- TP 2. Arbre de Kruskal. -

Le but de ce TP est de calculer, pour un ensemble V de points du plan, un arbre couvrant $T = (V, A)$ qui vérifie que la somme des distances des arêtes de A est minimale. Le calcul de cet arbre s'effectue par l'algorithme de Kruskal.

Langage. Programme en C++. Votre programme pourra commencer par :

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <fstream>
using namespace std;
int main()
  {int n;                              // Le nombre de points.
  cout << " Entrer le nombre de points : ";
  cin >> n;
  int m=n*(n-1)/2;                     // Le nombre de paires de points.
  int point[n][2];                     // Les coordonnées des points.
  int comp[n];                         // Cf TP 1.
  int edge[m][3];                      // Les paires de points et leur longueur.
  int arbre[n-1][2];                   // Les arêtes de l'arbre de Kruskal.
  int taille[n];                       // Cf TP 1.
  vector<int> complist[n];             // Cf TP 1.
  }
```

- Exercice 1 - Création d'un ensemble aléatoire V de n sommets dans le plan.

L'ensemble de sommets V est $\{0, \dots, n-1\}$. L'ensemble des positions des éléments de V est stocké dans un tableau **point** de taille $n \times 2$ vérifiant que **point**[i][0] est l'abscisse du point i (entre 0 et 611) et **point**[i][1] est l'ordonnée du point i (entre 0 et 791).

Ecrire une fonction void pointrandom(int n, int point[][2]) qui engendre aléatoirement le tableau **point**.

- Exercice 2 - Création du tableau des distances.

Ecrire une fonction void distances(int n, int m, int point[][2], int edge[][3]) qui engendre le tableau **edge** de taille $m \times 3$ de telle sorte que :

- Pour chaque paire $\{i, j\}$ avec $i < j$, il existe un k qui vérifie **edge**[k][0] $= i$ et **edge**[k][1] $= j$.
- L'entrée **edge**[k][2] est le carré de la distance euclidienne du point i au point j.

- Exercice 3 - Tri du tableau edge.

Ecrire une fonction void tri(int m, int edge[][3]) qui trie le tableau **edge**, selon l'ordre croissant des valeurs de **edge**[k][2].

Le but de ce TP n'étant pas le tri, on pourra se limiter à un simple tri à bulles (tant qu'il existe deux entrées consécutives qui ne sont pas croissantes, on les inverse).

- Exercice 4 - Calcul de l'arbre couvrant de poids (distance) minimum.

Ecrire une fonction void kruskal(int m, int edge[][3], int taille[], int comp[], vector<int> complist[], int arbre[][2]) qui applique l'algorithme de Kruskal au tableau d'arêtes **edge** et construit le tableau **arbre** qui contient les $n-1$ arêtes de l'arbre de distance minimum.

On pourra reprendre la fonction composantes du TP1, et y apporter des modifications mineures.

- Exercice 5 - Affichage.

Utiliser la fonction AffichageGraphique que vous pouvez trouver à l'adresse

http ://www.lirmm.fr/∼bessy/AffichagePS

afin d'afficher le résultat dans le fichier Exemple.ps.
L'appel se fera par AffichageGraphique(n,point,arbre) ;

- Exercice 6 - Pour aller plus loin.

- Améliorer les performances de votre algorithme en utilisant un tri plus efficace, par exemple tri fusion.
- Montrer que les arêtes de l'arbre obtenu ne peuvent se croiser.
- Utiliser d'autres distances (Manhattan, sup,...) pour créer votre arbre.
- Utiliser l'arbre de Kruskal afin d'approximer le voyageur de commerce.

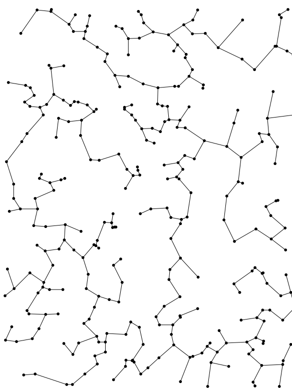

Fig. 1 – Un exemple d'arbre de Kruskal.