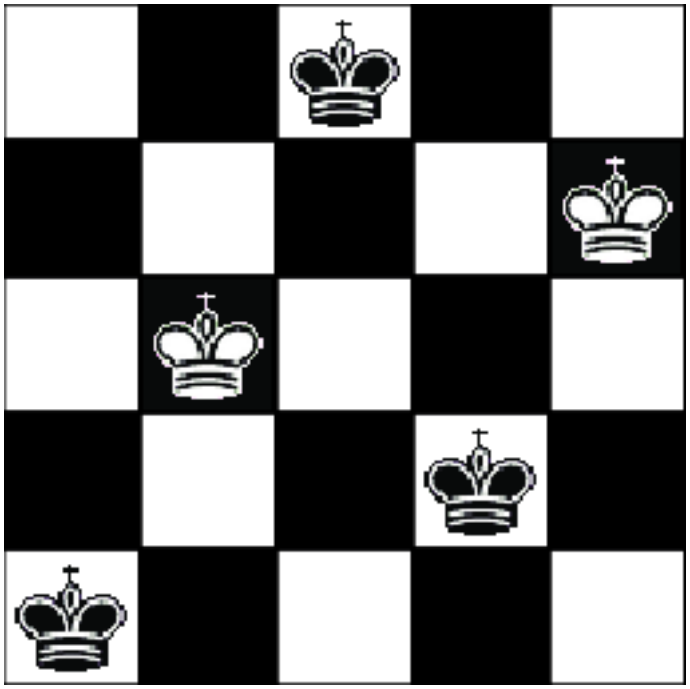
Université de sciences et de la technologie Houari Boumediene

La Résolution Du Problème Des n-reines Par Les Algorithmes Exactes Et A*

Partie 1 : Recherche basée espace des états



Présenté Par :

FERKOUS Sarah
KHAOUNI Amel

Professeur : M. MOULAI

26 Mars 2023

Table des matières

Liste des tableaux

Table des figures

Introduction

Notre projet étudiera la résolution du problème de n-reine par des méthodes exactes (BDF et DBF) puis par heuristiques, pour différencier ces approches et décider laquelle est la meilleure en fonction du temps d'exécution, de la complexité temporelle et spatiale (nombre de nœuds créés) et de nombreuses autres mesures de performance.

Ce travail montrera en détail la modélisation du problème de n-reine et comment on le présente en mémoire de la meilleure façon, et nous justifierons également la sélection structures de données utilisées.

L'essentiel est de savoir comment résoudre ce problème ? pour résoudre le problème des n-reines, on peut utiliser la méthode exacte pour les problèmes de petite taille et l'algorithme de recherche A* pour les problèmes de grande taille. Les deux méthodes ont leurs avantages et leurs inconvénients, et il est important de choisir la méthode appropriée en fonction de la taille et de la complexité du problème à résoudre. Dans cette partie vous verrez une description des algorithmes qui ont été adaptés au notre problème, Avec différentes descriptions de classes, tout cela sera implémenté en utilisant le langage Java.

La partie suivante est la partie test, vérification et une étude comparative. Avec une interface graphique (également développée en java) qui affichera la solution avec ces métriques de performance.

Chapitre 1

Approche Espace Des Etats

1.1 Description des concepts

1.1.1 Etat

Un état est une description complète de l'état actuel d'un système ou d'un problème donné. l'état représente souvent une configuration possible des variables du problème.

1.1.2 Opérateurs

Les opérateurs sont des actions qui transforment un état en un autre état. Les opérateurs sont utilisés pour explorer l'espace de recherche d'un problème et trouver une solution.

1.1.3 Successeurs

Les successeurs sont les états accessibles à partir d'un état donné en appliquant un opérateur. et sont utilisés pour explorer l'espace de recherche du problème.

1.2 Algorithme de Recherche en profondeur d'abord (DFS)

1.2.1 Description

Comme l'indique son nom, l'algorithme de recherche en profondeur d'abord démarre de la racine, explore le premier chemin de ses successeurs (une branche de l'arbre du haut vers le bas) jusqu'au moment ou le sommet n'a plus de successeurs non visités. Ensuite il remonte d'un niveau pour vérifier s'il ne reste pas de sommets à visiter (si il ne reste plus de sommets candidats, nous fermons alors le sommet courant), et ainsi de suite.

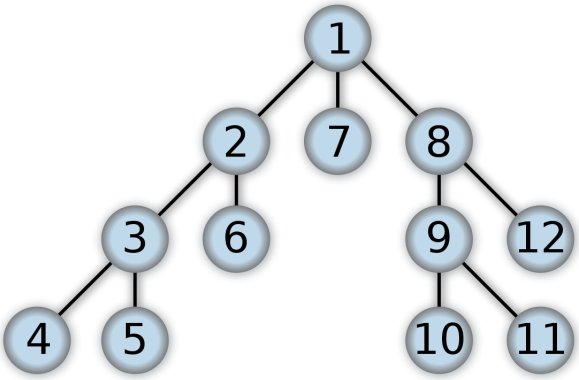


FIGURE 1.1 – Arbre de recherche (DFS)

Algorithm 1 DFS(G, s)

```
1:  for chaque sommet u de G do
2:      marquer u comme non visité
3:  end for
4:  empiler s sur une pile S
5:  while S n'est pas vide do
6:      u ← dépiler S
7:      if u n'est pas visité then
8:          marquer u comme visité
9:          for chaque voisin v de u do
10:             empiler v sur S
11:         end for
12:     end if
13: end while
```

1.2.2 Complexité

la complexité temporelle de l'algorithme dépend de la structure du graphe à parcourir, elle est de $O(|S| + |A|)$ où $|S|$ est le nombre de sommets et $|A|$ le nombre d'arcs si le graphe est représenté par une liste d'adjacence. Et $O(A^2)$ si le graphe est représenté par une matrice d'adjacence.
La complexité spatiale de l'algorithme DFS est de $O(|S|)$ dans le pire des cas, mais peut varier en fonction de l'implémentation.

1.3 Algorithme de Recherche en largeur d'abord (BFS)

L'algorithme de recherche en largeur d'abord (BFS, pour Breadth-First Search) est un algorithme de parcours des graphes qui explore tous les sommets d'un graphe en commençant par le sommet de départ et en visitant tous les voisins de ce sommet, puis tous les voisins de ces voisins, et ainsi de suite, jusqu'à ce que tous les sommets accessibles aient été visités.

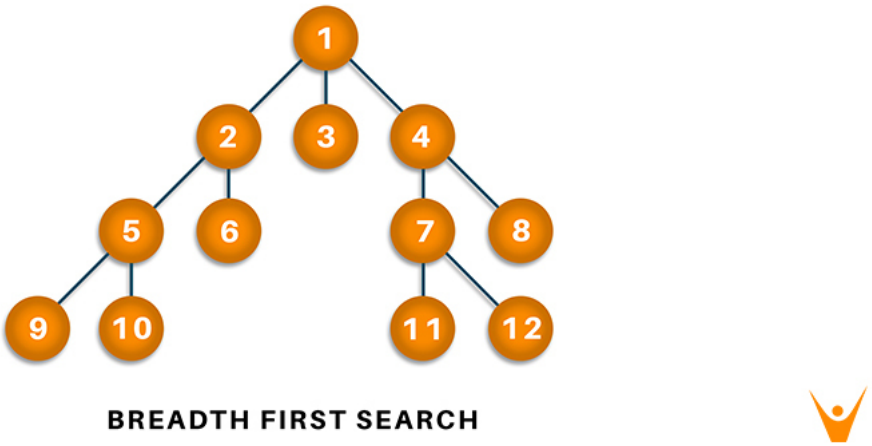


FIGURE 1.2 – Arbre de recherche (BFS)

Algorithm 2 BFS(G, s)

```
1:  for chaque sommet u de G do
2:      marquer u comme non visité
3:  end for
4:  créer une file F
5:  marquer s comme visité et l'ajouter à F
6:  while F n'est pas vide do
7:      u ← retirer le premier élément de F
8:      for chaque voisin v de u do
9:          if v n'est pas visité then
10:             marquer v comme visité et l'ajouter à F
11:         end if
12:     end for
13: end while
```

1.3.1 Complexité

Au niveau de la complexité, les algorithmes DFS et BFS sont identiques, car chaque arc n'est évalué qu'une seule fois. Ils diffèrent seulement par leur mode opératoire.

1.4 Méthodes heuristiques

1.4.1 heuristique

heuristique ou aide à découvrir, c'est une fonction associée à un état donné, elle permet d'accélérer la recherche, en développant les nœuds les plus prometteurs où la recherche est dirigée vers l'état but. Les heuristiques sont utilisées lorsque le problème possède une complexité exponentielle.
Une heuristique est **admissible** c.-à-d. que pour tout nœud n à partir duquel un nœud but peut être atteint, h(n) <= au cout du chemin optimal de n à un nœud but.

1.4.2 Algorithme A*

est un algorithme de recherche de chemin qui utilise une heuristique pour guider la recherche vers la solution la plus optimale. Sa fonction est définie comme suit : $f(n) = g(n) + h(n)$ tel que : $h(n)$ est l'heuristique et $g(n)$ est le coût du chemin déja parcouru.

1.4.3 Complexité

La complexité de l'algorithme A* dépend de plusieurs facteurs, notamment :

- la structure du graphe à explorer.
- la qualité de la fonction heuristique
- le nombre de nœuds à explorer pour atteindre la solution.

Dans le pire des cas, l'algorithme A* peut explorer tous les nœuds du graphe, ce qui correspond à une complexité en temps exponentielle en fonction du nombre de nœuds O(b^d), où **b** est le facteur de branchement moyen du graphe et **d** est la profondeur maximale de la solution.
Cependant, si la fonction heuristique est admissible, la complexité peut être considérablement réduite, en moyenne, à O($b^{d/2}$).

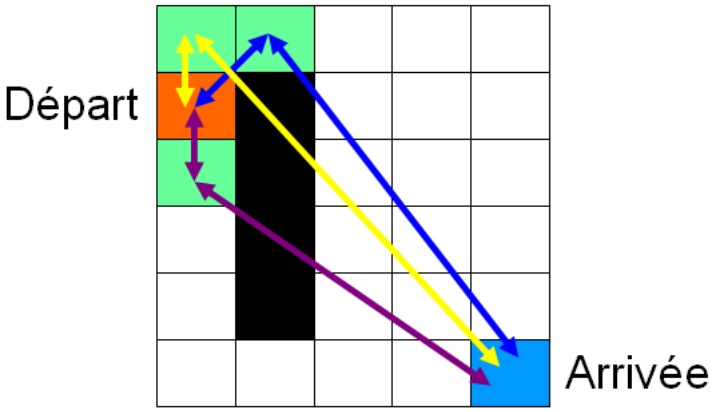


FIGURE 1.3 – A*

Algorithm 3 A*(départ, arrivée, heuristique)

Ouverts ← $depart$ ▷ Ensemble des noeuds à explorer
Fermés ← $null$ ▷ Ensemble des noeuds explorés
Coût ← $depart : 0$ ▷ Coûts cumulés pour chaque noeud atteint depuis le départ
Chemin ← $dpart : None$ ▷ Noeuds précédents pour chaque noeud atteint depuis le départ
while Ouverts n'est pas vide **do**
    actuel ← le noeud dans Ouverts avec le coût total le plus faible
    Ouverts ← Ouverts \ actuel
    Fermés ← Fermés ∪ actuel
    if actuel == arrivée **then**
        $cheminfinal = []$
        while actuel **do**
            $cheminfinal.append(actuel)$
            actuel ← Chemin[actuel]
        end while
        return $cheminfinal.reverse()$
    end if
    for voisin dans voisins(actuel) **do**
        coût ← Coût[actuel] + distance(actuel, voisin)
        if voisin dans Fermés et coût ≥ Coût[voisin] **then**
            $Continue$
        end if
        if voisin not in Ouverts or coût < Coût[voisin] **then**
            Coût[voisin] ← coût
            heuristiqueestimée ← coût + heuristique(voisin, arrivée)
            Chemin[voisin] ← actuel
            if voisin not in Ouverts **then**
                Ouverts ← Ouverts ∪ (voisin, heuristiqueestimée)
            else
                Ouverts ← Ouverts.mettreàjour(voisin, heuristiqueestimée)
            end if
        end if
    end for
end while
return None

Chapitre 2

Le Problème Des N-reines

2.1 Description du problème de n-reine

En **1848**, un joueur d'échecs allemand, **Max Bezzel** , pose le problème suivant : est-il possible de placer 8 reines sur l'échiquier de façon qu'aucune ne soit « en prise » (sous le feu d'une autre)? Rappelons en effet que pour qu'une reine ne soit menacée par aucune autre reine trois conditions doivent être remplies :

1. il ne doit pas y avoir d'autres reines sur sa colonne.
2. il ne doit pas y avoir d'autres reines sur sa ligne
3. il ne doit pas y avoir d'autres reines sur "ses diagonales".

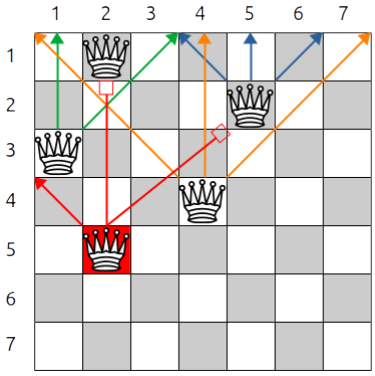
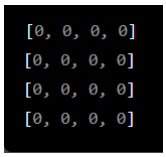


FIGURE 2.1 – Echiquier avec 7 reines

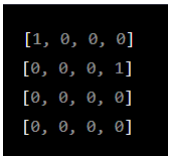
2.2 La modélisation du problème

2.2.1 La génération d'une instance du problème

Pour représenter une instance du problème des N-reines on peut utiliser une matrice (tableau à deux dimensions) de taille**nxn** où chaque case représente une position possible pour placer une reine tel que N est le nombre de reine.
Chaque case contient la valeur 0 pour indiquer qu'elle est vide, ou 1 pour indiquer qu'elle contient une reine



(a) matrice de 4x4 vide



(b) matrice de 4x4 avec 2 reines

FIGURE 2.2 – Structure de données de n-reine

2.2.2 La modélisation d'une solution au problème

Pour représenter la solution d'un problème des n reines, on peut utiliser un tableau à une dimension de longueur n, où chaque élément représente la colonne de la reine dans chaque ligne.
Voici le psesudo code de l'algorithme de génération d'une solution aléatoirement :

Algorithm 4 Generate solution Aléatoire

function GENERATE($solution$, n)
  for row de 1 a n **do**
    $col \leftarrow random(n)$
    $board[row][col] \leftarrow 1$
    $solution \leftarrow col$
  end for
end function

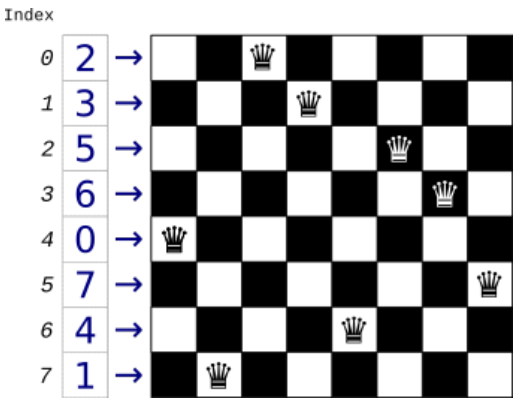


FIGURE 2.3 – Représentation d'une solution de problème de n-reine

2.2.3 La vérification de la validité d'une solution

1. Si deux reines sont sur la même colonne, elles se menacent mutuellement.
2. Si deux reines sont sur la même diagonale, elles se menacent mutuellement.

Pour vérifier cela, on peut calculer la différence de colonne et la différence de ligne entre les deux positions et vérifier si ces deux valeurs sont égales en valeur absolue.
Voici le pseudo code qui faire cette vérification :

```
function isValidSolution(solution)
    n ← length(solution)
    for i ← 0 to n − 1 do
        for j ← i + 1 to n − 1 do                ▷ Vérifier si deux reines sont sur la même colonne
            if solution[i] == solution[j] then
                return false
            end if                               ▷ Vérifier si deux reines sont sur la même diagonale
            if |, solution[i] − solution[j], | == |, i − j, | then
                return false
            end if
        end for
    end for
    return true
end function
```

2.2.4 L'Évaluation d'une solution

L'évaluation d'une solution pour le problème des n reines consiste à calculer le nombre de conflits entre les reines dans la solution donnée. Plus précisément, pour chaque paire de reines dans la solution, on vérifie si elles sont sur la même colonne ou sur la même diagonale. Si c'est le cas, on incrémente un compteur de conflits.

L'objectif est de minimiser le nombre de conflits dans la solution.
Une solution valide pour le problème des n reines a un nombre de conflits de zéro.
Voici le pseudo code qui faire cette évaluation :

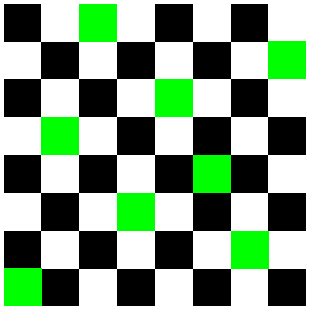
Algorithm 5 Évaluation d'une solution pour le problème des n reines

```
1:  function Évaluation(solution)
2:      conflits ← 0
3:      n ← taille(solution)
4:      for i ← 0 à n-1 do
5:          for j ← i+1 à n-1 do
6:              if solution[i] = solution[j] ou |solution[i] - solution[j]| = |i - j| then
7:                  conflits ← conflits + 1
8:              end if
9:          end for
10:     end for
11:     retourner conflits
12: end function
```

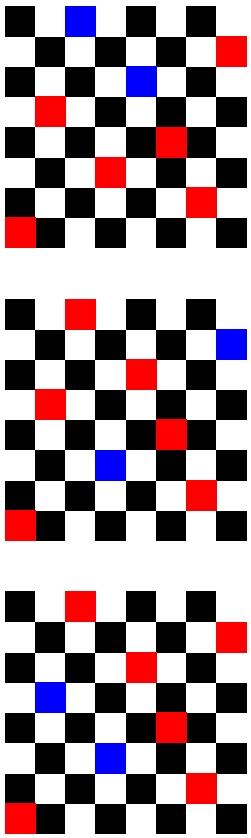
Exemple d'éxecution

Voici la solution proposé pour le problème de 8-reine :

2	7	4	1	5	3	6	0



Après l'execution de l'algorithme d'évaluation, il nous a affiché : Le nombre d'attaques pour la configuration [2,7,4,1,5,3,6,0] est **3**.



Chapitre 3

Implémentation En Java Du Problème

3.1 Algorithme DFS pour les N-reines

3.1.1 Description de l'algorithmes

L'algorithme DFS (DEPTH FIRST RESEARCH) fonctionne en explorant tous les états possibles de l'échiquier jusqu'à ce qu'il trouve une solution ou épuise toutes les options. Pour cela, il utilise une pile de liste d'entiers pour stocker l'état actuel de l'échiquier. À chaque étape, il développe tous les états possibles en ajoutant une nouvelle reine à une colonne différente si cette position est sûre, c'est-à-dire que la reine n'attaque pas les autres reines.
S'il trouve une solution, il la renvoie. Si la pile est vide et qu'aucune solution n'a été trouvée, il renvoie null

Arbre de Recherche pour DFS exemple echequier de 4x4 :

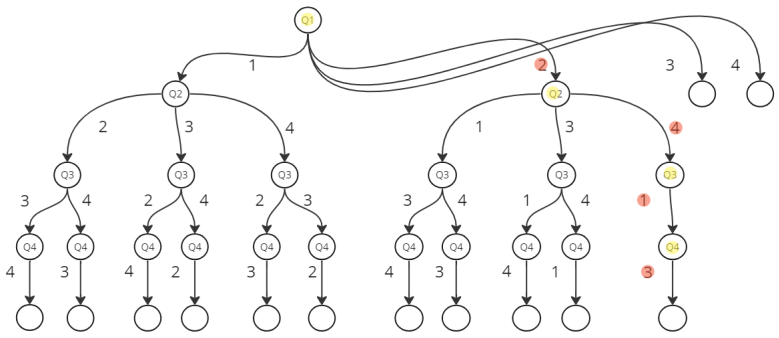


FIGURE 3.1 – Arbre de Recherche 4x4

Description des différentes classes

Le code correspond à une classe appelée "DFS" qui contient une méthode principale "main" et trois autres méthodes nommées "solveNQueens", "isAttacking" et "printSolution

La classe solveNQueens

pseudo-code

Algorithm 6 Résolution du problème des N-reines avec DFS itératif

```
1:  function DFS_Résoudre_N_Reines(n)
2:      solution ← null
3:      count[0] ← 0 // compteur des noeuds générés
4:      count[1] ← 0 // compteur des noeuds développés
5:      pile ← nouvelle pile de listes d'entiers
6:      empiler une liste vide dans la pile
7:      while la pile n'est pas vide ET solution est null do
8:          état ← dépiler une liste d'entiers de la pile
9:          incrémenter count[1] // incrémenter le compteur de noeuds développés
10:         if la taille de l'état est égale à n then
11:             solution ← état
12:         else
13:             for col allant de 0 à n-1 do
14:                 if le placement d'une reine dans la colonne col ne met pas en danger l'état then
15:                     nextState ← une copie de l'état avec la colonne col ajoutée
16:                     empiler nextState dans la pile
17:                     incrémenter count[0] // incrémenter le compteur de noeuds générés
18:                 end if
19:             end for
20:         end if
21:     end while
22:     retourner solution, count[0] et count[1]
23: end function
```

code en java

```java
// Resolution du probleme des N-reines avec DFS iteratif
public static List<Integer> solveNQueens(int n, int[] count) {
        List<Integer> solution = null;
        Stack<List<Integer>> stack = new Stack<>();
        stack.push(new ArrayList<>());
        count[0]++; // incrementer le compteur de noeuds generes
        while (!stack.isEmpty() && solution == null) {
                List<Integer> state = stack.pop();
                count[1]++; // incrementer le compteur de noeuds
                    developpes
                if (state.size() == n) {
                        solution = state;
```

```
                } else {
                    for (int col = 0; col < n; col++) {
                        if (!isAttacking(state, n-1-col)) {
                            List<Integer> nextState =
                                new ArrayList<>(state);
                            nextState.add(n-1-col);
                            stack.push(nextState);
                            count[0]++; // incrementer le
                                compteur de noeuds generes
                        }
                    }
                }
            }
            return solution;
        }
```

Explication

La méthode "solveNQueens" est responsable de la résolution du problème des N-reines en utilisant la recherche en profondeur d'abord (DFS) itératif.
Elle prend en entrée la taille de l'échiquier et un tableau d'entiers "count" qui sert à compter le nombre de nœuds générés et développés.
Elle utilise une pile pour stocker les états de l'échiquier et commence par ajouter une liste vide à la pile.
Elle continue à dépiler les états de la pile jusqu'à ce qu'elle trouve une solution ou que la pile soit vide. Pour chaque état déplilé, elle vérifie s'il s'agit d'un état final (tous les reines sont placées sur l'échiquier) ou non.
Si ce n'est pas le cas, elle génère les états suivants en ajoutant une nouvelle reine dans chaque colonne qui ne provoque pas d'attaque entre les reines déjà placées et ajoute ces nouveaux états à la pile.
Elle incrémente également le compteur de nœuds générés pour chaque nouvel état ajouté à la pile.
Si elle trouve un état final, elle renvoie cet état. Sinon, elle renvoie null.

La classe isAttacking

pseudo-code

Algorithm 7 Vérification de la validité du placement d'une nouvelle reine

```
1:  function isAttacking(state, col)
2:      row ← la taille de state
3:      for i ← 0 à row-1 do
4:          queenCol ← la colonne où se trouve la reine de l'état à la ligne i
5:          if queenCol est égal à col then
6:              retourner vrai
7:          end if
8:          if queenCol - col est égal à i - row then
9:              retourner vrai
10:         end if
11:         if queenCol - col est égal à row - i then
12:             retourner vrai
13:         end if
14:     end for
15:     retourner faux
16: end function
```

code en java

```java
// Verification de la validite du placement d'une nouvelle reine
public static boolean isAttacking(List<Integer> state, int col) {
        int row = state.size();
        for (int i = 0; i < row; i++) {
                int queenCol = state.get(i);
                if (queenCol == col) {
                        return true;
                }
                if (queenCol - col == i - row) {
                        return true;
                }
                if (queenCol - col == row - i) {
                        return true;
                }
        }
        return false;
}
```

Explication

La méthode "isAttacking" est une méthode utilitaire qui prend en entrée un état de l'échiquier (une liste d'entiers représentant les colonnes où les reines sont placées) et une colonne, et vérifie si une nouvelle reine placée dans cette colonne attaque l'une des reines déjà placées.
Elle vérifie les attaques diagonales et horizontales en comparant la différence entre les colonnes et les rangées des reines déjà placées avec celle de la nouvelle reine.

3.2 Algorithme BFS pour les N-reines

3.2.1 Description de l'algorithmes

L'algorithme BFS (Breadth-First Search ou parcours en largeur d'abord) dans ce cas consiste à explorer toutes les configurations possibles du placement de N reines sur un échiquier de taille NxN, en utilisant une structure de données de type file (queue) pour stocker les configurations à explorer.

L'algorithme commence par ajouter une configuration vide (sans reines placées) à la file, qui représente l'état initial de la recherche. Tant que la file n'est pas vide et qu'une solution n'a pas été trouvée, l'algorithme retire une configuration de la file et explore toutes les configurations possibles résultant du placement d'une nouvelle reine sur une colonne non-attaquée dans la configuration courante. Pour chaque configuration ainsi obtenue, elle est ajoutée à la file pour être explorée plus tard, et le compteur de noeuds générés est incrémenté.

Si une configuration de N reines est obtenue, elle représente une solution au problème et est renvoyée par l'algorithme. Sinon, l'algorithme continue de retirer des configurations de la file jusqu'à ce que toutes les configurations possibles soient explorées.
Le compteur de noeuds développés est incrémenté à chaque fois qu'une configuration est retirée de la file pour être explorée.

L'algorithme BFS garantit la découverte d'une solution optimale si elle existe, car il explore toutes les configurations de manière exhaustive par niveaux, c'est-à-dire qu'il explore toutes les configurations possibles avec 1 reine placée avant d'explorer celles avec 2 reines placées, et ainsi de suite jusqu'à ce qu'une configuration de N reines soit trouvée.

3.2.2 Description des différentes classes

Le code correspond à une classe appelée "BFS" qui contient une méthode principale "main" et trois autres méthodes nommées "solveNQueensBFS", "isAttacking" et "printSolution

classe solveNQueensBFS

pseudo-code

Algorithm 8 Fonction solveNQueens(n)

```
1:  function SOLVENQUEENS(n)
2:      solution ← null
3:      queue ← nouvelle Queue de Listes d'entiers
4:      queue.offer(nouvelle ArrayList d'entiers)
5:      count[0] ← 1 // incrémenter le compteur de noeuds générés
6:      while queue n'est pas vide et solution est null do
7:          state ← queue.poll()
8:          count[1] ← count[1] + 1 // incrémenter le compteur de noeuds développés
9:          if state.size() = n then
10:             solution ← state
11:         else
12:             for col de 0 à n-1 do
13:                 if non estAttacking(state, col) then
14:                     nextState ← nouvelle ArrayList d'entiers à partir de state
15:                     nextState.add(col)
16:                     queue.offer(nextState)
17:                     count[0] ← count[0] + 1 // incrémenter le compteur de noeuds générés
18:                 end if
19:             end for
20:         end if
21:     end while
22:     retourner solution
23: end function
```

code en java

```java
// Resolution du probleme des N-reines avec BFS
public static List<Integer> solveNQueens(int n, int[] count) {
        List<Integer> solution = null;
        Queue<List<Integer>> queue = new LinkedList<>();
        queue.offer(new ArrayList<>());
        count[0]++; // incrementer le compteur de noeuds generes
        while (!queue.isEmpty() && solution == null) {
                List<Integer> state = queue.poll();
                count[1]++; // incrementer le compteur de noeuds
                    developpes
                if (state.size() == n) {
                        solution = state;
```

```
                } else {
                    for (int col = 0; col < n; col++) {
                        if (!isAttacking(state, col)) {
                            List<Integer> nextState =
                                new ArrayList<>(state);
                            nextState.add(col);
                            queue.offer(nextState);
                            count[0]++; // incrementer le
                                compteur de noeuds generes
                        }
                    }
                }
            }
            return solution;
        }
```

Explication

Le code utilise une file d'attente pour stocker les états possibles, qui sont représentés sous forme de listes d'entiers.
Lorsque la file d'attente n'est pas vide, l'algorithme extrait le premier état de la file, vérifie s'il s'agit d'une solution, et sinon génère tous les états suivants possibles en ajoutant une nouvelle reine à une nouvelle colonne non menacée.
Les états suivants sont ajoutés à la fin de la file d'attente.
Le processus se répète jusqu'à ce qu'une solution soit trouvée ou que la file d'attente soit vide.

Le paramètre count est un tableau de deux entiers, qui sont utilisés pour compter le nombre de nœuds générés et développés.
count[0] est incrémenté à chaque fois qu'un nouvel état est généré et count[1] est incrémenté chaque fois qu'un état est développé, c'est-à-dire chaque fois qu'il est extrait de la file d'attente.

La fonction renvoie la première solution trouvée ou null si aucune solution n'a été trouvée.

classe isAttacking

La meme classe implémenter précédemment.

3.3 Algorithme A* pour les N-reines (heuristique 1)

3.3.1 Description de l'algorithmes

L'algorithme A* utilise une fonction heuristique pour guider la recherche et trouver la solution optimale en termes de coût.

L'heuristique utilisée ici est le nombre de conflits entre les reines, c'est-à-dire le nombre de paires de reines qui peuvent s'attaquer mutuellement.
Le code demande à l'utilisateur de saisir la taille de l'échiquier, exécute l'algorithme A* pour trouver une solution et affiche le résultat ainsi que le nombre de nœuds générés et développés, ainsi que le temps d'exécution de l'algorithme.

3.3.2 Description des différentes classes

heuristique 1

pseudo-code

Algorithm 9 Méthode heuristic(state : liste d'entiers) : entier

```
1:  function HEURISTIC(state : Liste d'entiers)
2:      n ← taille de state
3:      conflicts ← 0
4:      for i do0n-2
5:          for j doi+1n-1
6:              if state[i] = state[j] ou abs(state[i] - state[j]) = j - i then
7:                  conflicts ← conflicts + 1
8:              end if
9:          end for
10:     end for
11:     retourner conflicts
12: end function
```

code en java

```java
// Calcul de l'heuristique
public static int heuristic(List<Integer> state){
        int n = state.size();
        int conflicts = 0;
        for (int i = 0; i < n - 1; i++){
                for (int j = i + 1; j < n; j++){
                        if (state.get(i) == state.get(j) || Math.abs(
                            state.get(i) - state.get(j)) == j - i){
                                conflicts++;
                        }
                }
        }
        return conflicts;
}
```

Explication

Cette méthode calcule le nombre de conflits dans une configuration donnée de N-Queens en parcourant chaque paire de reines et en vérifiant si elles sont sur la même ligne, colonne ou diagonale. Le nombre de conflits est ensuite renvoyé en tant que valeur de la fonction.

classe A*(1)

pseudo-code

Algorithm 10 Méthode AStar(startState : liste d'entiers) : liste d'entiers

```
1:  function AStar(startState : Liste d'entiers)
2:      frontier : file de priorité de Noeud
3:      visited : ensemble de liste d'entiers
4:      n ← taille de startState
5:      start : Noeud initialisé avec startState, cost=0, heuristic=heuristic(startState) et parent=NULL
6:      Ajouter start à la frontier
7:      noeudsgénérés ← noeudsgénérés + 1
8:      while frontier n'est pas vide do
9:          node ← extraire le nœud de coût minimum de frontier
10:         noeudsdeveloppés ← noeudsdeveloppés + 1
11:         if node.heuristic = 0 then
12:             Retourner node.state
13:         end if
14:         Ajouter node.state à visited
15:         for i do0n-1
16:             for j do0n-1
17:                 if j ≠ node.state[i] then
18:                     newState ← copie de node.state
19:                     newState[i] ← j
20:                     if newState n'est pas dans visited then
21:                         child ← Noeud initialisé avec newState, cost=node.cost+1, heuristic=heuristic(newState) et parent=node
22:                         Ajouter child à la frontier
23:                         noeudsgénérés ← noeudsgénérés + 1
24:                     end if
25:
26:
27:
28:                     Retourner NULL
29:
```

code en java

```
public static List<Integer> AStar(List<Integer> startState){
        PriorityQueue<Noeud> frontier = new PriorityQueue
            <Noeud>();
        Set<List<Integer>> visited = new HashSet<List<
            Integer>>();
        int n = startState.size();
        Noeud start = new Noeud(startState, 0, heuristic(
            startState), null);
        frontier.add(start);
        noeudsgeneres++;
        while (!frontier.isEmpty()){
                Noeud node = frontier.poll();
                noeudsdeveloppes++;
                if (node.heuristic == 0){
                        return node.state;
                }
                visited.add(node.state);
                for (int i = 0; i < n; i++){
                        for (int j = 0; j < n; j++){
                                if (j != node.state.get(i)){
                                        List<Integer> newState =
                                            new ArrayList<Integer
                                            >(node.state);
                                        newState.set(i, j);
                                        if (!visited.contains(
                                            newState)){
                                                Noeud child = new
                                                    Noeud(newState,
                                                    node.cost + 1,
                                                    heuristic(
                                                    newState), node);
                                                frontier.add(child);
                                                noeudsgeneres++;
                                        }
                                }
                        }
                }
        }
        return null;
}
```

Explication

Cette méthode une file de priorité pour stocker les nœuds à explorer, en fonction de leur coût total (coût actuel + heuristique).
Elle utilise également un ensemble pour stocker les états visités et éviter les boucles infinies.
Pour chaque nœud extrait de la file de priorité, elle génère tous les états possibles en déplaçant une reine à une nouvelle position et ajoute ces états à la file de priorité si ils ne sont pas déjà visités.
L'algorithme continue jusqu'à ce qu'il trouve une solution ou que la file de priorité soit vide.
Si une solution est trouvée, l'algorithme renvoie l'état correspondant, sinon il renvoie NULL.

classe noeud

pseudo-code

Algorithm 11 Classe Noeud

1: $state$: liste d'entiers
2: $cost$: entier
3: $heuristic$: entier
4: $parent$: Noeud
5: **function** COMPARETO(other : Noeud) : entier
6: $f \leftarrow$ this.cost + this.heuristic
7: $otherF \leftarrow$ other.cost + other.heuristic
8: Retourner f - otherF
9: **end function**=0

code en java

```
private static class Noeud implements Comparable<Noeud> {
        List<Integer> state;
        int cost;
        int heuristic;
        Noeud parent;

        public Noeud(List<Integer> state, int cost, int
            heuristic, Noeud parent) {
                this.state = state;
                this.cost = cost;
                this.heuristic = heuristic;
                this.parent = parent;
        }

        public int compareTo(Noeud other) {
                int f = this.cost + this.heuristic;
```

```
                int otherF = other.cost + other.heuristic;
                return Integer.compare(f, otherF);
            }
        }
```

Explication

La classe Noeud est utilisée pour stocker l'état actuel d'une configuration de reines sur l'échiquier dans le problème des N-Reines. Elle contient les informations suivantes :

state : une liste d'entiers représentant la configuration actuelle de reines sur l'échiquier. Chaque entier correspond à la colonne où se trouve la reine dans cette ligne.

cost : le coût de l'état actuel, qui correspond au nombre de conflits entre les reines.
heuristic : la valeur heuristique de l'état actuel, qui estime le coût restant pour atteindre la solution.

parent : le noeud parent de l'état actuel dans l'arbre de recherche.

La méthode compareTo est une méthode qui compare deux noeuds et retourne un entier indiquant si l'objet actuel est plus petit, égal ou plus grand que l'objet comparé. Dans ce cas, elle calcule la valeur de la fonction d'évaluation f pour l'objet actuel et pour l'objet comparé (en utilisant la formule f = cost + heuristic) et retourne la différence entre ces deux valeurs. Si cette différence est positive, cela signifie que l'objet actuel est plus grand que l'objet comparé, s'il est négatif, cela signifie que l'objet actuel est plus petit que l'objet comparé, et s'il est zéro, cela signifie que les deux objets sont égaux en termes de coût et de valeur heuristique. Cette méthode est utilisée pour trier les noeuds dans la file de priorité de l'algorithme A* en fonction de leur coût total f.

3.4 Algorithme A* pour les N-reines (heuristique 2)

3.4.1 Description de l'algorithmes

Le programme utilise une heuristique pour guider la recherche vers la solution optimale. Il s'agit de la deuxième heuristique qui compte le nombre de cases vides sur l'échiquier qui peuvent être attaquées par les reines déjà placées. Le programme affiche la solution trouvée, le nombre de nœuds générés et développés, ainsi que le temps d'exécution de l'algorithme.

3.4.2 Description des différentes classes

heuristique 2

pseudo-code

Algorithm 12 Fonction heuristic

Require: $state$: Liste d'entiers
Ensure: $emptyCells$: Entier représentant le nombre de cases vides

1: $n \leftarrow$ taille de la liste $state$
2: $emptyCells \leftarrow 0$
3: $occupiedCols \leftarrow$ un nouvel ensemble vide
4: **for** $i \leftarrow 0$ **à** $n - 1$ **do**
5: $col \leftarrow state[i]$
6: **if** $col \notin occupiedCols$ **then**
7: Ajouter col à $occupiedCols$
8: **end if**
9: **end for**
10: $emptyCells \leftarrow n - taille(occupiedCols)$
11: **Retourner** $emptyCells$

code en java

```
public static int heuristic(List<Integer> state) {
        int n = state.size();
        int emptyCells = 0;
        Set<Integer> occupiedCols = new HashSet<Integer>();
        for (int i = 0; i < n; i++) {
                int col = state.get(i);
                if (!occupiedCols.contains(col)) {
                        occupiedCols.add(col);
                }
        }
        emptyCells = n - occupiedCols.size();
        return emptyCells;
}
```

Explication

L'heuristic prend en entrée une liste d'entiers state représentant une disposition des reines sur un échiquier et qui retourne le nombre de cases vides restantes sur l'échiquier.

La fonction commence par initialiser deux variables : n qui contient la taille de la liste state et emptyCells qui représente le nombre de cases vides.

Ensuite, on crée un ensemble occupiedCols qui va contenir les colonnes déjà occupées par les reines. On parcourt ensuite la liste state à l'aide d'une boucle for et à chaque itération, on récupère la colonne de la reine correspondante dans la variable col. Si la colonne n'est pas encore présente dans l'ensemble occupiedCols, alors on l'ajoute à cet ensemble.
Après la boucle, le nombre de cases vides est égal à la différence entre le nombre total de colonnes n et le nombre de colonnes présentes dans l'ensemble occupiedCols.

Enfin, la fonction retourne le nombre de cases vides emptyCells.

Le but de cette heuristique est de favoriser les états qui ont un grand nombre de cases vides car cela laisse plus de possibilités pour placer les reines restantes et donc une plus grande chance de trouver une solution.

classe A*(2)

pseudo-code

Algorithm 13 Méthode A*(startState : liste d'entiers) : liste d'entiers

```
1:  function A*(startState : Liste d'entiers)
2:      frontier : file de priorité de Noeud
3:      visited : ensemble de liste d'entiers
4:      n ← taille de startState
5:      start : Noeud initialisé avec startState, cost=0, heuristic=heuristic(startState) et parent=NULL
6:      Ajouter start à la frontier
7:      noeudsgénérés ← noeudsgénérés + 1
8:      while frontier n'est pas vide do
9:          node ← extraire le nœud de coût minimum de frontier
10:         noeudsdeveloppés ← noeudsdeveloppés + 1
11:         if node.heuristic = 0 then
12:             Retourner node.state
13:         end if
14:         Ajouter node.state à visited
15:         for i do0n-1
16:             for j do0n-1
17:                 if j ≠ node.state[i] then
18:                     newState ← copie de node.state
19:                     newState[i] ← j
20:                     if newState n'est pas dans visited then
21:                         child ← Noeud initialisé avec newState, cost=node.cost+1, heuristic=heuristic(newState) et parent=node
22:                         Ajouter child à la frontier
23:                         noeudsgénérés ← noeudsgénérés + 1
24:                     end if
25:
26:
27:
28:                     Retourner NULL
29:
```

code en java

```
public static List<Integer> AStar(List<Integer> startState){
    PriorityQueue<Noeud> frontier = new PriorityQueue
        <Noeud>();
    Set<List<Integer>> visited = new HashSet<List<
        Integer>>();
```

```
                int n = startState.size();
                Noeud start = new Noeud(startState, 0, heuristic(
                    startState), null);
                frontier.add(start);
                noeudsgeneres++;
                while (!frontier.isEmpty()) {
                        Noeud node = frontier.poll();
                        noeudsdeveloppes++;
                        if (node.heuristic == 0) {
                                return node.state;
                        }
                        visited.add(node.state);
                        for (int i = 0; i < n; i++) {
                                for (int j = 0; j < n; j++) {
                                        if (j != node.state.get(i)) {
                                                List<Integer> newState =
                                                    new ArrayList<Integer
                                                    >(node.state);
                                                newState.set(i, j);
                                                if (!visited.contains(
                                                    newState)) {
                                                        Noeud child = new
                                                            Noeud(newState,
                                                            node.cost + 1,
                                                            heuristic(
                                                            newState), node);
                                                        frontier.add(child);
                                                        noeudsgeneres++;
                                                }
                                        }
                                }
                        }
                }
                return null;
        }
```

Explication

Cette méthode une file de priorité pour stocker les nœuds à explorer, en fonction de leur coût total (coût actuel + heuristique).
Elle utilise également un ensemble pour stocker les états visités et éviter les boucles infinies.
Pour chaque nœud extrait de la file de priorité, elle génère tous les états possibles en déplaçant une reine à une nouvelle position et ajoute ces états à la file de priorité si ils ne sont pas déjà visités.
L'algorithme continue jusqu'à ce qu'il trouve une solution ou que la file de priorité soit vide.

Si une solution est trouvée, l'algorithme renvoie l'état correspondant, sinon il renvoie NULL.

classe noeud

La meme classe implémenter dans l'algorithme précédent.

Chapitre 4

Expérimentations

4.1 Environnement expérimental

	Marque	Systeme d'exploitation	Processeur
Sarah	DELL	64 B,Processeur x64.Windows 10.	Intel(R)I5-7 CPU @ 2.60GHz 2.70 GHz
Amel	DELL	64 B,Processeur x64.Windows 10.	Intel(R) Core(TM) i5-4310U CPU @ 2.00GHz 2.60 GHz

4.2 Interface graphique

Voici quelques captures d'écran décrivant notre interface graphique pour le problème de N-reines :

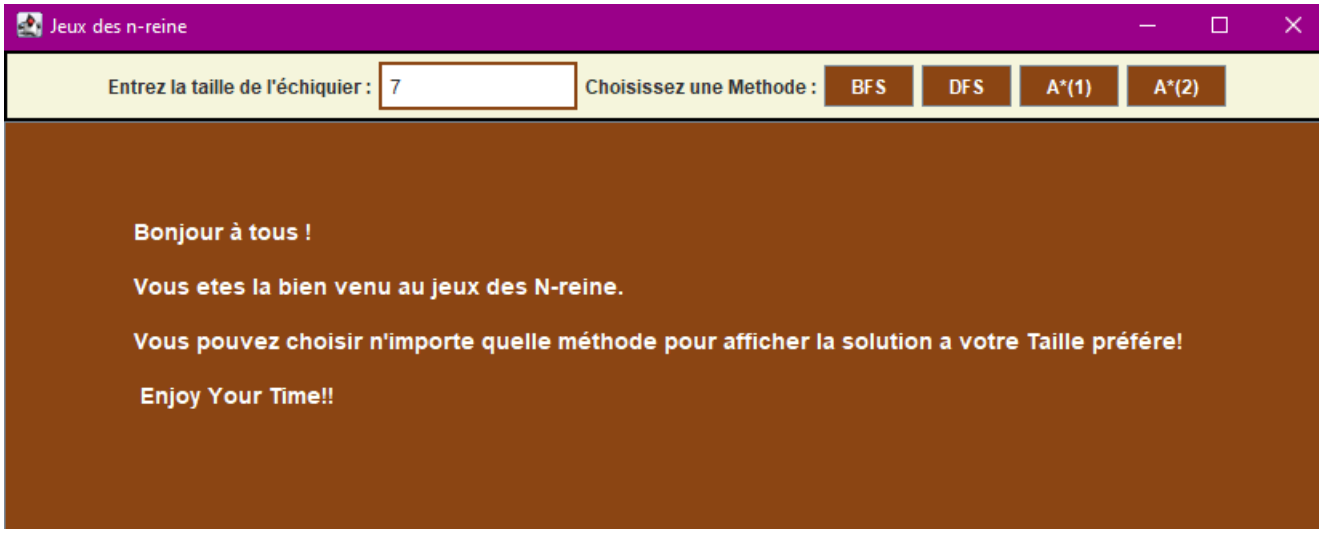


FIGURE 4.1 – Fenetre 1.1

Cette fenêtre contient des composants graphiques tels que des boutons, des zones de texte, des panneaux, etc. Les utilisateurs peuvent interagir avec ces composants pour choisir la taille de l'échiquier avec la zone de texte et la méthode de résolution du problème des n-reines.
Les quatre boutons sont nommés "BFS", "DFS", "A*(1)" et "A*(2)", qui représentent différentes méthodes pour résoudre le problème des n-reines avec la taille n choisit.

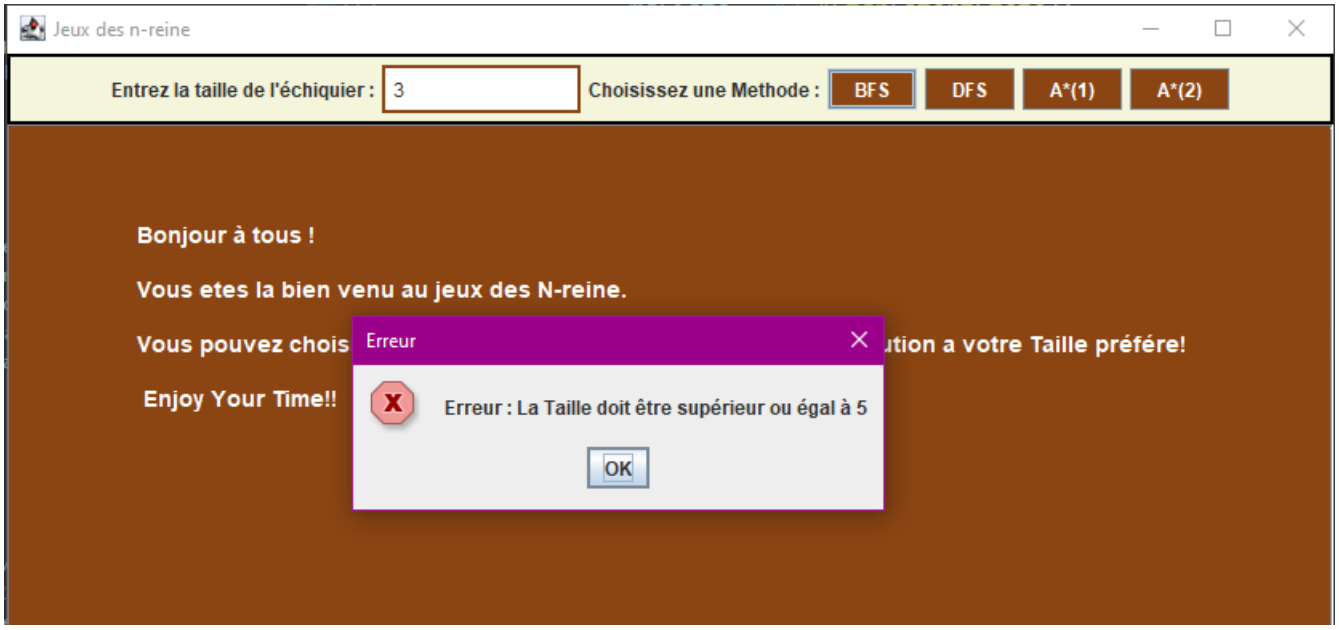


FIGURE 4.2 – Fenetre 1.2

Si la valeur de la variable "n" est inférieure à 5, une boîte de dialogue d'erreur affiche un message à l'utilisateur.

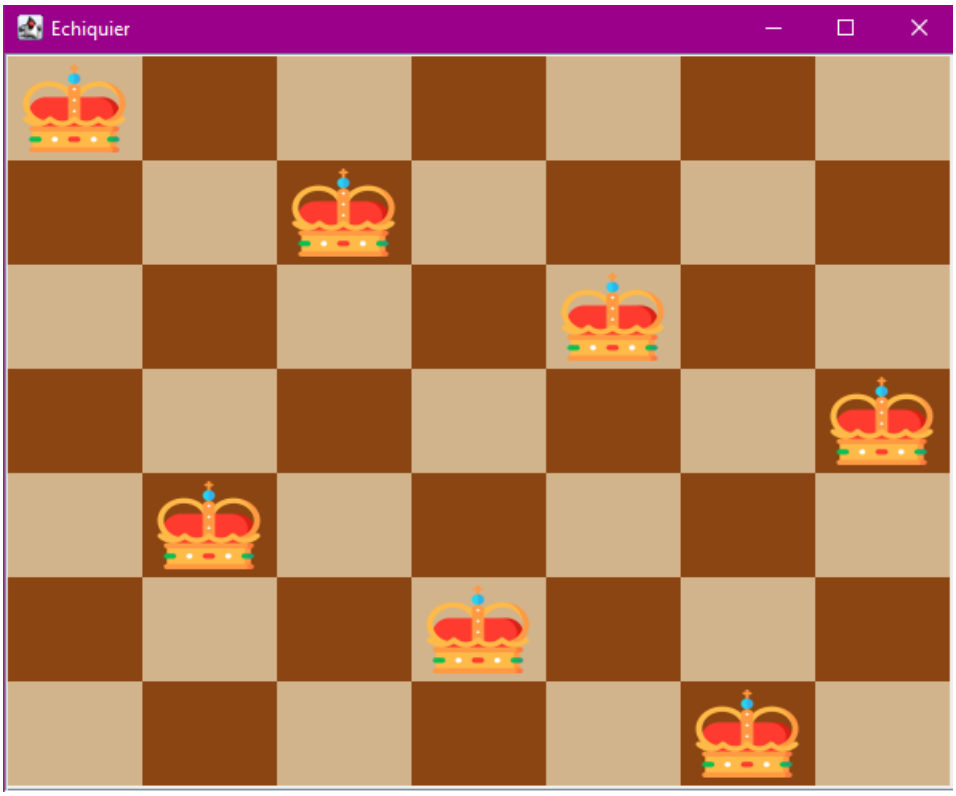


FIGURE 4.3 – Fenetre 2

Une fenêtre graphique contenant un échiquier, avec des cases colorées en marron ou beige selon leur position sur l'échiquier. Si une case contient une reine, une image de couronne est ajoutée à cette case.

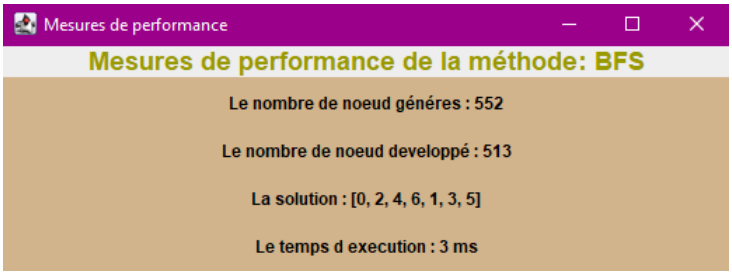


FIGURE 4.4 – Fenetre 3

Cette fenetre affiche des informations sur les mesures de performance pour la méthode choisit soit DFS, BFS, A*(1) ou A*(2).

4.3 Résultats expérimentaux

4.3.1 DFS pour N-reines

Taille	nœuds générés	nœuds développés	T.Execution(ms)	solution trouvée
5	12	6	1	[0, 2, 4, 1, 3]
7	23	10	1	[0, 2, 4, 6, 1, 3, 5]
9	61	42	1	[0, 2, 5, 7, 1, 3, 8, 6, 4]
11	84	53	2	[0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
13	155	112	2	[0, 2, 4, 1, 8, 11, 9, 12, 3, 5, 7, 10, 6]
15	1415	1360	9	[0, 2, 4, 1, 9, 11, 13, 3, 12, 8, 5, 14, 6, 10, 7]

TABLE 4.1 – résultats numériques DFS

4.3.2 BFS pour N-reines

heightTaille	nœuds générés	nœuds développés	T.Execution(ms)	solution trouvée
5	54	45	0	[0, 2, 4, 1, 3]
7	552	513	3	[0, 2, 4, 6, 1, 3, 5]
9	8394	8043	15	[0, 2, 5, 7, 1, 3, 8, 6, 4]
11	166926	164247	103	[0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
13	4674890	4601179	2238	[0, 2, 4, 1, 8, 11, 9, 12, 3, 5, 7, 10, 6]
15	Java heap space	Java heap space	Java heap space	Java heap space

TABLE 4.2 – résultats numériques BFS

4.3.3 A* pour N-reines avec H1

T	nds générés	nds développés	Temps(ms)	solution trouvée
5	97	6	8	[1, 4, 2, 0, 3]
7	491	13	7	[3, 6, 2, 5, 1, 4, 0]
9	851	13	9	[3, 8, 4, 7, 0, 2, 5, 1, 6]
11	1417	14	10	[5, 10, 4, 9, 3, 8, 2, 7, 1, 6, 0]
13	34554	228	50	[4, 12, 5, 11, 2, 6, 3, 9, 0, 8, 10, 1, 7]
15	8963	44	37	[1, 10, 5, 13, 6, 12, 0, 11, 14, 4, 9, 3, 8, 2, 7]
17	455134	1348	451	[9, 7, 5, 17, 2, 16, 6, 8, 3, 14, 4, 13, 1, 12, 18, 11, 0, 10, 15]

TABLE 4.3 – résultats numériques A*(1)

4.3.4 A* pour N-reines avec H2

T	nds générés	nds développés	Temps(ms)	solution trouvée
5	822	47	18	[2, 0, 1, 4, 3]
7	5475	141	26	[2, 4, 1, 3, 5, 0, 6]
9	147220	2194	145	[4, 7, 5, 2, 1, 6, 0, 3, 8]
11	1072169	10349	666	[2, 8, 1, 7, 9, 3, 4, 0, 5, 10, 6]
13	4230155	28575	2819	[3, 5, 8, 7, 11, 2, 6, 10, 1, 9, 0, 4, 12]
15	Java heap space	Java heap space	Java heap space	Java heap space

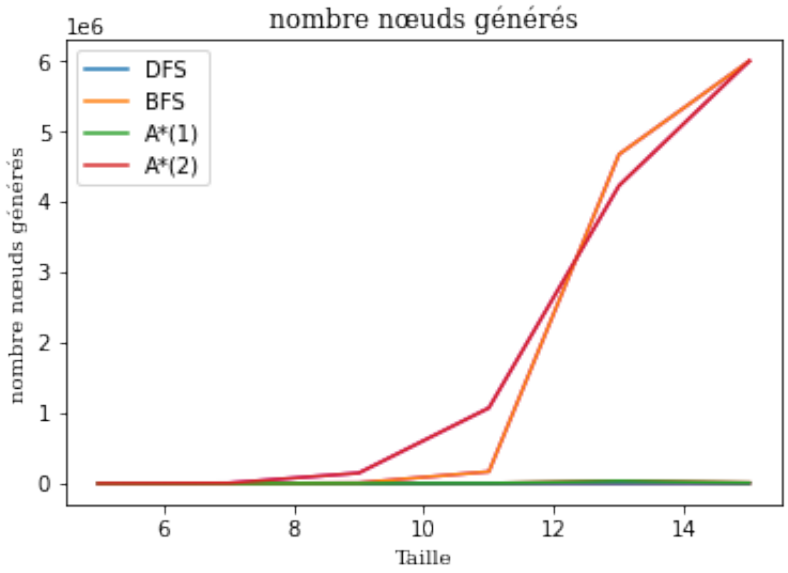
TABLE 4.4 – résultats numériques A*(2)

4.4 Etude comparative des 4 méthodes

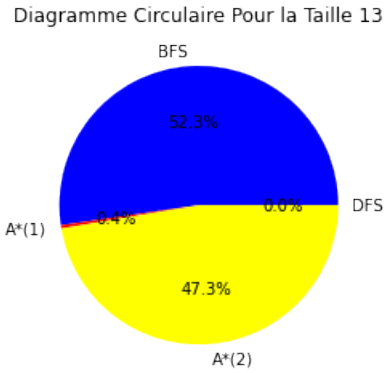
4.4.1 nombre nœuds générés

heightN	DFS	BFS	A*(1)	A*(2)
5	12	54	97	822
7	23	552	491	5475
9	61	8394	851	147220
11	84	166926	1417	1072169
13	155	4674890	34554	4230155
15	1415	Java heap space	8963	Java heap space

TABLE 4.5 – Comparaison entre le nombre de nœuds générés



(a) Graphe 1.1



(b) Graphe 1.2

FIGURE 4.5 – Graphes comparative des nœuds générés

Analyse

Selon le graphique 1.1 de la Fig. 4.5, On remarque que l'évolution du nombre de nœuds créés par les algorithmes BFS et A*(2) est très rapide a partir de taille 11, mais pour les deux autres algorithmes (DFS et A*(1)), le nombre de nœuds créés est proche de 0. le diagramme circulaire confirme ça, comme le montre le graphique 1.2 de la Fig. 4.5 on remarque que l'algorithme BFS prend la moitié du pourcentage de nœuds générés suivi de l'algorithme A*(2), sauf que les algorithmes DFS et A*(1), le nombre denœuds générés est négligeable et est proche de 0%.

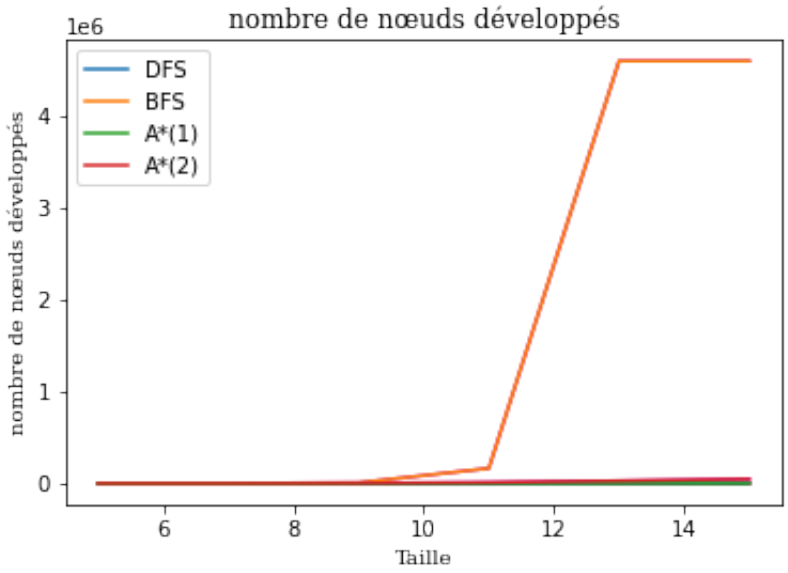
Conclusion

En conclusion, l'analyse des graphique montre que les algorithmes DFS et A*(1) sont beaucoup plus efficaces que les algorithmes BFS et A*(2) dans la création de nœuds. Ces résultats peuvent avoir des implications importantes pour le choix de l'algorithme à utiliser dans certaines applications où la création de nœuds est un facteur critique dans la performance de l'algorithme.

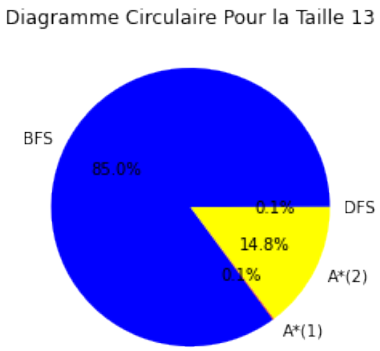
4.4.2 nombre de nœuds développés

heightN	DFS	BFS	A*(1)	A*(2)
5	6	45	6	47
7	10	513	13	141
9	42	8043	13	2194
11	53	164247	14	10349
13	112	4601179	228	28575
15	1360	Java heap space	44	Java heap space

TABLE 4.6 – Comparaison entre le nombre de nœuds développés



(a) Graphe 1.1



(b) Graphe 1.2

FIGURE 4.6 – Graphes comparative des nœuds développés

Analyse

D'après le diagramme circulaire 1.2 de la Fig. 4.6, l'algorithme BFS contient la plus grande partie du nombre de nœuds développés (85%) après c'est l'algorithme A*(2) avec 15%, sinon pour DFS et A*(1) avec un pourcentage de 0,1%. Et on remarque à travers le graphe 1.1 que les courbes de DFS, A*1 et A*2 sont très proches de zéro. sauf que BFS a une explosion combinatoire à partir de la taille 10.

Conclusion

Ces résultats peuvent être importants pour les applications où le nombre de nœuds développés est critique pour la performance de l'algorithme, car ils suggèrent que l'algorithme BFS est le plus efficace pour résoudre le problème et même pour A*(1)

4.4.3 temps d'exécution

heightN	DFS	BFS	A*(1)	A*(2)
5	1	0	8	18
7	1	3	7	28
9	1	15	9	145
11	2	103	10	666
13	2	2238	50	2819
15	9	Java heap space	37	Java heap space

TABLE 4.7 – Comparaison entre les temps d'exécution des 4 méthodes

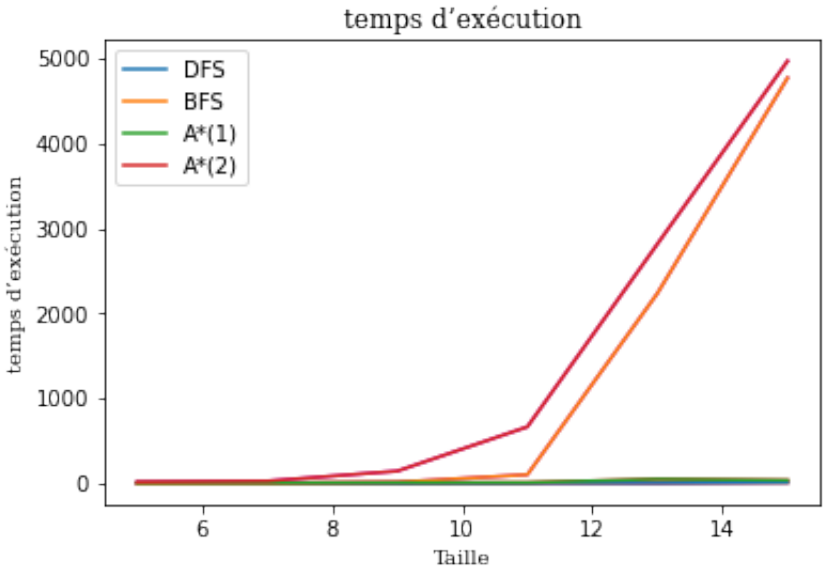


FIGURE 4.7 – Temps Execution Des 4 méthodes

Analyse

D'après la Fig. 4.7, le temps d'exécution augmente très rapidement que ce soit pour l'algorithme BFS ou A*2, il s'agit d'une explosion combinatoire à partir de la taille 11. mais pour les courbes DFS et A*1, la progression est stable et tend vers 0 en terme de temps d'exécution.

Conclusion

Ces résultats suggèrent que, pour les problèmes de grande taille, l'utilisation des algorithmes BFS et A*(2) peut entraîner des temps d'exécution excessifs(une complexité exponentielle), tandis que l'utilisation des algorithmes DFS et A*(1) peut être plus appropriée en termes de temps d'exécution.

4.4.4 solution trouvée

heightN	DFS	BFS
5	[0, 2, 4, 1, 3]	[0, 2, 4, 1, 3]
7	[0, 2, 4, 6, 1, 3, 5]	[0, 2, 4, 6, 1, 3, 5]
9	[0, 2, 5, 7, 1, 3, 8, 6, 4]	[0, 2, 5, 7, 1, 3, 8, 6, 4]
11	[0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9]	[0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
13	[0, 2, 4, 1, 8, 11, 9, 12, 3, 5, 7, 10, 6]	[0, 2, 4, 1, 8, 11, 9, 12, 3, 5, 7, 10, 6]
15	[0, 2, 4, 1, 9, 11, 13, 3, 12, 8, 5, 14, 6, 10, 7]	Java heap space

TABLE 4.8 – Comparaison entre les solutions trouvées 1

heightN	A*(1)	A*(2)
5	[1, 4, 2, 0, 3]	[2, 0, 1, 4, 3]
7	[3, 6, 2, 5, 1, 4, 0]	[2, 4, 1, 3, 5, 0, 6]
9	[3, 8, 4, 7, 0, 2, 5, 1, 6]	[4, 7, 5, 2, 1, 6, 0, 3, 8]
11	[5, 10, 4, 9, 3, 8, 2, 7, 1, 6, 0]	[2, 8, 1, 7, 9, 3, 4, 0, 5, 10, 6]
13	[4, 12, 5, 11, 2, 6, 3, 9, 0, 8, 10, 1, 7]	[3, 5, 8, 7, 11, 2, 6, 10, 1, 9, 0, 4, 12]
15	[1, 10, 5, 13, 6, 12, 0, 11, 14, 4, 9, 3, 8, 2, 7]	Java heap space

Table 4.9 – Comparaison entre les solutions trouvées 2

Analyse

Pour les solutions trouvées par les 4 algorithmes, il ressort clairement des tableaux 4.8 et 4.9 que DFS et BFS trouvent la même solution, mais A*1 et A*2 ils trouvent des solutions différentes. Notons que pour une taille donnée nous aurons 3 solutions différentes, celle de (DFS et BFS), l'une de A*1 et l'autre de A*2.

Conclusion

Les différentes solutions trouvées peuvent avoir des avantages et des inconvénients en termes de qualité et de temps de calcul. Il est donc important de considérer toutes les solutions possibles avant de choisir l'algorithme approprié pour un problème donné.

4.5 Conclusion de l'étude comparative

Pour la résolution du problème des n-reines, DFS et BFS utilisent une approche de recherche aveugle, qui consiste à explorer toutes les configurations possibles jusqu'à trouver une solution valide, tandis qu'A* utilise une approche de recherche heuristique qui utilise une fonction d'évaluation pour guider la recherche vers les solutions les plus prometteuses. Voici une comparaison des avantages et des inconvénients de chaque algorithme pour résoudre ce problème :

- DFS : il est simple à implémenter et nécessite peu d'espace mémoire, mais peut être lent pour les problèmes de grande taille, car il explore toutes les branches de l'arbre de recherche.
- BFS : il garantit la recherche de la solution optimale en termes de nombre de mouvements nécessaires. Cependant, il nécessite beaucoup d'espace mémoire et peut être lent pour les problèmes de grande taille.
- A* : A* peut être plus rapide et plus efficace que DFS et BFS. Cependant, la qualité de la solution dépend de la fonction heuristique choisie et de l'implémentation de l'algorithme. (Dans notre cas l'heuristique 1 est admissible mais la deuxieme non).

En général, DFS est plus adapté pour les petits problèmes de n reines, tandis que BFS et A* sont plus adaptés pour les problèmes plus grands. BFS est recommandé pour garantir la recherche de la solution optimale, tandis que A* est recommandé pour sa rapidité et son efficacité.

Remarque Dans notre étude nos résultats montrent que DFS est meilleur que A*(1), c'est parce que nous avons testé des tailles qui ne sont pas assez grandes grâce à notre environnement expérimental, mais il est clair que si on augmente la taille du problème on trouve que A*(1) est préférable. Même pour A*(2), ses résultats ne sont pas importants car la fonction heuristique n'est pas admissible, il faut donc plus de temps pour la rechercher, et plus d'espace mémoire

Conclusion et Perspectives

D'après les résultats présentés précédemment, le problème des n reines a été résolu en utilisant quatre algorithmes différents : DFS, BFS, A* avec heuristique 1, et A* avec heuristique 2.
Les performances de chaque algorithme ont été évaluées et comparées.
En conclusion, il est important de choisir l'algorithme approprié en fonction de la complexité du problème à résoudre et des exigences en matière de performances. Le rapport a démontré que, dans le cas du problème des n reines, DFS est l'algorithme le plus efficace, suivi de A* avec heuristique 1, BFS et A* avec heuristique 2.
Il est possible d'améliorer les résultats obtenus pour la résolution du problème des n reines en utilisant des algorithmes de recherche métaheuristique tel que l'algorithme génétique, local et taboue dans la suite du projet.

Réferences :

Cours Métaheuristiques, Mme H.DRIAS (USTHB)
Cours Métaheuristiques, Mme H.MESSAOUDI (USTHB)
le-probleme-des-8-reines [**interstices.info**].
N queens [**medium.com**].
chatgbt [**chatgbt.openai**].
n queens problem[**github**].

Annexe A

code source des interfaces graphiques

A.1 principal.java

```
package ProjetP1;
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import javax.swing.border.Border;
import javax.swing.border.LineBorder;
import ProjetP1.secondaire;
import ProjetP1.BFS;
import ProjetP1.DFS;
import ProjetP1.A;
import ProjetP1.A2;

public class principal extends JFrame implements ActionListener{
        private static int n;
        private static int nbrNoeudsGeneres;
        private static int nbrNoeudsDevelopes;
        private static long startTime,endTime,time;
        private static int[] tab = new int[n];
        private static int[] count = new int[2];
        private static List<Integer> maListe = new ArrayList<Integer
            >();
        private static String nomMethod;
        private JTextField textField;
        private JTextArea textArea;
        private JButton btnChoix1, btnChoix2, btnChoix3, btnChoix4;
```

```
    private JTextField textField2;
    private JTextArea textArea2;

    public principal() {
        super("Jeux des n-reine");
        // creation des composants de l'interface graphique
        textField = new JTextField(10);
        textArea = new JTextArea(20, 20);
        textArea.setEditable(false);
        // Ajout des marges au JTextArea
        Insets insets = new Insets(60, 80, 20, 20);
        textArea.setMargin(insets);
        textArea.setBackground(new Color(139, 69, 19));
        Font font = new Font("Arial", Font.BOLD, 14);
        textArea.setFont(font);
        textArea.setForeground(Color.WHITE);
        String texte = "Bonjour a tous !\n\nVous etes la bien venu au jeux
            des N-reine.\n\nVous pouvez choisir n'importe quelle methode
            pour afficher la solution a votre Taille prefere!\n\n Enjoy Your Time
            !!";
        // Afficher le texte dans le JTextArea
        textArea.setText(texte);
        btnChoix1 = new JButton("BFS");
        btnChoix1.addActionListener(this);
        btnChoix1.setBackground(new Color(139, 69, 19));
        btnChoix1.setForeground(Color.WHITE);
        btnChoix2 = new JButton("DFS");
        btnChoix2.addActionListener(this);
        btnChoix2.setBackground(new Color(139, 69, 19));
        btnChoix2.setForeground(Color.WHITE);
        btnChoix3 = new JButton("A*(1)");
        btnChoix3.addActionListener(this);
        btnChoix3.setBackground(new Color(139, 69, 19));
        btnChoix3.setForeground(Color.WHITE);
        btnChoix4 = new JButton("A*(2)");
        btnChoix4.addActionListener(this);
        btnChoix4.setBackground(new Color(139, 69, 19));
        btnChoix4.setForeground(Color.WHITE);
        // ajout des composants a la fenetre
        JPanel panel = new JPanel();
        panel.add(new JLabel("Entrez la taille de l'echiquier :"));
        Border line = BorderFactory.createLineBorder(new
            Color(139, 69, 19), 2);
        Border margin = BorderFactory.createEmptyBorder(5,
```

```
                5, 5, 5);
        Border compound = BorderFactory.
            createCompoundBorder(line, margin);
        textField.setBorder(compound);
        panel.add(textField);
        panel.setBackground(new Color(245, 245, 220));
        panel.setBorder(BorderFactory.createLineBorder(
            Color.BLACK, 2));
        add(panel, BorderLayout.NORTH);
        add(new JScrollPane(textArea), BorderLayout.CENTER)
            ;
        panel.add(new JLabel("Choisissez une Methode :"));
        panel.add(btnChoix1);
        panel.add(btnChoix2);
        panel.add(btnChoix3);
        panel.add(btnChoix4);
        // configuration de la fenetre
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
}
//Appel au fonctions
@Override
public void actionPerformed(ActionEvent e) {
        //BFS
        if (e.getSource() == btnChoix1) {
                // recuperation de la taille de la matrice
                n = Integer.parseInt(textField.getText());
                if (n < 5) {
                        JOptionPane.showMessageDialog(null, "
                            Erreur : La Taille doit etre superieur ou egal a 5", "
                            Erreur", JOptionPane.ERROR_MESSAGE);
                }else {
                        startTime = System.currentTimeMillis
                            ();
                        maListe = BFS.solveNQueens(n, count);
                        endTime = System.currentTimeMillis();
                        time = endTime - startTime;
                        nomMethod="BFS";
                        secondaire maFenetre = new secondaire
                            ();
                        maFenetre.setVisible(true);
```

```
                    JolieInterface maFenetre2 = new
                        JolieInterface();
                    maFenetre2.setVisible(true);
                    dispose();
                }
            }
            //DFS
            if (e.getSource() == btnChoix2) {
                // recuperation de la taille de la matrice
                n = Integer.parseInt(textField.getText());
                if (n < 5) {
                    JOptionPane.showMessageDialog(null, "
                        Erreur : La Taille doit etre superieur ou egal a 5", "
                        Erreur", JOptionPane.ERROR_MESSAGE);
                }else {
                    ArrayList<Integer> table = new
                        ArrayList<Integer>();
                    startTime = System.currentTimeMillis
                        ();
                    maListe = DFS.solveNQueens(n, count);
                    endTime = System.currentTimeMillis();
                    time = endTime - startTime;
                    //tab = maListe;
                    nomMethod="DFS";
                    secondaire maFenetre = new secondaire
                        ();
                    maFenetre.setVisible(true);

                    JolieInterface maFenetre2 = new
                        JolieInterface();
                    maFenetre.setVisible(true);
                    dispose();
                }
            }
            //HEUR1
            if (e.getSource() == btnChoix3) {
                // recuperation de la taille de la matrice
                n = Integer.parseInt(textField.getText());
                if (n < 5) {
                    JOptionPane.showMessageDialog(null, "
                        Erreur : La Taille doit etre superieur ou egal a 5", "
                        Erreur", JOptionPane.ERROR_MESSAGE);
                }else {
                    List<Integer> startState = new
```

```
                    ArrayList<Integer>();
                for (int i = 0; i < n; i++) {
                        startState.add(0);
                }
                startTime = System.currentTimeMillis
                    ();
                maListe = A.AStar(startState);
                endTime = System.currentTimeMillis();
                time = endTime - startTime;
                nbrNoeudsGeneres = A.noeudsgeneres;
                count[0]=nbrNoeudsGeneres;
                nbrNoeudsDevelopes = A.
                    noeudsdeveloppes;
                count[1]=nbrNoeudsDevelopes;
                /*
                for (int element : tab) {
                        maListe.add(element);
                }*/
                nomMethod="Heuristique 1";
                secondaire maFenetre = new secondaire
                    ();
                maFenetre.setVisible(true);

                JolieInterface maFenetre2 = new
                    JolieInterface();
                maFenetre.setVisible(true);
                dispose();
        }
}
//HEUR2
if (e.getSource() == btnChoix4) {
        // recuperation de la taille de la matrice
        n = Integer.parseInt(textField.getText());
        if (n < 5) {
                JOptionPane.showMessageDialog(null, "
                    Erreur : La Taille doit etre superieur ou egal a 5", "
                    Erreur", JOptionPane.ERROR_MESSAGE);
        }else {
                List<Integer> startState = new
                    ArrayList<Integer>();
                for (int i = 0; i < n; i++) {
                        startState.add(0);
                }
                startTime = System.currentTimeMillis
```

```
                                    ();
                                maListe = A2.AStar(startState);
                                endTime = System.currentTimeMillis();
                                time = endTime - startTime;
                                nbrNoeudsGeneres = A.noeudsgeneres;
                                count[0]=nbrNoeudsGeneres;
                                nbrNoeudsDevelopes = A.
                                    noeudsdeveloppes;
                                count[1]=nbrNoeudsDevelopes;
                                /*
                                for (int element : tab) {
                                        maListe.add(element);
                                }*/
                                nomMethod="Heuristique 2";
                                secondaire maFenetre = new secondaire
                                    ();
                                maFenetre.setVisible(true);

                                JolieInterface maFenetre2 = new
                                    JolieInterface();
                                maFenetre.setVisible(true);
                                dispose();
                        }
                }
        }
        public static void main(String[] args) {
                new principal();
        }
        public static List<Integer> getList() {
                return maListe;
        }
        public static String getNom() {
                return nomMethod;
        }
        public static int getNbGen() {
                return count[0];
        }
        public static int getNbDev() {
                return count[1];
        }
        public static int[] getSol() {
                return tab;
        }
        public static long getTime() {
```

```
                return time;
            }
        }
```

A.2 secondaire.java

```
package ProjetP1;
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import javax.swing.border.Border;
import javax.swing.border.LineBorder;
import ProjetP1.principal;

public class secondaire extends JFrame {
    //recuperer les valeurs envoyees par l'autre fenetre
    private static List<Integer> maList = principal.getList();
    // Dimensions de l'echiquier
    private static final int DIMENSION = maList.size();
    int[] tableau = maList.stream().mapToInt(Integer::intValue).
        toArray();
    public secondaire() {
        super("Echiquier");
        // Creation du plateau
        JPanel plateau = new JPanel(new GridLayout(
            DIMENSION, DIMENSION));
        // Parcours des cases
        for (int ligne = 0; ligne < DIMENSION; ligne++) {
            for (int colonne = 0; colonne < DIMENSION; colonne
                ++) {
                // Creation de la case
                JPanel caseEchiquier = new JPanel();
                // Coloration de la case en noir ou blanc
                if ((ligne + colonne) % 2 == 0) {
                    caseEchiquier.setBackground(
                        new Color(210, 180, 140));
                } else {
                    caseEchiquier.setBackground(
                        new Color(139, 69, 19));
                }
```

```
                                        // Ajout de la case sur le plateau
                                        plateau.add(caseEchiquier);
                                        // Si la case contient une reine, ajout de l'image de la
                                            reine
                                        if (tableau[ligne] == colonne) {
                                                ImageIcon iconeReine = new
                                                    ImageIcon(getClass().
                                                    getResource("img/../../crown(1).png"
                                                    ));
                                                Image imageReine = iconeReine.
                                                    getImage();
                                                Image imageRedimensionnee =
                                                    imageReine.
                                                    getScaledInstance(65, 55,
                                                    Image.SCALE_SMOOTH);
                                                ImageIcon
                                                    iconeReineRedimensionnee =
                                                    new ImageIcon(
                                                    imageRedimensionnee);
                                                JLabel labelReine = new JLabel(
                                                    iconeReineRedimensionnee);
                                                caseEchiquier.add(labelReine);
                                        }
                                }
                        }
                        // Creation du JScrollPane contenant le plateau
                        JScrollPane scrollPane = new JScrollPane(plateau);
                        // Ajout du JScrollPane a la fenetre
                        add(scrollPane);
                        // Configuration de la fenetre
                        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        setSize(500, 500);
                        setVisible(true);
                }
                public static void main(String[] args) {
                        new secondaire();
                }
        }
```

A.3 JolieInterface.java

```
        package ProjetP1;
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import ProjetP1.principal;

public class JolieInterface extends JFrame {
        private static String nomMethod = principal.getNom();
        private static int nbrNoeudsGeneres = principal.getNbGen();
        private static int nbrNoeudsDevelopes = principal.getNbDev();
        private static List<Integer> sol = principal.getList();
        private static long time = principal.getTime();
        private JButton button;
        public JolieInterface() {
                super("Mesures de performance");
                // Creation du JLabel pour le titre
                JLabel labelTitre = new JLabel("Mesures de performance de
                    la methode: " + nomMethod);
                labelTitre.setForeground(new Color(153,153,0));
                // Creation des labels pour les informations
                JLabel labelInfo1 = new JLabel("Le nombre de noeud
                    generes : " + nbrNoeudsGeneres);
                JLabel labelInfo2 = new JLabel("Le nombre de noeud
                    developpe : " + nbrNoeudsDevelopes);
                JLabel labelInfo3 = new JLabel("La solution : " + sol);
                JLabel labelInfo4 = new JLabel("Le temps d execution : " +
                    time + " ms");
                // Creation du JPanel contenant les labels d'information
                JPanel panelInfo = new JPanel(new GridLayout(4, 1));
                panelInfo.add(labelInfo1);
                panelInfo.add(labelInfo2);
                panelInfo.add(labelInfo3);
                panelInfo.add(labelInfo4);
                // Centrer les labels d'information
                labelInfo1.setHorizontalAlignment(JLabel.CENTER);
                labelInfo2.setHorizontalAlignment(JLabel.CENTER);
                labelInfo3.setHorizontalAlignment(JLabel.CENTER);
                labelInfo4.setHorizontalAlignment(JLabel.CENTER);
                // Ajouter un peu de coloration aux labels d'information
                labelInfo1.setForeground(Color.black);
                labelInfo2.setForeground(Color.black);
```

```
                labelInfo3.setForeground(Color.black);
                labelInfo4.setForeground(Color.black);
                panelInfo.setBackground(Color.WHITE);
                // Ajout du JPanel contenant les labels d'information a la region centre
                    de la fenetre
                add(panelInfo, BorderLayout.CENTER);
                // Creation du JPanel contenant le titre et le bouton
                JPanel panelTitre = new JPanel(new BorderLayout());
                panelTitre.add(labelTitre, BorderLayout.NORTH);
                // panelTitre.add(panelButton, BorderLayout.SOUTH); // Ajout du
                    panelButton en bas du panelTitre
                labelTitre.setFont(new Font("Arial", Font.BOLD, 18));
                labelTitre.setHorizontalAlignment(JLabel.CENTER);
                // Ajout du JPanel contenant le titre a la region nord de la fenetre
                add(panelTitre, BorderLayout.NORTH);
                // Configuration de la fenetre
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                setSize(300, 200);
                setLocationRelativeTo(null); // Centrer la fenetre sur l'ecran
                //setBackground(Color.BLACK);
                panelInfo.setBackground(new Color(210, 180, 140));
                setVisible(true);
        }
        public static void main(String[] args) {
                new JolieInterface();
        }
}
```