

TP1 : Recherche de chemins dans un graphe : LegoRover

TP Automatique

Département Génie Electrique et Informatique
INSA Toulouse
7 Février, 2022

Table des matières

1	Introduction	2
2	Mise en place de la partie Guidage	2
3	Mise en place de la partie Gestion de la trajectoire	3
4	Mise en place de la partie Gestion du plan	3
5	Extensions possibles	5
6	Conclusion	5
7	Annexe	6
7.1	La partie Guidage	6
7.2	La partie Gestion de la trajectoire	7
7.3	La partie Gestion du plan	8

1 Introduction

Ce TP se déroule en 3 parties principales.

Tout d'abord, nous allons chercher à réguler le robot Lego afin qu'il suive une ligne noire courbe (niveau 0) en mettant en place une commande de contrôle des moteurs droit et gauche en inverse, ensuite nous allons le configurer afin qu'il suive un chemin choisi (niveau 1), et enfin le robot devrait choisir de manière autonome le chemin à suivre en fonction d'un critère (vitesse ou distance) et de contraintes données, à l'aide de l'algorithme de Dijkstra (niveau 2).

2 Mise en place de la partie Guidage

La figure 8 nous montre le schéma de régulation à mettre en place pour la commande de bas niveau. Nous pouvons voir que la commande est inversée pour le moteur de gauche et de droite pour permettre aux roues du robot d'avoir des vitesses différentes, et donc en conséquence permettre au robot de tourner à gauche ou à droite selon les virages.

Afin de mettre en place la boucle de régulation proportionnelle autour d'une consigne en vitesse notée *speed* et d'une position centrale sur la ligne noire, nous avons complété le fichier *PiloteRoberto.java* de la manière suivante :

```
// Boucle de regulation Proportionnelle
erreur=0-linePosition;
Cde=erreur*kp;
LCD.drawString("linepos6 = " + Cde, 0,0);
motorD.setSpeed(speed+Cde); // speed - value in degrees/sec
motorG.setSpeed(speed-Cde); // speed - value in degrees/sec
```

FIGURE 1 – La boucle de régulation proportionnelle

Le code complet est décrit dans l'annexe 7.1.

Nous pouvons tester cette régulation en réel avec une valeur de vitesse constante réglée à 300°/s en complétant le fichier *test_BasNiveau.java* de la manière suivante :

```
//test sur la ligne courbe
listOfOrders.add(new Order(0, 2000, 300)); // tester le suivi de ligne
```

FIGURE 2 – Le code pour tester suivant la ligne

Le code complet est décrit dans l'annexe 7.1.

Le robot effectue bien la mission sur la ligne courbe sans dévier. Une augmentation de la vitesse est peut être envisageable.

Afin d'étudier davantage l'effet de la vitesse et de déterminer sa plage de valeurs acceptables, nous avons fait varier la vitesse entre 0 et 500 et analyser le comportement du robot en réel.

Le tableau comparatif des résultats de différentes vitesses :

La valeur de vitesse(degree/s)	100	300	400	450	500
Etat de mission	OK	OK	OK	FAIL	FAIL

Au-delà de 400°/s, le robot est très rapide et dévie de la trajectoire courbe. Pour une valeur de vitesse de 0°/s, le robot ne se déplace pas bien évidemment. **La plage de valeurs acceptable pour la vitesse est donc entre 0° et 400°.**

Afin de mettre en évidence les caractéristiques d'un régulateur proportionnel, nous avons fixé la vitesse à 300°/s et nous avons fait varier Kp entre 5 et 40.

Le tableau comparatif des résultats de différentes valeurs de Kp :

La valeur de Kp	Etat du mission
5	FAIL
10	OK
20	OK
30	OK
35	OK, mais le robot dévie légèrement d'un côté à l'autre en avançant
40	OK, mais le robot dévie largement d'un côté à l'autre en avançant

Nous pouvons clairement voir que pour des valeurs élevées de Kp (à partir de 30), l'erreur en entrée de la commande va être multipliée par un coefficient élevé, et donc la commande du régulateur cherche à corriger cette erreur avec une amplitude plus "forte" (d'où les mouvements brusques). Le capteur devient trop "sensible" avec le lineposition qui est le degré de déviation de la direction du robot par rapport à la trajectoire définie.

De plus, les valeurs de Kp qui sont plus petites que 10 donne un système peu "sensible" en fonction du changement de trajectoire. L'erreur en entrée de la commande va être multipliée par un coefficient petit. En conséquence, leur "Cde" est trop petite pour faire tourner le robot.

Pour conclure, la valeur de Kp doit être ni trop grande ni trop petite (entre 10 et 30).

3 Mise en place de la partie Gestion de la trajectoire

Nous avons écrit "à la main" une mission composée d'une séquence de trajectoires (allant du point A au point Q), à vitesse constante de 360°/s.

Pour cela, nous avons utilisé les commandes du type :

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

avec les angles en degrés, la distance en mm et la vitesse en degrés par seconde.

Nous avons dû mesurer les distances et les angles entre les différents points. Pour réaliser une mission allant du point A au point Q sur la carte, nous avons besoin de mesurer l'angle $\angle BCJ$ qui vaut 35° et l'angle $\angle JKQ$ qui vaut 75°. Afin de mettre en oeuvre la trajectoire voulue qui passe par les points A, B, C, J, K, P et Q, nous utilisons le code suivant :

```
//test qui fait A, B, C, K, Q
listOfOrders.add(new Order(0, 350, 360));
listOfOrders.add(new Order(0, 780, 360));
listOfOrders.add(new Order(-145, 770, 360));
listOfOrders.add(new Order(105, 720, 360));
```

FIGURE 3 – Le code de gestion de la trajectoire

Le code complet est décrit dans l'annexe 7.2.

Ici -145 signifie que le robot tourne 145° à gauche et 105 signifie que le robot tourne 105° à droite.

La longueur de la distance CK vaut 770mm et la distance KQ vaut 720mm.

Le robot effectue bien la mission en réel, malgré le fait que nos mesures de distances et d'angles n'étaient pas très précises.

4 Mise en place de la partie Gestion du plan

Afin d'élaborer un chemin pour le robot, le critère d'optimisation est le suivant : " **Minimiser la distance entre le point de départ et le point d'arrivé, donc chercher la somme de distances d_i entre les points la plus petite possible**". La formalisation sous forme de graph est donnée dans la Figure 9.

Nous pouvons donc compléter la fonction SetArc (Fichier Map.java dans l'annexe 7.3) dans le fichier Map.java de manière à définir le poids des arcs dans le graphe, qui correspond donc à la "distance" selon notre critère.

```
// definition du critere du Dijkstra
this.Map[li][col] = distance;
this.Map[col][li] = distance;
```

FIGURE 4 – Le code du critère de distance

Avant de passer à l'utilisation de l'algorithme de Dijkstra, nous proposons une méthode de recherche de chemin intuitive "glouton" (le pseudo code est dans l'Annexe 10) pour trouver le chemin entre un point de départ et un noeud fin : Nous pouvons par exemple utiliser un algorithme qui calcule tous les chemins possibles entre un point de départ et un point d'arrivée, et ensuite choisir le plus court (donc la valeur la plus faible), mais le problème est que cet algorithme est beaucoup plus coûteux en terme de calculs qu'un algorithme de Dijkstra par exemple.

Nous pouvons donc passer au fichier *Dijkstra.java* (voir Annexe 7.3). Le pseudo code de cette méthode est donné en Annexe dans la Figure 11.

Pour trouver le sommet le plus proche du sommet initial, nous avons complété la fonction *TrouvMin()* (Figure 5), et pour mettre à jour les différents tableaux de noeuds et de valeurs, nous avons complété la fonction *MajCcum(int noeud_retenue)* (Figure 6) pour trouver le sommet avec la plus courte distance, en utilisant l'algorithme de Dijkstra :

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

    int noeud_retenue = -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_retenue=i;
        }
    }
}
```

FIGURE 5 – Le code de la fonction TrouvMin

Le code de la Figure 5 correspond à la ligne "Choisir parmi les sommets non marqués celui dont le λ est minimum. Marquer ce sommet" dans le pseudo code de la Figure 11.

```
public void MajCcum(int noeud_retenue) {
    int i;

    double[][] map = this.map.getMap();

    for (i = 0; i < map[this.noeud_init].length; i++) {
        // si le critere du noeud i en passant par le noeud retenue est meilleur alors on change de chemin
        if ((tab_value[noeud_retenue]+map[noeud_retenue][i] < tab_value[i])) {
            tab_value[i]=tab_value[noeud_retenue]+map[noeud_retenue][i];
            tab_noeuds[i]=noeud_retenue;
        }
    }
}
```

FIGURE 6 – Le code de la fonction MajCcum

Le code de la Figure 6 correspond à la boucle "Pour .. fin pour" dans la partie "itérations" du pseudo code de la Figure 11.

Nous passons au test en réel en décommentant la ligne de code *"BuildPath.ComputeDijkstra();"*. Nous avons testé pour 3 différentes trajectoires :

- (1) A→Q : A-B-K-P-Q
- (2) A→S : A-B-C-G-R-S
- (3) A→T : A-L-O-N-T

Le robot prend bien les chemins listés pour chacune des trajectoires testées, qui correspondent aux chemins les plus courts possibles dans chaque cas.

Enfin, nous nous intéressons maintenant à faire un chemin le plus rapide en terme de temps de trajet. **Notre critère d'optimisation est donc la "vitesse"**. Nous adaptons le fichier *Map.java* (voir Annexe 7.3) :

```
// definition du critere du Dijkstra
this.Map[li][col] = 360-vitesse;
this.Map[col][li] = 360-vitesse;
```

FIGURE 7 – Le code du critère de vitesse

L'algorithme de Dijkstra cherche les pondérations avec les valeurs les plus petites, nous utilisons donc une méthode de valeur par complément (valeur max de vitesse - vitesse). **De cette manière, la valeur insérée sera plus petite le plus la vitesse de trajectoire est grande.**

Pour mettre en évidence cette méthode, il suffit de modifier les vitesses dans *SetMap()* (Annexe 7.3, *ShortestPath.java* → "Création des arcs") telle que toutes les trajectoires à part celles qu'on veut tester (pour comparer avec la trajectoire avec critère d'optimisation "distance") aient des vitesses beaucoup plus faibles.

Nous avons choisi de tester la première trajectoire (1) étudiée précédemment. Pour contrôler le résultat que nous allons obtenir, nous avons "imposé" un chemin :

A → Q : A-B-C-J-I-Q

Nous avons donc notamment diminué la vitesse des trajectoires B-K, K-P, P-Q et des autres sections à 100° pour mettre en évidence notre nouveau critère.

Le test en réel nous montre donc que le robot ne choisit plus le chemin le plus court, mais le chemin le plus rapide en terme de temps qui correspond bien à la trajectoire que nous avons définie et imposée.

5 Extensions possibles

- Afin de permettre au robot de partir d'un noeud initial, d'arriver à un noeud fin, en passant par des points particuliers précisés et non ordonnés, de manière optimale, nous pouvons par exemple limiter la vitesse des trajectoires des points que nous avons pas précisées (ou les diminuer), et ensuite utiliser l'algorithme de Dijkstra sur les points restant (et donc précisés) pour trouver le chemin le plus court en terme de vitesse ou distance.

- Si les ressources du robot sont limitées et qu'il peut par exemple effectuer une distance limitée (carburant limité), nous pouvons par exemple utiliser une boucle *while*. De cette manière, nous pouvons définir *Somme_{distance}* la somme des distances des sections que nous avons choisi. Après chaque utilisation de la fonction *TrouvMin()*, nous cumulons le résultat dans la variable *Somme_d*. Donc pour chaque boucle *while*(*Somme_d* ≤ *Distance_{max}*), nous pouvons faire tourner la fonction *TrouvMin()*.

- On imagine maintenant un rover sur mars, qui doit effectuer des points de mesures de plus ou moins grande importance, avec une quantité de ressource limitée. Afin de résoudre ce problème en maximisant les récompenses sur les points de mesures tout en garantissant que les ressources seront suffisantes, nous pouvons imaginer un algorithme qui combine les deux algorithmes précédents, en prenant comme facteur d'optimisation la distance cette fois-ci.

6 Conclusion

Ce TP nous a permis d'étudier en réel l'effet du gain d'un régulateur proportionnel sur le robot en variant la vitesse et la valeur de *Kp* du code. Nous avons aussi eu l'occasion de réfléchir quant aux différents algorithmes possibles pour la recherche d'un chemin pour le robot, en mettant en avant un critère d'optimisation différent pour chaque cas. Nous avons enfin pu voir que l'algorithme de Dijkstra est le plus optimal en terme de coût, et son implémentation a été possible en étudiant le code mis à disposition.

7 Annexe

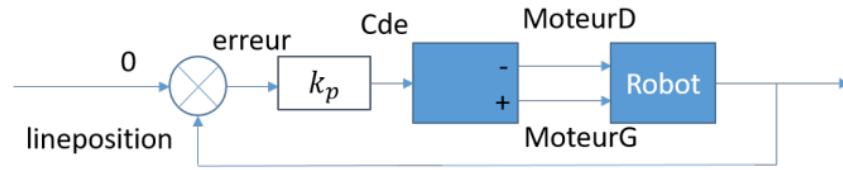


FIGURE 8 – Schéma de régulation

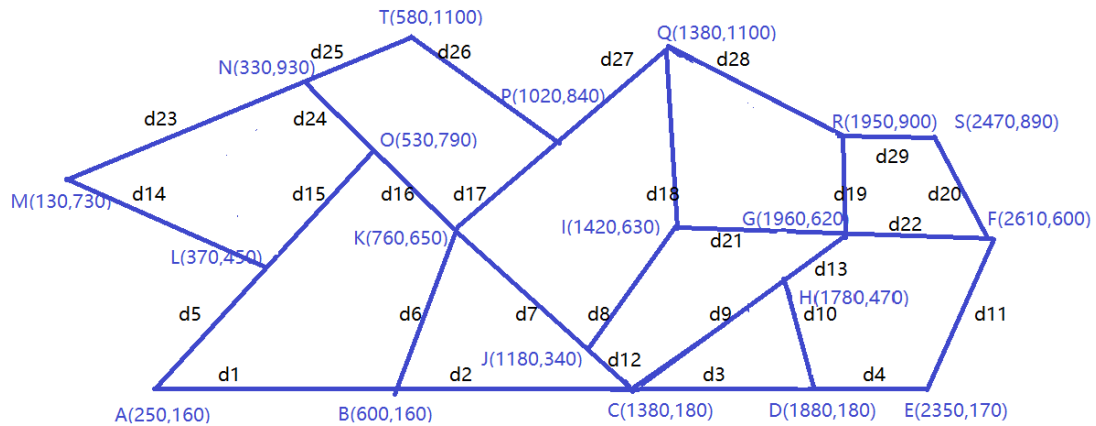


FIGURE 9 – Le plan de la mission

7.1 La partie Guidage

PilotRoberto.java

```
public void travels(float distance, int speed) {
    float linePosition;
    float erreur;
    float kp = 10;
    float Cde;
    float [] aux;

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

    // cr er variable pour stocker les donn es du senseur de couleur (blanc/noir)
    SampleProvider Miss = lineleader.getRedMode();
    float [] sample = new float [Miss.sampleSize()];

    lineleader.wakeUp();

    distanceParcourue = 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 r gulation Proportionnelle
    erreur=0-linePosition;
    Cde=erreur*kp;

    LCD.drawString("linepos6 = " + Cde, 0,0);
    motorD.setSpeed(speed+Cde); // speed - value in degrees/sec
    motorG.setSpeed(speed-Cde); // speed - value in degrees/sec

    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();
}

```

testBasNiveau.java

```

public class test_BasNiveau {

    public static void main(String[] args){
        ArrayList<Order> listOfOrders = new ArrayList<Order>();
        //le robot tourne dans le sens horaire

        //test sur la ligne courbe
        listOfOrders.add(new Order(0, 2000, 300)); // tester le suivi de ligne

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

```

7.2 La partie Gestion de la trajectoire

testBasNiveau.java

```

public class test_BasNiveau {

    public static void main(String[] args){
        ArrayList<Order> listOfOrders = new ArrayList<Order>();
        //le robot tourne dans le sens horaire

        //test qui fait A, B, C, K, Q
        listOfOrders.add(new Order(0, 350, 360)); //AB
        listOfOrders.add(new Order(0, 780, 360)); //BC
        listOfOrders.add(new Order(-145, 770, 360)); //CK
    }
}

```



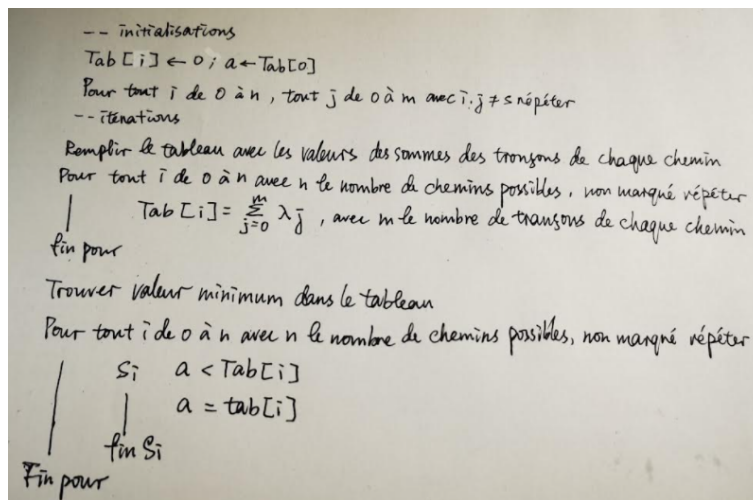
```

        listOfOrders.add(new Order(105, 720, 360)); //KQ

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

```

7.3 La partie Gestion du plan

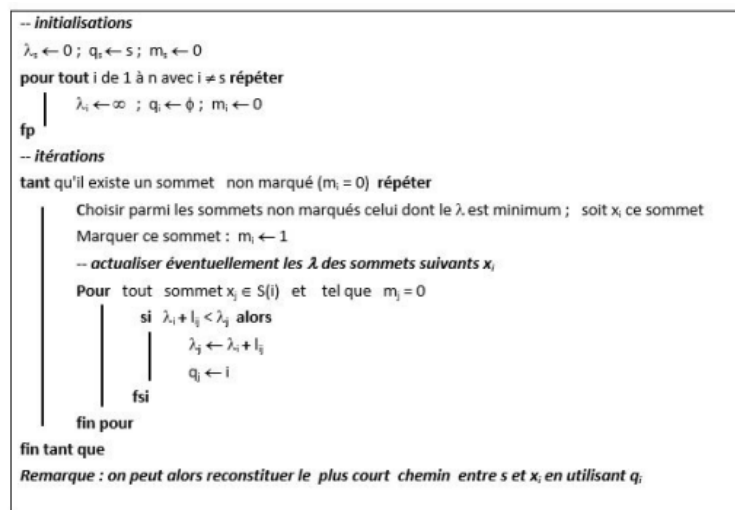


```

-- initialisations
Tab[i] ← 0 ; a ← Tab[0]
Pour tout i de 0 à n, tout j de 0 à m avec i, j ≠ s répéter
-- itérations
Remplir le tableau avec les valeurs des sommes des tronçons de chaque chemin
Pour tout i de 0 à n avec n le nombre de chemins possibles, non marqué répéter
    Tab[i] =  $\sum_{j=0}^m \lambda_j$ , avec m le nombre de tronçons de chaque chemin
fin pour
Trouver valeur minimum dans le tableau
Pour tout i de 0 à n avec n le nombre de chemins possibles, non marqué répéter
    si a < Tab[i]
    | a = Tab[i]
    | fin si
fin pour

```

FIGURE 10 – Le pseudo code de la méthode proposée



```

-- initialisations
 $\lambda_s \leftarrow 0$  ;  $q_s \leftarrow s$  ;  $m_s \leftarrow 0$ 
pour tout i de 1 à n avec i ≠ s répéter
     $\lambda_i \leftarrow \infty$  ;  $q_i \leftarrow \emptyset$  ;  $m_i \leftarrow 0$ 
fp
-- itérations
tant qu'il existe un sommet non marqué ( $m_i = 0$ ) répéter
    Choisir parmi les sommets non marqués celui dont le  $\lambda$  est minimum ; soit  $x_i$  ce sommet
    Marquer ce sommet :  $m_i \leftarrow 1$ 
    -- actualiser éventuellement les  $\lambda$  des sommets suivants  $x_i$ 
    Pour tout sommet  $x_j \in S(i)$  et tel que  $m_j = 0$ 
        si  $\lambda_i + l_{ij} < \lambda_j$  alors
             $\lambda_j \leftarrow \lambda_i + l_{ij}$ 
             $q_j \leftarrow i$ 
        fsi
    fin pour
fin tant que
Remarque : on peut alors reconstituer le plus court chemin entre s et  $x_i$  en utilisant  $q_i$ 

```

FIGURE 11 – Le pseudo code de l'algorithme de Dijkstra

- En utilisant un critère de distance
Map.java

```

public void SetArc(Point noeud_1, Point noeud_2, int vitesse) {
    //On attribue une ligne et une colonne pour le couple de point donnee
    String stringcol = noeud_1.getName();
    char car_col = stringcol.charAt(0);
    int col = (int) car_col - 65;
}

```

```

String stringli = noeud_2.getName();
char car_li = stringli.charAt(0);
int li = (int) car_li - 65;

this.SpeedMap[li][col] = vitesse;
this.SpeedMap[col][li] = vitesse;
// calcul de la distance entre ces deux point:
double distance = Math.sqrt(Math.pow(Math.abs(noeud_1.getX() - noeud_2.getX()), 2.0)

// definition du critere du Dijkstra
this.Map[li][col] = distance;
this.Map[col][li] = distance;
}

```

Dijkstra.java

```

public int TrouvMin() { //retourne le sommet le plus proche du sommet initial
    int noeud_retenue = -1;
    double [][] carte = this.map.getMap();
    double valmin = Double.MAX_VALUE;

    // recherche du noeud de valeur minimale non encore explore
    for (int i = 0; i < carte[noeud_init].length; i++) {
        if (tab_value[i] < valmin && tab_bool[i]==false) {
            valmin=tab_value[i];
            noeud_retenue=i;
        }
    }
    if (noeud_retenue != -1) {
        tab_bool[noeud_retenue] = true;
        this.fini--;
    } else {
        System.out.println("————>TrouvMin : pas de solution ");
    }
    return noeud_retenue;
}

public void MajCcum(int noeud_retenue) {
    int i;
    double [][] map = this.map.getMap();

    for (i = 0; i < map[this.noeud_init].length; i++) {
        // si le critere du noeud i en passant par le noeud retenue est meilleur alors on
        if ((tab_value[noeud_retenue]+map[noeud_retenue][i] < tab_value[i])) {
            tab_value[i]=tab_value[noeud_retenue]+map[noeud_retenue][i];
            tab_noeuds[i]=noeud_retenue;
        }
    }
}
}

```

- En utilisant un critère de vitesse

Map.java

```

public void SetArc(Point noeud_1, Point noeud_2, int vitesse) {
    //On attribue une ligne et une colonne pour le couple de point donnee
}

```

```

String stringcol = noeud_1.getName();
char car_col = stringcol.charAt(0);
int col = (int) car_col - 65;

String stringli = noeud_2.getName();
char car_li = stringli.charAt(0);
int li = (int) car_li - 65;

this.SpeedMap[li][col] = vitesse;
this.SpeedMap[col][li] = vitesse;
// calcul de la distance entre ces deux point:
double distance = Math.sqrt(Math.pow(Math.abs(noeud_1.getX() - noeud_2.getX()), 2.0)

// definition du critere du Dijkstra
this.Map[li][col] = 360-vitesse;
this.Map[col][li] = 360-vitesse;
}

```

ShortestPath.java

```

public void SetMap(){
    int nombre_noeud_map = 20;

    tab_point = new Point[nombre_noeud_map];
    int nb = 0;
    ancien_angle_x = 0;
    angle_x = 0;

    Map = new Map(nombre_noeud_map);
    Point A = new Point("A", 250, 160);
    tab_point[nb] = A;
    nb++;
    Point B = new Point("B", 600, 160);
    tab_point[nb] = B;
    nb++;
    Point C = new Point("C", 1380, 180);
    tab_point[nb] = C;
    nb++;
    Point D = new Point("D", 1880, 180);
    tab_point[nb] = D;
    nb++;
    Point E = new Point("E", 2350, 170);
    tab_point[nb] = E;
    nb++;
    Point F = new Point("F", 2610, 600);
    tab_point[nb] = F;
    nb++;
    Point G = new Point("G", 1960, 620);
    tab_point[nb] = G;
    nb++;
    Point H = new Point("H", 1780, 470);
    tab_point[nb] = H;
    nb++;
    Point I = new Point("I", 1420, 630);
    tab_point[nb] = I;
    nb++;
    Point J = new Point("J", 1180, 340);
}

```

```

tab_point[nb] = J;
nb++;
Point K = new Point("K", 760, 650);
tab_point[nb] = K;
nb++;
Point L = new Point("L", 370, 450);
tab_point[nb] = L;
nb++;
Point M = new Point("M", 130, 730);
tab_point[nb] = M;
nb++;
Point N = new Point("N", 330, 930);
tab_point[nb] = N;
nb++;
Point O = new Point("O", 530, 790);
tab_point[nb] = O;
nb++;
Point P = new Point("P", 1020, 840);
tab_point[nb] = P;
nb++;
Point Q = new Point("Q", 1380, 1100);
tab_point[nb] = Q;
nb++;
Point R = new Point("R", 1950, 900);
tab_point[nb] = R;
nb++;
Point S = new Point("S", 2470, 890);
tab_point[nb] = S;
nb++;
Point T = new Point("T", 580, 1100);
tab_point[nb] = T;

// Creation des arcs
// vitesseSurArc en pourcentage
Map.SetArc(A, B, 360);
Map.SetArc(A, L, 100);
Map.SetArc(B, C, 360);
Map.SetArc(B, K, 100);
Map.SetArc(C, D, 100);
Map.SetArc(C, J, 360);
Map.SetArc(C, H, 100);
Map.SetArc(D, E, 100);
Map.SetArc(D, H, 100);
Map.SetArc(E, F, 100);
Map.SetArc(F, G, 100);
Map.SetArc(F, S, 100);
Map.SetArc(G, H, 100);
Map.SetArc(G, I, 100);
Map.SetArc(G, R, 100);
Map.SetArc(I, J, 360);
Map.SetArc(I, Q, 360);
Map.SetArc(J, K, 100);
Map.SetArc(K, O, 100);
Map.SetArc(K, P, 100);
Map.SetArc(L, M, 100);
Map.SetArc(L, O, 100);

```

```
Map.SetArc(M, N, 100);  
Map.SetArc(N, O, 100);  
Map.SetArc(N, T, 100);  
Map.SetArc(P, Q, 100);  
Map.SetArc(P, T, 100);  
Map.SetArc(Q, R, 100);  
Map.SetArc(R, S, 100);  
Map.Affiche();  
}
```