MAP - SIM2

Planification de Trajectoire

Marine NEYRET
Imad EL HANAFI
Hicham KOUHKOUH
Promotion 2017

Maître de conférences :
M. Éric Lunéville

17 mars 2016

Table des matières

Introduction

Le problème de planification de trajectoire s'appuie sur un ensemble de techniques mathématiques et informatiques permettant de déterminer la meilleure trajectoire possible entre deux points tout en respectant certaines contraintes : présence d'obstacles, objet volumineux, ... Dans ce projet, on s'appuiera sur des méthodes déterministes, *ie.* complètes en résolution, pour résoudre un problème relativement simple, dans le sens où on est certain de l'existence et de l'unicité de la plus courte trajectoire.

C'est un domaine de recherche très actif, mêlant à la fois optimisation et théorie des graphes (Recherche Opérationnelle), tout en élaborant des algorithmes avec des considérations géométriques.

1 Méthode choisie

Afin de résoudre ce problème, nous allons, dans un premier temps, créer de nombreuses classes indépendantes qui vont nous permettre d'obtenir un ensemble de polygones (obstacles) et des points de départ et d'arrivée. Nous avons donc créé les classes :

- `Sommet` qui permettra d'obtenir des sommets tout en surchargeant les opérateurs qui nous seront utiles
- `Segment` qui permettra de créer un segment comme un vecteur de sommets
- `Polygone` qui permettra de créer un polygone, ou obstacle, comme liste de sommets
- `Vecteur` de taille deux, une classe annexe utile dans l'implémentation de fonctions des classes précédentes.

Dans un second temps, nous allons implémenter l'algorithme de Dijkstra qui nous permettra de calculer la trajectoire minimale entre le départ et l'arrivée. Pour cela, nous avons besoin des classes suivantes :

- `scene` qui permettra de créer la liste des polygones de la scène et de lui associer la liste de l'ensemble des sommets des polygones ainsi que le nombre de ces derniers
- `ArcPlanification` qui hérite de la classe `Arc` qui établit toutes les connexions possibles entre les sommets et leur associe une longueur (infinie quand cette dernière n'est pas possible)
- `Graphe` qui permet de récupérer une liste d'Arcs et une matrice globale de coûts
- `Matrice`, une classe annexe à la classe graphe afin de pouvoir créer la matrice globale des coûts associée à un certain Graphe
- `vecteur` de taille variable, une classe annexe utile pour l'algorithme.

Les classes `Matrice` et `Vecteur` sont des classes classiques dans la programmation scientifique, indispensables pour manipuler les différents objets et auxquelles plusieurs fonctions font appel. De la même façon, les classes `Sommet` et `Segment` sont classiques et elles sont simplement adaptées aux besoins de notre projet.

Enfin, il faut pouvoir lire un fichier d'entrée qui permet de récupérer les polygones que l'on souhaite avoir dans la scène. De plus, il faut pouvoir afficher la trajectoire minimale obtenue, ainsi que les polygones (obstacles) de la scène. Cette dernière étape sera effectuée à l'aide de MatLab.

2 Organisation du travail

Tout d'abord, pendant les deux premières séances, nous avons choisi de travailler en groupe afin d'avoir des classes et fonctions de base avec des notations cohérentes.

Lors de la troisième séance, Imad a développé l'algorithme de Dijkstra pendant que Marine et Hicham se sont attelés à la programmation des lectures de fichiers tout en finissant les fonctions des classes, notamment la fonction intersection.

Entre la troisième et la fin de la quatrième séance, nous avons essayé l'algorithme de Dijkstra sur plusieurs matrices et nous avons développé une première fonction `ConstructGraphe` qui va connaître de nombreuses modifications jusqu'à la fin. C'est aussi à ce moment que l'on a créé les fonctions d'accès (voir plus bas) et que nous avons commencé à réfléchir sur l'affichage sur MatLab.

Nous sommes arrivés à la cinquième séance en ayant bien avancé pendant les vacances : l'affichage sur MatLab et la fonction `ConstructGraph1` étaient bouclés. Une fois tous les fichiers mis bout à bout, des erreurs de segmentation sont apparues. Ces erreurs ont été réglées lors de la cinquième séance tout comme les problèmes relatifs aux obstacles (polygones non convexes, diagonales...).

Durant la dernière séance, nous avons réalisé la fonction `ConstructGraph2` associée au padding.

3 L'implémentation

Nous considérons que le fichier d'entrée du programme suit le modèle suivant :
`#Nombreobstacles`
`Chiffre`
`#Polygone`
`Chiffre` qui correspond au nombre de sommets
`x1 y1` les coordonnées du premier sommet du polygone
`x2 y2` les coordonnées du second sommet du polygone
`x2 y2` les coordonnées du troisième sommet du polygone, *etc*

Remarque : nous pouvons mettre autant de polygones que l'on veut et chaque polygone a autant de sommets que l'on veut. Cependant, nous considérons que les polygones sont strictement disjoints deux à deux.

3.1 Lecture de fichier

La lecture de fichier se fait en 3 étapes emboitées : la lecture de la scène, qui fait appel à la lecture d'un polygone, qui fait appel à la lecture d'un sommet. Les trois fonctions s'appellent `Lecture` et se différencient selon l'entrée.

La lecture de sommet récupère simplement les coordonnées de ce dernier.

Ensuite, la lecture de polygone commence par passer la ligne `#Polygone`, puis elle récupère le nombre de sommets dans le polygone avant de lire chaque sommet du polygone grâce à la lecture de sommet utilisée dans une boucle `while` qui s'arrête au nombre de sommets.

Enfin, la lecture de scène permet de lire le fichier d'entrée dans sa globalité. Elle commence par passer la ligne `#Nombreobstacles` afin de récupérer le nombre d'obstacles dans la scène. Pour notre scène, nous voulons alors avoir accès au nombre de sommets total dans la scène, à une liste de sommets dans l'ordre et une liste de polygones. Pour la liste des polygones nommée `polygones`, nous lisons le polygone avec la fonction `Lecture` précédente et nous utilisons pushback pour l'ajouter à la liste `polygones`. Parallèment, on incrémente `Nbsomscene`, le nombre de sommets de la scène. Pour la liste des sommets `sommetsPol`, nous utilisons un itérateur qui parcourt les sommets du polygone en cours et les ajoute dans la liste. Il reste alors à fermer le fichier.

Parallèlement, en utilisant un script MatLab, on lit ligne par ligne un fichier `.txt` qui suit le modèle précédent pour nous permettre d'afficher la scène.

3.2 La classe `Polygone`

Les polygones sont les objets phares de notre programme, dans la mesure où ils représentent les obstacles au milieu desquels nous nous frayons le plus court chemin.

Pour cela, il fallait envisager diverses possibilités d'obstacles : convexes, non convexes, et avec un nombre de sommets à priori indéfini.

Nous avons donc envisagé deux constructeurs de polygones à 3 ou 4 sommets qui reposent sur des `pushback` de sommets. Nous avons ensuite créé une fonction `add` interne à la classe, qui ajoute un sommet à un polygone à 4 sommets. Cette dernière permet donc de créer des polygones qui ont un nombre de sommets quelconque. Cependant, nous avons choisi une méthode de programmation qui repose sur un fichier d'entrée à lire donc ces fonctions ne nous ont pas servi par la suite : nous créons les polygones directement dans la lecture de la scène. Un constructeur par lecture nous a alors semblé plus adéquat.

Remarque : nous avons imposé que les sommets du polygone doivent être écrits (listés) dans un ordre qui évite toute intersection entre ses arêtes. Aussi, le nombre de sommets doit être au moins 3.

3.2.1 La fonction `normale`

C'est une fonction de la classe `Polygone`, qui prend en argument un entier (l'indice du sommet qui permet de situer le côté du polygone dont on veut la normale) et qui renvoie un vecteur unitaire. Autrement dit : `P.normale(int i)` renvoie la normale (vecteur unitaire) entrante relative au segment $[S_i, S_{i+1}]$.

Remarque : cette fonction sert en outre à s'orienter par rapport à l'intérieur ou l'extérieur du polygone, surtout quand il s'agit de polygones non convexes ou pour éliminer les arcs diagonaux.

3.2.2 La fonction `intersection`

C'est une fonction de la classe `Polygone`, qui prend en argument un `segment` et renvoie 1 s'il y a intersection avec le polygone courant et 0 sinon.

D'abord, on vérifie que les deux segments ne sont pas parallèles. Ceci en calculant la $3^{\text{ème}}$ composante de leur produit vectoriel (en effet, et comme on est dans un plan, les deux autres composantes sont nulles). Ensuite, on boucle sur l'ensemble des sommets du polygone en question et on teste à chaque fois le segment à partir du sommet où l'on se trouve avec le segment en argument. Pour ce faire, on paramétrise chacun des deux segments avec un α et un β qui représentent les coordonnées (barycentriques) du point d'intersection en fonction des deux sommets de chaque segment. On teste enfin si ces deux paramètres sont strictement inclus dans $]0, 1[$ (ainsi, si le point d'intersection est confondu avec l'un des sommets, alors la fonction renvoie 0, *ie.* pas d'intersection).

Cas particulier : quand le sommet sur lequel on se trouve (notons le C) appartient au segment en entrée, la fonction renvoie 0 (*ie.* pas d'intersection). On doit donc tester cette condition au début de la fonction en considérant les distances entre C et les sommets du segment d'entrée. Si par exemple le segment d'entrée est [A,B], on vérifie que $C \notin [A, B]$. Ceci élimine le cas problématique où B est l'un des sommet du polygone et A est dans la prolongation d'une diagonale partant de B.

3.2.3 La fonction `diag`

La fonction `diag` prend en argument deux entiers qui sont les indices de deux sommets d'un polygone et permet de vérifier si l'arc qui les relie se trouve à l'intérieur ou à l'extérieur du polygone. La fonction renvoie alors 0 ou 1 selon si l'arc est à l'extérieur ou pas.

Pour ce faire, on vérifiera la position des deux sommets (précédent et suivant) par rapport aux sommets en entrée et en sortie. On calcule alors des produits vectoriels, dont le signe permet de repérer l'arc par rapport au polygone.

3.3 Les fonctions d'accès

Lorsque nous codions notre programme, comme nous avons utilisé des listes, nous nous sommes vite rendu compte que l'accès aux différents polygones et différents sommets était lourd. Nous avons donc créé trois fonctions d'accès (aux sommets d'un polygone, aux polygones d'une scène et aux sommets d'une scène) afin que nos autres fonctions soient moins lourdes et plus compréhensibles. Les trois fonctions s'appellent `Acces` et se différencient selon la classe où elles se trouvent et leur sortie.

Chaque fonction d'accès prend en entrée un entier qui représente le polygone ou le sommet auquel on veut accéder. Elles utilisent ensuite un itérateur qui permet, à travers une boucle for, de se déplacer dans la liste et d'atteindre le polygone ou le sommet voulu.

La fonction d'accès à un sommet d'un polygone traite aussi des cas particuliers qui posaient problème dans les autres fonctions. En effet, comme dans la liste de sommets qui représente un polygone on ne remet pas le premier sommet en dernière position pour fermer le polygone, certaines de nos fonctions avaient des problèmes. Ainsi, nous avons ajouté les cas particuliers suivants :

— si l'entier d'entrée est -1, on considère que l'on veut accéder au dernier sommet
— si l'entier d'entrée est égal au nombre de sommets dans le polygone, on considère que l'on veut revenir au premier sommet pour fermer le polygone.

3.4 Les classes arc et Graphe

La classe `Arc` est une classe fantôme qui permet de coder l'algorithme de Dijkstra de façon neutre, c'est à dire que l'on peut réutiliser avec un autre type d'arc qui n'a pas forcément un coût égal à sa longueur.

Nous avons donc fait hériter de cette classe une classe `ArcPlanification` qui permet de créer les arcs propres à notre problème. Ainsi, nous pouvons construire des arcs qui sont composés de deux sommets et d'une longueur (celle entre les deux sommets).

La classe `Graphe` permet simplement de créer un graphe vide à utiliser dans les fonctions ConstructGraphe. Le graphe vide est composé d'une liste `arcs` d'Arc.

3.5 La fonction `ConstructGraphe1`

C'est la fonction maîtresse de notre programme, puisqu'elle brasse quasiment toutes les fonctions et classes que nous avons implémentées.

C'est une fonction externe à toutes les classes, qui prend en argument la scène (contenant l'ensemble des obstacles) ainsi que deux sommets : celui de départ et celui de l'arrivée, et qui renvoie un `Graphe`. Le plus important dans ce dernier c'est de récupérer la matrice de coût, qui pondère tous les arcs entre tous les sommets.

Cette fonction opère en deux grandes étapes :

(i) Construire la matrice de coût relative aux sommets des polygones entre eux
Ceci consiste à évaluer toutes les connexions entre tous les sommets des polygones et de mettre la longueur entre les sommets dans la matrice de coûts à la bonne position. On s'occupe donc d'abord des arcs au sein d'un polygone et on remplit la matrice de coût uniquement si le segment ne passe pas par l'intérieur du polygone. Puis on relie les sommets d'un polygone avec les sommets de tous les autres polygones. Ici, il faut tester les intersections entre le segment qui lie deux sommets de deux polygones différents et tous les polygones. On utilise la fonction `intersection` pour assigner à l'arc soit une valeur égale à la distance entre les deux sommets qu'il relie si cette fonction renvoie 0, soit une valeur infinie si elle renvoie 1.

(ii) Construire la matrice de coût "globale" qui prend en compte le départ et l'arrivée. Il faut tout d'abord vérifier que ces derniers ne sont ni sur un sommet du polygone, ni sur une arrête. Si c'est le cas, on décale infimement le départ ou l'arrivée afin que le chemin ne passe pas dans le polygone où il se trouve.

— Connecter le sommet de départ avec les sommets des polygones
On vérifie que le segment entre le départ ou l'arrivée et un sommet d'un polygone n'intersecte aucun polygone de la scène. La fonction intersection est utilisée comme précédemment.
— Connecter le sommet d'arrivée avec les sommets des polygones
On procède de la même manière.
— Connecter le sommet de départ et d'arrivée. On procède de la même manière sauf que l'on a qu'un seul segment ici.
— Inclure la matrice de coût des polygones dans la matrice globale des coûts

Test et validation

Pour vérifier cette fonction, on a effectué plusieurs tests simples dont on connait la solution, et on a affiché la matrice des coûts correspondant. Notamment, nous avons essayé la fonction avec des scènes simples avec un seul obstacle.

3.6 La fonction `ConstructGraphe2`

Elle reprend le même schéma que `ConstructGraphe1` mais avec un *double* en plus en argument qui correspond au rayon de l'objet circulaire qu'on suppose bouger et une nouvelle `scene`. En effet, cette dernière sera la nouvelle scène avec donc les nouveaux obstacles redimensionnés selon le rayon de l'objet qu'on considère bouger entre le point de départ et d'arrivée, selon la méthode du *padding*.

D'abord, on a besoin de La fonction `Theta`. C'est une fonction extérieure aux classes, qui prend en argument un polygone et qui renvoie un vecteur contenant tous les angles du polygone, dans le même ordre que ses sommets.

Ensuite, plusieurs cas sont à prendre en compte suivant chaque angle du polygone.

3.6.1 Si $0 < \theta \leq \frac{\pi}{2}$

On prolonge le sommet S_i suivant la directrice de sa deuxième arête, puis on crée plusieurs sommets dans le sens trigonométrique, en considérant respectivement la rotation de l'angle $k\frac{\theta_i}{n}$ pour créer le $k^{ième}$ sommet de la discrétisation

3.6.2 Si $\frac{\pi}{2} < \theta < \pi$

Deux cas sont à traiter :

le cas où les deux arêtes sont de longueur > au rayon de l'obstacle
On reprend le même schéma que dans la sous-section 3.6.1.

Le cas où l'une des arête est de longueur < au rayon de l'obstacle
Dans ce cas-ci, il y aura un problème dans le padding puisqu'on aura deux zones qui vont se superposer si on procède de la même manière. On a donc translaté le sommet en question suivant la bissectrice de l'angle extérieur $2\pi - \theta$ (qui est la même que celle de θ) avec une distance égale au rayon de l'obstacle.
Ce qui résout le problème.

3.6.3 Si $\pi < \theta < 2\pi$

Là encore, on décale le sommet suivant la bissectrice en procédant comme dans le $2^{ème}$ cas de 3.6.2.

3.6.4 La construction de `Graphe`

Une fois la nouvelle scène construite, on applique le code de `ConstructGraphe1` à cette dernière, en considérant un point ponctuel.

3.7 La fonction `dijkstra`

Basée sur l'algorithme de Dijkstra (comme son nom l'indique), cette fonction, externe à toutes les classes, prend en argument une matrice (celle des coûts, et qui est fournie par `ConstructGraphe`), `N` le nombre de sommets et un `Vector` qui va contenir les sommets du chemin minimal entre le départ et l'arrivée (et qui sera vide au début).

Pour implémenter l'algorithme de Dijkstra on a utilisé un vecteur `dist(N)` qui contiendra les distances minimales entre le sommet de départ et celui d'arrivée), un vecteur booléen `Set[N]` pour indiquer si on a visité un sommet i pendant les itérations de l'algorithme et enfin un vecteur de taille N qui donne le sommet qui précède i dans un chemin.

Test et validation

On a vérifié que notre code pour la fonction Dijkstra renvoie la bonne solution en le testant avec des exemples déjà résolu qu'on a trouvés sur internet ou qu'on a faits en cours de MAP-RO.

Par exemple, nous avons considéré le problème suivant :

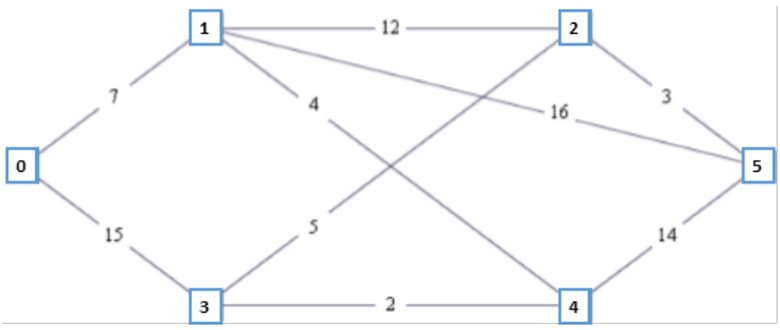

FIGURE 1 – Graphe test pour Dijkstra

La solution théorique pour aller de 0 à 5 passe par les points 1 4 3 2 et son poids est de 21.

Voici la solution donnée par notre algorithme de Dijkstra :

```
Affichage de la matrice des couts
0       7       1e+09   15      1e+09   1e+09
7       0       12      1e+09   4       16
1e+09   12      0       5       1e+09   3
15      1e+09   5       0       2       1e+09
1e+09   4       1e+09   2       0       14
1e+09   16      3       1e+09   14      0

Le point   Distance from Source
0               0
1               7
2               18
3               13
4               11
5               21
Solution :
le cout:
21
Trajet :
5
2
3
4
1
0
```

FIGURE 2 – Solution de notre algorithme de Dijkstra

On trouve bien la solution proposée par la correction du problème.

3.8 Résumé des fonctions majeures

On voit dans le schéma ci-dessous tous les liens entre les fonctions majeures de notre code et les classes.

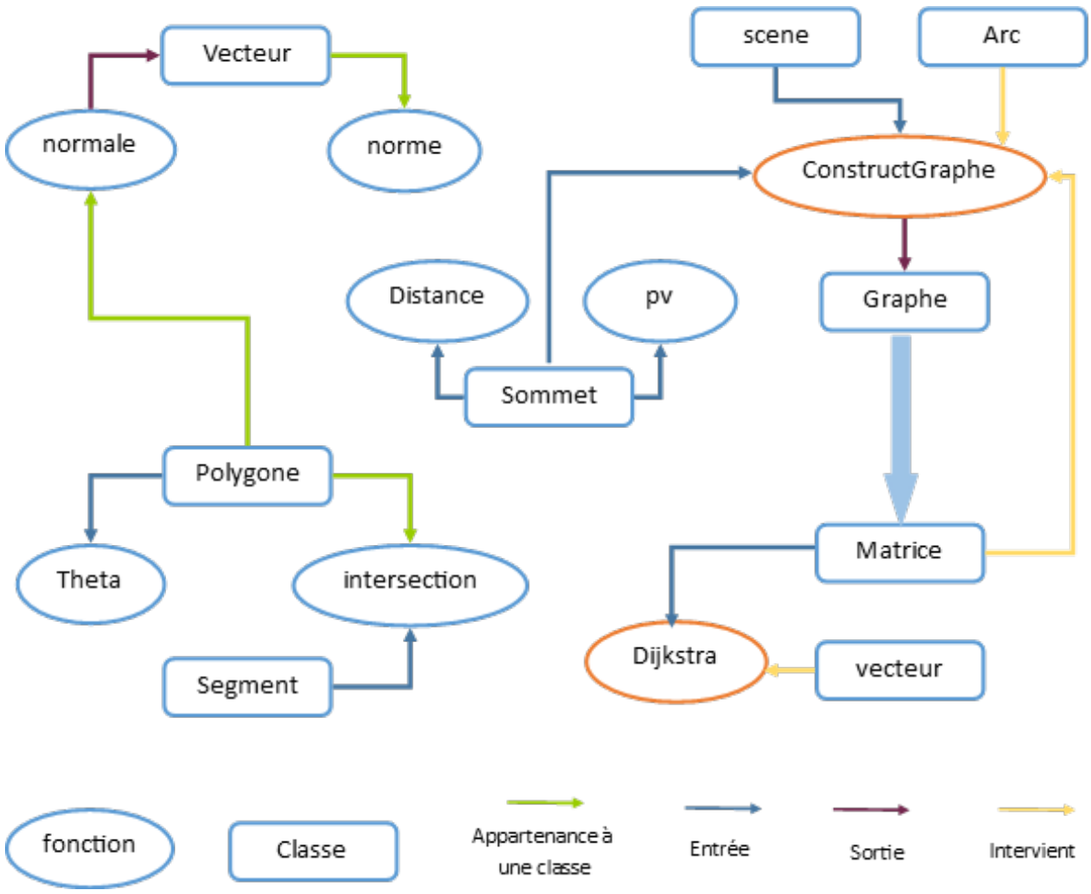

FIGURE 3 – Résumé des fonctions majeures

3.9 L'affichage sur MatLab

L'affichage s'effectue en deux étapes : afficher la scène, puis la solution.
L'affichage de la scène se fait en s'appuyant sur la lecture du fichier `test.txt`. La solution quant à elle est affichée selon si on utilise le *padding* ou pas. On a alors écrit deux scripts MatLab : `affichage.m` et `affichage1.m`. Dans le cas du padding, on affichera non pas des points, mais des cercles avec un rayon R et dont les centres sont les sommets donnés par Dijkstra.

Nos scripts suivent l'algorithme suivant :

— On récupère les sommets des polygones (depuis le fichier `test.txt`)
— On récupère les coordonnées des sommets du chemin optimal dans un fichier `sol.txt` grâce à une fonction auxiliaire `affichage` qui prend en argument le vecteur des solutions donné par Dijkstra, la scène, les points de départ et d'arrivée ainsi que le mode 0 ou 1 selon si on choisit le padding ou pas.
— On affiche la trajectoire en reliant ces sommets

4 Interface d'utilisateur

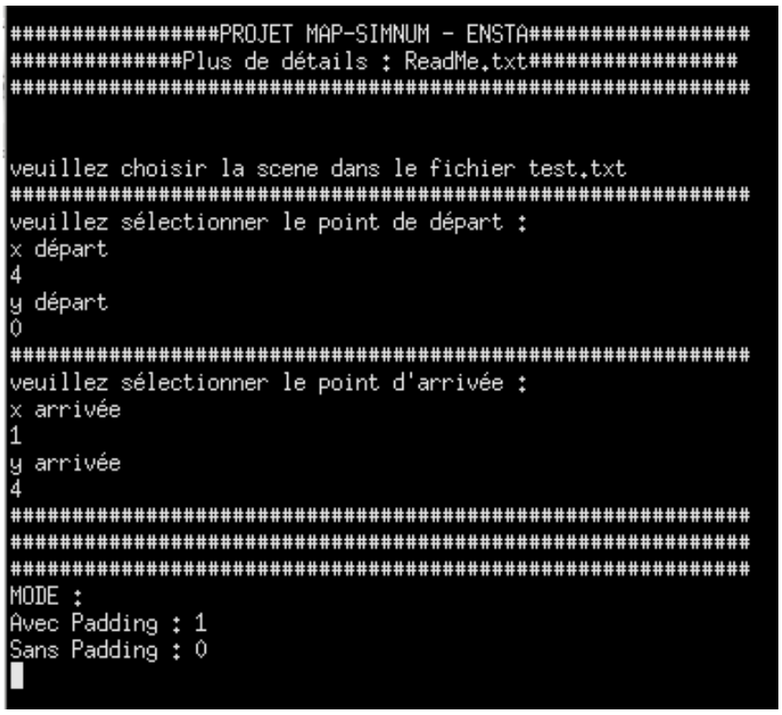

FIGURE 4 – Interface proposée à l'utilisateur

Grâce à l'interface qu'on propose, l'utilisateur peut définir le point de départ et le point d'arrivée de la trajectoire. Il peut aussi choisir le mode qu'il souhaite prendre pour la résolution : c'est à dire sans padding (`mode 0`) et avec padding (`mode 1`).

5 Les résultats

5.1 Premières validations

Une fois les fonctions majeures implémentées, nous avons commencé à les tester sur des cas simples afin de vérifier que la base fonctionne. Par exemple, si on considère une scène avec un polygone unique et convexe, on observe qu'il n'y a aucun problème ni avec ni sans padding. Le chemin le plus court est bien emprunté pour relier les deux points

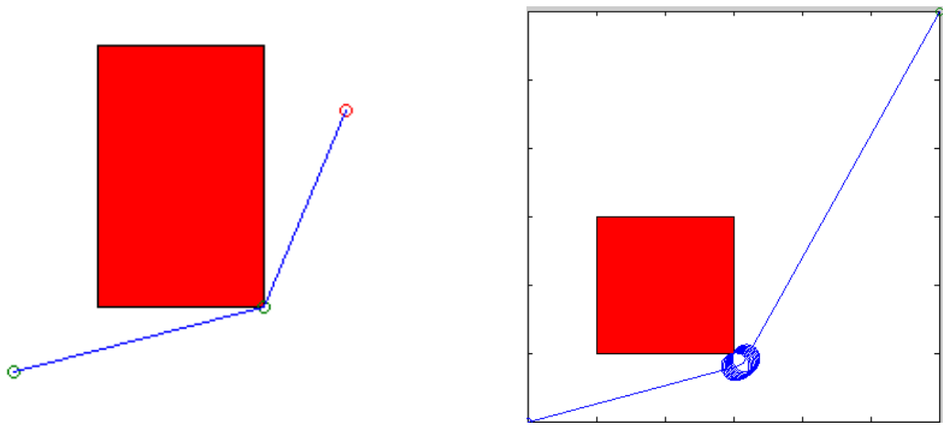

FIGURE 5 – Premiers tests (obstacle simple) avec et sans padding

Cependant, nous nous sommes vite rendu compte que certains cas particuliers n'étaient pas traités avec nos fonctions dans leur état à l'instant du test.

Nous avons rencontré des problèmes pour :

— les polygones non convexes
— les cas où les points de départ et/ou d'arrivée sont sur un polygone
— les cas où les points de départ et/ou d'arrivée sont dans le prolongement d'une diagonale

Ces problèmes ont été réglés pour le cas sans padding.

Pour le padding, les points de départ et d'arrivée sont considérés dans un obstacle dès lors que la distance avec l'obstacle est inférieure au rayon. Ceci est normal puisqu'on refait une nouvelle scène avec les obstacles redimensionés et on suit le schéma de `ConstructGraph1` où on n'a pas traité ce cas puisqu'il n'a pas de sens.

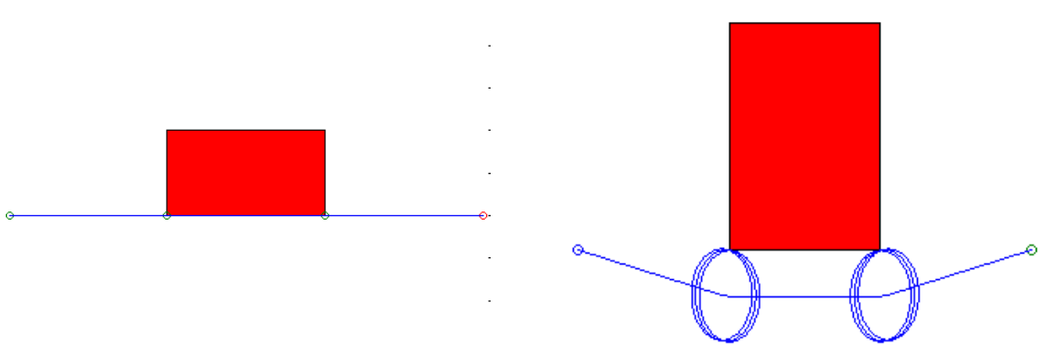

FIGURE 6 – Tests avec et sans padding sur des chemins confondus avec des arêtes

5.2 Résolution des problèmes

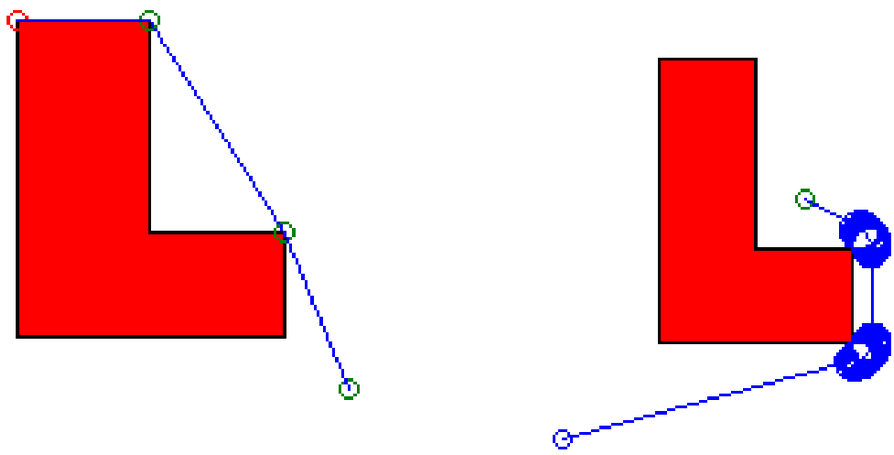

FIGURE 7 – Tests avec et sans padding sur des obstacles non convexes

Nous sommes ici dans le cas de polygones non convexes.

Dans la géométrie sans padding, nous remarquons qu'au niveau du creux du L nous passons directement du sommet i au sommet i+2, ce qui correspond bien au trajet le plus court. Nous voyons aussi que lorsque le point de départ ou d'arrivée appartient au polygone il n'y a pas de problème. En effet, le décalage infime de ce point ne change en rien a longueur de la trajectoire mais elle permet de sortir le point du polygone et donc de ne pas avoir un chemin qui passe à l'intérieur du polygone.

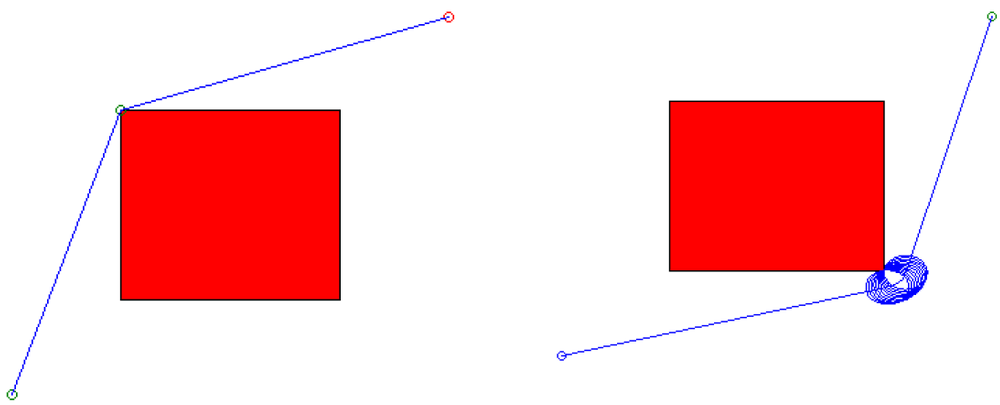

FIGURE 8 – Tests avec et sans padding quand le départ et l'arrivée sont sur un diagonale

Lorsque les points de départ et d'arrivée sont sur le prolongement d'une diagonale, nous évitons le problème qui est d'aller tout droit en suivant la diagonale, que ce soit avec ou sans padding. Ceci n'était initialement pas prévu par notre fonction d'intersection dans la mesure où nous utilisions des ouverts $]0, 1[$ pour paramétrer les coordonnées barycentriques du point d'intersection. On a donc rajouté de nouvelles conditions dans la fonction `intersection` qui vérifie justement si on est sur une diagonale ou pas.

5.3 Cas plus complexes

Nous avons par la suite utilisé des scènes où le nombre de polygones est plus grand et où les types de polygones varient.

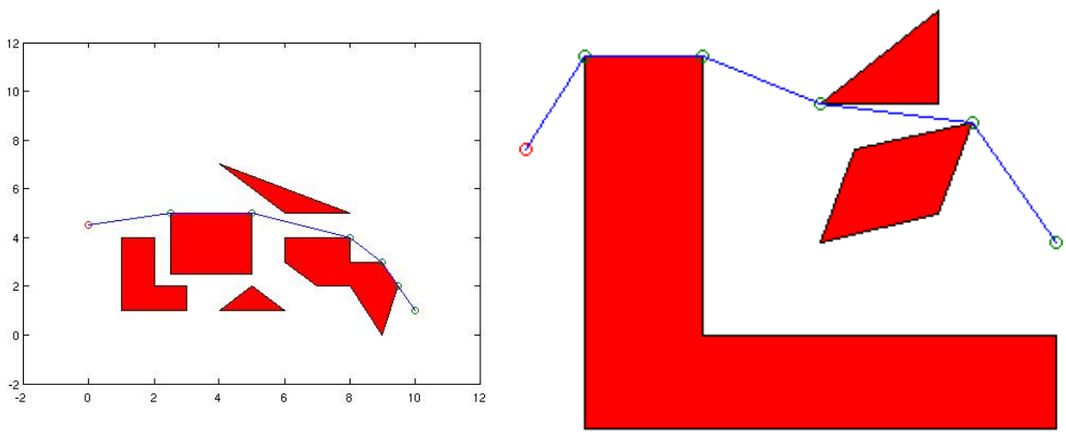

FIGURE 9 – Complexification de la scène pour des tests sans padding

Nous remarquons que la mise en commun de problèmes dans une même scène sur différents polygones n'affectent pas la bonne résolution de notre programme. Aucune déviation absurde du trajet n'apparaît.

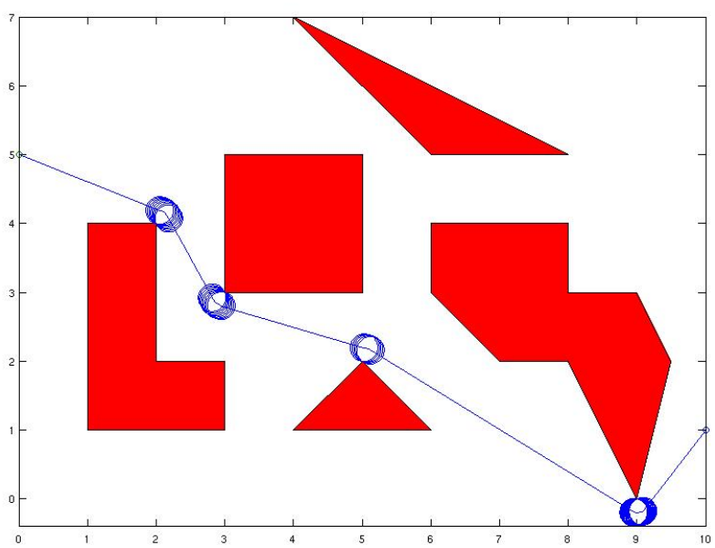

FIGURE 10 – Complexification de la scène pour des tests avec padding

Ici encore la solution donnée est bien la trajectoire minimale et on peut observer une très bonne application du padding quand les angles des polygones autour de la trajectoire sont inférieurs à Π.

5.4 Problème persistant

Cas particulier

Dans la fonction `Polygone::diag(int i, int j)`, on n'a pas traité le cas où on est sur un polygone non-convexe dont une arête appartient strictement à un arc reliant deux sommets non successifs.

Néanmoins, nous avons proposé une solution, mais n'avons pas pu l'implémenter correctement. Cette dernière consiste à évaluer l'orientation des sommets compris entre ces deux sommets (à l'aide du signe des produits vectoriels).

5.5 Cas où deux polygones sont partiellement confondus

On a pensé à adapter l'algorithme de Weiler-Atherton à notre code, pour trouver l'union de deux polygones dont l'intersection est non vide, et n'en faire qu'un seul polygone. Par manque de temps, nous n'avions pas pu coder l'algorithme.

Conclusion

Ce projet nous a permis d'utiliser et de mettre en pratique les connaissances acquises en MAP-SIM1. Nous sommes ainsi plus à l'aise avec la programmation orientée objet.

Le projet en lui même nous a grandement intéressé même si les nombre de cas particuliers n'ont pas été simples à résoudre.

On se rend finalement compte que l'on aurait du passer une séance au début à essayer de lister tous les cas particuliers possibles qui nous venaient en tête. Ceci aurait permis d'avoir des fonctions plus directes avec moins d'ajouts pour chaque cas particulier.

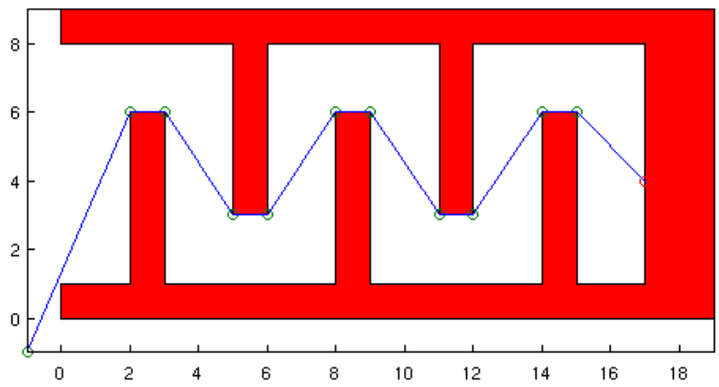

Figure 11 – TADAAAAAM !