Colas Rémi **1D1**
Filaux Nathan

Compte rendu
SAE S2.02

Introduction :

Le sujet que nous avons choisi est le problème des dames qui consiste à trouver les solutions possibles telles que chacune des n reines placées sur un échiquier de n dimensions ne s'attaquent pas entre elles. C'est à dire qu'aucune reines ne doivent être placées sur la même ligne, la même colonne ou la même diagonale. Il faut donc trouver une stratégie pour limiter le nombre de calculs, on pourra par exemple éliminer des cas lorsque l'on sait à l'avance qu'ils ne seront pas solutions. Si l'on n'optimise pas le programme, son exécution peut très vite devenir problématique si l'on souhaite trouver des solutions sur des échiquiers avec de grandes dimensions.
Le nombre de solutions pour n reines à une tendance exponentielle cependant ce n'est pas toujours le cas (comme pour 6 reines ou le nombre de solutions est inférieure à celui de 5 reines).
La principale difficulté avec cet exercice est de réussir à optimiser le programme pour trouver les solutions dans un temps raisonnable. Il faut alors trouver des

moyens de réduire le nombre de calculs tout en cherchant à trouver toutes les solutions possibles.
Pour faire cela il faut pouvoir stocker les solutions déjà trouvées mais aussi savoir reconnaître les cas où il n'y a pas de solutions. Il faut également trouver un moyen de savoir si deux reines s'attaquent que soit avant qu'on la pose ou même après pour vérifier si ce cas de figure est une solution ou non. De plus, il est nécessaire de trouver un moyen de parcourir toutes les possibilités afin de trouver toutes les solutions.
Pour résoudre ce problème il existe plusieurs algorithmes tels que celui de force brute, de backtracking, de recherche locale ou bien d'autres. Nous avons tous les deux opté pour un algorithme de backtracking avec de la récursivité, le premier en programmation objets et l'autre en programmation procédurale.
Le principe de l'algorithme de backtracking consiste à vérifier à chaque étape si le placement est valide. Si le placement n'est pas valide, on remonte en arrière et on essaie une autre combinaison et cela de façon récursive.

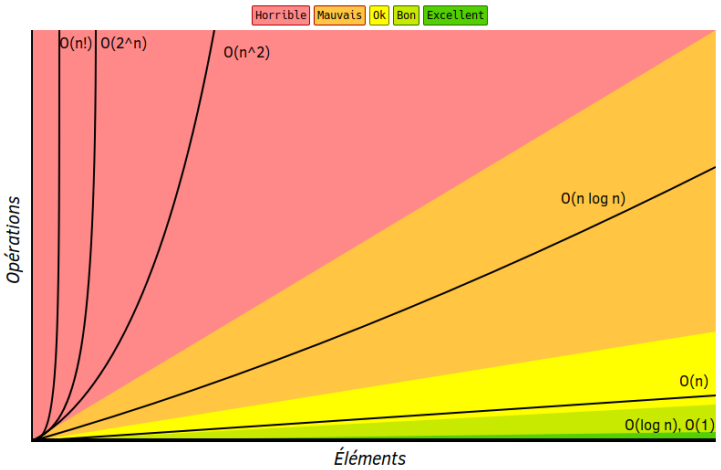
Programme 1:

Dans le premier algorithme, on imite un échiquier en programmation objet avec une classe "Plateau" de n dimensions comportant un tableau à deux dimensions dans lequel chaque case est représentée par "True" ou "False".
Si la case vaut "True" cela signifie que la case est libre pour placer une reine sans gêner celles déjà posées. A l'inverse si elle vaut "False" alors cela veut dire que la case est menacée. A chaque pose d'une reine ses coordonnés sont stockés sous la forme d'un tuple dans un tableau. La première classe "Plateau" possède une méthode pour poser une reine, une pour récupérer le placement des reines posées, une pour trouver toutes les cases libres (cases non menacés) et enfin une dernière méthode pour copier un état de l'échiquier, ce qui nous sera utile plus tard pour le backtracking.
Il existe également une deuxième classe ici pour résoudre le problème des huit reines. Cette classe "ProblemeReines" contient la dimensions de l'échiquier et un tableau pour stocker les différentes solutions. Il y 3 méthodes utiles pour trouver les différentes solutions, une pour avoir la taille de l'échiquier, un traitant de l'algorithme principale et un dernier pour récupérer les solutions stockées.
La méthode principale créer un échiquier "vide", puis pour chaque case libre il crée un sous échiquier dans lequel il explorera à son tour chaque case libre afin de trouver une solution.
Si le nombre de reines posées est égal à la dimension de l'échiquier alors les coordonnées des reines dans la classe "Plateau" sont ajoutées au tableau des solutions de "ProblemeReines".

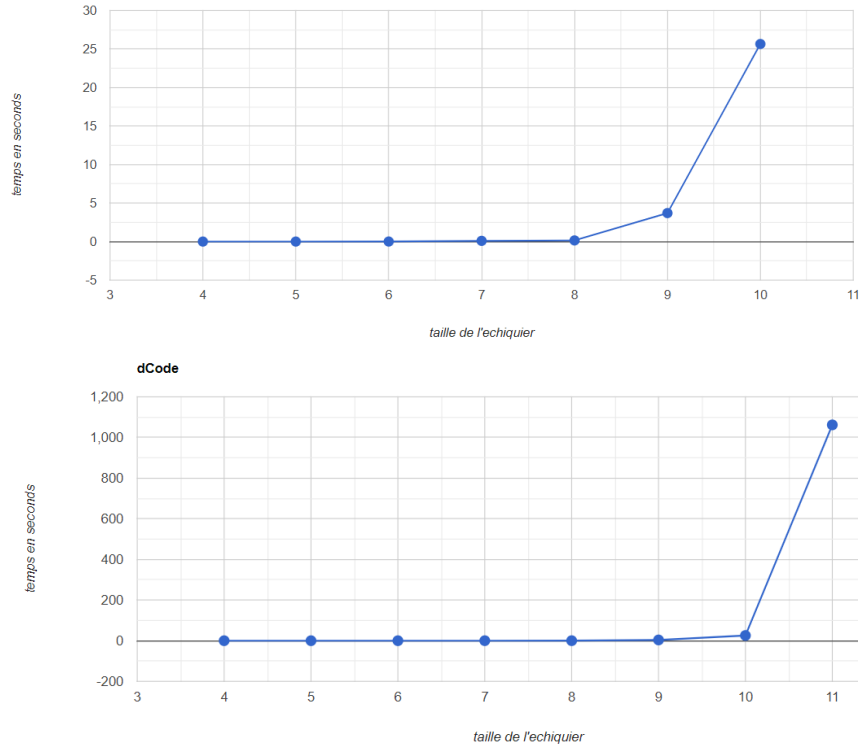
A la fin on peut utiliser la fonction “afficher” pour afficher à l’écran les solutions préalablement triés pour enlever les doublons.
Le problème avec cet algorithme est qu’il peut devenir très long si les dimensions augmentent. Par exemple pour 7*7 les calculs prennent une quinzaine de seconde alors que pour du 8*8 cela peut prendre plusieurs minutes avant de terminer son exécution. Cela est dû au fait que l’on parcourt plusieurs fois les mêmes cas de figure même si ce ne sont pas des solutions. Cela diffère du deuxième programme qui fonctionne par colonne et donc ne rencontre pas le problème du même paterne qui se répète.
Pour améliorer ce programme on pourrait déjà diviser les solutions que l’on parcourt au départ en se limitant au quart supérieur gauche pour les premières reines. Sinon on pourrait aussi regarder si on a déjà rencontré ce paterne pour ne pas avoir à le refaire à l’aide d’un arbre.

Programme 2:

Dans le second programme qui a un algorithme procédural récursif qui va placer les reines colonnes par colonnes et revenir en arrières si le placement est impossible .On crée une grille vide de la taille souhaitée puis la fonction “BonnePos” vérifiant si on peut placer une reine en regardant si elle n’est pas déjà alignée avec une autre sur la ligne, la colonne ou la diagonale. Une fois placé, on met les autres cases à 0 et on passe à la prochaine colonne. Une fois qu'une reine peut être placée dans la dernière colonne c’est que ce paterne de reines est une solution.
En revanche,si aucune reine ne peut être placée sur toute la colonne c’est que ce paterne n’est pas bon, donc on revient en arrière d’une colonne
Ensuite, dès qu'une solution ou une mauvaise solution est trouvée, elle est envoyée dans des listes distinctes pour les sauvegarder.
Pour tester si mon algorithme était performant j’ai donc regardé sa complexité en temps et également en taille.
Pour la complexité en temps je me suis référé à cette courbe qui nous montre si la complexité est bonne ou non.



https://buzut.net/cours/computer-science/time-complexity



Quand on compare les 2 courbes on remarque que mon algorithme à une très mauvaise complexité car la courbe est comparable a O(n^2). Malheureusement, le problème ne vient pas forcément de l'algorithme en lui-même mais plus du problème qu'il doit résoudre.En effet, à chaque fois que nous augmentons la taille de l'échiquier de 1 cela augmente considérablement le nombre de tests à faire donc selon moi peut importe que l'on exécute un code très optimisé ou non cela ne changera pas grand chose.
Pour la complexité en espace on va se retrouver encore une fois dans le même cas de figure. On passe de 724 solutions pour un échiquier en 10x10 , 2680 solutions en 11x11 puis 14200 solutions en 12x12. La courbe est là aussi plutôt exponentielle ce

qui va grandement complexifier la résolution à chaque augmentation de la taille de l'échiquier mais également occupé beaucoup d'espace disque.
Pour améliorer le programme je pense qu'on pourrait supprimer la vérification de la colonne car étant donné que le programme place les reines colonne par colonne, il ne peut pas déjà y avoir une reine dans la colonne où on doit placer une reine.Même si comme on n'a pu le voir précédemment ce ne sera pas très utile.
Pour améliorer sa complexité en espace je pourrais supprimer la sauvegarde des mauvais paterne car j'ai beaucoup plus de mauvais paterne à sauvegarder que de bon.

Comparaison:

Le second algorithme est donc bien meilleur dû à son court temps d'exécution il évite la redondance d'un même cas de figure ce qui réduit considérablement le nombre de calculs. De même pour trouver les cases libres car le deuxième programme ne change qu'une seule fois la valeur du tableau d'échiquier pour savoir si la case est libre alors que le premier change plusieurs fois la même valeur du tableau et ce sur chaque ligne, colonne et diagonale. Et il renseigne même des informations qui seront inutiles pour la suite.

Conclusion:

Bien que des optimisations d'algorithme puissent améliorer la vitesse de résolution du problème des reines, la croissance exponentielle du nombre de combinaisons possibles à explorer rend le temps de résolution long des échiquiers de grande taille.Peut un porte les optimisations,il est difficile d'obtenir une solution rapide et simple pour des échiquiers de grande taille.