Résolution du problème des N-Reines par différents algorithmes

Introduction

Dans le cadre de l'option Résolution de Problème nous devions réaliser un solveur pour le problème des N-Reines, ceci en utilisant plusieurs algorithmes de recherche.

J'ai réalisé cette application en utilisant le langage C++ sous l'environnement de développement (EDI) KDevelop4.

Présentation du problème

Le principe des N-Reines est assez simple, il faut placer N reines sur un échiquier de NxN cases ; Ceci sans que les reines ne soient en « prises ».

Exemple pour N=4 :

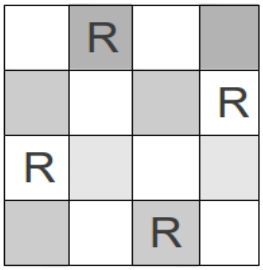


Illustration 1: Configuration Solution

L'illustration nous montre une configuration qui respecte les contraintes du problème. Tandis que sur l'illustration 2 nous voyons que 2 Reines sont en prises l'une de l'autre. Pour N=4 on trouve 2 solutions au problème, elles sont symétriques.

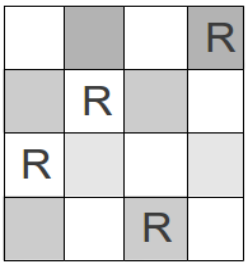


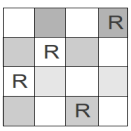
Illustration 2: Configuration Incorrecte

Le problème est discutable pour N>0, N ∈ Naturel.

Représentation des données (Classe : etape)

Pour représenter l'échiquier j'ai utilisé un vecteur d'entiers (`vector< int > position`), chaque membre i du vecteur représente la position verticale de la reine (en considérant la case du haut à gauche comme origine (0,0)) :

Exemple :

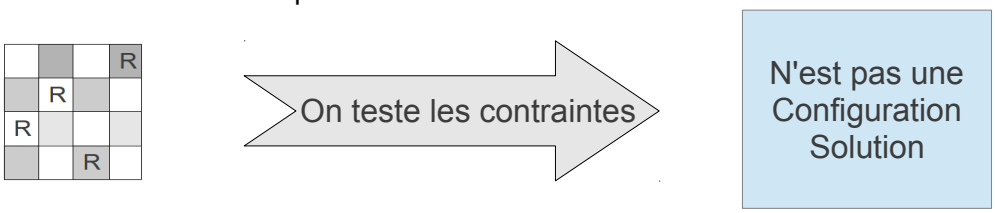


Correspond au vecteur :
{2,1,3,0}

Un premier algorithme « Générer puis Tester »

Présentation :

Une méthode générer puis tester implique qu'on génère une solution au problème, puis ensuite on teste si toutes les contraintes sont respectées.



Cette méthode induit qu'on génère tous les placements possibles de Reines sur l'échiquier, c'est à dire pour N=4, 256 configurations différentes (Si comme dans mon algorithme on considère qu'il n'y a pas deux Reines sur la même colonne. On en déduis que cette méthode est assez coûteuse en ressources.

Implémentation (generate_and_test) :

Les fonctions `generate_and_test` représentent mon implémentation de l'algorithme générer puis tester.

On génère des configurations complètes (des instances d' `etape`), c'est à dire N reines placées sur un échiquier de NxN, et ensuite on vérifie que les contraintes sont respectées grâce à la méthode `Check()` d' `etape`. Si la configuration respecte les contraintes on l'affiche au moyen de `etape::printPosition()`.

Affichage en console : « `Solution n°1 :[1 3 0 2 ]` »

Un second algorithme « Tester et Générer »

Présentation

Une méthode tester puis générer est une méthode qui implique qu'on teste à chaque génération on teste si la configuration obtenue est « viable » ou non. J'entends par viable le fait qu'il n'y ai pas de reines en prise l'une de l'autre, dans un cas pareil il ne sert à rien de placer les autres reines puisque au final la configuration complète ne respectera pas les contraintes.

Cette méthode est plus efficace que la précédente car on ne génère pas autant de configurations (256 pour `generate_and_test`, 18 pour `test_and_generate`)

Implémentation (test_and_generate) :

Les fonctions `test_and_generate` sont mon implémentation de la méthode tester et générer.

On place les reines dans l'ordre lexicographique, à chaque reine placée on vérifie la consistance de notre configuration (`etape`) grâce à `etape::Check()`. Si aucunes contraintes n'est violées on poursuit le placement, sinon on passe à la position suivante de la reine.

Si on arrive à une configuration complète et toutes les contraintes sont respectées, alors on affiche la configuration solution.

Un troisième algorithme : « Tester en amont »

Présentation :

Un algorithme tester en amont (forward checking), est une méthode qui vérifie avant d'instancier une valeur. Ici on vérifie où on peut placer la reine avant de la placer, de cette manière on ne place pas des reines « inutilement », toutes les reines qui sont placées sont légitimes. Ainsi on génère uniquement les configurations complètes solutions.

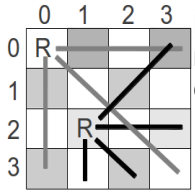
Cette méthode est encore moins gourmande que la précédente, on génère beaucoup moins de configurations, une configuration n'est générée que si elle solution partielle au problème, c'est à dire qu'elle respecte les contraintes de non-prises des reines déjà placée.

Implémentation (forward_checking) :

Les fonctions `forward_checking` représentent mon implémentation d'un algorithme tester en amont sur le problème des N-Reines.

Avant de placer chaque reine on calcul son domaine, c'est à dire quelles places ne respectent pas les contraintes. Ensuite on parcours le domaine, pour chaque places possibles on génère le reste de l' échiquer.

Exemple : Pour une configuration {0,2,-1,-1}, -1 signifiant que la reine n'est pas placée.



Le domaine de la reine 2 est vide {false, false, false, false} c'est à dire qu'on ne peut la placer sans qu'elle rompe les contrantes, on abandonne donc cette configuration partielle.

Utilisation de l'application :

Contenu de l'archive :

- le présent Rapport,
- un dossier NReine qui contient la version console de l'application
- un dossier NreineGUI qui contient la version avec interface graphique

Application console : Nreine/build/nreine

Pour lancer le programme :

-Se placer dans le dossier Nreine/build/

-Lancer la commande « `./nreine` »

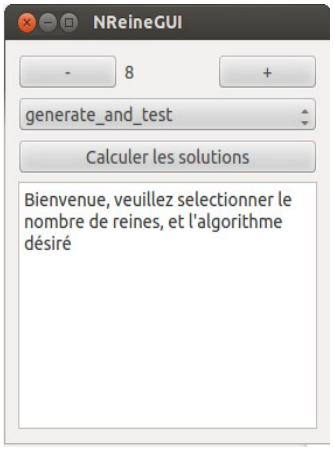
Options : « `-n nombre` » , nombre de reines.

« `-a algorithme` », algorithme choisit. (generate_and_test test_and_generate forward_checking)

« `-s nombre` », nombre de solutions que vous voulez afficher (optionnel, par défaut toutes les solutions sont affichées)

Exemple, calculer les 10 premières solutions pour N=12 avec l'algorithme `forward_checking` :

« `./nreine -n 12 -a forward_checking -s 10` »



Application graphique : NreineGUI/build/NReineGUI

Pour lancer le programme :

-Se placer dans le dossier NreineGUI/build/

-Lancer la commande « ./NReineGUI »

Option : « -c » utiliser la version console (Ex « ./NreineGUI -c -n 10 -a test_and_generate »

NB : la version graphique est beaucoup plus lente que la version console.

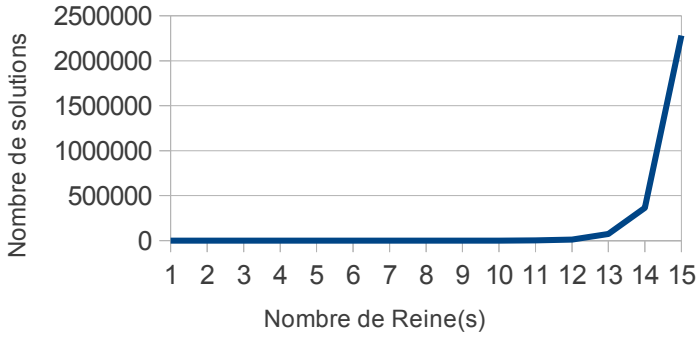
Quelques statistiques :

Ces statistiques ont été réalisées avec la commande :

« `./Nreine/build/nreine -n X -a A` » équivalente à « `./NreineGUI/build/NReineGUI -c -n X -a A` »

Où X représente le nombre de reines, et A l'algorithme choisit.

Nombre de solutions en fonction du nombre de Reines :



Temps d' exécution des algorithmes en fonction du nombre de Reines :