Projet d'Informatique Scientifique
X32I130

RICHARD Oscar - 680A

Sommaire

1 Commandes d'exécution

Une fois placé dans le dossier "algorithmes", on entre la commande "julia".

Ensuite on entre include("nom du fichier.jl") pour exécuter l'algorithme correspondant sur la map renseignée dans le fichier "config.jl" à la base du projet.

Ce dernier fichier permet aussi de modifier d'autres paramètres d'exécution, comme les points de départ et d'arrivée.

On peut ajouter d'autres fichier .map dans le dossier "maps".

2 Structure de données et fonctions communes

Le fichier "defs.jl" contient des structures de données et plusieurs fonction que les différents algorithmes appellent.

2.1 MutableBinaryHeap

C'est une structure de données du module DataStructures qui représente un tas binaire.

Un tas binaire est un arbre binaire où chaque noeud est plus petit ou égal à ses enfants pour un tas min (ce qui est utilisé ici), ou plus grand ou égal à ses enfants pour un tas max.

Cette propriété fait du tas binaire une structure de données efficace pour implémenter des priorités de file d'attente, utile dans notre cas pour continuer d'explorer le chemin le plus court.

Les tas binaires utilisés sont mutables, impliquant qu'on peut utiliser des fonctions de modifications sur ces tas, comme "pop" pour récupérer l'élément prioritaire (celui correspondant au plus court chemin actuel).

2.2 NodeData

C'est un enregistrement utilisé pour les algorithmes de Dijkstra et A star, comportant 3 attributs :

- position, qui correspond à la position de la case courante, donc un tuple de 2 entiers
- total_distance, qui correspond à la distance entre le point de départ et la position, donc un entier ou un flottant en fonction de l'algorithme

- heap_index, qui correspond à l'indice de la position dans le tas binaire

2.3 get_map_matrix

Il s'agit d'un parseur permettant de récupérer les informations contenues dans un fichier .map et de les transformer en matrice correspondant à la carte sur lesquels peuvent être exécutés les algorithmes.

On passe à cette fonction le chemin relatif entre le fichier .map et le fichier de l'algorithme appelant la fonction.

Les première et quatrième lignes d'un fichier .map ne sont pas utilisées, on se sert uniquement des deuxième et troisième lignes pour les dimensions de la matrice et du reste des lignes pour remplir cette dernière.

2.4 is_reachable

C'est un test vérifiant qu'une case est théoriquement atteignable, dans le sens où on teste si, en se trouve sur une case adjacente, on pourrait atteindre la case désirée avec un déplacement.

Cette fonction ne permet par exemple pas de savoir si une case est globalement atteignable, ou si elle est entourée d'obstacles barrant la route vers cette-dernière.

2.5 get_rebuilt_path

C'est une fonction utilisée par les algorithmes de Dijkstra et A star, permettant de reconstituer le chemin reliant le point de départ au point d'arrivée grâce à un dictionnaire (position => NodeData) implémenté dans ces algorithmes.

Le principe est qu'une clé de ce dictionnaire correspond à la case suivante par rapport à celle contenue dans NodeData.position

2.6 plural_adjuster

C'est une fonction permettant d'accorder en nombre certains mots, pour obtenir de meilleurs prints.

On passe la quantité associée au mot à accorder, ainsi que les 2 accords possibles.

2.7 print_solution

C'est une fonction utilisée par tous les algorithmes pour afficher proprement la taille du chemin, le nombre de case explorées et le chemin reconstitué.

3 Algorithmes

3.1 Algorithme flood fill

C'est un algorithme simple parcourant toutes les cases voisines à la case courante encore inexplorées, et qui stocke les résultats intermédiaires de distance entre la case de départ et la case actuelle pour reconstruire le chemin final s'il existe.

On le reconstruit en partant de la case d'arrivée, puis en cherchant récursivement la case voisine visitée présentant une distance au point de départ inférieure et consécutive à celle entre la case courante et le point de départ.

Cet algorithme met environ 0.2 secondes à s'exécuter et parcourt 127838 cases pour un chemin final d'une taille de 282 cases.

3.2 Algorithme de Dijkstra

Comme décrit précédemment, on utilise ici un MutableBinaryHeap permettant d'obtenir à chaque itération le chemin présentant la distance la plus petite entre son ultime case et le point de départ.

On traite donc en priorité les chemins qui s'apparentent à être ceux les plus courts.

Cet algorithme met environ 0.25 secondes à s'exécuter et parcourt 107370 cases pour un chemin final de 335 cases.

3.3 Algorithme A star

Pour les mêmes raison que dans l'algorithme de Dijkstra, cet algorithme utilise un MutableBinaryHeap.

Il utilise aussi une structure PathData, un enregistrement comportant 2 attributs :

- total_distance, qui correspond à la distance entre le point de départ et la position, donc un flottant

- position, qui correspond à la position de la case associée à la distance, donc un tuple d'entiers. Cet enregistrement est utilisé avec un dictionnaire (position => PathData) pour les calculs de "distance manhattan"

La fonction "manhattan_distance" mentionnée ci-dessus correspond à une heuristique utilisée pour estimer le coût (de mouvements) restant entre une case donnée et la case de destination. Le calcul est donné par

$$manhattan(case, dest) = abs(case.x - dest.x) + abs(case.y - dest.y) \quad (1)$$

Cet algorithme met environ 0.16 secondes à s'exécuter et parcourt 44184 cases pour un chemin final de 335 cases.

4 Difficultés rencontrées

L'organisation du début du projet a été assez compliqué pour moi. L'emploi du temps de la faculté condensait plusieurs CC la semaine de fin du mi-parcours, et ajouté au fait que le langage Julia était totalement nouveau pour moi il m'a été compliquer d'obtenir même le premier algorithme.

Julia ne dispose à ma connaissance pas d'un IDE qui lui est propre, j'ai donc codé avec un plugin Julia sur un IDE de JetBrains, que j'ai utilisé pour des projets antérieurs, mais l'instabilité de l'extension Julia (plantages) ralentissait fréquemment le développement.

J'ai néanmoins aimé faire ce projet, car cela m'a permis de découvrir le langage de programmation Julia, et de réaliser seul un projet d'informatique dont je suis fier.

5 Voies d'améliorations

Bien que les algorithmes de Dijkstra et A star parcourent moins de case que l'algorithme de flood fill (respectivement 127838, 107370 et 44184), ce dernier s'exécute en moins de temps que les deux autres. Le résultat de l'algorithme flood fill est moins pertinent que ceux des autres car il ne prend pas en compte les coûts de mouvements, mais peut-être les deux autres algorithmes présentent-ils des implémentations redondantes de structures de données (nodes_dict et true_path_data, deux dictionnaires pouvant peut-être être "fusionnés" en un), qui pourraient en conséquent être optimisées.

On pourrait aussi améliorer le code en retournant une structure contenant les informations passées à la fonction "print_solution"

6 Sources

Documents fournis, et

https://fr.wikipedia.org/wiki/Algorithme_A*

https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

https://fr.wikipedia.org/wiki/Algorithme_de_remplissage_par_diffusion