Rapport TP1 Automatique

Recherche de chemins dans un graphe
LegoRover

Vitrat Romain Bourlot Xavier

4AE-SE, Binôme 2
30 mai 2020

Table des matières

1 Introduction **1**

2 Guidage **1**
2.1 Tests de vitesse . . . 1
2.2 Tests de gain k_p . . . 2

3 Gestion de la trajectoire **2**

4 Gestion du plan **2**
4.0.1 Définition d'un graphe formalisant le sujet . . . 2
4.0.2 Plan de route par une méthode de "depth first search" . . . 3
4.0.3 Plan de route par une méthode de Dijsktra . . . 3
4.0.4 Choix du critère . . . 3

5 Extension **4**

6 Conclusion **4**

A Annexes **5**
A.1 Graphe vitesse . . . 5
A.2 Liens vidéos . . . 5

1 Introduction

L'objectif de la manipulation est de contrôler un robot type rover (2 roues motrices et une roue libre à l'avant) dans le but de le faire suivre une ligne, dans un premier temps. Dans un second temps, nous ferons évoluer le robot pour le faire suivre un parcours défini par des sommets et des distances. Enfin, nous envisagerons d'utiliser l'algorithme de Dijsktra pour établir un plus cours chemin entre 2 points du plan. La finalité serait d'optimiser l'algorithme pour que le robot ne repasse jamais par le même point lors d'un parcours à plusieurs arrêts.

2 Guidage

Le robot doit d'abord être capable de suivre une ligne. On dispose d'un capteur linéaire fournissant une valeur signée représentative de l'écart à la ligne. On utilise cette valeur pour maintenir l'alignement du robot, en commandant les deux moteurs, grâce à un correcteur proportionnel (cf Fig.1).

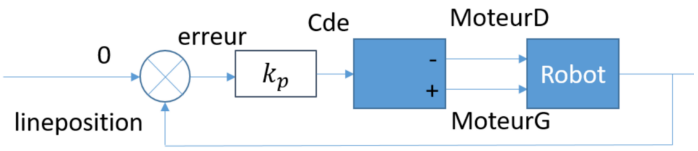

FIGURE 1 – Schéma bloc de l'asservissement de position

On contrôle la vitesse différentielle entre les deux moteurs pour changer l'angle du robot. A cette différence s'ajoute la vitesse en ligne droite du robot, qui est la même pour les deux moteurs.
Le code Java implémenté est le suivant :

```
motorG.setSpeed(speed+kp*lineposition);
motorD.setSpeed(speed-kp*lineposition);
```

Listing 1 – Code du guidage

On observe que lorsque le robot est trop à droite, $lineposition > 0$, donc on augmente la vitesse à gauche et on diminue la vitesse à droite pour compenser.
Afin de valider ce correcteur, il nous faut étudier l'influence des deux paramètres : vitesse et k_p.

2.1 Tests de vitesse

On réalise des tests de vitesse pour $k_p = 10$. Le code utilisé est le suivant (on effectue des aller-retour avec accroissement de la vitesse jusqu'a décrochage) :

```
for(int i=100;i<=500;i+=100){
    //tourne de 180°, puis avance de 2m avec une vitesse allant de 100    500 °/s
    listOfOrders.add(new Order(180,2000,i));
    listOfOrders.add(new Order(180,2000,i));
}
```

Listing 2 – Code de test en vitesse, $k_p = 10$

Les observations qualitatives sont reportées dans le tableau suivant :

Vitesse	Ligne droite	Virage
100	Stable	Peu d'oscillations
200	Stable	Quelques oscillations
300	Stable	A-coups en virage
400	Oscillations amorties	Oscillations franches
500	Critiquement stable	Perte trajectoire

TABLE 1 – Qualité de la régulation selon la vitesse

En conclusion, le choix d'une vitesse de 300 tr/s nous a semblé le plus pertinent, en effet, cela permettra au véhicule de suivre un chemin assez rapidement sans perdre la ligne ou trop osciller en virage.

2.2 Tests de gain k_p

De même, on effectue à vitesse constante (300 tr/s) des tests sur la valeur optimale de k_p, variant de 5 à 45.

k_p	Ligne droite	Virage
5	Stable	Perte ligne
10	Stable	Petits à-coups
15	Stable	Brusque dans ses changements de direction
25	Convenable	Brusque dans ses changements de direction
35	Critiquement stable	Brusque
45	Instable (Oscillations amplifiées jusqu'à décrochage)	Brusque

TABLE 2 – Qualité de la régulation selon le gain k_p

Pour la suite du TP, on prendra k_p=15 et une vitesse de $300°/s$.

3 Gestion de la trajectoire

Le but de la manipulation est ici de programmer manuellement un chemin à suivre pour le robot. On pourra également vérifier le bon fonctionnement du correcteur proportionnel en toute circonstance.
Les ordres sont constitués d'une rotation relative et d'une longueur à parcourir. On mesure sur le circuit les différents tronçons, afin de reconstituer une partie de la carte du circuit. On obtient les commandes suivantes :

```
1 //
2 listOfOrders.add(new Order(0,350,300));
3 listOfOrders.add(new Order(-70,500,300));
4 listOfOrders.add(new Order(35,850,300));
```

Listing 3 – Trajectoire définie manuellement de A à Q

Les commandes sont ajoutées à une liste d'ordres, que le robot exécutera de manière séquentielle. Nous n'avons pas relevé de problèmes particuliers sur cette partie du TP, il faut toutefois penser à bien positionner le centre de rotation du robot sur l'intersection pour garantir sa bonne orientation. Cela permet aussi au capteur de ligne de n'en percevoir qu'une à la fois.

4 Gestion du plan

Nous nous sommes à présent intéressés à gérer un plan de route pour le robot, et ceux en s'appuyant sur différentes méthodes et critères.

4.0.1 Définition d'un graphe formalisant le sujet

Voici la modélisation sous forme de graphe de la situation à laquelle est confronté le robot. Nous allons déterminer le meilleur critère dans une prochaine partie.

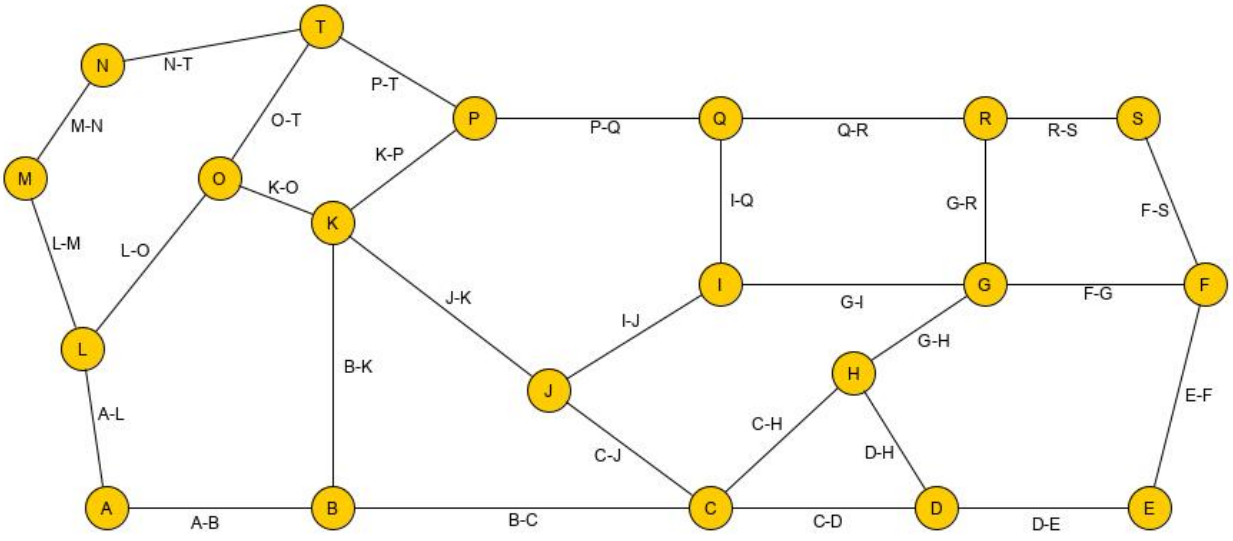

FIGURE 2 – Graphe représentant la situation à laquelle est confronté le robot, avec un critère de temps sur les arcs.

Cf A.1 pour le graphe avec un critère de vitesse.

4.0.2 Plan de route par une méthode de "depth first search"

```
// Fonction qui parcours la liste profond ment
    void DFSUtil(int v,boolean visited[],int last_node)
    {
        // Marque le sommet comme lu
        visited[v] = true;

        // Boucle utilisant la r currence pour parcourir tous les sommets.
        Iterator<Integer> i = liste[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n] || visited[n]==last_node)
                DFSUtil(n, visited);
        }
    }

    // Fonction permettant la travers e du graphe
    void DFS(int v)
    {
        boolean visited[] = new boolean[V];
        DFSUtil(v, visited);
    }
```

Listing 4 – Depth first search

Pour trouver un plus cours chemin grâce à cette méthode, on peut itérer plusieurs fois l'algorithme en changeant le 2nd point visité (décrémenter l'itérateur au lieu de l'incrémenter, partir du milieu, ...), il suffit ensuite de sommer les arcs en fonction du critère choisi.

4.0.3 Plan de route par une méthode de Dijsktra

L'algorithme a été complété de la manière suivante :

```

// recherche du noeud de valeur minimale non encore explor
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;

  }

}
```

Listing 5 – Recherche du noeud de valeur minimale non encore exploré

```
for (i = 0; i < map[this.noeud_init].length; i++) {
          //On met    jour la table si le chemin passant par le noeud marqu  est plus court que les
     pr c dents chemins enregistr s.
          if(tab_value[noeud_retenu] + map[noeud_retenu][i] < tab_value[i]){
              tabv_value[i] = tab_value[noeud_retenu] + map[noeud_retenu][i];
              tab_noeuds[i] = noeud_retenu;

          }

        }
```

Listing 6 – Mise à jour de la table si le chemin calculé à partir du nouveau noeud est plus court que celui précédemment enregistré

Une vidéo d'un parcours classique calculé par l'algorithme de Dijsktra est présente en annexe A.2.

4.0.4 Choix du critère

Un facteur important de performance lors de la recherche de plus court chemin est le choix d'un critère judicieux. Dans une première approche, on peut choisir la distance comme le critère à optimiser. Mais ce modèle ne tient pas compte d'éventuels ralentissements. Pour mettre en avant ce défaut, nous avons réalisé deux déplacements, du point B au point C, en utilisant deux critères différents, le premier de distance, et le deuxième de temps. Le segment BC comporte une limitation de vitesse définie logiciellement, rendant un détour par NOMPOINT ? plus judicieux pour le deuxième critère. Les critères de distance et de temps sont definis dans le code de la manière suivante :

```
//critere pour la distance
this.Map[li][col]=distance;
this.Map[col][li]=distance;
```

Listing 7 – Critère de distance

```
//critere pour optimiser le temps
this.Map[li][col]=distance/vitesse;
this.Map[col][li]=distance/vitesse;
```

Listing 8 – Critère de temps

Les vidéos illustrant ces codes sont accessibles en Annexe A.2.

5 Extension

Nous avons eu le temps de proposer une extension en réel de ce TP. Le but était de pouvoir élargir le nombre de points à parcourir. Notre implémentation ajoute une contrainte intermédiaire, de passer par le point A. Une application réelle de ce type de problème est l'ajout d'un arrêt dans un trajet calculé par GPS.
La méthode proposée ici pourrait être étendue à plusieurs points intermédiaires ordonnés. Le principe est simple, si on souhaite aller du point B au point C par exemple, on crée deux trajets $B \rightarrow A$ puis $A \rightarrow C$ et on applique l'algorithme de Dijkstra sur ces deux sous-trajets. Ensuite, on concatène les listes d'ordres pour définir le profil global de la mission.

```
//initialisation la carte
BuildPath.SetMap();
//selection des noeuds de debut et de fin via l'interface graphique
BuildPath.Selection();
//On retient le point final
Point temp=BuildPath.noeud_fin;
//On definit le point final du premier sous-trajet comme le point A (fixe pour le moment)
BuildPath.noeud_fin=BuildPath.tab_point[0];
//Calcul du sous-trajet optimal de Debut vers A
BuildPath.ComputeDijkstra();
try {
    BuildPath.ComputeOrder();
} catch (IOException e) {e.printStackTrace();}
//On stocke le sous-trajet
ArrayList<Order> listOffirstOrders=(ArrayList<Order>)(BuildPath.listOfOrders).clone();
//On creee le deuxieme sous-trajet de A vers Fin
BuildPath.noeud_init=BuildPath.tab_point[0];
BuildPath.noeud_fin=temp;
//Calcul du sous-trajet optimal de A vers Fin
BuildPath.ComputeDijkstra();
try {
    BuildPath.ComputeOrder();
} catch (IOException e) {e.printStackTrace();}
//On concatene les deux listes d'ordre pour obtenir le profil total de mission
Iterator<Order> itr = BuildPath.listOfOrders.iterator();
while(itr.hasNext()){
    listOffirstOrders.add(itr.next());
}
//execution de la mission
MissionPC mission = new MissionPC(listOffirstOrders);
mission.start();
```

Listing 9 – Extension

6 Conclusion

Pour conclure, le correcteur proportionnel nous a montré les limites de la méthode en fonction des différents cas d'utilisation, ce qui montre bien que certains systèmes sont stables par morceaux, et non pas partout. De plus, nous avons ensuite pu remarquer l'efficacité de la méthode de Dijkstra pour trouver un plus court chemin. L'amélioration que nous avons proposé permet de générer un parcours avec étape.

A Annexes

A.1 Graphe vitesse

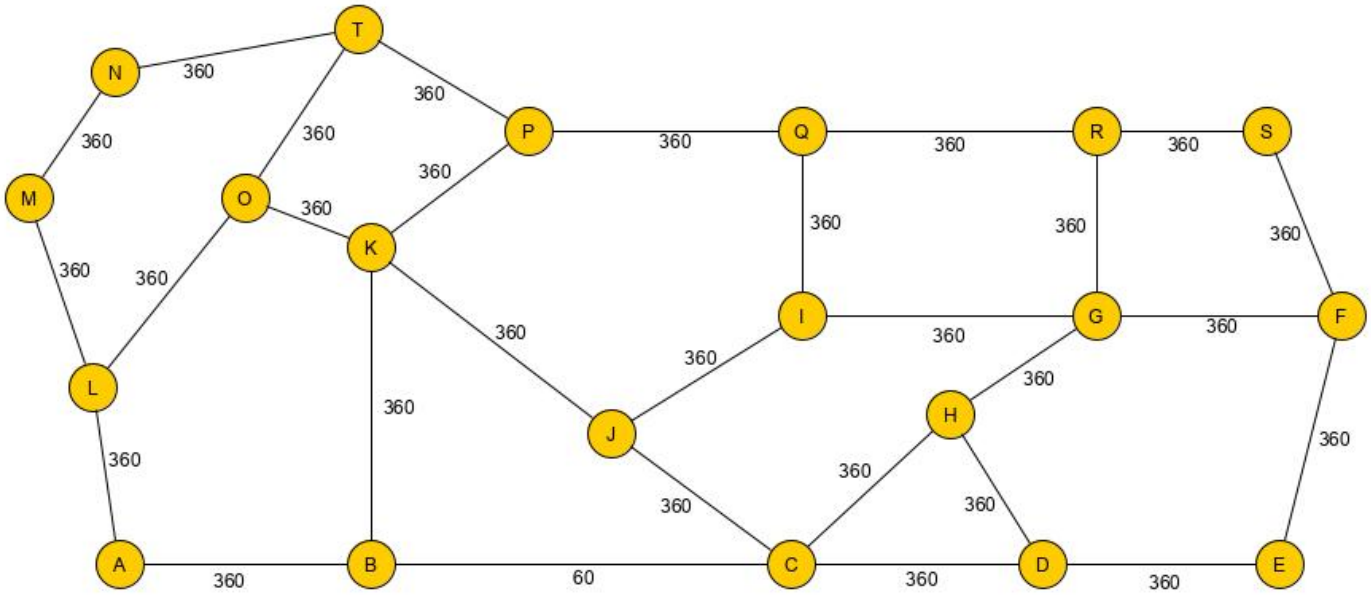

FIGURE 3 – Graphe représentant la situation à laquelle est confronté le robot, avec un critère de vitesse sur les arcs.

A.2 Liens vidéos

Vidéo d'un parcours quelconque calculé avec un algorithme de Dijsktra : Parcours classique

Vidéo du parcours B-C avec un critère d'optimisation de de la distance : Critère de distance

Vidéo du parcours B-C avec un critère d'optimisation de la durée : Critère de temps