X8II060

UFR Sciences et Techniques

M1 ALMA
2016-2017

Rapport projet Programmation par contrainte

Auteurs :
Montalvo Araya, Mario
Casonova

Février 2017

1 Introduction

Le travail demander consiste à développer un solveur CPS de la forme X,D,C avec X l'ensemble des variables, D l'ensemble des domaines et C l'ensemble des contraintes.

Le but est de baser le solveur sur un algorithme de type branch-and-prune comme préciser sur le sujet.

Nous avons principalement travaillé sur le problème des reines, en proposant plusieurs méthodes et implémentation nous avons pu tester leurs efficacités respectives pour ce problème. Nous avons aussi implémenté un backtracking sur le problème des magic squares.

2 Le backtracking

La méthode de backtracking consiste à essayer de réduire les domaines des différentes variables un par un en utilisant les contraintes du problème.

Nous vérifions à chaque "pas" de l'algorithme la validité des différentes valeurs que peut prendre une variable donnée grâce aux contrainte et si on trouve une valeur valide on avance.

Prenons un exemple sur un échiquier de 4*4 case :

La reine de la ligne 1 essaye de prendre la valeur 1 (la valeur indiquant sur quelle case de la ligne elle se trouve, comme dans le CM2). Les contraintes du problème ne l'interdisent pas, on continue.

La reine de la ligne 2 essaye de prendre la valeur 1. Pas possible à cause des contraintes donc elle essaye la valeur suivante. Même problème avec la valeur 2. Finalement les contraintes autorisent la valeur 3 donc l'algorithme continue et ainsi de suite.

Si nous arrivons à un ensemble de domaines réduit à une valeur pour chaque variable en respectant les contraintes alors nous avons une solution.

Avec cette méthode il n'y a pas de réduction des domaines à l'avance, nous nous contentons de vérifier les contraintes à chaque "pas".

3 Forward Checking

Cette méthode est une forme de propagation partielle de contraintes dans le cadre du backtracking.

Lors d'un backtracking classique, nous ne restreignons pas les domaines des variables et donc chaque valeur est examinée pour chaque variable. La première valeur passant le test des contraintes avec toutes les variables précédemment attribuées est affectée. Nous ne nous arrêtons uniquement lorsqu'aucune des valeur d'une variable donnée ne satisfait les contraintes du problème ou qu'une solution a été trouvée. De fait il arrive fréquemment que nous soyons en train de travailler sur une branche qui ne contient aucune solution possible sans le savoir.

Le foward checking permet de se rendre compte plus tôt qu'une certaine suite d'attributions va mener à une impasse quoi qu'il arrive. En effet, à chaque fois qu'une valeur est attribuée, nous restreignons les domaines de définition des variables encore non attribuées en suivant les contraintes du problème par rapport à cette attribution. Et quand l'un des domaines est réduit à une cardinalité de 0 nous sommes alors sûr que nous allons arriver à une impasse quoi qu'il arrive. Le fait d'éliminer à l'avance les valeurs in-

compatible des domaines des autres variables, nous évite plus tard de faire des test inutiles.

4 Partial Look Ahead

Le partial look ahead est un forward checking en plus poussé.

La méthode diffère du foward checking car lorsque l'un des domaines d'une variable est réduit à une seule valeur pendant la propagation des contraintes, nous attribuons cette variable et nous réitérons la méthode de pruning.

5 Les domaines

5.1 Version avec les classes Domaine

Nous avons choisi de représenter les domaines comme un ensemble ordonné d'entier pour le problème des dames.

Comme nous utilisons le java nous avons utilisé des **LinkedList<Integer>**. Cette structure nous offre des opérations en O(1) pour la suppression de valeur, ce qui nous intéresse lorsque nous prunons les domaines.

Précédemment nous utilisions des TreeSet mais la suppression d'une valeur quelconque se faisait en O(log n), ce qui ajoutait un coût très lourd à notre algorithme.

5.1.1 Les opérations

Nous avons implémenté différentes opérations sur les domaines, en voici la liste, avec leur complexités respectives.

— **La copie** : Nous utilisons pour cela un constructeur par recopie en java. L'opération se fait en O(n), il s'agit d'un opérateur de recopie classique :

```
                public Domain (LinkedList<Integer> valeurs){
        this.valeurs=valeurs;
        }
```

— **accès aux valeurs** : Comme nous utilisons des LinkedList nous ne pouvons pas accéder directement aux valeur. Nous sommes obligés de parcourir la liste jusqu'à la valeur choisie donc nous somme en **O(n)**. Dans tous les cas l'implémentation choisie en utilisant la classe Do-

maine nous oblige à avoir une complexité en **O(n)** pour cette opération.
— **suppressions aux bornes** : Elle se fait en temps constant, en utilisant les fonctions natives de Java pour les LinkedList **O(1)**
— **suppression valeur quelconque** : Elle se fait également en **O(1)** grâce aux méthodes fournies par Java car nous utilisons des itérateurs pour parcourir les listes.

5.2 Bactracking : un entier

Nous avons remarqué que pour effectuer un simple backtracking sur le problème des reines, le domaine de chaque variable peut se réduire à un simple entier que l'on incrémente à chaque fois que l'on teste une nouvelle valeur, sachant que nous les testons toutes et que tous les domaines sont égaux ($D_i = 1, 2, ..., n$).

5.3 Forward Checking : une matrice de booleens

Pour effectuer un forward checking nous sommes dans l'obligation d'avoir une représentation de tous les domaines, afin de pouvoir les restreindre si nécessaire. Nous avons donc choisi d'utiliser une matrice de booléens représentant l'échiquier, la valeur de la case (i, j) indiquant si la reine i peut se trouver sur la colonne j ou non. Dans notre cas la matrice est aplatie pour tenir dans une unique `ArrayList<Integer>`, ce choix reste à débattre.

5.3.1 Les opérations

La matrice est créée une seule fois et dispose d'une taille fixe. Il s'agit d'un tableau indexé, les complexités des opérations de lecture et d'écriture pour un index donné sont assurées en $O(1)$ par Java. Le parcours d'un domaine se fait en $O(n)$.

6 Les noeuds

6.1 Version avec les classes Node

Nous représentons un noeud comme une ArrayList de domaines. Cela permet un accès en O(1) aux différents domaine grâce à leurs index. Et comme nous ne supprimons pas de domaines, le fait que l'opération de suppression soit en O(n) ne nous dérange pas.

6.1.1 Les opérations

Nous avons implémenté différentes opérations sur les noeuds, en voici la liste, avec leur complexités respectives.

- **La copie** : Nous utilisons pour cela un constructeur par recopie en java. L'opération se fait en O(n), il s'agit d'un opérateur de recopie classique.
- **accès aux domaines** : Comme nous utilisons des ArrayList l'accès à un des domaines est en temps constant **O(1)**.
- **modification** : De même, l'utilisation des ArrayList nous assure la modification en temps constant **O(1)**
- **Affectation d'un domaine** : L'ajout des domaines se fait en fin d'ArrayList, avec un simple add(), opération en **O(1)**.
- **Tri des domaines** : Grâce à la méthode sort de java et un comparateur implémenté dans les classes domaines. Cette opération se fait en **O(n*log(n))**.

6.2 BackTracking : une liste chainée

Comme le bactracking travaille sur tous les domaines un par un et que tous les domaines sont identiques, il est inutile de maintenir une liste de tous les domaines, il suffit d'une liste des valeurs attribués (les affectations se font dans l'ordre, en commençant par la première variable) ou l'on ajoute ou enlève une valeur a la fin en fonction du parcours de l'arbre de recherche.

6.2.1 Les opérations

Les seules opérations directes sur les noeuds sont une suppression et une addition en fin de liste, dont la complexité en O(1) est assurée par Java.

La validation du noeud lors de l'affectation d'une variable requiers quant à elle un parcours de la liste ($n-1$ fois au pire) et donc une complexité en O(n) en utilisant un itérateur.

6.3 ForwardChecking : deux liste chainées

La première liste est identique à celle du ForwardChecking, elle contient les valeurs des variables attribuées. La deuxième contient les tailles des domaines, l'index correspondant à la ligne dans la matrice et la valeur à la taille du domaine à cet indice.

6.3.1 Les opérations

La liste des tailles est une `ArrayList` d'une taille fixe crée à l'initialisation et modifiée ensuite. Les opérations la concernant sont en $O(1)$ Les opérations sur la liste des valeur attribuées sont les mêmes que pour le BackTracking.

7 Les algorithmes

Il y a un main dans chacune des classes cité ci-dessous pour tester les implémentations des différentes méthodes.

7.1 BackTracking avec classe Node et Domaine

Les classes implémentant cette méthodes sont : BackTrackingQueen, BackTrackingTreeSetQueen.

Nous n'allons pas expliquer à nouveau la procédure du backtracking mais juste discuter de la complexité de l'algorithme en lui même.

D'après nos calculs, notre algorithme est en **O(n !)**. Ce qui nous empêche d'obtenir des résultats pour n $>$ 15 en des temps raisonnables.

En effet il faut voir comme il est décrit dans le cours que nous parcourons l'arbre de possibilité en entier jusqu'à l'obtention de toutes les feuilles. Sachant que l'on considère une "impasse" et une solution comme étant une feuille de l'arbre.

Cependant depuis la première version nous avons amélioré la méthode qui vérifie si une affectation est valide rendant l'algorithme beaucoup plus rapide. Auparavant à chaque fois qu'une nouvelle valeur était attribué nous re-vérifions en plus de la validité de cette dernière celle de toutes les précédente alors que leurs validités avait déjà était vérifié. Maintenant nous ne vérifions plus que la validité de la dernière valeur ajouté.

BackTrackingTreeSetQueen utilise les domaines avec les TreeSet la rendant beaucoup moins rapide que BackTrackingQueen qui utilise les domaines avec les LinkedList.

7.2 Backtracking avec linkedList<Integer>

La version proposée dans `QueensV3` a pour objectif d'effectuer le moins de parcours de boucle possible, afin de minimiser la complexité. Avec deux boucles imbriquées, l'algorithme a une complexité approximative de $O(n^2 * feuilles)$

7.3 BackTracking Magic Square

La classe implémentant cette méthode est : BackTrackingCarre.

Il s'agit de l'implémentation de la méthode du backtracking pour le problème des magics squares. Cependant étant donné la nature du problème cette méthode ne permet pas de trouver le nombre de solution possible pour un magic square de dimension supérieur à 4x4 en temps raisonnable.

Le temps de résolution pour un magic square croît beaucoup plus rapidement que le problème des queens quand on augmente les dimensions du magic square.

7.4 Forward Checking avec classe Node et Domaine

La classe implémentant cette méthode est : ForwardQueens.

Cette méthode a dans l'absolu la même complexité que le backtracking car elle est inhérente au problème.

Cependant comme nous faisons une propagation partielle des contraintes à chaque fois que nous attribuons une valeur valide cela permet de supprimer dans les domaines nous fixé les valeurs incompatibles avec la dernière valeur attribuée.

Cela rend la résolution du problème des reines beaucoup plus rapide.

7.5 Forward Checking avec classe Node et Domaine et un tri

La classe implémentant cette méthode est : ForwardAndTrieQueen.

Cette méthode fait quasiment que la même chose que la précédente cependant en plus de faire le forward checking nous trions les domaines à chaque appelle récursif.

De cette manière les domaines avec la plus petite cardinalité sont attribués en premier. De cette manière nous nous rendons compte plus tôt si une certaine attribution de domaine mène à une impasse ce qui accélère l'algorithme.

7.6 ForwardChecking avec matrice de booléens

La classe implémentant cette méthode est : QueensV6, QueensV5.

Encore une fois le but de cette version était de réduire au minimum de

nombre de parcours de tableau. La matrice d'entier nous permet d'avoir un accès en lecture et écriture en temps constant pour un indice donné. Les contraintes du problème des reines font que l'ont peut calculer les index des valeurs à pruner en fonction de la valeur nouvellement attribuée (mêmes diagonales et même colonne). Le pruning ne nécessite donc pas de parcours de domaine, nous nous contentons de parcourir les variables restant dans la liste et de pruner chaque valeur distinctement.
Dans la version `QueensV6` nous avons ajouté une liste des tailles des domaines afin de pouvoir s'arrêter de brancher si le pruning génère un domaine vide. Afin d'effectuer le backtracking, nous utilisons une liste d'index des valeurs qui ont été modifiées. Un parcours de cette liste en fin de branchement nous permet de restaurer la matrice de domaines dans sont état avant branching/-pruning et évite de devoir construire un nouvel ensemble de domaine ou de recopier intégralement l'ancien.

7.7 Partial Look Ahead

La classe implémentant cette méthode est : QueensPLH.

Pour pouvoir faire du Partial Look Ahead en partant de l'agorithme de ForwardChecking avec matrice de booléen, il nous fallait ajouter un moyen de trier les domaines afin d'obtenir les domaines réduits à une seule valeur par le pruning. Cette nouvelle fonctionnalité nous permettrais également de travailler sur le domaine le plus petit plutôt que de les prendre dans l'ordre. Plutôt que d'implémenter un tri sur les domaines (qui a une complexité en $n * log(n)$ nous avons choisi de sauvegarder le domaine le plus petit généré par le pruning (en utilisant la même boucle). De cette manière chaque nouvel appel de la méthode `branchAndPrune` se fait sur le plus petit domaine disponible. Nous avons également décider d'ajouter un ensemble d'entiers bi-dimentionnel pour représenter les valeurs encore disponible pour chaque domaine et un ensemble d'entiers pour indiquer les index des domaines non affectés. Ces ensembles sont implémentés par des `LinkedHashSet<E>` fournis par java et assurant les opérations `add()`, `contains(E e)` et `remove(E e)` en temps constant. Ces ajouts nous permettre d'effectuer moins de passage de boucle a chaque appel et nous a permis de réduire légèrement la complexité de notre algorithme.

7.8 Versions "1 solution"

Classes : QueensV5FC1Sol et QueensPLH1Sol.

Afin de pouvoir effectuer des comparaison nous avons implémenté des

versions s'arrêtant à la première solution trouvée. La plus efficace (QueensPLH1Sol) nous permet de trouver une solution pour 200 reines en 90 secondes.

8 Temps d'exécution

Machine utilisée :
— processeur Intel(R) i7, 4GHz
— memoire 16 Go
Pour le problème des queens :

Nombre de Case	12	13	15	16
BackTrackingTreeSetQueen	0.66s	3.12s	120s	-
BackTrackingQueen	0.509s	2.58s	105s	-
QueensV3	0.331s	1.621s	66.6s	-
ForwardQueens	0.263	1.199	40s	-
ForwardAndTrieQueen	0.243	1.033s	32s	-
QueensV6	0.218s	0.731	21s	139s
QueensPLH	0.35s	0.85s	19s	114s

Pour le magic square : 3x3 = 0.08s et 4x4 = 64s

Conclusion

En implémentant la méthode de backtracking nous avons très vite pu voir ses limites. En effet avec la complexité en $\mathbf{O(n!)}$ l'algorithme met un temps considérable pour trouver toutes les solutions sur un échiquier de 15 case.

Il est donc nécessaire que nous réussissions à appliquer les méthodes de contraction vu en cours pour arriver à une résolution des problèmes en un temps raisonnable.

Cependant même en appliquant des méthodes réduisant le nombre de possibilités (feuilles) nous nous retrouvons bloqués par la complexité inhérente aux problèmes. En effet, malgré toutes les tentatives d'amélioration que nous avons pu effectués sur le problème des dames, notre meilleure version ne calcule les solutions que pour une reine de plus en un temps équivalent.