Au delà des QCSP pour résoudre des problèmes de contrôle

Cédric Pralet, Gérard Verfaillie

Au delà des QCSP pour résoudre des problèmes de contrôle

Cédric Pralet **Gérard Verfaillie**

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

Résumé

Les problèmes de satisfaction de contraintes quantifiés (QCSP) sont souvent présentés comme les outils adéquats permettant de modéliser et de résoudre des problèmes de jeu à deux joueurs ou de planification dans l'incertain ou plus généralement des problèmes où l'objectif est de contrôler un système dynamique soumis à des événements incontrôlés. Ce papier montre que, pour de nombreux problèmes de ce type, l'approche standard de type QCSP ou QCSP+ n'est pas la plus appropriée. Les raisons principales en sont que, dans le cadre QCSP/QCSP+, (1) les évolutions possibles du système sont dépliées sur un nombre fixé d'étapes, (2) la notion d'état du système n'est pas explicitement prise en compte et (3) les algorithmes recherchent des stratégies gagnantes à mémoire complète définies comme des arbres de politique plutôt que des stratégies sans mémoire définies comme des fonctions depuis les états vers les décisions. Ce papier propose un nouveau cadre à base de contraintes qui n'a pas ces défauts. Les expérimentations montrent des améliorations de plusieurs ordres de grandeur par rapport à des solveurs de QCSP/QCSP+.

1 Introduction

Les problèmes de satisfaction de contraintes quantifiés (QCSP [3]) ont été introduits pour modéliser et résoudre des problèmes de satisfaction de contraintes (CSP) où la valeur prise par certaines variables est incertaine ou incontrôlable. Formellement, un QCSP est défini par deux éléments : un ensemble de contraintes C et une séquence de quantification $Q = Q_1 x_1 \dots Q_n x_n$ où chaque Q_i est un quantificateur existentiel ou universel ($\exists$ ou $\forall$). Un QCSP défini par $C = \{x_1 + x_3 < x_4, x_2 \neq x_1 - x_3\}$ et $Q = \exists x_1 \forall x_2 \exists x_3 \forall x_4$ doit être interprété comme : "Existe-t-il une valeur de x_1 telle que, pour toute valeur de x_2, il existe une valeur de x_3 telle que, pour toute valeur de x_4, les contraintes de C sont satisfaites ?". Résoudre un QCSP consiste à répondre oui ou non à cette question et, en cas de réponse positive, à produire une stratégie gagnante. Si $\mathbf{d}(x)$ désigne le domaine d'une variable x et A_x l'ensemble des variables quantifiées universellement qui précédent x dans la séquence de quantification, une telle stratégie est généralement définie comme un ensemble de fonctions $f_x : (\prod_{y \in A_x} \mathbf{d}(y)) \rightarrow \mathbf{d}(x)$: une fonction par variable x quantifiée existentiellement. Cet ensemble de fonctions peut être représenté par ce qu'on appelle un arbre de politique. Divers algorithmes ont été proposés ces dernières années pour résoudre un QCSP, depuis les premières techniques utilisant l'arc-cohérence quantifiée binaire ou ternaire (QAC [3, 8]) ou la transformation en formule booléenne quantifiée (QBF [4]) jusqu'aux techniques utilisant la règle dite de valeur pure, l'arc-cohérence quantifiée généralisée [9], le *backjumping* guidé par les conflits [5], la réparation de solutions [13] ou le parcours de droite à gauche de la séquence de quantification [14]. Récemment, une variante de QCSP dénommée QCSP+ [1] a été proposée pour rendre la modélisation plus aisée. L'idée est d'utiliser des séquences de quantification restreinte. Par exemple, une séquence de la forme $\exists x_1 [x_1 \geq 3] \forall x_2 [x_2 \leq x_1] \exists x_3, x_4 [(x_3 \neq x_4) \wedge (x_3 \neq x_1)]$ doit être interprétée comme : "Existe-t-il une valeur de x_1 telle que $x_1 \geq 3$ et, pour toute valeur de x_2 telle que $x_2 \leq x_1$, il existe des valeurs de x_3 et x_4 telles que $x_3 \neq x_4$, $x_3 \neq x_1$ et toutes les contraintes de C sont satisfaites ?".

QCSP et QCSP+ peuvent être utilisés pour modéliser des problèmes impliquant quelques alternances de quantificateur tels que des problèmes d'ordonnancement dans l'incertain [1]. Ils peuvent aussi être utilisés pour modéliser des problèmes impliquant un grand nombre d'alternances de quantificateur tels que des

problèmes de jeu à deux joueurs ou de planification dans l'incertain. Dans les problèmes de jeu, le but est de déterminer un premier coup pour le joueur 1 tel que, quel que soit le coup du joueur 2, il existe un second coup pour le joueur 1 tel que, quel que soit le coup du joueur 2 ... le joueur 1 gagne. La taille de la séquence de quantification dépend du nombre maximum de coups à considérer. La planification dans l'incertain et plus généralement le problème de contrôle de l'état d'un système dynamique soumis à des événements peuvent être vus comme un jeu contre la nature.

Le but de ce papier est de montrer que, quand l'état du système est complètement observable et quand l'évolution de l'état est markovienne, c'est-à-dire quand l'état courant ne dépend que de l'état et de l'événement précédents et non de tout l'historique des événements, une approche de type QCSP/QCSP+ n'est pas la plus pertinente. Le papier est organisé comme suit. Nous commençons par illustrer pourquoi une approche de type QCSP/QCSP+ n'est pas toujours appropriée (section 2). Nous introduisons ensuite un cadre dénommé MGCSP for *Markovian Game CSP* (section 3) et des algorithmes associés (section 4). Les résultats expérimentaux sont rapportés en section 5. Les preuves sont omises pour des raisons de place.

2 Exemple illustratif

Considérons le jeu *NimFibo*, un *benchmark* QCSP classique. Ce jeu implique deux joueurs A et B qui jouent alternativement. Initialement, N allumettes sont sur la table. Lors du premier coup, le joueur A peut prendre entre 1 et $N-1$ allumettes. Puis, lors de chaque coup, le joueur courant prend au moins une allumette et au plus deux fois le nombre d'allumettes prises par l'autre joueur lors du coup précédent. Le joueur qui prend la dernière allumette gagne. Le problème est de trouver une stratégie gagnante pour le joueur A.

2.1 Approche QCSP/QCSP+

Supposons que N est impair. Pour modéliser ce jeu comme un QCSP+, on peut introduire N variables $r_1, \ldots, r_N$ de domaine $[0..N]$ qui représentent le nombre d'allumettes restantes après chaque coup (N variables parce qu'il y a au plus N coups) et N variables de domaine $[1..N-1]$ qui représentent le nombre d'allumettes prises à chaque coup : a_1, a_3, ..., a_N pour le joueur A et b_2, b_4, ..., b_{N-1} pour le joueur B. Un modèle QCSP+ est alors le suivant :

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

La figure 1(a) représente une stratégie gagnante exprimée sous la forme d'un arbre de politique pour $N = 15$. Les nœuds circulaires représentent toutes les décisions possibles du joueur B tandis que les nœuds carrés représentent les décisions que doit prendre le joueur A pour gagner.

2.2 Approche à base d'état

L'état du système à chaque étape peut être représenté par trois variables d'état : une variable $j \in \{A, B\}$ représentant le joueur courant, une variable $r \in [0..N]$ représentant le nombre d'allumettes restantes et une variable $p \in [1..N]$ représentant le nombre d'allumettes prises par le joueur précédent. La décision prise lors de chaque tour peut être représentée par deux variables de décision a et b de domaine $[1..N-1]$ représentant le nombre d'allumettes prises respectivement par les joueurs A et B. La décision a est prise avant la décision b. La décision a est sous le contrôle du joueur A alors que la décision b ne l'est pas. Le but est d'atteindre, quelles que soient les décisions du joueur B, un état où $j = B$ (B doit jouer) et $r = 0$ (il ne reste plus d'allumettes). Une solution de ce problème de contrôle peut prendre la forme d'une politique $\pi : \{(A, r, p) \,|\, r \in [1..N], p \in [1..N]\} \rightarrow \mathbf{d}(a)$ qui associe une valeur de a à chaque état dans lequel A doit jouer et il reste au moins une allumette. Il n'est pas nécessaire de définir la politique π pour tous les états, mais uniquement pour ceux qui sont atteignables à partir de l'état initial en suivant π. La figure 1(b) montre une politique solution, tandis que la figure 1(c) représente les états atteignables en suivant cette politique, ainsi que les transitions entre états. Dans la figure 1(b), la deuxième ligne de la première colonne indique par exemple que, dans l'état $(A, 12, 1)$, le joueur A doit prendre une allumette ($a = 1$).

2.3 Comparaison entre les deux approches

Premièrement, l'approche QCSP oblige à borner le nombre d'étapes à considérer et a déplier les évolutions possibles du système sur le nombre maximum d'étapes (ici N). Le nombre de variables à considérer est proportionnel au nombre d'étapes. A l'opposé, l'approche

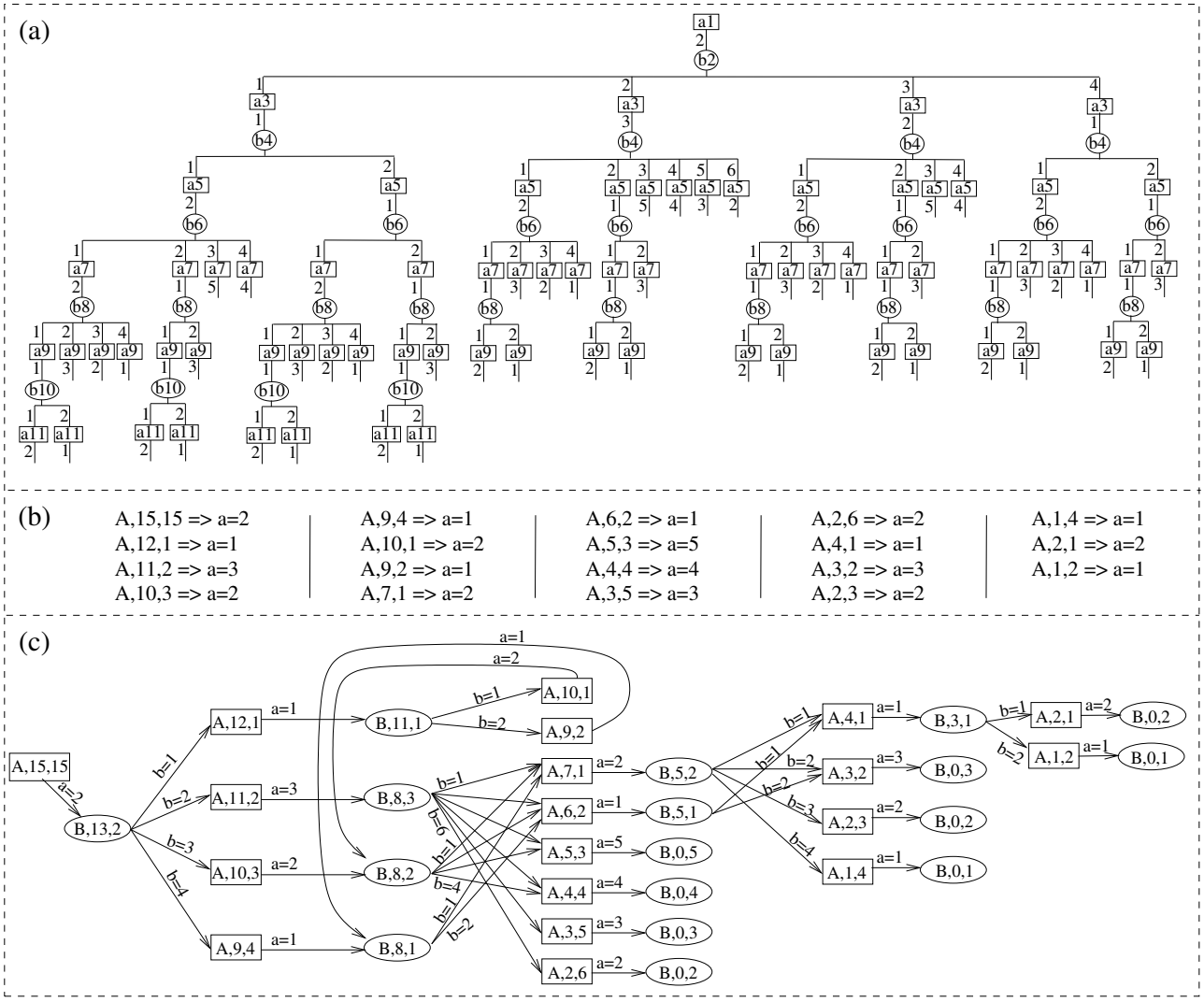


FIGURE 1 – Comparaison entre QCSP et modèle à base d'état sur le jeu *NimFibo* : (a) arbre QCSP de politique ; (b) politique à base d'état ; (c) graphe d'atteignabilité suivant cette politique.

à base d'état peut raisonner sur des horizons non bornés. Elle ne requiert pas que l'évolution du système soit dépliée, ce qui conduit à des modèles plus compacts.

Deuxièmement, on peut observer qu'avec $N = 15$, l'arbre de politique contient 48 feuilles et que la politique à base d'état contient seulement 19 paires (état,décision) alors que les deux induisent les mêmes séquences de décision et d'état. Le rapport en taille croît exponentiellement avec le nombre d'allumettes. La raison principale en est que, dans un contexte d'observabilité complète de l'état et de dynamique markovienne du système, l'arbre de politique mémorise trop d'information. Par exemple, dans le graphe d'atteignabilité de la figure 1(c), les deux séquences de décision $seq_1 : [a = 2, b = 1, a = 1, b = 1, a = 2]$ et $seq_2 : [a = 2, b = 3, a = 2]$ sont équivalentes car elles se terminent sur le même état $(B, 8, 2)$. La seule information utile pour prendre une décision dans cet état est l'état lui-même et non la trajectoire qui a permis de l'atteindre. En d'autres termes, rechercher des stratégies à mémoire complète sous forme d'arbre de politique comme cela est fait dans le cadre QCSP/QCSP+ revient à chercher dans un espace inutilement grand puisqu'il suffit de chercher des stratégies sans mémoire sous forme de politique.

Troisièmement, raisonner explicitement sur l'état permet de mémoriser qu'un état a déjà été exploré et le résultat de cette exploration (succès ou échec) et ainsi d'éviter toute nouvelle exploration. Par exemple, sur la figure 1(c), il n'est pas nécessaire d'explorer deux fois

les trajectoires débutant dans l'état $(B, 8, 2)$ (une fois quand cet état est atteint à partir de l'état $(A, 10, 1)$ et une autre fois quand il l'est à partir de l'état $(A, 10, 3)$). Ces notions de *good/nogood* sur les états ne sont pas gérés par l'approche QCSP qui peut uniquement gérer des *goods/nogoods* sur des ensembles de variables du modèle QCSP. Mémoriser de l'information sur les états déjà explorés utilise les principes de la programmation dynamique en avant [7].

Pour toutes ces raisons, nous pensons qu'il est nécessaire d'introduire un nouveau cadre à base de contraintes faisant explicitement appel à la notion d'état, permettant de modéliser et de résoudre efficacement des problèmes de contrôle de systèmes dynamiques complètement observables et markoviens.

3 Markovian Game CSP (MGCSP)

Dans ce qui suit, on considère que l'état du système est décrit par un ensemble S de variables. Toute affectation $s \in \mathbf{d}(S)$ est appelée un *état* (étant donné un ensemble ordonné X de variables, $\mathbf{d}(X)$ désigne le produit cartésien des domaines des variables de X). On considère de la même façon que les décisions prises à chaque tour par le $\exists$-joueur (décideur ou contrôleur) et le $\forall$-joueur (adversaire ou environnement) sont décrites par deux ensembles C et U de variables. Toute affectation $c \in \mathbf{d}(C)$ (resp. $u \in \mathbf{d}(U)$) est appelée une *décision* contrôlable (resp. incontrôlable).

3.1 Modèle du système

Pour représenter les états initiaux et finaux possibles, nous utilisons une relation d'initialisation $I \subseteq \mathbf{d}(S)$ et une relation de terminaison $E \subseteq \mathbf{d}(S)$. Pour représenter le fait que certaines décisions $c \in \mathbf{d}(C)$ (resp. $u \in \mathbf{d}(U)$) ne peuvent pas être prises dans certains états, du fait de règles de jeu ou de contraintes physiques, nous utilisons une relation de faisabilité $F_c \subseteq \mathbf{d}(S) \times \mathbf{d}(C)$ (resp. $F_u \subseteq \mathbf{d}(S) \times \mathbf{d}(U)$) telle que $F_c(s, c)$ (resp. $F_u(s, u)$) est vrai ssi la décision c (resp. u) est faisable dans l'état s. La dynamique du système est définie par une fonction de transition $T_c : \mathbf{d}(S) \times \mathbf{d}(C) \to \mathbf{d}(S)$ (resp. $T_u : \mathbf{d}(S) \times \mathbf{d}(U) \to \mathbf{d}(S)$) telle que $s' = T_c(s, c)$ (resp. $s' = T_u(s, u)$) signifie que s' est le résultat de l'application de la décision c (resp. u) dans l'état s.

Trois hypothèses sont faites pour garantir l'absence de blocage : premièrement, il existe au moins un état initial possible, c'est-à-dire un état s tel que $I(s)$ est vrai ; deuxièmement, pour tout état s non terminal ($E(s)$ faux), il existe au moins une décision faisable, c'est-à-dire une décision c (resp. u) telle que $F_c(s, c)$ (resp. $F_u(s, u)$) est vrai ; troisièmement, pour tout état s non terminal et pour toute décision c (resp. u) faisable dans s, $T_c(s, c)$ (resp. $T_u(s, u)$) est défini (il peut être non défini pour des décisions infaisables). Ces trois hypothèses ne sont en fait pas très contraignantes : si la première est violée, il est évident que le but ne peut pas être atteint ; si la seconde l'est, il suffit d'ajouter une valeur nulle à chaque variable de C (resp. U) pour garantir que ne rien faire est toujours faisable ; si la troisième l'est, la relation de faisabilité F_c (resp. F_u) peut être restreinte en considérant que les décisions qui n'induisent pas d'état suivant sont infaisables. Les relations I, E, F_c, F_u, T_c et T_u sont exprimées via des ensembles de contraintes. Tous ces éléments sont réunis dans la notion de Markovian Game CSP.

Définition 1 *Un MGCSP (Markovian Game CSP) est un n-uplet* $M = (S, I, E, C, U, F_c, T_c, F_u, T_u)$ *avec :*

- S *un ensemble fini de variables à domaine fini, appelées* variables d'état *;*
- I *un ensemble fini de contraintes sur* S*, appelées* contraintes d'initialisation *;*
- E *un ensemble fini de contraintes sur* S*, appelées* contraintes de terminaison *;*
- C *un ensemble fini de variables à domaine fini, appelées* variables contrôlables *;*
- U *un ensemble fini de variables à domaine fini, appelées* variables incontrôlables *;*
- F_c *et* F_u *des ensembles finis de contraintes respectivement sur* $S \cup C$ *et* $S \cup U$*, appelées* contraintes de faisabilité *;*
- T_c *et* T_u *des ensembles finis de contraintes respectivement sur* $S \cup C \cup S'$ *et* $S \cup U \cup S'$*, appelées* contraintes de transition *;*
- $\exists s \in \mathbf{d}(S),\ I(s)$ *;*
- $\forall s \in \mathbf{d}(S),\ \neg E(s) \to ((\exists c \in \mathbf{d}(C),\ F_c(s, c)) \wedge (\exists u \in \mathbf{d}(U),\ F_u(s, u)))$ *;*
- $\forall s \in \mathbf{d}(S),\ \neg E(s) \to ((\forall c \in \mathbf{d}(C),\ F_c(s, c) \to (\exists! s' \in \mathbf{d}(S),\ T_c(s, c, s'))) \wedge (\forall u \in \mathbf{d}(U),\ F_u(s, u) \to (\exists! s' \in \mathbf{d}(S),\ T_u(s, u, s'))))$

S' est une copie des variables de S représentant la valeur des variables d'état dans l'état suivant ; s' représente l'état suivant.

Pour illustrer cette définition, reconsidérons le jeu *NimFibo*. Dans cet exemple, l'ensemble S des variables d'état est constitué des variables j, r et p précédemment définies, représentant respectivement le joueur courant (valeur dans $\{A, B\}$), le nombre d'allumettes restantes (valeur dans $[0..N]$) et le nombre d'allumettes prises par le joueur précédent (valeur dans $[1..N]$). L'ensemble C (resp. U) des variables contrôlables (resp. incontrôlables) contient une seule variable a (resp. b) représentant le nombre d'allumettes prises par le joueur A (resp. B). Les différents ensembles de contraintes sont indiqués ci-dessous. I exprime que A

joue en premier et qu'il y a au départ N allumettes (la variable p est arbitrairement initialisée avec la valeur N). E exprime que le jeu se termine quand il ne reste plus d'allumettes. F_c et F_u expriment qu'à chaque étape, un joueur ne peut pas prendre plus deux fois le nombre d'allumettes prises par le joueur précédent. T_c et T_u définissent la fonction de transition du système : on change de joueur et le nombre d'allumettes restantes est diminué du nombre d'allumettes prises.

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

3.2 Problèmes d'atteignabilité

Un MGCSP décrit la dynamique du système considéré et induit un ensemble de *trajectoires* possibles.

Définition 2 *Soit $M = (S, I, E, C, U, F_c, T_c, F_u, T_u)$ un MGCSP. L'ensemble des trajectoires induites par M est l'ensemble des séquences (possiblement infinies) d'état et de décision $seq : s_1 \xrightarrow{c_1} s_2 \xrightarrow{u_2} s_3 \xrightarrow{c_3} s_4 \xrightarrow{u_4} s_5 \cdots$ telles que :*

- *$I(s_1)$ est vrai et, pour tout état s_i qui n'est pas le dernier de la séquence, $E(s_i)$ est faux ;*
- *pour toute transition $s_i \xrightarrow{c_i} s_{i+1}$ de seq, $F_c(s_i, c_i)$ et $T_c(s_i, c_i, s_{i+1})$ sont vrais ;*
- *pour toute transition $s_i \xrightarrow{u_i} s_{i+1}$ de seq, $F_u(s_i, u_i)$ et $T_u(s_i, u_i, s_{i+1})$ sont vrais.*

Pour contrôler le système et restreindre ces évolutions possibles, nous utilisons des *politiques* π qui sont des fonctions de l'ensemble $\mathbf{d}(S)$ des états vers l'ensemble $\mathbf{d}(C)$ des décisions : $\pi(s) = c$ spécifie de prendre la décision c dans l'état s. Une politique peut être partielle (non définie pour certains états). Des politiques partielles sont utiles pour définir le contrôle uniquement sur l'ensemble des états atteignables. Nous sommes aussi intéressés par des politiques applicables qui spécifient uniquement des décisions faisables. Ces éléments sont formalisés ci-dessous.

Définition 3 *Une politique pour un MGCSP $M = (S, I, E, C, U, F_c, T_c, F_u, T_u)$ est une fonction partielle $\pi : \mathbf{d}(S) \to \mathbf{d}(C)$. Pour toute politique π, $\mathbf{d}(\pi)$ désigne le domaine de π. Une politique π est applicable ssi pour tout $s \in \mathbf{d}(\pi)$, $F_c(s, \pi(s))$ est vrai.*

L'ensemble des trajectoires induites par une politique π est l'ensemble des trajectoires $seq : s_1 \xrightarrow{c_1} s_2 \xrightarrow{u_2} s_3 \xrightarrow{c_3} s_4 \xrightarrow{u_4} s_5 \cdots$ induites par M et telles que $c_i = \pi(s_i)$ pour toute transition $s_i \xrightarrow{c_i} s_{i+1}$ de seq. Une trajectoire induite par une politique π est complète dans les trois cas suivants :

- *elle est infinie ;*
- *elle est finie et, dans le dernier état s_j, c'est au tour du $\forall$-joueur et $E(s_j)$ est vrai (état terminal) ;*
- *elle est finie et, dans le dernier état s_j, c'est au tour du $\exists$-joueur et $E(s_j)$ est vrai ou $s_j \notin \mathbf{d}(\pi)$ (état terminal ou politique non définie).*

Différentes exigences peuvent être imposées sur les trajectoires. Nous nous focalisons ici sur des exigences d'*atteignabilité* qui imposent que les trajectoires se terminent dans un état satisfaisant une certaine condition.

Définition 4 *Un problème d'atteignabilité est une paire (M, G) avec M un MGCSP sur un ensemble S de variables d'état et G un ensemble fini de contraintes sur S, appelées* contraintes de but. *Une solution de ce problème est une politique applicable π pour M telle que toutes les trajectoires complètes induites par π soient finies et se terminent dans un état s_i tel que $G(s_i)$ est vrai.*

Le problème *NimFibo* est un problème d'atteignabilité (M, G) avec M le MGCSP défini en Section 3.1 et $G : (p = B) \wedge (r = 0)$ (exigence d'atteindre un état où B doit jouer et il ne reste plus d'allumettes). Une politique solution est donnée dans la figure 1(b).

3.3 Relation avec le problème QCSP/QCSP+

Étant donné un problème d'atteignabilité (M, G) et un entier N, considérons le QCSP+ suivant, noté $Q_N(M, G)$, qui pourrait être mis sous forme normale prénexe :

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

$Q_N(M, G)$ peut être lu comme : "Est-il vrai que, pour tout état initial possible s_1, soit s_1 est un état

but, soit il n'est pas terminal et il existe une décision faisable c_1 qui induit l'état suivant s_2 tel que, soit s_2 est un état but terminal, soit il n'est pas terminal et pour toute décision faisable u_2 qui induit l'état suivant s_3, soit s_3 est un état but, soit il n'est pas terminal et il existe une décision faisable $c_3 \ldots$ telle que, soit s_{N-1} est un état but, soit il n'est pas terminal et il existe une décision faisable c_{N-1} qui induit l'état suivant s_N qui est un état but terminal ?".

Proposition 1 *Étant donné un problème d'atteignabilité (M, G) et un entier N, il existe une stratégie gagnante pour le QCSP $Q_N(M, G)$ ssi il existe une politique solution π pour (M, G) telle que toutes les trajectoires complètes induites par π soient de longueur inférieure ou égale à N.*

La proposition 1 implique qu'il est possible de résoudre le problème d'atteignabilité (M, G) en résolvant le QCSP $Q_N(M, G)$. Cependant, l'approche n'est pas complète à moins de choisir une valeur de N suffisamment grande : il est possible que le QCSP n'ait pas de solution alors que le problème d'atteignabilité en a. L'approche n'est complète que si on choisit une valeur de N suffisamment grande, par exemple égale au nombre d'états possibles ($N = |\mathbf{d}(S)|$). Mais, comme $|\mathbf{d}(S)|$ peut être très grand et comme le nombre de variables et de contraintes de $Q_N(M, G)$ est linéaire en N, cette approche peut ne pas être concrètement applicable. Cette relation entre problème d'atteignabilité (en contexte non déterministe) et QCSP/QBF est le pendant de la relation existante entre problème d'atteignabilité en contexte déterministe et CSP/SAT [6]. Dans une autre direction, la proposition 1 peut être vue comme le pendant de la propriété des processus de décision markoviens (MDPs [12]) indiquant que tout MDP a une politique optimale stationnaire (ne dépendant que de l'état et non de l'étape).

De plus, en termes d'espace nécessaire pour mémoriser une stratégie gagnante, la taille des politiques peut être exponentiellement plus petite que celle des arbres de politique, ce qui peut être utile quand on veut embarquer un contrôleur à bord d'un système autonome à mémoire limitée. Plus précisément, si R_π désigne l'ensemble des états atteignables en suivant π, la politique π peut être mémorisée comme une table contenant $|R_\pi|$ paires (s, c). Si W est une stratégie gagnante équivalente pour $Q_N(M, G)$ (induisant les mêmes trajectoires que π) représentée sous la forme d'un arbre de politique, la stratégie W peut contenir jusqu'à $|\mathbf{d}(U)|^{N/2}$ feuilles, ce qui est exponentiel en N et toujours supérieur ou égal à $|R_\pi|$.

4 Algorithme

L'algorithme proposé pour résoudre des problèmes d'atteignabilité sur des MGCSP est inspiré de techniques de planification non déterministe [2]. Une différence est l'utilisation de la programmation par contraintes pour raisonner sur les différentes relations.

4.1 Vue globale

L'algorithme est composé de trois fonctions :

- **reachMGCSP** responsable de l'exploration des différents états initiaux possibles ;
- **exploreC** responsable de l'exploration des différentes différentes décisions contrôlables faisables dans un état ;
- **exploreU** qui fait la même chose pour les décisions incontrôlables.

La recherche explore un arbre Et/Ou dans lequel les nœuds Ou correspondent aux décisions contrôlables et les nœuds Et aux décisions incontrôlables. L'arbre est exploré en profondeur d'abord et seuls les états atteignables à partir des états initiaux sont considérés.

Au cours de la recherche, l'algorithme maintient une politique courante π. Il associe à chaque état s une marque $Mark(s) \in \{Solved, Bad, Processing, None\}$. Une marque *Solved* signifie que l'état s a déjà été visité et que la politique courante permet d'atteindre à coup sûr le but en partant de s. Une marque *Bad* signifie qu'il n'existe aucune politique permettant d'atteindre à coup sûr le but en partant de s. Une marque *Processing* est associé aux états en cours d'exploration. Une marque *None*, qui reste implicite, est associé aux autres états. Dans l'implémentation existante, les marques sont mémorisées dans une table de *hashage* initialement vide (tous les états avec la marque *None*). On suppose l'existence d'une variable d'état booléenne indiquant si c'est au tour du $\exists$-joueur ou du $\forall$-joueur.

Une spécificité de l'algorithme concerne la gestion des *boucles*. Une boucle est une situation dans laquelle un état marqué *Processing* est de nouveau rencontré. Quand une boucle est rencontrée, cela signifie que l'adversaire (nature ou autre joueur) peut générer une trajectoire qui boucle indéfiniment sans que le but soit atteint. Par exemple, supposons que la figure 2 représente l'ensemble des trajectoires possibles d'un système et que le but soit d'atteindre l'état s_f. Les trajectoires $seq_1 : s_a \xrightarrow{c:0} s_b \xrightarrow{u:0} s_c \xrightarrow{c:0} s_d \xrightarrow{u:1} s_e \xrightarrow{c:0} s_b$ et $seq_2 : s_a \xrightarrow{c:0} s_b \xrightarrow{u:0} s_c \xrightarrow{c:0} s_d \xrightarrow{u:1} s_e \xrightarrow{c:1} s_d$ bouclent respectivement sur s_b et s_d. En conséquence, la trajectoire $seq_3 : s_a \xrightarrow{c:0} s_b \xrightarrow{u:0} s_c \xrightarrow{c:0} s_d \xrightarrow{u:1} s_e$ ne peut pas être étendue en une solution. L'ensemble $J = \{s_b, s_d\}$ est appelé la *justification de bouclage* de seq_3. Il correspond à l'ensemble des états précédents sur lesquels des

boucles ont été détectées en essayant d'étendre seq_3. La marque de s_e, le dernier état de seq_3, ne peut cependant pas être positionnée à *Bad* parce que les boucles détectées dépendent de décisions prises avant s_e. Pour $seq_4 : s_a \overset{c:0}{\rightarrow} s_b \overset{u:0}{\rightarrow} s_c \overset{c:0}{\rightarrow} s_d$ et $seq_5 : s_a \overset{c:0}{\rightarrow} s_b \overset{u:0}{\rightarrow} s_c$, la justification de bouclage est $\{s_b\}$. Pour $seq_6 : s_a \overset{c:0}{\rightarrow} s_b$, elle est vide. La marque de s_b peut donc être positionnée à *Bad*. Plus généralement, un état dont l'exploration échoue peut être marqué *Bad* si la justification courante de bouclage est vide.

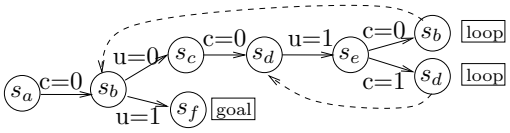


FIGURE 2 – Boucles dans la dynamique d'un système.

4.2 Pseudo-code

La fonction principale **reachMGCSP** prend en entrée un MGCSP M et un ensemble G de contraintes de but. Il retourne $(true, \pi)$ si (M, G) admet une politique solution π et $(false, \varnothing)$ sinon. Pour cela, la fonction **reachMGCSP** démarre avec une politique vide et analyse chaque état initial possible (la fonction *getSols* appelée en ligne 6 retourne l'ensemble des solutions d'un CSP et donc ici l'ensemble des états initiaux possibles). Chaque état initial s qui n'est pas un état but et dont la marque est différente de *Solved* est ensuite analysé. Si s est terminal ou est marqué *Bad*, le problème n'a pas de solution et $(false, \varnothing)$ est retourné (ligne 8). Sinon s est exploré par appel à la fonction **exploreC** (ligne 10).

```
1  Entrée : un MGCSP M et un ensemble G de
   contraintes de but
2  Sortie : une paire (b, π) avec b un booléen et π une
   politique
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)
```

La fonction **exploreC**(s, π) explore les décisions qui peuvent être prises dans l'état s. Elle retourne un triplet (b, π', J). b indique si la politique π peut être étendue de façon à ce que le but soit atteint à coup sûr depuis s. Si $b = true$, π' est la politique étendue. Si $b = false$, J est un ensemble d'états justifiant l'absence de solution à partir de s. J correspond à la justification de bouclage décrite précédemment. La première partie de **exploreC** (lignes 5 à 12) détermine toutes les décisions c qui sont faisables dans l'état s et tous les états suivants associés s' en raisonnant sur le CSP $F_c(S, C) \wedge T_c(S, C, S') \wedge (S = s)$. Si un état suivant s' obtenu en appliquant une décision c est un état but terminal ou est marqué *Solved*, alors il suffit de fixer $\pi(s) = c$ pour couvrir l'état s. La seconde partie de **exploreC** (lignes 13 à 34) parcourt l'ensemble des états suivants s' restant à explorer. Si s' est marqué *Solved*, alors l'état s est couvert (lignes 19 à 21). Sinon, si s' est marqué *Processing*, la justification de bouclage est étendue (lignes 22 et 23). Sinon, si s' est marqué *None*, il est exploré par appel à la fonction **exploreU** (ligne 26). Si cet appel retourne une politique solution étendue, s est marqué *Solved* et la politique étendue est retournée (lignes 27 à 29). Sinon, la justification de bouclage est étendue (ligne 30). Si tous les états suivants s' ont été explorés sans trouver de solution, alors s est retiré de la justification de bouclage, il est marqué *Bad* si la justification de bouclage est vide et un échec est retourné (lignes 31 à 34).

Le comportement de la fonction **exploreU** est similaire. Les seules différences sont les suivantes. Dans la première partie (lignes 5 à 15), un échec est retourné s'il existe un état suivant s' qui est un état non-but terminal ou qui est marqué *Bad*. Un échec est aussi retourné si une boucle est détectée. Dans la seconde partie (lignes 16 à 31), dans laquelle les états suivants restants sont explorés, si un état suivant s' est marqué *Bad*, un échec est retourné (lignes 20 à 22). Sinon, la fonction **exploreC** est appelée pour développer les différentes décisions faisables en s' (ligne 24).

Proposition 2 *L'algorithme* **reachMGCSP** *est correct et complet : il retourne* $(true, \pi)$ *avec* π *une politique solution si le problème d'atteignabilité* (M, G) *admet une solution et* $(false, \varnothing)$ *sinon.*

4.3 Améliorations algorithmiques

L'algorithme de base peut être amélioré sur différents points. Tout d'abord, dans la fonction **exploreC**, le CSP $F_c(S, C) \wedge T_c(S, C, S') \wedge G(S') \wedge E(S') \wedge (S = s)$ peut être considéré pour déterminer rapidement s'il existe une décision contrôlable permettant d'atteindre directement un état but terminal plutôt que d'énumérer toutes les solutions de $F_c(S, C) \wedge T_c(S, C, S') \wedge (S = s)$ et de vérifier en-

```
1  Entrée : un état s et une politique courante π
2  Sortie : un triplet (b, π', J) avec b un booléen, π' une
     politique étendue et J une justification de bouclage
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') ∧ E(s')) ∨ (Mark(s') = Solved) then
9              setMark(s, Solved) ;
10             return (true, π ∪ {(s, c)}, ∅)
11         else if ¬E(s') then
12             toExplore ← toExplore ∪ {(c, s')}
13     π' ← π
14     setMark(s, Processing)
15     J ← ∅
16     while toExplore ≠ ∅ do
17         Choose (c, s') ∈ toExplore ;
18         toExplore ← toExplore \ {(c, s')}
19         if Mark(s') = Solved then
20             setMark(s, Solved) ;
21             return (true, π' ∪ {(s, c)}, ∅)
22         else if Mark(s') = Processing then
23             J ← J ∪ {s'}
24         else if Mark(s') = None then
25             π' ← π' ∪ {(s, c)}
26             (covered, π', J') ← exploreU(s', π')
27             if covered then
28                 setMark(s, Solved) ;
29                 return (true, π', ∅)
30             else J ← J ∪ J' ; π' ← π' \ {(s, c)}
31     J ← J \ {s}
32     if J = ∅ then setMark(s, Bad)
33     else setMark(s, None)
34     return (false, π', J)
```

```
1  Entrée : un état s et une politique courante π
2  Sortie : un triplet (b, π', J) avec b un booléen, π' une
     politique étendue et J une justification de bouclage
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 ¬G(s') then
9              if E(s') ∨ (Mark(s') = Bad) ∨ (s = s')
               then
10                 setMark(s, Bad) ;
11                 return (false, π, ∅)
12             else if Mark(s') = Processing then
13                 return (false, π, {s'})
14             else if Mark(s') = None then
15                 toExplore ← toExplore ∪ {s'}
16     π' ← π
17     setMark(s, Processing)
18     while toExplore ≠ ∅ do
19         Choose s' ∈ toExplore ;
           toExplore ← toExplore \ {s'}
20         if Mark(s') = Bad then
21             setMark(s, Bad) ;
22             return (false, π', ∅)
23         else if Mark(s') = None then
24             (covered, π', J) ← exploreC(s', π')
25             if ¬covered then
26                 J ← J \ {s}
27                 if J = ∅ then setMark(s, Bad)
28                 else setMark(s, None)
29                 return (false, π', J)
30     setMark(s, Solved) ;
31     return (true, π', ∅)
```

suite si l'une d'elles satisfait $G(S') \wedge E(S')$. De façon similaire, il est possible de considérer le CSP $F_u(S,U) \wedge T_u(S,U,S') \wedge \neg G(S') \wedge E(S') \wedge (S = s)$ dans la fonction **exploreU** pour déterminer rapidement s'il existe une décision incontrôlable conduisant directement à un état non-but terminal.

En termes d'espace mémoire, l'algorithme mémorise les marques uniquement sur les états atteignables, mais ces états peuvent être très nombreux et la mémorisation coûteuse. Pour contourner cette difficulté, certaines marques peuvent être oubliées au cours de la recherche. L'algorithme reste valide, mais peut réexplorer certaines parties de l'espace de recherche. Cette option n'a pas été utilisée dans les expérimentations.

Finalement, nous considérons dans le cadre MGCSP que le $\exists$-joueur commence. Pour traiter des situations dans lesquelles le $\forall$-joueur commence, il suffit de remplacer l'appel à **exploreC** dans la fonction **reachMGCSP** par un appel à **exploreU**.

5 Expérimentations

Nous avons mené nos expérimentations sur un processeur Intel i5-520, 1.2GHz, 4GB RAM. Le temps de calcul maximum a été fixé à une heure. L'algorithme **reachMGCSP** est implémenté dans *Dyncode*, un outil développé au dessus de la librairie *Gecode*[1] de programmation par contraintes. Toute contrainte disponible dans *Gecode* peut être utilisée dans *Dyncode*. *Dyncode* a été initialement conçu pour traiter des problèmes de contrôle en environnement non déterministe et partiellement observable [11]. L'algorithme **reachMGCSP**, qui suppose un état complètement

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

observable, est cependant plus efficace que les algorithmes décrits dans [11]. Dans les expérimentations, nous avons utilisé *min-domain* pour l'heuristique de choix de variable et un ordre lexicographique pour le choix de valeur.

Nous avons d'abord comparé *Dyncode* avec *Qecode*[2], un solveur de QCSP+ construit sur *Gecode*. Nous avons mené des expérimentations sur trois jeux déjà modélisés dans la distribution *Qecode* : *NimFibo*, *Connect4* et *MatrixGame*. Les figures 3(a) à 3(c) montrent que, sur ces problèmes, *Dyncode* est supérieur à *Qecode* de plusieurs ordres de grandeur.

NimFibo Pour *NimFibo* (figure 3(a)), *Qecode* peut résoudre les instances jusqu'à 40 allumettes alors que *Dyncode* peut résoudre des instances impliquant plusieurs dizaines de milliers d'allumettes. La complexité temporelle observée avec *Dyncode* est même linéaire en fonction du nombre d'allumettes alors qu'on observe une évolution exponentielle avec *Qecode*. En d'autres termes, utiliser explicitement la notion d'état et mémoriser au cours de la recherche les états *good/nogood* casse la complexité.

Connect4 *Connect4* est un jeu à deux joueurs sur un tableau à 6 lignes et 7 colonnes. À chaque étape, un joueur place un jeton dans une colonne du tableau. Par gravité, le jeton tombe au bas de la colonne. Un joueur gagne s'il réussit à aligner 4 de ses jetons horizontalement, verticalement ou en diagonale. Le résultat du jeu est nul si le tableau est plein et sans aucun alignement. Comme dans la distribution *Qecode*, nous considérons une variante de ce jeu appelée *Connect4_Bounded* dont le but est de jouer sans perdre sur un nombre fixé d'étapes. La figure 3(b) montre que *Dyncode* résout plus d'instances que *Qecode*. Cette fois, on observe une évolution exponentielle avec les deux solveurs, mais l'évolution est plus lente pour *Dyncode*. La première explication est encore l'utilisation explicite de la notion d'état puisque, dans *Connect4*, plusieurs séquences de jeu peuvent conduire à la même configuration. La seconde est que *Qecode* crée initialement de nombreuses variables et contraintes pour déplier le problème sur l'horizon du jeu. Il peut ensuite réaliser ce qui est appelé une propagation en cascade sur le problème complet. A l'opposé, *Dyncode* propage les contraintes uniquement sur l'état courant et sur le suivant. Cela peut produire moins d'effacements, mais est réalisé beaucoup plus rapidement.

MatrixGame Dans le jeu *MatrixGame*, une matrice 0/1 de taille 2^d est considérée. À chaque tour, le $\exists$-joueur coupe la matrice horizontalement et garde le haut ou le bas. Le $\forall$-joueur coupe alors la matrice verticalement et garde la droite ou la gauche. Après d coups, la matrice est réduite à une case. Le $\exists$-joueur gagne si cette case contient la valeur 1. La figure 3(c) montre que *Dyncode* se comporte encore une fois mieux que *Qecode*, même si le modèle MGCSP est tel qu'un état ne peut pas être visité plus d'une fois. Nous pensons que c'est toujours parce que *Dyncode* ne raisonne que sur des CSP de petite taille.

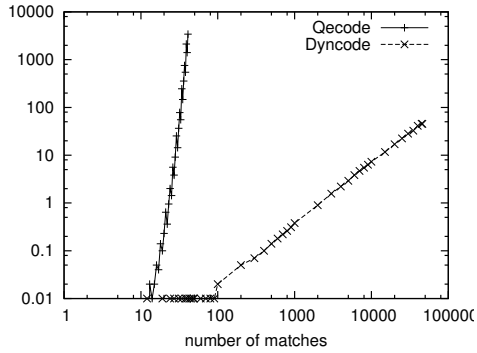
Dyncode a aussi été comparé avec *Queso*, un solveur de QCSP a priori plus efficace que *BlockSolve* or *QCSP-solve* [9]. Nous n'avons pas relancé *Queso* qui n'est plus maintenu et nous avons retenu les résultats indiqués dans [9], obtenus avec un processeur Pentium 4, 3.06GHz, 1GB RAM. Les résultats sont indiqués dans la figure 3(d) pour *Connect4*, mais cette fois avec des tableaux de taille $N \times M$ et l'objectif que le $\exists$-joueur remporte le jeu. Les résultats montrent que *Dyncode* est plus performant que *Queso*. Encore une fois, nous pensons que la notion d'état est ici vraiment utile pour éviter d'explorer plusieurs fois la même partie de l'espace de recherche. Alors que *Queso* utilise des techniques telles que la règle de valeur pure et la propagation de contraintes sur des contraintes disjonctives réifiées, la propagation de contraintes limitée réalisée par *Dyncode* limite le temps de calcul.

Dyncode pourrait être comparé avec d'autres outils : (a) des outils de planification non déterministe comme, par exemple, MBP [2], (b) des outils de synthèse de contrôleur venant de la communauté *model checking* [10] ou des outils de résolution de MDPs [12], en considérant les MGCSP comme des MDP avec des probabilités 0/1. Les algorithmes de résolution de MDP qui explorent tout l'espace d'état, tels que les algorithmes d'itération de la valeur ou de la politique, risquent d'être rapidement inefficaces. Des algorithmes de recherche dans un arbre Et/Ou peuvent être plus compétitifs, mais leur gestion des probabilités et des *backups* peut être pénalisante. Ces comparaisons expérimentales restent à faire. Un point important est le fait que ces outils n'offrent pas la flexibilité des modèles à base de contraintes.

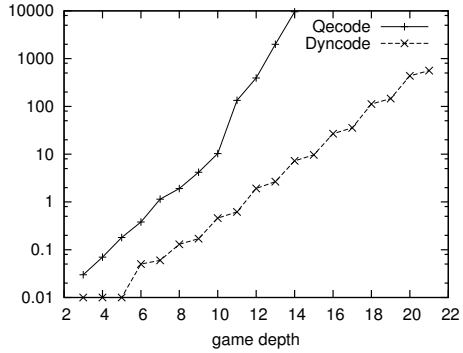
6 Conclusion

Ce papier a montré que, pour l'instant, utiliser le cadre QCSP/QCSP+ n'est pas la meilleure approche à base de contraintes pour modéliser et résoudre des problèmes de contrôle de systèmes dynamiques satisfaisant les hypothèses de dynamique markovienne et d'observabilité complète. L'utilisation de solveurs QCSP/QCSP+ est certainement plus appropriée pour résoudre des problèmes dans lesquels ces hypothèses ne sont pas vérifiées ou dans lesquels le nombre d'al-

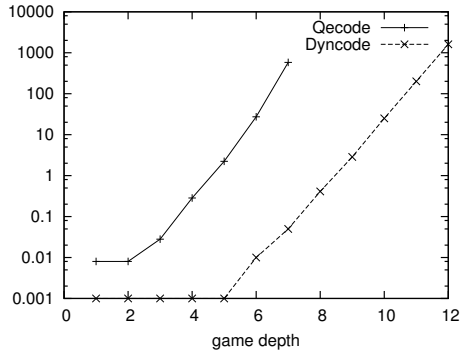
2. `http://www.univ-orleans.fr/lifo/software/qecode`



(a) *NimFibo*



(b) *Connect4_Bounded*



(c) *MatrixGame*

Taille du tableau		Temps de calcul (sec.)	
nCols	nLigns	*Queso* 3.06GHz,1GB RAM	*Dyncode* 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*

FIGURE 3 – Comparaison des temps de calcul obtenus avec *Dyncode*, *Qecode* (solveur QCSP+) et *Queso* (solveur QCSP) sur les jeux (a) *NimFibo*, (b) *Connect4_Bounded*, (c) *MatrixGame* et (d) *Connect4_Full*. Le temps de calcul en secondes est représenté en ordonnée sur une échelle logarithmique.

ternances de quantificateur reste limité. Pour le futur, nous envisageons d'étendre le cadre MGCSP pour modéliser des problèmes de contrôle dans lesquels le nombre de transitions incontrôlables entre deux étapes contrôlables n'est pas fixé. Des exigences sur les trajectoires plus générales que l'atteignabilité pourraient aussi être considérées.

Références

[1] Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. QCSP Made Practical by Virtue of Restricted Quantification. In *Proc. of IJCAI-07*, pages 38–43, 2007.

[2] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. of IJCAI-01*, pages 473–478, 2001.

[3] L. Bordeaux and E. Monfroy. Beyond NP : Arc-consistency for Quantified Constraints. In *Proc. of CP-02*, pages 371–386, 2002.

[4] Ian P. Gent, Peter Nightingale, and Andrew Rowley. Encoding Quantified CSPs as Quantified Boolean Formulae. In *Proc. of ECAI-04*, pages 176–180, 2004.

[5] Ian P. Gent, Peter Nightingale, and Kostas Stergiou. QCSP-Solve : A Solver for Quantified Constraint Satisfaction Problems. In *Proc. of IJCAI-05*, pages 138–143, 2005.

[6] H. Kautz and B. Selman. Planning as Satisfiability. In *Proc. of ECAI-92*, pages 359–363, 1992.

[7] R. Larson and J. Casti. *Principles of Dynamic Programming.* M. Dekker Inc., 1978.

[8] Nikos Mamoulis and Kostas Stergiou. Algorithms for Quantified Constraint Satisfaction Problems. In *Proc. of CP-04*, pages 752–756, 2004.

[9] Peter Nightingale. Non-binary Quantified CSP : Algorithms and Modelling. *Constraints*, 14(4) :539–581, 2009.

[10] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In *Proc. of the 7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380, 2006.

[11] C. Pralet, G. Verfaillie, M. Lemaître, and G. Infantes. Constraint-Based Controller Synthesis in Non-Deterministic and Partially Observable Domains. In *Proc. of ECAI-10*, pages 681–686, 2010.

[12] M. Puterman. *Markov Decision Processes, Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 1994.

[13] Kostas Stergiou. Repair-based Methods for Quantified CSPs. In *Proc. of CP-05*, pages 652–666, 2005.

[14] Guillaume Verger and Christian Bessiere. Blocksolve : a Bottom-up Approach for Solving Quantified CSPs. In *Proc. of CP-06*, pages 635–649, 2006.