Rapport Algorithmique avancée

Mini Projet : Labyrinthe

Camelia Mazouz
Groupe de TD : 03
N°étudiant : 22109872

Table des matières

1. Introduction

Ce document a pour but de détailler le code du projet Labyrinthe.

Vous trouverez une liste des fonctions implémentées et le rôle de chaque fonction.

Le fichier labyrinthe.txt présent dans le répertoire, représente les entrées claviers pour effectuer le test.

2. Arborescence

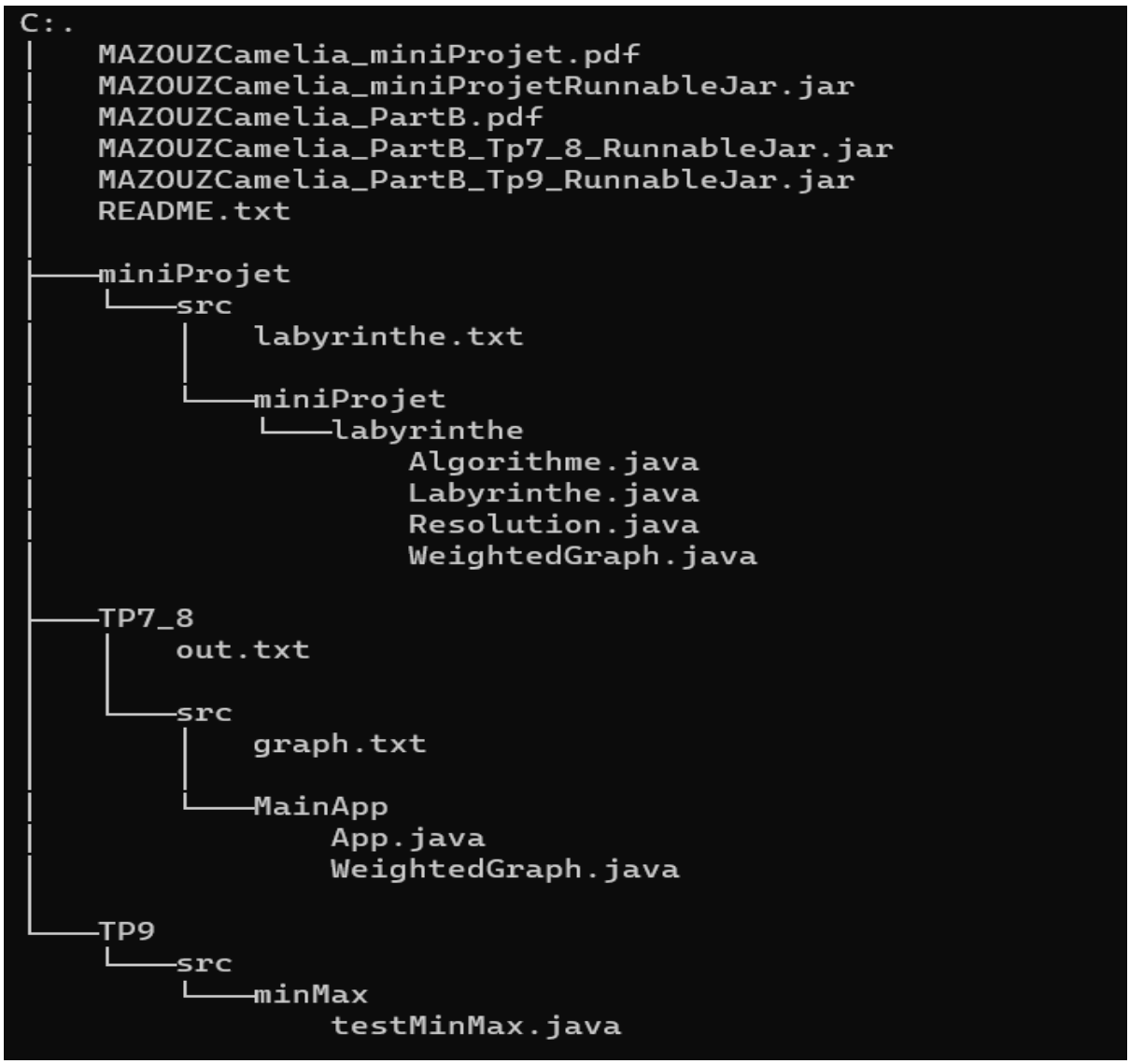

Figure 1

3. Exécution

- Exécution de la classe principale : Pour lancer le programme il suffit d'exécuter le fichier Resolution.java
- Exécution du jar : se déplacer dans le répertoire qui contient le jar exécutable et lancer la commande :
    - java -jar MAZOUZCamelia_miniProjetRunnableJar.jar

- Et mettre le chemin absolu du fichier qui contient le labyrinthe comme demandé dans la console.

Le fichier depuis lequel on récupère le graphe doit respecter le format suivant :

- Nombre d'instances
- Nombres de lignes
- Nombre de colonnes
- Labyrinthe

```
1
4
4
.D..
...F
.S..
####
```

Figure 2

Nombre de lignes, suivi du nombre de colonnes, suivi du labyrinthe.

4. Classes

Le projet est constitué de 4 classes :

a. **WeightedGraph.java** (classe du tp7_8) **:** cette classe gère les graphes, permet de créer un graphe avec des arêtes qui ont des poids.

b. **Algorithme.java** : qui contient l'algorithme A* (réalisé lors du tp7_8) utilisé pour trouver la sortie du labyrinthe.

c. **Labyrinthe.java** (inspirée du modèle de résolution de labyrinthe en C) **:** cette classe nous permet de gérer toutes les composantes d'un labyrinthe dont les dimensions du labyrinthe, le point de départ et d'arrivée du prisonnier, le point de départ du feu, une grille qui représente les lignes et colonnes du labyrinthe ainsi que les différentes fonctions qui gèrent le labyrinthe.

d. **Resolution.java :** cette classe nous permet de lire un fichier contenant une instance de labyrinthe, le résoudre et afficher le résultat final.

5. Choix d'implémentation

a. **Classe WightedGraph :** la modification apportée à cette classe est que si le nœud ajoute on précise son type : # pour un mur, pour une case vide, S pour sortie, D pour prisonnier et F pour feu. Ainsi que la fonction *initWeight* qui permet d'initialiser le temps individuel d'un sommet a l'infinie si ce dernier est un mur (car on ne peut pas traverser un mur), dans le cas contraire le temps assigné sera 1.

b. **Classe Algorithme.java :**
Chaque sommet (case du labyrinthe) a au maximum 4 voisins, donc pour le calcul de l'heuristique j'ai calculé la distance de Manhattan *(d(A,B) =* $|X_a-X_b|+|Y_a-Y_b|$*)* entre la position courante et la sortie.
Pour la recherche du plus court chemin il suffit de faire en sorte que A* ne passe pas par des murs (le temps individuel est à l'infini donc ce chemin ne sera jamais considéré) mais aussi de ne pas traverser les flammes, pour se faire il suffit de s'assurer que le voisin du sommet actuel ne soit pas de type 'F' et qu'on peut bouger le prisonnier.

c. **Resolution.java :** Dans cette classe la fonction *initLab* nous permet de lire un labyrinthe depuis le fichier passé en paramètres de la fonction et de retourner une instance du labyrinthe lu. Cette instance est récupérée dans la fonction main ou elle est ajoutée à une liste de labyrinthes. Sur chacun des labyrinthes de la liste *labyrinthes* on fait appelle à la fonction *run_instance* pour le résoudre.

d. **Classe Labyrinthe :** cette classe nous permet de gérer toutes les composantes d'un labyrinthe dont les dimensions du labyrinthe, le point de départ et d'arrivée du prisonnier, le point de départ du feu, une grille qui représente les lignes et colonnes du labyrinthe et qui nous permet de gérer l'affichage après les différentes modifications apportées sur le labyrinthe (pour ne pas écraser le contenu de départ notre graphe on fait toutes les modifications dans la grille) ainsi que les différentes fonctions qui gèrent le labyrinthe.

public boolean burn_around(int x, int y) :

- Paramètres : les coordonnées du feu
- Fonction qui propage le feu autour de lui, renvoie vrai si le feu a touché la sortie ou le joueur.

public boolean can_move_dir(int x, int y, char dir) :

- Paramètres : coordonnées du prisonnier et la direction dans laquelle on veut le faire bouger.
- Retourne vraie si le joueur peut bouger dans la direction *dir* faux sinon.

public int can_move(int x, int y):

- Paramètres : coordonnées du prisonnier
- Retourne le nombre de directions dans lesquelles le prisonnier peut bouger.

public boolean win_move(int x, int y)

- Paramètres : coordonnées du prisonnier
- Retourne vraie si le prisonnier est juste à cote de la sortie, faux sinon

public char run_instance()

- Fonction qui permet de lancer le jeu :
  - On récupère dans une liste le résultat de A*.
  - Si le nombre de coups du feu vers la sortie et inférieur au nombre de coups du prisonnier vers la sortie, le prisonnier ne peut pas s'échapper.
  - Si le feu et le prisonnier arrivent à la sortie au même moment le prisonnier perd.
  - Sinon, on propage le feu et on fait avancer le prisonnier :
    - On efface la position d'avant du joueur, et on le fait avancer

public String toString()

- Retourne une chaine de caractères qui contient le labyrinthe.