Projet d’informatique scientifique :

1. Compréhension des algorithmes

2. Implémentation et choix de structures de données

3. Sortie graphique

4. Approfondissement et algorithmes suboptimaux

1.

L’algorithme de recherche aveugle (floodfill) consiste en une recherche en largeur d’un environnement, c’est à dire un arbre de recherche où chaque branche est partiellement traversée une fois par niveau de profondeur de l’arbre, à l’opposé d’une recherche en profondeur qui traverse chaque branche dans leur intégralité les unes après les autres. Dans un espace à deux dimension, cette recherche se représente sous forme de losange qui s’étend du point de départ jusqu’à atteindre le point d’arrivée.

L’algorithme de dijkstra est une version du floodfill qui prend en compte le poids de déplacement variable entre deux états. Dans un espace de cases, ce poids est représenté par le ralentissement causé par le déplacement d’une case à une autre. L’algorithme de dijkstra fonctionne comme l’algorithme de floodfill, mais il priorise les case au poids le plus léger afin d’atteindre le point d’arrivée avec une distance minimale.

L’algorithme de A* fonctionne de manière similaire, mais il prend également en compte une valeur heuristique qui sert de minorant pour la distance entre deux case. Dans le cas de l’espace de cases, il s’agit de la distance à vol d’oiseau (l’heuristique de Manhattan). L’algorithme va donc prioriser les cases au poids le plus léger, mais aussi à la valeur heuristique h la moins élevée, pour atteindre le point d’arrivée plus rapidement que l’algorithme de dijkstra.

2.

Dans le contexte de ce projet, les cartes sont représentées comme des espace 2d où chaque pixel représente une case. J’ai donc décidé de les implémenter sous forme de matrice de caractères, où chaque élément de la matrice représente une case, et le caractère représente le type de la case. De manière générale, les algorithmes de pathfinding se font sur des graphes, mais ici le format des cartes permet de les implémenter de manière plus efficace avec des matrices, dont la construction donne les informations nécessaires pour obtenir les états adjacents et les distances heuristiques.

Les trois algorithmes sont programmés de manière similaire. On commence par le point de départ, et on lance une boucle qui continue jusqu’à ce que l’objectif soit atteint ou qu’on ait déterminé qu’il n’y a aucun chemin vers l’objectif. On regarde les case adjacentes au point courant, et si elles n’ont pas été visitées, on ajoute le chemin qui y mène à la liste des chemins à vérifier. Dans le cas de dijkstra et A*, on trie les chemins dans une liste triée par rapport à respectivement leurs valeurs g et g + h, de manière à prioriser les chemins les plus rapides. Si un de ces chemin finit sur l’objectif, on sort de la boucle et on renvoie ce chemin comme solution optimale. Sinon, on récupère le prochain chemin et on recommence à chercher les cases adjacentes. Si la liste de chemins à vérifier est vide, cela veut dire qu’il n’existe aucun chemin menant à l’objectif.

Pour implémenter les chemins à visiter, j'ai d'abord utilisé des vecteurs de points, mais avancer dans la recherche nécessite de connaître le dernier point du chemin traité. Pour pouvoir accéder à cette information en un temps constant, j'ai décidé d'implémenter les chemins comme des tuples qui contiennent les informations du chemin et du dernier point. Ce changement dans la structure de mes algorithmes a grandement augmenté les performances de mes fonctions (dans l'évaluation du 15 mars, le temps de recherche du chemin optimal est passé de 15 secondes à 0,05 secondes). Pour les algorithmes de Dijkstra et A*, j'ai également inclus les informations sur la distance parcourue et la distance heuristique vers l'objectif.

3.

Pour afficher les chemins trouvé à partir des algorithmes, j'ai utilisé les packages Images, Colors, et ImageView de Julia. Le programme affiche d'abord la carte initiale avec les points de départ et d'arrivée, puis affiche pour chaque fonction la carte modifiée par chaque chemin, avec les cases parcourues en rouge et les cases visitées en rose.

Pour transformer la matrice de caractères en image, je crée une matrice de valeurs RGB de même dimension, et je convertit chaque caractère de la matrice initiale en leur couleur correspondante. Puis j'utilise la fonction Images.save afin de sauvegarder l'image de la carte sous un fichier png, et la fonction ImageView.imshow afin d'afficher une fenêtre représentant l'image de la carte.

4.

En changeant les critères selon lesquels les chemins sont priorisés, on varie de l'algorithme de Dijkstra à l'algorithme glouton best-first. La fonction d'ajout trié que j'ai utilisée pour les algorithmes de Dijkstra et A* peut être modifiée pour inclure la variable flottante w, qui varie de 0 à 1 et détermine quel critère est priorisé.

Pour w = 0, la fonction est équivalente au best-first. Pour w = 0,5, la fonction est équivalente à A*, Pour w = 1, la fonction est équivalente à dijkstra. Cela signifie que plus w est bas, plus l'algorithme est rapide, mais plus le résultat s'éloigne de la solution optimale. Pour w $\in$ [0,5;1], la solution est toujours optimale, et il n'est pas intéressant de faire varier w dans cet intervalle car A* sera toujours la meilleure option. Pour w $\in$ [0;0,5], l'algorithme peut être plus rapide que A*, mais aussi renvoyer un résultat suboptimal.

J'ai écris une fonction WA* qui permet de spécifier la valeur de w et de faire tourner l'algorithme avec ce critère constant, puis j'ai écris une fonction DWA* où la valeur de w varie selon la distance par rapport à l'objectif et le poids de déplacement de la case courante. Ainsi, l'algorithme se comporte davantage comme best-first lorsqu'il est loin de l'objectif et peut se déplacer librement, et davantage comme dijkstra lorsqu'il est proche de l'objectif ou est ralenti par son environnement.