Semaine 8 – CM -Arbres Recouvrants de Poids Minimal (ARPM)

Exemple introductif.

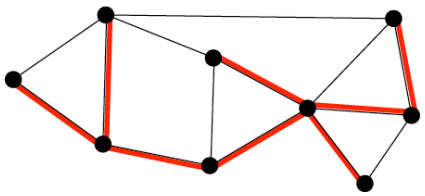
Vous souhaitez installer le câble dans le Calvados. Vous disposez pour cela d'une carte de l'ensemble du réseau routier (le câble est généralement disposé le long des routes). On vous demande de définir le réseau câblé de telle sorte que la longueur totale de câble soit minimale et qu'un certain nombre de lieux soient desservis.

On peut modéliser ce problème de câblage à l'aide d'un graphe non orienté connexe G = (S, A), où S associe un sommet à chaque lieu devant être desservi, et A contient une arête pour chaque portion de route entre deux lieux. Ce graphe est valué par une fonction coût qui spécifie pour chaque arête (u, v) la longueur de câble nécessaire pour connecter u à v.

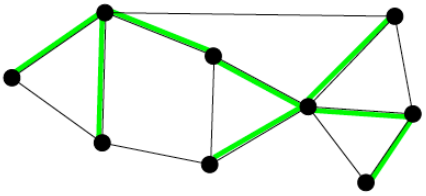
Il s'agit alors de trouver un sous-graphe connexe et sans cycle de ce graphe (autrement dit, un arbre) qui recouvre l'ensemble des sommets du graphe. Ce graphe est appelé arbre recouvrant (AR). On cherche à minimiser le poids total des arêtes de l'arbre. On dira qu'on cherche l'arbre couvrant de poids minimal (ARPM).

1. Définitions

- graphe non orienté *G*
- arbre *T* { sommets : tous les sommets de *G*
  arêtes : certaines arêtes de *G*



T arbre recouvrant de *G*



Un graphe peut avoir plusieurs arbres recouvrants.

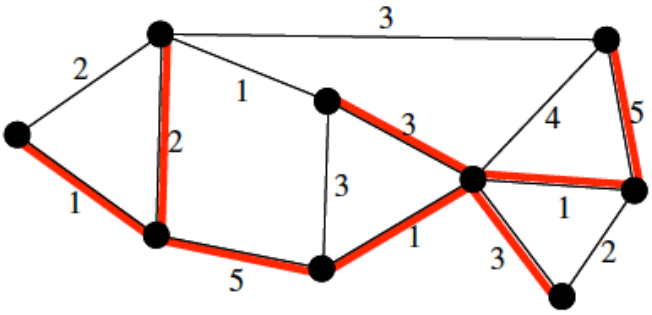
2. Enoncé du problème

Le problème consiste, dans un graphe non orienté connexe et valué, à trouver un sous-ensemble d'arêtes, formant un arbre, incluant tous les sommets (c'est-à-dire un arbre couvrant), tel que la somme des poids de chaque arête soit minimale (d'où le terme arbre couvrant de poids minimal).

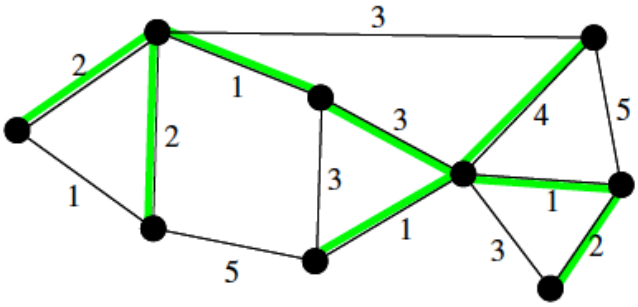
Soit G = (S, A) un graphe non orienté connexe valué par une fonction de poids w : A-> R+.

T est un ARPM de G ssi :

- T est un arbre (graphe connexe sans cycle) couvrant G
- T est de poids minimal. (Sigma pour e dans T des w(e) est minimale)



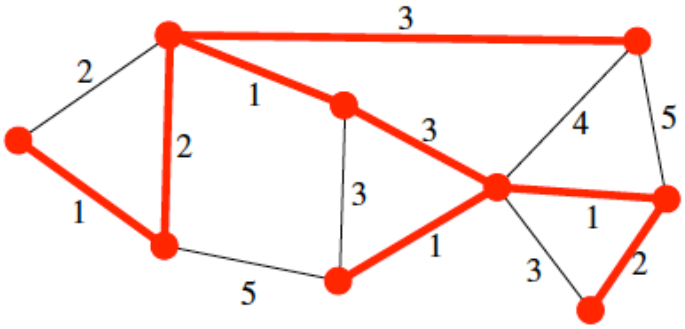
Poids de cet AR : 1+2+5+1+3+5+1+3 = 21



Poids de cet AR : 2+2+1+3+1+4+1+2 = 16

Solution force brute : générer tous les AR, pour chacun calculer leur poids et retenir le meilleur AR.
MAIS, pour un graphe G, le nombre d'AR peut être élevé, voir exponentiel : soit G={s, x1, x2, …, xn, t} avec une arête entre s et chaque xi et une arête de chaque xi à t.

3. Algorithme de KRUSKAL consiste à fusionner des sous arbres jusqu'à couverture complète. Chaque fusion se fait de la manière la plus économique possible



Poids de cet ARPM → 1+1+1+1+2+2+3+3 = 14

Principe.

Soit G = (V, E, W) un graphe non orienté valué connexe. Soit n = |V| le nombre de sommets de G.

- Partir d'une forêt constituée de n arbres isolés, où chaque arbre est réduit à un seul sommet.
- À chaque itération, créer un nouvel arbre en connectant deux sous arbres par une arête ne créant pas de cycle. Parmi toutes ces arêtes, choisir celle de coût minimal.
- S'arrêter lorsque la forêt ne comporte plus qu'un seul arbre.

Pseudo-code.

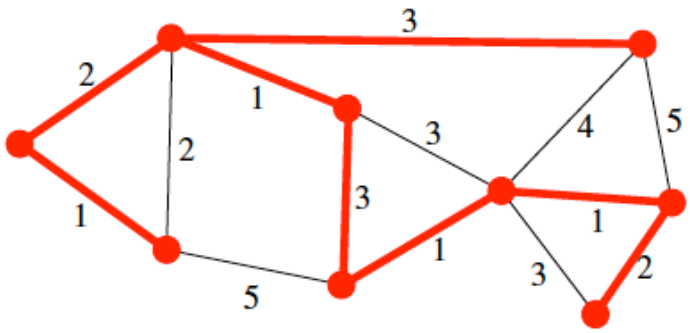
```
début
        pour chaque sommet v de V faire créer un arbre T_v réduit à sa racine v ;
        S := {} ;
        pour chaque arête (u, v) de E par ordre croissant faire[1]
                si l'ajout de (u, v) ne crée pas de cycle
                   alors
                        connecter le sous arbre T_u contenant u avec le sous arbre T_v contenant v ;
                        S := S + (u, v)
                fsi
        fpour
fin
```

En termes de graphes : l'ajout de (u, v) ne crée pas de cycle → les sommets u et v n'appartiennent pas à la même composante connexe.

Pseudo-code usuel.

- Initialiser T avec
  { sommets : tous les sommets de G
  { arêtes : aucune
- Traiter les arêtes de G l'une après l'autre par poids croissant :
  - Si une arête permet de connecter deux composantes connexes de T,
    - alors l'ajouter à T
    - sinon ne rien faire
  - Passer à l'arête suivante
- S'arrêter quand il n'y a plus d'arêtes
- Retourner T

4. Algorithme de PRIM choisit un premier sommet, puis fait grandir l'arbre depuis ce sommet. Il sélectionne l'arête incidente à ce sommet de poids minimal et l'ajoute à l'arbre et ainsi de suite. L'ARPM est construit de manière itérative, en sélectionnant à chaque pas l'arête de poids minimal qui relie un sommet de l'arbre avec un sommet en dehors de l'arbre.



Poids ARPM → 1+2+1+3+3+1+1+2 = 14

[1] Il n'est pas forcément nécessaire de passer en revue toutes les arêtes. En effet, « il ne reste plus qu'un seul arbre » peut se formuler : « le nombre d'arêtes ajoutées est (n-1) ».

Principe.
Soit G = (V, E, W) un graphe non orienté valué connexe. Soit n = |V| le nombre de sommets de G.

- Construire un arbre initial T réduit au seul sommet du graphe (choisi arbitrairement).
- À chaque itération, agrandir T en lui ajoutant le sommet libre accessible de plus petit poids.
- Stopper quand l'arbre est couvrant.

Pseudo code usuel.

- Initialiser T avec
  { sommets : un sommet de G qu'on choisit
  { arêtes : aucune
- Répéter :
  - Trouver toutes les arêtes de G qui relient un sommet de T et un sommet extérieur à T
  - Parmi celles-ci, choisir une arête de poids le plus petit possible
  - Ajouter à T cette arête et le sommet correspondant
- S'arrêter dès que tous les sommets de G sont dans T
- Retourner T

5. Conclusions

- *Notion de meilleur candidat admissible*

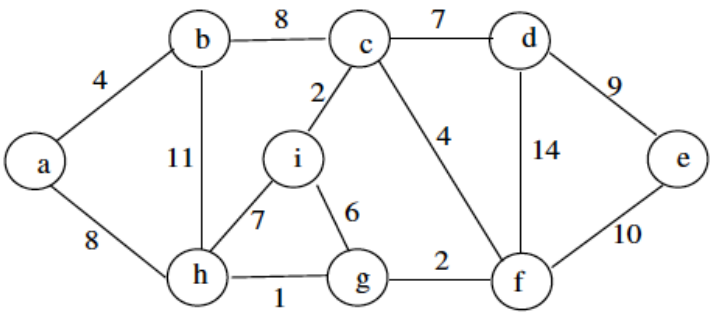
  - KRUSKAL → L'arête de poids minimal permettant de connecter deux composantes connexes.
  - PRIM→ L'arête de poids minimal parmi les arêtes permettant de relier un sommet de T à un sommet extérieur à T.

- *« Le » meilleur candidat admissible n'est pas forcément unique.*

- *Un graphe peut avoir plusieurs ARPMs* comme le montre l'exemple traité.

- Proximité entre l'algorithme de KRUSKAL avec construction de l'arbre de HUFFMAN et de l'algorithme de PRIM avec l'algorithme de plus court chemin de DIJKSTRA.

Exo. pour s'entraîner. Appliquer KRUSKAL puis PRIM (en prenant comme source le sommet a) au graphe ci-dessous :



Mise en œuvre de l'algo. de KRUSKAL en utilisant l'adt UNION-FIND[2]

(Présentation exo #2 du TD #3)

1. On s'intéresse au **type abstrait de données (ADT) « partition d'un ensemble »** (UNION-FIND) ou relation d'équivalence définie sur un ensemble. Il s'agit de trouver une structure de données appropriée pour gérer trois opérations :

1. Créer une relation d'équivalence « triviale » où chaque élément est seul dans sa classe.
2. Tester si deux éléments sont dans la même classe d'équivalence.
3. Faire l'union de deux classes d'équivalence pour n'en faire qu'une seule.

On notera **Sx** l'ensemble auquel appartient l'élément **x** (= la classe d'équivalence de **x**).

Ce type abstrait possède **trois primitives :**

- CRÉER-ENSEMBLE(x : Elément) crée un nouvel ensemble Sx dont le seul élément (et donc le représentant) est l'élément x.

- UNION(x, y : Elément) effectue l'union des ensembles disjoints Sx et Sy. Le représentant de (Sx U Sy) peut être un élément quelconque de cet ensemble. Bien souvent, on choisit, comme représentant de cette union, soit le représentant de Sx, soit le représentant de Sy.

- TROUVER-ENSEMBLE(x : Elément) qui retourne le représentant de l'ensemble Sx contenant l'élément x.

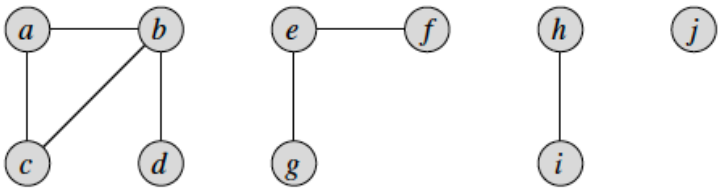
2. Utilisation de l'adt UNION-FIND pour exprimer le calcul des composantes connexes

La procédure COMPOSANTES-CONNEXES ci-dessous utilise les opérations d'ensembles disjoints pour calculer les composantes connexes d'un graphe. Après un pré traitement basé sur la procédure COMPOSANTES-CONNEXES, la fonction MÊME-COMPOSANTE indique si deux sommets se trouvent dans la même composante connexe.

```
procédure COMPOSANTES-CONNEXES(G : Graphe) ;
        Soit G = (S, A) ;
        pour chaque sommet u de S faire
                CREER-ENSEMBLE(u) ;
                pour chaque arête (u, v) de A faire
                        si TROUVER-ENSEMBLE(u) ≠ TROUVER-ENSEMBLE(v)
                                alors UNION(u, v)
                fpour
        fpour

fonction MEME-COMPOSANTE(u, v : Sommet) --> Booléen ;
        Retourner (TROUVER-ENSEMBLE(u) = TROUVER-ENSEMBLE(v))
```

Exemple :



[2] *Chap. 21, Introduction à l'algorithmique », T. Cormen, C. Leiserson, R. Rivest et C. Stein, Dunod, 2004.*

3. Utilisation de l'adt UNION-FIND pour exprimer l'algorithme de KRUSKAL

Ci-dessous, une formulation de l'algorithme de KRUSKAL utilisant les opérations d'ensembles disjoints.

```
fonction KRUSKAL (G : Graphe) --> ARPM ;
        soit G = (S, A, w)
        T := {} ;
        pour chaque sommet v de S faire CRÉER-ENSEMBLE(v) ;
        Trier les arêtes de A par ordre croissant selon w ;
        pour chaque arête (u, v) de E (par ordre croissant selon w) faire
                si TROUVER-ENSEMBLE(u) ≠ TROUVER-ENSEMBLE(v)
                        alors   T := T U {(u, v)} ;
                                UNION(u, v) ;
        fpour ;
        retourner(T)
```