Projet informatique

Planification de trajectoire

Réalisé par :
Aymen BEN AMMAR
Eya JLASSI

Encadré par :
M^me^ Sonia ALOUANE

2^ème^ année Techniques Avancées

Année universitaire : 2023/2024

Remerciements

En premier lieu, nous remercions notre enseignante et encadrante Mme. Sonia Alouane pour toutes les connaissances qu'elle nous a communiquées tout au long de l'élaboration de ce projet, pour sa grande disponibilité, et ses précieux conseils. Nous remercions également Mme Hedia Chaker, responsable de la filière Techniques Avancées, son sérieux remarquable et ses encouragements, qui nous ont été d'une très grande utilité

Nous remercions également, toutes les personnes qui nous ont conseillées et relues lors de la rédaction de ce projet, notamment nos camarades de la 2re année Techniques Avancées.

Enfin, nous adressons une pensée affective à nos parents, nos familles et nos amis pour leurs supports tout au long de ce projet.

Table des matières

Introduction

Dans le domaine de la robotique et de l'automatisation, la planification de trajectoire est un élément crucial pour assurer le déplacement efficace et sûr des robots dans leur environnement. En effet, la capacité à générer des trajectoires optimales tout en évitant les obstacles est essentielle pour la réalisation de tâches complexes telles que la navigation autonome, la manipulation d'objets et bien d'autres.

Dans ce projet, nous nous proposons d'explorer et d'implémenter des solutions de planification de trajectoire en utilisant le langage de programmation C++. Notre objectif est de concevoir un système capable d'identifier des trajectoires optimales pour un robot mobile dans des environnements dynamiques et souvent imprévisibles.

Nous aborderons les principaux concepts et techniques de planification de trajectoire, en nous concentrant sur des approches telles que la recherche d'espaces d'états, les algorithmes de recherche, la génération de chemins et la résolution de problèmes de mouvement.

Chapitre 1

Modélisation fonctionnelle

1.1 Description génerale

1.1.1 Principe du projet

Notre projet vise à définir le chemin optimal, généralement le plus efficace, qu'un objet doit emprunter pour se déplacer d'un point initial à un point final dans un environnement complexe, caractérisé par la présence de nombreux obstacles et des variations de terrain.

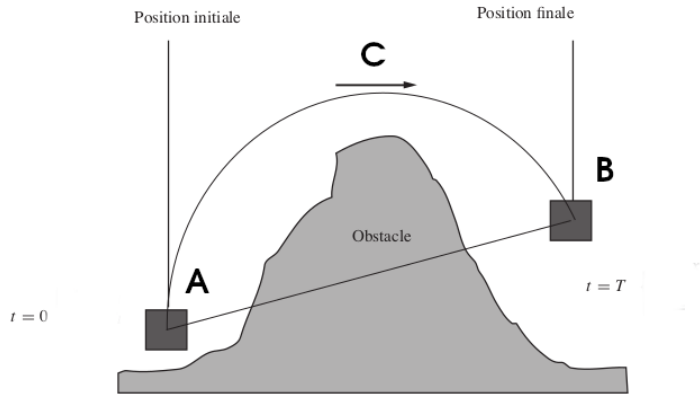
1.1.2 Définition d'un chemin optimal

La première étape de la planification de trajectoire consiste à établir clairement la notion de chemin, de manière à ce qu'elle puisse être ultérieurement identifiée par l'objet, qu'il soit considéré comme un point ponctuel ou un corps. Ensuite, il convient de définir les critères qui permettront de déterminer le chemin le plus efficace, en prenant en compte les caractéristiques spécifiques de l'objet en mouvement.

1. **Les extrémités du segment ne doivent pas être à l'intérieur d'un obstacle :**
   Cela garantit que le segment ne commence pas ou ne termine pas à l'intérieur d'un obstacle.

2. **Non-intersection avec les obstacles :**
   Le théorème de Pythagore énonce que la distance la plus courte entre deux points est une ligne droite les reliant. Cependant, il est impératif de ne pas céder à une simplification excessive de ce principe, car cela pourrait conduire

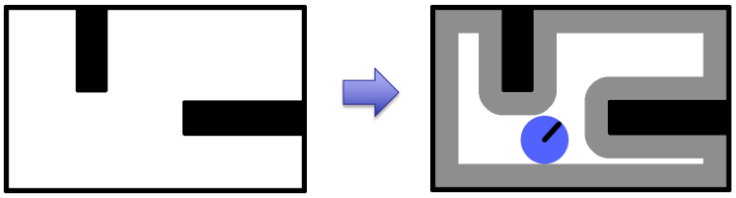
à la création d'un chemin qui traverse réellement l'intérieur d'un obstacle..



3. **Distance de sécurité :**
   Dans le cas ou l'objet n'est pas ponctuel, il faut garantir que l'objet n'intersecte pas l'obstacle, en d'autres termes, il faut s'assurer que le chemin maintient une distance de sécurité avec les obstacle telque :

$$\boxed{\text{Distance de sécurité} \geq \frac{\text{Largeur de l'objet}}{2}}$$

   Dans cette situation, l'espace des trajectoires envisageables serait restreint, comme le montre l'illustration dans la figure ci-dessous.



4. **Validation géométrique :**
   Vérifiez que le segment est spécifié correctement (longueur positive, points de départ et d'arrivée différents, etc.).

5. **Optimalité :** Un chemin ne doit pas seulement satisfaire l'ensemble des conditions précédentes pour être considéré valide, mais il doit également être optimal en termes de longueur. En d'autres termes, le chemin devrait être

le plus court possible tout en respectant les contraintes de non-intersection avec les obstacles, d'évitement des collisions, de distance de sécurité, et de respect des contraintes dynamiques le cas échéant.

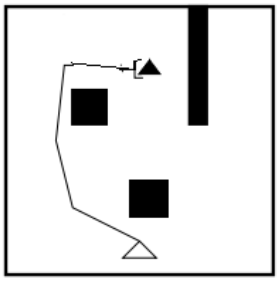


FIGURE 1.1 – Chemin non optimal

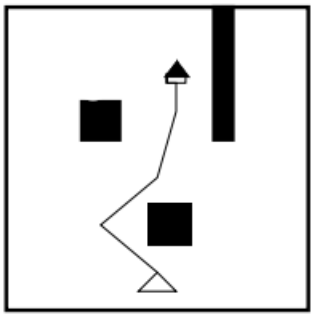


FIGURE 1.2 – Chemin optimal

1.1.3 Cahier de charge

On implantera les structures de données et algorithmes permettant de :

— Lire un document text contenant la localisation exacte des obstacles.

— Déterminer le chemin optimal en se basant sur l'algorithme de Dijkstra.

— Fournir le résultat de la génération sous forme de fichier, avec un encodage qui sera à définir.

— Visualiser le résultat de l'encodage en recourant à Matlab.

L'objectif de performance est d'être en mesure de générer des trajectoires optimales pour des itinéraires complexes en quelques minutes. Une applet MATLAB, permettant la visualisation des trajectoires à partir du fichier de sortie, sera également mise à disposition. A noter qu'outre la partie concernant le développement proprement dit, les élèves devront remettre un rapport comprenant :

— une description détaillée des choix de conception effectués,

— les liens entre la conception et le code effectivement implanté,

— des éléments permettant de se convaincre de la validité du planificateur de trajectoire(conception et implantation),

— des éléments permettant de juger de l'efficacité du planificateur de trajectoire(temps, mémoire).

1.2 Comportement entre l'utilisateur et le programme

L'interaction entre l'utilisateur et le programme de planification de trajectoire est conçue pour être intuitive et efficace. L'utilisateur démarre le programme en spécifiant les paramètres nécessaires tels que la carte de l'environnement, les coordonnées du point de départ et d'arrivée, ainsi que la localisation des obstacles à éviter. Le programme utilise ensuite ces informations pour générer une trajectoire optimale en évitant les obstacles.

L'interface utilisateur peut offrir des fonctionnalités telles que la visualisation de la carte avec les obstacles marqués, la possibilité de modifier dynamiquement la carte pour simuler différents scénarios, et l'affichage de la trajectoire générée avec des informations détaillées sur le chemin suivi.

Une fois la planification terminée, le programme affiche la trajectoire optimale à suivre.

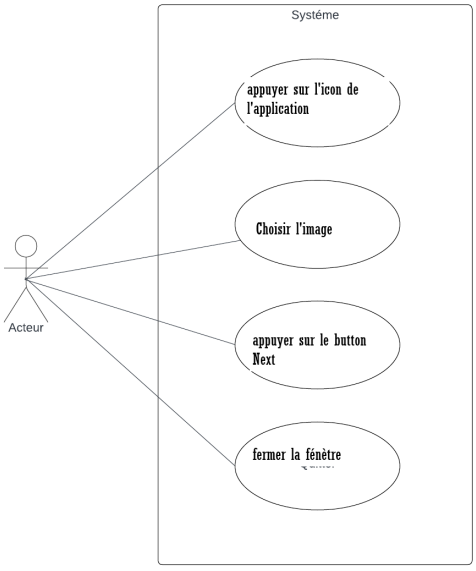


FIGURE 1.3 – Diagramme de cas d'utilisation

1.3 Les données d'entrée et de sortie

1.3.1 Les données d'entrée

En langage C, pour résoudre un problème de plannification de trajectoire, il est crucial de représenter les informations d'entrée de manière appropriée à l'aide de structures de données adaptées, comme des tableaux, des listes chaînées, des arbres ou des graphes. Les données d'entrée incluent généralement des informations sur l'environnement, telles que les obstacles, les points de départ et d'arrivée, ainsi que d'autres contraintes e sont fournies dans un fichier.txt.

Dans le contexte de la plannification de trajectoire, les structures de données peuvent être utilisées pour stocker des informations sur les cellules de la grille représentant l'environnement. Ces structures permettent de modéliser efficacement les relations spatiales entre les cellules, facilitant ainsi la recherche de chemins optimaux ou la génération de trajectoires.

1.3.2 Les données de sortie

En résolvant le problème de plannification de trajectoire en langage C, les données de sortie prendront la forme d'une visualisation .

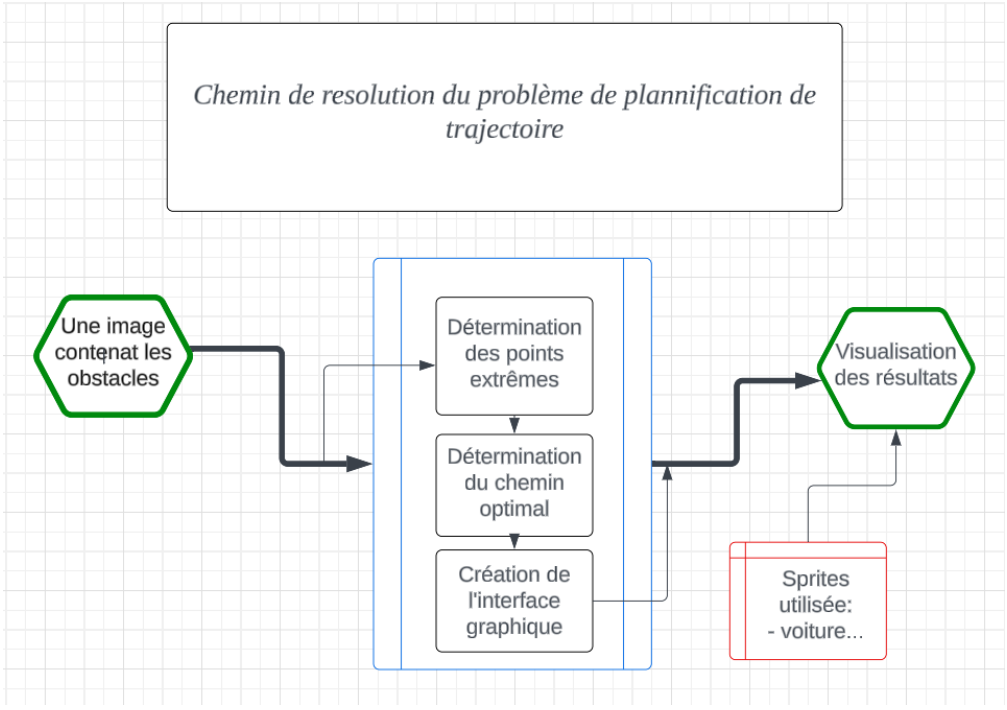


FIGURE 1.4 – Les données d'entrée et de sortie

1.4 La nature de traitement des données

1.4.1 Les étapes suivies

Le développement d'un projet de planification de trajectoire implique plusieurs étapes, depuis la visualisation des obstacles sur Matlab jusqu'à la visualisation des résultats. Voici une séquence d'étapes générales que nous pourriez suivre

- **Définition des Classes et Structures de Données :**
  - Créer les classes nécessaires pour représenter les éléments tels que les sommets, segments, obstacles, arcs, et le graphe des chemins.
- **Implémentation des Algorithmes de Base :**
  - Mettre en œuvre les fonctionnalités de base telles que la détection d'intersection entre un segment et un obstacle, la construction du graphe des chemins, etc.
- **Visualisation des Obstacles sur C++ :**
- **Determinations des sommets des Obstacles :**
  - Écrire un script C++ qui prend l'emplacement de fichier et retoune les sommets des obstacles.

— **Implémentation de l'Algorithme de Dijkstra :**
  — Mettre en œuvre l'algorithme de Dijkstra pour trouver le plus court chemin dans le graphe des chemins.
— **Visualisation des Trajectoires Calculées sur C++ :**
  — Afficher les trajectoires calculées sur un même graphique avec les obstacles.
— **Optimisation et Tests Complexes :**
  — Optimiser le code et effectuer des tests avec des configurations plus complexes, comprenant plusieurs obstacles de formes différentes.
— **Gestion des Objets Non Ponctuels (Cas des Disques) :**
  — Étendre le programme pour prendre en compte des objets non ponctuels, par exemple en utilisant la méthode du "padding" pour les disques.

1.4.2 Les algorithmes

Algorithme de Dijkstra

Le problème du plus court chemin demeure l'un des défis prépondérants en recherche opérationnelle, avec des implications cruciales dans divers secteurs tels que la logistique, les réseaux de transport, et la planification de trajectoire. Dans ce vaste éventail de méthodes, comprenant des approches renommées comme l'algorithme de Bellman-Ford et l'algorithme de Bellman, l'algorithme de Dijkstra émerge comme une solution particulièrement efficace.
cet algorithme offre une réponse précise et rapide au défi de trouver le chemin le plus court entre un point de départ et tous les autres points dans un graphe pondéré. Son approche itérative et sa capacité à fournir des résultats optimaux font de l'algorithme de Dijkstra un choix privilégié, notamment dans le domaine de la planification de trajectoire. En effet, grâce à sa performance remarquable, cet algorithme s'avère être un outil incontournable pour déterminer efficacement les chemins optimaux dans des environnements complexes caractérisés par de nombreux obstacles et des terrains variés.

Principe de l'algorithme de Dijkstra

Le principe de l'algorithme de Dijkstra repose sur la recherche itérative du chemin le plus court entre un nœud source et tous les autres nœuds d'un graphe pondéré dirigé. Voici les étapes fondamentales de l'algorithme :

1. **Initialisation :**
   L'algorithme commence par initialiser un tableau des distances à partir du

nœud source vers tous les autres nœuds. La distance initiale depuis le nœud source est mise à zéro, et les distances initiales vers tous les autres nœuds sont fixées à une valeur infinie. On utilise également une structure de données, souvent une file de priorité, pour maintenir les nœuds non encore traités.

2. **Sélection du nœud actuel :**
   À chaque itération, l'algorithme sélectionne le nœud non traité avec la plus petite distance actuelle depuis le nœud source. Au départ, cela est le nœud source lui-même.
3. **Mise à jour des distances :**
   Pour chaque nœud voisin du nœud actuel qui est encore dans l'ensemble des nœuds non traités, l'algorithme calcule la distance temporaire depuis le nœud source en passant par le nœud actuel. Si cette distance temporaire est plus courte que la distance enregistrée actuellement pour ce nœud voisin, la distance est mise à jour.
4. **Marquage du nœud actuel :**
   Le nœud actuel est marqué comme traité en le retirant de l'ensemble des nœuds non traités.
5. **Répétition :**
   Les étapes 2 à 4 sont répétées jusqu'à ce que tous les nœuds aient été traités ou que la distance vers un nœud cible spécifié ait été trouvée.
6. **Résultat :**
   Une fois terminé, le tableau des distances contient les distances les plus courtes depuis le nœud source vers tous les autres nœuds du graphe.

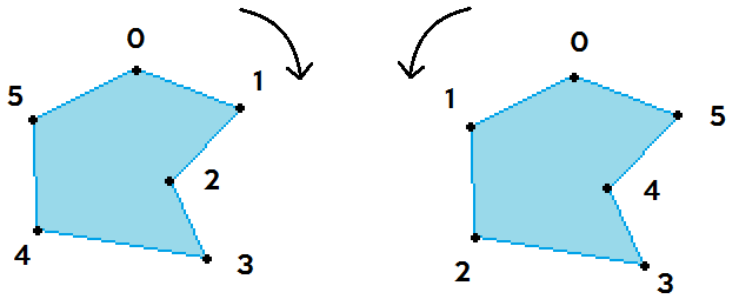
L'algorithme de Dijkstra garantit que la distance enregistrée pour chaque nœud après traitement est la plus courte distance possible depuis le nœud source. Cette propriété permet de reconstruire le chemin optimal vers n'importe quel nœud à partir du nœud source.

Chapitre 2

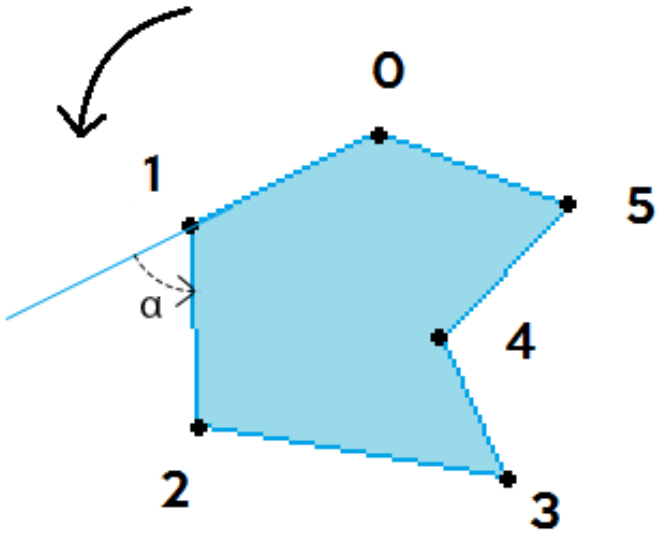
Modelisation operationnelle

2.1 Architecture

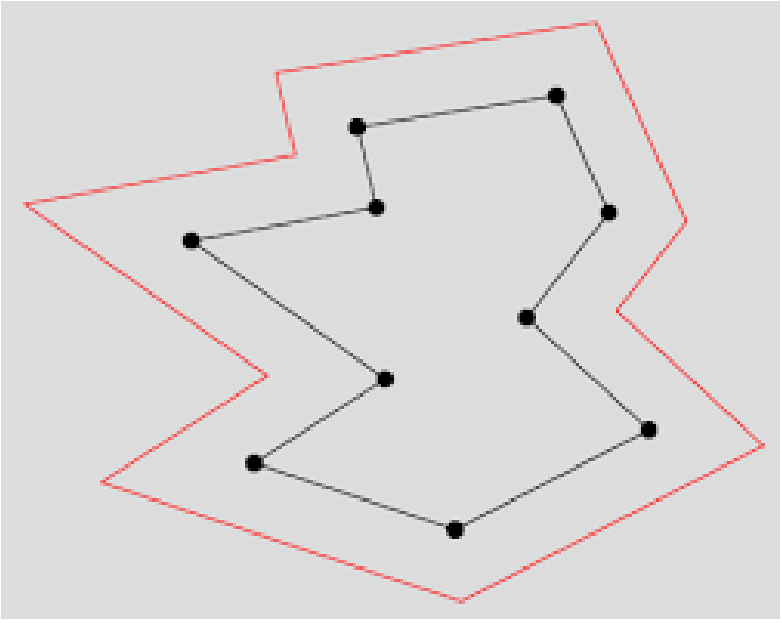
1. Définir les points de départ et d'arrivée à partir de leurs coordonnées 2D :
2. Déterminer à partir de l'image fournie les coordonnées des points extrêmes des obstacles
3. Vérifier si les obstacles sont dans le sens positive. Sinon, inverser le sens des points extrêmes qui le définissent.



Pour savoir le sens de l'obstacle, on s'appuie sur une propriété mathématique qui annonce que :
- Si une forme est dans le sens positive, la somme des angles αs est égale à $2 \times \pi$, sinon elle est égale à $-2 \times \pi$



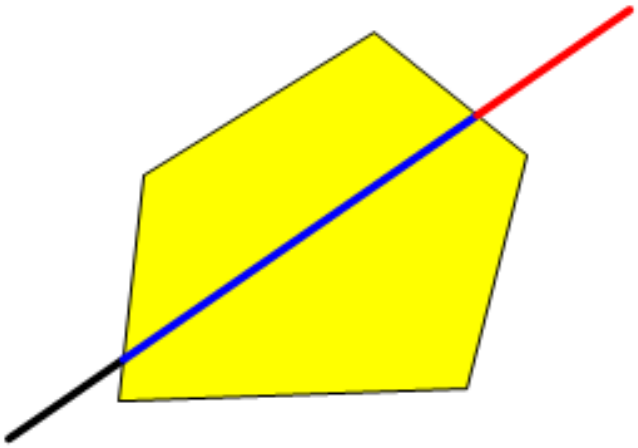
4. Ajouter du padding aux obstacles :
    - Définir la valeur du padding
    - Utiliser le fait que les obstacles sont dans le sens positive pour décaler ces points extrêmes d'une façon à maintenir la même distance entre les arcs de l'obstacle et les nouvelle arcs de l'obstacle virtuelle.



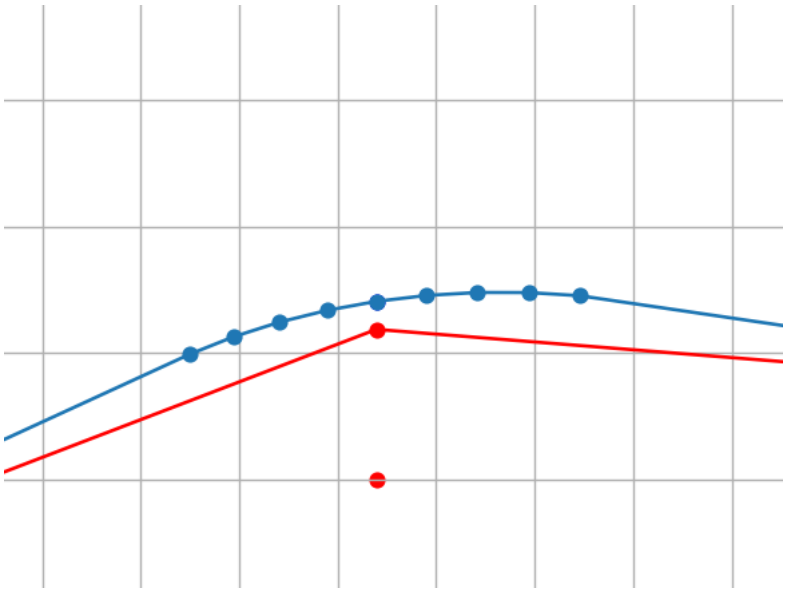
5. Définir le graphe à partir de l'ensemble de tous les points, comprenant :

- la point de départ
- la point d'arrivée
- les points extrêmes des obstacles

6. Eliminer les segments qui interceptent avec un obstacles en se basant sur le fait que les polynomes sont connexes.
   Ces segments sont :
   - soit reliant des points non voisins,
   - soit intercepte deux arcs d'un obstacle par leurs axes, non par les points d'extrémités.



7. Après avoir obtenu le graphe avec tous les arcs qui sont admissibles, on ajoute les angles.
   Pour cela, pour chaque point i :
   (a) On détermine tous les points avec lesquels il est relié.
   (b) Pour chaque point relié j, on crée un point ayant les coordonnées du point i et l'angle du vecteur défini par $P_j - P_i$
8. On relie tous ces nouveaux points pour obtenir les arcs finaux pour notre algorithme
9. Appliquer l'algorithme de Dijkstra en tenant compte de la différence des angles entre les points dans la pondération des arcs.
10. Ajouter une rotation dynamique au chemin obtenu en transformant chaque point en des points définissant un arc de cercle



2.2 Conception

2.2.1 Structure de données

1. **Sommet (*Vertex*) :**
   - *Description :*
     Représente un sommet avec des coordonnées (x, y) dans un plan.
   - *Fonctionnalités :*
     — Constructeur : Initialise un sommet avec des coordonnées spécifiques (par défaut, à l'origine).
     — Surcharge de l'opérateur d'assignation : Copie les coordonnées d'un autre sommet.
     — Opérateur d'égalité : Compare si deux sommets sont égaux avec une tolérance.

```cpp
class Sommet {
public:
    double x, y;
    double angle;

    Sommet(double x_ = 0., double y_= 0., double a_ = 0.) : x(x_), y(y_), angle(a_) {}

    Sommet& operator=(const Sommet& other);

    bool operator==(const Sommet& other) const;

    friend std::ostream& operator<<(std::ostream& out, const Sommet& s);
};
```

2. **Segment (*Segment*) :**

- *Description :*
  Représente un segment de ligne entre deux sommets.
- *Fonctionnalités :*
  - Constructeur : Initialise un segment avec deux sommets spécifiques.

```
class Segment {
public:
    Sommet start_Sommet, end_Sommet;

    Segment(const Sommet& start, const Sommet& end);
};
```

3. Obstacle (*Obstacle*) :

- *Description :*
  Représente un obstacle défini par une liste de sommets.
- *Fonctionnalités :*
  - Constructeur : Initialise un obstacle avec une liste de sommets et une tolérance.
  - Méthode d'intersection avec un segment : Vérifie si l'obstacle intercepte un segment.

```
class Obstacle {
public:
    std::vector<Sommet> vertices;

    Obstacle(const std::vector<Sommet>& vertices_, const double& eps, const bool& padding = true);

    bool intersectionWithSegment(const Segment& segment, const double& eps);

private:
    bool SegmentIntersect(const Sommet& s, const Segment& seg, const double& eps);

    std::vector<Sommet> with_Padding(const std::vector<Sommet>& vertices_, const double& eps);

    bool Sens_Positive(const std::vector<Sommet>& vertices_, const double& eps);
};
```

4. Arc(*Arcs*) :

- *Description :*
  Représente un arc entre deux sommets.
- *Fonctionnalités :*
  - Constructeur : Initialise un arc avec deux sommets spécifiques.

```
class Arc : public Segment {
public:
    double length;

    Arc(const Sommet& start, const Sommet& end);
};
```

5. Graph (*Graph*) :

- *Description :*
  Représente un graphe constitué d'arcs reliant des sommets.

- *Fonctionnalités :*
  - Constructeur : Initialise un graphe avec une liste d'arcs (par défaut, vide).
  - Méthode pour ajouter un arc au graphe.

```
class Graph {
public:
    std::vector<Arcs> arcs;

    Graph(const std::vector<Arcs>& first_arc = {});

    void AddArc(const Arcs& arc);
};
```

6. **arc_planification (*arc_planning*) :**
   - *Description :*
     Représente un graphe utilisé dans un contexte de planification de trajectoire.
   - *Fonctionnalités :*
     - Constructeur : Initialise un graphe avec une liste d'arcs, des sommets de départ et d'arrivée, une liste d'obstacles et une tolérance.
     - Méthodes pour trouver un chemin minimal et ajouter des angles aux sommets du graphe.

```
class arc_planification : public Graph {
public:
    Sommet start_Sommet, end_Sommet;
    std::vector<Obstacle> obstacles;
    double epsilon;

    arc_planification(const Graph& graph, const Sommet& start, const Sommet& end, const std::vector<Obstacle>& obs,
                      arc_planification::arc_planification(const Graph& graph, const Sommet& start, const Sommet& end,
                        const std::vector<Obstacle>& obs, const double& eps)
    void AddAngles();

    std::vector<Sommet> trouverCheminMinimal(const bool& dynamic = true);

private:
    std::vector<Sommet> listnodes();

    bool isInX(std::pair<int, int>& elem, std::vector<int>& X);

    std::vector<Sommet> result_dynamic(std::vector<Sommet> result, const double& eps);
};
```

2.2.2 Implémentation

On implémente cette logique dans une fonction *void* qu'on appelle "Trajectoire", elle prend en tant qu'arguments le point de départ, le point d'arrivée et le chemin vers l'image contenant les obstacles

```
std::vector<sf::Vector2f> Trajectory(const Sommet A, const Sommet B, const string& filename) {

    vector<Sommet> all_Sommets;
    all_Sommets.push_back(A); all_Sommets.push_back(B);
```

En utilisant la librairie OpenCV, on détermine les points extrêmes des obstacles. L'agorithme cherche à trouver les régions ayant les couleurs différentes à une couleur

initialement choisit (la couleur de la route, approxime ces formes à des polygones et retourne les coordonnées des points extrêmes de ces polygones.

```cpp
obstacles.cpp > ...
void detect_shapes(Mat& image, int image_width) {
    Mat gray;
    cvtColor(image, gray, COLOR_BGR2GRAY);

    // Appliquer un seuillage inverse pour binariser l'image
    Mat thresh;
    threshold(gray, thresh, 240, 255, THRESH_BINARY_INV);

    // Détecter les contours dans l'image
    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;
    findContours(thresh, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);

    // Initialiser le compteur de formes
    int shape_count = 0;

    // Initialiser une liste pour stocker les informations sur les formes
    vector<pair<int, vector<Point>>> shape_info_list;

    // Parcourir les contours détectés
    for (const auto& contour : contours) {
        // Approximer la forme du contour avec un polygone
        double epsilon = 0.04 * arcLength(contour, true);
        vector<Point> approx;
        approxPolyDP(contour, approx, epsilon, true);

        // Si la forme a au moins 3 côtés et moins de 10 côtés (pour éviter les formes complexes)
        if (approx.size() >= 3 && approx.size() < 10) {
            // Incrémenter le compteur de formes
            shape_count++;

            // Extraire et stocker les coordonnées des coins de la forme
            shape_info_list.push_back(make_pair(shape_count, approx));

            // Dessiner le contour de la forme
            drawContours(image, vector<vector<Point>>{approx}, -1, Scalar(0, 255, 0), 2);
```

Puis, on définit le graphe avec tous les arcs créés par les points des obstacles, de départ et d'arrivée.

Dans la définition de chaque obstacle, on vérifie que la somme des angles α est positive, cela est suffisant dû au fait qu'on sait que cette somme doit être soit $2 \times \pi$ soit $-2 \times \pi$. Sinon, on inverse le vecteur des points extrêmes.

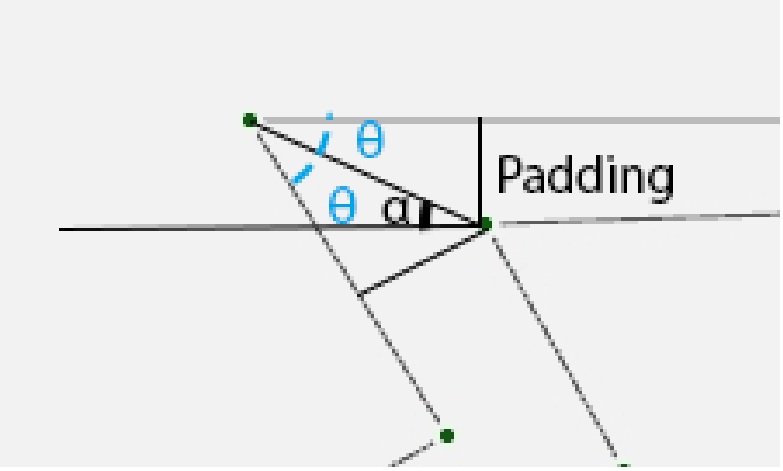
```cpp
class Obstacle {
public:
    std::vector<Sommet> vertices;


    Obstacle(const std::vector<Sommet>& vertices_, const double& eps, const bool& padding= true) {
        if(Sens_Positive(vertices_, eps)) {
            if(padding) vertices = with_Padding(vertices_, eps);
            else vertices = vertices_;
        } else {
            vector<Sommet> inv_vertices = vertices_;
            reverse(inv_vertices.begin(),inv_vertices.end());
            if(padding) vertices = with_Padding(inv_vertices, eps);
            else vertices = inv_vertices;
        }
    }
```

Le calcule de cette somme est implémenté dans la fonction "sens_positive" qui retourne une booléen :

```cpp
bool Sens_Positive(const std::vector<Sommet>& vertices_, const double& eps) {
    double somme_Ang=0;
    long long unsigned int n = vertices_.size();
    for(long long unsigned int b=0;b<n;b++) {
        long long unsigned int a = (b==0)? n-1:b-1;
        long long unsigned int c = (b==n-1)? 0:b+1;
        Sommet s1 = vertices_[a], s2 = vertices_[b], s3 = vertices_[c];

        //double eps = max(max(fabs(s1.x),fabs(s1.y)),max(fabs(s2.x),fabs(s2.y))) / pow(10,6);

        double ang1,ang2;
        if(fabs(s2.x - s1.x) < eps) ang1=(s2.y - s1.y)?M_PI_2:(-1)*M_PI_2;
        else {
            ang1 = atan((s2.y - s1.y)/(s2.x - s1.x));
            if((s2.x - s1.x)<0) ang1 += M_PI;
        }
        if(fabs(s3.x - s2.x) < eps) ang2=(s3.y - s2.y)?M_PI_2:(-1)*M_PI_2;
        else {
            ang2 = atan((s3.y - s2.y)/(s3.x - s2.x));
            if((s3.x - s2.x)<0) ang2 += M_PI;
        }

        double angdiff = ang2-ang1;
        while(angdiff<-M_PI) angdiff+=2*M_PI;
        while(angdiff>M_PI) angdiff-=2*M_PI;
        somme_Ang+=angdiff;
    }
    //cout<<somme_Ang<<'\n';
    return (somme_Ang>0);
}
```

La définition de l'obstacle virtuel avec le padding se fait avec la logique suivant :

1. On calcule les angles α et θ
2. on définit le vecteur de norme $||V|| = \frac{padding}{sin(\theta)}$ et d'angle α
3. on définit le nouveau point par la somme du point ancien avec ce vecteur



Voilà son implémentation :

```cpp
vector<Sommet> with_Padding(const std::vector<Sommet>& vertices_, const double& eps) {
    vector<Sommet> Curr_vertices_ = vertices_, vertices;
    for(int b=0; b<Curr_vertices_.size();b++) {
        int c = (b==Curr_vertices_.size()-1)? 0:b+1;
        int a = (b==0)? Curr_vertices_.size()-1:b-1;
        Sommet s1 = Curr_vertices_[a], s2 = Curr_vertices_[b], s3 = Curr_vertices_[c];

        //double eps = max(max(fabs(s1.x),fabs(s1.y)),max(fabs(s2.x),fabs(s2.y))) / pow(10,6);
        double pad = eps * 2 * pow(10,4);
        double ang1, ang2;
        if(fabs(s2.x - s1.x) < eps) ang1 = (s2.y >= s1.y)? M_PI_2: (-1)*M_PI_2;
        else {
            ang1 = atan((s2.y - s1.y)/(s2.x - s1.x));
            if((s2.x - s1.x)<0) ang1 += M_PI;
        }
        if(fabs(s3.x - s2.x) < eps) ang2 = (s2.y >= s3.y)? M_PI_2:(-1)*M_PI_2;
        else {
            ang2 = atan((s2.y - s3.y)/(s2.x - s3.x));
            if((s2.x - s3.x)<0) ang2 += M_PI;
        }

        double angdiff = ang1-ang2;
        while(angdiff<0) angdiff+=2*M_PI;
        while(angdiff>2*M_PI) angdiff-=2*M_PI;
        double ang = ang1-angdiff/2;

        vertices.push_back(Sommet(s2.x+pad*cos(ang)/sin(angdiff/2),s2.y+pad*sin(ang)/sin(angdiff/2)));
    }
    return vertices;
}
```

On définit un booléen dans la classe de l'obstacle qui détermine si un segment l'intercepte ou non avec la logique expliqué précédemment

```cpp
bool intersectionWithSegment(const Segment& segment, const double& eps) {
    // Détection d'intersection entre le segment et l'obstacle
    //double eps = max(fabs(segment.start_Sommet.y),fabs(segment.start_Sommet.x)) / pow(10,6);
    double m_segment,b_segment;
    if(fabs(segment.end_Sommet.x - segment.start_Sommet.x)>eps) {
        m_segment = (segment.end_Sommet.y - segment.start_Sommet.y) / (segment.end_Sommet.x - segment.start_Sommet.x);
        b_segment = segment.end_Sommet.y - m_segment * segment.end_Sommet.x;
    } else {
        m_segment = 0;
        b_segment = segment.end_Sommet.y;
    }
    int found = 0; bool first_Found = false;
    for(int i=0; i<vertices.size();i++) {
        if(found == 1) {
            found =2;
        } else {
            //cout<<vertices[i].y - m_segment * vertices[i].x - b_segment<<'\n';
            if(fabs(vertices[i].y - m_segment * vertices[i].x - b_segment) < eps) {
                if( ((vertices[i].y < segment.end_Sommet.y+(eps/3)) && (vertices[i].y > segment.start_Sommet.y-(eps/3)))
                    || ((vertices[i].y > segment.end_Sommet.y-(eps/3)) && (vertices[i].y < segment.start_Sommet.y+(eps/3))) ) {
                        found++;
                        //cout<<"found: "<<vertices[i]<<'\n';
                        if(i==0) first_Found=true;
                        else if(i==(vertices.size()-1) && first_Found) found =2;
                    }
            }
        }
        int j = (i==(vertices.size()-1))? 0:i+1;
        double x,y;
        if(fabs(vertices[j].x - vertices[i].x)<1e-4) {
            x = vertices[i].x;
            y = m_segment *x +b_segment;
```

```
    else if(fabs(segment.end_Sommet.x - segment.start_Sommet.x)>eps){
        double m = (vertices[j].y - vertices[i].y) / (vertices[j].x - vertices[i].x);
        double b = vertices[i].y - m * vertices[i].x;
        x = (b_segment - b) / (m - m_segment);
        if(fabs(m - m_segment) < eps) continue;
        y = m * x + b;
    } else {
        x=segment.end_Sommet.x;
        double m = (vertices[j].y - vertices[i].y) / (vertices[j].x - vertices[i].x);
        double b = vertices[i].y - m * vertices[i].x;
        y = m * x + b;
    }
    //cout<<x<<' '<<y<<'\n';
    if(SegmentIntersect(Sommet(x,y), Segment(vertices[i],vertices[j]), eps) && SegmentIntersect(Sommet(x,y), segment, eps)) {
        /*cout<<x<<' '<<y<<'\n';
        cout<<vertices[i]<<'\n';
        cout<<vertices[j]<<'\n';*/
        return true;
    }

}
```

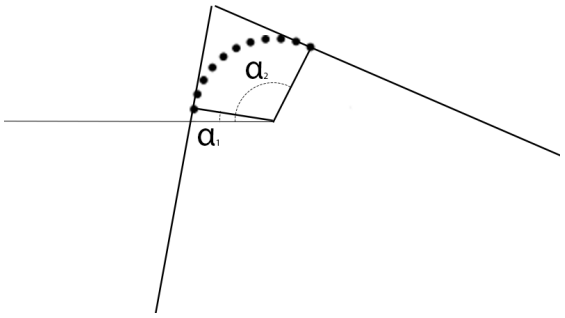
En utilisant tous ces fonctions, on crée le constructeur de la classe "arc_planification"

```
arc_planification(const Graph& graph, const Sommet& start, const Sommet& end, const vector<Obstacle>& obs, const double& eps)
    : Graph(graph), start_Sommet(start), end_Sommet(end), obstacles(obs), epsilon(eps) {
        for(long long unsigned int i=0;i<obstacles.size();i++) {
            for(auto iter = arcs.begin(); iter < arcs.end();) {
                if((obstacles[i]).intersectionWithSegment(*iter, eps)) {
                    iter = arcs.erase(iter);
                } else ++iter;
            }
        }
        int n=arcs.size();
        for(int i=0; i<n; i++) {
            Arc interm(arcs[i].end_Sommet, arcs[i].start_Sommet);
            (*this).AddArc(interm);
        }
    }
```

Après avoir ajouté les angles, on implémente l'algorithme de Dijkstra dans la fonction "trouverCheminMinimal" qui retourne le vecteur contenant les points à suivre

```
arc_planification(const Graph& graph, const Sommet& start, const Sommet& end, const vector<Obstacle>& obs, const double& eps)
    : Graph(graph), start_Sommet(start), end_Sommet(end), obstacles(obs), epsilon(eps) {
        for(long long unsigned int i=0;i<obstacles.size();i++) {
            for(auto iter = arcs.begin(); iter < arcs.end();) {
                if((obstacles[i]).intersectionWithSegment(*iter, eps)) {
                    iter = arcs.erase(iter);
                } else ++iter;
            }
        }
        int n=arcs.size();
        for(int i=0; i<n; i++) {
            Arc interm(arcs[i].end_Sommet, arcs[i].start_Sommet);
            (*this).AddArc(interm);
        }
    }
```

Il ne nous reste plus qu'à ajouter la rotation dynamique au chemin obtenu, pour déterminer les coordonnées des points suivantes :



L'implémentation se fait en suivant les étapes suivantes :

1. On calcule les angles α_1 et α_2
2. on définit le centre du cercle illustré dans la figure ci-dessous
3. on ajoute des points avec des angles équidistants entre α_1 et α_2, leur nombre dépendant de l'angle $\alpha_2 - \alpha_1$

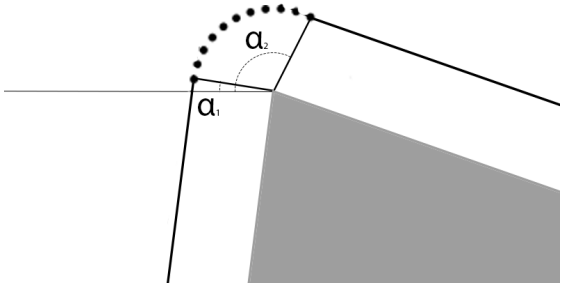
```
vector<Sommet> result_Physique;
result_Physique.push_back(result.front());
for(int b=1;b<n-1;b++) {
    Sommet s1 = result[b-1], s2 = result[b], s3 = result[b+1];

    double ang1,ang2;
    if(fabs(s2.x - s1.x) < eps) ang1=(s2.y - s1.y)?0:(-1)*M_PI;
    else {
        ang1 = atan((s2.y - s1.y)/(s2.x - s1.x)) - M_PI_2;
        if((s2.x - s1.x)<0) ang1 += M_PI;
    }
    if(fabs(s3.x - s2.x) < eps) ang2=(s3.y - s2.y)?0:(-1)*M_PI;
    else {
        ang2 = atan((s3.y - s2.y)/(s3.x - s2.x)) - M_PI_2;
        if((s3.x - s2.x)<0) ang2 += M_PI;
    }

    double angdiff = ang2-ang1;
    while(angdiff<-M_PI) angdiff+=2*M_PI;
    while(angdiff>M_PI) angdiff-=2*M_PI;

    if(angdiff<0) ang1 -= M_PI;

    double dis_angle = fabs(angdiff);
    int n_angles = 10-ceil(dis_angle*(10/M_PI));

    for(int i=0; i<=n_angles;i++) {
        double ang = ang1 + (angdiff)*i/n_angles;
        Sommet P = Sommet(s2.x+pad*cos(ang),s2.y+pad*sin(ang));
        result_Physique.push_back(P);
    }
}
result_Physique.push_back(result.back());
return result_Physique;
```

Toute l'implémentation des ces étapes est contenue dans la fonction "trouverCheminMinimal" de la classe arc_planification

```
Graph G= Graph();
for(int i =0;i<all_Sommets.size();i++) {
    for(int j = i+1; j<all_Sommets.size();j++) {
        G.AddArc(Arc(all_Sommets[i], all_Sommets[j]));
    }
}
arc_planification arc_plan = arc_planification(G, A, B, obs, eps);
arc_plan.AddAngles();

vector<Sommet> chemin = arc_plan.trouverCheminMinimal();

std::vector<sf::Vector2f> path;
while(!chemin.empty()) {
    Sommet CurrSom = chemin.back();
    chemin.pop_back();
    path.push_back(sf::Vector2f(CurrSom.x, CurrSom.y));
}
return path;
```

En raison du fait que l'algorithme de Dijkstra nous donne un chemin inversé et que la visualisation nécessite un vecteur de classes différentes de celle que nous avons définie, on inverse le chemin optimal tout en le transformant en un objet que nous nommons "path".

NB : Nous avons optimisé une alternative plus efficace vers la conclusion du projet, en créant le padding après avoir trouver la trajectoire optimale. Cette optimisation réside dans le fait que nous n'avons pas besoin de créer un padding autour des côtés non visités par notre objet et que cette démarche est beaucoup plus efficace en terme de ressources et temps d'exécution.



Chapitre 3

Gestion des données d'entrées et de sorties

3.1 Architecture

L'implémention de notre projet repose sur 3 parties essentielles :

— La gestion des données entrées.

— L'implémentation et l'exécution du code déjà expliqué en **??**.

— La visualisation des données de sortie.

Cependant, la solution finale que nous souhaiterons présenter dans ce projet est le développement d'une interface conviviale qui requiert peu, voire aucune, connaissance préalable en C++. Dans ce chapitre, nous nous concentrerons sur les étapes suivies et les méthodes mises en œuvre pour atteindre cet objectif.

3.1.1 Gestion des données d'entrées

3.1.1.1 Détection des sommets

Parcours de développement

L'exécution du code nécessite de fournir à l'exécuteur le nombre des obstacles et les sommets de chacun séparément afin qu'il puisse déterminer la position exacte de chaque obstacle et les arcs définissant son contour.

```
PS D:\Projects\TP_SIM202> g++ *.cpp
PS D:\Projects\TP_SIM202> ./a.exe
Entrer le point de d├®part: 280 208
Entrer le point d'arriv├®e: 39 106
Entrer le nombre d'obstacles: 3
Entrer le nombre de points de l'obstacle: 6
point n1: 309 252
point n2: 308 296
point n3: 54 296
point n4: 55 63
point n5: 113 63
point n6: 113 252
Entrer le nombre de points de l'obstacle: 4
point n1: 262 103
point n2: 244 193
point n3: 154 211
point n4: 172 121
Entrer le nombre de points de l'obstacle: 5
point n1: 191 100
point n2: 148 67
point n3: 164 14
point n4: 220 14
point n5: 236 66
(280,208),
(274.422,105.519),
(273.272,101.162),
(270.797,97.397),
(267.255,94.6127),
(263.011,93.0979),
(258.506,93.0096),
(196.141,107.265),
(194.847,107.389),
```

Dans les parties qui suivent, nous allons transmettre le parcours long et itératif d'essais et d'erreurs, qui nous avons poursuit pour attendre le résultat affiché dans le troisième chapitre.

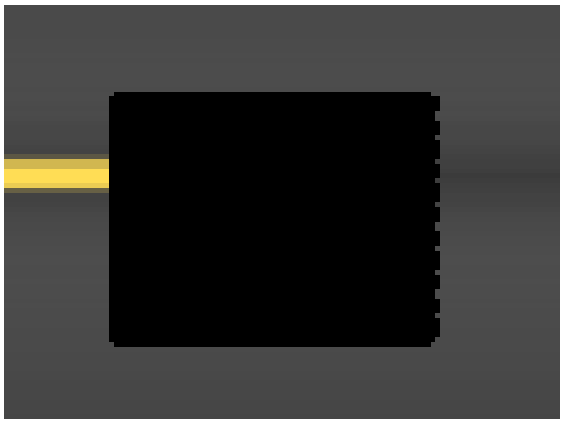
1ère étape :
Dans ce spectre et considérant que l'objectif de ce projet est d'assister l'utilisateur dans la recherche du chemin minimal entre un point de départ et un point d'arrivée sans rencontrer d'obstacles, la fourniture des coordonnées des sommets n'ai pas

assez facile pour l'utilisateur en raison de :

— La complexité de la simulation et la nécessité d'adapter un cas réel à nos besoins de modélisation.

— La spécification précise des coordonnées des sommets demande une expertise avancée en traitement d'image et en programmation C++.

2ème étape :
Une autre alternative aurait été de recevoir en entrée une image représentant la distribution des obstacles, d'afficher cette image à l'utilisateur et de lui demander de spécifier lui-même le nombre d'obstacles et de préciser les sommets en cliquant dessus :

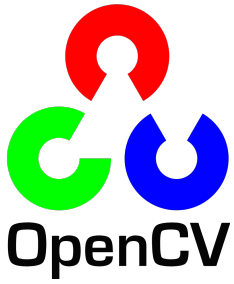


Cette version a été ultérieurement modifiée pour permettre à notre code C++ de déterminer à partir d'une image en noir et blanc le nombre d'obstacles ainsi que les coordonnées des points qui les composent. En effet, dans ce contexte, deux fonctions ont été implémentées :
La méthode "detectAndSaveContours" : Cette mèthode prend en entrée l'emplacement d'une image sous forme string et un vecteur de vecteur sommet. Cette méthode permet de former une image de chaque obstacle à part.
La méthode "findContour" : prend en paramètre l'image formée précédemment, crée un vecteur contenant les points qu'il forme et l'ajoute au vecteur "nodes".

```
void findContour(const sf::Image &image, std::vector<vector<Sommet>>& nodes);
void detectAndSaveContours(const std::string &imagePath, std::vector<vector<Sommet>>& nodes);
```

$\Longrightarrow$ Ces deux méthodes nous permettent de savoir le nombre des obstacles et les points qu'ils les forment pour les employer ultérieurement dans le dessin.

3ème étape :
Dans ce parcours, il ne reste plus qu'à détecter précisément les sommets de chaque obstacle. Cette étape nécessite l'intervention de techniques de traitement d'image, ce qui peut être laborieux et difficile à implémenter en C++. Pour trouver une solution efficace, nous avons décidé d'utiliser une nouvelle bibliothèque **OpenCV** et implementer un code qui ne permet de detecter les sommets des obstacles.




```cpp
#include <iostream>
#include <opencv2/opencv.hpp>
#include <classes.hpp>

// Définition de la fonction pour détecter les coins d'une image
void detectAndSaveContours(const string &imagePath, vector<vector<Sommet>>& coins) {
    // Charger l'image
    cv::Mat image = cv::imread(chemin_image);
    if (image.empty()) {
        std::cerr << "Erreur lors du chargement de l'image." << std::endl;
        return coins;
    }
    cv::Mat grayImage;
    cv::cvtColor(image, grayImage, cv::COLOR_BGR2GRAY);
    cv::Mat cornerStrength;
    cv::cornerHarris(grayImage, cornerStrength, 2, 3, 0.04);
    cv::normalize(cornerStrength, cornerStrength, 0, 255, cv::NORM_MINMAX);
    double threshold = 150; // Ajustez ce seuil selon vos besoins
    cv::Mat cornerMask = cornerStrength > threshold;
    for (int y = 0; y < cornerMask.rows; ++y) {
        for (int x = 0; x < cornerMask.cols; ++x) {
            if (cornerMask.at<uchar>(y, x) == 255) {
                coins.push_back(Sommet(x, y));
            }
        }
    }
}
```

3.1.1.2 Le choix du point de départ et d'arrivée

Pour les mêmes raisons évoquées précédemment, nous avons décidé de permettre à l'utilisateur de choisir ces deux positions par un simple clic.

Cependant, en observant ce processus, il est évident que l'utilisateur doit encore spécifier l'emplacement exact de l'image et avoir une compréhension de la structure des répertoires sur son appareil. Ainsi, nous n'avons pas encore atteint notre objectif : fournir un programme exécutable qui soit accessible sans nécessiter de connaissances informatiques avancées.

3.1.1.3 Insertion de l'image

Dans cette partie, nous avons décidé de résoudre ce problème en créant une interface GUI (Graphical User Interface) à l'aide de la bibliothèque *wxWidgets*. Cela permet à l'utilisateur de choisir l'image des obstacles qu'il souhaite grâce à un composant appelé "FilePickerCtrl".

3.1.2 Gestion des données de sortie

Une fois toutes les informations nécessaires à l'exécution de notre programme étaient fournies, le programme nous a retourné un vecteur "Trajectoire" indiquant le chemin minimal à suivre. À ce stade, nous avons décidé de permettre à l'utilisateur de visualiser le parcours en utilisant la bibliothèque "SFML" et de placer un objet en mouvement.

3.2 Conception

Dans ce même spectre, nous avons créé un projet **wxWidget** comportant essentiellement :
2 fichier.cpp

- show_the_pathMain.cpp
- show_the_pathApp.cpp

Et deux fichier header.hpp :

- show_the_pathMain.hpp
- show_the_pathApp.hpp

$\Longrightarrow$ Ces deux fichiers permettant la création de l'interface GUI (Graphical User Interface) et leur exécutions.

show_the_pathMain.hpp

```
class show_the_pathFrame: public wxFrame
{
    public:

        show_the_pathFrame(wxWindow* parent,wxWindowID id = -1);
        virtual ~show_the_pathFrame();

    private:
        wxString filePath; // Chemin du fichier sélectionné

        //(*Handlers(show_the_pathFrame)
        void OnQuit(wxCommandEvent& event);
        void OnAbout(wxCommandEvent& event);
        void OnFilePickerCtrl1FileChanged(wxFileDirPickerEvent& event);
        void OnPanel1Paint(wxPaintEvent& event);
        void OnButton1Click(wxCommandEvent& event);
        void OnToggleButton1Toggle(wxCommandEvent& event);
        void OnToggleButton1Toggle1(wxCommandEvent& event);
        //*)
```

Cette classe nous permet de créer la interface GUI, de choisir son design, offrir à l'utilisateur l'opportunité de choisir l'image qu'il souhaite en ouvrant les répertoires de son appareil, et d'executer le code C++ en appuyant sur le button "Suivant".

show_the_pathMain

```
IMPLEMENT_APP(show_the_pathApp);

bool show_the_pathApp::OnInit()
{
    //(*AppInitialize
    bool wxsOK = true;
    wxInitAllImageHandlers();
    if ( wxsOK )
    {
        show_the_pathFrame* Frame = new show_the_pathFrame(0);
        Frame->SetIcon(wxICON(aaaa));
        Frame->Show();
        SetTopWindow(Frame);
    }
    //*)
    return wxsOK;

}
```

```
#ifndef SHOW_THE_PATHAPP_H
#define SHOW_THE_PATHAPP_H

#include <wx/app.h>

class show_the_pathApp : public wxApp
{
    public:
        virtual bool OnInit();
};
```

Cette classe permet d'instancier l'interface, de l'afficher et de spécifier une icône pour l'application.

interface.hpp

Classe GraphVisualizer

La classe `GraphVisualizer` est une interface pour la visualisation graphique de graphes et de trajectoires. Elle utilise la bibliothèque SFML en C++ pour créer des fenêtres interactives et dessiner des graphes ainsi que des trajectoires. Voici une brève description de ses fonctionnalités :

— Le constructeur initialise un objet `GraphVisualizer` avec les textures des images de localisation, d'objet et d'arrière-plan. Il configure également les variables membres telles que `startClicked` et `endClicked`.

— La méthode `visualize` crée une fenêtre SFML permettant à l'utilisateur de cliquer pour définir les positions de départ et d'arrivée sur le graphe, calculer le chemin optimal et observer le résultat.

`drawlocImage` dessine l'image de la localisation (ou autre objet) à une position spécifique sur la fenêtre.

`findBoundingBox` calcule la boîte englobante du graphe en déterminant les coordonnées les plus à gauche, à droite, en haut et en bas des sommets.

```
float angle(const sf::Vector2f& s);
//Calculer les angles entre les positions

GraphVisualizer(const std::vector<vector<Sommet>>& nodes,const std::string& backgroundImage) :
// Constructeur de la classe GraphVisualizer, initialise les textures et les sprites avec les images spécifiées
// et initialise les indicateurs de clic de début et de fin.

void visualize() :
//Fonction qui lance la visualisation du graphe et permet à l'utilisateur de sélectionner les points de départ et d'arrivée.

bool loadTexture(sf::Texture& texture, const std::string& imagePath) :
//Fonction privée qui charge une texture à partir d'un chemin d'image spécifié.

void drawGraph(sf::RenderWindow& window) :
//Fonction privée qui dessine le graphe en reliant les sommets avec des lignes.

void drawlocImage(sf::RenderWindow& window,sf::Sprite &sprite, double X, double Y) :
//Fonction privée qui dessine une image à une position spécifiée dans la fenêtre.

sf::FloatRect findBoundingBox(const std::vector<std::vector<Sommet>>& nodes) :
//Fonction privée qui trouve la boîte englobante du graphe, c'est-à-dire le rectangle délimitant tous les sommets du graphe.
#endif // INTERFACE_HPP_INCLUDED
```

Chapitre 4

Développement

4.1 Implémentation

Dans une première partie, nous avons utilisé la bibliothèque wxWidgets pour lancer un projet GUI. Nous avons créé une interface en utilisant les inclusions suivantes :

```
#include <wx/msgdlg.h>
#include <wx/string.h>
#include <wx/utils.h>
```

Notre interface repose principalement sur deux outils : `FilePickerCltr`, qui nous permet de choisir une image, et un bouton nommé `Suivant` (`OnToggleButton`). Une fois ce bouton pressé, notre programme principal se lance.

Ensuite, notre programme commence par charger les images nécessaires pour la visualisation. À ce stade, l'utilisation d'une bibliothèque de traitement d'images devient indispensable. Nous avons donc fait appel à la bibliothèque SFML :

```
#include <SFML/Graphics.hpp>
```

Ensuite, nous avons implémenté plusieurs fonctions :

- `GraphVisualizer visualizer (nodes,filePath.ToStdString())` : initialise une instance de `GraphVisualizer`.
- `visualizer.visualize( )` : dessine l'arrière-plan, les obstacles, et récupère les positions de départ et d'arrivée obtenues par un simple clic de la part de l'utilisateur.

4.2 Test

Nous avons réalisé plusieurs tests approfondis pour évaluer et valider les fonctionnalités de notre programme de planification de trajectoire en C++. Ces tests ont été effectués à différentes étapes du développement, permettant ainsi de détecter et de corriger les éventuelles erreurs ou anomalies.

Tout d'abord, nous avons effectué des tests unitaires pour chaque fonctionnalité clé du programme. Ces tests nous ont permis de vérifier si les fonctions individuelles fonctionnaient correctement et renvoyaient les résultats attendus.

Ensuite, nous avons réalisé des tests de validation en générant différents types de labyrinthes, tels que des labyrinthes de tailles variées, avec différentes configurations de passages et de murs. Nous avons vérifié si le programme produisait des labyrinthes cohérents et sans erreurs, en nous assurant que tous les passages étaient connectés et qu'il n'y avait pas de boucles ou de chemins inaccessibles.

De plus, nous avons effectué des tests de performance pour évaluer l'efficacité de notre algorithme de planification de trajectoire. Nous avons mesuré le temps d'exécution de différentes images, afin d'optimiser notre programme et garantir qu'il fonctionnait rapidement et de manière efficace.

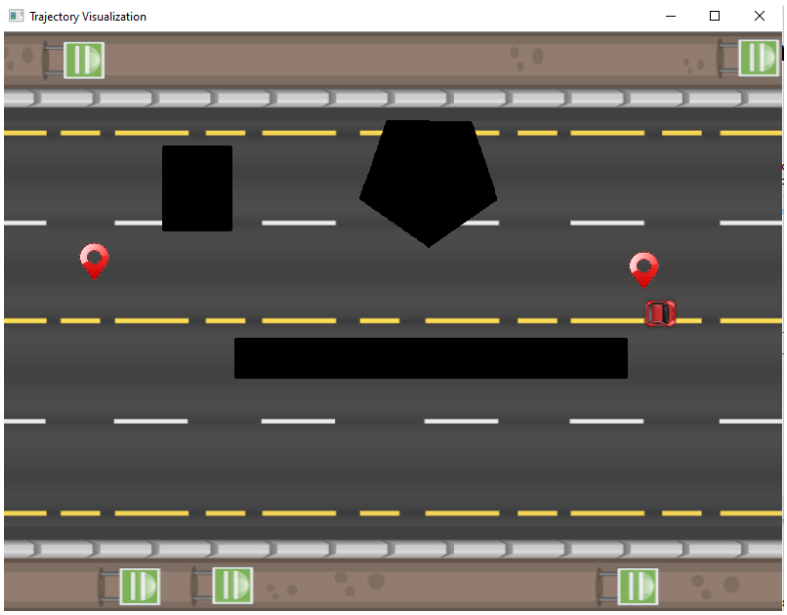
En parallèle, nous avons également effectué des tests d'intégration pour vérifier l'interaction harmonieuse entre les différentes parties du programme. Nous avons vérifié que les modules se connectaient correctement, que les données étaient transmises correctement et que le programme fonctionnait de manière globale sans conflits ni erreurs.

Enfin, nous avons sollicité des utilisateurs pour tester notre programme et nous fournir des retours d'expérience. Leurs commentaires et suggestions nous ont permis d'apporter des améliorations supplémentaires, d'identifier des scénarios d'utilisation spécifiques et de résoudre les problèmes auxquels nous n'aurions pas pensé initialement.

Grâce à ces tests rigoureux, nous avons pu valider la robustesse, la précision et les performances de notre programme de planification de trajectoire en C++. Ils ont joué un rôle essentiel pour garantir la qualité du logiciel et offrir une expérience utilisateur fluide et satisfaisante.

Exécution


Plannification de Trajectoire
Plannifiez votre trajet facilement :
Browse
Next
Graph Visualization



4.3 Difficultés rencontrées et solutions adaptées

Au cours de la création de ce projet, divers problèmes nous ont rencontré :

1. Lors de l'implémentation de l'algorithme, nous avons rencontré des difficultés pour visualiser la sortie et évaluer sa validité. Pour remédier à cela, nous avons fait appel à nos connaissances antérieures en utilisant un code Python approprié pour dessiner les coordonnées renvoyées.
2. L'installation et la configuration des bibliothèques ont été une tâche complexe. Les différences entre les versions de MinGW, ainsi que celles des versions récentes de SFML et de wxWidget, ont posé un défi majeur. De plus, le manque de tutoriels et de documentation sur la résolution des erreurs rencontrées a amplifié la difficulté. Cette situation a nécessité beaucoup de temps, d'espace de stockage sur l'ordinateur et d'essais pour adapter les codes aux différentes versions utilisées.

   Pour surmonter ces problèmes, nous avons dû être plus attentifs à la configuration des logiciels et des bibliothèques. Nous nous sommes également familiarisés avec l'utilisation de l'invite de commande CMD, du logiciel CMake, ainsi que des environnements de développement tels que VSCode et Code : :Blocks. Cette expérience nous a permis de mieux gérer les défis liés à l'installation et à la configuration des outils de développement.

3. L'installation de la bibliothèque OpenCV s'est avérée être notre défi ultime, que nous n'avons malheureusement pas réussi à surmonter. Cependant, cela nous a poussés à explorer d'autres solutions. Nos tentatives d'installation de cette bibliothèque ont rencontré plusieurs obstacles :

   - La non-disponibilité de la bibliothèque pour le GCC+Mingw.
   -L'obligation d'utiliser le logiciel CMake pour créer les fichiers d'inclusion adaptés à notre version de Mingw.
   -La recherche de méthodes de configuration et la résolution de problèmes tels que "Point d'entrée non trouvé", pour lesquels nous avons échoué à trouver des solutions.

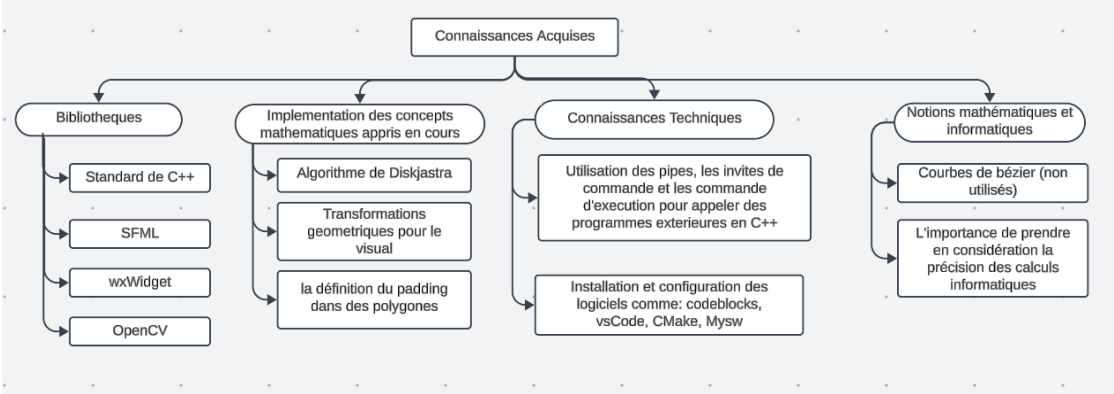
   Face à ces difficultés, nous avons décidé de redoubler d'efforts et d'explorer les connaissances acquises lors de notre parcours scolaire. Nous avons ainsi envisagé :

   - L'implémentation de cette bibliothèque avec le langage de programmation Python.
   -La découverte de cette bibliothèque pour chacun des langages C++ et Python, en approfondissant nos compétences dans ces domaines respectifs.

   Effectivement, ces alternatives nous ont permis de maintenir notre élan dans le projet malgré les obstacles rencontrés lors de l'installation de la bibliothèque OpenCV.

4. L'appel d'un programme externe en C++ s'avère généralement plus simple en utilisant un programme C++ avec une invite de commande, car cela facilite la connexion entre les deux programmes. Notamment, cela se révèle pratique lorsque le programme Python nécessite de spécifier le chemin de l'image pour déterminer les coins. Cependant, lors de l'implémentation de wxWidget et l'élimination de l'utilisation de bash, cette connexion est devenue impossible. Cela nous a obligés à acquérir des connaissances sur l'exécution par l'invite de commande et à modifier notre code initial en conséquence.

4.4 Connaissances Acquises



Conclusion

Pour mener à bien ce projet, une répartition efficace des tâches a été réalisée entre les membres de l'équipe. Aymen Ben Ammar s'est concentré sur la mise en œuvre de l'algorithme de planification de trajectoire, tandis qu' Eya Jlassi s'est occupée de la conception et de l'implémentation de l'interface graphique utilisateur, depuis l'utilisation de wxWidget jusqu'à la visualisation avec SFML.

Aymen Ben Ammar, fort de son expertise en programmation, a développé les fonctionnalités du programme de planification de trajectoire en utilisant le langage C++. Il a veillé à respecter les principes de conception logicielle et à maintenir un code clair et efficace.

De son côté, Eya Jlassi a effectué des recherches approfondies sur les bibliothèques nécessaires à la création de l'interface utilisateur et à la visualisation et crée l'interface de l'application et la représentation visuelle des données d'entrée et de sortie.

Malgré cette répartition des tâches, ils ont travaillé en étroite collaboration pour résoudre les problèmes et les dysfonctionnements des deux parties du programme, assurant ainsi une compréhension globale du projet par chacun et favorisant l'apprentissage mutuel.

Le rapport final est le fruit d'un travail collectif. Ils ont élaboré le plan ensemble et structuré les informations de manière claire, en incluant les objectifs du projet, la description des fonctionnalités implémentées, les résultats obtenus et les conclusions tirées.

Chacun d'entre eux a contribué au développement du code du programme en mettant en pratique les concepts et les techniques appris lors des cours de SIM201 et RO201 portant sur le langage C++.

La collaboration entre Eya et Aymen a été essentielle pour la réussite de ce

projet. Leur expertise complémentaire et leur communication constante nous ont permis d'aboutir à un programme fonctionnel de génération de labyrinthe en C, accompagné d'un rapport précis et détaillé sur le processus de développement.

Ce travail d'équipe leur a permis de valoriser leurs compétences individuelles tout en assurant une cohésion globale dans la réalisation du projet.

En plus de la répartition des tâches, ils ont adopté une approche méthodique et organisée en utilisant le cycle de conception descendant. Ce processus itératif leur a permis de diviser le projet en sous-tâches gérables, facilitant ainsi le développement et la coordination entre les membres de l'équipe.

Ils ont commencé par une phase de planification, au cours de laquelle ils ont défini les objectifs globaux, identifié les fonctionnalités clés et établi une feuille de route pour le développement. Cela leur a donné une vision claire du projet et des étapes nécessaires pour atteindre leurs objectifs.

Ensuite, ils ont entamé la phase de conception en analysant les exigences du projet et en établissant une structure globale pour leur programme en C++. Cette approche leur a permis de structurer leur travail de manière efficace, en veillant à ce que chaque étape soit soigneusement planifiée et exécutée.

En adoptant le cycle de conception descendant, ils ont amélioré leur efficacité, facilité la collaboration et assuré que le résultat final était conforme à leurs objectifs initiaux.