Fennich Zineb

Compte Rendu PartB Labyrinthe (Dijkstra et statégie A*)

Dans ce projet, nous cherchons à trouver le plus court chemin dans une carte en 2 dimensions. La carte est représentée sous la forme d'un **graphe pondéré**, où chaque case est un sommet et où les connexions (arêtes) se font entre une case et ses **8 voisines** (haut, bas, gauche, droite et diagonales). Les **poids** des arêtes sont définis en fonction du **coût** (temps) pour parcourir chaque case, tel que spécifié dans un fichier texte.

Deux algorithmes classiques de recherche de plus court chemin sont implémentés :

1. **Dijkstra**
2. **A* (A-Star)**

L'objectif est de **comparer leurs performances** sur différentes cartes et **discuter** des circonstances où l'un ou l'autre est préférable.

1. Description du projet

- **Lecture de la carte** :
  Un fichier texte (ex. `graph.txt`) décrit :
  1. Le nombre de lignes (`nlines`) et de colonnes (`ncols`).
  2. Les différents types de cases possibles, chacun associé à un **coût** et une **couleur** pour l'affichage.
  3. Les lignes du « dessin » de la carte : chaque caractère représente un type de case.
  4. Les coordonnées du **point de départ** et du **point d'arrivée** (start, finish).
- **Création du graphe** :
  On utilise la classe `WeightedGraph.Graph` pour stocker une liste de sommets (`vertexlist`). Chaque sommet possède :
  1. Un identifiant (`num`) qui correspond au numéro de la case (la case (i,j)est typiquement numérotée i×ncols+j).
  2. Un coût individuel (`indivTime`) qui représente le temps pour parcourir cette case.
  3. Des propriétés nécessaires aux algorithmes (distance courante `timeFromSource`, prédécesseur `prev`, etc.).
- **Connexité 8 directions** :
  Chaque case est reliée à ses 8 voisines (lorsqu'elles existent dans les limites de la carte). Pour le **poids** de l'arête entre deux cases A et B, on peut, par exemple, prendre **(tA+tB)/2** si on veut la moyenne des coûts individuels. Les instructions signalent qu'il faut prêter attention aux **poids diagonaux** (souvent plus grands que les poids horizontaux/verticaux).

- **Algorithme Dijkstra** :
  On l'implémente dans la méthode `Dijkstra(...)`.
  Cet algorithme parcourt le graphe en calculant la distance minimale depuis le nœud de départ jusqu'aux autres nœuds. Il stocke dans une structure de données (une `PriorityQueue` en Java) le sommet à visiter ayant la plus petite distance courante.
  À chaque itération :
  1. On extrait le sommet le plus proche (distance minimale).
  2. On met à jour les distances de ses voisins si l'on découvre un chemin plus court.
  3. On répète jusqu'à ce que la destination soit atteinte ou que tous les nœuds accessibles soient visités.
- **Algorithme A*** :
  Implémenté dans la méthode `AStar(...)`.
  C'est une variante de Dijkstra **informée** par une **heuristique**.
  1. La distance courante g(n) est additionnée d'une estimation du coût restant h(n) (ex. la distance euclidienne vers la destination).
  2. La file de priorité choisit alors le sommet ayant le **plus petit** f(n)=g(n)+h(n).
  3. Les voisins sont explorés en fonction de ce critère.

L'avantage d'A* réside dans le fait qu'il **réduit** souvent la zone de recherche en privilégiant les sommets plus proches de la destination d'après l'heuristique. Toutefois, cela ne fonctionne bien que si l'heuristique est **admissible** (c'est-à-dire qu'elle ne surestime jamais le coût réel pour atteindre la destination).

- **Comparaison** :
  On compare généralement Dijkstra et A* sur :
  - Le **nombre de nœuds explorés**.
  - Le **temps d'exécution** (indirectement lié, surtout si la taille de la carte est importante).
  - La **qualité** du chemin (les deux doivent trouver un chemin optimal, à condition que l'heuristique d'A* soit admissible).

```
Choisissez l'algorithme à utiliser :
1. Dijkstra
2. A*
Entrez le numéro de l'algorithme (1 ou 2) : 1
Destination trouvée avec Dijkstra!
Done! Using Dijkstra:
    Number of nodes explored: 4154
    Total time of the path: 340.0
Chemin écrit dans out.txt
```

```
Choisissez l'algorithme à utiliser :
1. Dijkstra
2. A*
Entrez le numéro de l'algorithme (1 ou 2) : 2
Destination trouvée avec A*!
Done! Using A*:
    Number of nodes explored: 4123
    Total time of the path: 341.0
Chemin écrit dans out.txt
```

Les résultats montrent que, sur cette carte en particulier, **A*** et **Dijkstra** aboutissent à des chemins presque équivalents (340.0 contre 341.0) en termes de coût total, mais avec un léger écart quant au **nombre de sommets explorés** (A* : 4123, Dijkstra : 4154). Cela illustre qu'A* peut réduire le nombre de nœuds visités grâce à son **heuristique** (ici, la distance euclidienne), mais que la **qualité** (et donc le coût total du chemin) est quasiment identique lorsque l'heuristique n'est pas trop sous-estimée. En pratique, on constate souvent qu'A* est plus performant que Dijkstra en termes de nœuds explorés, tout en conservant le même chemin optimal, dès lors que l'heuristique est admissible.U**ne heuristique admissible** signifie qu'elle ne surestime jamais le coût réel restant pour atteindre la destination. Par conséquent, lors de la sélection du prochain nœud à explorer, A* se montre plus « avisé » que Dijkstra : au lieu de considérer uniquement la distance déjà parcourue (comme Dijkstra), il ajoute une estimation du coût restant. Cela oriente la recherche plus directement vers la cible, évitant dans bien des cas l'exploration exhaustive de zones éloignées de l'objectif. Dès lors, A* explore généralement **moins de nœuds** que Dijkstra, tout en conservant la **même optimalité** du chemin si l'heuristique est effectivement admissible.

- **Affichage** :
  Le code affiche la carte, la progression de l'algorithme (les sommets explorés), et finalement le chemin trouvé. Chaque type de case est dessiné avec sa couleur (ex. bleu, vert, etc.). Pendant l'exécution, un `Thread.sleep(10)` dans la boucle montre visuellement l'algorithme parcourir la carte.

2. Implémentation

- **Lecture du fichier** :
  Dans la méthode `main`, on lit :
  - Les métadonnées (`nlines`, `ncols`).
  - Les types de terrains et leurs couleurs.
  - Les lignes décrivant la carte (chaque caractère = un type).
  - Les positions de départ et d'arrivée.

- **Construction du graphe** :
  À chaque caractère lu, on crée un sommet (vertex) avec le `indivTime` approprié. On numérote les sommets de **0** à **(ncols*nlines - 1)**.
  Ensuite, on connecte chaque sommet à ses voisins (jusqu'à 8 voisins), en calculant le poids (éventuellement la somme ou la moyenne des coûts).
- **Dijkstra** :
  La méthode `Dijkstra` applique l'algorithme de Dijkstra classique :
  - `resetGraph(graph)` pour remettre tous les `timeFromSource` à l'infini, etc.
  - Mettre la distance du départ à **0**.
  - Utiliser une `PriorityQueue<WeightedGraph.Vertex>` triée par `timeFromSource`.
  - Extraire le sommet de distance minimale, mettre à jour ses voisins, etc.
- **A*** :
  La méthode `AStar` est similaire, mais utilise `f = g + h` (où `g` est `timeFromSource` et `h` est `calculateHeuristic(...)`). On prend la **distance euclidienne**.
  - La `PriorityQueue` est triée par `v.f`, c'est-à-dire la somme `timeFromSource + heuristic`.
  - L'algorithme s'arrête quand on atteint la destination ou quand la file est vide.
- **Affichage** :
  La classe `Board` gère l'affichage. Elle dessine chaque case avec la couleur associée, ainsi que le chemin final. Les appels à `board.update(graph, current)` permettent de montrer en direct l'exploration.

3. Analyse de complexité

Dijkstra

- Dans le pire des cas (tous les sommets sont visités), la complexité de Dijkstra avec un **tas** (ou `PriorityQueue`) est en général: **O((V+E)logV)**
  - V est le nombre de sommets (ici, **nlines×ncolsnlines**).
  - E est le nombre d'arêtes (au plus 8xV pour la connexité 8).
  - Sur une grille, on peut estimer E≈8VE, donc la complexité est à peu près **O(V logV)**.

A*

- A* a la **même** borne supérieure en théorie **O((V+E) logV)**.

- **En pratique**, l'heuristique va souvent réduire drastiquement la portion du graphe explorée, rendant l'algorithme **plus rapide** que Dijkstra si l'heuristique est **admissible** et **cohérente** (ex. distance euclidienne dans une grille).
- Si l'heuristique est **mauvaise** (par exemple, surestime beaucoup ou n'est pas admissible), A* peut perdre son avantage et explorer presque autant que Dijkstra.

4. Heuristique utilisée

- Dans le code, la fonction `calculateHeuristic(int current, int end, int ncols)` applique la **distance euclidienne :**

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- Cette heuristique est **admissible** dans le cas d'une grille où le coût de déplacement est proportionnel à la distance, car elle ne surestime pas le coût réel pour aller à la destination (en partant du principe que se déplacer d'une case à une autre correspond environ à une unité de distance, ou à $\sqrt{2}$ pour la diagonale).
- Les **performances** d'A* seront souvent **meilleures** qu'avec Dijkstra, puisque la file de priorité tend à prioriser les cases proches de la destination.