

Compte Rendu de Projet : Problème des huit reines

Benjamin DUMONT-GIRARD, Guéno  PHILIPPON, Lucas TORRES, Tom MORICEAU 1C2

Programme 1 (Benjamin) :

Pour r soudre le probl me des N-Reines, j'ai opt  pour une approche simple mais efficace en utilisant un algorithme de backtracking. Le programme est organis  autour de deux fonctions principales: "is_safe" et "backtrack".

Dans un premier temps, je souhaitais r diger le programme en anglais comme j'aime le faire dans d'autres langages afin de m'entra ner   cette pratique courante en entreprise.

Malheureusement, devant la complexit  de l'impl mentation de cette m thode dans un langage que je ne ma trise pas assez, j'ai d  me r soudre   en laisser une bonne partie en fran ais afin de m'aider dans le debugging et de me permettre de faire un programme propre, minimaliste et fonctionnel .

La fonction "is_safe" v rifie si une reine peut  tre plac e en toute s curit  sur une certaine case de l' chiquier en v rifiant les lignes, les colonnes et les diagonales. La fonction "backtrack" utilise une approche r cursive pour placer les reines sur l' chiquier en explorant diff rentes configurations.

En termes de structure de programme, tout se d roule dans une seule fonction, "solve_n_queens". En lan ant le programme et en sp cifiant la taille de l' chiquier, le programme r sout le probl me des N-Reines en cons quence.

Concernant les performances, le temps d'ex cution varie en fonction de la taille de l' chiquier. Pour une grille de 8x8, par exemple, le temps d'ex cution est raisonnable. Pour des grilles plus grandes, le temps peut augmenter significativement, mais le programme reste efficace pour des tailles raisonnables d' chiquier qui ne d passent pas les grilles de 20 par 20 cases..

Bien que le programme soit basique, il fournit une base solide pour r soudre le probl me des N-Queens.

Le programme utilise une approche de retour en arri re pour trouver des solutions valables. Bien qu'il puisse exister des m thodes plus avanc es pour r soudre ce probl me, mes connaissances en python ne me permettaient pas de faire un programme beaucoup plus complexe. j'ai donc opt  pour cette m thode que je connaissais d j .

Programme 2 (Guénoilé) :

Ce programme résout le problème classique des N-reines, où le but est de placer N reines sur un plateau de taille $N \times N$ de telle sorte qu'aucune reine ne puisse se menacer en ligne, en colonne ou en diagonale. Le programme utilise une approche récursive pour explorer toutes les configurations possibles et trouver toutes les solutions valides. Les techniques de résolution sont le BackTracking.

Fonctionnalités :

1. Demander la taille du plateau de jeu : inviter l'utilisateur à saisir la taille du plateau de jeu.
2. Vérifier si la reine est en sécurité : La fonction vérifie si la reine peut être placée en toute sécurité dans la case donnée sans menacer une autre reine déjà placée.
3. Afficher les solutions : Le programme affiche toutes les solutions valides pour le jeu N Queens.
4. Demande de vérification des coordonnées de la reine : L'utilisateur peut préciser les coordonnées de la reine pour vérifier le nombre de fois qu'elle apparaît dans la solution.
5. Comptez le nombre de fois qu'une reine apparaît : La fonction compte le nombre de fois qu'une reine donnée apparaît dans la solution.
6. Afficher les résultats de l'occurrence de la reine : Affichez les solutions dans lesquelles une reine spécifique apparaît.
7. Afficher toutes les solutions : permet à l'utilisateur de visualiser toutes les solutions trouvées.
8. Afficher le temps d'exécution : Le programme affiche le temps d'exécution du CPU nécessaire pour trouver toutes les solutions.

Ce programme fournit une implémentation puissante et efficace pour résoudre le problème des N-reines. Il permet à l'utilisateur d'interagir en spécifiant la taille de la carte et en visualisant les solutions pour des configurations spécifiques. Grâce à une approche récursive entièrement optimisée, le programme peut gérer des palettes de plus grande taille tout en fournissant des résultats précis et rapides.

Temps:

grille 6*6	0.0013144039999999982 secondes
grille 8*8	0.023177438999999998 secondes
grille 12*12	16.464367887 secondes

Programme 3 BFS (Lucas) :

Ce programme Python est spécialement conçu pour résoudre le problème des N-Reines en prenant en compte certaines contraintes. Il emploie le BFS, une méthode de parcours efficace pour explorer systématiquement toutes les configurations potentielles.

Le programme calcul donc toutes les possibilités possibles pour les grilles 6x6, 8x8 et 12x12. Ensuite, il affiche le nombre de solutions, et le temps d'exécution. Il permet également d'afficher les solutions pour offrir un aperçu visuel à l'utilisateur.

Pour rentrer plus précisément dans le fonctionnement du BFS et le fonctionnement de mon programme voici son processus de résolution. Tout d'abord l'algorithme prend pour référence 3 grilles (6x6), (8x8) et (12x12). Ensuite la fonction "est_valide" vérifie si le placement d'une nouvelle reine est valide et aucune reine déjà placée sur l'échiquier n'est dans la même colonne, ni sur les mêmes diagonales que la nouvelle reine. "Bfs_n_reines" utilise un algorithme BFS pour explorer systématiquement toutes les configurations possibles. Elle commence avec un échiquier vide, représenté par une liste vide, et explore toutes les possibilités de placement de reines colonne par colonne. À chaque étape, si un placement est valide, il est ajouté à la file d'attente pour une exploration plus approfondie. Enfin, lorsqu'une solution est trouvée, elle est ajoutée à la liste des solutions puis est affichée.

L'intégration du temps CPU utilisé par nos différents programmes offre un moyen pratique de comparer leurs performances respectives.

Pour conclure, j'ai choisi l'algorithme de résolution (BFS) car la création de l'algorithme me paraissait intéressante surtout que par rapport à toutes les approches de mon groupe c'est la plus légère et la plus rapide avec un temps de résolution inférieur à ceux de tous les autres programmes. On en a donc conclu que c'était l'approche la plus adaptée si l'objectif était un gain de performance. De plus une deuxième version du programme est disponible et implémente un affichage graphique avec la modélisation d'un échiquier pour l'utilisateur ainsi qu'un menu pour choisir la taille de l'échiquier parmi celles proposées.

Les temps de résolution :

Grille 6x6 : 0.0003 secondes

Grille 8x8 : 0.0053 secondes

Grille 12x12 : 3.4611 secondes

Programme 4 (Tom) :

J'ai basé mon programme de résolution du problème des N-Reines sur la Programmation Orientée Objet, approche qui peut paraître étonnante au premier abord mais qui m'a permis de garder une certaine lisibilité dans mon code, du fait d'avoir divisé mon programme en deux fichiers (un fichier de classes, et un fichier main).

Pour ma méthode de résolution j'ai opté pour le backtracking que l'on retrouve dans la méthode *résoudre* de la classe *Échiquier*, qui fait appel à la fonction *est_menacee* qui vérifie si la reine que l'on souhaite placer n'entre pas en conflit avec les reines déjà positionnées sur le plateau d'après les règles des échec.. Si ce n'est pas le cas on place la reine à l'aide de *placer_reine*, et après avoir terminé l'appel récursif de *résoudre* et si l'on entre toujours pas dans la condition d'arrêt, on appelle la fonction *retirer_reine* et on réessaye avec un positionnement de la reine précédente différent, jusqu'à trouver une solution qui est ensuite ajoutée à la liste des solutions.

En ce qui concerne le fichier principal (*main.py*), on retrouve une fonction *main* assez classique qui va demander à l'utilisateur la taille du plateau, s'il veut placer la première reine ou non, puis afficher toutes les solutions disponibles en respectant les conditions données. On se retrouve ensuite face à un menu venant de la fonction *menu* appelée à la fin du *main*, qui nous propose :

- D'essayer avec une autre valeur (on relance alors la fonction *main*)
- D'afficher graphiquement les résultats, j'ai alors utilisé la bibliothèque *matplotlib* pour afficher d'abord un premier affichage, et on demande ensuite à chaque fois à l'utilisateur si il veut voir la prochaine solution (pour éviter un affichage graphique de 14200 solutions pour une grille 12x12, par exemple).
- D'essayer avec une valeur aléatoire (comprise entre 1 et 10) grâce à la bibliothèque *random*
- De quitter le programme

En termes de performances, mon programme reste assez lent, on voit par exemple que pour 8 reines, il met 0.0184 secondes, qui reste évidemment plus lent que le BFS de Lucas par exemple, pour une raison évidente d'efficacité.

Quelques statistiques de temps d'exécution:

- **6 reines** : 0.001 secondes
- **8 reines** : 0.0184 secondes
- **10 reines** : 0.5314 secondes
- **12 reines** : 18.6646 secondes

Les aspects uniques de mon programme restent donc l'utilisation de la Programmation Orientée Objet et le menu interactif qui propose plusieurs choses à faire, puisque mon but était de pousser l'engagement utilisateur au maximum.