Placement de tâches dans un NOC

Projet L3 II-IUP

Rodney Kenneth ABOUE
Antoine PECOUT
Victor PIRIOU
Anaël QUEANT
Matteo RIO
Licence 3 Informatique

JUIN 2020

Enseignant : Laurent LEMARCHAND

Établissement : Université de Bretagne Occidentale UFR Sciences

Table des matières

1. Présentation du problème

L'application permet de répondre au problème suivant : un drone de surveillance doit patrouiller dans un port, puis revenir à son point de départ. On cherche à lui faire parcourir la distance la plus courte possible, tout en maximisant la surface mise à portée de son radar.

On prend en compte le fait que le robot effectue sa patrouille dans un unique polygone qui englobe la totalité des points à surveiller, et part d'un point défini à l'avance. Certaines zones de son parcours sont des zones interdites, soit des polygones par lesquels le robot ne doit pas passer.

Le but serait de pouvoir dessiner sur le site data.shom.fr la zone à surveiller par le robot et les zones interdites, puis de générer des points dans le polygone englobant, entre lesquels le robot pourrait se déplacer. Ensuite, il faudrait calculer un chemin pour le robot, de manière à ce que le radar couvre une surface de points la plus grande possible, tout en ayant un parcours le plus court possible. Ensuite, on voudrait pouvoir donner en sortie deux fichiers PDF : l'un détaillant le problème, et le second donnant la solution à ce problème en indiquant le chemin du drone.

2. Utilisation

2.1. data.shom.fr

Pour définir la zone du visée il faut se rendre dans la section « dessin » de data.shom.fr et créer une nouvelle carte. Cliquer sur l'outil de dessin de polygone et dessiner une première figure en mer qui sera le secteur pris en compte par l'algorithme.

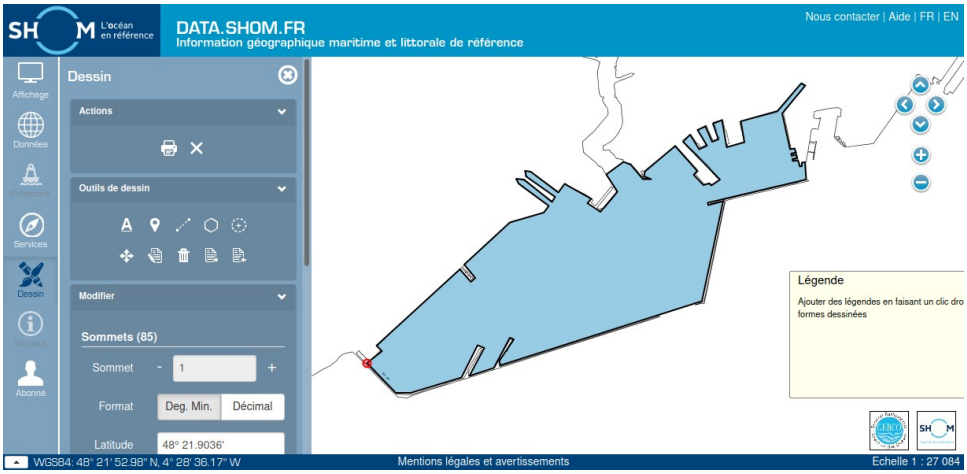

Figure 1: exemple de carte minimale sur le port de brest, data.shom.fr

Les bases sont posées, les étapes suivantes de dessin sont facultatives. Par la suite il est possible de placer d'autres polygones à l'intérieur du premier. Ces derniers seront des zones interdites interprétées comme des obstacles physiques infranchissables. Un point de départ du chemin solution peut éventuellement être saisi. Pour cela il faut utiliser l'outil d'ajout de point et placer un et un seul point dans la première figure et hors d'un polygone interdit.

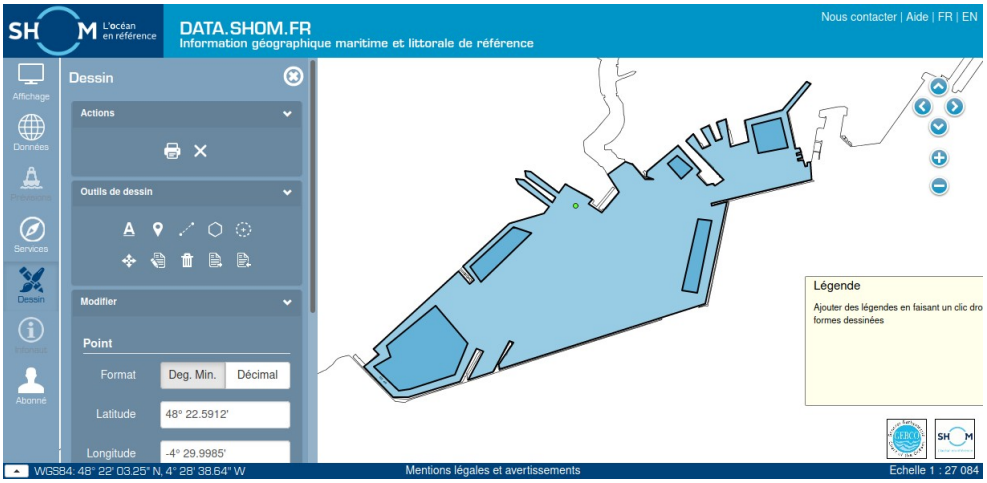

Figure 2: exemple de carte finale sur le port de brest, data.shom.fr

Le dessin de la carte est terminé il est temps d'exporter la carte via l'outil d'export en KML. Le ficher définissant les limites du problème est maintenant prêt à être résolu par l'application.

2.2. Application

3. Réalisation

3.1. Structures de données

Le programme utilise deux structures de données : la structure point (pointp), et la structure liste de polygones (liste_polygone) dont les champs sont décrits sur la capture d'écran ci-dessous.

```
11 typedef struct st_point{
12     int id;
13     double x;
14     double y;
15     char etat;
16
17     pointp* next;
18 } pointp;
19
20 typedef struct st_liste_polygone{
21     pointp* polygone;
22     liste_polygone* next;
23 }liste_polygone;
24
```

La structure point contient deux doubles x et y, qui indiquent ses coordonnées, récupérées via la lecture d'un export KML du site data.shom.fr. Chaque point contient également un pointeur "next" vers un autre point, ainsi, chaque point peut être considéré comme maillon d'une liste chaînée. Les points et les listes de points sont regroupés sous le même type . Les champs "état" et "id" ont été laissés mais ne sont pas utilisés. Lorsque l'on crée une liste de points, le premier point est alloué mais ses champs restent vides, il sert simplement à indiquer le début de la liste. La liste commence donc à partir du second point.

La liste de polygones fonctionne de la même manière. Elle a un champ "polygone" qui est un pointeur vers une liste de points, le point courant, et un champ "next", qui contient un pointeur vers une autre liste de polygones.

Ces structures de données sont allouées, remplies, manipulées et libérées en interne, via les fonctions du fichier "structure_point_polygone.c". Ce fichier permet : d'initialiser une liste de polygones, d'y ajouter un élément, d'afficher la liste en console et de la supprimer. Les mêmes opérations sont disponibles pour les listes de points, avec en plus des fonctions de recherche (non-utilisées) et un calcul de taille de la liste.

Détaillons quelques-unes de ces fonctions :

- init_point : Ne prend pas de paramètre. La fonction alloue un point et retourne un pointeur vers celui ci.
- add_point_liste : Prend en paramètre une liste de points, et deux doubles correspondant aux coordonnées x et y d'un point à ajouter. La fonction parcours la liste de points passée en paramètre et si un point avec les mêmes coordonnées existe déjà dans la liste, la fonction retourne -1. Sinon elle crée un nouveau point avec les

coordonnées données et ajoute un pointeur vers ce point dans le champs « next » du dernier élément de la liste. Puis renvoie l’id unique du point.

- init_liste_polygone : Ne prend pas de paramètre. La fonction alloue une liste de polygone, et appelle "init_point" pour allouer son champ "polygone" qui correspond à la première liste de points. Le pointeur vers la structure ainsi créée est retourné par la fonction.
- add_polygone_liste : Prend en paramètre une liste de point et une liste de polygones. La fonction ajoute à la fin de la liste de polygone en paramètre, la liste de points en paramètre. Aucune valeur n'est renvoyée, mais si l'une des deux listes vaut NULL, l'exécution s'arrête.

–

3.2. Traitement de l’entrée

L’entrée de l’application est un ficher kml exporté du site data.shom.fr. Cette partie est dédiée à expliquer comment les données du ficher importé sont récupérées. Pour voir comment utiliser data.shom.fr pour réaliser un ficher reconnu par l’application voir « 2.1 data.shom.fr».

Dans le programme cette fonctionnalité est codée par la fonction «extraire polygone ». La fonction prend en paramètres le nom du ficher et un pointeur vers une liste de polygones. Elle renvoie un pointeur vers un point si un point de départ à été détecté et NULL sinon. Elle remplit également la liste passée en paramètre avec les polygones détectés.

La fonction ouvre le ficher kml avec un fopen et va le parcourir jusqu’à ce que aucun nouveau polygone ne soit découvert. Nous avons remarqué qu’il y a une balise <coordinates></coordinates> si et seulement si il y a un polygone/point. Donc le programme cherche la chaîne «<coordinates> » grâce à une suite de fgetc. Si une telle chaîne est présente alors la fonction retourne et le curseur de parcours est avancé à la suite de la chaîne. Sinon on arrive à la fin du ficher et le parcours, et donc le traitement de l’entrée est terminé.

Il y a donc de nouvelles coordonnés à enregistrer. Nous allons donc créer et initialiser une nouvelle liste chaînée de points pour stocker les valeurs. Dans la balise les données sont organisées comme ceci : «>%f,%f%f,%f (…)%f,%f< ». Nous allons donc récupérer les couples de coordonnés avec un fscanf, créer un point, l’ajouter à la liste puis tester le caractère suivant. Si c’est un ‘ ‘ alors il y a d’autres points à saisir sinon c’est un ‘<’ et l’enregistrement est terminé pour ce polygone. Une fois l’enregistrement fini regarde si la liste comporte un seul élément. Si c’est le cas alors elle est stockée dans le point de départ, sinon elle est ajoutée à la liste de polygones.

Ensuite la fonction reboucle et teste la présence de nouvelles coordonnées. Si le teste est négatif elle retourne.

3.3. Génération des points

La génération des points va se faire via les fonctions du fichier « gen_point.c » et « gen_point.h ».

3.3.1 La structure utilisée et sa création

Dans un premier temps, afin de stocker les sommets et les futurs points, nous avons mis en place une structure « polygone ».

```
typedef struct{
        //Un sommet du polygone
        pointp * sommetPoly;
        //Liste de pointeur de point se trouvant à l'interieur du polygone
        pointp ** listePointPoly;
        //Nombre de point dans le poly hors sommet (nécessaire pour parcourir la liste)
        int nbPoint;
}polygone;
```

Dans cette structure, nous stockons l'adresse d'un seul sommet (les autres sont liés à cet unique sommet via un pointeur), une liste dynamique de pointeur de point qui va servir à stocker l'ensemble des points du polygone et le nombre de point afin de pouvoir parcourir cette liste sans tenter d'exécuter une action sur une case inexistante.

Pour créer un élément de cette structure, nous avons mis en place une fonction(polygone* creerPoly(pointp*)) qui va prendre en argument un sommet du polygone et va vous renvoyer un pointeur vers une structure « polygone » avec la liste de point générer à 0 et son nombre prenant la valeur 0.

```
// Création d'un structure polygone à partir d'un sommet
polygone * creerPoly(pointp * unSommet){
        polygone * lePoly = (polygone*)malloc(1*sizeof(polygone));
        lePoly->nbPoint = 0;
        lePoly->sommetPoly = unSommet;
        lePoly->listePointPoly=(pointp**)malloc(lePoly->nbPoint*sizeof(pointp*));
        return lePoly;
}
```

La fonction creerPoly renvoi un pointeur vers le polygone. Elle prend en argument un pointeur vers un point qui doit être un sommet (**Attention** : aucune vérification est faite pour savoir si c'est bien un sommet). On alloue un polygone, on met récupère le pointeur vers « sommetPoly », on alloue la liste de points à une taille 0 et on met la variable nbPoint à 0.

3.3.2 Le lancement de la génération de point

Dans un second temps, afin de générer automatiquement les points, nous avons mis en place la fonction « gen_point_polygone ». L'objectif de cette fonction est de vérifier si le point de départ est bien dans le polygone, puis de démarrer la génération de tous les points possible en fonction de la distance et de stocker ces points dans la structure polygone vu précédemment tout en vérifiant que les points générés soit dans le polygone.

```
// Vérifier le point de départ et lancer ensuite la création des points
bool gen_point_polygone(polygone* lePolygone, pointp* pointDepart, float distanceVoisin){
        if(!isInside(pointDepart, lePolygone)){
                return false;
        }
        lePolygone->listePointPoly = (pointp **) realloc(lePolygone->listePointPoly, ++(lePolygone->nbPoint)*sizeof(pointp*));
        pointp* nouveauPoint = init_point_liste_points(); // pour avoir un id de point correct (compris entre 0 et le nb de pts de la grille-1)
        nouveauPoint->x = pointDepart->x;
        nouveauPoint->y = pointDepart->y;
        lePolygone->listePointPoly[lePolygone->nbPoint - 1] = nouveauPoint;
        gen_point(lePolygone, lePolygone->listePointPoly[lePolygone->nbPoint-1], distanceVoisin);
        return true;
}
```

La fonction va prendre en paramètre un pointeur vers une structure du polygone englobant, le point de départ de la génération de point et la distance entre les différents points générés. La fonction va donc d'abord s'assurer que le point de départ se trouve bien dans le polygone, puis va ré-allouer la place de la liste de point pour en ajouter 1, on va ensuite générer un nouveau point et mettre les informations du point de départ dans celui-ci afin d'éviter les modifications sur l'ID qui aurait pus avoir lieu avant l'exécution de cette fonction, enfin ajouter le point à la liste de point et commencer la génération de manière récursive via la fonction « gen_point() » .

3.3.3 La vérification de la présence d'un point dans un polygone

Dans la fonction précédente, nous avons pu voir l'utilisation de la fonction isInside(). Dans cette section, nous allons voir et expliquer l'ensemble des méthodes autour de cette vérification.

Dans cette fonction, nous avons en argument un pointeur vers le point à vérifier et le polygone dans lequel on veut savoir si le point s'y trouve ou non. La stratégie appliquée pour faire cette vérification est de créer un segment horizontal entre le point de départ et un point généré à la même valeur de coordonnée y, puis de compter le nombre d'intersections entre ce segment et les arêtes du graphe. Si le nombre d'intersections est impair ou que les 3 points sont colinéaire et que le point à tester est sur l'arête, alors on considère que le point se trouve dans le polygone.

```
// Returns true if the point p lies inside the polygon[] with n vertices
bool isInside(pointp * lePoint, polygone* lePolygone){
        // Create a point for line segment from p to infinite
        pointp* extreme = init_sommet();
        extreme->x = INF;
        extreme->y = lePoint->y;
        pointp* sommet = lePolygone->sommetPoly;
        // Count intersections of the above line with sides of polygon
        int count = 0;
        do{
                // Check if the line segment from 'p' to 'extreme' intersects
                // with the line segment from 'polygon[i]' to 'polygon[next]'
                if (doIntersect(sommet, sommet->next, lePoint, extreme)){
                        // If the point 'p' is colinear with line segment 'i-next',
                        // then check if it lies on segment. If it lies, return true,
                        // otherwise false
                        if (orientation(sommet, lePoint, sommet->next) == 0){
                                return onSegment(sommet, lePoint, sommet->next);
                        }
                        count++;
                }
                sommet = sommet->next;
        } while (sommet->id != lePolygone->sommetPoly->id);
        // Return true if count is odd, false otherwise
        return count&1;  // Same as (count%2 == 1)
}
```

Pour commencer, on génère un pointeur vers le point dit « extrême », on lui donne une valeur « INF » pour la coordonnée X (définie à 10 000 en temps que variable global, l'utilisation d' « INT_MAX » pourrait causer des problèmes d'overflow) et la même valeur Y que le point à tester. Ensuite, nous testons sur l'ensemble des arêtes l'intersection de la droite formée par le point à tester et le point « extrême » (fonction présentée à page 11/18). S'il y a une intersection, on vérifie si les 2 sommets de l'arête en cours et le point à tester sont colinéaires (fonction présentée à page 12/18). Si les points sont colinéaires, on retourne si le point se trouve sur l'arête ou non (fonction présentée à page 13/19). Sinon, on finit la boucle et on retourne « TRUE» si le compte d'intersection est impair.

Maintenant que nous avons vu isInside(), on va rentrer dans les détails de la fonction doIntersect(). L'objectif de cette fonction est de vérifier que les 2 droites formés par les 4 points en paramètre se croise dans leur intervalle respectif.

```
bool doIntersect(pointp* p1, pointp* q1, pointp* p2, pointp* q2){
        // Find the four orientations needed for general and
        // special cases
        int o1 = orientation(p1, q1, p2);
        int o2 = orientation(p1, q1, q2);
        int o3 = orientation(p2, q2, p1);
        int o4 = orientation(p2, q2, q1);

        // General case
        if (o1 != o2 && o3 != o4){
                return true;
        }

        // Special Cases
        // p1, q1 and p2 are colinear and p2 lies on segment p1q1
        if (o1 == 0 && onSegment(p1, p2, q1)) return true;

        // p1, q1 and p2 are colinear and q2 lies on segment p1q1
        if (o2 == 0 && onSegment(p1, q2, q1)) return true;

        // p2, q2 and p1 are colinear and p1 lies on segment p2q2
        if (o3 == 0 && onSegment(p2, p1, q2)) return true;

        // p2, q2 and q1 are colinear and q1 lies on segment p2q2
        if (o4 == 0 && onSegment(p2, q1, q2)) return true;

        return false; // Doesn't fall in any of the above cases
}
```

La fonction prend en paramètre 4 points et renvoie un booléen. On récupère ensuite leur orientation et on les compare 2 à 2 pour savoir s'il y a une intersection direct ou non. S'il n'y a pas d'intersection direct, on va regarder si un des tests renvois une colinéarité et si le point se trouve sur le segment. Si l'un des tests est bon, on renvoie vrai. Si aucun des tests est correct, alors la fonction va retourner faux, car il n'y a aucune intersection entre les 2 segments.

La fonction « orientation » est beaucoup utilisée dans les fonctions précédentes, nous allons la détailler ici. L'objectif est de comparer les vecteurs via les 3 points fournis en paramètre afin de déterminer si les points sont colinéaires ou non. Pour savoir si les 3 points sont colinéaires, nous calculons les 2 vecteurs puis on regarde si la soustraction de (x*y') - (x'*y) = 0 . Si c'est égale à 0, alors les points sont colinéaire, sinon si la valeur est positif alors c'est le sens horaire, sinon c'est le sens non-horaire.

```
int orientation(pointp* p, pointp* q, pointp* r){
        int val = ((q->y - p->y) * (r->x - q->x)) - ((q->x - p->x) * (r->y - q->y));
        if (val == 0) return 0; // colinear
        return (val > 0)? 1: 2; // clock or counterclock wise
}
```

En argument on envoie les 3 points qui vont former les vecteurs. Ensuite, on compare les vecteur « pq » et « qr » afin de savoir s'ils sont colinéaires. Si la valeur rendue après le calcul (x*y') - (x'*y) est 0, alors on retourne 0 pour annoncer qu'ils sont colinéaires. Si ce n'est pas colinéaire, on regarde si c'est dans le sens horaire ou non.

On a vu précédemment l'utilisation de la fonction onSegment (dans la fonction isInside), il est temps de la détailler. L'objectif de cette fonction est de vérifier que les coordonnées d'un point se trouve bien dans un intervalle créé par 2 points (une arrête dans notre cas). Il suffit donc de comparer les valeur minimums et maximums en X et Y avec les valeurs de coordonnées du point à tester.

```
// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(pointp* p, pointp* q, pointp* r){
        if ( q->x <= max(p->x, r->x) && q->x >= min(p->x, r->x) && q->y <= max(p->y, r->y) && q->y >= min(p->y, r->y)){
                return true;
        }
        return false;
}
```

La fonction prend en argument 3 points (« p » et « r » sont les points correspondant aux extrémités et « q » le point à tester) et retourne un booléen. Nous testons si la coordonnée X du point « q » est compris entre le minimum et le maximum des coordonnées en X des extrémités puis nous testons si la coordonnée Y du point « q » est compris entre le minimum et le maximum des coordonnées en Y des extrémités. Si les coordonnées sont bien comprises dans les intervalles, alors on renvoie vrai, sinon nous renvoyons faux.

3.3.4 La génération de point via récursivité

Maintenant que nous avons vu la structure, vu le lancement et comment on vérifier la présence d'un point dans un polygone, nous allons nous expliquer la méthode récursive de la génération de point. **Attention :** nous n'allons pas présenter l'ensemble de la fonction, mais seulement la première partie, car le reste est exactement la même chose avec simplement les valeurs de coordonnées X et Y qui sont modifiés.

La génération de point par récursivité est très simple : on créé un point avec des valeurs X et Y modifier par rapport aux coordonnées du point envoyé en argument (X+distance, X-distance, Y+distance, Y-distance), on vérifie ensuite que celui-ci est dans le polygone, puis on vérifie que ce point n'existe pas déjà, enfin, on l'ajoute à la liste des points de la structure polygone et on réutilise la fonction sur le nouveau point. On réitère l'opération 3 fois pour générer des point tout autour du point donné en argument.

```
// Générer l'ensemble des points de manière récursive
void gen_point(polygone* lePolygone, pointp* pointDepart, float distanceVoisin){
        pointp* nouveauPoint = init_point_liste_points();
        //Point à l'OUEST
        // Ajustement des coordonnées avec la distance de création
        nouveauPoint->x = pointDepart->x - distanceVoisin;
        nouveauPoint->y = pointDepart->y;
        //On vérifie que le nouveau point se trouve bien dans le polygone
        if(isInside(nouveauPoint, lePolygone) == true){
                // On vérifie si le point n'existe pas
                bool existant = false;
                for(int i = 0; i < lePolygone->nbPoint && existant == false; i++){
                        if(nouveauPoint->x == lePolygone->listePointPoly[i]->x && nouveauPoint->y == lePolygone->listePointPoly[i]->y){
                                existant = true;
                        }
                }
                // Si il n'existe pas
                if(!existant){
                        // On aggrandie l'allocation en mémoire du tableau de point
                        lePolygone->listePointPoly = (pointp **)realloc(lePolygone->listePointPoly, ++(lePolygone->nbPoint)*sizeof(pointp*));
                        lePolygone->listePointPoly[lePolygone->nbPoint - 1] = nouveauPoint;
                        // On appel de nouveau la fonction avec le point fraîchement ajouté à la liste
                        gen_point(lePolygone, lePolygone->listePointPoly[lePolygone->nbPoint-1], distanceVoisin);
                        nouveauPoint = init_point_liste_points();
                }
        }
```

En argument : le polygone englobant qui va stocker l'ensemble des points, le « point de départ » qui correspond au point autour duquel on va générer des points et la distance à laquelle les points vont être générés par rapport au point de départ. On initialise un nouveau point, on lui donne ensuite des coordonnées X et Y (nous modifions 2 fois X pour générer des points à l'EST et l'OUEST, et 2 fois Y pour générer des points au NORD et au SUD). Après, on va vérifier si le nouveau point est toujours dans le polygone via isInside. S'il y est, on va vérifier que ce point n'existe pas déjà en comparant les coordonnées pour éviter les doublons. S'il n'existe pas encore, on va ré-allouer le tableau de point avec une taille en plus, puis ajouter le point à la fin du tableau. On va ensuite relancer la fonction avec le point fraîchement créé.

Pour continuer l'exécution de la fonction, on recrée un point à la fin pour ne pas bloquer l'exécution par la suite et éviter d'initialiser des pointeurs inutile. Si jamais lors de la dernière partie de la fonction, le point n'est pas correct, on « free » l'adresse.

3.3.5 Suppression des points des polygones interdits

Maintenant que nous avons générer l'ensemble des points du polygone englobant, on va retirer les points des polygones interdit.

La méthode utilisée ici est très simple, on va tester l'ensemble des points de la liste et regarder si le point testé est dans le polygone interdit. Si celui-ci est dedans, on va le remplacer par le dernier point de la liste.

```
// Retire l'ensemble des points du polygone qui se trouvent dans le polygone interdit
void delete_point_polygone(polygone* lePoly, polygone* lePolyInterdit){
        for(int i = 0; i< lePoly->nbPoint; i++){
                if(isInside(lePoly->listePointPoly[i], lePolyInterdit) == true){
                        free(lePoly->listePointPoly[i]);
                        lePoly->listePointPoly[i] = lePoly->listePointPoly[lePoly->nbPoint-1];
                        lePoly->listePointPoly = (pointp**)realloc(lePoly->listePointPoly, --(lePoly->nbPoint)*sizeof(pointp*));
                        i--;
                }
        }
}
```

On passe en argument l'adresse du polygone englobant et l'adresse du polygone interdit. On parcourt la liste de point contenu dans la structure du polygone englobant. Nous testons ensuite via isInside() si le point à la position « i » de la liste se trouve dans le polygone interdit. S'il se trouve dedans, on libère l'adresse à la position « i » qui est devenu inutile, on le remplace par le dernier point, nous ré-allouons le pointeur avec une taille de point en moins et on réduit « i » afin de tester le point qui a pris la position i.

3.4. Génération de la grille

La génération de la grille consiste à générer le fichier tspp.dat.

Attention : Pour modifier les distances pour considérer 2 points comme voisins ou couvert, il faut modifier les variables globales « distanceCouverture » et « distanceVoisin » se trouvant au début du fichier « gen_graph.c »

3.4.1 Génération du graphe

Afin de générer le graphe, nous avons créé une fonction genererGraphe() . L'objectif de cette fonction est de tester l'écart entre les points et si l'écart est inférieur à une des 2 valeurs correspondant à la distance entre voisins ou la distance de couverture, alors on remplit un fichier avec ces informations. Pour tester la distance entre 2 points, nous faisons la distance euclidienne et nous la comparons à une valeur fixe.

Ne pouvant pas mettre une seule capture d'écran sans que cela ne soit illisible, nous allons couper cette fonction en plusieurs partie afin d'expliquer celle-ci.

```
void genererGraphe(liste_polygone* lesPolygones, polygone* lePolygone){
        bool accesDirect;
        float distance;

        // Création du fichier tspp.dat en supprimant une ancienne version si le fichier existe déjà
        FILE * fichier = fopen("tspp.dat", "w");

        fprintf(fichier ,"alpha=0.05;\n");
        fprintf(fichier ,"n=%d;\n",lePolygone->nbPoint);
```

La fonction prend en paramètre la liste de polygones tirée de l'export kml pour tester les obstacles et le polygone pour avoir la liste de points.

Pour commencer, nous ouvrons le fichier « tspp.dat » en mode écriture uniquement, ce qui permet d'effacer l'ancien contenu automatiquement. Nous écrivons ensuite la valeur alpha et le nombre de points du polygone dans le fichier.

```
// Mise en place des voisins
fprintf(fichier ,"Edges= {\n");
for(int i = 0; i < lePolygone->nbPoint; i++){
        for(int j = i + 1; j < lePolygone->nbPoint; j++){
                distance = sqrtf(pow((lePolygone->listePointPoly[i]->x - lePolygone->listePointPoly[j]->x),2)
                                + pow((lePolygone->listePointPoly[i]->y - lePolygone->listePointPoly[j]->y),2));
                accesDirect = !intersection(lePolygone, i, j, lesPolygones);
                if(distance <= distanceVoisin && accesDirect){
                        fprintf(fichier, "< %d %d >,\n", lePolygone->listePointPoly[i]->id, lePolygone->listePointPoly[j]->id);
                }
        }
}
// Retire la dernière virgule
fseek(fichier, -2, SEEK_END);
fprintf(fichier, "\n};\n");
```

Ensuite, nous ajoutons les voisins dans le fichier. Afin de les ajouter, nous allons faire deux boucles, la première va parcourir l'ensemble des points du polygone, le second va parcourir le reste de la liste. Nous allons ensuite faire la distance euclidienne entre les 2 points d'indice « i » et « j ». On va ensuite vérifier s'il y a un accès direct en testant une intersection entre le segment formé par les 2 points et la liste de polygones. S'il y a un accès direct et que la distance est inférieur à la variable global correspondante à la distance entre voisins, alors on écrit dans le fichier les 2 points. Enfin, on retire la dernière virgule après avoir finis la boucle et fermons la partie correspondant aux voisins.

```
// Mise en place des points couverts
fprintf(fichier, "CoveredBy = [\n");
for(int i = 0; i < lePolygone->nbPoint; i++){
        fprintf(fichier, "{ ");
        for(int j = 0; j< lePolygone->nbPoint; j++){
                distance = sqrtf(pow((lePolygone->listePointPoly[i]->x - lePolygone->listePointPoly[j]->x),2)
                                + pow((lePolygone->listePointPoly[i]->y - lePolygone->listePointPoly[j]->y),2));
                accesDirect = !intersection(lePolygone, i, j, lesPolygones);
                if(distance <= distanceCouverture && accesDirect){
                        fprintf(fichier, "%d, ", lePolygone->listePointPoly[j]->id);
                }
        }
        // Retire la dernière virgule
        fseek(fichier, -2, SEEK_END);
        fprintf(fichier, " },\n");
}
// Retire la dernière virgule
fseek(fichier, -2, SEEK_END);
fprintf(fichier, "\n];\n");
fclose(fichier);
```

On réitère l'opération pour la couverture, mais cette fois, nous comparons par rapport à la distance de couverture, et on écrit différemment dans le fichier. On retire de nouveau la virgule, et on ferme le fichier après avoir fini.

3.4.2 Vérifier si il n’y a pas d’obstacle

Afin d’éviter des voisins qui n’aurait pas lieu d’être à cause d’un obstacle, nous avons mis en place une fonction « intersection » qui va vérifier qu’il n’y a pas d’intersection avec un des polygones de la liste de l’export kml de départ. Pour ce faire, nous allons parcourir la liste de polygones est utiliser la fonction doIntersect() présenté dans la génération de point pour vérifier.

```
bool intersection(polygone *lePolygone, int i, int j, liste_polygone* polygones){
	liste_polygone* lesPolygones = polygones;
	pointp* sommet;
	// Parcours de l'ensemble des polygones contenus dans la liste (Premier polygone vide)
	while(lesPolygones->next != NULL){
		lesPolygones = lesPolygones->next;
		sommet = lesPolygones->polygone;
		// pour chaque arrête [sommet - sommet->next] du polygone
		do{
			if(doIntersect(lePolygone->listePointPoly[i], lePolygone->listePointPoly[j], sommet, sommet->next)){
				return true;
			}
			sommet = sommet->next;
		}while(sommet != lesPolygones->polygone);
	}
	return false;
}
```

En argument, il y a le polygone englobant, les 2 indices de la liste et la liste de polygones. On récupère l’adresse de la liste de polygones afin de pouvoir la parcourir sans modifier l’adresse de départ. On boucle tant qu’il y a un autre polygone. On récupère ensuite un sommet et on boucle sur l’ensemble des arêtes du polygone en cours. Nous testons ensuite s’il y a une intersection entre l’arête en cour et le segment formé des points donnés par les indices. S’il y a une intersection, alors on retourne True directement, sinon on continue de parcourir. Si aucune intersection est trouvée, alors la fonction retourne False.

3.5. Sortie

3.5.1 Format PDF

Dans cette application il est possible de produire deux pdf. L'un contenant le problème initial (polygones et grille) et un autre contenant le graphe résolu avec les différents états des points de la grille et le chemin solution du problème. Ces sorties sont nommées « grille_out_probleme_.pdf » et « grille_out_resultat.pdf » par défault. Pour cette partie nous avons utilisé DISLIN, une librairie permettant de produire des dessins scientifiques.

/ INSERER LES PDF DE SORTIE*/*

Les fonctions prennent en paramètre ce qu'elle doivent afficher : la liste de polygones, la grille non résolue pour le premier affichage, et les listes de points solution, vus et non-vus pour le second. Elles commencent par une initialisation qui définit le type de sortie (ici en pdf) et le nom du ficher, puis initialise DISLIN et place les titres et noms des axes. Les polygones et les points vont êtres dessinés dans un plan orthonormal. On va déterminer les extremums en x et y des points des polygones grâce à la fonction def_bornes pour placer ce plan. On définit un pas égal au dixième de la différence des extremums d'un axe. Ce pas va servir à la graduation et à ce que les points aux extremums ne se superposent pas aux axes. On génère maintenant le graphique.

Pour dessiner un polygone on rentre les coordonnées de tout ses points dans des tableaux x et y, en suite on utilise DISLIN curves pour relier les points par des droites et tracer le polygone. Pour un point, on dessine un symbole grâce à ses coordonnées et un entier qui définit sa forme avec DISLIN rlsymb. Avec un simple parcours on peut aisément passer du dessin de polygone/point au dessin de liste. Pour les deux sorties on commence par tracer les polygones. Le polygone englobant en bleu et les autres en noir. En suite on dessine les différentes listes de points de chacun des problèmes avec les couleurs voulues. Pour la sortie du résultat fait un autre polygone qui sera le chemin solution.

3.5.2 Interface graphique

3.6. Interface graphique

4. Architecture logicielle

Voici un schéma de l'architecture logicielle