ROB316 - TP3 : Path planning using A*

Dajing Gu

Décembre 2020

1 Introduction

1.1 Résumé du Cours

La planification du chemin est de chercher un chemin entre la position courant et la destination (bien présenter suite de nœuds et de liens ou trajectoire), qui est constituée de trois parties : la planification dans une carte, la discrétisation des espaces de recherche et la recherche de chemin dans un graphe.

La conception du **plan** est de présenter la suite de nœuds et de liens ou trajectoire à suivre. Il aussi décide dans le plan l'action associée à chaque nœud ou à chaque position.

Afin de réaliser la **discrétisation des espaces**, on a tout d'abord besoin d'obtenir l'**espace des configurations** (positions parcourues par le robot) en considérant la forme du robot et les contraintes dans les cartes topologiques ou métriques. Ensuite on discrétise cette espace en cellules (régulières, trees) ou chemins (diagramme de Voronoi, treillis avec échantillonage aléaoire).

Enfin on fait **la recherche de chemin dans un graphe** sur le graphe discrétisé déjà obtenu. Il y a beaucoup d'algorithmes, qui estiment le coût des chemins possibles et choisissent ce dont le coût est le moindre. À la base de **Dijkstra** qui s'occupe du cas de poids et **Wavefront**, qui permet la parallélisation, **A*** est créé, dont la partie heuristique permet de ne parcourir qu'une partie des états. Et finalement, **D* Lite**, qui permet la replanification en environnement dynamique, est présenté.

1.2 Introduction du TP

En utilisant python, on analyse l'influence de paramètres différents dans A*: l'influence heuristique, l'influence d'environnement et à la fin l'influence de poids de chaque noeud.

2 L'influence heuristique

Dans cette section, on étudie l'influence heuristique. En gardant la carte, les positions de *start* et *end* par défaut, on change seulement le paramètre de `heuristic_weight`.

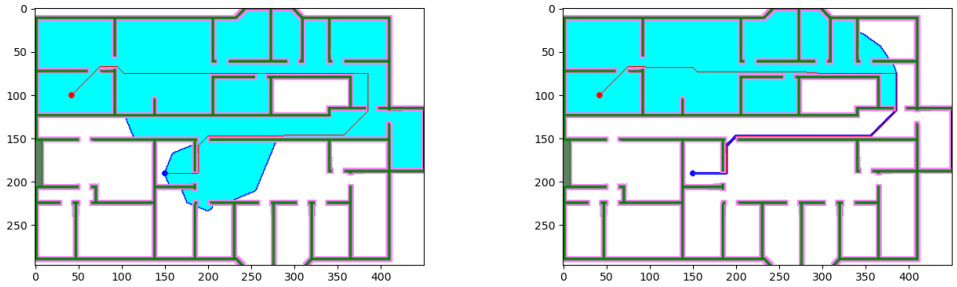
2.1 Question 1

Quand le paramètre `heuristic_weight` = 0, l'algorithme de A* ne considère que la distance entre le point actuel *current* et le point de départ *start*, cela est identique à l'algorithme **Dijkstra**. Le résultat de différents `heuristic_weight` est représenté dans la table 2.1

`heuistic_weight`	Temps exécution (s)	Longuer chemin
0.0	2.29	685.97
1.0	1.55	685.97

Table 2.1: Résultat pour `heuistic_weight` = 0.0 et 1.0

Selon la table 2.1 et la figure 2.1, on trouve que les longueurs de chemin calculées sont égaux. L'avantage pour `heuistic_weight` = 0 est que il peut toujours trouver le chemin le plus court. Mis le temps d'exécution pour `heuistic_weight` = 0.0 est plus grande, qui est un désvantage. Ceci peut être expliqué par la plus grande zone de recherche.



(a) `heuistic_weight` = 0.0 (b) `heuistic_weight` = 1.0

Figure 2.1: Résultat pour `heuistic_weight` = 0.0 et 1.0

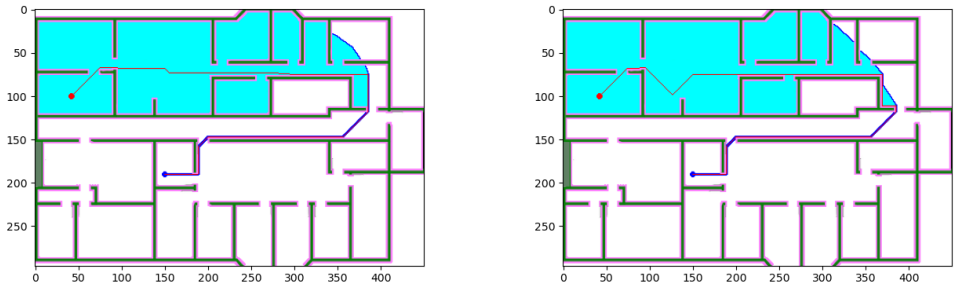
2.2 Question 2

Cette fois le paramètre `heuristic_weight` = 5, et l'algorithme de A* devient plus proche à l'algorithme d'optimal greedy (ou *Best First Search*). Il fait plus de attention sur le point de destination.

`heuistic_weight`	Temps exécution (s)	Longuer chemin
1.0	1.55	685.97
5.0	1.55	705.1

Table 2.2: Résultat pour `heuistic_weight` = 0.0 et 1.0

L'avantage pour `heuistic_weight` = 5.0 en théorie est qu'il est plus rapide de trouver un chemin, même si la table 2.2 et la figure 2.2 présente un cas au contraire. Le désavantage est que le chemin le plus court n'est pas trouvé puisque la longueur du chemin calculé est plus grande.



(a) `heuistic_weight` = 1.0 (b) `heuistic_weight` = 5.0

Figure 2.2: Résultat pour `heuistic_weight` = 1.0 et 5.0

La raison de ce phénomène est présenté dans les deux formulas codé en python. Cette fois l'algorithme fait trop si beaucoup d'attention sur le point de destination (la distance entre le point actuel et le point de destination, la distance entre le point départ et le point de destination) que il néglige le coût pour se déplace du point actuel *current* au point *next*. Cette méthode permet de trouver un chemin rapidement, mais la caractéristique de le plus court ne peut pas être assurée.

```
frontier[self.start] = heuristic_weight * math.dist(self.goal, self.start)
priority = new_cost + heuristic_weight * self.heuristic(self.goal, next)
```

3 Influence de l'environnement

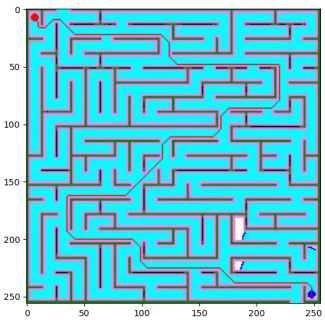
3.1 Question 3

Dans cette section on va comparer la performance de l'algorithme avec `heuistic_weight` = 1.0 et `heuistic_weight` = 0.0 en utilisant des cartes différentes.

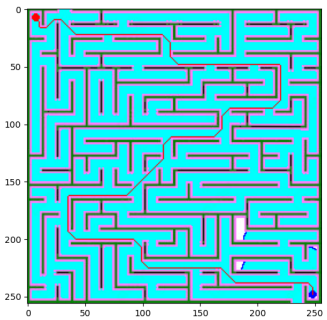
Dans la carte 'labyrinthe', il y a beaucoup d'obstacles. Le temps d'exécution et la zone de recherche sont presque la même pour tous les deux valeurs.

`heuistic_weight`	Temps exécution (s)	Longuer chemin
0.0	1.04	785.75
1.0	1.10	785.75

Table 3.1: Résultat pour `heuistic_weight` = 0.0 et 1.0 dans la carte 'labyrinthe'



(a) `heuistic_weight` = 0.0



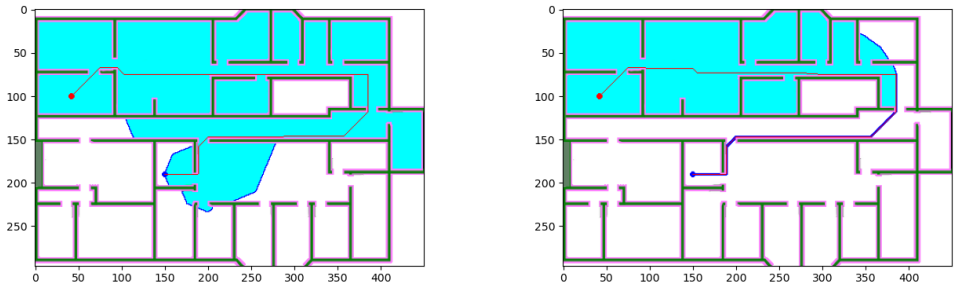
(b) `heuistic_weight` = 1.0

Figure 3.1: Résultat pour `heuistic_weight` = 0.0 et 1.0 dans la carte 'labyrinthe'

Dans la carte 'office', il y a des obstacles mais pas trop. comme on a déjà analysé dans la question 2, Le temps d'exécution et la zone de recherche pour `heuistic_weight` = 1.0 sont un peu plus petits que ceux de `heuistic_weight` = 0.0.

`heuistic_weight`	Temps exécution (s)	Longuer chemin
0.0	2.29	685.97
1.0	1.55	685.97

Table 3.2: Résultat pour `heuistic_weight` = 0.0 et 1.0 dans la carte 'office'



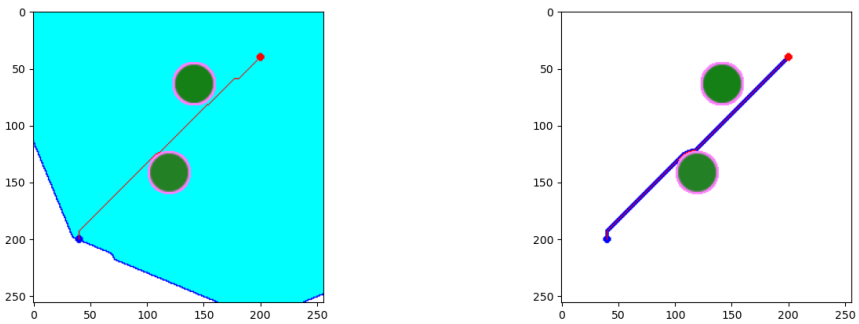
(a) `heuistic_weight` = 0.0 (b) `heuistic_weight` = 1.0

Figure 3.2: Résultat pour `heuistic_weight` = 0.0 et 1.0 dans la carte 'office'

Dans la carte 'freespace', il y a seulement quelque obstacles. Cette fois, Le temps d'exécution et la zone de recherche pour `heuistic_weight` = 1.0 sont beaucoup plus petits que ceux de `heuistic_weight` = 0.0.

`heuistic_weight`	Temps exécution (s)	Longuer chemin
0.0	3.72	230.37
1.0	8.02×10^{-2}	230.37

Table 3.3: Résultat pour `heuistic_weight` = 0.0 et 1.0 dans la carte 'freespace'



(a) `heuistic_weight` = 0.0 (b) `heuistic_weight` = 1.0

Figure 3.3: Résultat pour `heuistic_weight` = 0.0 et 1.0 dans la carte 'freespace'

En fonction des figures et tables présentés dans cette section, on peut déduire que dans l'environnement où il y a seulement quelques obstacles, l'algorithme A* avec `heuistic_weight` = 1.0 se comporte le meilleur.

4 Les noeuds avec poids

Par défaut, il n'y pas de poids sur les noeuds. Tous les noeuds sont les mêmes et le meilleur chemin va passer à proximité d'obstacles et travers des passages étroits.

4.1 Question 4

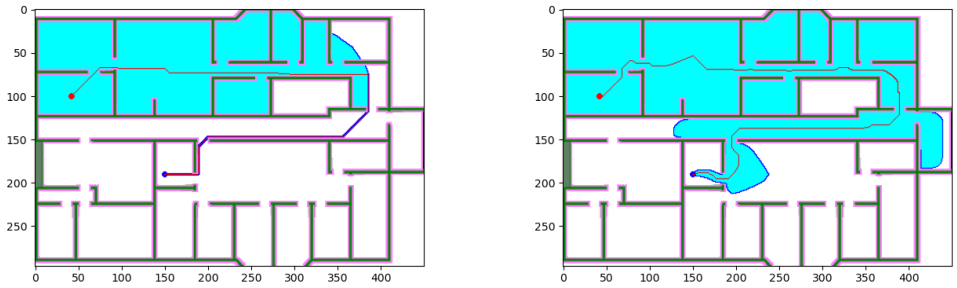
Dans cette section on va ajouter le poids sur chaque noeud pour que le chemin ne passe pas très près des obstacles. La fonction de `cv2.distanceTramsform(map, cv2.DIST_L2, 3)` est utilisé pour calculer la distance à l'obstacle le plus proche pour chaque position libre de la carte.

```
self.distance = cv2.DistanceTransform(self.map, cv2.DIST_L2, 3)
self.dist_weight = 60/(self.distance + 0.00001)
```

On a ajouté deux lignes dans la fonction d'initialisation de class `pathfind` pour calculer la distance et le poids pour chaque noeud. La distance est sur le dénominateur car on a besoin de s'éloigner l'obstacle. Plus le point est proche d'obstacle, plus grande la valeur le poids. Une déviation de 0.0001 est ajouté à la distance pour éviter la division par zéro. On a aussi défini un nouveau coût:

```
new_cost = cost_so_far[current] + math.dist(current, next) + self.dist_weight[next]
```

Le résultat est présenté dans la figure 4.1. Même si plus de zone est recherchée et même si le chemin trouvé n'est plus le plus court, le chemin du robot a réussi d'éviter les obstacles.



(a) le cas sans poids

(b) le cas avec poids

Figure 4.1: Comparaison avec le résultat avec et sans poids