Le projet choisi est le Scénario 2 : Graphes et Image

Ce document retrace les efforts effectués concernant l'implémentation du projet. Il décrit les classes et méthodes utilisés, les techniques suivies et réponds aux questions posés.

I. Généralités

Le langage de programmation choisi pour le projet est Java.
Tout d'abord, le programme calcule le plus court chemin entre le point en haut à gauche du graphe et celui à son opposé en bas à droite. Ce chemin est déterminé grâce à l'algorithme A-Star, déclinaison de Dijkstra, et selon la différence d'intensité en niveau de gris des pixels adjacents (haut, bas, droite, gauche). Il est affiché à l'aide de JavaFX par une ligne rouge, directement sur le graphe.
Concernant le graphe, celui-ci peut être déterminé par deux méthodes différentes :

i. Par un algorithme aléatoire, qui étant donné un nombre de pixels renvoie une matrice d'intensité depuis laquelle est déduis le plus court chemin.
ii. Par une image donnée, convertie en un graphe et dont les arêtes sont l'intensité de ses pixels en niveaux de gris.

Le projet est rendu via une archive java, que l'on peut exécuter avec la commande suivante, qui indique le chemin vers la librairie JavaFX (noté cheminVersLib dans la commande et téléchargeable sur *gluonhq.com/products/javafx)*, au cas où elle ne serait pas installée :

```
→ java --module-path cheminVersLib --add-modules javafx.controls -jar CodeMiniProjet.jar
```

Si JavaFX est installée, la commande peut être allégée telle que :

```
→ java -jar CodeMiniProjet.jar
```

Des arguments peuvent aussi être pris par le programme, selon la méthode de définition du graphe choisi, ces commandes sont précisées dans la partie **III. Résultat** de ce document, et dans le **README.txt.**

L'extraction de l'archive jar laissera apparaître l'arborescence du projet :

```
/CodeMiniProjet
├── /graphe
│   ├── Pixel.class
│   ├── Arete.class
│   ├── Graphe.class
│   └── AEtoile.class
├── /ui
│   └── MainApp.class
├── /META-INF
│   └── MANIFEST.MF
└── module-info.class
```

Précision :

i. Le répertoire META-INF contient le fichier MANIFEST.MF indiquant une unique chose : la classe main à exécuter.
ii. Le fichier module-info.class définit les informations sur le module de l'application permettant d'organiser le code en modules clairement délimités et de gérer leurs dépendances.

II. Classes et méthodes

Dans le package **graphe**, les classes suivantes sont implémentés :

- **Pixel**
  Cette classe représente un pixel dans une image en termes de ses coordonnées (x, y) et de son intensité lumineuse (un nombre entier représentant l'intensité du pixel).
  - *equals* redéfinit la méthode equals pour comparer deux objets Pixel. Elle compare les coordonnées (x, y) pour savoir s'ils se trouvent à la même position dans l'image. On implémente cette méthode car essentielle pour les HashMap implémentés dans les prochaines classes.
  - *hashCode* redéfinit la méthode hashCode, qui génère un code unique pour chaque objet Pixel basé sur ses coordonnées (x, y). Cette méthode est elle aussi importante pour l'utilisation de Pixel dans les HashMap, afin d'assurer une répartition efficace des objets dans ces structures. Si l'on ne redéfinie pas la méthode hashCode, Java utilisera la version par défaut fournie par la classe Object, qui renvoie un entier basé sur l'adresse mémoire de l'objet. Cela peut poser des problèmes de collision, et de mauvaise distribution des objets.

- **Arete**
  La classe Arete représente une arête dans le graphe. Chaque arête est définie par deux pixels : un point de départ (depuis) et un point d'arrivée (vers), ainsi qu'un poids représentant la différence d'intensité entre les deux pixels voisins.

- **Graphe**
  La classe représente un graphe où les sommets sont des pixels d'une image, et les arêtes représentent les relations de voisinage entre ces pixels. Les arêtes sont pondérées en fonction de la différence d'intensité entre les pixels voisins.
  - *ajouterPixel* ajoute un pixel au graphe en l'insérant dans la liste d'adjacences.
  - *ajouterArete* ajoute une arête entre deux pixels donnés. Le poids de l'arête est calculé comme la différence absolue d'intensité entre les pixels concernés.
  - *getAretes* retourne la liste des arêtes sortantes d'un pixel donné.
  - *getListeAdjacence* retourne l'ensemble complet de la liste d'adjacences du graphe, c'est-à-dire une Map qui associe chaque pixel à ses voisins.
  - *depuisMatrice* est une méthode statique permettant de créer un graphe à partir d'une matrice d'intensités, typiquement utilisée pour représenter une image. Elle crée un pixel pour chaque élément de la matrice, puis ajoute des arêtes entre les pixels voisins, en respectant la connectivité 4.

- **AEtoile** (*Dijkstra remplacé par A*, voir raisons dans la partie IV. Efforts & Axes d'amélioration)*
  Cette classe implémente l'algorithme A* (A-Star), un algorithme de recherche de chemin efficace qui utilise une fonction heuristique pour estimer le coût du chemin restant. Elle se concentre sur le calcul du chemin le plus court entre deux pixels dans un graphe en prenant en compte à la fois de la distance déjà parcourue depuis le point de départ et d'une estimation de la distance restante jusqu'à la destination.
  - *plusCourtChemin* détermine le plus court chemin entre deux pixels dans un graphe : L'algorithme A* explore le graphe en priorisant les nœuds qui semblent les plus prometteurs pour atteindre la destination, en se basant sur un score total $f(n) = g(n) + h(n)$. Ici, $g(n)$ représente le coût exact pour atteindre le nœuds n depuis le point de départ, et $h(n)$ est une estimation heuristique du coût restant jusqu'à l'arrivée. L'algorithme commence par le point de départ, en évaluant les nœuds voisins et en mettant à jour leurs scores si un chemin plus court est trouvé. Les nœuds sont stockés dans une file de priorité pour traiter en premier ceux avec le plus petit score $f(n)$. Lorsque la destination est atteinte, l'algorithme reconstruit le chemin en retraçant les prédécesseurs depuis la destination jusqu'au point de départ.
  - *heuristique* calcule une estimation du coût entre deux pixels, et est utilisée pour guider l'algorithme. Ici, on utilise la distance de Manhattan comme heuristique car semble adaptée pour ce genre de problème. Le choix de l'heuristique est détaillé dans la partie IV. Efforts et Axes d'amélioration.

Dans le package **ui**, la classe principale est implémenté :

- **MainApp**
  La classe MainApp étend Application de JavaFX, ce qui en fait une application graphique pour Java. Elle sert de point d'entrée pour l'interface utilisateur de l'application, en affichant un graphe ou une image et en permettant de visualiser le plus court chemin entre deux pixels en utilisant l'algorithme A*.
  - *executerGrapheAleatoire* génère un graphe aléatoire sous forme d'une matrice d'intensités, crée le graphe à partir de cette matrice, puis utilise l'algorithme A* pour trouver le plus court chemin. Elle affiche ensuite le graphe et le chemin sur un canvas JavaFX.
  - *executerGrapheDepuisImage* charge une image depuis un fichier spécifié, la convertit en une matrice d'intensités en niveaux de gris, puis crée un graphe à partir de cette matrice. Elle utilise également A* pour trouver le plus court chemin et affiche le tout sur un canvas JavaFX.
  - *chargerImage* charge une image depuis un fichier et la convertit en une matrice d'intensités en niveaux de gris.
  - *dessinerImage* dessine l'image ou le graphe sur le canvas. Chaque pixel est dessiné comme un rectangle avec une couleur de gris correspondant à son intensité.
  - *dessinerChemin* cette méthode dessine le chemin trouvé par A*. Elle trace des lignes rouges entre les pixels du chemin.

i. Fonctionnement général
   La classe MainApp, via sa méthode main, récupère les arguments entrés en ligne de commande lors de l'exécution. Deux choix sont possibles pour l'utilisateur, soit obtenir le plus court chemin d'un graphe aléatoire, soit obtenir le plus court chemin depuis une image. Ce choix est détaillé dans la partie III. Résultat. La méthode start() est la méthode principale de l'application, appelant selon le choix précédent *executerGrapheAleatoire* ou *executerGrapheDepuisImage*.
ii. Interaction avec le package graphe
   Les méthodes *executerGrapheAleatoire* et *executerGrapheDepuisImage* créent toutes deux un objet Graphe en utilisant la matrice d'intensités. Ensuite, le programme utilise l'algorithme A* pour trouver le plus court chemin entre deux pixels.
iii. Intégration avec JavaFX :
   Pour l'affichage, la classe crée une scène avec un Canvas où chaque pixel du graphe est dessiné sous forme de rectangle. La méthode *dessinerImage* dessine chaque pixel en fonction de sa valeur d'intensité dans la matrice (en niveaux de gris), ce qui permet de visualiser l'image ou le graphe aléatoire. La méthode *dessinerChemin* est responsable de l'affichage du chemin trouvé par l'algorithme de Dijkstra. Le chemin est représenté par des lignes rouges reliant les pixels successifs du chemin optimal.

III. Résultats

Pour tester le programme, il faut entrer les arguments au lancement du programme selon le choix du mode.

i. Si l'on veut obtenir le plus court chemin d'un graphe aléatoire, les arguments sont le nombre de pixels du graphe (le résultat sera un graphe de nombrePixels × nombrePixels) et la taille des pixels. La commande est :
   → ... CodeMiniProjet.jar aleatoire *taillePixels nombrePixels*
ii. Si l'on veut obtenir le plus court chemin depuis une image, il faut tout d'abord faire attention à prendre une image qui n'est pas d'une trop grande résolution, et faire varier la la taille des pixels pour que le résultat soit affichable. La commande est :
   → ... CodeMiniProjet.jar image *taillePixels cheminImage*

Attention au ratio nombrePixels (ou résolutionImage) / taillePixels pour que tout le graphe soit affiché entièrement.

Ci-dessous quelques exemples de résultats :
La ***figure 1*** représente un graphe aléatoire généré lors de l'exécution du programme sans argument. La vérification des résultats obtenus peut être via les labyrinthes ***figure 2*** et ***figure 3***, où l'on constate que l'on obtient des chemins satisfaisant, bien que ce ne soient pas exactement les plus courts.

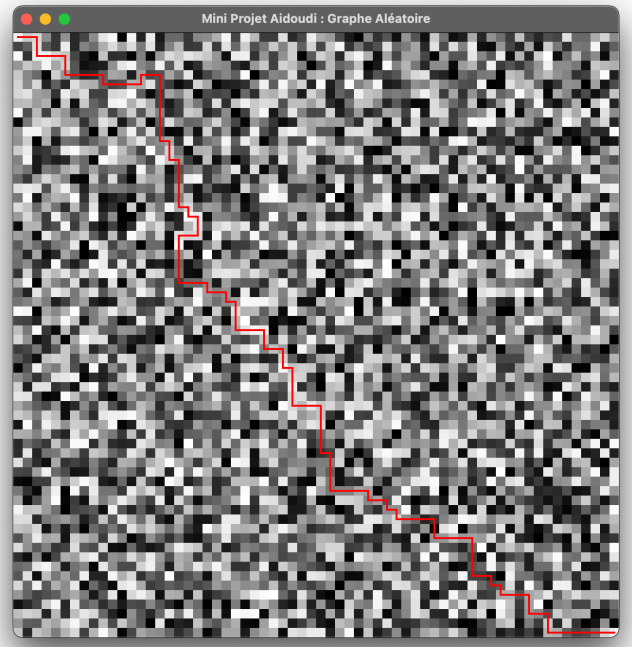


figure 1
graphe aléatoire de 64 × 64 pixels

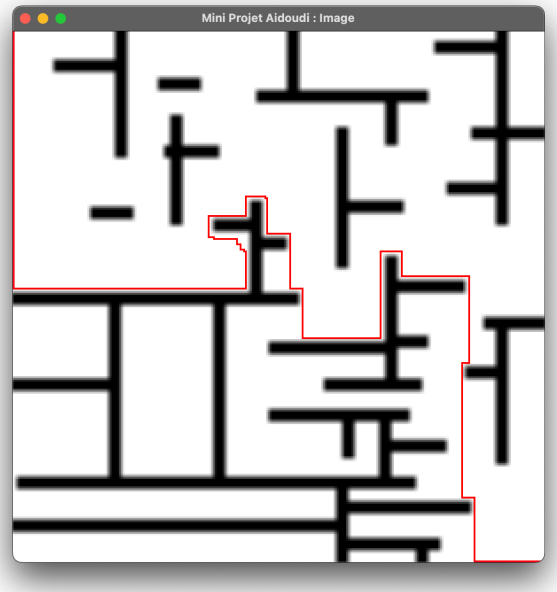


figure 2
labyrinthe1.png disponible dans le dossier du projet

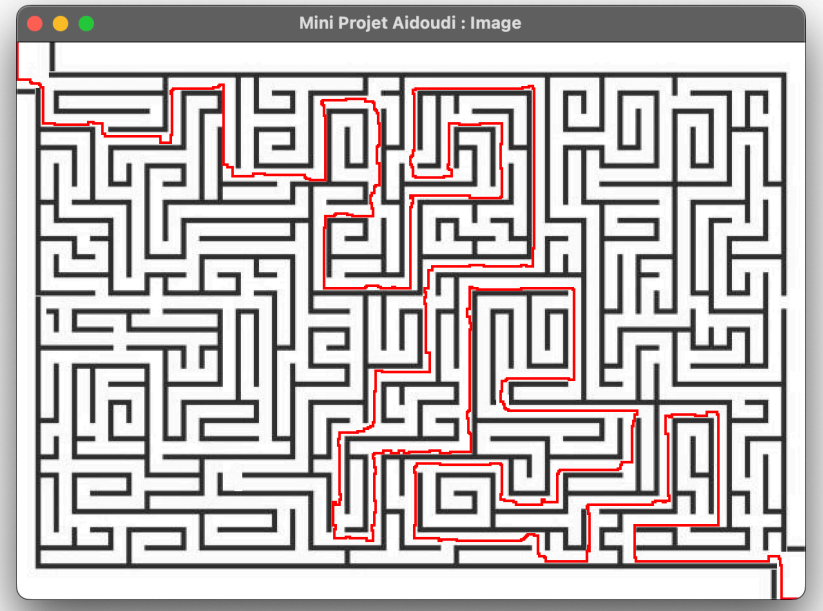


figure 3
labyrinthe2.jpg disponible dans le dossier du projet

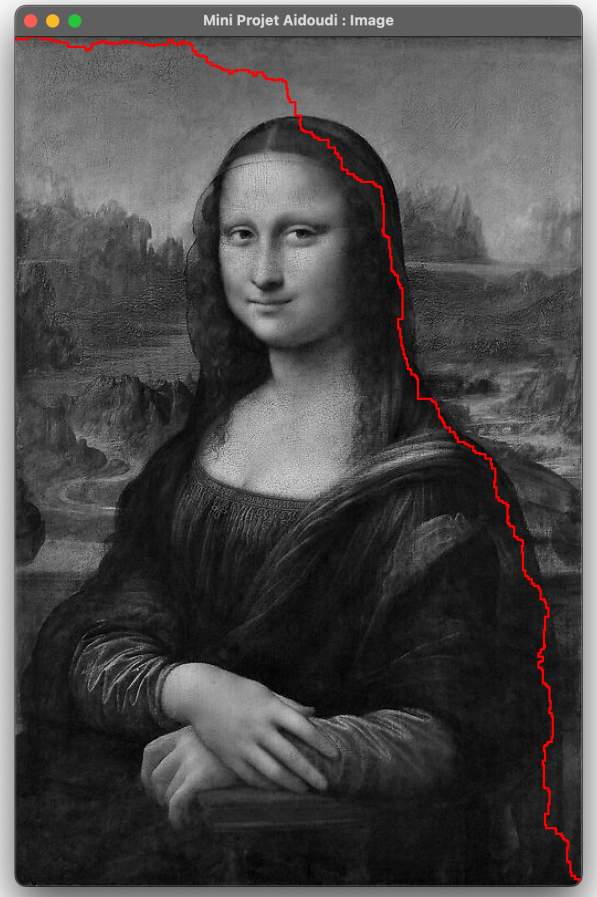
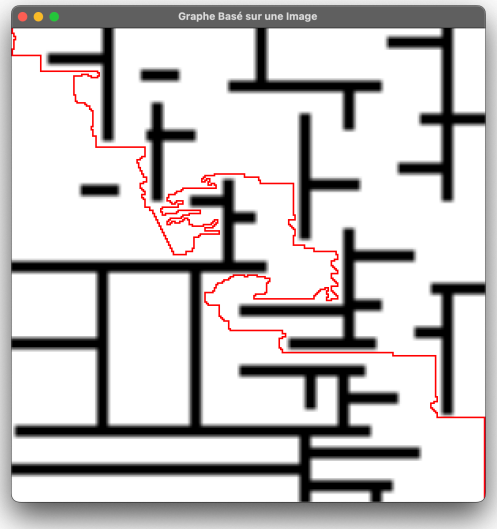


figure 4
La Joconde de Léonard de Vinci

IV. Efforts & Axes d'amélioration

Les classes Pixel, Aretes, et Graphe ne sont pas celles qui ont rendu l'implémentation compliquée. En revanche, l'utilisation de JavaFX, et l'implémentation et le test de l'algorithme de résolution du plus court chemin sont elles des étapes plus délicates.

Tout d'abord, pour JavaFX, il a fallu apprendre à faire le lien entre les algorithmes et la fenêtre affichée, c'est-à-dire afficher la matrice d'intensité dans le cas d'un graphe aléatoire, ou encore afficher le résultat donné par AEtoile directement sur la fenêtre. Aussi, il a fallu apprendre à partager un projet JavaFX via un jar, ce qui peut s'avérer fastidieux.



Ensuite, concernant les algorithmes de résolution du plus court chemin, j'ai dans un premier temps essayer de trouver une solution avec l'algorithme de Dijkstra, mais je ne suis pas parvenu à trouver une implémentation donnant des solutions aussi satisfaisantes que celles avec A*, malgré de nombreux changements. Par exemple, j'ai obtenu le résultat ci-contre, bien plus long que celui obtenu avec A*.
J'ai ainsi abandonné Dijkstra et remplacé A*. Celui-ci m'as immédiatement donné une meilleure solution, celle dans la partie III. Résultat. Son implémentation était aussi plus facile selon moi que celle de Dijkstra.

Concernant l'heuristique utilisée, l'utilisation de la distance de Manhattan est justifiée par la nature orthogonale du graphe, où les déplacements sont restreints aux axes horizontaux et verticaux. Elle garantit également une exécution plus rapide pour des grandes grilles, car elle évite les opérations de racine carrée nécessaires au calcul de la distance Euclidienne, ce qui est un avantage en termes de performance dans des scénarios où les différences de précision sont insignifiantes. De plus, la différence entre les résultats obtenus avec ces deux heuristiques est minime, car les chemins calculés sont généralement similaires, surtout sur des grilles où les coûts de déplacement sont uniformes (labyrinthe) ou faiblement variés (image).

Concernant les axes d'amélioration, on peut entre autres citer

i. Le temps d'exécution pour de grands graphes : par exemple, un graphe généré aléatoirement de 1024 par 1024 pixels implique une dizaine de secondes de résolution, comparé a un graphe de 128 par 128 pixels qui lui est résolu quasiment immédiatement.
ii. L'affichage pour des images avec une trop grosse résolution : l'image doit être downscale par l'utilisateur vers une définition en dessous les 500 par 500 pixels (si possible), et la taille des pixels doit être abaissé.
iii. L'interface avec JavaFX et l'exécution du programme avec des arguments : on pourrait présenter les choses autrement avec une fenêtre gérant toutes les entrées-sorties et évitant d'utiliser des arguments à l'exécution.

V. Sources

Algorithme A* : *redblobgames.com/pathfinding/A*/implementation.html* , *datacamp.com/tutorial/A*-algorithm*

Distance de Manhattan : *datacamp.com/fr/tutorial/manhattan-distance*

JavaFX : *examples.javacodegeeks.com/java-development/desktop-java/javafx/javafx-canvas-example/* , *jenkov.com/tutorials/javafx/canvas.html* , *reddit.com/r/JavaFX/comments/mz9mqq/how_do_i_distribute_my_javafx_application/*