RRT : « Rapidly exploring Random Trees »
Méthode probabiliste pour la planification de mouvement

PARTIE THEORIQUE :

A - Introduction

La méthode RRT basique, introduite par S. *LaValle* en 1998 à l'Université d'Illinois (aux Etats Unis d'Amérique), représente un intermédiaire entre les méthodes probabilistes classiques (Probabilistic Road Maps – 1996 – *Kavraki*, *Svestka, Latombe et Overmars*) et la technique de marche aléatoire (introduite par *Motwani* et *Hsu*). Elle consiste à faire évoluer un arbre partant de la configuration initiale afin d'atteindre une configuration but. RRT est représentée par un graphe topologique noté *G*. L'arbre a pour racine la configuration initiale q_{init}. Il est composé de *k* arrêtes et il est construit selon l'algorithme 1 :

Algorithme 1

Build_RRT(q_{init},k)
1. G.init(q_{init});
2. Pour i=1 to k
3. q_{rand} = rand_config();
4. Etendre (G, q_{rand}) ;
5. Retourne G ;

Etendre (G, q_{rand})
1. q_{near} = Plus_proche_noeud(q_{rand} , G);
2. Si Nouvelle_config(q_{rand} ,q_{near} , q_{new} ,Δq) alors
3. G.ajoute_noeud(q_{new}) ;
4. G.ajoute_arrrête(q_{near} , q_{new} , Δq) ;
5. Si q_{new} = q_{rand} alors
6. Retourne *Atteint*;
7. Sinon
8. Retourne *Etendu*;
9. Retourne Echec ;

La fonction *rand_config()* génère aléatoirement une configuration q_{rand} dans l'espace libre. La fonction *Etendre* permet de sélectionner à chaque itération le nœud de l'arbre *G* le plus proche (fonction *Plus_proche_noeud()*) à q_{rand} selon une métrique (distance euclidienne par exemple). La fonction *Nouvelle_config()* effectue un mouvement dans la direction de q_{rand} avec un incrément Δq. Ce dernier paramètre peut être choisi dynamiquement au cours de l'exécution. Il peut être aussi fonction de

la distance aux obstacles afin d'accélérer l'extension de l'arbre. Plus le robot est loin des obstacles, plus le pas est grand. La Figure 1 illustre une itération de l'algorithme RRT basique.

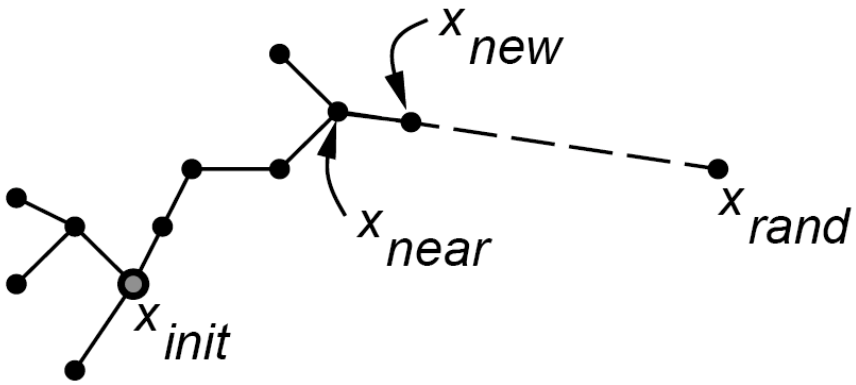

Fig. 1 : Illustration d'une itération de l'algorithme 1.

Par ailleurs, au lieu d'étendre l'arbre RRT avec un pas incrémental, la fonction *Etendre()* pourrait être itérer jusqu'à ce que la configuration aléatoire ou un obstacle soit atteint. Dans ce cas, la fonction *Connecter()* présentée ci-dessous remplacerait la fonction *Etendre()*. Le principal avantage de cette fonction est qu'un long chemin peut être construit avec un seul appel à la fonction *Plus_proche_noeud()* (fonction pouvant être coûteuse en terme de temps de calcul).

Connecter (G, q_{rand})
1. Répéter
2. S = Etendre (G, q_{rand}) ;
3. jusqu'à not(S = Etendu)
4. Retourne Echec ;

B – RRT Bidirectionnelle

Depuis, plusieurs variantes de l'algorithme basique ont été proposées dans la littérature. Dans ce TP, nous nous intéresserons à la version bidirectionnelle de RRT. L'algorithme de cette version consiste à faire évoluer deux arbres, l'un initial G_{init} et l'autre final G_{but}, ayant pour racines respectives la configuration initiale et la configuration but. La construction des deux arbres est décrite dans l'algorithme 2 :

Algorithme 2

RRT_Bidirectionnelle(q_{init},q_{but})
1. G_{init} .init(q_{init}), G_{but} .init(q_{but}), ;
2. Pour i=1 to k
3. q_{rand} = rand_config();
4. Si not(Etendre (G_{init} , q_{rand}) = *Echec*) alors
5. Si (Etendre(G_{but} , q_{new}) = *Atteint*) alors
6. Retourne Chemin(G_{init} , G_{but}) ;
7. Echanger (G_{init} , G_{but}) ;
8. Retourne Echec;

L'algorithme *RRT_Bidirectionnelle()* devise le temps de calcul entre deux processus :

1. Explorer l'espace des configurations,
2. Etendre les deux arbres l'un vers l'autre.

A chaque itération l'un des arbres (G_{init} , G_{but}) est étendu et une connexion du plus proche nœud de l'autre arbre au nouveau nœud est testée. De la même façon que pour RRT basique, la fonction *Connecter()* peut remplacer la fonction *Etendre()* nous permettant ainsi d'obtenir un meilleur compromis exploration/temps de calcul. La Figure 2 illustre une itération de l'algorithme RRT bidirectionnelle.

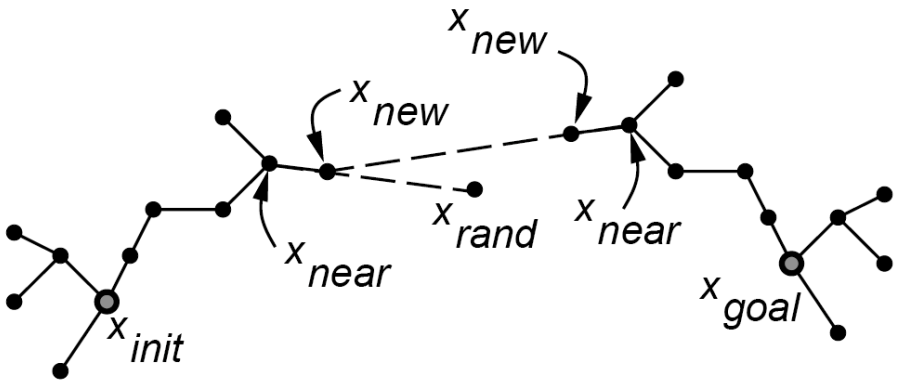

Fig. 1 : Illustration d'une itération de l'algorithme 2.

PARTIE EXPERIMENTALE :

Dans la partie expérimentale, nous allons considérer un cas simple où l'espace de configuration est limité à 2 dimensions.

A – RRT basique

Dans cette partie, nous allons nous intéresser à l'algorithme basique de RRT permettant d'explorer l'espace de configuration. Le but est d'obtenir une couverture de l'espace des configurations semblable à celle représentée dans la figure 3.

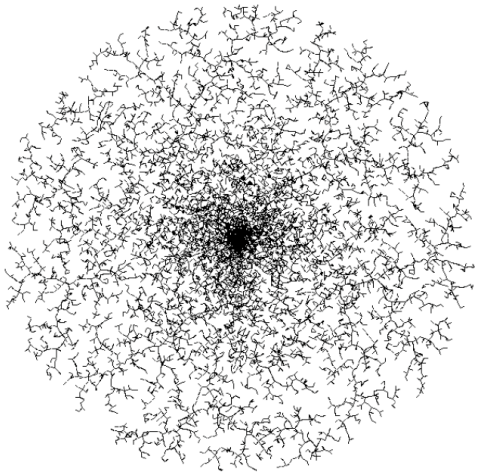

Fig. 3 : Échantillonnage polaire.

Pour cela, nous vous conseillons de suivre les étapes suivantes :

1. Ecrire une fonction matlab *Explore* permettant de générer itérativement K points dans un environnement 2D. Commencer par définir une configuration initial *start* (au centre de l'environnement par exemple) et une structure d'arbre *rrt* ayant comme racine *start* et les deux champs suivants :
   - rrt(1).coords : permet de stocker les coordonnées des nœuds de l'arbre,
   - rrt(1).parent : permet de stocker le père du nœud généré.

2. Connecter les différents nœuds générés selon une stratégie de votre choix (à la racine par exemple) et tracer sur la même figure les arrêtes.

3. Modifier la stratégie de connexion en choisissant le plus proche voisin. Conclure.

4. Modifier votre fonction en prenant en considération le paramètre Δq défini dans l'algorithme 1 (partie théorique).

5. Modifier la technique d'échantillonnage utilisée pour la génération des nœuds (échantillonnage polaire). Déduire l'influence de la technique choisie sur le graphe construit.

6. Comment modifier la fonction *Explore* pour l'utiliser dans un problème de planification de mouvement (RRT bidirectionnelle en l'absence d'obstacles).

A – <u>RRT bidirectionnelle</u>

Nous allons nous intéresser dans cette partie à l'algorithme RRT bidirectionnelle.

1. Récupérer le répertoire RRT. Tester l'algorithme en faisant varier les différents paramètres ainsi que les environnements.

2. Proposer votre propre environnement et tester l'algorithme en présence de passage étroit.

3. Ecrire la fonction *smoothpath* permettant de lisser le chemin solution.