Université Picardie jules vernes
UFR des sciences
Département informatique

Rapport d'implémentation

Algorithme de A*

Réalisé par :

Massila LAKAF

Mohamed amine RAHMOUNI

1. L'algorithme Dijkstra avec plusieurs sorties :

Introduction

L'algorithme initial applique la méthode de Dijkstra pour trouver le chemin le plus court dans un labyrinthe. Il fonctionne correctement lorsque le labyrinthe possède une seule entrée et une seule sortie. Cependant, si plusieurs sorties sont possibles, il ne garantit pas forcément de trouver la plus proche. L'objectif de cette partie est d'analyser ce problème et de proposer une amélioration.

Analyse de l'algorithme actuel

L'algorithme commence par lire le fichier contenant la description du labyrinthe, puis construit une représentation sous forme de graphe. Il applique ensuite l'algorithme de Dijkstra en utilisant une file de priorité pour explorer les chemins possibles, en partant du point d'entrée et en avançant jusqu'à atteindre la sortie définie. Une fois la sortie trouvée, il reconstruit le chemin en remontant les nœuds et affiche le résultat.

Le problème est qu'il ne prend en compte qu'une seule sortie, celle définie dans le fichier. Si une autre sortie plus proche existe, l'algorithme ne l'explorera pas et ne choisira donc pas forcément le chemin optimal.

Proposition d'amélioration

Pour gérer plusieurs sorties, une modification est nécessaire. Plutôt que de s'arrêter dès qu'une sortie est atteinte, l'algorithme doit comparer toutes les sorties possibles et sélectionner la plus proche.

Une solution consiste à :

1. Ajouter toutes les sorties dans une liste :

```
# Extraire les point de sortie
point2 = [tuple(map(int, ligne.strip().split())) for ligne in lignes[2].strip().split(";")]
```

2. L'algorithme continue d'explorer toutes les sorties possibles et retient la plus proche :

```
# Vérifier si on a atteint une sortie
if current_node.position in exits:
    if best_exit is None or current_node.dist < best_end_node.dist:
        best_exit = current_node.position
        best_end_node = current_node
```

3. On reconstruit le chemin depuis la sortie la plus proche :

```
path = get_path(end_node)
```

4. On modifie le fichier maze.txt pour qu'il contienne plusieurs sorties :

```
5 5
4 0
0 4; 0 2; 0 3
1 1 0 0 0
1 0 0 1 0
1 1 0 0 0
0 0 0 0 1
0 1 1 1 1
```

5. Résultat :

```
[1, 1, 5, 0, 0]
[1, 0, 5, 1, 0]
[1, 1, 5, 0, 0]
[0, 5, 0, 0, 1]
[5, 1, 1, 1, 1]
nombre de sommets explorés:  14
```

Le chemin est indique par le numéro 5.

Cette modification permettrait d'assurer que le chemin trouvé est réellement le plus court, quel que soit le nombre de sorties.

Conclusion

L'algorithme initial fonctionne bien, mais il est limité lorsqu'il y a plusieurs sorties. En modifiant son comportement pour qu'il explore toutes les issues possibles et choisisse la plus proche, on améliore son efficacité et on s'assure qu'il trouve toujours le chemin optimal.

2. Algorithme A* :

- L'algorithme A* trouve toujours le chemin le plus court pour sortir ?

Pour que A* trouve toujours le chemin optimal, l'heuristique doit respecter deux propriétés essentielles :

a) L'heuristique doit être admissible

Une heuristique est admissible si elle ne surestime jamais le coût réel pour atteindre l'objectif ; cela signifie que l'heuristique sous-estime ou évalue correctement la distance restante. Grâce à cela, A* ne risque pas d'ignorer un bon chemin en croyant qu'il est trop coûteux.

b) L'heuristique doit être monotone

Une heuristique est monotone si le coût estimé ne diminue jamais lorsqu'on avance dans le graphe, évitant ainsi des réévaluations inutiles. Une heuristique monotone assure que *le premier chemin trouvé par A est forcément optimal*, car aucun autre chemin découvert plus tard ne pourra être meilleur.

Correspondance entre l'explication et l'algorithme

1. **Création des listes ouverte et fermée**
    a. L'algorithme stocke les nœuds à explorer dans une liste ouverte et les nœuds déjà explorés dans une liste fermée.
    b. Cela permet d'éviter de réexplorer inutilement des chemins déjà analysés, ce qui est cohérent avec l'idée que *A* ne revient pas sur un chemin déjà prouvé optimal* (grâce à l'heuristique monotone).
2. **Exploration du nœud avec le plus petit coût F(n)**

   a. À chaque itération, on prend le nœud dont la valeur F=G+H est la plus petite.
   b. Cela garantit que le premier chemin trouvé vers un nœud est le meilleur tant que H(n) est monotone.
3. **Extension des nœuds adjacents**
   a. Si un voisin a déjà été exploré (liste fermée), on l'ignore, ce qui évite de revenir inutilement en arrière.
   b. Si un voisin est nouveau, on l'ajoute à la liste ouverte et on enregistre son coût actuel, ce qui assure que chaque nœud est visité au bon moment et avec le bon coût.
   c. Si un voisin est déjà dans la liste ouverte mais qu'on trouve un chemin plus court pour l'atteindre, on met à jour ses valeurs. C'est ce qui garantit qu'on explore d'abord le chemin optimal avant d'envisager des alternatives.

▪ Mise en place de A*

Il existe plusieurs heuristiques pour estimer la distance à la sortie :

- **Manhattan** : adaptée aux déplacements strictement verticaux et horizontaux, mais inadaptée ici car notre algorithme autorise les diagonales. On le voit dans la génération des voisins, où les mouvements diagonaux sont inclus :

```
for voisins in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:
```

Par exemple, pour aller de $(2,3)$ à $(5,7)$, la distance est $|5-2|+|7-3|=3+4=7$

L'heuristique Manhattan sous-estimerait donc trop le coût du chemin.

- **Euclidienne** : mesure la distance en ligne droite et fonctionne bien, mais elle nécessite un calcul de racine carrée, ce qui alourdit les performances.
- **Octile** : similaire à Chebyshev, mais suppose un coût plus élevé pour les déplacements diagonaux, ce qui ne correspond pas à notre problème où tous les mouvements ont le même poids.
- **Chebyshev** : parfaitement adaptée, car elle correspond au coût réel d'un déplacement quand les mouvements diagonaux et orthogonaux ont le même poids.

Nous avons choisi **Chebyshev** car elle est simple, rapide à calculer et reflète exactement la manière dont notre algorithme explore le labyrinthe.

Implémentation de A* avec Chebyshev

Nous avons modifié l'algorithme de Dijkstra pour intégrer A* avec l'heuristique de **Chebyshev**. Cette modification permet de réduire le nombre de sommets explorés en donnant une direction à la recherche au lieu d'explorer uniformément comme Dijkstra.

1. Ajout de l'heuristique Chebyshev :

Voici la fonction heuristique Chebyshev qu'on a implémenté :

```
# Fonction heuristique H basée sur la distance de Chebyshev
def heuristique_chebyshev(node, end):
    return max(abs(end[0] - node.position[0]), abs(end[1] - node.position[1]))
```

2. Calcul du coût total F(u) = G(u) + H(u) :

G : coût depuis le départ, H : estimation du coût restant (heuristique), F: coût total

- g : Permet de savoir combien coûte le chemin pris.
- h : Permet de donner une estimation du coût restant.
- F = g + h : Aide à choisir le meilleur chemin possible.

Nous avons modifié la gestion des nœuds en ajoutant la fonction heuristique. Le coût total F(u) est maintenant la somme de la distance réelle parcourue (G) et de l'heuristique (H), ce qui oriente mieux la recherche :

```
new_node.dist = current_node.dist + 1  # G(u) = coût réel depuis le départ
new_node.h = heuristique_chebyshev(new_node, end)  # H(u) = heuristique Chebyshev
new_node.f = new_node.dist + new_node.h  # F(u) = G(u) + H(u)
```

3. Mise à jour de la file ouverte avec F(u) au lieu de G(u) :

*Dans Dijkstra, la priorité était donnée uniquement à la distance G. Avec A**, on utilise maintenant F(u) = G(u) + H(u), ce qui permet d'explorer en priorité les nœuds qui semblent les plus prometteurs :

```
# Regarde si pas dans la open list, y ajouter.
if not in_open_list:
    heapq.heappush(open_list, new_node)
```

4.matrice initiale et Résultat obtenu :

Matrice initiale :

```
5 5
4 0
0 4
1 1 0 0 0
1 0 0 1 0
1 1 0 0 0
0 0 0 0 1
0 1 1 1 1
```

Résultat :

```
[1, 1, 0, 0, 5]
[1, 0, 0, 1, 5]
[1, 1, 5, 5, 0]
[0, 5, 0, 0, 1]
[5, 1, 1, 1, 1]
nombre de sommets explorés:  9
```

Le chemin est indique par le numéro 5.