Fennich Zineb

Compte-rendu : Mini Projet Prisonnier LabyrinthEscape avec A*

1) Présentation Générale

Le projet **LabyrinthEscape** consiste à résoudre un labyrinthe dans lequel un prisonnier doit rejoindre la sortie (**S**) avant d'être rattrapé par le feu (**F**). Les murs (**#**) bloquent aussi bien la progression du prisonnier que celle du feu.

Le programme se décompose en **trois grandes parties** :

1. **Lecture des grilles** (labyrinthes) depuis un fichier `input.txt`.
2. **Interface graphique** (*LabyrinthGUI* et *LabyrinthPanel*) pour :
    - Afficher la grille (murs, feu, départ, sortie…).
    - Lancer la résolution en appuyant sur un bouton « Résoudre ».
    - Visualiser le chemin trouvé en **bleu**, si le prisonnier peut s'échapper.
3. **Algorithme de résolution** (*LabyrinthEscape*), qui combine :
    - Un **BFS multi-source** pour déterminer à quel instant le feu atteint chaque case.
    - Un **A* (A-star) modifié** pour trouver une route sécurisée du départ vers la sortie, en évitant de se faire rattraper par les flammes.

2) Composants Importants

2.1. Lecture et Interface Graphique

- **LabyrinthGUI**
    - Lit *T* labyrinthes dans un fichier `input.txt` (pour chacun, on récupère **`N`** (nombre de lignes) et **`M`** (nombre de colonnes)).
    - Stocke ces labyrinthes dans une liste (`List<char[][]>`).
    - Affiche un **labyrinthe à la fois** et propose des boutons « Précédent », « Suivant » pour naviguer entre les différentes grilles.
    - Un bouton « Resoudre » appelle la fonction d'**A*** (dans la classe *LabyrinthEscape*). S'il existe un chemin permettant au prisonnier d'atteindre la sortie avant le feu, celui-ci est tracé en **bleu** dans la grille.
- **LabyrinthPanel**
    - Reçoit la grille (un tableau 2D de `char`) et la dessine en couleurs :
        - **noir** pour les murs `#`,
        - **rouge** pour le feu `F`,
        - **cyan** pour le départ `D`,
        - **jaune** pour la sortie `S`,
        - **vert** pour le vide.

    - Peut afficher un **chemin** (liste de `Node`) en **bleu** quand la résolution trouve une solution.
    - Gère une **légende** à droite pour rappeler la signification des couleurs.

Cette interface, construite en Java (Swing), permet donc de **visualiser** à la fois le labyrinthe, la propagation du feu et le chemin de fuite trouvé par l'algorithme.

2.2. Classe Principale *LabyrinthEscape*

- **computeFireTime(...)**
    - Réalise un **BFS multi-source** : on place initialement dans la file **toutes** les positions du feu (`F`) avec un temps = `0`, et on propage ce temps aux cases voisines.
    - Lorsqu'une case (x,y) est atteinte par le feu au temps *t*, les voisins sont atteints au temps *t+1* (sauf murs ou limites).
    - On enregistre ces temps dans `fireTime[x][y]`, ce qui permettra à A* d'éviter de se déplacer dans une case au moment où le feu l'atteint (ou plus tard).
- **aStarWithFire(...)**
    - Implémente un **A*** classique, mais avec la contrainte de ne pas arriver dans une case (nx,ny) au temps *gCost+1* si *gCost+1 ≥ fireTime[nx][ny]*.
        - En d'autres termes, on **évite** d'entrer dans une case en feu ou sur le point de l'être au même instant.
    - On utilise une `PriorityQueue<Node>` où chaque nœud contient :
        - `gCost` : le temps parcouru depuis le départ,
        - `hCost` : l'**heuristique**, ici la **distance de Manhattan** $|x - x_{end}| + |y - y_{end}|$
        - `fCost() = gCost + hCost`.
    - Tant qu'il existe des nœuds à explorer dans la PriorityQueue, on prend celui dont le `fCost` est minimal et on vérifie ses voisins (haut, bas, gauche, droite).
    - Si l'on parvient à la position de la sortie (**S**), on remonte la chaîne des parents pour reconstruire le chemin.

Cette combinaison BFS (pour le feu) + A* (pour le prisonnier) fait en sorte que le prisonnier n'emprunte **que** des cases où il arrive strictement avant le feu.

2.3. Classe *Node*

- Stocke les **coordonnées** (x,y) d'une case, un `gCost`, un `hCost`, et un `parent`.
- Compare deux `Node` via la méthode `compareTo(...)`, selon `fCost() = gCost + hCost`.
- Permet ainsi la gestion automatique de la file de priorité (la case la plus prometteuse est explorée en premier).

3) Analyse de l'Algorithme A*

3.1. Principe de l'A*

L'algorithme **A*** est un **Dijkstra informé** par une heuristique.

- **Open set** (PriorityQueue) : on y place le nœud de départ, puis on y insère tout nœud pour lequel on découvre un meilleur chemin.
- **Heuristique (hCost)** : On a choisi la **distance de Manhattan**, $|x - x_{end}| + |y - y_{end}|$. Cette heuristique est *admissible* quand on se déplace en 4 directions orthogonales (pas de diagonale), car elle ne surestime pas le coût réel pour atteindre la sortie.
- **Optimalité** : si l'heuristique ne surestime jamais, A* **trouvera** un chemin optimal (s'il existe).

3.2. Modification : contrainte avec le feu

Pour **éviter** le feu, la condition **tentativeG<fireTime[nx][ny]** assure que l'on ne puisse pas passer par une case (nx,ny) au moment où le feu l'occupe (ou plus tôt). Cela rend l'algorithme plus **sélectif**. Même si un nœud semblait court en distance pure, il est éliminé s'il se trouve en proie aux flammes.

4) Distance Manhattan

Dans une **grille** où les **déplacements ne sont autorisés que dans les quatre directions orthogonales** (haut, bas, gauche, droite), la **distance de Manhattan** est généralement mieux que la distance euclidienne.

Rappel : Distance de Manhattan vs. Distance Euclidienne

- **Distance de Manhattan**
  $$d_{\text{Manhattan}}((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

  Elle correspond au plus court chemin lorsqu'on ne peut se déplacer **que** dans des directions orthogonales (pas de diagonale).

- **Distance Euclidienne**
  $$d_{\text{Euclidienne}}((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
  Elle représente la distance « à vol d'oiseau », plus adaptée quand on peut se déplacer librement dans toutes les directions (y compris en diagonale).

Pourquoi la distance Manhattan dans ce projet ?

1. **Admissibilité de l'heuristique**
    - Pour que l'algorithme *A** garantisse un **chemin optimal**, l'**heuristique** h(n) doit être **admissible**, c'est-à-dire **ne jamais surestimer** la distance réelle qui reste à parcourir.
    - Si le prisonnier peut **seulement** se déplacer vers le haut, le bas, la gauche ou la droite, alors la **distance réelle** pour aller de (x1,y1) à (x2,y2) ne peut être **inférieure** à la distance de Manhattan. Au contraire, une distance euclidienne (à vol d'oiseau) pourrait être **trop optimiste** (puisqu'elle supposerait la

possibilité de se déplacer en diagonale), et donc risquerait de **surestimer** l'avantage de certains trajets.

2. **Conformité aux mouvements autorisés**
   - Dans une grille 4 directions, le **coût** de passer d'une case à une autre correspond typiquement à 1 mouvement orthogonal.
   - La distance de Manhattan **colle** parfaitement à ce modèle : elle additionne simplement les différences en x et en y.
   - La distance euclidienne, en revanche, suppose qu'on puisse couper en diagonale (ou qu'un déplacement de (x1,y1) à (x2,y2) puisse se faire suivant un segment droit), ce qui n'est **pas** autorisé ici.
3. **Risques de non-admissibilité**
   - Avec la distance euclidienne, on pourrait avoir
     $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < |x_1 - x_2| + |y_1 - y_2|$
   - Du coup, si A^* s'appuie dessus, l'heuristique peut **sous-estimer** la vraie distance orthogonale (ou, plus problématique, pourrait la surestimer dans certains scénarios de coûts de passage).
   - Tout écart rend l'heuristique potentiellement **non admissible** pour un déplacement strictement à 4 directions.

5) Complexité

4.1. BFS du feu

Le **BFS multi-source** (`computeFireTime`) parcourt chaque cellule au plus **une fois** pour y définir son temps d'arrivée du feu. Sur une grille de taille N×MN \times MN×M, cela donne une **complexité O(NxM).**

4.2. A* modifié

- Dans le pire cas, A* peut insérer dans la `PriorityQueue` chaque cellule (et parfois plusieurs fois si le `gCost` s'améliore).
- Les opérations d'insertion et d'extraction dans une `PriorityQueue` coûtent O(log(V)), où V est le nombre de sommets en mémoire.
- Sur une grille N×M, la borne standard est donc **O((NxM) x log(NxM))** en pire cas.

En pratique, si l'heuristique est pertinente (distance de Manhattan dans un labyrinthe en 4 directions), A* **réduit fortement** la zone explorée, évitant d'examiner les zones trop éloignées.