

TP1 — Robotique : Résoudre un labyrinthe avec A*

Quang-Trung Luu

quang-trung.luu@universite-paris-saclay.fr

Gif-sur-Yvette, le 6 février 2026

1 Objectifs pédagogiques

Ce TP a pour objectif d'introduire les étudiants à la résolution de problèmes de planification de trajectoire en environnement discret. À travers la modélisation d'un labyrinthe et l'implémentation des algorithmes Dijkstra et A*, les étudiants apprendront à représenter un environnement sous forme de grille 2D, à gérer des contraintes de déplacement et à calculer un chemin optimal entre un point de départ et un point d'arrivée.

2 Pré-requis et consignes

- Langage : Python 3
- Travail : binôme (sauf indication contraire).
- Le code doit être propre, commenté, et testable.

3 Algorithme Dijkstra

L'algorithme de Dijkstra est un algorithme glouton permettant de calculer les plus courts chemins à partir d'un sommet source vers tous les autres sommets d'un graphe pondéré dont les poids des arêtes sont non négatifs. À chaque étape, l'algorithme sélectionne le sommet ayant la distance provisoire minimale, puis met à jour (relaxe) les distances de ses voisins. En répétant ce processus, il garantit l'obtention des plus courts chemins optimaux. Le pseudocode de l'algorithme de Dijkstra est donné dans l'Algorithme 1.

Algorithme 1 : Algorithme Dijkstra.

Input : Graph $G = (V, E)$, source s

Output : Arrays $\text{dist}[\cdot]$, $\text{prev}[\cdot]$

1 **Function** Dijkstra(G, s) :

2 **foreach** $v \in V$ **do**
3 $\text{dist}[v] \leftarrow \infty$
4 $\text{prev}[v] \leftarrow \text{UNDEFINED}$

5 $Q \leftarrow V;$
6 $\text{dist}[s] \leftarrow 0;$
7 **while** $Q \neq \emptyset$ **do**
8 $u \leftarrow \arg \min_{x \in Q} \text{dist}[x];$
9 $Q \leftarrow Q \setminus \{u\};$
10 **foreach** $(u, v) \in E$ **do**
11 $\text{alt} \leftarrow \text{dist}[u] + w(u, v);$
12 **if** $\text{alt} < \text{dist}[v]$ **then**
13 $\text{dist}[v] \leftarrow \text{alt};$
14 $\text{prev}[v] \leftarrow u;$

15 **return** $\text{dist}[\cdot], \text{prev}[\cdot];$

4 Algorithme A*

L'algorithme A* est un algorithme de recherche de chemin largement utilisé en robotique mobile, en intelligence artificielle et en planification de trajectoire. Il permet de trouver un chemin optimal entre un point de départ et un point d'arrivée dans un environnement discret ou continu, tout en limitant le nombre de nœuds explorés. A* peut être vu comme une extension informée de l'algorithme de Dijkstra, dans laquelle une heuristique guide la recherche vers la cible.

Le principe fondamental de A* repose sur l'évaluation de chaque position candidate à l'aide d'une fonction de coût

$$f(n) = g(n) + h(n) \quad (1)$$

Le terme $g(n)$ représente le coût réel du chemin depuis le point de départ jusqu'au nœud, tandis que $h(n)$ est une estimation du coût restant pour atteindre l'objectif. Dans un labyrinthe représenté sous forme de grille, deux distances sont principalement employées : la distance de Manhattan et la distance euclidienne.

- La distance de Manhattan correspond au nombre minimal de déplacements nécessaires lorsque seuls les mouvements horizontaux et verticaux sont autorisés. Pour une cellule (x, y) et la cellule d'arrivée (x_g, y_g) , elle est définie par

$$h_{\text{Manhattan}}(x, y) = |x - x_g| + |y - y_g| \quad (2)$$

Cette heuristique est particulièrement adaptée aux grilles à voisinage 4-connecté et ne surestime jamais le coût réel pour atteindre l'objectif.

- La distance euclidienne représente la distance géométrique directe entre deux cellules et est donnée

par

$$h_{\text{Euclidienne}}(x, y) = \sqrt{(x - x_g)^2 + (y - y_g)^2} \quad (3)$$

Elle est plus appropriée lorsque des déplacements diagonaux ou continus sont autorisés, mais fournit une estimation moins précise dans le cas d'une grille strictement 4-connectée.

Le pseudocode de l'algorithme de A* est donné dans l'Algorithme 2

Algorithme 2 : Algorithme A*

```

Input : Graph  $G = (V, E)$ , start  $s$ , goal  $t$ , heuristic  $h(\cdot)$ 
Output : Shortest path (via  $\text{prev}[\cdot]$ ) and cost  $\text{g}[t]$  (if reachable)

1 Function AStar( $G, s, t, h$ ) :
2   foreach  $v \in V$  do
3      $\text{g}[v] \leftarrow \infty$ 
4      $\text{prev}[v] \leftarrow \text{UNDEFINED}$ 
5    $\text{g}[s] \leftarrow 0$ 
6    $Q \leftarrow \{s\}$  // open set
7   while  $Q \neq \emptyset$  do
8      $u \leftarrow \arg \min_{x \in Q} (\text{g}[x] + h(x))$ 
9     if  $u = t$  then
10      return  $\text{g}[t], \text{prev}[\cdot]$ 
11     $Q \leftarrow Q \setminus \{u\}$ 
12    foreach  $(u, v) \in E$  do
13       $\text{alt} \leftarrow \text{g}[u] + w(u, v)$ 
14      if  $\text{alt} < \text{g}[v]$  then
15         $\text{g}[v] \leftarrow \text{alt}$ 
16         $\text{prev}[v] \leftarrow u$ 
17        if  $v \notin Q$  then
18           $Q \leftarrow Q \cup \{v\}$ 
19    return  $\infty, \text{prev}[\cdot]$  // goal unreachable
  
```

Dans ce TP, les déplacements étant limités aux **quatre directions** principales, la distance de Manhattan sera utilisée comme heuristique pour l'algorithme A*, afin de guider efficacement la recherche tout en garantissant l'obtention d'un chemin optimal.

5 Représentation du labyrinthe (maze)

Le labyrinthe est représenté par une grille bidimensionnelle de taille $H \times W$. Chaque cellule de la grille correspond à une position potentielle du robot. Une cellule peut être soit franchissable, soit bloquée par un obstacle. Vous pouvez choisir un encodage simple, par exemple :

- 0 : cellule libre (movable area)
- 1 : obstacle (mur / non franchissable)

La reward map est une matrice séparée (même taille) :

- reward négative pour encourager les chemins courts (ex : -1 par pas)
- reward positive pour certains bonus (ex : +5)
- reward finale à l'arrivée (ex : +100)

6 Travail demandé

Partie A : Écrire la classe Maze

Une classe `Maze` devra être implémenter permettant de créer et de manipuler un labyrinthe de taille arbitraire. Cette classe devra stocker les dimensions du labyrinthe, la grille représentant les obstacles, la matrice de récompense, ainsi que les coordonnées du point de départ et du point d'arrivée.

La classe devra également fournir des méthodes permettant de vérifier si une cellule appartient bien à la grille, si elle est franchissable, et d'identifier les cellules voisines accessibles depuis une position donnée. Dans ce TP, on considérera un voisinage à quatre directions, correspondant aux déplacements vers le haut, le bas, la gauche et la droite.

Partie B : Fonctions de génération (movable areas, obstacles, reward)

Des fonctions devront être écrites afin de générer les zones franchissables et les obstacles du labyrinthe. Les obstacles pourront être placés de manière aléatoire ou déterministe, mais les cellules correspondant au point de départ et au point d'arrivée devront impérativement rester franchissables.

La matrice de récompense devra être initialisée de manière cohérente. Un choix classique consiste à appliquer une pénalité uniforme à chaque déplacement, par exemple une valeur négative constante, afin d'encourager les chemins courts. Des bonus peuvent être ajoutés sur certaines cellules, et une récompense significative devra être définie pour la cellule d'arrivée.

Partie C : Résolution avec A* : solve()

Une méthode `solve()` devra être implémentée dans la classe `Maze`. Cette méthode devra retourner la liste ordonnée des cellules constituant le chemin optimal entre le point de départ et le point d'arrivée. Si aucun chemin n'existe, la méthode devra retourner `None`. L'algorithme devra utiliser une file de priorité pour gérer les cellules à explorer, mémoriser les meilleurs coûts trouvés et reconstruire le chemin final à partir des relations de parenté entre cellules.

7 Travaux et Questions

1. Tester un labyrinthe sans obstacle, un labyrinthe comportant des obstacles simples, ainsi qu'un cas où aucun chemin n'est possible. Il faudra également vérifier que les contraintes sur les cellules de départ et d'arrivée sont toujours respectées.
2. En extension, les étudiants pourront proposer l'ajout de déplacements diagonaux avec un coût adapté, une visualisation graphique du labyrinthe et du chemin.
3. Faire une comparaison entre Dijkstra et A*.
4. Testez les deux algorithmes avec des poids négatifs. Commentez les résultats.

5. Quelles sont les différences entre les deux algorithmes ? Lequel est le plus performant ? Quelle est la complexité de chacun des deux algorithmes ?
6. Listez les scénarios dans lesquels A* ne peut pas faire mieux que Dijkstra. Dans quel cas A* devient-il strictement équivalent à Dijkstra ?
7. Si l'heuristique $h(n)$ est parfaite (c'est-à-dire exactement égale au coût réel restant), combien de sommets A* explore-t-il ?

Références

1. De Kai. Lecture 10 : Dijkstra's Shortest Path Algorithm. Hong Kong University of Science and Technology (HKUST). [Online]. Available : <https://www.cse.ust.hk/~dekai/271/notes/L10/L10.pdf>
2. Siyang Chen. The A* Search Algorithm. [Online]. Available : https://courses.cs.duke.edu/fall11/cps149s/notes/a_star.pdf