Sorbonne Université

Projet

4M016

Optimisation d'un trajet sur un graphe

Auteurs :
Nicolas Quero
Anas Boudi

6 janvier 2019

Table des matières

1 Introduction

On imagine que l'on entreprend un voyage à bord d'un véhicule tout terrain ; d'un point A on cherche à relier un point B. Pour cela on est amenés à chercher le meilleur chemin c'est-à-dire celui qui minimisera nos efforts. Est-il possible de construire un programme qui trouve le meilleur chemin pour un relief donné ?

Le but du projet est de trouver le meilleur chemin qui permet de relier deux points d'une surface munie d'un relief. On considère que le meilleur chemin correspond au chemin qui minimise la distance à parcourir et les pentes (positives ou négatives) rencontrées au cours du trajet.

Pour ce faire, nous allons discrétiser l'espace et le modéliser par un graphe. Ensuite, l'algorithme de Dijkstra nous permettra de trouver le chemin le moins coûteux pour relier deux points de ce graphe. La dernière étape du projet consistera à tracer en trois dimension le chemin obtenu.

2 Construction d'un graphe

2.1 Construction théorique

On considère un véhicule se déplaçant sur un terrain $\Omega = [0,1] \times [0,1]$, on ne perd pas de généralité en se plaçant sur Ω car il est en bijection avec $\mathbb{R}^2$ et on définit une fonction $f : \Omega \longmapsto \mathbb{R}^2$ qui représente le relief du terrain. À partir de ces éléments nous allons construire un graphe.

D'abord nous allons discrétiser l'espace en $N_x \times N_y$ points, on définit aussi les pas $\Delta x = \frac{1}{N_x - 1}$ et $\Delta y = \frac{1}{N_y - 1}$. Pour la suite du projet, on se restreint au cas $N_x = N_y$. Nous avons désormais les points du graphe, il reste à en définir les arêtes.

On définit une distance δ telle que pour tout couple de points $(X_n, X_m)_{0 \leqslant n,m \leqslant N_x N_y - 1}$ espacés d'une distance inférieure à δ on trace un arête.

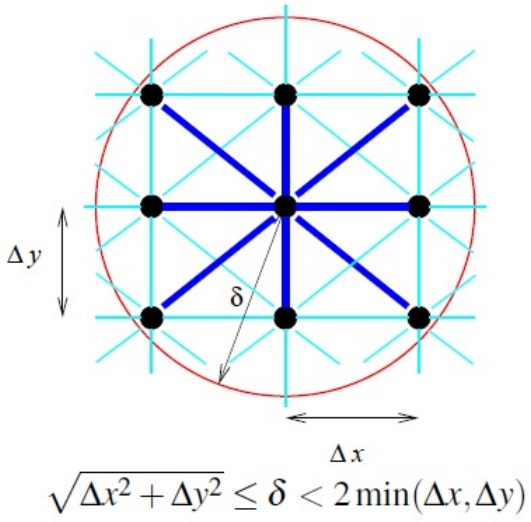

FIGURE 1 – *Construction des arêtes à l'aide de* δ

Pour ce projet, on autorisera toujours le déplacement en diagonale donc on pose $\sqrt{\Delta x^2 + \Delta y^2} \leqslant \delta < min(\Delta x, \Delta y)$.

Enfin, on va associer à chaque arête entre un couple $(X_n, X_m)_{0 \leqslant n,m \leqslant N_x N_y - 1}$ un coût $c_{X_n X_m}$ qui dépend de la longueur de l'arête mais aussi de la pente du relief défini par la fonction f.

$$c_{X_n X_m} = \alpha \frac{|X_n - X|}{max(\Delta x, \Delta y)} + (1-\alpha)\frac{|f(X_m) - f(X_n)|}{|X_n - X_m|}, 0 \leqslant \alpha \leqslant 1$$

2.2 Construction informatique

On commence par définir une fonction "grille" qui construit un tableau de N_x lignes et N_y colonnes. Chaque case du tableau correspond aux coordonnées d'un sommet, on utilise la classe R2 pour stocker les coordonnées.

```
R2**    grille(int const N_x,int const N_y)
{
        int n;//compteur
        int m;//compteur
        double dx;//pas
        double dy;//pas
```

```
        R2 **tab(0);//Tableau des sommets

        n = 0;
        tab = new R2 *[N_x];//Alloue N_x lignes
        while(n < N_x)
        {
                tab[n] = new R2[N_y];//Alloue N_y colonne à chaque ligne
                n++;
        }

        n = 0;
        dx = 1.0/(N_x - 1.0);// 1.0 obligatoire
        dy = 1.0/(N_y - 1.0);//On définit les pas.

//On remplit le tableau avec les coordonnées
        while(n < N_x)
        {
                m = 0;
                while(m < N_y)
                {
                                tab[n][m] = R2(n*dx,m*dy);//Coordonées du sommet
                        m++;
                }
                n++;
        }
        return(tab);
}
```

L'étape suivante est de définir les arêtes du graphe. Pour cela on va utiliser une matrice d'adjacence.

D'abord on associe à chaque sommet un unique nombre qu'on appelera "clef". On donne les clefs par ordre croissant en partant d'en haut à gauche du tableau des sommets. Pour un exemple simple cela donne :

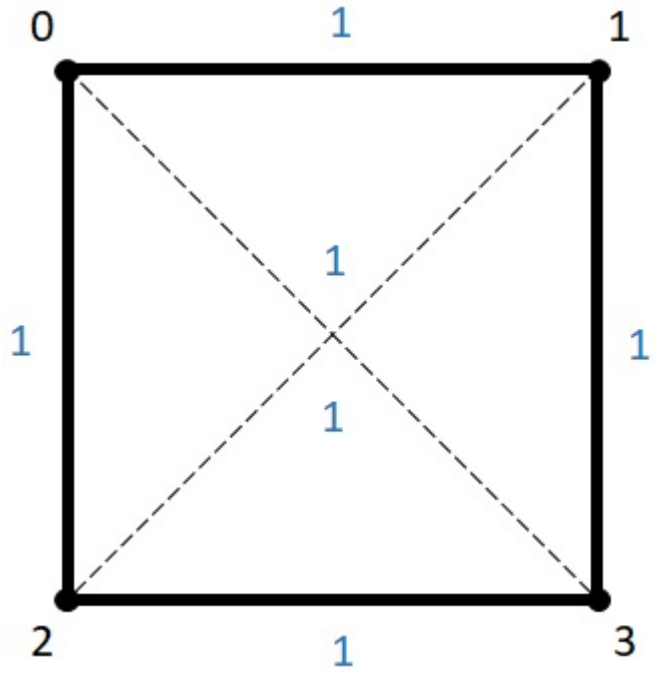

FIGURE 2 – Clefs données pour 4 sommets, tous les coûts sont égaux à 1 (en bleu)

Ainsi, lorsqu'on veut stocker les arêtes et leur coût on a juste à définir une matrice carrée de taille $N = N_x \times N_y$ qu'on appellera "matrix". Pour tout $(n, m) \in N^2$, $matrix[n][m]$ prend la valeur 0 s'il n'y a pas d'arête et la valeur du coût entre les sommets n et m s'il y a une arête : c'est ce qu'on appelle une matrice d'adjacence. Pour notre exemple on a :

clefs des sommets	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

FIGURE 3 – Matrice d'adjacence associée à la figure 2

On remarque que la matrice est symétrique, cela nous sera utile plus tard.

On doit maintenant définir un moyen de savoir si deux clefs n et m de la matrice d'adjacence pointent sur deux sommets de distance inférieure au δ défini plus haut, ceci afin de savoir s'il y a une arête entre ces deux sommets.

Soit un tableau de sommets "tab" avec N_x lignes et N_y colonnes. On défnit les clefs de la manière suivante : pour tout $(i, j) \in N_x \times N_y$ le sommet $tab[i][j]$ prend la clef $n = i \times N_y + j$. Ainsi à partir de (i, j) on connaît la clef n du sommet, de même en connaissant n on connaît (i, j) puisque $n \equiv j \pmod{N_y}$ et $i = \frac{n-j}{N_y}$.

Donc pour tout couple de clefs (n, m) on peut obtenir leurs coordonnées dans le tableau des sommets et vérifier que leur distance est inférieure à notre δ.

On programme donc une fonction qui va nous permettre de construire une matrice d'adjacence à partir des sommets créés dans "grille", on l'appelle "adjacency".

2.3 Initialisation

Voici comment on commence :

```
    double**        adjacency(R2** const& tab, int N_x, int N_y)
{
        int N;Nombre de sommets
        int b;//Permet d'économiser des opérations par tour
        int c;//Permet d'économiser des opérations par tour
```

```
        int d;//Permet d'économiser des opérations par tour
        int e;//Permet d'économiser des opérations par tour
        double dx;//Pas
        double dy;//Pas
        double delta;//Limite au-delà de laquelle on ne trace pas l'arête
        double **matrix(0);//Matrice d'adjacence
        int n;//Compteur
        int m;//Compteur

//Chaque sommet porte un numéro qui est donné dans l'ordre croissant
//en partant d'en haut à gauche de la grille
//Ainsi, le sommet tab[i][j] porte le numéro i*N_y +j

        N = N_x * N_y;
        dx = 1.0/(N_x - 1.0);
        dy = 1.0/(N_y - 1.0);
        delta = 2*fmin(dx,dy);
        matrix = new double *[N];

}
        n = -1;
        while(++n < N)
                matrix[n] = new double[N];

//On initialise la matrice d'adjacence à 0
        n = -1;
        while(++n < N)
        {
            m = -1;
            while(++m < N)
                matrix[n][m] = 0;
        }
```

Notons d'abord que l'on a donné comme paramètres des "const &" puisque le but de cet algorithme est de construire la matrice d'adjacence : nous ne modifierons pas les paramètres. La référence nous permet d'optimiser l'utilisation de la mémoire. Pour l'initialisation, on défini des variables "int" qui nous permettront de gagner quelques opérations plus tard. Notre matrice

d'adjacence est bien de taille $N = N_x \times N_y$, on l'initialise à 0 car au début aucune arête n'est tracée entre les sommets.

2.4 Remplissage de la matrice

Dans cette boucle on va remplir la matrice d'adjacence. n et m correspondent à des clefs de notre tableau, grâce à elles on va pouvoir parcourir toute la matrice.

```
{
        n = 0;
        while(n < N)
        {
//On ne remet pas forcément m à 0 car on ferait des comparaisons inutiles
                m = fmax(0,n-N_y-1);

//La matrice est symétrique on arrête donc m à n + 1
                while(m < n + 1)
                {
                    c = n%N_y;
                    b = (n-c)/N_y;
                    e = m%N_y;
                    d = (m-e)/N_y;

                    if(dist(tab[b][c],tab[d][e]) < delta && n!= m)
                            matrix[n][m] = 1;//Coût défini à 1 pour l'exemple
                    matrix[m][n] = matrix[n][m];

//On saute les comparaisons inutiles sauf si N_y == 2
//car cela conduit à une boucle infinie
                    if(m == n - N_y + 1 && N_y != 2)
                              m = m + N_y - 3;
                    m++;
                }
                n++;
        }
        return(matrix);
}
```

Puisque n et m correspondent à des clefs de notre tableau de sommet, il existe deux uniques couples (b,c),(d,e) tels que $n = b \times N_y + c$ et $m = d \times N_y + e$ donc $c \equiv n \pmod{N}_y$ et $b = (n - c)/N_y$, on fait de même pour m. Si $dist(tab[b][c], tab[d][e]) < \delta$ et si les sommets n et m sont différents on trace l'arête.

Dans l'algorithme, certaines comparaisons ont été évitées par souci d'optimisation. Voici un petit schéma qui explique pourquoi :

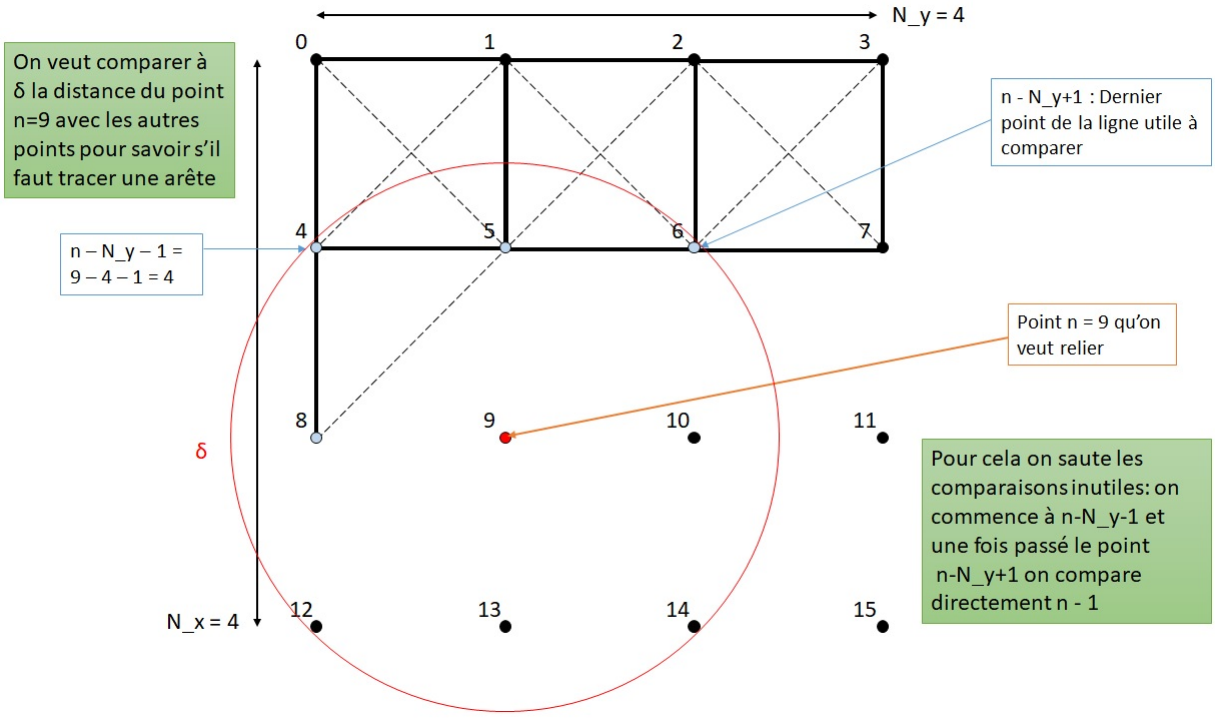

FIGURE 4 – Schéma explicatif des comparaisons utiles

Nous avons désormais à disposition les sommets avec leurs coordonnées ainsi que leurs arêtes et leurs coûts. Nous avons donc fini la construction de notre graphe, nous pouvons commencer la recherche du plus court chemin entre deux points grâce à l'algorithme de Dijkstra.

3 Recherche du chemin le moins coûteux entre deux points

3.1 Présentation de l'algorithme de Dijkstra

$I = N_x \times N_y$ désigne le nombre total de sommets du maillage.

L'algorithme de Dijkstra permet de trouver le chemin le plus court entre deux points d'un graphe. Pour ce faire, on va prendre la matrice d'adjacence en entrée ainsi que notre point de départ. On déclare un tableau *Poids* de taille I contenant le poids nécessaire pour aller du point de départ à chaque sommet. On déclare aussi un tableau *Adj* de taille I contenant l'antécédant le plus proche de chaque sommet.

L'idée est de mettre tous les poids au départ à l'infini (sauf celui du point de départ qui vaut 0) puis de regarder quel est le sommet de poids minimal nommé min_d puis de modifier le poids des sommets qui lui sont accessibles s' il est moins couteux de passer par min_d. Le poids du sommet devient dans ce cas le poids de min_d additionné au coût de l'arête reliant les deux sommets que l'on trouve dans la matrice d'adjacence et min_d devient alors l'antécédent du sommet dont le poids a été modifié. On fait cela en boucle pour au final obtenir un tableau d'antécédents que l'on peut parcourir en partant du point d'arrivée désiré pour récursivement arriver jusqu'au point de départ. On a alors le chemin le plus court.

Le code complet est disponible en annexe, on analysera dans la suite le code petit à petit.

3.2 Mise en oeuvre de l'algorithme

3.2.1 Initialisation

```
int* dijkstra(double** const &Mat_a, int const& begin_i)
{
    double Poids[I];
    bool Parcourus[I];
    int* Adj = new int[I];
    for(int i=0; i<I; i++)
```

```
    {
        Poids[i]=DBL_MAX; Parcourus[i]=false; Adj[i]=-1;
    }
    Poids[begin_i]=0.;
}
```

On prend en entrée la matrice d'adjacence Mat_a et l'indice du point de départ $begin_i$. On crée les tableaux $Poids$ et Adj, ainsi qu'un tableau de booléens $Parcourus$ de taille I. Pour k une clef, $Parcourus[k]$ vaudra $true$ si l'on est déjà passé par k et $false$ sinon.

A la ligne 6 on instancie $Poids$ à l'infini pour tous les sommets sauf à l'indice $begin_i$ car le poids pour aller de $begin_i$ à $begin_i$ est évidemment de 0. $Parcourus$ est instancié à false pour tous les sommets car on n'est encore passé par aucun sommet, et Adj est instancié à -1 pour faire comprendre qu'aucun point n'a encore d'adjacent.

3.2.2 Recherche du sommet de poids minimal

```
for(int i=0; i!=n_arrivee; i++)
{
    int min_d = Trouve_min(Poids, Parcourus);
    Parcourus[min_d]=true;
}
```

On commence à parcourir nos sommets en appelant à chaque tour le sommet de poids minimal et qui n'a pas encore été parcouru. Pour cela, on fait appel à la fonction $Trouve_{min}(Poids, Parcourus)$.

```
int Trouve_min(double* Poids, bool* Parcourus)
{
    double min = DBL_MAX;
    int sommet_i;
    for(int i=0; i<I; i++)
    {
        if(Parcourus[i]==false && Poids[i]<=min)
        {
            sommet_i=i; min=Poids[i];
```

```
        }
    }
    return sommet_i;
}
```

Cette fonction va déclarer un minimum à l'infini et l'indice du minimum associé. Ensuite elle va parcourir tous les sommets et si le poids d'un sommet est inférieur au minimum et que le sommet n'a pas été parcouru, on modifie le minimum et l'indice en conséquence. Après avoir parcouru tous les sommets on retourne l'indice du sommet minimal que l'on notera min_d. On indique alors dans le programme principal que l'on est passé par min_d.

3.2.3 Condition de changement du poids d'un sommet

On va maintenant regarder si le fait de passer par min_d ne rend pas le poids plus faible pour chaque sommet d'indice noté k.

```
for(int k=0; k<I; k++)
{
    if(!Parcourus[k] && Mat_a[min_d][k]!=0 && Poids[min_d]!=DBL_MAX
    && Poids[min_d]+Mat_a[min_d][k] < Poids[k])
    {
        Poids[k]=Poids[min_d]+Mat_a[min_d][k];
        Adj[k]=min_d;
    }
    if(min_d-N_y-1>0 && k < min_d - N_y - 1) // On saute les comparaisons inutiles
        k = min_d - N_y - 2;
    else if(k > min_d + N_y + 1)
        k = I;
    else if(k%N_y == min_d%N_y + 1)
        k = k + N_y - 3;
}
```

Pour ce faire, si :
- on n'est pas déjà passé par k (cela voudrait dire revenir en arrière)
- que l'arête existe (donc que $Mat_a[min_d][k]! = 0$, sinon l'arête n'existe pas)
- que $Poids[min_d]$ n'est pas infini (sinon avec l'arithmétique modulaire des

ordinateurs on reviendra à 0 en additionnant les poids)
- et que $Poids[min_d] + Mat_a[min_d][k] < Poids[k]$ (cela veut dire que passer par min_d est moins coûteux, $Mat_a[min_d][k]$ contenant le poids de l'arète $[min_d][k]$)

Alors on change le poids du sommet k qui est plus faible en passant par min_d, et change son antécédant en min_d car il faut passer en dernier par min_d dans le chemin au poids moindre reliant $begin_i$ à k. (le if et les deux else en-dessous sautent simplement les comparaisons inutiles de la même manière que dans adjacency.cpp).

Après avoir parcouru tous les k possibles, on calcule un nouveau min_d, on le note comme parcouru et on réitère les opérations précédentes. On calcule en tout au plus I fois min_d. On trouvera toujours un min_d car notre graphe est connexe, donc le premier tour de boucle changera le poids des points connectés au point de départ, et le deuxième changera le poids des points connectés au minimum des points connectés au point de départ etc. jusqu'à tomber sur $n_arrivee$. De plus, on a bien min_d calculé au plus une seule fois par point car un point parcouru n'est plus pris en compte.

Une fois les deux boucles parcourues, on a bien regardé le point d'arrivée et le chemin minimal menant jusqu'à lui. On retourne alors le tableau d'antécédents.

3.3 Valeur du coût dans la Matrice d'Adjacence

Le coût est défini dans la matrice d'adjacence. Pour modifier aisément la fonction associée au coût, la fonction prise en compte dans la Matrice d'Adjacence est la fonction $f(x, y)$ dans "points_gnu.cpp", on n'a alors plus qu'à modifier le return de la fonction f dans "points_gnu.cpp". Par exemple si l'on souhaite utiliser la fonction $crat$ il suffit de changer le return de la fonction f en $return(crat(x, y))$; ou $return(gauss(x, y))$; pour utiliser la fonction $gauss$.

Par défaut les points (x,y) ont des valeurs allant de (0,0) à (1,1). Si les valeurs qui nous intéressent sont situées en dehors de cet intervalle il faut réajuster les valeurs de x et y avant le return comme dans la fonction $crat(x, y)$ (tout intervalle réel fermé borné est en bijection avec [0,1] donc il n'y a pas de problème).

3.4 Ecrire les données dans un fichier grâce aux antécédents

Dans le main on récupère maintenant le tableau d'antécédents. On cherche désormais à afficher le plus court chemin allant de n_{depart} à $n_{arrivee}$ (le départ et l'arrivée sont à définir par l'utilisateur dans le début de *dijkstra.cpp*, voir annexe). Pour cela on définit un tableau $t_{ant}[t]$ où t est le nombre de points dans le chemin(t est calculé dans le main, voir annexe), on remplit ensuite ce tableau avec les antécédents de chaque point.

Par exemple si le chemin le plus court pour aller du point 1 au point 7 est de passé par 1 puis 2 puis 4 puis 7, alors l'antécédent de 7 est 4 (ou $t_{ant}[7] = 4$), l'antécédent de 4 est 2 (ou $t_{ant}[4] = 2$) et l'antécédent de 2 est 1. On est retombé sur le point de départ donc on s'arrête.

```
1     n = n_arrivee;
2     t_ant[0] =n;
3     int j = 1;
4
5     while(n!=n_depart)
6     {
7         n = Adj[n];
8         t_ant[j] = n;
9         j++;
10    }
```

Une fois le tableau rempli, on ouvre "Maillage.txt" et "Chemin.txt" en écriture. Dans "Maillage.txt" on veut écrire les coordonnées de tous les points, celles-ci sont stockées dans notre grille G créée dans la première ligne du main. Comme les indices que l'on considère vont de 0 à I-1, il faut les réajuster de manière à retrouver x et y les coordonnées des points de notre grille.

Les coordonnées d'une clef i de notre grille est $(x, y) = G[\frac{i-i\%N_x}{N_x}][i\%N_x]$ comme expliqué plus haut. On écrit alors ces deux coordonnées séparées d'une tabulation dans le fichier et pour la coordonnée z, il suffit d'ajouter $f(x, y)$ qui est la valeur de z. On répète cela pour tous les points, séparés d'un saut à la ligne et on a alors tous les points du maillage dans "Maillage.txt".

```
11 ofstream fic1("Maillage.txt", ios::out | ios::trunc);
12 if(fic1)
13 {
14     cout << "MAILLAGE OUVERT" << endl;
15     double x, y;
16     for(int i = 0; i<I; i++)
17     {
18         x = G[i/N_x][i%N_x][0];
19         y = G[i/N_x][i%N_x][1];
20         fic1 << x << '\t' << y << '\t' << f(x,y) << endl;
21     }
22 }
23 else
24     {cout << "PB MAILLAGE" << endl;}
25
```

Pour le chemin il suffit de faire la même chose dans le fichier "Chemin.txt" mais uniquement avec les points du tableau t_{ant}.

On peut désormais afficher le résultat dans gnuplot à l'aide de la commande splot "Maillage.txt" with dots, "Chemin.txt" with lines pal pour bien voir le chemin.

```
26
27     ofstream fic3("trace.txt", ios::out | ios::trunc);
28
29     if(fic3)
30     {
31         fic3 << "splot \"Maillage.txt\" with dots,
32         \"Chemin.txt\" with lines pal" << endl;
33     }
34     fic3.close();
35
36     system("gnuplot --persist trace.txt");
37
38
```

Certains résultats jugés pertinents ont été sauvegardés dans le dossier Résultats du projet. Pour afficher vos propres résultats avec le code il faut modifier $N_x, n_{depart}, n_{arrivee}$ et a dans le début de "dijkstra.cpp" et la fonction du coût f dans "points_gnu.cpp". Attention, comme annoncé au début de ce mémoire, l'algorithme ne fonctionne correctement que lorsque $N_x = Ny$.

3.5 Résultats sur Gnuplot

Pour vérifier que le coefficient du coût a est bien pris en compte on a ci-dessous deux résultats, le premier où $a = 0$ (donc on ne prend en compte que le coût de la fonction et pas la distance entre deux points, et le deuxième où $a = 1$ (donc seule la distance est prise en compte).

Les images étant en 3D, il sera plus intéressant de les visualiser sur gnuplot.

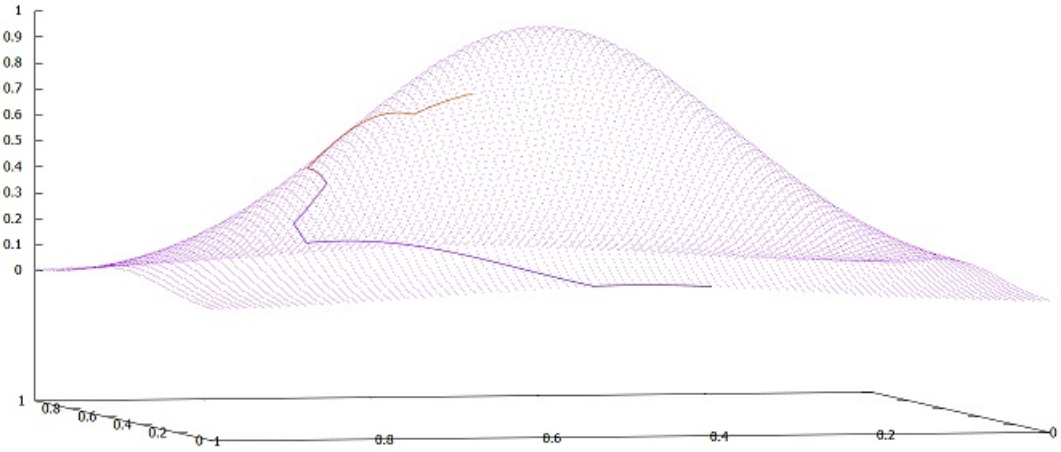

FIGURE 5 – Fonction *gauss* avec $a = 0$

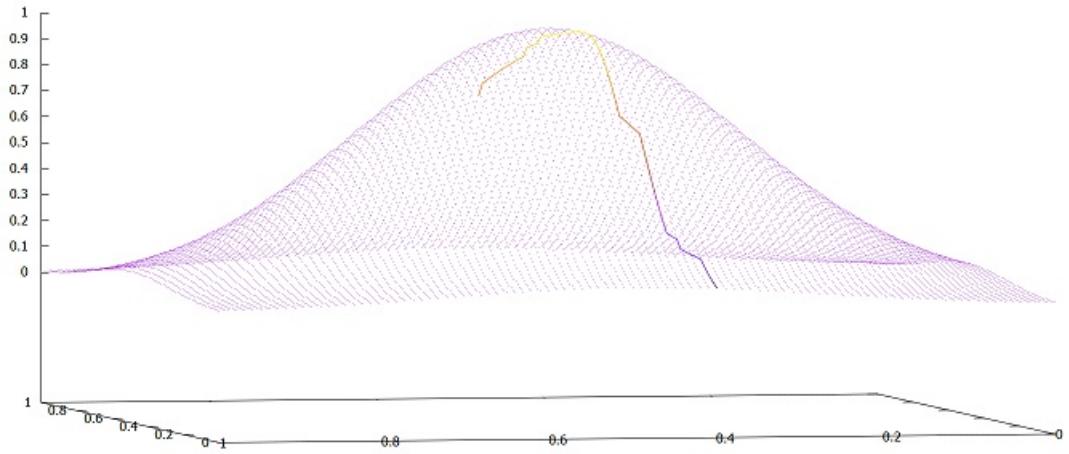

FIGURE 6 – Fonction *gauss* avec $a = 1$

On obtient bien des résultats cohérents. On peut regarder un autre exemple utilisant la fonction *crat*.

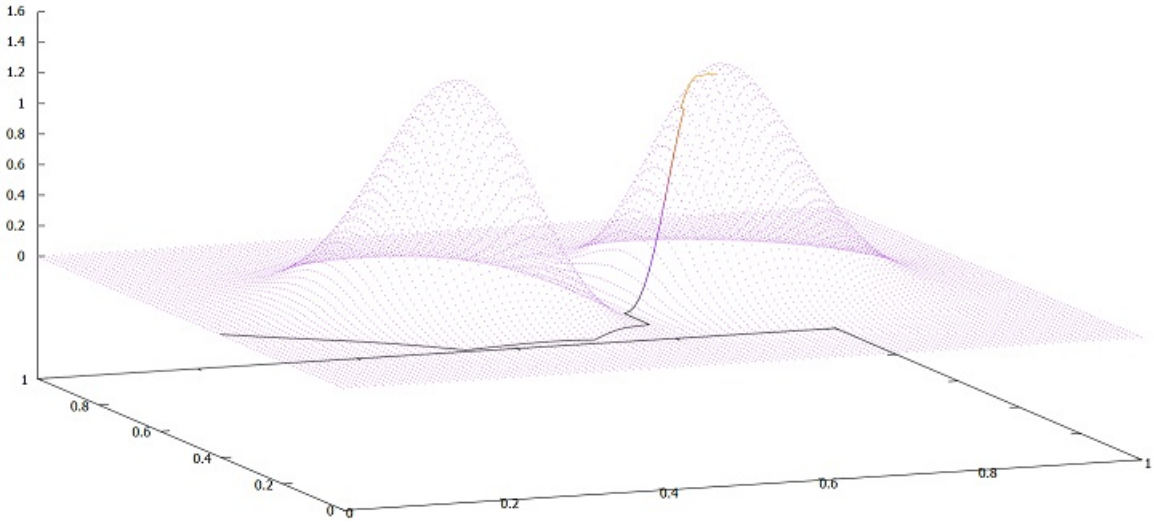

FIGURE 7 – Fonction *crat* avec $a = 0,5$

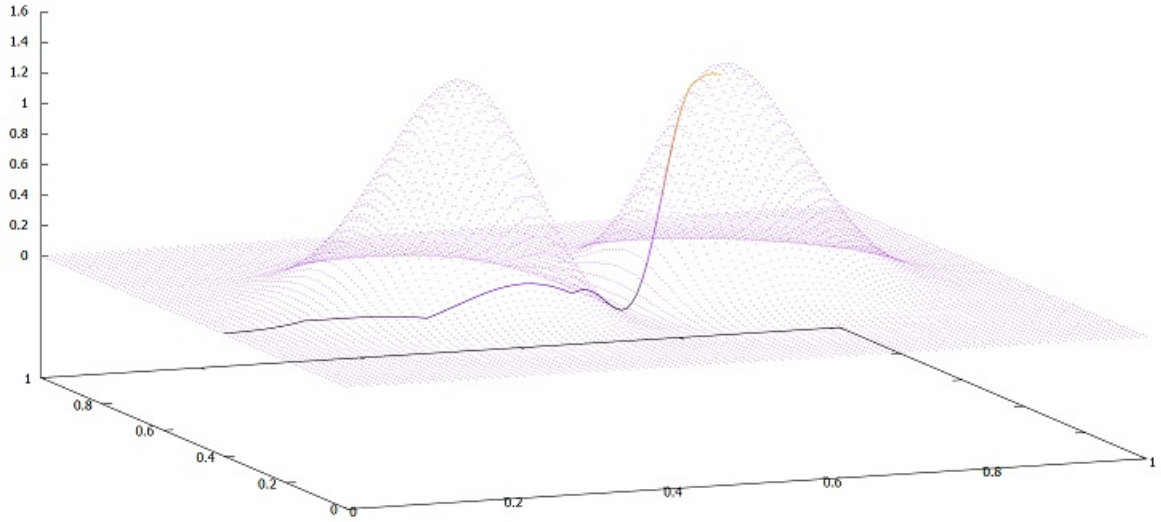

FIGURE 8 – Fonction *crat* avec $a = 0,9$

4 Conclusion

4.1 Complexité du code

La complexité de notre projet repose sur deux parties principales : la construction de la matrice d'adjacence et l'algorithme de Dijkstra.

4.1.1 Construction de la matrice d'adjacence

La matrice d'adjacence est une matrice en $I^2 = N_x^2 \times N_y^2$. Pour générer cette matrice on réalise pour chaque case un nombre constant d'opération. Le programme a donc une complexité en $O(I^2)$ où N est le nombre de points de la matrice.

Aussi, il faut prendre en compte la taille de l'allocation de mémoire qui peut être très conséquente. Avec $N_x = N_y = 100$, on a $I = 10^4$ et on obtient alors une matrice d'adjacence de taille 10^8. La taille de l'allocation d'un pointeur étant de 8 octets, notre matrice a finalement une taille de 8×10^8 octets soit $0,8$ Go de RAM utilisée pour notre matrice d'adjacence. En passant N_x et N_y à 200 la matrice demandera $3,2$ Go de RAM ce qui est trop gourmand pour de nombreuses machines. On se limitera alors à des tableaux

100 par 100 pour éviter des problèmes d'allocation de mémoire (on évite naturellement d'écrire plusieurs Go de données temporaires sur la mémoire disque, d'autant plus que la vitesse d'écriture de la RAM est bien plus élevée).

4.1.2 Complexité de Dijkstra

Trouver le noeud de poids minimal demande $O(I)$ opérations. On le refait au plus I fois, donc on a fait $O(I^2)$ opérations. On recalcule les poids au plus une fois par arête ce qui rajoute un terme linéaire.

La complexité de l'algorithme de Dijkstra est donc en $O(I^2)$ dans le pire des cas.

4.2 Conclusion

Notre projet permet de trouver le chemin le moins coûteux d'un véhicule qui se déplacerait entre deux points d'un terrain en trois dimensions. Le programme finalement obtenu a été réalisé en utilisant une matrice d'adjacence et l'algorithme de Dijkstra. Enfin, on a pu représenter les chemins les plus courts pour certains exemples à l'aide de gnuplot.

On peut imaginer retrouver l'implémentation de notre projet dans un jeu vidéo en 3D pour y afficher le chemin le plus court menant le personnage incarné par le joueur à son objectif en évitant au mieux les obstacles. Cependant, la complexité en espace mémoire du programme est assez élevée, il ne faudrait donc pas prendre en compte trop de points pour ne pas surcharger la RAM du joueur, ou bien trouver un moyen de contourner ce problème en modifiant dijkstra pour ne pas avoir recours à la matrice d'adjacence. Cela serait peut-être faisable en injectant les conditions de la matrice d'adjacence et le calcul du coût directement dans dijkstra.

5 Sources

5.1 Allocation de mémoire pour un tableau de plusieurs dimensions

$https://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/CPP--Tableaux-2D.html$

5.2 Matrices d'adjacence

$https://www.pairform.fr/doc/1/32/180/web/co/Matrice.html$

5.3 Explications et exemples pour l'algorithme de Dijkstra

$http://yallouz.arie.free.fr/terminale_cours/graphes/graphes.php?page=g3$

$https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra$

$https://www.math.u-bordeaux.fr/\ gstauffe/cours/MSE3211A/MSE3211A\_3.pdf$

$https://openclassrooms.com/fr/courses/1125906-le-pathfinding-avec-dijkstra$

A Annexes

Les commentaires des programmes ont été retirés pour plus de clarté.

A.1 Programme de construction de la matrice d'adjacence

```
39 double** adjacency(R2** const &tab)
40 {
41     int N;
42     int b;
43     int c;
44     int d;
45     int e;
46     double dx;
47     double dy;
48     double delta;
49     double **matrix(0);
50     int n;
51     int m;
52 
53     N = N_x * N_y;
54     dx = 1.0/(N_x - 1.0);
55     dy = 1.0/(N_y - 1.0);
56     delta = 2*fmin(dx,dy);
57     matrix = new double *[N];
58     n = -1;
59 
60     while(++n < N)
61         matrix[n] = new double[N];
62 
63     n = -1;
64     while(++n < N)
65     {
66         m = -1;
67         while(++m < N)
68         matrix[n][m] = 0;
```

```
69      }
70
71      n = 0;
72      while(n < N)
73      {
74          m = fmax(0,n-N_y-1);
75          c = n%N_y;
76          b = (n-c)/N_y;
77
78          while(m < n + 1)
79          {
80
81              e = m%N_y;
82              d = (m-e)/N_y;
83              if(dist(tab[b][c],tab[d][e]) < delta && n!= m)
84                  matrix[n][m] = a*dist(tab[b][c],tab[d][e])/fmax(dx,dy)
85                  +(1-a)*abs(f(tab[b][c][0], tab[b][c][1]) -
86                  f(tab[d][e][0], tab[d][e][1]))/
87                  dist(tab[b][c],tab[d][e]);
88
89              matrix[m][n] = matrix[n][m];
90
91              if(m == n - N_y + 1 && N_y != 2)
92                  m = m + N_y - 3;
93              m++;
94          }
95          n++;
96      return(matrix);
97  }
```

A.2 Algorithme de Dijkstra

```
98  // *** Valeurs à modifier en fonction du résultat souhaité *** \\
99
100 int N_x = 100; // Notre graphe est de taille N_x*N_y (N_x=N_y)
101
102 int n_depart = 40; // Point de départ
103 int n_arrivee = 6754; // Point d'arrivée
104
105 double a = 0.5; // a est le coefficient du coût
106
107 // ********************************************************** \\
108
109 int* dijkstra(double** const &Mat_a, int const& begin_i)
110 {
111     double Poids[I];
112     bool Parcourus[I];
113     int* Adj = new int[I];
114     for(int i=0; i<I; i++)
115     {
116         Poids[i]=DBL_MAX; Parcourus[i]=false; Adj[i]=-1;
117     }
118     Poids[begin_i]=0.;
119
120     int min_d;
121
122     for(int i=0; min_d!=n_arrivee; i++)
123     {
124         min_d = Trouve_min(Poids, Parcourus);
125         Parcourus[min_d]=true;
126
127         for(int k=0; k<I; k++)
128         {
129             if(!Parcourus[k] && Mat_a[min_d][k]!=0 && Poids[min_d]!=DBL_MAX
130             && Poids[min_d]+Mat_a[min_d][k] < Poids[k])
131             {
132                 Poids[k]=Poids[min_d]+Mat_a[min_d][k];
133                 Adj[k]=min_d;
```

```
134             }
135             if(min_d-N_y-1>0 && k < min_d - N_y - 1)
136                 k = min_d - N_y - 1;
137             else if(k > min_d + N_y + 1)
138                 k = I;
139             else if(k%N_y == min_d%N_y + 1)
140                 k = k + N_y - 3;
141         }
142     }
143     return Adj;
144 }
145
146 int Trouve_min(double* Poids, bool* Parcourus)
147 {
148     double min = DBL_MAX;
149     int sommet_i;
150     for(int i=0; i<I; i++)
151     {
152         if(Parcourus[i]==false && Poids[i]<=min)
153         {
154             sommet_i=i; min=Poids[i];
155         }
156     }
157     return sommet_i;
158 }
```

A.3 points_gnu.cpp

```
159 double crat(double x, double y)
160 {
161     x = (x-0.5)*6;
162     y = (y-0.5)*6;
163     return(2*(pow(x,2)+pow(x,2))*exp(-pow(x,2)-pow(y,2)));
164 }
165
166 double gauss(double x, double y)
167 {
168     return(exp(-10*(pow(x-0.5,2.0)+pow(y-0.5,2.0))));
```

```
169 }
170
171 double f(double x, double y) // Fonction du coût
172 {
173     return(gauss(x,y));
174 }
```

Il faut modifier dans ce fichier la fonction f si l'on souhaite modifier le coût utilisé dans la matrice d'adjacence (f est la fonction du coût).

A.4 main.cpp

```
175 int main(){
176         R2 ** G = grille(N_x,N_y);
177         double** A = adjacency(G);
178         int* Adj = dijkstra(A, n_depart);
179         int n=n_arrivee;
180         int t=1;
181         while(n!=n_depart) // On calcule la taille de t_ant.
182         {
183             n=Adj[n];
184             t++;
185         }
186
187         int t_ant[t];
188         n=n_arrivee;
189         t_ant[0]=n;
190         int j=1;
191         while(n!=n_depart) // On remplit t_ant.
192         {
193             n=Adj[n];
194             t_ant[j]=n;
195             j++;
196         }
197
198         ofstream fic1("Maillage.txt", ios::out | ios::trunc);
199         ofstream fic2("Chemin.txt", ios::out | ios::trunc);
200         if(fic1)
```

```
201         {
202             cout << "MAILLAGE OUVERT" << endl;
203             double x, y;
204             for(int i = 0; i<I; i++)
205             {
206                 x = G[i/N_x][i%N_x][0];
207                 y = G[i/N_x][i%N_x][1];
208                 fic1 << x << '\t' << y << '\t' << f(x,y) << endl;
209             }
210         }
211         else {cout << "PB MAILLAGE" << endl;}
212 
213         if(fic2)
214         {
215             cout << "CHEMIN OUVERT" << endl;
216             double x, y;
217             for(int i=0; i<t; i++)
218             {
219                 cout << t_ant[i] << endl;
220                 x = G[t_ant[i]/N_x][t_ant[i]%N_x][0];
221                 y = G[t_ant[i]/N_x][t_ant[i]%N_x][1];
222                 fic2 << x << '\t' << y << '\t' << f(x,y) << endl;
223             }
224         }
225         else {cout << "PB CHEMIN" << endl;}
226 
227         for(int i = 0; i<I; i++)
228         {
229          delete [] A[i];
230         }
231         for(int i = 0; i<N_x; i++)
232         {
233             delete [] G[i];
234         }
235         delete [] A; delete [] G; delete [] Adj;
236         return 0;
237 }
```