INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

TP1
Recherche de chemin dans un graphe : LegoRover

4AE-SE
Groupe 3

SOMMAIRE

INTRODUCTION

Le principe du TP se décompose en trois parties, correspondant chacune aux différents niveaux de commande dans une architecture d'autonomie. Cela se fait à travers un robot Lego composé de deux roues motrices et d'une roue libre pour l'équilibre, mais aussi d'une grande carte avec des lignes noires. Le but de notre implémentation est de commander le robot pour qu'il suive tout d'abord une ligne noire puis qu'il suive une certaine trajectoire et enfin qu'il choisisse un chemin à suivre selon un certain critère (distance ou durée).

MISE EN PLACE DE LA PARTIE GUIDAGE

Notre Robot Lego doit suivre, dans cette partie, une ligne noire. Cela correspond à la commande de bas niveau sur le robot.

Afin que le robot Lego puisse suivre la ligne noire, il est composé d'un capteur LineLeader composé de 8 couples photodiodes/phototransistors. Lorsque celui-ci détecte du noir, le bit se met à 1. La figure ci-dessous montre le principe du suivi de ligne :

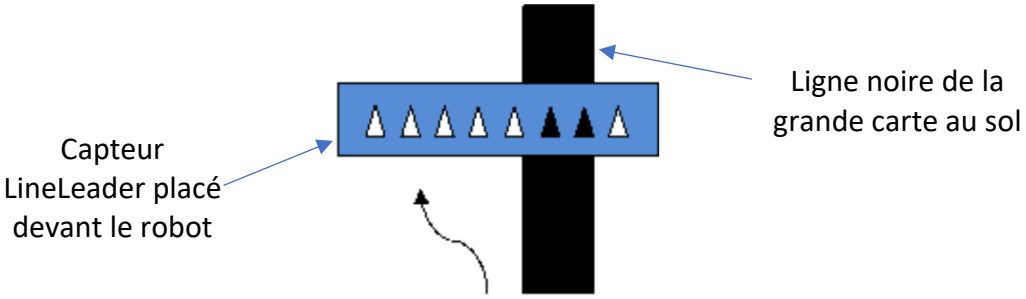

Figure 1 : Schéma de principe du Lineleader

Schéma de régulation

Le principe de la commande suit le schéma de régulation suivant :

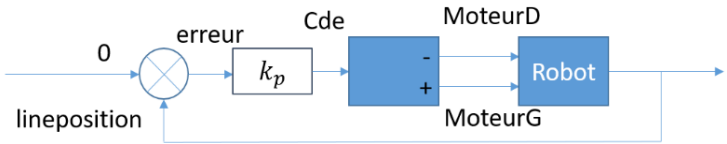

Figure 2 : Schéma de régulation de la vitesse des moteurs

Comme vous pouvez le voir sur la *figure 2*, l'erreur entre « 0 » et la position du robot par rapport à la ligne est multiplié par un gain k_p que nous choisirons. Cette commande est transformée en deux vitesses différentes, envoyées sur chacune des deux roues pour corriger la trajectoire. La commande envoyée sur les deux moteurs est différente puisque le robot doit pouvoir suivre des courbes.

Mise en place de la boucle de régulation

Pour transmettre cette commande au robot, nous complétons le fichier PilotRoberto.java (voir figure 3, annexe 1). Nous avons implémenté la commande correspondant à la *figure 2*.

Test en réel de la boucle de régulation

Vous pouvez regarder le fonctionnement de notre robot avec le QRCode n°1 suivant. Cette vidéo est réalisée avec ces données : k_p=10 et speed=300 degrés/sec.

QRcode n°1 : Vidéo robot partie guidage

Le robot suit correctement la ligne jusqu'à la fin (le capteur joue bien son rôle), et va à une vitesse relativement lente et faisant des zigzags. Les zigzags sont liés au gain k_p. Nous allons donc augmenter la vitesse et déterminer jusqu'à quelle valeur nous pouvons aller sans décrocher de la ligne. Nous modifierons ensuite le gain k_p pour déterminer sa plage de variation acceptable.

Variation de la vitesse entre 0 et 500

Par la suite, afin de déterminer les plages de données acceptables, nous testons directement sur notre robot Lego différentes vitesses. Nous observons qu'avec « speed>400 », notre robot ne suit pas correctement la ligne noire ; il n'a pas le temps d'analyser ce que le capteur de couleur reçoit afin de rester sur sa ligne.
De même, pour une vitesse « speed<100 », le robot est tellement lent qu'il dévie de sa trajectoire. Par exemple, pour « speed=10 », le robot n'avançait avec qu'une seule roue, le faisant dévier de sa trajectoire. Nous avons dû le réinitialiser en lui retirant sa batterie.
La plage de valeurs acceptable pour la vitesse est ainsi : speed$\in [100; 400]$.

Variation de K_p entre 5 et 40

Nous procédons de la même manière afin d'évaluer les effets du régulateur proportionnel k_p sur la commande du robot.
Nous observons que pour un $k_p$trop faible (k_p=5), le robot Lego ne parvient pas à suivre la ligne noire tandis que pour un k_p élevé (k_p=40), celui-ci essaye au maximum de la suivre. Il va alors zigzaguer plus fréquemment pour être le plus précis possible.
La plage de valeurs acceptable pour k_p est ainsi : $k_p \in [8; 40]$

Ainsi la partie guidage nous a permis de pouvoir réguler la trajectoire du robot sur du suivi de ligne. La vitesse choisie et le gain k_p influent sur la manière et la rapidité dont le robot parcourt la ligne entière.

MISE EN PLACE DE LA PARTIE GESTION DE LA TRAJECTOIRE

En reprenant le fichier test_BasNiveau.java, le but est d'imposer un chemin et de vérifier que le robot Lego respecte bien la trajectoire donnée. Cela correspond à la commande de niveau 1 dans une architecture d'autonomie.

Création d'une mission à la main de A à Q

Pour aller du point A au point Q, nous faisons passer le robot par les points B et K. Nous implémentons notre mission (voir figure 4, annexe 2) grâce à la commande « listOfOrders.add(new Order(int angle, int distance, int speed)) ; » avec les angles en degrés, la distance en mm et la vitesse en degres/sec. Le principe est de relever les distances et les angles sur la carte de la salle de TP. Il faut faire plus particulièrement attention à l'orientation des angles.

Test en réel de la mission

Comme vous pouvez le voir sur la vidéo (voir QRCode n°2), notre robot est bien implémenté puisqu'il suit parfaitement sa trajectoire.

QRcode n°2 : Vidéo robot partie gestion de trajectoire

MISE EN PLACE DE LA PARTIE GESTION DU PLAN

L'objectif de cette partie est de réaliser une commande de niveau 2. Le robot doit pouvoir déterminer tout seul le meilleur chemin à prendre pour atteindre sa destination. La notion de « meilleur » dépendra de ce que nous souhaitons implémenter (exemple : le plus rapide chemin, le plus court chemin, le chemin le plus économique etc…)

Formalisation du problème

Il est demandé d'automatiser la décision de trajectoire pour aller d'un point à un autre sur la carte. Notre critère sera celui du plus court trajet en termes de distance. Notre problème consiste donc à trouver le plus court chemin à partir du graphe (Figure 5, annexe 3). Chaque arc représente la distance entre 2 points. Les sommets sont les points atteignables pouvant être une destination voulue ou un départ.

Implémentation du critère de distance

La fonction SetArc() du fichier Map.java nous permet de régler le critère de distance. (Voir figure 6, annexe 4).

Méthode de recherche de chemin intuitive

Afin de trouver le plus court chemin, nous pourrions calculer l'ensemble des chemins possibles pour aller d'un point à un autre sans repasser deux fois par le même point. Ceci ressemblerait à un arbre de décision (voir figure 7, annexe 5). Puis parmi les extrémités de l'arbre, extraire celui dont la distance est la plus courte. La figure 8, annexe 6, montre ce que donnerait cette méthode algorithmiquement.

Méthode de l'algorithme de Dijkstra

L'annexe mis à disposition pour ce TP propose un algorithme de résolution pour ce type de problématique. Il s'agit de l'algorithme de Dijkstra. La figure 9, 10, 11 et 11 bis de l'annexe 7 indique les différentes étapes de cette méthode.
La figure 10 correspond à notre implémentation de la fonction Trouvmin().
La figure 11 correspond à notre implémentation de la fonction MajCcum(int nœud_retenu).
Et la figure 11 bis correspond à la partie « reconstitution du plus court chemin entre le point de départ et le point d'arrivé » de l'algorithme de Dijkstra proposé dans l'annexe du sujet.

Test en réel de la mission

Nous avons testé l'algorithme sur le trajet A → E. Comme vous pouvez le voir sur la vidéo (voir QRCode n°3), notre robot prend le chemin A → B → C → D → E, comme on pouvait s'y attendre. Ce chemin étant tout droit, on pouvait être sûr que ce serait le plus court en termes de distance.

QRcode n°3 : Vidéo robot partie gestion du plan : critère de distance

Nous remarquons cependant que le robot est lent entre B et C. Nous allons voir si le critère de temps sur ce même trajet (A → E), modifie le chemin parcouru.

Chemin le plus rapide

Nous voulons désormais trouver le chemin le plus rapide. Nous avons donc modifier la fonction SetArc() du fichier Map.java pour modifier notre critère (figure 12, annexe 8). Au vu de ce que nous avons observé sur le trajet A → E, le robot ne devrait cette fois-ci pas emprunter le chemin B-C.

Test en réel de la mission

Nous avons testé l'algorithme avec le critère de temps sur le même trajet que précédemment (voir QRCode n°4). Notre robot a bien évité le chemin B-C car il en existait un plus rapide pour atteindre la destination du sommet E. Si on regarde d'un peu plus près le code qui est fourni, on se rend compte que la vitesse programmée sur le chemin BC est bien inférieure à la vitesse des autres chemins (voir figure 13, annexe 9). Ceci explique le comportement du robot.

QRcode n°4 : Vidéo robot partie gestion du plan : critère de durée

EXTENSIONS POSSIBLES

Il est possible d'imaginer d'autres critères contraignant le choix d'un chemin. Par exemple, il serait possible d'implémenter un algorithme pour passer par différents points de manière optimale. Nous pourrions, de la même manière que l'algorithme glouton déjà proposée, réaliser un arbre de décision calculant tous les chemins possibles pour aller d'un point de départ à un point d'arrivée (voir figure 14, annexe 10). Pour ce faire, de manière algorithmique, il faudrait déjà calculer le plus court chemin entre chacun des points de passage, puis reporter ces valeurs sur les arcs de l'arbre de décision. Et enfin, il suffirait de calculer la distance totale de chacune des séquences et choisir la plus courte.

CONCLUSION

Ce TP nous a permis d'appréhender la notion de graphe et d'optimisation de chemin dans un graphe. Nous avons apprécié commander le robot suivant les différents niveaux de commandes existants. En effet, le premier niveau correspondait à suivre la ligne noire, le deuxième correspondait à suivre un chemin défini, le dernier correspondait au choix automatique du meilleur chemin pour atteindre une destination. Nous avons compris la notion de « critère » désignant l'élément à minimiser pour choisir le meilleur chemin.

Ce TP nous a donné un avant-goût du cours de « programmation linéaire, graphes et optimisation combinatoire », et a également permis de l'illustrer. Nous nous sommes rendu compte que ce cours s'appliquait sur des cas concrets de la vie courante.

ANNEXES

Annexe 1

```
151         // Boucle de régulation Proportionnelle
152
153         //erreur = lineposition; rajoute une ligne de code pour rien
154         Cde = linePosition*kp;
155         LCD.drawString("linepos6 = " + Cde, 0,0);
156         motorD.setSpeed(300-Cde); // speed - value in degrees/sec
157         motorG.setSpeed(300+Cde); // speed - value in degrees/sec
```

Figure 3 : Ajout de code dans le fichier PilotRoberto.java

Annexe 2

```
15          //Trajet de 1 à Q en passant par : A, B, K, Q
16          //listOfOrders.add(new Order(degré de rotation, distance en mm, vitesse du robot));
17          //Départ du trajet au point A
18          listOfOrders.add(new Order(0, 370, 360)); //en direction de B
19          listOfOrders.add(new Order(-75, 520, 360)); //en direction de K
20          listOfOrders.add(new Order(38, 760, 360)); //en direction de Q
21          //Arrivée au point Q
```

Figure 4 : Ajout de code dans le fichiertest_BasNiveau.java

Annexe 3

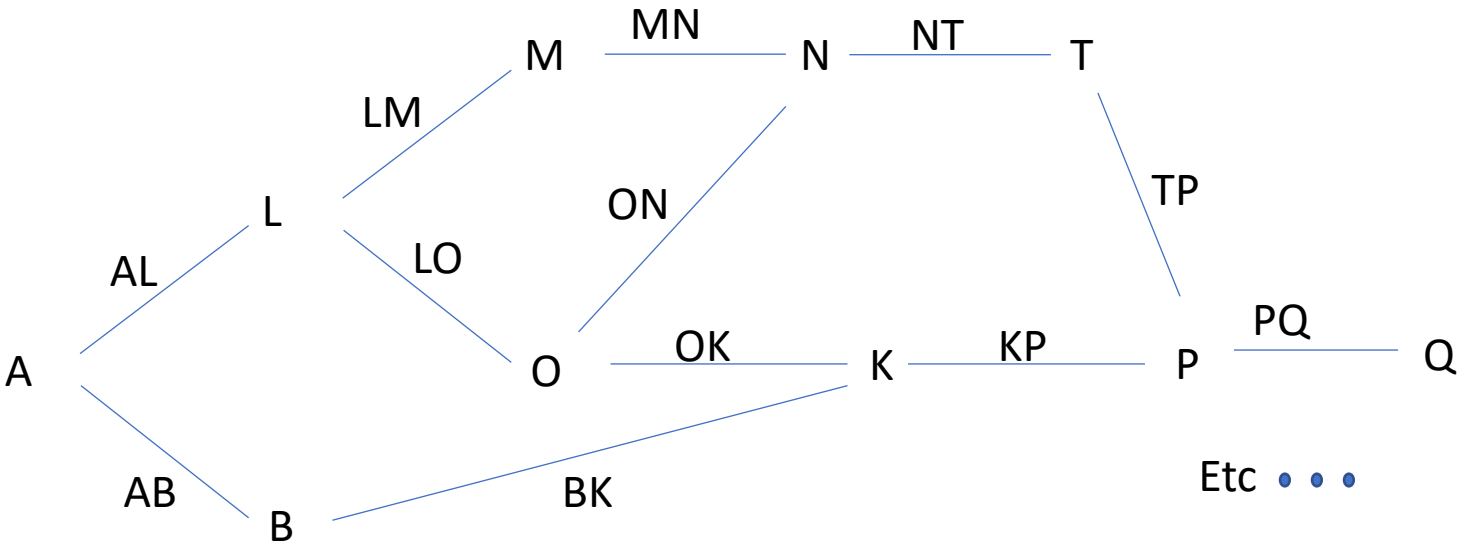

Figure 5 : Graphe valué abrégé schématisant notre carte au sol

Annexe 4

```
49          // définition du critère du Dijkstra
50          this.Map[li][col] = distance;
51          //Attribution du critère entre deux points
52          //ex: entre A et B
53          this.Map[col][li] = distance;
54          //Attribution du même critère entre les deux même point mais dans l'autre sens
55          //ex: entre B et A
```

Figure 6 : Partie de la fonction SetArc() définissant le poids des arcs suivant la distance

Annexe 5

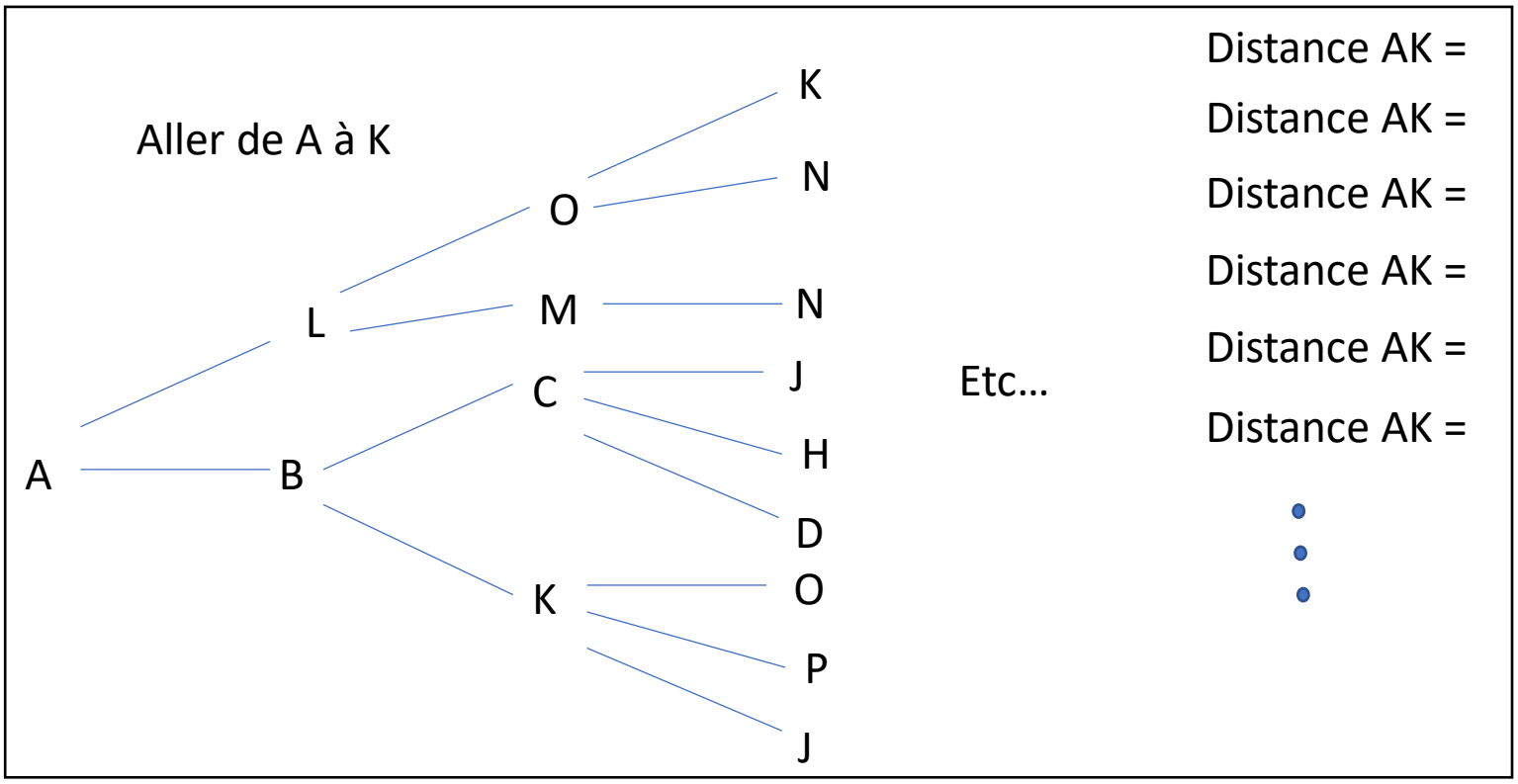

Figure 7 : Arbre de décision pour trouver le plus court chemin

Annexe 6

```
tableau_noeud = [A,B,C,...];
Distance = [parcourt1, parcourt2, parcourt3,...];
int k = 0;
int MinDistance;
//soit par exemple A le noeud de départ et E le noeud d'arrivée
while (tous les chemins possibles n'ont pas encore été explorés)
{
    Distance [i] = calcul de la distance d'un chemin i de A à E;

}
for (int i; i<Distance.length; i++)
{
    if (Distance[i]<MinDistance)
    {
        MinDistance = Distance[i];
    }

}
```

Figure 8 : Algorithme glouton de recherche du plus court chemin

Annexe 7

```
// 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 pr□c□dents
        }
        this.tab_value[noeud_init] = 0; // le noeud init est à 0 de lui-meme
        // FIN INITIATISATION ---------------------------------------------
```

Figure 9 : Partie initialisation de l'algorithme de Dijkstra

```
52    public int TrouvMin() { //retourne le sommet le plus proche du sommet initial
53
54        int noeud_retenu = -1;
55        double[][] carte = this.map.getMap();
56        double valmin = Double.MAX_VALUE;
57
58        // recherche du noeud de valeur minimale non encore exploré
59        for (int i = 0; i < carte[noeud_init].length; i++) {
60            if (this.tab_bool[i] == false && this.tab_value[i] < valmin ) {
61                valmin = this.tab_value[i];
62                noeud_retenu = i;
63            }
64        }
65
66        //Marquage du sommet non marqué dont le lambda est minimum
67        if (noeud_retenu != -1) {
68            tab_bool[noeud_retenu] = true;
69            this.fini--; //on décrémente notre variable pour savoir
70            //quand on aura passÃ® en revue tous les noeuds
71
72        } else {
73            System.out.println("------->TrouvMin : pas de solution ");
74        }
75
76        return noeud_retenu;
77    }
```

Figure 10 : Fonction TrouvMin() complétée

```
79    // mise à jour de tab_value : tableau des valeurs de critère cumulées
80    public void MajCcum(int noeud_retenu) {
81        int i;
82
83        double[][] map = this.map.getMap();
84
85        for (i = 0; i < map[this.noeud_init].length; i++) {
86            // si le critère du noeud i en passant par le noeud retenu est meilleur alors on change de chemin
87            if ((this.tab_value[noeud_retenu]+map[noeud_retenu][i] < this.tab_value[i]) && (tab_bool[i] == false)) {
88                this.tab_value[i] = this.tab_value[noeud_retenu] + map[noeud_retenu][i];
89                this.tab_noeuds[i] = noeud_retenu;
90            }
91        }
92    }
```

Figure 11 : Fonction MajCum() complétée

```java
    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
        //Reconstitution du plus court chemin entre le point final et le 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;
    }
```

Figure 11 Bis : Fin de l'algorithme de Dijkstra dans le fichier Dijkstra.java

Annexe 8

```java
        // définition du critère du Dijkstra
        this.Map[li][col] = distance/vitesse;
        //Attribution du critère entre deux points
        //ex: entre A et B
        this.Map[col][li] = distance/vitesse;
        //Attribution du même critère entre les deux même point mais dans l'autre sens
        //ex: entre B et A
```

Figure 12 : Partie de la fonction SetArc() définissant le poids des arcs suivant la durée

Annexe 9

```java
// Creation des arcs
//Map.SetArc(PointDeDépart, PointDArrivee, vitesseSurArc);
// vitesseSurArc en pourcentage
Map.SetArc(A, B, 360);
Map.SetArc(A, L, 360);
Map.SetArc(B, C, 60);
Map.SetArc(B, K, 360);
Map.SetArc(C, D, 360);
Map.SetArc(C, J, 360);
Map.SetArc(C, H, 360);
Map.SetArc(D, E, 360);
Map.SetArc(D, H, 360);
Map.SetArc(E, F, 360);
Map.SetArc(F, G, 360);
Map.SetArc(F, S, 360);
Map.SetArc(G, H, 360);
Map.SetArc(G, I, 360);
Map.SetArc(G, R, 360);
Map.SetArc(I, J, 360);
Map.SetArc(I, Q, 360);
Map.SetArc(J, K, 360);
Map.SetArc(K, O, 360);
Map.SetArc(K, P, 360);
Map.SetArc(L, M, 360);
Map.SetArc(L, O, 360);
Map.SetArc(M, N, 360);
Map.SetArc(N, O, 360);
Map.SetArc(N, T, 360);
Map.SetArc(P, Q, 360);
Map.SetArc(P, T, 360);
Map.SetArc(Q, R, 360);
Map.SetArc(R, S, 360);
```

Figure 13 : Vitesse programmée sur chaque arc dans le fichier ShortestPath.java

Annexe 10

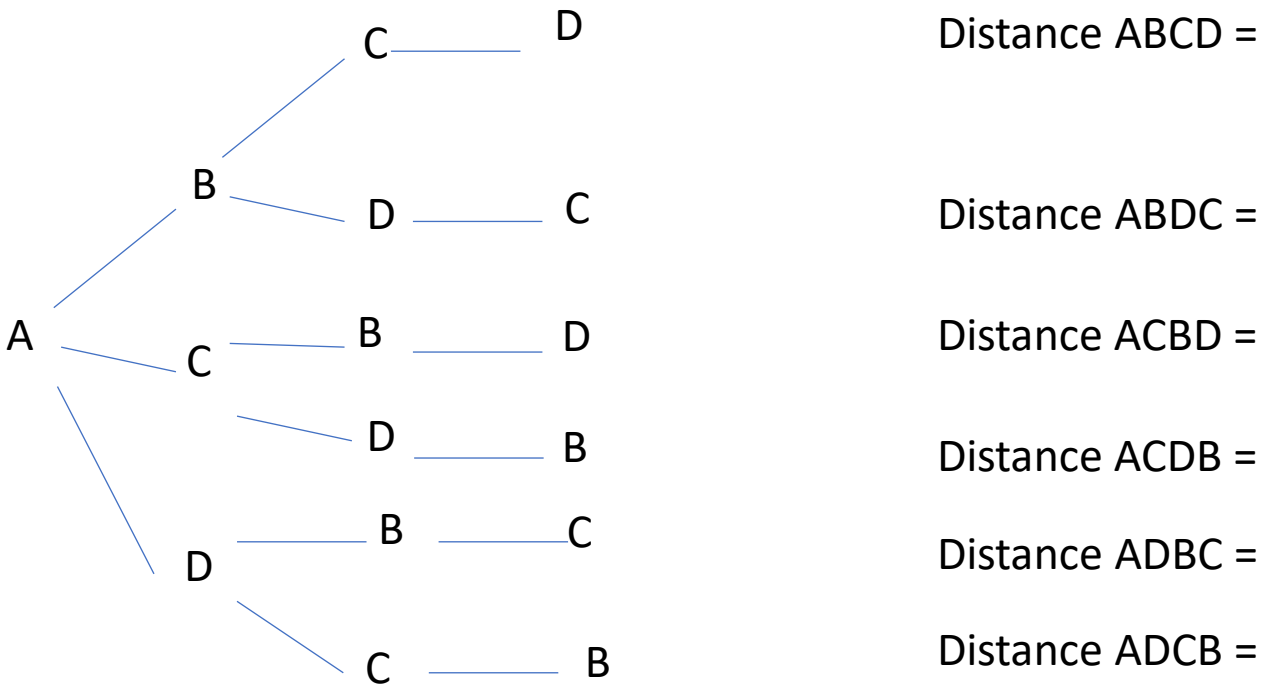

Figure 14 : Recherche du chemin optimal pour passer par 4 points A, B, C, D en commençant par A