Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département d'Intelligence Artificielle
et Sciences de Données

Master I Systèmes Informatiques Intelligents

Résolution des problèmes NP-COMPLET

'' Problèmes du N-Reines ''

Réalise par :

- DJENANE Nihad 191931040689 G1
- M'BAREK Lydia 181831064011 G1

Année universitaire
2022/2023

Table des matières

List des figures

List des tableaux

Introduction

Les méthodes de résolution de problèmes en Intelligence Artificielle (IA) sont des approches algorithmiques qui permettent aux ordinateurs de résoudre des problèmes complexes. Le problème des N Reines est un exemple courant de problème qui peut être résolu à l'aide de ces méthodes. Ce problème est considéré comme un problème NP-complet, ce qui signifie qu'il est difficile à résoudre pour les échiquiers de grande taille, mais facile à vérifier si une solution proposée est correcte.

Dans cette première partie du rapport, nous allons explorer les performances de différentes méthodes de recherche pour résoudre le problème des N-Reines. Nous allons commencer par étudier les performances de deux méthodes de recherche aveugles : l'algorithme de recherche en largeur (BFS) et l'algorithme de recherche en profondeur (DFS). Ensuite, nous introduirons une méthode de recherche heuristique l'algorithme A* qui utilise une fonction d'évaluation pour guider la recherche vers des solutions plus prometteuses. Nous comparerons les performances de ces méthodes en termes de temps d'exécution et de nombre de nœuds explorés pour résoudre le problème des N-Reines.

Chapitre 1 : Approche espace d'états

En Intelligence Artificielle, la résolution de problèmes est une discipline clé pour atteindre des solutions efficaces à des problèmes complexes. L'approche de l'espace des états est l'une des méthodes les plus courantes de résolution de problèmes en IA. Elle consiste à modéliser un problème sous forme d'un ensemble d'états possibles, et à utiliser des opérateurs pour passer d'un état à un autre jusqu'à atteindre le but.

Les concepts clés dans l'approche de l'espace des états sont les suivants :

- Un état représente une configuration du problème.
- Les opérateurs sont les actions qui peuvent être appliquées à un état pour générer un nouvel état.
- Les successeurs sont les nouveaux états générés par l'application d'un opérateur à un état.

Les algorithmes de recherche dans l'espace des états varient en termes de complexité et de performance, et peuvent être classés en deux catégories : les méthodes non informées (ou aveugles) et les approches informées (ou heuristiques).

Dans cette première partie, s'en intéresse à ces deux catégories.

1. Les méthodes aveugles

Ces méthodes utilisent des algorithmes non informés, réalisent une recherche exhaustive sans utiliser des connaissances sur la structure de l'espace d'états pour l'optimisation de la recherche. Parmi les méthodes aveugles les plus courantes, on trouve la recherche en largeur d'abord (BFS), la recherche en profondeur d'abord (DFS), la recherche de profondeur limite...

1.1 Recherche en largeur d'abord (BFS)

Consiste à explorer tous les successeurs d'un nœud avant de passer au niveau suivant. Cette stratégie est complète car on arrive toujours à trouver une solution si elle existe, mais peut être coûteuse en termes temps et mémoire...

La complexité de la recherche en largeur d'abord est de l'ordre $O(b^d)$, où b est le facteur de branchement (le nombre moyen de successeurs pour chaque nœud) et d est la profondeur du nœud but le moins profond.

Algorithme BFS ;
Entrée : s : état initial ; F : ensemble des états finaux ;
Sortie : une solution si succès sinon échec var : Ouverte, Fermée : file de nœuds initialement vide ;
début
insérer le nœud initial s dans la file Ouverte ;

L : **si** Ouverte est vide alors échec
sinon continuer ;
défiler n le premier nœud de la file Ouverte et l'insérer dans Fermée ;
si il n'existe pas de successeur de n **alors aller** à L ;

Déterminer les successeurs de n et les enfiler dans Ouverte ;
créer un chaînage de ces nœuds vers n ;
si parmi les successeurs, il existe un état final **alors succès** :
la solution est la chaîne des nœuds allant du nœud courant à la racine
sinon aller à L ;
fin

1.2 Recherche en profondeur d'abord (DFS)

Elle explore autant que possible le premier successeur avant de passer au suivant.

Cette stratégie est plus économique en termes de mémoire, car elle ne stocke en mémoire que les nœuds du chemin menant au but et non tous les nœuds du graphe.

La complexité est de l'ordre de $O(b^m)$, où b est le facteur de branchement et m est la profondeur maximale de l'arbre.

La recherche en profondeur limitée est similaire à la recherche en profondeur d'abord, mais elle limite la profondeur de la recherche à un certain niveau.

Algorithme DFS ;
Entrée : s : état initial ; F : ensemble des états finaux ;
Sortie : une solution si succès **sinon échec** ;
var : Ouverte : pile de nœuds initialement vide ;
Fermée : file de nœuds initialement vide ;
Début empiler le nœud initial s dans Ouverte ;
L : **si** Ouverte est vide **alors** échec **sinon continuer** ;
dépiler n, le premier nœud de Ouverte et l'enfiler dans Fermée ;
si le seuil de profondeur est atteint **alors aller** à L **sinon continuer** ;
déterminer les successeurs de n et les empiler dans Ouverte ;
créer un chaînage de ces nœuds vers n ;
si parmi les successeurs, il existe un état final **alors** succès :
la solution est la chaîne des nœuds allant du nœud courant à la racine
sinon aller à L ;
fin

2. Les méthodes heuristiques

Les méthodes heuristiques utilisent une estimation de la solution pour guider la recherche dans l'espace de recherche et évaluer les alternatives en fonction de leur capacité à se rapprocher de la solution recherchée.

L'algorithme A* utilise une fonction heuristique qui estime la distance entre l'état courant et l'état but pour évaluer chaque état et choisir le successeur le plus prometteur.

Algorithme A*

Entrée : s état initial ; F ensemble des états finaux ; c(n_i , n_j) coût n_i et n_j ; h fonction heuristique
Sortie : une solution si succès sinon échec
var : Ouverte : liste de nœuds triée selon la fonction f et initialement vide ;
Fermée : file de nœuds initialement vide ;

```
début
    f(s) := g(s) + h(s) ;
    insérer s dans la liste Ouverte avec f(s) ;
    tant que Ouverte n'est pas vide faire
        début
            retirer n le premier nœud de la liste Ouverte qui a la plus petite valeur de f;
            enfiler n dans Fermée ;
            si n est un état final alors succès utiliser le chainage arrière pour obtenir la solution
            sinon si n a des successeurs alors pour chaque successeur ni faire
                    début f(ni ) := g(ni ) + h(ni ) ;
                        si ni n'est ni dans Ouverte ni dans Fermée alors
                            début enfiler ni avec f(ni ) dans Ouverte selon l'ordre croissant de f;
                                    créer un chaînage de ni vers n ;
                             fin
                        sinon si f(ni ) est inférieure à la valeur de ni dans Ouverte ou Fermée
                              alors début remplacer cette valeur par f(ni ) ;
                                          mettre à jour le chainage arrière ;
                         fin
                    fin
             fin
    fin
```

Chapitre 2 : Le problème de N-Reines

1. Représentation du problème

Le problème des N-Reines est un problème classique consiste à placer N reines sur un plateau d'échecs de N*N cases de manière à ce qu'aucune reine ne puisse menacer une autre reine. En d'autres termes, aucune paire de reines ne doit être sur la même rangée, colonne ou diagonale. On peut décrit le problème formellement comme suit :

Étant donné un échiquier de taille N*N, le but est de placer N reines sur cet échiquier de telle manière que :

- Aucune reine ne se trouve sur la même ligne qu'une autre reine.
- Aucune reine ne se trouve sur la même colonne qu'une autre reine.
- Aucune reine ne se trouve sur la même diagonale qu'une autre reine.

Par exemple, Voici une solution possible pour N = 7 :

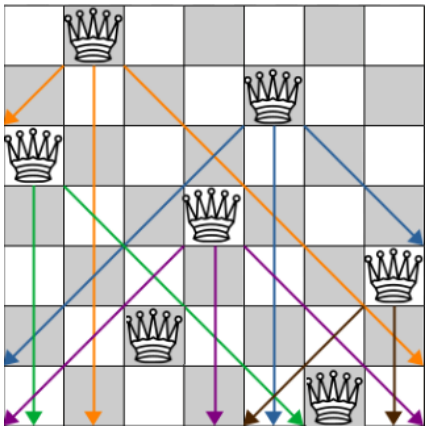

Figure 1 : Solution possible du problème N-Reines pour N = 7

2. Modélisation du problème

Nous avons adopté une approche de modélisation du Problème des N-Reines en utilisant une matrice booléenne de taille N*N, où chaque élément représente une case sur l'échiquier.
Alors :

- Un état est une matrice booléenne de taille N × N qui représente une configuration particulière de reines placées sur l'échiquier.
  Exemple : Pour N = 4

$$\text{Etat initiale} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Etat finale} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- L’opérateur dans ce problème est d’ajoute une reine à une case vide de l'échiquier, à condition que cette action ne crée pas de conflit avec une autre reine déjà présente sur l'échiquier.
- Les successeurs sont tous les états qui peuvent être atteints à partir de l'état actuel en appliquant une seule action d'opérateur.

3. Fonction d’évaluation

Cette fonction permet d'évaluer un état donné en calculant le nombre de reines menacées sur l'échiquier.

La méthode "enAttack()" prend en entrée une case de l'échiquier et renvoie true si cette case est menacée par une autre reine présente sur l'échiquier.

```
public int evaluate ( boolean[][] board){
    int attack = 0 ;

    for ( int i = 0 ; i < board.length; i++)
        for ( int j = 0; j < board.length ; j++)
            if ( board[i][j] && this.enAttcak(board, i, j) ) attack++;
    return attack;
}

private boolean enAttcak (boolean[][] board , int row , int col ){
    int diagonal ;
    for ( int i = 0; i < board.length ; i++ ){
        // verifier la colone
        if ( i != row && board[i][col] == true ) return true;
        // verifier diagonal superieur a droit
        diagonal = i - row + col ;
        if (diagonal  >= 0 && diagonal < board.length && i != row && diagonal != col &&  board[i][ diagonal ] == true)
            return true;
        // verifier diagonal superieur a gauche
        diagonal = row + col - i;
        if (diagonal  >= 0 && diagonal < board.length &&  i != row && diagonal != col && board[i][ diagonal ] == true)
            return true;
    }
    return false;
}
```

Figure 2 : Fonction d'évaluation

Chapitre 3 : Implémentation

Cette partie du rapport présente l'implémentation de notre modélisation et résolution du problème des N reines en décrivant les différentes classes principales de notre projet Java.

1. Classe NQueensProblem

Cette classe permet d'implémenter notre approche de modélisation du problème. Elle a comme attribue la taille du problème et la matrice lui-même.

La méthode "initBoard()" permet d'initialiser l'échiquier à vide en définissant toutes les valeurs de la matrice à "false".

```
public class NQueensProblem {
    private int size;
    public boolean[][] board ;

    public  NQueensProblem( int size  ) {
        this.size = size;
        board = new boolean [this.size][this.size];
        this.initBorad();
    }

    public int getSize() {
        return size;
    }

    public void initBorad() {
        for(int i = 0; i< size; i++)
            for (int j = 0; j < size; j++)
                board[i][j]= false;
    }
}
```

Figure 3 : Classe NQueensProblem

2. Classe Solution

Cette classe nous aide à implémenter les algorithmes de résolution du problème des N reines. Elle hérite de la classe NQueensProblem. Elle possède des attributs tels que "nbrNodeGener" (nombre de nœuds générés) et "nbrNodeDevelop" (nombre de nœuds développés) pour suivre les performances de l'algorithme de recherche de solution.

Elle contient plusieurs méthodes utiles telles que «ifSafe» qui permet de vérifier si une reine peut être placée en toute sécurité sur une certaine position de l'échiquier, et d'autres méthodes qui facilitent la mise en œuvre des algorithmes de résolution.

Elle est déclarée abstraite pour permettre la création de classes de solution spécifiques en héritant de cette classe de base.

```java
public abstract class Solution extends NQueensProblem  {

    public long nbrNodeGener;
    public long nbrNodeDevelop;

    public Solution( int size ){

    abstract public boolean solveN_Queens(boolean[][] board);

    public boolean ifSafe( boolean[][] board , int row , int col ){

        int diagonal ;
        for ( int i = 0; i < row ; i++ ){
            // verifier la colone
            if ( board[i][col] == true ) return false;
            // verifier diagonal superieur a droit
            diagonal = i - row + col ;
            if (diagonal  >= 0 && diagonal < this.getSize() &&  board[i][ diagonal ] == true) return false;

            // verifier diagonal superieur a gauche
            diagonal = row + col - i;
            if (diagonal  >= 0 && diagonal < this.getSize() && board[i][ diagonal ] == true) return false;
        }
         return true;
     }

    public void printBorad(boolean[][] borad){
    public void printTable(boolean[][] board){
    public int evaluate ( boolean[][] board){
    private boolean enAttcak (boolean[][] board , int row , int col ){
    public static void copyMatrix(boolean[][] source, boolean[][] destination) {
}
```

Figure 4 : Classe Solution

3. Résolution à l'aide de l'algorithme DFS

Nous avons créé une classe DFSsolution pour implémenter l'algorithme de recherche en profondeur d'abord DFS. (Voir la figure 4).

Elle hérite de la classe Solution, elle a également son propre attribut "temp" qui est une matrice temporaire pour stocker la solution trouvée pendant la recherche, et un booléen "solutionFounded" qui est utilisé pour vérifier si une solution a été trouvée.

La classe DFSsolution utilise la méthode "solveN_Queens" pour explorer l'arbre des solutions possibles en générant des successeurs à chaque étape. Elle stocke temporairement les solutions trouvées. Si une solution est valide elle copier la matrice dans board et retourne true.

```java
public class DFSsolution extends Solution {
    private boolean[][] temp;
    private boolean solutionFounded;
    public DFSsolution(int size) {
    @Override
    public boolean solveN_Queens(boolean[][] board) {
         return this.solveN_Queens(board, 0);
    }

    public boolean solveN_Queens(boolean[][] borad, int row) {
        if ( this.solutionFounded ) return true;
        this.nbrNodeDevelop++;
        if ( row == board.length) {
            this.solutionFounded = true;
            Solution.copyMatrix(temp, borad);
            return true;
        }
        else {
            for ( int col = 0; col < temp.length; col++){
                if ( this.ifSafe(temp, row, col) ){
                   this.nbrNodeGener++;
                   temp[row][col] = true;
                   solveN_Queens(board , row + 1);
                   temp[row][col] = false;
                }
            }
        }
        return solutionFounded;
    }
}
```

Figure 5 : Classe DFSsolution

4. Résolution à l'aide de l'algorithme BFS

Nous avons implémenté l'algorithme de recherche en largeur d'abord BFS de manière similaire à notre implémentation précédente pour l'algorithme DFS.
Nous avons créé une classe BFSsolution hérite de la classe "Solution". Nous avons implémenté la méthode "solveN_Queens" prend en entrée une matrice "board" représentant l'état initial, et renvoie un booléen indiquant si une solution a été trouvée ou non. La méthode commence par initialiser la file "listOvert" avec tous les états possibles pour la première ligne de l'échiquier. Ensuite, elle développe tous les successeurs possibles dans ce niveau avant du passer à un autre. Si un état but est trouvé, la méthode copie cette configuration dans la matrice "board" et renvoie true. Sinon, la méthode continue à développer les nœuds jusqu'à ce que toutes les configurations possibles aient été explorées sans succès, auquel cas elle renvoie false.
Cette classe utilise également les variables "nbrNodeGener" et "nbrNodeDevelop", pour calculer le nombre de nœuds générés et développés pendant l'exécution de l'algorithme.

```java
public class Noued {
    public int row;
    public boolean[][] etat;

    public Noued(int size){
            this.row = 0;
            etat = new boolean[1][size];
            for(int i = 0; i < size; i++)
                this.etat[0][i] = false;
        }

    public Noued(boolean[][] etat, int row , int col , int size  ) {
        this.etat = new boolean[ row + 1 ][size];
        this.row = row;
        for ( int i = 0; i < row ; i++ )
          for ( int j = 0 ; j < size ; j++)
             this.etat[i][j] = etat[i][j];

        for ( int j = 0; j < size; j++)
             this.etat[row][j] = false;

        this.etat[row][col] = true;
        }
    }
```

Figure 6 : Classe Noued

```java
public class BFSsolution extends Solution {
    public BFSsolution(int size) {
    @Override
    public boolean solveN_Queens(boolean[][] board){

       int  col ;
       Noued nouedCurrent;
       Noued tempNoued;
       LinkedList<Noued> listOvert = new LinkedList<Noued>();
       // noued contient l'echequier vide
       nouedCurrent = new Noued(this.getSize());

       for ( col = 0; col < this.getSize(); col++ ){
           tempNoued = new Noued(nouedCurrent.etat , 0  , col , this.getSize() );
           listOvert.add(tempNoued);
           this.nbrNodeGener++;
       }
       while ( ! listOvert.isEmpty() ) {
           nouedCurrent = listOvert.removeFirst();
           this.nbrNodeDevelop++;
           if ( nouedCurrent.row + 1 == this.getSize() ){
               Solution.copyMatrix(nouedCurrent.etat, board);
               return true;
           }
           for ( col = 0; col < this.getSize() ; col++ ){
               if (this.ifSafe(nouedCurrent.etat, nouedCurrent.row + 1  , col)){
                   tempNoued = new Noued(nouedCurrent.etat , nouedCurrent.row + 1  , col , this.getSize() );
                   listOvert.add(tempNoued);
                   this.nbrNodeGener++;
               }
           }
       }
       return false;
    }
}
```

Figure 7 : Classe BFSsolution

5. Résolution à l'aide de l'algorithme A*

Nous avons proposé deux heuristiques pour résoudre le problème des N reines :

- Une heuristique statique qui utilise une matrice pour représenter chaque case contenant le nombre de cases qu'il peut attaquer.
- Une heuristique dynamique qui calcule le nombre de cases restant en sécurité pour chaque état.

5.1 Implémentation du l'heuristique statique

Les deux figures ci-dessous montre notre implémentation de l'algorithme A* en utilisant l'heuristique statique.

Nous avons créé deux classes :

La classe "Heuristique1Solution" est une sous-classe de la classe "Solution". A l'attribut "heuristique", qui est une matrice d'entiers stockant les valeurs de l'heuristique prédéfinie. Elle implémente la méthode "solveN_Queens" qui fonctionne sur le même principe que celle du BFS, à la différence qu'elle trie les nouds dans la file selon leur coût total.

La classe "NouedH" est une aide à notre implémentation. Elle hérite de la classe "Noued" possède un attribut supplémentaire "f", qui représente le coût total de l'état du nœud. Pour calculer "f", nous utilisons la formule suivante : f(n) = g(n) + h(n), où "g(n)" correspond au nombre de lignes restantes (size - row) et "h(n)" est la valeur heuristique. La méthode "addNouedSortH" est utilisée pour trier la file de nœuds par ordre croissant de leur coût total.

```
public class NouedH extends Noued {

    public int f;  // f(n) = g(n) + h(n)

    public NouedH(boolean[][] etat, int row , int col , int size , int h   ){
        super( etat , row , col , size);
        this.f = size - row + h ;
    }

    // constructeur pour creer le 1 er noud
    public NouedH(int size) {
        super(size);
    }
    public void addNouedSortH(LinkedList<NouedH> list){
        int i = 0;
        while ( i < list.size() && list.get(i).f < this.f ) i++;
        list.add( i , this );
    }
}
```

Figure 8 : Classe NouedH

```java
public class Heuristique1Solution extends Solution {
    int[][] heurstique;
    public Heuristique1Solution(int size) {
        super(size);
        this.heurstique = new int[size][size];
        for ( int i = 0; i < size; i++)
            for ( int j = 0; j < size; j++)
                // cette fromule permet de calculer nombre des case qui'il peut etre  attcke par cette case(i,j)
                this.heurstique[i][j] = Math.min((size - ( i + 1 )), ( size - ( j + 1 ))) +
                                        Math.min( ( size - ( i + 1 )), j);
    }
    public boolean solveN_Queens(boolean[][] board){
       int col;
       NouedH nouedCurrent;
       NouedH tempNoued;
       LinkedList<NouedH> listOvert = new LinkedList<NouedH>();
       nouedCurrent = new NouedH(this.getSize());
       for ( col = 0; col < this.getSize(); col++ ){
           tempNoued = new NouedH(nouedCurrent.etat , 0  , col , this.getSize() , this.heurstique[0][col] );
           tempNoued.addNouedSortH(listOvert);  his.nbrNodeGener++;
       }
       while ( ! listOvert.isEmpty() ) {
           nouedCurrent = listOvert.removeFirst(); this.nbrNodeDevelop++;
           if ( nouedCurrent.row + 1 == this.getSize() ){
               Solution.copyMatrix(nouedCurrent.etat, board);
               return true;
           }
           for ( col = 0; col < this.getSize() ; col++ )
               if ( this.ifSafe(nouedCurrent.etat, nouedCurrent.row + 1  , col)){
                   tempNoued = new NouedH(nouedCurrent.etat , nouedCurrent.row + 1  , col ,this.getSize() ,
                                                  this.heurstique[nouedCurrent.row + 1][col] );
                   this.nbrNodeGener++;tempNoued.addNouedSortH(listOvert);
               }
       }
       return false;
    }
```

Figure 9 : Classe Heustique1Solution

5.2 Implémentation du l'heuristique dynamique

Nous avons suivi la même approche que pour l'implémentation de l'heuristique 1 pour créer la classe "Heuristique2Solution". Cette dernière hérite également de la classe "Solution" et implémente la méthode "solveN_Queens" avec le même code que celui utilisé pour "Heuristique1Solution". La seule différence se situe dans la création du nœud.

Nous avons ajouté un nouveau constructeur dans la classe "NouedH", qui permet de créer un nœud et de calculer la valeur de "f" en utilisant la méthode "estimationH1". Cette méthode évalue l'heuristique "h" en se basant sur le nombre de reines déjà placées sur le plateau. On a f(n) = g(n) + h(n) ou :

h(n) est la valeur retourner par la fonction "estimationH1".

g(n) le nombre de lignes restants.

La méthode privée "estimationH1" renvoie le nombre de cases restantes sécurisées pour un état donné de l'échiquier.

```java
public class NouedH extends Noued {
    public int f;  // f(n) = g(n) + h(n)
    public NouedH(boolean[][] etat, int row , int col , int size  ){
        super( etat , row , col , size);
        this.f = this.estimationH1(size)  + size - row - 1 ;
    }
    private int estimationH1(int size ){
        int i , j , row , col , safe , diagonal;
        int[] position = new int[this.etat.length];
        safe = 0;
        boolean isSafe;
        for ( i = 0; i < this.etat.length; i++ )
            for ( j = 0; j < size; j++)
                if ( this.etat[i][j] ) {
                    position[i] = j;
                    break;
                }
        for ( i = this.etat.length; i < size; i++ )
            for (  j = 0; j < size; j++){
                isSafe = true;
                for ( int k = 0; k < this.etat.length && isSafe; k++)  {
                    if ( position[k] == j ){ isSafe=false; continue;}
                    // verifier diagonal superieur a droit
                   diagonal = j + ( i - k );
                    if ( diagonal >=0 && diagonal < size && this.etat[k][diagonal] == true ) { isSafe=false; continue;}
                    // verifier diagonal superieur a gauche
                    diagonal = j - (i - k) ;
                     if (diagonal  >= 0 && diagonal < size && this.etat[k][ diagonal ] == true) { isSafe=false; continue;}
                }
                if(isSafe)safe++;
            }
        return safe ;
    }
    // constructeur pour creer le 1 er nouds qui represente la matrice vide
    public NouedH(int size) {
        super(size);
    }
```

Figure 10 : Classe NouedH version 2

```java
public class Heuristique2Solution extends Solution {

    public Heuristique2Solution(int size) {
        super(size);
    }
 public boolean solveN_Queens(boolean[][] board){
       int col;
       NouedH nouedCurrent;
       NouedH tempNoued;
       LinkedList<NouedH> listOvert = new LinkedList<NouedH>();

       nouedCurrent = new NouedH(this.getSize());

       for ( col = 0; col < this.getSize(); col++ ){
           tempNoued = new NouedH( nouedCurrent.etat , 0  , col , this.getSize() );
           tempNoued.addNouedSortH(listOvert);
           this.nbrNodeGener++;
       }
       while ( ! listOvert.isEmpty() ) {
           nouedCurrent = listOvert.removeFirst();
           this.nbrNodeDevelop++;
           if  ( nouedCurrent.row + 1 == this.getSize()  ) {
               Solution.copyMatrix(nouedCurrent.etat, board);
               return true;
            }
           for ( col = 0; col < this.getSize() ; col++ ){
                if (this.ifSafe(nouedCurrent.etat, nouedCurrent.row + 1  , col)) {
                    tempNoued = new NouedH( nouedCurrent.etat , nouedCurrent.row + 1  , col , this.getSize() );
                    tempNoued.addNouedSortH(listOvert);
                    this.nbrNodeGener++;
                }
           }
        }
        return false;
    }
}
```

Figure 11 : Classe Heurstique2Solution

Chapitre 4 : Expérimentation

1. Environnement Expérimentale

Les tests ont été effectués sur un ordinateur équipé d'un processeur Intel Core i7-7700HQ, de 16 Go de RAM et fonctionnant sous Windows 10 comme système d'exploitation.

2. Interface graphique

Notre interface graphique pour le problème des N-reines est simple et efficace. Elle se compose de deux sections principales la zone de l'échiquier au départ est vide et la zone d'options .Dans la zone de l'échiquier vide, l'utilisateur peut visualiser la solution une fois qu'elle est trouvée. Dans la zone d'options, l'utilisateur peut spécifier la valeur de N. Ensuite, il peut sélectionner l'algorithme de résolution souhaité et cliquer sur le bouton "Lancer" pour commencer la recherche de solution.

Une fois le bouton "Lancer" cliqué, l'interface affiche la solution du problème, ainsi que des statistiques telles que le nombre de nœuds générés et développés, et le temps d'exécution.

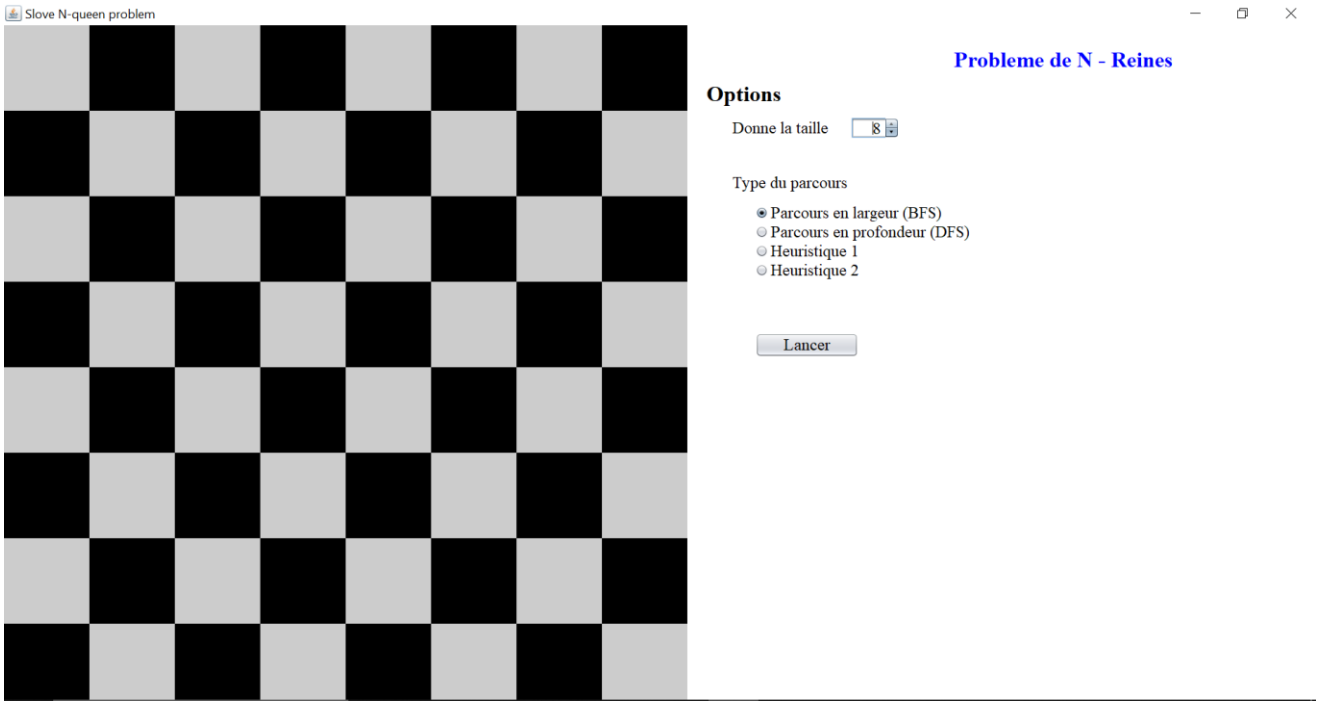

Figure 12 : Capture d'écran de l'interface avant la recherche d'une solution pour le problème des N-reines

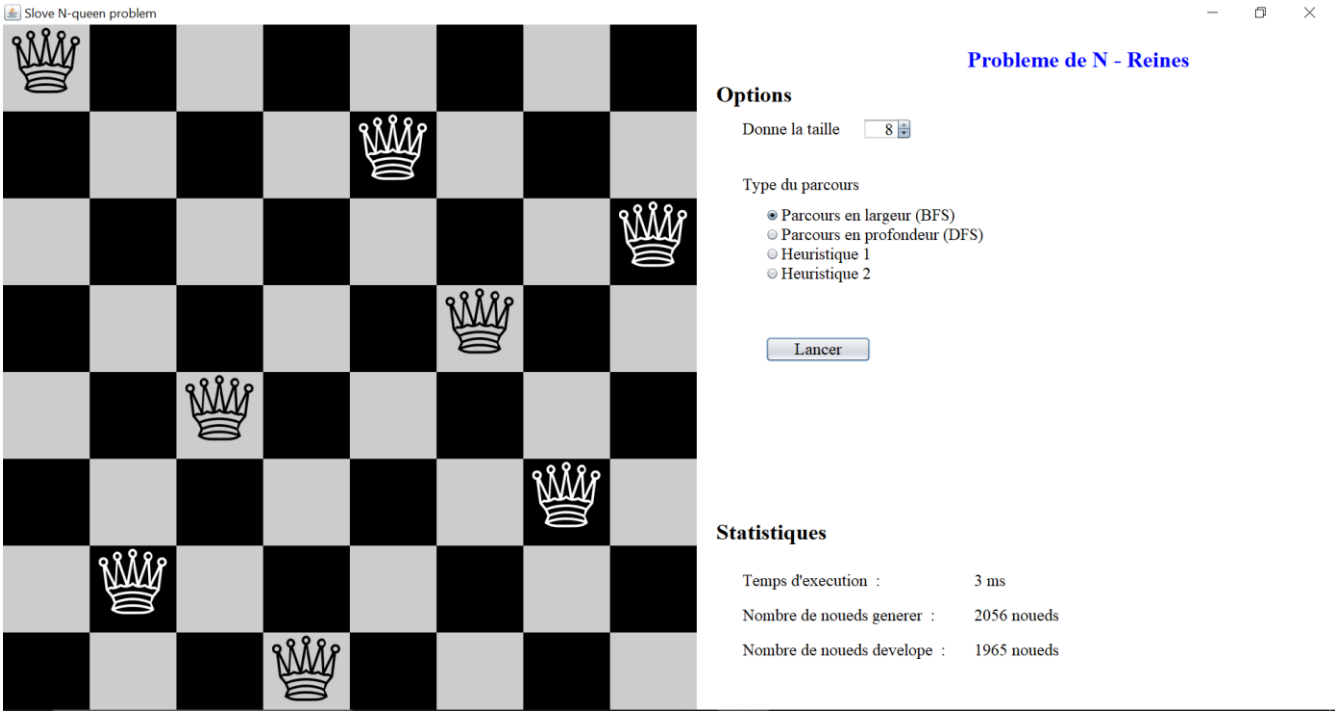

Figure 13 : Capture d'écran de l'interface après la recherche d'une solution pour le problème des N-reines

3. Illustration des algorithmes à l'aide d'un exemple

3.1 Exemple d'utilisation de l'algorithme DFS pour le problème des 4 reines

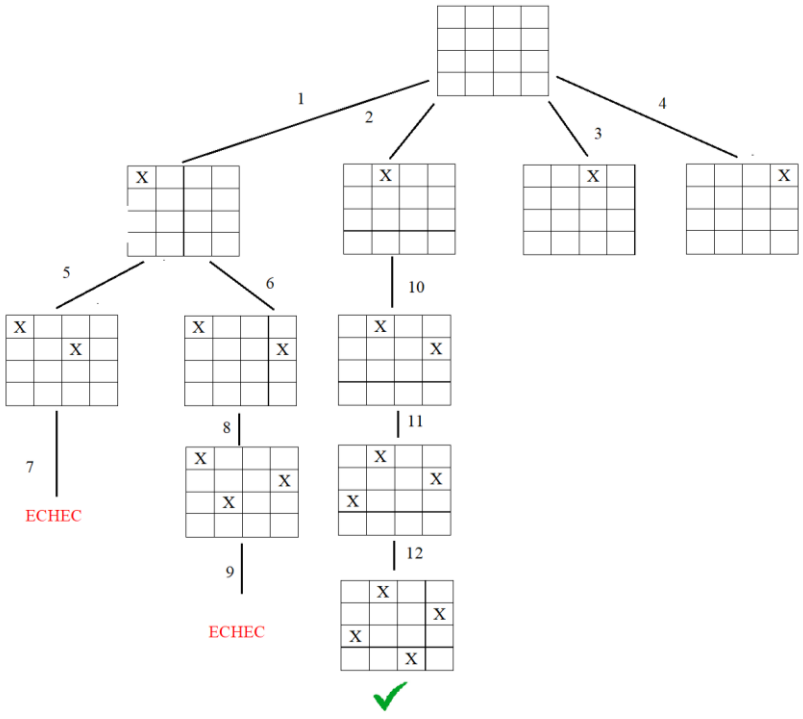

Figure 14 : Construction de l'arbre de recherche du l'algorithme DFS pour la résolution du problème des 4 reines

3.2 Exemple d'utilisation de l'algorithme BFS pour le problème des 4 reines

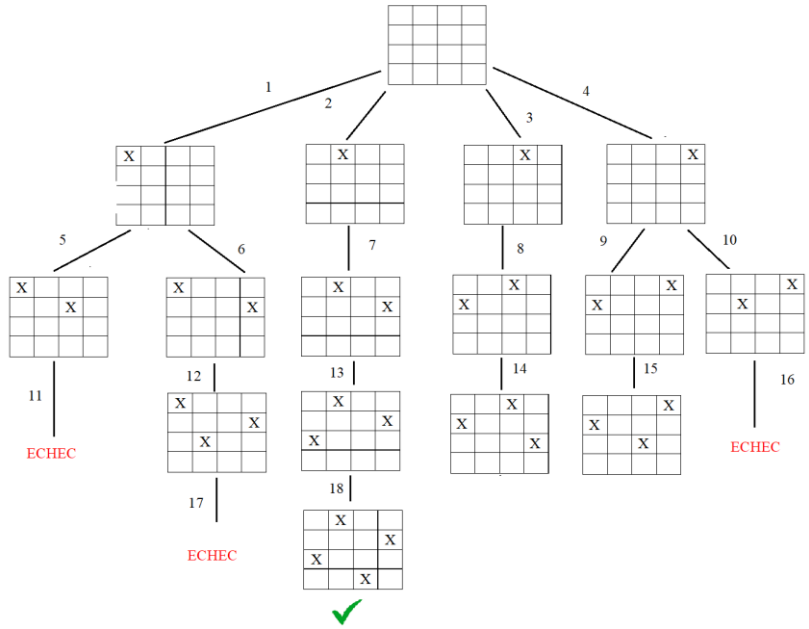

Figure 15 : Construction de l'arbre de recherche du l'algorithme BFS pour la résolution du problème des 4 reines

3.3 Exemple d'utilisation de l'algorithme A* avec heuristique 1 pour le problème des 4 reines

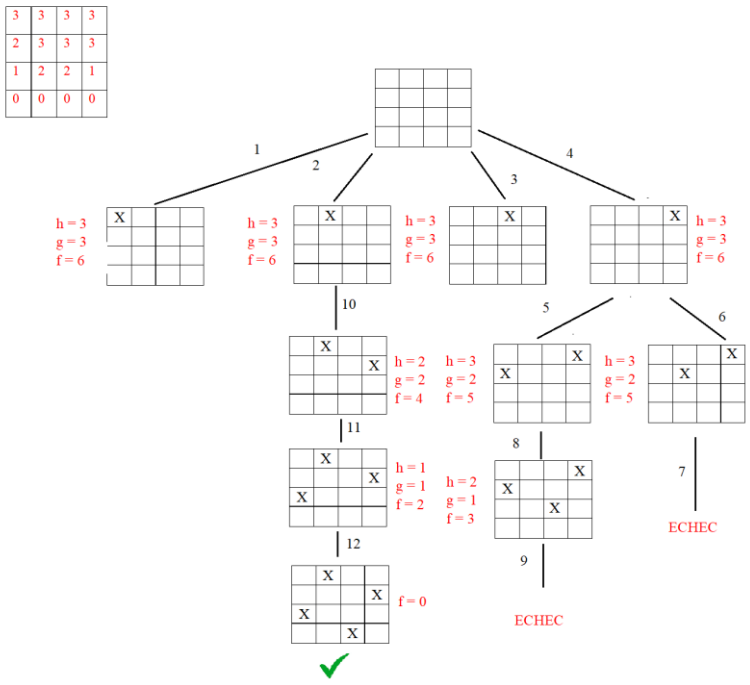

Figure 16 : Construction de l'arbre de recherche du l'algorithme A* avec heuristique 1 pour la résolution du problème des 4 reines

3.4 Exemple d'utilisation de l'algorithme A* avec heuristique 2 pour le problème des 4 reines

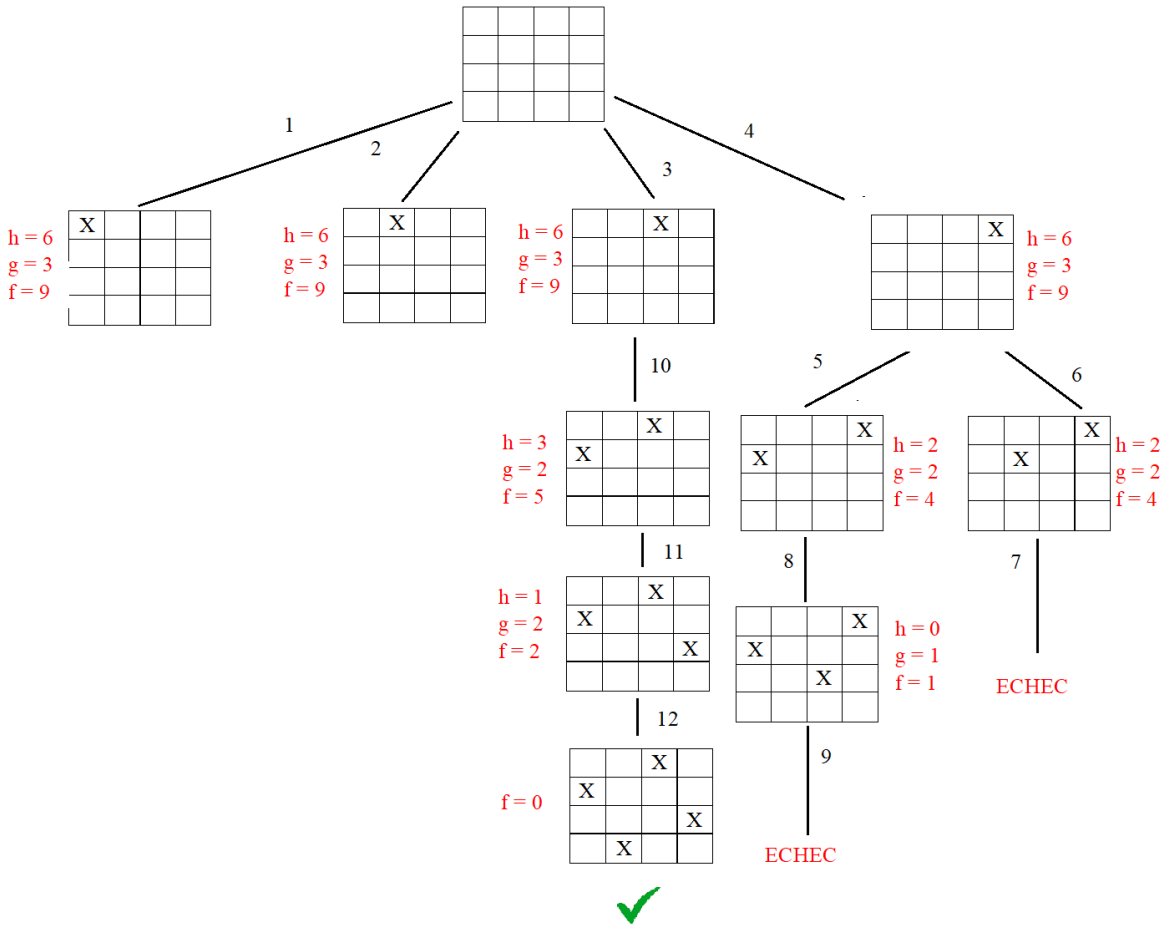

Figure 17 : Construction de l'arbre de recherche du l'algorithme A* avec heuristique 2 pour la résolution du problème des 4 reines

4. Résultat expérimentaux

Nous avons testes les 4 algorithmes pour résoudre le problème des N-reines, en mesurant le temps d'exécution, le nombre de nœuds générés et développés, ainsi que la première solution trouvée pour différentes valeurs de N.

Nous représentons ici la solution trouvée sous forme de tableau, où chaque colonne représente une ligne du l'échiquier, et chaque chiffre représente la colonne où la reine doit être placée pour résoudre le problème.

Les résultats expérimentaux obtenus sont présentés ci-dessous.

4.1 Algorithme DFS

Table 1 : Résultats des tests de l'algorithme DFS

Taille	Nombre de nœuds génères	Nombre de nœuds développes	Temps d'exécution (ms)	Solution
8	124	114	0	0 | 4 | 7 | 5 | 2 | 6 | 1 | 3
10	124	103	0	0 | 2 | 5 | 7 | 9 | 4 | 8 | 1 | 3 | 6
12	295	262	1	0 | 2 | 4 | 7 | 9 | 11 | 5 | 10 | 1 | 6 | 8 | 3
14	1944	1900	5	0 | 2 | 4 | 6 | 11 | 9 | 12 | 3 | 13 | 8 | 1 | 5 | 7 | 10
16	10112	10053	7	0 | 2 | 4 | 1 | 12 | 8 | 13 | 11 | 14 | 5 | 15 | 6 | 3 | 10 | 7 | 9
18	41377	41300	64	0 | 2 | 4 | 1 | 7 | 14 | 11 | 15 | 12 | 16 | 5 | 17 | 6 | 3 | 10 | 8 | 13 | 9
20	199733	199636	90	0 | 2 | 4 | 1 | 3 | 12 | 14 | 11 | 17 | 19 | 16 | 8 | 15 | 18 | 7 | 9 | 6 | 13 | 5 | 10
22	1737308	1737189	859	0 | 2 | 4 | 1 | 3 | 9 | 13 | 16 | 19 | 12 | 18 | 21 | 17 | 7 | 20 | 11 | 8 | 5 | 15 | 6 | 10 | 14
24	411755	411609	396	0 | 2 | 4 | 1 | 3 | 8 | 10 | 13 | 17 | 21 | 18 | 22 | 19 | 23 | 9 | 20 | 5 | 7 | 11 | 15 | 12 | 6 | 16 | 14
26	397880	397700	267	0 | 2 | 4 | 1 | 3 | 8 | 10 | 12 | 14 | 20 | 22 | 24 | 19 | 21 | 23 | 25 | 9 | 6 | 15 | 11 | 7 | 5 | 17 | 13 | 18 | 16
28	3006508	3006299	2208	0 | 2 | 4 | 1 | 3 | 8 | 10 | 12 | 14 | 16 | 22 | 24 | 21 | 27 | 25 | 23 | 26 | 6 | 11 | 15 | 17 | 7 | 9 | 13 | 19 | 5 | 20 | 18
30	56429852	56429620	52961	0 | 2 | 4 | 1 | 3 | 8 | 10 | 12 | 14 | 6 | 22 | 25 | 27 | 24 | 21 | 23 | 29 | 26 | 28 | 15 | 11 | 9 | 7 | 5 | 17 | 19 | 16 | 13 | 20 | 18
32	87491694	87491426	64011	0 | 2 | 4 | 1 | 3 | 8 | 10 | 12 | 14 | 5 | 17 | 23 | 25 | 29 | 24 | 30 | 27 | 31 | 26 | 28 | 15 | 18 | 9 | 7 | 16 | 11 | 20 | 6 | 13 | 22 | 19 | 21
34	2294605599	2294605285	2322714	0 | 2 | 4 | 1 | 3 | 8 | 10 | 12 | 14 | 5 | 17 | 19 | 25 | 27 | 30 | 32 | 26 | 28 | 33 | 31 | 29 | 11 | 9 | 6 | 15 | 18 | 7 | 21 | 13 | 24 | 16 | 23 | 20 | 22 |

D'après les résultats, on remarque que DFS a produit des résultats très rapide pour les petites valeurs de N, chaque fois que la valeur de N augmente, le nombre de nœuds génère augmente également, cela implique que le temps d'exécution augmente. Notez que cet algorithme arrive toujours à trouver une solution si elle existe.

4.2 Algorithme BFS

Table 2 : Résultats des tests de l'algorithme BFS

Taille	Nombre de nœuds génères	Nombre de nœuds développes	Temps d'exécution (ms)	Solution
8	2056	1965	5	0 | 4 | 7 | 5 | 2 | 6 | 1 | 3 |
10	35538	34815	46	0 | 2 | 5 | 7 | 9 | 4 | 8 | 1 | 3 | 6
12	856188	841989	1137	0 | 2 | 4 | 7 | 9 | 11 | 5 | 10 | 1 | 6 | 8 | 3
14	27358552	26992957	116032	0 | 2 | 4 | 6 | 11 | 9 | 12 | 3 | 13 | 8 | 1 | 5 | 7 | 10
16	-	-	-	-

```
run:
Pour N = 16 :
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.LinkedList.linkLast(LinkedList.java:142)
        at java.util.LinkedList.add(LinkedList.java:338)
        at solution.BFSsolution.solveN_Queens(BFSsolution.java:52)
        at test.Main.main(Main.java:34)
C:\Users\client\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 24 minutes 36 seconds)
```

Figure 18 : Erreur lors de l'execution du l'algorithme BFS

Les résultats dans le tableau, montrent que l'algorithme BFS est capable de trouver des solutions pour les petites valeurs de N. Cependant, lorsque la valeur de N augmente, le nombre de nœuds générés (le nombre d'états à explorer) augmente également. En conséquence, pour des valeurs de N supérieures ou égales à 16, l'algorithme BFS n'a pas pu fournir de solution en raison d'une erreur de type java.lang.outOfMemoryError qui se produit en raison du nombre important de nouds génères. (Voir la figure au-dessus)

4.3 Algorithme A* pour N-Reines

A/- Heuristique 1 :

Table 3 : Résultats des tests de l'algorithme A* (Heurstique 1)

Taille	Nombre de nœuds génères	Nombre de nœuds développes	Temps d'exécution (ms)	Solution
8	140	128	2	7 | 3 | 0 | 2 | 5 | 1 | 6 | 4
10	184	162	2	9 | 0 | 5 | 3 | 1 | 7 | 2 | 8 | 6 | 4
12	183	148	3	11 | 0 | 10 | 7 | 4 | 1 | 8 | 2 | 9 | 6 | 3 | 5
14	3787	3738	10	13 | 0 | 12 | 7 | 5 | 3 | 1 | 10 | 8 | 11 | 2 | 4 | 6 | 9
16	19155	19098	29	15 | 0 | 14 | 11 | 1 | 5 | 7 | 12 | 3 | 13 | 2 | 9 | 6 | 4 | 10 | 8
18	33891	33808	55	17 | 0 | 16 | 1 | 12 | 10 | 6 | 3 | 14 | 2 | 4 | 13 | 15 | 8 | 11 | 5 | 7 | 9
20	46095	45991	93	19 | 0 | 18 | 1 | 17 | 13 | 3 | 7 | 9 | 4 | 14 | 5 | 15 | 2 | 16 | 11 | 6 | 8 | 10 | 12
22	158034	157919	201	21 | 0 | 20 | 1 | 19 | 2 | 14 | 7 | 10 | 3 | 15 | 17 | 4 | 6 | 18 | 16 | 11 | 13 | 8 | 5 | 9 | 12
24	479512	479366	637	23 | 0 | 22 | 1 | 21 | 2 | 20 | 14 | 9 | 13 | 4 | 7 | 17 | 3 | 5 | 18 | 16 | 19 | 10 | 12 | 15 | 6 | 8 | 11
26	803347	803183	1161	25 | 0 | 24 | 1 | 23 | 2 | 22 | 3 | 15 | 10 | 12 | 19 | 6 | 8 | 17 | 20 | 4 | 21 | 18 | 5 | 13 | 11 | 7 | 14 | 16 | 9
28	7676439	7676243	12315	27 | 0 | 26 | 1 | 25 | 2 | 24 | 3 | 18 | 16 | 14 | 20 | 7 | 21 | 6 | 8 | 5 | 22 | 4 | 19 | 23 | 11 | 13 | 17 | 9 | 12 | 10 | 15
30	19345659	19345427	30898	29 | 0 | 28 | 1 | 27 | 2 | 26 | 3 | 25 | 19 | 15 | 13 | 7 | 22 | 8 | 23 | 5 | 20 | 9 | 4 | 6 | 21 | 12 | 24 | 16 | 18 | 10 | 14 | 11 | 17 |
32	505161639	505161362	1051804	31 | 0 | 30 | 1 | 29 | 2 | 28 | 3 | 27 | 21 | 26 | 16 | 12 | 15 | 9 | 7 | 4 | 8 | 24 | 22 | 6 | 25 | 23 | 17 | 5 | 14 | 11 | 20 | 10 | 19 | 13 | 18
34	1036965065	1036964750	1917282	33 | 0 | 32 | 1 | 31 | 2 | 30 | 3 | 29 | 4 | 22 | 20 | 18 | 15 | 27 | 8 | 10 | 7 | 9 | 6 | 24 | 28 | 25 | 23 | 5 | 26 | 14 | 12 | 17 | 21 | 13 | 11 | 16 | 19

En examinant les résultats du test de l'algorithme A* avec l'heuristique 1, on peut remarquer que plus la taille du problème augmente, plus le nombre de nœuds générés et développés augmente également. Cela est dû au fait que l'espace de recherche devient plus grand avec des problèmes de plus grande taille, ce qui augmente la complexité de l'algorithme.

Cependant, le temps d'exécution reste relativement faible, ce qui suggère que l'algorithme est efficace pour résoudre ce type de problème

B/- Heuristique 2:

Table 4 : Résultats des tests de l'algorithme A* (Heuristique 2)

Taille	Nombre de nœuds génères	Nombre de nœuds développes	Temps d'exécution (ms)	Solution
8	86	75	2	7 | 3 | 0 | 2 | 5 | 1 | 6 | 4
10	121	98	2	9 | 7 | 4 | 2 | 0 | 5 | 1 | 8 | 6 | 3
12	358	325	7	11 | 9 | 7 | 4 | 2 | 0 | 6 | 1 | 10 | 5 | 3 | 8
14	1958	1915	19	13 | 11 | 9 | 7 | 2 | 4 | 1 | 10 | 0 | 5 | 12 | 8 | 6 | 3
16	14814	14759	55	15 | 13 | 11 | 14 | 3 | 7 | 2 | 4 | 1 | 10 | 0 | 9 | 12 | 5 | 8 | 6 |
18	36205	36128	138	17 | 15 | 13 | 16 | 10 | 3 | 6 | 2 | 5 | 1 | 12 | 0 | 11 | 14 | 7 | 9 | 4 | 8
20	6824	6725	49	19 | 17 | 15 | 18 | 16 | 1 | 7 | 5 | 2 | 0 | 12 | 4 | 11 | 3 | 14 | 10 | 13 | 9 | 6 | 8
22	50041	49922	262	21 | 19 | 17 | 20 | 18 | 1 | 9 | 7 | 5 | 3 | 0 | 15 | 4 | 12 | 16 | 13 | 2 | 10 | 6 | 14 | 11 | 8
24	45728	45583	275	23 | 21 | 19 | 22 | 20 | 1 | 10 | 6 | 11 | 9 | 2 | 0 | 17 | 4 | 12 | 16 | 18 | 3 | 15 | 13 | 8 | 14 | 5 | 7
26	331835	331659	1860	25 | 23 | 21 | 24 | 22 | 1 | 11 | 15 | 20 | 4 | 10 | 5 | 9 | 2 | 0 | 3 | 17 | 19 | 16 | 18 | 12 | 8 | 13 | 7 | 14 | 6
28	148238	148036	943	27 | 25 | 23 | 26 | 24 | 1 | 12 | 2 | 22 | 11 | 7 | 20 | 3 | 6 | 8 | 19 | 21 | 18 | 5 | 17 | 4 | 0 | 10 | 15 | 9 | 14 | 16 | 13
30	2804567	2804337	20797	29 | 27 | 25 | 28 | 26 | 1 | 13 | 19 | 24 | 14 | 23 | 6 | 10 | 5 | 2 | 0 | 18 | 3 | 7 | 22 | 4 | 21 | 16 | 20 | 15 | 12 | 9 | 17 | 8 | 11
32	1223333	1223072	10452	31 | 29 | 27 | 30 | 28 | 1 | 14 | 2 | 26 | 13 | 25 | 4 | 12 | 15 | 11 | 3 | 7 | 23 | 19 | 22 | 6 | 0 | 5 | 21 | 18 | 24 | 10 | 16 | 20 | 9 | 17 | 8
34	17636869	17636575	175671	33 | 31 | 29 | 32 | 30 | 1 | 15 | 21 | 28 | 16 | 27 | 19 | 17 | 11 | 5 | 3 | 6 | 0 | 2 | 8 | 26 | 23 | 4 | 24 | 18 | 25 | 13 | 22 | 20 | 14 | 10 | 7 | 9 | 12

Selon les résultats des tests de l'algorithme A* avec l'heuristique 2, il est clair que le temps d'exécution ne croît pas de manière exponentielle.

Cet algorithme est capable de trouver des solutions dans un temps raisonnable.

5. Etude comparative pour 4 méthodes

L'objectif de cette étude comparative est d'évaluer les performances de quatre méthodes dans un contexte spécifique afin de déterminer celle qui est la plus efficace.

Pour réaliser cette évaluation, nous avons créé une table pour comparer la solution trouve pour les quatre algorithmes et tracé trois graphes : le premier pour le nombre de nœuds générés, le deuxième pour le nombre de nœuds développés et le dernier pour le temps d'exécution.

Table 5 : Comparaison des solutions trouvées pour les quatre méthodes

Taille	Solution BFS	Solution DFS
8	0 | 4 | 7 | 5 | 2 | 6 | 1 | 3	0 | 4 | 7 | 5 | 2 | 6 | 1 | 3
10	0 | 2 | 5 | 7 | 9 | 4 | 8 | 1 | 3 | 6	0 | 2 | 5 | 7 | 9 | 4 | 8 | 1 | 3 | 6
12	0 | 2 | 4 | 7 | 9 | 11 | 5 | 10 | 1 | 6 | 8 | 3	0 | 2 | 4 | 7 | 9 | 11 | 5 | 10 | 1 | 6 | 8 | 3
14	0 | 2 | 4 | 6 | 11 | 9 | 12 | 3 | 13 | 8 | 1 | 5 | 7 | 10	0 | 2 | 4 | 6 | 11 | 9 | 12 | 3 | 13 | 8 | 1 | 5 | 7 | 10

Taille	Solution heuristique 1	Solution heuristique 2
8	7 | 3 | 0 | 2 | 5 | 1 | 6 | 4	7 | 3 | 0 | 2 | 5 | 1 | 6 | 4
10	9 | 0 | 5 | 3 | 1 | 7 | 2 | 8 | 6 | 4	9 | 7 | 4 | 2 | 0 | 5 | 1 | 8 | 6 | 3
12	11 | 0 | 10 | 7 | 4 | 1 | 8 | 2 | 9 | 6 | 3 | 5	11 | 9 | 7 | 4 | 2 | 0 | 6 | 1 | 10 | 5 | 3 | 8
14	13 | 0 | 12 | 7 | 5 | 3 | 1 | 10 | 8 | 11 | 2 | 4 | 6 | 9	13 | 11 | 9 | 7 | 2 | 4 | 1 | 10 | 0 | 5 | 12 | 8 | 6 | 3

D'après le tableau, nous pouvons observer que les deux algorithmes de recherche en profondeur DFS et en largeur BFS parviennent à la même solution. Cela est dû au fait que les nœuds développés dans BFS sont également développés dans DFS, ce qui garantit que la solution est la même pour les deux algorithmes. Cependant, lorsqu'il s'agit d'utiliser des heuristiques, chaque algorithme développe ses propres nœuds pour rechercher une solution.

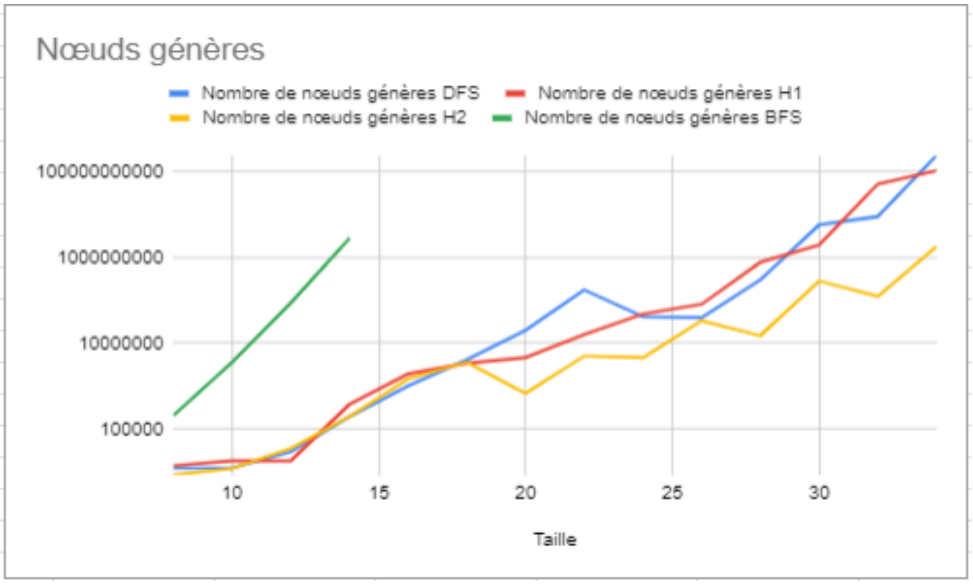

Figure 19 : Graphique pour comparer le nombre de nœuds générés pour les quatre méthodes

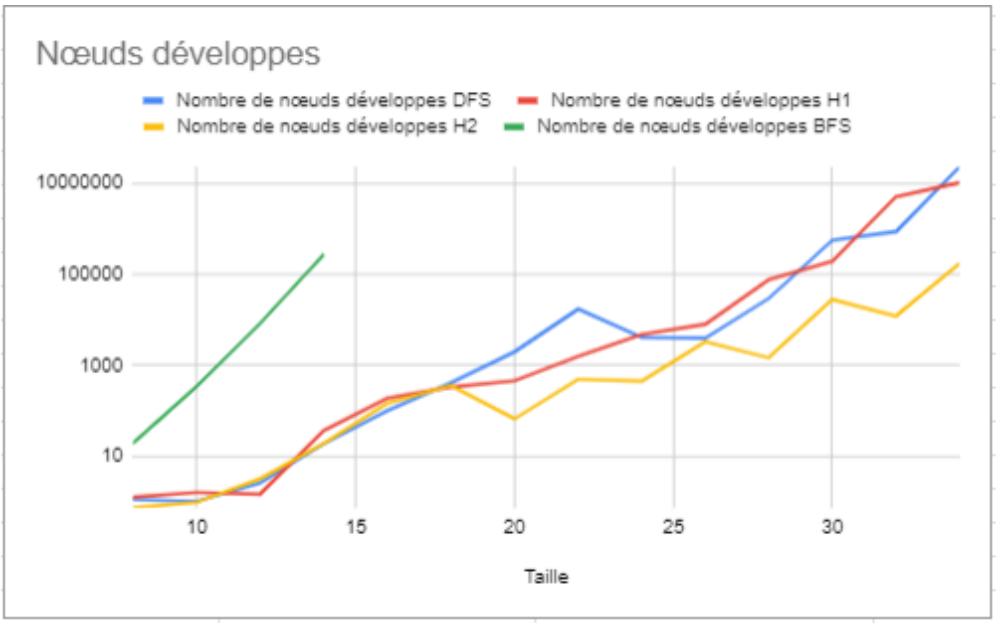

Figure 20 : Graphique pour comparer le nombre de nœuds développe pour les quatre méthodes

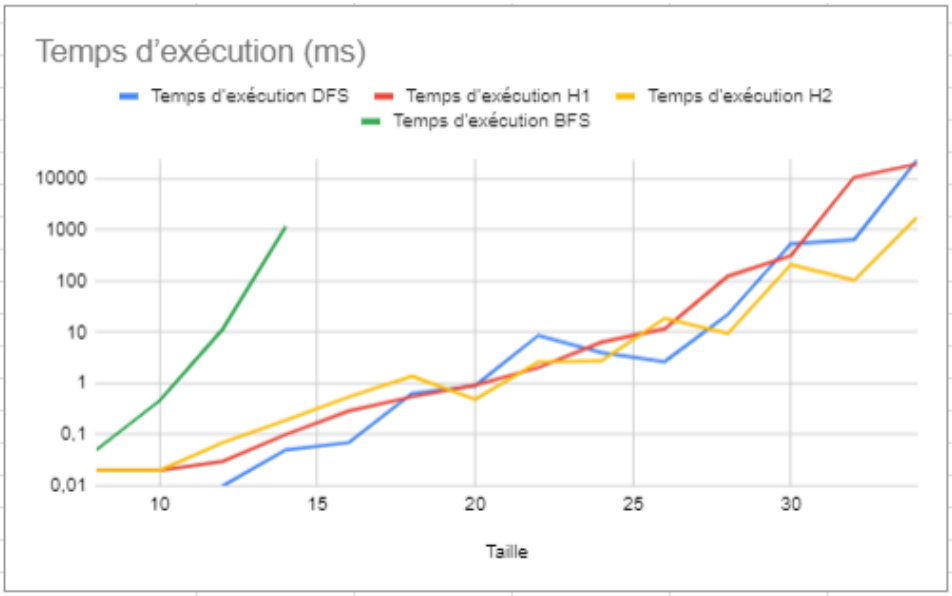

Figure 21 : Graphique pour comparer le temps d'exécution pour les quatre méthodes

D'après les graphes, on observe que l'algorithme BFS est très coûteux en termes de temps et de mémoire. En effet, il génère et développe un grand nombre de nœuds, ce qui entraîne une augmentation considérable du temps d'exécution pour trouver une solution par rapport aux autres algorithmes.

En ce qui concerne les trois algorithmes restants (DFS, A* avec H1 et A* avec H2), nous pouvons constater que pour les petites tailles de problèmes, il n'y a pas de différence significative dans les performances de ces algorithmes. Cependant, lorsque la taille du problème augmente, nous avons constaté que l'algorithme A* avec H2 est devenu plus performant en termes de mémoire et de temps d'exécution comparé aux autres algorithmes. Cela est dû à la meilleure heuristique utilisée par cet algorithme (H2), qui a permis de réduire le nombre de nœuds explorés et donc d'améliorer les performances. En d'autres termes, l'algorithme A* avec H2 est capable de trouver une solution plus rapidement et en utilisant moins de mémoire pour les problèmes de grande taille.

En conclusion, l'algorithme BFS est le plus adapté pour résoudre le problème N-Reines pour des tailles de problèmes relativement petites, tandis que l'algorithme A* avec heuristique 2 est plus adapté pour des tailles de problèmes plus grandes.

Conclusion générale

En conclusion, nous avons étudié les performances de différents algorithmes de recherche pour résoudre le problème des N-Reines. Nous n’avons constaté que les algorithmes de recherche aveugle tels que BFS peuvent être efficaces pour des tailles de problèmes relativement petites, mais peuvent devenir impraticables pour des tailles de problèmes plus grandes en raison de la quantité élevée de nœuds générés. D'autre part, les algorithmes de recherche heuristique tels que A* peuvent être plus efficaces pour résoudre des problèmes plus grands en utilisant des heuristiques pour guider la recherche de solutions.

Il est important de noter que d'autres approches peuvent être utilisées pour résoudre ce problème, telles que les algorithmes génétiques que nous allons implémenter dans la partie 2 de ce rapport. Les algorithmes génétiques utilisent des techniques d'évolution pour trouver des solutions potentielles, et peuvent être efficaces pour résoudre des problèmes difficiles ou complexes. En fin de compte, le choix de l'algorithme dépendra des détails du problème spécifique et des objectifs de l'application.

Reference

[1] Intelligence Artificielle et Résolution de Problèmes, Dr H.Moulai

[2] Randomness in Heuristic: an experimental Investigation for the maximum satisfiability problem, Mme H.DRIAS (USTHB)

[3] Cours Résolution de Pb Master SII, H. AZZOUNE