LU2IN013

Rapport de projet

Motif dans les graphes

SEVI Melvin
CHELOUCHE Keryan
NEVEU Marie

Encadrant : M.TABOURIER Lionel

1 Motivation et notions importantes

Un graphe est un ensemble d'objets qualifiés de **noeuds** dont la relation entre une paire de noeuds est modélisée par un objet qualifié d'**arête ou arc**. De nombreux systèmes peuvent être représentés par des graphes notamment en biologie, en chimie organique. On retrouve aussi ces objets dans les domaines des transports et des télécommunications.

Par exemple, dans le cas de la chimie moléculaire, les molécules sont représentées par des graphes où les noeuds correspondent aux atomes et les arêtes représentent les liaisons (fortes ou faibles) entre atomes formant les molécules. On peut également représenter le métro parisien par un graphe où les noeuds représentent les stations et les arêtes, la présence d'une correspondance entre deux stations. Enfin dans le cas des réseaux sociaux, les noeuds représentent les individus et les arêtes représentent les liens d'amitiés entre individus.

L'analyse de ces graphes est complexe et afin d'extraire des informations de ces graphes, nous avons besoin, par exemple, d'identifier des sous-structures fréquentes. Notre objectif sera donc de reproduire les résultats de l'article de Milo et al. : *"Superfamilies of Evolved and Designed Networks"*[1], c'est-à-dire identifier certains motifs spécifiques dans un graphe réel, permettant d'en comparer la présence avec des graphes généré aléatoirement.

1.1 Theorie des graphes

Graphe : Soit $G = (V, E)$ un graphe où V est l'ensemble de noeuds et E l'ensemble des arêtes. On définit la cardinalité, c'est-à-dire le nombre de noeuds de G par $|V| = n$ et le nombre d'arêtes par $|E| = m$.

Orientation : Un graphe peut être orienté ou non orienté. Un graphe est dit orienté si chacune des arêtes reliant deux noeuds a une direction. Les éléments de E sont des couples et non des ensembles. Soit $u, v \in V$ alors si $(u, v) \in E \Rightarrow v$ on dit que v successeur de u et u est un prédécesseur de v. Un graphe est dit non orienté si chacune des arêtes reliant deux noeuds n'a pas d'orientation. Les arêtes sont des ensembles et non des couples. Soit u, v alors si $\{u, v\} \in E$; on dit u et v sont adjacents.

Degré : Dans un graphe non orienté ; on note $\Gamma(u)$ l'ensemble des voisins du noeud u et le degré $d(u) = |\Gamma(u)|$ d'un noeud u est le nombre d'arcs incidents à u. Soit $\{u, v\}$ on dira que l'arête $\{u, v\}$ est incidente à u et v. Dans le cas d'un graphe orienté, soit u un noeud on distinguera le degré entrant $d^-(i) = |\Gamma^-(u)|$, qui est le nombre d'arêtes entrantes de u et le

1. https ://www.researchgate.net/publication/8462722_Superfamilies_of_Evolved_and_Designed_Neworks

degré sortant $d^+(i) = |\Gamma^+(u)|$ de u qui est le nombre d'arêtes sortantes de u. Soit (u, v) ; on dira que (u, v) est un arc entrant de v et un arc sortant de u. On définit alors le degré de u comme la somme $d(u) = d^-(u) + d^+(u)$

Chaîne : Une chaîne est une séquence de noeuds et d'arêtes $\nu = v_1 e_1 v_2 e_2 \cdots v_n e_n v_{n+1}$ avec $v_i \in V$ pour $i \in \{1, \ldots, n+1\}$ et $e_i = \{v_i, v_{i+1}\} \in E$ pour $i \in \{1, \ldots, n\}$. Une chaîne élémentaire est une chaîne qui ne passe pas deux fois par le même noeud.

Graphe connexe : Un graphe est connexe si, pour tout couple $(u, v) \in V^2$, il existe une chaîne élémentaire entre u et v.

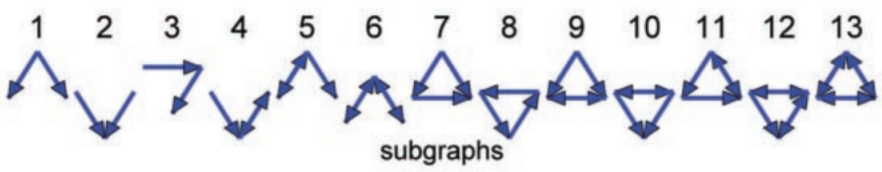
Composante connexe : Soit $x \sim y$ la relation d'équivalence définie sur l'ensemble E des points d'un graphe non orienté par $x \sim y \iff$ il existe une chaîne d'extrémités x et y. Les classes de la relation d'équivalence $\sim$ sont appelées les composantes connexes du graphe non orienté.

Faible connexité : Pour un graphe orienté, on dit qu'il est faiblement connexe, si en oubliant l'orientation des arêtes, le graphe est connexe.

Sous-graphe : Un sous-graphe de $G = (V(G), E(G))$ est un graphe $H = (V(H), E(H))$ tel que $V(H) \subset V(G)$; $E(H) \subseteq E(G)$ et pour tout $\{i, j\} \in E(H)$; $i, j \in V(H)$. On appellera un sous graphe connexe un motif.

Triangles : On définira un triangle dans un graphe non orienté comme un motif où trois noeuds possède tous un lien avec les deux autres :

Triades : On définira les triades comme tous motifs faiblement connexe à 3 noeuds. Il s'agit des motifs suivant :



Densité : Soit $G = (V, E)$ un graphe. La densité δ de G est la proportion d'arcs qui

apparaissent dans le graphe, c'est-à-dire :

$$\delta = \frac{nombre\ d'aretes\ du\ graphe}{nombre\ d'aretes\ maximum\ du\ graphe} = \frac{m}{\binom{n}{2}} = \frac{2m}{n(n-1)}.$$

Degré moyen : Le degré moyen d'un graphe $G = (V, E)$ est défini de la manière suivante :

$$\overline{d} = \frac{nombre\ total\ de\ degre\ du\ graphe}{nombre\ de\ noeud\ du\ graphe} = \frac{\sum_{i=1}^{n} d(i)}{n} = \frac{2m}{n}.$$

2 Choix d'implémentation et structure de données permettant de représenter un graphe

Représentation d'un graphe non orienté à l'aide d'une matrice d'adjacence :

Un graphe non orienté $G = (V, E)$ peut être représenté par une matrice définie de la manière suivante. Soit $\mathcal{M}_G = \{m_{ij}\}_{1 \leq i,j \leq n}$, la matrice d'adjacence de taille n associé au graphe G. On se concentre ici dans le cas des graphes non pondérés, c'est-à-dire $m_{ij} = 1$ si $\{i, j\} \in E$ et 0 sinon. Par exemple le graphe $G = (\{1, 2, 3, 4\}), (\{1, 2\}, \{2, 4\}, \{3, 4\}, \{1, 3\}, \{2, 3\}))$ est représenté par la matrice :

$$\mathcal{M}_G = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Remarque : la matrice d'adjacence d'un graphe non orienté est symétrique.

Représentation d'un graphe non orienté à l'aide d'une liste d'adjacence :

Un graphe non orienté $G = (V, E)$ peut aussi être représenté par une liste de noeuds de V du graphe en associant à chacun des noeuds la liste de ses noeuds adjacents. Par exemple, soit le graphe représenté comme précédemment, sa liste d'adjacence est celle ci : [1,2,3,4] avec : [1]→ [2,3] ; [2] → [1,3,4] ; [3] → [1,2,4] ; [4] → [2,3].

Notons qu'un graphe orienté peut aussi être représenté à l'aide d'une matrice d'adjacence. En considérant les notations précédentes, on a aura $a_{ij} = 1$ si $(i, j) \in E$ et 0 sinon.

On optera pour une implémentation en python de nos graphes et plus précisément pour la représentation en liste d'adjacence ; que ce soit pour des graphes orientés ou non.
Ainsi, nous implémenterons nos graphes par des dictionnaires ayant pour clé les indices des noeuds du graphes et pour valeurs les listes d'adjacences des noeuds. Dans le cas des graphes orientés, on se dotera de deux dictionnaires ; un pour les listes de prédécesseurs et un autre pour les listes de successeurs. L'avantage de cette implémentation par rapport aux matrices

d'adjacence est la place occupée en mémoire. En effet, soit un graphe $G = (V, E)$, une matrice d'adjacence de taille n prend plus de place mémoire que n listes de taille n au maximum, car dans un graphe réel, il est rare que tous les noeuds soient voisins entre eux et donc que listes d'adjacences d'un graphe réel soient de taille n. On obtient une complexité spatiale en $\Theta(n^2)$ pour la matrice d'adjacence et en $\Theta(n+m)$ pour la représentation en liste d'adjacence.

Complexité des primitives classiques pour la représentation des graphes avec liste d'adjacence :

Dans un graphe non orienté :

existence arête (i, j)	parcours des adjacents de i
$\mathcal{O}(d(i))$	$\Theta(d(i))$

Dans un graphe orienté :

existence arête (i, j)	parcours des successeurs de i	parcours des prédécesseurs de i
$\mathcal{O}(d^+(i))$	$\Theta(d^+(i))$	$\Theta(d^-(i))$

(On suppose disposer de la liste des prédécesseurs et de la liste des successeurs d'un noeud)

3 Algorithmes de recherche de motifs dans des graphes et étude de complexité

3.1 Recherche de motifs triangulaires dans des graphes non orientées

Dans cette partie on cherche à étudier des algorithmes de recherche de motifs dans différents types de graphe.
Pour commencer on recherchera des motifs triangulaires dans des graphes non orientés, puis on s'intéressera dans une prochaine partie aux graphes orientés. Afin d'analyser le comportement de nos algorithmes, on aura besoin de tester ceux-ci sur un grand nombre de graphes de différentes tailles.
Pour cela, on effectuera nos recherches dans deux types de graphes générés aléatoirement : des graphes dit "d'Erdos Renyi" et des graphes générés à l'aide du "configuration model". Ces deux termes seront défini plus tard.
On étudiera en particulier 3 types d'algorithme de recherche et l'on comparera leur efficacité. On calculera leur complexité théorique que l'on comparera à nos résultats expérimentaux. De plus, on s'intéressera aux propriétés spécifiques des graphes d'Erdos Renyi et du *configuration model*.

3.1.1 Étude général des 3 algorithmes de recherche de motifs triangulaires

Algorithme n°1 : Recherche basique

Notre première algorithme de recherche de motifs triangulaire est le plus trivial. Pour chaque noeuds, on regarde sa liste de voisins et on vérifie si le noeud en question et n'importe lequel de ses voisins ont un voisin en commun en faisant un double parcours des deux listes d'adjacences à chaque fois : Si c'est le cas on a un trouvé un triangle. Notre algorithme stocke et renvoie sous forme de liste tous les motifs triangulaires sous forme d'ensemble de la forme $\{i, j, k\}$. Pour s'assurer que l'on ne compte pas deux fois le même motif triangulaire ; que ce soit le motif triangulaire (i, j, k) ou le motif (j, i, k) par exemple, on s'assurera de ne retenir que des triplets de la forme (i, j, k) avec $i < j < k$. Cela est explicité dans le pseudo code ci-dessous. En notant Δ, le degré maximal d'un noeud pour un graphe G, d'après notre pseudo code ; notre première algorithme de recherche de triangle a donc (puisque non orienté) une complexité en $\mathcal{O}(n\Delta^3)$:

Pseudo code :

Algorithm 1: RechercheTriangle n°1

Data: Graphe $G = (V, E)$
Result: retourne l'ensemble des motifs triangles du graphe G

```
for S1 ∈ V do
    for S2 ∈ Γ(S1) do
        if S2 > S1 then
            for S3 ∈ Γ(S2) do
                if S3 ∈ Γ(S1) and S3 > (S2) then
                    res = res ∪ {S1, S2, S3}
return res
```

Algorithme n°2 : Premier algorithme optimisé en triant le graphe

Dans le cas de notre algorithme optimisé, on se propose de trier toutes les listes d'adjacence de notre graphe par ordre croissant et ainsi au lieu de parcourir de manière triviale deux listes pour les comparer, on cherche l'intersection de deux listes triées. La fonction sorted() prédéfinie en python que nous utilisons pour trier nos listes a une complexité en $\mathcal{O}(\Delta \times log(\Delta))$. Donc la complexité de tri du graphe est en $\mathcal{O}(n \times \Delta \times log(\Delta))$ puisque on trie en tout n listes. On doit également déterminer la complexité de notre algorithme de recherche de l'intersection de deux listes triées. Soit une liste triée A et une liste triée B alors le nombre de comparaisons effectués dans le pire des cas est $|A| + |B|$ donc la comparaison est en $O(\Delta)$. Enfin, d'après notre pseudo code la complexité de notre algorithme optimisé est donc en $\mathcal{O}(n\Delta^2 + n\Delta log(\Delta)) = \mathcal{O}(n(\Delta^2 + \Delta log(\Delta)))$ soit en $\mathcal{O}(n\Delta^2)$. En effet, pour chaque noeud, on parcourt seulement sa liste de voisin et on compare pour chaque voisin les deux listes triées de voisins du noeud courant et du noeud voisin ; ce qui donne bien la complexité énoncée.

Pseudo code :

Algorithm 2: RechercheTriangle n°2

Data: Graphe $G = (V, E)$ dont les listes d'adjacences sont triées
Result: retourne l'ensemble des motifs triangulaires du graphe G

```
1 for S1 ∈ V do
2     for S2 ∈ Γ(S1) do
3         if S2 > S1 then
4             intersection ← intersection_liste_triee(Γ(S1), Γ(S2))
5             for S3 ∈ intersection do
6                 if (S3 > S2) then
7                     res = res ∪ (S1, S2, S3)
8 return res
9
```

Algorithme n°3 : Second algorithme optimisé en parcourant les liens

On a également une autre manière de détecter des motifs triangulaires dans un graphe non orienté en « parcourant directement les liens ». C'est à dire que pour un noeud i donné on devrait être capable de déterminer si i est voisin d'un noeud j en parcourant l'ensemble des arcs du graphe. Notre algorithme va donc pour chaque noeud du graphe, marquer chaque noeud de sa liste de voisin et pour chacun de ses voisin, on parcourt aussi sa liste de voisin, ainsi si un des voisins de celui-ci a été marqué alors on a trouvé un motif triangulaire. On s'assure que l'on ne compte pas plusieurs fois les mêmes motifs de la même manière que pour notre second algorithme. On marquera chaque noeud à l'aide d'un dictionnaire ayant pour clé les indices des noeuds et comme valeur : 1 si le noeud a été marqué et un 0 sinon (ou un tableau de booléen t ou $t[i] = 1$ si le noeud i a été marqué et vaut 0 sinon) et puisque la vérification du marquage se fait en $\mathcal{O}(1)$, cet algorithme a donc une complexité en $\mathcal{O}(n\Delta^2)$.

Pseudo code :

Algorithm 3: RechercheTriangle n°3

Data: Graphe $G = (V, E)$
Result: retourne l'ensemble des motifs triangulaires du graphe G

```
for S1 ∈ V do
    for S2 ∈ Γ(S1) do
        Marquage(S2)
    for S2 ∈ Γ(S1) do
        if S2 > S1 then
            for S3 ∈ Γ(S2) do
                if estMarque(S3) and S3 > S2 then
                    res = res ∪ (S1, S2, S3)
return res
```

3.1.2 Étude des algorithmes de recherches de motifs dans des graphes d'Erdos Renyi

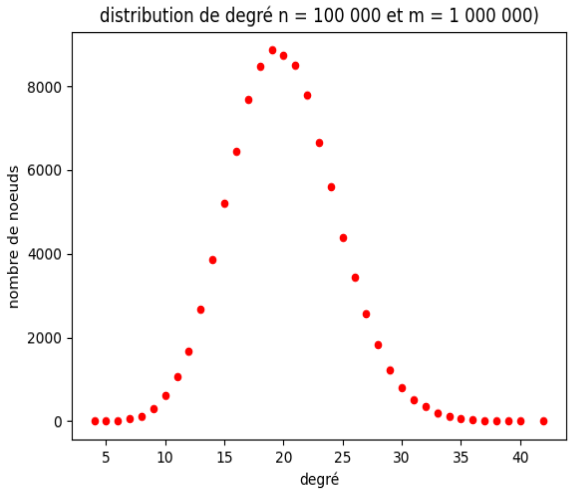
Protocole et Définition

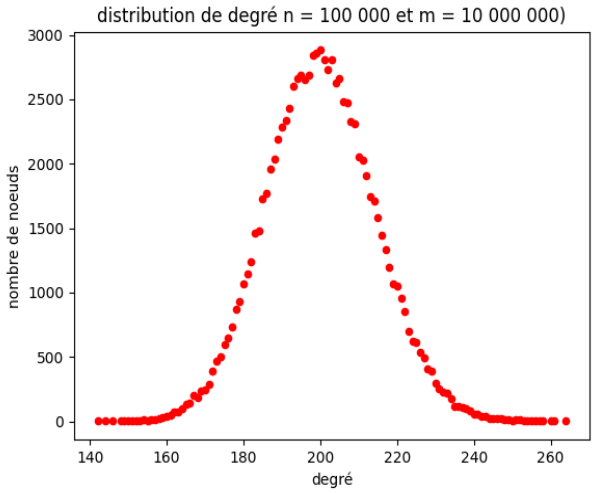
Étudions à présent expérimentalement nos algorithmes sur des graphes générés aléatoirement d'Erdos Renyi en suivant un protocole précis. Le modèle d'Erdos Renyi est le modèle le plus simple et connue permettant de générer des graphes aléatoirement. Il y a deux variantes équivalentes de ce modèle dont le modèle $\mathcal{ER}(n, m)$ qui consiste à générer un graphe à n noeuds à m liens en générant m liens aléatoirement. Pour cela, on choisit aléatoirement m couples de noeuds différents pour former les liens. Pour l'étude on adoptera le modèle $\mathcal{ER}(n, m)$ et on suivra le protocole suivant : on mesurera les temps d'exécution de nos 3 fonctions de recherche de motifs dans des graphes d'Erdos Renyi en faisant évoluer les paramètres n et m, c'est à dire en fixant tout d'abord le nombre de noeud n, puis en fixant le nombre d'arête m du graphe et enfin en fixant la densité d. Mais avant cela, on détermine à quoi correspond Δ dans un graphe de d'Erdos Renyi ou du moins on approxime Δ. Pour cela ; on trace les distributions de degrés de certains graphes d'Erdos Renyi.

Degré moyen et distribution de degré

Dans le modèle d'Erdos Renyi $\mathcal{ER}(n, m)$, les liens sont générés aléatoirement donc les degrés sont répartis de manière uniforme entre les noeuds. On peut donc approximer le degré d'un noeud à celui du degré moyen $\overline{d}$ du graphe, car l'extrême majorité des noeuds auront un degré très proche de $\overline{d}$. Par exemple, si on trace les distributions de degré pour des graphes

d'Erdos Renyi de taille 10^5 à 10^6 arêtes et 10^7 arêtes, on obtient les distributions de degrés suivantes :





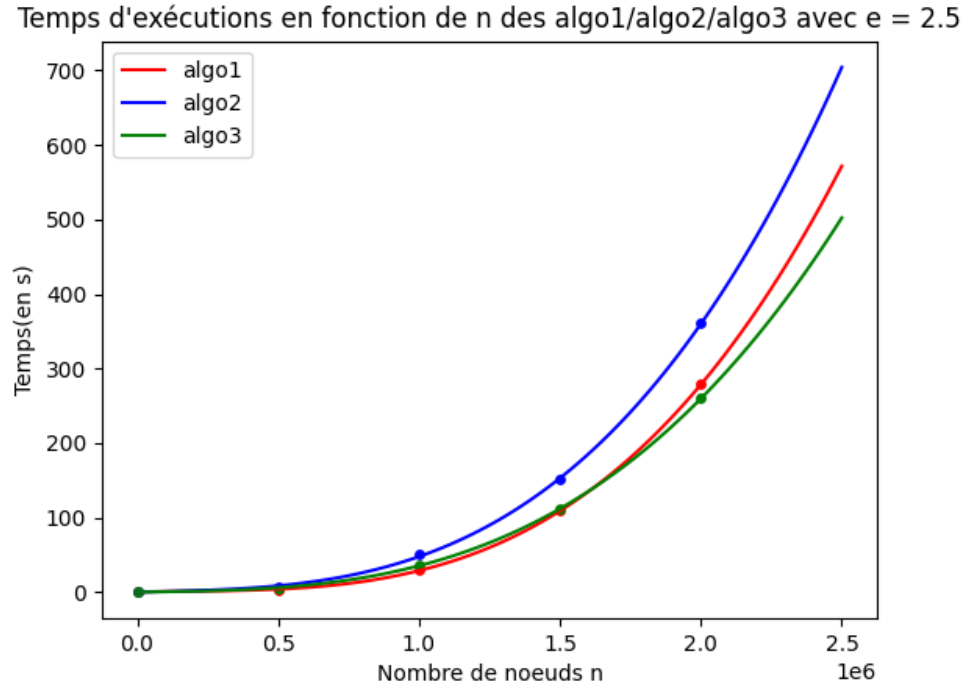
On vérifie bien que $\overline{d}$ est environ égale à $\frac{2*10^6}{10^5} = 20$ pour notre première distribution et $\frac{2*10^7}{10^5} = 200$ pour la seconde. Notre hypothèse sur la distribution de degré d'un graphe d'Erdos Renyi se vérifie donc bien dans ces deux cas là.

Étude de complexité

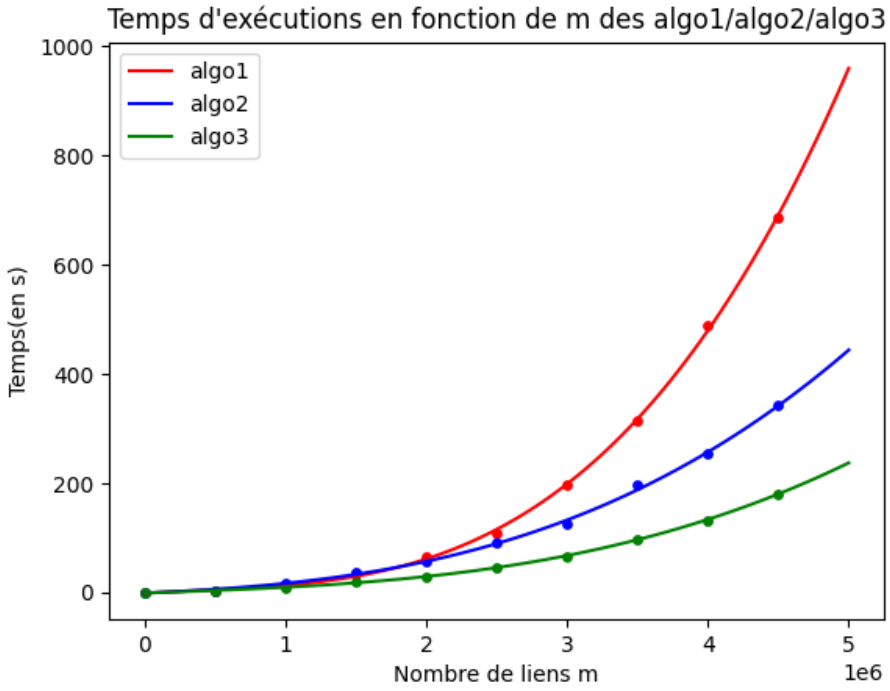
Calculons la complexité pire cas approchée de la recherche de triangles pour notre premier algorithme non optimisé en fonction de n et m. D'après nos calculs de complexité effectuées dans la première partie, en approximant le degré maximum Δ par le degré moyen dans graphe de Erdos Renyi on obtient que cet algorithme a une complexité en $\mathcal{O}(m^3/n^2)$.
En effectuant les mêmes procédés pour nos seconds et dernier algorithme on obtient pour les deux une complexité en $\mathcal{O}(m^2/n)$.

Étude expérimentale et comparaisons des temps d'exécutions

On observe sur la figure (a) qu'à densité fixé, l'algorithme 1 devient plus lent que l'algorithme 3 au alentour de 1.75 million de noeuds. A 2.5 million, il est toujours plus rapide que l'algorithme 2 mais devrait finir par la dépasser. Ceci est en accord avec nos résultats théoriques, à densité fixé, l'algorithme 1 est en $\mathcal{O}(\delta^3 n^4)$ tandis que les algorithmes 2 et 3 sont tout deux en $\mathcal{O}(\delta^2 n^3)$. Comme δ est fixé, l'algorithme 1 finira par être le plus lent pour n fixé. Cependant, pour n assez petit, comme $\delta^2 > \delta^3$, les algorithmes 2 et 3 sont plus lent.
On observe sur la figure (b) qu'à n fixé, l'algorithme 1 dépasse l'algorithme 2 au environ d'un million de liens et l'algorithme 3 à 2 millions de liens. Pour m assez grand (supérieur à 2 millions), l'algorithme n°1 est donc le plus lent suivi du n°2 et donc le n°3 où on "parcourt les liens" du graphes est le plus rapide. Cela est en accord avec nos résultats théoriques. $\mathcal{O}(m^3)$ pour l'algorithme n°1 et $\mathcal{O}(m^2)$ pour les algorithmes n°2 et n°3 (n fixe).



(a) d fixé à 0.00001

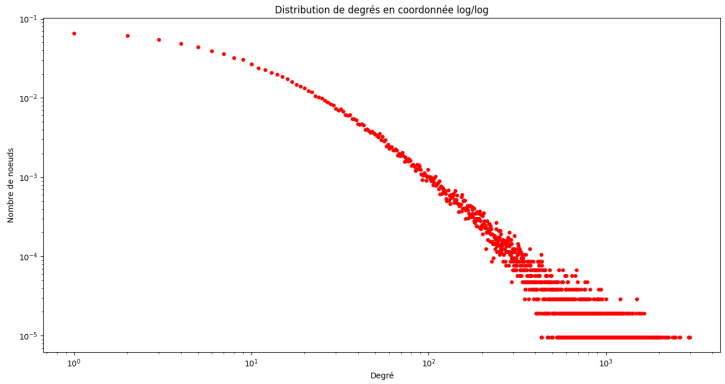


(b) n fixé à 100 000

3.1.3 Étude des algorithmes de recherches avec le "configuration model"

Avec les graphes d'Erdos Renyi, nous obtenons des distributions de degré normales, cependant, comme beaucoup de phénomène physique et biologique, les distributions de degrées des graphes réelles suivent approximativement une loi de puissance. On dit que ce sont des distributions en longue traine (ou *long-tailed* distribution en anglais). C'est le cas par exemple de la distribution du nombre de followers par compte twitter ou du réseau social américain Livemocha ou encore de la distribution du nombre de connexions des aéroports dans le monde.

Exemple :



(a) Distribution de degrés du graphe du réseau Livemocha

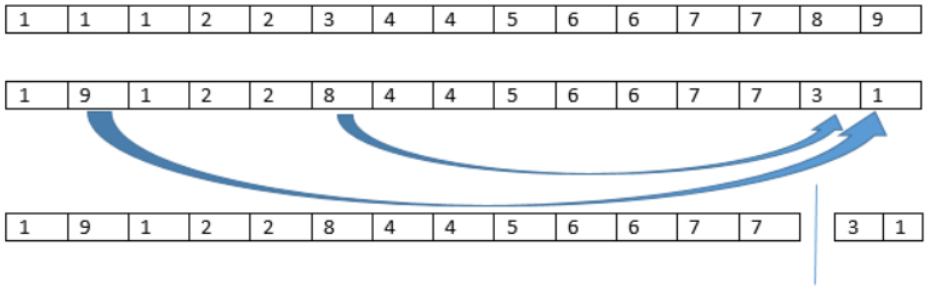
A l'aide du configuration model, une méthode de génération de graphe aléatoire à partir d'une distribution de degré donnée (dans notre cas, un loi de puissance), nous pourrons donc tester nos algorithmes de recherche de motifs triangulaires sur des graphes ayant des distributions similaires aux graphes réels. On utilisera le même protocole que que pour les graphes Erdos Renyi.

Génération de distribution de degré et algorithme du configuration model

Nous devons donc dans un premier temps générer des distributions de degré suivant une loi de puissance. Pour cela (et uniquement pour cela), nous utiliserons networkx, un module python A partir d'une distribution de degré, on peut alors créer un graphe de la manière suivante :
On crée un tableau où chaque noeud est représenté dans un nombre de cases égale à son degré.
On échange ensuite les deux dernières cases avec deux cases aléatoires, les deux cases ainsi

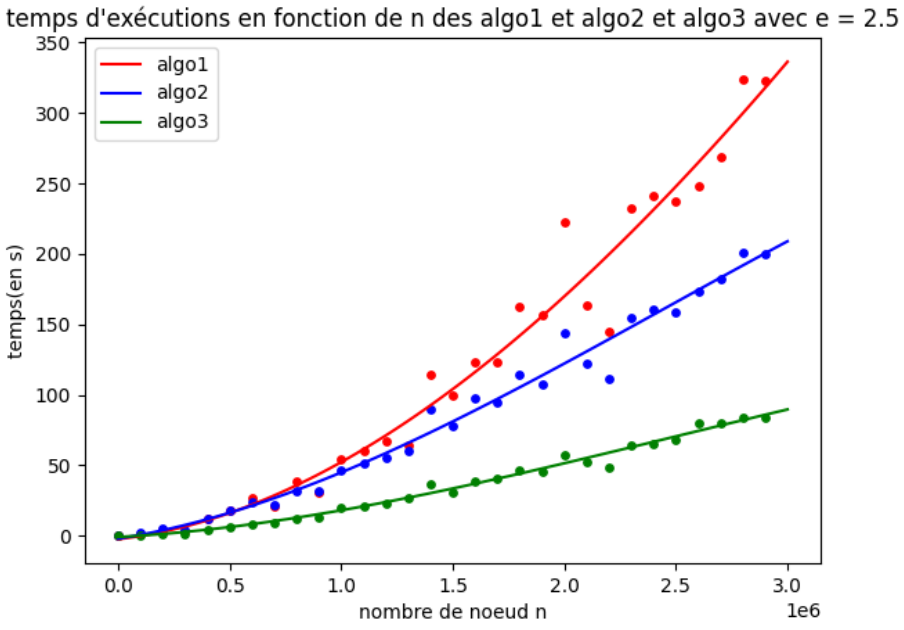
sélectionnées représentent alors un lien entre les deux noeuds.
On recommence le processus avec le tableau privé de ses 2 dernières cases.
Enfin, on supprime les doublons et les liens entre un noeud et lui-même et on a terminé.

Exemple :



Étude expérimentale

On peut donc tracer les 3 courbes des temps d'exécutions en fonction de n avec l'exposant fixé à 2.5 pour nos 3 algorithmes avec le *configuration model* et on obtient le graphique suivant :



On observe que notre algorithme n°3 est plus rapide que de nos deux autres algorithmes de recherche, cependant il y plus de fluctuations notamment si on compare les courbes des algorithmes n°1 et n°2. En effet, contrairement au modèle d'Erdos-Renyi, la distribution en

longue traine entraîne que certains noeuds possède beaucoup plus de liens que la moyenne et sont donc plus long à traiter. Si ces noeuds sont beaucoup appelé lors de la recherche de triangles, alors le traitement de ce graphe sera plus long que ce que le nombre de noeuds laisserai supposer (et pareillement, si de telles noeuds sont peu appelés, le traitement sera plus rapide qu'attendu).

3.2 Recherche de triades dans des graphes orientés

Dans cette partie, on s'intéresse à la recherches de motifs orientés dans des graphes orientés, ici les motifs recherchés sont des triades.

On procédera pareillement à une étude de complexité que nous comparerons à une étude expérimentale, sauf que on utilisera seulement des graphes du *configuration model* puisqu'ils sont plus proche des graphes réels que l'on veut étudier.

Cependant, comme nous travaillons maintenant avec des graphes orientés, ils nous faut un moyen de les générer. En effet, nous voulons conserver à la fois les arcs sortants de chaque noeuds, mais aussi ses arcs entrants, pour largement simplifier le parcours des prédécesseurs d'un noeud

Dans notre cas on s'intéressera seulement à la recherche d'une triade où ils existent un arc entrant et sortant entre chaque noeud avec les deux autres ; c'est à dire à la triade n°13 de la figure en introduction. On note que l'on pourrait facilement adapter l'algorithme pour trouver une autre sorte de triades.

3.2.1 Étude de complexité dans le cas général

Notre première algorithme pour détecter ce dernier type de triade se rapproche de celui de la recherche d'un motif triangulaire dans un graphe non orienté. L'algorithme que nous utiliserons est parfaitement explicité dans le pseudo code suivant.

Pseudo code :

```
Data: G = (V,E)
Result: Retourne l'ensemble des triades du graphe G
1 res ← {}
2 for S1 in V do
3     for S2 in Γ+(S1) do
4         if S1 ∈ Γ+(S2) then
5             for S3 ∈ Γ+(S2) do
6                 if S2 ∈ Γ+(S3) and S1 ∈ Γ+(S)3 and S3 ∈ Γ+(S1) then
7                     res ← res ∪ (S1,S2,S3)
8 return res
```

Étude de complexité :

Notons Δ^+ le degré entrant maximum et Δ^- le degré sortant maximum. Alors étant donné que la complexité de la primitive "in list" pour savoir si un élément est présent dans une liste est en O(n) pour une liste de taille n, d'après notre pseudo code notre algorithme a une complexité en $\mathcal{O}(n(\Delta^+)^3)$. On remarque qu'on pourrait effectuer des optimisations similaires au cas non orienté (Par exemple un marquage)

3.2.2 Étude expérimentale

On trace alors la courbes des temps d'exécutions de notre algorithme de recherches de triades en fonction de n et on remarque une courbe très semblable au cas non orienté. On déduit que comme attendu, la complexité reste dans le même ordre de grandeur.

4 Étude des algorithmes sur des graphes réels

On s'intéresse maintenant aux graphes réels. On veut pouvoir déterminer si un motif est sur ou sous représenté dans un graphe réels donné par rapport à des graphes aléatoires ayant la même distribution de degré. Pour cela on compare le nombre de motifs présent dans le graphe à son nombre dans un graphe généré aléatoirement. On utilise comme outil le Z-score.

Le Z-score

On calcule le Z-score d'un graphe de la manière suivante :

$$Z_i = \frac{N_{reali} - < Nrand_i >}{std(Nrand_i)}$$

où $Nreal_i$ est le nombre de motifs "i" du graphe réel, $< Nrand_i >$ est la moyenne du nombre de motifs "i" présent dans les graphes aléatoires généré et $std(Nrand)$ est l'écart type des graphes aléatoires généré. Les Z-score obtenu ainsi ont cependant des amplitudes différentes en fonction du nombre de noeuds du graphe. Pour comparer des graphes de tailles différentes, il faut donc normaliser ce Z-score, pour obtenir le profil d'importance (ou *significance profile* en anglais, abrégé en SP). On le calcule de la manière suivante :

$$SP_i = \frac{Z_i}{\sqrt{\sum Z_i^2}}$$

Avec le SP, on a un score allant de -1 à 1 traduisant de la sur ou sous présence d'un motif dans un graphe par rapport au graphe aléatoire, que l'on peut utilisé pour comparer entre eux différent graphe réels.

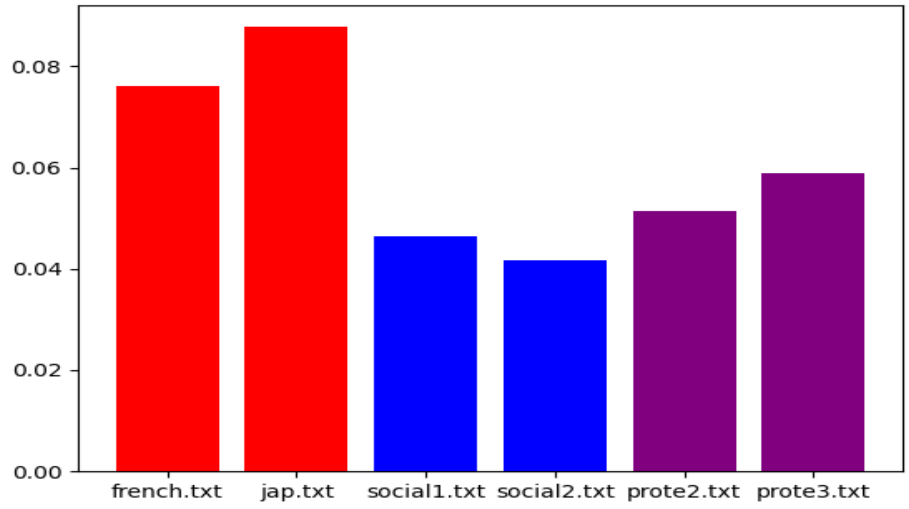
Etude expérimentale

Calculons le Z-score et le SP de plusieurs graphes réels. Nous nous limiterons à la recherche de triangles dans le cas non orienté. Mais on peut facilement s'adapter au cas orienté et même à des motifs à plus de 3 noeuds.

On comparera chaque graphe réel à 100 *configuration model* généré à partir de sa distribution de degré. Parmi les graphes que l'on utilisera, certains modélise des choses proches : Les graphes french.txt et jap.txt modélisent tout les deux l'adjacence des mots dans des phrases, tandis que social1 et social2 modélisent tout deux des réseaux sociaux et enfin prote2 et prote3 modélisent la structure des protéines. On dira qu'il sont de la même famille.

nom	n	m
french.txt	8317	24295
jap.txt	2701	8300
social1.txt	67	182
social 2.txt	32	96
prote2.txt	53	213
prote3.txt	96	123

SP des différents graphes :



On remarque que les graphes de même famille (apparaissant de la même couleur sur le graphe) ont des SP très similaire mais différent de ceux d'une autre famille. On remarque aussi que quelque soit le type de graphe, le SP est positif, ce qui semble montrer qu'il existe plus de triangles dans un graphe réel que dans un graphe généré aléatoirement de même distribution de degrée.

Il serait intéressant d'étendre le travail aux graphes orientés. En effet, en calculant le SP d'un graphe pour chacune des 13 triades, on pourrait obtenir un profil beaucoup plus précis

de ce graphe (On appelle ce profil le TSP du graphe pour *Triad Significance Profile*).
Ce travail est fait et developpé dans l'article de Milo et al :*"Superfamilies of Evolved and Designed Networks"*

5 Annexes

Lien vers le code du projet : https ://github.com/melLaflame/projet-motifs-dans-les-graphes