# TP 1 - Recherche de chemins dans un graphe : LegoRover

## Introduction

*FIGURE 1 - Photo du robot Lego EV3*

Dans ce TP nous allons commander le robot Lego EV3 (voir *FIGURE 1*) par la conception d'une commande de guidage permettant au robot de suivre une ligne noire à différents niveaux d'automatisation. A l'aide du langage de programmation orienté objet Java nous allons d'abord mettre en place une commande de guidage avec un rebouclage sur la sortie et gain proportionnel. Ensuite, nous donnerons des directives de trajectoire sur une carte. Enfin, nous appliquerons les principes de l'algorithme de Dijkstra pour rendre le robot autonome par rapport à la gestion du plan.

## 1. Mise en place de la partie Guidage : niveau 0

La première étape de ce TP consiste à demander au robot rover de suivre une ligne noire. Le robot disposant d'un capteur Line Leader, il est capable de détecter quand il se trouve sur la ligne ou non. Ainsi, l'objectif de cette partie est de traiter l'information de ce capteur pour demander aux deux roues directrices du robot de réajuster la trajectoire.

### a. Mise en place de la boucle de régulation proportionnelle

Le guidage des roues du système se fait suivant la commande suivante:

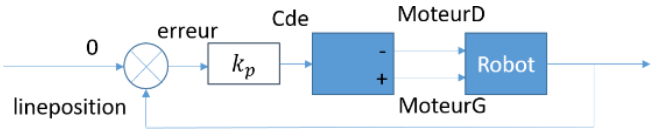


*FIGURE 2 - Schéma bloc d'une commande de régulation des roues de rover par retour sur la sortie et gain proportionnel Kp*

Il s'agit d'une régulation par retour sur la sortie avec gain proportionnel kp sur l'erreur entre la position sur la ligne noire captée par le robot et le référentiel (milieu de la ligne noire). L'erreur est amplifiée d'un gain kp pour agir sur les moteurs. L'action sur les moteurs droite et gauche est schématisée par les symboles plus(+) et moins(-) pour permettre au robot de tourner[1].

Alors, les commandes ne sont pas les mêmes en fonction de la roue contrôlée. En effet, les deux roues n'auront pas le même comportement pour un même objectif donné au rover.

[1] Voir code en gras dans l'annexe n°1

### b. Résultats des tests réalisés sur le rover

Nous commençons tout d'abord par calibrer notre robot dans la rubrique noir, et on lance le programme pilote.java pour que le robot soit en attente de mission (voir Annexe).

Nous réalisons une première série de tests pour vérifier le respect du suivi de courbe du robot pour différentes vitesses de rotation des roues imposées :

| *vitesse (°/s)* | *100* | *150* | *200* | *350* | *400* |
|---|---|---|---|---|---|
| *Comportement du robot* | *très bon suivi de la ligne, le robot a parfaitement suivi la ligne tout au long de sa trajectoire* | *très bon suivi de la ligne, le robot a parfaitement suivi la ligne tout au long de sa trajectoire* | *très bon suivi de la ligne, le robot a parfaitement suivi la ligne tout au long de sa trajectoire* | *suivi de ligne maintenu malgré quelques écarts* | *déviations quand angle de la courbe noire trop grand* |

Ainsi, la vitesse de rotation des roues influe sur la niveau de précision du robot. Plus on demande une vitesse importante et plus il est difficile d'avoir une information sur la position du robot par rapport à la ligne qui sera continue. Ainsi, pour un gain kp de 10, la plage de vitesses admissibles est de [100;350]m/s.

Nous jouons maintenant sur la valeur du gain proportionnel Kp placé dans la chaîne directe de régulation :

| *Kp\ vitesse (°/s)* | *350* | *400* |
|---|---|---|
| *5* | *Déviation* | *Déviation* |
| *10* | *ok* | *Déviation* |
| *20* | | *ok* |
| *30* | *ok* | *Dépassement au début de la ligne droite puis oscillations amorties avant de disparaître* |
| *40* | *Oscillations en fin de circuit au moment où la courbe devient une ligne droite* | |

Cette deuxième partie de test met en avant l'effet sur la stabilité d'un système de l'ajout de gain dans la chaîne directe de régulation d'un système. Nous constatons bien que mettre du gain nous permet d'asservir le comportement du rover mais également que lorsque le gain devient trop important, le système perd en stabilité : il connaît des oscillations. Ainsi, pour une vitesse entre 350 et 400 m/s nous favorisons un gain de 20.

## 2. Mise en place de la partie Gestion de la trajectoire : niveau 1

La deuxième étape de ce TP consiste uniquement à donner des directives d'angle de rotation, de distance et de vitesse de déplacement. Ainsi, nous serons capable de déplacer le robot d'un point x à un point y en lui donnant les mouvements à réaliser pour cela. Remarque : il ne s'agit pas ici de suivre une ligne noire mais de donner des directives mesurables au rover.

### a. Le code pour réaliser la mission d'aller du point A au point Q

Nous avons réalisé des mesures sur la carte afin de donner une liste d'ordre contenant l'angle de rotation, la vitesse et la distance (voir figure n° 3 ci-dessous).

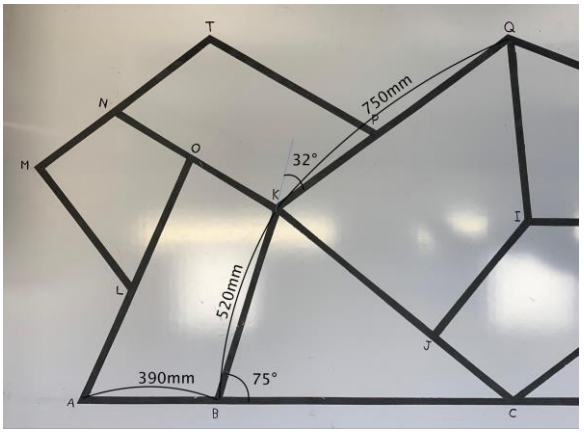


*FIGURE 3 - Représentations des mesures réalisées sur le plan*

Cela correspond alors au code mis dans le fichier tes_BasNiveau.java :

```
listOfOrders.add(new Order(0, 390, 360));
listOfOrders.add(new Order(-75, 520, 360));
listOfOrders.add(new Order(32, 750, 360));
```

### b. Validation de la mission

Nous avons noté des problèmes au niveau de la précision de nos mesures qui ont pu rendre imprécis le suivi de la trajectoire. Néanmoins, le respect des consignes a bien été observé et la mission validée.

## 3. Mise en place de la partie Gestion du plan : niveau 2

La troisième étape de ce TP consiste à donner la mission au rover de suivre un chemin correctement, soit en appliquant le suivi de courbe vue dans le première partie. Mais également de le faire en fonction d'un critère et de contraintes données. Nous nous sommes notamment penchés sur des critères de rapidité et de distance. Nous verrons alors que l'algorithme dit de Dijkstra[2] nous a permis de répondre à cette problématique.

### a. Le principe des algorithmes de recherche de chemin avec les graphs

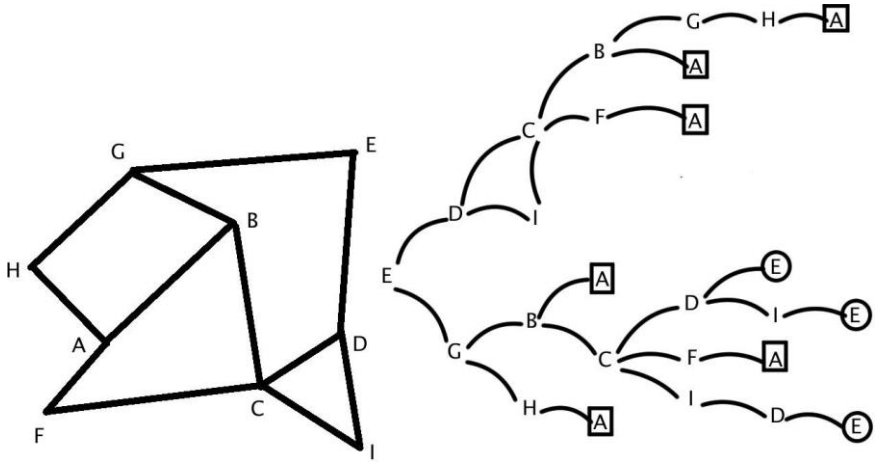


*FIGURE 4 - Graph pour la recherche d'un algorithme trouvant un chemin entre le point de départ E et le noeud fin A*

[2] voir code complété et annoté en annexe n°2

Nous avons essayé de trouver un algorithme qui retourne un chemin entre un point de départ et un nœud fin. Pour cela nous avons créé une liste de parcours qui est dynamiquement modifiée par l'algorithme lors de la recherche d'un chemin jusqu'à arriver à notre point d'arrivée. Nous avons aussi pris en compte les chemins qui bouclent sans arriver à notre nœud fin en analysant sur le graphe de la *FIGURE 4* les nœuds de bifurcation. Nous avons trouvé une version standard et une récursive de l'algorithme de recherche pensé en Python[3] (pour la simplicité du langage).

### b. Critère du plus court chemin

Pour mettre en place l'algorithme de Dijtskra avec pour critère le chemin le plus court, on écrit dans le fichier Map.java :

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

Ainsi, on obtient le trajet suivant :

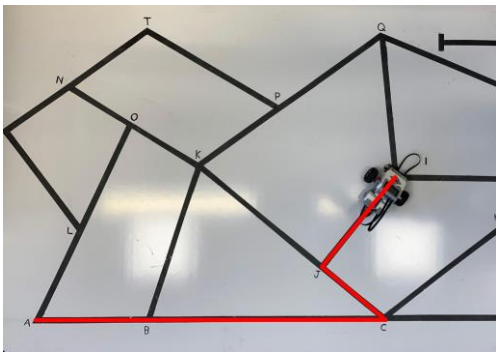


*FIGURE 5 - Tracé du trajet du robot rover pour respecter le critère de plus petite distance*

Alors, nous avons vérifié en mesurant, et ce trajet correspond bien à celui dont la distance est la plus courte. Nous pouvons ainsi valider la manipulation.

### c. Critère du chemin le plus rapide

Pour mettre en place l'algorithme de Dijtskra avec pour critère le chemin le plus rapide, on écrit dans le fichier Map.java :

```
// définition du critère du Dijkstra
this.Map[li][col] = 400-vitesse ;
this.Map[col][li] = 400-vitesse ;
```

Il est nécessaire de passer un critère à "vitesse_max - vitesse" car le critère se base sur la méthode Trouvmin(). Ainsi, plus notre vitesse sera grande et plus l'écart avec 400m/s (notre vitesse max) sera petit. La méthode Trouvmin() détectera alors ce plus petit écart et donc l'algorithme trouvera bien le chemin le plus rapide.

[3] voir pseudo code en annexe n°3

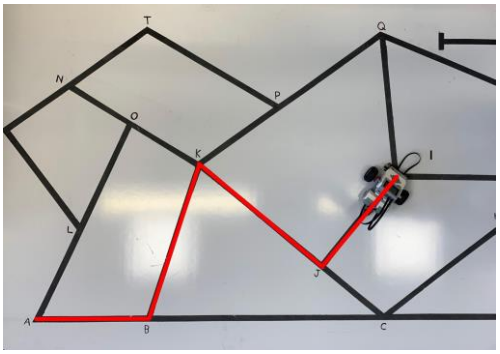


*FIGURE 6 - Tracé du trajet du robot rover pour respecter le critère de plus petite durée*

Alors, nous avons constaté que le temps mis pour réaliser ce trajet a bien été plus court que dans la partie précédente de chemin le plus rapide. Nous pouvons alors, là aussi, valider la manipulation réalisée.

## Conclusion

Ce TP nous a d'abord appris à contrôler le suivi d'une ligne noire par un robot par un modèle en boucle fermée. Nous avons ensuite compris qu'il était nécessaire de régler des paramètres tels que la vitesse du robot et le gain de commande à partir de plusieurs tests en réel afin d'en trouver des plages de valeurs acceptables. Enfin nous avons établi le lien entre la théorie des graphes et la recherche du chemin le plus court avec un critère de distance puis un critère de vitesse. La dernière partie du TP nous a permis de vérifier le bon fonctionnement de l'algorithme de Dijkstra pour la détermination du plus court chemin.

## Table des annexes

## Annexe n°1: Fichier PiloteRoberto.java

```
public void travels(float distance, int speed) {
              float linePosition;
              float erreur;
              float kp = 30;
              float Cde ;
              //speed = 3*speed;
              float[] aux;

              pilot.setLinearSpeed(speed);
              pilot.setAngularSpeed(rotateSpeed);


              // créer variable pour stocker les données du senseur de couleur
              SampleProvider Miss = lineleader.getRedMode();
              float[] sample = new float[Miss.sampleSize()];

              lineleader.wakeUp();

              distanceParcourue = 0;
       //     LCD.drawString("distance = " + distanceParcourue, 0,0);

              motorD.suspendRegulation();
              motorG.suspendRegulation();;
              pilot.forward();

              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 = 0-linePosition; //calcul de l'erreur du système
                     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/se


                     distanceParcourue = pilot.getMovement().getDistanceTraveled();

                     LCD.drawString("distance = " + distanceParcourue, 0,2);

              }
              while(distanceParcourue < distance){

                     motorG.setSpeed(speed);
                     motorD.setSpeed(speed);

                     distanceParcourue = pilot.getMovement().getDistanceTraveled();
                     LCD.drawString("distance = " + distanceParcourue, 0,3);
              }
              pilot.stop();
       }
```

## Annexe n°2: Fichier Dijkstra.java

```
public class Dijkstra {
        Map map;
        int noeud_init;
        int noeud_fin;
        int[] tab_noeuds; //tab_noeuds[i] = le noeud precedent de i
        double[] tab_value; //tableau des valeurs cumulées
        boolean[] tab_bool; //tableau pour savoir si on est déjà passé sur le point
        int fini;

   public Dijkstra(Map map, Point noeud_start, Point noeud_end) {

        this.map = map;
        this.noeud_init = noeud_start.recupInt();//On travaille sur noeud_init qui est un int
        this.noeud_fin = noeud_end.recupInt(); //On travaille sur noeud_fin qui est un int
        int NbNoeuds = this.map.getNbNoeud();

        this.fini = NbNoeuds - 1; // l'algo se finit si on explore tous les noeuds --> si fini arrive à 0
        this.tab_bool = new boolean[NbNoeuds]; //tableau des noeuds qu'il reste Ã parcourir
        this.tab_value = new double[NbNoeuds]; //tableau des valeurs les plus petites
        this.tab_noeuds = new int[NbNoeuds];// tableau des prÃ©cÃ©dents de chaque noeud

        // 1-  INITIALISATION ----------------------------------------------------------
        for (int i = 0; i < this.map.getMap().length; i++) {
        this.tab_bool[i] = false;//init du tableau de booleens
        this.tab_value[i] = Double.MAX_VALUE;//init du tableau des chemins les plus courts
        this.tab_noeuds[i] = -1;//init du tableau des noeuds précédents
        }
        this.tab_value[noeud_init] = 0; // le noeud init est Ã  0 de lui-meme
        // FIN INITIATISATION ----------------------------------------------------------
   }


   // 2- ITERATION ----------------------------------------------------------
   public void Run_Algo() {
        int noeud = this.noeud_init;
        while   (this.fini   !=   0)   {   //   3-      TANTQUE   IL   EXISTE   UN   NOEUD   NON   MARQUÉ
----------------------------------------------
        noeud = TrouvMin();  // 4- TROUVER LE SOMMET LE PLUS PROCHE ----------------------------------------------_
        MajCcum(noeud);// 5- METTRE A JOUR TABLEAU DES VALEURS CUMULEES  ----------------------------------
        } // FIN TANTQUE IL EXISTE UN NOEUD NON MARQUÉ ----------------------------------
   }
  // FIN ITERATION ----------------------------------------------------------


   // 3- TROUVER LE SOMMET LE PLUS PROCHE  ----------------------------------
   public int TrouvMin() {
        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++) {
                if (tab_value[i] < valmin  && tab_bool[i]==false) {
                                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;
   }
  // FIN TROUVER LE SOMMET LE PLUS PROCHE  ----------------------------------


 // 5- METTRE A JOUR TABLEAU DES VALEURS CUMULÉES  ----------------------------------
  public void MajCcum(int noeud_retenu) {
        int i;

        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
                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;
```

```
                }
        }
    } // FIN METTRE A JOUR TABLEAU DES VALEURS CUMULÉES  --------------------------------
   // 6- CONSTRUIRE LE TABLEAU FINAL  ----------------------------------------------
   public int[] Tab_final() {
          int nombre_noeuds = 0;
          int boucle = this.noeud_fin;

          System.out.println("critere final: " + tab_value[boucle]);
          //on cherche le nombre de noeud que l'on va passer en revue pour arriver à notre point de départ
          while (boucle != this.noeud_init) {
          boucle = tab_noeuds[boucle];
          nombre_noeuds++;
          }

          int[] tab_resul = new int[nombre_noeuds + 1];
          boucle = this.noeud_fin;

          //on remplit notre tableau final en remontant
          while (nombre_noeuds > -1) {
          //System.out.println("tab_resul[nombre_noeuds]: " + boucle + " est a: " + tab_noeuds[boucle]);
          tab_resul[nombre_noeuds] = boucle;
          boucle = tab_noeuds[boucle];
          nombre_noeuds--;
          }
          return tab_resul;
    }

    //sort un tableau de point à  partir d'un tableau de int
    public Point[] Tab_final_point(int[] tab_int, Point[] tab_point) {
          int i;
          Point[] resultat = new Point[tab_int.length];

          for (i = 0; i < tab_int.length; i++) {
          resultat[i] = tab_point[tab_int[i]];
          }
          return resultat;
    }
    // FIN CONSTRUIRE LE TABLEAU FINAL  ---------------------------------

}
```

## Annexe n°3 : Pseudo algorithme

Initialisation:

```
pt_depart=''
pt_arivee  =''
pt_courant=''
l_parcours=[pt_depart]  #avec tous les points de la
l_points_restants=[] #avec tous les points de la map au début, on enlève ceux parcourus on y met le point de départ car parcourus dès l'initialisation
l_suivant=[] #tous les points après notre point trouvés par la fonction Trouve_suivant
l_bifurcation=[] #liste des points de bifurcation
```

Fonction : (.py)

```
Def Trouve_chemin(pt_courant) :
        l_suivant=Trouve_suivant(pt_courant) #return la liste l_suivant des points suivant le point courant
        For pt_suivant in (l_suivant & l_points_restants):
                if pt_suivant != pt_arrivee:
                        If bifurcation(pt_courant) : #return TRUE si plusieurs points suivant restants
                                l_bifurcation.append(pt_courant) #liste de points de bifurcation
                if pt_suivant != pt_depart:
                                l_parcours.append(pt_suivant)
                                l_points_restants.pop(pt_courant) #enlève le point parcouru à la liste de points restants pour éviter les aller-retour
                                pt_courant=pt_suivant
                        else : #on a bouclé au point de départ sans trouver le point d'arrivée
                                pt_courant=l_bifurcation[-1] #on revient à la bifurcation
                                l_bifurcation[-1].pop
                                i= l_parcours.index(l_bifurcation[-1]) #on relève à partir de quand on a eu la bifurcation
                        l_parcours.pop[i :] #on supprime le parcours après la bifurcation
                        Trouve_suivant(pt_courant) #on remet à jour la liste de points suivants
                Else :
                        Break
Return(l_parcours)
```

Fonction (avec récursivité) :

```
Def Trouve_chemin_recursif(pt_courant) :
        l_suivant=Trouve_suivant(pt_courant)
        For pt_suivant in l_suivants:
                While pt_suivant!=pt_arrivee:
                        pt_courant=pt_suivant
                        Trouve_chemin_recursif(pt_suivant) # récursivité
                l_parcous.append(pr_suivant)
Return(l_parcours)
```

**Annexe Robot**