INSTITUT POLYTECHNIQUE DE PARIS
ENSTA PARIS

CSC_5RO16_TA, Planification et Contrôle

TP2, Rapidly Exploring Random Trees

by
Guilherme NUNES TROFINO

supervised by
David FILLIAT

Confidentiality Notice
Non-confidential and publishable report

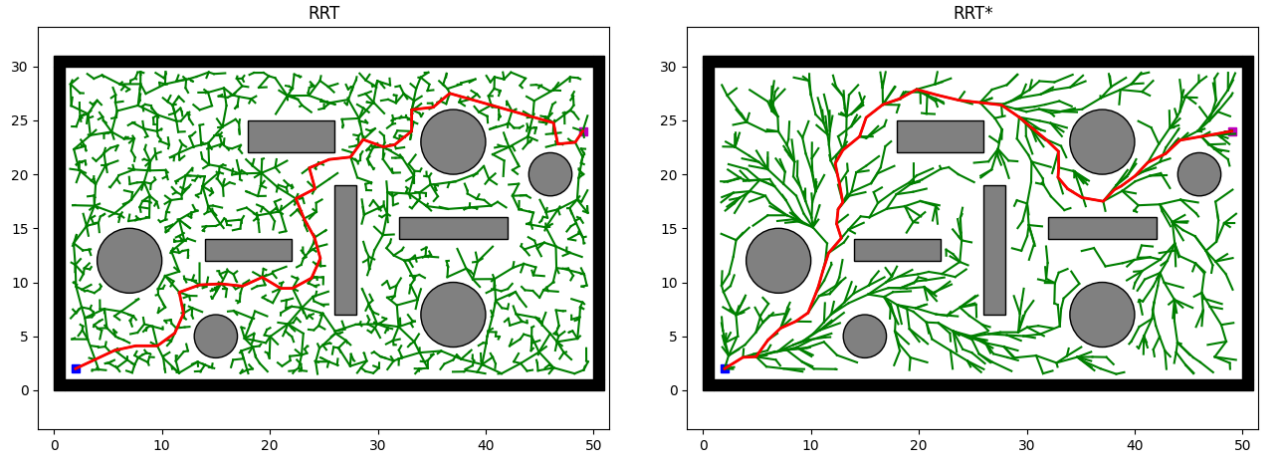
ROBOTIQUE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET COMMUNICATION

Paris, FR
23 décembre 2024

Table des matières

1. Question 1

Tout d'abbord, il est important de noter que les algorithmes RRT et RRT* sont des méthodes stochastiques. Cela implique que pour obtenir des résultats significatifs et fiables, plusieurs expérimentations doivent être réalisées. Ces répétitions permettent de lisser les variations inhérentes à la nature aléatoire de ces algorithmes. L'environment 1 est présenté ci-dessous pour chaque algorithme :



(a) algorithme RRT (b) algorithme RRT*

FIGURE 1.1 : Environment 1

Les résultats obtenus après plusieurs expériences sont synthétisés ci-dessous :

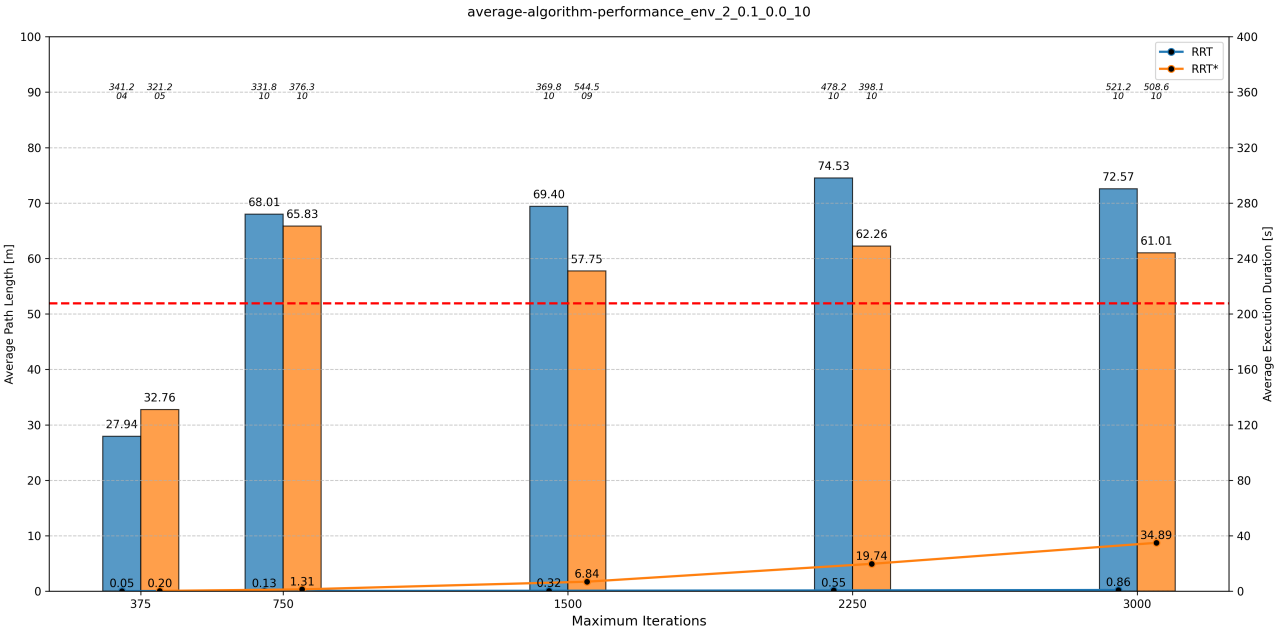


FIGURE 1.2 : Performance Moyenne des Algorithmes avec `step`=2

L'image précédente illustre les résultats obtenus pour les algorithmes RRT et RRT* exécutes sur l'environment 1 avec `step`=2, `goal_sample_rate`=0.1, `corner_sample_rate`=0.0 et `repetions`=10.

> **Remarque.** Dans ce projet, chaque image aura sur son titre les informations sur l'exécution : `average-algorithm-performance_i_j_k_l_m` :
>
> 1. `i` : `environnement` : environnement utilisé ;
> 2. `j` : `step` : pas utilisé ;
> 3. `k` : `goal_sample_rate` : goal sample rate utilisé ;
> 4. `l` : `corner_sample_rate` : corner sample rate utilisé ;
> 5. `m` : `repetions` : repetions realisées.

La ligne rouge représente le **chemin optimal théorique** entre le point de départ et le point d'arrivée, calculé en ignorant les obstacles. Cette ligne sert de référence pour évaluer la qualité des solutions générées par les algorithmes.

Même si les résultats pour `max_iterations`=375 sont en dessous de cette ligne, ce n'est pas surprenant car, parmi les 10 expériences, seulement 6 chemins ont été trouvés par RRT et 3 par RRT*.

Les chifres autour de la ligne de 90 mètres représentent respectivement la quantité moyenne d'itérations effectuées, au dessus, et la quantité de chemins trouvés, en dessous.

> **Remarque.** Les barres illustrent la longeur moyenne des chemins trouvés en mètres avec l'échelle à gauche et les lignes montrent le temps moyen de calcul en secondes avec l'échelle à droite.

Au fur et à mesure que le nombre maximale d'itéractions augmente les algorithmes trouvent de chemins plus courts en générale et le temps de calcul augmente. Il faut noter que l'algorithme RRT* trouve souvent des chemin plus courts au coût d'un temps de calcul plus significatif.

2. Question 2

Afin d'étudier l'influence des différents paramètres sur les performances des algorithmes, `step` a été fixé à 1 et 16 pour l'environnement 1. Les résultats sont présentés ci-dessous :

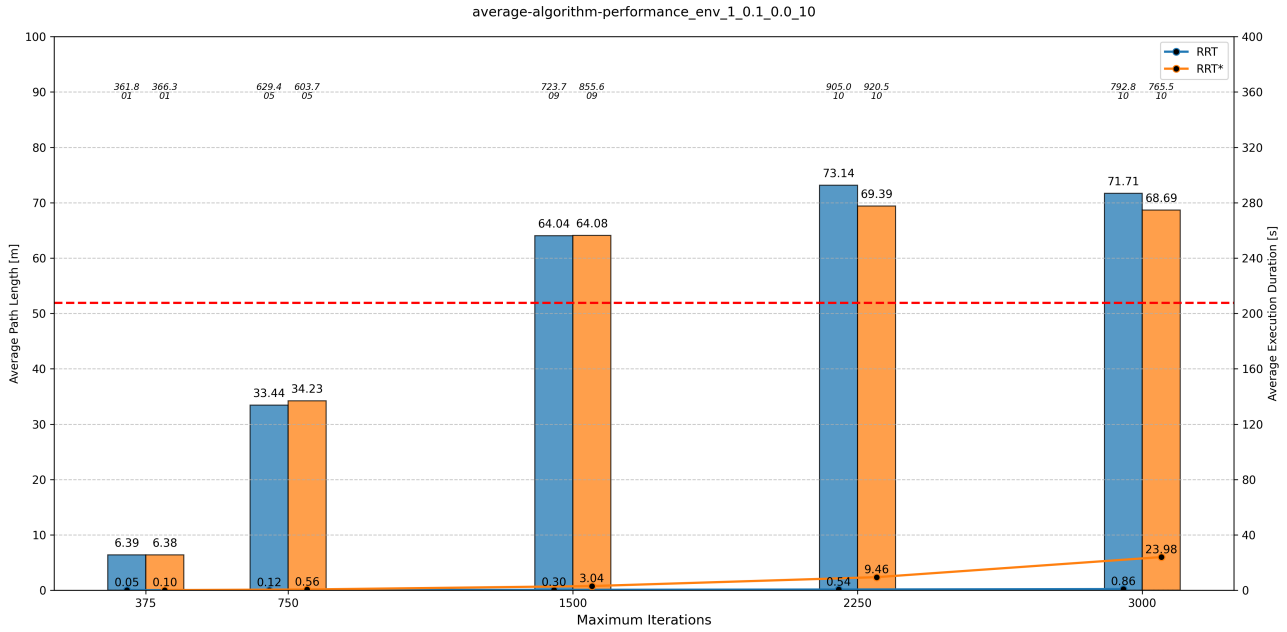


FIGURE 2.1 : Performance Moyennes des Algorithmes avec `step`=1

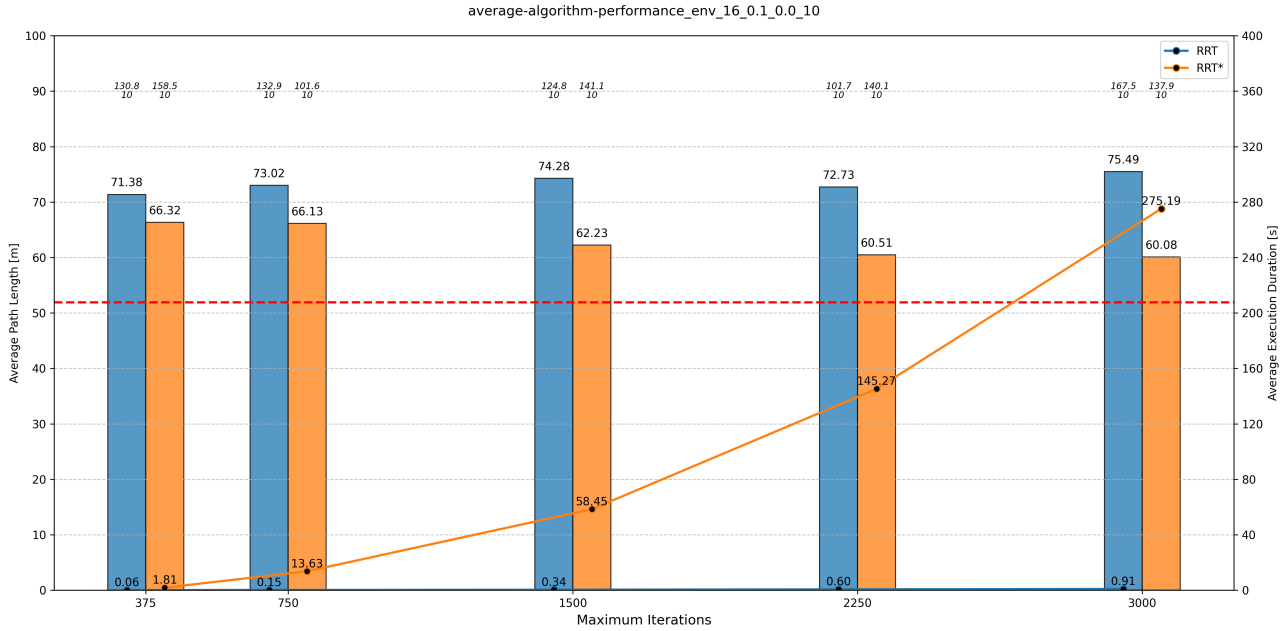


FIGURE 2.2 : Performance Moyennes des Algorithmes avec `step`=16

Lorsque le `step` est fixé à 1, les algorithmes RRT et RRT* recontrent davantage de difficultés pour trouver un chemin. En moyenne, seulement 1 chemin a été trouvé parmi les 10 expériences réalisés pour chaque

algorithme lorsque `max_iterations`=375. Ce n'est qu'à partir de `max_iterations`=2250 que les algorithmes trouvent systématiquement des chemins.

> **Remarque.** Des valeurs de `step` plus grands, comme 16, permettent aux algorithmes d'explorer une zone plus vaste avec le même nombre d'itérations. Cela augmente significativement les chances de trouver un chemin.

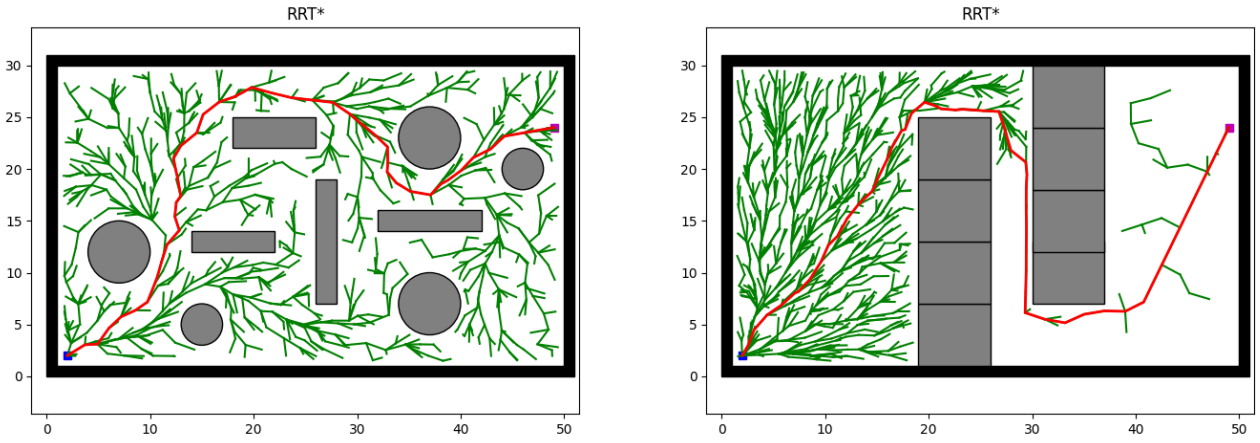
Comme décrit dans la Question 1, le comportement de l'algorithme RRT* se distingue également ici par de chemins plus courts en moyenne, mais au prix d'un temps de calcul plus élevé. Cette fois, cependant, l'augmentation de la valeur de `step` a un impact plus significatif sur le temps de calcul de l'algorithme RRT* que sur celui de l'algorithme RRT.

> **Remarque.** Des valeurs `step` plus grands, comme 16, permettent aux algorithmes d'explorer une zone plus vaste. Cela entraîne une augmentation du recalcul des chemins pour le RRT*, ce qui augmente le temps de calcul de manière significative.

3. Question 3

Lorsque les algorithmes tentent d'étendre leur arbre vers un point aléatoire à partir du nœud le plus proche, ils rencontrent des difficultés dues à la densité des obstacles. Dans un environnement où les obstacles sont nombreux et rapprochés, la probabilité de trouver un chemin libre de collision entre le nœud le plus proche et le point aléatoire est faible.

Par conséquent, les algorithmes ont tendance à se concentrer dans des zones où les obstacles sont moins denses, ce qui ralentit la croissance de l'arbre. De plus, lorsque aucun chemin libre de collision entre le nœud le plus proche et le point aléatoire, l'arbre risque de stagner. Ainsi, la croissance rapide de l'arbre devient difficile dans des environnements densément peuplés d'obstacles.



(a) environment 1 (b) environment 2

FIGURE 3.1 : Comparison des Environments

4. Question 4

Afin de proposer une alternative au problème présenté dans la question précédente, nous avons implémenté une version simplifiée de l'algorithme OBRRT. L'idée consiste à générer des points en tenant compte des obstacles pour augmenter les chances que l'arbre traverse des zones difficiles. Pour cela, la fonction `generate_random_node()` du fichier `obrrt.py` a été implémentée comme suit :

```python
def generate_random_node(self):
    if np.random.random() < self.goal_sample_rate and self.corner_sample_rate < 1 - self.goal_sample_rate:
        return self.s_goal

    if np.random.random() < self.corner_sample_rate:
        for _ in range(1000):
            x, y, w, h = self.env.obs_rectangle[np.random.randint(len(self.env.obs_rectangle))]

            node = Node((
                np.random.uniform(x - 0.25 * w, x + 1.25 * w),
                np.random.uniform(y - 0.25 * h, y + 1.25 * h)
            ))

            if not self.utils.is_inside_obs(node):
                return node

    delta = self.utils.delta

    return 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)
    ))
```

Les résultats de cette implementation, comparés aux autres méthodes, sont présentés ci-dessous pour 50 exécutions de l'algorithme avec un nombre maximum de 1500 itérations :

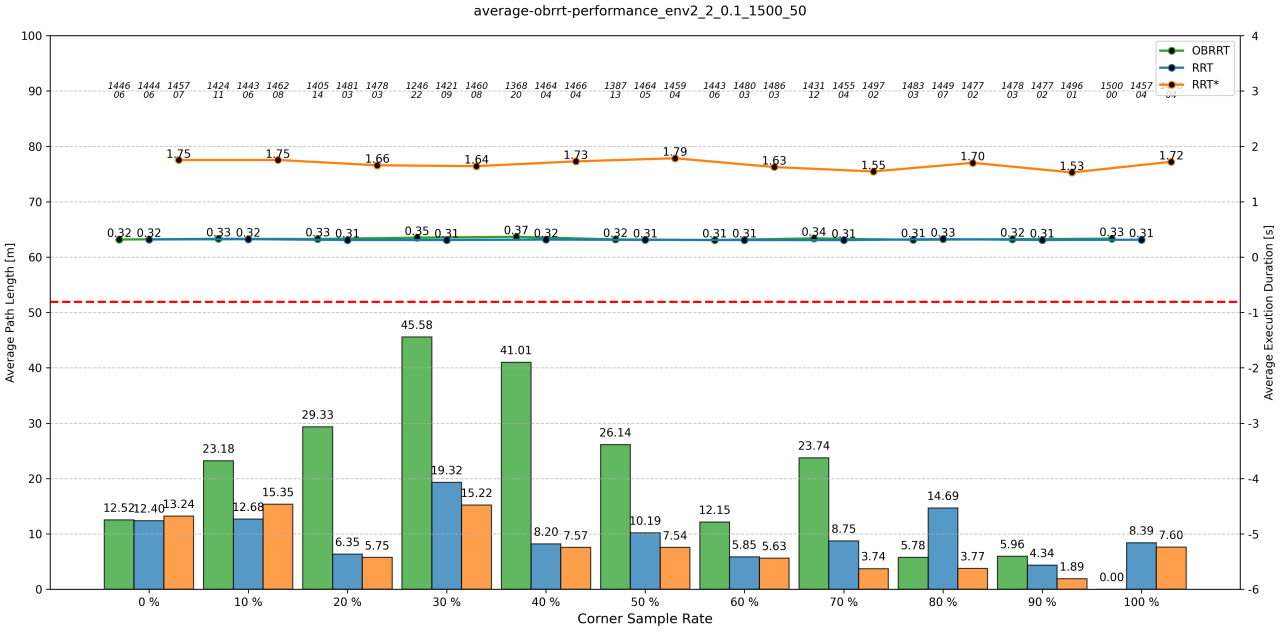
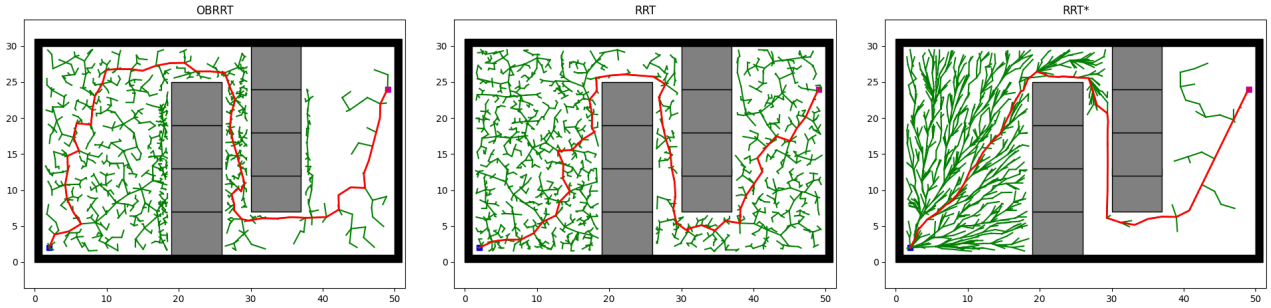


FIGURE 4.1 : Performance Moyenne des Algorithmes OBRRT, RRT et RRT*

On remarque que avec `corner_sample_rate` = 0 % il n'y a pas de différence entre les algorithmes. Cependant, avec `corner_sample_rate` = 100 % l'algorithme OBRRT ne parvient pas à trouver de chemin, car il ne peut pas relier les points d'interêt situés trop loin des obstacles et l'origine.

Ceci dit il est possible de voir que pour des valeurs intermédiaires de `corner_sample_rate` l'algorithme OBRRT a une performance supérieure aux autres algorithmes.

Des exemples de chemins trouvés par les différents algorithmes sont montrés ci-dessous :



(a) algorithme OBRRT (b) algorithme RRT (c) algorithme RRT*

FIGURE 4.2 : Environment 2, exemple de chemin

On a observe que l'algorithme OBRRT a trouvé, en plus du chemin menant à l'objectif, des chemins autour des obstacles avec un concentration assez importante.

Par ailleurs, l'algorithme RRT* a rencontré davantage de difficulté pour trouver un chemin dans cet environment. En raison de son besoin d'un plus grand nombre d'itéractions pour surmonter les obstaces, il n'a pas pu explorer efficacement l'environment au-delà de ces obstacles. Cela s'explique par la nature même de l'algorithme, qui est plus sensible à la densité des points dans l'espace de recherche.