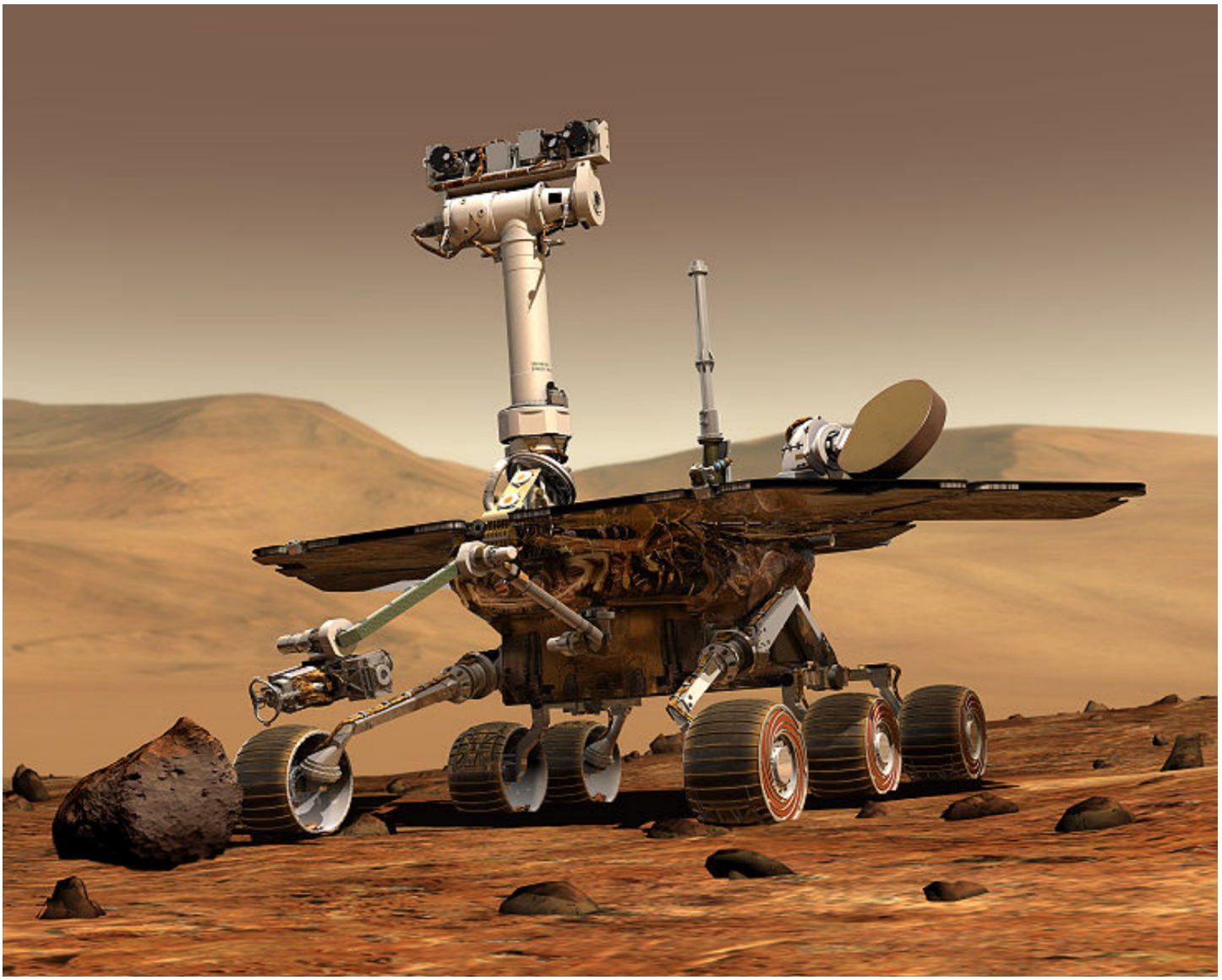
Compte rendu TP1 : Recherche de chemins dans un graphe Lego Rover



Date de réalisation 31 janvier 2023
Encadrante : Gwendoline Le Corre

Table des matières

I. Introduction

Au cours de ce TP, nous allons chercher à commander le robot LegoRover (2 roues motrices + une roue pour l'équilibre) à différents niveaux. Dans un premier temps il s'agira de commander le robot (niveau 0) afin que ce dernier suit une ligne noire dessinée au sol grâce à son capteur. Dans un deuxième temps nous chercherons à contrôler la trajectoire du robot en lui indiquant l'ordre des points à atteindre. Enfin nous nous consacrerons à la dernière partie sur la mise en place du guidage plan. Il s'agira pour nous de faire choisir au robot de manière autonome le chemin le plus optimal suivant different critère (distance / temps).

II. Préparation

Nous disposons d'un robot et le but est d'implémenter différents niveau d'automatisation de celui-ci:

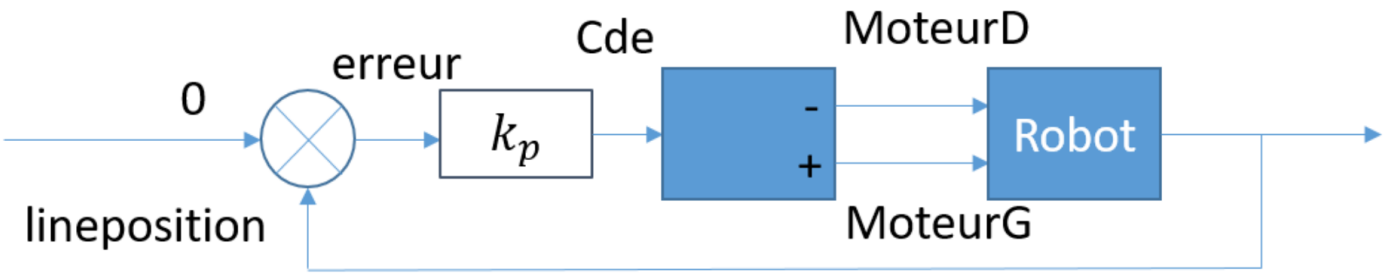
- Niveau 0 : suivre une ligne noire au sol
- Niveau 1 : suivre correctement le chemin voulu
- Niveau 2 : choisir correctement un chemin à suivre en fonction d'un critère et de contraintes données.

Le Robot MARS (@IP : 10.0.0.1) dispose de :

- 2 roues motrices et une roue libre pour l'équilibre.
- Capteur LineLeader qui envoie un bit à 1 quand le robot est sur la ligne noire.
- Un microprocesseur ARM9 qui contient un code embarqué dans la brique.

III. Mise en place de la partie guidage : Niveau 0

Figure du schéma de régulation



Explication du schéma de régulation

- k_p : il s'agit du gain pour la régulation proportionnelle que nous décidons. On prendre $k_p = 10$
- $erreur = 0 - lineposition$
- $Cde = commande = k_p * erreur$: commande envoyée aux moteurs des roues du robot pour modifier sa trajectoire.
- $lineposition$: donne la position du robot en fonction de la bande noire.
    - Si $lineposition = 0$ pas de correction car le robot est sur la ligne noire
    - Si $lineposition > 0$ le robot doit tourner à gauche
    - Si $lineposition < 0$ le robot doit tourner à droite

⇒ Les commandes envoyées aux 2 moteurs sont différentes ce qui permet de faire tourner le robot. w

Mise en oeuvre de la boucle de régulation proportionnelle

Nous avons complété le fichier PilotRoberto.java : pour faire la boucle grâce à la consigne de vitesse *speed* et la position centrale sur la ligne *lineposition* :

Code :

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

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

        erreur = 0 -linePosition;
        Cde= kp*erreur;

        // Boucle de régulation Proportionnelle

        motorG.setSpeed(speed-Cde); //degrees/sec
        motorD.setSpeed(speed+Cde);

        LCD.drawString("linepos6 = " + Cde, 0,0);
        motorD.forward();
        motorG.forward();


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

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

}
```

Détermination de la vitesse limite pour en fonction du kp

k_P	Vitesse limite (degrés/sec)
5	345
10	350
40	400

IV. Mise en place de la partie gestion de la trajectoire : Niveau 1

Nous devons vérifier que le robot gère bien la trajectoire. Il faut valider le bon séquencement d'une mission écrite à la main. Nous demandons au robot d'aller de A à Q en passant par O et K.

Fonction utilisée

listOfOrders.add(new Order(int angle, int distance, int speed)) :

- angle à prendre pour changer la trajectoire en degrés
- distance à parcourir en mm
- vitesse de déplacement du robot en degrés par seconde.

Test d'une mission allant de A à Q

Fonction de test : test_basNiveau();

```
        //test qui fait A, O, K, Q
listOfOrders.add(new Order(-68, 695, 200)); // A à O
listOfOrders.add(new Order(95, 265, 200)); // O à K
        listOfOrders.add(new Order(-66, 760, 200)); //K à Q
```

Cela fonctionne, le robot fait bien le chemin demandé.

V. Mise en place de la partie gestion du plan : Niveau 2

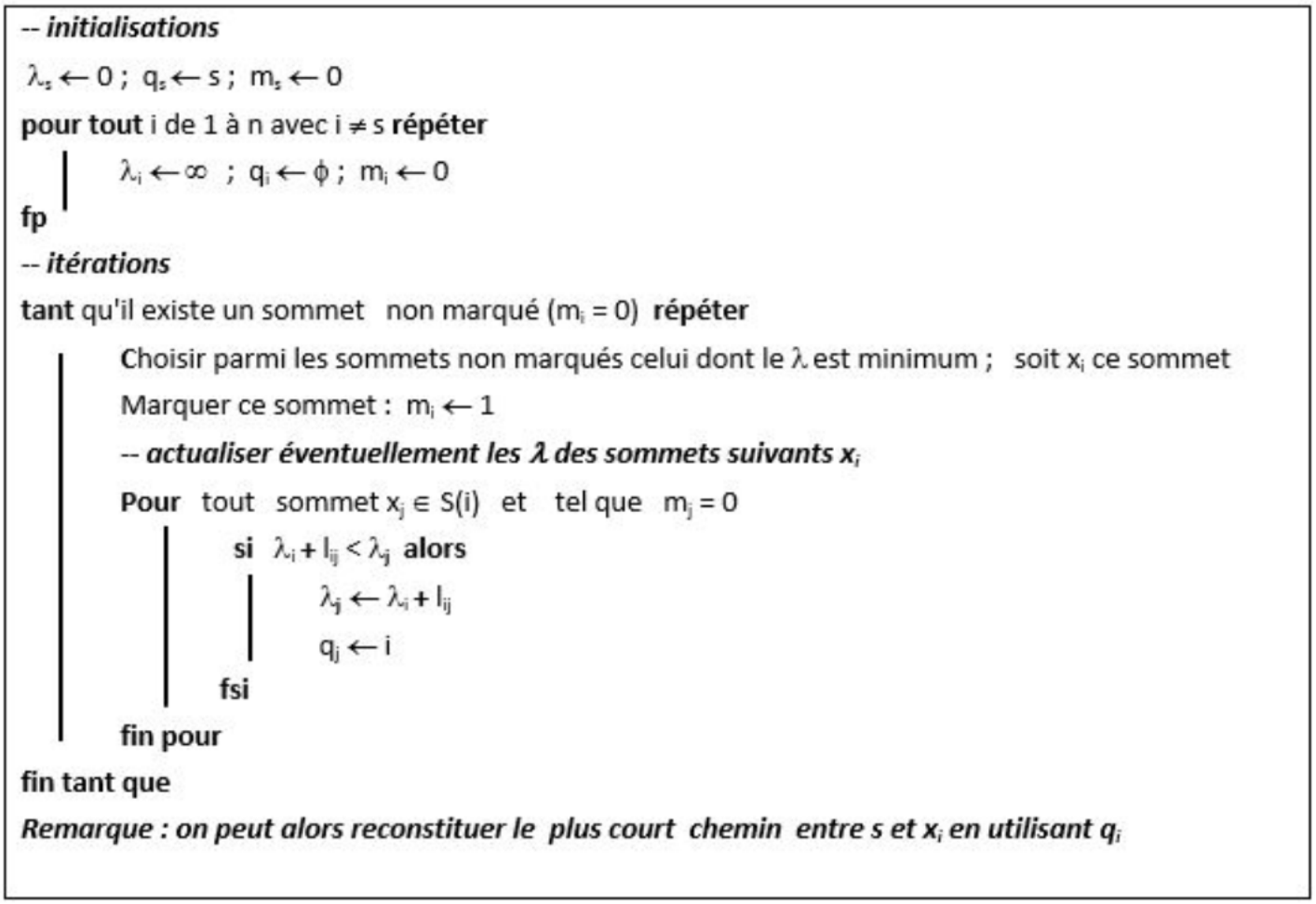
Nous allons mtn élaborer une stratégie pour élaborer un plan de mission en définissant un graphe avec des sommets (les points visitables) et des arcs (chemins entre les points). Tout cela à l'aide de l'Annexe A "Théorie des graphes" et nous utiliserons l'algorithme de Dijkstra que nous allons expliquer non exhaustivement.

Quelques explications sur l'algorithme de Dijkstra

Le but est de construire un tableau tel que chaque case concernant un sommet i contienne :

- λ = tab_value : le poids du sommet/nœud, nous avons de critère au choix (la distance/le temps).
- q = tab_noeuds : le sommet/nœud prédécesseur (le point d'où on vient).
- m = tab_bool : un booléen qui est vrai quand le sommet/nœud est marqué.

À chaque itération, l'algorithme met à jour ces trois arguments. Nous avons donc autant d'itérations que de sommets.
Voilà le pseudo-algorithme :



Définition de notre critère du Dijkstra

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

Implémentation de la fonction TrouvMin()

Elle retourne le sommet (nœud) le plus proche du sommet initial selon le critère choisi. Voilà le code.

```
        // 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
```

Implémentation de la fonction MajCum()

Elle retourne le tableau des valeurs des poids de chauqe sommet (tab_value) ainsi que le sommet précédent (tab_noeuds)
Voilà le code :

```
        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[i]>map[i][noeud_retenu]+tab_value[noeud_retenu]) {
                        tab_value[i]=map[i][noeud_retenu]+tab_value[noeud_retenu];
                        tab_noeuds[i]=noeud_retenu;
                }


        }
```

Test d’une mission allant de A à I pour deux critères différents

- Si le critère est la distance la plus courte alors le chemin choisi est A,B,C,J,I.
- Si le critère est le temps le plus court alors le chemin choisi est A,B,K,J,I. En effet il s’agit d'une distance parcourue plus longue car il évite le tronçon B/C qui est un chemin en terre à vitesse réduite.

⇒ Notre algorithme focntionne

VI. Extensions possibles

- Afin d'être plus optimal au niveau du trajet du Rover il est possible d'imaginer et d'implémenter un algorithme pour passer par différents points. Par exemple, si on souhaite partir d'un nœud initial et arriver à un nœud final en passant par certains points de manière optimale, on pourrait au niveau algorithmique calculer le plus court chemin entre chacun des points de passage. On réaliserait ainsi un arbre de décision calculant tous les chemins possibles et on reporterait les précédentes valeurs sur les arcs de l'arbre. Enfin par un simple calcul de la distance totale de chacun des chemins on trouverait facilement le plus court.

- De la même manière que précédemment, si les ressources du robot sont limitées, il serait judicieux de trouver le chemin le plus court. Connaissant la consommation du Rover, par exemple pour une distance donnée, il serait facilement prévisible des trajets que nous pouvons nous permettre d'effectuer et ceux dont il est impossible d'en effectuer la totalité.
  De plus, on peut aussi utiliser une boucle while qui actualise une variable Somme_Chemins (somme des distances des chemins choisis par la fonction TrouverMin(). Ainsi tant que cette somme est inférieure à la distance limite, le robot peut continuer son avancée.

- Afin que le robot effectue des points de mesure de plus ou moins grande importance avec une quantité de ressource limitée, on pourrait imaginer un algorithme qui combine les deux derniers algorithmes présentés précédemment. De cette façon, le chemin choisi serait optimal et respecterait les contraintes liées aux ressources du robot.

VII. Conclusion

Pour conclure ce TP nous a permis de comprendre et d'appréhender différents niveaux de commande pour un système autonome. La première partie sur la mise en place du guidage nous a tout d'abord permis de comprendre la calibration et le fonctionnement du système. Cela nous a parallèlement permis de comprendre le fonctionnement du logiciel Eclipse malgré une très mauvaise connaissance du langage Java. Ce TP nous a aussi permis d'appliquer et d'implémenter des concepts de cours tel que l'algorithme de Dijkstra (sert à résoudre le problème du plus court chemin). A l'aide de la notion de critère à minimiser, nous avons pu choisir un chemin optimal pour le robot. Ce critère pouvait être de différentes natures : distance dans un premier temps puis le temps grâce à une petite conversion. Enfin nous avons pu imaginer différentes solutions ou extensions possibles afin de rester optimal pour d'autres contraintes imposées (ressources limitées , points imposés, etc).

Finalement, ce TP en plus d'être assez ludique s'est révélé très intéressant pour nous car c'est la première fois qu'on a pu commander un système à différents niveaux.