QCSP guidé par Monte Carlo

Jean-Mathieu Chantrein et Vincent Barichard et Igor Stéphan

LERIA, Université d'Angers, 2, bd Lavoisier, 49045 Angers Cedex 01

{jean-mathieu.chantrein|vincent.barichard|igor.stephan}@info.univ-angers.fr

Résumé

Nous présentons dans cet article une coopération entre un solveur de contraintes quantifiées et un algorithme de type Monte Carlo. Ce dernier sert d'heuristique sur l'ordre du choix des valeurs des domaines des variables lors de la résolution. Il échantillonne l'espace de recherche pour se focaliser sur les zones qui ne satisfont pas le CSP sous-jacent. Il réordonne ensuite le choix des valeurs des domaines fait par le solveur, donnant la priorité par dualité aux zones de l'espace de recherche ayant le plus de chance de contenir une solution.

Abstract

We present in this article a cooperation between a solver for problems with quantified constraints and a Monte Carlo algorithm. The latter acts as a heuristic on the order of the values of domains variables chosen during the search. It samples the search space in order to focus on the areas which do not satisfy the underlying CSP. It reorders the choice of values made by the solver, and gives a higher priority by duality to the search space areas which seem to have the best chance of containing solutions.

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 [2]. L'étude des QCSP est récente mais il existe un réel intérêt pour mettre au point des techniques efficaces afin de les résoudre (cf. [10] pour un panorama). 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. Prenons le problème du boulanger qui 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 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 $w_1, w_2, w_3, w_4, f \in \{1,\ldots,40\}$, $c_1, c_2, c_3, c_4 \in \{-1, 0, 1\}$, alors $\exists w_1 \exists w_2 \exists w_3 \exists w_4 \forall f \exists c_1 \exists c_2 \exists c_3 \exists c_4 (\sum_{i=1}^{4} w_i.c_i = f)$. Les variables quantifiées existentiellement w_1, w_2, w_3, w_4 représentent les poids que le boulanger doit acheter ; la variable quantifiée universellement f représente le poids de farine à peser ; les variables quantifiées existentiellement c_1, c_2, c_3, c_4 représentent la position sur la balance de chaque poids acheté i. Si le poids est mis sur le même plateau que la farine pesée ($c_i = -1$), sur l'autre plateau ($c_i = 1$) ou omis ($c = 0$). Les valeurs des c_1, c_2, c_3, c_4 sont différentes selon la valeur de f. Le problème du boulanger admet une unique stratégie gagnante $\{1, 3, 9, 27\}$ (modulo les permutations des variables w_i).

Le domaine des QCSP est à l'intersection de deux domaines : le domaine des CSP [11] et celui des QBF [3]. 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 pour bénéficier du catalogue des contraintes. Cette approche est la nôtre : nous implantons QuaCode, un solveur QCSP, au-dessus de Gecode, une bibliothèque de classes pour la gestion des contraintes dans un CSP, sans changer le cœur de la bibliothèque.

Cette extension, si elle accroît les possibilités de modélisation, accroît aussi la complexité de décision de NP-complet à PSPACE-complet. Dans le problème de satisfiabilité des formules propositionnelles (ou SAT),

les heuristiques stochastiques ont montré leur efficacité [9] mais elles se sont montrées inefficaces dans PSPACE [5] pour la raison suivante : le test d'un candidat solution pour NP est polynomial alors que pour PSPACE il est co-NP-complet. Mais une autre voie est possible dans l'emploi d'une heuristique stochastique par la coopération avec un solveur complet. Cette approche a déjà été proposée dans le domaine des CSP avec succès [7] par une coopération entre un solveur CSP complet et une heuristique de type Monte Carlo [1] et pour les QBF avec un moindre succès [4] par une coopération entre un solveur QBF complet et une colonie de fourmis. Nous proposons dans cet article de mettre en coopération notre solveur QCSP complet QuaCode avec une heuristique stochastique de type Monte Carlo pour guider QuaCode. Pour éviter l'écueil du test co-NP-complet sur l'acceptation d'un candidat comme solution, nous proposons de focaliser l'exploration sur les parties non solutions de l'espace de recherche pour le CSP sous-jacent.

L'article est articulé ainsi : la section 2 présente les notions nécessaires à la compréhension de notre proposition et en particulier l'algorithme de recherche quantifié complet sur lequel elle est basée ; la section 3 présente l'architecture de la coopération entre un solveur QCSP complet et une heuristique stochastique de type Monte Carlo ainsi que des résultats expérimentaux montrant la pertinence de cette approche ; enfin la section 4 dresse une conclusion avec des perspectives nouvelles de recherche.

2 Préliminaires

Le symbole $\exists$ représente le quantificateur existentiel et le symbole $\forall$ représente le quantificateur universel ($\overline{\exists} = \forall$ et $\overline{\forall} = \exists$). 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 n-uplet $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$: $\mathbf{V}$ est un ensemble de n variables, ordre est une bijection de $\mathbf{V}$ dans $[1..n]$, quant est une fonction de $\mathbf{V}$ dans $\{\exists, \forall\}$ ($\text{quant}(x)$ dénote le quantificateur associé à la variable x), $\mathbf{D}$ est une fonction de $\mathbf{V}$ dans l'ensemble des domaines $\{D(x_1), \ldots, D(x_n)\}$ telle que, pour toute variable $x_i \in \mathbf{V}$, $D(x_i)$ en dénote son domaine, i.e. l'ensemble fini de toutes les valeurs possibles, $\mathbf{C}$ est un ensemble de contraintes. Si $x_{j_1}, \ldots, x_{j_m}$ 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 $D(x_{j_1}) \times \cdots \times D(x_{j_m})$. Dans ce qui suit, pour chaque $i \in [1..n]$, $q_{x_i} = \text{quant}(x_i)$ et $D_{x_i} = D(x_i) = D_i$.

Un QCSP $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$ est généralement représenté par sa formule en logique du premier ordre $q_{x_1} x_1 \ldots q_{x_n} x_n \bigwedge_{c_j \in \mathbf{C}} c_j$ avec $x_1 \in D_{x_1}$, …, $x_n \in D_{x_n}$, $\text{ordre}(x_i) = i$, pour chaque $i \in [1..n]$. Avec cette représentation $q_{x_1} x_1 \ldots q_{x_n} x_n$ est appelé « lieur »(un lieur vide étant noté ε). Par exemple, le QCSP $(\{x, y, z, t\}, \text{ordre}, \text{quant}, \{\{0, 1, 2\}\}, \mathbf{C})$ avec

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

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

L'ensemble $\mathcal{T}(Q)$ avec $x_1 \in D_{x_1}$, …, $x_n \in D_{x_n}$, $Q = q_{x_1} x_1 \ldots q_{x_n} x_n$ est l'ensemble des arbres tels que

- chaque nœud feuille est étiqueté avec le symbole $\square$ et a la profondeur n,
- chaque nœud interne à la profondeur i, $0 \leq i < n$, est étiqueté avec la variable x_{i+1},
- chaque arc reliant un nœud de profondeur i à l'un de ses fils est étiqueté par un élément de $D_{x_{i+1}}$,
- toutes les étiquettes des arcs reliant un nœud à l'un de ses fils sont différentes.

Soit $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$ un QCSP tel que $\mathbf{V} = \{x_1, \ldots, x_n\}$, avec $x_1 \in D_{x_1}$, …, $x_n \in D_{x_n}$. Une stratégie est un arbre de $\mathcal{T}(q_{x_1} x_1 \ldots q_{x_n} x_n)$ tel que

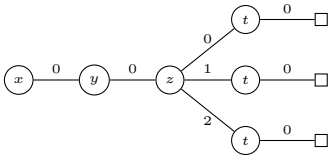
- chaque nœud étiqueté par une variable existentiellement quantifiée admet un unique fils et
- chaque nœud étiqueté par une variable universellement quantifiée dont le domaine est de taille k admet k nœuds fils.

Un scénario est une séquence d'étiquettes $v_1, \ldots, v_n$ sur un chemin $(x_1, v_1), \ldots, (x_n, v_n)$, $v_i \in D_{x_i}$ pour tout i, $1 \leq i \leq n$, d'un arbre $\mathcal{T}(q_{x_1} x_1 \ldots q_{x_n} x_n)$. Un scénario $v_1, \ldots, v_n$ pour un QCSP $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$ tel que $\mathbf{V} = \{x_1, \ldots, x_n\}$ est un scénario gagnant si $(\bigwedge_{1 \leq i \leq n} x_i = v_i) \wedge (\bigwedge_{c_j \in \mathbf{C}} c_j)$ est vrai ; un tel scénario correspond à l'instanciation complète $x_1 = v_1$, …, $x_n = v_n$; c'est un scénario gagnant si l'instanciation satisfait toutes les contraintes. Une stratégie est une stratégie gagnante si tous les scénarios sont des scénarios gagnants. S'il n'y a pas de quantificateur, la stratégie $\square$ est toujours une stratégie gagnante.

Par exemple, la stratégie suivante est une stratégie gagnante pour le QCSP $\exists x \exists y \forall z \exists t((x = (y * z) + t) \wedge$

1. Une variante intéressante du problème de résolution des QCSP est de ne pas rechercher une solution mais simplement une approximation de celle-ci dans le cadre d'un calcul en temps réel à la volée. Cette approximation peut être réalisée grâce à une heuristique de type Monte Carlo [8].

$(t \leq x)), x, y, z, t \in \{0, 1, 2\}$ 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 gagnant, puisque $0 = (0 * 2) + 0$.

Nous pouvons donner une sémantique plus intuitive pour les QCSP : un QCSP $\forall x QC$ avec $x \in \mathbf{V}$ admet une stratégie gagnante si et seulement si, pour tout $v \in D_x$, $Q(C \wedge (x = v))$ admet une stratégie gagnante et un QCSP $\exists x QC$ avec $x \in \mathbf{V}$ admet une stratégie gagnante si et seulement si, pour au moins un $v \in D_x$, $Q(C \wedge (x = v))$ admet une stratégie gagnante.

Il n'y a, à notre connaissance que trois solveurs QCSP, excepté le nôtre : BlockSolve[12], Queso[6] et Qecode [1]. BlockSolve est basé sur un algorithme de Fourier-Motzkin par élimination de quantificateurs. Qecode est aussi un solveur basé sur un algorithme de recherche quantifié mais est dédié à une restriction des QCSP (QCSP+), qui nécessite une forme restreinte de la quantification, et qui est utilisé principalement pour spécifier des jeux finis à deux joueurs. Queso est basé sur un algorithme de recherche quantifié et son approche est très proche de la nôtre. Qecode est toujours maintenu tandis que les deux autres ne le sont plus.

L'algorithme 1 présente la structure d'un algorithme de recherche quantifié pour un QCSP basé sur cette sémantique récursive intuitive. Cet algorithme est basé sur une boucle perpétuelle. Au départ, la pile de retour arrière des points de choix pour les variables $pileRA$ est vide. La fonction $atteintPointFixe$ calcule le point fixe de la propagation de l'ensemble des contraintes et retourne trois codes possibles (différents) : *échec* si une contrainte a été violée, *succès* si l'ensemble des contraintes est vrai et *branche* sinon. Dans le cas du code *échec*, les derniers points de choix sur des variables universellement quantifiées sont sautés puisqu'ils ne peuvent plus mener à un succès, le dernier point de choix sur une variable existentiellement quantifiée x est dépilé de la pile de retour arrière et une nouvelle contrainte $(x = v)$ est ajoutée à l'ensemble courant des contraintes pour poursuivre l'exploration sur une nouvelle valeur possible v. Si un tel point de choix sur une variable quantifiée existentiellement n'existe pas (cela signifie qu'il n'y a que des points de choix sur des variables quantifiées universellement ou aucun point de choix) alors il n'existe pas de stratégie gagnante et l'algorithme retourne faux. Dans le cas du code *succès*, les derniers points de choix sur des variables existentiellement quantifiées sont sautés puisqu'un seul succès est suffisant selon la sémantique du quantificateur existentiel, le dernier point de choix sur une variable x quantifiée universellement est dépilé de la pile de retour arrière et une nouvelle contrainte $(x = v)$ est ajoutée à l'ensemble courant des contraintes pour obtenir des stratégies gagnantes pour les autres valeurs possibles de x en poursuivant par v. Si un tel point de choix sur une variable quantifiée universellement n'existe pas (cela signifie qu'il n'y a que des points de choix sur des variables quantifiées existentiellement ou aucun point de choix) alors le QCSP a une stratégie gagnante et l'algorithme retourne vrai. Sinon l'appel à $atteintPointFixe(C)$ a retourné le code *branche* : l'algorithme poursuit sa recherche en profondeur en parcourant le lieur Q et sélectionne la variable suivante non-instanciée x et empile tous les choix de valeur pour cette variable sur la pile de retour arrière. Le premier point de choix (x, v) est alors dépilé de la pile de retour arrière et la nouvelle contrainte $(x = k)$ est ajoutée à l'ensemble des contraintes pour explorer cette première possibilité pour la variable x. L'instruc-

Algorithme 1 Un algorithme de recherche quantifié pour solveur de QCSP

Entrée: Un QCSP QC
Entrée: Une séquence de domaines $< D_1, \ldots, D_n >$
Entrée: Une liste d'échanges $SIBus.liste$
Sortie: **vrai** si le QCSP QC admet au moins une stratégie gagnante et **faux** sinon
  $pileRA := \emptyset$
  tant que vrai faire
   selon $atteintPointFixe(C)$ **faire**
    cas *échec*
     retour sur le dernier choix existentiel (x, v) de $pileRA$
     si aucune **alors retourner** faux
     sinon ajouter à C la contrainte $(x = v)$
    cas *succès*
     retour sur le dernier choix universel (x, v) de $pileRA$
     si aucune **alors retourner** vrai
     sinon ajouter à C la contrainte $(x = v)$
    cas *branche*
     sélectionner la variable non instanciée suivante x $ordonne(x, SIBus.liste, D(x))$
     pour tout $v \in D(x)$ empiler (x, v) dans $pileRA$
     sélectionner le premier choix (x, v) de $pileRA$
     ajouter à C la contrainte $(x = v)$
   fin selon
  fin tant que

tion $ordonne(x, SIBus, D(x))$ sera expliquée dans la suite de l'article.

3 QCSP guidé par Monte Carlo

La figure 1 présente l'architecture actuelle de la coopération entre le solveur QCSP complet et l'algorithme de Monte Carlo : 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 de type Monte Carlo via un mécanisme intermédiaire de communication que nous nommons « SIBus » (pour *Search Information Bus*).

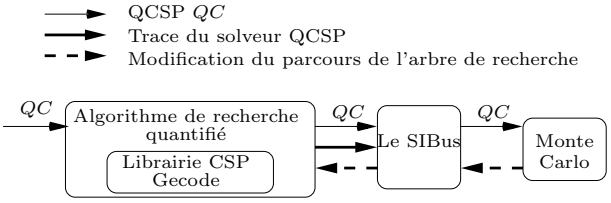


FIGURE 1 – L'architecture actuelle de la coopération entre le solveur complet et l'algorithme de type Monte Carlo via le SIBus.

L'algorithme stochastique de type Monte Carlo qui sert d'heuristique à l'algorithme complet de recherche quantifié ne calcule pas une stratégie au QCSP car le test qui vérifie qu'une stratégie est une stratégie gagnante est co-NP-complet. Or l'intérêt des méthodes de type Monte Carlo est de calculer rapidement et en très grand nombre des solutions potentielles. Ainsi WalkQSat [5], une extension aux QBF de l'algorithme de recherche locale WalkSat [9], 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.) Le schéma de fonctionnement général de l'algorithme de type Monte Carlo 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 manières 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 décrit au paragraphe 3.1 ; l'algorithme de type Monte Carlo est décrit au paragraphe 3.2 ; la prise en compte des résultats de l'heuristique par le solveur complet est décrit au paragraphe 3.3 ; deux améliorations immédiates possibles à apporter sont discutées au paragraphe 3.5.

3.1 Le SIBus

Le SIBus est le centre de l'architecture que nous proposons pour développer des outils dans le domaine des QCSP : il est un intermédiaire qui permet la communication asynchrone entre différents processus aussi bien en entrée qu'en sortie. Les deux fonctions actuelles sont le stockage de la trace du solveur complet QCSP (i.e. l'ensemble des points de choix ouverts) ainsi qu'une liste d'échanges $SIBus.liste$ constituée de triplets (variable, valeur, valeur). Un tel triplet spécifie pour une variable l'échange à réaliser dans son domaine entre le rang de la première et de la seconde valeur. (Un domaine est alors considéré comme une séquence de valeurs et non plus un simple ensemble.) Cette liste est en entrée et en sortie de l'algorithme de type Monte Carlo ainsi que de l'algorithme complet. Dans la conclusion nous décrivons l'architecture complète de notre proposition ainsi que les autres fonctionnalités connectables au SIBus.

3.2 Une heuristique de type Monte Carlo

L'algorithme 2 prend en entrée un ensemble de paramètres qui décrivent un CSP (i.e. un nombre de variables n, une séquence de n variables $x_1, \ldots, x_n$, une séquence de n domaines et un ensemble de contraintes C), une liste d'échanges $SIBus.liste$ et de deux paramètres de contrôle pour l'heuristique : la température T et la fréquence f. Il manipule également une fonction cfl qui associe à chacune des valeurs des domaines des variables le nombre de conflits dans lesquels elle apparaît. L'algorithme commence par une phase d'initialisation qui affecte à chaque variable x_i une valeur v_i du domaine D_i de manière aléatoire et qui met à 0 le nombre de conflits enregistré pour toutes les valeurs des domaines de chaque variable. Elle calcule ensuite $nbCfls$, le nombre de conflits engendrés par l'instance initiale. Le compteur qui compte le nombre d'itérations de l'algorithme est mis à zéro. L'algorithme entre alors dans une boucle perpétuelle qui va modifier l'instance courante et va apprendre par renforcement les zones de l'espace de recherche où la chance de trouver une stratégie gagnante pour le QCSP est la plus faible. Après l'incrémentation du compteur qui compte le nombre de tests, un indice k correspondant à l'indice de la variable actuellement la moins impliquée dans des conflits est choisi. La valeur courante de la

Algorithme 2 Heuristique de type Monte Carlo.

Entrée: Un nombre de variables n
Entrée: Une séquence de variables $s =< x_1, \ldots, x_n >$
Entrée: Une séquence de domaines $< D_1, \ldots, D_n >$
Entrée: Un ensemble de contraintes C
Entrée: Une température T
Entrée: Une fréquence f
Entrée/Sortie: Une liste d'échanges $SIBus.liste$

```
pour 1 ≤ i ≤ n faire
  v_i := rand(D_i)
  pour 1 ≤ k ≤ |D_i| faire
    cfl(x_i, v_k) := 0
  fin pour
fin pour
nbCfls := evConflits(cfl, s, < v_1, ..., v_n >, C)
compteur := 0
tant que true faire
  compteur := compteur + 1
  k := choix(cfl, s, < v_1, ..., v_n >)
  v_sauv := v_k
  v_k := rand(D_k)
  nNbCfls := evConflits(cfl, s, < v_1, ..., v_n >, C)
  si nNbCfls > nbCfls alors
    nbCfls := nNbCfls
  sinon
    δ := nbCfls − nNbCfls
    si metropolis(δ, T) alors
      nbCfls := nNbCfls
    sinon
      v_k := v_sauv
    fin si
  fin si
  si compteur = f alors
    pour 1 ≤ i ≤ n faire
      D'_i := reordonne(cfl, x_i, D_i)
      envoyer(x_i, D_i, D'_i, SIBus.liste)
      D_i := D'_i
    fin pour
    compteur := 0
  fin si
fin tant que
```

variable x_k est sauvegardée et une nouvelle valeur est tirée au hasard dans son domaine. La nouvelle instance est évaluée, et en même temps, la fonction cfl est mise à jour pour prendre en compte les nouveaux conflits. Le nombre de conflits de l'instance testée est renvoyé puis stocké dans la variable $nNbCfls$. Si le nombre de conflits de la nouvelle instance est supérieure à celui de l'instance courante alors la nouvelle instance remplace la courante, sinon la règle d'acceptation de Métropolis est appliquée. Cette dernière est celle du recuit simulé qui accepte une dégradation de l'évaluation selon une probabilité liée à la température T fournie en paramètre à l'algorithme. Si la dégradation est acceptée alors la nouvelle instance remplace la courante sinon la valeur de la variable x_k est restaurée. Si le compteur atteint la fréquence f, chaque domaine est réordonné en fonction du nombre de conflits retourné par cfl pour chacune de ses valeurs. Les valeurs présentant le moins de conflits sont mises en avant car elles ont plus de chance de participer à une stratégie gagnante. Ensuite, la liste des permutations des valeurs des domaines permettant d'obtenir le nouvel ordre à partir de l'actuel est envoyée sur le SIBus. Le nouvel ordre remplace l'ordre courant pour la suite de l'algorithme.

3.3 Prise en compte des choix heuristiques par le solveur complet

La fonction *ordonne* présente dans l'algorithme 1 applique l'ensemble des échanges de valeurs du domaine de la variable x considérée à partir de la liste d'échanges $SIBus.liste$ calculée par l'algorithme stochastique et envoyée sur le SIBus. L'ordre final du domaine de la variable x est le résultat de toutes les permutations envoyées par l'algorithme stochastique sur le SIBus depuis la dernière ouverture de la variable x par l'algorithme complet [2]. Cette liste d'échanges contient d'autres triplets pour d'autres variables mais ces échanges ne sont pas appliqués. L'appel à cette fonction s'insère dans l'algorithme 1 de recherche quantifiée lors de la sélection d'une nouvelle variable non instanciée du lieur. Le domaine ainsi partiellement réordonné favorise par dualité les zones de l'espace de recherche où le CSP est le plus susceptible d'être vrai et donc potentiellement les zones de l'espace de recherche où sont le plus susceptible d'être présentes les stratégies gagnantes.

3.4 Résultats expérimentaux

Pour étudier et valider expérimentalement notre approche, nous avons utilisé le solveur QCSP QuaCode, implémenté une méthode de type Monte Carlo ainsi

2. Une variable est réouverte lors d'un retour en arrière.

que le bus logiciel SIBus permettant aux différents algorithmes de communiquer entre eux en envoyant et écoutant les informations véhiculées dessus. Le problème du boulanger déjà présenté dans cet article va être utilisé comme problème test, nous l'avons pour cela décliné en deux instances distinctes. En effet, le problème du boulanger admet une unique stratégie gagnante $\{1, 3, 9, 27\}$ (modulo les permutations des variables w_i). La première stratégie gagnante est trouvée rapidement, nous avons donc construit une instance plus difficile à résoudre en éliminant la première stratégie gagnante. L'instance 1 correspond à l'instance initiale. L'instance 2 empêche le solveur de trouver une stratégie gagnante où $w_1 = 1$, ainsi, la première stratégie gagnante sera $\{3, 1, 9, 27\}$.

Les résultats présentés au Tableau 1 synthétisent la moyenne, la valeur minimale et maximale d'un ensemble d'informations récoltées sur 1000 exécutions indépendantes. Toutes les expérimentations ont été faites sur un ordinateur équipé d'un processeur Intel Core^{tm} i7-2620M CPU à 2.70GHz (deux cœurs) avec 4 Go Ram tournant sous Linux 32-bits. La température a été fixée à 4.5 et la fréquence à 1000.

Nous comparons QuaCode seul avec QuaCode utilisant les indications d'une méthode de type Monte Carlo (QuaCode+MC). L'algorithme de type Monte Carlo guide QuaCode en envoyant sur le SIBus des suggestions d'échanges de l'ordre de parcours de deux valeurs du domaine d'une variable. QuaCode a la liberté d'accepter ces suggestions et de les appliquer (il envoie alors sur le SIBus l'information comme quoi le changement est accepté) ou de les ignorer (par exemple lorsque le domaine de la variable en question a déjà été ouvert). Pour chaque approche, le temps de calcul ainsi que le nombre de nœuds explorés par QuaCode sont affichés. Lorsque la méthode de type Monte Carlo est exécutée, nous avons également reporté le nombre de demandes envoyées sur le SIBus et le nombre de demandes validées par QuaCode (cf. Tableau 1a). De plus, au Tableau 1b, nous avons reporté le nombre d'itérations de la méthode de type Monte Carlo.

Notons que seul le temps d'exécution de QuaCode est pertinent. En effet, quelque soit l'approche, c'est QuaCode qui détermine s'il existe une stratégie gagnante ou non, l'algorithme de type Monte Carlo est exécuté en parallèle voire sur une autre machine et s'arrête aussitôt que QuaCode a fini la résolution du problème. QuaCode seul étant un algorithme complètement déterministe, chaque exécution se déroule de la même manière. Ainsi, le nombre de nœuds explorés est toujours le même et le temps de calcul varie peu d'une exécution à l'autre.

En comparant les temps de calcul sur l'instance 1, nous observons que celui de QuaCode+MC est toujours inférieur au temps de calcul minimal requis pour QuaCode seul. Cela se confirme sur l'instance 2. Dans le meilleur des cas, QuaCode+MC est 3 fois plus rapide que QuaCode seul sur l'instance 1 et presque 10 fois plus rapide sur l'instance 2. En comparant maintenant le nombre de nœuds explorés, il apparaît que QuaCode+MC peut parfois explorer plus de nœuds que QuaCode seul. Toutefois, cela ne détériore que légèrement les temps de calcul. Notons que les nombres de nœuds reportés dans les tableaux correspondent aux nombres de nœuds indiqués par la bibliothèque CSP sous-jacente Gecode. Ce nombre de nœuds est supérieur au nombre de nœuds réellement explorés ceci est dû aux mécanismes internes de la bibliothèque quant à sa manière de gérer les retours arrières. Ainsi, même si quelques nœuds supplémentaires sont ouverts, ils peuvent être peu coûteux en temps de calcul. C'est pourquoi, malgré un nombre de nœuds légèrement supérieur sur certaines exécutions, les statistiques des temps de calcul sont peu impactées.

En comparant maintenant le nombre de demandes envoyées sur le bus avec le nombre de demandes acceptées, il apparaît qu'un grand nombre de demandes ne sont pas retenues par QuaCode. En analysant les traces des deux algorithmes, nous observons que la plupart des demandes non retenues correspondent à des demandes d'échanges de valeurs de domaines de variables déjà ouvertes par QuaCode. Changer l'ordre de parcours de ces domaines pourrait dans certains cas compromettre la complétude de l'algorithme, toutes ces demandes sont donc systématiquement ignorées. Afin de mieux cibler ces demandes d'échanges envoyées sur le SIBus, l'algorithme de type Monte Carlo devrait donc tenir compte de la trace de QuaCode afin de connaître les domaines ouverts et se focaliser sur les zones non encore explorées.

Sur les instances testées ici, QuaCode+MC surpasse QuaCode seul. L'algorithme de type Monte Carlo agit comme une heuristique dynamique de guidage pour l'algorithme complet. Cet algorithme a un coût, c'est pourquoi il est important de pouvoir l'exécuter en parallèle sur un autre cœur du processeur ou sur une autre machine. Les processeurs multi-cœur étant aujourd'hui très largement répandus, une approche pouvant s'exécuter sur deux cœurs au lieu d'un rentabilise au mieux les atouts de la machine utilisée.

3.5 Discussion

Nous discutons deux améliorations possibles qui n'ont pas encore été testées : l'une propose de tenir compte de la trace du solveur complet pour empêcher l'algorithme de type Monte Carlo de tester des zones obsolètes de l'espace de recherche tandis que l'autre propose d'aller plus loin et d'intensifier la recherche

		Temps (ms)			Nœuds explorés			Échanges demandés			Échanges réalisés		
		Moy.	Min	Max	Moy.	Min	Max	Moy.	Min	Max	Moy.	Min	Max
Instance 1	QuaCode	3971	3822	4137	19130	19130	19130	/	/	/	/	/	/
	QuaCode+MC	1457	1021	2593	13453	7661	27988	1127	565	1215	581	256	1215
Instance 2	QuaCode	152341	146377	156286	1171084	1171084	1171084	/	/	/	/	/	/
	QuaCode+MC	70151	1498	132221	901853	14662	1697205	25239	1020	57264	19571	518	49015

(a) Résultats pour l'algorithme QuaCode

		Nombre d'itérations		
		Moy.	Min	Max
Instance 1	Monte Carlo	29293	12000	64000
Instance 2	Monte Carlo	1935350	32000	3692237

(b) Résultats pour l'algorithme de type Monte Carlo

TABLE 1 – Synthèse des résultats sur le problème du boulanger.

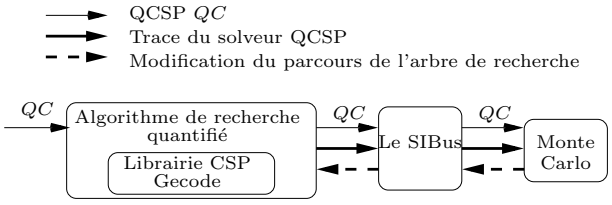


FIGURE 2 – L'architecture de la coopération entre le solveur complet et l'algorithme de Monte Carlo via le « SIBus » avec prise en compte de la trace du solveur

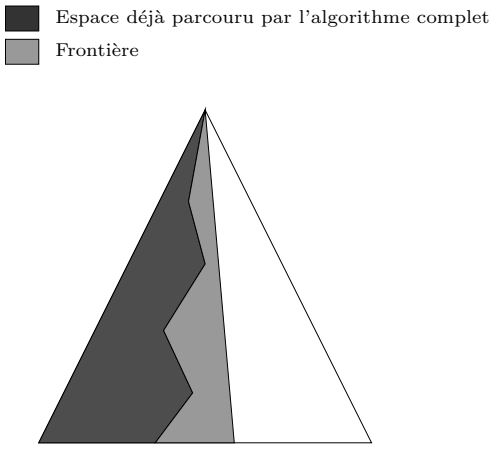


FIGURE 3 – Espace déjà parcouru par l'algorithme complet et frontière

stochastique sur la « frontière » courante de la recherche complète.

L'architecture présentée dans cet article ne tient pas compte de la trace du solveur. Ainsi, le solveur progresse dans l'espace de recherche et décide pour certaines parties de celui-ci de l'impossibilité de l'existence d'une stratégie gagnante mais l'algorithme de type Monte Carlo n'en est pas informé et, lui, continue de parcourir cet espace initial (même si c'est dans un autre ordre).

La figure 2 présente une première amélioration qui consiste à ne pas faire parcourir cet espace : la différence avec l'architecture actuelle présentée en figure 1 est la flèche qui part du SIBus et qui s'oriente vers l'algorithme de type Monte Carlo. L'espace réel de recherche déjà parcouru par l'algorithme complet (en gris foncé et moyen sur la figure 3) n'est que sous approximé par l'algorithme de type Monte Carlo (en gris foncé sur la même figure) car la trace ne fait état que des valeurs sur lesquelles l'algorithme complet a branché et non des propagations et, de plus, ce dernier a poursuivi sa recherche pendant une itération complète de l'algorithme de type Monte Carlo. L'espace de recherche de ce dernier est donc une sur approximation de l'espace réellement nécessaire à explorer (en gris moyen et blanc toujours sur la même figure).

Une seconde amélioration est l'intensification de la recherche à la frontière. Il est probablement plus intéressant de guider au plus proche de son parcours l'algorithme complet que d'aller obtenir de la connais-

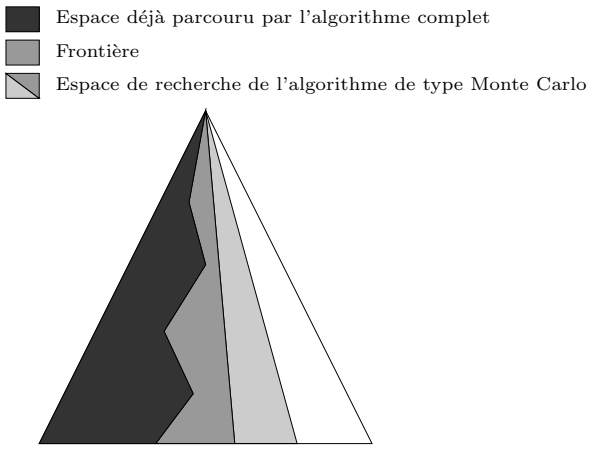


FIGURE 4 – Espace de recherche de l'algorithme de type Monte Carlo

sance sur un espace de recherche qui ne sera jamais parcouru soit parce que une stratégie gagnante aura déjà été obtenue soit parce que l'on aura excédé les ressources en temps ou en espace. L'espace de recherche pour l'algorithme de type Monte Carlo est alors une zone plus restreinte (en gris moyen et gris clair sur la figure 4) qui se déplace dans les zones inexplorées (en blanc toujours sur la même figure).

4 Conclusion

La figure 5 présente l'architecture actuelle du projet QuaCode contenant l'architecture, présentée dans cet article, de coopération entre un solveur complet et un algorithme de type Monte Carlo sans la prise en compte de la trace par l'heuristique mais disposant d'un analyseur de trace pour gérer la construction d'une (des) stratégie(s) gagnantes(s) mais aussi de certificats.

La figure 6 présente l'architecture à terme du projet QuaCode dans laquelle l'algorithme de type Monte Carlo prend en compte la trace du solveur QCSP complet et ne recherche que légèrement au delà de la frontière comme décrit dans la discussion au paragraphe 3.5. Cette architecture intègre un ensemble d'heuristiques complètes greffées sur le SIBus qui s'inspireront de l'apprentissage de lemmes. Mais cette architecture peut aussi prendre en compte d'autres greffons basés sur d'autres heuristiques stochastiques comme les algorithmes génétiques ou des méthodes tabous.

Les résultats expérimentaux préliminaires ont été effectués sur des contraintes arithmétiques pour lesquelles le calcul de la valeur de conflit est relativement aisée à concevoir. Il n'en est pas nécessairement de même pour d'autres contraintes issues du large cata-

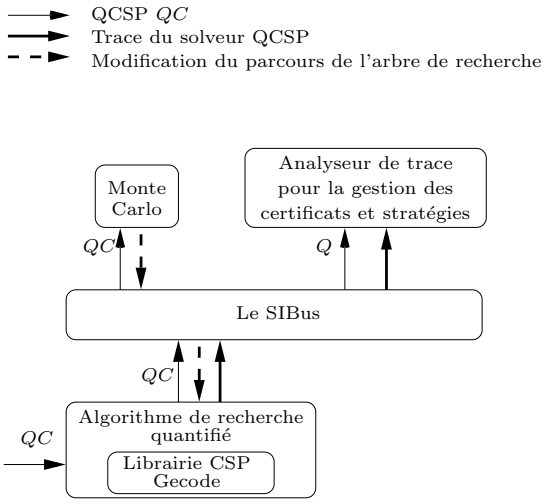


FIGURE 5 – L'architecture QuaCode actuelle

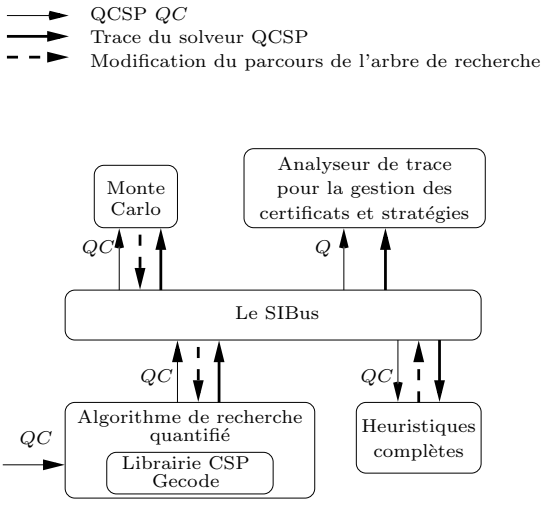


FIGURE 6 – L'architecture QuaCode à terme

logue de Gecode et une réflexion est à apporter sur la manière de choisir la fonction d'évaluation $evConflits$.

Références

[1] M. Benedetti, A. Lallouet, and J. Vautard. Modeling adversary scheduling with QCSP+. In *Proceedings of the 23th ACM Symposium on Applied Computing (SAC'08)*, pages 151–155, 2008.

[2] L. Bordeaux and E. Monfroy. Beyond NP : Arc-Consistency for Quantified Constraints. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 371–386, 2002.

[3] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 262–267, 1998.

[4] F. Garreau. QBF et colonie de fourmis, master thesis, Université of Angers, 2012.

[5] I.P. Gent, H.H. Hoos, A.G.D. Rowley, and K. Smyth. 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)*, 2003.

[6] I.P Gent, P. Nightingale, A. Rowley, and K. Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 172(6-7) :738–771, 2008.

[7] M. Loth, M. Sebag, Y. Hamadi, and M. Schoenauer. Bandit-Based Search for Constraint Programming. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP'13)*, pages 464–480, 2013.

[8] B. Satomi, Y. Joe, A. Iwasaki, and M. Yokoo. Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 655–661, 2011.

[9] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, 1994.

[10] I. Stéphan. Un panorama sur les procédures de décision séquentielles pour le problème de validité des formules booléennes quantifiées. *Revue d'Intelligence Artificielle*, 26-1&2 :163–196, 2012.

[11] E. Tsang. Foundations of constraint satisfaction. *Academic Press, London*, 1993.

[12] G. Verger and C. Bessiere. 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)*, pages 635–649, 2006.