TP : Graphes

Le but de ce TP est d'implanter l'algorithme de Kruskal pour la recherche d'un arbre couvrant de poids minimum et l'algorithme de Dijkstra pour le calcul de plus courts chemins. Le problème de l'arbre couvrant de poids minimal a été présenté en cours. La donnée de ce problème est un graphe pondéré. Pour la représentation du graphe, vous utiliserez des listes d'adjacences, puis vous construirez la matrice d'adjacence correspondante.

Génération aléatoire des graphes

Vous commencerez par écrire un petit programme indépendant pour générer des graphes aléatoirement. Ce programme demandera à l'utilisateur les informations suivantes :
- le nombre de sommets,
- la probabilité d'existence d'une arête entre 2 sommets donnés,
- l'intervalle des valeurs possibles pour les poids (que l'on choisira entiers).

La façon la plus simple de générer aléatoirement un graphe consiste à fixer le nombre d'arêtes que l'on veut à partir du nombre de sommets et de la densité, puis de tirer au hasard des couples de sommets pour constituer ces arêtes (en vérifiant que l'arête n'existe pas déja). Le graphe généré sera simplement affiché à l'écran, en respectant le format suivant :

```
<nombre de sommets>  <nombre d'aretes>
<sommet> <sommet> <poids>
      ....
<sommet> <sommet> <poids>
```

Lecture et représentation du graphe.

Dans un second programme, vous implémenterez les algorithmes qui nous intéressent. Ce second programme devra donc lire les données générées par le programme précédent, et représenter le graphe par des listes d'adjacences. Vous écrirez une fonction qui construit la matrice d'adjacence à partir des listes d'adjacences.

Algorithme de Kruskal (Voir cours)

Pour respecter la complexité théorique $O(n \log n)$, on doit être capable de tester en $O(\log n)$ le fait que deux sommets appartiennent à la même composante connexe. Il faut donc utiliser une structure de données pour représenter des ensembles disjoints (ici, les composantes connexes). Les opérations élémentaires pour ces structures de données sont :
- `trouver-ensemble(x)` : rechercher à quel ensemble appartient un élément `x` donné,
- `union(E,F)` : faire l'union de deux ensembles disjoints `E` et `F`.

Vous implémenterez ces opérations avec des *forêts*. Chaque arbre de la forêt représente un ensemble de sommets (ses noeuds). Les informations que l'on maintient pour chaque noeud sont le numéro du père, et le *rang* du noeud. Par convention on dira que le père de la racine d'un tel arbre est elle-même. Le rang d'un noeud correspond à l'évaluation (en fait une borne supérieure) de la hauteur de l'arbre sous-jacent. Un arbre de ce type est identifié sans ambiguité par sa racine. Ainsi `trouver-ensemble(x)` renverra le numéro de la racine de l'arbre auquel appartient `x`—il est facile alors de déterminer si deux sommets figurent dans le même arbre ou non. L'union de deux ensembles se fait simplement en fusionnant les arbres correspondants (la racine de l'un prends pour père la racine de l'autre).
Par souci d'efficacité, on utilisera *l'union par rang* (on choisit l'arbre supposé le moins haut pour jouer le rôle du nouveau fils) et *la compression de chemins* (lors du calcul de `trouver-ensemble(x)`, on modifie les peres des noeuds intermédiaires afin qu'ils aient tous la racine comme père), deux techniques grâce auxquelles l'opération `trouver-ensemble` se fait avec une complexité $O(\log n)$ en moyenne.
Vous reprendrez les fonctions qui ont été présentées en cours. Le plus simple est d'utiliser des tableaux et les numéros des sommets.

Algorithme de Dijkstra

Voir cours.