Planification de trajectoire par l'algorithme RRT

David FILLIAT

2 janvier 2020

1 Introduction

Dans ce TP, nous allons étudier la méthode de planification de trajectoire appelée Rapidly Exploring Random Trees [1] et quelques unes de ses variantes. Pour cela, nous utiliserons le code MATLAB réalisé par Olzhas Adiyatov et Atakan Varol légèrement modifié, que vous téléchargerez sur ma page.

Ces programmes permettent (entre autres) d'exécuter différentes variantes de RRT pour un robot se déplaçant en 2D dans des environnements contenant des obstacles polygonaux.

2 Prise en main

Pour utiliser ce code, vous devez commencer par définir une variable contenant les paramètres du problème, c'est à dire la carte, le point de départ et le point d'arrivée. Par exemple :

```
> carte = struct('name','bench_june1.mat','start_point',[-15.5 -5.5],'goal_point',[7 -3.65]);
```

Le champ `name` doit contenir le nom d'un fichier `.mat` du répertoire `maps/`.

Ensuite, l'appel à la fonction de planification RRT se fait par :

```
> rrt(map, max_iter,  is_benchmark, rand_seed, variant)
```

avec :

- `map` : une structure du type de celle définie ci-dessus
- `max_iter` : le nombre maximum d'itérations de construction de l'arbre
- `is_benchmark` : un booléen pour activer/désactiver l'affichage
- `rand_seed` : une graine pour le générateur aléatoire afin de pouvoir relancer le code dans les mêmes conditions si besoin
- `variant` : le nom d'un fichier matlab définissant les fonctions de base de RRT (échantillonnage, calcul de distance, affichage ...) pour le problème visé. Nous travaillerons avec `FNSimple2D` qui définit un robot se déplaçant de manière holonome en 2D.

Voici un exemple d'utilisation :

```
> res=rrt(carte, 8e3,0,cputime*1000,'FNSimple2D');
```

Trois variantes de RRT sont proposées :

- `rrt` : l'algorithme de base
- `rrt_star` : une variante qui modifie localement l'arbre des positions afin que chaque point soit relié à son voisin le plus proche du départ [2].
- `rrt_star_fn` : une variante de rrt_star qui impose une limite au nombre de noeuds pour réduire la consommation de mémoire [3].

Enfin, une fonction `benchmark2D(map_name,start_point,goal_point,algo,variant,max_iter)` permet de lancer 5 exécutions d'un algorithme sur la carte décrite dans le fichier `map_name` (par exemple `'bench_june1.mat'`) pour obtenir des statistiques sur le nombre de fois ou l'algorithme échoue à trouver un chemin, la longueur moyenne des chemins découverts et le nombre d'itérations pour trouver un chemin au but. Par exemple :

```
> benchmark2D('bench_june1.mat',[-15.5 -5.5],[7 -3.65],'rrt','FNSimple2D',5000)
```

3 Question 1 - Optimalité du chemin

Dans cette section, utilisez le problème défini par :

```
> carte = struct('name','bench_june1.mat','start_point',[-15.5 -5.5],'goal_point',[7 -3.65]);
```

Testez les trois algorithmes `rrt`, `rrt_star` et `rrt_star_fn` sur ce problème en faisant varier le nombre maximal d'itérations. Que pouvez-vous constater sur les longueurs moyennes des chemins ? Sur les temps de calculs ? Quel est l'algorithme qui offre le meilleur compromis entre le temps de calcul et l'optimalité du chemin ?

4 Question 2 - Planification pour des passages étroits

Dans cette section, utilisez le problème défini par :

```
> carte = struct('name','RSE12.mat','start_point',[-18.5 -5.5],'goal_point',[18 -3.65]);
```

Testez l'algorithme `rrt` avec ces paramètres :

```
> benchmark2D('RSE12.mat',[-18.5 -5.5],[18 -3.65],'rrt','FNSimple2D',5000)
```

Que constatez vous ?

Pour remédiez à cela, modifiez le fichier `FNSimple2D_Obst.m` afin l'implémenter une variante de l'algorithme `OBRRT` [4]. Dans cet algorithme, l'idée est d'échantillonner des points en prenant en compte les obstacles afin d'augmenter les chances que l'arbre de trajectoire passe dans les zones difficiles.

Commencez par implémenter une version très simple dans laquelle vous échantillonnerez 5% des points autour des sommets des obstacles. Pour cela, vous devez modifier la fonction `sample` à la ligne 177 du fichier `FNSimple2D_Obst.m`. Vous aurez besoin des variables suivantes :

- `this.obstacle.num` : le nombre d'obstacles de la carte
- `this.obstacle.output{obs_ind}` : une matrice contenant les coordonnées des sommets de l'obstacle `obs_ind`. La matrice comporte 2 colonnes (coordonnées x et y) et une ligne par sommet.

Evaluez les performances que vous obtenez avec 5000 itérations en comparaison avec la méthode `FNSimple2D` :

```
> benchmark2D('RSE12.mat',[-18.5 -5.5],[18 -3.65],'rrt','FNSimple2D_Obst',5000)
```

En fonction du temps restant, inspirez vous de l'article [4] pour implémenter d'autres méthodes d'échantillonnage et tentez de réduire le nombre d'itérations nécessaire à l'algorithme pour trouver un chemin.

Références

[1] Steven M. Lavalle, James J. Kuffner, and Jr. Rapidly-Exploring Random Trees : Progress and Prospects. In *Algorithmic and Computational Robotics : New Directions*, pages 293–308, 2000.

[2] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *Int. J. Rob. Res.*, 30(7) :846–894, June 2011.

[3] O. Adiyatov and H. A. Varol. Rapidly-exploring random tree based memory efficient motion planning. In *2013 IEEE International Conference on Mechatronics and Automation*, pages 354–359, Aug 2013.

[4] S. Rodriguez, Xinyu Tang, Jyh-Ming Lien, and N. M. Amato. An obstacle-based rapidly-exploring random tree. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 895–900, May 2006.