THÉORIE DES GRAPHES
ORGANISATION & PROJETS 2024–2025

Pour le **lundi 7 octobre** 2024, 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. Par exemple, s'il vous est demandé explicitement de codé la recherche d'un plus court chemin, il ne faut pas prendre la fonction toute faite dans `NetworkX`.

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 6 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'hésitez pas à consulter et comparer plusieurs sources pour sélectionner les plus pertinentes. Il est dommage de voir que des étudiants se contentent souvent d'une unique référence `Wikipedia`, faites preuve d'esprit critique. Lisez les quelques conseils donnés en fin de document.

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 9 octobre 2024** : choix individuel des modalités d'examen, répartition en groupes et choix des sujets.
— **jeudi 12 décembre 2024 (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 16 décembre 2024** — ordre de passage à déterminer : présentation orale.
— **janvier 2025** : 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 ; faire un projet seul est aussi possible) 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. 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 étudiants s'engagent à respecter la Charte ULiège d'utilisation des intelligences artificielles génératives dans les travaux universitaires.

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.). Le principe de base est de proposer un thème qu'il vous est loisible de développer.

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) Construction du graphe de Rauzy d'ordre n d'une longue chaîne de caractères. Pour la définition, voir la première page de ce document. On donne au programme une chaîne construite à partir d'un nombre fini de règles de substitution. Par exemple $f : a \mapsto ab, b \mapsto ba$. On itère f à partir du symbole a (il est nécessaire que le symbole de départ soit substitué par une chaîne débutant par le même caractère). A chaque étape, les lettres du mot sont remplacées par leur image par f. Ainsi, a est remplacé par ab qui est lui-même remplacé par $f(a)f(b) = abba$. A son tour remplacé par $f(a)f(b)f(b)f(a) = abbabaab$ et ainsi de suite. On donne au programme la taille ℓ du préfixe à construire. Ensuite, dans ce préfixe, on passe en revue les facteurs de longueur n y apparaissant et on construit le graphe d'ordre n. Ainsi, les paramètres du programme sont ℓ, n et les règles définissant f. On affichera le graphe obtenu. Idéalement, on aimerait voir comment évolue ce graphe quand ℓ et f sont fixés et que n grandit.

(2) Implémentation du Leiden algorithm pour la détection de communautés au sein d'un graphe. A partir d'un graphe fourni en entrée (par ex. via un fichier), faire tourner cette modification de l'algorithme "de Louvain" pour chercher des sous-ensembles de sommets fortement

connectés (et définissant ainsi une communauté). On soignera les sorties du programme et on le testera sur des graphes de grandes taille.

(3) Le graphe de `Wikipedia` a pour sommets les pages de l'encyclopédie en ligne et il existe un arc d'un article vers un autre, si le premier fait référence au second. Dans ce projet, il faut extraire une partie du graphe présent en ligne (ceci représente déjà un challenge informatique important) et construire le graphe associé. Dans la structure de données, on ne conservera de chaque page que son titre et les liens (orientés) entre celles-ci. On pourra sauver le résultat dans un fichier (format au choix). On essayera alors d'extraire un maximum d'informations pertinentes en exploitant des notions de théorie des graphes.

(4) La fonction de Grundy $g : V \to \mathbb{N}$ attribue à chaque sommet d'un graphe (simple) orienté sans cycle $G = (V, E)$ une valeur définie récursivement :

   (a) Si un sommet u n'a pas d'arêtes sortantes, alors $g(u) = 0$.

   (b) Pour tout sommet u, la fonction est calculée comme suit :

$$g(u) = \text{mex}(g(\text{succ}(u)))$$

   où $\text{succ}(u) = \{v \in V : (u, v) \in E\}$ est l'ensemble des successeurs de u, et **mex** (minimum excluant) désigne le plus petit entier positif ou nul n'étant pas dans l'ensemble des valeurs $g(v)$ des successeurs $v \in \text{succ}(u)$.

   Ecrire un programme qui teste si un graphe orienté est ou non sans cycle. Ensuite, pour un graphe sans cycle fourni en entrée (par ex. dans un fichier) calculer la valeur de Grundy de chaque sommet. On essaiera ensuite de représenter le graphe (s'il est de petite taille) en positionnant les sommets de même valeur sur un même niveau.

(5) Modéliser, à petite échelle, le système de recommandation utilisé par Netflix. En particulier, partir de données réelles ou fictives que l'on pourra charger à partir d'un fichier pour faire tourner la méthode.

(6) Construction du graphe du Rubik's cube : chaque sommet représente une configuration du Rubik's cube et deux sommets sont connectés s'il existe un mouvement pour passer de l'un à l'autre. On utilisera des représentations "standards" des configurations et des mouvements. Le graphe étant de très grande taille, on n'en considérera qu'une partie. On pourra par exemple, explorer des chemins pour essayer d'arriver à la résolution du jeu.

(7) Même genre d'énoncé que pour le projet précédent mais cette fois pour le jeu Quadratis. Chaque configuration du jeu correspond à un sommet du graphe et les sommets sont connectés en fonction des coups joués. Il est ici possible de créer le graphe complet associé au jeu et d'en explorer les propriétés (quelle est la "pire" configuration de départ, statistiques, etc.).

(8) Algorithmes d'approximation pour le *problème du voyageur de commerce* (1). Voir la page Wikipedia pour la description du problème. Implémenter deux approches d'approximation pour ce problème du voyageur de commerce.

Il vous est demandé d'implémenter l'heuristique du plus proche voisin et de la comparer avec l'*algorithme de Christofides–Serdyukov*. Vous fournirez plusieurs exemples sur lesquels appliquer vos procédures. Votre programme doit pouvoir prendre en entrée un fichier décrivant un réseau. Idéalement, vous réaliserez votre propre "benchmarking" pour comparer les performances de ces deux méthodes.

(9) Algorithmes d'approximation pour le *problème du voyageur de commerce* (2). Voir la page Wikipedia pour la description du problème. Implémenter deux approches d'approximation pour ce problème du voyageur de commerce. Il vous est demandé d'implémenter l'heuristique du plus proche voisin et de la comparer avec la *technique de l'échange de paires* qui consiste à supprimer itérativement deux arêtes et à les remplacer par deux arêtes différentes qui reconnectent les fragments créés par la suppression d'arêtes en un nouveau circuit améliorant le précédent (faire une petite recherche bibliographique). Vous fournirez plusieurs exemples sur lesquels appliquer vos procédures. Votre programme doit pouvoir prendre en entrée un fichier décrivant un réseau. Idéalement, vous réaliserez votre propre "benchmarking" pour comparer les performances de ces deux méthodes.

(10) Algorithmes d'approximation pour le *problème du voyageur de commerce* (3). Voir la page Wikipedia pour la description du problème. Implémenter deux approches d'approximation pour ce problème du voyageur de commerce. Il vous est demandé d'implémenter l'heuristique du plus proche voisin et de la comparer avec la méthode de recuit simulé. On pourra consulter la référence Section 1.5. Vous fournirez plusieurs exemples sur lesquels appliquer vos procédures. Votre programme doit pouvoir prendre en entrée un fichier décrivant votre réseau. Idéalement, vous réaliserez votre propre "benchmarking" pour comparer les performances de ces deux méthodes.

(11) Implémenter le test de planarité décrit dans la référence "GRAPH THEORY WITH APPLICATIONS" de Bondy et Murty, 5e édition 1982, (facilement accessible en ligne) Section 9.8, pages 163–164. Faire une recherche bibliographique d'autres tests de planarité et brièvement les décrire. On fournira un graphe dans un fichier et votre programme doit décider si le graphe est ou non planaire.

(12) Implémenter la recherche d'un flot maximum dans un réseau de transport décrit dans la référence "GRAPH THEORY WITH APPLICATIONS" de Bondy et Murty, 5e édition 1982, (facilement accessible en ligne) Section 11.3, pages 198–202, et se basant sur le théorème *Max flow–Min cut*. On fournira les données d'un problème dans un fichier (un réseau et les capacités des arcs) et votre programme fournira une solution (si elle existe). Cette solution doit être donnée sous la forme d'un fichier formaté. Quelles applications réelles pourriez-vous envisager ?

(13) A l'instar de la recherche d'une chaîne de caractères dans un texte, on peut rechercher la présence d'un sous-graphe dans un graphe. Étant donnés deux graphes G et H décrits dans des fichiers, votre programme

doit déterminer si H y apparaît comme sous-graphe et fournir l'indication de l'endroit où ce sous-graphe a été détecté. Dans un second temps, votre programme doit aussi fournir toutes les occurrences de H dans G. Notez bien qu'il s'agit d'un problème combinatoirement difficile, le temps de calcul peut être important et il sera probablement nécessaire de vous limiter à des graphes de taille modeste. Pour les graphes de petite taille, avoir une sortie graphique mettant en évidence les occurrences de H dans G est en plus, s'inspirer de la fonction `GrapHighlight` de Mathematica.

(14) Algorithmes de routage dans les réseaux. L'infrastructure du réseau Internet peut être modélisée par un graphe. Le processus consistant à trouver un chemin d'une source vers toutes les destinations du réseau est appelé *routage*. Celui-ci est réalisé au moyen d'un protocole qui établit, au sein de chaque routeur, des tables de routage consistantes. Ce travail (ambitieux) comporte d'abord une part de recherche bibliographique, vous modéliserez un tel réseau par un graphe. Ensuite, vous y implémenterez le distance-vector algorithm. Attention : avoir un modèle qui rend compte de la complexité d'Internet n'est pas simple (simuler la non-fiabilité de certains liens ou routeurs, avoir un trafic variable au cours du temps, etc.). Vous avez toute la liberté de proposer votre modèle.

(15) Construire le graphe des produits achetés conjointement sur `Amazon` en utilisant la base de données `amazon0302` ou `amazon0312` disponible sur graphchallenge.mit.edu. Analyser le graphe ainsi construit et essayer d'extraire un maximum d'informations pertinentes en exploitant des notions de théorie des graphes. En particulier, pourriez-vous fournir un algorithme de recommandation d'achats ?

(16) Implémenter l'algorithme de Tarjan permettant de détecter les composantes fortement connexes d'un graphe (simple) orienté. Vous implémenterez également une fonction permettant de générer aléatoirement un graphe orienté par le modèle de Barabási–Albert (adapté aux graphes orientés). Vous pourrez ainsi générer de "grands" graphes et afficher des statistiques sur leurs composantes f. connexes (nombre de sommets, taille des composantes, distribution, ...). Avoir une sortie "graphique" visualisant les sorties est un plus (mais pour de grands graphes, il faudra faire preuve d'ingéniosité en affichant par exemple uniquement le condensé — graphe acyclique des composantes connexes du graphe).

(17) Dans le problème *Sparsest Cut problem*, l'objectif est de partitionner un graphe (simple non orienté) donné en deux ou plusieurs grandes "parties" en supprimant le moins d'arêtes possible. Les problèmes de partitionnement de graphes tels que celui-ci occupent une place centrale dans de nombreuses applications. Formellement, étant donné un graphe simple non orienté $G = (V, E)$, on définit *l'éparpillement* (sparsity) d'un ensemble non vide $S \subset V$ de sommets, où $\#S \leq (\#V)/2$ comme le ratio $\alpha(S)$ entre le nombre d'arêtes de G joignant les sommets de S et de $V \setminus S$ et $\#S$. L'éparpillement du graphe est alors défini comme le minimum des $\alpha(S)$ pour tous les sous-ensembles de sommets

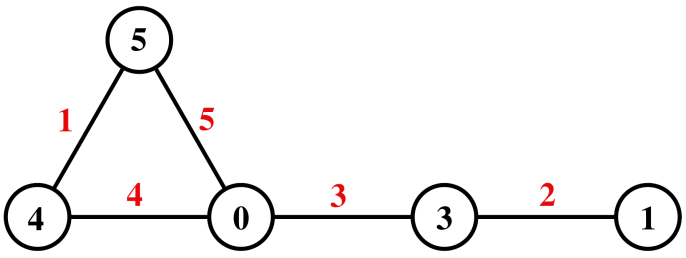
vérifiant $\#S \leq (\#V)/2$. L'objectif est de trouver un sous-ensemble S dont l'éparpillement est minimal. Attention, il s'agit d'un problème combinatoirement difficile, le temps de calcul peut être important et il sera probablement nécessaire de vous limiter à des graphes de taille modeste. Vous pouvez aussi chercher des heuristiques donnant des solutions approchées en un temps raisonnable.

(18) Pour un graphe simple non orienté, on peut définir la notion de *largeur d'arbre* (ou "treewidth"). La définition formelle est donnée ici ou Wikipedia. Pour un graphe de taille modeste (car le problème est combinatoirement difficile), votre programme doit fournir les décompositions en arbre et calculer la valeur de largeur d'arbre du graphe fourni en entrée. Puisqu'on travaillera sur des graphes de petite taille, avoir une sortie graphique illustrant les concepts est un plus.

(19) Soit $G = (V, E)$ un graphe pondéré non orienté. Le problème considéré ici concerne le maintien efficace d'informations sur un arbre couvrant G de poids minimal (ou une forêt couvrante minimale si G n'est pas connexe), lorsque G subit dynamiquement des modifications, tels que des insertions d'arêtes, des suppressions d'arêtes et des mises à jour de poids d'arêtes. On attend de l'algorithme dynamique qu'il effectue les opérations de mise à jour plus rapidement que si l'on recalculait l'arbre minimal tout entier à partir de zéro.

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

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



1. cette condition garantit $\#V \leq \#E + 1$.

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. Pour aller plus loin, vous pouvez produire un générateur d'arbres ou utiliser la bibliothèque house of graphs

(22) *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 : $\forall u \notin N$, $\exists v \in N : (u, v) \in E$.
Étant 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 : les sommets représentent les positions (ou configuration) du jeu et les arcs représentent les options disponibles depuis une position données. Considérez en particulier, le jeu de Nim, le jeu de Chomp et le jeu de Tribonacci (il suffit de lire les règles des trois jeux). Dans chaque cas, l'utilisateur fournira la position initiale du jeu. Ceci assure de ne considérer qu'un graphe fini.

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 ! Même si Wikipédia regorge d'informations utiles, il est (très) dommage de se limiter à cette seule entrée. S'approprier un sujet nécessite la consultation de **plusieurs** sources.
— 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 ?