Sommaire :

Sofiane kebci

Manuel d'utilisation du programme:

Afin d'exécuter mon programme, il suffit de lancer la classe Main.java, on vous propose en premier de choisir le fichier à parcourir, après avoir choisi ce dernier, vous avez la possibilité de lancer le parcours en profondeur, en largeur ou bien le meilleur d'abord. Si vous optez pour la 3ème option, il vous restera qu'un à déterminer quelle heuristique utiliser pour le parcours, je vous propose 5 :

-Misplacement heuristic
-EuclideanDistance heuristic
-ManhattanDistance heuristic
-MisplacedCorners heuristic
-WeightedTiles heuristic

Les explications des heuristiques seront décrites dans la suite du rapport.
Après avoir fait votre choix, le programme vous renvoie le résultat sous la forme suivante :

Number of treated states: xxxx states
Number of waiting states: xxxx states
---Result ResultFound— or ---Result ResultNotFound—

Et directement le programme et renvoyer au début pour choisir un fichier.

Explications concernant l'implémentation des algorithmes de parcours:

La classe FileReader comporte les méthodes:

- ArrayList<char[][]> readFileData : qui lit un fichier qui renvoie un liste de deux éléments : la matrice initiale et finale.
- ArrayList<String> findFileNames : qui parcours le dossier Taquin/problems et qui renvoie une liste des noms de tous les fichiers présents à l'intérieur.

Forme générale des 3 algorithmes : (En profondeur, en largeur et le meilleur d'abord)

1-Un constructeur qui initialise les grilles initiale et finale ainsi que le nombre de lignes de colonnes.

2-Une méthode boolean runAlgoSearch qui commence par ajouter la grille initiale à la liste open (au début ou à la fin selon l'algorithme qu'on veut implémenter) puis parcours cette dernière tant qu'elle n'est pas vide. Dans la boucle while :

-Il prend l'élément en tête de la liste et vérifie si c'est le même que l'état final, si c'est le cas, il renvoie le résultat sinon il continue.
-Il parcourt la grille à la recherche de la ligne et de la colonne de la case vide.
-Il appelle la méthode neighborCell 4 fois, avec en arguments la liste close, la liste open,la ligne et la colonne de chacune des 4 cases voisines à la case vide, la grille et finalement la ligne et la colonne de la case vide.
-Il ajoute la grille à la liste close.

3-Une méthode void neighborCell qui crée d'abord une copie de la grille et puis vérifie si la case voisine existe (si elle n'est pas en dehors de la grille), si c'est le cas, le programme

permute les deux cases(La case voisine et la case vide). Si cette nouvelle grille n'est pas dans open et dans close, on l'ajoute à la liste d'attente sinon on continue.

4- Une méthode boolean isNotIn qui vérifie si une grille est dans une liste donnée en arguments.

5-Une méthode char[][] deppCopy qui renvoie une copie d'une grille pour ne pas provoquer involontairement des modifications inattendues de la source ou de la copie.

6-Une méthode boolean canBeAccessed qui vérifie si une case existe dans une grille donnée.

7-Une méthode bolean equals qui vérifie si deux grille sont identiques.

Description des heuristiques:

- **Misplacement heuristic :**

Cette heuristique calcule simplement le nombre de tuiles mal placées par rapport à leur position cible dans l'état final, et utilise ce nombre comme mesure de la distance entre les deux états.

- **EuclideanDistance heuristic :**

Cette heuristique calcule la distance euclidienne entre l'état actuel du jeu et l'état final souhaité en considérant chaque case comme un point dans un espace à deux dimensions, où la position de chaque tuile est déterminée par ses coordonnées x et y dans la grille de jeu.

La distance euclidienne entre deux points est calculée en utilisant la formule mathématique suivante :

distance = racine carrée de $[(x2 - x1)^2 + (y2 - y1)^2]$

- **ManhattanDistance heuristic :**

Cette heuristique calcule la distance de Manhattan entre l'état actuel du jeu et l'état final souhaité en considérant la distance horizontale et verticale entre chaque tuile dans la grille de jeu. Contrairement à la distance euclidienne, la distance de Manhattan ne prend pas en compte la diagonale.

La distance de Manhattan entre deux points est calculée en utilisant la formule mathématique suivante :

distance = $|x2 - x1| + |y2 - y1|$

- **MisplacedCorners heuristic :**

Cette heuristique ressemble à l'heuristique de mauvaise localisation mais consiste à considérer seulement les coins du puzzle, qui sont les quatre tuiles situées aux coins de la grille de jeu. Cela permet de prendre en compte le fait que les coins sont des zones particulièrement difficiles à déplacer dans le jeu du Taquin, et que la recherche d'une solution optimale doit se concentrer sur ces tuiles.

- **WeightedTiles heuristic :**

Cette heuristique utilise une matrice de poids pour attribuer un poids spécifique à chaque tuile dans la grille de jeu. Les tuiles qui sont proches de leur position finale souhaitée et qui sont importantes pour la résolution du puzzle, comme les coins et les cases éloignées du centre, ont un poids plus élevé que les autres tuiles.

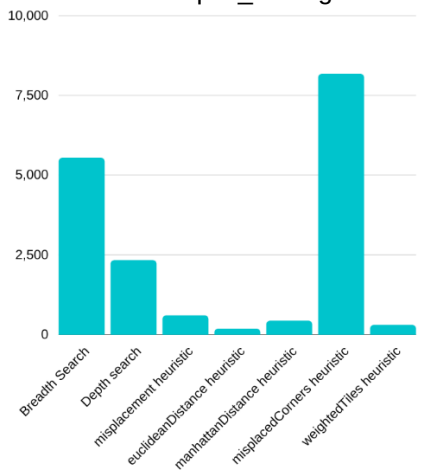
Analyse de la comparaison des différents parcours et heuristiques accompagnée de tableaux et/ou courbes:

Méthode	Matrice 1	Matrice 2	Matrice 3	Matrice 4	Matrice 4
Breadth Search	5538	6336	6392	20160	3
Depth search	2333	>507758	4917	20160	1
Misplacement heuristic	600	23	1295	20160	1
EuclideanDistance heuristic	181	40	470	20160	1
ManhattanDistance heuristic	435	36	339	20160	1
MisplacedCorners heuristic	8171	77291	1823	20160	1
WeightedTiles heuristic	302	20	1611	20160	1

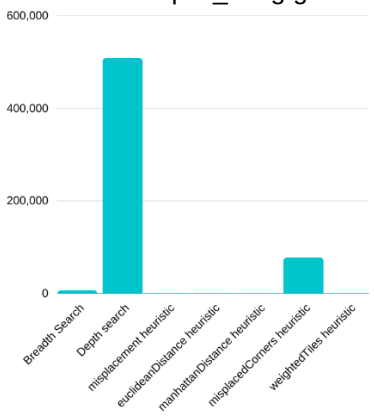
Avec :

Unité : Nombre d'états traités

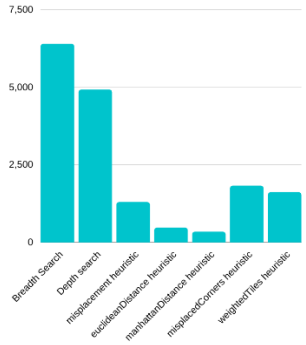
Matrice 1 : taquin_2x4b.grid.txt



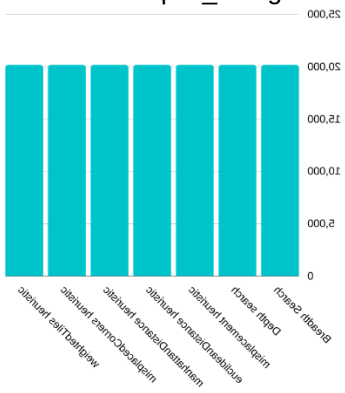
Matrice 2 : taquin_4x4g.grid.txt



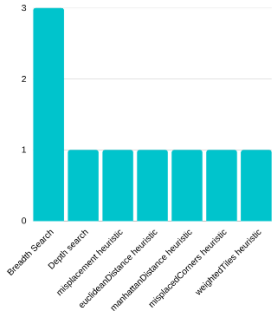
Matrice 3 : taquin_2x4c.grid.txt



Matrice 4 : taquin_2x4.grid.txt



Matrice 5 : taquin_3x3e.grid.txt



On se rend compte que les heuristiques nous permettent de trouver le résultats en très peu d'états et de ce fait, fait gagner beaucoup de temps de calcul, comme on peut bien le voir sur la figure de la matrice 2, le nombre d'états qu'il faut traiter pour arriver au résultats final en utilisant le parcours en longueur est >507758 (au bout de 6h j'ai arrêté le programme) alors que l'heuristique des poids nous permet de le trouver en seulement 20 états.

Les seuls moments ou les parcours en profondeur et en largeur ainsi que les heuristiques arrivent à la fin avec le même le nombre d'états traité sont : soit le résultat final n'existe pas comme on peut l'apercevoir sur la figure de la matrice 4, ou bien quand la matrice final est facilement atteignable (figure de la matrice 5), le nombre d'états traités par les heuristiques est le même que celui du parcours en profondeur.

Maintenant, si on comparait les heuristiques entre elles, on peut conclure qu'il est souvent très difficile de savoir lequel utiliser, laquelle est la plus effective et la plus rapide pour atteindre l'état final. Par example, pour la matrice 1, le meilleur choix serait d'utiliser l'heuristique de la distance euclidienne, en revanche pour la matrice 2, pour être plus performant, faudrait plutôt utiliser l'heuristique des poids ou bien celle des case mal placées, tandis que pour la 3ème matrice, l'heuristique de la distance de Manhattan est la plus efficace.

Pour conclure, on peut dire que les heuristiques sont très importantes pour guider la recherche et afin réduire la complexité d'un programme drastiquement, mais le choix de l'heuristique peut être difficile à faire car il n'y a pas de méthode précise et complète pour y arriver.