Compte Rendu TME 3

Wenzhuo ZHAO, Zhaojie LU, Chengyu YANG, Zhen HOU

Mars 2021

1 Simple Benchmark

Nous générons des graphes à la taille de 400 noeuds, avec les contraintes suivantes:

- Chaque paire de noeuds dans un même cluster ont une probabilité p d'être connectée
- Chaque paire de noeuds dans de différents clusters ont une probabilité q d'être connectée, dont $q \leq p$

Évidemment, la fraction $\frac{p}{q}$ peut avoir un impact sur les communautés de noeuds. Voici des graphes dessinés par la librairie de Python networkx[4] avec différentes valeurs de p, q et $\frac{p}{q}$.

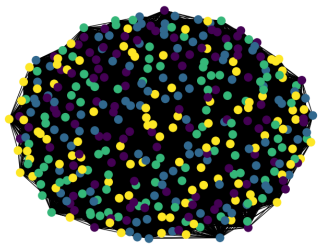

Figure 1: Graphe de 400 noeuds, avec p = 0.3, q = 0.3

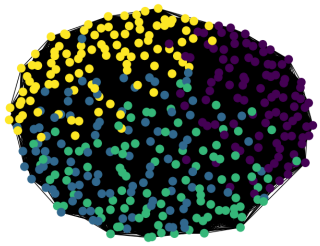

Figure 2: Graphe de 400 noeuds, avec p = 0.6, q = 0.3

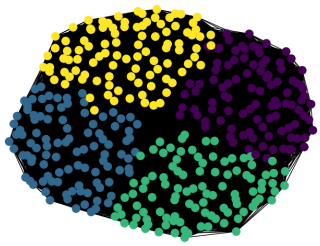

Figure 3: Graphe de 400 noeuds, avec p = 0.8, q = 0.2

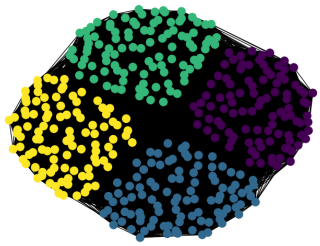

Figure 4: Graphe de 400 noeuds, avec p = 0.8, q = 0.1

Avec l'augmentation de la valeur $\frac{p}{q}$, le recouvrement entre communautés (*overlapping community*) a bien réduit. En revanche, la baisse de la valeur $\frac{p}{q}$ rend les communautés plus mêlées.

2 Label Propagation

La propagation de labels consiste à ce que chaque noeud puisse recevoir un label par un vote majoritaire de ses voisins. Nous implémentons cet algorithme qui rend une liste de longueur n contenant les labels correspondants à chaque noeud. Un graphe décrit avec cette liste de labels comme la communauté est dessiné ci-dessous.

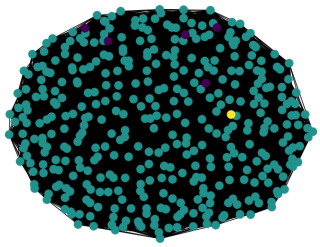

Figure 5: Communautés de graphe de 400 noeuds en Section "Simple benchmark", avec p = 0.6, q = 0.3

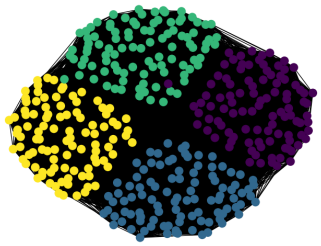

Figure 6: Communautés de graphe de 400 noeuds en Section "Simple benchmark", avec p = 0.8, q = 0.1

On peut observer une tendance à ce que les noeuds dans un graphe dense 5sont rangés dans une même communauté, néanmoins les noeuds dans un graphe avec des clusters moins connectés sont rangés dans de différentes communautés Cette méthode soit simple à mettre en oeuvre, elle présente une grande instabilité due au non déterminisme de l'algorithme et peut dans certains cas ne pas détecter de structures communautaires.
Notre implémentation de cet algorithme fonctionne aussi bien sur des graphes à la grande taille. Sur le graphe [1] à la taille de 10^6 noeuds et arêtes, le programme prend 18 secondes.

3 Nouvel Algorithme

Dans cette section, nous utilisons une implémentation de l'algorithme Louvain en C++ et introduisons une approche divisive afin de mieux étudier la détection de communautés.

3.1 Louvain

La méthode de Louvain[2] est un algorithme hiérarchique d'extraction de communautés applicable à de grands réseaux. Il s'agit d'un algorithme glouton avec une complexité temporelle de $O(n \log n)$. Nous utilisons cette implémentation sur des jeux de données générés en section "Simple Benchmark".

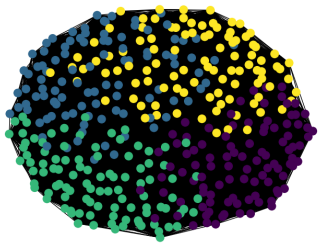

Figure 7: Communautés de graphe de 400 noeuds en Section "Simple benchmark", avec p = 0.6, q = 0.3

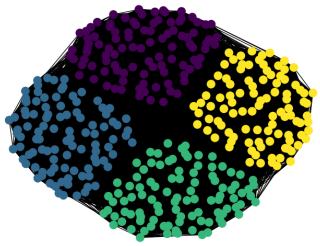

Figure 8: Communautés de graphe de 400 noeuds en Section "Simple benchmark", avec p = 0.8, q = 0.1

Pour un mouvement de noeud i vers une autre communauté S, le calcul de coût qui est l'écart de modularité[6] ΔQ ne dépend que de la communauté S et le degré du noeud i, donc il est rapidement calculé et peut être facilement parallélisé en complexité quasi linéaire.
En terme de modularité, cet algorithme renvoie un meilleur résultats par rapport celui de l'algorithme Label Propagation. On peut observer une grande différence de résultats sur figure 7 et figure 5 en appliquant ces 2 algorithmes et nous pouvons dire que l'algorithme Louvain a une meilleur stabilité.

3.2 Une approche divisive

Une approche divisive consiste à considérer initialement le graphe entier comme une seule communauté. Puis, itérativement supprimer un lien du graphe de sorte que la modularité soit optimisée. Le choix du lien à supprimer à chaque itération consiste à choisir le lien dont la centralité d'intermédiarité [3] est maximale. Nous obtiendrons un "dendrogramme" à la fin.

Algorithm 1: Divisive approach

```
Data: Graph G
Result: List partition
/* la modularité Q                                          */
max_Q = 0;
/* partitions, initialement contenant tous les noeuds       */
partition = [[G.nodes]];
while length(G.edges) > 0 do
    edge = G.highestEdgeBetweenness();
    G.remove(edge);
    components = G.connectedComponents();
    if length(partitions) != length(components) then
        /* Calcul de la modularité                          */
        q = calculateQ(components, G);
        if q > max_Q then
            max_Q = q ;
            partitions = components ;
return partition
```

Chaque fois qu'un arête est retiré dans le graphe, la centralité d'intermédiarité doit être recalculée et il est assuré qu'au moins un arête entre deux communautés a une valuer maximum de la centralité d'intermédiarité. Ce processus de recalcul apporte l'inconvénient de cet algorithme que la complexité est de l'ordre de $O(n^3)$. Grâce à la librairie NetworkX [4], des fonctions comme

calculer la centralité d'intermédiarité peuvent être importées en Python. En tenant compte que cet algorithme va opérer sur le graphe, il est donc nécessaire de faire une copie de graphe avant l'utilisation de cet algorithme. Prenons un graphe avec 128 noeuds et 1024 arêtes généré par LFR Benchmark [5] et voici une visualisation de communautés renvoyées par cet algorithme.

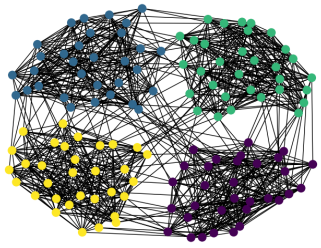

Figure 9: Communautés de graphe de 128 noeuds et de 1024 arêtes

4 Expérimentation

Dans cette section, nous allons comparer les algorithmes mentionnés ci-dessus par la scalabilité et le temps d'exécution sur des graphes de différentes tailles. Ensuite, à l'aide des métriques (modularité, AMI, RI et NMI) nous comparons la précision de ces algorithmes.

- Pour un graphe avec 128 noeuds et 1024 arêtes:
  - LPA Modularity: 0.462646484375
    Louvain Modularity: 0.6474609375
  - LPA AMI: 0.737737439924118
    Louvain AMI: 1.0
  - LPA RI: 0.6523205048773107
    Louvain RI: 1.0
  - LPA NMI: 0.7583588079622381
    Louvain NMI: 1.0
  - dont pour notre divisive approach, on a:
    Divisive Algorithm Modularity: 0.6474609375
    Divisive Algorithm AMI: 1.0
    Divisive Algorithm RI: 1.0
    Divisive Algorithm NMI: 1.0

- Pour un graphe avec 1000 noeuds et 8000 arêtes:
  - LPA Modularity: 0.7422699999999998
    Louvain Modularity: 0.86793
  - LPA AMI: 0.9097944664743935
    Louvain AMI: 1.0
  - LPA RI: 0.7952786791472185
    Louvain RI: 1.0

– LPA NMI: 0.9253864956345751
Louvain NMI: 1.0

On peut constater que l'algorithme Louvain a une meilleur précision entre ses résultats de labels et les labels réels par rapport celle de l'algorithme Label Propagation. Pour notre divisive approache, le nombre de tests n'est pas assez nombreux mais on a aussi un bon résultar sur cet algorithme.

Appendix A Source Code

Le source code de ce TME est disponible sur notre répertoire de GitHub [7].

References

[1] *Amazon*. http://snap.stanford.edu/data/com-Amazon.html.

[2] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. DOI: 10.1088/1742-5468/2008/10/p10008. URL: https://doi.org/10.1088/1742-5468/2008/10/p10008.

[3] Linton C. Freeman. "A Set of Measures of Centrality Based on Betweenness". In: *Sociometry* 40.1 (1977), pp. 35–41. ISSN: 00380431. URL: http://www.jstor.org/stable/3033543.

[4] Aric Hagberg, Pieter Swart, and Daniel S Chult. "Exploring network structure, dynamics, and function using networkx". In: (Jan. 2008). URL: https://www.osti.gov/biblio/960616.

[5] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. "Benchmark graphs for testing community detection algorithms". In: *Phys. Rev. E* 78 (4 Oct. 2008), p. 046110. DOI: 10.1103/PhysRevE.78.046110. URL: https://link.aps.org/doi/10.1103/PhysRevE.78.046110.

[6] M. E. J. Newman. "Modularity and community structure in networks". In: *Proceedings of the National Academy of Sciences* 103.23 (2006), pp. 8577–8582. ISSN: 0027-8424. DOI: 10.1073/pnas.0601602103. eprint: https://www.pnas.org/content/103/23/8577.full.pdf. URL: https://www.pnas.org/content/103/23/8577.

[7] W.Zhao. *Détection de communautés en Python*. https://github.com/valeeraZ/Sorbonne_CPA_Graph/tree/master/TME3.