COURS DE THÉORIE DES GRAPHES
ORGANISATION PRATIQUE & PROJET

Pour le **lundi 18 octobre** 2021, 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). Citez vos sources (construire une petite bibliographie/sitographie) et détaillez votre contribution par rapport à ces références. N'héistez pas à consulter et comparer plusieurs sources pour sélectionner les plus pertinentes.

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/démonstration 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 18 octobre 2021** : choix individuel des modalités d'examen, répartition en groupes et choix des sujets.
— **jeudi 9 décembre 2021 (minuit)** : 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 13 décembre 2021** — 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 3 groupes — le sujet choisi par l'éventuel groupe de 3 étudiants ne peut être attribué que deux fois). 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. Enparticulier, 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.).

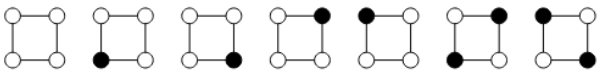
Vérifiez que votre solution tourne même sur les cas pathologiques (par exemple, quel est le comportement attendu, si le graphe fourni n'est pas connexe, ne satisfait pas aux hypothèses, si le fichier est mal structuré, etc.). Essayez de construire un ensemble "témoin" de graphes "tests" sur lesquels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

(1) Etant donné un fichier reprenant les PAE d'étudiants (liste d'étudiants et les cours choisis), construire un graphe non orienté dont les sommets sont les cours dispensés et il existe une arête entre deux sommets si au moins un étudiant possède les deux cours correspondants dans son cursus (graphe d'incompatibilité). Déterminer un horaire d'examen compatible minimisant le nombre de jours d'examen nécessaires. Il s'agit en fait d'un problème de coloriage, voir par exemple J. Randall Brown, *Chromatic Scheduling and the Chromatic Number Problem*, `https://www.jstor.org/stable/2629029` [1]

(2) Etant donné un graphe fortement connexe fourni en entrée (possibilité de charger un fichier, fournir un fichier exemple pour un graphe de 100 sommets), l'utilisateur sélectionne un sommet de départ v et un entier n. On souhaite estimer la probabilité de revenir en v avec un chemin de longueur exactement n, puis de longueur *au plus* n. On souhaite également compter le nombre de tels chemins. Quand un agent qui parcourt le graphe se trouve en un sommet, on suppose que le choix du successeur se fait de façon de uniforme : s'il y a t successeur, chacun a une probabilité $1/t$ d'être choisi. Expérimenter avec d'autres distributions de probabilité. Fournir des "statistiques", par exemple, sur un échantillon de n chemins, quel est le plus fréquent, etc.

(3) Soit $i(G)$, le nombre d'ensembles différents de sommets indépendants d'un graphe simple non orienté G. Par exemple, pour un cycle à 4 sommets, $i(C_4)$ vaut 7, comme on le voit ci-contre (où les ensembles indépendants sont formés de sommets noircis).

1. accessible depuis une machine connectée au réseau ULiège



Pouvoir calculer $i(G)$ pour un graphe fourni dans un fichier. Etant donné un graphe non orienté k-régulier à n sommets, vérifier numériquement le résutlat suivant : $i(G) \leq (2^{k+1}-1)^{n/(2k)}$. Vérifier aussi que l'égalité est atteinte si n est divisible par $2k$ et G est l'union disjointe de $n/(2k)$ copies du graphe biparti complet $K_{k,k}$. Source : `https://arxiv.org/abs/1610.09210v2`
Des bibliothèques de graphes sont disponibles sur *house of graphs*, `https://hog.grinvin.org/`

(4) Le problème de l'empereur Constantin : *Defendens Imperium Romanum : A Classical Problem in Military Strategy*, comment distribuer des légions pour sécuriser l'empire romain ? Lire et s'inspirer de l'article `https://www.jstor.org/stable/2589113` [2] de C. S. S. ReVelle and K. E. Rosing et développer un algorithme pour chercher des solutions (optimales ou non) à des problèmes comparables (on se limitera à des graphes d'une vingtaine de sommets).

(5) Soit G un graphe simple non orienté. Un *coloriage équitable* est l'attribution d'une couleur à chaque sommet de telle sorte que des sommets voisins reçoivent des couleurs distinctes (coloriage propre) et les tailles des ensembles de sommets ayant reçu deux couleurs quelconques diffèrent d'au plus une unité. Sur des graphes de petite taille (que l'on peut fournir dans un fichier), trouver un coloriage équitable utilisant un nombre minimum de couleurs.

(6) Variante du problème précédent, implémenter l'algorithme (difficile) décrit par Kierstead et al.
`https://link.springer.com/article/10.1007%2Fs00493-010-2483-5`
permettant d'obtenir un coloriage équitable à $r+1$ couleurs quand les sommets ont un degré au plus r.

(7) On considère un graphe représentant le réseau des contacts au sein d'une population. Chaque sommet représente un agent et les arêtes relient les agents susceptibles d'interagir. Chaque agent sera marqué comme sain, infecté ou guéri. Au départ, seuls quelques agents seront marqués de façon aléatoire comme infectés, tandis que tous les autres seront marqués comme sains (susceptibles d'être infectés). A chaque étape (temps discret), chaque agent choisira un sous-ensemble aléatoire de ses voisins dans le réseau de contacts, et il les "rencontrera". Si un agent susceptible rencontre un agent infecté, il devient lui aussi infecté. Les rencontres sont considérées comme symétriques, donc l'agent qui initie la rencontre n'a pas d'importance. Enfin, les agents infectés se rétablissent après une certaine période (nombre de pas à définir). Pour les détails, s'inspirer de `https://community.wolfram.com/groups/-/m/t/1907703`

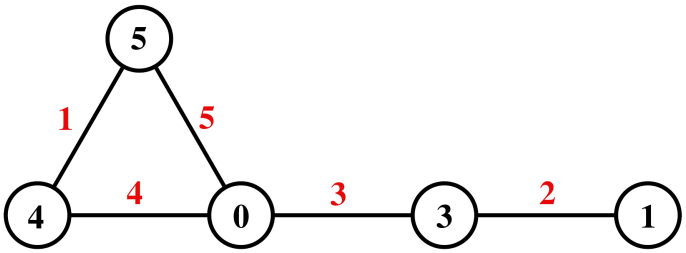
2. accessible depuis une machine connectée au réseau ULiège

(8) 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 (possibilité de charger un fichier, fournir un fichier exemple pour un graphe de 100 sommets). Il est demandé de consulter d'autres sources que l'unique page Wikipédia. Il sera nécessaire d'implémenter des arcs pondérés.

(9) Projet comparable au précédent. 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 (possibilité de charger un fichier, fournir un fichier exemple pour un graphe de 100 sommets). Il est demandé de consulter d'autres sources que l'unique page Wikipédia. Il sera nécessaire d'implémenter des arcs pondérés.

(10) *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`
Possibilité de charger un fichier et fournir un fichier exemple pour un graphe de 50 sommets minimum.

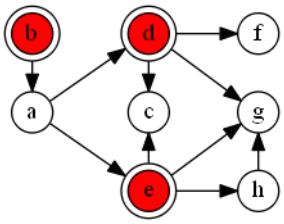
(11) Un graphe simple non orienté $G = (V, E)$ possède une évaluation "gracieuse" (en anglais, *graceful labeling* pour vos recherches) s'il existe une bijection $f : V \to \{0, \dots, \#V - 1\}$ associant à chaque sommet un entier, telle que pour toute paire d'arêtes distinctes $\{x, y\}$ et $\{u, v\}$, $|f(u) - f(v)| \neq |f(x) - f(y)|$. Autrement dit, la numérotation des sommets (en noir sur la figure en exemple) induit une numération univoque des arêtes (en rouge).



On conjecture que tout arbre possède une telle évaluation. Fournir un programme donnant une évaluation gracieuse pour tout arbre d'au plus 25 sommets. Possibilité de charger un fichier (on supposera que le

fichier fourni décrit un arbre). Référence : A Computational Approach to the Graceful Tree Conjecture, `https://arxiv.org/abs/1003.3045` Pour aller plus loin, vous pouvez produire un générateur d'arbres ou utiliser la bibliothèque *house of graphs* `https://hog.grinvin.org/`

(12) *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 (en rouge sur la figure en exemple) 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. Possibilité de charger un fichier et fournir un fichier pour un graphe sans cycle de 50 sommets minimum. On appliquera ensuite l'algorithme à des graphes issus de la théorie des jeux et définis comme suit : 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 $x - x' = y - y' > 0$.

(13) *Le problème de l'isomorphisme de graphes.* On donne deux graphes simples (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. On pourra également appliquer le programme à deux copies d'un même graphe pour en chercher des automorphismes.

(14) *Algorithme de Kosaraju—Sharir.* Implémentation de cet algorithme permettant de rechercher les composantes fortement connexes d'un graphe orienté. On peut se limiter au cas de graphes simples. On comparera, d'un point de vue théorique, cet algorithme avec celui de Tarjan. Dans un second temps, fournir le graphe acyclique des composantes connxes.

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 ?