Alexis Dupuis
Corentin Chauvin-Hameau

SYMOU – RRT pour un robot (1, 1)
Note explicative sur le code

Rapide présentation du problème

Le but est ici de générer une trajectoire entre deux points pour un robot mobile de type (1, 1). L'environnement concerné peut contenir des obstacles, et l'algorithme utilisé est RRT.

Structure du programme

Le code est composé de trois fichiers Python :

- **map_utilitary.py** : contient des fonctions permettant essentiellement de charger une image depuis un fichier image.
- **a_star.py** : implémentation de l'algorithme A*
- **RRT.py** : le fichier principal contenant tout ce qui traite de la résolution du problème.

map_utilitary.py

La stratégie pour représenter l'environnement est d'utiliser une image dans laquelle la couleur de chaque pixel représente un état : obstacle, vide, départ ou arrivée. Nos deux ordinateurs n'ayant pas openCV, cette carte est convertie en un fichier texte dans lequel ce sont des caractères qui représente les états.

La fonction *loadImage* charge une image dans une matrice openCV. Elle peut être convertie dans un fichier texte à l'aide de *saveMapAsText*. *loadMapFromText* charge la carte à partir d'un fichier texte. Enfin, *imageToText* permet de faire d'un coup ce que font les trois fonctions précédentes.

a_star.py

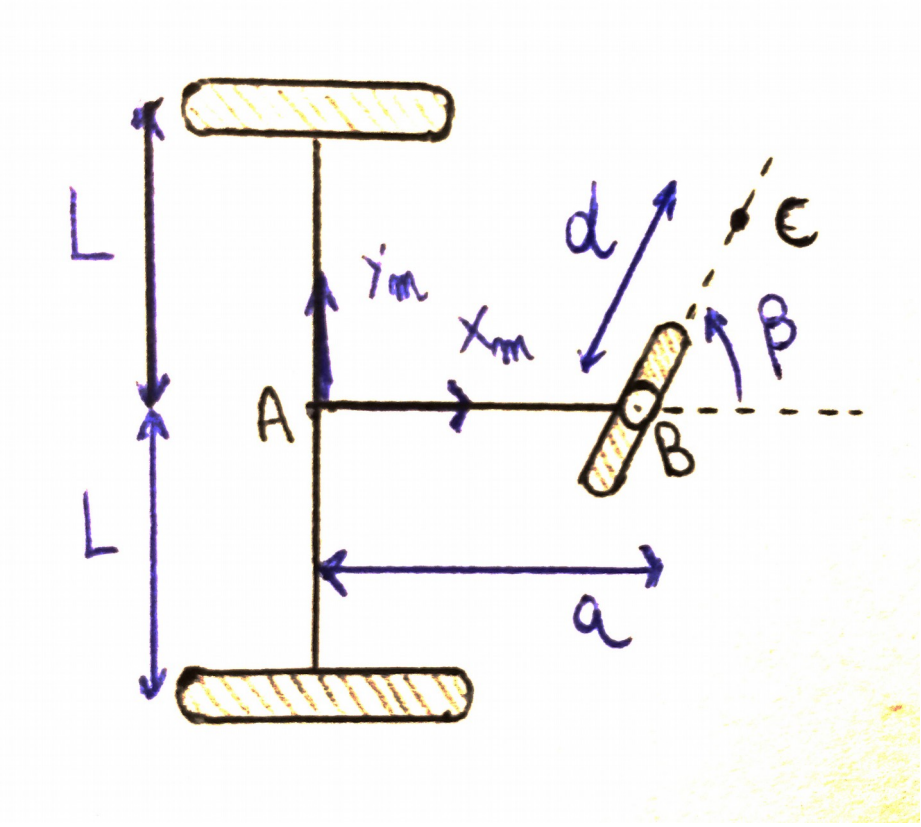
Les nœuds des arbres utilisés pour l'algorithme RRT sont représentés par des objets Python *Node*. Chacun de ces nœuds possède en attributs une liste de taille 4 *q* représentant la configuration du nœud, ainsi qu'une liste *neighbors* contenant les nœuds voisins. Ainsi, on n'utilise pas de matrices d'adjacence pour stocker les relations entre noœuds, ce qui rend d'une part le parcours de l'arbre rapide, et d'autre part ne prend pas de place en mémoire.
Un arbre est lui représenté par une liste de *Node*. Ceci est uniquement dans un but de stockage. En effet, pour parcourir l'arbre, il suffit de connaître sa racine (un objet *Node*) et de regarder dans sa liste de voisins.

L'algorithme A* est intégralement contenu dans la fonction *a_star* qui est une copie quasi-conforme de l'implémentation de Wikipédia :
https://en.wikipedia.org/wiki/A*_search_algorithm#Pseudocode.
La fonction *reconstructPath* permet de reconstruire le chemin allant du départ à l'arrivée.
La fonction *h* permet à la fois d'estimer la distance d'un nœud à l'arrivée, et de mesurer la distance entre deux nœuds voisins. Il s'agit simplement de la distance euclidienne dans l'espace des configurations.
Le coût *g* de chaque nœud par rapport au départ est représenté comme un attribut de l'objet *Node*.

RRT.py

La fonction principale est ici *buildRRT* qui applique l'algorithme RRT. Partant de deux nœuds, elle construit alternativement deux arbres jusqu'à les avoir merge un certains nombres de fois ou avoir dépassé le nombre d'itération maximum.
randomNode se charge de tirer q_rand, de trouver q_near le plus proche dans l'arbre en cours, puis de générer la commande pendant un certain temps (cf annexe) pour fabriquer q_new si aucun obstacle n'a été rencontré.
extendRRT permet d'ajouter q_new à l'abre en cours, et d'ajouter les liens de voisinage à q_new et q_near.
Enfin, *displayTree* affiche à la fin les deux arbres ainsi que le chemin trouvé par A* s'il existe (il y a un bug de couleurs si le chemin n'est pas valide).

Annexe : cinématique et "sugar tracking"



Si on omet la rotation de chacune des roues, on peut écrire $q=(x,y,\theta,\beta)$. Si on déroule le cours de VETIN, on peut trouver que le vecteur de commande s'écrit $u=(u_m,u_s)=(\dot{m}x,\dot{\beta})$.
Ensuite, on veut que notre point C ait une certaine vitesse $v_d=(v_x,v_y)=\frac{v_{max}}{\sqrt{(x_G-x_C)^2+(y_G-y_C)^2}}(x_G-x_C,y_G-y_C)$ vers le point objectif G (en l'occurrence q_rand).

On a donc l'équation suivante : $\begin{cases}\dot{m}x\cos\theta-a\dot{\theta}\sin\theta-d(\dot{\theta}+\dot{\beta})\sin(\theta+\beta)=v_x\\ \dot{m}x\sin\theta+a\dot{\theta}\cos\theta+d(\dot{\theta}+\dot{\beta})\cos(\theta+\beta)=v_y\\ \dot{m}x=\frac{L}{\tan\beta}\dot{\theta}\end{cases}$. En isolant $\dot{\theta}$ et $\dot{m}x$, on obtient le système inversible (presque tout le temps) suivant :

$$\begin{pmatrix}A & B\\ C & D\end{pmatrix}\begin{pmatrix}\dot{\theta}\\ \dot{\beta}\end{pmatrix}=\begin{pmatrix}v_x\\ v_y\end{pmatrix} \text{ avec } \begin{matrix}A=\frac{L}{\tan\beta}\cos\theta-a\sin\theta-d\sin(\theta+\beta)\\ C=\frac{L}{\tan\beta}\sin\theta+a\sin\theta+d\cos(\theta+\beta)\end{matrix} \text{ et } \begin{matrix}B=-d\sin(\theta+\beta)\\ D=d\cos(\theta+\beta)\end{matrix}$$

On en déduit alors $\begin{pmatrix}\dot{\theta}\\ \dot{\beta}\end{pmatrix}$ d'où on peut déduire $\begin{pmatrix}\dot{m}x\\ \dot{\theta}\end{pmatrix}$ que l'on intègre pour générer la trajectoire du robot.

Le seul problème peut que l'on peut avoir est quand $\beta=0$. À ce moment, on se retrouve avec $\dot{\beta}=\frac{\cos\theta v_y-\sin\theta v_x}{d}$ (le calcul n'est pas beau à voir, mais tout se simplifie miraculeusement) et on pose $\dot{m}x=v_{max}$. Dès lors, on arrive tout de même à faire l'intégration.