GRI – Cours 3

Clémence Magnien Christophe Prieur

5 février 2015

Cœur et périphérie, généralisation : k-cœurs, k-périphérie

k-cœur : sous-graphe $H = (C, E(C))$ de G tel que, pour tout $v \in C$, $d^\circ\ (v) \geq k$. On peut étudier les k-cœurs comme une généralisation de la connexité.

k-périphérie : sommets qui sont dans le $k-1$-cœur mais pas dans le k-cœur (périphérie=1-périphérie).

Remarque : un k-cœur n'est pas nécessairement connexe.

Pour calculer les k-cœurs :
- calculer la périphérie, puis
- calculer la 2-périphérie, puis
- calculer la 3-périphérie, etc.

Pour être efficace, il faut stocker tous les sommets de degré entre 1 et une valeur k dans k ensembles.

L'algorithme est obtenu en modifiant celui du calcul de la périphérie, comme suit :
- pour chaque sommet v
  - si `deg[v]` $\leq k$ alors ajouter v à `ensdeg[deg[v]]`
- pour k allant de 1 à k
  - mettre tous les sommets de `ensdeg[`k`]` dans la file
  - calculer la périphérie comme précédemment, en mettant à jour `ensdeg` dès qu'on modifie un `tmpdeg[v]`
- à la fin, `ensdeg[`k`]` contient la k-périphérie.

Ensemble en C (pour un nombre d'éléments borné)

- Un tableau de taille n (borne supérieure sur la taille de l'ensemble) qui contient les éléments de l'ensemble.
- Un entier qui contient le nombre d'éléments
- Un tableau de taille n qui contient l'indice, dans le premier tableau, de chacun des éléments présents dans l'ensemble, et -1 pour les éléments absents.

Les opérations :

ajout *si l'élément est absent*, on l'ajoute à la fin, on met à jour son indice et on incrémente la taille.

suppression on échange l'élément à supprimer avec le dernier élément, on met à jour les deux indices et on décrémente la taille.

Connexité en *streaming*

On calcule la taille des composantes connexes du graphe sans le stocker en mémoire, en lisant directement les liens un par un. On considère qu'au départ tous les nœuds sont dans des composantes isolées. À chaque fois qu'on considère un lien entre u et v, on fait l'union de la composante de u et de celle de v.

À la fin on a les composantes connexes.

Avantages : pas besoin de stocker le graphe, même complexité en pratique que le parcours en largeur.

Idée : les composantes connexes sont des ensembles disjoints, on les stocke dans un tableau `set` sous forme d'arbre. Chaque nœud pointe sur son père. La racine est telle que `set[u] == u`.

Version n134(8i31(ue)]TJ/F44 9.9626 Tf 14.944 -24ó Td [(nitia(listsion)-838(:)
ocitis(on):

set[ue] u

Compression des chemins : pendant la recherche de la racine, on fait pointer chaque nœud visité vers la racine.

`initialisation`
- pour tout sommet v faire :
  - set[v] = v
  - rang[v] = 0

`find(u)`
- si u != set[u]
  - set[u] = `find`(set[u])
- return `set[u]`

`union(u,v)`
- `link(find(u), find(v))`

`link(u,v)`
- si rang[u] > rang[v] :
  - set[v] = u
- sinon
  - set[u] = v
  - si rang[u] == rang[v] et u!=v alors rang[v] ++

`find(u)` (version plus efficace non récursive)
- p = u
- tant que set[p] != p
  - p = set[p]
- v = u
- tant que set[v] != v
  - tmp = set[v]
  - set[v] = p
  - v = tmp
- return p ;

Remarque : Le mot clé `union` existe en C, et une fonction `link` est prédéfinie. Utilisez d'autres nom pour ces deux fonctions.

Complexité de la version finale

Rangs uniquement : $\mathcal{O}(m \log n)$
(quand on n'utilise pas la compression de chemins, le rang est égal à la profondeur, et la profondeur maximale est $\log n$).

Compression de chemins + rangs : $\mathcal{O}(m\alpha(n))$ avec $\alpha(n) < 4$ en pratique. La complexité est donc linéaire en pratique (formellement elle est très légèrement surlinéaire).