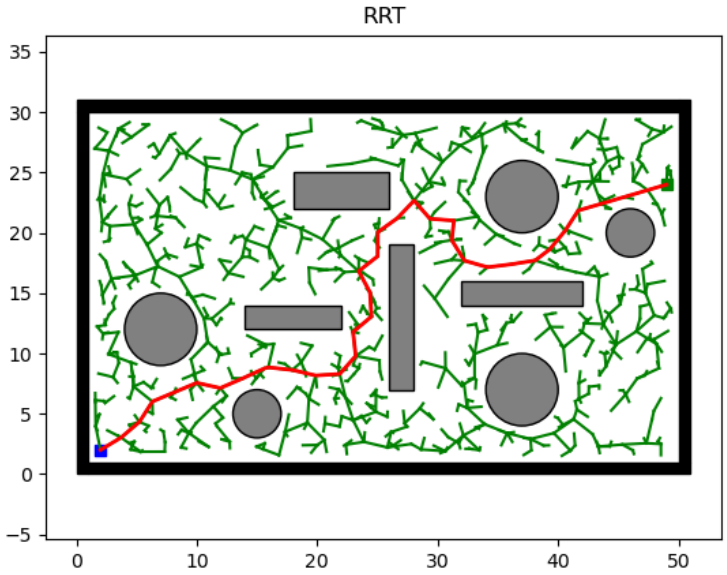
TP4 - Rapidly Exploring Random Trees

ROB316 - Planification et contrôle

Bastien HUBERT



ENSTA Paris - janvier 2023

1 RRT vs RRT* :

1.1 Question 1 :

On se place dans un premier temps dans l'environnement `env.Env()`, correspondant à une pièce rectangulaire parsemée d'obstacles de taille moyenne.

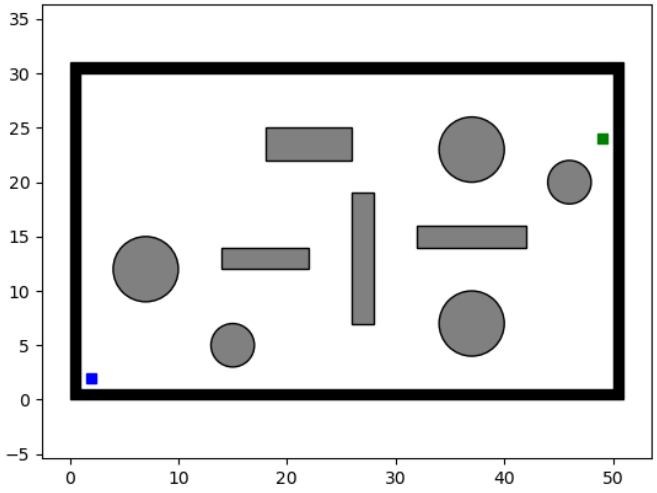


Figure 1: L'environnement `env.Env()`

Les algorithmes RRT et RRT* étant stochastiques, on observe de fortes variations dans le nombre d'itérations et la distance du chemin trouvé pour les mêmes conditions. Dans toute la suite, on fera donc une moyenne sur 10 exécutions des algorithmes.

Par exemple, pour RRT avec `iter_max = 1500`, on obtient les résultats suivants :

```
Found path in 507 iterations, length : 76.26563305363254
Found path in 363 iterations, length : 72.42004199513777
Found path in 376 iterations, length : 78.25803655326779
Found path in 283 iterations, length : 63.563162658255955
Found path in 256 iterations, length : 70.60211951348744
Found path in 434 iterations, length : 68.13456092520539
Found path in 374 iterations, length : 72.08658845825103
Found path in 240 iterations, length : 68.01881251580612
Found path in 686 iterations, length : 64.20381600687286
Found path in 534 iterations, length : 71.08146403411524
```

Le tableau ci-dessous correspond aux valeurs moyennes de `nb_iter` et `get_path_length(path)` calculées par RRT et RRT* pour différentes valeurs de `iter_max` :

	RRT		RRT*	
max_iter	iter_moy	len_moy	iter_moy	len_moy
500	316.0	70.98	312.0	66.97
1000	435.7	66.89	372.8	68.45
1500	447.8	70.35	402.5	63.07
2000	440.0	73.09	420.4	63.08
2500	422.5	71.94	442.8	61.57
3000	433.8	71.84	421.1	60.62

Table 1: Métriques moyennes de RRT et RRT* en fonction de `iter_max`

On constate que RRT n'est pas sensible aux variations de `iter_max` (tant qu'il trouve un chemin en moins de `iter_max` itérations), tandis que le chemin calculé par RRT* est de plus en plus court et nécessite de plus en plus d'itérations à mesure que `iter_max` augmente. Ces observations sont cohérentes avec la théorie, car on sait que RRT ne permet pas l'amélioration des chemins calculés, tandis que RRT* peut restructurer l'arbre calculé pour optimiser un chemin déjà trouvé.

1.2 Question 2 :

Le tableau ci-dessous correspond aux valeurs moyennes de `nb_iter` et `get_path_length(path)` calculées par RRT et RRT* pour différentes valeurs de `step_len` (`iter_max = 1500` ici) :

	RRT		RRT*	
step_len	iter_moy	len_moy	iter_moy	len_moy
0.5	1027.2	69.66	1067.0	68.32
1	905.5	72.63	657.6	70.24
1.5	628.7	70.31	613.2	66.32
2	328.1	66.14	390.1	64.27
2.5	396.2	75.15	310.1	62.43
3	373.1	72.14	326.0	61.86

Table 2: Métriques moyennes de RRT et RRT* en fonction de `step_len`

On constate que les chemins calculés par les deux algorithmes sont en moyenne de même taille pour toutes les valeurs de `step_len`, ce qui semble indiquer que ce paramètre a peu d'impact sur l'efficacité de la couverture de l'espace par l'arbre. En revanche, le nombre moyen d'itérations augmente clairement lorsque `step_len` diminue : puisque chaque point généré par l'algorithme est plus proche de son prédécesseur, il faut plus de points (donc d'itérations) pour trouver un chemin.

2 Planification dans un couloir étroit :

2.1 Question 3 :

Dans cette partie, on considère la même pièce rectangulaire que dans `env.Env()`, mais en remplaçant les obstacles éparpillés par deux murs épais formant un couloir étroit qui sépare la pièce en deux.

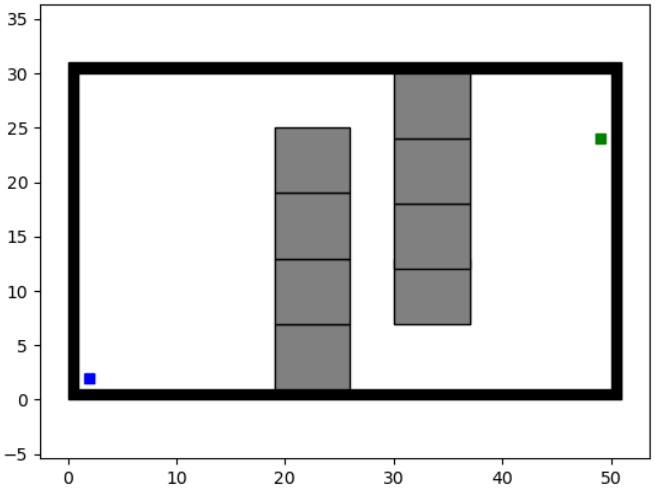


Figure 2: L'environnement `env.Env2()`

Les arbres construits par RRT et RRT* ont du mal à se propager dans le couloir en raison de la méthode d'expansion de l'arbre : en effet, le point "suivant" qui sera ajouté à l'arbre est choisi aléatoirement dans le cercle de rayon `step_len` autour du point courant. À l'intérieur du couloir, ce cercle est en très grande partie dans les murs (voire de l'autre côté si `step_len` est assez grand), ce qui invalide le tirage du nouveau point et empêche l'exploration du couloir. Ce problème est d'autant plus visible que `step_len` est grand, car une plus grande partie du cercle sera invalidé par les obstacles.

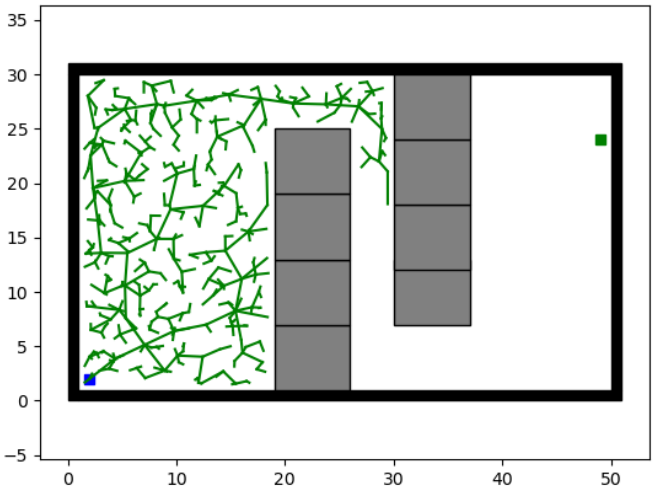


Figure 3: RRT n'arrive pas à avancer dans le couloir

2.2 Question 4 :

La nouvelle fonction `generate_random_node` est décrite ci-après :

```
def generate_random_node( self, goal_sample_rate, proportion ):
        if np.random.random() < goal_sample_rate:
            return self.s_goal

        delta = self.utils.delta

        if np.random.random() > proportion: # cas de base
            next_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 ) )
            )
        else:
            delta = 10 * delta # pour balayer plus large
            collision = True
            while ( collision ):
                # choix d'un obstacle aléatoire
                ( x, y, w, h ) = self.env.obs_rectangle[np.random.randint( len( self.env.obs_rectangle ) )]
                # choix d'un coin aléatoire de l'obstacle
                delta_x, delta_y = (int)( np.random.random() > 0.5 ), (int)( np.random.random() > 0.5 )
                x_corner, y_corner = x + delta_x * w, y + delta_y * h
                # création d'un nouveau noeud
                next_node = Node(
                    ( np.random.uniform( x_corner + delta, x_corner - delta ),
                      np.random.uniform( y_corner + delta, y_corner - delta ) )
                )
                # on recommence si il y a collision
                collision = self.utils.is_inside_obs( next_node )

        return next_node
```

Elle impose de rajouter une variable `proportion` (choisie globale) en début de programme, et de rajouter cette variable dans l'appel de `generate_random_node` dans `planning` :
`node_rand = self.generate_random_node( self.goal_sample_rate, proportion )`.

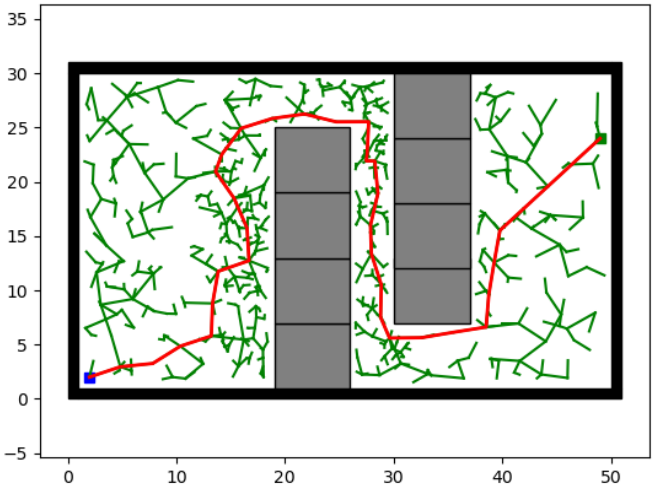


Figure 4: Un exemple d'arbre obtenu après modification de `generate_random_node`

Le tableau ci-dessous correspond aux valeurs moyennes de `nb_iter` et `get_path_length(path)` calculées par RRT pour les paramètres par défaut (`step_len = 2` et `iter_max = 1500`) dans `env.Env2`, en faisant varier `proportion`. Il présente aussi le pourcentage de chemin trouvés par l'algorithme en itérations limité :

proportion	iter_moy	len_moy	taux de réussite
0	1302.0	106.2	10%
0.1	1089.5	106.82	20%
0.25	1037.0	103.68	60%
0.5	1127.3	101.62	70%
0.75	1086.6	103.38	70%
0.9	1237.7	106.27	60%
1	942.2	105.58	50%

Table 3: Métriques moyennes de RRT et RRT∗ en fonction de `step_len`

On constate que les performances de RRT sont grandement améliorées par la nouvelle version de `generate_random_node` (la version initiale peut être retrouvée en imposant `proportion = 0`), avec un maximum de performance pour des valeurs de `proportion` entre 0.5 et 0.75. Il est logique que cette version améliore le taux de réussite de RRT car se diriger vers les coins d'un obstacle est une heuristique simple pour sortir d'un labyrinthe d'obstacles. Toutefois, se focaliser sur ces coins peut nuire à l'exploration du reste de l'espace et dégrader la qualité du chemin trouvé.