Structures de données persistantes pour le backtracking et le parallélisme en PPC

Julien Vion
LAMIH CNRS UMR 8201, UVHC, 59313 Valenciennes
julien.vion@univ-valenciennes.fr

Résumé

Nous présentons les structures de données utilisées par le solveur *Concrete* pour représenter l'état d'un problème au cours de la recherche. L'utilisation de structures purement fonctionnelles permet de réaliser très facilement un backtracking non chronologique ou une résolution en parallèle.

Abstract

We present the data structures used by the *Concrete* constraint solver to represent the state of a problem during search. The use of purely functional data structures enable us to implement non-chronological backtracking and parallel resolution easily.

Les *structures de données persistantes* [4] permettent de conserver les anciennes versions d'elles-mêmes lorsqu'elles sont modifiées. Elles sont fortement apparentées aux structures de données *immutables*, c'est-à-dire que leur contenu ne peut pas être modifié. À la place, lors d'une opération de modification (l'ajout d'un élément, par exemple), une nouvelle structure de données est créée, contenant les données mises à jour. La structure de données persistante la plus simple est la liste chainée, représentée par une paire $(\textit{tête}, \textit{queue})$, la tête étant le premier élément de la liste et la queue la liste des autres éléments, ou la liste vide $\langle\rangle$. Prenons l'exemple de la liste chainée $xs = \langle 1, 2, 3\rangle = (1, (2, (3, \langle\rangle)))$. Ajouter la valeur 0 en tête de la liste ne nécessite pas de dupliquer d'information, mais simplement de créer une nouvelle paire $xs' = (0, xs)$. La liste xs elle-même n'est pas modifiée et peut toujours être accédée. La plupart des structures de données arborescentes peuvent être facilement adaptées pour être persistantes.

Ces structures de données ont été grandement développées dans les années 1990 conjointement au développement des langages fonctionnels [10]. Les langages fonctionnels modernes comme Haskell, OCaml, Scala ou Clojure incluent tous une librairie de structures de données persistantes dont les performances sont comparables aux structures traditionnelles.

Les structures persistantes permettent d'implanter très facilement un backtracking. Par exemple, pour effectuer une recherche en profondeur d'abord (e.g., MAC), on maintient simplement une pile contenant les différentes versions des structures de données à chaque niveau de l'arbre de recherche. Un autre avantage des structures immutables est qu'elles permettent d'implémenter très facilement un backtracking non chronologique, ou encore une parallélisation des algorithmes de recherche : il n'est plus nécessaire de synchroniser l'accès aux informations si celles-ci ne peuvent pas être modifiées. Les stratégies de type *Embarassingly Parallel Search* [11] ou autres cousines du *MapReduce* nécessitent de dupliquer l'état complet du problème pour chaque sous-problème à évaluer. Une structure persistante permet de ne mémoriser que les éléments modifiés de chaque sous-problème.

Le solveur *Concrete* [13] est entièrement développé en Scala. Dans cet article, nous présentons les structures de données choisies pour représenter les domaines des variables et les états des contraintes. Il s'agit d'une première ébauche de classification de diverses structures de données, et nous espérons que ce retour d'expérience sera utile aux développeurs de solveurs désireux d'exploiter au mieux les fonctionnalités apportées par les langages fonctionnels.

1 Représentation du problème

Un problème de satisfaction de contraintes est généralement représenté par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, avec $\mathcal{X} = \{X_1, \ldots, X_n\}$ les variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ les domaines de définition de ces variables et $\mathcal{C} =$

$\{C_1, \ldots, C_e\}$ les contraintes. D_i est le domaine de la variable X_i. $(\mathcal{X}, \mathcal{C})$ forme un hypergraphe dont les variables sont les sommets et les contraintes les arêtes. On note vars(C) les variables impliquées par une contrainte C. Chaque contrainte définit l'ensemble des instanciations autorisées des variables qu'elle implique. On associe à chaque contrainte un *propagateur* qui détecte des valeurs qui n'apparaissent dans aucune instanciation autorisée [5]. Certains propagateurs peuvent détecter que toutes les instanciations des variables sont autorisées : la contrainte est alors dite « universelle » ou *entailed* et peut être désactivée.

Le modèle de résolution général des CSP est une recherche en profondeur d'abord. À chaque nœud de l'arbre, on effectue une hypothèse qui réduit le domaine d'une voire plusieurs variables, puis on effectue un filtrage en exécutant les propagateurs des contraintes jusqu'au point fixe. Pour des raisons de performances, les propagateurs sont généralement incrémentaux : en effet, tout au long d'une branche de l'arbre de recherche, chaque propagateur est appelé de très nombreuses fois. À chaque appel, les domaines des variables ont uniquement « diminué », c'est-à-dire qu'ils ont perdu des valeurs. Il est généralement possible d'utiliser cette propriété pour améliorer grandement les performances, voire la complexité des algorithmes de propagation [9, 2]. En général, cela nécessite d'associer un *état* à chaque contrainte, qui devra être maintenu au cours de la recherche.

Pour réaliser la recherche en profondeur d'abord, il faut être capable de *backtracker*, c'est-à-dire annuler une hypothèse et ses conséquences quand elle conduit à une contradiction. Pour ce faire, le solveur *Concrete* maintient une pile d'états au cours de la recherche. Chaque état de problème contient trois ensembles d'informations : • le contenu du domaine de chaque variable, • l'état de chaque contrainte si nécessaire, et • pour chaque variable, l'ensemble des contraintes actives qui l'impliquent.

2 Représentation des domaines

Concrete supporte les domaines basés sur les booléens et les entiers relatifs bornés. La plus petite valeur d'un domaine D est notée $\underline{D}$ et la plus grande $\overline{D}$. Les domaines booléens (également utilisés pour représenter les domaines entiers inclus dans $\{0, 1\}$) ne présentent pas de difficulté particulière. Un domaine booléen ne peut prendre que quatre états : *vide*, *vrai*, *faux* ou *indéterminé*. Chacun est représenté par un objet singleton immutable.

Sur les autres domaines entiers, les deux opérations les plus importantes sont 1. l'énumération des éléments à partir d'une valeur arbitraire, et 2. le filtrage selon un prédicat. Ces deux opérations suffisent à implanter un algorithme de filtrage général comme AC-3 ou une de ses variantes. Le contrôle très rapide de la présence d'une valeur ou la possibilité de retirer une valeur arbitraire est également souvent utile pour implanter des propagateurs spécifiques. Toute structure traditionnellement utilisée pour réaliser un ensemble mathématique, comme une table de hash, pourrait convenir. Les solveurs de contraintes utilisent souvent une structure de données backtrackable par *« trailing »* [7] ou un *Sparse Set* [3] pour implémenter de tels domaines. Ces structures ne sont que partiellement persistantes : il n'est pas possible de modifier une ancienne version de la structure de manière transparente. Elles ne peuvent pas être utilisées dans le cadre du parallélisme.

Pour réaliser un filtrage aux bornes, la possibilité d'effectuer une « coupe », i.e., supprimer rapidement tous les éléments supérieurs ou inférieurs à une valeur donnée est également importante. Cette observation conduit à observer les domaines comme des ensembles ordonnés. Les ensembles ordonnés sont traditionnellement implémentés sous forme d'arbres équilibrés, mais dans ce cas le test de présence et la suppression d'un élément sont en $O(\log |D|)$, et le filtrage en $O(|D| \log |D|)$.

Les opérations d'union et d'intersection entre deux domaines ou ensembles permettent d'exprimer facilement de nombreuses propriétés de consistance. Par exemple, on peut considérer la recherche du support d'une valeur $a \in \mathrm{dom}(X)$ dans la contrainte C portant sur (X, Y) comme une intersection entre les supports de X pour C et le domaine de Y. Cette observation nous a conduit par le passé à développer des algorithmes basés sur une représentation par vecteurs de bits des domaines [8], qui ont été depuis repris par de nombreux solveurs. Les contraintes *min/max* ou *element* peuvent également être partiellement exprimées par des opérations d'union/intersection.

Les domaines sur les nombres entiers dans *Concrete* sont polymorphiques. Sous une même interface, ils peuvent être réalisés soit par un objet représentant l'ensemble vide, soit par une valeur singleton pour représenter les variables affectées, soit par un intervalle de nombres entiers, soit par des vecteurs de bits. Ces derniers sont la représentation la plus générale.

La représentation par intervalles est utilisée si toutes les valeurs entières comprises entre $\underline{D}$ et $\overline{D}$ sont présentes dans D (on dit que D est *convexe*). Dans ce cas, seule la paire $(\underline{D}, \overline{D})$ est stockée en mémoire. L'espace de stockage est minimal, et les opérations de coupe ainsi que la suppression du plus petit ou du plus grand élément peuvent être réalisées en temps constant. En cas de suppression d'un élément au milieu de l'intervalle, un vecteur de bits est créé et retourné. Au contraire, si

après une modification de vecteur de bits on constate que $\overline{D} - \underline{D} = |D| - 1$, alors D est convexe et est représenté par un intervalle.

Un vecteur de bits est un tableau d'entiers naturels en base 2, représentant chacun b bits (généralement 64). On peut tester en $O(1)$ la présence d'un bit par des opérations de rotation et masquage bit-à-bit disponibles à bas niveau dans la plupart des langages de programmation[1]. Les bits sont indexés à partir de 0. Pour représenter un ensemble de valeurs arbitraires de manière compacte, on associe au vecteur de bits une fonction bijective associant à chaque index la valeur correspondante, et vice-versa. Cette fonction peut être réalisée par une structure de données ad-hoc, mais plus simplement, nous associons à chaque domaine un « offset » tel que pour un domaine D, le bit d'index $i - \mathit{offset}$ est à *vrai* ssi $i \in D$. Cela permet de représenter des valeurs négatives, ou optimiser la représentation d'un domaine commençant par une « grande » valeur. Cette représentation est en $O(\overline{D} - \underline{D})$, ce qui n'est pas optimal mais le surcout est rarement mesurable en pratique. Pour réaliser les opérations d'union et intersection entre deux domaines, il faut également que les associations index/valeur des opérandes correspondent. Les opérations de rotation à gauche ou à droite permettent d'établir la correspondance des offsets de manière efficace. De plus, *Concrete* conserve le dernier ajustement d'offset demandé en cache, notamment pour réaliser efficacement la technique décrite dans [8]. Ensuite, on peut réaliser l'opération d'union ou d'intersection par paquets de b bits à l'aide des opérateurs bit-à-bit du langage hôte.

Il est possible que $\underline{D} > \mathit{offset}$ suite à la suppression des premiers éléments du domaine, mais si $\underline{D} - \mathit{offset} \geq b$, un décalage est automatiquement réalisé et l'offset est mis à jour. Si $\overline{D} - \mathit{offset} < b$, un seul entier est nécessaire pour représenter le domaine. Nous utilisons le polymorphisme pour optimiser ce cas particulier.

Le principal inconvénient de la représentation par vecteurs de bits est qu'elle n'est pas naturellement persistante ou immutable. Cependant, sa relative compacité en fait un bon candidat pour des stratégies de recopie [12]. Pour obtenir un comportement véritablement persistant, il faut réaliser une copie du domaine à chaque fois que celui-ci est modifié. Retirer un seul élement du domaine nécessite de recopier l'ensemble du tableau (soit $\lceil \overline{D} - \underline{D} / b \rceil$ entiers de b bits à recopier). Si on retire ainsi chaque élément un par un, on peut totaliser $O(|D|^2)$ opérations. Lorsque l'on fait appel à la fonction de filtrage par un prédicat de cout $O(f)$ (cas le plus courant), il est possible de ne réaliser la copie qu'une seule fois et ainsi de faire l'ensemble du traitement de manière optimale en $O(|D|\, f)$. Cette fonctionnalité est importante car elle garantit de ne pas dégrader la complexité des algorithmes de propagation. L'opération de coupe inférieure jusqu'à la valeur $c \leq \overline{D}$ nécessite de recopier $\lceil c - \underline{D} / b \rceil$ entiers de b bits, et vice versa pour la coupe supérieure. Cette complexité est identique (au facteur b près) aux structures traditionnelles avec *trailing*.

[1] Certains langages comme Java ou Scala intègrent directement les vecteurs de bits dans leur bibliothèque standard.

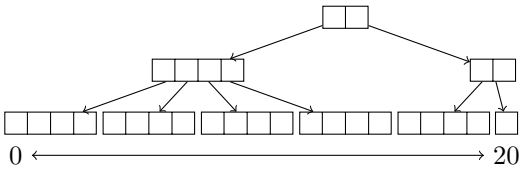

FIG. 1 : Un vecteur immutable de 21 éléments ($m = 4$)

3 Représentation de l'état du problème

La question de rendre persistante l'état global du problème reste entière. Bien sûr, on peut envisager de recopier l'état du domaine de chaque variable et de chaque contrainte à chaque nœud de l'arbre de recherche. Si les structures de données des domaines et des contraintes sont déjà persistantes, cela nécessite un traitement en $O(n + e)$ à chaque nœud. Le plus souvent, l'état d'une minorité de variables et de contraintes sont modifiés après un filtrage. Il est alors indispensable, pour des problèmes de grande taille, de ne sauvegarder que ce qui est nécessaire. On peut alors utiliser une technique de *trailing* classique pour gérer le backtrack. Cette technique n'est pas applicable si plusieurs traitements se font en parallèle.

Une alternative séduisante consiste à utiliser une structure de données persistante pour stocker l'état global du problème. Une référence sur une version de la structure de données est alors suffisante pour sauvegarder et restaurer l'état du problème à tout moment en temps constant, y compris dans le cadre de traitements en parallèle. Pour le stockage de l'état des domaines, nous avons besoin d'une structure de données capable d'associer un domaine à chaque variable, et permettant une lecture et mise à jour rapide dans un cadre persistant. Si les variables sont indexées de 0 à $n - 1$, on peut utiliser un simple tableau de domaines pour réaliser cette association. Cependant, comme évoqué précédemment, il faut utiliser une structure moins naïve qu'un tableau pour ne pas avoir à copier l'ensemble des domaines à chaque nœud, mais uniquement ceux qui ont été modifiés, tout en gardant de bonnes performances en lecture. La même remarque s'applique pour les contraintes et leurs états.

La structure de *vecteur immutable* introduite par le

langage de programmation Clojure [6] et inspirée par les *Ideal Hash Tries* [1] réalise un bon équilibre entre les performances de lecture et de modification. Les vecteurs immutables sont implantés sous forme d'arbres m-aires. La figure 1 illustre un vecteur immutable quaternaire. Chaque nœud est un tableau d'au plus m éléments. Les feuilles contiennent les données, les nœuds intermédiaires forment un chemin depuis la racine jusqu'à l'élément recherché. Pour un vecteur de taille n, la profondeur de l'arbre est de $O(\log_m n)$. Pour modifier un élément, on recopie chacun des nœuds conduisant à cet élément, soit $O(m \log_m n)$ opérations. Le choix typique de $m = 32$ réalise un bon compromis entre la vitesse de consultation et la vitesse de modification sur les machines modernes, en prenant notamment en compte les caches matériels et les optimisations sur les calculs de modulo. Dans le langage Java, il n'est pas possible de créer un tableau de plus de 2^{31} éléments. Cette restriction s'applique sur la plupart des structures de données du langage et est rarement remise en cause. Le vecteur immutable représentant un tel tableau n'aurait qu'une profondeur de 7 avec $m = 32$. En pratique, les opérations sur les vecteurs immutables peuvent être considérées de complexité constante.

4 Performances et conclusion

Une fois les structures de données définies, l'implémentation de n'importe quel algorithme de propagation et de recherche est relativement simple, à condition que tous les états des contraintes soient persistants. Certaines structures partiellement persistantes comme les *Sparse Sets*, voire non persistantes comme les *résidus* peuvent également être utilisées sous conditions (synchronisation, backtracking chronologique…) Cependant, bien que de complexité constante ou presque, les surcouts que représentent les recopies de domaine et parcours des vecteurs immutables ne sont pas nécessairement négligeables.

La longueur de cet article ne permet pas de rendre compte de benchmarks détaillés pour chaque structure de données. Nous avons comparé les performances des solveurs *CSP4J* [14] et *Concrete 3*. Ces deux solveurs utilisent les mêmes algorithmes de branchement et de propagation, mais *CSP4J* était entièrement basé sur des structures de *trailing* alors que *Concrete 3* se base sur des structures persistantes. L'impact du cout des vecteurs immutables est pratiquement négligeable devant des algorithmes de propagation de complexité au moins quadratique. Cependant, *Concrete 3* peut se révéler jusqu'à deux fois plus lent que *CSP4J* sur des propagateurs très rapides (contraintes binaires ou arithmétiques aux bornes). Des benchmarks plus complets sont en cours de réalisation. L'intérêt de ces structures de données est clairement dans leur simplicité d'utilisation, et les perspectives pour réaliser facilement et sans surcout des algorithmes de backtrack non-chronologiques ou parcourus en parallèle.

Références

[1] P. Bagwell. *Ideal Hash Trees*. Rapp. tech. EPFL, 2001.

[2] C. Bessière, J.-C. Régin, R.H.C. Yap et Y. Zhang. "An Optimal Coarse-Grained Arc Consistency Algorithm". In : *Artificial Intelligence* 165.2 (2005), p. 165–185.

[3] P. Briggs et L. Torczon. "An Efficient Representation for Sparse Sets". In : *ACM Letters on Programming Languages and Systems* 2.1–4 (1993), p. 59–69.

[4] J.R. Driscoll, N. Sarnak, D. Sleator et R. Tarjan. "Making Data Structures Persistent". In : *J. of Computer and System Science* 38 (1989), p. 86–124.

[5] P. van Hentenryck, Y. Deville et CM. Teng. "A Generic AC Algorithm and its Specializations". In : *Artificial Intelligence* 57 (1992), p. 291–321.

[6] R. Hickey. *The Clojure Programming Language*. `http://clojure.org/`. 2006.

[7] C. Lecoutre et R. Szymanek. "GAC for Positive Table Constraints". In : *Proc. CP'06*. Nantes, France, 2006, p. 284–298.

[8] C. Lecoutre et J. Vion. "Enforcing AC using Bitwise Operations". In : *Constraint Programming Letters* 2 (2008), p. 21–35.

[9] R. Mohr et T.C. Henderson. "Arc and Path Consistency Revisited". In : *Artificial Intelligence* 28 (1986), p. 225–233.

[10] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[11] J.-C. Régin, M. Rezgui et A. Malapert. "Embarrasingly Parallel Search". In : *Proc. 19th Itl Conference on Principles and Practice of CP*. 2013, p. 596–610.

[12] C. Schulte. "Comparing Trailing and Copying for Constraint Programming". In : *Proceedings of ICLP'99*. 1999, p. 275–289.

[13] J. Vion. *Concrete : a CSP Solving API for the JVM*. `http://github.com/concrete-cp`. 2006–2014.

[14] J. Vion. *Constraint Satisfaction Problem For Java*. `http://cspfj.sourceforge.net/`. 2006–2012.