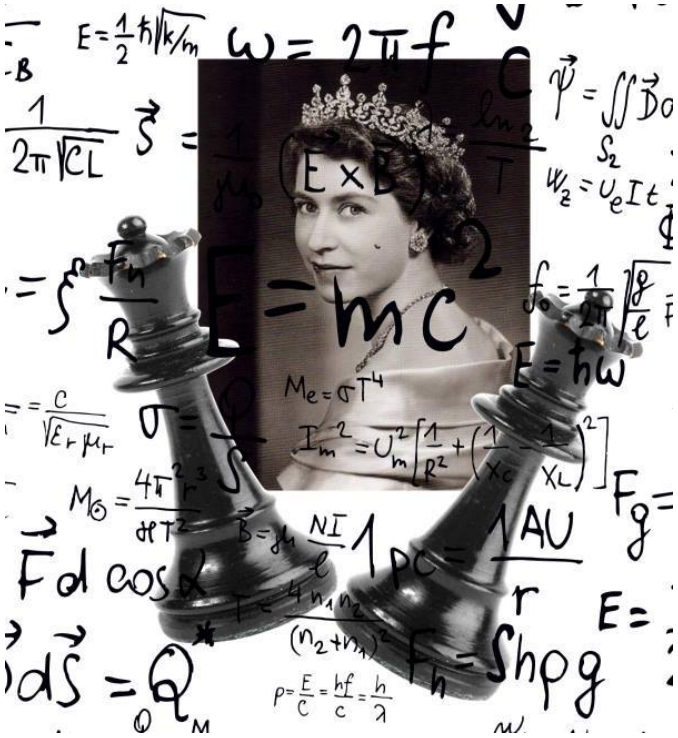
Projet de Programmation par Contraintes : Élisabeth

T.Béziers La Fosse, D.Bordet, J. Clayton, A. Giraudet, B. Moreau

4 mars 2016



Sommaire

1 Introduction

1.1 Problématique et but du projet

Dans le cadre du module de Programmation par Contraintes en Master 1 ALMA, nous avons eu pour objectif d'implémenter un solveur capable de résoudre le problème des *N-Queens*.

Ce projet est étendu sur trois séances, durant lesquelles nous avons dû concevoir ce solveur, capable d'effectuer une recherche complète des solutions, mais aussi une recherche locale, et enfin comparer les résultats d'exécutions sur des problèmes de taille variable.

Pour implémenter notre solveur, nous avons pris la décision d'utiliser le langage de Programmation Python3. Nous avons hésité avec $C++$, mais avons finalement choisi ce premier. Même si ses performances sont légèrement moins bonnes qu'avec $C++$, nous apprécions sa simplicité d'utilisation et sa gestion des listes et dictionnaires particulièrement adapté à la façon dont nous avons pensés notre solveur. De plus aucun autre module de notre Master ne nous a donné l'occasion de nous en servir, si bien que nous avons vu ici une opportunité pour nous réhabituer à ce langage particulièrement au goût du jour.

Dans ce rapport nous verrons en première partie le problème des *N-Queens* avec plus de précision. Ensuite nous parlerons de la recherche complète que nous avons implémenté, puis la recherche locale. Enfin nous comparerons nos résultats d'exécution.

1.2 Problème des N-Queens

Le problème des *N-Queens* est un problème très célèbre dans le monde des mathématiques. Ce dernier, évoqué pour la première fois en 1850 par Franz Nauck consiste à poser n reines sur un échiquier de n cases sans qu'elles puissent s'attaquer. Ainsi elles ne doivent pas :

— Être sur la même ligne
— Être sur la même colonne
— Être sur la même diagonale

Ce problème est relativement complexe, car il n'existe pas d'algorithme polynomial.

En testant toutes les possibilités de placement des reines sur un échiquier de taille 8×8 avec un algorithme de recherche exhaustive, on se retrouve à parcourir 64^8 placements possibles, soit 2^{48}, si bien qu'on arrive rapidement à

des temps d'exécution bien trop longs. Néanmoins on peut réduire facilement ce temps d'exécution avec la programmation par contraintes.

1.2.1 Contraintes

On doit ainsi modéliser les contraintes dans notre programme pour réduire drastiquement le nombre d'itérations avant de trouver une solution. Les contraintes, en langage naturel, sont donc :

— Il ne doit y avoir qu'une seule reine par ligne.
— Il ne doit y avoir qu'une seule reine par colonne.
— Il ne doit pas y avoir plus d'une reine par diagonale ;

Ainsi pour modéliser ceci sous la forme d'un *CSP*, on utilisera les variables suivantes : x_i : Reine de la ligne i. $x_i = j : j$, numéro de la colonne où est placée la reine.

Avec ce modèle on est sûr qu'il n'y aura pas d'attaques sur une même ligne, puisqu'on n'y autorise qu'une seule reine. Les contraintes sont donc les suivantes :

— Pas d'attaque sur les colonnes : $allDifferent(x_i)$ $\forall i \in [1, n]$. $\forall i \neq j$, $x_i \neq x_j$
— Pas d'attaque sur les diagonales : $\forall i \neq j, |x_i - x_j| \neq |i - j|$, et $\forall 1 \leq i < j \leq n : |x_i - x_j| \neq j - i$

2 Complete Search

2.1 Branch & Prune

La recherche complète consiste à trouver toutes les solutions d'un problème, ou bien à prouver qu'il n'existe pas de solutions. Pour effectuer cette recherche, on utilise un algorithme de *Branch & Prune*, fortement inspiré de celui vu en cours.

On part d'un nœud vide, et pour chaque possibilité d'assignement de la variable suivante on crée une branche, c'est le *branching*.

On assigne donc nos variables x_i une par une et on retire les valeurs qui ne valident pas les contraintes du domaine des variables pour les branches suivantes, c'est le *pruning*. S'il n'y a plus de valeur possible dans le domaine d'une variable, c'est que l'assignement ne valide pas les contraintes. On peut donc fermer la branche et passer à la suivante.

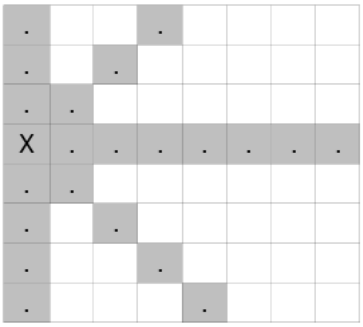
Ainsi les branches dont toutes les variables sont assignées représentent une solution. Et de cette manière, en récupérant tous ces assignements différents, nous récupérons l'ensemble des solutions du problème.

2.1.1 Backtracking

Le *backtracking* est semblable au Branch & Prune sauf que l'algorithme ne "prune" pas. En effet, il créé des branches pour tous les élément du domaine de chaque variables sans exception. on ne vérifie la satisfiabilitée des affectations que quand toutes les variables ont été affectées.
Cet algorithme est donc évidemment moins efficace que le Branch & Prune.

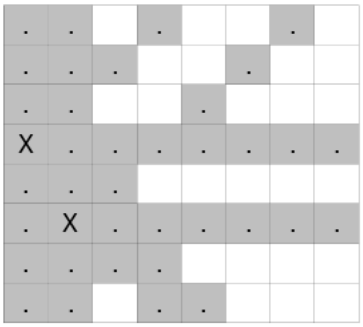
Dans l'exemple suivant, on assigne la variable montrée par la croix. Lors de notre pruning, on réduit donc du domaine des variables des noeuds suivants toutes les cases grisées, c'est à dire les cases sur la même colonne, et sur les diagonales, qui violent les contraintes. https ://thewalnut.io/app/release/11/

FIGURE 1 – Première Reine



On assigne ensuite une seconde variable, et on recommence tant que le domaine de la variable suivante n'est pas vide. Auquel cas la branche ne peut aboutir à une solution et on l'abandonne.

FIGURE 2 – Seconde Reine



Ainsi on peut trouver toutes les solutions possibles au problème des NQueens.

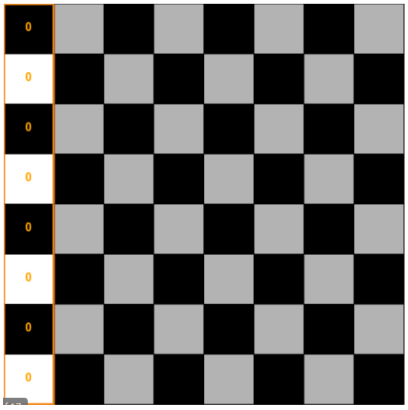
3 Local Search

La recherche locale de notre programme, par rapport à la recherche complète, s'arrête dès qu'elle trouve une solution au problème. L'algorithme implémenté se nomme *min-conflicts algorithm*.

Pour trouver une solution, ce dernier numérote chaque cellule de la ligne courante par le nombre de collision avec d'autres reines. En effet, si une cellule se trouve sur la même ligne qu'une autre reine, La cellule prend 1 pour valeur. Si elle est en plus sur la diagonale d'une seconde reine, elle prend 2 comme valeur. Ainsi pour placer nos reines, on choisira toujours les cases valant 0.

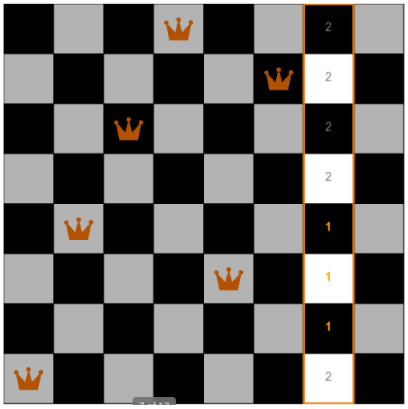
Lorsque plusieurs cellules de la ligne courante ont 0 comme valeur, on choisira la première parcourue. On aurait pu aussi la choisir de manière aléatoire, mais afin d'améliorer le temps d'exécution c'est la première solution que nous avons choisi.

FIGURE 3 – Échiquier vide



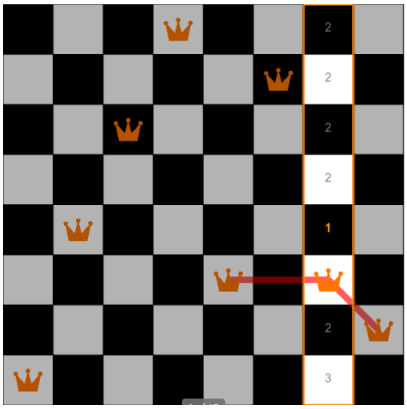
Ainsi on commence par la première ligne, en ayant 0 comme valeur sur chaque cellule.

FIGURE 4 – Premier parcours



On place la première reine arbitrairement, on calcule ensuite l'indice de chaque cellule des lignes suivantes, jusqu'à avoir placé une reine sur chaque ligne. Si l'une d'elle ne comporte pas de cellule indicée à 0, dans ce cas on place la reine sur l'indice minimum.

FIGURE 5 – Retour en arrière

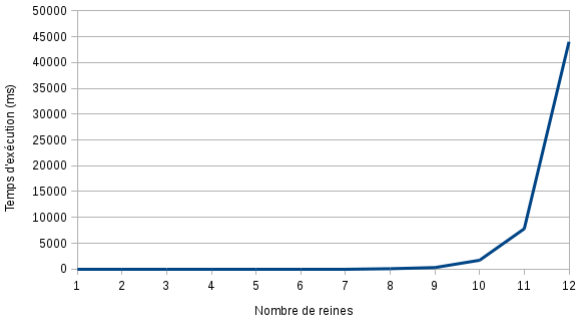


Enfin, lorsque la reine de la dernière ligne a été placée, on se rend sur la reine violant une contrainte la plus proche. On la déplace sur la cellule de la ligne qui a l'indice minimum. Si cette cellule n'est pas à 0, c'est qu'une nouvelle reine viole une contrainte, on se rend à nouveau sur cette reine, et on la déplace. On recommence tant que des contraintes sont violées.

4 Résultats d'exécution

4.1 Complete search

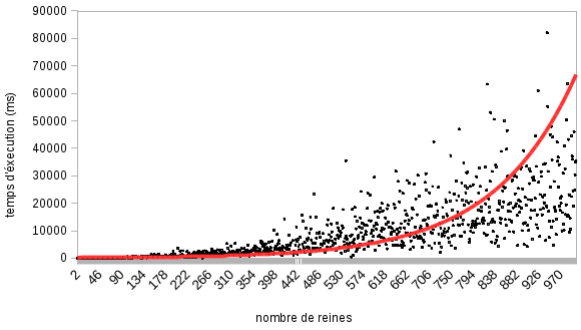
FIGURE 6 – Recherche complète



Voici les résultats d'exécution de l'algorithme de recherche complète de solutions. On voit assez rapidement que le temps d'exécution croit de manière exponentielle, puisque à partir de 12 reines, la résolution dure 44 secondes. Nous n'avons pas fait plus d'exécutions pour une raison temporelle évidente.

4.2 Local search

FIGURE 7 – Recherche locale



Voici ici les résultats d'exécution de l'algorithme de recherche locale d'une solution. Par rapport à l'algorithme précédent, on observe que l'on peut résoudre des problèmes avec un nombre plus important de reines. La croissance est elle

aussi exponentielle, mais permet des résultats bien plus rapides qu'avec une recherche complète.

5 Conclusion

Pour conclure ce projet, l'implémentation des algorithmes de recherche de solutions a été un exercice très intéressant, et nous a notamment permis de comprendre beaucoup plus aisément les algorithmes vus en cours. En voyant les résultats d'exécution, on comprend rapidement que l'algorithme de recherche locale est bien plus rapide que celui de recherche complète. Si bien qu'il vaut mieux utiliser ce dernier dans le cas où l'on a besoin de toutes les solutions. Ainsi l'algorithme de recherche locale est plutôt efficace dans le cadre d'un problème de satisfiabilité.

6 Sources

Image	Source
1 2	http ://www.cs.cornell.edu/ wdtseng/icpc/notes/bt2.pdf
3 4 5	https ://thewalnut.io/app/release/11/