Planification de mouvement : RRT et PRM

L'objectif de ce TP est de vous faire programmer et manipuler les deux principaux algorithmes de planification de mouvement : les RRT (Rapidly exploring Random Trees) et les PRM (Probabilistic Roadmaps). Ces algorithmes seront programmés pour un mobile à 6 degrés de liberté (3 degrés de liberté en translation et 3 degrés de liberté en rotation).

Introduction

Comme vous le savez, les algorithmes de type RRT et PRM ont besoin de faire des requêtes du type plus proche voisin (pour les RRT) ou k plus proches voisins (pour les PRM). Etant donné que l'espace des configurations peut-être de grande dimension, beaucoup des structures de données permettant de faire des requêtes de voisinage telles que les grilles régulières, Kd-Trees ou extensions des quadtrees / octrees en N dimensions ne sont pas vraiment adaptées à ce type d'espace. Dans le cadre de ce TP, nous utiliserons une structure de données appelée « Vantage Point tree »[1] (ou « VP-Tree » en abrégé) qui est implémentée par la classe *MotionPlanning::VPTree*. Durant le TP, vous n'aurez pas à directement manipuler cette structure de données car des classes l'encapsulant vous sont fournies :

- Un arbre pouvant stocker des configurations et proposant des requêtes de voisinage vous est fourni dans la classe *MotionPlanning::SixDofConfigurationTree*.
- Un graphe pouvant stocker des configurations et proposant des requêtes de voisinage vous est fourni dans la classe *MotionPlanning::SixDofConfigurationGraph*.

Les algorithmes de planification devront être implémentés par héritage de la classe *MotionPlanning::SixDofPlannerBase* qui a pour rôle de fournir un certain nombre de fonctionnalités communes aux algorithmes de planification : tirage aléatoire d'une configuration, tests de collisions et calcul de distance entre configurations par exemple. Cette classe met aussi à votre disposition une classe interne nommée *Configuration* représentant une configuration du mobile à 6 degrés de liberté.

Partie 1 : Préparer le terrain : distance et méthode locale

Dans un premier temps, nous allons compléter la classe *MotionPlanning::SixDofPlannerBase* afin de programmer les méthodes nécessaires à la réalisation des algorithmes de planification de chemin.

Question 1.1 : La classe *MotionPlanning::SixDofPlannerBase* met à disposition la méthode *configurationDistance(const Configuration &, const Configuration &)* permettant de calculer une distance entre deux configurations. Pour calculer cette distance, nous allons utiliser la formule suivante. Soit $C_1 = (T_1, Q_1)$ et $C_2 = (T_2, Q_2)$ deux configurations, nous définissons la distance entre deux configurations comme suit :

$$d(C_1, C_2) = w_T \left|\left|T_1 - T_2\right|\right| + w_R \text{acos}(|Q_1 . Q_2|) / (0.5 * \pi)$$

Dans cette formule, w_T et w_R correspondent à des poids donnant plus ou moins d'importance à la translation ou la rotation. Pour les besoins du TP, ces poids pourront être fixés à 1. Le symbole « . » entre les deux quaternions représente le produit scalaire entre ces derniers (méthode *dot* de la classe *Math::Quaternion*). Cette distance a la bonne propriété de respecter l'inégalité triangulaire, ce qui la rend compatible avec l'utilisation du « Vantage Point Tree » utilisé pour les requêtes de voisinage entre

[1] Voir https://en.wikipedia.org/wiki/Vantage-point_tree et https://fribbels.github.io/vptree/writeup pour plus d'informations sur cette structure de données.

configurations. Programmez le corps de la méthode *configurationDistance* (la déclaration de la méthode est faite et son implémentation se trouve dans le fichier cpp, elle est actuellement vide).

Question 1.2 : Comme vous le savez, la méthode locale utilise la détection de collision pour vérifier que le chemin d'interpolation entre deux configurations se trouve dans C-free. C'est le rôle de la méthode *bool doCollide(Configuration c1, Configuration c2, float dq)* de la classe *MotionPlanning::SixDofPlannerBase*. Cette méthode utilisera une détection de collision discrète pour laquelle *dq* est la distance maximale entre deux configurations testées le long du chemin d'interpolation entre *c1* et *c2*. Programmez le corps de cette méthode (la déclaration est fournie et son implémentation vide se trouve dans le fichier cpp associé). Vous pourrez vous aider de la méthode *doCollide(Configuration)* pour savoir si une configuration donnée est en collision avec l'environnement.

Partie 2 : Programmation de l'algorithme de RRT.

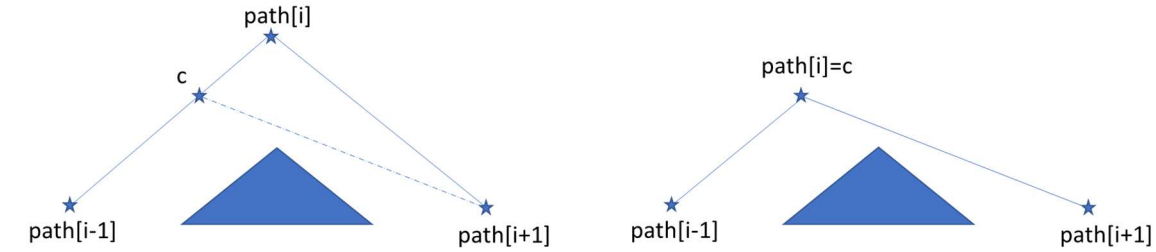
Question 2.1 : Nous allons maintenant programmer l'algorithme de RRT. Pour ce faire, créez une classe *MotionPlanning::RRT* héritant de *MotionPlanning::SixDofPlannerBase* qui a pour rôle d'implémenter l'algorithme des RRT dans la méthode *plan* héritée de *MotionPlanning::SixDofPlannerBase*. Dans cette méthode, les paramètres *start* et *target* correspondent aux configurations que l'on souhaite relier par un chemin. Le paramètre *radius* correspond à la distance maximale entre deux configurations stockées dans le RRT et *dq* correspond à la distance maximale entre deux configurations lors de l'utilisation de la méthode locale (Cf. question 1.2). Vous pourrez vous aider de la classe *MotionPlanning::SixDofConfigurationTree* pour stocker l'arbre de configurations associé au RRT et effectuer les requêtes de voisinage.

Question 2.2 : Pour tester votre planificateur RRT référencez la classe Application::SIAA_TP4_MotionPlanning (fichier « Application/TP4_MotionPlanning.h ») auprès du lanceur d'application dans le main. Dans un second temps, décommentez le cas 0 dans le switch de la méthode *computePlan* de la classe SIAA_TP4_MotionPlanning. Un bon paramétrage pour la méthode *plan* est de mettre *radius* à 0.1 lorsque cela vous est demandé.

Comme vous pouvez certainement le remarquer, le mouvement de votre mobile est quelque peu erratique. Ceci est lié à la nature aléatoire de l'algorithme de planification. Dans la classe *MotionPlanning::SixDofPlannerBase* vous trouverez la méthode *optimize* qui prend en paramètre un chemin (vecteur de configurations) et *dq* (distance maximale entre deux configurations pour l'utilisation de la méthode locale. L'implémentation de cette méthode est actuellement vide.

Question 2.3 : Une première optimisation de chemin consiste à supprimer du chemin toute configuration d'indice *i* pour laquelle les configurations d'indice *i-1* et *i+1* peuvent être reliées par la méthode locale. Programmez cette optimisation dans la méthode *optimize* de la classe SixDofPlannerBase. Modifiez votre code de manière à optimiser le chemin lorsque ce dernier est produit par la méthode *plan*. Observez le résultat.

Question 2.4 : Une seconde optimisation consiste à essayer de lisser le chemin en complément de l'optimisation précédente. Pour ce faire, vous pouvez tirer aléatoirement un indice de configuration *i* tel que $i \in]0; size - 1[$ où *size* est le nombre de configurations le long du chemin. Il s'agit ensuite de tirer aléatoirement une configuration *c* sur le chemin reliant les configurations d'indices *i-1* et *i.* Si cette configuration peut être reliée à la configuration à la configuration d'indice *i+1* alors la configuration *c* remplace la configuration d'indice *i*. Cet algorithme est illustré dans la Figure 1 : Optimisation de chemin de la question 2.4. Cette opération doit être répétée un certain nombre de fois pour optimiser le chemin.



Tirage aléatoire de c sur le chemin *(path[i-1], path[i])*
Vérification du chemin *(c, path[i])* avec la méthode locale

Le chemin *(c, path[i])* est libre de collision
La configuration *path[i]* est remplacée par *c*.

Figure 1 : Optimisation de chemin de la question 2.4

Partie 3 : Programmation de l'algorithme de PRM

Question 3.1 : Nous allons maintenant programmer l'algorithme de PRM. Pour ce faire, créez une classe *MotionPlanning::PRM* héritant de la classe *MotionPlanning::SixDofPlannerBase*. Créez une méthode *void grow(size_t nbNodes, size_t k, float dq, size_t maxSamples)* ayant pour but de construire / faire grossir la PRM. La signification des paramètres de cette méthode est la suivante :

- *nbNodes* : nombres de nœuds à effectivement ajouter à la PRM.
- *k* : nombre de plus proches voisins à prendre en compte pour tenter de relier une configuration à la PRM.
- *dq* : distance maximale entre deux configurations lors du test de chemin via la méthode locale.
- *maxSamples* : nombre maximum d'échantillons à tirer. Si ce nombre est atteint, la construction s'arrête même si moins de *nbNodes* nœuds ont été ajoutés à la PRM.

Pour vous faciliter la vie, la classe MotionPlanning::SixDofConfigurationGraph vous est fournie. Elle permet de stocker la structure de graphe de la PRM, d'effectuer des requêtes de k plus proches voisins et vous permet de savoir à quelle composante connexe un nœud du graphe appartient.

Question 3.2 : Programmez la méthode *plan* qui utilisera la PRM construite par l'appel à la méthode *grow* pour planifier un chemin. Le paramètre *radius*, pour l'utilisation d'une PRM, ne sera pas utilisé car ce dernier n'a de signification que pour le planificateur RRT. La méthode *plan* utilisera un algorithme A* sur le graphe de la PRM afin de planifier un chemin. Rappelez vous que si la source et la cible du chemin ne peuvent pas se connecter à la même composante connexe, aucun chemin en peut être trouvé.

Question 3.3 : Testez votre algorithme en décommentant le cas 1 dans la méthode computePlan de la classe SIAA_TP4_MotionPlanning. Si l'algorithme échoue lors de la planification, vous pouvez faire grossir la PRM à chaque échec jusqu'à ce que la source et la cible du chemin soient dans la même composante connexe.