Projet de Programmation & Algorithmique Avancée

Université Paris 7 — BioInformatique — M1

A rendre pour le xxx

Le but de ce projet est d'implémenter les algorithmes de calcul d'un arbre couvrant de poids minimum (ACPM) d'un graphe.

1 L'algorithme de Kruskal

1.1 Notations et définition du problème

Dans tout le sujet $G = (V, E)$ désigne un graphe non-orienté, connexe, et valué par une fonction de poids $w : E \mapsto \mathbb{R}^+$. On note $n = |V|$ et $m = |E|$.

Un ACPM de G est un graphe $T = (V, A)$ vérifiant :

1. T est un arbre (il a $n - 1$ arêtes et est connexe et sans cycles)
2. $A \subseteq E$ est un ensemble d'arêtes de G (T est couvrant)
3. Le poids de T (somme des poids des arêtes de A) est minimal parmis tous les arbres couvrants G

1.2 Algorithme générique

Les algorithmes de Prim et Kruskall suivent le schéma suivant :

Données: Un graphe $G = (V, E)$ de n sommets valué par w
Résultat: Les arêtes d'un ACPM de G
$A = \emptyset$
pour *i de 1 à n-1* **faire**
  Choisir l'arête e
  $A = A \cup \{e\}$
retourner A

1.3 Algorithme de Kruskall

On maintient l'**invariant** : $F = (V, A)$ est une forêt couvrante (un sous-graphe acyclique, composé des n sommets de V formant un ou plusieurs arbres). Au début il y a n arbres, tous vides. À chaque étape on relie deux arbres entre eux.

Le choix de l'arête e à l'étape i est l'arête la plus légère qui relie deux arbres différents.

De façon équivalente, l'algorithme est souvent présenté ainsi :

Données: Un graphe $G = (V, E)$ de n sommets valué par w
Résultat: Les arêtes d'un ACPM de G
$A = \emptyset$
Soient $e_1..e_m$ les arêtes de G **triées par ordre croissant**
pour *i de 1 à m* **faire**
  si $A \cup \{e_i\}$ *ne contient pas de cycle* **alors**
    $A = A \cup \{e\}$
retourner A

2 Structure de Détection des cycles

Soit $e_i = \{x, y\}$ l'arête que l'on essaie d'insérer à A. Pour savoir si $A \cup \{e_i\}$ contient un cycle il faut juste tester si x et y sont dans la même composante connexe. En effet si non on relie deux arbres différents donc on ne crée pas de cycle, alors que si oui on rajoute une arête entre deux sommets du même arbre ce qui crée necessairement un cycle.

On maintient alors une **liste des composantes connexes**. Au début il y a n composantes connexes : chacune est constituée d'un seul sommet. Puis quand on relie la compsante de x à celle de y, cela fusionne les deux composantes. Ainsi nous avons une structure de donnée abstraite qui utilise trois opérations :
- `Initialiser(V: ensemble)` initialise la structure avec une composante par élément de V
- `Union(x,y : éléments)` fusionne la composante de x à celle de y
- `Find(x,y : éléments) renvoie booléen` répond vrai si x et y sont dans la même composante, faux sinon.

La version définitive de Kruskal est alors :

Données: Un graphe $G = (V, E)$ de n sommets valué par w
Résultat: Les arêtes d'un ACPM de G
`Initialiser(V)`
$A = \emptyset$
Soient $e_1..e_m$ les arêtes de G **triées par ordre croissant**
pour *i de 1 à m* **faire**
  Soit $e_i = \{x, y\}$
  si *Find(x,y)*$=faux$ **alors**
    $A = A \cup \{e\}$
    `Union(x,y)`
retourner A

Vous devez réaliser *quatre* implémentations différentes de la structure des composantes.

2.1 Méthode Indice

Dans la première on donne un numéro quelconque aux composantes. Notons $x.num$ le numéro de la composante de l'élément x.
- `Initialiser(V)` donne à chaque $x \in V$ un numéro de composante différent (un numéro de sommet par exemple)
- `Union(x,y)` renumérote. Soit $c = y.num$. Tous les sommets de numéro c (incluant y lui-même) sont renumérotés par `Union(x,y)` pour prendre le numéro $x.num$
- `Find(x,y)` répond vrai si si $x.num = y.num$

2.2 Méthode Liste

Dans cette deuxième méthode on maintient, pour chaque composante, la liste chainée de ses éléments :

– `Initialiser(V)` crée V listes d'un élément, une par élément de V ;
– `Union(x,y)` colle la tête de la liste de y à la fin de la queue de la liste de x, suivant la méthode usuelle de fusion de listes.
– `Find(x,y)` peut se faire de deux façons différentes : soit en recherchant x dans la liste de y, soit en regardant si x et y ont même tête de liste.

Notez que les listes doivent être doublement chaînées car parcourues dans les deux sens, mais qu'il est inutile de maintenir une liste des listes, ou une liste des têtes de liste.

2.3 Méthode Arbre (aussi appellée Union-Find)

Dans cette troisième méthode les sommets de chaque composante sont organisés en un arbre. **Cet arbre n'a rien a voir avec le graphe G** !! L'implémentation de la structure est abstraite et indépendante d'un graphe. Tout simplement on rajoute aux éléments de V un pointeur `père` ce qui crée des structures d'arbres enracinés (orientés) et définit implicitement des racines (éléments sans père).

Dans chacun des cas on a une procédure `Racine(x)` qui remonte les pointeurs `père` jusqu'à la racine. On a alors :
– `Initialiser(V)` se contente de mettre à `null` les pointeurs `père`
– `Union(x,y)` fait alors `père(Racine(y))=Racine(x)` : on fusionne les deux arbres en faisant que la racine de l'un devient fille de la racine de l'autre
– `Find(x,y)` répond alors simplement *vrai* si `Racine(x)==Racine(y)`.

2.4 Première amélioration de la méthode Arbre : fusion hiérarchique

On crée un autre champs par élément : `taille`. Si x est racine de sa composante, **et seulement dans ce cas-là**, alors $x.taille$ est le nombre d'éléments de la composante de x. L'arbre de plus petite taille est ajouté au plus grand :

`Union(x,y : éléments)`;
rx = Racine(x);
ry = Racine(y);
si $rx.taille \geq ry.taille$ **alors**
  $ry.pere = rx$;
  $rx.taille = rx.taille + ry.taille$;
sinon
  $rx.pere = ry$;
  $ry.taille = rx.taille + ry.taille$;

2.5 Deuxième amélioration de la méthode Arbre : compression de chemins

Lors d'une requête `Racine(x)` on obtient une racine rx comme résultat. La compression de chemins consiste à, pour tous les sommets parcourus depuis x inclus jusqu'à rx exclus, de changer le père de ce sommet pour en faire rx. Cela peut se faire en stockant, dans une pile ou autre structure, ces sommet (ils forment le chemin reliant x à la racine rx).

3 Travail demandé

Implémenter l'algorithme de Kuskall de huit façons différentes :
– Le graphe peut être représenté en
  – listes d'adjacences, ou
  – matrices d'adjacence
– La structure des composantes peut être :

- Indice
- Liste
- Arbre simple
- Arbre avec les deux implémentations

3.1 Compteur d'usage

Pour évaluer empiriquement la complexité de vos algorithmes, on doit compter tous les accès aux structures. Donc tout champs doit être accessible uniquement via un accesseur qui incrémente une variable `static`.

Ce compteur sera donc incrémenté au moins $deg(v)$ fois lors des parcours des voisins d'un sommet v, et même n fois si stockage en matrice. Parcourrir une liste L contenant un élément l'incrémentera au moins $|L|$ fois, etc etc. Les différentes classes auront chacune leur compteur. Vous pouvez même en prévoir plusieurs : par champ, en lecture ou en écriture, etc. À la fin du programme seront affichés les statistiques (valeurs de tous ces compteurs et total).

Les appels aux constructeurs donneront lieu à incrément d'autres compteurs (un par constructeur) qui seront aussi affichés séparéments en fin de programme.

Votre note dépendra de la valeur des compteurs pour des exemples donnés. En particulier pour la meilleure implémentation elle devra être le plus bas possible.

Tout usage d'un champ ou d'un constructeur n'utilisant pas ce compteur sera sanctionné.