- TP 4. Plus courts chemins. Algorithme de Dijkstra. -

Le but de ce TP est de calculer un arbre des plus courts chemins (en terme de distance euclidienne) issu d'un sommet dans un graphe G dont les sommets sont des points du plan et les arêtes xy sont toutes les paires $\{x, y\}$ de sommets dont la distance est inférieure à une valeur fixée d_{max}.

Langage. Programme en C++. Votre programme pourra contenir :

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <fstream>
#include <cmath>
using namespace std;
typedef struct coord{int abs; int ord;} coord;

const int N=1400;
const int M=(N*(N-1))/2;

void pointRandom(int n,coord point[]);
float distance(coord p1,coord p2);
void voisins(int n,int dmax,coord point[],vector<int> voisin[],int &m);
void voisins2arete(int n,vector<int>voisins[],int arete[][2]);
void affichageGraphique(int n,int m,coord point[],int arete[][2],string name);
bool existe(int n,float dis[],bool traite[],int &x);
void dijkstra(int n,vector<int> voisin[],coord point[],int pere[]);
int construireArbre(int n,int arbre[][2],int pere[]);

int
main()
{
  int n;                            // Le nombre de points.
  cout << "Entrer le nombre de points: ";
  cin >> n;
  int dmax=50;                      // La distance jusqu'a laquelle on relie deux points.
  coord point[N];                   // Les coordonnees des points.
  vector<int> voisin[N];            // Les listes de voisins.
  int arbre[N-1][2];                // Les aretes de l'arbre de Dijkstra.
  int pere[N];                      // La relation de filiation de l'arbre de Dijkstra.
  int m;                            // Le nombre d'aretes
  int arete[M][2];                  // Les aretes du graphe
  return EXIT_SUCCESS;
}
```

Ce début de code est récupérable là : `http://www.lirmm.fr/~montassier/hlin501/tp4.cc`

- Exercice 1 - Création du graphe.
Reprendre la fonction `void pointRandom(int n,coord point[])` du TP2 qui engendre aléatoirement le tableau **point** représentant un ensemble aléatoire de n points dans le plan. Rappelons que **point** est de taille n, l'abscisse du point i (entre 0 et 612) est stockée dans **point**$[i]$.abs et l'ordonnée (entre 0 et 792) est stockée dans **point**$[i]$.ord.

Écrire une fonction `void voisins(int n,int dmax,coord point[],vector<int> voisin[],int &m)` qui, pour tout sommet i, construit la liste **voisin**$[i]$ vérifiant qu'un point $j \neq i$ apparait dans **voisin**$[i]$ si et seulement si la distance euclidienne du point i au point j est au plus égale à d_{max}.

- Exercice 2 - Affichage du graphe.
Écrire `void affichageGraphique(int n,int m,coord point[],int arete[][2],string name)`, inspirée de la fonction d'affichage du TP3, qui permet d'afficher le graphe créé dans l'Exercice 1 à l'aide d'un fichier *Graphe.ps*. Tester sur au moins 300 points.

- Exercice 3 - Arbre de Dijkstra.
Écrire une fonction `void dijkstra(int n,vector<int> voisin[],coord point[],int pere[])` sur le modèle de l'algorithme vu en cours. La racine de l'arbre des plus courts chemins est le sommet 0. En sortie, le tableau **pere** représente l'arbre des plus courts chemins. Ainsi, tout sommet i distinct de la racine et accessible depuis celle-ci vérifie que **pere**$[i]$ est différent de -1, valeur donnée à l'initialisation.

- Exercice 4 - Affichage de l'arbre.
Écrire une fonction `int construireArbre(int n,int arbre[][2],int pere[])` qui remplit le tableau **arbre** avec toutes les arêtes i**pere**$[i]$ et retourne le nombre k de ces arêtes (i.e. le nombre de points accessibles depuis la racine moins un.)

Utiliser la fonction `void affichageGraphique(int n,int m,coord point[],int arete[][2],string name)` pour créer un fichier *Arbre.ps* qui représente l'arbre.

- Exercice 5 - Pour aller plus loin.
Répondre ou améliorer les points suivants :

- Lorsque d_{max} est très grand, que constatez-vous ?
- Les arêtes de l'arbre de Dijkstra peuvent-elles se couper ?
- Essayer des métriques différentes (sup, manhattan).

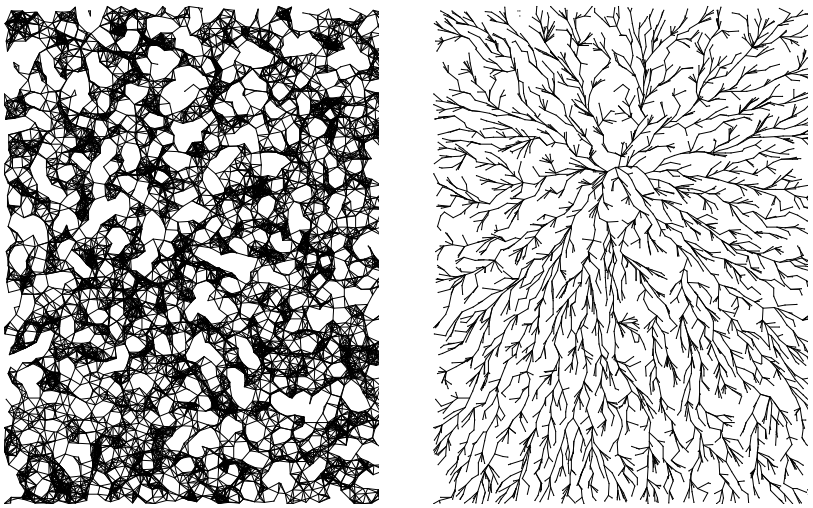

FIGURE 1 – Un exemple d'arbre de plus courts chemins.