ROB316 - Planification et contrôle
TP4 - Path planning using RRT

COELHO RUBACK Arthur
SANTOS SANO Matheus

Palaiseau, France

Décembre 2023

RRT vs RRT*

Question 1

La valeur *iter_ max* est le nombre maximum d'itérations que l'algorithme effectue pour trouver le chemin vers l'objectif. Ainsi, si cette valeur est trop faible, l'algorithme risque de ne pas avoir la possibilité d'explorer suffisamment l'environnement pour trouver un chemin vers l'objectif. En revanche, si *iter_ max* est trop grand, l'algorithme risque de passer plus de temps que nécessaire pour trouver le chemin.

Pour analyser le comportement des algorithmes RRT et RRT* en fonction du nombre maximal d'itérations, la taille moyenne des chemins trouvés par l'algorithme et le temps de calcul ont été analysées. Comme les deux algorithmes étant stochastiques, ils ont été simulés plusieurs fois car, selon la loi des grands nombres, plus on effectue de simulations, plus la probabilité de la moyenne arithmétique des résultats observés sera proche de la probabilité réelle. Ainsi, 100 simulations RRT ont été réalisées avec *iter_ max* très petit, très grand et intermédiaire. Les résultats obtenus sont présentés ci-dessous :

iter_ max	Distance moyenne du trajet	Temps de calcul
500	46.25	205.03
1500	59.25	334.67
3000	62.97	363.46
5000	64.80	374.02
10000	66.19	384.01

TABLE 1 – Distance moyenne du trajet et temps de calcul pour différentes valeurs de *iter_ max*. 100 simulations du RRT ont été réalisées pour chaque valeur de *iter_ max*.

Comme le montre le Tableau 1, lorsque *iter_ max* est petit, la distance moyenne du chemin est la plus faible, car dans certaines simulations, 500 itérations ne suffisent pas à l'algorithme RRT pour trouver un chemin. La Figure 1 illustre la distance du chemin trouvé pour différentes simulations et pour différentes valeurs de *iter_ max*.

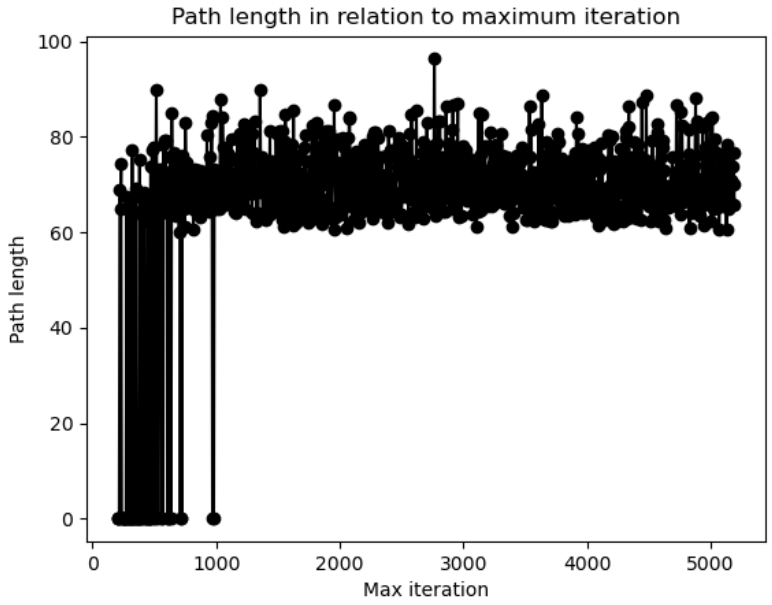


FIGURE 1 – Distance moyenne du trajet par rapport à *iter_ max*.

L'analyse de la Figure 1 montre que lorsque *iter_ max* est petit, il n'y a pas assez d'itérations pour que l'algorithme trouve le chemin vers le but, avec une distance égale à 0. Lorsque *iter_ max* est grand, le nombre d'itérations est suffisant pour que l'algorithme trouve le chemin. Toutefois, comme le montrent le Tableau 1 et la Figure 1, une forte augmentation du nombre maximal d'itérations n'affecte pas la distance moyenne du chemin obtenu ni le temps de calcul. On peut donc conclure que les performances de l'algorithme RRT ne s'améliorent pas si le nombre maximal d'itérations est trop élevé, car l'algorithme RRT n'a pas besoin d'un nombre excessif d'itérations pour trouver le chemin jusqu'à l'objectif.

Les performances de l'algorithme RRT*, en revanche, sont influencées par le nombre maximal d'itérations. Comme RRT* tente de modifier l'arbre tout au long de son exécution pour trouver le chemin le plus court vers l'objectif, le temps de calcul est équivalent au nombre d'itérations effectuées par l'algorithme, c'est-à-dire le nombre total d'itérations *iter_ max*, comme le montre le Tableau 2.

iter_ max	Distance moyenne du trajet	Temps de calcul
500	63.60	500
1500	63.49	1500
3000	63.87	3000
5000	63.83	5000
10000	63.82	10000

TABLE 2 – Distance moyenne du trajet et temps de calcul pour différentes valeurs de *iter_ max*. 100 simulations du RRT* ont été réalisées pour chaque valeur de *iter_ max*.

Comme RRT* recherche le chemin optimal, on peut voir dans le Tableau 2 qu'une forte augmentation du nombre maximal d'itérations n'améliore pas les performances de l'algorithme RRT*, puisqu'il n'a pas besoin d'un nombre excessif d'itérations pour trouver le chemin optimal.

On peut donc conclure que le nombre maximal d'itérations doit être suffisamment élevé pour que l'algorithme RRT* puisse trouver le chemin optimal, mais qu'une augmentation trop importante nuit à ses performances, car elle augmente le temps de calcul pour obtenir le même chemin optimal que celui obtenu avec un plus petit nombre d'itérations.

Question 2

Le paramètre *step_ len* représente la distance maximale que l'algorithme RRT parcourt à chaque itération pendant l'expansion de l'arbre. En d'autres termes, la longueur du pas par laquelle l'algorithme étend l'arbre du nœud le plus proche à un nouveau nœud a une taille maximale de *step_ len*.

Ainsi, si *step_ len* est trop petit, l'expansion de l'arbre devient très lente et, par conséquent, l'exploration de l'espace devient moins efficace. Comme l'expansion est faible à chaque itération, il faut un grand nombre d'itérations pour que l'algorithme atteigne sa destination. En considérant $step\_len = 0.5$ et $iter\_max = 1500$, le résultat est illustré à la Figure 2.

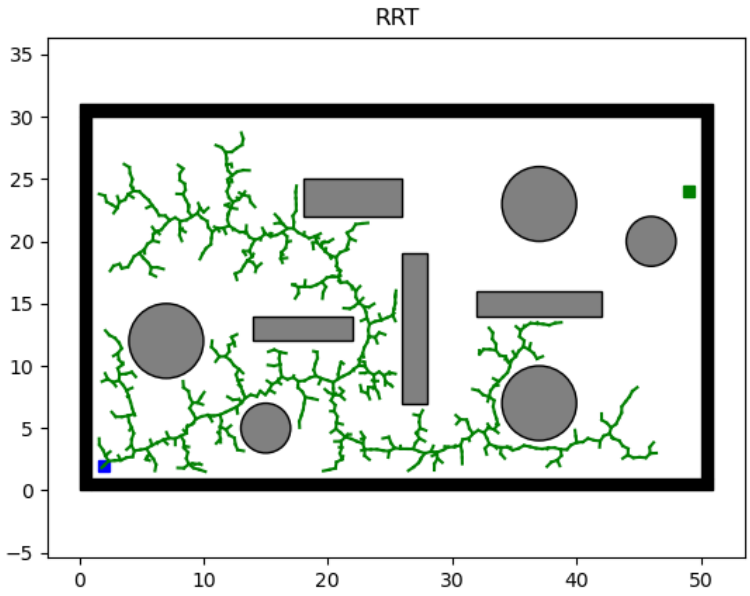


FIGURE 2 – Chemin trouvé avec $step\_len = 0.5$ et $iter\_max = 1500$.

Dans la Figure 2, on peut voir que l'algorithme n'a pas été en mesure de trouver un chemin vers le but parce qu'il n'y a pas eu assez d'itérations pour que l'algorithme s'étende suffisamment pour trouver ce chemin. Par conséquent, en augmentant le nombre maximal d'itérations, on obtient le résultat illustré à la Figure 3.

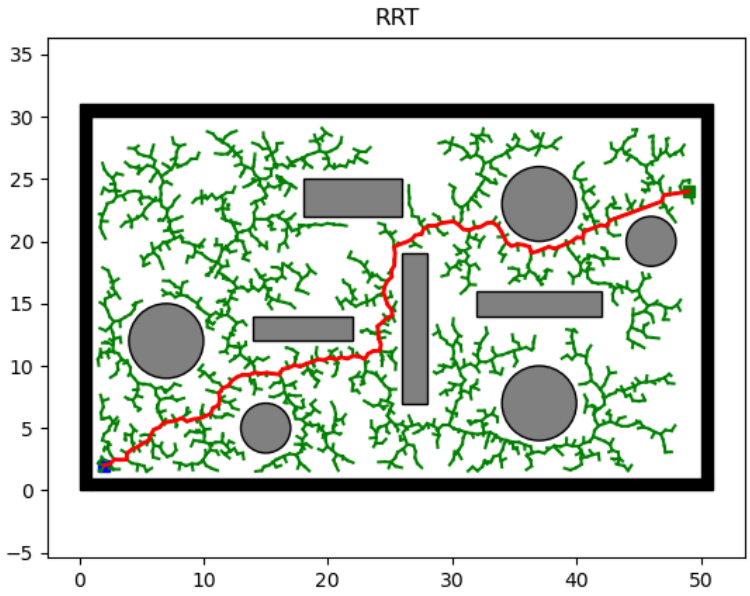


FIGURE 3 – Chemin trouvé avec $step\_len = 0.5$ et $iter\_max = 3000$.

Avec un nombre d'itérations suffisamment important, l'algorithme parvient à identifier un chemin. Cependant, comme $step\_len$ est encore trop petit, l'algorithme est encore trop lent pour trouver le chemin vers le but, comme on peut le voir dans le Tableau 3.

En revanche, si la valeur de *step_len* est très grande, l'expansion de l'arbre est très importante à chaque itération, et l'algorithme peut sauter certaines régions de l'espace qui n'ont pas encore été explorées. Ainsi, l'algorithme peut parcourir des distances inutiles, sans avoir mieux exploité une distance plus courte pour atteindre l'objectif. Ce phénomène est illustré par la Figure 4.

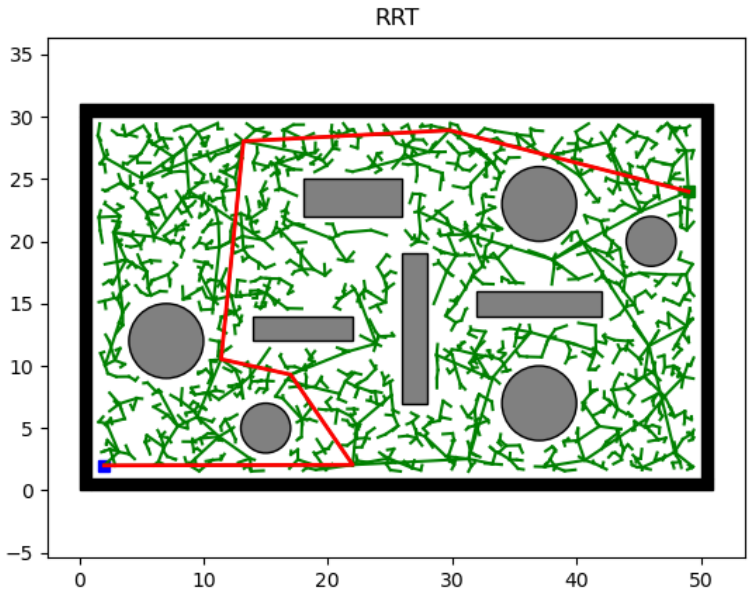


FIGURE 4 – Chemin trouvé avec *step_len* = 20 et *iter_max* = 3000.

Comme l'expansion de l'arbre à chaque itération est très importante, le nombre d'itérations nécessaires à l'algorithme pour atteindre l'objectif est inférieur à celui obtenu pour le défaut, comme le montre le Tableau 3.

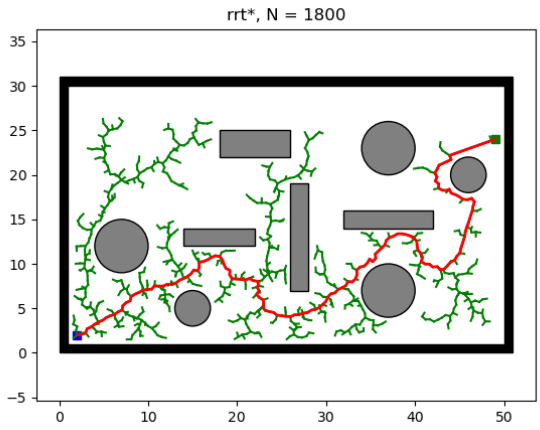
step_len	Nombre moyen d'itérations (100 simulations)
0.5	1752.03
2	425.94
20	230.15

TABLE 3 – Nombre moyen d'itérations par rapport à *step_len* avec *iter_max* = 3000. 100 simulations ont été réalisées.

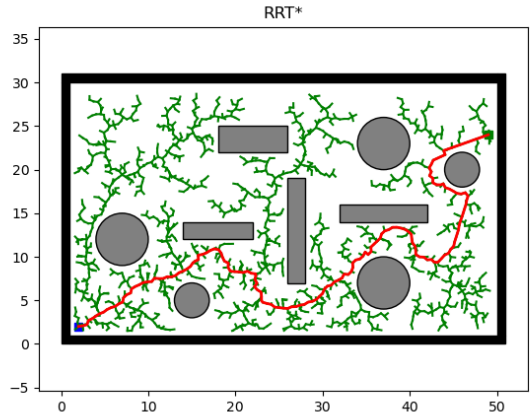
En bref, il est nécessaire de trouver un *step_len* suffisamment grand pour que l'algorithme atteigne son but plus rapidement, mais en même temps le *step_len* ne doit pas être trop grand pour que l'algorithme puisse mieux explorer l'environnement.

L'algorithme RRT*, à son tour, permet de trouver le chemin optimal vers le but, c'est-à-dire le chemin le plus court. Lorsque on varie la valeur de *step_len*, on obtient des résultats similaires à ceux de RRT. Cependant, comme l'algorithme RRT* reconstruit l'arbre pour toujours minimiser le coût total du chemin, il est possible d'analyser le comportement du chemin tout au long de l'exécution de RRT*.

Lorsque *step_len* est trop petit, l'algorithme est trop lent et nécessite toujours plus d'itérations pour trouver le chemin vers l'objectif. De plus, le chemin trouvé jusqu'à l'objectif final change légèrement au cours des itérations, car l'algorithme explore minutieusement l'environnement à chaque petit pas. C'est ce que montre la Figure 5.



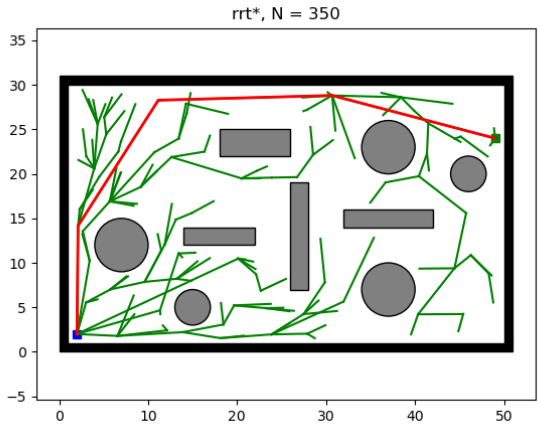
(a) Chemin trouvé après 1800 itérations.



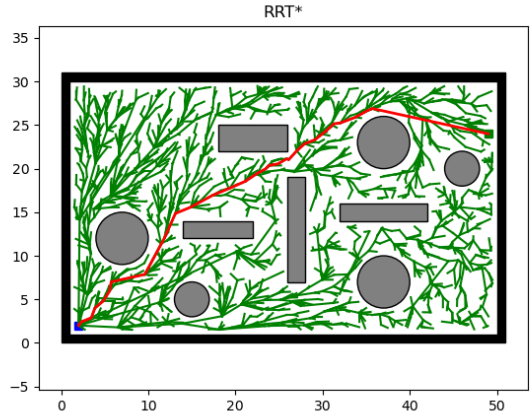
(b) Chemin final trouvé après toute l'exécution de RRT*.

FIGURE 5 – Chemin trouvé à différents moments de l'exécution du RRT* avec $step\_len = 0.5$ et $iter\_max = 3000$.

Lorsque *step_len* est trop grand, le chemin atteint son but rapidement, mais n'explore pas l'environnement. Ce manque d'exploration approfondie de l'environnement signifie que le chemin trouvé par l'algorithme change beaucoup au cours des itérations, puisqu'un chemin plus optimal est toujours trouvé après avoir exploré plus zones. Ce phénomène est illustré par la Figure 6.



(a) Chemin trouvé après 350 itérations.



(b) Chemin final trouvé après toute l'exécution de RRT*.

FIGURE 6 – Chemin trouvé à différents moments de l'exécution du RRT* avec $step\_len = 20$ et $iter\_max = 3000$.

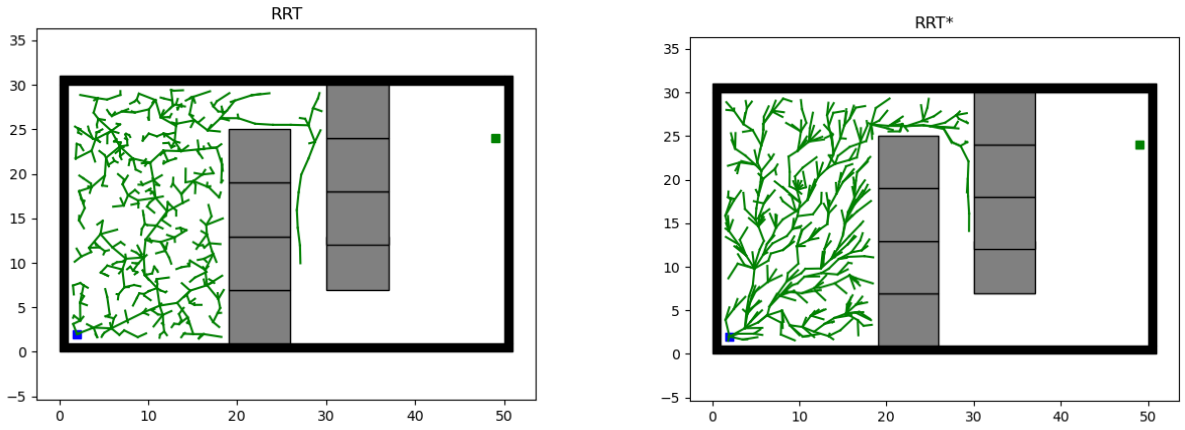
Planification dans des couloirs étroits

Question 3

Dans ce nouvel environnement, les algorithmes RRT et RRT* ont plus des difficultés à étendre l'arbre en raison du plus grand nombre d'obstacles et de leur configuration dans l'environnement. Les deux algorithmes tentent d'étendre l'arbre vers des points aléatoires, mais ils vérifient si la ligne menant à ce point traverse des obstacles. Si c'est le cas, elle ne relie pas ce point aléatoire dans l'arbre et ce point est rejeté. Dans ce nouvel environnement, la configuration des obstacles rend difficile l'expansion de l'arbre, car la plupart des points aléatoires sélectionnés sont rejetés.

Par conséquent, le seul moyen pour l'arbre de s'étendre est de passer par l'espace étroit entre les deux obstacles. Pour que l'algorithme suive effectivement ce chemin et évite ces obstacles, il faut plusieurs itérations. Il est donc difficile pour l'arbre de croître rapidement dans l'environnement.

Le résultat obtenu par les algorithmes RRT et RRT* avec les paramètres initiaux est présenté dans la Figure 7.



(a) Résultat obtenu par l'algorithme RRT. (b) Résultat obtenu par l'algorithme RRT*.

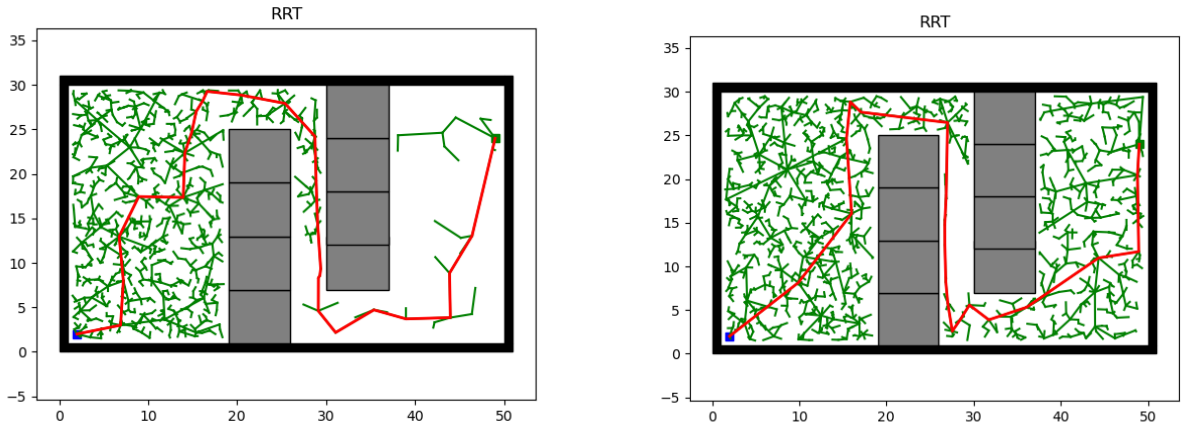
FIGURE 7 – Résultats obtenus par RRT et RRT* dans l'environnement Env2.

Comme le montre la Figure 7, les algorithmes ont des difficultés à trouver un chemin vers le but. Avec ces paramètres, il est possible d'obtenir un chemin, mais, dans le cas illustré sur la Figure 1, l'arbre se développe très lentement, ce qui rend plus difficile l'obtention d'un chemin vers le but.

Question 4

Pour que l'algorithme puisse atteindre son objectif avec cette configuration d'obstacles, une variante simple de l'OBRRT a été implémentée. Dans cette nouvelle stratégie, on échantillonne des points sur les bords des obstacles afin que l'algorithme puisse "suivre" le mur de l'obstacle. On considère les bords supérieur, inférieur, droit et gauche d'un obstacle choisi au hasard. De plus, il est possible de varier le pourcentage de points échantillonnés en utilisant cette nouvelle stratégie. L'implémentation est illustrée ci-dessous dans la Figure 13.

Cette adaptation de l'algorithme permet de passer plus facilement par le chemin étroit entre les obstacles, puisque l'algorithme "suit" leurs murs plus rapidement. En faisant varier le pourcentage de points échantillonnés par cette stratégie, on obtient différentes performances de l'algorithme. Si ce pourcentage est faible, la plupart des nœuds échantillonnés sont générés de manière aléatoire et ne se trouvent pas nécessairement sur le bord d'un obstacle. De cette manière, l'algorithme a plus de liberté pour explorer l'environnement.

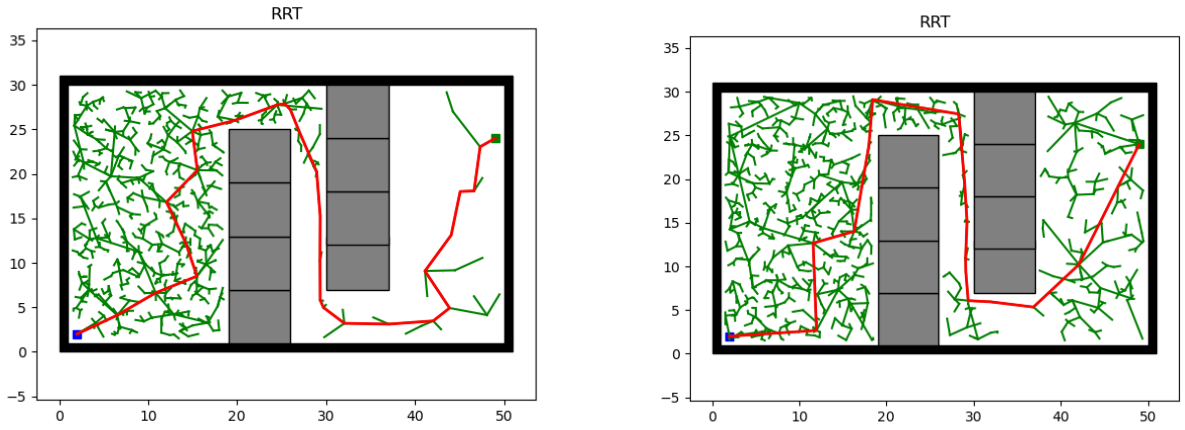


(a) Résultat obtenu par l'algorithme RRT avec *step_len* 5.

(b) Résultat obtenu par l'algorithme RRT avec *step_len* 10.

FIGURE 8 – Résultats obtenus par RRT avec pourcentage 0% et *iter_max* 3000.

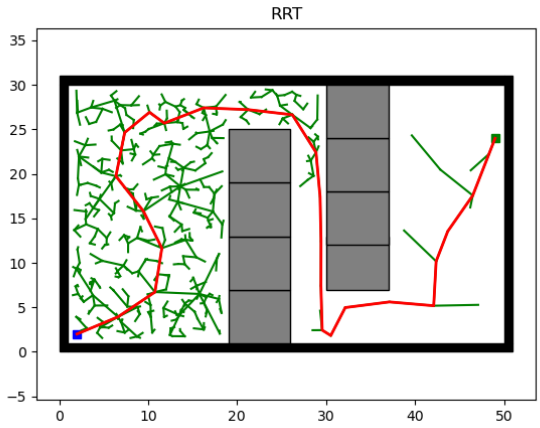
Lorsque l'on augmente le pourcentage de nœuds obtenus par la nouvelle méthode, le nombre de points échantillonnés est nécessairement proche d'un obstacle, ce qui laisse moins de liberté à l'algorithme pour explorer l'environnement. Cependant, l'algorithme parvient à suivre le mur d'un obstacle le long de chemins plus étroits, comme le montrent les Figures 9 et 10.



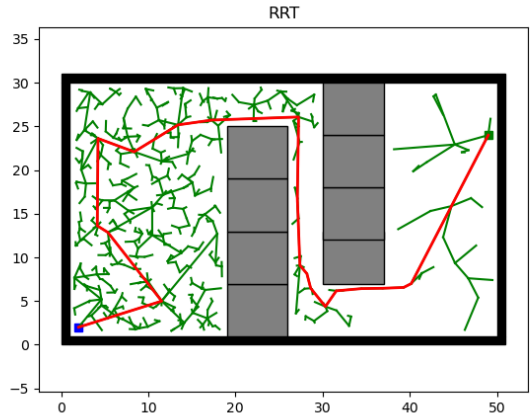
(a) Résultat obtenu par l'algorithme RRT avec *step_len* 5.

(b) Résultat obtenu par l'algorithme RRT avec *step_len* 10.

FIGURE 9 – Résultats obtenus par RRT avec pourcentage 25% et *iter_max* 3000.



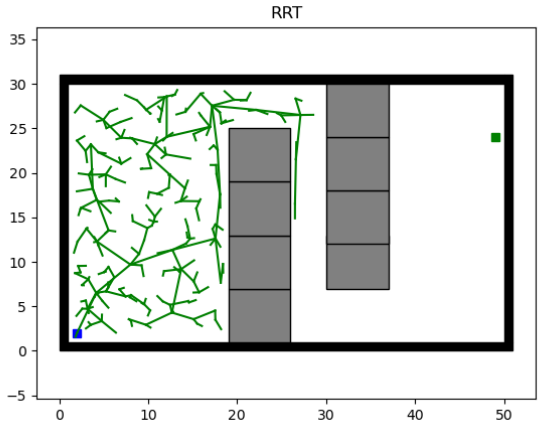
(a) Résultat obtenu par l'algorithme RRT avec *step_len* 5.



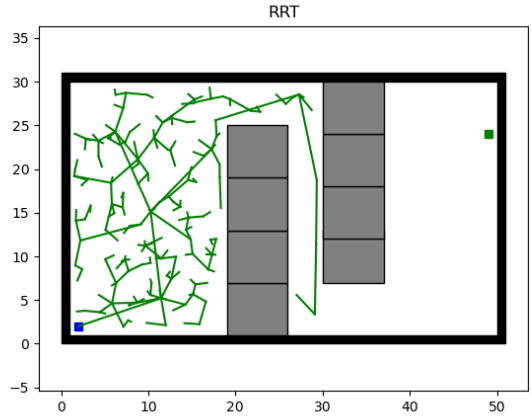
(b) Résultat obtenu par l'algorithme RRT avec *step_len* 10.

FIGURE 10 – Résultats obtenus par RRT avec pourcentage 50% et *iter_max* 3000.

Si on augmente trop ce pourcentage, la plupart des points échantillonnés sont proches de l'obstacle, forçant presque l'algorithme à suivre le mur d'un obstacle. De cette manière, l'algorithme n'explore pas correctement l'environnement et met plus de temps à atteindre l'objectif. Comme le montrent les Figures 11 et 12, l'algorithme ne parvient même pas à atteindre son objectif.



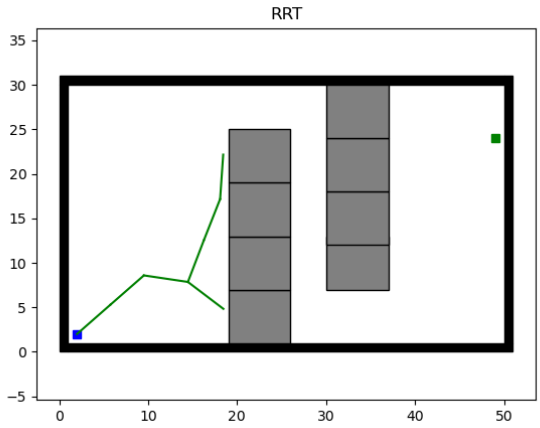
(a) Résultat obtenu par l'algorithme RRT avec *step_len* 5.



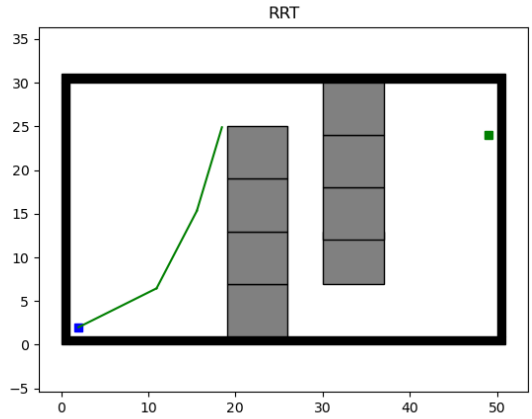
(b) Résultat obtenu par l'algorithme RRT avec *step_len* 10.

FIGURE 11 – Résultats obtenus par RRT avec pourcentage 75% et *iter_max* 3000.

Dans le cas où le pourcentage est de 100%, tous les points échantillonnés sont obtenus par cette nouvelle méthode et l'algorithme ne cherche donc à atteindre l'objectif qu'en suivant le mur d'un obstacle. De cette manière, l'algorithme n'atteint pas son but, comme le montre la Figure 12.



(a) Résultat obtenu par l'algorithme RRT avec *step_len* 5.



(b) Résultat obtenu par l'algorithme RRT avec *step_len* 10.

FIGURE 12 – Résultats obtenus par RRT avec pourcentage 100% et *iter_max* 3000.

En résumé, le pourcentage doit être suffisamment élevé pour que l'algorithme puisse passer les points situés au bord des obstacles et, par conséquent, traverser plus facilement le chemin étroit. Cependant, il ne faut pas trop augmenter le pourcentage, car cela empêcherait l'algorithme d'explorer efficacement l'environnement et de, surtout, suivre les obstacles, ce qui nuirait à ses performances.

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

    # Percentage of points sampled using a simple variant of the OBRRT
    percentage_strategy = 0.3
    if np.random.random() <= percentage_strategy:
        new_node_generated = False
        while not new_node_generated:
            # Get random obstacle
            x_obs, y_obs, w_obs, h_obs = self.env.obs_rectangle[np.random.randint(len(self.env.obs_rectangle))]

            # Decide which node around the obstacle (within the obstacles edge) will be selected
            id_selected_node = np.random.randint(4)

            if id_selected_node == NODE_LEFT:
                # The left edge of the obstacle
                new_node = Node((np.random.uniform(x_obs - delta, x_obs),
                                 np.random.uniform(y_obs - delta, y_obs + h_obs + delta)))
            elif id_selected_node == NODE_RIGHT:
                # The right edge of the obstacle
                new_node = Node((np.random.uniform(x_obs + w_obs, x_obs + w_obs + delta),
                                 np.random.uniform(y_obs - delta, y_obs + h_obs + delta)))
            elif id_selected_node == NODE_BELOW:
                # The edge below the obstacle
                new_node = Node((np.random.uniform(x_obs - delta, x_obs + w_obs + delta),
                                 np.random.uniform(y_obs - delta, y_obs)))
            elif id_selected_node == NODE_TOP:
                # The top edge of the obstacle
                new_node = Node((np.random.uniform(x_obs - delta, x_obs + w_obs + delta),
                                 np.random.uniform(y_obs + h_obs, y_obs + h_obs + delta)))

            # Verify if the new node is not inside an obstacle
            if self.utils.is_inside_obs(new_node):
                new_node_generated = True
    else:
        new_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)))

    return new_node
```

FIGURE 13 – Code d'une variante simple de l'OBRRT.