Compte rendu TP1 : Recherche de chemins dans un graphe avec un LegoRover

IBLED Mathilde et MARTIN Gautier
4AE-SE4

TP réalisé le 9 mars 2021

Encadrants :
Elodie Chanthery, Gwendoline Le Corre, Charles Poussot-Vassal

Table des matières

Introduction 3
I. Mise en place de la partie guidage 3
II. Mise en place de la partie Gestion de la trajectoire 4
III. Mise en place de la partie Gestion du plan 5
IV. Mise en place d'une nouvelle extension 6
Conclusion 6
ANNEXES 7

Introduction

L'objectif de ce TP est de mettre en place le guidage et le suivi autonome (sur plusieurs niveaux) d'un robot LegoRover le long d'une ligne noire sur piste. Le robot doit savoir trouver son chemin et aller d'un point A à un point x, sans jamais dévier du circuit. Nous mettrons en place un algorithme d'optimisation du chemin en fonction de critère prédéfini. Dans un dernier temps, nous ajoutons une fonction annexe de gestion de carburant virtuel.

I. Mise en place de la partie guidage

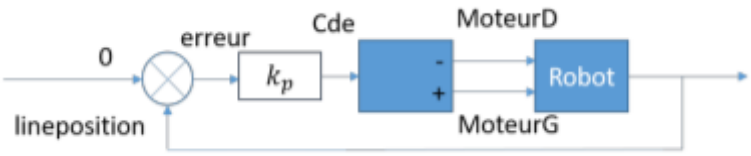

Figure 1 - Schéma de régulation

Le robot doit suivre la ligne noire à partir d'une commande de vitesse notée speed en degré par seconde. L'asservissement est réalisé en fonction de la position par rapport à la ligne noire (acquisition par capteur LineLeader). Pour pouvoir suivre la trajectoire et assurer les différents virages, la commande envoyée au moteurD doit être différente de celle du moteurG. Ainsi les moteurs peuvent ne pas tourner à la même vitesse. Pour tourner à gauche il faut que la vitesse du moteurD soit supérieure à la vitesse du moteurG et inversement. La programmation de cette régulation se trouve dans l'annexe (1).

Nous avons testé plusieurs valeurs de Kp sur le robot pour une vitesse constante de 300°/rad.

Kp	Résultats sur la trajectoire du robot
2	Le robot ne tourne pas
10	Le robot ne tourne pas complètement dans les virages
20	Le robot suit très bien la ligne
40	Le robot suit la ligne mais oscille beaucoup

Ensuite, nous décidons de garder Kp=20 car c'est le meilleur compromis. Nous faisons maintenant varier la vitesse du robot.

Vitesse (°/rad)	Résultats sur la trajectoire du robot
20	Le robot n'avance pas
300	Le robot suit précisément la trajectoire
400	Le robot a des difficultés à suivre la trajectoire
470	Le robot a d'autant plus de difficultés
500	Le robot est à la limite de ne plus suivre la trajectoire imposée

Concernant la valeur de Kp, il est clair qu'une valeur trop faible mène à un temps de réponse trop important pour le robot, qui n'arrive pas à suivre la trajectoire. Lorsque nous testons des valeurs hautes (au-delà de 30), un dépassement se fait ressentir et le robot en arrive même à des oscillations pour le test Kp=40.
Pour ce qui est de la vitesse, une valeur trop basse ne permet tout simplement pas au robot de se mouvoir. Nous augmentons la vitesse jusqu'à trouver la valeur optimale qui permet au robot d'aller le plus vite possible sans avoir de difficultés pour suivre la trajectoire. En effet, pour des valeurs trop hautes, ce dernier va trop vite dans les virages pour que l'asservissement ait le temps de se faire.
Ainsi, toutes ces observations nous mènent à choisir les caractéristiques optimales : Kp égale 20 et la vitesse égale 300°/rad.

II. Mise en place de la partie Gestion de la trajectoire

Notre robot suit bien les lignes noires, nous allons instaurer un système de suivi de trajectoire, avec pour plateforme de test la piste suivante :

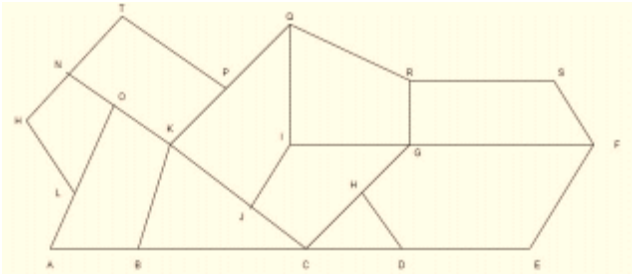

Figure 2 - Schéma représentatif de la piste

Notre premier test consiste à faire circuler le robot du point A au point Q, pour cela il doit passer par le point B et le point K.
Nous utilisons la fonctionnalité : *listOfOrders.add (new Order(int angle, int distance, int speed))* pour créer les trajectoires entre les différents points du chemin voir annexe(2). Nous avons mesuré les angles entre l'axe de la voiture et le chemin à suivre. L'angle indiqué dans le code doit être négatif pour tourner à gauche et positif pour tourner à droite.

Notons dans cette expérimentation qu'il est délicat d'entrer une trajectoire à la main pour le robot. Une première expérimentation est d'abord nécessaire pour comprendre qu'un angle négatif correspond à une rotation vers la gauche du robot. Il faut ensuite être relativement précis dans la mesure des angles et distances entre les points quand bien même le robot arrive à corriger sa trajectoire grâce à son asservissement.
In fine, nous réussissons à rejoindre le point Q depuis le point A malgré une légère déviation au croisement.
La démonstration est visible sous le lien suivant : https://youtu.be/PM_cJSTN1EA

III. Mise en place de la partie Gestion du plan

Nous allons maintenant mettre en place le suivi autonome du parcours. Pour cela, nous mettons en place deux critères d'optimisation, tout d'abord la distance puis le temps, voir annexe (3) et (4).
Pour se faire nous mettons en place l'algorithme de Dijkstra basé sur la théorie des graphes et vu en cours peu avant le TP. Le graphe peut être représenté avec la figure 2, les sommets étant les points (A à T), les poids des arcs peuvent être modélisés par la distance qui sépare les points. Cet algorithme nous permettra donc d'optimiser la distance entre deux points demandés. Nous utilisons l'algorithme de Dijkstra dans ce cas précis uniquement car les poids des arcs (distances) sont positifs et que nous avons des circuits. C'est un algorithme de plus court chemin.

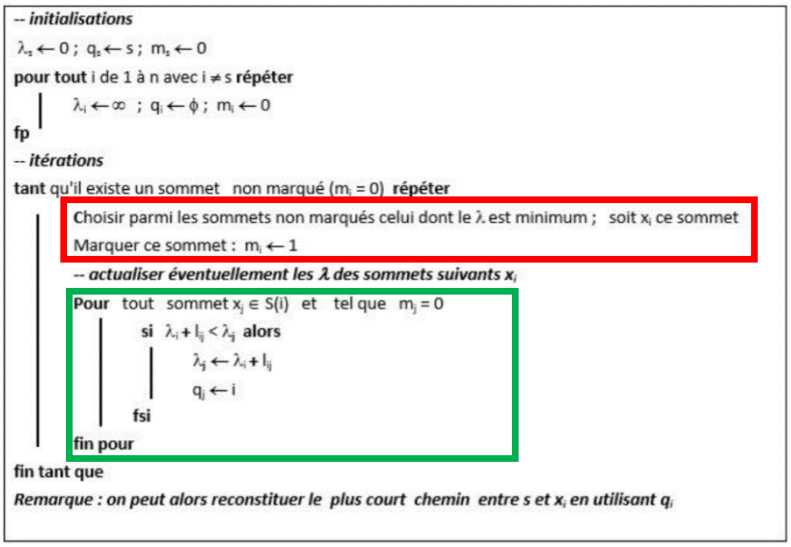

Figure 3 - Algorithme de Dijkstra

Le sommet de référence est le plus récemment marqué, de cette base (ou point de départ pour initialiser), de cette base nous recherchons le sommet non marqué le plus proche : nœud_retenu dans le code en annexe (5).

À partir de ce nœud retenu, nous regardons tous les sommets non marqués qui lui sont adjacents, si pour atteindre ce sommet, le chemin est plus court en passant par le nœud retenu que par le sommet de référence, on actualise le chemin le plus court (voir annexe (6)). L'algorithme s'arrête quand tous les sommets sont marqués.

Nous faisons le test sur le chemin de A vers K, il y a plusieurs possibilités de chemins, les plus courts visuellement sont A, B, K (90 cm) ou A, O, K (83 cm). Sans critère de distance, le robot choisit naturellement le chemin ABK. En implantant notre code, le robot choisit maintenant le chemin AOK, effectivement plus court : https://youtu.be/1Tbmx4hnWJ0

Pour tester le critère de temps, nous commandons au robot d'aller du point A au point F. De base, pour atteindre ce point, le robot emprunte notamment les segments CD-DE-EF. Nous baissons drastiquement la vitesse sur l'arc DE. Après implémentation de l'algorithme avec pour critère la vitesse, arrivé au point C, le robot n'emprunte plus CD-DE-EF, maintenant trop long, mais préfère passer par CG-GF : https://youtu.be/e5lsETmFdjg

IV. Mise en place d'une nouvelle extension

Les parties principales de ce TP ayant été traitées avant la fin de l'heure, nous avons eu l'opportunité de mettre en place une extension possible de notre algorithme. Pressés par le temps, nous nous sommes dirigés vers la gestion d'un carburant virtuel pour le robot. L'algorithme est simple, nous donnons une valeur de carburant de base à notre robot (ici 1000 pour l'exemple), à chaque passage entre deux points séparés d'une distance x, ce même x est retiré de la valeur carburant du robot. Avant que ce dernier ne commence la transition entre deux points, il vérifie son niveau de carburant, si celui-ci est inférieur à 300 le robot ne prend aucun risque et s'arrête à sa position actuelle. Le code se retrouve dans l'annexe (7).
Une amélioration possible est de comparer la distance entre les deux points suivants et le niveau de carburant, si ce dernier est plus grand alors le robot peut continuer son chemin.

Dans la vidéo suivante, nous demandons au robot de parcourir le chemin le plus court entre A et E, la distance entre les deux dépassant bien entendu un mètre (1000 mm). Dans la première vidéo, le code de gestion du carburant n'est pas embarqué, le robot arrive bien à destination. Dans la deuxième version, avec gestion du carburant, arrivé au point C, le niveau est passé sous la barre des 300, le robot ne rejoindra donc jamais le point D et s'arrête en ce point.

Lien de la vidéo concernant la gestion du carburant : https://youtu.be/_yPrPezpyr8

Conclusion

La grande idée autour de ce TP était d'observer les différents niveaux de commande pour un système autonome (voir figure 4 ci-dessous). Au fur et à mesure de l'avancement, nous tendons alors vers un système de plus en plus complexe. D'un simple suivi de ligne (niveau 0) nous arrivons à un robot qui est capable de décider de sa propre trajectoire en fonction d'un critère d'optimisation (niveau 2). La partie gestion de plan nous aura également permis de mettre en place l'algorithme de plus court chemin, l'algorithme de Dijkstra vu en cours avant ce TP. Nous aurons pu constater son efficacité et sa facilité d'implémentation dès lors que le robot a la piste en mémoire.
Malgré notre efficacité pendant le TP nous n'aurons pas eu beaucoup de temps à consacrer aux extensions possibles. Nous avons dû nous tourner sur l'extension la plus basique (gestion d'un carburant limité) alors que la « tournée du facteur » était probablement la plus enrichissante.

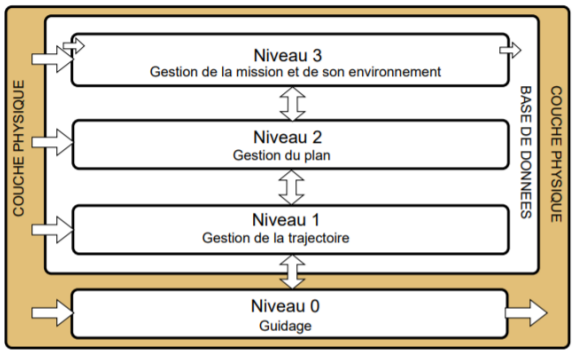

Figure 4 - Niveaux de commande dans une architecture d'autonomie

ANNEXES

(1) Code de la programmation de la régulation des moteurs

```
erreur=linePosition;
Cde=kp*erreur;
LCD.drawString("linepos6 = " + Cde, 0,0);
motorD.setSpeed(speed-Cde); // speed - value in degrees/sec
motorG.setSpeed(speed+Cde); // speed - value in degrees/sec
```

```
//test sur la ligne courbe
listOfOrders.add(new Order(0, 2000, 360)); // tester le suivi de ligne
```

(2) Mise en place du suivi d'un chemin de A à Q

```
//test qui fait A, B, K, Q
listOfOrders.add(new Order(0, 340, 300));
listOfOrders.add(new Order(-75, 500, 300));
listOfOrders.add(new Order(35, 800, 300));
```

On indique au robot l'angle, la distance, et la vitesse de sa trajectoire

(3) Critère d'optimisation : la distance

```
public void SetArc(Point noeud_1, Point noeud_2, int vitesse) {

    //On attribue une ligne et une colonne pour le couple de point donné
    String stringcol = noeud_1.getName();
    char car_col = stringcol.charAt(0);
    int col = (int) car_col - 65;

    String stringli = noeud_2.getName();
    char car_li = stringli.charAt(0);
    int li = (int) car_li - 65;

    this.SpeedMap[li][col] = vitesse;
    this.SpeedMap[col][li]= vitesse;
    // calcul de la distance entre ces deux point:
    double distance = Math.sqrt(Math.pow(Math.abs(noeud_1.getX() - noeud_2.getX()), 2.0) + Math.pow(Math.abs(noeud_
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // définition du critère du Dijkstra
    this.Map[li][col] = distance ;
    this.Map[col][li] = distance ;
```

Entouré en rouge : le critère d'optimisation choisi

(4) Critère d'optimisation : le temps

```
//permet de mettre un poids entre les noeuds 1 et 2 (commence Ã  1)  avec A=1.. et de mettre la vitesse sur l'arc
public void SetArc(Point noeud_1, Point noeud_2, int vitesse) {

    //On attribue une ligne et une colonne pour le couple de point donné
    String stringcol = noeud_1.getName();
    char car_col = stringcol.charAt(0);
    int col = (int) car_col - 65;

    String stringli = noeud_2.getName();
    char car_li = stringli.charAt(0);
    int li = (int) car_li - 65;

    this.SpeedMap[li][col] = vitesse;
    this.SpeedMap[col][li]= vitesse;
    // calcul de la distance entre ces deux point:
    double distance = Math.sqrt(Math.pow(Math.abs(noeud_1.getX() - noeud_2.getX()), 2.0) + Math.pow(Math.abs(noeud_1.getY() - noe

    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // définition du critère du Dijkstra
    this.Map[li][col] = distance/vitesse ;
    this.Map[col][li] = distance/vitesse ;


}
```

(5) Algorithme de Dijkstra : partie recherche du nœud le plus proche

```
// recherche du noeud de valeur minimale non encore exploré
for (int i = 0; i < carte[noeud_init].length; i++) {
     if (tab_value[i]<valmin && tab_bool[i]==false) {
         valmin=tab_value[i];
         noeud_retenu=i;
     }
}
```

Détermination du sommet adjacent non marqué le plus proche -> nœud_retenu

(6) Algorithme de Dijkstra : partie actualisation du chemin le plus court

```
for (i = 0; i < map[this.noeud_init].length; i++) {
    if (tab_value[noeud_retenu]+map[noeud_retenu][i]<tab_value[i]) {
        tab_value[i]=tab_value[noeud_retenu]+map[noeud_retenu][i];
        tab_noeuds[i]=noeud_retenu;
    }
}
```

On évalue tous les sommets adjacents non marqués du nœud retenu puis on actualise le chemin le plus cours en conséquence

(7) Programmation de l'extension dans la fonction ComputeOrders()

```
int Carburant = 1000;

Carburant = Carburant - distance;
 if (Carburant<300) {
     System.out.println("Plus de Carburant");
 break;
 }
```

Mise à jour de la valeur carburant à chaque trajet entre deux sommets, si le carburant est inférieur à 300 avant d'atteindre le prochain sommet, le robot s'arrête