Le problème de satisfaction de contraintes quantifiées et les jeux à deux joueurs à horizon fini : le projet QuaCode

Vincent Barichard, Igor Stéphan

Université d'Angers, 2 boulevard Lavoisier, 49045 Angers, France
vincent.barichard@univ-angers.fr,igor.stephan@univ-angers.fr

RÉSUMÉ. *Le problème de satisfaction de contraintes quantifiées (QCSP) est une généralisation du problème de satisfaction de contraintes (CSP) pour lequel les variables peuvent être quantifiées aussi bien existentiellement qu'universellement. Le concept de QCSP offre un cadre naturel pour traiter des problèmes PSPACE comme par exemple les jeux à deux joueurs à horizon fini et information complète ou la planification sous incertitude. Nous présentons à travers trois exemples comment les QCSP peuvent être utilisés pour modéliser des jeux à deux joueurs : le* Jeu de Nim*, le* MatrixGame *et le* Puissance Quatre*. Les solveurs QCSP de l'état de l'art ont un défaut majeur : ils explorent un espace combinatoire plus important que l'espace de recherche naturel du problème original car ils sont inaptes à reconnaître que certains sous-problèmes sont nécessairement vrais. Nous proposons dans le cadre des jeux à deux joueurs à horizon fini une solution efficace pour utiliser les solveurs QCSP et une réponse aussi simple qu'élégante à ce parcours superflu. Nous proposons un solveur QCSP construit au-dessus de la librairie CSP GeCode que nous comparons aux solveurs de l'état de l'art.*

ABSTRACT. *Quantified Constraint Satisfaction Problems (QCSP) are a generalization of Constraint Satisfaction Problems (CSP) in which variables may be quantified existentially and universally. QCSP offers a natural framework to express PSPACE problems as finite two-player games with perfect information or planing under uncertainty. We present how QCSP may be used to model two-player games on three classical games : Nim game, MatrixGame and Connect Four. State-of-the-art QCSP solvers have an important drawback: they explore much larger combinatorial space than the natural search space of the original problem since they are unable to recognize that some sub-problems are necessarily true. We propose a very simple and elegant solution to use efficiently QCSP to design finite two-player games. Our QCSP solver built over GeCode, a CSP library, obtained very good results compared to state-of-the-art QCSP solvers.*

MOTS-CLÉS : *jeu à deux joueurs à horizon fini, problème de satisfaction de contraintes quantifiées, QCSP.*

KEYWORDS: *finite two-players game, quantified constraint satisfaction problem, QCSP.*

DOI:10.3166/RIA.31.337-365

1. Introduction

Le problème de satisfaction de contraintes quantifiées (ou QCSP pour *Quantified Constraint Satisfaction Problem*) est une généralisation du problème de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problem*) dans laquelle les variables peuvent être non seulement quantifiées existentiellement (comme dans les CSP) mais aussi universellement (Bordeaux, Monfroy, 2002). Un QCSP est une alternance de variables existentiellement ou universellement quantifiées portant sur des domaines finis suivie par un CSP. Un CSP est alors une collection de contraintes portant sur des variables prenant leurs valeurs dans des domaines finis. Un QCSP peut être vu comme un jeu à deux joueurs à horizon fini dans lequel les variables existentiellement quantifiées représentent les coups d'un joueur A et les variables universellement quantifiées les coups d'un joueur B. Un QCSP est valide si le joueur A a une stratégie pour gagner c'est-à-dire une stratégie pour affecter des valeurs aux variables existentiellement quantifiées de telle manière que, quelles que soient les valeurs affectées aux variables universelles par le joueur B, le CSP est vrai. Le joueur A tente ainsi de rendre vraies toutes les contraintes du CSP tandis que le joueur B tente au moins d'en violer une.

Voici un exemple issu de la combinatoire : le problème du boulanger. Ce problème consiste à aider un boulanger souhaitant acquérir quatre poids distincts à choisir dans l'intervalle $\{1, \ldots, 40\}$ kg, lui permettant de peser n'importe quelle quantité entière de farine dans l'intervalle $\{1, \ldots, 40\}$ kg en utilisant une balance de Roberval. Pour la pesée, chaque poids p_i peut être placé de n'importe quel côté de la balance ou ne pas être utilisé. Ce problème peut être modélisé par le QCSP suivant : sachant que $p_1, p_2, p_3, p_4, f \in \{1, \ldots, 40\}$, $e_1, e_2, e_3, e_4 \in \{-1, 0, 1\}$, alors

$$\exists p_1 \exists p_2 \exists p_3 \exists p_4 \forall f \exists e_1 \exists e_2 \exists e_3 \exists e_4 (\sum_{i=1}^{4} p_i \times e_i = f).$$

Les emplacements e_i indiquent pour chaque poids p_i s'il est placé sur le plateau à côté de la farine (-1), sur l'autre plateau (1) ou absent de la balance (0). Pour chaque poids de farine à peser les emplacements sont différents et dépendent de ce poids. Le problème du boulanger admet une unique stratégie gagnante $\{1, 3, 9, 27\}$ (modulo les permutations des variables p_i) si on considère que ce qui importe au boulanger est de savoir quels poids il doit acheter. Sinon le reste de la stratégie gagnante est une fonction $\{1, \ldots, 40\} \times \{1, 2, 3, 4\} \to \{-1, 0, 1\}$.

L'étude des QCSP est récente mais il existe un réel intérêt pour mettre au point des techniques efficaces pour résoudre de tels problèmes (Chen *et al.*, 2015 ; Börner *et al.*, 2009 ; Gent *et al.*, 2004 ; 2008 ; 2005 ; Bordeaux *et al.*, 2005 ; Benedetti *et al.*, 2007 ; Bordeaux, Zhang, 2007 ; Börner *et al.*, 2003 ; Benedetti *et al.*, 2008 ; Verger, Bessiere, 2008 ; Pralet, Verfaillie, 2011 ; Mamoulis, Stergiou, 2004). Cette extension est pleine de promesses car elle permet de coder de manière plus compacte certains problèmes et même d'en modéliser d'autres qui ne peuvent l'être en CSP. Mais cette extension accroît la complexité du problème de décision, passant de NP-complet (pour un CSP classique) à PSPACE-complet. De même, le calcul d'au moins une stratégie gagnante

pour un QCSP relève de la classe de complexité PSPACE (Stockmeyer, Meyer, 1973). Il n'y a donc qu'un mince espoir (gardant les hypothèses classiques de la théorie de la complexité (Papadimitriou, 1994)) d'obtenir un algorithme efficace pour résoudre un QCSP quelconque. Mais il en est de même pour SAT et CSP dont la complexité est NP-complet or nous connaissons les succès des solveurs SAT et CSP pour certaines classes de problèmes. Il est possible qu'il en soit de même pour PSPACE, comme le suggère les nombreuses classes non triviales de QCSP, dans le cadre des jeux combinatoires, identifiées comme étant polynomiale en temps (Börner *et al*., 2009).

Le domaine des QCSP est à l'intersection de deux domaines : le domaine des CSP (Tsang, 1993) et celui des QBF (Cadoli *et al*., 1998) (les formules booléennes quantifiées ou QBF étant des QCSP restreintes aux booléens). Il n'y a, à l'aune de nos connaissances que trois solveurs QCSP, excepté le nôtre, QuaCode (Barichard, Stéphan, 2014 ; Barichard, Stéphan, 2014) : BlockSolve (Verger, Bessiere, 2006), Queso (Gent *et al*., 2008) et Qecode (Benedetti *et al*., 2008). BlockSolve est basé sur un algorithme de Fourier-Motzkin par élimination de quantificateurs. Queso est basé sur un algorithme de recherche quantifié et son approche est très proche de la nôtre. Qecode est aussi un solveur basé sur un algorithme de recherche quantifié mais est dédié à une extension des QCSP (QCSP+), qui propose en plus une forme restreinte de la quantification, et qui est utilisé principalement pour spécifier des jeux finis à deux joueurs. Ce dernier solveur est toujours maintenu tandis que les deux autres ne le sont plus.

Comme un solveur QCSP basé sur un algorithme de recherche quantifié peut être vu comme une extension d'un solveur CSP basé sur un algorithme de recherche, il est particulièrement pertinent de chercher à construire un solveur QCSP au-dessus de technologies pour solveur CSP sans les changer. Cette approche est la nôtre : nous implantons QuaCode, un solveur QCSP, au-dessus de GeCode, une bibliothèque de classes pour la gestion des contraintes d'un CSP, sans changer le cœur de la bibliothèque.

La plupart des solveurs de l'état de l'art, tout comme QuaCode ont un inconvénient majeur : ils parcourent un espace combinatoire bien plus grand que l'espace de recherche du problème original. Si, pour la plupart des problèmes, il n'y a pas de solution systématique, il n'en va pas de même pour les jeux à deux joueurs à horizon fini. Nous proposons dans ce cadre un théorème qui prouve l'élimination du parcours superflu.

Nous présentons tout d'abord en section 2 le concept de problème de satisfaction de contraintes quantifiées. Nous spécifions, en section 3, trois jeux à deux joueurs à horizon fini en QCSP : le *Jeu de Nim*, le *MatrixGame* ainsi que le *Puissance Quatre*. Nous montrons en particulier toute l'élégance de ces spécifications en QCSP. Nous abordons, en section 4, la spécification d'un solveur QCSP comme extension d'un solveur CSP. Nous présentons, en section 5, le « talon d'Achille » des QCSP, c-à-d. le problème de l'exploration des espaces de recherche inutile par construction dans les solveurs QCSP comme extensions de solveurs CSP et la solution que nous y apportons. Enfin, en section 6, nous présentons notre solveur QCSP QuaCode qui intègre

notre solution au problème du « talon d'Achille » et le comparons à l'état de l'art. Nous continuons par une discussion dans laquelle nous présentons QuaCode comme un système ouvert, capable en tant que solveur complet de coopérer avec des méthodes incomplètes via une interface pour orienter sa recherche. Nous concluons en donnant un ensemble de perspectives.

2. Problème de satisfaction de contraintes quantifiées

2.1. Syntaxe des QCSP

Le symbole $\mathcal{PS}$ représente les symboles propositionnels, le symbole $\exists$ représente le quantificateur existentiel et le symbole $\forall$ représente le quantificateur universel. Le symbole $\wedge$ représente la conjonction logique, le symbole $\vee$ représente la disjonction logique, le symbole $\rightarrow$ représente l'implication logique, le symbole $\leftrightarrow$ représente l'équivalence logique, le symbole $\top$ représente ce qui est toujours vrai et le symbole $\bot$ représente ce qui est toujours faux. Le symbole $\equiv$ représente l'équivalence entre formules.

Un QCSP est un quintuplet $(\mathbf{V}, \mathbf{order}, \mathbf{quant}, \mathbf{D}, \mathbf{C})$: $\mathbf{V}$ est un ensemble de n variables, **order** est une bijection de $\mathbf{V}$ dans $[1..n]$, **quant** est une fonction de $\mathbf{V}$ dans $\{\exists, \forall\}$ ($\mathbf{quant}(v)$ dénote le quantificateur associé à la variable v), $\mathbf{D}$ est une fonction de $\mathbf{V}$ dans l'ensemble des domaines $\{D(v_1), \ldots, D(v_n)\}$ telle que, pour toute variable $v_i \in \mathbf{V}$, $D(v_i)$ en dénote son domaine, i.e. l'ensemble fini de toutes les valeurs possibles ($D(v)$ est le domaine associé à la variable v), $\mathbf{C}$ est un ensemble de contraintes. Si $v_{j_1}, \ldots, v_{j_m}$ sont les variables d'une contrainte $c_j \in \mathbf{C}$ alors la relation associée à c_j est un sous-ensemble du produit cartésien $D(v_{j_1}) \times \cdots \times D(v_{j_m})$. Dans ce qui suit, pour chaque $i \in [1..n]$, $q_{v_i} = \mathbf{quant}(v_i)$ et $D_{v_i} = D(v_i)$.

Un QCSP $(\mathbf{V}, \mathbf{order}, \mathbf{quant}, \mathbf{D}, \mathbf{C})$ est généralement représenté par sa formule en logique du premier ordre $q_{v_1} v_1 \ldots q_{v_n} v_n \bigwedge_{c_j \in \mathbf{C}} c_j$ avec $v_1 \in D_{v_1}$, …, $v_n \in D_{v_n}$, $\mathbf{order}(v_i) = i$, pour chaque $i \in [1..n]$. Avec cette représentation $q_{v_1} v_1 \ldots q_{v_n} v_n$ est appelé « lieur » (un lieur vide étant noté ε).

Par exemple, le QCSP $(\{x, y, z, t\}, \mathbf{order}, \mathbf{quant}, \{\{0, 1, 2\}\}, \mathbf{C})$ avec

$$\begin{cases} \mathbf{order} = \{(1, x), (2, y), (3, z), (4, t)\}, \\ \mathbf{quant} = \{(x, \exists), (y, \exists), (z, \forall), (t, \exists)\}, \\ D(x) = D(y) = D(z) = D(t) = \{0, 1, 2\}, \\ \mathbf{C} = \{(x = (y * z) + t), (t \leq x)\} \end{cases}$$

est noté : $\exists x \exists y \forall z \exists t((x = (y * z) + t) \wedge (t \leq x))$ avec $x, y, z, t \in \{0, 1, 2\}$.

2.2. Sémantique des QCSP

Une stratégie est un arbre pour un lieur $Q = q_{v_1} v_1 \ldots q_{v_n} v_n$ avec $v_1 \in D_{v_1}$, …, $v_n \in D_{v_n}$, $Q = q_{v_1} v_1 \ldots q_{v_n} v_n$ est un arbre tel que :

– chaque nœud feuille est étiqueté avec le symbole $\square$ et à la profondeur n,

– chaque nœud interne à la profondeur k, $0 \leq k < n$, est étiqueté avec la variable v_{k+1},

– chaque arc reliant un nœud de profondeur k à l'un de ses fils est étiqueté par un élément de D_{v_k},

– toutes les étiquettes des arcs reliant un nœud à l'un de ses fils sont différentes,

– chaque nœud étiqueté par une variable existentiellement quantifiée admet un unique fils et

– chaque nœud étiqueté par une variable universellement quantifiée dont le domaine est de taille k admet k nœuds fils.

Un scénario est une séquence d'étiquettes $val_1, \ldots, val_n$ sur un chemin (v_1, val_1), …, (v_n, val_n), $val_i \in D_{v_i}$ pour tout i, $1 \leq i \leq n$, d'une stratégie pour un lieur $q_{v_1} v_1 \ldots q_{v_n} v_n$. Un scénario correspond à la notion d'instanciation dans un CSP où toutes les variables sont instanciées à une valeur de leur domaine. Un scénario val_1, …, val_n pour un QCSP $(\mathbf{V}, \mathbf{order}, \mathbf{quant}, \mathbf{D}, \mathbf{C})$ tel que $\mathbf{V} = \{v_1, \ldots, v_n\}$ est un scénario prometteur si $(\bigwedge_{1 \leq i \leq n} v_i = val_i) \wedge (\bigwedge_{c_j \in \mathbf{C}} c_j)$ est vrai ; un tel scénario correspond à l'instanciation complète $v_1 = val_1$, …, $v_n = val_n$; c'est un scénario prometteur car il satisfait toutes les contraintes. Un scénario prometteur correspond à la notion de solution dans un CSP. Une stratégie est une stratégie gagnante si tous les scénarios qui la composent sont des scénarios prometteurs. Ses scénarios sont les scénarios gagnants. Un scénario qui n'est dans aucune stratégie gagnante est un scénario perdant. Un scénario est en conflit avec le CSP sous-jacent si $(\bigwedge_{1 \leq i \leq n} v_i = val_i) \wedge (\bigwedge_{c_j \in \mathbf{C}} c_j)$ est faux (au moins une contrainte est violée). Un scénario peut-être à la fois prometteur et perdant.

Nous pouvons donner une sémantique plus intuitive, récursive, pour la définition d'une stratégie gagnante d'un QCSP. Prenons un QCSP QC, deux cas sont possibles : soit la première variable est quantifiée universellement et nous avons le QCSP $\forall x QC$, soit elle est quantifiée existentiellement et nous avons le QCSP $\exists x QC$. Le QCSP $\forall x QC$ avec $x \in D_x$ admet une stratégie gagnante si et seulement si, pour tout $val \in D_x$, $Q(C \wedge (x = val))$ admet une stratégie gagnante. Le QCSP $\exists x QC$ avec $x \in D_x$ admet une stratégie gagnante si et seulement si, pour au moins un $val \in D_x$, $Q(C \wedge (x = val))$ admet une stratégie gagnante.

Par exemple, la stratégie représentée à la figure 1 est une stratégie gagnante pour le QCSP

$$\exists x \exists y \forall z \exists t((x = (y * z) + t) \wedge (t \leq x)), x, y, z, t \in \{0, 1, 2\}$$

puisque $(0 = (0 * 0) + 0)$, $(0 = (0 * 1) + 0)$ et $(0 = (0 * 2) + 0)$. Le scénario $0, 0, 2, 0$, qui correspond à l'instanciation complète $(x = 0)$, $(y = 0)$, $(z = 2)$ et $(t = 0)$, est un scénario prometteur puisque $0 = (0 * 2) + 0$ et est aussi un scénario gagnant puisque faisant partie de la stratégie gagnante ci-dessous. Le scénario $2, 2, 0, 2$, qui correspond à l'instanciation complète $(x = 2)$, $(y = 2)$, $(z = 0)$ et $(t = 2)$, est aussi un scénario prometteur puisque $2 = (2 * 0) + 2$ mais est un fait un scénario perdant

puisqu'il ne fait partie d'aucune stratégie gagnante ($\exists t((2 = (2 * 1) + t) \wedge (t \leq x)), t \in \{0, 1, 2\}$ ne peut avoir de solution). Le scénario $2, 2, 2, 2$ est en conflit avec le CSP $((x = (y * z) + t) \wedge (t \leq x))$.

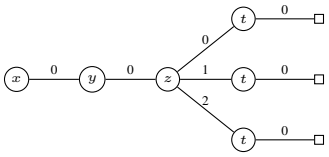


Figure 1. Exemple de stratégie gagnante

Nous utilisons aussi la représentation sous forme d'un QCSP+. En effet, un QCSP+ recouvre syntaxiquement les QCSP de la forme

$$
\begin{array}{l}
Q\phi = \exists x_1 \forall y_1 \ldots \exists x_{n-1} \forall y_{n-1} \exists x_n \\
R_\exists(x_1) \wedge (R_\forall(x_1, y_1) \rightarrow \\
\quad (\ldots \quad (R_\exists(x_1, y_1, \ldots, x_{n-1}) \wedge \\
\qquad\quad (R_\forall(x_1, y_1, \ldots, x_{n-1}, y_{n-1}) \rightarrow \\
\qquad\quad (R_\exists(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n) \wedge \\
\qquad\quad F(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n)))) \ldots))
\end{array}
$$

avec $x_1 \in D_{x_1}, y_1 \in D_{y_1}, \ldots, x_{n-1} \in D_{x_{n-1}}, y_{n-1} \in D_{y_{n-1}}, x_n \in D_{x_n}$.

Cela peut apparaître comme une restriction, mais il n'en n'est rien puisqu'il suffit de mettre $R_\exists(x_1, y_1, \ldots, x_i) \equiv \top$ pour tout i, $i \leq n$, et $R_\forall(x_1, y_1, \ldots, x_i, y_i) \equiv \top$ pour tout i, $i < n$, pour obtenir à nouveau un QCSP quelconque $QF(x_1, \ldots, x_n)$.

3. Exemples de jeux à deux joueurs à horizon fini modélisés en QCSP

Parmi les jeux à horizon fini, certains sont bien adaptés pour être modélisés par un QCSP. En effet, le large choix de contraintes disponibles dans une bibliothèque CSP permet de modéliser les règles et les conditions de victoires de jeux complexes. Le lieur lui permet de représenter chaque joueur ainsi que l'alternance des tours de jeux. Nous présentons ici trois jeux : le *Jeu de Nim*, le *MatrixGame* et le *Puissance Quatre*. Ces trois jeux sont sélectionnées car ils permettent d'apprécier différentes caractéristiques des QCSP. Le *Jeu de Nim* est un jeu dans lequel des règles doivent être satisfaites entre chaque tour de jeu. Son modèle en QCSP fait donc apparaître une hiérarchie de règles qui est prise en compte lors de la résolution. Le *MatrixGame* s'exprime très simplement grâce à l'utilisation d'une contrainte globale, il démontre l'intérêt d'un formalisme puissant de haut niveau tel que les CSP. Enfin, le *Puissance Quatre* est un jeu populaire qui nécessite un grand nombre de variables et de contraintes pour être modélisé. Bien que difficile à résoudre pour la taille de grille usuelle, c'est un excellent challenge pour solveur QCSP.

3.1. Le Jeu de Nim

Le *Jeu de Nim* est un jeu à deux joueurs qui se joue avec un tas de pièces ou d'allumettes. Le but du jeu est d'être celui qui se saisira de la dernière allumette. Chaque joueur peut prendre de une à trois allumettes. Dans la variante de Fibonacci, le minimum est d'une allumette et le maximum pour le premier tour est du nombre initial d'allumettes moins une. Puis chaque joueur peut prendre d'une jusqu'au double d'allumettes que son adversaire à pris au tour précédent. Le joueur existentiel débute. Avec un nombre pair n d'allumettes, le QCSP est le suivant (x_i est le nombre d'allumettes choisies au tour i): $R_\exists(1) = \top$ (Le joueur existentiel choisit entre 1 et $n-1$ allumettes, toutes les valeurs du domaine de x_1 sont possibles) et pour tout i, $1 < i \leq \frac{n}{2}$, $R_\exists(i) = (x_i \leq 2 * y_{i-1}) \wedge (x_i + \Sigma_{1 \leq j < i}(x_j + y_j) \leq n)$ et pour tout i, $1 \leq i \leq \frac{n}{2}$, $R_\forall(i) = (y_i \leq 2 * x_i) \wedge (\Sigma_{1 \leq j \leq i}(x_j + y_j) \leq n)$ (chaque joueur prend d'une jusqu'au double d'allumettes que son adversaire à pris au tour précédent et ne peut pas prendre plus d'allumettes que ce qu'il reste dans le tas) :

$$\exists x_1 \forall y_1 \ldots \exists x_{n-1} \forall y_{n-1} \\ R_\exists(1) \wedge (R_\forall(1) \rightarrow (R_\exists(2) \wedge (R_\forall(2) \rightarrow (\ldots (R_\forall(n-1) \rightarrow \bot)))))$$

avec $x_1, y_1, \ldots, x_{n-1}, y_{n-1} \in [1..n-1]$.

Pour $n = 4$, nous obtenons le QCSP:

$$\begin{array}{l} \exists x_1 \forall y_1 \exists x_2 \forall y_2 \\ \quad (((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)) \rightarrow \\ \quad (((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \wedge \\ \quad (((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)) \rightarrow \bot))) \\ \quad \text{avec } x_1, y_1, x_2, y_2 \in \{1, 2, 3\} \end{array} \quad (1)$$

La figure 2 montre la première stratégie gagnante (sous forme d'arbre) obtenue à partir de l'arbre de recherche de la figure 3 par une simple application de la sémantique des quantificateurs.

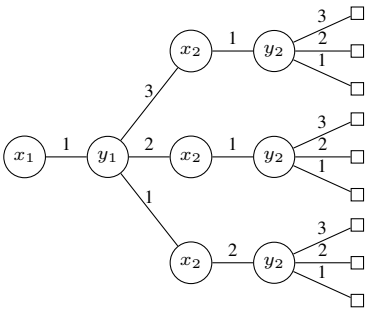


Figure 2. Première stratégie gagnante pour le problème Nim avec la variante de Fibonacci ($n = 4$)

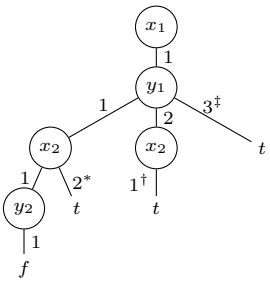


Figure 3. Arbre de recherche pour le problème Nim avec la variante de Fibonacci ($n = 4$)

Pour l'instanciation $(x_1 = 1), (y_1 = 1), (x_2 = 1)$, le QCSP est réduit à $\forall y_2(((y_2 \leq 2) \wedge (y_2 \leq 1)) \rightarrow \bot) \equiv \bot$ avec $y_2 \in \{1, 2, 3\}$ et pour l'instanciation $(x_1 = 1)$, $(y_1 = 1)$, $(x_2 = 2)$ ($*$ de la figure 3), le QCSP est réduit à $\forall y_2(((y_2 \leq 4) \wedge (y_2 \leq 0)) \rightarrow \bot) \equiv \top$ avec $y_2 \in \{1, 2, 3\}$.

Pour l'instanciation $(x_1 = 1)$, $(y_1 = 2)$, $(x_2 = 1)$ († de la figure 3), le QCSP est réduit à $\forall y_2(((y_2 \leq 2) \wedge (y_2 \leq 0)) \rightarrow \bot) \equiv \top$ avec $y_2 \in \{1, 2, 3\}$.

Pour l'instanciation $(x_1 = 1)$, $(y_1 = 3)$ ($\ddagger$ de la figure 3), le QCSP est réduit à $\exists x_2 \forall y_2 \lambda \equiv \top$ avec $x_2, y_2 \in \{1, 2, 3\}$ et $\lambda = ((3 \leq 2) \rightarrow (((x_2 \leq 6) \wedge (x_2 \leq 0)) \wedge (((y_2 \leq 2 * x_2) \wedge ((x_2 + y_2) \leq 0)) \rightarrow \bot)))$. Pour chacune des instanciations partielles $*$, † et $\ddagger$, toutes les branches sont vraies.

3.2. Le MatrixGame

Le *MatrixGame* est un jeu à deux joueurs de d tours. Il est joué sur une matrice de $0/1$ de taille 2^d. À chaque tour, le joueur existentiel coupe horizontalement la matrice en deux et décide s'il faut conserver la partie haute ou basse. Puis, le joueur universel coupe la matrice verticalement et décide s'il faut conserver la partie gauche ou droite. Le joueur existentiel gagne si la dernière case contient 1.

Le *MatrixGame* est modélisé de la manière suivante :

$$\text{Soit le lieur} \quad \exists x_1 \forall x_2 \ldots \exists x_{n-1} \forall x_n \quad x_i \in \{0, 1\} \tag{1}$$

$$\text{alors} \quad Board[Idx] = 1 \tag{2}$$

$$\text{avec} \quad Idx = \sum_{i=1}^{n} access_i x_i, \quad Idx \in [0; l] \tag{3}$$

$$l = 2^d * 2^d$$

$$n = 2 * d$$

Le lieur (1) définit les variables de décision du problème, chacune prend sa valeur dans $\{0, 1\}$. Une alternance de quantificateurs correspond à un tour de jeu. Ainsi, pour un problème de d-tours, il y a $n = 2 * d$ variables de décisions et la matrice de jeu comporte $l = 2^{d^2}$ cases. La matrice de jeu est l'unique entrée du problème. Chacune de ses cases étant égale soit à 0 soit à 1. C'est le vecteur $Board$ utilisé dans la contrainte (2) qui représente la matrice de jeu dans notre modèle. La variable Idx représente l'ensemble des indices des cases encore accessibles d'ici la fin du jeu. Pour gagner, le joueur existentiel doit donc vérifier la contrainte 2 qui assure qu'il reste au moins une case égale à 1 dans le reste des cases atteignables, et ce jusqu'au dernier tour. Dans un CSP, cette relation est modélisée grâce à la contrainte `element`.

Le vecteur $access$ utilisé dans la contrainte (3) permet de connecter les variables de décision aux valeurs de la matrice de jeu. Il permet en effet de retrouver la case finale (la dernière restante) de la matrice de jeu en fonction des variables de décision x_i. Il est calculé avant la résolution du problème, il est donc constant pour celui-ci. Sa taille correspond au nombre de variables de décision du problème (2 au minimum). Il est calculé de la manière suivante :

$$access_n = 1$$
$$access_{n-1} = l$$
$$access_i = access_{i+2} * 2$$

Pour un jeu à 3 tours, nous obtenons le vecteur suivant :

32	4	16	2	8	1

Ainsi, il est possible de calculer l'indice de la case finale. Soit la partie réalisée par la séquence de coupes (0,1,1,0,1,1), l'indice (modélisée par la variable Idx) de la case finale est 29, ce qui correspond à la case de coordonnées ligne 4 et colonne 5. Si la valeur de cette case est égale à 1, le joueur existentiel gagne. Cette partie correspond à un scénario prometteur.

Résoudre le QCSP du *MatrixGame* signifie découvrir une stratégie gagnante pour ce jeu. Cette stratégie permet au joueur existentiel de gagner quelques soient les coups de son adversaire. Dans ce jeu, il n'y a pas de règle à vérifier au cours de la partie comme pour le *Jeu de Nim*. Seule la valeur de la dernière case est pertinente. Ainsi, l'arbre de recherche est complet et nécessite d'explorer complètement tous les scénarios.

3.3. Le Puissance Quatre

Le jeu du *Puissance Quatre* est un jeu bien connu qui se joue à deux joueurs sur une grille comptant le plus souvent 6 rangées et 7 colonnes. Le joueur qui gagne la

partie est le premier à aligner 4 de ses pions (horizontalement, verticalement ou en diagonal). Tour à tour, les deux joueurs placent un pion dans la colonne de leur choix, le pion descend alors jusqu'à la position la plus basse disponible dans la colonne. Si toute la grille est remplie et qu'aucun joueur n'a réalisé d'alignement, la partie est déclarée nulle.

La modélisation donnée ici peut être utilisée pour un nombre variables de lignes et de colonnes variable mais supérieur à quatre (le nombre minimum de pions à aligner). Le nombre de colonnes sera appelé nbC et le nombre de lignes nbL. Elle est traduite et adaptée de celle de P. Nightingale dans (Nightingale, 2009). L'approche est la suivante : à chaque tour de jeu est associée une grille représentant l'état du jeu à ce tour, celle-ci sera liée aux grilles des tours précédents et suivants. Chaque case d'une grille peut prendre trois états : libre, occupée par le joueur 1 ou occupée par le joueur 2. Pour assurer la cohérence des grilles au fur et à mesure des tours de jeux, des contraintes de liaisons seront ajoutées au modèle. Ainsi, si au tour de jeu k, la case de coordonnées $(2, 4)$ est occupée par le joueur 1, dans toutes les grilles des tours $\geq k$, la case de coordonnées $(2, 4)$ sera également occupée par le joueur 1.

Le nombre de variables de décision correspond au nombre de cases dans la grille de départ, soit $nbL \times nbC$ variables chacune à valeur dans l'intervalle $[1..nbC]$. Le lieur est formé d'une alternance de quantificateurs $\exists\forall$ fonction du nombre de variables de décisions. Nous obtenons le problème suivant :

$$\text{Soit le lieur} \quad \exists m^1 \forall u^2 \dots \exists m^{n-1} \forall u^n \tag{4}$$

$$\text{avec } \{m^i, u^i\} \in [1 \dots nbC] \tag{5}$$

$$i \in [1 \dots nbL \times nbC] \tag{6}$$

$$\text{tel que} \quad \mathcal{C} \tag{7}$$

Ici, $\mathcal{C}$ représente l'ensemble des contraintes du problème à satisfaire.

Pour chaque tour de jeu, donc pour chaque i, il est nécessaire d'introduire un ensemble de variables intermédiaires comme suit :

– Si i est pair (tour du joueur $\forall$), alors ajout d'une nouvelle variable $\exists m^i$ qui sera liée à la variable $\forall u^i$ correspondante.

– $\exists b^i_{r,c} \in \{rond, croix, vide\}$ avec $r \in [1 \dots nbL]$ et $c \in [1 \dots nbC]$. Chaque $b^i_{r,c}$ représente l'état du tableau après le mouvement i.

– $\exists h^i_c \in [0 \dots nbL], c \in [1 \dots nbC]$ représentant la hauteur de la colonne c après le tour i.

– $\exists vainqueur^i \in \{rond, croix, aucun\}$ représentant le gagnant après le tour i.

– $\exists aligne^i \in \{0, 1\}$ représentant la présence d'un alignement après le tour i.

– $\exists al^i_z \in \{0, 1\}$ représentant la présence d'un alignement dans chaque, ligne, colonne ou diagonale, numérotée z après le tour i (un alignement ne concerne que les

pions du joueur qui joue le tour i).

– $\exists mh_c^i \in \{0,1\}, c \in [1 \dots nbC]$, qui est égale à 1 quand le pion au tour i a été placé dans la colonne c.

– $\exists pos_{r,c}^i \in \{0,1\}, r \in [1 \dots nbL], c \in [1 \dots nbC]$, qui est égale à 1 quand la case de coordonnées (r,c) est libre pour le tour i.

– $\exists tour\_fini^i \in \{0,1\}$ qui est égal à 1 quand le tour est terminé et que toutes les contraintes du tour sont satisfaites.

Ces variables sont liées entre elles ainsi qu'aux variables de décisions grâce aux 7 groupes de contraintes suivants :

1. Si i est pair (tour du joueur $\forall$), la contrainte suivante est ajoutée pour chaque colonne c : $(vainqueur^{i-1} \neq aucun) \vee (h_c^{i-1} = nbL) \vee ((u^i = c) \rightarrow (m^i = c))$

2. Si i est pair, un pion $croix$ est placé dans la grille. Pour chaque colonne c, la valeur de la variable mh_c^i est choisie en fonction du mouvement effectué par le joueur (a-t-il été effectué dans cette colonne ou non). Pour cela, l'ensemble des contraintes suivantes est ajouté :

- $((m^i = c) \wedge (h_c^{i-1} \neq nbL)) \leftrightarrow ((mh_c^i = 1) \vee (tour\_fini^i = 1))$
- Pour chaque $r \in [1 \dots nbL], c \in [1 \dots nbC]$:

remplir la colonne complètement

$(h_c^{i-1} = r-1) \rightarrow (pos_{r,c}^i = 1)$

$(b_{r,c}^i \neq rond \wedge b_{r+1,c}^i = aucun \wedge b_{r+2,c}^i = aucun \wedge \dots \wedge b_{nbL,c}^i = aucun) \Leftrightarrow (pos_{r,c}^i = 1)$

connecter mh_c^i *à* $b_{r,c}^i$

$(mh_c^i = 1 \wedge h_c^{i-1} = r-1) \rightarrow (b_{r,c}^i = croix)$

$(mh_c^i \neq 1 \wedge h_c^{i-1} = r-1) \rightarrow (b_{r,c}^i = aucun)$

Si i est impair, des contraintes analogues sont ajoutées en inversant $rond$ et $croix$.

3. Les contraintes suivantes assurent la cohérence entre la grille du tour $i-1$ et la grille du tour i : pour chaque $r \in [1 \dots nbL], c \in [1 \dots nbC]$, $((b_{r,c}^{i-1} = rond) \rightarrow (b_{r,c}^i = rond)) \wedge ((b_{r,c}^{i-1} = croix) \rightarrow (b_{r,c}^i = croix))$

4. Les contraintes suivantes assurent la cohérence entre la hauteur d'une colonne et la grille de jeu : pour chaque $r \in [1 \dots (nbL+1)], c \in [1 \dots nbC]$: $(b_{r-1,c}^i \neq aucun \wedge b_{r,c}^i = aucun) \rightarrow (h_c^i = r-1)$.

5. Détection d'un alignement : soit bg_z un groupe numéroté z de quatre variables formant un alignement de 4 pions (lignes, colonnes ou diagonales). Par exemple, le groupe $bg_1 = (b_{1,1}^i, b_{2,1}^i, b_{3,1}^i, b_{4,1}^i)$). Alors $\max(z) = nbL + nbC + 2*(nbL-3) + 2*(nbC-3) - 2$. Si i est impair, pour tout z, $(aligne^{i-1} = 1 \vee bg_z[1] \neq rond \vee bg_z[2] \neq rond \vee bg_z[3] \neq rond \vee bg_z[4] \neq rond) \Leftrightarrow al_z^i \neq 1$. Si i est pair, des contraintes analogues sont ajoutées en inversant $rond$ et $croix$.

6. Détection globale d'un alignement au tour i : $l_1^i \vee l_2^i \vee \dots \vee l_{\max(z)}^i \Leftrightarrow aligne^i$.

7. Si i est impair, les contraintes suivantes assurent la détermination du vainqueur :

- $(vainqueur^{i-1} = rond) \rightarrow (vainqueur^i = rond)$
- $(vainqueur^{i-1} = croix) \rightarrow (vainqueur^i = croix)$
- $(vainqueur^{i-1} = aucun \wedge aligne^i = 1) \rightarrow (vainqueur^i = rond)$
- $(vainqueur^{i-1} = aucun \wedge aligne^i = 0) \rightarrow (vainqueur^i = aucun)$.

Si i est pair, des contraintes analogues sont ajoutées en inversant $rond$ et $croix$.

Certaines contraintes font référence à celles du tour précédent, pour les contraintes du premier tour (pour $i = 1$), les valeurs suivantes sont utilisés : $vainqueur^0 = aucun, al^0 = 0, b^0_{r,c} = aucun, b^i_{0,c} = rond, h^0_c = 0, tour\_fini_0 = 1$. Il faut de plus assurer que le joueur $\exists$ remporte le jeu, donc $vainqueur^{nbL \times nbC} = rond$. Pour le premier tour, la symétrie est cassée en retirant la partie gauche du domaine de départ. Ainsi, le domaine de la variable $\exists m_1$ devient $\{\lfloor \frac{nbC}{2} \rfloor \dots nbC\}$.

4. Un solveur QCSP comme extension d'un solveur CSP

Pour un solveur QCSP conçu comme une extension d'un solveur CSP, l'ensemble des contraintes est considéré comme une conjonction. Quand un problème est représenté pour être une entrée d'un tel solveur QCSP, les contraintes sont issues d'un ensemble de contraintes primitives du langage. Cependant un problème représentant un jeu à deux joueurs est rarement exprimé comme tel mais plutôt par une grande formule logique, que nous appellerons QCSP *complexe* (ie. contenant des contraintes booléennes qui ne sont pas des conjonctions). Par exemple, dans un jeu fini à deux joueur sur n tours, le problème est exprimé en deux parties. Il commence par une alternance de vérifications des règles pour le joueur existentiel ($R_\exists$) d'une part et pour son adversaire, le joueur universel, ($R_\forall$) d'autre part. Il termine par une vérification des conditions de victoire pour le joueur existentiel (VC):

$$
\begin{array}{l}
\exists x_1 \forall y_1 \dots \exists x_{n-1} \forall y_{n-1} \exists x_n \\
R_\exists(x_1) \wedge (R_\forall(x_1, y_1) \rightarrow \\
\quad (\dots \quad (R_\exists(x_1, y_1, \dots, x_{n-1}) \wedge \\
\qquad (R_\forall(x_1, y_2, \dots, x_{n-1}, y_{n-1}) \rightarrow \\
\qquad (R_\exists(x_1, y_2, \dots, x_{n-1}, y_{n-1}, x_n) \wedge \\
\qquad VC(x_1, y_2, \dots, x_{n-1}, y_{n-1}, x_n)))) \dots))
\end{array} \tag{8}
$$

Pour transformer ce QCSP complexe en une entrée pour un solveur QCSP, des symboles propositionnels existentiellement quantifiés (dits de Tseitin) sont introduits. Ainsi le QCSP du *Jeu de Nim* (1) est réécrit en :

$$\begin{array}{l}\exists x_1 \forall y_1 \exists x_2 \forall y_2 \exists o_1 \exists o_2 \\ \quad (((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)) \rightarrow o_1) \wedge \\ \quad (o_1 \leftrightarrow (((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \wedge o_2)) \wedge \\ \quad (o_2 \leftrightarrow (((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)) \rightarrow \bot)) \\ \quad \text{avec } x_1, y_1, x_2, y_2 \in \{1, 2, 3\}, o_1, o_2 \in \textbf{BOOL}\end{array} \quad (9)$$

Algorithme 1 Un algorithme de recherche quantifié pour solveur de QCSP

Entrée: Un QCSP QC
Sortie: **vrai** si le QCSP QC admet au moins une stratégie gagnante et **faux** sinon

```
pileRetourArrière := ∅
tant que true faire
 selon atteintPointFixe(C) faire
  cas échec
   retour sur le dernier choix existentiel (x, k) de pileRetourArrière
   si aucune alors retourner faux
   sinon ajouter à C la contrainte (x = k)
  cas succès
   retour sur le dernier choix universel (x, k) de pileRetourArrière
   si aucune alors retourner vrai
   sinon ajouter à C la contrainte (x = k)
  cas branche
   sélectionner la variable non instanciée suivante x
   pour tout k ∈ D(x) empiler (x, k) dans pileRetourArrière
   sélectionner le premier choix (x, k) de pileRetourArrière
   ajouter à C la contrainte (x = k)
 fin selon
fin tant que
```

Nous verrons dans la section suivante que cette transformation n'affecte pas uniquement la modélisation du problème mais aussi le parcours de l'espace de recherche associé. Pour l'instant nous décrivons l'algorithme 1 qui présente la structure d'un algorithme de recherche quantifié pour un solveur QCSP conçu comme une extension d'un solveur CSP. Cet algorithme est basé sur une boucle perpétuelle. Au départ le pile de retour arrière portant sur le choix des variables est vide. La fonction $reachfixpoint$ calcule le point fixe de la propagation et retourne trois différents codes possibles : *échec* s'il y a une contrainte violée, $succès$ si l'ensemble des contraintes est vrai et $branche$ sinon. Dans le cas de *échec*, les derniers choix sur les variables universellement quantifiées sont sautés car ils ne peuvent pas mener à un succès, le choix sur la dernière variable existentiellement quantifiée (x, k) est dépilé de la pile de retour arrière et la nouvelle contrainte $(x = k)$ est insérée dans l'ensemble de contraintes courant pour explorer cette nouvelle possibilité pour la variable x existentiellement quantifiée. S'il n'y a plus de choix sur une quelconque variable existentiellement quantifiée (il ne reste plus que des choix sur des variables universellement

quantifiées ou plus du tout) alors le QCSP n'a pas de stratégie gagnante et l'algorithme retourne faux. Dans le cas de $succès$, les derniers choix sur les variables existentiellement quantifiées sont sautés car un seul succès est suffisant de par la sémantique du quantificateur existentiel, le choix sur la dernière variable universellement quantifiée (x, k) est dépilé de la pile de retour arrière et la nouvelle contrainte $(x = k)$ est insérée dans l'ensemble de contraintes courant pour continuer à construire une stratégie gagnante pour toutes les valeurs possibles de x. S'il n'y a plus de choix sur une quelconque variable universellement quantifiée (il ne reste plus que des choix sur des variables existentiellement quantifiées ou plus du tout) alors le QCSP a une stratégie gagnante et l'algorithme retourne vrai. Sinon, l'appel à $reachfixpoint(C)$ a retourné $branche$: l'algorithme continue d'avancer à travers le lieur Q et sélectionne la prochaine variable x non instanciée et empile sur la pile de retour arrière tous les choix de la variable. Le premier choix de la variable (x, k) est dépilé de la pile de retour arrière et la nouvelle contrainte $(x = k)$ est insérée dans l'ensemble de contraintes pour explorer la première possibilité de la variable x.

5. Solveur QCSP et jeux à deux joueurs à horizon fini

La plupart des solveurs de l'état de l'art, quasiment tous basés sur l'introduction des symboles propositionnels de Tseitin, ont un inconvénient majeur, le « talon d'Achille » : ils parcourent un espace combinatoire bien plus grand que l'espace de recherche du problème original [1] qui est su comme étant inutile par construction (ce qui est le cas par exemple pour les sous-arbres $*$, † et ‡ de l'arbre de recherche de la figure 4 pour le solveur QCSP sur le *Jeu de Nim*). Cette notion inclut la capture des actions illégales du joueur universel mais aussi, par exemple, la détection de la fin du jeu avant le dernier tour qui est aussi une source d'accroissement de l'espace de recherche exploré [2].

Même si le QCSP complexe et le QCSP issu de l'introduction de symboles propositionnels de Tseitin sont équivalents, le second est bien plus complexe à résoudre car de nombreuses instances satisfiables sont explorées durant la recherche. Ces branches sont alors combinées (au niveau des variables universellement quantifiées) pour cons-

1. Le problème de l'exploration d'un espace de recherche trop grand a été déjà pointé du doigt dans la communauté QBF (Ansotegui *et al*., 2005) : Dans le cas d'un jeu à deux joueurs, les contraintes booléennes deviennent nécessairement vraies grâce à une action illégale du joueur universel qui conduit immédiatement à la victoire du joueur existentiel. Dans leur enfance, les solveurs QBF avaient du mal à détecter que les contraintes booléennes étaient nécessairement vraies sous une assignation partielle. Comme les solveurs QBF sont basés pour la plupart sur un algorithme de recherche quantifié pouvant être vu comme une extension de l'algorithme de recherche des solveurs SAT, les premiers solveurs QBF étaient quasiment tous basés sur des solveurs SAT existants. Ces solveurs QBF ont traités principalement le problème du « talon d'Achille » en changeant le cœur du solveur (introduisant par exemple des variables indicatrices qui capturent les actions illégales du joueur universel (Ansotegui *et al*., 2005), exploitant la représentation sous forme de circuit (Goultiaeva, Bacchus, 2010) ou plus généralement tenant compte de la dualité des problèmes QBF (Sabharwal *et al*., 2006)).

2. Dans (Ansotegui *et al*., 2005), si le joueur existentiel a déjà gagné alors toute nouvelle action du joueur universel est considérée comme illégale.

truire des stratégies gagnantes. Une propriété de la logique sous-jacente très importante pour l'efficacité est perdue dans le solveur : la propriété d'absorption du $\top$ par rapport à la disjonction i.e. si un argument d'une disjonction n-aire est vrai alors il est inutile de calculer les autres arguments, la disjonction est vraie. Si $R(x_1, \ldots, x_i)$ est vrai pour une instanciation quelconque alors il n'est pas nécessaire de résoudre $q_{x_{i+1}} x_{i+1} \ldots q_{x_n} x_n F(x_1, \ldots, x_n)$ pour cette instanciation mais le solveur le fait. La figure 4 l'illustre en explicitant l'arbre de recherche pour le *Jeu de Nim* selon le QCSP (9).

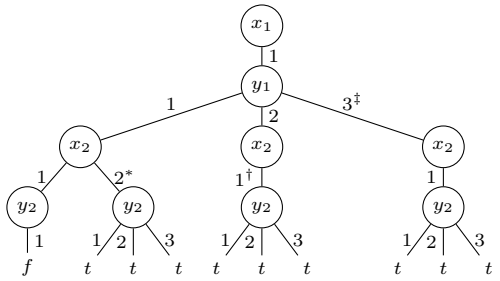


Figure 4. Arbre de recherche pour le problème Nim avec la variante de Fibonacci $(n = 4)$ *pour un solveur QCSP*

Par construction, la propriété d'absorption de $\bot$ par rapport à la conjonction est assurée par un solveur (Q)CSP : si une contrainte est fausse, puisque la file des contraintes est une conjonction n-aire, il n'est pas nécessaire de calculer les autres contraintes, le solveur échoue puis revient en arrière.

Dans un solveur pour QCSP, la propriété d'absorption de $\top$ par rapport à la disjonction n'est plus assurée. Même si pour une instanciation quelconque la conjonction de contraintes est vraie (et donc $((o_1 \vee o'_1) \leftrightarrow \top)$ est aussi vrai), le solveur QCSP tentera de résoudre le QCSP $q_{x_{i+1}} x_{i+1} \ldots q_{x_n} x_n \exists o_1 \ldots \exists o_m C(x_1, \ldots, x_n)$ pour cette instanciation (où C est l'ensemble de contraintes issue de F par introduction des symboles propositionnels o_1, …, o_m) . En d'autres termes, pour détecter un succès, dans un solveur QCSP, toutes les contraintes doivent être satisfaites et tant qu'il demeure des variables non instanciées ou des contraintes non résolues, l'algorithme 1 de recherche quantifié ne peut pas décider.

Si la puissance d'un solveur QBF relève principalement de l'algorithme de recherche quantifié et de sa représentation interne (incluant des techniques d'apprentissage de clauses et de cubes), la puissance d'un solveur (Q)CSP relève plus de son catalogue de contraintes. Mais il apparaît bien plus difficile d'implanter les techniques liés à la dualité dans les solveurs QCSP que dans les solveurs QBF puisqu'il faut alors disposer des contraintes duales. Ainsi, il nous est apparu utile de construire un solveur QCSP au-dessus des technologies CSP *sans les toucher*. Pour traiter le problème du « talon d'Achille » sans toucher à la librairie CSP, nous proposons un outil très simple pour les solveurs QCSP, l'*outil coupure*, inspiré de la coupure de Prolog, comme étant un outil sous la responsabilité de celui qui spécifie le QCSP, pour élaguer l'espace de

recherche des parties qui sont par construction inutiles. Cet outil ne demande qu'une seule chose : que nous soyons capable de dire au solveur QCSP si le QCSP courant est résolu.

L'outil *coupure* est défini, pour un ensemble de contraintes S et une contrainte e comme $coupure(S, e)$: si $\neg(\bigwedge_{c \in S} c \rightarrow e)$ est vrai alors le QCSP courant est résolu.

Le théorème suivant met en exergue que cet outil est particulièrement puissant dans les jeux à deux joueurs à horizon fini puisqu'il permet d'éliminer le « talon d'Achille » : les stratégies gagnantes sont conservées (premier point du théorème) alors que l'espace de recherche du QCSP initial est recouvré (second point du théorème).

THÉORÈME 1. — *Soit $Q\phi$ le QCSP complexe suivant spécifiant un jeu à deux joueurs :*

$$\begin{array}{l} Q\phi = \exists x_1 \forall y_1 \ldots \exists x_{n-1} \forall y_{n-1} \exists x_n \\ R_\exists(x_1) \wedge (R_\forall(x_1, y_1) \rightarrow \\ \quad (\ldots \quad (R_\exists(x_1, y_1, \ldots, x_{n-1}) \wedge \\ \qquad\quad (R_\forall(x_1, y_1, \ldots, x_{n-1}, y_{n-1}) \rightarrow \\ \qquad\quad (R_\exists(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n) \wedge \\ \qquad\quad VC(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n)))) \ldots)) \end{array}$$

avec $x_1 \in D_{x_1}, y_1 \in D_{y_1}, \ldots, x_{n-1} \in D_{x_{n-1}}, y_{n-1} \in D_{y_{n-1}}, x_n \in D_{x_n}$.

Soient C l'ensemble de contraintes obtenu de la réécriture de la formule ϕ par l'introduction des symboles propositionnels de Tseitin o_1, …, o_m et l'ensemble

$$\begin{array}{l} \Sigma = \{coupure(\{R_\forall(x_1, y_1), \ldots, R_\forall(x_1, y_1, \ldots, x_{i-1}, y_{i-1})\}, \\ \quad (R_\exists(x_1, y_1, \ldots, x_i) \rightarrow R_\forall(x_1, y_1, \ldots, x_i, y_i)) \ ) \mid 1 \leq i < n \ \} \end{array}$$

de coupures. Les propositions suivantes sont vérifiées ($Q_\exists = \exists o_1 \ldots \exists o_m$) :

– Soit s une stratégie pour $QQ_\exists$. s est une stratégie gagnante pour $QQ_\exists C$ si et seulement si s est une stratégie gagnante pour $QQ_\exists(C \wedge \Sigma)$.

– Soit T un arbre de recherche pour Q. T est un arbre de recherche de $Q\phi$ si et seulement si T est un arbre de recherche pour $QQ_\exists(C \wedge \Sigma)$.

PREUVE. — Le premier point du théorème est une conséquence immédiate du second puisque si les arbres de recherche sont identiques, les stratégies gagnantes aussi. Prouvons le second point. Soit la propriété P suivante : « T' est un sous arbre de hauteur $2i$ d'un arbre de recherche pour $Q\phi$ si et seulement si T' est aussi un sous arbre d'un arbre de recherche pour $QQ_\exists(C \wedge \Sigma)$ ». La preuve se fait sur les alternances de quantificateurs $\exists\forall$. Nous notons $R_\exists(x_1, y_1, \ldots, x_k)$ et $R_\forall(x_1, y_1, \ldots, x_k, y_k)$ respectivement $R^k_\exists$ et $R^k_\forall$.

Supposons $i = 1$. Si $R^1_\exists$ est vrai, soit $R^1_\forall$ est vrai alors le nœud est ouvert (la branche n'est pas décidée) aussi bien dans T_ϕ (l'arbre de recherche pour $Q\phi$) que $T_{(C \wedge \Sigma)}$ (l'arbre de recherche pour $QQ_\exists(C \wedge \Sigma)$) car dans ce dernier cas la coupure $coupure(\emptyset, (R^1_\exists \rightarrow R^1_\forall))$ n'est pas appliquée ($(R^1_\exists \rightarrow R^1_\forall) \equiv \top$), soit $R^1_\forall$ est faux et

alors c'est un nœud fermé et $succès$ aussi bien dans T_ϕ car $(R^1_\exists \wedge (R^1_\forall \rightarrow \psi)) \equiv \top$ que dans $T_{(C \wedge \Sigma)}$ car dans ce dernier cas la coupure $coupure(\emptyset, (R^1_\exists \rightarrow R^1_\forall))$ est appliquée ($(R^1_\exists \rightarrow R^1_\forall) \equiv \bot$). Si $R^1_\exists$ est faux alors le nœud est fermé et $échec$ aussi bien dans T_ϕ car $(R^1_\exists \wedge (R^1_\forall \rightarrow \psi)) \equiv \bot$ que dans $T_{(C \wedge \Sigma)}$ car dans ce dernier cas la coupure $coupure(\emptyset, (R^1_\exists \rightarrow R^1_\forall))$ n'est pas appliquée ($(R^1_\exists \rightarrow R^1_\forall) \equiv \top$).

Nous supposons que la propriété est vraie au rang $i-1$ et nous démontrons qu'elle reste vraie au rang i. Un nœud dans le sous arbre est soit ouvert soit fermé et dans ce dernier cas, le résultat est alors $succès$ ou $échec$. Par hypothèse de récurrence, T' est un sous arbre de hauteur $2(i-1)$ d'un arbre de recherche T_ϕ pour $Q\phi$ si et seulement si T' est aussi un sous arbre d'un arbre de recherche $T_{(C \wedge \Sigma)}$ pour $QQ_\exists(C \wedge \Sigma)$. Les lieurs sur Q étant identiques et l'algorithme de construction des arbres étant le même un nœud est ouvert dans l'un si et seulement s'il est ouvert dans l'autre. Ce qui signifie que pour tout k, $k < i$, les $R^k_\forall$ et les $R^k_\exists$ sont vrais. Au rang i, seule la coupure $coupure(\{R^1_\forall, \ldots, R^{i-1}_\forall\}, (R^i_\exists \rightarrow R^i_\forall))$ peut s'appliquer. Maintenant un nœud de profondeur $2i$ est un nœud soit :

(ouvert) : $\Rightarrow$ Si le nœud est un nœud ouvert de T_ϕ alors $R^i_\exists$ et $R^i_\forall$ sont vrais donc la coupure $coupure(\{R^1_\forall, \ldots, R^{i-1}_\forall\}, (R^i_\exists \rightarrow R^i_\forall))$ ne s'applique pas (car $(R^i_\exists \rightarrow R^i_\forall) \equiv \bot$) donc le nœud est aussi un nœud ouvert dans $T_{(C \wedge \Sigma)}$.

$\Leftarrow$ Si le nœud est un nœud ouvert dans $T_{(C \wedge \Sigma)}$ alors $R^i_\exists$ et $R^i_\forall$ sont vrais donc le nœud est aussi un nœud ouvert dans T_ϕ.

(fermé et $succès$) : $\Rightarrow$ Si le nœud est un nœud fermé et $succès$ dans T_ϕ alors $R^i_\exists$ est vrai et $R^i_\forall$ est faux donc la coupure $coupure(\{R^1_\forall, \ldots, R^{i-1}_\forall\}, (R^i_\exists \rightarrow R^i_\forall))$ s'applique puisque ($(R^i_\exists \rightarrow R^i_\forall) \equiv \bot$) donc le nœud est aussi fermé et $succès$ dans $T_{(C \wedge \Sigma)}$.

$\Leftarrow$ Si le nœud est un nœud fermé et $succès$ dans $T_{(C \wedge \Sigma)}$ alors c'est par application de la coupure $coupure(\{R^1_\forall, \ldots, R^{i-1}_\forall\}, (R^i_\exists \rightarrow R^i_\forall))$ donc $(R^i_\exists \rightarrow R^i_\forall) \equiv \bot$ donc $R^i_\forall$ est faux et $R^i_\exists$ est vrai donc le nœud est aussi un nœud fermé et $succès$ dans T_ϕ ($(R^i_\exists \wedge (R^i_\forall \rightarrow \psi)) \equiv \top$).

Enfin, un nœud de profondeur $2i-1$ est un nœud

(fermé et $échec$) $\Rightarrow$ Si le nœud est un nœud fermé et $échec$ dans T_ϕ alors une des contraintes de C est violée or la coupure $coupure(\{R^1_\forall, \ldots, R^{i-1}_\forall\}, (R^i_\exists \rightarrow R^i_\forall))$ est non applicable. En effet, tous les $\{R^1_\forall, \ldots, R^{i-1}_\forall\}$ sont vrais et $R^i_\exists$ est faux. C'est donc aussi un nœud fermé et $échec$ dans $T_{(C \wedge \Sigma)}$.

$\Leftarrow$ Si le nœud est un nœud fermé et $échec$ dans $T_{(C \wedge \Sigma)}$ alors une des contraintes de C est violée donc c'est aussi un nœud fermé et $échec$ dans T_ϕ.

■

Dans le cas d'un jeu à deux joueurs à horizon fini, chaque coupure

$$coupure(\quad \{R_\forall(x_1, y_1), \ldots, R_\forall(x_1, y_1, \ldots, x_{i-1}, y_{i-1})\}, \\ (R_\exists(x_1, y_1, \ldots, x_i) \rightarrow R_\forall(x_1, y_1, \ldots, x_i, y_i)))$$

s'exprime en logique ainsi :

$$\begin{array}{rlll} \neg(& (& \bigwedge_{1\leq k\leq i} R_\forall(x_1,y_1,\ldots,x_{k-1},y_{k-1})) \\ & \rightarrow & (R_\exists(x_1,y_1,\ldots,x_i) \rightarrow R_\forall(x_1,y_1,\ldots,y_{i-1},x_i,y_i)) \ ) \\ \equiv \quad (& (& \bigwedge_{1\leq k\leq i} R_\forall(x_1,y_1,\ldots,x_{k-1},y_{k-1})) \\ & \wedge & R_\exists(x_1,y_1,\ldots,x_i) \wedge \neg R_\forall(x_1,y_1,\ldots,y_{i-1},x_i,y_i) \ ) \end{array}$$

ce qui permet d'arrêter dès que le joueur universel triche (i.e. les joueurs ont respecté les règles jusqu'au coup i pour le joueur existentiel et $i-1$ pour le joueur universel puis le joueur universel a triché à son coup i).

Dans le cas de la modélisation du *Jeu de Nim*, le théorème demande d'ajouter deux nouvelles coupures :

$$\begin{array}{l} coupure(\emptyset, (\top \rightarrow ((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)))) \\ coupure(\quad \{(x_2 \leq 2 * y_1), ((x_1 + y_1 + x_2) \leq 4)\}, \\ \qquad\qquad (((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \rightarrow \\ \qquad\qquad ((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)))) \end{array}$$

La première coupure élimine le sous arbre sous ‡ de la figure 4 :

$$[x_1 \leftarrow 1][y_1 \leftarrow 3](\neg((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)))$$

est vrai.

La seconde coupure élimine les deux sous arbres sous $*$ et † de la figure 4 : $C = (x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)$ et

$$\begin{array}{l} e = (((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \rightarrow \\ \qquad ((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4))) \end{array}$$

alors

$(*)$ $[x_1 \leftarrow 1][y_1 \leftarrow 1][x_2 \leftarrow 2](C)$ est vrai et $[x_1 \leftarrow 1][y_1 \leftarrow 1][x_2 \leftarrow 2](e)$ est faux puisque $y_2 \geq 0$ alors $[x_1 \leftarrow 1][y_1 \leftarrow 1][x_2 \leftarrow 2](\neg(C \rightarrow e))$ est vrai.

$(\dagger)$ $[x_1 \leftarrow 1][y_1 \leftarrow 2][x_2 \leftarrow 1](C)$ est vrai et $[x_1 \leftarrow 1][y_1 \leftarrow 2][x_2 \leftarrow 1](e)$ est faux puisque $y_2 \geq 0$ alors $[x_1 \leftarrow 1][y_1 \leftarrow 2][x_2 \leftarrow 1](\neg(C \rightarrow e))$ est vrai.

Cela permet de retrouver l'arbre de recherche de la figure 3.

Dans le cas du *Puissance Quatre*, le modèle présenté au paragraphe 3.3, bien que fonctionnel, a l'inconvénient de devoir compléter intégralement tous les tours de jeu pour déterminer le vainqueur même si celui-ci a été décidé plus tôt dans le jeu. Pour résoudre ce problème, il est possible d'utiliser la détection de valeurs pures (voir (Nightingale, 2009)). Ce problème peut aussi être résolu pour l'ajout de coupures. Ainsi, pour chaque tour de jeu i tel que i est pair, la coupure suivante est ajoutée : $coupure(\{vainqueur^i = rond, tour\_fini_i = 1\}, \bot)$. Cela garantit que la partie se termine dès que le vainqueur a été déterminé.

Le lien entre QCSP et jeux à deux joueurs a été présenté pour le cas où la stratégie calculée est une stratégie gagnante pour un jeu à horizon fini et à connaissance parfaite. Mais dans la pratique, une telle approche est intenable car soit il n'y a pas de stratégie gagnante initiale auquel cas le calcul réalisé n'est d'aucune aide, soit l'espace de recherche est tellement grand qu'il est illusoire de vouloir le parcourir, soit le jeu n'est pas à horizon fini ; voire les trois. Si nous disposions d'une fonction d'évaluation f pour une instanciation partielle, alors nous pourrions remplacer la condition de victoire $VC(x_1, y_1, \ldots, x_n)$ d'un QCSP par un test de franchissement de seuil s : $f(x_1, y_1, \ldots, x_n) \geq s$. La stratégie alors calculée n'est plus une stratégie gagnante (au sens du jeu) mais une stratégie qui garantit au joueur $\exists$ que quoique le joueur $\forall$ joue, son $n^{ème}$ coup aura une évaluation supérieure ou égale au seuil.

6. Le solveur QuaCode et l'état de l'art

Nous avons développé QuaCode [3] un solveur QCSP construit au-dessus de la librairie CSP GeCode [4]. Ainsi, toute contrainte présente dans GeCode est disponible dans QuaCode. QuaCode implémente l'algorithme 1 pour chercher une stratégie gagnante.

Nous avons réaliser un ensemble de tests sur des Intel i5, 1.8GHz, 4GB RAM tournant sous Linux. Chaque exécution n'utilise qu'un seul cœur et le temps maximum de calcul est fixé à une demie heure. Dans notre expérimentation, nous comparons QuaCode aux deux autres solvers de l'état de l'art : Queso [5] et Qecode [6]. Queso est un solver de QCSP écrit *from scratch* en Java par P. Nightingale. Queso n'est plus maintenu mais les sources sont toujours disponibles. Qecode est un solver QCSP/QCSP+ basé sur GeCode. Pour résoudre un QCSP+, Qecode construit des problèmes CSP dont toutes les variables sont quantifiées existentiellement. L'algorithme de recherche en profondeur d'abord de GeCode est appelé à chaque fois qu'il faut résoudre un CSP et les résultats sont capturés et analysés par Qecode. Pour résoudre un QCSP+, de nombreux CSP doivent être résolus. La conception du QCSP+ doit être prise soigneusement en compte car Qecode nécessite de connaître l'alternance des quantificateurs pour connecter les pièces du QCSP+. Notons que le nombre de nœuds retourné par Qecode est sous évalué comparé au vrai nombre de nœuds ouverts. En effet, lorsque Qecode fournit à GeCode un CSP dans lequel il reste des variables à instancier et donc des nœuds à ouvrir, ceux-ci ne sont pas comptabilisé dans le résultat affiché. La valeur affichée est donc plus en lien avec ceux des CSP résolus par GeCode que ceux réellement ouverts.

Pour évaluer QuaCode versus l'état de l'art, nous utilisons le *Jeu de Nim*, le *MatrixGame* et le *Puissance Quatre* présentés au paragraphe 3.

---

3. http://quacode.barichard.com
4. http://www.gecode.org
5. http://pn.host.cs.st-andrews.ac.uk/
6. http://www.univ-orleans.fr/lifo/software/qecode/QeCode.html

6.1. Le MatrixGame

Même si le *MatrixGame* est souvent présenté comme un problème QCSP+, il n'y a pas de règle à satisfaire entre chaque tour. Toutes les contraintes dépendent de l'ensemble des variables. Ainsi, ce problème ne réclame aucune des spécificités d'un QCSP+ et peut être modélisé en un QCSP sans perte d'efficacité. Toutes les instances utilisées admettent une stratégie gagnante.

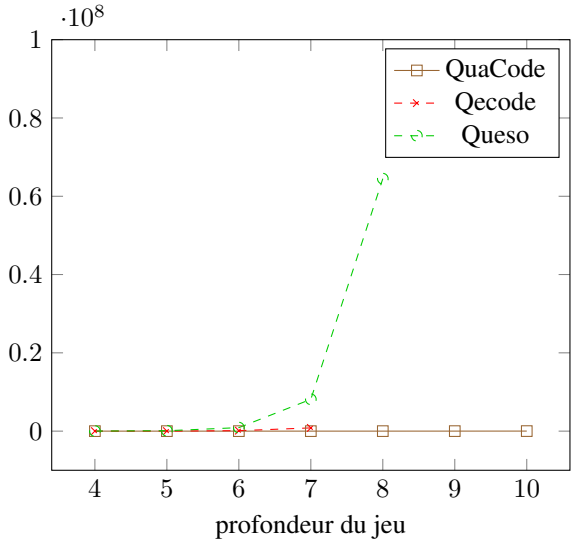


Figure 5. Nombre de propagations pour le MatrixGame

Nous notons que Qecode dépasse le temps maximum accordé après simplement une profondeur du jeu de 7 et Queso de 8. Nous observons une rupture qui apparaît tôt ou tard selon le solveur. Cela est illustré par l'explosion du nombre de propagations de Queso pour une profondeur de 8 (cf. figure 5) et par l'augmentation du nombre de nœuds de Qecode pour une profondeur de 7 (cf. figure 6). Queso et QuaCode étant basés sur des algorithmes de recherche quantifiés similaires, la différence de performance s'explique par la librairie de contraintes sous-jacente. En effet, QuaCode est basé sur GeCode et bénéficie de toutes ses contraintes. Contraintes qui ont été largement éprouvées par la communauté CSP.

6.2. Le Jeu de Nim

Nous abordons maintenant le *Jeu de Nim*. Les figure 7 et 8 reportent respectivement le nombre de propagations et le nombre de nœuds de QuaCode comparés à Queso et Qecode. Nous constatons en particulier que le nombre de propagations et de nœuds de Queso s'accroissent significativement par rapport à ceux de QuaCode et Qecode. De fait, Qecode est conçu pour être efficace sur ce genre de QCSP et ainsi

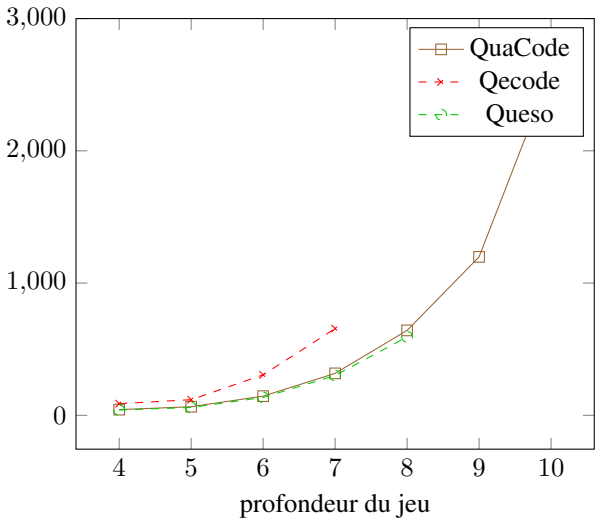


Figure 6. Nombre de nœuds pour le MatrixGame

évite les branches trivialement satisfaites de l'arbre de recherche comme le fait l'outil de $coupure$ dans le cas de QuaCode. Les résultats de Qecode et QuaCode sont similaires. Cela montre que l'on peut obtenir la même efficacité qu'un solveur dédié grâce à l'outil $coupure$.

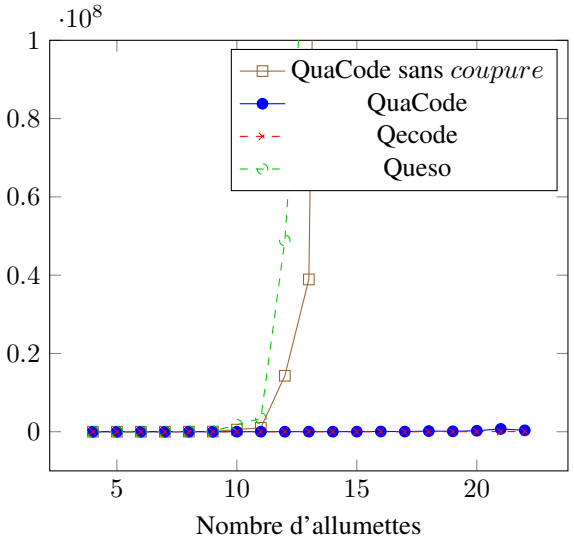


Figure 7. Nombre de propagations pour le jeu de Nim

La figure 9 montre que lorsque le nombre d'allumettes dépasse 12, le temps de calcul de Queso excède alors la demie heure tandis que QuaCode et Qecode nécessitent

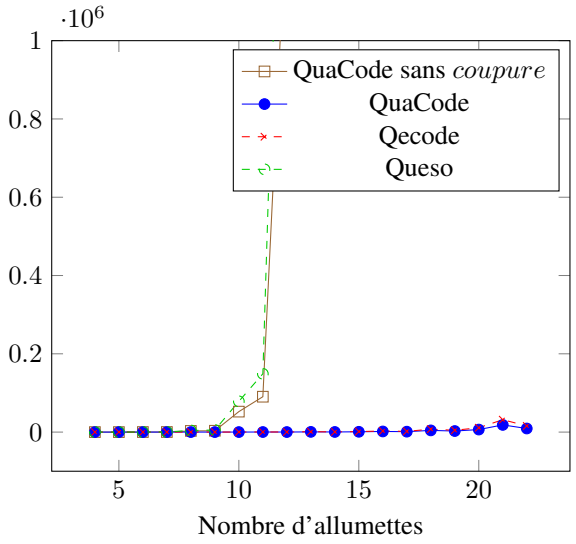


Figure 8. Nombre de nœuds pour le jeu de Nim

moins d'une seconde. QuaCode surperforme Qecode quelques soient les instances. Pour la dernière instance, c'est même deux fois plus rapide. QuaCode et Qecode sont tous les deux basés sur GeCode ainsi les contraintes sont utilisées de la même manière. Mais en Qecode, la résolution des CSP est laissée à GeCode et aucune connaissance ne venant du QCSP ne peut être utilisée pendant cette phase. QuaCode n'apparaît pas comme une *boîte noire* pour la recherche d'un scénario et utilise toutes les connaissances disponibles du problème pour clore au plus vite la recherche. Cela explique le gain d'efficacité de QuaCode sur Qecode.

Ces figures montrent aussi que si l'outil de $coupure$ n'est pas mis en œuvre le « talon d'Achille » apparaît comme pour Queso.

6.3. Le Puissance Quatre

Le dernier problème testé est le *Puissance Quatre* déjà étudié par P. Nightingale dans (Nightingale, 2009). Nous utilisons le modèle de base et comparons, selon différents paramètres, Queso et QuaCode. Malheureusement, ce problème ne dispose pas de formulation pour Qecode ; ainsi nous n'utilisons pas Qecode pour ces tests. Dans (Nightingale, 2007), l'auteur adapte la méthode des *valeurs pures*, issue des *littéraux monotones* pour SAT (Bennaceur, 2004), aux QCSP. Ainsi, Queso peut élaguer des valeurs dans les domaines des variables universelles pour éviter des branches inutiles de l'arbre de recherche. Ce mécanisme est puissant mais il est très couteux et doit être utilisé avec prudence. Le *Puissance Quatre* est un classique des jeux à deux joueurs et bénéficie pleinement du théorème 1 portant sur l'utilisation de la coupure. Notre but ici est de comparer un solveur qui utilise l'optimisation basé sur les valeurs

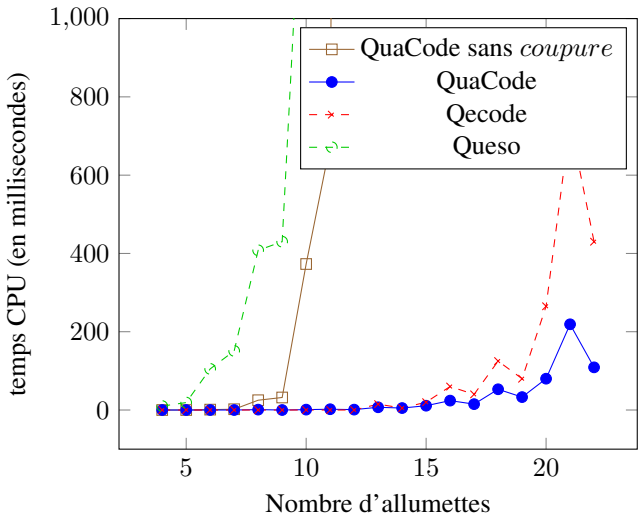


Figure 9. Comparison des temps de calcul pour le jeu de Nim

pures avec un solveur qui exploite le théorème 1 et l'outil de coupure. Notons que la modélisation utilisée du *Puissance Quatre* est favorable à l'utilisation des valeurs pures. En effet, elle ajoute des variables existentielles *ombres* qui facilitent la détection des valeurs pures.

Tableau 1. Comparaison pour le Puissance Quatre *des solveurs QCSP selon différents paramètres (« Opt. » pour « options », "VP" pour « valeurs pures », « QC » pour « QuaCode », « Qo » pour « Queso », « TD » pour « temps dépassé » i.e. plus que 1 800 000 ms)*

Taille du plateau		Propagations, nœuds et temps				
col	ligne	Solveur	Opt.	Propagations	Nœuds	Temps en (ms)
4	4	QC		798 636 092	928 645	142 180
4	4	QC	coupure	6 766 190	7 800	885
4	4	Qo		512 177 815	845 296	55 207
4	4	Qo	VP	20 761 455	25 141	2 813
4	5	QC				TD
4	5	QC	coupure	83 364 319	76 117	11 874
4	5	Qo				TD
4	5	Qo	VP	1 037 360 167	775 784	112 718
5	4	QC				TD
5	4	QC	coupure	940 296 419	979 244	138 768
5	4	Qo				TD
5	4	Qo	VP			TD

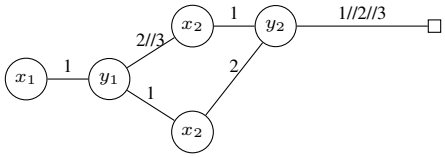
L'ensemble des résultats sont présentés dans le tableau 1. Aucun des solveurs (sans les valeurs pures ou l'outil de coupure) n'est à même de résoudre des instances avec plus de 4 lignes ou colonnes dans le temps imparti. Cela montre que l'élagage fait par les valeurs pures ou l'outil de coupure est indispensable pour résoudre efficacement le problème. De fait, pour chaque instance, le nombre de nœuds explorés et le nombre de propagations pour Queso avec valeurs pures et QuaCode avec l'outil de coupure sont très inférieurs à ceux pour Queso ou QuaCode seuls. Ces résultats se traduisent en une réduction drastique du temps de calcul. La seconde observation est que l'utilisation de l'outil coupure élague plus de nœuds que l'utilisation des valeurs pures. En effet, QuaCode avec l'outil coupure surperforme Queso avec valeurs pures sur toutes les instances. Sur l'ultime instance, l'unique solveur qui la résout dans le temps imparti est QuaCode.

7. Discussion

Dans le cadre des solveurs QCSP, deux points d'efficacité sont à discuter : celui de la représentation de la stratégie et celui des solveurs de type QuaCode dont l'efficacité ne peut pas reposer uniquement sur l'algorithme 1 de recherche.

7.1. Pour une gestion efficace des stratégies

Encore une fois sous les hypothèses classiques de la théorie de la complexité, il y a peu d'espoir de pouvoir encoder d'une manière ou d'une autre toute stratégie, gagnante ou non, sous une forme polynomiale (Coste-Marquis *et al.*, 2006). La représentation classique présentée dans la section 2 est celle d'une arborescente complète qui est évidemment utile pour la définition de la sémantique des QCSP mais inutilisable en pratique (car exponentielle en nombre de nœuds par rapport au nombre de variables universellement quantifiées). De fait, une stratégie peut être optimisée en ne stockant pas un arbre mais un graphe orienté acyclique (ou DAG) qui partage les sous-arbres identiques ; de même les arcs partant d'un nœud étiqueté par un variable universelle et partageant le même sous arbre peuvent aussi être factorisés. Le DAG ci-dessous représente la stratégie gagnante de la figure 2 pour le *Jeu de Nim* :



En fait pour une stratégie calculée par un algorithme de recherche comme celui de l'algorithme 1, le DAG représentant la stratégie restreint aux variables sur lesquelles l'algorithme à brancher associé à l'ensemble des contraintes initiales et à l'algorithme de propagation est suffisant pour décrire toute stratégie. Ainsi le solveur de CSP sous-jacent sert aussi bien à calculer la stratégie *off line* qu'à la parcourir *on line*.

7.2. Pour un solveur QCSP plus efficace

Une des possibilités pour rendre le solveur plus efficace est d'apprendre la structure du problème en stockant des *nogoods*, respectivement des *goods* pour élaguer des parties de l'espace de recherche car redondants en échec, respectivement en succès (Bacchus, Stergiou, 2007). Ces techniques sont nées dans le cadre des CSP (Prosser, 1993) et se sont montrées très performantes dans le cadre des QBF et de la recherche dirigée par les conflits pour SAT (Marques-Silva, K.Sakallah, 1996) et la programmation logique non monotone (Gebser *et al*., 2007). Mais cette option de recherche pour rendre efficace un solveur QCSP demande de modifier le cœur du solveur CSP sous-jacent et se révèle donc difficile à réaliser, car contraire à sa philosophie, dans le cas de QuaCode.

Une autre des possibilités pour rendre les solveurs QCSP plus efficace est de guider leur recherche soit grâce à des heuristiques internes de choix de variable dans le bloc de quantificateurs considéré [7] ou de choix de valeur à explorer (Stynes, Brown, 2009), soit grâce à des heuristiques externes basées sur la résolution en parallèle et coopérative avec des métaheuristiques, type algorithme de Monte Carlo (Kocsis, Szepesvári, 2006). C'est ce dernier choix qui a était fait prioritairement pour permettre à QuaCode d'être un système ouvert. La figure 10 présente l'architecture actuelle de la coopération entre le solveur QCSP complet et l'algorithme stochastique : le solveur QCSP complet basé sur l'algorithme 1 de recherche quantifiée utilisant la bibliothèque CSP GeCode est guidé heuristiquement de manière asynchrone par un algorithme stochastique via un mécanisme intermédiaire de communication que nous nommons « SIBus » (pour *Search Information Bus*).

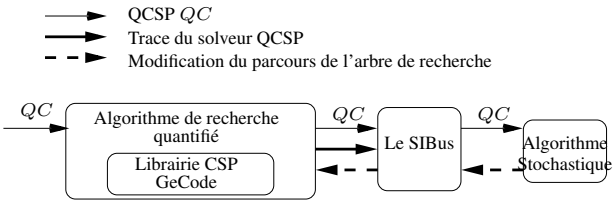


Figure 10. L'architecture actuelle de la coopération entre le solveur complet et l'algorithme stochastique via le SIBus

Si dans le cas des CSP (et de SAT), la vérification que l'instanciation calculée est bien un modèle est polynomiale en temps, il n'en est pas de même pour les QCSP (et les QBF) pour lesquelles le test est co-NP-complet (Kleine Büning, Zhao, 2004). Or l'intérêt des méthodes de type stochastique est de calculer rapidement et en très grand nombre des solutions potentielles. Ainsi WalkQSat (Gent *et al*., 2003), une extension aux QBF de l'algorithme de recherche locale WalkSat (Selman *et al*., 1994),

7. Deux quantificateurs universels ou deux quantificateurs existentiels peuvent être permutés sans mettre en cause la sémantique du QCSP mais il n'en va pas de même en général d'un quantificateur existentiel avec un quantificateur universel. Les heuristiques de choix de variable relèvent donc du choix (dynamique) d'une permutation dans un bloc de quantificateurs identiques.

a montré les limites de l'approche métaheuristique pour le calcul des solutions des QBF et a été abandonnée. Par contre, tester si un scénario est perdant (ou gagnant) est polynomial. Il est alors garanti qu'un tel scénario perdant ne pourra appartenir à aucune stratégie gagnante (tandis qu'un scénario gagnant peut ou peut ne pas appartenir à une stratégie gagnante). L'algorithme stochastique qui sert d'heuristique à l'algorithme complet de recherche quantifié ne calcule donc pas une stratégie au QCSP. Le schéma de fonctionnement général de l'algorithme stochastique en coopération avec l'algorithme complet est donc le suivant : l'algorithme stochastique, par des exécutions polynomiales sur des instances complètes sur le CSP, cherche à identifier des espaces non solutions du CSP sous-jacent. Il réordonne ensuite l'ordre de parcours des valeurs des domaines des variables de manière à retarder l'exploration de ces espaces par le solveur complet. Il guide ainsi le solveur QCSP vers des zones de l'espace de recherche ayant plus de chances de contenir une stratégie gagnante sans modifier la complétude de ce dernier. Le SIBus est libre et disponible avec des exemples de communication sur simple demande.

8. Perspectives et conclusion

Le formalisme QCSP permet de modéliser des problèmes sous-contraintes complexes et dans lesquels certaines variables peuvent être quantifiées universellement. Ce formalisme est bien adapté à la représentation des jeux à deux joueurs à horizon fini. En effet, la richesse du formalisme CSP d'une part, fournit les outils pour modéliser un ensemble d'interactions complexes entre les variables de décisions permettant ainsi de représenter les règles du jeu. La possibilité de quantifier universellement des variables d'autre part, permet de représenter un adversaire à battre. Pour résoudre ces problèmes, nous proposons QuaCode un solveur QCSP, adapté à ce formalisme, construit au-dessus de la bibliothèque CSP GeCode. GeCode fournit sa bibliothèque de contraintes à laquelle ont été ajoutés un algorithme de résolution quantifié et l'outil coupure, permettant de rendre la résolution plus efficace en tirant parti de certaines particularités des QCSP.

Toutefois, beaucoup de jeux comme le *Puissance Quatre*, n'ont pas de stratégie gagnante pour une partie vierge (non commencée). En effet, si tel était le cas, le jeu n'aurait que peu d'intérêt. Malgré tout, des bonnes stratégies existent et parmi celles-ci, il est possible d'en extraire les meilleures. Pour pouvoir modéliser ce problème de manière simple et élégante, il faudrait relâcher le caractère *omniscient* du joueur universel. Nous travaillons actuellement à fournir un cadre légèrement plus souple que les QCSP tout en conservant la simplicité et la richesse du formalisme initial. Nous pensons que cela permettra de modéliser plus naturellement des jeux à deux joueurs réels pour lesquels il n'existe pas de stratégie gagnante initiale.

Bibliographie

Ansotegui C., Gomes C., Selman B. (2005). Achilles' heel of QBF. In *Proceedings of the 20th national conference on artificial intelligence (aaai'05)*, p. 275-281.

Bacchus F., Stergiou K. (2007). Solution directed backjumping for QCSP. In *Proceedings of the 13th international conference on principles and practice of constraint programming (cp'07)*, p. 148-163.

Barichard V., Stéphan I. (2014). The *cut* tool for QCSP. In *Proceedings of the 26th ieee international conference on tools with artificial intelligence (ictai'14)*, p. 883-890.

Barichard V., Stéphan I. (2014). L'outil *coupure* pour les QCSP. In *Actes des dixièmes journées francophones de programmation par contraintes (jfpc'14).*

Benedetti M., Lallouet A., Vautard J. (2007). QCSP made practical by virtue of restricted quantification. In *Proceedings of the 20th international joint conference on artificial intelligence (ijcai'07)*, p. 38-43.

Benedetti M., Lallouet A., Vautard J. (2008). Modeling adversary scheduling with QCSP+. In *Proceedings of the 23th acm symposium on applied computing (sac'08)*, p. 151-155.

Bennaceur H. (2004). A comparison between SAT and CSP techniques. *Constraints*, vol. 9, n° 2, p. 123-138.

Bordeaux L., Cadoli M., Mancini T. (2005). CSP Properties for Quantified Constraints: Definitions and Complexity. In *Proceedings of the 20th national conference on artificial intelligence (aaai'05)*, p. 360-365.

Bordeaux L., Monfroy E. (2002). Beyond NP: Arc-Consistency for Quantified Constraints. In *Proceedings of the 8th international conference on principles and practice of constraint programming (cp'02)*, p. 371-386.

Bordeaux L., Zhang L. (2007). A solver for quantified Boolean and linear constraints. In *Proceedings of the 2007 acm symposium on applied computing (sac '07)*, p. 321-325.

Börner F., Bulatov A., Chen H., Jeavons P., Krokhin A. (2009). The complexity of constraint satisfaction games and QCSP. *Information and Computation*, vol. 207, n° 9, p. 923-944.

Börner F., Bulatov A., Jeavons P., Krokhin A. (2003). Quantified Constraints: Algorithms and Complexity. In *Proceedings of the 17th international workshop on computer science logic (csl'03)*, p. 58-70.

Cadoli M., Giovanardi A., Schaerf M. (1998). An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 15th national conference on artificial intelligence (aaai'98)*, p. 262-267.

Chen H., Madelaine F., Martin B. (2015). Quantified Constraints and Containment Problems. *Logical Methods in Computer Science*, vol. 11, n° 3.

Coste-Marquis S., Fargier H., Lang J., Le Berre D., Marquis P. (2006). Representing Policies for Quantified Boolean Formulae. In *Proceedings of the 10th international conference on principles of knowledge representation and reasoning (kr'06)*, p. 286-296.

Gebser M., Kaufmann B., Neumann A., Schaub T. (2007). Conflict-Driven Answer Set Solving. In *Proceedings of the 20th international joint conference on artificial intelligence (ijcai'07)*, p. 386-392.

Gent I., Hoos H., Rowley A., Smyth K. (2003). Unsing Stochastic Local Search to Solve Quantified Boolean Formulae. In *Proceedings of the 9th international conference on principles and practice of constraint programming (cp'03).*

Gent I., Nightingale P., Rowley A. (2004). Encoding Quantified CSPs as Quantified Boolean Formulae. In *Proceedings of the 16th european conference on artificial intelligence (ecai'04)*, p. 176-180.

Gent I., Nightingale P., Rowley A., Stergiou K. (2008). Solving quantified constraint satisfaction problems. *Artificial Intelligence*, vol. 172, n° 6-7, p. 738-771.

Gent I., Nightingale P., Stergiou K. (2005). QCSP-solve: A solver for quantified constraint satisfaction problems. In *Proceedings of 9th international joint conference on artificial intelligence (ijcai'05)*, p. 138-143.

Goultiaeva A., Bacchus F. (2010). Exploiting Circuit Representations in QBF Solving. In *Proceedings of the 13th international conference on theory and applications of satisfiability testing (sat'10)*, p. 333-339.

Kleine Büning H., Zhao X. (2004). On Models for Quantified Boolean Formulas. In *Logic versus approximation, in lecture notes in computer science 3075.*

Kocsis L., Szepesvári C. (2006). Bandit based monte-carlo planning. In *Proceedings of the 17th european conference on machine learning (ecml'06)*, vol. 4212 of LNCS, p. 282-293.

Mamoulis N., Stergiou K. (2004). Algorithms for Quantified Constraint Satisfaction Problems. In *Proceedings of the 10th international conference on principles and practice of constraint programming (cp'04)*, p. 752-756.

Marques-Silva J., K.Sakallah. (1996). GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of ieee/acm international confenrence on computer-aided design (iccad'96)*, p. 220–227.

Nightingale P. (2007). *Consistency and the Quantified Constraint Satisfaction Problem, PhD thesis, University of St Andrews.*

Nightingale P. (2009). Non-binary quantified CSP: algorithms and modelling. *Constraints*, vol. 14, n° 4, p. 539-581.

Papadimitriou C. (1994). *Computational complexity*. Addison-Wesley.

Pralet C., Verfaillie G. (2011). Beyond QCSP for Solving Control Problems. In *Proceedings of the 17th international conference on principles and practice of constraint programming (cp'11)*, p. 744-758.

Prosser P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, vol. 9, p. 268-299.

Sabharwal A., Ansótegui C., Gomes C., Hart J., Selman B. (2006). QBF Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. In *Proceedings of the 9th international conference on theory and applications of satisfiability testing (sat'06)*, p. 382-395.

Selman B., Kautz H., Cohen B. (1994). Noise strategies for improving local search. In *Proceedings of the 12th national conference on artificial intelligence (aaai'94)*, p. 337-343.

Stockmeyer L., Meyer A. (1973). Word problems requiring exponential time. In *Proceedings of the 5th annual acm symposium on theory of computing (stoc '73)*, p. 1-9.

Stynes D., Brown K. (2009). Value ordering for quantified CSPs. *Constraints*, vol. 14, n° 1, p. 16-37.

Tsang E. (1993). Foundations of constraint satisfaction. *Academic Press, London.*

Verger G., Bessiere C. (2006). BlockSolve: a Bottom-Up Approach for Solving Quantified CSPs. In *Proceedings of the 12th international conference on principles and practice of constraint programming (cp'06)*, p. 635-649.

Verger G., Bessiere C. (2008). Guiding Search in QCSP$^+$ with Back-Propagation. In *Proceedings of the 14th international conference on principles and practice of constraint programming (cp'08)*, p. 175-189.