ROB316 - TP4 : Planification de trajectoire en utilisant RRT

Dajing Gu

Décembre 2020

1 Introduction

1.1 Résumé du Cours

Il y a deux niveaux de planification, **la planification locale** et celle de globale. On a déjà étudié la planification globale dans TP3, qui stocke les informations environnementales dans une carte et utilise cette carte pour trouver un chemin réalisable. Mais cela ne convient pas dans des environnements inconnus. Dans ce TP, on va étudier la planification de trajectoire locale, qui ne prend en compte que les informations d'environnement instantanées du robot, qui nous aide à réduire le temps de calcul.

La planification de trajectoire est constitué de quatre parties: l'évitement d'obstacles, la planification réactive, la recherche de chemin stochastique et la exploration. **Vector Field Histogram**, **Fenêtre dynamique** et champs de potentiels peuvent être utilisés pour réaliser l'évitement d'obstacles. L'algorithme **BUG** est utilisé dans la carte inconnue afin de finir la planification réactive. Dans la recherche de chemin stochastique, l'algorithme **RRT** et ses variantes sont utilisés. Et l'exploration d'un environnement inconnu peut être fait par l'algorithme **A***.

1.2 Introduction du TP

En utilisant python, on analyse l'influence de paramètres différents dans `RRT`: le nombre d'itération, la taille du pas, et le type d'environnement.

2 RRT vs RRT*

Dans cette section, on garde la carte, les positions de *start* et *end* par défaut et on fait des expériences pour comparer l'algorithme de RRT de RRT*.

Question 1

On a testé les deux algorithmes `RRT` et `RRT*` en modifiant le nombre maximal d'itération.

En théorie, RRT s'arrete quand il trouve une trajectoire, mais RRT* va continuer sans cesse jusqu'à le nombre maximal d'itération pour trouver une trajectoire optimale. Le résultat de différents valeurs est représenté dans la table 2.1 et 2.2. Une cycle d'itération 50 sur RRT est utilisé pour éviter les erreurs stochastiques. ET sur RRT*, on a fait plusieurs expériences.

Selon la table 2.1 et la table 2.2, on trouve que quand le nombre maximal d'itération n'est pas assez grande, l'algorithme RRT et RRT* ne peuvent pas réussir à trouver la trajectoire et il y aura des échecs.

Pour RRT, on constate qu'il n'y pas une relation directe entre le nombre maximal d'itération et les deux autres valeurs. la longueur de trajectoire ne peut pas être amélioré par l'augmentation du nombre d'itération. Mais pour RRT*, le cas est différent. Plus le nombre d'itération est grand, plus la longueur de trajectoire est petite, qui signifie que plus d'itérations nous permet d'obtenir une trajectoire plus optimale. Le temps pour trouver la première trajectoire est changé mais le temps total de computation est toujours égale au nombre maximal d'itération.

Nombre maximal d'itération	Temps de computation	Longueur de trajectoire
100	NAN	NAN
200	NAN	NAN
500	243	49.29
1000	414	71.50
2000	476	71.64
5000	430	72.42
10000	404	70.95
20000	465	71.55
50000	438	70.21

Table 2.1: Résultat de différent nombre maximal d'itération pour RRT

Nombre maximal d'itération	Temps de computation	Longueur de trajectoire
100	NAN	NAN
200	NAN	NAN
500	500	71.94
1000	1000	70.60
2000	2000	60.20
5000	5000	59.68
10000	10000	59.15
20000	20000	56.70
50000	50000	56.09

Table 2.2: Résultat de différent nombre maximal d'itération pour RRT*

Question 2

On maintenant étudie l'influence de `step_len` sur les deux algorithme en modifiant sa valeur. Le nombre maximal d'itération est mis par défaut (10000).

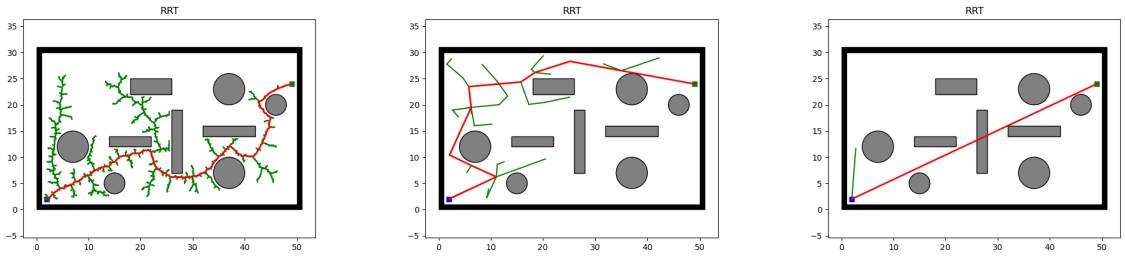
`step_len`	Temps de computation	Longueur de trajectoire
0.1	8260	68.84
0.2	4252	73.55
0.5	1575.94	71.43
1	791.12	71.93
2	409.98	71.04
5	238.12	71.69
10	172.38	72.20
20	90.56	66.75
50	4.96	51.9
100	4.89	51.89

Table 2.3: Résultat de différente `step_len` pour RRT

Selon les résultats présentés dans la table 2.3 et 2.4, on peut trouver qu'une longueur de pas trop courte ralentira la vitesse de recherche de l'algorithme, et trop longue entraînera la croissance à franchir les obstacles

Plus précisément, pour RRT, quand `step_len` est petit, ça va prendre beaucoup de temps pour trouver une trajectoire, comme présenté dans la figure 1(a). Plus `step_len` est grand, moindre le temps dont il a besoin de trouver une trajectoire. La longueur associée est aussi plus petite. Quand la valeur de `step_len` est grande, l'algorithme peut trouver rapidement une trajectoire. Pourtant, quand la valeur est trop grande,

la trajectoire trouvée n'est plus correcte car l'évitement d'obstacles n'est pas fait, comme présenté dans la figure 1(c)



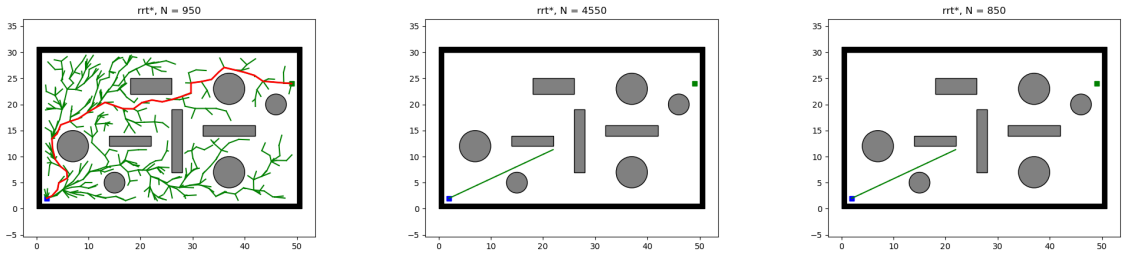
(a) `step_len` = 0.1 (b) `step_len` = 10 (c) `step_len` = 100

Figure 2.1: Résultat de `step_len` = 0.1, 10 et 100 pour RRT

La loi résumée précédemment n'est pas assez évidente dans la table 2.4 à cause de la caractère de stochastique. Car le nombre d'itération réel est toujours égale à le nombre maximal d'itération, on ne peut rien dire sur la relation entre le temps d'exécution et le `step_len`. Mais on peut quand même constater que la longueur minimale de trajectoire est obtenue quand le `step_len` n'est ni trop petit ni trop grand.

`step_len`	**Temps de computation**	**Longueur de trajectoire**
0.1	10000	64.79
0.2	10000	69.48
0.5	10000	64.25
1	10000	59.12
2	10000	59.15
5	10000	57.97
10	10000	57.60
20	10000	66.75
50	10000	56.66
100	10000	57.35

Table 2.4: Résultat de différente `step_len` pour RRT*



(a) `step_len` = 0.1 (b) `step_len` = 10 (c) `step_len` = 100

Figure 2.2: Résultat de `step_len` = 0.1, 10 et 100 pour RRT*

3 Planification dans un couloir étroit

Dans cette section, l'environnement Env2 (`environnement = env.Env2 ()`) est utilisé et tous les autre paramètres sont mis par défaut. Un couloir étroit se situe au milieu de la carte, qui rend difficile la recherche de trajectoire.

Question 3

Nombre de réussite	Temps de computation	Longueur de trajectoire
2	881	103.12

Table 3.1: Résultat de RRT dans un couloir étroit dans 50 cycles

Selon la table 3.1 et la figure 3.1, on trouve que il est beaucoup plus difficile de faire pousser l'arbre rapidement dans cet environnement et il y a beaucoup de fois d'échecs (il y a seulement 2 cas de réussite dans 50 cycles). Cela peut être expliqué par le principe de RRT. Quand l'arbre essaie de grandir vers un point aléatoire à partir du nœud le plus proche, il va juger si la ligne entre le point et le nœud le plus proche passe à travers des obstacles. Si oui, ce point aléatoire va être éliminé.

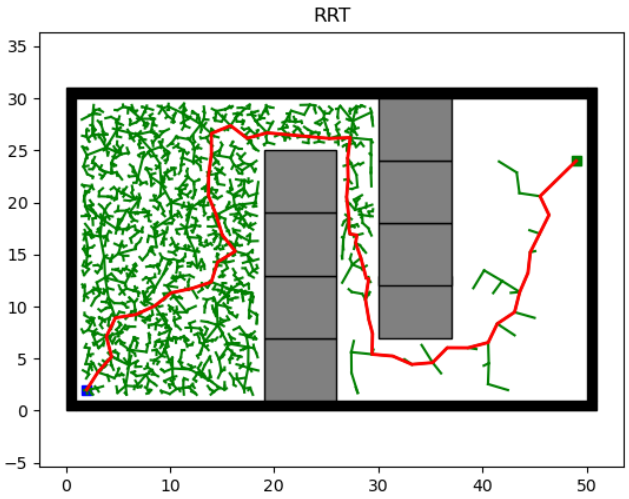


Figure 3.1: Résultat de RRT dans un couloir étroit

Dans la carte utilisée dans cette section, la largeur du couloir n'est pas assez grande en comparaison avec le `step_len`, qui nous empêche de agrandir l'arbre car les nouveaux points aléatoires sont toujours éliminés .

Question 4

Pour résoudre le problème qui se passe dans la question 3, on a modifié le code `rrt.py`. On a modifié la fonction `generate_random_node(self, goal_sample_rate)` en utilisant l'algorithme **OBRRT**. L'idée de cet algorithme est d'échantillonner des points en tenant compte des obstacles afin d'augmenter les chances que l'arbre traverse des zones difficiles. Le code associé est présenté ci-dessous.

```
1 def generate_random_node(self, goal_sample_rate):
```

```
    if np.random.random() < goal_sample_rate:
        return self.s_goal

    delta = self.utils.delta

    node = Node((np.random.uniform(self.x_range[0] + delta, self.x_range[1] - delta),
                 np.random.uniform(self.y_range[0] + delta, self.y_range[1] - delta)))
    a = np.random.random()

    delta = 8 * self.utils.delta
    if a < 0.5:
        while 1:
            id = np.random.randint(len(self.env.obs_rectangle))
            [x, y, w, h] = self.env.obs_rectangle[id]

            node_list =[Node((np.random.uniform(x - delta, x),
                              np.random.uniform(y - delta, y + h + delta))),
                        Node((np.random.uniform(x + w, x + w + delta),
                              np.random.uniform(y - delta, y + h + delta))),
                        Node((np.random.uniform(x - delta, x + w + delta),
                              np.random.uniform(y - delta, y))),
                        Node((np.random.uniform(x - delta, x + w + delta),
                              np.random.uniform(y + h, y + h + delta)))
                        ]
            node = node_list[np.random.randint(len(node_list))]

            if self.utils.is_inside_obs(node):
                break
    return node
```

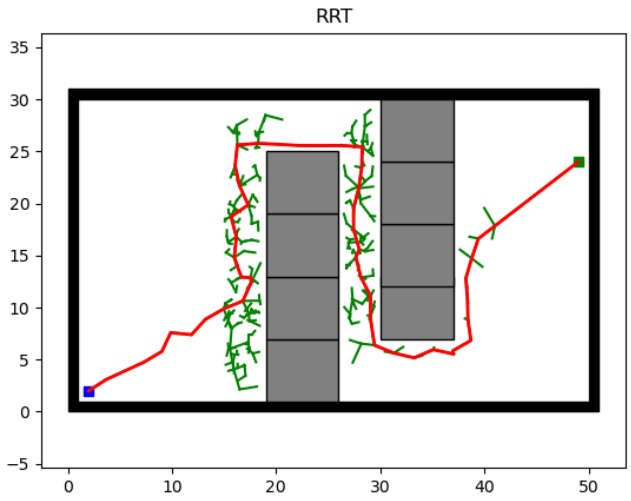


Figure 3.2: Résultat de nouveau algorithme (pourcentage = 100%)

On a testé le comportement de l'algorithme en choisissant différent pourcentage de points générés au tour d'obstacle. Quand le pourcentage égale 0, c'est le même cas comme l'algorithme original, quand le pourcentage est 1, c'est à dire que tous les points générés sont au tour d'obstacles. Selon les résultats présentés dans la table 3.2 et la figure 3.2, on trouve que plus de points sont générés au tour d'obstacles et le nombre de réussite a augmenté beaucoup en comparaison avec le cas original, qui satisfait à notre attente.

pourcentage (%)	**Nombre de réussite**	**Temps de computation**	**Longueur de trajectoire**
0	2	957	100.10
20	10	835.2	105.46
40	22	810.55	103.12
60	16	739.25	103.08
80	23	741.78	101.67
100	16	732.125	101.09

Table 3.2: Résultat de différent pourcentage pour RRT (dans 50 cycles)