TP2: Path planning using RRT
RO16 - Planification et control

Gallego, Natalia
AST
ENSTA Paris
natalia.gallego@ensta-paris.fr

I. Introduction

Dans ce TP, nous approfondirons la mise en œuvre et l'analyse de l'algorithme **Rapidly Scanning Random Trees (RRT)** et de sa variante améliorée, **RRT***. Ces algorithmes permettent une découverte efficace de chemins sans collision dans des environnements complexes.

La première partie de ce travail se concentre sur la comparaison des algorithmes **RRT** et **RRT*** en termes de qualité de chemin et d'efficacité de calcul en faisant varier des paramètres tels que le nombre maximum d'itérations et la taille des pas. Cela nous aidera à comprendre les compromis entre le temps de calcul et les longueurs de chemin résultantes.

La deuxième partie aborde un environnement plus difficile contenant des couloirs étroits, où l'algorithme **RRT** standard a du mal à faire pousser l'arbre efficacement en raison de l'espace confiné. Pour résoudre ce problème, nous modifierons l'implémentation RRT en incorporant une variante simple de l'algorithme **RRT** basé sur les obstacles (**OBRRT**). Cette adaptation implique d'échantillonner stratégiquement des points aux coins des obstacles pour améliorer la capacité de l'arbre à naviguer dans les zones difficiles.

Tout au long de ces travaux pratiques, nous mènerons des expériences, analyserons les résultats et modifierons la base de code fournie pour évaluer les performances des algorithmes dans différentes conditions et améliorations.

II. Question 1: Variation du nombre maximum d'itérations

A. RRT

L'algorithme **RRT** (Rapidly-Exploring Random Tree) génère des chemins dans un environnement bidimensionnel entre un point de départ et un objectif. Le paramètre itérations contrôle le nombre maximum de tentatives dont dispose l'algorithme pour explorer et connecter des nœuds dans l'espace de recherche. La modification de cette valeur affecte les performances de l'algorithme :

Lorsque la valeur des itérations est faible (250, 500), l'arbre a moins de possibilités de s'étendre, ce qui peut avoir pour conséquence qu'aucun chemin viable vers l'objectif ne soit trouvé dans des environnements complexes. Comme le montrent les résultats du tableau I, dans plusieurs exécutions, aucun chemin n'a été trouvé, reflétant des itérations insuffisantes pour combler l'écart. D'un autre côté, le plus petit nombre d'itérations réduit le temps total requis, puisque l'algorithme s'arrête rapidement s'il ne trouve pas de solution. Un exemple de cette configuration peut être vu sur la figure 1.

Lorsque nous ne travaillons pas avec de grandes valeurs d'itérations (1000, 1500, 2000), plus de nœuds sont générés, augmentant la probabilité de trouver un chemin vers l'objectif même dans des scénarios plus compliqués, il est donc normal que des chemins soient trouvés davantage. de manière cohérente. De plus, l'algorithme a de meilleures chances de se rapprocher d'un chemin plus court vers la cible, bien que cela ne soit pas toujours garanti en raison de la nature aléatoire du **RRT**. À mesure que le nombre d'itérations augmente, le temps nécessaire pour terminer le processus augmente également.

Un exemple de cette configuration peut être vu sur la figure 2.

	RRT		
	Iterations	Length	Time
250	206,00		0,21 ± 0,034
500	422,00 ± 54,79	72,38 ± 4,78	0,26 ± 0,03
1000	376,00 ± 167,06	71,85 ± 8,71	0,41 ± 0,03
1500	448,00 ± 129,13	73,75 ± 4,78	0,61± 0,07
2000	399,80 ± 177,65	74,45 ± 6,81	0,81 ± 0,03

TABLE I
RRT SELON NOMBRE D'ITÉRATIONS

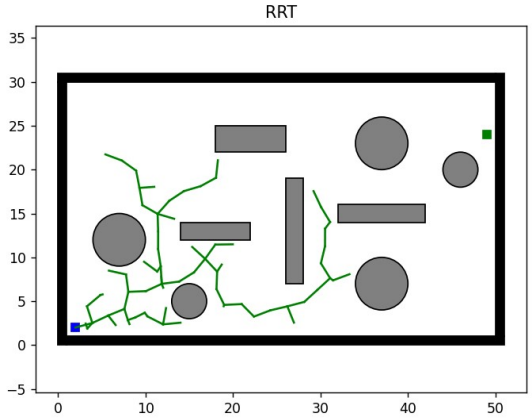

Fig. 1. RRT avec un faible nombre d'itérations

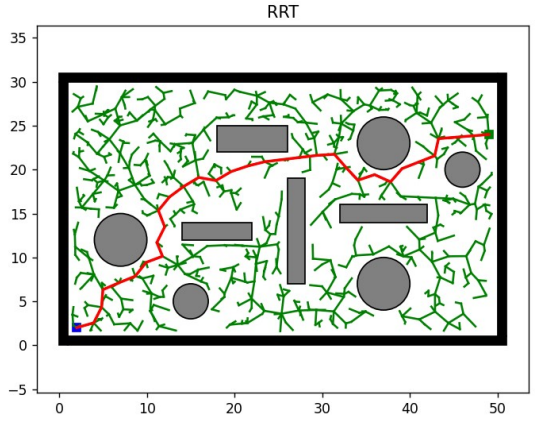

Fig. 2. RRT avec un nombre élevé d'itérations

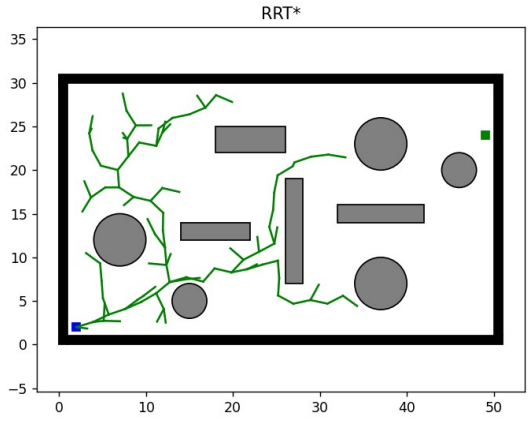

Fig. 3. RRT* avec un faible nombre d'itérations

B. *RRT**

L'algorithme **RRT*** (Rapidly-Exploring Random Tree Star) est une version améliorée de l'algorithme **RRT**, qui cherche non seulement à explorer l'espace de recherche, mais aussi à optimiser le chemin en ajustant les connexions entre les nœuds pour obtenir un chemin plus court et plus sûr. . Le paramètre itérations, dans le cas de **RRT***, comme auparavant, contrôle le nombre maximum de tentatives d'exploration et de connexion de nœuds dans l'espace de recherche.

Pour cette raison, d'une manière générale, la manière dont ce paramètre affecte le résultat reste constante par rapport à ce qui a été décrit précédemment avec **RRT**, comme le montre par exemple le tableau II.

	RRT*		
	Iterations	Length	Time
250	209,00	68,55	0,10 ± 0,032
500	350,20 ± 88,55	68,27 ± 5,40	0,55 ± 0,12
1000	355,60 ± 213,18	65,96 ± 5,93	4,22 ± 1,62
1500	379,20 ± 125,21	63,64 ± 2,40	11,00± 1,14
2000	643,00 ± 258,01	63,64 ± 3,56	15,56 ± 3,35

TABLE II
RRT* SELON NOMBRE D'ITÉRATIONS

En tant que **RRT***, il recherche non seulement un chemin viable, mais optimise également les connexions entre les nœuds tout au long de l'exploration, garantissant que le chemin final est asymptotiquement optimal et aussi court que possible. Cela signifie que **RRT*** peut mettre plus de temps à trouver la solution en raison de son processus d'optimisation, mais offre une trajectoire de meilleure qualité par rapport à **RRT**, qui est plus rapide mais moins efficace en termes de longueur de trajet. Cela donne les résultats visibles sur les figures 3, 4, où dans ces dernières on voit un chemin plus optimisé par rapport au cas de **RRT**.

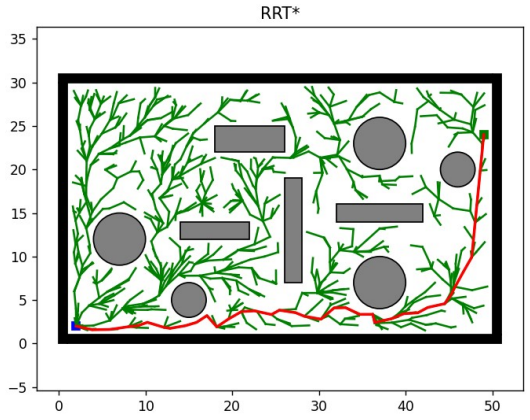

Fig. 4. RRT* avec un nombre élevé d'itérations

C. *Comparaison des résultats RRT et RRT**

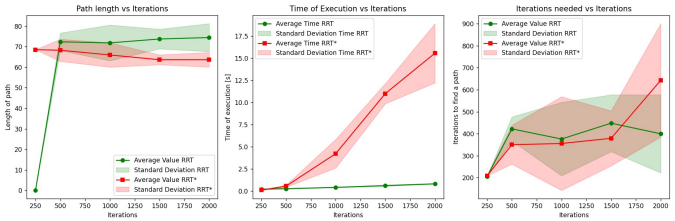

Fig. 5. Comparaison des résultats RRT et RRT*

En comparant les résultats obtenus avec **RRT** et **RRT*** (Figure 5) en fonction des itérations, de la longueur du chemin et du temps d'exécution, plusieurs différences clés sont mises en évidence qui reflètent les caractéristiques de chaque algorithme. Dans le cas de **RRT**, avec un faible nombre d'itérations (250), la longueur du chemin est de 206,00 unités et le temps d'exécution est relativement rapide à seulement 0,21 seconde. À mesure que les itérations augmentent, la longueur du chemin tend également à augmenter, atteignant une valeur de 448,00 unités avec 1500 itérations. Le temps d'exécution, quant à lui, augmente progressivement, atteignant

0,81 seconde en 2000 itérations. Ce comportement montre que **RRT** a la capacité de trouver des chemins rapidement, mais ces chemins ne sont pas forcément les plus efficaces en termes de distance.

En revanche, dans **RRT***, la longueur du trajet est nettement plus courte que dans **RRT**. Même avec un faible nombre d'itérations, comme 250, le chemin a une longueur de 209,00 unités, mais à mesure que les itérations augmentent, la longueur diminue considérablement, atteignant 63,64 unités pour 1500 et 2000 itérations. Cela reflète l'approche d'optimisation de **RRT***, qui ajuste continuellement les connexions entre les nœuds pour trouver des itinéraires plus courts et plus efficaces. Cependant, cette amélioration de la qualité du chemin entraîne une augmentation considérable du temps d'exécution. Alors qu'en **RRT** le temps d'exécution reste inférieur à 1 seconde même avec 2000 itérations, en **RRT*** le temps augmente progressivement, atteignant 15,56 secondes en 2000 itérations.

III. Question 2: Variation du step_len

A. RRT

Le paramètre **step_len** dans l'algorithme **RRT** contrôle la distance maximale que chaque nouveau nœud peut parcourir lors du développement de l'arborescence. Sa valeur a un impact direct sur l'efficacité et la qualité de la solution générée par l'algorithme.

Par exemple, si **step_len** est petit, l'arbre grandit de manière plus détaillée, explorant l'espace plus précisément. Cela peut être bénéfique dans des environnements complexes où il est nécessaire de trouver des chemins plus serrés autour des obstacles. Cependant, en raison de la faible distance parcourue par l'arbre à chaque itération, le nombre d'itérations nécessaires pour trouver un chemin peut être plus important, entraînant une augmentation du temps total d'exécution.

D'un autre côté, si **step_len** est grand, l'arbre avance rapidement, parcourant de plus longues distances à chaque pas. Cela peut réduire le nombre d'itérations nécessaires pour connecter le nœud de départ au nœud cible, diminuant ainsi le temps d'exécution global. Cependant, un pas_len trop grand peut faire sauter l'arbre par-dessus de petits obstacles ou des zones étroites, ce qui pourrait empêcher l'algorithme de trouver un chemin viable ou générer des trajectoires plus longues. Même si le temps d'exécution pourrait être moindre, la qualité de la route pourrait en être affectée négativement.

Ces comportements peuvent être vus dans le tableau des résultats III, où les valeurs des différentes variables sont vues en fonction de la valeur de **step_len**.

B. RRT*

Le paramètre **step_len**, dans le cas de **RRT***, comme avec **RRT**, détermine la distance maximale qu'un nouveau nœud peut parcourir à chaque itération, influençant ainsi la précision et l'efficacité de l'algorithme.

Lorsque le paramètre **step_len** est petit, l'algorithme génère des arbres plus détaillés, explorant l'espace de manière plus approfondie tout en effectuant de petits pas. En revanche, un **step_len** plus grand permet à l'algorithme d'atteindre des zones distantes plus rapidement, réduisant ainsi le nombre d'itérations nécessaires, mais au risque de perdre des détails fins dans l'espace de recherche.

	RRT		
	Iterations	Length	Time
0,2			0,524 ± 0,07
1	840,60 ± 392,04	74,00 ± 4,87	0,58 ± 0,04
2	448,00 ± 129,13	73,75 ± 4,78	0,61 ± 0,07
5	306,40 ± 110,23	70,67 ± 4,16	0,61 ± 0,05
10	241,60 ± 182,00	68,76 ± 1,30	0,61 ± 0,06

TABLE III
RRT SELON LEN_STEP

Les résultats de cet algorithme où la valeur de step_len varie peuvent être visualisés dans le tableau IV.

	RRT		
	Iterations	Length	Time
0,2			0,77 ± 0,08
1	829,40 ± 275,57	73,07 ± 6,28	4,89 ± 2,21
2	379,20 ± 125,21	63,64 ± 2,40	11,00 ± 1,14
5	267,20 ± 64,99	61,08 ± 1,99	17,82 ± 2,68
10	159,60 ± 55,27	62,81 ± 1,72	49,52 ± 11,66

TABLE IV
RRT* SELON LEN_STEP

C. Comparaison des résultats RRT et RRT*

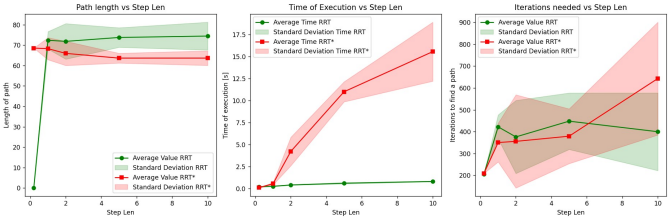

Fig. 6. Comparaison des résultats RRT et RRT*

En comparant les résultats obtenus (Figure6) pour les algorithmes **RRT** et **RRT*** avec différentes valeurs de **step_len**, plusieurs différences clés peuvent être observées en termes de le nombre d'itérations, la longueur du chemin et le temps d'exécution.

Pour les faibles valeurs de **step_len** (0,2 et 1), les deux algorithmes nécessitent un nombre considérable d'itérations pour trouver une solution, mais la longueur des chemins générés par **RRT*** est plus courte que **RRT**. Cela suggère que **RRT*** permet une plus grande optimisation du chemin, même avec un petit **step_len**, ce qui améliore la qualité du chemin au prix d'un temps d'exécution accru. Par exemple, pour **step_len** de 1, **RRT** utilise 840,60 itérations et produit un chemin de longueur 74,00 unités, tandis que **RRT*** utilise 829,40 itérations et obtient une longueur de 73,07, avec un durée d'exécution de 4,89 secondes.

À mesure que le **step_len** augmente (de 2 à 10), les deux algorithmes réduisent le nombre d'itérations requises, car de plus grandes distances entre les nœuds permettent une exploration plus rapide de l'espace de recherche. Cependant, **RRT*** conserve un avantage en termes de longueur de chemin. Par exemple, avec un **step_len** de 10, **RRT*** produit un chemin de 62,81, tandis que **RRT** obtient un chemin plus long de 68,76, bien que les deux algorithmes prennent à peu près le même temps d'exécution. (environ 0,61 seconde pour **RRT** et 49,52 secondes pour **RRT***).

IV. Question 3: environnement 2

Pour cet exercice, les paramètres ont été utilisés :

- environment = env.Env2()
- x_start = (2, 2)
- x_goal = (49, 24)
- iter_max = 1500
- goal_sample_rate = 0.1
- step_len = 2

Dans cet exercice, nous avons tenté de revoir le comportement du système lorsqu'il tente de travailler avec le deuxième environnement TP. Comme on peut le voir sur la figure 7 et dans les valeurs trouvées dans le tableau V, pour l'algorithme **RRT** il est impossible de générer un chemin entre le point initial et le but du système, cette difficulté est due à une raison principale liée à la structure de la carte et aux caractéristiques de l'algorithme.

Comme nous pouvons le constater, l'environnement contient un couloir étroit au centre, ce qui rend difficile l'exploration de l'espace de recherche. En raison de la nature aléatoire de l'algorithme **RRT**, la génération de nouveaux nœuds a tendance à être assez rare, et lorsque l'arbre tente de croître jusqu'à un point aléatoire à partir du nœud le plus proche, il est très probable que les nouveaux nœuds tomberont en dehors de la zone étroite. couloir ou qui sont bloqués par des obstacles. L'arbre a de fortes chances de ne pas trouver de chemin vers l'objectif car les nouveaux nœuds ne sont pas sur le bon chemin pour traverser le couloir, ce qui augmente le nombre d'itérations infructueuses. Lorsque l'arbre tente de se connecter à un nœud aléatoire, s'il se trouve dans une zone inaccessible ou très proche d'un obstacle, sa capacité à croître est limitée. Cela est particulièrement vrai dans des environnements complexes comme celui dans lequel nous travaillons, où des zones étroites nécessitent un contrôle plus précis sur la direction dans laquelle l'arbre pousse. Dans des situations comme celle-ci, l'algorithme **RRT** a du mal à naviguer dans ces couloirs étroits sans guidage ni méthode d'exploration plus intelligente.

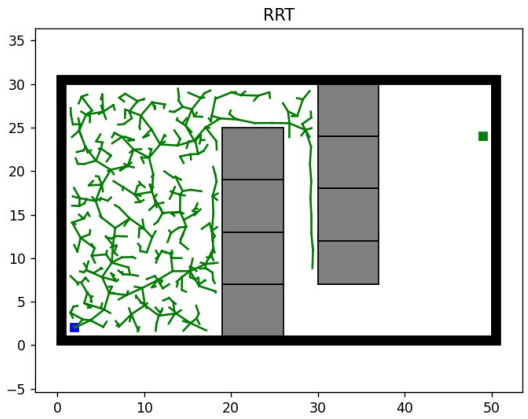

Fig. 7. Résultats Env2 avec RRT

		RRT
Iterations	Length	Time
		0,79
		0,65
		0,58
		0,64
		0,66

TABLE V
RRT* SELON LEN_STEP

V. Question 4: Amélioration de l'algorithme RRT avec l'implémentation d'OBRRT [1]

OBRRT (Optimization-Based Rapidly-exploring Random Tree) est une variante de l'algorithme classique **RRT** (Rapidly-exploring Random Tree) conçu pour répondre aux limites du **RRT** dans des scénarios avec des chemins étroits ou des zones de grande complexité.

OBRRT utilise une approche basée sur l'optimisation pour guider la croissance de l'arborescence vers les zones pertinentes de l'espace de configuration. Cela le rend mieux adapté aux scénarios avec des chemins étroits ou des régions spécifiques que le **RRT** standard trouve difficile à explorer.

L'OBRRT utilise un coût ou une mesure pour guider la croissance de l'arbre vers les régions où les solutions sont les plus susceptibles d'être trouvées. Au lieu de choisir des points complètement au hasard, **OBRRT** évalue les expansions vers des points qui maximisent la probabilité de progression vers l'objectif. Après avoir trouvé un chemin initial, **OBRRT** applique des techniques d'optimisation pour améliorer la solution trouvée. Ceci est utile pour trouver des solutions plus fluides ou plus efficaces par rapport au **RRT** standard.

Ci-dessous le code qui a été conçu pour exécuter ce nouvel algorithme.

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

        delta = self.utils.delta
        x1 = self.x_range[0] + delta
        x2 = self.x_range[1] - delta
        y1 = self.y_range[0] + delta
```

```
        y2 = self.y_range[1] - delta

        node = Node((
                np.random.uniform(x1, x2),
                np.random.uniform(y1, y2)
            ))
        p = np.random.random()

        delta *= 8

        if p < 0.2:
            while True:
                id = np.random.randint(len(
                     self.env.obs_rectangle))
                x, y, w, h = self.env.obs_rectangle[
    id]
                xa = x - delta
                xb = x + w + delta
                ya = y - delta
                yb = y + h + delta
                node_list = [
                    Node((np.random.uniform(xa, x),
                          np.random.uniform(ya, yb))
    ),
                    Node((np.random.uniform(x + w,
    xb),
                          np.random.uniform(ya, yb))
    ),
                    Node((np.random.uniform(xa, xb),
                          np.random.uniform(ya, y)))
    ,
                    Node((np.random.uniform(xa, xb),
                          np.random.uniform(y + h,
    yb)))
                ]

                node = node_list[
                     np.random.randint(len(node_list
    ))]
                if not self.utils.is_inside_obs(node
    ):
                    break
        return node
```

	Failure(%)	Av. Iter	Av. Lenght	Av. Time
		p = 0.2		
0	70	1174.87	111.43	0.55
20	62	1029.79	102.10	0.49
40	80	979.00	103.99	0.37
60	90	1407.40	100.55	0.29
80	100	–	–	0.16
100	100	–	–	0.03
		p = 0.5		
0	76	1148.92	109.70	0.65
20	44	1064.79	103.26	0.49
40	40	1119.06	102.49	0.35
60	60	1212.50	100.46	0.21
80	80	1315.45	99.75	0.10
100	100	–	–	0.02
		p = 0.8		
0	90	1115.00	106.90	0.80
20	20	888.82	102.55	0.74
40	40	1051.52	101.55	0.42
60	60	1338.89	99.37	0.20
80	80	1415.85	97.56	0.09
100	100	–	–	0.02

TABLE VI
RÉSULTAT AVEC L'IMPLÉMENTATION D'OBRRT

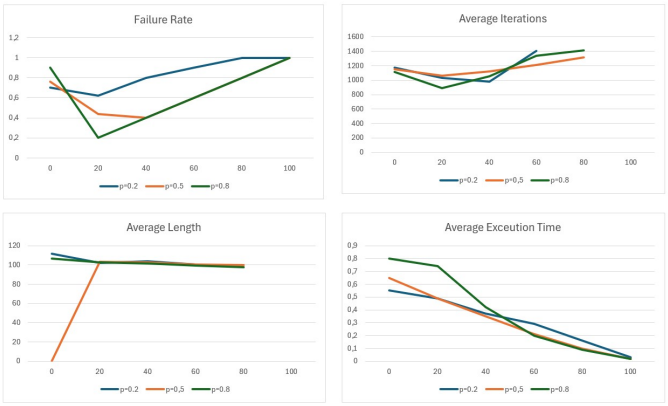

Fig. 8. Résultat avec l'implémentation d'OBRRT. Pour différents p.

Le code présenté est une implémentation d'une fonction qui génère des nœuds aléatoires dans le contexte d'un algorithme basé sur **RRT** (tel que **OBRRT**). Cette approche intègre un échantillonnage ciblé vers les régions proches des obstacles, ce qui constitue une particularité des méthodes basées sur l'optimisation. La fonction **generate_random_node** sélectionne des points d'échantillonnage de manière probabiliste, avec une probabilité (**p**) de générer des nœuds près des bords des obstacles dans l'espace de configuration. Dans ces cas, un obstacle rectangulaire est identifié aléatoirement dans l'environnement et un nœud est généré autour de ses bords, élargissant sa région avec un facteur delta. Ce mécanisme augmente la probabilité que l'arbre explore des chemins proches des obstacles, améliorant ainsi les performances dans les scénarios avec des chemins étroits. Le code garantit également que les nœuds générés ne se trouvent pas à l'intérieur d'obstacles, garantissant ainsi la validité des points d'échantillonnage pour la planification.

Le tableau VI et la Figure 8présente les résultats de l'algorithme OBRRT basé sur le paramètre appelé **p**, lié à la probabilité que des nœuds soient générés à proximité d'obstacles lors de la recherche.

Comme nous pouvons le constater, à mesure que la valeur de **p** augmente, le taux d'échec (% Failure) a également tendance à augmenter. Par exemple, lorsque $p = 0,2$, le taux d'échec est relativement faible (70% dans le cas de 0, et monte à 100% lorsque **corner˙sample˙rates = 100**). Cette augmentation du taux d'échec peut être liée à la difficulté de l'algorithme à générer des chemins viables dans des configurations plus étroites, où il est plus susceptible de rester bloqué sur des obstacles ou des chemins impossibles.

On observe qu'en général le nombre moyen d'itérations diminue lorsque **p** est grand, surtout. Concernant la longueur moyenne du chemin, elle a tendance à être plus courte à mesure que **p** augmente. Cela suggère que même si l'algorithme échoue plus souvent, lorsqu'il trouve une solution, celle-ci est plus courte.

Enfin, le temps moyen a tendance à diminuer à mesure que **p** augmente. Ce comportement est attendu car une valeur plus élevée de **p** réduit le nombre d'itérations nécessaires, et donc également le temps de calcul. Cependant, le temps est nettement plus court pour les cas où **corner˙sample˙rates = 100**, malgré les échecs, ce qui peut indiquer que l'algorithme s'arrête rapidement faute de trouver de solution.

La figure 9 montre un cas réussi de génération d'arbre en utilisant OBRRT, où l'on peut voir que même dans le cas d'un chemin très étroit, l'algorithme est capable de trouver un chemin qui relie le point initial et le point souhaité.

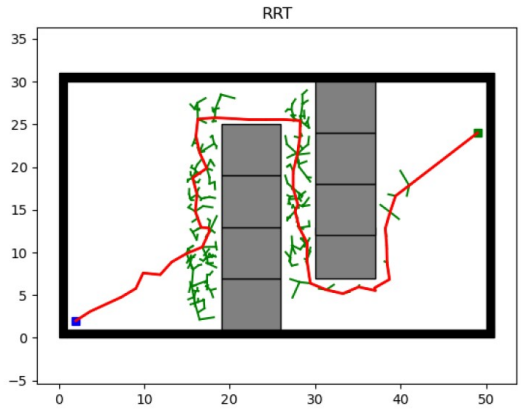

Fig. 9. Résultat avec l'implémentation d'OBRRT. p=0,2

VI. Conclusions

Dans ces travaux pratiques, les algorithmes **RRT** et **RRT*** ont été analysés et comparés en termes de longueur moyenne du chemin et de temps de calcul en faisant varier le nombre maximum d'itérations. Les résultats montrent que, bien que les deux algorithmes puissent trouver des chemins viables dans l'environnement donné, **RRT*** a tendance à générer des trajectoires plus courtes et de meilleure qualité que **RRT**, au prix d'un temps de calcul accru en raison de l'optimisation itérative effectuée par **RRT***. Concernant les temps de calcul, il a été observé que l'augmentation du nombre d'itérations augmente significativement le temps d'exécution dans les deux algorithmes, mais **RRT*** est plus affecté par cette variation du fait de son processus de raffinement.

En modifiant le paramètre de longueur de pas (**step_len**), il a été constaté que de petites valeurs amènent les algorithmes à générer des trajectoires plus tortueuses et plus détaillées, mais au prix d'un plus grand nombre d'itérations et de temps de calcul. En revanche, des valeurs élevées de **step_len** permettent de trouver des chemins plus rapidement, mais avec moins de précision et, dans certains cas, des chemins plus longs. Ces résultats suggèrent qu'un équilibre dans la longueur des pas est crucial pour obtenir de bonnes performances en termes de qualité de trajectoire et d'efficacité de calcul.

Dans la deuxième partie du travail, une variante de l'algorithme **OBRRT** a été implémentée pour planifier dans des environnements à couloirs étroits, comme celui défini par Env2. Cette approche a considérablement amélioré la capacité de l'algorithme à surmonter les difficultés survenues lors de la tentative de croissance de l'arbre jusqu'à des points aléatoires dans des zones comportant des obstacles. La modification de la fonction **generate_random_node** a permis à l'algorithme d'afficher de meilleures performances dans les zones difficiles en donnant la priorité à la génération de points dans les zones sans obstacles proches des bords des obstacles. En évaluant les performances en fonction du pourcentage de points échantillonnés dans ces domaines stratégiques, une amélioration significative de l'efficacité de la planification des trajectoires a été observée.

En conclusion, les résultats des travaux pratiques montrent que les algorithmes de planification **RRT** et **RRT*** sont efficaces pour résoudre des problèmes de planification dans des espaces libres d'obstacles, mais nécessitent des ajustements des paramètres pour optimiser l'équilibre entre qualité de trajectoire et temps de calcul. De plus, l'introduction de stratégies adaptatives telles que **l'OBRRT** dans des environnements d'obstacles complexes peut améliorer considérablement l'efficience et l'efficacité de l'algorithme dans des scénarios difficiles tels que des couloirs étroits.

VII. GitHub

Vous pouvez vous référer à la référence suivante pour voir les codes de nœud du projet: GitHub/RO16/TP2

References

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