Path Finding Project

Développé par NTWARI Robin

1/03/2025

1 Introduction

Ce projet implémente quatre algorithmes de recherche de chemin : **BFS**, **Dijkstra**, **A*** (Arrivée*) et **Glouton**. Le terrain est chargé depuis un fichier `.map` (ou `.txt`) à partir de la 5ème ligne. La grille est constituée de cases normales (représentées par `'.'`) et d'obstacles (représentés par `'@'`, `'T'`, `'S'` et `'W'`). Chaque algorithme vérifie que les points de départ et d'arrivée ne se trouvent pas sur un obstacle, puis calcule un chemin (optimal pour BFS, Dijkstra et A*; approché pour l'algorithme glouton).

Remarque sur l'affichage : Bien que le chemin soit correctement calculé, l'affichage console ne reflète pas fidèlement la structure du terrain en raison du wrapping automatique des lignes par le terminal. C'est pourquoi le résultat est sauvegardé dans un fichier texte, offrant ainsi une représentation fidèle du chemin.

2 Installation et Configuration

- **Dépendances :** Aucune bibliothèque externe n'est requise pour le fonctionnement de base, sauf pour l'affichage coloré en console si vous souhaitez utiliser `printstyled` (disponible dans certains environnements).
- **Configuration du fichier d'entrée :** Le fichier d'entrée doit être au format `.map` ou `.txt` et contenir le terrain à partir de la 5ème ligne. Veillez à ce que les lignes soient de longueur identique pour un affichage correct.

3 Description des Fonctions et Algorithmes

`chargerFichier`
: Lit le fichier d'entrée à partir de la 5ème ligne, retire les espaces superflus et construit la grille sous forme d'un vecteur de vecteurs de caractères.
Complexité : $O(n)$, où n est le nombre de lignes lues.

`isValidCoordinate`
: Vérifie si une coordonnée (tuple) se trouve dans les limites de la grille.
Complexité : $O(1)$.

`cout_mouvement`
: Retourne le coût de déplacement selon le type de case (ex. arbre, eau, sable, obstacle, etc.).
Complexité : $O(1)$.

`get_voisins`
: Renvoie les voisins immédiats (haut, bas, gauche, droite) d'une case, en vérifiant leur validité dans la grille.
Complexité : $O(1)$ (nombre constant de voisins).

`print_console_path`
: Ces fonctions construisent une copie de la grille, remplacent le chemin par `'*'`, le point de départ par `'D'` et l'arrivée par `'G'`. La première affiche le résultat en console, tandis

que la seconde écrit la représentation dans un fichier texte.

Note : L'affichage console est limité par le wrapping automatique, c'est pourquoi la sauvegarde dans un fichier est recommandée pour une représentation fidèle.

Algorithmes de path finding (BFS, Dijkstra, A*, Glouton)

Chaque algorithme suit ces étapes :

- Vérification de la validité des coordonnées de départ et d'arrivée.
- Parcours de la grille en utilisant des structures comme **files** (pour BFS), **dictionnaires** pour mémoriser les coûts et prédécesseurs, et **ensembles** pour suivre les nœuds visités.
- Reconstruction du chemin à partir des prédécesseurs.

Complexités approximatives :

- **BFS** : $O(N + E)$, efficace pour des grilles de taille modérée.
- **Dijkstra** : $O(N^2)$ avec une implémentation simple, optimisable avec un tas de priorité.
- **A*** : $O(N + E)$ en pratique grâce à l'utilisation de l'heuristique (distance de Manhattan utilisé dans le code).
- **Glouton** : Complexité réduite mais ne garantit pas le chemin optimal.

Choix des structures : j'ai choisi d'utiliser un dictionnaire et un ensemble pour leur simplicité et leur clarté dans l'implémentation, même si des structures plus avancées pourraient être utilisées pour améliorer les performances sur de grandes instances notamment des tas plus spécifiquement pour les algos Dijkstra et A* mais comme je ne suis pas à l'aise avec leur implémentation, j'ai utilisé les structures mentionnées dessus qui sont tout autant efficace.

4 Affichage du Résultat

Le calcul du chemin est correct pour chacun des algorithmes, mais l'affichage dans la console est souvent dégradé par le wrapping automatique des lignes, surtout pour les grands fichiers d'entrée. Pour contourner cette limitation, j'avais tenté de développer une interface graphique mais je ne m'en sortais pas raison pour laquelle j'ai opté pour un affichage de type console mais coloré pour distinguer le départ, l'arrivée et le chemin trouvé.

5 Utilisation

Les fonctions d'algorithmes sont conçues pour être appelées directement depuis la REPL ou via un script, par exemple :

```
algoAstar("battleground.map", (12, 25), (25, 65))
```

De même, les autres algorithmes (BFS, Dijkstra, Glouton) peuvent être appelés en fournissant directement le nom du fichier et les coordonnées de départ et d'arrivée sous forme de tuples. Aucune variable globale n'est requise, ce qui élimine le risque d'erreur `UndefVarError` que j'ai rencontrée plusieurs fois lors de la programmation du code.

6 Limitations et Amélioration

- **Affichage console** : La représentation visuelle du chemin dans la console est limitée par le wrapping automatique. L'utilisation d'une interface graphique (par exemple avec Plots.jl) ou l'export vers un fichier permet d'obtenir une représentation fidèle.
- **Optimisation des algorithmes** : Pour les grandes instances, l'utilisation de structures de données plus efficaces (comme un tas de priorité pour Dijkstra et A*) pourrait améliorer les performances.

7 Conclusion

Ce projet offre une implémentation des 4 algorithmes de recherche de chemin, en mettant l'accent sur la simplicité et la clarté de l'implémentation. Bien que le calcul du chemin soit exact, l'affichage console peut être dégradé par les limitations du terminal. La représentation graphique pourrait permettre d'obtenir une représentation correcte du chemin trouvé.