Université des Sciences et de la Technologie
Houari Boumediene Faculté Informatique

TP Métaheuristiques

Problème des N-Reines

Élèves :
Manel OUCHAR
Zakaria YOUSFI
SII G03

4 juillet 2023

Table des matières

Table des figures

Chapitre 1

Introduction Générale

Le problème des n-reines est un défi fascinant dans le domaine de la science informatique, des mathématiques et de la recherche opérationnelle. Ce problème, qui consiste à placer n reines sur un échiquier de taille n x n de manière à ce qu'aucune reine ne puisse menacer une autre, peut sembler simple, mais en réalité, il est très complexe et nécessite des algorithmes sophistiqués pour être résolu efficacement.

Dans ce rapport de projet, nous allons dans un premier temps, explorer les différentes stratégies de recherche en espace d'états : méthodes de recherche aveugles et des méthodes utilisant des heuristiques avec une explication de leur complexité et des concepts clés tels que l'état, les opérateurs et les successeurs.

Dans un second temps, nous allons nous concentrer sur le problème des N-Reines en décrivant la modélisation utilisée pour représenter une instance du problème et sa fonction d'évaluation pour calculer le nombre de reines en attaque.

Ensuite, nous allons décrire notre implémentation en Java des différents algorithmes pour le problème, ainsi que les différentes classes implémentées.

En expérimentant les différentes méthodes de résolution de ce problème, nous allons présenter les résultats de nos expérimentations, nous allons également discuter des avantages et des limites de chaque méthode en présentant une analyse comparative de la performance de ces dernières. Aussi, Nous allons fournir une interface graphique pour visualiser les résultats de manière plus intuitive.

Enfin, nous allons conclure notre travail en résumant les résultats obtenus et en proposant des perspectives pour des recherches futures dans le domaine des problèmes d'optimisation et de satisfaction de contraintes.

Chapitre 2

Approche Espace des États

2.1 État, Opérateurs et Successeurs

Une approche à espace des états est une méthode qui consiste à représenter le problème de façon symbolique, puis à trouver les relations entre les différents états possibles du système afin de déterminer la solution optimale[1].

Dans cette dernière, un **état** représente la situation du problème à un moment donné. Les **opérateurs** sont des actions qui transforment un état en un autre état. Les **successeurs** sont les états atteignables à partir de l'état actuel en appliquant les opérateurs disponibles[2][3].

Il existe de nombreuses méthodes de résolution dans une approche à espace d'états. Dans notre étude, on s'intéresse aux suivantes :

- **Méthodes Exactes (aveugles) :**
  - Recherche en profondeur d'abord (DFS).
  - Recherche en largeur d'abord (BFS).
- **Méthodes Heuristiques :**
  - Recherche A*.

2.2 Recherche en profondeur d'abord (DFS)

2.2.1 Description

La **Recherche en Profondeur D'abord (DFS)** est un algorithme de parcours d'arbre ou de graphe qui explore le plus profond chemin possible jusqu'à atteindre un nœud feuille, avant de revenir en arrière et d'explorer les autres chemins. Il commence à la racine de l'arbre ou à un nœud arbitraire et utilise une pile pour stocker les nœuds visités et à visiter. Lorsqu'un nœud est visité, il est marqué comme tel pour éviter qu'il ne soit visité à nouveau.[4][5]

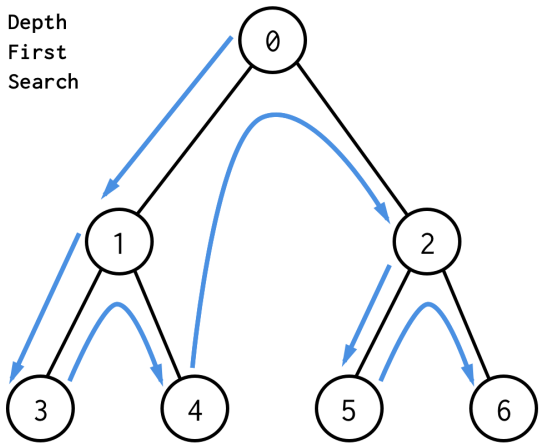

FIGURE 2.1 – Recherche en profondeur d'abord (DFS) sur un arbre binaire.

2.2.2 Pseudo-code

Ci-dessous le pseudo-code de cet algorithme :

```
fonction parcoursEnProfondeur(noeud: etat initiale):
    On initialise une pile vide pour stocker les noeuds a visiter.

    marquer noeud comme visite;
    pile.empiler(noeud);

    tant que nonVide(pile) faire
        //Recuperation du dernier element de la pile
        noeud_actuel = pile.depiler();

        //Traitement du noeud actuel
```

```
12              si(etatBut(noued_actuel)) alors
13                  retourner(noued_actuel)
14              fsi;
15
16              //Ajout des enfants a la pile
17              pour chaque enfant enf de noeud_actuel faire
18                  si enfant n'a pas ete visite alors
19                      marquer enfant comme visite;
20                      pile.empiler(enf);
21                  fsi;
22              fait;
23          fait;
24      fin.
```

Listing 2.1 – Pseudo-code de l'algorithme de recherche en profondeur d'abord DFS.

2.2.3 Complexité

Lorsqu'on utilise la méthode BFS pour explorer un système d'états, la complexité temporelle dépend principalement du nombre d'états possibles dans le système et du nombre d'opérateurs de transition entre les états.

Plus précisément, la **complexité temporelle** de BFS pour un système d'états est de l'ordre de $\mathbf{O}(b^d)$, ou b est le facteur de branchement moyen du système (c-à-d le nombre moyen d'opérateurs de transition par état) et d est la profondeur maximale du système (c-à-d la distance maximale entre l'état initial et l'un des états cibles).

En d'autres termes, la complexité temporelle de BFS dans un système d'états augmente de manière exponentielle avec la profondeur du système et le facteur de branchement moyen. Cela signifie que si le facteur de branchement moyen est élevé et la profondeur maximale du système est grande, le temps de calcul nécessaire pour trouver une solution peut devenir très élevé.

La **complexité spatiale** de DFS pour un système d'états est de l'ordre de $\mathbf{O}(b * d)$, où b est le facteur de branchement moyen du système et d est la profondeur maximale du système. Cela est dû au fait que DFS empile chaque nœud visité sur une pile, et dans le pire des cas, la pile peut contenir jusqu'à d nœuds en même temps. Donc la complexité spatiale est proportionnelle à la profondeur maximale d ainsi qu'au facteur de branchement moyen b.

2.3 Recherche en largeur d'abord (BFS)

2.3.1 Description

La **Recherche en Largeur D'abord (BFS)** est un algorithme de parcours qui explore tous les nœuds d'un arbre ou d'un graphe en visitant tous les voisins du nœud courant avant de passer aux voisins des voisins.[6][7]

L'algorithme de recherche en largeur d'abord utilise une file d'attente pour stocker les nœuds à visiter. Au début, la racine de l'arbre est placée dans la file d'attente. Ensuite, l'algorithme retire un nœud de la file d'attente, visite tous ses enfants et les ajoute à la fin de la file d'attente. L'algorithme continue de retirer des nœuds de la file d'attente jusqu'à ce qu'elle soit vide, ce qui signifie que tous les nœuds ont été visités.

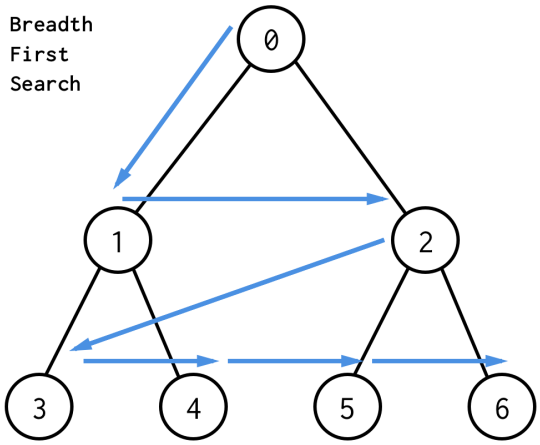

FIGURE 2.2 – Recherche en largeur d'abord (BFS) sur un arbre binaire.

2.3.2 Pseudo-code

Ci-dessous le pseudo-code de cet algorithme :

```
fonction parcoursEnLargeur(noeud: etat initiale):
    On initialise une file vide pour stocker les noeuds a visiter.

    marquer noeud comme visite;
    enfiler(file, noeud);

    tant que nonVide(file) faire
        // On retire le premier noeud de la file
        noeud_actuel = defiler(file);
```

```
10
11              //Traitement du noeud actuel
12              si(etatBut(noued_actuel)) alors
13                  retourner(noued_actuel)
14              fsi;
15
16              //Ajout des enfants a la file
17              pour chaque enfant de noeud_actuel faire
18                  si enfant n'a pas ete visite alors
19                      marquer enfant comme visite;
20                      enfiler(file, enfant);
21                  fsi;
22              fait;
23          fait;
24      fin.
```

Listing 2.2 – Pseudo-code de l'algorithme de recherche en largeur d'abord BFS.

2.3.3 Complexité

La **complexité temporelle** de BFS pour un système d'états est de $\mathbf{O}(b^d)$, où b est le facteur de branchement moyen du système et d est la profondeur maximale de la recherche. Cela signifie que l'algorithme visitera chaque état du système au plus une fois, et le temps nécessaire pour le faire sera proportionnel au nombre total d'états dans le système, qui est exponentiel dans le pire des cas.

La **complexité spatiale** de BFS pour un système d'états est également de $\mathbf{O}(b^d)$, car il doit maintenir une file d'attente des états non explorés qui peut potentiellement contenir chaque état du système à un moment donné pendant la recherche. En pratique, cependant, l'utilisation réelle de l'espace de BFS peut être considérablement inférieure à cette limite de pire cas, surtout si le système a un facteur de branchement élevé mais une profondeur maximale relativement faible.

2.4 Recherche A*

2.4.1 Définition d'heuristique

En algorithmique, une **heuristique** est une méthode de calcul qui fournit rapidement une solution, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile. Une heuristique s'impose quand les algorithmes de résolution exacte (DFS, BFS..etc)

sont impraticables, à savoir de complexité polynomiale ou exponentielle. Généralement, une **heuristique** est conçue pour un problème particulier, en s'appuyant sur sa structure propre, mais il existe des approches fondées sur des principes généraux.[8]

2.4.2 Description

La recherche **A*** est un algorithme de recherche de chemin dans un graphe ou un arbre, entre un nœud initial et un nœud final. Il utilise une fonction heuristique $\boldsymbol{h}$ sur chaque nœud pour estimer le coût restant pour atteindre le nœud final, en plus du coût déjà accumulé depuis le point de départ pour atteindre ce nœud $\boldsymbol{g}$.[9][10]
Donc, la fonction d'évaluation d'un noeud n sera de la forme : $\mathbf{f(n) = g(n) + h(n)}$

Ainsi, l'algorithme choisit de parcourir en premier les nœuds ayant le coût total le plus faible **f(n)** afin de se rapprocher du point d'arrivée plus rapidement.

2.4.3 Pseudo-code

Ci-dessous le pseudo-code de cet algorithme :

```
1  fonction A*(root: noued l'etat initiale, cout: fonction de cout, h:
   fonction heuristique):
2      initialiser une file ouvert et y ajouter le noeud racine;
3      initialiser un hashset ferme;
4
5      tant que nonVide(ouvert) faire
6          //Recuperation du premier element de la file
7          noeud_actuel = ouvert.tete();
8          ajouter noued_actuel dans ferme
9
10         //Traitement du noeud actuel
11         si(etatBut(noued_actuel)) alors
12             retourner(noued_actuel)
13         fsi;
14
15         //Ajout des enfants a ouvert
16         pour chaque enfant enf de noeud_actuel faire
17             enf = noeud.create()
18             enf.cost = noued_actuel.cost + cout(noued_actuel,enf)
19             enf.h = calculerHeuristic(enf)
20             enf.f = enf.cost + enf.h
21             si (ferme ne contient pas enf) alors
22                 ouvert.enfiler(enf);
23             fsi;
```

```
            fait;

            trier ouvert selon la valeur de f
        fait;
    fin.
```

Listing 2.3 – Pseudo-code de l'algorithme A*.

2.4.4 Complexité

La complexité de l'algorithme **A*** dépend principalement de la qualité de la fonction heuristique utilisée et de la taille de l'arbre à explorer.

Dans le pire des cas, la complexité temporelle de l'algorithme est exponentielle, ce qui peut le rendre inéfficace pour les arbres larges et peu profonds, avec un grand nombre de nœuds de décision à chaque niveau. Cependant, dans la pratique, l'algorithme **A*** est généralement très efficace pour trouver le chemin le plus court dans des arbres de taille raisonnable.

De plus, il existe des variantes de l'algorithme, comme l'**A* itératif** et l'**A* bidirectionnel**, qui peuvent améliorer la performance dans certains cas particuliers.

Chapitre 3

Le problème des N-reines

3.1 Description du problème

3.1.1 Historique

Le problème des **N-Reines** est un problème mathématique qui consiste à placer N reines sur un échiquier de taille N x N sans qu'aucune ne puisse menacer une autre.
Ce problème a été formulé pour la première fois en 1848 par le mathématicien allemand ***Max Bezzel***, mais c'est l'anglais ***William Rowan Hamilton*** qui lui a donné son nom actuel en 1850. Depuis lors, de nombreux mathématiciens ont travaillé sur ce problème et ont proposé différentes méthodes pour le résoudre. Le problème des **N-Reines** est considéré comme un exemple classique de problème de placement et il continue d'être étudié aujourd'hui dans le cadre de la théorie des graphes et de l'informatique théorique[11].

3.1.2 Contraintes

La règle de base du problème est de placer N reines sur un échiquier de taille N x N sans qu'elles ne se menacent mutuellement. Chaque reine doit être placée sur une ligne différente, une colonne différente et aucune diagonale ne doit contenir plus d'une reine[12].

Cette contrainte est représentée dans l'échiquier de la figure 3.1. Nous utilisons l'indexation usuelle des tableaux, autrement dit la case de coordonnées (i,j) est située sur la i-ème ligne et la j-ème colonne dans le sens de lecture. Les cases menacées par la reine placée dans la case (4,4) sont marquées par un croix verte.

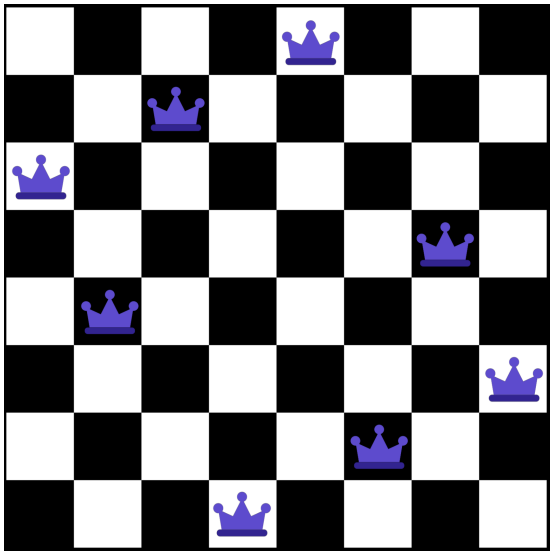

FIGURE 3.1 – Solution de 8 Reines.

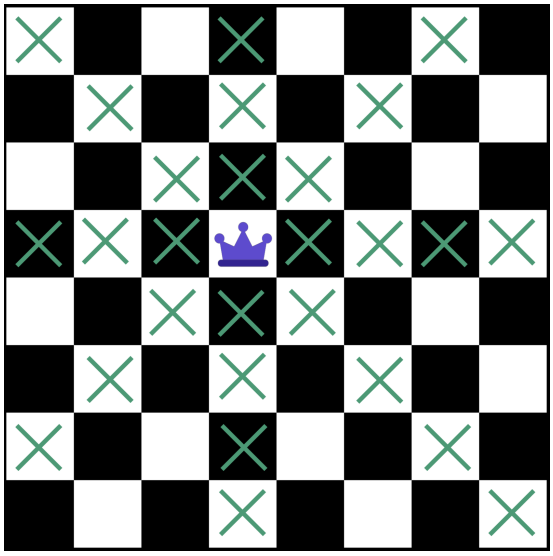

FIGURE 3.2 – La reine dans la case (4,4) menace les cases marquées d'une croix sur l'échiquier.

Le but de ce problème est de trouver toutes les solutions possibles pour placer les reines en respectant la règle citée auparavant. On remarque que le nombre de solutions possibles augmente rapidement avec la taille de l'échiquier.

3.2 Structures de données

L'une des représentations courantes du problème des N-Reines est l'utilisation d'un

vecteur pour représenter la configuration des reines sur l'échiquier, tel qu'on attribut une colonne à chaque reine puis on essaye toutes les attributions de lignes possibles sur cette colonne. Donc, pour **n = 8** on aura à faire **n ! = 8 ! = 40320 permutations**.

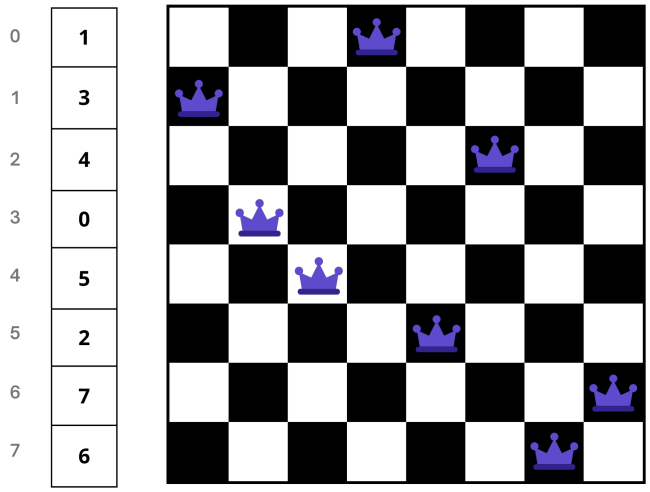

FIGURE 3.3 – Modélisation d'une solution de 8 reines.

Cette représentation est simple et efficace, car elle permet de facilement représenter l'emplacement des reines sur l'échiquier et de vérifier si la configuration est valide. Contrairement à la représentation de la matrice entière qui est plus couteuse car pour **n = 8** reines on aura **n² ! / n !(n² - n) ! = 64 ! / (8 ! * 56 !) = 4 426 165 368 possibilitées** de placements de ces dernières sur l'échiquier. [13]

Ainsi, chaque élément du tableau représente la position d'une reine sur une rangée spécifique de l'échiquier comme le montre l'exemple ci-dessous.

En utilisant cette représentation, on peut facilement vérifier si deux reines se menacent mutuellement en calculant la différence absolue entre leurs positions en colonne et en ligne. Si cette différence est égale à 0 ou n (ou l'un de leurs multiples), alors les reines se menacent mutuellement et la configuration n'est pas valide.

3.3 Fonction d'évaluation

Afin de déterminer si une configuration donnée de n reines sur un échiquier de n x n cases est une solution valide ou non, on fait appel à une fonction d'évaluation. Cette dernière vérifie si aucune reine ne peut attaquer une autre reine, c'est-à-dire si aucune paire de reines ne se trouve sur la même ligne, la même colonne ou la même diagonale.

Si la configuration donnée satisfait cette condition, alors elle est considérée comme une solution valide.

Ci-dessous le pseudo-code de cette fonction :

```
fonction evaluer(echiquier):
    conflits = 0;

    Pour i de 0 a taille(echiquier) - 1 faire
        Pour j de i+1 a taille(echiquier) - 1  faire

            // abs() est une fonction qui retourne la valeur absolut
            Si (echiquier[i] == echiquier[j]) OU
                abs(echiquier[i] - echiquier[j]) == abs(i - j) alors
                    conflits = conflits + 1;
            fsi;
        fait;
    fait;
    retourner conflits;
FIN.
```

Listing 3.1 – Pseudo-code de la fonction d'évaluation.

Chapitre 4

Implémentation en Java

4.1 Classes générales

4.1.1 Node

Cette classe définit le nœud qui est utilisé pour représenter un état de l'échiquier lorsqu'on utilise les algorithmes DFS et BFS. Elle contient une variable **echiq** qui est une liste d'entiers représentant les positions de chaque reine sur l'échiquier suivant la modélisation expliquée auparavant.

4.1.2 Result

Cette classe est utilisée pour stocker les résultats des algorithmes de recherche DFS et BFS, elle contient :

- **L'attribut listeSol** : qui est de type **Node** et représente la solution du problème.
- **L'attribut nbrNodeGen** : qui est un entier représentant le nombre de noeuds générés avant la découverte de la première solution.
- **L'attribut nbrNodeDev** : qui est un entier représentant le nombre total de noeuds développés pendant la recherche.

4.1.3 Node1

Cette classe est utilisée pour représenter un état du problème des n reines lorsqu'on utilise l'algorithme A*. Chaque instance de la classe contient :

— **L'attribut echiq** : une liste d'entiers représentant les positions des reines sur l'échiquier.
— **L'attribut cost** : représente le coût cumulatif (la fonction g) de chaque nœud dans l'arbre de recherche. Ici, le coût est le nombre de déplacements nécessaires pour atteindre le noeud actuel à partir du nœud initial (profondeur du noeud dans l'arbre).
— **L'attribut heuristic** : représente l'estimation de la distance ou du coût restant pour atteindre l'objectif depuis un nœud donné (la fonction h).
— **La méthode compare()** : implémentée pour permettre de comparer deux nœuds en fonction de leur coût total, c'est-à-dire leur coût actuel plus l'heuristique restante estimée jusqu'à la solution finale.

4.1.4 Result1

Cette classe est utilisée pour stocker les résultats des algorithmes de recherche A*, elle contient les même attributs que la classe **Result** sauf pour l'attribut **listeSol** qui est ici de type **Node1**.

4.1.5 Util

Cette classe contient des méthodes d'affichage qui nous ont été utile lors des tests. Elle contient aussi la fonction **evaluation()** qui permet de vérifier si une configuration de l'échiquier est une solution valide. La méthode prend en paramètres la liste représentant l'échiquier et la taille n de l'échiquier.

La méthode vérifie si chaque paire de reines placées sur l'échiquier ne se menacent pas mutuellement. Elle vérifie les diagonales et les colonnes pour s'assurer qu'aucune paire de reines n'est en position de se menacer. Si la configuration est valide, la méthode retourne true. Si elle n'est pas valide, la méthode retourne false.

4.2 Algorithme DFS

Pour l'implémentation de l'algorithme DFS, nous avons créé une classe appelée **DFS**. Cette classe contient une seule méthode **successeursDFS()** qui prend en entrée la taille de l'échiquier n, et retourne un objet de la classe **Result**.

La fonction commence par initialiser un nœud racine qui représente l'état initial de l'échiquier (vide). Ce nœud est ensuite empilé dans une pile appelée **ouvert**.

La boucle principale de la recherche s'exécute tant que la pile ouvert n'est pas vide. Pour chaque itération, un nœud est retiré de la pile ouvert et vérifié pour voir s'il représente une solution valide à l'aide de la fonction d'évaluation **Util.evaluation()**. Si c'est le cas, le nœud est stocké dans result.listeSol et la recherche se termine. Sinon, les successeurs du nœud sont générés et empilés dans la pile ouvert.

Enfin, la méthode renvoie l'objet **Result** contenant la solution, le nombre de nœuds générés ainsi que le nombre de noeuds développés.

4.3 Algorithme BFS

Pour cette implémentation nous avons créer la classe **BFS** qui contient une méthode statique **successeursBFS()** qui prend en paramètres un entier n représentant la taille de l'échiquier et renvoie un objet de type **Result**.

La méthode commence par initialiser un objet **Result**, un objet **Node** représentant l'état initial de l'échiquier, et une liste ouvert représentant la file d'attente des nœuds à explorer. Elle ajoute l'état initial à la file d'attente et initialise les variables des statistiques.

Ensuite, la méthode commence à explorer les nœuds (ordonnées par la fonction **compare()** de la classe) en tête de file tant qu'il y a des nœuds dans la file. Pour chaque nœud en tête de file, elle vérifie si la configuration de l'échiquier est une solution en utilisant la méthode **Util.evaluation()**. Si c'est le cas, elle affecte le nœud courant à la variable **listeSol** de l'objet **Result**. Sinon, elle génère tous les successeurs possibles du nœud courant.

Enfin, la méthode renvoie l'objet **Result** contenant la liste des solutions, le nombre de nœuds générés avant de trouver la première solution ainsi que le nombre de noeuds développés.

4.4 Algorithme A*

Pour cet algorithme nous avons créé deux classes **Astar1** et **Astar2** qui implémentent le même algorithme A* mais avec des heuristiques différentes :

4.4.1 Heuristique du conflit minimal

Cette heuristique est une fonction qui compte le nombre de conflits entre les paires de reines sur l'échiquier. Pour chaque paire, elle vérifie si elles se trouvent sur la même colonne ou sur la même diagonale. Si c'est le cas, elle augmente le compteur de conflits et le retourne en sortie. Ce nombre donne une estimation de la distance entre l'état actuel et l'état final.

Cette fonction heuristique est **admissible**, ce qui signifie qu'elle ne surestime jamais la distance à l'état final. Cela est dû au fait que placer une reine sur l'échiquier ne peut qu'abaisser ou maintenir le nombre de conflits, mais ne peut jamais l'augmenter.

4.4.2 Heuristique de la distance Euclidienne

Cette heuristique est une fonction qui mesure la distance euclidienne entre les positions actuelles des reines sur l'échiquier et les positions finales souhaitées. Elle compare toutes leurs positions, en évaluant la distance qui les sépare en prenant en compte le nombre de paires de reines pouvant s'attaquer mutuellement.

Celle-ci est **admissible**, car elle sous-estime toujours le nombre de mouvements nécessaires pour résoudre le problème.

Les deux classes **Astar1** et **Astar2** contiennent trois méthodes principales :

- **La méthode successeursAstar()** : qui implémente l'algorithme A* et qui prend la taille de l'échiquier en entrée et retourne un objet de type **Result1**. Cette méthode commence par initialiser une file de priorité **ouvert** et un noeud racine. Ensuite, elle l'ajoute à la file de priorité et boucle tant que la file n'est pas vide. Pour chaque itération, elle extrait le noeud avec la plus petite valeur de coût heuristique de la file, vérifie s'il s'agit d'une solution valide avec la fonction **Util.evaluation()** et, si c'est le cas, retourne ce noeud en tant que **Result1**. Sinon, elle génère tous les successeurs possibles du noeud actuel en appelant la méthode **generateSuccessors()**, ajoute ses successeurs à la file de priorité et continue la recherche.
- **La méthode generateSuccessors()** : qui prend en entrée un nœud et un entier n qui est la taille de l'échiquier. Elle crée une liste de **successeurs** pour ce nœud en générant tous les états possibles qui peuvent être atteints à partir de cet état et la retourne en sortie.
- **La méthode calculateHeuristic()** : qui calcule la fonction heuristique h d'un noeud donné en paramètre. Elle dépend de l'heuristique utilisée.

Chapitre 5

Expérimentations

Dans ce chapitre, nous présentons les résultats de l'application des algorithmes DFS, BFS et A* pour la résolution du problème des n reines. Nous examinons en particulier les statistiques relatives aux temps d'exécution, au nombre de nœuds générés avant la première solution, ainsi qu'au nombre de nœuds développés pour chaque algorithme.

L'objectif principal de cette étude est de déterminer l'efficacité de ces algorithmes pour résoudre ce problème, en utilisant différentes mesures. Nous discutons également des avantages et des inconvénients de chaque algorithme.

5.1 Environnement expérimental

5.1.1 Matériel

L'implémentation de la solution a été réalisée sur un ordinateur ayant les configurations matérielles suivantes :

— Un microprocesseur Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz.
— Une mémoire de 16,0 Go de RAM.
— Système d'exploitation Windows 11 Famille 64bits.

5.1.2 Logiciel

Pour l'environnement de développement, nous avons opté pour : IntelliJ IDEA 3.3.2022.

5.2 Interface graphique

5.2.1 Design

Afin de visualiser les résultats de manière plus intuitive, nous avons développé une interface graphique qui constituée de 2 parties : une partie à droite pour afficher l'échiquier et une partie à gauche contenant des menus, des bouttons de contrôle ainsi que des statistiques de la résolution.

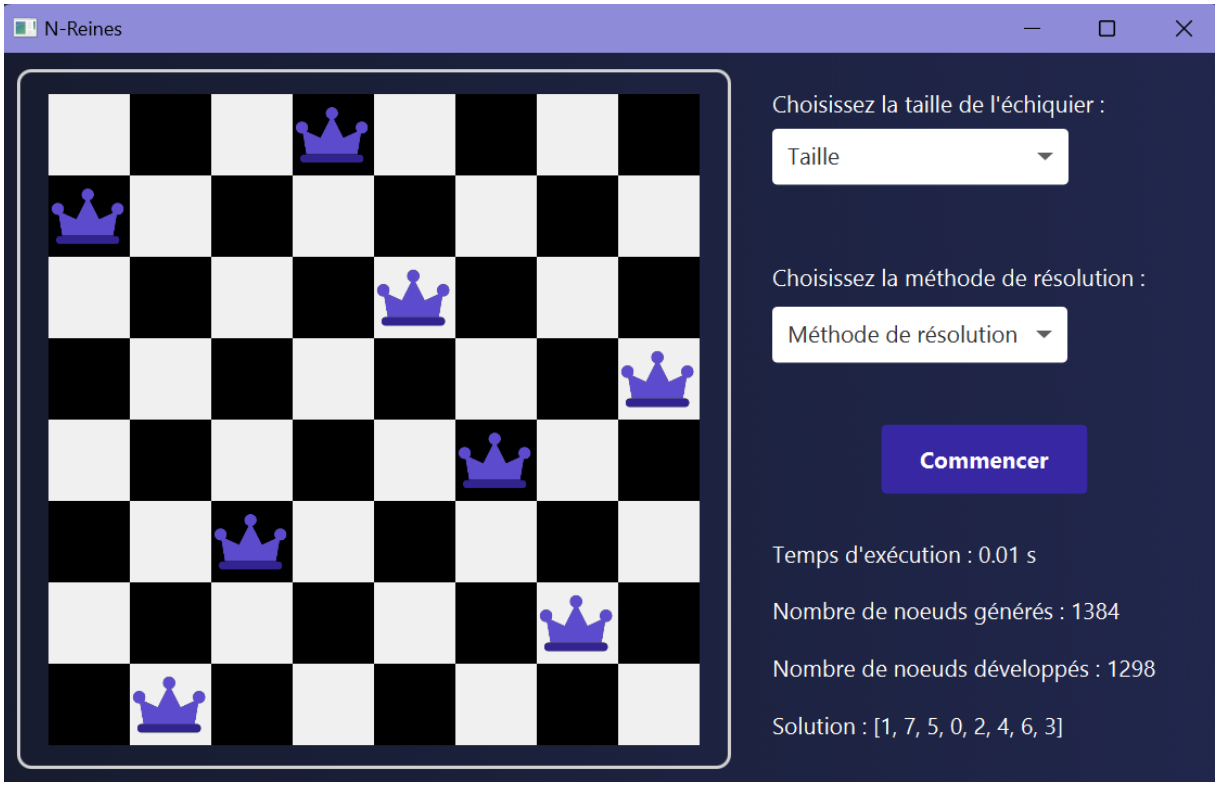

FIGURE 5.1 – Interface graphique

Le premier menu permet à l'utilisateur de seléctionner la taille de son échiquier (8X8, 12X12...etc). Le deuxième menu lui permet de seléctionner la méthode de résolution souhaitée (DFS, A*...etc).

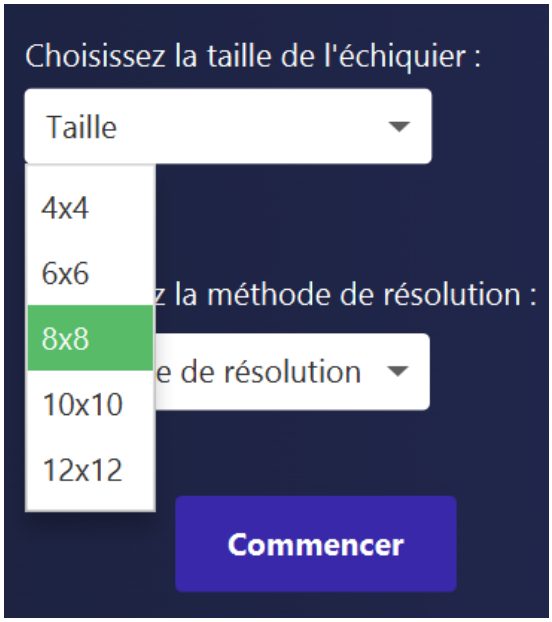

FIGURE 5.2 – Menu taille échiquier

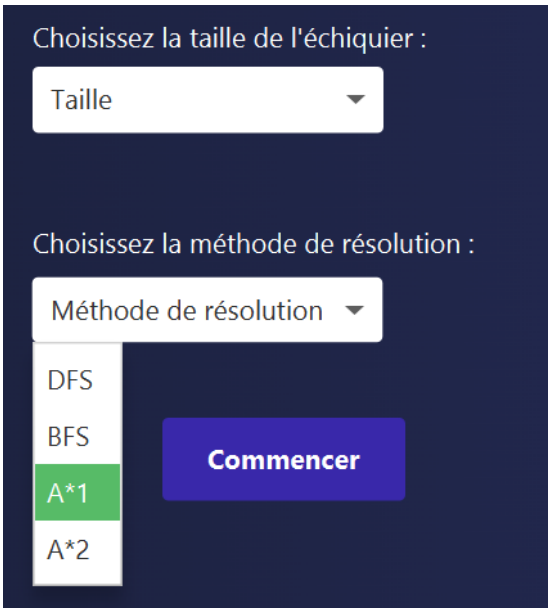

FIGURE 5.3 – Menu méthode

Après avoir séléctionner la taille de l'échiquier et la méthode de résolution, l'utilisateur peut clicker sur le boutton **"Commencer"** qui lance la recherche selon la méthode séléctionnée, place les reines dans leurs positions et affiche également des statistiques relatives à la recherche tel que ; le temps d'exécution, le nombre de noeuds générés, le nombre de noeuds développés et la solution affichée sur l'échiquier.

5.3 Résultats expérimentaux

5.3.1 DFS pour N-reines

L'application de notre solution DFS sur un intervalle croissant de taille d'échiquier N, nous a permis de dresser le tableau de résultats suivant :

DFS				
N	**Temps d'exécution (s)**	**Nombre de noeuds générés**	**Nombre de noeuds développés**	**Solution**
6	0.002699	512	324	[4, 2, 0, 5, 3, 1]
7	0.002299	513	325	[6, 4, 2, 0, 5, 3, 1]
8	0.013849	7732	4888	[7, 3, 0, 2, 5, 1, 6, 4]
9	0.01075	20898	13211	[8, 6, 3, 1, 7, 5, 0, 2, 4]
10	0.055949	160349	101361	[9, 7, 4, 2, 0, 5, 1, 8, 6, 3]
11	0.337899	1256183	794061	[10, 8, 6, 4, 2, 0, 9, 7, 5, 3, 1]
12	2.17435	12356500	7810799	[11, 9, 7, 4, 2, 0, 6, 1, 10, 5, 3, 8]
13	24.440899	128763398	81393993	[12, 10, 8, 11, 4, 1, 3, 0, 9, 7, 5, 2, 6]

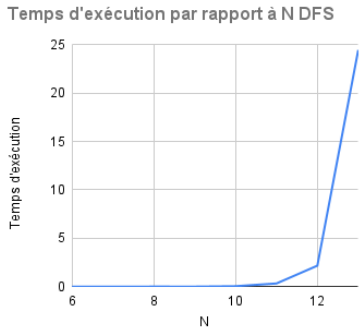

FIGURE 5.4 – Temps d'exécution par rapport à N DFS

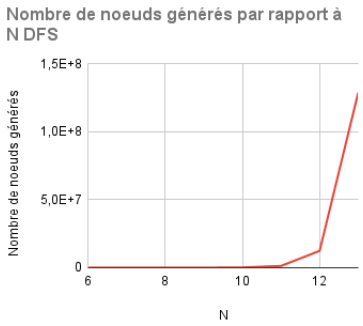

FIGURE 5.5 – Nombre de noeuds générés par rapport à N DFS

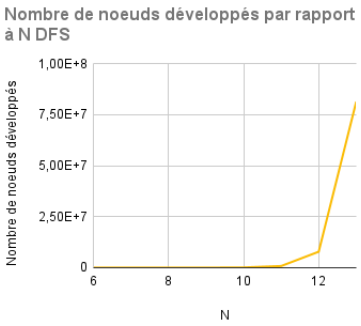

FIGURE 5.6 – Nombre de noeuds développés par rapport à N DFS

5.3.2 BFS pour N-reines

L'application de notre solution BFS sur un intervalle croissant de taille d'échiquier N, nous a permis de dresser le tableau de résultats suivant :

BFS				
N	**Temps d'exécution (s)**	**Nombre de noeuds générés**	**Nombre de noeuds développés**	**Solution**
6	0.0037	3381	1237	[1, 3, 5, 0, 2, 4]
7	0.010599	22547	8660	[0, 2, 4, 6, 1, 3, 5]
8	0.045	181725	69281	[0, 4, 7, 5, 2, 6, 1, 3]
9	0.365099	1617626	623530	[0, 2, 5, 7, 1, 3, 8, 6, 4]
10	7.14055	16158389	6235301	[0, 2, 5, 7, 9, 4, 8, 1, 3, 6]
11	space error	space error	space error	space error

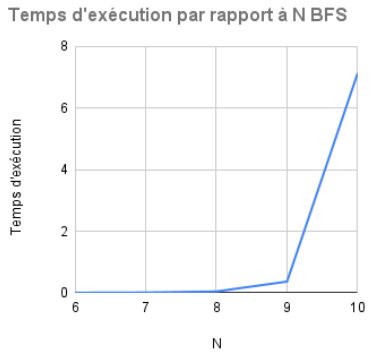

FIGURE 5.7 – Temps d'exécution par rapport à N BFS

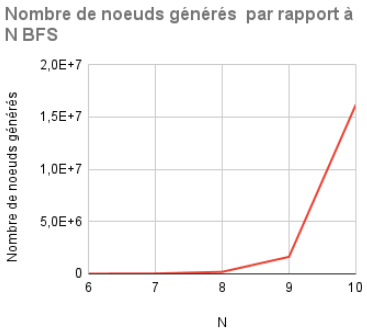

FIGURE 5.8 – Nombre de noeuds générés par rapport à N BFS

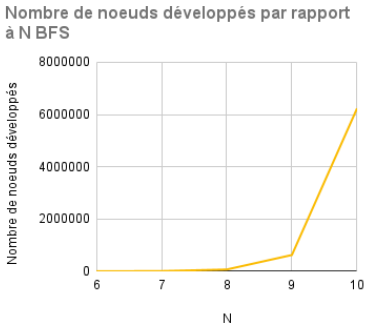

FIGURE 5.9 – Nombre de noeuds développés par rapport à N BFS

5.3.3 A* pour N-reines

5.3.3.1 Heuristique conflits minimum

L'application de notre solution A* avec l'heuristique de conflits minimum sur un intervalle croissant de taille d'échiquier N, nous a permis de dresser le tableau de résultats suivant :

A* avec heuristique conflits minimum				
N	**Temps d'exécution (s)**	**Nombre de noeuds générés**	**Nombre de noeuds développés**	**Solution**
6	0.006	73	49	[1, 3, 5, 0, 2, 4]
7	0.007	88	48	[1, 6, 4, 2, 0, 5, 3]
8	0.00805	1384	1298	[1, 7, 5, 0, 2, 4, 6, 3]
9	0.016250	5185	5046	[1, 8, 5, 3, 6, 0, 2, 4, 7]
10	0.012549	5374	5271	[1, 9, 2, 6, 8, 3, 0, 4, 7, 5]
11	0.119299	53723	53592	[1, 10, 2, 6, 8, 0, 4, 9, 7, 5, 3]
12	0.432050	204896	204771	[1, 11, 2, 5, 10, 8, 4, 0, 3, 6, 9, 7]
13	2.649850	1311762	1311623	[1, 12, 2, 9, 7, 4, 10, 3, 0, 11, 5, 8, 6]

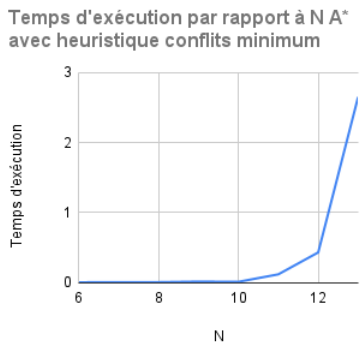

FIGURE 5.10 – Temps d'exécution par rapport à N A* avec heuristique conflits minimum

FIGURE 5.11 – Nombre de noeuds générés par rapport à N A* avec heuristique conflits minimum

FIGURE 5.12 – Nombre de noeuds développés par rapport à N A* avec heuristique conflits minimum

5.3.3.2 Heuristique distance Euclidienne

L'application de notre solution A* avec l'heuristique de distance Euclidienne sur un intervalle croissant de taille d'échiquier N, nous a permis de dresser le tableau de résultats

suivant :

A* avec heuristique distance Euclidienne				
N	**Temps d'exécution (s)**	**Nombre de noeuds générés**	**Nombre de noeuds développés**	**Solution**
6	0.001350	207	67	[1, 3, 5, 0, 2, 4]
7	0.001650	428	114	[4, 1, 5, 2, 6, 3, 0]
8	0.0031	655	130	[2, 4, 1, 7, 5, 3, 6, 0]
9	0.0261	4259	919	[4, 8, 1, 3, 6, 2, 7, 5, 0]
10	0.003599	11438	242	[4, 2, 9, 3, 6, 8, 1, 5, 7, 0]
11	0.354400	25215	4330	[5, 2, 9, 1, 3, 8, 10, 7, 4, 6, 0]
12	3.906600	79628	12574	[5, 1, 4, 9, 3, 8, 2, 11, 6, 10, 7, 0]
13	1.390800	47857	7068	[3, 6, 11, 2, 10, 1, 4, 8, 12, 9, 7, 5, 0]

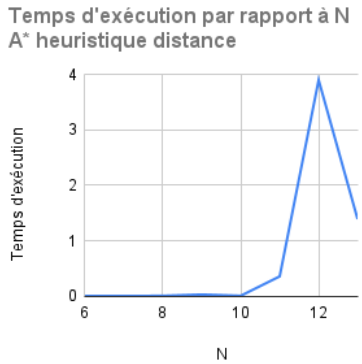

FIGURE 5.13 – Temps d'exécution par rapport à N A* avec heuristique distance Euclidienne

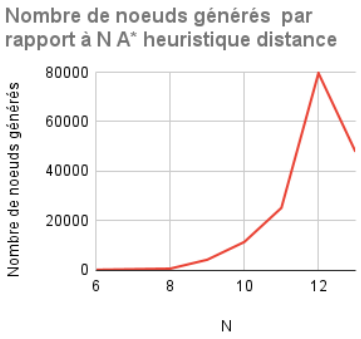

FIGURE 5.14 – Nombre de noeuds générés par rapport à N A* avec heuristique distance Euclidienne

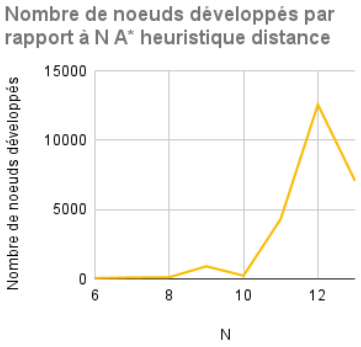

FIGURE 5.15 – Nombre de noeuds développés par rapport à N A* avec heuristique distance Euclidienne

5.4 Etude comparative

5.4.1 Temps d'exécution

Temps d'exécution (s)				
N	**DFS**	**BFS**	**A* conflits minimum**	**A* distance Euclidienne**
6	0.002699	0.0037	0.006	0.001350
7	0.002299	0.01059	0.007	0.001650
8	0.013849	0.045	0.00805	0.0031
9	0.01075	0.365099	0.016250	0.0261
10	0.055949	7.14055	0.012549	0.003599
11	0.337899	space error	0.119299	0.354400
12	2.17435	space error	0.432050	3.906600
13	24.440899	space error	2.649850	1.390800

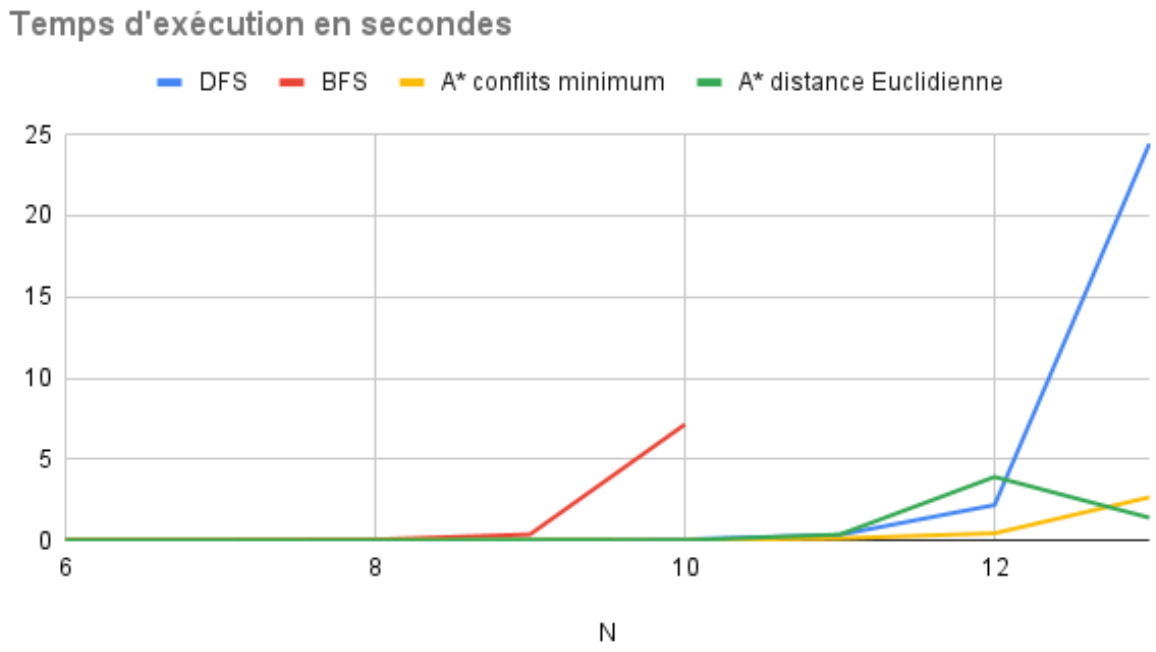

FIGURE 5.16 – Temps d'exécution par rapport à N des algorithmes DFS, BFS et A*

5.4.1.1 Analyse des résultats

On peut voir que pour des tailles d'échiquiers plus petites (6-8), DFS est le plus rapide, suivi de BFS et des deux variantes d'A*. Cependant, pour des tailles plus grandes, A* avec heuristique conflits minimal est généralement le plus rapide, suivi de A* avec heuristique distance euclidienne, puis DFS et enfin BFS. On peut également noter que pour des tailles d'échiquiers plus grandes (11-13), BFS a échoué à trouver une solution en raison du grand nombre de noeuds à explorer.

5.4.2 Nombre de noeuds générés

Nombre de noeuds générés				
N	**DFS**	**BFS**	**A* conflits minimum**	**A* distance Euclidienne**
6	512	3381	73	207
7	513	22547	88	428
8	7732	181725	1384	655
9	20898	1617626	5185	4259
10	160349	16158389	5374	1437
11	1256183	space error	53723	79627
12	12356500	space error	204896	47856
13	128763398	space error	1311762	166272

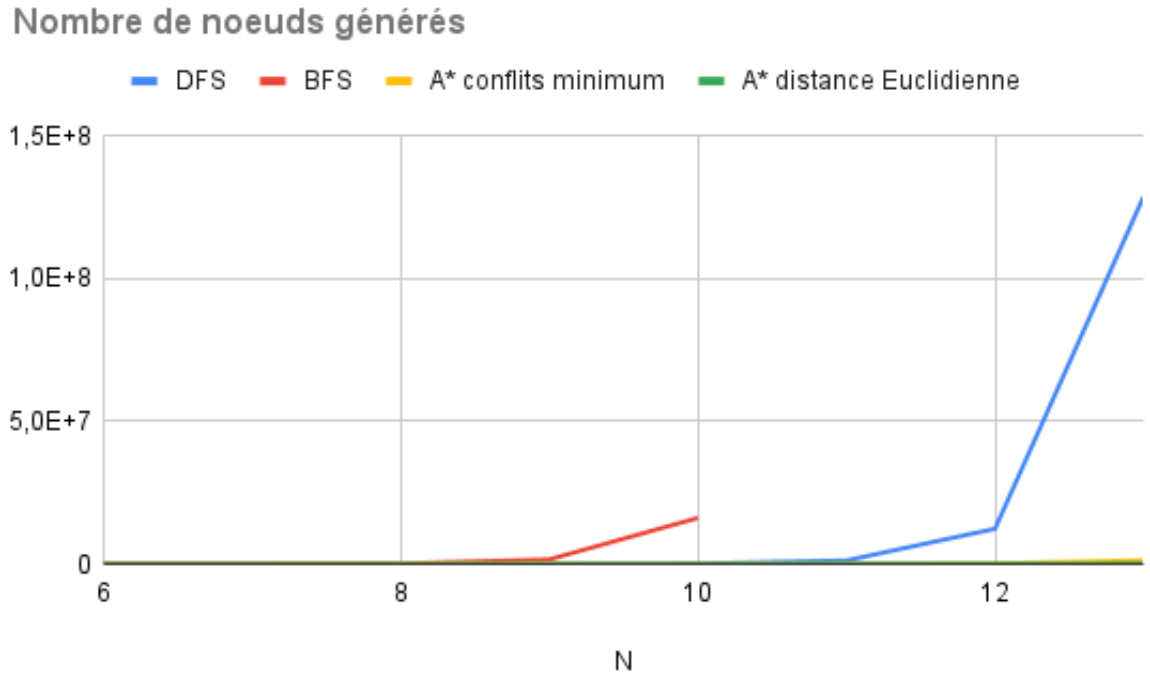

FIGURE 5.17 – Nombre de noeuds générés par rapport à N des algorithmes DFS, BFS et A*

5.4.2.1 Analyse des résultats

On peut observer que le nombre de noeuds générés par l'algorithme DFS est plus faible que BFS. En revanche, l'algorithme BFS explore tout l'arbre de recherche, ce qui rend son nombre de noeuds générés très élevé pour des instances du problème de taille importante.

En comparaison, les deux algorithmes A* génèrent un nombre de noeuds intermédiaire. L'heuristique du conflit minimal a des performances légèrement supérieures à l'heuristique de la distance euclidienne. Cela s'explique par le fait que l'heuristique du conflit minimal est spécifique au problème des n reines car elle prend en compte les conflits entre les reines, ce qui conduit à une exploration plus rapide des configurations les plus prometteuses. Tandis que l'heuristique de la distance euclidienne est plus générale.

5.4.3 Nombre de noeuds développés

Nombre de noeuds développés				
N	**DFS**	**BFS**	**A* conflits minimum**	**A* distance Euclidienne**
6	324	1237	49	67
7	325	8660	48	114
8	4888	69281	1298	130
9	13211	623530	5046	919
10	101361	6235301	5271	242
11	794061	space error	53592	4330
12	7810799	space error	204771	12574
13	81393993	space error	1311623	7068

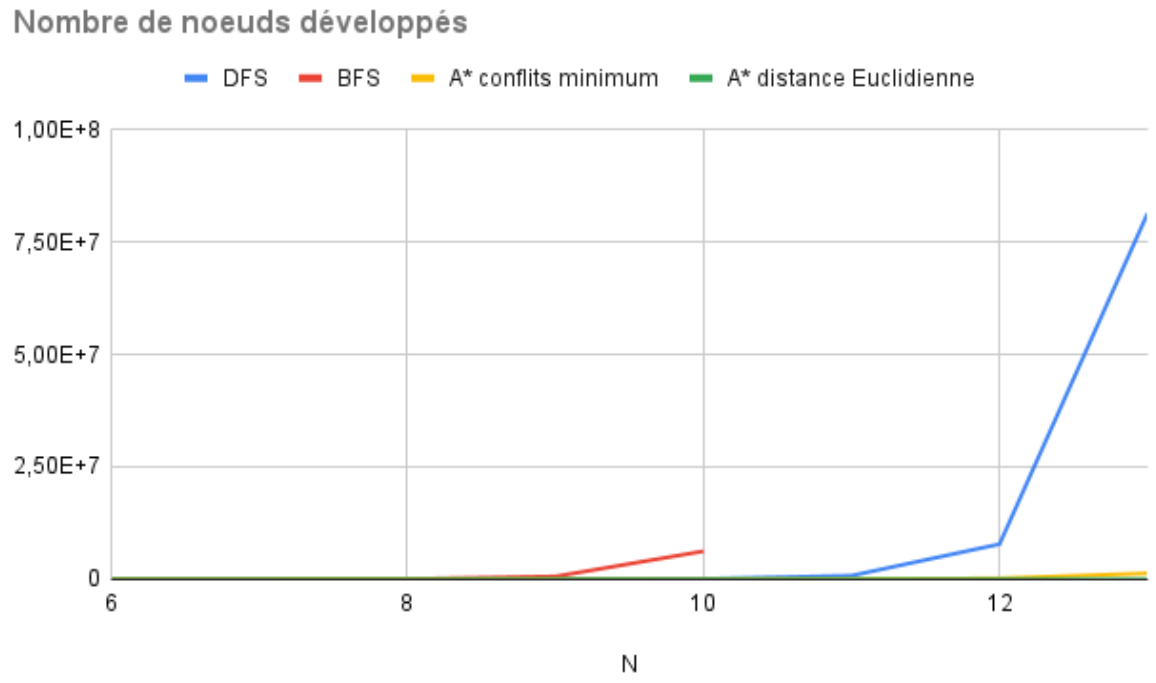

FIGURE 5.18 – Nombre de noeuds développés par rapport à N des algorithmes DFS, BFS et A*

5.4.3.1 Analyse des résultats

Ces résultats montrent que les algorithmes de recherche A* avec heuristique conflits minimal et A* avec heuristique distance euclidienne sont plus efficaces que les algorithmes de recherche en profondeur d'abord et en largeur d'abord pour le problème des n reines. En effet, ils génèrent beaucoup moins de noeuds pour toutes les tailles de N.
Cependant, l'heuristique de conflit peut être plus difficile à utiliser pour de grandes instances du problème car on remarque qu'elle développe un nombre de noeuds similaire à DFS pour ces dernières, tandis que l'heuristique de distance euclidienne reste relativement constante.

Le BFS et le DFS ont développé un grand nombre de nœuds pour les plus grandes tailles d'échiquier, ce qui peut être attribué à leur méthode de recherche exhaustive.

Chapitre 6

Conclusion et perspectives

En conclusion, les résultats obtenus montrent les avantages et les inconvénients des différentes approches de recherche pour résoudre le problème des n reines. Les algorithmes de recherche non informés tels que DFS et BFS sont rapidement limités par la taille du problème, tandis que les algorithmes de recherche informés tels que A* avec les heuristiques de conflit et de distance euclidienne sont plus efficaces en termes de nombre de nœuds générés.

Pour des problèmes de taille moyenne, l'heuristique de conflits minimum semble être une bonne option. Cependant, pour des problèmes de plus grande taille, il peut être nécessaire d'explorer d'autres approches de recherche ou d'heuristiques plus sophistiquées pour obtenir des résultats optimaux.

Des perspectives intéressantes pourraient inclure l'utilisation de techniques d'apprentissage automatique pour élaborer des heuristiques plus efficaces ou l'exploration d'autres types d'algorithmes de recherche tels que la recherche locale ou les algorithmes évolutifs.

En fin de compte, la résolution du problème des n reines reste un défi intéressant et complexe pour les chercheurs en intelligence artificielle et en informatique, et les résultats obtenus dans cette étude ouvrent la voie à des recherches futures dans ce domaine.