THÉORIE DES GRAPHES
ORGANISATION & PROJETS 2023–2024

Pour le **lundi 9 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 2023** : choix individuel des modalités d'examen, répartition en groupes et choix des sujets.
— **jeudi 7 décembre 2023 (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 11 décembre 2023** — ordre de passage à déterminer : présentation orale.
— **janvier 2023** : 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. 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.). Le principe 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) Algorithmes d'approximation pour le *problème du voyageur de commerce* (1). Voir la page `https://en.wikipedia.org/wiki/Travelling_salesman_problem` 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 `https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm` et de la comparer avec l'*algorithme de Christofides–Serdyukov*, cf. `https://en.wikipedia.org/wiki/Christofides_algorithm`. 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.

(2) Algorithmes d'approximation pour le *problème du voyageur de commerce* (2). Voir la page `https://en.wikipedia.org/wiki/Travelling_salesman_problem` 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 `https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm` 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. On pourra consulter la référence `https://citeseerx.ist.psu.edu/doc/10.1.1.92.1635` (section 3)

(3) Algorithmes d'approximation pour le *problème du voyageur de commerce* (3). Voir la page `https://en.wikipedia.org/wiki/Travelling_salesman_problem` 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 `https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm` et de la comparer avec la *méthode de recuit simulé*, `https://en.wikipedia.org/wiki/Simulated_annealing`. On pourra consulter la référence `https://citeseerx.ist.psu.edu/doc/10.1.1.92.1635` (section 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.

(4) Le problème des horaires. Prendre la référence "GRAPH THEORY WITH APPLICATIONS" de Bondy et Murty, 5e édition 1982, facilement accessible en ligne, par exemple `https://www.zib.de/groetschel/teaching/WS1314/BondyMurtyGTWA.pdf` On y présente à la section 6.3, le *timetable problem* (pages 97 à 100). Implémenter la solution décrite. On fournira les données d'un problème dans un fichier structuré et votre programme donnera une solution (si elle existe). Essayer de fournir un exemple réaliste se basant sur votre propre expérience d'étudiant.e. Avoir une sortie "graphique" visualisant les sorties est un plus.

(5) 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.

(6) 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 ?

(7) Dans les notes du cours (disponibles en ligne), à la section IV.3, on introduit le polynôme chromatique d'un graphe. Grâce à la proposition IV.3.10, écrire un programme prenant en entrée un graphe simple non orienté (par exemple, décrit dans un fichier) et qui en calcule le polynôme chromatique. Pour vous aider, Mathematica dispose d'une fonction similaire : `ChromaticPolynomial[g,z]`. La sortie fournie par votre programme devra pouvoir être entrée à un logiciel (de votre choix) permettant d'en tracer le graphique (et ainsi en estimer les zéros). C'est un plus si votre programme fournit ce graphique et l'estimation des zéros. Peut-on envisager d'autres méthodes pour calculer ce polynôme chromatique ?

(8) 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.

(9) 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* `https://en.wikipedia.org/wiki/Distance-vector_routing_protocol`. 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.

(10) Le jeu "Clash Royale" est application sur smart-phone dans laquelle deux joueurs s'affrontent en ligne. Le jeu comporte plus d'une centaine de cartes et, lors d'un match, chaque joueur dispose d'un ensemble (deck) de 8 cartes. Vous devez extraire les données disponibles pour l'application cf. `https://developer.clashroyale.com/#/` (par exemple, les 25 derniers decks joués par les top players, comme sur `royaleapi.com`) pour construire un graphe dont les sommets sont les cartes et les arêtes pondérées représentent le nombre de fois que deux cartes se sont retrouvées dans un même deck. Ensuite, analysez le graphe obtenu pour obtenir un maximum d'informations sur les cartes les plus jouées, les decks les plus joués, les combinaisons retenues, etc. Libre à vous de pousser l'analyse.

(11) Construire le graphe des produits achetés conjointement sur `Amazon` en utilisant la base de données `amazon0302` ou `amazon0312` disponible sur `https://graphchallenge.mit.edu/data-sets` Analyser le graphe ainsi construit et essayer d'extraire un maximum d'informations pertinentes en exploitant des notions de théorie des graphes.

(12) Construire le graphe des collaborations entre chercheurs en relativité générale à partir des données nommées `ca-GrQc` disponibles sur `https://graphchallenge.mit.edu/data-sets` Analyser le graphe ainsi construit et essayer d'extraire un maximum d'informations pertinentes en exploitant des notions de théorie des graphes. Il serait en particulier intéressant d'extraire des communautés,
cf. `https://www.analyticsvidhya.com/blog/2020/04/community-detection-graphs-networks/`

(13) Graphes et bioinformatique : Étudier comment les graphes sont utilisés en bioinformatique pour résoudre des problèmes tels que l'alignement de séquences génétiques. On consultera l'article très accessible "A Graph Theoretical Approach to DNA Fragment Assembly", J. Kaptcianos, Amer. *J. of undergraduate research*, vol 7.
`https://www.ajuronline.org/uploads/Volume%207/Issue%201/71B-KaptcianosArt.pdf` Comme détaillé dans la source, vous devrez produire le graphe de de Bruijn d'une séquence ADN fournie en entrée et vous calculerez le nombre de circuits eulériens de celui-ci (faire varier la longueur de la séquence et la taille des blocs considérés pour construire le graphe). Pour aller plus loin, vous considérerez le "Eulerian superpath problem" décrit à la page 12 de l'article et proposerez une solution algorithmique.

(14) 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) `https://en.wikipedia.org/wiki/Barabási-Albert_model`. 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).

(15) 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.

(16) Pour un graphe simple non orienté, on peut définir la notion de *largeur d'arbre* (ou "treewidth"). La définition formelle est donnée ici `https://link.springer.com/referenceworkentry/10.1007/978-0-387-30162-4_431` ou `https://en.wikipedia.org/wiki/Treewidth` 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.

(17) 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. cf. `https://link.springer.com/referenceworkentry/10.1007/978-0-387-30162-4_156`

(18) La définition usuelle d'un coloriage propre des sommets exige que des sommets adjacents reçoivent des couleurs différentes. Dans les années 1980, des "relaxations" de cette définition ont été introduites. L'une d'elles, appelée *coloration défectueuse*, permet à un nombre fixe c de sommets adjacents à un sommet v dans le graphe d'avoir la même couleur que v. Une application est l'allocation de ressources partagées à des moments différents entre des utilisateurs. Quel niveau de concurrence c (taille du défaut) est acceptable ? De nombreux articles ont été écrits et sont encore écrits sur ces colorations. Étant donné un graphe simple non orienté fourni dans un fichier et un paramètre c, écrire un programme qui fourni une coloration avec un défaut c et un nombre minimum de couleurs. On pourra s'inspirer de `https://community.wolfram.com/groups/-/m/t/2602941`

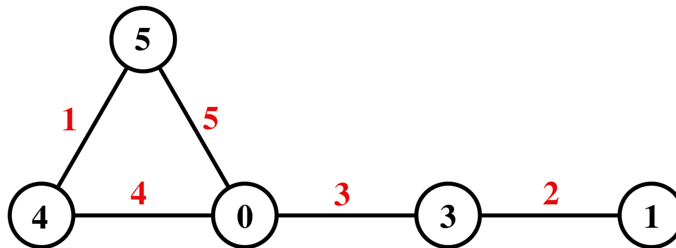
On peut aussi pour un nombre fixe de couleurs, déterminer le défaut minimum permettant d'obtenir un coloriage.

(19) 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`

(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. `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.

(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 $f : V \to \{0, \dots, \#E\}$ 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 $\{1, \dots, \#E\}$. 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/`

(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

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

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 `https://en.wikipedia.org/wiki/Nim`, le jeu de Chomp `https://en.wikipedia.org/wiki/Chomp` et le jeu de Tribonacci `http://www.numdam.org/item/10.1051/ita:2007039.pdf` (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.

(23) Un sommet est *simpliciel* si l'ensemble de ses voisins est une clique. Un graphe "cordal" est un graphe pour lequel tout cycle de longueur au moins 4 possède une corde, i.e., une arête n'appartenant pas au cycle et joignant deux sommets du cycle. Tester si un graphe (simple non orienté) est ou non "cordal". Pour un graphe cordal, vérifier sur des exemples qu'un tel graphe contient toujours un sommet simpliciel et qu'une fois celui-ci supprimé, le graphe obtenu est encore cordal. Itérer la procédure pour supprimer un à un les sommets du graphe.

(24) *Algorithmes de Kruskal et Boruvka.* Étant 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 multi-graphe 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 ! 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 ?