Rapport de Projet SIM202

Planification de trajectoire

Pierre-Antoine CHEMINOT
Gonzague BEUTTER
Maxence JOSEPH
Promo 2023

TP réalisé avec C++ et Matlab

Mars 2021

Table des matières

1 Présentation du Projet

L'objectif de ce projet est de déterminer la trajectoire que doit suivre un objet pour se déplacer d'un point de départ A à un point d'arrivée B. Cette trajectoire doit être la meilleure dans la mesure où la distance parcourue par l'objet doit être minimale, tout en évitant un certain nombre d'obstacles.

Dans un premier temps, nous nous intéresserons aux cas où l'objet est assimilé à un point qui ne peut suivre que des trajectoires rectilignes. De même, les obstacles considérés seront des polygones.

Lorsqu'il n'y a pas d'obstacles, la distance minimale entre les points de départ et d'arrivée et celle du segment [AB].

Lors de la présence d'un obstacle, cette distance minimale sera obtenue par une concaténation de segments entre les points de départ, d'arrivée, et certains sommets des obstacles.

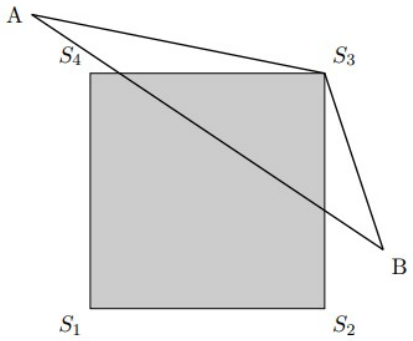

Figure 1 – Plus court chemin de A à B avec et sans obstacles

Il est bien-sûr envisageable qu'il y ai plusieurs obstacles. L'ensemble de ces derniers constituera un graphe global, qui est un ensemble de sommets et de segments, et qui contiendra également les points de départ et d'arrivée. Ce graphe sera construit par concaténation d'obstacles. En effet, le point de départ sera considére comme un obstacle (constitué d'un seul sommet), et sera concaténé au premier obstacle pour former le début du graphe. De manière itérartive, on ajoute un par un les différents obstacles au graphe global, jusqu'au point d'arrivée (lui même considéré comme un obstacle). Chacun des segments du graphe possède un poids, qui s'intérprète comme la distance entre les deux sommets formant le segment.

L'enjeu est alors de déterminer les segments valides qui peuvent être parcourus par l'objet. Il s'agit de ceux constituant les arêtes des obstacles pris individuellement, mais également certains segments reliant deux sommets d'obstacles différents. On attribue alors aux segments non valides un poids infini. En effet, le but étant de minimiser la distance entre A et B, donner une longueur infinie à un segment l'exclu automatiquement des choix possibles par l'objet.

Pour résoudre ce problème, plusieurs outils mathématiques peuvent être utiles.

- En orientant les obstacles et en initialisant ses sommets dans l'ordre trigonométrique, le calcul des normales des arêtes est utile pour déterminer la concavité ou la convexité de la zone de l'obstacle engendrée par les segments en questions.
- De même, savoir si deux segments s'interceptent demeure fondamental pour construire le graphe global.

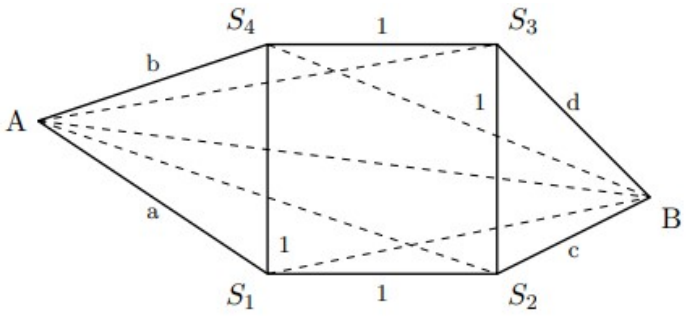

FIGURE 2 – Exemple de graphe

2 Implémentation algorithmique

L'implémentation algorithmique est réalisée grâce au logiciel C++. Nous définissons pour cela plusieurs classes pour résoudre le problème.

2.1 La classe Point

La stratégie énoncée précédemment consiste à considérer les points de départ et d'arrivée comme des obstacles. Cependant, un obstacle est un ensemble de segments qui relie des sommets et qui possèdent chacun un poids. Il est donc nécessaire dans un premier temps de construire un objet *Point* de $\mathcal{R}^2$ qui possède toutes les propriétés que l'on peut attendre d'un tel objet.

Notre classe *Point* doit naturellement posséder deux coordonnées réelles (x,y) décrivant sa position dans le plan.

Nous définisson par ailleurs un constructeur, qui, par défaut (c'est à dire dans le cas où aucune coordonnées n'est spécifiées lors de la construction du point), crée le point (0,0).

Nous aurons besoin ultérieurement de toutes les opérations usuelles entre deux points. Nous implémentons donc des fonctions internes (qui agissent que le point en question) et externes (qui créent un nouveau point) correspondant aux opérations usuelles d'addition, de soustraction, de multiplication et de division par un scalaire.

Enfin, il nous semble utile de surcharger des opérateurs permettant d'effectuer des tests d'égalité entre deux points, ainsi que qu'une fonction externe qui a partir deux point u et v, renvoie le vecteur $\overrightarrow{uv}$, qui est alors considéré comme un point car seul la connaissance de ses coordonnées nous intéresse.

2.2 La classe Arc

Revenons à notre objectif qui est dans un premier temps de construire une classe *Obstacle*, constituée de points et de segments pondérés. Nous avons jusqu'à présent définis une classe *Point* qui va nous être utile dans cette construction. Nous souhaitons maintenant implémenter informatiquement une classe *Segment*. Cependant, pour ne pas perdre en généralité, nous choisissons d'abord de créer une

classe *Arc*, dont un cas particulier sera un segment.

En effet, dans le cas de notre projet, le poids attribué à chaque segment représente la distance entre les deux point formant le segment. Cette distance entre deux points $A(a_x, a_y)$ et $B(b_x, b_y)$ se calcul grâce à la formule : $d = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$. Toutefois, nous pouvons envisager dans une autre situation que ce poids peut représenter une quantité différente, qui peut être négative par exemple. Définir une classe mère *Arc* possédant une classe fille *Segment* nous garantit de pouvoir utiliser ultérieurement la classe *Arc* à d'autres fins (dans le cadre d'un autre projet par exemple).

Nous implémentons ainsi une classe *Arc*, constituée de deux points A et B (construit grâce à la classe *Point*), et d'un nombre réel qui est un poids.

La classe *Segment* hérite de *Arc*, et l'appel de son constructeur permettra d'attribuer au poids la valeur de la distance entre les deux points A et B.

2.3 La classe Obstacle

Nous avons dès à présent tous les éléments pour créer un obstacle.

Un tel objet est ainsi constitué de points, représentant ses sommets, aisni qu'un ensemble de segments qui relient certains sommets entre eux. Nous pouvons utiliser les classes définies précédemment.

Un obstacle est donc représenté par *nbr_sommets* sommets rangés dans un vecteur de points nommé *Sommets*.

Nous décidons d'orienter nos obstacles dans le sens trigonométrique, ce qui implique que les sommets correspondant doivent être rangés dans cet ordre dans le vecteur *Sommets*. Cela nous amène ensuite à définir un vecteur normal sortant à chaque segment, qui est en fait assimilé à un point par la connaissance de ses coordonnées. Enfin, nous implémentons une fonction *concave_convexe()* qui renvoie TRUE si l'obstacle est convexe, et FALSE dans le cas concave ; ainsi qu'une fonction *transfo* qui renvoie l'angle algébrique entre les segments AB et BC formés par les trois points A,B et C.

Ces fonctions nous servirons ultérieurement lors de la construction du graphe final.

2.4 La classe Graphe

Nous sommes maintenant en mesure de créer un obstacle. Néanmoins, il est envisageable que notre objet, voulant se déplacer de A à B, soit confronté à une multitude d'obstacles. L'ensemble de ces obstacles forme un graphe global, qui contient donc l'ensemble des sommets de tous les obstacles en plus des points de départ et d'arrivée. Plus qu'une liste de points, ce graphe doit être en mesure de fournir l'ensemble des segments que l'objet peut parcourir sur son chemin. Comment construire cet ensemble ?

Ce dernier ne se limite pas à la concaténation de tous les segments constituant les différents obstacles. Autrement, notre objet ne ourrait pas se déplacer entre les obstacles. Il ne peut pas non plus

être construit à partir de tous les segments qui relieraient tous les points de tous les obstacles. En effet dans ce cas, certains segments reliant deux sommets d'obstacles différents pourraient intersecter un autre segment propre à un obstacle. Toute la difficulté réside donc dans la détermination de l'ensemble de ces segments, que nous appelerons "segments valides".

Finalement, la classe *Graphe* doit contenir plusieurs attributs :

- Les points de départ et d'arrivée,
- L'ensemble des segments valides,
- Ainsi que le nombre de sommets totaux. En réalité, cette dernière information nous facilite l'implémentation des fonctions qui vont suivre.

On suppose que le constructeur de la classe *Graphe* prenne en entrée deux points, que sont les point de départ et d'arrivée, ainsi qu'une liste d'obstacle que l'on va devoir traiter pour construire la liste des segments valides.

Pour cela, nous décidons de séparer l'implémentation en plusieurs fonctions distinctes, que l'on va ensuite utiliser pour construire la liste de segments valides.

- Commençons par créer la fonction *IsIn*, qui, à partir d'un point A et d'un vecteur de points, renvoie TRUE si et seulement si le point A appartient à l'ensemble des points du vecteur.

- Intéressons nous ensuite à la fonction *intersect*. Celle-ci prend quatre points A, B, C et D, et renvoie TRUE si et seulement si les segments AB et CD s'intersectent.

- On peut alors implémenter la fonction *sumObstacles*. A partir de deux obstacles *ob1* et *ob2*, celle-ci renvoie un nouvel obstacle qui contient tous les sommets des deux obstacles *ob1* et *ob2*, et qui, grâce aux fonctions *intersect* et *IsIn*, n'a pu retenir que les segments valides, c'est à dire ceux n'intersectant pas un segment déjà existant.
  L'ensemble de ces segments existant est constitué au départ de tous les segments constituant les deux obstacles pris séparemment ; puis il s'agrandit au fur que l'on teste toutes les combinaisons de points (p_1, p_2) possibles (avec $p_1 \in ob1$ et $p_2 \in ob2$), où le segment formé par (p_1, p_2) ne croise pas l'un de ces segments déjà existant.

Il suffit alors de parcourir la liste d'obstacles donnée en arguments du constructeur du graphe, et déterminer la liste finale des segments valides en appliquant la fonction *sumObstacle* à tous les obstacles de la liste.

De plus, une précision concernant la fonction *sumObstacles* doit être apportée. Dans un souci de complexité, cette dernière ne renvoie pas vraiment un nouvel obstacle. En réalité, elle part de l'obstacle $ob1$ qu'elle modifie au fur et à mesure. En outre, elle change son nombre de sommets, et actualise la liste des segments valides. C'est pourquoi cette liste de segments valides ne se trouve pas dans la classe *Graphe*, mais dans la classe *Obstacle*.
Après avoir parcouru tous les obstacles, cette liste de segments valides est finalisée, et nou ssommes en mesure de la copier dans l'attribut *graphe_* de la classe *Graphe*.

2.5 L'algorithme de Dijsktra

Nous avons construit un graphe qui est une représentation de tous les segments valides, c'est à dire de tous les segments que l'objet peut parcourir pour aller du point de départ A au point d'arrivée B.

Chacun de ces segments a un poids qui représente sa longueur.

Notre objectif initial est de trouver le plus court chemin pour aller de A à B. Ce plus court chemin est forçément obtenu en parcourant certains segments de l'ensemble des segments valides. La question soulevée maintenant est donc la suivante : comment trouver les segments à parcourir pour minimiser la distance parcourue ?

2.5.1 Principe

Pour cela, nous allons implémenter l'algorithme de Dijkstra, couramment utilisé pour résoudre les problèmes de plus courts chemins.

On commence par numéroter les sommets du graphe de 1 à n. Quitte à renuméroter, nous pouvons supposer que l'objet part du sommet 1. On note E l'ensemble des segments valides, et d(i,j) la distance pour aller du sommet i au sommet j.

- L'algorithme prend en entré une matrice C, appelée matrice des coûts, et qui est telle que :

$$c_{ij} = \begin{cases} 0 & \text{si } i = j \\ \infty & \text{si} \quad i \neq j \quad \text{et} \quad (i,j) \notin E \\ d(i,j) & \text{si} \quad i \neq j \quad \text{et} \quad (i,j) \in E \end{cases}$$

- Il renvoie un vecteur $l = (l(1), \cdot, l(n))$ et un vecteur $p = (p(1), \cdot, p(n))$ tels que :
  - l(j) donne le coût minimal du chemin reliant le sommet 1 au sommet j,
  - p(j) donne le numéro du sommet qui précéde le sommet i dans le chemin de longueur minimal.

Algorithme 1 Algorithme de Dijsktra

Entrées : N nombre de sommets du graphe, c matrice des coûts du graphe, d matrice des poids du graphe, S, T

pour $j = 1, N$ **faire**
  $l(j) = c_{1j}, \quad p(j) = 0$
  si $j > 1$ **et** $c_{1j} < \infty$ **alors**
    $p(j) = 1$
  fin de test si
fin de boucle pour
while T n'est pas vide **faire**
  Choisir i dans T tel que $l(i)$ minimum
  Retirer i de T et l'ajouter à S
  pour chaque successeur $j \in T$ de i **faire**
    si $l(j) > l(i) + d(i,j)$ **alors**
      $l(j) = l(i) + d(i,j), \quad p(j) = i$
    fin de test si
  fin de boucle pour
end while

Sorties : l, p

FIGURE 3 – Algorithme de Dijkstra

Cet algorithme a l'avantage de nous donner directement le coût minimal pour aller du sommet 1 vers n'importe quel sommet. Dans notre cas, si l'on suppose que le point d'arrivée B porte le numéro n, il suffit de regarder la valeur l(n) pour conaître cette distance minimale.

De plus, la valeur de p(n) nous indique le sommet qui précède le point d'arrivée dans le chemin recherché. De manière récursive, nous pouvons en déduire l'ordre des sommets que notre objet doit parcourir pour réaliser la mission demandée.

2.5.2 Construction de la matrice des coûts

Afin d'utiliser l'algorithme de Dijkstra, nous devons implémenter une fonction permettant de créer la matrice des coûts.

On rappelle que le coefficent c(i,j) représente la distance pour aller du sommet i au sommet j. Si le segment formé par les points i et j n'est pas valide, cette valeur est $+\infty$. De même, il est naturel de dire que c(i,i)=0.

On commence par construire une fonction *initC* ayant pour but d'initialiser la matrice C. Cette fonction réalise trois tâches distinctes :

- Elle crée une matrice C de taille $n \times n$,
- Elle initialise tous ses coefficients en leur attribuant la valeur $+\infty$,
- Elle place la valeur 0 sur les coefficients diagonaux.

On crée ensuite une fonction *buildMatrixC*. Après avoir appelé la fonction d'initialisation détaillée précédemment, *buildMatrixC* parcours le graphe *graphe_*, et attribut la distance d(i,j) aux coefficients c(i,j). On a pour cela besoin d'un tableau de points *memory* qui, lors du parcours du graphe, attribut à chaque sommet un numéro arbitraire (sauf pour les points de départ et d'arrivée qui possèdent respectivement les numéros 1 et n). Ce tableau permet de faire le lien avec les coefficients (i,j) de la matrice C, puisque l'on identifie chaque point au numéro attribué par memory.

Il est a noter que cette fonction utilise une autre version de la fonction *IsIn*. En outre, cette autre fonction *IsIn* prend en argument un point A, le tableau *memory* ainsi que le nombre de sommets du graphe, et renvoie l'index du point A dans le tableau *memory*. Si ce dernier ne contient pas le point A, alors *IsIn* renvoie -1.

2.5.3 Chemin Final

Finalement, on obtient un vecteur p et l par l'algorithme de Dijkstra comme décrit précédemment. Le vecteur p ne nous donnant pas l'ordre des points à parcourir directement, nous décidons de conclure grâce à une fonction *cheminFinale*, qui, à partir du vecteur p et du tableau *memory*, renvoie un vecteur *chemin* donnant les sommets à parcourir dans l'ordre pour aller du point de départ A=p(1) au point d'arrivée B=p(n) par la chemin le plus court.

3 Résultats