Université Paris Cité **Langages à objets avancés**
M1 **Année 2022—2023**

TP Noté
Manipulation de multigraphes pondérés

1 Généralités

Ce TP noté compte pour 20% de la note finale, il vous permet de faire le point sur les premières choses que nous avons vues ensemble et qui sont propres à c++. Il nous permettra aussi de vous fournir un premier retour avant le projet (40%) et l'examen (40%).

Le travail est à réaliser en binôme, c'est un format qui vous permettra d'avoir un regard critique et éventuellement une aide tout le long de sa réalisation. La dernière séance de TP avant les vacances est également réservée à l'accompagnement de ce sujet. Vous avez presque deux semaines pour le réaliser, et nous souhaitons absolument qu'il soit terminé avant les vacances. Le dépôt est à faire sur Moodle au plus tard le dimanche 30/10/2022 à 23h59 sous forme d'une **archive tar** (utiliser impérativement la commande `tar` [1]). Vous êtes trop nombreux pour que nous acceptions des retards ou des envois par mail. Merci de respecter cette date limite, et d'anticiper les problèmes techniques.

Derniers conseils : avant de vous lancer dans la programmation, réfléchissez bien au diagramme de classes. Quelles sont les relations qui permettent de respecter les consignes ? Attention, il y a des décisions conceptuelles à prendre qui ne sont pas explicitement écrites. Réfléchissez bien également à ce qui doit être déclaré constant dans votre programme, à ce qui doit être passé par référence en argument ou retour de vos méthodes et fonctions.

2 Description de l'objectif

Vous allez écrire un ensemble de classes qui permettent de créer et manipuler des multigraphes pondérés. Un graphe est défini par un ensemble de sommets et d'arêtes.

Une arête est définie, **une fois pour toutes**, par deux sommets, ses extrémités. Elle a également un poids (entier) qui, lui, peut être modifié. Il peut y avoir plusieurs arêtes (chacune avec son poids propre) avec les mêmes extrémités, c'est pourquoi on parle de multi-graphe.

Les sommets ont une étiquette qui est une chaîne de caractères. Il faudra bien faire attention à ne pas multiplier les créations de sommets. Si les arêtes a et b d'un graphe ont le sommet s en commun, alors elles doivent référencer le **même** sommet. Il en est de même pour les arêtes qui peuvent appartenir à plusieurs graphes.

1. `tar cvf mon_fichier.tar fic1 fic2 ...` pour créer l'archive `mon_fichier.tar` composée des fichiers `fic1`, `fic2`, ...

Par ailleurs, vous créerez un *garbage collector* qui aura pour fonction de comptabiliser toutes les créations de pointeurs sur des :

— sommets lors de la construction d'une arête,

— arêtes lors de la construction d'un graphe,

Il permettra que les libérations nécessaires soient faites quand le moment de la destruction sera venu.

Pour chaque classe définie ci-dessous, vous devrez pouvoir en afficher une instance en surchargeant l'opérateur `<<`.

Enfin, pour mettre à l'épreuve votre modélisation vous implémenterez un algorithme classique de calcul d'arbre couvrant.

3 Travail à faire

Nous détaillons ici les classes que vous avez à écrire, et ce que nous attendons de votre part.

Graphe

On devra pouvoir créer un graphe des différentes façons suivantes :

— à partir d'un ensemble de sommets et d'un ensemble d'arêtes ;

— à partir d'un graphe existant.

Étant donné un graphe G, on doit pouvoir lui appliquer les méthodes suivantes :

— `ajoute_sommet` qui, étant donné soit un sommet existant, soit une étiquette, crée le sommet dans ce dernier cas et ajoute le sommet au graphe ;

— `ajoute_arete` qui, étant donné soit une arête existante, soit deux sommets existants et un poids, soit deux étiquettes et un poids, crée les sommets et l'arête si nécessaire et les ajoute au graphe ;

— `poids` qui retourne le poids de G, c'est-à-dire la somme des poids de ses arêtes ;

— `symetrise` qui transforme le graphe en s'assurant que si l'arc (u, v) existe alors l'arc (v, u) de même poids existe aussi (quitte à l'ajouter).

— `kruskal` L'objectif pour nous est que vous ayez une manipulation non triviale à faire avec les classes que vous avez écrites pour les graphes. Gardez cette question pour la fin.
Cet algorithme est relativement simple et vous est certainement familier puisque c'est l'un des premiers algorithmes de graphes qu'on étudie. Il a été écrit initialement dans le cas de graphes non orientés, mais il est évidement applicable à des graphes "symétrisés" par la méthode définie juste précédemment. Il permet d'obtenir un graphe "symétrisé" couvrant de poids minimal. Vous pouvez trouver sa description en pseudo langage sur Wikipédia :
`https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal`.
Voici quelques explications supplémentaires :

— `pour chaque sommet v faire créerEnsemble(v)` cela consiste simplement à étiqueter chaque sommet par des entiers différents.

— Le tri n'a pas à être optimisé. Vous pouvez utiliser celui que vous considérez être le plus simple à écrire.

— `find(u)` revient [2] juste à lire l'entier actuellement attribué au sommet u

— `union(u,v)` revient juste à donner le même numéro à tous les sommets qui ont le même numéro que u ou que v, vous pouvez pour cela utiliser le numéro de u par exemple.

Arête

On devra pouvoir créer une arête des différentes façons suivantes :

— à partir de deux étiquettes (chaînes de caractères) et d'un poids (entier) ;

— à partir de deux sommets existants et d'un poids ;

— à partir d'une arête existante.

Sommet

On devra pouvoir créer un sommet des différentes façons suivantes :

— à partir d'une étiquette (chaînes de caractères) ;

— à partir d'un sommet existant.

Prévoyez un attribut entier pour les sommets qui servira à faire le marquage, cela vous sera utile pour implémenter Kruskal.

GC

Cette classe est le *garbage collector*. Elle doit donc permettre de stocker les références des pointeurs sur sommet et arête créés.

Diagramme UML

Faites un diagramme UML en respectant bien la syntaxe vue en cours. **Joignez une image ou un pdf** mais pas le fichier au format d'un outil qui vous est local (bouml, StarUML....)

Makefile et tests

Votre programme devra pouvoir être compilé avec la commande `make`. Tout programme qui ne respectera pas cette consigne aura une note très dégradée.

Il faut présenter des tests pour que nous puissions constater que vos différentes constructions et destructions fonctionnent, ainsi que les actions requises sur un graphe.

2. les interprétation données ici pour `find` et **union** permettent de simplifier les choses

Rédigez vos affichages afin que nous soyons bien guidés : ce qui est effectué / attendu doit être exprimé dans la console.

Si vous avez prévus des tests que vous souhaitez séparer, pour que nous ayons une ergonomie uniforme, faites en sorte que nous puissions les lancer avec la commande `make test_i`, et regroupez dans `make all` un descriptif court de la nature des tests que vous nous présentez.

Modalités de rendu

Vous rendrez une archive au format `tar` contenant les fichiers suivants :

— le makefile,

— vos fichiers `.cpp` et `.hpp`,

— un fichier `binome` qui contient les nom, prénom et numéro d'étudiant de chaque étudiant du binôme, écrivez les dans cet ordre, et en allant à la ligne pour le second binôme.

— le diagramme UML au format `pdf` ou un format image (`jpg`, `ppm`, ...).