Compte Rendu du Projet *Labyrinthe A* et Propagation du Feu*

BEN SLAMA Sana

28 décembre 2024

1 Introduction

Dans ce mini-projet, nous avons pour objectif de déterminer si un prisonnier peut s'échapper d'un labyrinthe en évitant la propagation d'un feu. Le problème se décline de la façon suivante :

- La grille du labyrinthe est constituée de cases libres, de murs, d'une position de départ du prisonnier (`D`), d'une position de sortie (`S`) et de positions en feu (`F`).
- Le feu se propage dans les directions orthogonales (haut, bas, gauche, droite).
- Le prisonnier se déplace également en quatre directions orthogonales, à raison d'un pas par unité de temps.
- On souhaite savoir si le prisonnier peut atteindre la sortie **avant** que le feu n'investisse la même case.

Nous avons choisi d'employer principalement deux idées :

1. Une **propagation du feu** en utilisant un algorithme de type *BFS*.
2. Une **recherche de chemin** pour le prisonnier en utilisant l'algorithme **A*** avec l'**heuristique de Manhattan**.

2 Propagation du feu

2.1 Algorithme utilisé

Pour la propagation du feu, on a opté pour un **BFS**. L'utilisation de BFS pour propager le feu garantit que chaque cellule est visitée dans l'ordre exact de son accès par les flammes. Cela est essentiel pour une synchronisation correcte entre la propagation du feu et les déplacements du prisonnier.
Concrètement, toutes les cases en feu initial (`F`) sont placées dans une file `Queue` au temps $t = 0$. On exécute ensuite une recherche en largeur classique pour déterminer le temps t auquel chaque case libre est atteinte par le feu.

- **Initialisation** :

- On crée un tableau `fireTime` de même dimension que le labyrinthe, initialisé à `Integer.MAX_VALUE` (sous-entendu : le feu n'y est pas encore arrivé).
- Pour chaque case marquée `F`, on met `fireTime[row][col] = 0` et on l'enfile dans la `Queue`.

— **Propagation** :
- On dépile une position en feu, disons (r, c), associée à un temps t.
- Toutes les cases voisines (N, S, E, O) libres (*non murées*) sont affectées à l'instant $t + 1$ (si cela les améliore, c'est-à-dire si $t + 1$ est inférieur à la valeur précédente de `fireTime`).
- On poursuit jusqu'à épuisement de la `Queue`.

Cette approche permet de connaître pour chaque case le **temps d'arrivée du feu** de façon synchronisée, avec une complexité en $O(N \times M)$ pour une grille de $N \times M$.

2.2 Structure de données utilisée

Nous avons choisi une `Queue` (ou `LinkedList`) standard pour implémenter le BFS. Toutes les cases incendiées initialement constituent les sources. Cela permet d'avoir un **BFS** en une seule passe. Cette structure est particulièrement adaptée pour un parcourt en largeur, car l'insertion et l'extraction s'effectuent en temps constant amorti.

3 Recherche de chemin pour le prisonnier (A*)

3.1 Pourquoi A*

A* est l'un des algorithmes les plus adaptés pour résoudre des problèmes de recherche de chemin dans des graphes ou des grilles, grâce à sa combinaison d'une recherche exhaustive (comme BFS) et d'une heuristique guidée.Cela en fait un choix optimal pour les labyrinthes où la recherche exhaustive complète serait trop coûteuse.

3.2 Choix de l'heuristique : Manhattan

Dans le cadre de ce projet, l'heuristique de Manhattan est particulièrement efficace car elle exploite les déplacements orthogonaux permis dans le labyrinthe. Elle garantit une exploration ciblée des chemins les plus prometteurs tout en réduisant le nombre de nœuds explorés inutilement. Ici, le **prisonnier se déplace uniquement en 4 directions** (orthogonales). L'heuristique la plus appropriée pour ce type de déplacement est la distance **Manhattan**, définie par :

$$h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|.$$

Elle sous-estime toujours le coût réel pour atteindre la sortie, car en l'absence de murs, c'est précisément le nombre minimal de pas orthogonaux à effectuer. Ainsi, elle est **admissible** et **cohérente**, ce qui garantit qu'A* trouvera la solution optimale.

3.3 Algorithme A*

Nous avons une priority queue (`PriorityQueue` en Java) ordonnée selon $f(n) = g(n) + h(n)$:

— $g(n)$ = nombre de pas déjà effectués (distance parcourue depuis le départ `D`).

— $h(n)$ = distance de Manhattan jusqu'à la sortie `S`.

Nous extrayons en priorité la position dont le score $f(n)$ est minimal, et nous propageons aux voisins (N, S, E, O).

3.4 Comparaison avec le feu

Afin d'éviter que le prisonnier n'arrive sur une case en même temps (ou après) que le feu, nous appliquons la condition :

$$\text{if } g(n+1) < \texttt{fireTime}[r][c] \text{ alors le prisonnier peut avancer.}$$

Si cette condition n'est pas satisfaite, la case est considérée comme inaccessible à temps.

4 Structure de données et amélioration de la complexité

4.1 Structures de données

— **Tableau 2D** pour stocker la grille (`char[][]`).

— **Tableau 2D d'entiers** `fireTime` pour mémoriser le temps d'arrivée du feu dans chaque cellule.

— **Tableau 2D d'entiers** `dist` pour enregistrer le coût (nombre de pas) déjà effectué pour atteindre chaque cellule par le prisonnier.

— **Queue** pour le BFS multi-source du feu.

— **PriorityQueue** (tas binaire) pour l'algorithme A*, ce qui nous permet de traiter en priorité les positions les plus prometteuses selon $f(n)$.

Les structures de données utilisées jouent un rôle essentiel dans l'efficacité globale du programme. Le tableau fireTime mémorise les temps d'arrivée du feu pour chaque cellule, garantissant une gestion synchrone et cohérente. La PriorityQueue optimise la sélection des mouvements les plus prometteurs pour A*. En résumé, cette combinaison permet de gérer efficacement les

deux aspects critiques du problème : la propagation rapide du feu et les déplacements guidés du prisonnier. En pratique, la structure de PriorityQueue réduit le temps d'exploration en triant dynamiquement les chemins viables selon leur priorité.

4.2 Complexité

- **Propagation du feu (BFS)** : $O(N \times M)$ dans le pire des cas, où $N \times M$ est la taille de la grille. On ne visite chaque case qu'une seule fois en améliorant ses valeurs de temps d'arrivée.
- **A*** :
  - Dans le pire des cas, A* peut se comporter comme un BFS sur l'ensemble des cases, mais **l'heuristique Manhattan** réduit considérablement le nombre de cases explorées si la grille est ouverte.
  - La structure `PriorityQueue` en Java opère en $O(\log V)$ pour l'insertion et l'extraction, où V représente le nombre de sommets (cases) en attente. Concrètement, la complexité haute reste $O(V \log V)$, mais pratiquement, on a un gain réel dû à l'heuristique (moins de chemins explorés).

Cette combinaison BFS + A* + heuristique admissible nous permet de gagner beaucoup de temps de calcul comparé à un pur BFS, car nous guidons la recherche du chemin avec l'heuristique de Manhattan.

5 Conclusion

Grâce à l'algorithme de **propagation du feu** par BFS et à la **recherche A*** (avec **Manhattan**), on peut rapidement déterminer si le prisonnier peut ou non échapper à l'incendie.

- **L'heuristique de Manhattan** est particulièrement adaptée au déplacement en 4 directions.
- Le **BFS multi-source** pour le feu calcule le temps d'arrivée sur chaque case efficacement.
- L'utilisation d'une **PriorityQueue** dans A* diminue le nombre de nœuds visités de façon significative, en se focalisant sur les chemins les plus prometteurs.

En conclusion, pour un problème de taille raisonnable, la solution A* avec Manhattan et le BFS pour la propagation du feu a permis de déterminer efficacement si le prisonnier peut sortir à temps ou non.