introduced to model and solve CSP involving uncertainty or uncontrollability on the value taken by some variables. From a formal point of view, a QCSP is defined by two elements: a set of constraints C, and a quantification sequence $Q = Q_1 x_1 \ldots Q_n x_n$ where each Q_i corresponds to an existential or universal quantifier ($\exists$ or $\forall$). A QCSP defined by $Q = \exists x_1 \forall x_2 \exists x_3 \forall x_4$ and $C = \{x_1 + x_3 < x_4, x_2 \neq x_1 - x_3\}$ is then to be interpreted as "Does there exist a value for x_1 such that for every value taken by x_2 there exists a value for x_3 such that for every value of x_4 constraints $x_1 + x_3 < x_4$ and $x_2 \neq x_1 - x_3$ are satisfied?". Solving a QCSP means answering yes or no to the previous question, and producing a winning strategy if the answer is yes. If $\mathbf{d}(x)$ denotes the domain of variable x and if A_x denotes the set of universally quantified variables that precede x in the quantification sequence, such a winning strategy is generally defined as a set of functions $f_x : \prod_{y \in A_x} \mathbf{d}(y) \rightarrow \mathbf{d}(x)$ (one function per existentially quantified variable x). This set of functions can be represented as a so-called *policy tree*. Various algorithms were defined in the last decade for solving QCSPs, from earlier techniques based on binary or ternary quantified arc-consistency (QAC [1, 2]) or translation into quantified boolean formulas [3], to techniques based on pure value rules, n-ary quantified generalized arc consistency [4], conflict-based backjumping [5], solution repair [6], or right-left traversal of the quantification sequence [7]. Recently, an adaptation of QCSP called QCSP+ [8] was proposed to make QCSP

more practical from the modeling point of view. The idea in QCSP+ is to use restricted quantification sequences instead of standard quantification sequences. The former look like $\exists x_1[x_1 \geq 3]\, \forall x_2[x_2 \leq x_1]\, \exists x_3, x_4[(x_3 \neq x_4) \wedge (x_3 \neq x_1)]\, C$, and must be interpreted as "Does there exist a value for x_1 satisfying $x_1 \geq 3$ such that for every value of x_2 satisfying $x_2 \leq x_1$, there exists values for x_3 and x_4 satisfying $x_3 \neq x_4$ and $x_3 \neq x_1$ such that all constraints in C are satisfied?".

QCSP and QCSP+ can be used to model problems involving a few quantifier alternations such as adversary scheduling problems [8]. They can also be used to model problems involving a larger number of quantifier alternations, such as two-player games or planning under uncertainty. For two-player games, the goal is to determine a first play for player 1 such that for every play of player 2, there exists a play of player 1 such that for every play of player 2 ... player 1 wins the game. The size of the quantification sequence is fixed initially depending on the maximum number of turns considered. Planning under uncertainty can be seen as a game against nature and interpreted similarly. More generally, a large number of quantifier alternations are often useful when QCSP/QCSP+ is used to model problems of control of the state of a dynamic system subject to events.

The goal of this paper is to show that when that state is completely observable at each step and when the evolution of the current state is Markovian (i.e. the state of the system at a given step depends only on the last state and on the last event, and not on the whole history of events), then using pure QCSP/QCSP+ is (currently) not the best approach. The paper is organized as follows. We first illustrate why using pure QCSP/QCSP+ in this context is not always appropriate (Section 2). We then define a new framework called MGCSP for Markovian Game CSP (Section 3), and associated algorithms (Section 4). Experiments show orders of magnitude improvements on some standard QCSP benchmarks (Section 5). Proofs are omitted for space reasons.

2 Illustrating Example

Let us consider a QCSP benchmark called the *NimFibo* game. This game involves two players, referred to as A and B, who play alternatively. Initially, there are N matches on a table. At the first play, player A can take between 1 and N-1 matches. Then, at each turn, each player takes at least one match and at most twice the number of matches taken by the last player. The player taking the last match wins. The problem is to find a winning strategy for player A.

The QCSP/QCSP+ approach Let us assume that N is odd. To model the NimFibo game as a QCSP+, it is first possible to introduce N variables $r_1, \ldots, r_N$ of domain $[0..N]$ representing the number of matches remaining after each turn (N variables because there are at most N turns). It is also possible to introduce decision variables of domain $[1..N-1]$ modeling the number of matches taken by each player at each turn: a_1, a_3, ..., a_N for player A and b_2, b_4, ..., b_{N-1} for

player B. A QCSP+ modeling the NimFibo game is then:

$$\exists a_1, r_1[r_1 = N - a_1]\, \forall b_2, r_2[b_2 \leq 2a_1, r_2 = r_1 - b_2]$$
$$\exists a_3, r_3[a_3 \leq 2b_2, r_3 = r_2 - a_3]\, \forall b_4, r_4[b_4 \leq 2a_3, r_4 = r_3 - b_4] \ldots$$
$$\exists a_N, r_N[a_N \leq 2b_{N\text{-}1}, r_N = r_{N\text{-}1} - a_N]\ \mathit{True}$$

Figure 1(a) gives a winning strategy expressed as a policy tree for $N = 15$ matches. Nodes depicted by circles model all possible decisions of player B. Nodes depicted by squares represent decisions to be made by player A to win the game.

State-based approach Basically, the state of the system at each step is defined by three state variables: one variable $p \in \{A, B\}$ specifying the next player, one variable $r \in [0..N]$ modeling the number of remaining matches, and one variable $l \in [1..N]$ specifying the number of matches taken by the last player. Two kinds of decisions modify the state (p, r, l): plays of player A and plays of player B, modeled respectively by decision variables a and b of domain $[1..N-1]$ representing the number of matches taken by each player at each turn. These decisions are sequentially made. Variable a is controllable by player A whereas variable b is not. The goal is to reach, whatever the plays of player B, a state such as $(B, 0, l)$ in which B cannot take any match, while ensuring that states of the form $(A, 0, l)$ in which A cannot take any match are never reached before. A solution to this control problem can be defined as a decision policy $\pi : \{(A, r, l) \,|\, r \in [1..N], l \in [1..N]\} \rightarrow \mathbf{d}(a)$ associating a value for a with states $s = (A, r, l)$ in which A must play and there is at least one remaining match. Policy π does not need to be specified for all states: it only needs to be specified for states which are reachable using π. A solution policy is given in Figure 1(b). Figure 1(c) gives the states which are reachable using this policy, as well as the possible transitions between states.

Comparison between the two approaches With 15 matches, the policy tree contains 48 leaves and the decision policy contains only 19 (state,decision) pairs, although both policies induce the same sequences of states. The size ratio grows exponentially when the number of matches increases. The main reason is that the Markovian nature of the system considered together with the complete observability of the state at each step entail that the strategy encoded as a policy tree memorizes too many elements. For instance, on the reachability graph of Figure 1(c), the two sequences of plays $seq_1 : [a = 2, b = 1, a = 1, b = 1, a = 2]$ and $seq_2 : [a = 2, b = 3, a = 2]$ are equivalent because they end up in the same state, $(B, 8, 2)$. The only useful information to be taken into account to act from that state is the state itself, and not the entire trajectory used to reach it. Said differently, searching for strategies defined as "memoryfull" policy trees as in QCSP/QCSP+ is searching in a uselessly large search space, since searching for "memoryless" decision policies π is sufficient.

Second, explicitly reasoning over the notion of state enables to memorize whether a state s has already been successfully explored, which means that it has been already proved that the goal can always be reached from s. This is

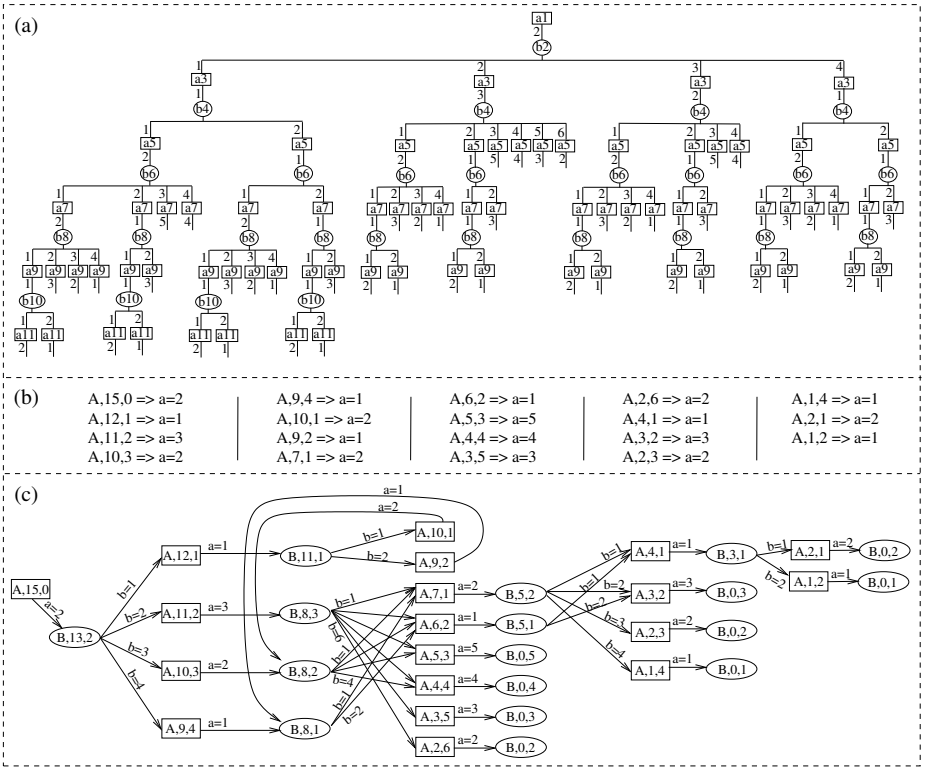


Fig. 1. Comparison between QCSP and state-based models on the NimFibo game: (a) QCSP policy tree; (b) state-based policy; (c) reachability graph using the policy

equivalent to memorizing *goods* over states. For example, when exploring reachable states of Figure 1(c), trajectories rooted in state $(B, 8, 2)$ do not need to be explored twice (once when $(B, 8, 2)$ is reached from $(A, 10, 1)$ and once when $(B, 8, 2)$ is reached from $(A, 10, 3)$). Similarly, if the exploration below state s leads to a dead-end, then s can be recorded as a nogood, potentially reused later during search to avoid exploring s again. These notions of goods and nogoods *over states* are not handled by QCSP solvers, which can only record goods and nogoods *over variables* of the unfolded model. Recording information about states already explored uses the principles of forward dynamic programming [9].

Third, the state-based approach is able to reason over unbounded horizons. It does not require the system evolutions to be unfolded over a fixed number of steps. This leads to models which are more compact than QCSP models.

For all these reasons, we believe that there is a need to introduce a new constraint-based framework based explicitly on the notion of state, in order to efficiently model and solve control problems for completely observable and Markovian dynamic systems.

3 Markovian Game-CSP (MGCSP)

In the following, S denotes the set of variables describing the system state. State variables used here differ from those used in Strategic CSP [10]. Every assignment $s \in \mathbf{d}(S)$ is called a state (given an ordered set of variables X, $\mathbf{d}(X)$ denotes the Cartesian product of the domains of the variables in X). We consider two additional sets of variables denoted C and U respectively, corresponding to the decision made by the $\exists$-player (resp. the $\forall$-player) at each decision step. Every assignment $c \in \mathbf{d}(C)$ (resp. $u \in \mathbf{d}(U)$) is called a controllable (resp. uncontrollable) decision.

3.1 Control Model

To represent the possible initial and final states, we use a relation $I \subseteq \mathbf{d}(S)$ ("init") and a relation $E \subseteq \mathbf{d}(S)$ ("end"). To represent that some decisions $c \in \mathbf{d}(C)$ (resp. $u \in \mathbf{d}(U)$) cannot be made in some states, due to game rules or to physical constraints, we use a relation $F_c \subseteq \mathbf{d}(S) \times \mathbf{d}(C)$ (resp. $F_u \subseteq \mathbf{d}(S) \times \mathbf{d}(U)$) called feasibility relation such that $F_c(s, c)$ (resp. $F_u(s, u)$) holds iff making decision c (resp. u) is possible in state s. The state evolution scheme is defined by a transition function $T_c : \mathbf{d}(S) \times \mathbf{d}(C) \to \mathbf{d}(S)$ (resp. $T_u : \mathbf{d}(S) \times \mathbf{d}(U) \to \mathbf{d}(S)$), such that $s' = T_c(s, c)$ (resp. $s' = T_u(s, u)$) means that s' is the state resulting from the application of decision c (resp. u) in state s.

Three assumptions are made to guarantee that system evolutions cannot be blocked: first, there exists at least one possible initial state s, that is one state such that $I(s) = \mathit{true}$; second, for every state s which is not terminal ($E(s) = \mathit{false}$), there exists at least one feasible decision c (resp. u), that is one decision such that $F_c(s, c) = \mathit{true}$ (resp. $F_u(s, u) = \mathit{true}$); third, for every state s which is not terminal and every decision c (resp. u) feasible in s, $T_c(s, c)$ (resp. $T_u(s, u)$) is defined; it may be undefined for infeasible decisions. These three assumptions are actually undemanding: if the first assumption is violated, then it is obvious that the goal cannot be reached; if the second assumption is violated, it suffices to add a dummy value for every variable in C (resp. U) to guarantee that at least doing nothing is always feasible; if the third assumption does not hold, then feasibility relation F_c (resp. F_u) can be strengthened by considering as infeasible decisions that induce no successor state. Relations I, E, F_c, F_u, T_c, and T_u are expressed by sets of constraints. All previous elements are gathered in the notion of Markovian Game CSP.

Definition 1. *A Markovian Game Constraint Satisfaction Problem (MGCSP) is a tuple* $M = (S, I, E, C, U, F_c, T_c, F_u, T_u)$ *with:*

- S *a finite set of finite domain variables called* state variables*;*
- I *a finite set of constraints over* S *called* initialization constraints*;*
- E *a finite set of constraints over* S *called* termination constraints*;*
- C *a finite set of finite domain variables called* controllable variables
- U *a finite set of finite domain variables called* uncontrollable variables*;*

- F_c *and* F_u *finite sets of constraints over* $S \cup C$ *and* $S \cup U$ *respectively, called* feasibility constraints*;*
- T_c *and* T_u *finite sets of constraints over* $S \cup C \cup S'$ *and* $S \cup U \cup S'$ *respectively, called* transition constraints*;*
- $\exists s \in \mathbf{d}(S),\ I(s)$*;*
- $\forall s \in \mathbf{d}(S),\ \neg E(s) \rightarrow ((\exists c \in \mathbf{d}(C),\ F_c(s,c)) \wedge (\exists u \in \mathbf{d}(U),\ F_u(s,u)))$*;*
- $\forall s \in \mathbf{d}(S),\ \neg E(s) \rightarrow ((\forall c \in \mathbf{d}(C),\ F_c(s,c) \rightarrow (\exists! s' \in \mathbf{d}(S),\ T_c(s,c,s'))) \wedge (\forall u \in \mathbf{d}(U),\ F_u(s,u) \rightarrow (\exists! s' \in \mathbf{d}(S),\ T_u(s,u,s'))))$ *("$\exists!$" stands for "there exists a unique").*

To illustrate the framework, consider the NimFibo game again. In this example, the set of state variables S contains variables p, r, and l defined previously, representing respectively the next player (value in $\{A, B\}$), the number of remaining matches (value in $[0..N]$), and the number of matches taken at the last play (value in $[1..N]$). Set C (resp. U) contains a unique variable a (resp. b) of domain $[1..N-1]$ representing the number of matches taken by player A (resp. B). The different constraint sets are given below. I expresses that initially, there are N matches and variable l is assigned a default arbitrary value (since there is no last play). E expresses that the game ends when there is no match left. F_c and F_u express that at each step, a player can take at most twice the number of matches taken by the last player. T_c and T_u define the transition function of the system: e.g., the number of remaining matches r' is decreased by the number of matches taken (given a variable x, x' denotes the value of x at the next step).

$$I : (p = A) \wedge (r = N) \wedge (l = N) \qquad E : (r = 0)$$
$$F_c : (a \leq r) \wedge (a \leq 2 \cdot l) \qquad F_u : (b \leq r) \wedge (b \leq 2 \cdot l)$$
$$T_c : (p' = B) \wedge (r' = r - a) \wedge (l' = a) \qquad T_u : (p' = A) \wedge (r' = r - b) \wedge (l' = b)$$

3.2 Reachability Control Problems

A MGCSP describes the dynamics of the system considered and induces a set of possible trajectories.

Definition 2. *Let* $M = (S, I, E, C, U, F_c, T_c, F_u, T_u)$ *be a MGCSP. The set of trajectories induced by* M *is the set of (possibly infinite) sequences of state transitions* $seq : s_1 \xrightarrow{c_1} s_2 \xrightarrow{u_2} s_3 \xrightarrow{c_3} s_4 \xrightarrow{u_4} s_5 \cdots$ *such that:*

- $I(s_1)$ *holds, and for every state* s_i *which is not the last state of the sequence,* $E(s_i)$ *does not hold (*s_i *is not terminal),*
- *for every transition* $s_i \xrightarrow{c_i} s_{i+1}$ *in seq,* $F_c(s_i, c_i)$ *and* $T_c(s_i, c_i, s_{i+1})$ *hold,*
- *for every transition* $s_i \xrightarrow{u_i} s_{i+1}$ *in seq,* $F_u(s_i, u_i)$ *and* $T_u(s_i, u_i, s_{i+1})$ *hold.*

In order to control the system and restrict its possible evolutions, we use so-called *decision policies* π, which are mappings from states $s \in \mathbf{d}(S)$ to decisions $c \in \mathbf{d}(C)$. $\pi(s) = c$ means that decision c is made when state s is encountered. Such a decision policy can be partial in the sense that $\pi(s)$ may be undefined for some states $s \in \mathbf{d}(S)$. Partial policies are useful to define the controller behavior

only over the set of reachable states of the system. We are also interested in applicable policies, which have the particularity to specify only feasible decisions. These elements are formalized below.

Definition 3. *A policy for a MGCSP* $M = (S, I, E, C, U, F_c, T_c, F_u, T_u)$ *is a partial function* $\pi : \mathbf{d}(S) \to \mathbf{d}(C)$. *The domain of a policy* π *is defined as* $\mathbf{d}(\pi) = \{s \in \mathbf{d}(S) \,|\, \pi(s) \text{ defined}\}$.

The set of trajectories induced by π *is the set of trajectories for* M *of the form* $s_1 \overset{c_1}{\to} s_2 \overset{u_2}{\to} s_3 \overset{c_3}{\to} s_4 \overset{u_4}{\to} s_5 \cdots \overset{u_{i-1}}{\to} s_i$ *obtained by following* π, *i.e. such that* $c_j = \pi(s_j)$ *for every transition* $s_j \overset{c_j}{\to} s_{j+1}$ *in the sequence. The trajectory is said to be complete if it is infinite or if* $s_i \notin \mathbf{d}(\pi)$.

π *is said to be applicable iff for every trajectory* $s_1 \overset{c_1}{\to} s_2 \overset{u_2}{\to} s_3 \overset{c_3}{\to} s_4 \overset{u_4}{\to} s_5 \cdots \overset{u_{i-1}}{\to} s_i$ *induced by* π, *either* $s_i \notin \mathbf{d}(\pi)$, *or* $s_i \in \mathbf{d}(\pi)$ *and* $F_c(s_i, \pi(s_i))$ *(i.e. the policy specifies decisions which are feasible).*

Several requirements can be imposed on system-state trajectories. We focus here on *reachability* requirements, imposing to find an applicable policy π so that all trajectories induced by π satisfy a given condition at some step.

Definition 4. *A* reachability control problem *is a pair* (M, G) *with* M *a MGCSP over a set of state variables* S, *and* G *a finite set of constraints over* S *called* goal constraints. *A solution to this problem is an applicable policy* π *for* M *such that all complete trajectories* $s_1 \overset{c_1}{\to} s_2 \overset{u_2}{\to} s_3 \overset{c_3}{\to} s_4 \overset{u_4}{\to} s_5 \cdots$ *induced by* π *are finite and end in a state* s_n *such that* $G(s_n)$ *holds.*

The NimFibo problem corresponds to reachability control problem (M, G) with M the MGCSP defined in Section 3.1 and $G : (p = B) \wedge (r = 0)$ (requirement of reaching a state in which there is no match left and B must play). A possible solution policy π for $N = 15$ is given in Figure 1(b).

3.3 Relationship with QCSP/QCSP+

To relate QCSP and reachability control problems (M, G), let us consider the following QCSP+, which could be put in prenex normal form:

$$
\begin{aligned}
&Q_N(M,G) : \forall S_1[I(S_1)] \\
&\quad G(S_1) \vee (\neg E(S_1) \wedge \exists C_1, S_2[F_c(S_1, C_1) \wedge T_c(S_1, C_1, S_2)] \\
&\quad G(S_2) \vee (\neg E(S_2) \wedge \forall U_2, S_3[F_u(S_2, U_2) \wedge T_u(S_2, U_2, S_3)] \\
&\quad G(S_3) \vee (\neg E(S_3) \wedge \exists C_3, S_4[F_c(S_3, C_3) \wedge T_c(S_3, C_3, S_4)] \\
&\quad G(S_4) \vee (\neg E(S_4) \wedge \forall U_4, S_5[F_u(S_4, U_4) \wedge T_u(S_4, U_4, S_5)] \\
&\quad ... \\
&\quad G(S_{N\text{-}1}) \vee (\neg E(S_{N\text{-}1}) \wedge \exists C_{N\text{-}1}, S_N[F_c(S_{N\text{-}1}, C_{N\text{-}1}) \wedge T_c(S_{N\text{-}1}, C_{N\text{-}1}, S_N)] \\
&\quad G(S_N))...))))
\end{aligned}
$$

$Q_N(M, G)$ can be read as: "Does it hold that for every possible initial state s_1, either $G(s_1)$ is satisfied, or s_1 is not terminal and there exists a feasible

decision c_1 inducing successor state s_2 such that either the goal is reached in s_2, or s_2 is not a terminal state and for every feasible decision u_2, inducing successor state s_3, either the goal is reached in s_3 or s_3 is not terminal and there exists a feasible decision c_3 ... such that either $G(s_{N-1})$ holds or s_{N-1} is not terminal and there exists a feasible decision c_{N-1} inducing a state s_N satisfying the goal?".

Proposition 1. *There exists a winning strategy for QCSP $Q_N(M, G)$ given in Equation 1 if and only if there exists a solution policy $\pi : \mathbf{d}(S) \rightarrow \mathbf{d}(C)$ for reachability control problem (M, G) such that all complete trajectories induced by π have less than N steps.*

Proposition 1 implies that if a QCSP can be put in a form similar to $Q_N(M, G)$, in which the notion of state is made explicit, then searching for a solution policy for (M, G) suffices to solve the initial QCSP. However, there may exist a solution policy for (M, G) and no winning strategy for $Q_N(M, G)$ because $Q_N(M, G)$ models a control problem over a *bounded* horizon. It is possible to take N high enough, for instance equal to the number of possible states ($N = |\mathbf{d}(S)|$). But as $|\mathbf{d}(S)|$ can be huge and as the number of variables and constraints in $Q_N(M, G)$ is linear in N, this approach may not be practically applicable. The problem does not arise with the MGCSP approach in which we just describe the transition function of the system instead of unfolding the model. In another direction, Proposition 1 can be seen as a counterpart of a property of Markov Decision Processes (MDPs [11]) stating that every MDP has an optimal policy which is stationary.

Next, in terms of space needed to record a winning strategy, the size of policies can be exponentially smaller than the size of policy trees, which can be useful when embedding a controller on-board an autonomous system having limited memory. More precisely, if R_π denotes the set of states reachable using π, policy π can be recorded as a table contains $|R_\pi|$ (s, c) pairs. On the other hand, let W be an equivalent winning strategy for $Q_N(M, G)$, that is a winning strategy inducing the same trajectories as π. Strategy W expressed as a policy tree may contain $|\mathbf{d}(U)|^{N/2}$ leaves, which is exponential in N, and which can be shown to be always greater than or equal to $|R_\pi|$.

A last remark concerns the semantics of the notion of goal. In QCSP, goal constraints are specified just after the rightmost quantifier in the quantification sequence. But there may exist winning strategies which never reach the goal: these strategies instead block the adversary at some step. In MGCSPs, the goal is guaranteed to be reached along every trajectory. Both notions of goal are however equivalent here due to the assumption of existence of a feasible decision in every non-terminal state, which ensures that no blocking can occur in $Q_N(M, G)$.

4 Algorithm

The algorithm proposed for solving reachability control problems over MGCSP is inspired by techniques for planning in non-deterministic domains [12]. One difference is the use of constraint programming to reason over the different relations.

General description The algorithm is composed of three functions:

- **reachMGCSP**, responsible for exploring the different initial states,
- **exploreC**, responsible for exploring the different feasible controllable decisions $c \in \mathbf{d}(C)$ in a current s,
- **exploreU**, responsible to do the same for uncontrollable decisions $u \in \mathbf{d}(U)$.

Search behaves as an And/Or search in which Or nodes correspond to decisions in C and And nodes to decisions in U. The search space is explored in a depth-first manner, and only states which are reachable from initial states are considered.

During search, the algorithm maintains a current policy π. It also associates, with each state s, a mark $Mark(s)$ in $\{SOLVED, BAD, PROCESSING, NONE\}$. Mark *SOLVED* means that state s has already been visited during search and there already exists in current policy π a recipe to reach the goal starting from s. Mark *BAD* means that there does not exist any solution policy starting from s. Mark *PROCESSING* is associated with states on the trajectory currently explored. Mark *NONE*, which is not explicitly stored, is associated with all other states. In the implementation, state marks are recorded in a hash table, which is empty initially (all marks set to *NONE*).

A specificity of the algorithm concerns the handling of loops. A loop is a situation in which a state marked *PROCESSING* is encountered again. When a loop is detected and the goal has not been reached yet, this means that the adversary has a way to generate an infinite loopy trajectory in which the goal is never reached. For example, assume that Figure 2 represents the set of feasible trajectories of a system. Trajectories $seq_1 : s_a \overset{c:0}{\to} s_b \overset{u:0}{\to} s_c \overset{c:0}{\to} s_d \overset{u:1}{\to} s_e \overset{c:0}{\to} s_b$ and $seq_2 : s_a \overset{c:0}{\to} s_b \overset{u:0}{\to} s_c \overset{c:0}{\to} s_d \overset{u:1}{\to} s_e \overset{c:1}{\to} s_d$ respectively loop over s_b and s_d. Therefore, trajectory $seq_3 : s_a \overset{c:0}{\to} s_b \overset{u:0}{\to} s_c \overset{c:0}{\to} s_d \overset{u:1}{\to} s_e$ cannot be extended to a solution. Set $J = \{s_b, s_d\}$ is called the *loop justification* of seq_3. It corresponds to the set of past states over which loops are detected when trying to reach the goal from seq_3. The mark of s_e, the last state of seq_3, cannot however be set to *BAD* because loops discovered depend on decisions made before s_e. For $seq_4 : s_a \overset{c:0}{\to} s_b \overset{u:0}{\to} s_c \overset{c:0}{\to} s_d$ and $seq_5 : s_a \overset{c:0}{\to} s_b \overset{u:0}{\to} s_c$, the loop justification is $\{s_b\}$. For $seq_6 : s_a \overset{c:0}{\to} s_b$, the loop justification is empty ($J = \varnothing$). This means that in seq_6, no state explored strictly before s_b is involved in the loops discovered after s_b. The mark of s_b can then be set to *BAD*. More generally, a state whose exploration does not succeed can be marked as *BAD* if the current loop justification is empty.

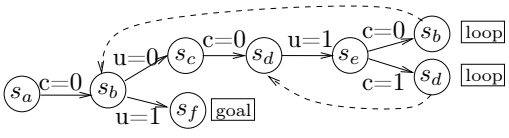


Fig. 2. Behavior of algorithm **reachMGCSP** in face of loops

Pseudo-code Main function **reachMGCSP** takes as input a MGCSP M and a set of goal constraints G. It returns $(true, \pi)$ if M admits a solution policy π, $(false, \varnothing)$ otherwise. To do that, function **reachMGCSP** starts with an empty policy and analyzes every possible initial state s, i.e. every state s satisfying the initialization constraints in I (function *getSols* used in line 6 returns the set of solutions of a CSP). Every initial state s which does not satisfy the goal and whose mark differs from *SOLVED* is then studied. If s is terminal or has mark *BAD*, then the control problem has no solution and $(false, \varnothing)$ is returned (line 8). Otherwise, s is explored further using a call to function **exploreC** (line 10).

```
1  Input: a MGCSP M and a set of goal constraints G
2  Output: a pair (b, π) with b a boolean and π a policy
3  reachMGCSP(M, G)
4  begin
5      π ← ∅
6      foreach s ∈ getSols(I(S)) do
7          if ¬G(s) ∧ (Mark(s) ≠ SOLVED) then
8              if E(s) ∨ (Mark(s) = BAD) then return (false, ∅)
9              else
10                 (covered, π, .) ← exploreC(s, π)
11                 if ¬covered then return (false, ∅)
12     return (true, π)
```

Function **exploreC**(s, π) explores the possible decisions that can be made is state s. It returns a triple (b, π', J). b specifies whether policy π given in input can be extended so that the goal can always be reached starting from s. If $b = true$, π' is the extended policy covering s. If $b = false$, J is a set of states justifying the absence of solution starting from s. J corresponds to the loop justification described previously. The first part of **exploreC** (lines 5 to 10) determines all decisions c that are feasible in state s and all associated possible successor states s', by reasoning over CSP $F_c \wedge T_c \wedge (S = s)$. If some successor state s' obtained by applying c satisfies the goal or has mark *SOLVED*, then it suffices to set $\pi(s) = c$ to cover state s. The second part of function **exploreC** (line 11 to 27) traverses the set of successor states s' to be explored further. If s' has mark *SOLVED*, then a solution is found to cover state s (line 17). Otherwise, if s' has mark *PROCESSING*, the loop justification is extended (line 19). Otherwise, if s' has mark *NONE*, it is further explored via a call to function **exploreU** (line 22). If this call returns an extended solution policy, then the mark of s is set to *SOLVED* and the new solution policy is returned (line 23). Otherwise, the loop justification is extended (line 24). If all successor states have been explored without finding a solution, then current state s is removed from the loop justification, its mark is set to *BAD* if the loop justification is empty, and an inconsistency result is returned (lines 25 to 27).

```
1  Input: a state s and a current policy π
2  Output: a triple (b, π', J) with b a boolean, π' a policy, and J a set of states
3  exploreC(s, π)
4  begin
5      toExplore ← ∅
6      foreach sol ∈ getSols(F_c(S, C) ∧ T_c(S, C, S') ∧ (S = s)) do
7          (s, c, s') ← (sol^↓S, sol^↓C, sol^↓S')
8          if G(s') ∨ (Mark(s') = SOLVED) then
9              setMark(s, SOLVED); return (true, π ∪ {(s, c)}, ∅)
10         else if ¬E(s') then toExplore ← toExplore ∪ {(c, s')}
11     π' ← π
12     setMark(s, PROCESSING)
13     J ← ∅
14     while toExplore ≠ ∅ do
15         Choose (c, s') ∈ toExplore; toExplore ← toExplore \ {(c, s')}
16         if Mark(s') = SOLVED then
17             setMark(s, SOLVED); return (true, π' ∪ {(s, c)}, ∅)
18         else if Mark(s') = PROCESSING then
19             J ← J ∪ {s'}
20         else if Mark(s') = NONE then
21             π' ← π' ∪ {(s, c)}
22             (covered, π', J') ← exploreU(s', π')
23             if covered then setMark(s, SOLVED); return (true, π', ∅)
24             else J ← J ∪ J'; π' ← π' \ {(s, c)}
25     J ← J \ {s}
26     if J = ∅ then setMark(s, BAD) else setMark(s, NONE)
27     return (false, π', J)
```

Function **exploreU** behaves similarly. The only differences are as follows. In the initial phase (lines 5 to 11), search can be pruned if there exists a successor state s' whose mark equals BAD, or such that s' is a terminal non-goal state. An inconsistency result is also directly returned if a loop is detected. For the second phase, in which the rest of the successor states are more finely studied, if the mark of one successor state s' equals BAD, then an inconsistency result is returned (line 17). Otherwise, a call to **exploreC** is invoked, in order to develop the different possible decisions that can be made in s' (line 19).

Proposition 2. *Algorithm* **reachMGCSP** *is sound and complete: it returns* $(true, \pi)$ *with* π *a solution policy if reachability control problem* (M, G) *admits a solution, and* $(false, \varnothing)$ *otherwise.*

Algorithmic improvements The basic algorithm can be improved on several points. First, in **exploreC**, CSP $F_c(S, C) \wedge T_c(S, C, S') \wedge G(S')$ can be considered in order to faster determine whether there exists a controllable decision c allowing the goal to be directly satisfied, instead of enumerating all solutions

```
1  Input: a state s and a current policy π
2  Output: a triple (b, π', J) with b a boolean, π' a policy, and J a set of states
3  exploreU(s, π)
4  begin
5      toExplore ← ∅
6      foreach sol ∈ getSols(F_u(S,U) ∧ T_u(S,U,S') ∧ (S = s)) do
7          s' ← sol^{↓S'}
8          if (E(s') ∧ ¬G(s')) ∨ (Mark(s') = BAD) ∨ (s = s') then
9              setMark(s, BAD); return (false, π, ∅)
10         else if Mark(s') = PROCESSING then return (false, π, {s'})
11         else if Mark(s') = NONE then toExplore ← toExplore ∪ {s'}
12     π' ← π
13     setMark(s, PROCESSING)
14     while toExplore ≠ ∅ do
15         Choose s' ∈ toExplore; toExplore ← toExplore \ {s'}
16         if Mark(s') = BAD then
17             setMark(s, BAD); return (false, π', ∅)
18         else if Mark(s') = NONE then
19             (covered, π', J) ← exploreC(s', π')
20             if ¬covered then
21                 J ← J \ {s}
22                 if J = ∅ then setMark(s, BAD) else setMark(s, NONE)
23                 return (false, π', J)
24     setMark(s, SOLVED); return (true, π', ∅)
```

of $F_c \wedge T_c$ and then checking whether one of them satisfies G. Similarly, it is possible to consider, in **exploreU**, CSP $F_u(S,U) \wedge T_u(S,U,S') \wedge E(S') \wedge \neg G(S')$ to search for adversary strategies which directly lead to a non-goal terminal state.

In terms of space consumption, the algorithm records marks over reachable states only. But the set of such states may be large and recording marks may become expensive. To overcome this difficulty, some state marks could be forgotten during search: the algorithm then remains valid but may re-explore some parts of the search space. This option has not been used in the experiments.

Last, we consider in the MGCSP framework that the $\exists$-player begins to play. To handle situations in which the $\forall$-player begins, it suffices to replace the call to **exploreC** in function **reachMGCSP** by a call to **exploreU**.

5 Experiments

We ran our experiments on an Intel i5-520, 1.2GHz, 4GB RAM. Maximum computation time is set to one hour. Algorithm **reachMGCSP** is implemented in *Dyncode*, a tool developed on top of constraint programming library Gecode.[1]

[1] `http://www.gecode.org/`

Any constraint available in Gecode can be used in Dyncode. Dyncode was initially introduced in [13] to handle control problems involving non-determinism and partial observability. Algorithms in [13] are less efficient than dedicated algorithm **reachMGCSP** over deterministic and completely observable problems. For the experiments, we used min-domain for the variable choice heuristics and lexicographic ordering for the value choice.

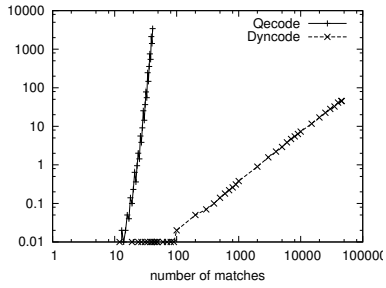
We first compared Dyncode with Qecode[2], a QCSP+ solver based on Gecode. We performed experiments on three games already encoded in the Qecode distribution: NimFibo, Connect4, and MatrixGame. Figures 3(a) to 3(c) show that on these problems, Dyncode outperforms Qecode by several orders of magnitude.

For NimFibo (Figure 3(a)), Qecode can solve problems up to 40 matches, whereas Dyncode can solve instances involving several tens of thousands of matches. The time complexity observed with Dyncode is even linear in the number of matches, whereas there is an exponential blowup with Qecode. In other words, explicitly using the notion of state and recording good/bad states encountered during search breaks the problem complexity.

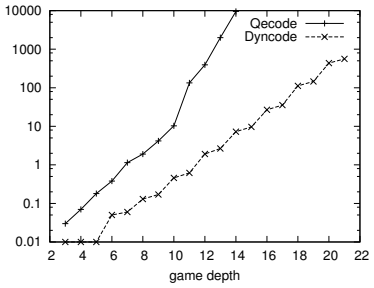
Connect4 is a two-player game over a six-row $\times$ seven-column board. At each step, a player puts a token in one column of the board. This token falls at the bottom of the column by the effect of gravity. A player wins if he manages to align four of his tokens horizontally, vertically, or diagonally. The game is null if there is no alignment and the board is full. As in the Qecode distribution, we consider here that the goal is to play without losing the game over a fixed number of steps. This variant is referred to as Connect4_Bounded. Figure 3(b) shows that Dyncode solves more instances than Qecode for this game. This time, there is an exponential blowup for both solvers, but the slope of the blowup is weaker for Dyncode. The first reason is again the explicit use of the notion of state, since in Connect4, several sequences of plays can lead to the same configuration. The second point is that Qecode initially creates many variables and constraints to encode the problem over a fixed horizon. In theory, it then performs so-called *cascade propagation* over the whole problem. On the opposite, Dyncode propagates constraints just over the current state and the next state. This may achieve less pruning, but this is performed much faster. We also realized experiments with random value choice heuristics and restarts, to see the effect of search diversification. We observed that randomization in Dyncode speeds search on some executions (e.g. Connect4_Bounded for game depth equal to 25 may be solved in about 30 minutes), but that it can lead to memory problems on other executions, due to the recording of state marks.

In MatrixGame, a 0/1 matrix of size 2^d is considered. At each turn, the $\exists$-player cuts the matrix horizontally and decides to keep the top or bottom part. The $\forall$-player then cuts the matrix vertically and keeps the left or right part. After d turns, the matrix is reduced to a single cell. The $\exists$-player wins if this cell contains a 1. Figure 3(c) shows that Dyncode performs better for this game, whose MGCSP encoding is such that the same state is never encountered twice. We believe that Dyncode is faster because it reasons over smaller CSPs.

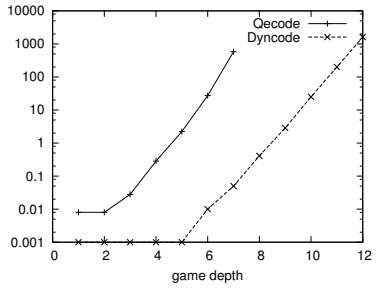
[2] `http://www.univ-orleans.fr/lifo/software/qecode`



(a) NimFibo



(b) Connect4_Bounded



(c) MatrixGame

Board size		CPU time (sec.)	
ncols	nrows	Queso	Dyncode
		3.06GHz,1GB RAM	1.2GHz,4GB RAM
4	4	1.05	0.44
4	5	9.13	1.47
5	4	63.57	6.44
5	5	1749.94	64.9
5	6	16012.50	1621.8

(d) Connect4_Full

Fig. 3. Comparison of computation times obtained with Dyncode, Qecode (QCSP+ solver), and Queso (QCSP solver) on games (a) NimFibo, (b) Connect4_Bounded, (c) MatrixGame, (d) Connect4_Full; y-axis represents CPU time in seconds

Dyncode was also compared with Queso, a QCSP solver which was shown in [4] to outperform other QCSP solvers such as BlockSolve or QCSP-solve. We did not rerun Queso, which is not maintained anymore, but instead directly took the results provided in [4], obtained with a Pentium 4, 3.06GHz, 1GB RAM. Results are given in Figure 3(d) for Connect4, but this time with boards of size $N \times M$, and in which the goal for the $\exists$-player is to win the game. The results show that Dyncode outperforms Queso. Again, we believe that the notion of state is really useful here to avoid re-exploring several times the same part of the search space. Whereas Queso uses techniques called pure value rules and constraint propagation for reified disjunction constraints, the limited constraint propagation performed by Dyncode reduces computation times.

Dyncode could be compared against other tools: (a) tools for non-deterministic planning, e.g. MBP [12]; (b) tools for controller synthesis from the model checking community [14]; (c) tools for solving MDPs [11], by considering MGCSPs as MDPs with 0/1 probabilities. MDP algorithms exploring the whole state space, such as basic value/policy iteration, may not scale well for games. And/Or search algorithms exploring only reachable states may be more competitive, but their management of probabilities and Bellman backups may induce extra computa-

tion times. These comparisons are left for future work. An important aspect is the fact that these tools do not offer the flexibility of constraint-based models.

6 Conclusion

This paper showed that, at the moment, using QCSP/QCSP+ is not the best constraint-based way of solving control problems for dynamic systems satisfying the Markovian and complete observability assumptions. The strengths of quantified constraints are more appropriate for solving problems in which these assumptions are violated, or problems involving just a few alternations of quantifiers. In the future, we plan to extend MGCSPs to model control problems in which the number of uncontrollable transitions between two controllable steps is not fixed. Requirements more general than reachability could also be considered. This should be a step to cross-fertilize advances in constraint programming and work performed in the automata and model checking community.

References

1. Bordeaux, L., Monfroy, E.: Beyond NP: Arc-consistency for Quantified Constraints. In: Proc. of CP-02. (2002) 371–386
2. Mamoulis, N., Stergiou, K.: Algorithms for Quantified Constraint Satisfaction Problems. In: Proc. of CP-04. (2004) 752–756
3. Gent, I.P., Nightingale, P., Rowley, A.: Encoding Quantified CSPs as Quantified Boolean Formulae. In: Proc. of ECAI-04. (2004) 176–180
4. Nightingale, P.: Non-binary Quantified CSP: Algorithms and Modelling. Constraints **14**(4) (2009) 539–581
5. Gent, I.P., Nightingale, P., Stergiou, K.: QCSP-Solve: A Solver for Quantified Constraint Satisfaction Problems. In: Proc. of IJCAI-05. (2005) 138–143
6. Stergiou, K.: Repair-based Methods for Quantified CSPs. In: Proc. of CP-05. (2005) 652–666
7. Verger, G., Bessiere, C.: Blocksolve: a Bottom-up Approach for Solving Quantified CSPs. In: Proc. of CP-06. (2006) 635–649
8. Benedetti, M., Lallouet, A., Vautard, J.: QCSP Made Practical by Virtue of Restricted Quantification. In: Proc. of IJCAI-07. (2007) 38–43
9. Larson, R., Casti, J.: Principles of Dynamic Programming. M. Dekker Inc. (1978)
10. Bessiere, C., Verger, G.: Strategic Constraint Satisfaction Problems. In: Proc. of CP-06 Workshop on Modelling and Reformulation. (2006) 17–29
11. Puterman, M.: Markov Decision Processes, Discrete Stochastic Dynamic Programming. John Wiley & Sons (1994)
12. Bertoli, P., Cimatti, A., Roveri, M., Traverso, P.: Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In: Proc. of IJCAI-01. (2001) 473–478
13. Pralet, C., Verfaillie, G., Lemaître, M., Infantes, G.: Constraint-Based Controller Synthesis in Non-Deterministic and Partially Observable Domains. In: Proc. of ECAI-10. (2010) 681–686
14. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Proc. of the 7th International Conference on Verification, Model Checking and Abstract Interpretation. (2006) 364–380