TP n°1 - Recherche de chemin dans un graphe : LegoRover

Isaure Badier - Raphaël Bernard - Clémence Arnal

31 janvier 2022

Table des matières

1 Introduction

Le but de cette manipulation est de commander un robot progressant grâce à deux roues motrices. La commande du robot se fait à plusieurs niveaux. D'abord nous allons programmer le robot pour qu'il suive une ligne noire dessinée au sol. Ensuite, nous testerons la gestion de la trajectoire du robot en lui demandant d'exécuter des ordres que nous lui donnerons. Enfin la dernière partie concernera la gestion du plan avec calcul du plus court chemin en terme de distance puis de temps.

2 Guidage et suivi de ligne noire

2.1 Programmation et régulation

L'objectif de cette partie est de programmer le robot pour qu'il suive une ligne noire disposée au sol. Pour cela, le robot est muni d'une ligne de capteurs de luminosité qui renvoient un 1 devant du noir. Nous allons mettre en place une boucle de régulation proportionnelle autour d'une consigne en position, comme le présente la figure 1.

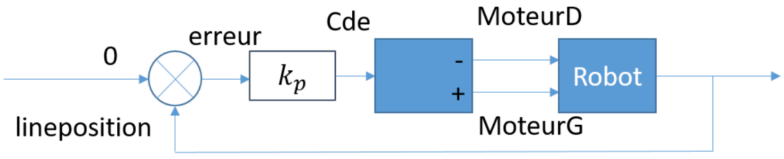

FIGURE 1 – Boucle de régulation proportionnelle

Pour cela, on utilise la fonction `CapteurLigne()` qui nous renvoie un 0 si le robot est bien positionné, mais un nombre négatif s'il est trop à droite et positif s'il est trop à gauche. Comme nous souhaitons mettre en place une régulation proportionnelle, la commande se calcule simplement avec $Cde = k_p \times erreur$, l'erreur étant la différence entre 0 et *LinePosition*. Enfin, cette commande est appliquée sur chaque roue, en l'additionnant à la vitesse pour la roue droite et en la soustrayant pour la roue gauche. Le code est présenté sur la figure 2.

```
while(distanceParcourue < (distance - distanceApproach)){

    Miss.fetchSample(sample,0); //lecture du senseur
    aux = sample; //auxiliaire par la lecture
    linePosition = this.CapteurLigne(aux); // lecture du senseur


    // Boucle de régulation Proportionnelle
     erreur = - linePosition ;
     Cde = erreur * kp ;
     motorD.setSpeed(speed + Cde);
     motorG.setSpeed(speed - Cde);
```

FIGURE 2 – Code Java pour la boucle de régulation proportionnelle

Ainsi, si le robot est positionné à gauche de la ligne, `CapteurLigne()` renvoie un résultat positif. La commande va donc être négative : le moteur gauche va accélérer tandis que le droit va ralentir. Le robot va donc tourner vers la droite et se rapprocher de la ligne. Si cette opération est répétée à une fréquence suffisamment élevée, le robot va pouvoir suivre la ligne noire en rectifiant sa position. Pour une vitesse de rotation de 300 degrés par seconde, on obtient un résultat qui correspond à nos attentes, comme montré sur la vidéo ci-dessous (lien ici).

2.2 Variations du facteur k_p et de la vitesse

En ayant maintenant un algorithme fonctionnel, nous pouvons tester l'influence du facteur k_p et de la vitesse sur le comportement du robot. En augmentant la vitesse tout en laissant le k_p à 10, nous pouvons remarquer un décrochage du robot dans les virages, comme sur la vidéo au lien suivant (ici). Le robot va trop vite pour permettre à l'algorithme de s'exécuter et de rectifier la position avant qu'il ne quitte définitivement la ligne. La valeur limite de vitesse se trouve aux alentours de 350 degrés par seconde.

Lorsque l'on augmente le k_p, on voit apparaître des oscillations plus saccadées dans le comportement du robot, il rectifie plus vite sa position par petits a-coup. En revanche, cela permet d'augmenter la vitesse au delà du seuil trouvé au-dessus. Voici en table 1 les résultats que nous avons pu obtenir.

k_p	5	10	40
Vitesse limite de décrochage (°/s)	345	350	400

TABLE 1 – Mesures expérimentales pour trouver la vitesse limite en fonction du k_p

3 Gestion de la trajectoire

Pour tester la gestion de la trajectoire, nous avons simplement commandé le robot en terme de distance à parcourir, d'angle de rotation, et de vitesse d'avancement. Pour réaliser le parcours du point A au point Q en passant par les points C, J et I, nous avons effectué des mesures sur la carte, présentées dans le tableau 2.

Parcours	A → C	C → J	J → I	I → Q
Distance (mm)	1100	250	390	490
Angle (°)	0	-145	90	-50

TABLE 2 – Mesures des angles et distances sur la carte

Nous avons ensuite programmé le robot en lui donnant de nouvelles missions, chacune correspondant à une partie du parcours. Le code est présenté en figure 3 et la vidéo d'exécution est disponible ici.

```
//test qui fait a,b,c,j,i,q
listOfOrders.add(new Order(0, 1100, 360)); // a vers c
listOfOrders.add(new Order(-145, 250, 360)); // c vers j
listOfOrders.add(new Order(90, 390, 360)); // j vers i
listOfOrders.add(new Order(-50, 490, 360)); // i vers q

MissionPC mission = new MissionPC(listOfOrders);
mission.start();
```

FIGURE 3 – Code Java pour la gestion de la trajectoire

4 Gestion du plan

Pour mettre en place la gestion du plan, nous utiliserons l'algorithme de Dijkstra, un algorithme de recherche qui consiste à calculer la valeur d'un critère (minimal ou maximal). Dans cette partie, nous cherchons à faire choisir au robot le plus court chemin possible en fonction du point de départ et d'arrivée fourni.

4.1 Définition du poids des arcs

Pour calculer la valeur du plus court chemin, il faut d'abord définir le poids des arcs entre les points. Pour cela, nous définissons le critère de notre algorithme de recherche dans la fonction `SetArc()`. Notre objectif étant de parcourir le plus court chemin, notre critère est la distance (voir Figure 4).

```
// définition du critère du Dijkstra - critère de distance
this.Map[li][col] = distance;
this.Map[col][li] = distance;
```

FIGURE 4 – Fonction **SetArc()**

4.2 Algorithme

Une fois que nous avons défini le poids des arcs, nous suivons l'algorithme de Dijkstra pour parcourir le graphe. A chaque itération, la première chose à faire est de trouver le point non marqué (c'est-à-dire où le robot n'est pas encore passé) dont le critère est minimum. Cette opération est réalisée par la fonction `TrouveMin()`, présente en figure 5. Il s'agit simplement de parcourir le tableau `tab_value`, qui contient les critères de chaque point, et de rechercher le minimum.

```
public int TrouvMin() { //retourne le sommet le plus proche du sommet initial

    int noeud_retenu = -1;
    double[][] carte = this.map.getMap();
    double valmin = Double.MAX_VALUE;

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

    if (noeud_retenu != -1) {
        tab_bool[noeud_retenu] = true;
        this.fini--; //on dÃ©crÃ©mente notre variable pour savoir quand on aura passÃ© en revue tous les noeuds

    } else {
        System.out.println("------->TrouvMin : pas de solution ");
    }

    return noeud_retenu;
}
```

FIGURE 5 – Fonction **TrouveMin()**

Une fois le sommet avec le critère minimum identifié, il faut marquer le sommet pour indiquer au robot qu'il a déjà étudié ce point. Il suffit de mettre à jour la case correspondante au sommet dans le tableau `tab_bool`, ce qui est fait également sur la figure 5.

La prochaine étape est de recalculer le critère des points non marqués pour savoir si le robot doit ou non changer de chemin. Pour chaque points non marqué, on additionne le critère du point identifié (renvoyé par `TrouveMin()`) et le poids de l'arc qui sépare ce poids et le non marqué. Si l'on trouve un nouveau critère qui est plus faible que précédemment, cela signifie que le chemin pour aller au point

non marqué est plus court en passant par le point identifié. Dans ce cas, il faut mettre à jour le tableau `tab_noeuds`. Le code est présenté sur la fonction `MajCcum()`, sur la figure 6.

```
// mise à jour de tab_value : tableau des valeurs de critère cumulées
public void MajCcum(int noeud_retenu) {
    int i;
    double aux ;
    double[][] map = this.map.getMap();

    for (i = 0; i < map[this.noeud_init].length; i++) {
        // si le critère du noeud i en passant par le noeud retenu est meilleur alors on change de chemin
        //A compléter
        // actualize tab_value : additionne critère du point retenu et la distance (poids) de l'arc entre le point i et le poinbt retenu
        if (tab_bool[i] == false) {
            aux = tab_value[noeud_retenu] + map[noeud_retenu][i] ;
            if (aux < tab_value[i]) {
                tab_value[i] = aux ;
                tab_noeuds[i] = noeud_retenu ;
            }

        }
    }
}
```

FIGURE 6 – Fonction **MajCcum()**

4.3 Test de l'algorithme en utilisant le critère de distance (plus court chemin)

Après avoir complété l'algorithme sur notre fichier Eclipse et dé-commenté la ligne qui permet de lancer l'exécution de l'algorithme de Dijkstra, notre robot gère son plan et se déplace en utilisant le plus court chemin. En effet, en donnant au robot l'ordre d'aller du point A au point N, il calcule le plus court chemin et se rend au point d'arrivée comme on peut le voir sur la vidéo (lien ici).

Sur la figure 7, voici ce que donne l'exécution du programme sur la console d'Eclipse. Nous pouvons voir que l'algorithme a calculé un critère d'environ 93 cm. En mesurant sur la carte la distance entre les différents points, nous arrivons à une distance totale de 97 cm entre A et N. En prenant en compte les erreurs de mesures, nous savons que notre algorithme fonctionne et que le critère correspond bien à la distance entre les points.

```
======== Calcul du Dikjstra ======
critere final: 933.744093405297
nombre de points visités: 4
valeur: A
valeur: L
valeur: O
valeur: N
Fin Dij
C'est parti !
```

FIGURE 7 – Message d'exécution sur la console

5 Conclusion

Malgré notre manque de connaissance en programmation orientée objet et en parcours de graphe, nous avons pu mener à bien une bonne partie de ce TP. La première partie, sur le guidage et le suivi de ligne noire, permet de prendre en main le robot et le logiciel Eclipse sur un algorithme assez simple. La deuxième partie, sur la gestion de la trajectoire, est rapide et constitue une initiation au langage de programmation orientée objet. Enfin, la dernière partie sur le parcours de graphe est la plus intéressante, tout en étant la plus compliquée. Les algorithmes ne sont pas simples à comprendre, surtout en ayant aucune connaissances sur les graphes. Néanmoins, nous avons un algorithme fonctionnel pour un parcours de graphes avec critère de distance. Nous n'avons en revanche pas eu le temps d'utiliser un critère de temps ou de réfléchir aux extensions possibles. Le TP est donc très intéressant, en plus d'être assez ludique, mais manque peut-être d'un peu de temps.