Sarah Ghiri – 000334719

Evguéniy Starygin – 000443325

8 decembre 2017

INFO-F203 – Algorithmique 2

Projet : "Cycles et hypergraphes"

1. Gestion des hypergraphes

L'hypergraphe est généré aléatoirement via la fonction graph_generator() avec entre 7 et 15 sommets et entre 2 et 5 hyperarêtes. Elle génère le graphe d'incidence de l'hypergraphe via la bibliothèque *Networkx*[1] et l'affiche avec *matplotlib*[2] qui est gérée par la fonction bipartite_draw(g).
Le choix de *Networkx* est motivé par les vastes fonctions entourant les graphes permettant un gain de temps et certaines fonctions permettant de vérifier nos propres implémentations, comme is_chordal(G) ou chordal_graph_cliques(G)[3].

On choisit de représenter l'hypergraphe via son graphe d'incidence, car on en a besoin pour trouver ou non un cycle dans le sens de Berge, si ce dernier est Berge-acyclique on peut alors ne pas construire de graphe primal pour l'alpha acyclicité. Mais pas seulement ce dernier convient très bien pour les autres acyclicités comme beta et gamma car il suffit de voir les voisins des sommets "e" qui sont les hyperarêtes pour avoir les sommets contenu dans une hyperarêtes, ect

Exemples d'exécution

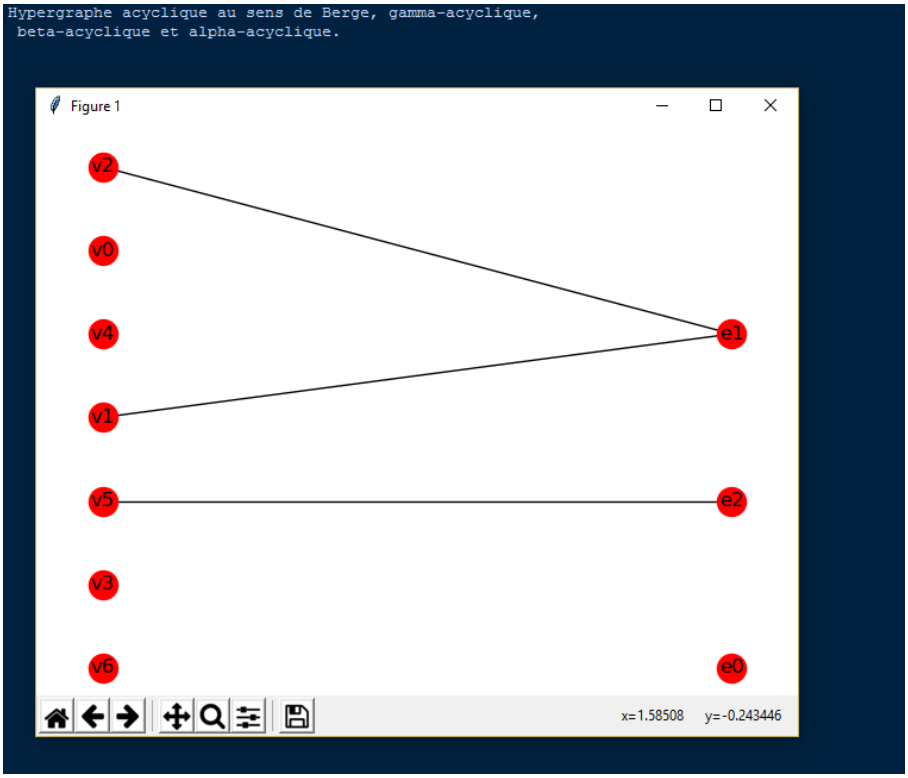
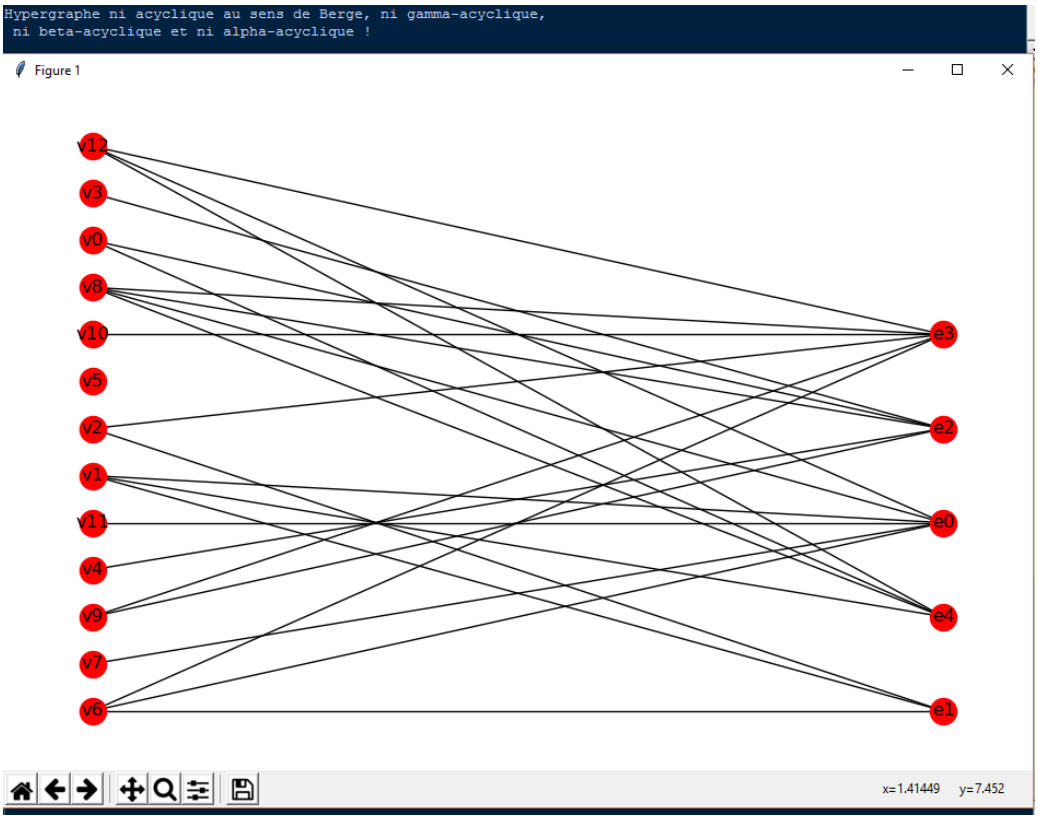


Figure 1

Le script affiche dans le terminal si le script est berge, gamma, beta, alpha acyclique et puis affiche la représentation du graphe d'incidence de l'hypergraphe via *mathplotlib*. L'exemple ci-dessus est berge-acyclique, du coup il est également gamma, beta et alpha (berge ⇒ γ-acyclique ⇒ β-acyclique ⇒ α-acyclique).

Celui-ci ci-dessous possède un cycle au sens de berge, gamma, beta et alpha.

[1] https://networkx.github.io/documentation/networkx-1.10/index.html
[2] https://matplotlib.org/2.1.0/index.html
[3] https://networkx.github.io/documentation/networkx-1.10/reference/algorithms.chordal.html



2. Acyclicité selon Berge

Un hypergraphe (**V**, **E**) est acyclique au sens de Berge si son graphe d'incidence (**v**, **e**) est acyclique. Notre fonction **berge(g)** prend donc le graphe d'incidence d'un hypergraphe en paramètre et y effectue un parcours en profondeur. S'il tombe sur un nœud déjà visité, il y a un cycle et il renvoie False.
Dans le cas présent, il est possible d'avoir des sous-graphes de l'hypergraphe non reliés, d'où la première boucle while. Elle itère sur une liste possédant tous les sommets sauf ceux avec un voisin ou moins, pour éviter un parcours inutile. Au final, cette boucle lancera des parcours avec les nœuds débutant un sous-graphe indépendant et pas avec les autres, vu qu'on supprime de cette liste directement ceux visités lors d'un autre parcours de sous-graphe.

Ensuite, on entre dans la boucle while du stack dans laquelle l'exploration commence vraiment, la fonction parcourra chaque sommet une fois du sous-graphe qu'on parcourt (sauf si elle détecte un cycle).
Pour chaque sommet en cours de visite, on parcourt également sa liste de voisins (d'arêtes en quelque sorte) dans la boucle for. Le précédent du sommet, s'il existe, sera aussi repris en compte vu que le graphe est non orienté.
Nous obtenons donc une complexité finale linéaire en **O(v+2e)**, **v** étant le nombre de sommets du graphe d'incidence, donc **V**+**E** et **e** le nombre d'arêtes du graphe, pris deux fois car le graphe est non-orienté. Notons que **v** sera possiblement inférieur à **V**+**E**, car aucun parcours n'est lancé à partir de sommets isolés (le premier while itère sur une liste de sommets les excluant), donc plus le graphe en possède et plus **v** < **V**+**E.**

3. Gamma-acyclicité

Pour déterminer la gamma-acyclicité d'un graphe, nous avons utilisé un ensemble de règles[4] permettant d'éliminer des sommets et hyperarêtes d'un hypergraphe successivement. Si l'hypergraphe se vide, il est gamma-acyclique.
Le premier while semblable à celui de la fonction alpha-acyclic, vérifie qu'il y a bien des suppressions qui ont eu lieu et qu'il reste des nœuds. Dedans, on parcourt d'abord tous les nœuds (sommets et hyperarêtes vu qu'il s'agit ici du graphe d'incidence de l'hypergraphe), donc **n** itérations. Rappelons que nos hyperarêtes possèdent des "e" dans leur nom pour les distinguer des sommets "v".
S'il le nœud est un sommet et est isolé, il est supprimé (première règle d'élimination). S'il s'agit d'une hyperarête. Ceci ce fait en **O(n)**. S'il s'agit d'une hyperarête, il relance une boucle sur toute la liste des nœuds, nous en sommes en **O(n^2)**. Enfin chaque hyperarête parcourra ses voisins **n**-1 fois dans cette deuxième boucle imbriquée avec les fonctions any et all utilisés pour la deuxième condition de suppressions des hyperarêtes. Ceci lui fera pratiquement passer par tous les sommets (vu que ceux isolés ont été supprimés) et nous en arrivons a **O(n^3)** dans le meilleur cas et en moyenne, quand on arrive à supprimer tous les nœuds directement sans devoir réitérer le while**.**

4. Beta-acyclicité

Un hyper graphe H = (V, E) est β-acyclique si en appliquant successivement les 2 règles suivantes, on obtient un hypergraphe vide[5] :

1. Si un sommet est un nest point (Un sommet v dans un hypergraphe H est un nest point si l'ensemble des hyperarêtes de H le contenant forme une chaîne pour l'inclusion, c.-à-d pour toutes hyperarêtes E et F contenant v, E ⊂ F ou F ⊂ E), alors on le retire de H.

2. Si une hyperarête est vide, alors on la retire de E.

Dans le cas où l'on arrive à un point où l'application de ces règles n'est plus possible et que l'hypergraphe n'est pas vide alors on détecte un β cycle. Un hypergraphe est également β-acyclique s'il est γ-acyclique (γ-acyclique ⇒ β-acyclique) et (β-acyclique ⇒ α-acyclique). Cette règle est utilisée dans la fonction beta_acyclic qui prend en paramètre le graphe d'incidence de l'hypergraphe.

La complexité de l'algorithme est en **O(n^2).**

[4] https://www.imj-prg.fr/theses/pdf/2009/david_duris.pdf, page 39
[5] https://www.imj-prg.fr/theses/pdf/2009/david_duris.pdf, page 42-43

5. Alpha-acyclicité

Notre fonction détectant l'alpha-acyclicité d'un graphe se base sur les définitions données dans l'énoncé du projet :

« un hypergraphe est α-acyclique si son graphe primal est cordal et que toute clique maximale (au sens de l'inclusion) de taille deux ou plus dans le graphe primale est une hyper-arête dans l'hypergraphe. »

Nous commençons donc par convertir le graphe d'incidence en graphe primal (**v**, **e**), ce qui se fait en **O(n²)**.

Pour déterminer la cordalité du graphe, nous utilisons la notion *d'élimination simpliciale*. On parcourt tous les sommets du graphe, on vérifie si un sommet forme une clique avec ses voisins et on le supprime dans ce cas. Si aucune suppression n'a été faite après le parcours de tous les sommets restant, il n'est pas cordal. Sinon, on continue jusqu'à atteindre ce cas ou jusqu'à ce qu'il ne reste plus de sommets (alors il est cordal).

Nous partons d'une liste de nœuds excluant les sommets orphelins, dans le cas de la cordalité il aurait aussi été possible d'exclure ceux ne possédant qu'un voisin, mais nous détectons en même temps les cliques et celles si peuvent être de taille 2 ou plus. Lors de l'*élimination simpliciale*, nous détectons au final toutes les cliques du graphe. On enregistre donc celles de taille maximale dans une liste, pour à la fin tester la condition avec les hyperarêtes de base ce qui se fera en **O(n)**, on ne parcourt que la liste de tous les nœuds du graphe d'incidence de l'hypergraphe pour avoir une liste des sommets inclus dans chaque hyperarête.

La boucle while principale s'arrête quand tout a été éliminé (ou si on ne peut plus rien faire et qu'il n'est pas cordal). Dans cette boucle, un indice **i** qui est itéré à chaque tour permet de parcourir tous les sommets (remis à 0 quand dépassement de la taille de la liste).

Dans le meilleur des cas possible, chaque sommet visité peut être supprimé et l'est donc directement (la taille de la liste sera réduite et vu que **i** augmente, il saute un sommet après l'élimination d'un autre et quand **i** aura atteint la taille de la liste on repart au début). On visite donc **n** sommets dans ce while. Pour le pire des cas, on supprime toujours un par tour de liste, par exemple le dernier visité de la liste dans son état courant. Donc si n = 5, on en visite 5, puis 4, puis 3… On arrive donc à [**n**(**n**+1)]/2 itérations. Pour chacun de ses sommets qu'on visite, on lance la fonction **detect_clique** qui itère sur tous les voisins (appelons ce total **m**) d'un sommet et qui pour chacun des voisins vérifient si les autres voisins de la liste ont les mêmes voisins donc (**m**-1).

Dans le meilleur des cas, la première boucle de **detect_clique** aura visité toutes les arêtes du graphe au final, si on tombe sur un sommet simplicial à chaque fois et en prenant en compte cette deuxième boucle imbriquée et aussi grâce au fait qu'on élimine des nœuds dans le premier while, on peut avoir $2(\mathbf{n}-1)^2 - 3(\mathbf{n}-1)$ itérations (le graphe doit également être connexe).

On arrive donc à une complexité moyenne en **O(n²)** au final pour le tout.