COURS DE THÉORIE DES GRAPHES
ORGANISATION PRATIQUE & PROJET

Pour le **lundi 12 octobre** 2020, chaque étudiant aura choisi les modalités d'examen le concernant :

1) projet d'implémentation, examen écrit (exercices et théorie vue au cours, énoncés et définitions) ;
2) pas de projet, examen écrit (exercices et partie théorique étendue).

Les délégués des différentes sections fourniront, par mail, la **liste des choix retenus**.

La note ci-dessous détaille le projet d'implémentation.

1. Extrait de l'engagement pédagogique MATH-0499

[...] Un projet d'implémentation, par groupes de deux, intervient (pour ceux qui en font le choix) dans la note finale. Ce projet nécessite en plus de fournir un code C, la production d'un rapport écrit court devant faciliter la compréhension du code et la défense orale de celui-ci (questions individuelles). Sauf mention explicite, les différents groupes ne peuvent ni collaborer, ni s'inspirer du code d'un autre groupe. [...]

2. Le code source

Le code source sera fourni en `C` "standard" et utilisera les bibliothèques usuelles. Il sera correct, efficace et intelligible. Votre code doit pouvoir être compilé, sans erreur (ni 'warning'), sous `gcc`. Si des options particulières sont nécessaires à la compilation, par exemple `--std=c99`, il est indispensable de le mentionner en préambule (ou de fournir un Makefile). Un code ne compilant pas entraîne une note de zéro au projet. Une alternative est de fournir le code source en `Python 3` "standard". On peut utiliser des modules usuels mais il faut bien évidemment coder les parties principales inhérentes au projet choisi (et ne pas utiliser une libraire toute faite). On peut par exemple utiliser `NetworkX` pour les manipulations basiques de graphes.

Quelques consignes qu'il est indispensable de respecter :

— Le choix des noms de variables et de sous-programmes doit faciliter la lecture et la compréhension du code.
— L'emploi de commentaires judicieux est indispensable : entrée/sortie des différents sous-programmes, points clés à commenter, boucles, etc.
— Enfin, l'indentation et l'aération doivent aussi faciliter la lecture de votre code en identifiant les principaux blocs.

Un code peu clair, même si le programme "tourne", sera pénalisé.

Une interface rudimentaire `graphes.c`/`graphes.h` est disponible en ligne sur `http://www.discmath.ulg.ac.be/`. Celle-ci est détaillée à la fin des

notes de cours (chapitre V). Libre à vous de l'utiliser ou non, voire de l'améliorer.

3. Le rapport

Le rapport ne doit pas être un copier-coller du code source (ce dernier étant fourni par ailleurs). Le rapport, au format `pdf` et idéalement rédigé sous LaTeX, est court : maximum 5 pages. Il doit décrire la stratégie utilisée, les choix opérés, les grandes étapes des différentes procédures ou fonctions. Il pourra aussi présenter les difficultés/challenges rencontrés en cours d'élaboration ou reprendre certains résultats expérimentaux (benchmarking sur des exemples types ou générés aléatoirement).

4. La présentation orale

La présentation est limitée à **10 minutes** maximum. Sans que cela soit nécessaire, les étudiants ont le droit d'utiliser un ordinateur (pour faire tourner leur programme, pour présenter leur code, pour présenter leur travail avec un support type "power point"). Un projecteur vidéo est à disposition. Cette présentation se veut être une synthèse/explication du travail fourni.

Elle est suivie par une **séance de questions**. Le but de ces questions est de déterminer la contribution et l'implication de chacun. Ainsi, des questions différentes seront posées individuellement et alternativement aux deux membres du groupe. L'ensemble présentation/questions ne devrait pas dépasser 20 minutes. Un ordre de passage des différents groupes sera déterminé.

5. Dates importantes

— **lundi 12 octobre 2020** : choix individuel des modalités d'examen, répartition en groupes et choix des sujets.
— **vendredi 11 décembre 2020** : dépôt du code et du rapport sous forme d'une archive envoyée par mail au titulaire du cours. Cette archive contiendra deux répertoires, un pour le code à compiler, l'autre pour le rapport.
— **lundi 14 décembre 2020** — ordre de passage à déterminer : présentation orale.
— **janvier 2020** : examen écrit (commun pour tous).

6. Les projets

Sauf problème, les étudiants proposent une **répartition par groupes** de deux (en cas d'un nombre impair d'étudiants, un unique groupe de 3 étudiants sera autorisé) et l'**attribution des sujets** aux différents groupes (un même sujet ne peut pas être donné à plus de 2 groupes — le sujet choisi par l'éventuel groupe de 3 étudiants ne peut être attribué à un second groupe). La répartition devra être validée par le titulaire du cours.

Si un accord entre les étudiants n'est pas trouvé, le titulaire procédera à un tirage au sort (des groupes et des sujets).

Le plagiat est, bien entendu, interdit : il est interdit d'échanger des solutions complètes, partielles ou de les récupérer sur Internet. Citer vos sources !

Néanmoins, vous êtes encouragés à discuter entre groupes. En particulier, il vous est loisible d'utiliser des fonctions développées par d'autres groupes (et qui ne font pas partie du travail qui vous est assigné). Mentionner les sources utilisées/consultées.

Les projets listés ci-dessous sont "génériques", il est loisible à chaque groupe d'aller plus loin, d'adapter et de développer plus en avant les fonctionnalités de son code (par exemple, meilleure gestion des entrées/sorties, optimisation des structures de données, fournir des exemples "types" dans un fichier, etc.).

Vérifiez que votre solution tourne même sur les cas pathologiques (par exemple, quel est le comportement attendu, si le graphe fourni n'est pas connexe, ne satisfait pas aux hypothèses, si le fichier est mal structuré, etc.). Essayez de construire un ensemble "témoin" de graphes "tests" sur lesquels faire tourner votre code. Tous les projets n'ont pas la même difficulté, il en sera tenu compte pour la cotation.

(1) Implémentation de l'algorithme de *Ford-Fulkerson* permettant de rechercher un flot maximum dans un réseau de transport ; une première page permettant de s'initier au sujet est donnée par
`https://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm`
Le rapport doit expliquer la notion de flot maximum, l'algorithme implémenté et le code doit tourner sur des réseaux de taille suffisante. Il est demandé de consulter d'autres sources que l'unique page Wikipédia.

(2) Implémentation de l'algorithme de *Dinic* permettant de rechercher un flot maximum dans un réseau de transport ; une première page permettant de s'initier au sujet est donnée par
`https://en.wikipedia.org/wiki/Dinic's_algorithm`
Le rapport doit expliquer la notion de flot maximum, l'algorithme implémenté et le code doit tourner sur des réseaux de taille suffisante. Il est demandé de consulter d'autres sources que l'unique page Wikipédia.

(3) *Implémentation de l'algorithme de Bellman–Ford.* Cet algorithme, non vu au cours, est une alternative à l'algorithme de Dijkstra permettant de trouver un plus court chemin. Il s'agit donc, dans un premier de se familiariser avec la méthode (recherche bibliographique) pour, dans un second temps, l'implémenter et le comparer avec l'algorithme de Dijkstra. `https://en.wikipedia.org/wiki/Bellman-Ford_algorithm`

(4) *Implémentation de l'algorithme de Karger.* Cet algorithme (probabiliste), non vu au cours, permet de trouver un ensemble de coupure minimum. Il s'agit donc, dans un premier de se familiariser avec la méthode (recherche bibliographique) pour, dans un second temps, l'implémenter. `https://en.wikipedia.org/wiki/Karger's_algorithm`

(5) Un graphe simple non dirigé $G = (V, E)$ possède une évaluation "gracieuse" (graceful labeling) s'il existe une bijection $f : V \to \{1, \ldots, \#V\}$ telle que pour toute paire d'arêtes distinctes $\{x, y\}$ et $\{u, v\}$, $|f(u) - f(v)| \neq |f(x) - f(y)|$. Autrement dit, la numérotation des sommets

induit une numération univoque des arêtes. On conjecture que tout arbre possède une telle évaluation. Fournir un programme donnant une évaluation gracieuse pour tout arbre d'au plus 30 sommets. Référence : A Computational Approach to the Graceful Tree Conjecture, `https://arxiv.org/abs/1003.3045`

(6) *Noyau d'un graphe.* On donne un graphe simple et orienté $G = (V, E)$ sans cycle. Le noyau de G est l'unique sous-ensemble N de sommets vérifiant les deux propriétés suivantes
— stable : $\forall u, v \in N$, $(u, v) \notin E$ et $(v, u) \notin E$
— absorbant : $\forall u \notin N$, $\exists v \in N : (u, v) \in E$.
Etant donné un graphe, tester s'il est sans cycle et dans ce cas, être en mesure de fournir son noyau. On appliquera l'algorithme à des graphes issus de la théorie des jeux : étant donnés deux entiers $k, \ell \geq 0$ considérés comme paramètres fournis par l'utilisateur, on considère le graphe dont les sommets sont les couples (x, y) avec $x \leq k$ et $y \leq \ell$. Il y a un arc de (x, y) à (x', y') si et seulement si $x = x'$ et $y' < y$ ou, $x' < x$ et $y = y'$. Dans un second temps, on ajoutera aux arcs précédents, des arcs de (x, y) à (x', y') avec la condition $x - x' = y - y' > 0$.

(7) *Recherche d'un "matching" dans un graphe biparti.* On présentera tout d'abord la notion de matching (définition, existence d'un matching parfait, applications, difficulté du problème) avant d'implementer un algorithme (par exemple, l'algorithme de Hopcroft–Karp) recherchant un matching de taille maximum au sein d'un graphe biparti.

(8) *Génération aléatoire de graphes, modèle de Barabási–Albert.* On présentera d'abord le modèle d'un point de vue théorique (cf. `https://arxiv.org/abs/cond-mat/0106096`). Ensuite, on l'implémentera pour générer des graphes aléatoires. L'implementation devra permettre d'afficher graphiquement les graphes obtenus (bonus : une animation). Le but est de pouvoir générer des graphes de grande taille. On pourra, par exemple, générer une sortie utilisable par un utilitaire de visualisation de graphes, comme GraphViz `http://www.graphviz.org/`.

(9) *Le problème de l'isomorphisme de graphes.* On donne deux graphes (tous les deux orientés ou non) et on veut décider s'ils sont ou non isomorphes. En cas de réponse positive, on donnera un isomorphisme entre eux. Remarque : il s'agit d'un problème connu pour être algorithmiquement difficile. Le recours à des heuristiques est le bienvenu.

(10) *Test de planarité.* Etant donné un graphe simple, déterminer si celui-ci est ou non planaire (on pourra utiliser le théorème de Kuratowski). On peut par exemple implémenter la méthode originale de John Hopcroft, Robert Tarjan, *Efficient Planarity Testing* (1974). Le programme peut être adjoint d'une fonction supplémentaire : si un graphe n'est pas planaire, mettre en évidence un sous-graphe homéomorphe à K_5 ou $K_{3,3}$. Le programme fournirait ainsi une preuve de non-planarité.

(11) *Construction d'un graphe réel de collaborations.* Pouvoir interroger une base de données comme `http://www.ams.org/mathscinet/` ou `orbi.ulg.ac.be` pour construire un graphe (pondéré non orienté) dont les sommets sont les auteurs et une arête existe entre deux auteurs s'ils ont une publication commune. Le poids d'une arête est fourni par le nombre de publications communes. En interrogeant ces bases de données, l'utilisateur pourra fournir certains critères (pour ne pas obtenir un graphe trop gros) comme une liste de noms, certaines dates, etc. ou construire tous les sommets à distance au plus d d'un sommet donné. Permettre le calcul d'un plus court chemin entre deux sommets.

(12) Un graphe (simple, non orienté) est *k-dégénéré* s'il est possible de supprimer un à un ses sommets de telle sorte que chaque sommet supprimé soit de degré au plus k dans le sous-graphe induit par les sommets restant. Ce projet comporte deux parties : une procédure de test pour déterminer si un graphe est k-dégénéré (k est un paramètre) et en cas de réponse positive, fournir une suite convenable de sommets à supprimer. Ensuite, générer des graphes k-dégénérés maximaux à n sommets (k, n sont des paramètres). La maximalité signifie que le graphe obtenu est k-dégénéré et que si l'on ajoute une quelconque arête, il n'a plus cette propriété.

(13) Un graphe *parfait* (simple et non orienté) est défini comme suit. Soit $\alpha(G)$, le cardinal maximal d'un ensemble de sommets indépendants de G. Soit $k(G)$, le nombre minimal de cliques présentes dans G et dont la réunion contient tous les sommets de G (certaines arêtes peuvent manquer et un même sommet peuvent appartenir à plusieurs cliques). Un graphe G est *parfait* si $\alpha(G) = k(G)$. Dans ce projet, on demande de calculer ces deux quantités $\alpha(G)$ et $k(G)$. Ensuite, pouvoir engendrer des graphes parfaits à n sommets (n étant un paramètre).

(14) *Algorithmes de Kruskal et Boruvka.* Etant donné un graphe simple connexe pondéré, trouver un arbre de poids minimal en utilisant d'une part l'algorithme de Kruskal et d'autre part l'algorithme de Boruvka. Si on fournit un multigraphe en entrée, le remplacer par un graphe 'équivalent' (boucles supprimées et multi-arêtes remplacées par un unique arête de poids minimal). La première partie de ce travail nécessite une petite recherche bibliographique et le rapport expliquera et comparera les deux algorithmes. Dans la mesure du possible, si plusieurs arbres réalisent la meilleure borne, fournir ces alternatives. cf. par exemple
`https://en.wikipedia.org/wiki/Kruskal's_algorithm`
`https://en.wikipedia.org/wiki/Boruvka's_algorithm`

Quelques conseils :

— Pensez à l'utilisateur qui teste votre programme : préparer un makefile, donner des conseils sur l'utilisation (fournir quelques fichiers de test), quelles entrées fournir, quelles sorties attendues ? Décrivez un exemple typique d'utilisation.

— Préparez une petite bibliographie, citer les sources utilisées (même les pages Wikipédia !). Si vous avez exploiter une source, un autre cours, mentionnez-le explicitement !

— Avez-vous tester votre programme sur de gros graphes ? De quelles tailles ? Eventuellement produire un petit tableau de "benchmarking" indiquant, sur une machine donnée, le temps de calcul en fonction des tailles de graphes testés.
— Relisez (et relisez encore) votre rapport ! Faites attention à l'orthographe (accords, conjugaison), au style.
— Si vous développez des heuristiques, avez-vous des exemples de graphes (ou de familles de graphes) qui se comportent mal par rapport à cette heuristique ?