Projet de simulation numérique

Planification de trajectoire

Réalisé par :

Khaled Bouguila

Mohamed Dziri

Faïez Khalfallah

Encadré par :

Mme Sonia Alouane

2A Mathématiques Appliquées

Année universitaire : 2022/2023

Table des matières

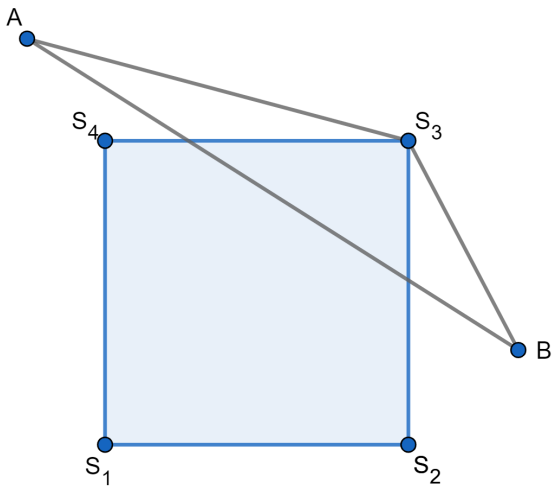
Introduction

La planification de trajectoire est un problème crucial dans de nombreux domaines tels que la robotique, l'automatisation industrielle, la navigation aérienne et la logistique. Le but de la planification de trajectoire est de trouver le chemin le plus court ou le plus optimal entre deux points tout en évitant les obstacles sur le trajet. Ce projet vise à résoudre ce problème en utilisant la modélisation des obstacles sous forme de sommets et de chemins reliant ces obstacles par un graphe. L'algorithme de Dijkstra est ensuite utilisé pour trouver le chemin le plus court dans ce graphe, permettant ainsi de planifier la trajectoire à suivre par l'objet. La première partie de ce projet consiste en la modélisation des entités mises en jeu dans le problème et leur implémentation en C++. Le code est conçu en suivant les principes de la programmation orientée objet pour assurer la modularité et la réutilisabilité du code. Les obstacles sont représentés par des courbes fermées dans le plan. Par une méthode de discrétisation bien adaptée, on transforme ces courbes en des polygones dont les sommets sont utilisés pour créer un graphe où chaque sommet représente un point libre de l'obstacle. Les arêtes du graphe sont formées par les chemins reliant les sommets de chaque obstacle. Ensuite, l'algorithme de Dijkstra est mis en oeuvre pour trouver le plus court chemin entre les points de départ et d'arrivée en évitant les obstacles. Pour visualiser les résultats, on a eu recours à Python, notamment les librairies `matplotlib.pyplot` et `matplotlib.animation`.

Afin de résoudre ce problème, notre démarche débutera en assimilant l'objet en déplacement à un point matériel. Nous étendrons ensuite nos recherches aux objets non ponctuels, pouvant être réguliers ou irréguliers sans rotation autour d'eux-mêmes. Enfin, nous proposerons une méthode pour résoudre le même problème, mais cette fois-ci avec la prise en compte de la rotation.

1 Principe de la méthode

En l'absence de dynamique, la trajectoire minimale reliant un point A à un point B est un segment $[AB]$, à condition qu'il n'intercepte pas d'obstacles. Si le segment croise un obstacle, il est évident que la trajectoire minimale reliant A et B passera par un sommet de l'obstacle.



Ainsi, nous allons créer un graphe en utilisant les sommets des obstacles, où chaque arc sera une connexion entre deux sommets ayant une valeur correspondant à la longueur du segment reliant ces sommets, si le segment ne croise aucun obstacle. Si le segment rencontre un obstacle, la valeur de l'arc correspondant sera infinie. Ensuite, nous ajouterons à ce graphe les points de départ et d'arrivée prédéfinis, ainsi que les connexions possibles avec tous les sommets des obstacles.

2 Implémentation

Les classes et les fonctions associées sont définies en `C++` dans les fichiers `trajectoire.cpp` et `trajectoire.hpp`.

2.1 Sommet

Un **sommet** est un point dans le plan auquel on associe deux coordonnées `Sx` et `Sy` ainsi que deux entiers `idx` et `sens` qui n'ont pas d'intérêt initialement (On en parlera avec détails dans 4.3)

2.2 Segment

Un **segment** est constitué de deux sommets `deb` et `fin` qui constituent ses extrémités. Notons bien que le segment est orienté. Cela implique que la permutation de `deb` et `fin` fait changer le segment.

2.3 Obstacle

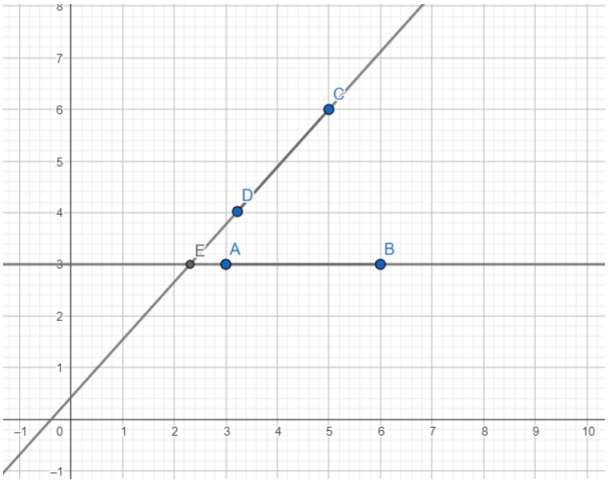
Nous utiliserons une classe **obstacle** qui décrira chaque obstacle par ses sommets ordonnés. Il sera important d'avoir les normales pour chaque arête de l'obstacle afin de détecter son intérieur. Le sens de création des sommets est considéré trigonométrique par convention. On supposera, sans perte de généralité, que tous les obstacles traités sont convexes (on en parlera en détails dans 4.1).

Dans cette classe, on a implémenté une fonction qui détecte l'intersection entre un segment (ouvert) et un obstacle. Pour ce faire, on a eu recours à deux autres fonctions.

2.3.1 La fonction `intersects_segment`

Cette fonction est membre de la classe **segment** et prend en argument un `const segment& seg_obs`. Elle retourne un booléen qui prend la valeur `true` s'il y a intersection entre le segment courant `*this` et `seg_obs`.

Pour cela, la fonction commence par créer une nouvelle base à partir du vecteur directeur du segment courant et de sa normale. Elle projette ensuite les extrémités du segment courant et celles du segment `seg_obs` dans cette nouvelle base. Ce faisant, on retourne toujours vers le cas affiché dans la figure, où $[AB]$ désigne le segment courant qui se trouve maintenant en position horizontale et $[DC]$ le segment `seg_obs` :



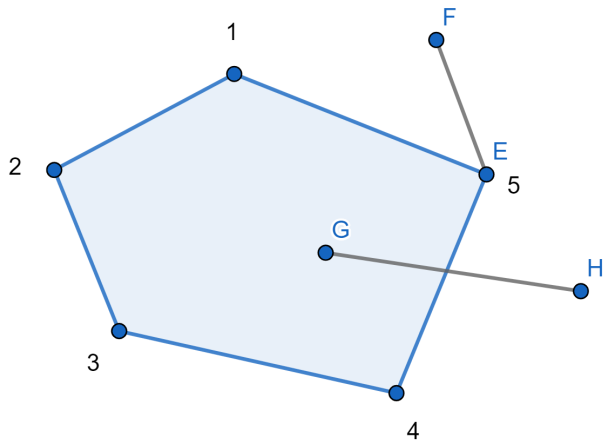
Elle calcule ensuite les coordonnées x et y du point d'intersection E des droites passant par ces deux segments et vérifie si ce point d'intersection se trouve à l'intérieur des deux segments. Une condition nécessaire et suffisante pour qu'il y ait intersection est donc :

$$\begin{cases} \min(x_A, x_B) \leq x_E \leq \max(x_A, x_B) \\ \min(y_C, y_D) \leq y_E \leq \max(y_C, y_D) \end{cases}$$

2.3.2 La fonction `position`

C'est une fonction membre de la classe **obstacle**. Elle prend en argument un `const segment & seg` (un segment $[AB]$) et retourne un `vector<int>` contenant

deux indices. Ces indices dépendent de la position des points A et B par rapport à l'obstacle courant `*this`. En résumé, si l'indice est i, un entier positif ou nul, alors le point en considération coincide avec le sommet i de l'obstacle. Si l'indice est égal à -1, le point est dans l'intérieur strict de l'obstacle. S'il est égal à -2, le point est extérieur. L'exemple qui suit donne une bonne illustration :



`this->position`$([FE]) = [-2, 5]$
`this->position`$([GH]) = [-1, -2]$

Ces deux fonctions nous permettant finalement d'implémenter la fonction désirée.

2.3.3 La fonction `intersect`

C'est une fonction membre de la classe **obstacle**. Elle prend en argument un `const segment & seg` et retourne un booléen selon que l'intersection est trouvée ou non.

- Si un des deux points est intérieur, on retourne `true`.
- Si les deux points A et B sont des sommets de l'obstacle, on vérifie si le segment $[AB]$ est un segment de l'obstacle ou pas.
- Si les deux extrémités sont à l'extérieur ou une extrémité est à l'extérieur et l'autre est un sommet de l'obstacle, la fonction vérifie si le segment intersecte avec n'importe quel segment de l'obstacle.
- Sinon, la fonction retourne `false`.

2.4 Construction du graphe des chemins

2.4.1 La classe arc

La classe **arc** est une classe utilisée dans l'implémentation de l'algorithme de Dijkstra pour la recherche du chemin le plus court. Elle représente un arc reliant deux sommets `deb` et `fin` et contient un membre `longueur` représentant le coût de l'arc.

2.4.2 La calsse graphe

On s'intéresse maintenant à l'implémentation de la classe **graphe**. En effet, un **graphe** possède deux membres : le membre `noeuds` qui est un vecteur d'objets de la classe **sommet** (`vector<sommet>`) et le membre `arcs` qui est un vecteur d'objets de la classe **arc** (`vector<arc>`). Le constructeur du graphe (initialement non orienté) prend en argument une liste d'**obsatcles** sous la forme `const vector<obstacle>& obstacles` et deux **sommets** `const sommet& depart` et `const sommet& arrivee`. Le constructeur met à jour les noeuds en ajoutant tous les sommets puis fait parcourir les neouds pour ajouter tous les arcs possbiles entre eux.

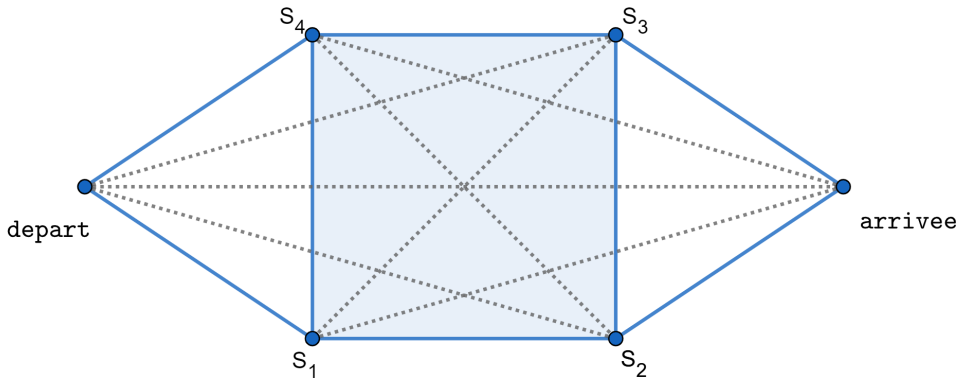


FIGURE 1 – Les arcs du graphe sont en bleu tandis que les traits en pointillés sont des arcs de longueur infinie

3 Objet ponctuel

Après avoir mis en place les classes nécessaires, nous pouvons entamer la première partie : l'objet ponctuel. La traitement de cette partie ne nécessite que l'implémentation d'un algorithme de recherche : nous avons opté pour **l'algorithme de Dijkstra**.

3.1 L'algorithme de Dijkstra

3.1.1 Principe

On dispose d'un graphe G avec n sommets numérotés de 1 à n. Supposons qu'on s'intéresse aux chemins partant du sommet 1. Pour cela, on construit un vecteur ℓ de longueur n, tel que $\ell(j)$ soit égal à la longueur du plus court chemin allant de 1 au sommet j. Au départ, on initialise ce vecteur comme suit :

— S'il existe un arc reliant 1 à j, $\ell(j) = d(1, j)$.

— Sinon, $\ell(j) = +\infty$.

dans laquelle $d(i, j)$ est le poids (la longueur) de l'arc allant de i à j.

On construit également un autre vecteur p pour stocker le chemin reliant le sommet 1 au sommet voulu. La valeur $p(i)$ donne le sommet qui précède i dans le chemin.

On considère ensuite deux ensembles de sommets :

— S, l'ensemble des sommets visités, initialisé à $\{1\}$;

— T, l'ensemble des sommets non visités, initialisé à $\{2, 3, ..., n\}$

Le principe de l'agorithme consiste donc à chaque itération en :

(i) Trouver l'indice réalisant le minimum du vecteur ℓ et d'identifier le sommet correspondant à cette valeur qu'on notera i.

(ii) On ajoute ce sommet à S et on le retire de l'ensemble T.

(iii) On parcourt les successeurs du sommet i, c.-à-d. pour chaque successeur j de i non encore visité, on compare $\ell(j)$ et $\ell(i) + d(i, j)$. Deux cas se présentent :

— Si $\ell(j) > \ell(i) + d(i, j)$, le chemin le plus court trouvé jusqu'à présent allant de 1 à j est $1 \to \ldots \to i \to j$. Dans ce cas, on met à jour les vecteurs ℓ et p : $\ell(j) = \ell(i) + d(i, j)$ et $p(j) = i$.

— Si $\ell(j) \leq \ell(i) + d(i, j)$, le meilleur chemin trouvé est celui duquel on dispose et aucune modification sur ℓ et p n'est effectuée.

On itère ce processus jusqu'à ce que $T = \emptyset$, c'est-à-dire jusqu'à ce que tous les sommets soient visités. À chaque étape, le vecteur ℓ donne le coût minimal des chemins de 1 aux sommets de S. Après la fin de l'algorithme, on obtient un chemin de 1 à k de la façon suivante :

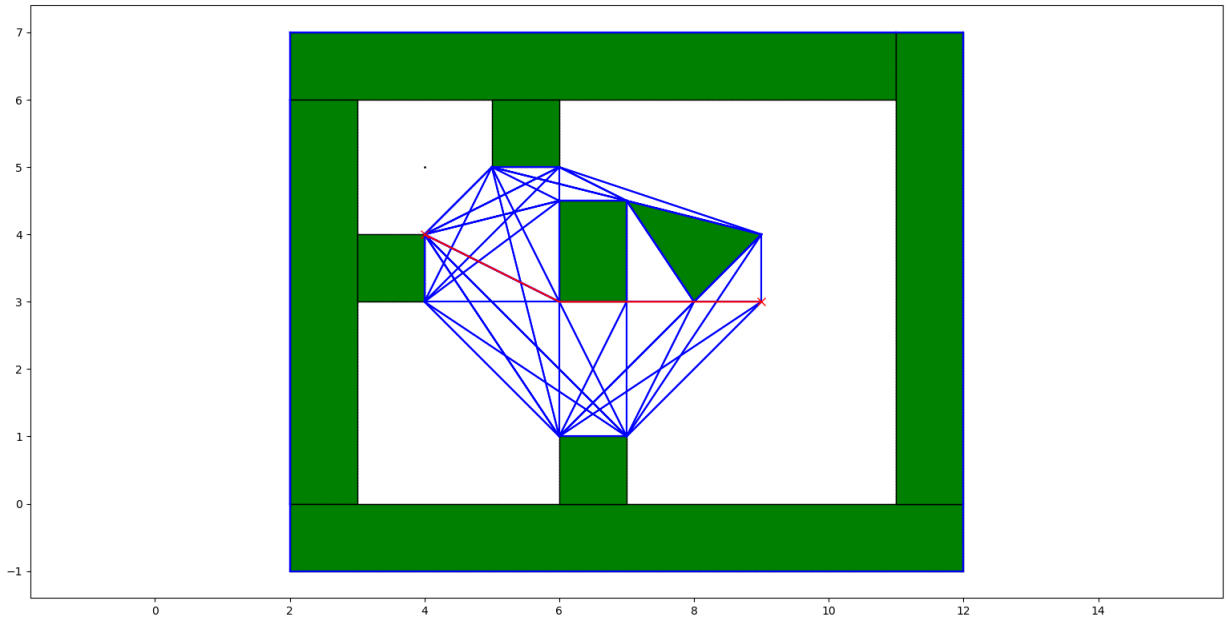
```
i=k
chemin = [k]
Tant que i > 1 faire :
    i=p(i)
    ajouter i au début de chemin
```

3.1.2 Implémentation

Pour l'implémentation de l'algorithme, on a créé une fonction `Dijkstra` qui prend un argument un graphe `const graphe &G`, un noeud de départ `const sommet &start` et un noeud d'arrivée `const sommet &end`. Elle exécute l'algorithme et retourne un vecteur de sommet `vector<sommet>` qui contient les sommets à suivre dans l'ordre pour aller de `start` à `end`.

3.1.3 Validation du code

Pour valider le travail de la première partie, nous avons eu recours au graphe de la figure ci-dessous. Le programme s'éxecute en un temps quasi-nul et affiche le résultat voulu.



Les obstacles sont colorés en vert et le graphe des chemins possibles est en bleu. Dans cet exemple, on désire aller du point $(4, 4)$ vers le point $(9, 3)$. Le chemin suivi par le point est de couleur rouge : c'est bien le chemin de coût minimal.

4 Objet non ponctuel

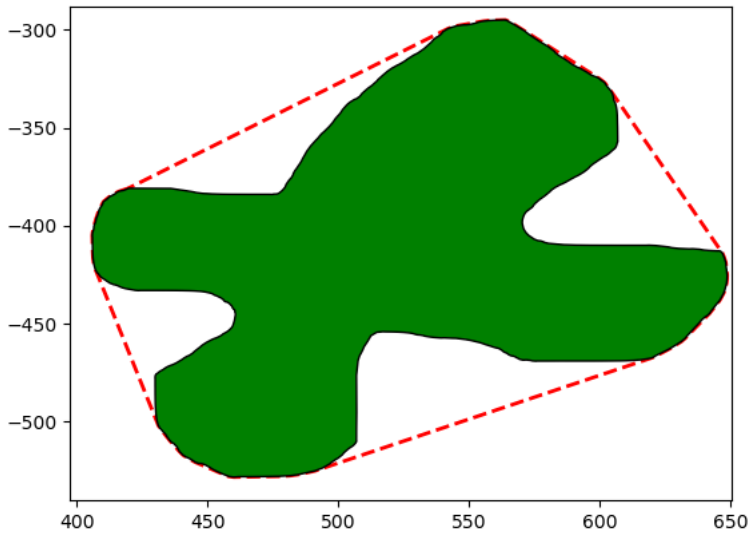
Nous pouvons généraliser la méthode précédente pour le cas où l'objet n'est pas ponctuel mais plutôt un objet de forme quelconque. Pour ce faire, on peut utiliser la technique de « padding »

4.1 Le « padding »

Cela implique de modifier les obstacles en ajoutant une couche autour de l'objet en mouvement, qui génère un nouvel obstacle épaissi. En d'autres termes, on considère l'ancien obstacle comme étant épaissi de la couche générée par le glissement de l'objet autour de l'obstacle.

Pour cela, on a implémenté la fonction `padding` qui prend en argument l'objet qui se déplace, représenté par un ensemble de sommets (`const vector<sommet>& objet`) et l'obstacle qu'on souhaite modifier `const obstacle& obs`. Elle renvoie l'obstacle modifié avec la technique de « padding ».

L'obstacle et/ou l'objet peuvent être de forme quelconque. Grâce à la fonction `to_convexe`, on enlève les concavités de l'objet et/ou l'obstacle comme le montre la figure ci-dessous et puis on réalise le « padding » sur le nouvel obstacle/objet. Comme le chemin optimal ne passe jamais par une concavité (à moins que le départ ou l'arrivée soient situés dans une concavité), le résultat final est le même.



L'idée qu'on a développée pour réaliser le « padding » consiste à parcourir les segments de l'obstacle. On considère un segment i et le segment suivant $i+1$. Pour chaque segment, on trouve le sommet de l'objet qui serait en contact avec le segment lorsque l'objet touche l'obstacle de ce côté et ceci est à l'aide de la fonction `trouverIdxPtProche`. Puis, on translate ces deux segment en suivant leurs vecteurs normaux vers ces deux points. Ensuite, on ajoute à l'obstacle entre ces deux segments les symétriques des segments de l'objet situés entre les deux points trouvés i et $i+1$.

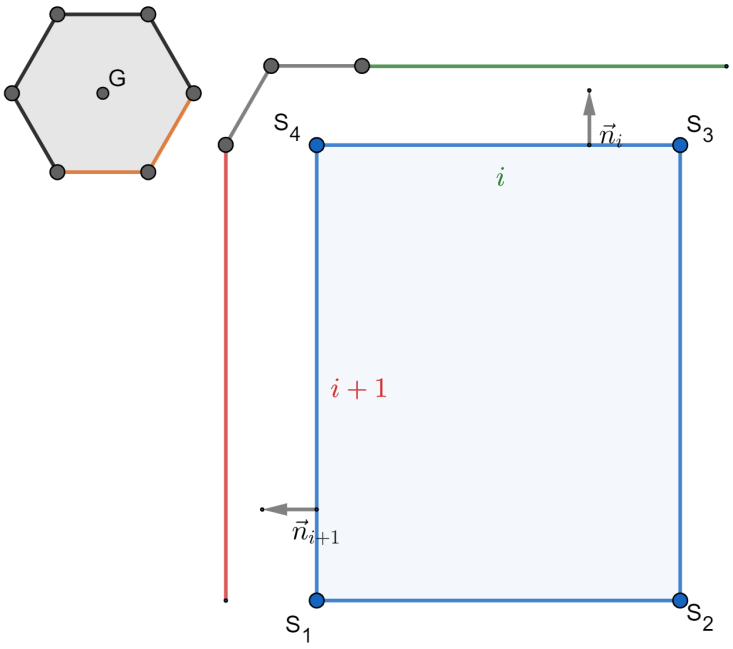


FIGURE 2 – Principe de la méthode du « padding »

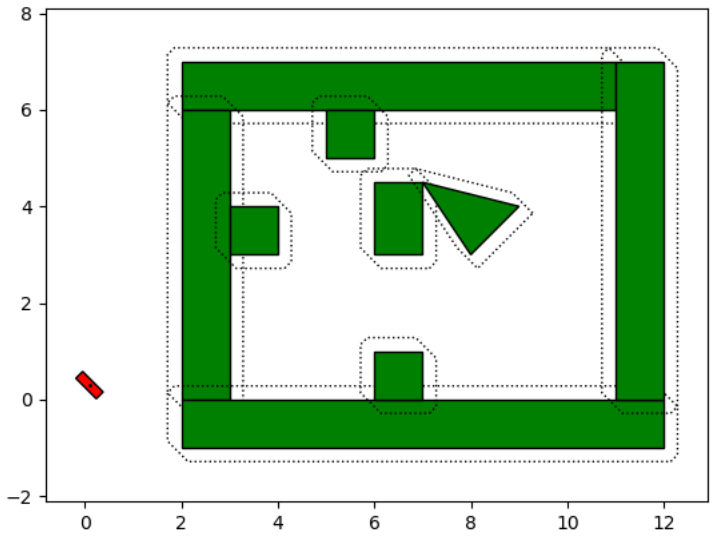
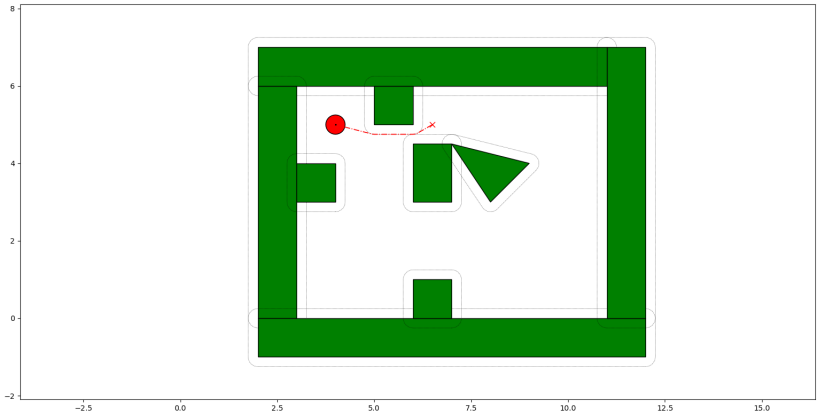


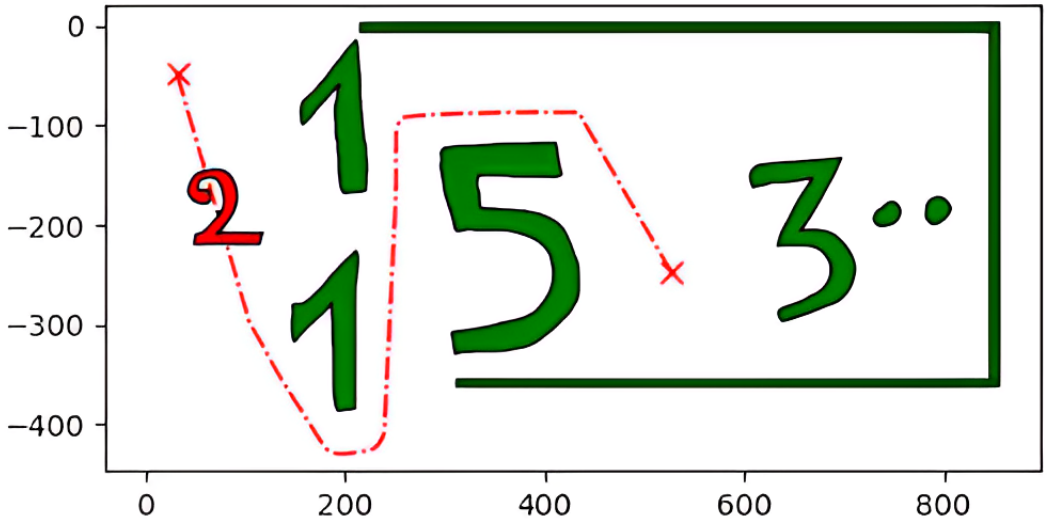
FIGURE 3 – « Padding » réalisé sur les obstacles ici colorés en vert

4.2 Validation du code



Les obstacles originaux sont colorés en vert, l'objet en rouge. Les nouvelles frontières des obsatcles générées par le « padding » sont en pointillés. On obtient le résultat attendu.

On donne aussi un deuxième exemple où l'objet et les obstacles sont sous forme de chiffres déssinés à la main.



4.3 Extension : objet tournant

La méthode peut encore être étendue pour un objet pouvant tourner sur lui même. Pour résoudre ce problème, une approche consiste à discrétiser les directions suivant un nombre donné d'angles pour se ramener au cas d'un polygone. La trajectoire est décrite par la position et l'orientation de l'objet. Dans ce cas, on travaille dans un espace $\mathbb{R}^2 \times [0, 2\pi[$ appelé espace (point, direction), où les obstacles deviennent des domaines de $\mathbb{R}^3$, chaque « *layer* » du domaine correspondant à une direction donnée de l'objet.

Pour ce faire, on introduit les éléments suivants :

— `n_rot` : un entier représentant le nombre de directions que peut prendre l'objet. `n_rot`= 8 correspondra à la discrétisation de l'intervalle $[0, 2\pi[$ à l'ensemble $\{2k\pi/8, \quad k \in [|0, 7|]\}$.

— `idx` : un membre de la classe **sommet** représentant l'indice du « *layer* » dans lequel se trouve le sommet ($0 \leq$ `idx`$<$ `n_rot`).

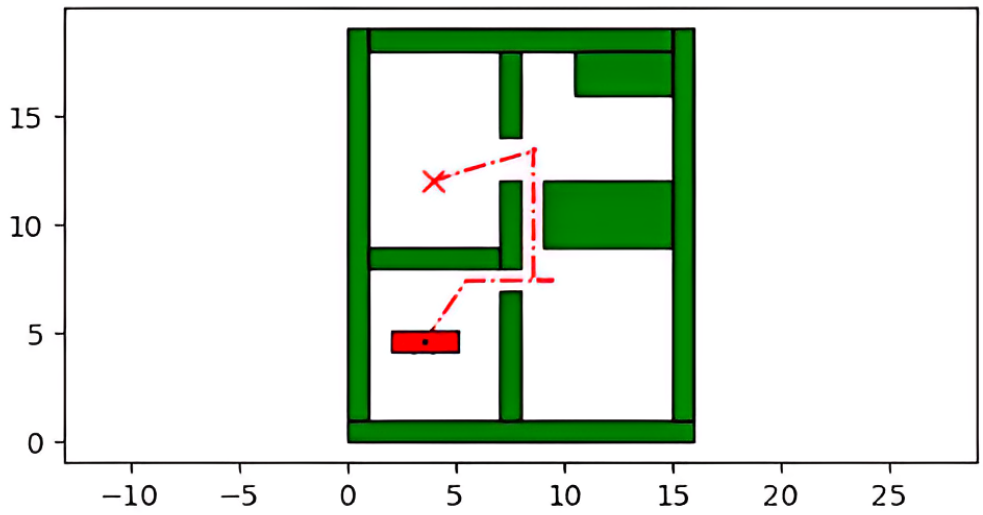
— `sens` : membre de la classe **sommet** représentant le sens de rotation dans lequel tournera l'objet en sortant du sommet (la valeur `1` correspondra à un sens direct et `-1` à un sens indirect).

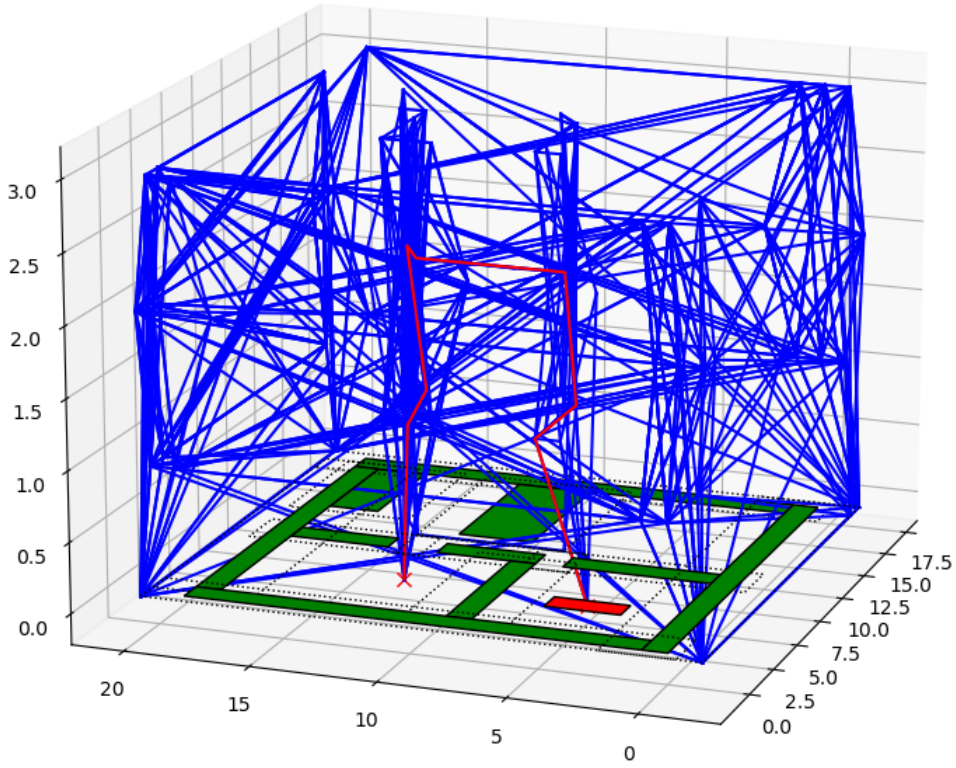
— `cout_tour` : le coût en termes de distance d'un pas de rotation.

La fonction `padding3D` réalise maintenant le « padding » sur chaque obstacle mais sur chaque « layer » afin de prendre en compte la capabilité de l'objet à tourner sur lui-même. On note aussi que le graphe est déosrmais *orienté* dû au fait que la rotation n'est pas toujours possible dans chaque noeud. Il suffit juste de vérifier si l'objet peut tourner au point de départ pour qu'il n'y ait pas blocage.

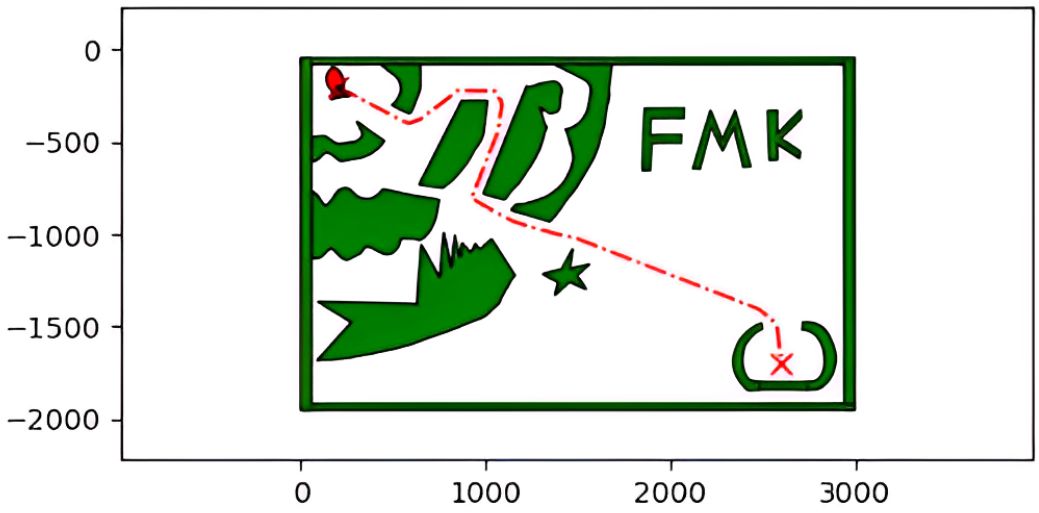
4.3.1 Validation du code

L'objet est de forme rectangulaire. On voit le chemin suivi par l'objet en rouge et c'est bien le résultat voulu. Une illustration du graphe des chemins en trois dimensions est donnée par la figure qui suit.





On finit par un exemple où l'objet est sous forme d'un poisson et cherche à dépasser les obstacles pour arriver dans la concavité.



5 Organisation du travail et difficultés rencontrées

L'organisation du travail de ce projet a été essentielle pour sa réussite. Nous avons commencé par une phase de planification, en définissant les tâches à accomplir et en les répartissant entre les membres de l'équipe.

Pour faciliter la communication et la collaboration, nous avons utilisé des outils de gestion de projet en ligne tels que GitHub. Ceci nous a permis de suivre l'avancement des tâches, de partager des fichier et de travailler ensemble en temps réel.

Les réunions régulières en classe étaient d'un aide très précieux pour discuter de l'avancement du projet et résoudre les problèmes éventuels. Nous avons également utilisé une messagerie instantanée pour rester en contact régulier et partager des informations importantes entre les réunions.

Toutefois, la réalisation de ce projet a été parsemée de nombreux défis, que ce soit des problèmes théoriques, de codage, de planification, de communication ou encore de documentation. Dans un premier temps, des problèmes théoriques ont surgi, liés à la compréhension et à l'application de concepts abstraits tels que l'implémentation de la méthode Dijkstra, la conception de la fonction `intersect`, le « padding » ou encore le traitement du problème de l'objet tournant. La non convexité de l'objet/des obstacles nous a aussi posé un problème qu'on a éventuellement résolu. De plus, nous avons rencontré des difficultés de codage lors de l'assemblage des différentes parties du code, l'adaptation des variables et la validation, ainsi que la création d'exemples d'implémentation adéquats. Des difficultés de planification se sont également présentées, notamment pendant les périodes de vacances, pour organiser des créneaux de travail de groupe afin d'avancer plus rapidement dans le projet. En outre, certaines parties du code sont très gourmandes en temps de calcul, en particulier lors du traitement de l'objet tournant dans un graphe assez large. Le calcul prenait plusieurs heures. Enfin, on a rencontré des problèmes de documentation. Il est essentiel de bien documenter le code pour permettre aux autres membres de l'équipe de le comprendre et de le modifier facilement. Cela peut prendre du temps, mais c'est un aspect important du travail en équipe sur un projet de programmation.

Dans l'ensemble, l'organisation rigoureuse et le travail coordonné nous a permis de mener à bien le projet dans les délais impartis et de produire un résultat final de qualité. Nous avons également appris l'importance de la planification et de la communication pour la réussite d'un projet d'équipe.

Conclusion

En conclusion, ce projet de planification de trajectoire est un exemple de la manière dont la modélisation mathématique et les algorithmes peuvent être utilisés pour résoudre des problèmes pratiques dans divers domaines. Ce rapport se concentre sur les détails de l'implémentation de la méthode et fournit une analyse approfondie de la performance de l'algorithme de Dijkstra pour résoudre le problème de la planification de trajectoire.

Ce projet a été une expérience riche en apprentissage et en défis pour l'ensemble de l'équipe. Nous avons été confrontés à des problèmes de différents types. Cependant, grâce à notre travail collaboratif et à notre détermination, nous avons réussi à surmonter ces obstacles et à mener à bien le projet dans les plus brefs délais.

Ce projet nous a permis de développer nos compétences en programmation, en travail d'équipe et en résolution de problèmes. Nous avons également appris l'importance de la communication et de la documentation dans le travail en équipe sur un projet de programmation.

Enfin, nous tenons à remercier Mme Alouane pour son soutien et ses conseils tout au long du projet. Nous sommes convaincus que ce projet a été une expérience enrichissante pour tous les membres de l'équipe et nous espérons avoir l'occasion de travailler ensemble sur d'autres projets à venir.