Cours de Graphes – Fiche de TP

Marc Gengler
Département Informatique
Polytech Marseille — usage interne

Année 2016-17

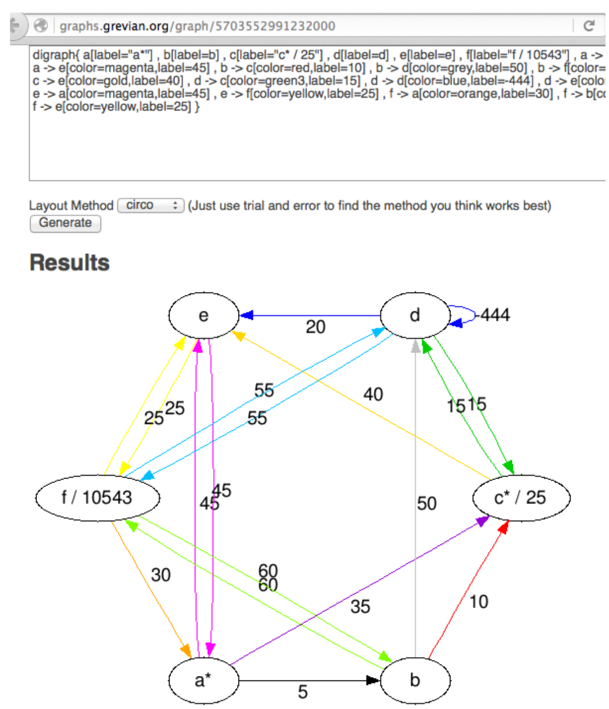
Les fichiers *.h et *.c du présent répertoire constituent l'énoncé du TP associé au cours de graphes. `make` construit un exécutable `monprog`, l'exécute et compare le résultat obtenu au fichier `la_reponse_de_reference`. Seul le fichier `ma_contribution.c` doit être modifiée pour compléter les corps des procédures et fonctions fournies. Le programme principal `main` dans `main.c` commence par dérouler une procédure `exemples` qui permet de se familiariser avec l'environnement fourni. Ensuite, il appelle `ma_contribution`, toujours définie dans `main.c`. Celle-ci fait appel aux fonctionnalités décrites dans `ma_contribution.c` et qui doivent être écrites dans le contexte du TP.

Le fichier `fonctions_graphe.h` donne les signatures des fonctionnalités offertes à l'utilisateur, ainsi qu'une brève description de leurs effets. Le fichier `fonctions_graphe.c` donne l'implantation des ces mêmes fonctionnalités. Ce fichier n'est pas commenté, car il n'est pas sensé être lu par l'utilisateur. Le fichier `fonctions_graphe.c` est compilé vers une librairie `libgra.a` qui sera incluse dans la compilation du programme principal.

Le fichier `reponse_de_reference` donne la réponse obtenue par l'auteur du TP. Elle sert à guider les élèves, mais ne constitue pas la seule réponse possible. En effet, l'ordre de visite des sommets est souvent libre, de même que le choix des couleurs des arcs et arêtes. La réponse obtenue lors des TP peut donc légèrement différer de la réponse de référence. L'environnement fourni permet de colorier les arcs et arêtes à la guise du programmeur. Attention, ce coloriage n'a rien à voir avec les classiques notions de coloriage des arêtes connue en théorie des graphes.

La visualisation des graphes se fait à l'aide de l'outil libre `GraphViz`. Comme celui-ci n'est pas fourni par les services de l'Ecole, on se connectera au site Internet `http://graphs.grevian.org/` qui offre une plate-forme de visualisation. Ce site fournit un descriptif de la syntaxe acceptée en entrée et donne accès, à travers le bouton `Make a Graph`, à une fenêtre qui permet de visualiser un graphe. Il suffira d'un *copier-coller* vers cette fenêtre pour obtenir un affichage des graphes. Ceci peut paraître fastidieux, mais constitue la meilleure solution actuellement disponible. Parfois, le site refuse des syntaxes pourtant correctes.

L'utilisateur appellera `imprime_graphe` ou d'autres fonctions d'impression définies dans `fonctions_graphe.h`. Celle-ci crée la syntaxe qu'il suffit de copier et coller dans le cadre de lecture du site web. Le bouton `Generate` visualise alors le graphe. Le bouton `Layout Method` permet de choisir entre plusieurs politiques d'arrangement des sommets. L'utilisateur choisira ce qui lui convient le mieux. L'option `dot`, et éventuellement `circo`, convient le mieux aux applications de ce TP.



La figure permet de découvrir la plupart des options d'affichage. On a ici affaire à un graphe orienté à cause de l'arc `a -> b` et les arêtes sont dessinées comme deux arcs. Dans un graphe non orienté, les arêtes seraient dessinées comme traits sans flèche. Les arcs et arêtes peuvent être coloriés et/ou pondérés. Les fonctionnalités offertes permettent aussi d'afficher des graphes de flots, comme le montre le dernier graphe de `la_reponse_de_reference`. Un arc sera alors annnoté de la forme `34 / 56`.

Les sommets sont appelés `a`, `b`, etc. Un sommet *mouillé* porte une étoile, comme `a*`. Un sommet qui porte un poids s'écrit `f/10543`, par exemple.

Les modalités de fonctionnement du TP et de son contrôle sont fixées par les enseignant(e)s en charge des TP. La note finale du cours Algo 2 est obtenue en attribuant 15/20 à l'écrit final et 5/20 au TP.

Les diverses fonctions et procédures à réaliser

Ce paragraphe donne un descriptif des différentes fonctions à réaliser. Toutes les opérations de manipulation des graphes se feront exclusivement à l'aide des fonctionnalités dans `fonctions_graphe.h` et on n'est pas autorisé à écrire `graphe->pondere`, etc. La plupart des questions demandent à colorier les graphes et on pourra souvent commencer par écrire un code correct sans coloriage des arêtes, avant de l'adapter pour intégrer le coloriage.

int graphe_non_oriente (t_gra graphe) ; Ce prédicat dit si, oui ou non, le graphe fourni est symétrique. Une réponse `1` signifie que le graphe est symétrique, c'est-à-dire *non orienté*. Le graphe peut être pondéré ou non. Cette fonction vérifie que pour tout arc `a -> b`, on a aussi l'arc `b -> a`. Elle ne prend qu'une dizaine de lignes.

void fermeture_reflexive (t_gra graphe) ; Cette procédure calcule la fermeture réflexive d'un graphe non pondéré. Tout graphe pondéré devra être rejeté via `assert`. L'opération de *fermeture réflexive* consiste à ajouter tout arc de la forme `a -> a` qui manquerait. On observe qu'un arc `a -> a` et une arête `a -- a` sont la même chose. La procédure ne prend que quelques lignes.

void fermeture_reflexive_pondere (t_gra graphe , int poids) ; Ceci est la version pondérée de la fermeture réflexive d'un graphe. Celui-ci doit être pondéré et accepter le poids proposé. A nouveau, quelques lignes de code suffisent.

int graphe_AR (t_gra graphe) ; Ce prédicat dit si, oui ou non, le graphe donné est *anti-réflexif*, c'est-à-dire s'il ne comporte aucun arc de la forme `a -> a`. Tout graphe est accepté et c'est fait en quelques lignes.

int graphe_ARAS (t_gra graphe) ; Ce prédicat dit si, oui ou non, le graphe donné est *anti-réflexif* et *anti-symétrique*. Le graphe peut être pondéré ou non. Un graphe est *anti-symétrique* si la présence d'un arc `a -> b` interdit l'existence de l'arc inverse `b -> a`. Donc, lorsque `a = b`, l'existence de `a -> a` interdit celle de `a -> a` et le graphe est également `anti-réflexif`. Quelques lignes suffisent.

int degre_graphe (t_gra graphe) ; Cette fonction calcule le degré d'un graphe non orienté, pondéré ou non. Elle vérifie bien-sûr l'acceptabilité du graphe fourni. Le *degré d'un graphe* est égal au degré du sommet dont le degré est le plus élevé parmi tous les sommets. Le *degré d'un sommet* est égal au nombre de voisins du sommet. On choisit de dire qu'une arête réflexive de la forme `a -- a` compte pour deux dans le calcul des voisins de `a`. Dix lignes de code suffiront amplement pour produire une solution.

int connexe_vague (t_gra graphe , int view) ; Cette fonction lance des vagues dans un graphe non orienté, pondéré ou non. Elle rend comme valeur le nombre de composantes connexes trouvées. Le paramètre `view` est une option booléenne qui indique si les étapes du calcul doivent être affichées ou non.

Cette fonction implante un algorithme dit *de la vague* qui consiste à choisir un sommet quelconque non encore considéré, à lancer un parcours *en largeur* à partir de ce sommet et à déterminer tous les autres sommets qui sont atteints par ce parcours. On utilisera les états `sec` et `mouillé` associés aux sommets et qui sont accessibles via les fonctions et procédures `mouille`, `tremper` et `secher`.

La fonction `int cherche_sommet_sec ( t_gra graphe )` sert à chercher un sommet de départ pour la vague. Elle retourne, soit l'indice d'un sommet sec qui est un nombre positif, soit la valeur `-1` qui indique le fait qu'il n'y a plus de tel sommet. La fonction fait quelques lignes.

Tant que l'on trouve un sommet de départ `a`, on lance une vague depuis `a`. `connexe_vague` rend le nombre de composantes connexes ainsi trouvées. Une vague qui part de `a` va d'abord toucher les voisins de `a`, ensuite les voisins des voisins, etc. Il s'agit donc d'un parcours en largeur qui sera implanté à l'aide des fonctions de gestion de file offertes dans `fonctions_graphe.h`.

Au départ, tous les sommets sont dans l'état `sec`. Le point de départ `a` de la vague reçoit l'état `mouillé` et est introduit dans la file d'attente. Lorsque la vague touche un sommet `sec`, celui-ci est `mouillé` et introduit dans la file d'attente, ce qui signifie que la vague continue. Lorsque la vague touche un sommet déjà `mouillé`, elle s'arrête car le sommet n'est pas introduit dans la file d'attente. La progression de la vague est terminée lorsque la file d'attente est vide. Tous les sommets `mouillés` font alors partie de la composante connexe.

D'abord, on écrira un simple parcours en largeur qui mouille les sommets. Ensuite, on y introduit une pondération des sommets. Le point de départ aura le poids `0`, ses voisins le poids `1`, etc. Chaque arête où arc reçoit une couleur en fonction de l'étape à laquelle elle/il appartient. Si `view` est vrai, on imprime en plus le graphe à chaque étape; cf. `la_reponse_de_reference`. A la fin, on `sèche` les sommets. Il faut une trentaine de lignes de code.

int est_un_arbre (t_gra graphe) ; Ce prédicat s'applique aux seuls graphes non orientés et anti-réflexifs et dit si, oui ou non, le graphe donné est un arbre. Il commence par compter le nombre d'arêtes à l'aide de la fonction `int nombre_aretes ( t_gra graphe )` pour vérifier s'il y a une arête de moins que le nombre de sommets ou non ? Si tel est le cas, on compte le nombre de composantes connexes via l'appel `connexe_vague( graphe , NON ) ;` pour déterminer si le graphe en comporte une seule ou non ?

void parcours_profondeur_niveaux (t_gra graphe , int depart) ; Elle calcule les *plus courts chemins* depuis le sommet `depart` vers les autres sommets. Elle effectue un parcours en profondeur en incrémentant un compteur à chaque appel récursif. Tout sommet prend comme poids la plus petite valeur avec laquelle il a été touché.

La procédure énumère tous les chemins depuis le point de départ. Ceci fournit bien une solution, mais elle est très inefficace, car le nombre de chemins dans un graphe augmente très vite avec le nombre de sommets. Comme un graphe peut comporter des cycles et des circuits, il faut marquer les sommets qui se situent déjà sur le chemin en construction. Ceci se fait à l'aide des marques `sec` et `mouillé`.

La procédure `parcours_profondeur_niveaux` initialise les poids de tous les sommets à $+\infty$, sauf celui du sommet de départ qui démarre à `0`. Elle passe le contrôle à la procédure récursive `void parcours_profondeur_niveaux_rec ( t_gra graphe , int sommet , int niveau )` qui effectue le parcours. Son code nécessite en gros dix lignes.

void tri_topologique (t_gra graphe) ; Cette procédure s'applique à tout graphe pour lequel on a vérifié qu'il est anti-réflexif et anti-symétrique. Le tri topologique consiste à attribuer à tout sommet un poids strictement plus grand que les poids de tous ces prédécesseurs. De plus, on coloriera les arcs comme illustré dans le fichier `la_reponse_de_reference`.

La fonction `int cherche_sommet_sec_et_predecesseurs_mouilles ( t_gra graphe )` cherche un sommet non valué dont tous les prédécesseurs possèdent déjà un poids. En TD, on a montré qu'un tel choix est toujours possible si le graphe donné est un `Directed Acyclic Graph (DAG)`, c'est-à-dire un graphe anti-réflexif, anti-symétrique et sans circuits.

La procédure `tri_topologique` ne vérifie pas l'absence de circuits. Si jamais le graphe comporte des circuits, la fonction `cherche_sommet_sec_et_predecesseurs_mouilles` ne trouvera tôt ou tard plus de sommet qui convient. Elle avertira alors par un message explicite et rendra comme valeur `-1` pour indiquer à `tri_topologique` que les calculs peuvent être arrêtés. On obtiendra donc un graphe dans lequel certains sommets possèdent un poids et d'autres non. `tri_topologique` nécessite en gros quinze lignes et la fonction de recherche une dizaine.

void multiplie (t_gra graphe) ; `multiplie` calcule les plus courts chemins pour un graphe non pondéré, orienté ou non. Elle commence par fermer réflexivement le graphe et appliquer ensuite la multiplication de matrice suivante

$$M^{2p}(i,j) = \max_k M^p(i,k) \times M^p(k,j)$$

Ici, i, j et k parcourent les sommets. Le produit de la p^e puissance de M, c'est-à-dire M^p, avec elle-même fournit la puissance $2p$ de M, donc M^{2p}. Il suffit de calculer M^m, avec $m \geq$ `taille_graphe( graphe )` -1. Chaque produit utilisera une nouvelle couleur pour les arcs et arêtes.

La définition ci-dessus suppose que l'existence de l'arc `i -> j` correspond à $M(i,j) = 1$ et que son absence correspond à $M(i,j) = 0$. Or, ce comportement est exactement celui produit par `get_arc( graphe , i , j )`.

Le calcul manipulera à tout moment deux graphes, à savoir l'ancien graphe qui correspond à M^p et le nouveau graphe qui correspond à M^{2p} et qui est en cours de construction. A l'itération suivante, le nouveau graphe joue bien-sûr le rôle d'ancien graphe. On pourra facilement copier des graphes grâce à `copie_graphe`.

On introduira les optimisations vues en TD, à savoir éviter de recalculer la diagonale ou toute entrée déjà égale à `1`. De même, on arrêtera de calculer de nouvelles puissances, dès que deux puissances successives sont identiques.

void floyd_warshall (t_gra graphe) ; Cette procédure s'applique à un graphe non pondéré, orienté ou non. Chaque itération utilise une nouvelle couleur. Il faut à nouveau deux graphes, l'ancien et le nouveau en cours de construction. 15 lignes de code suffisent.

void multiplie_pondere (t_gra graphe) ; Cette procédure s'applique à un graphe pondéré, orienté ou non, pour calculer les chemins *les plus légers* entre toute paire de sommets. Le graphe est d'abord fermé réflexivement avec le poids `0`, ensuite on utilise la multiplication de matrices en minimisant les sommes des poids.

Attention, la formulation vue en TD repose sur le fait qu'un arc absent ou une arête absente possède le poids $+\infty$, neutre pour la minimisation. Dans la représentation retenue ici, `get_arc( graphe , i , j )` retourne `0` pour un arc `i -> j` absent ou une arête `i --j` absente. Consulter leur poids constitue une erreur signalée par `assert`.

Les optimisations possibles se limitent au faits d'éviter de recalculer la diagonale et d'arrêter le calcul de nouvelles puissances dès que deux itérations produisent le même résultat. Chaque itération introduit une nouvelle couleur.

void floyd_warshall_pondere (t_gra graphe) ; Cette s'applique à tout graphe pondéré, orienté ou non, pour calculer les chemins *les plus légers* entre toute paire de sommets. Les remarques de mise en garde faites pour `multiplie_pondere` s'appliquent. Chaque itération introduit une nouvelle couleur.

void dijkstra (int depart , t_gra graphe , int table_predecesseurs[ ]) ; Cette procédure calcule les chemins *les plus légers* du sommet `depart` vers tous les autres sommets. Le graphe est pondéré, orienté ou non. Pour que l'algorithme de *Dijkstra* fonctionne il faut que tous les poids soient positifs ou nuls. Ceci est vérfiée à l'aide du prédicat `int verifie_ponderation ( t_gra graphe )`. A la différence de ce qui était supposé en cours, il n'est pas garanti que l'ensemble des sommets du graphe puisse être atteint depuis le sommet `depart`.

La table `table_predecesseurs[ ]` passée en argument sert à mémoriser, pour tout sommet `u`, le prédécesseur `v` du sommet `u` le long du chemin le plus léger du `depart` à `u`. Ce tableau est initialisé convenablement par `dijkstra`.

L'algorithme commence par initialiser le poids de tous les sommets à $+\infty$ et celui de `depart` à `0`. Tous les sommets sont supposés dans l'état `sec`. Il s'agit ensuite de chercher le sommet `u` qui est `sec` et dont le poids n'est pas infini et est minimal parmi tous les sommets secs. La fonction `int cherche_sec_sommet_min ( t_gra graphe )` cherche un tel sommet. Comme on ne suppose pas que tous les sommets puissent nécessairement être atteints depuis le sommet `depart`, il se peut qu'à un moment donné tous les sommets secs aient un poids infini. Dans ce cas, la fonction `cherche_sec_sommet_min` rend `-1`, ce qui aura comme effet de terminer la procédure `dijkstra`.

Si, par contre, `cherche_sec_sommet_min` trouve un sommet `u`, alors il devient `mouillé` et on appelle l'opération `relax( graphe , table_predecesseurs , u , v )` pour tout sommet `v` qui est `sec` et voisin de `u` au sens où l'arc `u-> v` existe.

Finalement, `void relax ( t_gra graphe , int table_predecesseurs[ ] , int pred , int sommet )` propose à `sommet` un nouveau poids qui est la somme de celui de `pred` plus le poids de l'arc `pred -> sommet`. Si cette proposition est meilleure que l'actuel poids de `sommet`, il faut mettre à jour le poids de `sommet` et indiquer `pred` comme étant son prédécesseur.

void dijkstra_maximise_le_min (int depart , t_gra graphe , int table_predecesseurs[ ]) ; Cette procédure est comparable à la précédente à ceci près qu'elle cherche le meilleur chemin de `depart` à tous les autres sommets, le critère de qualité étant le fait que le maillon faible (l'arc de plus petit poids sur le chemin) ait une valeur aussi élevée que possible. Il s'agit donc de maximiser un minimum.

`int cherche_sec_sommet_max ( t_gra graphe )` et `void relax_maximise_le_min ( t_gra graphe , int table_predecesseurs[ ] , int pred , int sommet , int depart )` sont des adaptations assez immédiates des fonctions et procédures précédentes. Pour cette dernière, il faut juste observer que, dans le cas où `pred` est égal à `depart`, le maillon faible du chemin `depart -->> pred` n'est pas égal au poids de `pred`, mais bien infini.

void ford_et_fulkerson (void) ; Cette procédure donne l'algorithme de calcul du meilleur flot à travers un graphe de flot. La vérification de la correction du graphe de flot donné repose sur les trois fonctions `graphe_AR`, `verifie_ponderation` et `parcours_profondeur_niveaux` qu'il faudra écrire avant de pouvoir "décommenter" les trois lignes indiquées et mettre en commentaire les deux lignes de solution provisoire qui suivent. Sinon, il suffit de compléter la procédure en codant les deux procédures ci-dessous.

`void calcule_residuel ( t_gra graphe_flot , t_gra graphe_residuel , int i , int j )` permet de calculer l'éventuel arc résiduel du sommet `i` vers le sommet `j` dans le graphe résiduel `graphe_residuel`. Il faut donc calculer, à partir du graphe de flot `graphe_flot`, la marge qu'il y a dans le sens de `i` vers `j`. Si celle-ci est strictement positive, le graphe résiduel reçoit l'arc pondéré `i -> j` dont le poids est égal à la marge.

`void adapte_flot ( t_gra graphe_flot , int depuis , int vers , int valeur )` permet enfin de modifier le flot dans le graphe de flot `graphe_flot` entre les sommets `depuis` et `vers`. Le flot en question est augmenté d'une quantité égale à `valeur`. On rappelle que ceci revient à augmenter le flot dans un éventuel arc `depuis -> vers` et/ou de diminuer le flot dans un éventuel arc `vers -> depuis`.