Robin Engels 000426221
Maël Panouillot 000444914

## Rapport du projet d'algorithmique:
# Les Hypergraphes

Génération de l'hypergraphe:
Lors de la génération de l'hypergraphe, un dictionnaire est utilisé.
D'abord, dans la fonction d'initialisation, trois valeurs, max_aretes, max_noeuds et proba sont reçues en paramètres. Alle vaudront respectivement le nombre maximum d'arêtes, de nœuds et la probabilité qu'un nœud appartienne à une arête.
La clef de chaque entrée du dictionnaire vaut le numéro du nœud qui lui est associée, et la valeur correspondant à cette clef est une liste comprenant toutes les arêtes auxquelles ce nœud appartient.

La méthode __str__ implémentée sert uniquement à afficher une liste des nœuds et à signaler à quelles arêtes ces nœuds appartiennent.

La méthode affiche_graphe_bipartie() sert à afficher le graphe incident de l'hypergraphe, grâce aux librairies netwokX et matplotlib, prend en paramètre un objet Graph.

La méthode affiche_graphe_primal() sert à afficher le graphe primal de l'hypergraphe, grâce aux librairies networkX et matplotlib, prend en paramètre un objet Graph.

Cyclicité au sens de Berge:
La méthode qui vérifie si le graphe est cyclique au sens de Berge est appelée berge(). Elle ne prend aucun paramètres (excepté le 'self' étant donné que c'est la méthode d'une classe), et le graphe qui est analysé est un attribut de la classe Graph().

Lorsque la méthode berge() est appelée, une variable et trois attributs de la classe Graph() sont créés. La variable point_de_depart servira pour analyser tout les points du graphe un par un grâce à une boucle, l'attribut points_de_depart_a_eviter[] est une liste contenant tout les points qui ont ete analyses, et à partir desquels il ne serait pas pertinent de faire une analyse complète du graphe, car elle a déjà été faite en partant d'un autre point. Ensuite, l'attribut points_visites[] stocke touts les noeuds qui ont ete visites, l'attribut aretes_a_eviter[] stocke les aretes par lesquelles l'algorithme est passe, et l'attribut result est un booléen qui vaut True uniquement si un cycle a ete trouve.

Ensuite, une boucle nettoie les différents attributs, et lance la méthode cherche_aretes() avec point_de_depart en parametre, puis incrément la variable point_de_depart pour passer à un autre point du graphe, et boucle jusqu'à ce que la valeur de point_de_depart soit plus grande que la taille du graphe, ou que l'attribut result vale True.
Enfin, la méthode renvoie le statut de l'attribut result, qu'il soit vrai ou faux.

Puis, la méthode cherche_aretes prend en parametre l'objet self et un point du graphe. La variable arete vaudra le numéro d'une des arêtes qui est connectée au point donné en paramètre. Ensuite une boucle vérifie si les arêtes connectees au point n'ont pas déjà été visitée (et exclu la dernière arête visitée, car elle sera forcément connectée au point). Si c'est le cas, alors le graphe est cyclique, et la boucle se termine. Si ce n'est pas le cas, l'arête est ajoutée dans la liste self.aretes_a_eviter, pour eviter que lors d'un appel récursif un peu plus loin, on ne repasse deux fois par la même arete, puis la fonction cherche_points est appelée, avec l'arête en question envoyée en paramètres. Enfin, l'arête est retirée de la liste self.aretes_a_eviter pour pouvoir y accéder de nouveau, et la varialbe arete est incrémentée. La boucle s'arrête lorsque l'attribut result vaut True, ou que toutes les arêtes connectées a ce point ont ete visitées.

Enfin, la fonction cherche_point prend en paramètre l'objet self et une arete, et recherche touts les points connectes à cette arête, pour les comparer aux points déjà visites.
D'abord, la fonction recherche tout les points connectes a l'arete en regardant point par point s'il est connecté à l'arête entré en paramètre. Si oui, alors le point est ajouté à la liste

points_connectes.
Ensuit, une boucle parcourt la liste points_connectes, et compare ces éléments à la liste points_visites, et si un élément de points_connectes fait partie de la liste points_visites, alors le graphe est cyclique. Sinon, le points est ajoute à la liste points_visites, la fonction cherche_aretes est appelée avec le point en paramètre, et enfin le point est retiré de la liste points_visites, et la boucle passe à l'élément suivant de points_connectes.
La boucle s'arrête si l'attribut result vaut vrai, ou que tout les éléments de points_connectes ont été visités.

La fonction cherche_aretes et cherche_points sont récursives entre elles, c'est à dire que dans certain cas, la fonction cherche_aretes appellera la fonction cherche_points, et réciproquement.

La fonction berge() est de complexité O(n*m), n= nombre de nœuds de l'hypergraphe et m = nombre d'arêtes de l'hypergraphe. La fonction Berge appelle la fonction cherche_aretes sur un noeud, qui elle-même appelle la fonction cherche_noeuds sur toutes les arêtes, et c'est éxécuté n*m fois au total, parce que la fonction ne repasse pas deux fois par un même noeud et une même arête.

## Alpha-Cyclicité:

La fonction find_clique (Complexité : $O(n*m^2)$, m = nombre de voisin a parcourir et n = nombre de noeuds) prends en paramêtre la taille maximum des cliques qu'on veux trouver ( par défaut elle est réglée a "l'infini" ). La fonction va parcourir tout le graphe et pour chaque noeuds vérifie que ces voisin ont au moins les même voisin que lui si c'est le cas c'est ce noeuds fait partie d'une clique. Si plusieurs clique de même taille sont trouvée elle sont renvoyée sous forme de liste.

La fonction voisin ( Complexité O(n*m), m = nombre d'hyper arrete dans lequelle le noeuds ce trouve ) va simplement regarder dans tout les hyper-arrete au quelle le noeuds donnée en paramêtre est connecté et en faire une liste de noeuds voisin.

La fonction is_chordal (Complexité : O(n^n)) vérifie que le graphe est bien cordal. Elle va d'abord chercher les plus grandes cliques du graphe et pour chaque clique trouvé les sommets simpliciaux si il y en a et les supprimé. Si aucun sommet simplicial n'est trouvé on réduit la taille maximum des prochaines clique a cherché et on recommence. Si aucun sommet simplicial n'est trouvé une fois que la taille max est égal a 1 le graphe n'est pas cordal. Si le graphe fini par se vider de tout ses noeuds il est cordal.

La fonction clean_graph (Complexité : O(n)) , supprime tout les noeuds qui n'ont pas de voisin