Des contraintes globales prêtes à brancher

Guillaume Richaud, Xavier Lorca, Narendra Jussien

Des contraintes globales prêtes à brancher

Guillaume Richaud **Xavier Lorca** **Narendra Jussien**

École des Mines de Nantes, LINA UMR CNRS 6241, F–44307 Nantes Cedex 3, France
{guillaume.richaud,xavier.lorca,narendra.jussien}@emn.fr

Résumé

Les contraintes globales représentent un outil indispensable dans la résolution et la modélisation de problèmes combinatoires. Il s'agit d'un concept clé de la programmation par contraintes qui, par sa généralité, peut s'étendre à d'autres paradigmes. Pourtant, concevoir, implémenter et maintenir des contraintes globales sont des exercices difficiles, très liés au solveur sous-jacent dans lequel elles sont implémentées. Cet article vise à jeter les bases de contraintes *prêtes à brancher* clairement découplées du solveur hôte. Ainsi, trois points sont particulièrement étudiés : comment une contrainte globale est liée à un solveur ? comment rendre indépendantes du solveur les structures de données internes des contraintes ? dans quelle mesure ce découplage introduit-il un surcoût ?

Abstract

Global constraints represent invaluable modeling tools for Constraint Programming (CP). They represent more than ever a key feature for the technology and even more generally for all search paradigms. However, designing, implementing and maintaining global constraints is a tedious task, from a practical point of view, generally strongly related to the solver. In this context, this paper aims at paving the way for pluggable constraints that allows a clear separation of the global constraints from the host solver. For this purpose, three bolts are studied : (1) How a global constraint is connected with its host solver ? (2) How the internal data structures of a global constraint could be made independent from the related solver ? (3) Is there a real overhead due to this decoupling ?

1 Introduction

La programmation par contraintes est considérée maintenant comme une technique de choix pour traiter de problèmes combinatoires complexes. Par contre, la quête du Graal décrite par Eugene Freuder il y a quelques années n'est toujours pas achevée : résoudre un problème avec la programmation par contraintes n'est toujours pas à la portée d'un seul clic de souris. Choisir le bon solveur, l'intégrer à l'applicatif existant, modéliser proprement et efficacement le problème, trouver la bonne heuristique de recherche restent des étapes à la fois nécessaires et réservées à des experts du domaine. Récemment, certaines de ces étapes sont traitées par l'initiative de Jacob Feldman : CP inside[1]. Cette initiative a pour but de rendre la technologie contraintes disponible aux utilisateurs finaux par l'intermédiaire d'interfaces communément utilisées dans des outils commerciaux ou non (comme Microsoft Excel, Google apps, etc.). La possibilité d'embarquer la technologie dans des outils grand-public permettra d'introduire les moteurs décisionnels basés sur les contraintes comme des composants naturels des applicatifs métiers traditionnels.

Dans cet article, nous voudrions traiter un point important et orthogonal : la capacité de *brancher* une contrainte globale sur différents solveurs. Jusqu'à présent, les contraintes globales restent un composant-clé des solveurs de contraintes. Elles permettent de traiter efficacement de nombreux problèmes difficiles grâce à leur capacité à encapsuler des sous-problèmes récurrents dans un algorithme de résolution à base de propagation/filtrage. Elle représentent un des éléments phare de la programmation par contraintes et plus généralement des techniques de résolution (y compris les stratégies de recherche locale - hybrides, les solveurs à base d'explications/nogoods, etc.). Mais, développer une nouvelle contrainte globale est une tâche ardue et souvent ingrate. En effet, l'information généralement manipulée en provenance des variables (via leur domaine) est généralement trop légère pour développer un algorithme de filtrage efficace. Ainsi, les contraintes globales maintiennent généralement une structure de données interne complexe (on peut se référer au cas des contraintes sur les graphes pour s'en convaincre) pour implémenter des algorithmes de filtrage efficaces. Ainsi, la complexité d'im-

[1] http://4c.ucc.ie/web/posters/CPInsidePoster.pdf

plémentation d'une contraintes globale repose généralement sur le maintien efficace de cette structure de données interne [5, 2]. Aujourd'hui, les contraintes globales sont liées à un solveur de contraintes donné car elles reposent généralement sur les structures backtrackables de ce solveur. Ces structures sont utilisées pour le maintien efficace de propriétés utilisées par les algorithmes de filtrage (les composantes connexes d'un graphe par exemple) quand le solveur doit revenir en arrière. Ces structures sont généralement fortes consommatrices de CPU (avec des approches à base de recalcul) ou de mémoire (pour les approches à base de trailing ou de recopie) et ne permettent ainsi pas d'accéder aux dernières nouveautés du langage support (par exemple, dans le cas de Java, les *generics*). Ainsi, l'intérêt même de développer une nouvelle contrainte globale peut être remis en question car reposant sur des technologies dépassées.

L'objectif de cet article est de défricher les bases de ce que pourrait être un cadre général pour une séparation claire des contraintes globales de leur solveur hôte. Au delà même, il s'agit aussi de permettre l'utilisation des contraintes globales dans des paradigmes de recherche plus génériques : solveur à base d'explications [10] ou techniques de recherche locale [21]. Il ne s'agit, en aucun cas, d'un nouveau cadre de description des contraintes globales à l'image de ce que sont, par exemple, les propriétés de graphes [3] ou les contraintes *Range* et *Roots* [7], mais bien d'un schéma d'implémentation pour découpler une contrainte globale du solveur dans lequel elle a été développée (voir la figure 1). Nous nous intéressons à trois points particuliers :

- comment une contrainte globale est effectivement reliée au solveur sous-jacent ?
- comment les structures de données internes d'une contrainte globale peuvent être rendues indépendantes du solveur sous-jacent ?
- est-ce que le découplage introduit un sur-coût ?

Un premier pas significatif dans la fourniture d'un tel cadre général consiste à fournir un squelette général pour une contrainte indépendante de tout solveur (voir la figure 1). Ainsi, une contrainte *prête à brancher* peut être définie à l'aide de deux sortes d'objets : un type de donnée abstrait (TDA) qui décrit la structure de données interne et des algorithmes de filtrage. Les événements liés aux variables du solveur devant être traités par la contrainte, ils doivent en être *compris* pour qu'elle puisse mettre à jour le TDA et, de manière symétrique, les mises à jour du TDA doivent être répercutées du côté solveur. Ceci implique que ces mises à jours soient elles-même converties en événements sur les variables. C'est le rôle que joue l'interface présentée dans la figure 1.

Par la suite, nous commençons par décrire le contexte général de notre travail (section 3). Puis, nous discutons les différents points clés qui permettent de découpler proprement une contrainte de son solveur hôte (section 4). Après une introduction générale à l'approche (section 4.1), ce schéma de développement est appliqué à deux contraintes globales : la contrainte `boundAllDiff` [13] (section 4.2) et la contrainte `tree` [1] (section 4.3) ; et ce, dans le contexte de deux solveurs : `Choco` (`choco.emn.fr`) et `Gecode` (`gecode.org`). Alors, nous présentons une évaluation de l'utilisation pratique de ce schéma (section 5) en commençant par la comparison de chacune des contraintes *classiques* (`boundAllDiff` et `tree`) avec sa version prête à brancher (section 5.1). Ensuite, nous présentons un exemple pratique mêlant ces deux contraintes : le problème de chemin hamiltonien (section 5.2). Enfin, nous terminons par une discussion générale sur l'avenir de ces contraintes *prêtes à brancher*.

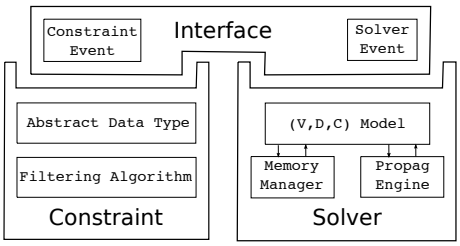

FIG. 1 – Une contrainte globale idéalement découplée de son solveur hôte.

2 Préliminaires

La programmation par contraintes (PPC) [12, 14, 20] est un domaine de recherche dont l'objectif est de fournir des outils logiciels déclaratifs et flexibles pour résoudre des problèmes combinatoires. Un problème de satisfaction de contraintes (CSP) est défini par un ensemble $\mathcal{V} = \{v_1, \ldots, v_n\}$ de variables (au sens mathématique du terme), un ensemble $\mathcal{D} = \{dom(v_1), \ldots, dom(v_n)\}$ de domaines qui représentent l'ensemble fini des valeurs possibles pour chacune des variables et un ensemble $\mathcal{C}$ de contraintes (relation logique sur un sous-ensemble des variables). Une solution pour un CSP est une affectation des variables (une valeur pour chaque variable) qui vérifie simultanément l'ensemble des contraintes du problème.

L'idée centrale de la programmation par contraintes est basée sur une technique de propagation-recherche qui consiste à explorer l'ensemble des affectations possibles pour les variables d'un CSP. Évidemment, cet espace de recherche est de taille exponentielle par nature. Ainsi, l'objectif est de détecter en amont (et donc de retirer) des portions qu'il est inutile d'explorer (car ne contenant aucune solution) de cet espace (sans évidemment explorer effectivement ces portions). Ceci est réalisé en utilisant des techniques de *filtrage* (identification et retrait de valeurs *inconsistantes* des domaines des variables en tenant compte des

contraintes du problème).

Dans ce contexte, les *contraintes globales* [6] représentent un outil indispensable de modélisation et de résolution dans le paradigme. En effet, une contrainte globale est un ensemble compact d'algorithmes et de solutions pour des sous-problèmes récurrents dans les CSP. Elles sont considérées comme des contraintes comme les autres mais généralement encapsulent un ensemble de contraintes définies sur un ensemble conséquent de variables.

Les contraintes globales offrent une vue concise et efficace du sous-problème par lequel elles sont définies. Elles utilisent de manière active la structure sous-jacente de ce sous-problème et proposent ainsi des algorithmes de filtrage particulièrement efficaces. En effet, la connaissance explicite embarquée dans cette structure permet une meilleure propagation (et donc une meilleure recherche de solution) en évitant de redécouvrir continuellement les mêmes incohérences (un tel phénomène est généralement appelé *thrashing*). Habituellement, les algorithmes mis en œuvre ont une complexité élevée mais ce surcoût est largement compensé par le pouvoir de filtrage apporté par la contrainte.

3 État de l'art

La conception, l'implémentation et la maintenance d'une contrainte globale est une tâche complexe d'un point de vue pratique. Aujourd'hui, des outils ou des concepts sont proposés pour faciliter le processus. Ainsi en est-il de la dérivation automatique d'algorithmes de filtrage pour des contraintes globales, des tentatives d'unification des architectures de solveurs ou encore des boîtes à outils pour l'implémentation de contraintes globales (ici dans le cadre des contraintes sur les graphes). Nous détaillons ces trois propositions.

La première proposition part du principe que l'implémentation d'une contrainte (globale) devrait être limitée en utilisant des propriétés de reformulation. Ainsi, les automates permettent d'exprimer la plupart des contraintes globales [3] et de produire automatiquement des vérificateurs de contraintes et des algorithmes de filtrage pour les contraintes [4] grâce à trois (méta) contraintes globales (`regular`, `cost regular` et `multi-cost regular`). Mais, toutes les contraintes ne peuvent pas être représentées par des automates et même si une telle représentation existe, le niveau de consistance atteint pour la génération automatique est fréquemment significativement plus faible qu'une approche dédiée. Une autre limitation de l'utilisation des automates est leur forte consommation mémoire. En effet, la plupart du temps, la taille de l'automate utilisé pour dériver un algorithme de filtrage n'est pas polynomial.

La seconde proposition est liée à l'architecture des solveurs. En deux mots, les solveurs de contraintes traitent la propagation selon l'une des deux philosophies suivantes : centrée sur les événements (comme dans choco) qui sont alors au cœur des files de propagation (la variable - et ce qui lui arrive - est alors au centre du système) ou centrée sur la propagation (comme dans gecode) où c'est plutôt la contrainte (ou plus précisément ses propagateurs - algorithmes directionnels de filtrage) qui est la plaque tournante du système. Une démarche intuitive introduite par [11] tente de concilier ces deux points de vue : une méthode générique (un *advisor*) permet d'interfacer de manière sûre les contraintes et les propagateurs qui lui sont liés. De cette façon, les *advisor*s représentent un moyen élégant de prendre en compte une propagation incrémentale dans les solveurs centrés sur la propagation tels que gecode. Ils peuvent être généralisés pour être considérés comme outils génériques de propagation dans les solveurs centrés sur les événements. Ainsi, une certaine souplesse est introduite dans le processus de propagation.

La troisième proposition est liée est la complexité de concevoir et d'implémenter des contraintes globales dans un domaine spécifique. Dans le cas des contraintes sur les graphes, un cadre générique a été proposé nommé `LS(Graph)`) par [8]. Ce cadre générique permet de s'affranchir de la complexité de la représentation des structures internes pour les contraintes sur les graphes. En effet, dans ce domaine, les structures de données sont clés pour obtenir des algorithmes efficaces surtout dans un contexte lié à la gestion du retour arrière. Plusieurs approches ont été proposées : une gestion *backtrackable* de la structure de données et le maintien incrémental d'une structure de données *ad hoc* [16]. On parle alors, dans ce dernier cas, d'une structure de données *pleinement dynamique*. C'est un premier pas vers une contrainte globale découplée de son solveur hôte.

Dans la suite de cet article, nous reprenons les deux dernières propositions afin de réaliser une séparation claire entre un solveur et ses contraintes. Plus précisément, nous montrons comment la notion d'*advisor* peut permettre d'interface proprement des contraintes *prêtes à brancher* avec différents solveurs. En effet, un développeur peut, dans ce contexte, spécifier précisément quand une contrainte doit être propagée selon l'observation des domaines des variables. Nous montrons aussi la complexité de rendre générique les structures de données internes des contraintes globales *prêtes à brancher* dans la gestion du retour arrière.

4 Les contraintes prêtes à brancher dans la pratique

Passer d'une contrainte dépendante du solveur à une contrainte prête à brancher nécessite de spécifier comment la contrainte considérée va interagir avec ledit solveur. Cela est généralement assez simple. En effet, en pratique, les algorithmes de résolution des contraintes gèrent les réveils

des contraintes selon les événements qui surviennent dans le domaine des variables sur lesquelles elles portent (dans le cadre des solveurs centrés événements), ou d'après des algorithmes généraux qui réalisent la propagation d'une manière donnée (pour les autres).

Dans le cadre des contraintes découplées des solveurs, les modifications dans les domaines des variables depuis le dernier réveil de la contrainte doivent être détectées et ensuite, traduites en événements gérables par la structure de données de la contrainte. De manière symétrique, nous avons à traduire les informations de filtrage de la contrainte en événements compréhensibles par le solveur (retraits de valeur, mise à jour des bornes, etc.) et applicables sur les domaines des variables qui peuvent être interprétées par l'algorithme de résolution basé sur les contraintes. Cette traduction bidirectionnelle des informations est appelée *interface* dans la suite. On peut remarquer qu'une interface peut être vue comme vision d'un niveau plus élevé des *advisors* introduits par [11], dans le sens où une interface s'occupe de diriger la stratégie des événements entre les contraintes et le solveur. Par conséquent, elle décide si une contrainte doit être propagée ou si le processus peut être retardé.

La transformation d'une contrainte globale en une contrainte prête à brancher devient complexe lorsqu'on considère des contraintes à états explicites. Pour résumer, une *contrainte à état explicite* gère sa propre structure de données afin de maintenir certaines propriétés qui sont utilisées par l'algorithme de filtrage pour retirer les valeurs inconsistantes dans les domaines des variables sur lesquelles la contrainte porte. Plus précisément, une contrainte globale s'appuie énormément sur les structures de données backtrackables du solveur hôte utilisé, et, dans le but de la rendre indépendante du solveur, nous devons gérer le retour arrière dans la contrainte explicitement. Par exemple, les contraintes arithmétiques binaires classiques ($\leq$, $=$, $\neq$) ne requièrent pas plus d'information que l'état du domaine de la variable. A l'opposé, les contraintes globales basées sur les graphes doivent modéliser une représentation du graphe dans le but de représenter efficacement et de manière expressive les propriétés liées (composantes connexes, composantes fortement connexes, nœuds dominants, arbres recouvrants, etc).

Dans la suite, nous présentons l'architecture d'un cadre d'interface pour implémenter des contraintes prêtes a brancher. Ensuite, nous illustrons une telle interface avec les contraintes `boundAllDiff` [13], qui est une contrainte globale simple (dans le sens qu'elle ne fait intervenir aucune structure de données interne), et `tree` [1], qui est une illustration typique d'une contrainte à état explicite (la structure de données interne représente un graphe orienté et des propriétés portant sur ce graphe). Finalement, nous branchons ces deux contraintes à deux solveurs différents.

4.1 Modèle des contraintes prêtes à brancher

Cette section étudie un modèle générique pour les contraintes prêtes a brancher. L'architecture est résumée par la figure 2. Elle est basée sur une décomposition en trois étapes, largement inspirée par le bien connu pattern *Observer* du génie logiciel orienté objet.

Une *Interface de Solveur* propose un ensemble de méthodes pour traduire les événements des variables du solveur en événements génériques gérables par l'interface dans la contrainte. Plus précisément, le solveur traduit les événements survenant sur les variables en événements génériques, et soumet ces événements au gestionnaire. Par exemple, supposons que le domaine d'une variable a changé, des valeurs ont été retirées (durant le filtrage ou l'énumération) ou ont été ajoutées (à cause d'un retour-arrière), alors un nouvel événement générique est généré et transmis au gestionnaire d'événements. Une *Interface de Contrainte* propose un ensemble de méthodes pour traduire les événements génériques fournis par le gestionnaire en événements compréhensibles par les structures de données internes de la contrainte. Les décisions de retrait (sur la structure interne) effectuées par les algorithmes de filtrage sont traduites de manière symétrique en événements génériques et sont renvoyés au gestionnaire. Un *Gestionnaire d'événements* qui distribue chaque événement générique. Il peut être vu comme une structure observable qui notifie les nouveaux éléments postés dans la queue d'événements génériques de l'interface de la contrainte prête à brancher correspondante ou de l'interface du solveur sous-jacent. Typiquement, si une variable du solveur a été instanciée, l'interface du solveur construit et poste un nouvel événement générique dans la queue du gestionnaire. Ensuite, le gestionnaire notifie l'interface de la contrainte que de nouveaux événements sont disponibles. De manière similaire, si l'algorithme de filtrage change la structure interne de la contrainte alors, un événement générique est produit et posté dans la queue du gestionnaire. L'interface du solveur est alors notifiée qu'un nouvel événement est disponible. Cependant, on peut remarquer que même si une traduction d'un événement du solveur en un événement contrainte est généralement direct, l'opération inverse peut être compliquée. Par exemple, dans le cadre des contraintes basées sur des graphes, les événements haut niveau du solveur peuvent être des retraits de valeurs dans les domaines des variables. De tels événements sont interprétés comme un ensemble de retraits d'arc dans la structure de données modélisant le graphe. De manière similaire, un retour-arrière consiste en un ajout d'un ensemble de valeurs dans les domaines des variables, et est interprété comme l'ajout d'un ensemble d'arcs dans la structure de données modélisant le graphe.

Dans la suite, nous ne détaillerons pas l'implémentation de notre interface générique mais nous détaillerons les différentes difficultés qui peuvent être rencontrées en fonc-

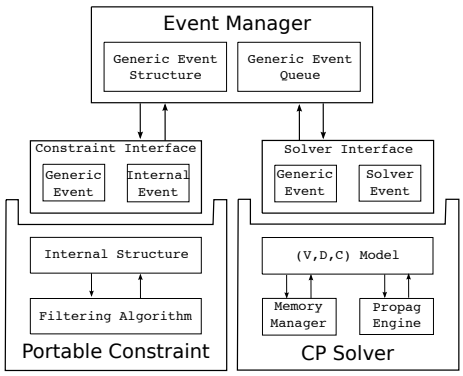

FIG. 2 – Interfaçage d'un solveur et d'une contrainte *prête à brancher* par l'intermédiaire du gestionnaire d'événements.

tion du type de contrainte et des algorithmes de résolution. Nos expérimentations sont basées sur les solveurs `Choco` et `Gecode`. Nous avons décidé volontairement de considérer deux types de contraintes globales : la contrainte `boundAllDiff` qui n'utilise aucune structure de données internes et la contrainte `tree` dont la structure de données interne est basée sur une représentation de graphe qui implique de véritables traductions des événements touchant les domaines en événements liés au graphe.

4.2 Une interface pour une contrainte globale simple : le cas `boundAllDiff`

Nous montrons d'abord, que dans le cas des contraintes globales qui ne requièrent pas de structures de données backtrackables (c.-à-d., toutes les inférences sur les domaines sont faites par un raisonnement direct sur les domaines des variables) alors, proposer une version prête à brancher de ces contraintes est facile. Basé sur l'algorithme de consistance de bornes introduit dans [13], cette section présente une implémentation prête à brancher de la contrainte `boundAllDiff`. Nous soulignons que la difficulté d'implémenter une telle contrainte prête à brancher est uniquement lié à l'interface avec le solveur.

Soit $v_1, v_2, \ldots, v_n$ un ensemble de variables avec les domaines finis $dom(v_1)$, $dom(v_2), \ldots,$ $dom(v_n)$, et des contraintes de différences deux à deux dont l'ensemble est noté `allDiff`$(v_1, \ldots, v_n)$ défini par $\{(e_1, \ldots, e_n) \mid \forall e_i \in dom(v_i), e_i \neq e_j \text{ for } i \neq j\}$. Il s'agit d'une contrainte très étudiée. Le premier algorithme de filtrage pour rendre les domaines consistants a été introduit par Régin dans [15]. Ici, nous rappelons l'algorithme de filtrage propageant la consistance aux bornes introduit dans [13]. Afin de fournir des approches de filtrage rapides et pouvant s'appliquer à des problèmes de grande taille, plusieurs algorithmes de filtrage (qui atteignent un niveau de consistance plus faible que la consistance de domaine) ont été introduits. Le plus rapide est l'algorithme de consistance aux bornes, `boundAllDiff`. Intuitivement, au plus n variables peuvent avoir leur domaine inclus dans un intervalle contenant n valeurs. Dans un intervalle de Hall (un intervalle I de taille n, tel qu'il y ait n variables dont le domaine est contenu dans I), une solution utilisera toutes les valeurs pour ces variables rendant ses valeurs inutilisables pour les autres variables. Ainsi, pour maintenir la consistance aux bornes, nous devons vérifier pour chaque intervalle I qu'il y a, au moins, autant de valeurs que de variables dont le domaine est inclus dans I. Et pour chaque intervalle de Hall I, nous devons interdire toutes les valeurs de I dans les autres variables.

Definition 1 (`boundAllDiff` – consistance) *Une contrainte `boundAllDiff`, portant sur les variables $(x_1, \ldots, x_n)$, est consistante aux bornes si et seulement si les conditions suivantes sont vérifiées : (1) $|D_i| \geq 1 (i = 1, \ldots, n)$, (2) pour chaque intervalle I : $|K_I| \leq |I|$, et (3) pour chaque intervalle de Hall I, $\{min(x_i), max(x_i)\} \cap I = \emptyset$ for all $x_i \notin K_I$.*

Pour chaque réveil, la contrainte `boundAllDiff` trie les variables et initialise plusieurs compteurs, ainsi aucune structure de données n'est maintenue entre deux appels de la procédure de filtrage. Les informations requises par la contrainte est l'ensemble des variables avec les valeurs maximum et minimum de chaque domaine. Alors, pour être capable de brancher la contrainte `boundAllDiff` dans différents solveurs, l'interface doit fournir une vue générique des variables. Ainsi, pour une contrainte `boundAllDiff` prête à brancher, le cadre générique nécessite l'implémentation de : **(1)** L'interface du solveur donne le minimum et le maximum du domaine de chaque variable et, envoie une vue de ces domaines au gestionnaire d'événements. Symétriquement, les événements de la contrainte envoyés par le gestionnaire d'événements sont traduits en événements de solveur et sont finalement convertis en modification de domaine des variables. **(2)** Le gestionnaire d'événements peut être vu ici comme une fonction bijective qui convertit les événements solveur en événements contrainte et vice-versa. **(3)** L'interface contrainte contient l'algorithme de filtrage `boundAllDiff`. Les événements contrainte générés par l'algorithme de filtrage consistent uniquement en un ensemble de modifications portant sur les valeurs minimale et maximale des vues des variables dans la structure de la contrainte.

La complexité de l'interface dépend des informations requises par la contrainte et des informations disponibles dans l'algorithme de recherche. Généralement, les valeurs maximale et minimale pour une variable sont accessibles en temps constant, aussi l'interface a une complexité de $O(n)$. De plus, la structure de données et l'algorithme de filtrage sont exactement les mêmes dans la version prête à brancher ou dans la version *ad hoc* de la contrainte.

4.3 Le cas des contraintes globales à états : la contrainte tree

Dans le cas de contraintes globales qui reposent sur une structure de données backtrackable (quelques inférences ont besoin de détecter de manière répétée un motif caché dans les domaines des variables) alors, proposer une version prête à brancher de ces contraintes est plus délicat, et des algorithmes *pleinement dynamiques* sont nécessaires pour fournir une contrainte indépendante du solveur. De plus, comme une contrainte prête à brancher n'est pas dépendante du solveur hôte, un choix plus large de structures de données est offert. Cette liberté permet de choisir et d'implémenter la plus efficace. Cette section illustre ce point avec la contrainte `tree` [1].

La contrainte `tree` est représentée par un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ dans lequel les nœuds représentent les variables, $\mathcal{V} = \{v_1, \ldots, v_n\}$, les arcs représentent la relation de succession directe entre eux, $dom(v_i) = \{j \mid (v_i, v_j) \in \mathcal{E}\}$, et `ntree` est une variable qui spécifie le nombre d'arbres dans la forêt (<u>`ntree`</u> et $\overline{\texttt{ntree}}$ correspondent respectivement à la valeur minimal et à la valeur maximale de $dom(\texttt{ntree})$). Une contrainte $\texttt{tree}(\texttt{ntree}, \mathcal{G})$ spécifie que le graphe orienté associé $\mathcal{G}$ devrait être une forêt de `ntree` arbres, formellement :

Definition 2 *Une* ground instance *de la contrainte* $\texttt{tree}(\texttt{ntree}, \texttt{VER})$ *est une* solution *si et seulement si (1) le graphe orienté associé $\mathcal{G}$ est composé de* `ntree` *composantes connexes, et (2) chaque composante connexe de $\mathcal{G}$ n'a pas de circuit impliquant plus d'un nœud (notons que chaque composante contient exactement un nœud possédant une boucle et qui correspond à la racine de l'arbre).*

Étant donné un graphe orienté $\mathcal{G}$ et un ensemble de *de racines potentielles*[2], le nombre minimal d'arbres (<u>`ntree`</u>) pour partitionner le graphe orienté $\mathcal{G}$ associé à une contrainte `tree` est le nombre de puits (c.-à-d. de nœuds sans arc sortant exceptée la boucle sur lui-même) du graphe réduit[3] de $\mathcal{G}$, et le nombre maximum d'arbres ($\overline{\texttt{ntree}}$) pour partitionner le graphe orienté $\mathcal{G}$ est le nombre de racines potentielles.

Ensuite, nous détaillons l'algorithme de filtrage de la contrainte `tree`. Lorsque la variable `ntree` doit atteindre la valeur $\overline{\texttt{ntree}}$, l'algorithme force, pour chaque racine potentielle, l'arc boucle (qui représente le fait qu'il s'agit d'une racine potentielle). Dans le cas où `ntree` doit atteindre <u>`ntree`</u>, l'algorithme retire, pour chaque racine potentielle qui n'appartient pas à une composante fortement connexe puits de $\mathcal{G}$, la boucle sur lui-même. Finalement, pour n'importe quel `ntree` la règle principale de filtrage associée avec la contrainte est basée sur la détection des *noeuds dominants* du graphe[4]. Alors, l'algorithme de filtrage doit détecter tous les nœuds de j de $\mathcal{G}$ tel qu'il existe un nœud i pour lequel j domine toutes les racines potentielles de $\mathcal{G}$ par rapport à i. Les arcs infaisables dans $\mathcal{G}$ pour une contrainte `tree` sont les arcs sortant (j, k), où j est un nœud dominant, tel qu'il n'existe pas un chemin de i à une racine potentielle de $\mathcal{G}$ utilisant l'arc (j, k).

Figure 3 décrit le squelette de la contrainte `tree` telle qu'elle a été implémentée dans le solveur `choco`. Ici, les effets des événements survenant sur les variables du graphe consistent en plusieurs modifications de la structure du graphe. Par exemple, si un arc (i, j) est retiré de $\mathcal{G}$ (c.-à-d., $j \notin dom(v_i)$) alors, ce retrait peut : **(1)** diminuer le nombre de racines potentielles, si $i = j$. Cela mène à une mise à jour de $\overline{\texttt{ntree}}$. **(2)** augmenter le nombre de composantes puits. Cela mémé a une augmentation de <u>`ntree`</u>. **(3)** augmenter le nombre de composantes fortement connexes (cfc) dans $\mathcal{G}$. Cela mène à modifier le graphe réduit $\mathcal{G}_r$ associé avec $\mathcal{G}$. **(4)** créer un nouveau nœud dominant dans $\mathcal{G}$.

Si un retrait de valeur dans une variable (`ntree` ou une variable décrivant un nœud du graphe) impliqué dans la contrainte `tree` mène a un domaine vide il est nécessaire de revenir dans un état antérieur consistant. Pour ce faire, il est possible d'utiliser les structures de données *backtrackables* fournies par le solveur de contraintes utiliséÈ [5]. L'implémentation initiale de la contrainte `tree` utilisait ces structures de données afin d'enregistrer dynamiquement et restaurer les propriétés de graphe comme les composantes fortement connexes et les nœuds dominants du graphe orienté $\mathcal{G}$. Ainsi, l'utilisation de ces structures de données backtrackables dédiées fait que la contrainte `tree` initiale est dépendante du solveur.

L'état de l'art classique des algorithmes de graphe propose plusieurs algorithmes incrémentaux permettant de maintenir les propriétés de graphe impliquées dans la contrainte `tree` comme les composantes fortement connexes et la fermeture transitive [9], ou les nœuds dominants [19]. Deux questions demeurent : est-il vraiment nécessaire d'utiliser des structures de données backtrackables lorsque des algorithmes incrementaux en ajout et en retrait existent ? Si aucune structure backtrackable n'est utilisée (c.-à.-d. qu'il n'est pas nécessaire de *trailer* les modifications des structures de données utilisées dans la contrainte),

[2] Une *racine potentielle* d'un graphe orienté $\mathcal{G}$ est un nœud qui peut potentiellement être racine d'un arbre dans une solution satisfaisant la contrainte `tree`.

[3] Le *graphe réduit* $\mathcal{G}_r$ est dérivé d'une graphe orienté donné $\mathcal{G}$ en associant à chaque composante fortement connexe, un nœud de $\mathcal{G}_r$ et à chaque arc de $\mathcal{G}$ qui connecte différentes composantes fortement connexes, correspond un arc dans $\mathcal{G}_r$.

[4] Étant donné un graphe orienté $\mathcal{G}$, un nœud j *domine* un nœud k par rapport à un nœud i si et seulement si n'importe quel chemin de i à k atteint le nœud j avant le nœud k

[5] Le solveur Choco propose plusieurs structures *backtrackables* utilisant des entiers, des booléens et des bitsets. Ils sont basés sur une approche de *trailing* [18] qui enregistre, pour chaque événement modifiant la structure de données, les informations nécessaires pour défaire ses effets.

Initial awakening of the `tree` constraint :
– Compute $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
– If there is at least one solution satisfying the constraint then, do propagation related to the constraint :
  1. Update `ntree` according to $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
  2. Propagate according to the dominator nodes of $\mathcal{G}$.
  3. Propagate according to the values of $\max(\texttt{ntree})$ and $\min(\texttt{ntree})$.

Each time an event occurs on a domain variable of the `tree` constraint do :
– If this event occurs on a domain variable modeling `ntree` then :
  1. Update `ntree` according to $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
  2. Propagate according to $\max(\texttt{ntree})$ and $\min(\texttt{ntree})$.
– If this event occurs on a domain variable modeling a node of $\mathcal{G}$ do :
  1. Update `ntree` according to $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
  2. Propagate according to new dominator nodes of $\mathcal{G}$.

FIG. 3 – Squellete d'implémentation de la contrainte `tree`au sein de *choco*.

quelles sont les relations entre les contraintes et le solveur ?

Le goulot d'étranglement de la complexité de la contrainte `tree` est lié au fait de maintenir de manière répétée plusieurs propriétés de graphe (composantes fortement connexes – cfc, fermeture transitive, nœud dominant) du graphe orienté $\mathcal{G}$ entre deux étapes de recherche. De manière basique, une nouvelle approche implémentant une telle contrainte et respectant la séparation des préoccupations introduite dans la figure 2, peut être décomposée de la manière suivante : premièrement, *l'interface de la contrainte* est décomposée en une structure de la contrainte et en un algorithme de filtrage. La *structure de la contrainte* est basée sur une structure de données générique, incrémentale en ajout et en retrait, qui modélise le graphe orienté $\mathcal{G}$ et ses propriétés associées (c.-à.-d., cfc et la fermeture transitive). Cette partie contient les primitives permettant de mettre à jour la structure de données en fonction des retraits et des ajouts d'arcs. Ces primitives, pour résumer, calculent chaque propriété en ne considérant qu'un graphe partiel nécessaire[6] du graphe d'origine $\mathcal{G}$. *L'algorithme de filtrage*, basé sur les propriétés maintenues par le graphe (représenté par la structure de la contrainte), retire les arcs de $\mathcal{G}$ qui sont inconsistants avec la contrainte `tree`. Deuxièmement, *l'interface du solveur* et le *gestionnaire d'événements* qui assurent une relation bidirectionnelle entre les événements survenant sur le domaine des variables (c.-à-d. retraits/restaurations des valeurs dans les domaines) et des événements survenant dans le graphe orienté $\mathcal{G}$ (retrait/ajout d'arcs).

Dans l'implémentation actuelle de la contrainte `tree` prête à brancher, chaque fois qu'un événement survient dans le domaine d'une variable, cet événement est d'abord interprété par l'interface du solveur pour produire un événement interne. Ensuite, la structure de données interne est mise à jour avec cet événement interne. Alors, l'algorithme de filtrage dédié à la contrainte `tree` est appliqué et les événements internes résultants sont traduits en événements génériques et sont envoyés au gestionnaire d'événements. Nous pouvons maintenant parler du gestionnaire d'événements. À partir des domaines des variables impliquées dans un CSP, généralement, quatre types d'événements peuvent être distingués : le retrait de valeurs dans les domaines, l'instanciation de variables, la mise à jour d'une borne inférieure et la mise à jour d'une borne supérieure. Cependant, les instanciations et les mises à jour de bornes peuvent facilement être décomposées en un ensemble de retraits. Ainsi, pour chaque type d'événement envoyé par l'algorithme de recherche à l'interface du solveur, une traduction de l'événement en un ensemble de retraits dans la structure de la contrainte est réalisée. Cependant, tous les retraits d'arcs ne sont pas gérés de la même manière par le gestionnaire d'événements. En effet, l'événement à l'origine de l'ensemble des retraits est considéré afin d'améliorer l'efficacité des mises à jour de la structure de la contrainte. Par exemple, un ensemble de retraits liés a un événement d'instanciation survenant sur une variable entraîne une modification locale dans le voisinage du nœud correspondant dans le graphe orienté $\mathcal{G}$ associé avec la contrainte. Cette information peut être prise en compte dans le but de réaliser ces modifications efficacement.

5 Évaluation

Dans cette partie, nous nous proposons d'évaluer le comportement des contraintes *prêtes à brancher*. Toutes les expérimentations ont été réalisées avec `choco` (version $1.2.04$) et `gecode` (version $1.3.1$) sur un processeur Intel Xeon cadencé à 2.4GHz et possédant 1Go de RAM, mais dont seulement 128Mo alloués à la machine virtuelle java.

Cette évaluation est réalisée en deux étapes. La première (section 5.1), évalue le sur-coût engendré par le découplage. Pour cela, nous nous focalisons sur la contrainte `boundAllDiff` dans le contexte des n reines (tableau 1). Puis, la contrainte `tree` montre la portabilité effective de sa version *prête à brancher* pour les solveurs `Choco` et `Gecode` (tableau 2). La deuxième étape (section 5.2), évalue le comportement des contraintes *prêtes à brancher* dans un environnement hétérogène (on retrouve à la fois des

[6] Étant donné un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, un *graphe partiel* $\mathcal{G}'$ de $\mathcal{G}$ est défini par $(\mathcal{V}' \subseteq \mathcal{V}, \{(i,j) \in \mathcal{E} \mid i,j \in \mathcal{V}'\})$

boundAllDiff	nReines					
	25	50	75	100	150	200
prête à brancher	22	35	1538	2461	5741	14217
ad-hoc	19	35	1531	2435	5711	14006

TAB. 1 – Comparaison des temps de calcul (ms) pour les versions prête à brancher et *ad hoc* de la contrainte `boundAllDiff`.

contraintes *ad hoc* et des contraintes *prêtes à brancher* dans le modèle). Pour cela, nous nous intéressons au problème de chemin hamiltonien (tableau 3).

5.1 Sur-coût lié à la portabilité

Nous évaluons le sur-coût d'interfaçage des contraintes `boundAllDiff` et `tree` avec deux solveurs de contraintes : `Choco` et `Gecode`. L'interface traduit une information liée au solveur en des informations génériques. L'implémentation de cette interface dépend du niveau d'information fourni par le solveur. `Choco`, centré événement, est capable de fournir aux contraintes les modifications ayant provoqué leur réveil. `Gecode`, quant à lui centré propagateur, ne peut fournir cette information. Pour la contrainte `tree` branchée sur `Choco`, l'interface présente une complexité temporelle directement liée au nombre de modifications de domaines réalisées tandis que pour `Gecode`, l'interface doit passer en revue le domaine de chaque variable pour identifier les modifications. Dans le cas de `boundAllDiff`, l'interface n'a uniquement besoin de connaître les valeurs maximale et minimale des domaines des variables aussi bien pour l'un que l'autre des solveurs.

Nous résolvons les n reines à 25, 50, 75, 100, 150 and 200 reines. Le problème est modélisé à l'aide de 3 contraintes `boundAllDiff` (en version prête à brancher d'une part et en version *ad hoc* d'autre part). Cette contrainte ne gère aucune structure de données interne, ainsi le sur-coût pour la version prête à brancher est minimal. Pour 200 reines, le temps passé dans les trois modules d'interface (une interface par contrainte) ne représente que 1.5% du temps global de calcul (tableau 1).

On peut noter qu'une contrainte *prête à brancher* peut-être utilisée avec n'importe quel problème. Le tableau 2 montre les détails du temps de calcul utilisé par les différentes composantes de la contrainte `tree`, tant pour `Choco` que pour `Gecode`. Pour des graphes d'ordre$\{25, 50, 75, 100, 150, 200\}$, et de densité $\{0.05, 0.2, 0.4, 0.5, 0.6, 0.8, 0.95\}$, 50 instances d'un problème de partionnement de graphe sont générées (soit au total 2100 graphes). Notons que les deux solveurs utilisent ici la même heuristique de choix de variable. La colonne "Interface" du tableau 2 montre parfaitement le sur-coût engendré par l'interface dans Gecode. On note aussi ce que le temps passé dans la contrainte elle-même est le même dans les deux solveurs : c'est ce que montrent les colonnes "Contrainte" et "Structure".

5.2 Comportement en pratique

Cette section résume les résultats expérimentaux des contraintes prêtes à brancher utilisées pour résoudre un problème de chemin hamiltonien. Cette évaluation montre l'efficacité des contraintes prêtes à brancher dans un contexte pratique. Pour cela, le problème de chemin hamiltonien est modélisé à l'aide des contraintes `boundAllDiff` et `tree`.

L'évaluation est réalisée en observant toutes les combinaisons possibles (ad hoc/prête à brancher) des contraintes `tree` et `boundAllDiff` avec le solveur choco. Pour chaque ordre de graphe dans $\{25, 50, 75, 100, 150, 200\}$, et pour des densités $\{0.10, 0.25, 0.40, 0.50, 0.65, 0.75, 0.90\}$, 50 instances sont générées (soit globalement 2100 graphes). Ici, un timeout de 5 minutes a été fixé et une heuristique aléatoire de choix de variable a été utilisée.

Les résultats présentés dans le tableau 3 montrent qu'utiliser des contraintes prêtes à brancher n'a pas d'impact sur l'efficacité, même en cas de combinaison avec des contraintes *ad hoc*. Même, on peut voir que la version prête à brancher de la contrainte `tree` est bien meilleure que la version *ad hoc*. Ceci est principalement dû à la nature complètement incrémentale de cette contrainte (aucune structure backtrackable n'est utilisée). Enfin, la contrainte `boundAllDiff` dans sa version prête à brancher est équivalente à sa version *ad hoc*.

6 Discussion

Cet article tente d'identifier les limites du découplage d'une contrainte globale du solveur sous-jacent. Nous avons montré dans les paragraphes précédents qu'il n'y a pas de frein majeur à un tel découplage. En fait, cette question revient à lever deux interrogations :

6.1 Est-ce que l'implémentation d'une contrainte prête à brancher est difficile ?

Les contraintes globales sont généralement implémentées à l'aide des structures dédiées et de l'API d'un solveur donné. Ces contraintes, le plus souvent, embarquent des structures de données persistantes utilisées pour maintenir leur propre niveau de consistance pendant la recherche de solution. En réalité, cette interférence du solveur avec l'architecture interne de la contrainte est plutôt une limitation pour l'efficacité et l'expressivité d'une implémentation. Par exemple, les nouveautés proposées par les langages ne sont pas, la plupart du temps, directement utilisables étant donné le temps de latence induit par le solveur lui-même (par exemple, les *generics* en Java).

Ordre du graphe	temps Choco (ms)			temps Gecode (ms)		
	Interface	Contrainte	Graphe	Interface	Contrainte	Structure
25	5	10	27	6	10	27
50	5	57	229	24	56	228
75	11	190	863	58	186	879
100	18	440	2287	120	439	2310
150	50	1466	9524	368	1329	9596
200	82	3214	27551	812	3009	27097

TAB. 2 – Temps de calcul pour `tree` branchée sur les deux solveurs (Choco et Gecode).

Ordre du graphe	contrainte `tree` ad hoc		contrainte `tree` à brancher	
	ad hoc `boundAllDiff`	à brancher `boundAllDiff`	ad hoc `boundAllDiff`	à brancher `boundAllDiff`
25	57	58	56	52
50	5529	5577	6224	6235
75	10858	10745	10311	10126
100	19201	19267	14956	14855
150	60683	60612	27271	27380
200	156055	155793	45099	45919

TAB. 3 – Résolution du problème de chemin hamiltonien combinant contraintes ad hoc et prêtes à brancher (ms).

Ainsi, dans le contexte des contraintes prêtes à brancher la question des structures de données internes n'en est plus une car toute latitude est laissée au développeur. Par contre, la vraie question est la manière de mettre à jour la représentation interne de la contrainte face aux événements fournis par le solveur. Une réponse satisfaisante doit prendre en compte la complexité du maintien des propriétés maintenues par la représentation interne. Intuitivement, trois cas peuvent être distingués : dans le premier cas, les propriétés sont calculées à chaque appel de la contrainte sans aucun effet mémoire (comme dans la section 4.2). Ainsi, après chaque modification les propriétés sont recalculées et la contrainte peut les exploiter ; dans le deuxième cas, les propriétés ne peuvent être recalculées de manière efficace mais il existe des algorithmes complètement incrémentaux qui permettent de prendre en compte les modifications (cas de la section 4.3). Dans le cas des propriétés de graphe, par exemple, de nombreux travaux existent pour prendre en compte dynamiquement des modifications de structures. Ainsi, [17] propose un algorithme de maintien d'un arbre recouvrant de poids minimal devant des ajouts/retraits d'arêtes ; dans le dernier cas, il est nécessaire d'embarquer dans la contrainte des mécanismes explicites de retour arrière (selon diverses méthodes : trailing, copie, recalcul). Là, il est tout de même nécessaire d'être capable de savoir depuis la contrainte si un retour arrière a eu lieu ou non.

6.2 Est-ce que réaliser l'interface pour une contrainte prête à brancher est difficile ?

Les solveurs manipulent des événements sur les variables décrivant des modifications des domaines : retraits de valeur(s), mise à jour de borne(s), etc. Ces événements sont compris et traités par toutes les contraintes portant sur les variables du solveur. Mais, pour les contraintes prêtes à brancher, ces événements sont généralement vides de sens *per se*. C'est pourquoi l'interface doit les traiter pour transformer ces événements du solveur en événements sur les structures de données internes de la contrainte.

Dans la première approche de découplage, les événements du solveur peuvent traduit en *quelque chose a changé*. Cette information permet de recalculer les propriétés à maintenir.

Dans la seconde approche, les événements du solveur sont transformés en des mises à jour de la structure de données interne traitées ensuite par les algorithmes incrémentaux fournis par la contrainte. Ces mises à jour dépendent du type de modifications que sont capables de traiter les algorithmes en question. Par exemple, dans la section 4.3, les événements du solveur ont été transformés en des retraits (ou ajouts) d'arêtes dans le graphe de référence.

Enfin dans la dernière approche, les événements du solveur sont transformés en des événements compris par la structure backtrackable générique utilisée.

La limite principale au découplage des contraintes globale n'est pas une question de génie logiciel mais plutôt une question d'interopérabilité des langages de programmation utilisés pour implémenter les contraintes. Par exemple, les contraintes de choco sont développées en Java, celles de Gecode en C++, celles de Comet en Comet et les contraintes de Sicstus Prolog en C. Évidemment des outils existent pour traiter ces problématiques. Par exemple, nous avons utilisé JNI (Java Native Interface) pour interfacer nos contraintes prêtes à brancher avec Gecode (à travers le module Gecode/J). JNI permet à du code Java tournant sur une machine virtuelle d'appeler et d'être appelé par d'autres applications écrites dans d'autres langages de programmation. Les coûts engendrés par une telle interface supplémentaire sont liés au type d'arguments pas-

sés entre les différent langages. Mais, en faisant attention, les applications Java peuvent appeler du code natif de manière relativement efficace. De plus, le sur-coût de JNI devient négligeable quand le code natif est fort consommateur de cpu.

Dans le cas de notre interface, les paramètres sont des messages d'événements (sur les variables) ainsi JNI est une solution envisageable. Par contre, pour une application codée en C, appeler une fonction Java est peu efficace car il est alors nécessaire d'utiliser des mécanismes réflexifs dans Java [23][7]. Il existe d'autres approches pour assurer l'interopérabilité des langages : communication par TCP/IP ou par IPC (inter-process communication) par exemple. Les applications Java en particulier peuvent utiliser les technologies objet distribuées comme l'API IDL[8].

7 Conclusion

Cet article s'est proposé d'étudier les liens entre les contraintes et les solveurs dans lesquels elles sont implémentées pour montrer qu'un découplage est généralement possible. Un tel découplage permet d'utiliser des contraintes globales complexes (assez difficiles à implémenter) avec différents solveurs. Nous avons montré qu'une limitation reposait sur la gestion de la mémoire pour le retour arrière tant sur les domaines des variables que pour les structures de données internes manipulées par les contraintes. Nous avons proposé une discussion sur la manipulation d'une mémoire distribuée pour les structures de données de chaque contrainte. En particulier, nous avons montré que dans le cas de la contrainte `tree` et plus généralement quand des algorithmes complètement incrémentaux existent, un tel découplage est efficace.

De plus, pour une contrainte globale donnée, un découplage entre filtrage et structure de données interne est un point central qui permet une véritable séparation des préoccupations (importante d'un point de vue génie logiciel). En effet, améliorer un algorithme de filtrage est différent d'améliorer la gestion des structures de données. Mais, on se rend compte que pour beaucoup de contraintes c'est ce qui fait la différence (en particulier pour les contraintes sur les graphes comme la contrainte `tree`). Enfin, nous montrons que le frein principal au découplage des contraintes globales n'est pas uniquement un problème de génie logiciel mais reste au niveau de l'interopérabilité des langages de programmation.

Nos travaux futurs concernent la mise à disposition d'un ensemble de contraintes globales sur les graphes et sur un ensemble d'interfaces pour gérer efficacement l'intéropérabilité entre langage (avec une focalisation sur Java et C/C++). Bien sûr, toutes les contraintes d'un système n'ont pas nécessairement vocation à être découplées (par exemple, les contraintes arithmétiques ou toute contrainte ne nécessitant pas le maintien d'un état interne). L'objectif à long terme de cette réflexion est la promotion de la notion de contrainte globale en dehors du strict cadre de la programmation par contraintes classique en en proposant l'utilisation, par exemple, dans des stratégies de recherche locale ou des solveurs à base d'explications.

[7] `http://java.sun.com/docs/books/performance/1st_edition/html/JPNativeCode.fm.html`

[8] `http://java.sun.com/docs/books/jni/html/intro.html#1811`

Références

[1] N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *CP-AI-OR'05*, volume 3524 of *LNCS*, pages 64–78, 2005.

[2] N. Beldiceanu, X. Lorca, and P. Flener. Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4), 2008.

[3] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future. *Constraints*, 12(1) :21–62, 2007.

[4] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In *CP'04*, volume 3258 of *LNCS*, pages 107–122, 2004.

[5] Nicolas Beldiceanu, Mats Carlsson, Emmanuel Poder, R. Sadek, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic *k*-dimensional objects. In *CP'07*, LNCS, pages 180–194, 2007.

[6] C. Bessière and P. van Hentenryck. To Be or Not to Be... a Global Constraint. In *CP'03*, LNCS, pages 789–794, 2003.

[7] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. The range and roots constraints : Specifying counting and occurrence problems. In *IJCAI*, pages 60–65, 2005.

[8] Pham Quang Dung, Y. Deville, and P. van Hentenryck. Ls(graph) : A local search framework for constraint optimization on graphs and trees. In *SAC*, March 2009.

[9] D. Eppstein, Z. Galil, and G. Italiano. *Dynamic graph algorithms*. CRC Press, 1997.

[10] Narendra Jussien and Vincent Barichard. The PaLM system : explanation-based constraint programming. In *Proceedings of TRICS workshop of CP*, pages 118–133, Singapore, September 2000.

[11] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In *CP'07*, LNCS, pages 409–422, 2007.

[12] J-L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10 :29–127, 1978.

[13] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint. In *IJCAI*, pages 245–250, 2003.

[14] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artif. Intell.*, 28(2) :225–233, 1986.

[15] J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.

[16] Jean-Charles Régin. Maintaining arc consistency algorithms during the search without additional space cost. In *CP*, volume 3709 of *LNCS*, pages 520–533, 2005.

[17] Jean-Charles Régin. Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In *CPAIOR'08*, LNCS, pages 233–247, 2008.

[18] Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In *ICLP'99*, pages 275–289, 1999.

[19] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Incremental Computation of Dominator Trees. *ACM Transactions on Programming Languages and Systems*, 19(2) :239–252, March 1997.

[20] P. van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, 1989.

[21] Pascal van Hentenryck and Laurent Michel. Growing COMET. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, *Trends in Constraint Programming*, chapter 17, pages 291–297. ISTE, 2007.

[22] W. J. van Hoeve. The alldifferent constraint : A survey. *CoRR*, cs.PL/0105015, 2001.

[23] Steve Wilson and Jeff Kesselman. *Java Platorm Performance : Strategies and Tactics*. Addison-Wesley, 2000.