Rapport du Projet de programmation par contraintes

Baptiste Canovas-Virly
Lydia Souriot

30 mars 2017

Table des matières

Introduction

Le but de ce projet est de réaliser un solveur CSP capable de résoudre des problèmes de programmation par contrainte classiques, en utilisant une stratégie de branch and prune.
Pour cela nous avons donc dû commencer par sélectionner un problème de base : le problème des dames.
Ainsi que deux algorithmes de propagation de contraintes : Le forward checking, et la consistance d'arcs.

Chapitre 1

Représentation d'un problème

1.1 Description des classes

1.1.1 Domaines

Le domaine d'une variable est représenté sous la forme d'une pile d'entiers ordonnés. Nous avons choisi cette représentation pour sa dimension variable, et parce que l'on considère que l'ordre d'accès aux variables (plus petite ou plus grande d'abord), n'a pas d'importance.

1.1.2 Contraintes

Les contraintes modélisent structurellement une partie du problème des reines. En effet, pour ce problème, on retrouve 4 types de contraintes différentes :
- Une reine par ligne
- Une reine par colonne
- Une reine par diagonale montante
- Une reine par diagonale descendante.

Sachant que l'on a choisit de modéliser le problème en posant une variable par ligne pour représenter le numéro de la colonne où se situe la reine, la première contrainte est déjà réglée. Restent les trois autres :
- Une reine par colonne : Alldiff(Xi)
- Une reine par diagonale montante : pour tout i et j différents, xi + i != xj + j
- Une reine par diagonale descendante : pour tout i et j différents, xi - i != xj -j

On peut donc construire une structure contenant i, j, xi, et xj, pour toutes ces contraintes, en forçant xi + i à être différent de xj + j.

1.1.3 Problème

Un problème est une structure regroupant les domaines admissibles pour les variables, les contraintes associées, ainsi que la taille des données et le nombre de contraintes.

1.1.4 Noeuds

Nous avons choisit de modéliser les noeuds par une pile contenant tous ces noeuds. Un noeud est l'ensemble des domaines possibles pour chaque variable du problème. La pile est justifiée par la recherche en profondeur : une liste aurait permis une recherche en largeur, mais nous avons préféré pouvoir obtenir le plus rapidement possible une solution plutôt que de les avoir toutes à la fin.

1.2 Complexité des opérations

1.2.1 Domaines

Copie :
Accès aux valeurs (à partir de leur valeur) : O(n)
Accès aux valeurs (à partir de leur indice) : O(n)
Suppression d'une valeur aux bornes :
-borne inférieure : O(1)
-borne supérieure : O(n)
Suppression d'une valeur quelconque : O(n)

1.2.2 Noeuds

Copie :
Accès aux domaines : O(1)
Modification et affectation d'un domaine : O(n)

Chapitre 2

Résolution

2.1 Backtracking

Le principe du Backtracking est de tester les branchements possibles, en s'arrêtant de tester une branche dès lors qu'aucune solution viable n'est possible sur celle-ci.
Pour l'implémentation, nous avons utilisé l'algorithme donné par le cours, et qui a théoriquement une complexité très grande. En effet, il pourrait générer toutes les possibilités de combinaison entre les domaines, s'il n'était pas restreint par les contraintes.
La complexité théorique des CSP est : O($S * d^n$)
Avec :
- S la taille du CSP
- d la taille maximale d'un domaine
- n le nombre de variables.

2.2 Pruning

L'ajout d'une composante de Pruning sur un algorithme de Backtracking permet d'obtenir un algorithme dit de BranchAndPrune. Le but n'est plus de simplement vérifier la possibilité qu'une branche admet une solution ou pas, mais de réduire les domaines possibles à chaque embranchement pour diminuer l'espace de recherche, et donc la complexité de l'algorithme total.

2.2.1 Forward Checking

Dans le cas du Forward Checking, nous nous intéressons aux contraintes dont il ne reste qu'une valeur à fixer. Dans certains cas il est possible de la fixer alors directement. Mais pour nos contraintes du problème des dames, il n'est possible que de supprimer les valeurs non admissibles du domaine de cette variable non fixée. Il y avait deux possibilités d'implémentation de cet algorithme : soit en se basant sur les variables fixées, soit en se basant sur les contraintes.

Dans le premier cas, il faut alors parcourir à chaque fois l'ensemble des contraintes pour déterminer où se trouve chaque variable fixée (à moins de disposer d'un index). Le deuxième cas parcourt toutes les contraintes et réduit le domaine des variables libres dans les contraintes où il n'en reste qu'une. D'un point de vue complexité, le deuxième est bien sur largement plus intéressant, puisqu'il faut pour le premier : O($n * m * n$) (nombre de variables * nombre de contraintes * calcul de la taille des variables de la contrainte) et pour le deuxième O($m * n$) (nombre de contraintes * calcul de la taille des variables de la contrainte).

Il aurait été possible de diminuer le temps de calcul en intégrant une taille dans la structure des domaines, mais la complexité aurait été augmentée ailleurs, et la suppression de domaines aurait toujours été en O(n). Les gains auraient donc été limités.

2.2.2 Consistance d'arcs

La consistance d'arc vise à vérifier que pour chaque valeur du domaine de chaque variable, il existe une solution. C'est à dire que chaque contrainte contenant une variable donnée pour toutes les valeurs de son domaine, il existe au moins une valeur dans les domaines des autres variables de la contrainte considérée. Si ce n'est pas le cas, la valeur du domaine n'étant pas consistante est enlevée.

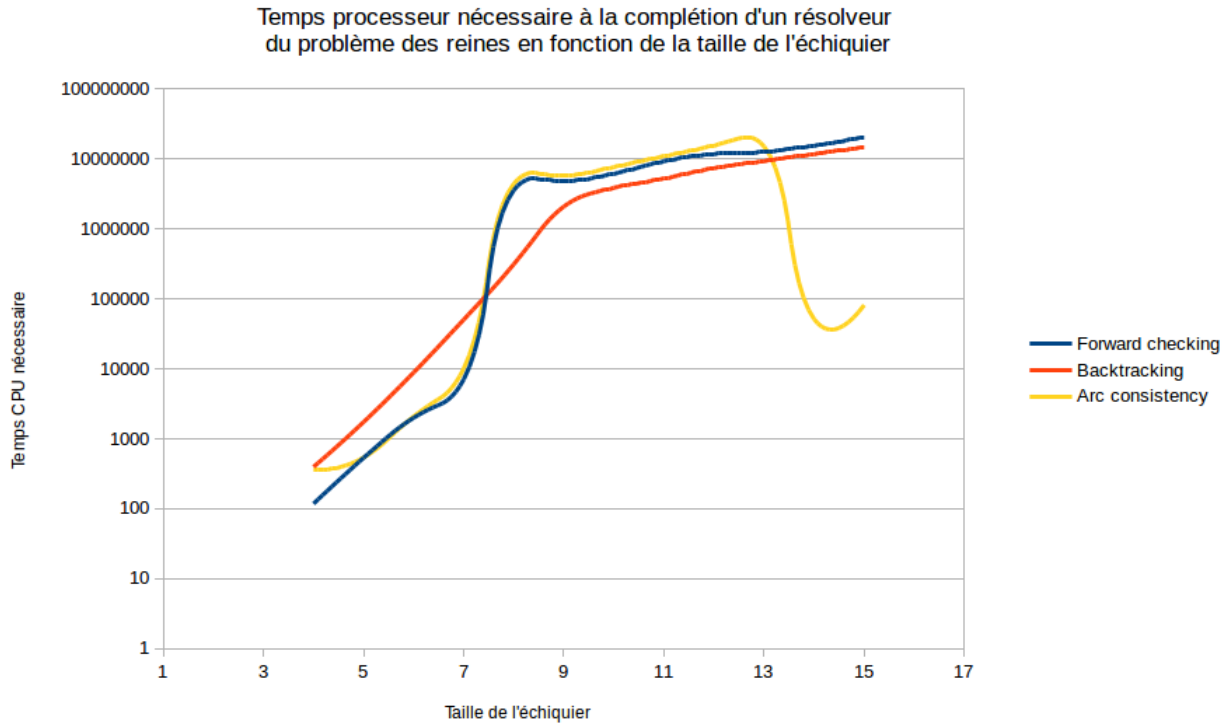
Au niveau de l'implémentation, il n'y a pas beaucoup de choix : il faut vérifier la consistance pour chaque contrainte. Nous avons donc réalisé une boucle sur ces contraintes qui compare après le domaine de la première variable avec le domaine de la seconde, ce qui suffit puisque nous avons des contraintes symétriques. Potentiellement, la complexité est de $O(m * n^2)$.

Par rapport au Forward Checking, la complexité est plus importante, mais la réduction des branches l'est également plus. Est ce donc intéressant ? Nous allons voir cela dans le chapitre suivant.

Chapitre 3

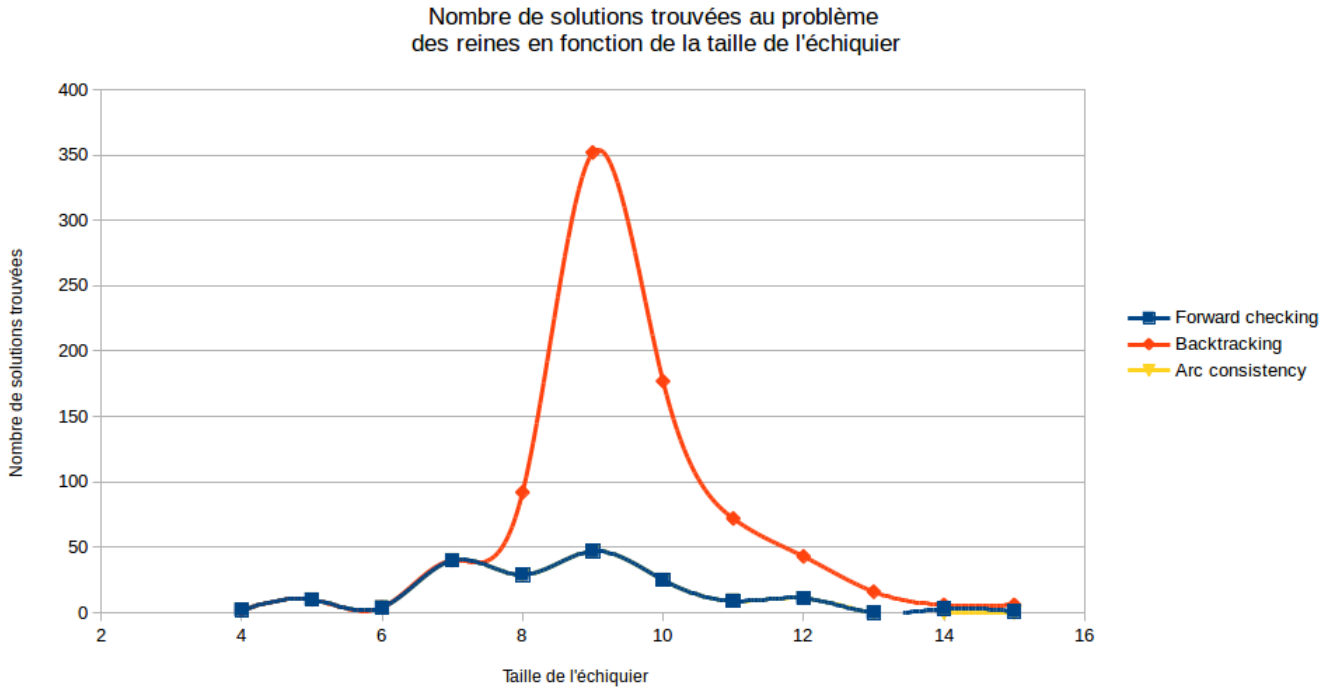
Résultats

3.1 Comparatif des temps d'exécution



On remarque que les résultats sont aberrants. Utilisant Valgrind et gdb, on remarque des problèmes de fuites mémoires et accès illégaux, allouer de la mémoire sans l'initialiser à zéro nous amène très rapidement à une segmentation fault.

3.2 Comparatif du nombre de résultats trouvés



Le nombre de solutions trouvées n'est pas moins anormal, pour les mêmes raisons Remarque : on a certaines symétries, mais pas toutes, et on a vérifié pour les petites tailles d'instances, ce ne sont que des résultats valides.

Conclusion

Nous voulions implémenter un problème supplémentaire, mais nos efforts se sont entièrement consacrés au debugging de notre code C++, puis, après avoir décidé de repartir de zéro, de notre code C.
Nos lacunes en debugging ne nous ont pas permis d'avancer malgré la grande quantité de temps que nous avons consacré au projet.