

SAE 2.02

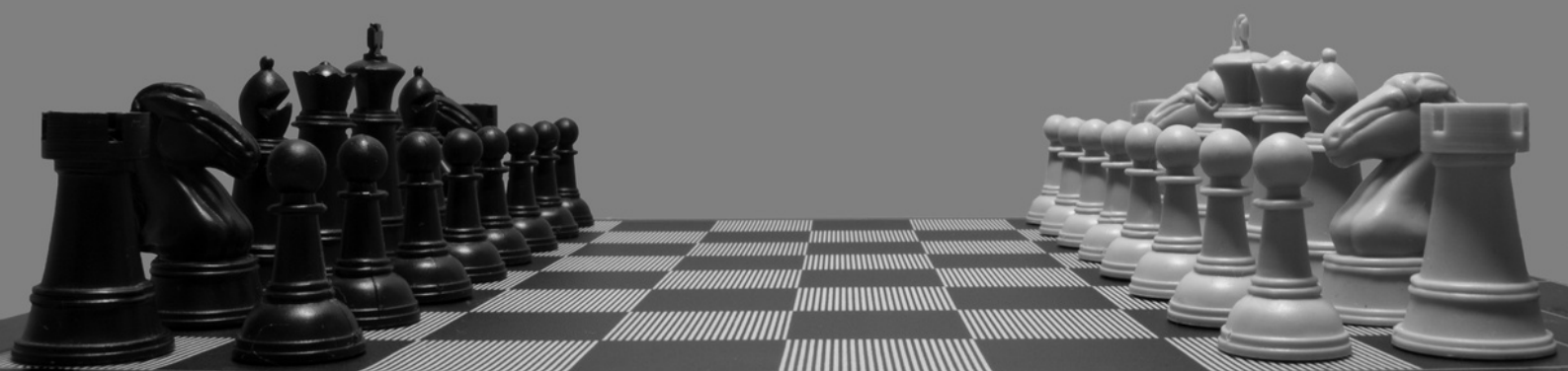
EXPLORATION ET COMPARAISON ALGORITHMIQUE



Le problème des huit dames



Info - 1A
Yanis PONTTHOU | Yohann MEAR | Dorian POLICE





SOMMAIRE

I. Introduction

II. Description Algorithme n°1

III. Description Algorithme n°2

IV. Comparaison des Algorithmes

V. Conclusion



I. INTRODUCTION







Le problème des huit dames



Le but du problème des huit dames est de placer huit dames d'un jeu d'échecs sur un échiquier de 8×8 cases sans que les dames puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée).

Par conséquent, deux dames ne doivent jamais partager la même rangée, colonne, ou diagonale.

I. INTRODUCTION

Descriptif du problème:

Le problème des 8 reines n'est pas nouveau. Pendant de nombreuses années, beaucoup de mathématiciens, y compris Gauss ont travaillé sur ce problème, qui est un cas particulier du problème généralisé des n-dames, posé en 1850 par Franz Nauck, et qui consiste à trouver les positions de l'échiquier où les reines ne se menacent pas entre elles. Ainsi, chaque dame ne peut être sur la même ligne, même colonne et même diagonale d'une autre reine, car dans ce cas elles se menaceraient entre elles.

Afin de pouvoir résoudre ce problème, il faut ainsi analyser tous les cas, pour un échiquier de taille $n \times n$ (exemple 8×8 qui admet 92 solutions), avec le nombre de dames qui peuvent être placées en même temps sur l'échiquier, ainsi que leurs positions qu'on représentera grâce au programme. Pour répondre à ce problème, on peut donc utiliser un arbre de décision, pour représenter les cases de l'échiquier où on peut placer une nouvelle reine après en avoir déjà placé une. Un arbre de décision est un schéma qui représente les possibilités de choix qui sont interconnectées, sous la forme graphique d'un arbre.

I. INTRODUCTION

Processus de résolution du problème:

Premièrement nous avons étudié le sujet en groupe, puis chaque membre du groupe à fait ses propres recherches concernant le problème : articles, vidéos, etc... puis nous avons choisis de coder un premier algorithme, qui utilise le principe du backtracking, qui nous permet de trouver toutes les solutions pour un échiquier de taille $n*n$. Un algorithme de backtracking est un type d'algorithme qui explore toutes les possibilités et qui effectue un retour en arrière lorsque la solution proposée n'est pas la bonne. Nous avons donc chacun de notre côté codé cette même base afin que pour les spécifications suivantes la personne qui l'ai codée ai plus d'aisance à rajouter les nouvelles fonctionnalités. Une fois la base programmée, nous avons essayé de répondre au problème : créer un algorithme qui, pour une position de dame imposée, nous permet de trouver les solutions, s'il existe des solutions pour celle-ci. C'est à partir de ce moment là que nous avons choisi deux chemins différents : le premier a pris le chemin de modifier l'algo en prenant en compte dans le processus de recherche de solutions le placement de cette première dame, et le deuxième chemin qui recherche toutes les solutions et qui récupère une fois fini les solutions contenant le placement de la dame voulu.

II. Description Algorithme n°1

Algorithme n°1

Les différentes fonctions :

- est_valide(plateau, ligne, col).
- backtrack(plateau, ligne).
- resoudre reines(n).

Fonction est_valide(plateau, ligne, col) :

Cette fonction permet de renvoyer un booléen, afin de vérifier si une dame peut être placée à la ligne et à la colonne du plateau. La fonction utilise une première boucle for, qui parcourt les lignes. On compare ensuite 3 cas possibles dans le if :

- Si une dame est déjà placée dans la même colonne (`plateau[i] == col`)
- Si une dame est déjà placée dans la même diagonale montante (d'en bas à gauche vers en haut à droite)
- Si une dame est déjà placée dans la même diagonale descendante (d'en haut à gauche vers en bas à droite)

Si jamais un de ces 3 cas est réalisé, la variable "valide" qui permet de savoir si l'on peut placer le pion dans la case prend la valeur False. On renvoie ensuite la valeur du résultat.



II. Description Algorithme n°1

Algorithme n°1

Les différentes fonctions :

- est_valide(plateau, ligne, col).
- backtrack(plateau, ligne).
- resoudre reines(n).

Fonction backtrack(plateau, ligne) :

Passons désormais, après avoir déterminé les critères pour qu'une case soit valide, à l'utilisation du backtracing. Nous définissons d'abord, au début de la fonction, le cas de base, si la ligne est égale à n . Si jamais elle l'est, on renvoie ainsi une copie du tableau, car cela voudra dire qu'on aura placé des dames à toutes les lignes sans qu'elles se menacent mutuellement (sauf dans le cas où la taille de l'échiquier n'admet aucune solution).

BACK ◀



II. Description Algorithme n°1

Algorithme n°1

Les différentes fonctions :

- est_valide(plateau, ligne, col).
- **backtrack(plateau, ligne).**
- resoudre reines(n).

Fonction backtrack(plateau, ligne) :

On crée par la suite un tableau, res, pour stocker les résultats des solutions au problème des huit reines. Si c'est la 1ère ligne (où on impose la dame), la colonne prend la valeur de la 1ère dame, on stocke ensuite la colonne où la dame est placée au sein de la ligne, puis on ajoute à la variable res les solutions possibles pour la prochaine ligne. Dans le cas où ce n'est pas la première ligne, on parcourt l'échiquier, noté n, et on fait un appel à la fonction définie au préalable, est_valide, pour vérifier qu'on peut placer la reine. On fait ensuite de même qu'auparavant, c'est-à-dire qu'on stocke la colonne où la dame est placée au sein de la ligne, puis on ajoute à la variable res les solutions possibles pour la prochaine ligne. Le résultat est ensuite renvoyé.



II. Description Algorithme n°1

Algorithme n°1

Les différentes fonctions :

- est_valide(plateau, ligne, col).
- backtrack(plateau, ligne).
- resoudre_reines(n).

Fonction resoudre_reines(n) :

Les deux précédentes fonctions sont définies dans la fonction `resoudre_reines`. Dans cette fonction, on demande d'abord dans quelle colonne on souhaite placer la 1ère dame. On initialise ensuite la variable du plateau avec : $[-1] * n$. Cela permet d'initialiser chaque case du plateau comme vide, les reines n'ayant pas encore été placées. On réutilise ensuite la fonction `backtrack` définie précédemment pour afficher les solutions possibles, en précisant comme 2ème argument la ligne à partir de laquelle les reines seront placées.



II. Description Algorithme n°1

Algorithme n°1

Les différentes fonctions :

- est_valide(plateau, ligne, col).
- backtrack(plateau, ligne).
- resoudre reines(n).

Fonction resoudre reines(n) :

On demande ensuite si l'utilisateur veut afficher toutes les solutions, ce qui permettra d'afficher les solutions à la fois sous la forme de tableau (avec des . pour les endroits où il n'y a pas de reines) mais également en affichant sous forme de liste leurs numéros de colonne. On parcourt ensuite, grâce à des boucles "for" imbriquées, le tableau contenant toutes les solutions pour ensuite les afficher une à une. Si, parmi les solutions, la dame doit être placée dans la colonne de la ligne, la lettre R (utilisée pour représenter la dame) est placée dans la colonne. Sinon, c'est un . (utilisé pour représenter les cases vides) qui est placé pour les endroits où il n'y a pas de reine.



II. Description Algorithme n°1

Algorithme n°1

Les différentes fonctions :

- est_valide(plateau, ligne, col).
- backtrack(plateau, ligne).
- resoudre reines(n).

Les erreurs et problèmes rencontrés :

- Difficultés pour comprendre ce qu'on attendait de nous, à la fois dans le programme mais également dans le compte rendu
- Difficultés pour afficher uniquement les solutions "possibles" (exemple : au départ de l'algorithme, des solutions étaient trouvées pour $n = 4$)



III. Description Algorithme n°2

Algorithme n°2

Les différentes fonctions :

- **est_valide(plateau, ligne, colonne)**
- **n_reines(n, ligne=0, plateau=None, solutions=None)**
- **affiche_solutions(sol, n, x, y)**
- **TransformerSolutionsEnCoordo(liste)**

Fonction est_valide(plateau, ligne, col) :

Cette fonction permet de renvoyer un booléen, afin de vérifier si une dame peut être placée à la ligne et à la colonne du plateau. La fonction **est_valide** fonctionne de la façon où elle va regarder à partir de la première ligne jusqu'à la ligne actuelle en prenant les dames déjà posées sur chaque ligne et vérifiant si les dames déjà posées ne rentrent pas en conflit avec la dame que l'on veut poser sur la ligne actuelle en tenant en compte la règle des échecs: premièrement en vérifiant que les deux dames ne soient pas sur la même colonne puis sur la même diagonale pour cela j'ai utilisé une propriété mathématique tirée de la géométrie dans l'espace qui est : si $|x_a - x_b| == |y_a - y_b|$, alors ils sont sur la même diagonale. Si aucun cas n'est détecté, alors la fonction renvoie **True**, car la nouvelle reine ne sera pas menacée par les précédentes reines.


queen

III. Description Algorithme n°2

Algorithme n°2

Les différentes fonctions :

- `est_valide(plateau, ligne, colonne)`
- `n_reines(n, ligne=0, plateau=None, solutions=None)`
- `affiche_solutions(sol, n, x, y)`
- `TransformerSolutionsEnCoordo(liste)`

Fonction n_reines(n, ligne=0, plateau=None, solutions=None) :

- **n** est la taille de l'échiquier et donc le nombre de reines à placer
- **ligne** est l'indice de la ligne actuelle que nous sommes en train de considérer. Elle est initialisée à 0.
- **plateau** est un tableau qui représente l'échiquier. Il est initialisé à `None`, mais lors de la première itération, on crée un tableau de longueur **n** contenant des `-1`.
- **solutions** est une liste de listes contenant toutes les configurations possibles des **n** reines sur l'échiquier. Elle est initialisée à `None`.

La fonction commence par initialiser le tableau **plateau** et la liste de **solutions** si nécessaires. Ensuite, elle vérifie si elle a atteint la dernière ligne de l'échiquier, dans ce cas, elle ajoute la configuration actuelle à la liste des solutions et retourne la fonction pour continuer la récursivité.

III. Description Algorithme n°2

Algorithme n°2

Les différentes fonctions :

- `est_valide(plateau, ligne, colonne)`
- `n_reines(n, ligne=0, plateau=None, solutions=None)`
- `affiche_solutions(sol, n, x, y)`
- `TransformerSolutionsEnCoordo(liste)`

Fonction `n_reines(n, ligne=0, plateau=None, solutions=None)` :

Si elle n'est pas à la dernière ligne, elle parcourt toutes les colonnes de la ligne actuelle, vérifie si une reine peut être placée sur cette case avec la fonction `est_valide` et, si c'est le cas, elle marque la case comme occupée en y plaçant une reine (en stockant le numéro de colonne) et appelle récursivement la fonction `n_reines` pour passer à la ligne suivante. Si la fonction `est_valide` renvoie `False` pour toutes les colonnes de la ligne actuelle, alors il n'y a pas de solution possible avec cette configuration, donc l'algorithme retourne la fonction et vu que nous sommes sur de la récursivité ça veut dire que l'on ferme une boîte parmi toutes les boîtes ouvertes et que l'on retourne en arrière en essayant avec une nouvelle solution.



III. Description Algorithme n°2

Algorithme n°2

Les différentes fonctions :

- `est_valide(plateau, ligne, colonne)`
- `n_reines(n, ligne=0, plateau=None, solutions=None)`
- **`affiche_solutions(sol, n, x, y)`**
- **`TransformerSolutionsEnCoordo(liste)`**

Fonction `affiche_solutions(sol, n, x, y)` :

Cette fonction est très simple, elle va parcourir les solutions contenues dans la liste `sol`, puis elle va trouver et ajouter à la liste `l` les solutions pour lesquelles le positionnement de la dame passé en paramètre est présent et vérifié. Cette fonction permet de connaître les solutions pour une dame donnée parmi une liste de solutions.

`TransformerSolutionsEnCoordo(liste)` :

Cette fonction permet de transformer une liste qui représente une solution en coordonné pour permettre une compréhension du résultat plus simple. On se base sur le fait que nous avons créé nos liste de solutions avec l'indice de la liste = la ligne.



IV. Comparaison des Algorithmes

Les deux algorithmes utilisent la même logique afin de trouver les positionnements des reines.

Cependant ils ne répondent pas de la même manière au problème demandé. Le premier va permettre à l'utilisateur de rentrer la colonne de la première ligne pour la première dame, ce qui permet d'éviter un nombre important de calculs car il limite les solutions possibles. Le second part du principe de calculer toutes les solutions possibles pour un échiquier de taille n puis ensuite récupère parmi toutes les solutions celles qui admettent une dame aux coordonnées souhaitées.

Voilà la grosse différence entre les deux algorithmes. Au niveau des inconvénients, le premier algorithme ne permet pas d'avoir les solutions pour une dame qui est placée au-delà de la première ligne. L'inconvénient du second algorithme est qu'il est très lourd : plus nous allons augmenter la taille de l'échiquier, plus le temps de réponse est long, car il va s'occuper de trouver toutes les solutions pour un échiquier, contrairement au premier algo qui va trouver toutes les solutions mais en limitant les possibilités à la première ligne, ce qui va donc permettre sur des échiquiers plus grand, et grâce au principe de la récursivité de limiter le temps d'exécution.

IV. Comparaison des Algorithmes

Prenons un exemple :

Si nous imposons une reine sur la colonne 0 et la ligne 3, l'algorithme n°2 va tester toutes les solutions de l'échiquier sans prendre en compte la dame imposée, et, par la suite, afficher toutes les solutions possibles dans le cas où la dame est imposée à la colonne 0 et à la ligne 3.

Exemple de résultat : Pour n = 8

[2, 5, 7, 0, 4, 6, 1, 3], [2, 5, 7, 1, 3, 0, 6, 4]

Solution 12:

.	.	.	R
.	R	.	.	.
.	R
R
.	R
.	R	.	.
.	R
.	.	.	.	R

Solution 13:

.	.	.	R
.	R	.	.	.
.	R
.	R
.	.	.	.	R
R
.	R	.	.
.	R

V. Conclusion

Conclusion :

En conclusion, comme amélioration nous avons pensé que nous aurions pu incorporer au sein du premier algorithme, qui utilise le backtracking, une solution pour permettre de placer une reine sur une autre ligne. Cela permettrait ainsi, comme dans le deuxième algorithme, d'afficher les solutions si l'on désire placer la reine sur une autre ligne. Malgré tout, nous sommes très satisfait du premier algorithme qui utilise le backtracking, car il est plus efficace pour les recherches de solutions, notamment lorsque l'échiquier est plus grand. Cela peut s'expliquer car le deuxième algorithme parcourt une fois de plus l'échiquier, partant de l'emplacement où on place la 1ère dame.



V. Conclusion



Conclusion :

L'avantage principal du 1er algorithme est sa rapidité : le deuxième algorithme est en moyenne 3x plus long à exécuter pour un échiquier de taille 12*12.

Si on prend le cas le plus "logique", où l'utilisateur saisit la première dame sur la première ligne, le premier algorithme est ainsi plus efficace. Malgré tout, si on veut connaître les solutions pour le cas où la dame est placée sur une autre ligne, il faut privilégier le deuxième algorithme car le premier n'affiche pas les reines sur les lignes qui précèdent la 1ère dame imposée. Chaque algorithme est ainsi très efficace pour ce qu'on lui demande : si l'on veut un algorithme plus rapide, mais où la première dame doit être placée uniquement sur la première ligne, mieux vaut choisir le premier algorithme. Dans l'autre cas, où la vitesse d'exécution de l'algorithme n'est pas primordial, l'algorithme numéro 2 sera préférable. Il est malgré tout important de préciser que, pour un échiquier plus petit, comme un échiquier de 4*4, la différence de vitesse d'exécution est légère..

