BlockSolve : une approche bottom-up des QCSP

Guillaume Verger, Christian Bessière

BlockSolve : une approche bottom-up des QCSP

Guillaume Verger **Christian Bessiere**

LIRMM (CNRS/Université de Montpellier),
161 rue Ada, 34392 Montpellier Cedex 5, France.
verger@lirmm.fr bessiere@lirmm.fr

Résumé

Les problèmes de satisfaction de contraintes quantifiées (QCSP) sont une généralisation des problèmes de satisfaction de contraintes (CSP), dans lesquels chaque variable est quantifiée soit existentiellement, soit universellement. Ce type de problème s'inscrit notamment dans le domaine de la planification de tâches en présence de l'incertain. Les méthodes existantes de résolution de QCSP parcourent l'arbre de recherche dans l'ordre des variables du problème. Ces méthodes utilisent des généralisations des méthodes de propagation de contraintes pour les QCSP. Nous présentons BlockSolve, un algorithme de résolution de QCSP qui factorise les branches de l'arbre de recherche en blocs, et qui n'utilise que des méthodes de propagation des CSP classiques.

1 Introduction

Les problèmes de satisfaction de contraintes quantifiées (QCSP) sont une extension des problèmes de satisfaction de contraintes (CSP). La différence est l'utilisation de quantificateurs sur les variables, ce qui permet une plus grande expressivité de problèmes. Évidemment, le coût de cette expressivité est une augmentation de la complexité. Les CSP se situent en général dans la classe des problèmes NP-Complet, et les QCSP se situent dans la classe P-SPACE-Complet, les problèmes de P-SPACE étant considérés plus difficiles que ceux de NP[1]. P-SPACE est la classe de complexité des problèmes se résolvant avec une complexité spatiale polynomiale, sans borne sur la complexité temporelle. Cette généralisation des CSP suit la généralisation du problème de satisfaction de formules booléennes (SAT) en problème de satisfaction de formules booléennes quantifiées (QBF).

[1]en acceptant la conjecture P $\neq$ NP

La manière la plus naturelle de résoudre une instance de QBF ou de QCSP est d'instancier les variables dans l'ordre dans lequel elles sont données dans le problème, de la plus extérieure vers la plus intérieure. C'est une approche que l'on appelle *top-down*. La plupart des solvers QBF implémentent la technique top-down. Ces solvers étendent les algorithmes propres à SAT aux QBF. Cependant, Biere [2], ou encore Pan et Vardi [9] ont proposé d'autres techniques pour résoudre des instances de QBF. Les deux méthodes essaie d'éliminer les variables dans l'ordre inverse, c'est-à-dire de la plus intérieure vers la plus extérieure, dans une approche appelée *bottom-up*. L'approche bottom-up est motivée par le fait que les heuristiques qui sont utilisées pour résoudre les instances de SAT voient leur efficacité grandement réduite lors de l'ajout de quantificateurs universels. La contrepartie de cette approche est le coût plus élevé en espace.

Du coté des QCSP, la recherche n'en est encore qu'à ses débuts. Bordeaux et Montfroy [3] ont étendu la notion d'arc-consistance (AC) des CSP aux QCSP. Mamoulis et Stergiou [8] ont défini un algorithme d'AC pour les QCSP à contraintes binaires. Quelques solveurs basés sur ces techniques ont été récemment développés pour les QCSP, dont QCSP-Solve, par Gent, Nightingale et Stergiou [6]. Stergiou a de plus développé un autre solveur basé sur les méthodes de réparation [11]. QCSP-Solve est le solveur référence pour les QCSP : il est plus rapide que les méthodes utilisées jusqu'ici, c'est-à-dire la réécriture d'un QCSP en QBF pour la résolution via un solveur QBF.

Dans cet article, nous introduisons BlockSolve, le premier algorithme bottom-up qui résout des QCSP. Nous commencerons par définir les différentes notions dont nous aurons besoin dans cet article, puis nous décrirons l'algorithme BlockSolve, en commençant par une explication intuitive qui se traduira par un exemple de fonctionnement. L'approche du problème que nous avons avec

BlockSolve est assez différente de l'approche classique de ce genre de solveurs.

Premièrement, BlockSolve *n'utilise que des techniques des CSP classiques*. Nous ne définirons donc pas de généralisation de ces techniques aux QCSP. BlockSolve utilise un solveur de contraintes classique et un algorithme de propagation de contraintes comme *forward checking* (FC) [7] ou MAC [10].

Deuxièmement, BlockSolve *ne parcourt pas l'arbre dans le sens des variables du problème*, mais commence par le bas de l'arbre, et remonte jusqu'à la racine, dans une approche bottom-up. De plus il essaie de fusionner les noeuds de l'arbre dont les sous-arbres sont équivalents. La technique de factorisation utilisée ici pour BlockSolve est très proche de celle utilisée par Fargier *et al.* dans le cadre des Mixed CSP [4]. Les Mixed CSP sont des QCSP dans lesquels la séquence de variables est seulement composée de deux ensembles consécutifs, un ensemble quantifié universellement et l'autre existentiellement. Fargier *et al.* décomposent Mixed CSP pour les résoudre grâce à la technique de *subproblem extraction* décrite par Freuder et Hubbe dans [5]. Cette manière d'aborder le problème coûte en espace, nous l'étudierons après avoir décrit le comportement de BlockSolve.

Suite à ça nous expérimenterons notre algorithme sur des jeux de données aléatoires, en le comparant à QCSP-Solve, au niveau du temps de calcul et du nombre de noeuds de l'arbre visités. Nous discuterons des résultats produits par les deux algorithmes.

2 Préliminaires

Dans cette section nous définissons les différentes notions qui vont nous permettre de décrire le fonctionnement de notre algorithme de résolution de QCSP, BlockSolve.

Définition 1 (réseau de contraintes quantifiées) *Un réseau de contraintes quantifiées est une formule de la forme $\mathcal{QC}$ dans lequel :*

- *$\mathcal{Q}$ est une* **séquence** *de variables quantifiées $Q_i x_i, i \in [1, n]$, avec $Q_i \in \{\exists, \forall\}$, et x_i une variable au sens des CSP, avec un domaine de valeurs $D(x_i)$,*
- *$\mathcal{C}$ est une conjonction de contraintes $(c_1 \wedge ... \wedge c_m)$ portant sur les variables de $\mathcal{Q}$.*

Dans cet article, nous nous limitons aux contraintes binaires, c'est à dire impliquant deux variables. Une contrainte portant sur les variables x_i et x_j est notée c_{ij}.

Définissons maintenant ce qu'est une **solution** d'un réseau de contraintes quantifiées :

Définition 2 (arbre solution) *Une solution d'un réseau de contraintes quantifiées $\mathcal{QC}$ est un arbre dont :*

- *le noeud racine r n'a pas d'étiquette,*
- *chaque noeud s à distance i $(1 \leq i \leq n)$ de la racine r est étiqueté par l'instanciation $(x_i \leftarrow v)$ avec $v \in D(x_i)$,*
- *pour chaque noeud s à distance i de la racine, le nombre de successeurs de s dans l'arbre est $|D(x_{i+1})|$ si x_{i+1} est une variable universelle, ou 1 si x_{i+1} est une variable existentielle. Si x_{i+1} est universelle, chaque valeur w de $D(x_{i+1})$ apparaît dans l'étiquette de l'un des successeurs de s,*
- *pour chaque feuille, l'intanciation de $x_1, \ldots, x_n$ définie par les étiquettes des noeuds de r à la feuille satisfait toutes les contraintes dans $\mathcal{C}$.*

Dans un réseau de contraintes quantifiées, à l'inverse d'un réseau de contraintes classique, l'ordre des variables fait partie des données, en cela $\mathcal{Q}$ est une séquence et non un ensemble de variables.

Exemple 1 *Le réseau $\exists x_1 \forall x_2, x_1 \neq x_2, D(x_1) = D(x_2) = \{0, 1\}$ est insoluble, il n'existe pas de valeur pour x_1 dans $D(x_1)$ qui soit compatible avec toutes les valeurs de $D(x_2)$ pour x_2.*

Exemple 2 *Le réseau $\forall x_2 \exists x_1, x_1 \neq x_2, D(x_1) = D(x_2) = \{0, 1\}$ est soluble, pour chaque valeur que peut prendre x_2, x_1 peut prendre une valeur qui lui est différente.*

Notons qu'un réseau dans lequel tous les quantificateurs sont existentiels est un réseau de contraintes classique. Le réseau classique **correspondant** au réseau $\mathcal{QC}$ est défini par $X\mathcal{C}$, dans lequel X est l'ensemble des variables (non quantifiées) de $\mathcal{QC}$.

De la définition 2 découle naturellement la notion de **problème** de satisfaction de contraintes quantifiées :

Définition 3 (QCSP) *Un problème de satisfaction de contraintes quantifiées (QCSP) est un problème dont la donnée est un réseau de contraintes quantifiées et dont l'ensemble de solutions est l'ensemble des arbres solution de ce réseau*

Notons que cette définition des QCSP, même si différente dans la présentation, est équivalente aux définitions récursives habituelles. L'avantage de notre définition est qu'elle définit formellement ce qu'est un solution d'un QCSP.

Exemple 3 *Soit le problème $\exists x_1 \exists x_2 \forall x_3 \forall x_4 \exists x_5 \exists x_6$, $(x_1 \neq x_5) \wedge (x_1 \neq x_6) \wedge (x_2 \neq x_6) \wedge (x_3 \neq x_5) \wedge (x_4 \neq x_6) \wedge (x_3 \neq x_6)$. Chaque variable a le même domaine $D = \{0, 1, 2, 3\}$.*

La figure 1 nous montre une solution de ce problème.

Nous définissons maintenant la notion de **bloc** qui va nous suivre tout au long de l'article et grâce à laquelle nous allons exprimer les règles sur l'ordre des variables.

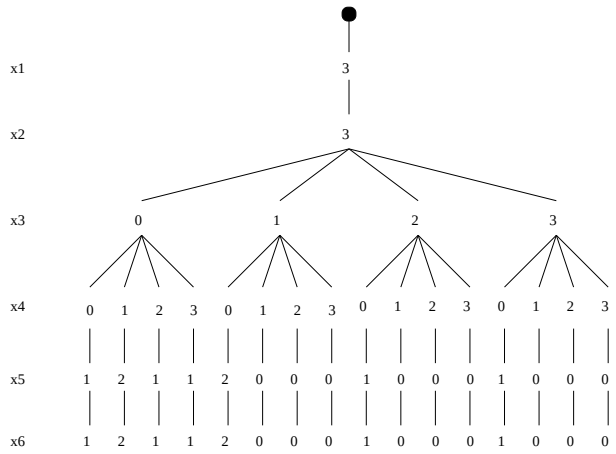


FIG. 1 – Une solution de l'exemple 3

Définition 4 (bloc) *Un bloc est une sous-séquence de variables consécutives dans $\mathcal{Q}$ de même quantificateur et de longueur maximale. Un bloc sera appelé* bloc universel *(resp.* bloc existentiel*) si les variables le composant sont quantifiées universellement (resp. existentiellement).*

Il est possible d'échanger l'ordre de deux variables à l'intérieur d'un même bloc, par contre le problème n'est plus le même lorsque deux variables de deux blocs différents sont échangées.

Nous dirons d'une variable x_i qu'elle est *intérieure* (resp. *extérieure*) à x_j si le bloc de x_i est placé plus tard (resp. plus tôt) dans la séquence que celui de x_j.

La notion de bloc, et le fait de pouvoir intervertir des variables au sein d'un même bloc, nous conduit à une nouvelle définition d'un **arbre solution**, qui est une contraction d'un arbre solution tel que défini plus haut :

Définition 5 (arbre solution) *Une solution d'un réseau de contraintes quantifiées $\mathcal{QC}$ est un arbre dont :*

- *le noeud racine r n'a pas d'étiquette,*
- *chaque noeud à distance i de la racine r représente le i^{eme} bloc de $\mathcal{Q}$,*
- *chaque noeud à distance i de la racine r est étiqueté par une instanciation de toutes les variables du i^{eme} bloc si c'est un bloc existentiel, ou par une union de produits cartésiens de sous-domaines de ses variables si c'est un bloc universel,*
- *pour chaque noeud s de profondeur i, le nombre de successeurs de s dans l'arbre est égal à 1 si le $(i+1)^{eme}$ bloc est existentiel, et au moins 1 si le $(i+1)^{eme}$ bloc est universel. Si le $(i+1)^{eme}$ bloc est universel, toutes les combinaisons de valeurs des domaines de ses variables apparaît dans l'étiquette de l'un des fils de s.*
- *pour chaque feuille, une instanciation de $x_1, \ldots, x_n$ définie par les étiquettes des noeuds existentiels entre r et la feuille et par n'importe quelle combinaison dans les étiquettes de noeuds universels satisfait toutes les contraintes de $\mathcal{C}$.*

Notons que le noeud racine n'est là que pour avoir un arbre et non une forêt lorsque le premier bloc est universel.

La figure 2 nous montre une solution du QCSP de l'exemple 3. Le noeud racine ne figure pas ici, car le premier bloc est existentiel. Le réseau est divisé en trois blocs, le premier et le dernier étant existentiels et le second universel. Les noeuds existentiels sont complètement instanciés, c'est-à-dire que pour chaque variable du bloc, une seule valeur lui correspond. Cette instanciation est une solution du CSP restreint aux variables du bloc. Le bloc universel, au contraire, est étiqueté par une union de produits cartésiens. La signification d'un tel noeud est que les valeurs autorisées pour les variables du bloc sont celles des tuples de cette étiquette. Cela correspond à une fusion des noeuds de l'arbre pour lesquels les sous-arbres sont équivalents. L'arbre de la figure 2 est donc la version condensée de l'arbre de la figure 1, dans laquelle les branches identiques ont été regroupées.

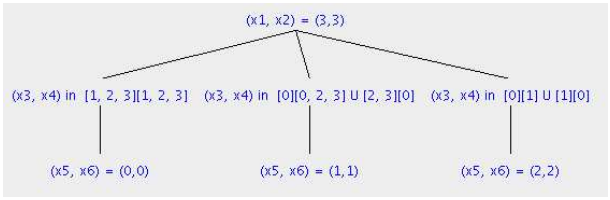


FIG. 2 – Solution de l'exemple 3

Notre algorithme `BlockSolve` utilise la notion de bloc, tant pour créer l'arbre solution sous la forme que nous avons décrite que pour résoudre le QCSP. Plus particulièrement, `BlockSolve` découpe un réseau de contraintes quantifiées en **niveaux**.

Définition 6 (niveau) *Un problème $\mathcal{P} = \mathcal{QC}$ se décompose en p **niveaux** $0 \ldots p-1$. Chaque niveau k est formé d'un bloc universel, noté $bloc_k^{\forall}$, et du bloc existentiel lui succédant dans $\mathcal{Q}$, noté $bloc_k^{\exists}$. Si le premier bloc de $\mathcal{Q}$ est existentiel, il forme à lui seul le niveau 0 et si le dernier bloc est universel il forme à lui seul le niveau $p-1$.*

Pour un niveau k donné, on obtient un sous-problème $\mathcal{P}_k$ pour lequel les variables prises en compte sont celles dont le niveau i est tel que $i \geq k$, et l'ensemble de contraintes $\mathcal{C}_k$ la restriction de $\mathcal{C}$ dans laquelle on ne garde que les contraintes dont toutes les variables impliquées sont dans $\mathcal{P}_k$. $\mathcal{P}_1$ correspond au problème $\mathcal{P}$ en entier. `BlockSolve` utilise cette notion de niveau afin de résoudre le problème. Le principe est de résoudre le sous-problème $\mathcal{P}_p$, puis de résoudre les sous-problèmes $\mathcal{P}_{p-1}$, ..., jusqu'à $\mathcal{P}_1 = \mathcal{P}$.

3 L'algorithme `BlockSolve`

Cette section est dédiée à la description de l'algorithme `BlockSolve`. Dans un premier temps nous décrivons le fonctionnement de l'algorithme à l'aide d'un exemple. Enfin nous traitons de l'algorithme plus en détail, et de sa complexité.

Comme Gent et al. le font dans QCSP-Solve, nous commençons par appliquer un pré-traitement qui permet de se débarrasser des contraintes de la forme $\forall x_i \forall x_j\ c_{ij}$ ainsi que celles de la forme $\exists x_i \forall x_j\ c_{ij}$ une fois pour toutes. En effet, pour les contraintes $\forall x_i \forall x_j\ c_{ij}$, s'il existe un couple (v_i, v_j) de valeurs pour x_i et x_j qui ne satisfasse pas c_{ij}, alors le problème est inconsistant. Sinon la contrainte sera toujours satisfaite et elle est donc inutile. Pour les contraintes de la forme $\exists x_i \forall x_j\ c_{ij}$, on supprime toutes les valeurs v_i du domaine $D(x_i)$ qui ne sont pas compatibles avec une valeur v_j de $D(x_j)$. Si $D(x_i)$ se vide de cette manière, le problème est inconsistant, sinon la contrainte est inutile car les valeurs conflictuelles ont été supprimées.

`BlockSolve` démarre une fois le pré-traitement appliqué. `BlockSolve` utilise les techniques classiques de propagation des CSP. Il n'utilise pas d'adaptation de ces techniques pour les QCSP et s'intègre donc dans un solveur classique comme Choco [1], duquel il hérite les algorithmes de propagation.

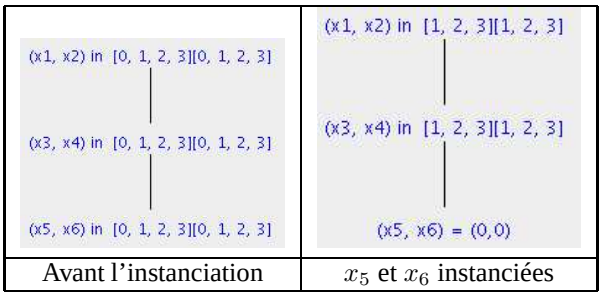
Nous allons dérouler `BlockSolve` sur un exemple simple avant de rentrer dans la description détaillée de l'algorithme.

3.1 Comprendre par l'exemple

Le problème que nous traitons ici est celui de l'exemple 3, dont la solution nous est présentée dans la figure 2. Les différents tableaux montrent le déroulement de l'algorithme sur l'exemple.

L'intuition derrière `BlockSolve` est de partir des variables du dernier bloc et de remonter vers la racine en instanciant les variables existentielles. Chaque instanciation d'une variable $\exists x_i$ avec une valeur v_i peut nous amener à supprimer des valeurs incompatibles avec v_i dans les domaines des variables extérieures à x_i (FC suffit à faire ça). Supprimer une valeur v_j dans le domaine d'une variable $\exists x_j$ est similaire au cas des CSP : tant que $D(x_j)$ n'est pas vide, on peut continuer le processus de remontée. Supprimer v_k du domaine d'une variable $\forall x_k$ supérieure à x_i signifie qu'il faudra trouver une autre instanciation des variables existentielles intérieures à x_k qui accepte v_k. Mais l'instanciation courante n'est pas jetée pour autant. Elle peut être compatible avec tout un ensemble de tuples du produit cartésien des domaines des blocs existentiels précédents. Plus gros est cet ensemble de tuples, plus grande est la factorisation des branches, et plus grand est le gain par rapport à une recherche "de haut en bas".

Pour un niveau k, `BlockSolve` cherche une solution au sous-problème $\mathcal{P}_{k+1}$. Au début, `BlockSolve` cherche donc à instancier le dernier bloc. Ici, c'est le bloc composé de x_5 et x_6. `BlockSolve` essaie d'instancier ces variables comme si le réseau de contraintes quantifiées était un réseau de contraintes classiques.

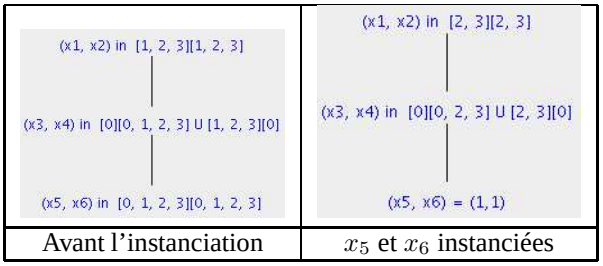


Avant l'instanciation	x_5 et x_6 instanciées

Pour l'instanciation $x_5 = 0, x_6 = 0$, FC réduit les domaines des autres variables. Comme il n'y a pas de contrainte entre (x_1, x_2) et (x_3, x_4) (voir pré-traitement) et que les contraintes sont toutes binaires, FC assure que tous les tuples restants de (x_1, x_2) sont compatibles avec l'instanciation. Mais FC n'assure pas qu'il existe une solution pour (x_1, x_2). Il garantit seulement que si on trouve une instanciation de (x_1, x_2) sans plus penser à (x_5, x_6), elle sera compatible avec ce dernier bloc.

FC a supprimé la valeur 0 pour les variables universelles x_3 et x_4 de la même façon que pour les existentielles x_1 et x_2. A ce stade, pour les combinaisons de valeurs sur (x_3, x_4) contenant 0 pour x_3 ou 0 pour x_4, `BlockSolve` n'a pas encore trouvé d'instanciation pour le dernier bloc. Il faudra donc trouver une solution partielle sur (x_5, x_6) pour les tuples de (x_3, x_4) qui n'étaient pas compatibles avec l'instanciation $x_5 = 0, x_6 = 0$.

On répète l'étape précédente avec les tuples de $D(x_3) \times D(x_4)$ pour lesquels on n'a pas encore d'instanciation de (x_5, x_6), soit $\{0\} \times \{0, 1, 2, 3\} \cup \{1, 2, 3\} \times \{0\}$ (les tuples de $D(x_3) \times D(x_4)$ ne sont donc pas totalement restaurés). Notons que les domaines $D(x_1)$ et $D(x_2)$ ne sont pas restaurés car on veut que les valeurs que prendront x_1, x_2 soient valables pour tous les tuples sur (x_3, x_4).



Avant l'instanciation	x_5 et x_6 instanciées

`BlockSolve` a trouvé l'instanciation $x_5 = 1, x_6 = 1$, ce qui a supprimé la valeur 1 pour x_1, x_2, x_3 et x_4. Il reste donc encore les tuples $(0, 1)$ et $(1, 0)$ non couverts pour (x_3, x_4).

Avant l'instanciation | x_5 et x_6 instanciées

Dernière étape, cette fois-ci l'instanciation de x_5 et x_6 ne réduit pas l'ensemble de tuples de (x_3, x_4). L'algorithme a donc trouvé une solution pour $\mathcal{P}_1$ et a réduit les domaines des variables plus haut dans l'arbre. Il ne reste qu'à instancier x_1 et x_2, mais comme les domaines ont été réduits à une valeur et qu'elles sont compatibles, la résolution est terminée et l'arbre solution est celui de la figure 2. Notons que les premières instanciations de x_5 et x_6 sont bien compatibles avec $x_1 = 3, x_2 = 3$, car le tuple $(3, 3)$ était présent dans l'ensemble de tuples compatibles au moment des instanciations successives de x_5 et x_6.

3.2 Description de l'algorithme

Dans cette section, nous examinons plus en détail `BlockSolve`, présenté dans l'algorithme 1. Comme vu dans l'exemple précédent, nous avons besoin de garder en mémoire les ensembles des tuples des variables de chaque blocs. C'est ce qui est fait dans les 2 tableaux $T_\forall[]$ et $T_\exists[]$.

`BlockSolve` fonctionne ainsi : pour un niveau k donné (on commence au niveau 1), on essaie de résoudre le sous-problème $\mathcal{P}_{k+1}$ tout en restant compatible avec les valeurs des variables extérieures à $\mathcal{P}_{k+1}$. Une fois le $\mathcal{P}_{k+1}$ résolu, `BlockSolve` tente de résoudre le niveau k en instanciant les variables existentielles. Les domaines des variables universelles du niveau k ont été réduits par les instanciations des variables existentielles des niveaux k à p, p étant le nombre de niveaux. `BlockSolve` va donc tenter de résoudre $\mathcal{P}_k$ de la même manière que précédemment, mais pour les valeurs des variables universelles qui n'ont pas été traitées. Dans le cas d'un échec, on retourne au niveau supérieur, et la recherche continue.

Le premier appel de `BlockSolve` se fait avec en paramètres le problème $\mathcal{P}_1$ et les ensembles de tuples des valeurs des variables des blocs, au niveau 1.

3.2.1 Pas à pas

A la ligne 1, le cas de terminaison renvoie $true$ quand le nombre de niveaux de $\mathcal{P}$ est dépassé.
A la ligne 2, on sauvegarde $T_\forall[k]$ et $T_\exists[k]$ afin de pouvoir restaurer le contexte avant de remonter au niveau précédent.
A la ligne 3, on cherche à trouver une solution du sous-problème $\mathcal{P}_k$ pour chacun des tuples de $T_\forall[k]$. Pour chaque solution partielle trouvée, on va réduire $T_\forall[k]$ jusqu'à ce qu'il soit vide. Qu'il soit vide indique que l'on a trouvé une solution partielle pour tous les tuples de $T_\forall[k]$.
A la ligne 4, on sauvegarde l'état du problème plus haut (les niveaux 0 à k) dans l'arbre avant de lancer la recherche plus profondément.
A la ligne 5, si le résultat de la résolution de $\mathcal{P}_{k+1}$ est $true$, les ensembles $T_\forall[i], \forall i \leq k$ et $T_\exists[i], \forall i \leq k$ sont compatibles (et ont été éventuellement réduits) avec la solution trouvées dans le sous-problème $\mathcal{P}_{k+1}$.
A la ligne 6, les ensembles $C_\forall$ et $C_\exists$ contiennent les tuples du niveau compatibles avec la solution partielle de $\mathcal{P}_{k+1}$ (si solved est $true$) ou les tuples incompatibles (solved est $false$).
De la ligne 7 à la ligne 10 (boucle repeat-until), l'algorithme cherche à instancier les variables existentielles du niveau k pour tous les tuples de $T_\forall[k]$ qui représentent les valeurs des variables universelles pour lesquelles on a résolu le sous-problème $\mathcal{P}_{k+1}$.
A la ligne 8, on essaie d'instancier les variables existentielles sachant qu'elles doivent être compatibles avec la solution partielle de $\mathcal{P}_{k+1}$. La fonction `solve-level`[2] modifie les ensembles de tuples $T_\forall[]$ et $T_\exists[]$: si on réussit à instancier les variables existentielles du niveau, il est possible que l'on ait réduit l'ensemble $T_\forall[k]$ depuis la résolution partielle de $\mathcal{P}_{k+1}$ (ligne 5). Dans ce cas, on sait qu'il reste des tuples de valeurs des variables de $bloc_k^\forall$ compatibles avec la solution partielle de $\mathcal{P}_{k+1}$ calculée précédemment pour lesquels on n'a pas encore trouvé d'instanciation valide. On va maintenant chercher à instancier de nouveau les variables de $bloc_k^\exists$ avec les tuples de $T_\forall[k]$ pour lesquels on a la solution partielle mais pas encore l'instanciation. C'est ce qui est fait à la ligne 9, avant de revenir au début de la boucle. Dans le cas où `solve-level` renvoie faux, il faut sortir de la boucle.
A la ligne 11, on a instancié les variables de $bloc_k^\exists$ autant que c'était possible pour la solution partielle de $\mathcal{P}_{k+1}$ (et au moins une fois). On va alors devoir résoudre le problème dans les tuples de $T_\forall[k]$ pour lesquels on n'a pas trouvé de solution à $\mathcal{P}_{k+1}$ et $bloc_k^\exists$.
L'algorithme passe par la ligne 12 lorsqu'une solution partielle de $\mathcal{P}_{k+1}$ a été trouvée, mais qu'il n'existe pas d'instanciation valide des variables de $bloc_k^\exists$.
A la ligne 13, on met à jour les domaines des variables de $bloc_k^\exists$, comme on n'a pas trouvé de solution qui satisfasse à la fois $\mathcal{P}_{k+1}$ et les variables de $bloc_k^\exists$. On sait alors que pour les restrictions sur les domaines de variables des blocs précédant le niveau courant , il n'existe pas de solution pour les variables de $bloc_k^\exists$ dans les domaines $C_\exists$. On peut donc les supprimer de $T_\exists[k]$, afin de vérifier $\mathcal{P}_k$ pour les domaines restant.

[2] un appel à `solve-level` est en fait un appel à un solveur de CSP classique, qui n'instancie que les variables existentielles du niveau, et qui applique FC

```
   Data: problème P_P, T_∀[], T_∃[], niveau k
   Result: Vrai si le problème a une solution, Faux sinon
   begin
1      if k > nombre de niveaux then return true;
2      A_∀ ← T_∀[k] ; A_∃ ← T_∃[k];
3      while T_∀[k] ≠ ∅ do
4          B_∀[0..k] ← T_∀[0..k] ; B_∃[0..k] ← T_∃[0..k];
5          solved ← BlockSolve(k + 1);
6          C_∃ ← T_∃[k] ; C_∀ ← T_∀[k];
           if solved then
7              repeat
                   D_∀ ← T_∀[k];
8                  instantiated ←solve-level(k);
9                  if instantiated then T_∃[k] ← C_∃ ; T_∀[k] ← D_∀ − T_∀[k];
                   else T_∀[k] ← D_∀;
10             until T_∀[k] = ∅ ∨ ¬instantiated ;
11             if C_∀ ≠ T_∀[k] then T_∃[k] ← B_∃[k] ; T_∀[k] ← B_∀[k] − (C_∀ − T_∀[k]);
12             else solved ← false
           if ¬solved then
13             T_∃[k] ← B_∃[k] − C_∃ ;
14             if T_∃[k] = ∅ then
                   T_∀[k] ← A_∀ ; T_∃[k] ← A_∃;
                   return false;
15             T_∃[0..k − 1] ← B_∃[0..k − 1] ; T_∀[0..k] ← B_∀[0..k];
16     T_∀[k] ← A_∀ ; T_∃[k] ← A_∃;
       return true;
   end
```

Algorithm 1 – `BlockSolve`

A la ligne 14, on a fini d'explorer l'arbre de recherche au niveau des variables de $bloc_k^{\exists}$: il n'existe plus de tuple de valeurs des domaines de ces variables valide. On en déduit que le problème $\mathcal{P}_k$ n'est pas soluble.
A la ligne 15, `BlockSolve` se replace dans la même partie de l'arbre qu'avant la descente qui s'est soldée par un échec, mais avec les domaines des existentielles modifiés (ligne 13).
A la ligne 16, on a trouvé des solutions partielles pour tous les tuples des variables de $bloc_k^{\forall}$, $\mathcal{P}_k$ est résolu.

3.2.2 Une vue plus globale de `BlockSolve`

Plusieurs choses sont à garder à l'esprit lors du fonctionnement de `BlockSolve`.

Les contraintes sont des contraintes binaires, forward-checking suffit donc à assurer que lors de la résolution d'un problème $\mathcal{P}_k$, toutes les variables des problèmes extérieurs (i.e. $\mathcal{P}_i, \forall i < k$) ne gardent comme valeurs possibles seulement celles qui sont compatibles avec les valeurs prises par les variables du problème $\mathcal{P}_k$. Donc il est possible "d'oublier" la solution partielle de $\mathcal{P}_k$ quand on essaie d'instancier les variables existentielles au niveau i.

La fonction `solve-level` est un algorithme classique de résolution de contraintes à qui l'on demande d'instancier les variables existentielles du niveau. Pour lui, toutes les variables du problème sont des variables existentielles. De plus il doit disposer d'un algorithme de propagation de contraintes comme forward-checking, ou plus puissant. Évidemment plus l'algorithme de propagation sera précis, moins on explorera de sous-arbres inutilement, exactement de la même manière que pour les CSP classiques.

`BlockSolve` peut donner la solution d'un QCSP sous la forme décrite dans la définition 5. Pour construire une telle solution, nous modifions l'algorithme : `BlockSolve` prend en paramètre supplémentaire un noeud d'arbre. Ce noeud correspond au bloc existentiel précédant $\mathcal{P}_k$. Quand une solution de $\mathcal{P}_{k+1}$ est trouvée, elle est décrite sous la forme d'un arbre. Pour chaque instanciation des variables de $bloc_k^{\exists}$, on crée le noeud universel et le noeud existentiel correspondants, et l'on y greffe la solution de $\mathcal{P}_{k+1}$. La branche créée est ensuite attachée au noeud passé en paramètre, afin que l'on puisse le récupérer au niveau $k-1$. Dans le cas d'un échec, on n'attache pas de noeud.

3.3 Complexité spatiale

`BlockSolve` garde en mémoire les tuples des universels et des existentiels pour lesquels on a déjà visité l'arbre de recherche. La taille de ces ensembles est bien sûr exponentielle avec le nombre de variables du bloc. Nous savons que QCSP se place dans P-SPACE, qui est la classe des problèmes de *décision* se résolvant en utilisant un espace polynomial en fonction de la taille des données. Notre algorithme utilise plus de place que ce que peut utiliser un autre algorithme de résolution de QCSP, comme QCSP-Solve.

Mais il est sûrement souhaitable qu'un algorithme résolvant un QCSP donne une solution du problème plutôt que de se contenter de dire si oui ou non le problème a une solution. Exhiber la solution est coûteux en espace, car la taille de l'arbre est exponentielle. Au niveau théorique donc, tout algorithme donnant une solution d'un QCSP a besoin d'une taille exponentielle en mémoire.

Nous stockons ces tuples sous la forme d'unions de produits cartésiens. Quand tous les tuples sont autorisés, un bloc est représenté par le produit cartésien des domaines des valeurs. Garder les tuples sous cette forme utilise moins de place que de définir en extension les valeurs du bloc. De plus l'opération de soustraction entre ensembles de tuples est plus rapide.

4 Expérimentations

Dans cette section, nous présentons des résultats expérimentaux sur des problèmes aléatoires. Nous comparons notre méthode à QCSP-Solve au niveau du temps CPU, et du nombre de noeuds visités.

4.1 Caractéristiques de `BlockSolve`

`BlockSolve` a été développé en Java à l'aide de la librairie de contraintes Choco [1]. Dans notre implantation, avant de lancer la procédure principale, `BlockSolve` trie les variables de chaque bloc entre elles, par nombre de contraintes liées décroissant. Il est nécessaire aussi de créer les contraintes des tuples de blocs.

Avant chaque instanciation de variable, le solveur de contraintes que nous utilisons (la fonction `solve-level`) trie le domaine de la variable afin d'affecter en premier la valeur qui est compatible avec le maximum de tuples dans les blocs supérieurs. Pour ce faire, il instancie la variable, puis par arc-consistance réduit les domaines des autres variables, et calcule le nombre de branches (le nombre de tuples des blocs) dans lequel il est. Le but étant de se placer dans un maximum de branches en même temps. Ce test est assez long, mais il diminue fortement le nombre de noeuds visités pendant la recherche. De plus il permet de ne pas couper trop souvent les ensembles de tuples, et donc de pouvoir les maintenir à une taille relativement faible.

4.2 Le générateur de problèmes aléatoires

Les instances de problèmes ont été créées à l'aide d'un générateur créé à partir du modèle proposé dans [6]. Dans ce modèle, les problèmes contiennent 3 blocs, deux existentiels et un universel. Il prend en compte 7 paramètres : $< n, n_\forall, n_{pos}, d, p, q_{\forall\exists}, q_{\exists\exists} >$ où n est le nombre de variables, $n_\forall$ le nombre de variables universelles, n_{pos} la position de la première variable universelle dans la séquence, d le nombre de valeurs dans le domaine de chaque variable, p la densité du problème, c'est-à-dire le pourcentage du nombre de contraintes sur le nombre de contraintes possibles. Les contraintes possibles sont de la forme $\forall x_i \exists x_j\ c_{ij}$ ou de la forme $\exists x_i \exists x_j\ c_{ij}$ car les autres contraintes peuvent être supprimées par prétraitement. $q_{\exists\exists}$ indique la proportion de tuples autorisés dans les contraintes de type $\exists x_i \exists x_j\ c_{ij}$. Pour $q_{\forall\exists}$, on génère une bijection de $D(x_i)$ vers $D(x_j)$, et $q_{\forall\exists}$ est la proportion de tuples de la bijection autorisés. Les tuples de $D(x_i) \times D(x_j)$ n'appartenant pas à la bijection sont autorisés. Ceci pour éviter que les problèmes soient surcontraints.

Nous ajoutons un paramètre au générateur, $b_\forall$ qui est le nombre de blocs universels dans le problème. Les variables sont séquencées comme suit : $n_{pos} - 1$ existentielles suivies de $n_\forall$ universelles, puis encore $n_{pos} - 1$ existentielles suivies de $n_\forall$ universelles...

4.3 Résultats expérimentaux

Nous présentons dans cette section les résultats obtenus sur des instances générées aléatoirement par le générateur. Nous faisons varier $q_{\exists\exists}$ de 5% à 95%, et pour chacune des valeurs de $q_{\exists\exists}$, 100 instances ont été générées.

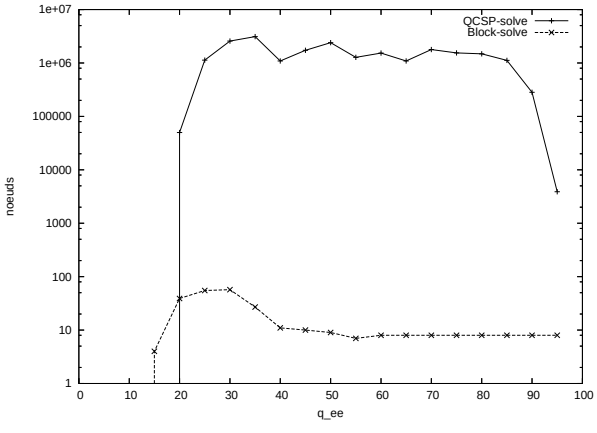


FIG. 3 – Nombre de noeuds pour $n = 15, n_\forall = 7, b_\forall = 1, d = 15, p = 30, q_{\forall\exists} = 50$ où $q_{\exists\exists}$ varie.

Les trois premières figures sont tirées de tests sur des

problèmes de 15 variables réparties en 3 blocs, le premier étant existentiel. Le bloc universel contient 7 variables, les existentiels 4 variables. Dans ces problèmes, la transition de phase se situe à $q_{\forall\exists} = 25$ La figure 3 compare QCSP-Solve et `BlockSolve` en terme de nombre de noeuds visités. Nous pouvons voir ici que `BlockSolve` explore beaucoup moins de noeuds que QCSP-Solve. Mais n'oublions pas qu'à chaque noeud, `BlockSolve` effectue quantité d'opérations : il trie le domaine de la variable courante suivant le nombre de tuples de valeurs des blocs extérieurs avec lesquels ses valeurs sont compatibles. De plus quand un bloc est instancié, il effectue des différences entre les ensembles de tuples des blocs.

Si l'on analyse le nombre de noeuds explorés, on s'aperçoit que QCSP-Solve, qui est meilleur pour les problèmes inconsistants (pour les petites valeurs de $q_{\exists\exists}$), a l'air de résoudre difficilement les problèmes ayant une solution. `BlockSolve` quant à lui trouve une solution sans backtrack. Ce qui signifie que pour les problèmes faciles, il existe une instanciation du bloc intérieur qui est consistant avec tous les tuples du bloc universel.

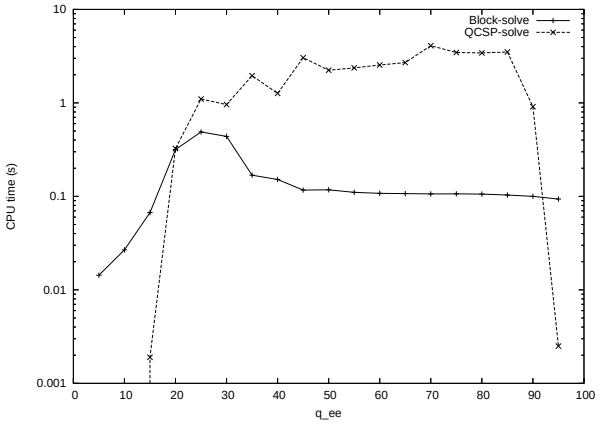


FIG. 4 – temps cpu pour $n = 15, n_\forall = 7, b_\forall = 1, d = 15, p = 30, q_{\forall\exists} = 50$ où $q_{\exists\exists}$ varie. (transition de phase à $q_{\exists\exists} = 25$)

La figure 4 montre les résultats en terme de temps CPU. Si l'on compare à la figure 3, il est clair que `BlockSolve` prend beaucoup de temps à explorer un noeud. Notons que QCSP-Solve détermine plus vite que `BlockSolve` qu'un problème est inconsistant ($q_{\exists\exists} < 25$), mais que `BlockSolve` trouve une solution beaucoup plus rapidement lorsqu'il en existe une ($q_{\exists\exists} > 25$).

Il est intéressant de voir que `BlockSolve` est plus stable que QCSP-Solve, comme le montre la figure 5. Dans cette figure, chaque point représente une instance de problème. Pour les problèmes satisfiables, il est difficile de prévoir en combien de temps QCSP-Solve va trouver une solution.

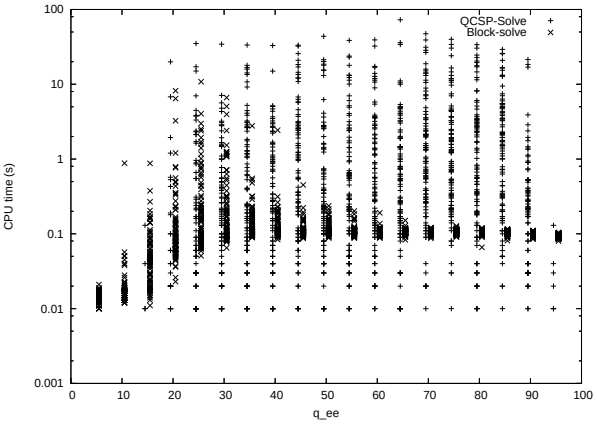
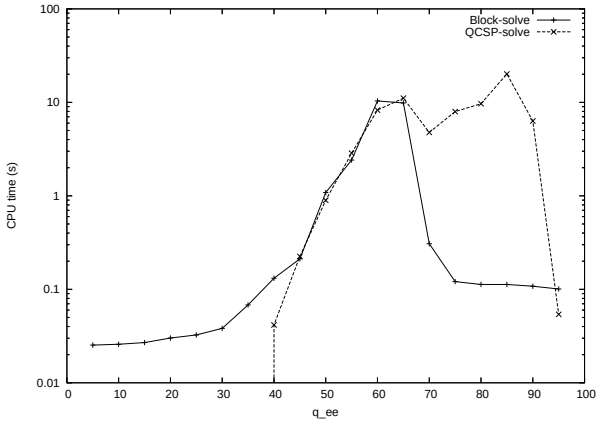
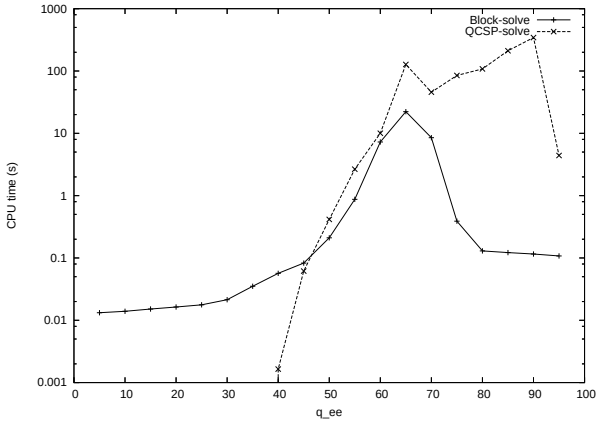


FIG. 5 – Dispersion du temps cpu sur $n = 15, n_\forall = 7, b_\forall = 1, d = 15, p = 30, q_{\forall\exists} = 50$



temps cpu pour $n = 25, n_\forall = 5, b_\forall = 2,$
$d = 8, p = 20, q_{\forall\exists} = 50$
(transition de phase à $q_{\exists\exists} = 55$)



temps cpu pour $n = 28, n_\forall = 4, b_\forall = 3,$
$d = 8, p = 20, q_{\forall\exists} = 50$
(transition de phase à $q_{\exists\exists} = 60$)

FIG. 6 – Problèmes de 25 variables dans 5 blocs (haut) et 28 variables dans 7 blocs (bas)

Nous avons lancé les deux algorithmes sur des problèmes de plus de trois blocs. La figure 6 présente les résultats des tests pour des instances de cinq blocs ($\exists\forall\exists\forall\exists$) de cinq variables chacun, et des tests pour des instances de sept blocs de quatre variables chacun. Ces tests montrent que le comportement général des deux algorithmes reste semblable même quand le nombre de blocs augmente. Pour les problèmes insolubles, QCSP-Solve détecte l'inconsistance plus vite, mais `BlockSolve` trouve une solution plus rapidement pour les problèmes solubles.

5 Conclusion

Nous avons présenté `BlockSolve`, un solveur bottom-up de QCSP qui s'appuie sur des algorithmes de résolution de CSP classiques. Son originalité réside dans le fait de traiter les variables existentielles du bas de l'arbre vers le haut, permettant ainsi une factorisation des branches. Plus cette factorisation est importante, plus on réduit le nombre de noeuds de l'arbre de recherche à visiter.

Les résultats obtenus par expérimentation sont positifs : la factorisation des branches permet une bonne régularité tant dans la rapidité de réponse que dans le nombre de noeuds visités pour les instances de problèmes satisfiables. Le nombre de noeuds visités par `BlockSolve` reste souvent très faible, bien plus que pour le solveur référence QCSP.

Les travaux futurs vont se diriger principalement vers l'utilisation de techniques plus avancées de backtracking, ainsi que vers la généralisation de `BlockSolve` à des contraintes n-aires.

Remerciements

Nous tenons particulièrement à remercier Kostas Stergiou, Peter Nightingale et Ian Gent, qui nous ont gracieusement fourni le code de QCSP-Solve.

Références

[1] Choco : librairie de contraintes en java, http ://choco.sourceforge.net/.

[2] Armin Biere. Resolve and expand. In *SAT*, 2004.

[3] L. Bordeaux and E. Montfroy. Beyond np : Arc-consistency for quantified constraints. In *Proceedings CP'02*, pages 371–386, Ithaca NY, 2002.

[4] Hne Fargier, Jme Lang, and Thomas Schiex. Mixed constraint satisfaction : A framework for decision problems under incomplete knowledge. In *AAAI/IAAI, Vol. 1*, pages 175–180, 1996.

[5] Eugene C. Freuder and Paul D. Hubbe. Extracting constraint satisfaction subproblems. In *IJCAI*, pages 548–557, 1995.

[6] I.P. Gent, P. Nightingale, and K. Stergiou. QCSP-solve : A solver for quantified constraint satisfaction problems. In *Proceedings IJCAI'05*, pages 138–143, Edinburgh, Scotland, 2005.

[7] R.M. Haralick and G.L. Elliott. Increasing tree seach efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[8] N. Mamoulis and K. Stergiou. Algorithms for quantified constraint satisfaction problems. In *Proceedings CP'04*, pages 752–756, Toronto, Canada, 2004.

[9] Guoqiang Pan and Moshe Y. Vardi. Symbolic decision procedures for qbf. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2004.

[10] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA, 1994.

[11] K. Stergiou. Repair-based methods for quantified csps. In *Proceedings CP'05*, pages 652–666, Sitges, Spain, 2005.