GRI – Cours 4

Christophe Prieur Clémence Magnien

19 février 2015

Coe cient de clustering d nitions et calcul

Clustering : lié aux triangles

Proba que deux nœuds soient reliés s'ils ont un voisin en commun

Intuitivement, cela correspond à l'adage : *es a is de es a is sont es a is*. Calculer le coefficient de clustering d'un graphe permet de savoir à quel point c'est le cas pour un graphe donné.

Coe cient de clustering local

Soit un sommet v, son coefficient de clustering est égal au nombre d'arêtes entre ses voisins, divisé par le nombre d'arêtes total qu'il pourrait y avoir.

Si on note $N(v)$ les voisins de v et $E(N(v)) = \{(u_1, u_2) \in E, u_1, u_2 \in N(v)\}$, alors

$$lcc(v) = \frac{2|E(N(v))|}{d^\circ(v)(d^\circ(v) - 1)}.$$

Proche de la notion de densité.

Le coefficient de clustering local d'un graphe G, noté $lcc(G)$, est la moyenne du coefficient de clustering de tous les sommets *qui ont un degr sup rieur ou ga à* .

Coe cient de clustering global

$$gcc(G) = \frac{3N_\Delta}{N_\vee},$$

où N_Δ est le nombre de triangles dans le graphe, et $N_\vee$ le nombre de triplets connexes (trois sommets et au moins deux arêtes).

Probabilité que quand deux sommets ont un voisin en commun, le troisième lien existe.

Calcul des triangles

Triangles auxquels appartient un sommet

Pour un sommet v, on veut trouver tous les triangles qui le contiennent.

Solution naïve : pour toute paire (u_1, u_2) de voisins de v, regarder s'il existe un lien entre u_1 et u_2.

Complexité : $d^\circ(v)(d^\circ(v)-1)/2$ paires de voisins de v (plus le temps pour trouver si le lien entre u_1 et u_2 existe). Coûteux pour les sommets de fort degré.

Méthode alternative : Pour chaque sommet v : pour chaque voisin u de v, regarder tous les voisins de u et regarder s'ils sont reliés à u. L'algorithme 1 permet de faire cette dernière étape en temps constant.

Algorithm 1: Lister les triangles d'un sommet v

```
A = tableau de taille n, initialisé à 0 ; for chaque voisin u de v do
    A[u] = 1 ;
for chaque voisin u de v do
    for chaque voisin w de u do
        if A[w] == 1 then
            Afficher un triangle entre u, v et w ;
```

Attention cependant car il faut initialiser le tableau à 0 : cette méthode est moins efficace que la méthode naïve pour les sommets de faible degré.

Triangles auxquels appartient une arête

Pour une arête (u, v), on veut savoir combien de triangles la contiennent.

Il y a un triangle si un sommet w est voisin à la fois de u et v. w appartient aux deux tableaux d'adjacence de u et de v, on veut donc calculer leur intersection.

Pour calculer l'intersection facilement, les tableaux d'adjacence **doivent être triés**.

Algorithm 2: Calcul de l'intersection de deux listes d'adjacences triées

```
iu = 0;
iv = 0;
while (iu < d°(u)) and (iv < d°(v))) do
    if (g->links[u][iu] < g->links[v][iv]) then
        iu++;
    else
        if (g->links[u][iu] > g->links[v][iv]) then
            iv ++ ;
        else
            Triangle between u, v, and g->links[u][iu];
            iu++ ; iv++ ;
```

Cette méthode est longue si u ou v a un fort degré. En pratique on a un certain nombre de nœuds qui ont un fort degré, il faut donc améliorer la méthode.

Algorithme final

Pour chaque nœud, on veut calculer à combien de triangles il appartient.

En utilisant la méthode ci-dessus, on peut voir le triangle u, v, w en calculant l'intersection des tableaux d'adjacences de u et v, de u et w, ou de v et w.

Il faut réduire le plus possible la taille des tableaux dont on va calculer l'intersection. On commence par **trier les sommets par degrés décroissants et renuméroter le graphe**. Puis :

— on ne considère que les liens (v, u) tels que $v < u$

— on ne cherche les sommets w voisins de u et v que si $w < v$ (et donc $w < u$).

Intuition : si v a un fort degré :

— u aura un degré plus faible que lui

— il y aura très peu de voisins w de v avec $w < v$

On peut donc dans ce cas arrêter de calculer l'intersection des voisinages dès qu'on voit un sommet $x > v$.

Si v a un faible degré, u aura un degré encore plus faible que lui et les deux tableaux d'adjacence de u et v seront courts.

Tri des sommets par ordre décroissant.

```
void qsort(void *tab, size_t nb_elem, size_t taille_elem,
                  int(*compar)(const void *, const void *));
```

`compar(*u,*v)` : fonction qui renvoie :

— < 0 si u doit être avant v

— 0 s'ils sont identiques

— > 0 si u doit être après v.

On veut trier par degré *d croissant*, donc `compar` peut renvoyer par exemple : $d^\circ(v) - d^\circ(u)$

Renumérotation du graphe. On a un graphe tel que : $d^\circ(0) = 10, d^\circ(1) = 20, d^\circ(2) = 18, d^\circ(3) = 16$. Après tri par degré décroissant on a le tableau :

1	2	3	0
0	1	2	3

qui indique que 1 doit être remplacé partout par 0, 2 doit être remplacé par 1, etc.

On calcule le tableau *inverse* :

3	0	1	2
0	1	2	3

On parcourt tous les tableaux d'adjacence du graphe et on remplace partout `u` par `inverse[u]`.

Permutation des tableaux d'adjacence et des degrés :

```
memcopy(copylinks,g->links,g->n*sizeof(*(g->links)))
memcopy(copydeg,g->degrees,g->n*sizeof(*(g->degrees)))
```

Pour chaque sommet `u` :

— `g->links[inverse[u]] = copylinks[u]`

— `g->degrees[inverse[u]] = copydeg[u]`

(on remplace partout `u` par `inverse[u]`, donc le nouveau degré de `inverse[u]` est l'ancien degré de `u`)

Tri des tableaux d'adjacence. On trie ensuite chaque tableau d'adjacence avec `qsort`.

L'algo final est donné dans l'algorithme 3.

Algorithm 3: Algo final

```
Result: Tableau tr qui contient le nombre de triangle de chaque sommet
tr : tableau de taille n initialisé à 0;
for v = 0; v < n; v++ do
    for i=0;i < d°(v) ;i++ do
        u = i-ème voisin de v;
        if u > v then
            iu = 0;
            iv = 0;
            while iu < d°(u)) and iv < d°(v)) and g-> inits u[ iu <v)
            and g-> inits v[ iv <v) do
                // Ce while est une variante de l'algorithme 2;
                if g-> inits u[ iu <g-> inits v[ iv then
                    iu++;
                else
                    if g-> inits u[ iu >g-> inits v[ iv then
                        iv ++;
                    else
                        tr[u]++;
                        tr[v]++;
                        tr[g->links[u][iu]]++;
                        iu++;
                        iv++;
```

Amélioration possible au niveau du deuxième for : on peut chercher le premier sommet $u > v$ dans le tableau d'adjacence de v par une recherche dichotomique.

Calcul des coe cients de clustering partir du no bre de triangles

Une fois qu'on connait le nombre de triangles auxquels appartient chaque sommet, le calcul des deux coefficients de clustering est facile.

Clustering local pour un sommet v

$|E(N(v))|$ = nombre de triangles qui contiennent v.

Clustering global

Nombre de triangles dans le graphe : somme de toutes les cases du tableau, divisée par trois.

Nombre de triplets : pour chaque sommet v, son nombre de triplets est $d^\circ(v)(d^\circ(v)-1)/2$. On fait la somme sur tous les sommets du graphe.

Valeurs en pratique et co paraison

En pratique, les deux coefficients de clustering sont forts pour les graphes de terrain : plusieurs ordres de grandeur au dessus de la densité.

Pour le clustering global, cela signifie que deux sommets qui ont un voisin en commun ont nettement plus de chances d'être reliés que deux sommets choisis au hasard.

Si un sommet a degré k, il a $k(k-1)/2$ paires de voisins. Il est donc rare qu'un sommet de fort degré ait un fort coefficient de clustering local. De manière générale, plus le degré d'un nœud est fort, plus son coefficient de clustering est faible.