

Rapport du Projet Algorithmique

Outils pour la conception d'algorithmes

Détection de communautés dans des réseaux sociaux

Réalisé par :

GHANNOUM Jihad - IATIC 4

KHIARI Slim - IATIC 4

NOUIRA Nessrine - IATIC 4

TOIHIR Yoa - IATIC 4

M. Manoussakis George : Encadrant

Mme Cohen Johanne : Encadrante

Projet IATIC4 effectué du 25/09/2021 au 14/10/2021

TABLE DES MATIÈRES

| | |
|--|-----------|
| Introduction | 3 |
| Partie 1 : Cahier des charges | 4 |
| 1.1 Description du projet : | 4 |
| 1.2 Technologies et outils utilisés : | 4 |
| 1.3 Organisation du groupe : | 4 |
| 1.4 Structure de données d'un graphe : | 5 |
| Partie 2 : Algorithmes de génération de graphes aléatoires | 6 |
| 1.1 Génération aléatoire des arêtes du graphe : | 6 |
| 1.2 Barabasi-Albert : | 6 |
| 1.2.1 Présentation théorique : | 6 |
| 1.2.2 Description de l'implémentation : | 6 |
| 1.1 Version sans pivot : | 7 |
| 1.1.1 Présentation théorique : | 7 |
| 1.1.2 Description de l'implémentation : | 7 |
| 1.2 Version avec pivot : | 7 |
| 1.2.1 Présentation théorique : | 7 |
| 1.2.2 Description de l'implémentation : | 8 |
| 1.3 Version avec ordonnancement des sommets : | 9 |
| 1.3.1 Présentation théorique : | 9 |
| 1.3.2 Description de l'implémentation : | 9 |
| Partie 3 : Algorithmes d'énumération des cliques en fonction de la dégénérescence | 11 |
| 1.1 Algorithme en fonction de la dégénérescence : | 11 |
| 1.1.1 Présentation théorique : | 11 |
| 1.1.2 Description de l'implémentation : | 11 |
| 1.2 Algorithme en fonction de la dégénérescence et du degré maximal : | 12 |
| 1.2.1 Présentation théorique : | 12 |
| 1.2.2 Description de l'implémentation : | 12 |
| Partie 4 : Comparaison de graphes en terme de temps et d'espace d'exécution | 13 |
| Exemple d'exécution des algorithmes | 14 |
| Conclusion | 19 |
| Bibliographie | 20 |

Introduction

Dans ce rapport, nous présenterons le problème d'énumération de cliques maximales dans un graphe non-orienté. Ce dernier a de nombreuses applications, notamment dans les réseaux sociaux. Ainsi, après avoir présenté le problème en détail, nous verrons plusieurs algorithmes le résolvant de manière plus ou moins efficace. Nous les comparerons ensuite afin de voir si leurs performances empiriques sont les mêmes que celles auxquelles on peut s'attendre théoriquement.

Partie 1 : Cahier des charges

1.1 Description du projet :

Dans le cadre de notre deuxième année du cycle ingénieur à l'ISTY, il nous est proposé un projet de 6 semaines. Ce projet nous permettra de mettre en pratique nos connaissances, nos compétences professionnelles et surtout de développer l'esprit d'équipe à travers un cahier de charges ayant pour finalité le développement d'un outil qui vise à détecter les communautés dans les réseaux sociaux, c'est-à-dire des ensembles de nœuds très densément connectés entre eux.

En théorie des graphes, on modélise souvent les communautés par des cliques maximales.

Soit $G = (V, E)$ un graphe avec V : l'ensemble des sommets et E : l'ensemble des arêtes de G . Une clique K de G est un ensemble de sommets du graphe tous connectés deux à deux.

On dit que la clique K est maximale si elle n'est incluse dans aucune autre clique de G , c'est-à-dire qu'il n'existe pas de sommet dans $V \setminus K$ connecté à tous les sommets de K .

Le problème de décision de la clique est NP-complet (c'est l'un des 21 problèmes NP-complet de Karp). Le problème de trouver la clique maximale est à la fois de complexité paramétrée intraitable et difficile à approximer. Lister toutes les cliques maximales peut nécessiter un temps exponentiel car il existe des graphes avec un nombre exponentiel de cliques maximales. Par conséquent, une grande partie de la théorie sur le problème de la clique est consacrée à l'identification de familles particulières de graphes qui admettent des algorithmes plus efficaces, ou à établir la difficulté de calcul du problème général dans divers modèles de calcul.

1.2 Technologies et outils utilisés :



Le développement de ce projet sera réalisé en langage Python sous l'IDE Pycharm et en utilisant les deux bibliothèques **NetworkX** et **Matplotlib**.




NetworkX est une bibliothèque Python qui permet d'instancier des graphes composés de nœuds et des arêtes. Grâce à cette bibliothèque, la manipulation de ces graphes est simplifiée.

Matplotlib est une bibliothèque destinée à tracer et visualiser des données sous formes de graphiques.

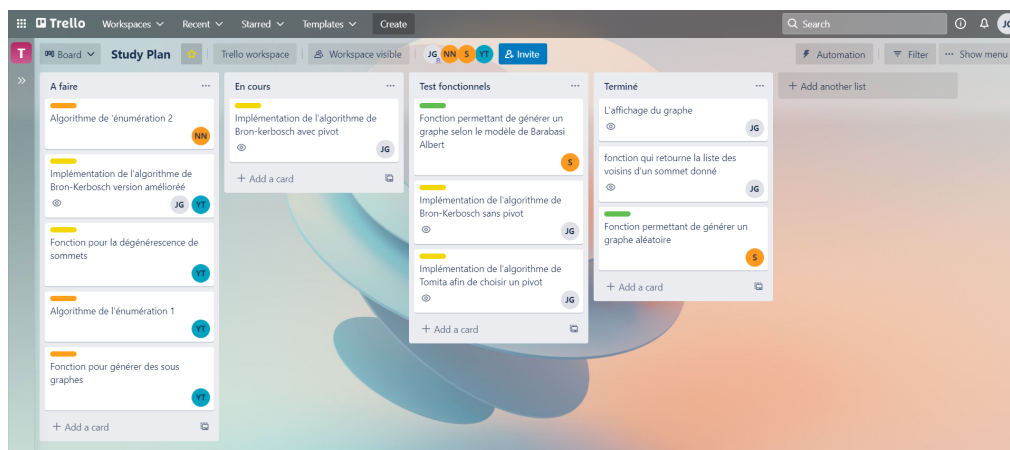
1.3 Organisation du groupe :

Pour pouvoir réaliser le projet en groupe, nous allons utiliser les outils de gestion de projet : **Trello et GitHub.**

Trello est un outil de gestion de projet en ligne qui repose sur une organisation des projets en planche listant des cartes chacune représentant des tâches. Les cartes sont attribuées à des utilisateurs et sont mobiles d'une planche à l'autre, traduisant leur avancement.

Le vert  correspond à la première partie du projet, le jaune  à la deuxième partie, et l'orange  à la troisième.

Notre planche Trello se présente comme ceci :



GitHub est un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git.

Nous utiliserons GitHub afin de disposer du même niveau d'avancement du projet tout au long de sa réalisation.



1.4 Structure de données d'un graphe :

Pour pouvoir modéliser un graphe dans le langage de programmation que nous avons choisi, nous avons décidé de représenter un graphe en tant qu'objet auquel est associé de multiples fonctions.

Un graphe est algorithmiquement une liste d'adjacence qui est représentée dans le langage Python, sous forme de dictionnaire qui est un type de collection associant une clé à une valeur. Les clés sont dans notre cas les sommets et les valeurs associées sont les voisins de ces sommets.

Toutes les fonctions liées aux graphes ont été développées dans la classe correspondante.

Partie 2 : Algorithmes de génération de graphes aléatoires

Dans cette partie, nous allons présenter deux façons de générer des graphes aléatoires. D'abord, nous allons voir une méthode "classique" que nous détaillons dans la partie 1.1 en considérant toutes les arêtes pouvant se former, ensuite la méthode de Barabasi Albert que nous détaillons dans la partie 1.2.

1.1 Génération aléatoire des arêtes du graphe :

Dans l'algorithme qu'on propose, on ajoute une arête dans le graphe à former aléatoirement à partir d'un nombre aléatoire qu'on appelle "probabilité".

Si "probabilité" est inférieur à 1 et supérieur à 0, on ajoute le sommet voisin dans la liste d'adjacence du sommet concerné. On répète ce processus pour tous les sommets du graphe.

Par la suite, on effectue une mise à jour afin d'avoir les mêmes sommets dans la liste d'adjacence du sommet voisin et celle du sommet concerné.

1.2 Barabasi-Albert :

1.2.1 Présentation théorique :

Le modèle de génération de graphe aléatoire Barabasi-Albert fait apparaître des sommets fortement connectés au graphe.

On dit que ce modèle génère des réseaux "sans échelle".

Le réseau commence par un graphe complet de m_0 nœuds (dans notre cas $m_0 = 3$).

Ensuite, de nouveaux nœuds sont ajoutés au réseau. Chaque nœud est connecté à $m \leq 3$ nœuds existants ont déjà.

Le nouveau nœud est connecté à un nœud i avec la probabilité suivante :

$$p_i = \frac{\text{degré}(i)}{\Sigma \text{degré}}$$

1.2.2 Description de l'implémentation :

Les m nouvelles arêtes à ajouter sont données en paramètres de la méthode.

On commence par initialiser le graphe avec un graphe triangle c'est-à-dire une clique à trois nœuds.

On calcule la somme des degrés des nœuds du graphe et on parcourt ces derniers.

Pour chaque nœud i , on compare la probabilité p_i à un nombre aléatoire "random".

Si "random" est strictement inférieur à p_i alors on ajoute le nœud i dans le graphe.

On répète ce fonctionnement pour chaque m .

Partie 3 : Algorithmes de Bron-Kerbosch

L'algorithme de Bron-Kerbosch est un algorithme d'énumération de retour sur trace récursif utilisé pour trouver toutes les cliques maximales d'un graphe non orienté. Il est connu qu'un graphe à n sommets possède au maximum $3^{n/3}$ cliques maximales. Ceci donne donc une borne inférieure en temps de $O(1.44^n)$ dans le pire des cas.

Dans cette partie, nous expliquons l'idée de base de cet algorithme. En outre, nous présentons deux techniques permettant d'améliorer le temps d'exécution.

1.1 Version sans pivot :

1.1.1 Présentation théorique :

Étant donné trois ensembles de nœuds disjoints P , R et X , cet algorithme cherche toutes les cliques maximales contenant tous les sommets de R , certains de P , mais aucun de X .

- P : l'ensemble des sommets candidats (qui n'ont pas été encore pris en compte) pour être ajoutés à la potentielle clique.
- R : un sous-ensemble des sommets de la potentielle clique.
- X : l'ensemble contenant les sommets déjà traités ou appartenant déjà à une clique.
- $P \cup X$: l'ensemble de tous les sommets qui sont adjacents à chaque sommet de R .

L'ensemble X est utilisé pour éviter d'énumérer les cliques maximales plus d'une fois.

1.1.2 Description de l'implémentation :

À chaque appel récursif, l'algorithme va parcourir les sommets de P l'un après l'autre; si P est vide, soit X est vide et R est renvoyé en tant que clique maximale, soit l'algorithme retourne sur ses traces.

Pour chaque sommet v de P , l'algorithme fait un appel récursif dans lequel v est ajouté à R , et P et X sont restreints à l'ensemble des voisins de v ; cet appel retourne toutes les extensions de cliques de R qui contiennent v . Puis v est déplacé de P à X pour que ce nœud ne soit plus pris en considération dans les futures cliques.

1.2 Version avec pivot :

1.2.1 Présentation théorique :

La version sans pivot est inefficace pour les graphes qui contiennent beaucoup de cliques maximales, pour cela Bron et Kerbosch ont également introduit une optimisation pour leur algorithme de base en choisissant un élément pivot pour réduire le nombre d'appels récursifs et donc permettre un retour en arrière plus rapide.

L'idée est basée sur l'observation que pour tout sommet $[u \in P \cup X]$, le pivot soit le sommet u lui-même, soit l'un de ses non-voisins $[P \cup X \setminus N(u)]$ doit être contenu dans toute clique contenant R .

Si ni u ni aucun de ses non-voisins ne sont inclus dans une clique maximale contenant R , cette clique ne peut pas être maximale, car u peut être ajouté à cette clique en raison du fait que seuls les voisins de u ont été ajoutés à la clique originale R .

Par conséquent, choisir un pivot arbitraire $u \in P \cup X$ et itérer juste sur u et tous ses non-voisins diminuent le nombre d'appels récurifs dans la boucle 'for' qui mènent à des cliques non maximales. L'algorithme **Pivot_tomita** a montré que dans le cas où u est choisi dans $P \cup X$ de manière à ce que u ait le maximum de voisins, alors le temps d'exécution pour un graphe $G = (V, E)$ est en $O(3^{|V|/3})$.

1.2.2 Description de l'implémentation :

Pour implémenter l'algorithme Bron-Kerbosch avec pivot, nous allons devoir implémenter un algorithme qui permet de choisir un pivot [**Pivot_tomita**].

L'algorithme Pivot-Tomita prend en entrée deux ensembles (P et X) et retourne un sommet pivot u où $|P \cap N(u)|$ est maximale.

La première partie consiste à initialiser un sommet u (pivot) au premier élément de la liste $P \cup X$, puis à initialiser une variable `degre_max` à la taille de la liste $P \cap N(u)$.

À la deuxième étape, on parcourt tous les sommets de la liste $P \cup X \setminus \{u\}$ afin de vérifier si la taille de la liste $P \cap N(v)$ [v : sommet en cours] est supérieure à la valeur initiale de la variable `degre_max`. Si c'était le cas, on réaffecte le sommet u à v et `degre_max` à $|P \cap N(v)|$.

Enfin, on retourne un pivot u avec $|P \cap N(u)|$ maximale

Nous disposons désormais de l'algorithme donnant un pivot u , nous allons pouvoir l'utiliser pour implémenter la version avec pivot.

Nous commençons par vérifier si la clique R donnée est maximale ou non, si $|P \cup X| = 0$, alors il n'y a aucun sommet qui peut être ajouté à la clique. Par conséquent, la clique est maximale et peut être ajoutée à la solution. Sinon, la clique n'est pas maximale et on va créer une variable *pivot* et on lui affecte le sommet retourné par l'algorithme **pivot_tomita**

Pour chaque sommet dans la liste $P \setminus N(pivot)$, on va réaliser un appel vers l'algorithme Bron-Kerbosch avec pivot. Nous utilisons le mot-clé "**yield from**" en Python qui permet d'enregistrer à chaque itération les cliques maximales retournées par l'algorithme appelé puis à retourner toutes ces données une fois l'algorithme terminé.

Nous retiendrons à chaque itération le sommet traité de P et nous allons ajouter ce même sommet à X .

1.3 Version avec ordonnancement des sommets :

1.3.1 Présentation théorique :

Nous pouvons utiliser une méthode alternative pour améliorer l'algorithme de Bron Kerbosch. Cette méthode consiste à mettre de côté la stratégie impliquant le pivot au niveau externe de la récursion de Bron-Kerbosch sans pivotement et d'ordonner l'ordre dans lequel les sommets d'un graphe sont traités récursivement par l'algorithme à ce même niveau. On appelle cet ordre, l'ordre de dégénérescence.

La dégénérescence correspond à la plus petite valeur d tel que tout sous-graphe G , contient un sommet dont le degré maximal est égal à d .

L'ordre de dégénérescence correspond par conséquent à un ordre tel que chaque sommet a d ou moins de d voisins qui apparaissent plus tard dans l'ordre.

Cette version énumère toutes les cliques maximales en un temps $O(d(n-d)3^{d/3})$ pour un graphe détenant n sommets. La complexité en espace s'élève à $O(n+d\Delta)$, Δ étant égal au degré maximal du graphe.

1.3.2 Description de l'implémentation :

Pour pouvoir implémenter l'algorithme Bron-Kerbosch en utilisant l'ordre de dégénérescence nous allons devoir implémenter l'algorithme permettant d'obtenir l'ordre de dégénérescence d'un graphe G .

L'algorithme permettant d'obtenir l'ordre de dégénérescence retourne une liste L qui contiendra la liste des sommets.

La première étape consiste à initialiser une liste L vide, puis à initialiser une liste D qui contiendra des sommets stockés à l'indice correspondant à leur degré. On finit par initialiser un entier k à 0 qui représentera, à la fin de l'algorithme, la dégénérescence du graphe.

À la deuxième étape, on parcourt la liste D jusqu'à trouver une cellule de D laquelle il y a des sommets présents. Une fois trouvée, on modifie la valeur de notre entier k à la valeur maximale entre l'indice de la cellule sur laquelle on s'est arrêtée et la valeur actuelle de k .

Ensuite, on sélectionne aléatoirement un sommet v dans la case de D dans laquelle nous trouvons actuellement. On ajoute v au début de la liste L et on le retire de la cellule de D dans laquelle il se trouve. Pour tous les voisins w de v qui ne sont pas encore dans la liste L , on leur retire un degré puis on les déplace à leur nouvel indice correspondant dans D . Pour réaliser cela, on récupère le nombre de leurs voisins dans la liste d'adjacence du graphe, en retirant les sommets qui sont déjà dans L hormi v . Nous incluons v dans la liste des voisins car nous venons tout juste de l'ajouter à la liste L et nous cherchons à récupérer les indices actuels des voisins w de v pour pouvoir ensuite les supprimer de D à ces indices. Pour filtrer les voisins de w , nous avons utilisé la fonction "*filter()*" de Python qui permet de filtrer une liste en fonction de certaines conditions.

Après avoir récupéré la liste des voisins filtrés d'un sommet, on peut déduire, à partir de la taille de la liste, son degré et par conséquent son indice dans D.

Nous pouvons désormais retirer les voisins w de leurs indices actuels dans D puis on les ajoute ensuite dans l'indice juste en dessous de leur ancien indice dans D.

On répète la seconde étape n fois, n étant égal au nombre de sommets du graphe.

Enfin, on retourne la liste L contenant la liste des sommets respectant l'ordre de dégénérescence.

Nous disposons désormais de l'algorithme donnant l'ordre de dégénérescence d'un graphe, nous allons pouvoir l'utiliser pour implémenter la version améliorée de l'algorithme de Bron-Kerbosch.

Nous commençons par initialiser une liste P qui correspond aux sommets du graphe passé en paramètres. On initialise ensuite une liste vide X.

On crée une liste L et on lui affecte la liste retournée par l'algorithme retournant la liste des sommets dans un ordre de dégénérescence.

Pour chaque sommet dans cette liste L, on va réaliser un appel vers l'algorithme Bron-Kerbosch avec pivot. On va utiliser le mot-clé "*yield from*" en Python.

On retire à chaque itération le sommet traité de P et on va ajouter ce même sommet à X.

Partie 3 : Algorithmes d'énumération des cliques en fonction de la dégénérescence

Il existe d'autres algorithmes d'énumération de cliques maximales mis à part les algorithmes de Bron-Kerbosch.

Les deux algorithmes que nous allons présenter dans cette partie dépendent de la dégénérescence du graphe.

1.1 Algorithme en fonction de la dégénérescence :

1.1.1 Présentation théorique :

Le premier algorithme consiste à retourner toutes les cliques maximales sans duplication en utilisant le fait que chaque clique maximale soit prise en compte une seule fois et qu'elle soit un sous-graphe d'un seul et même graphe G_i , G_i étant lui-même un sous-graphe de G . Les sommets des cliques maximales sont stockés dans une table de hachage sans répétition.

Le temps d'énumération repose totalement sur la dégénérescence du graphe.

Cette version énumère toutes les cliques maximales en un temps $\alpha O(f(k+1)q)$ avec q la taille maximale d'une clique, α correspondant au nombre de cliques maximales et $f(k+1)$ étant équivalent au temps d'énumération d'une clique maximale d'un graphe détenant un ordre de dégénérescence égal à $k+1$. La complexité en espace quant à elle est en $O(\alpha q)$.

1.1.2 Description de l'implémentation :

La première partie de l'implémentation de l'algorithme est centrée sur le calcul de la dégénérescence du graphe passé en paramètre.

On initialise une table de hachage qui contiendra la liste d'adjacence du graphe en suivant l'ordre de dégénérescence.

On initialise ensuite une table de hachage vide nommée T , dans laquelle on stockera les cliques maximales puis on crée un nouveau graphe à partir de la liste d'adjacence respectant l'ordre de dégénérescence.

On va récupérer tous les sous-graphes du graphe sur lequel nous travaillons, à l'aide d'une fonction de génération de graphe que nous avons implémentée. Cette fonction prend en paramètre une liste de sommets du graphe qui n'ont pas encore été traités et qui n'appartiennent par conséquent à aucun sous-graphe, la liste des sous-graphes que l'on a pour l'instant et les voisins des sommets traités. La fonction commence par vérifier si la liste des sous-graphes que l'on a est vide, si c'est le cas, alors on sélectionne un sommet v parmi les sommets du graphe qui n'ont pas encore été traités. Sinon, la sélection est réalisée dans la liste retournée entre l'intersection entre les sommets non-traités et les voisins des sommets traités. Ensuite, on va appeler deux fois récursivement la fonction de génération de graphes, une fois avec en paramètres la liste des sommets traités sans celui qu'on vient tout juste de sélectionner, avec la même liste de sous-graphes et la même liste de voisins, et une deuxième fois avec la liste des sous-graphes avec le sommet v ajouté et la liste des voisins avec les voisins de v en plus.

Le système se répète jusqu'à ce qu'il n'y ait plus de sommet à traiter, on retourne alors la liste des sous-graphes.

La deuxième étape commence en parcourant tous les sous-graphes de G . Pour chaque sous-graphe G_i de G on va calculer toutes ses cliques maximales. Pour toutes les cliques maximales de G_i , on va trier ses sommets en suivant l'ordre de dégénérescence du graphe G . Pour réaliser cela, on va faire l'intersection entre l'ordre de dégénérescence du graphe G et la liste des sommets de la clique en respectant l'ordre du premier paramètre.

Ensuite, on va vérifier si la clique maximale que l'on traite est déjà incluse dans la table de hachage T . Si la clique est déjà présente alors on la rejette, sinon on insère la clique en tant que "*valeur*" et le numéro du sous-graphe G_i en tant que "*clé*".

Enfin, on retourne la table de hachage pour la récupérer.

1.2 Algorithme en fonction de la dégénérescence et du degré maximal :

1.2.1 Présentation théorique :

Cet algorithme représente une version modifiée du précédent algorithme, au lieu d'utiliser une table de hachage, l'algorithme va calculer si la clique est bien maximale dans le graphe G . On peut montrer que cela est le cas pour une clique maximale de G_i lorsque tout sommet de la clique n'a pas de voisin d'ordre inférieur à v_i dans σ qui soit connecté à tous les sommets de la clique. Cet algorithme fonctionne donc sur le même principe que le précédent, mais va effectuer cette vérification au lieu de tester sur la table de hachage.

1.2.2 Description de l'implémentation :

Pour l'implémentation du deuxième algorithme d'énumération, nous avons commencé par la récupération de degré maximum de graphe G qui a une importance dans la complexité de notre algorithme.

Par la suite, l'implémentation de cet algorithme est semblable à celle de l'algorithme précédent. Cependant nous avons fait directement le calcul de toutes les cliques maximales à la place de la table de hachage.

Tout d'abord, nous avons développé deux fonctions "*verifier_adjacence*" et "*verifier_rank_adjacence*" pour les employer dans notre algorithme. La première fonction vérifie qu'un sommet x est adjacent à un autre sommet dans la liste de sommets. La deuxième fonction vérifie les sommets qui ont un voisin commun d'ordre inférieur, et aussi l'adjacence des sommets.

Ensuite nous avons parcouru toutes les cliques maximales: pour chaque clique, nous parcourons tous ses sommets et à son tour elle va parcourir tous les voisins du sommet S_i .

Pour vérifier les conditions d'algorithme :

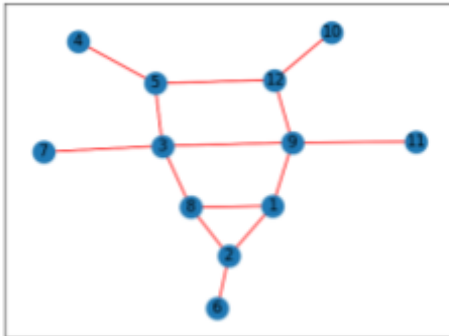
Si les sommets ont un voisin commun d'ordre inférieur dans σ qui est adjacent à tous les sommets de la clique, nous allons utiliser les deux fonctions qu'on a créées pour tester si les conditions sont vérifiées, on sort des deux boucle for c'est à dire que la clique n'est pas maximale alors on rejette afin d'éviter les tests inutiles, sinon on insère la clique dans la liste des cliques maximales.

Intéressant à la complexité de cet algorithme : la complexité en temps pour la première partie de calcul de la dégénérescence de G et les listes d'adjacences dégénérées qui coûtent

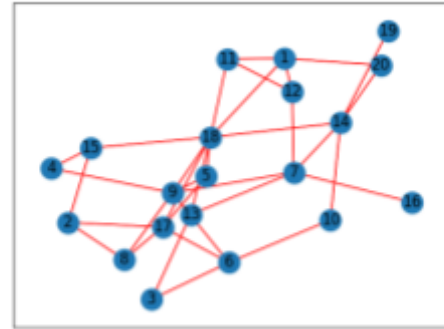
chacunes $O(m)$ temps est la même que la complexité de l'algorithme d'énumération précédent.

Mais pour la partie d'énumération, ça dépend de nombre de clique maximale car pour chaque clique maximale K calculée dans les graphes $G_i / i \in [n]$, si ses sommets ont un voisin commun de "rank" inférieur dans σG au sommet donc la complexité $O(\Delta)$ selon le nombre sommets à vérifier. La complexité en temps pour tout l'algorithme $O(\infty f(k+1)qK\Delta)$. Concernant la complexité en espace: l'algorithme parcourt chaque graphe G_i de manière itérative. Nous avons donc juste besoin de l'espace pour stocker les ensembles de sommets de chaque $G_i / i \in [n]$, qui est de taille au plus $O(k)$, itérativement, et l'espace pour trouver chaque clique dans un graphe d'ordre $k + 1$.

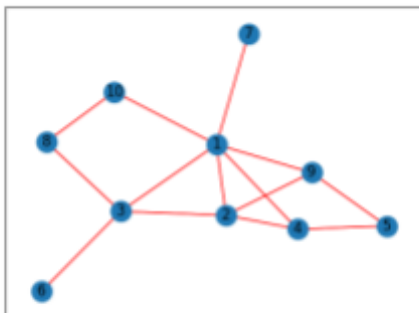
Partie 4 : Comparaison de graphes en terme de temps et d'espace d'exécution



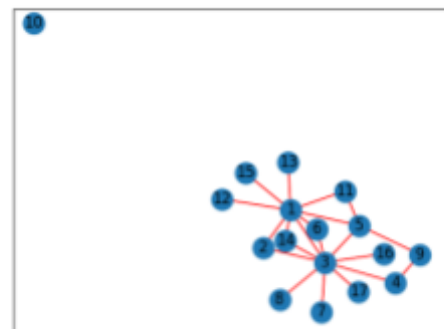
Un graphe G1 avec 12 sommets.



Un graphe G2 avec 20 sommets.



Un graphe G3 avec 10 sommets et $m = 8$.



Un graphe G4 avec 17 sommets et $m = 15$.

| Complexité en temps | | | | |
|---|-------------|-------------|-------------|-------------|
| | G1 | G2 | G3 | G4 |
| Bron-Kerbosch sans pivot $O(1.44^n)$ | 79,4968472 | 1469,771568 | 38,33759992 | 492,2235243 |
| Bron-Kerbosch avec pivot $O(3^{n/3})$ | 81 | 1516,381107 | 38,9407384 | 505,460369 |
| Bron-Kerbosch version améliorée $O(d * 3^{d/3})$ | 4,326748711 | 149,7660353 | 54 | 389,407384 |
| Enumération des cliques maximales v1 $\alpha O(f(k+1)q)$ | 108 | 189 | 90 | 144 |
| Enumération des cliques maximales v2 $O(\infty f(k+1)qK\Delta)$ | 288 | 1008 | 360 | 960 |

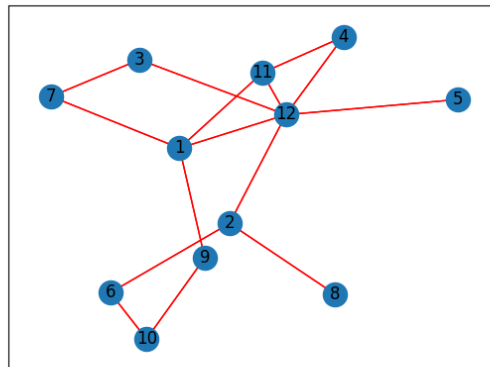
*L'unité de temps est en seconde

| Complexité en espace | | | | |
|---|----|----|----|----|
| | G1 | G2 | G3 | G4 |
| Bron-Kerbosch | 20 | 36 | 22 | 37 |
| Enumération des cliques maximales v1 | 36 | 63 | 30 | 48 |
| Enumération des cliques maximales v2 | 12 | 21 | 10 | 16 |

Exemple d'exécution des algorithmes

- Exemple 1 :

Ce premier graphe a été généré à partir de l'algorithme de génération aléatoire des graphes avec $|V| = 12$:

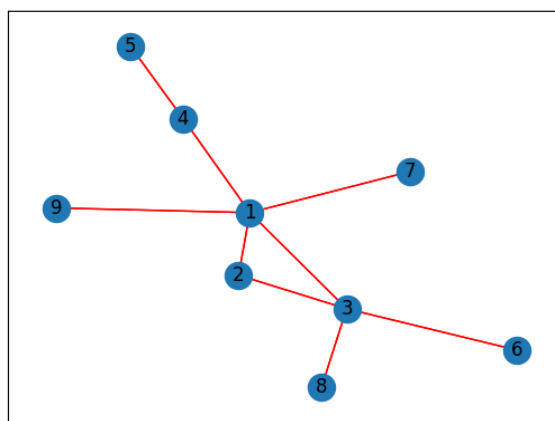


| Cliques maximales obtenues | | | | |
|----------------------------|--------------------------|---------------------------------|--------------------------------------|--------------------------------------|
| Bron-Kerbosch sans pivot | Bron-Kerbosch avec pivot | Bron-Kerbosch version améliorée | Enumération des cliques maximales v1 | Enumération des cliques maximales v2 |
| [1, 9] | [6, 2] | [10, 9] | [4, 11, 12] | [2, 8] |
| [1, 11, 12] | [6, 10] | [10, 6] | [7, 3] | [2, 12] |
| [1, 7] | [7, 1] | [4, 11, 12] | [5, 12] | [2, 6] |
| [2, 8] | [7, 3] | [1, 9] | [10, 9] | [11, 12, 1] |
| [2, 12] | [8, 2] | [1, 11, 12] | [2, 12] | [11, 12, 4] |
| [2, 6] | [9, 1] | [1, 7] | [3, 12] | [5, 12] |
| [3, 12] | [9, 10] | [7, 3] | [2, 8] | [10, 9] |
| [3, 7] | [12, 2] | [2, 8] | [7, 1] | [10, 6] |
| [4, 11, 12] | [12, 3] | [2, 12] | [10, 6] | [9, 1] |
| [5, 12] | [12, 5] | [2, 6] | [11, 12, 1] | [3, 12] |
| [6, 10] | [12, 11, 1] | [3, 12] | [1, 9] | [3, 7] |

| | | | | |
|---------|-------------|---------|--------|--------|
| [9, 10] | [12, 11, 4] | [5, 12] | [2, 6] | [7, 1] |
|---------|-------------|---------|--------|--------|

- Exemple 2 :

Ce second graphe a été généré selon le modèle de Barabasi-Albert avec 9 sommets et $m = 7$



:

| Cliques maximales obtenues | | | | |
|----------------------------|--------------------------|---------------------------------|--------------------------------------|--------------------------------------|
| Bron-Kerbosch sans pivot | Bron-Kerbosch avec pivot | Bron-Kerbosch version améliorée | Enumération des cliques maximales v1 | Enumération des cliques maximales v2 |
| [1, 2, 3] | [1, 2, 3] | [2, 1, 3] | [8, 3] | [7, 1] |
| [1, 4] | [1, 4] | [9, 1] | [6, 3] | [4, 1] |
| [1, 7] | [1, 7] | [6, 3] | [1, 7] | [4, 5] |
| [1, 9] | [1, 9] | [1, 4] | [5, 4] | [8, 3] |
| [3, 8] | [5, 4] | [1, 7] | [3, 2, 1] | [1, 9] |
| [3, 6] | [6, 3] | [5, 4] | [9, 1] | [1, 2, 3] |
| [4, 5] | [8, 3] | [3, 8] | [1, 4] | [6, 3] |

Conclusion

Ce projet a été une très bonne expérience pour l'ensemble des membres de l'équipe. Chacun de nous a réussi à finir ses tâches dans les délais malgré les difficultés rencontrées qui ont été surmontées grâce à l'entraide. Nous avons atteint notre objectif principal qui est la lecture et la compréhension des articles scientifiques fournis avec le sujet.

Pour résumer, l'algorithme de Barabasi-Albert est plus efficace que la génération "classique" des graphes, la version sans pivot de Bron-Kerbosch est inefficace pour les graphes qui contiennent beaucoup de cliques maximum, afin d'améliorer l'algorithme de Bron-Kerbosch on utilise la version avec ordonnancements des sommets, et il existe d'autres algorithmes d'énumération de cliques maximales qui dépendent de la dégénérescence du graphe.

Bibliographie

1. C. Bron, J. Kerbosch,
Algorithm 457, finding all cliques of an undirected graph
Comm. ACM, 16 (1973), pp. 575-577
dl.acm.org/doi/10.1145/362342.362367
2. Tomita Etsuji , Tanaka Akira, and Takahashi Haruhisa (2006)
The worst-case time complexity for generating all maximal cliques and computational experiments
Theoretical Computer Science
doi.org/10.1016/j.tcs.2006.06.015
3. N.Chiba, T. Nishizaki
Arboricity and subgraph listing algorithms
SIAM J. Comput
[Arboricity and Subgraph Listing Algorithms | SIAM Journal on Computing | Vol. 14, No. 1 | Society for Industrial and Applied Mathematics](https://doi.org/10.1137/S0036141003425019)
4. A.-L. Barabasi, R. Albert
Emergence of scaling in random networks
Science 286 (5439) (1999) 509-512
doi.org/10.1126/science.286.5439.509
5. George Manoussakis
A new decomposition technique for maximal clique enumeration for sparse graphs
Theoretical Computer Science
doi.org/10.1016/j.tcs.2018.10.014
6. David E., Maarten L., and Darren S.
Listing All Maximal Cliques in Sparse Graphs in Near-optimal Time
ISAAC 2010: Algorithms and Computation pp 403-414
dl.acm.org/doi/abs/10.1145/2543629