Projet : Algorithme de Kruskal.

Le projet consise à programmer l'algorithme de KRUSKAL de calcul d'un plus petit arbre recouvrant et d'en donner une petite application.

Le travail demandé est à faire en binôme ou en solo, pas de trinôme et devra être rendu au plus tard le 20 janvier 2025 ! Un espace de rendu est ouvert sur UPdago. Le travail à rendre consiste en un programme (sous forme des fichiers sources) et un rapport ; si nous avons des doutes sur la paternité de certains codes fournis, nous nous réservons le droit de convoquer quelques équipes pour un oral de contrôle.

Si vous avez des questions, n'hésitez pas à les poser, nous y répondrons, soit par mail, soit par un ajout sur la page du projet dans l'ENT. Lisez bien toutes les directives qui suivent, elles sont importantes pour la plupart.

Le travail de programmation.

La programmation devra être **modulaire et soignée**, quel que soit le langage utilisé (vous pouvez utiliser les langages suivants : Ada, C, C++, Java **(en version 1.8 uniquement)** et Ocaml.

Quelques éléments concernant la programmation :

- Vous implémenterez les graphes simples pondérés non orientés de deux manières différentes :
  - par leur listes de successeurs (liste d'adjacence)
  - par leur matrice d'adjacence ;

  Cela vous permettra de comparer les performances de vos implémentations en fonction de la taille et de la densité du graphe manipulé.
- en matière de structures de données **vous n'êtes pas autorisés à utiliser des composants tout faits, vous devez tout implémenter par vous-mêmes**. Par exemple, inutile de vous/-nous demander si vous pouvez utiliser une structure d'ArrayList ou TreeSet de Java, la réponse est « **NON** » ; et cette réponse sera la même s'il s'agit quelque soit la collection (`heap`, `queue` ou structure structure de données de la bibliothèque standard de Java ou C++, ou... de C++, d'Ocaml, etc.
- les sommets des graphes manipulés seront les entiers de 1 à n, l'entier n étant l'ordre du graphe ; attention, nous n'examinerons aucun projet qui numéroterait les sommets différemment ;
- On supposera que les coûts des arcs sont des nombres entiers et on ne manipulera pour cela que le type int ; pour ce genre d'algorithme, on a besoin de représenter un coût infini, on ne se cassera pas la tête et on supposera que l'entier maximum représentable est suffisant pour donner une bonne approximation de l'infini (`Integer.MAX_VALUE` vaut $2^{31} - 1 = 2147483647$ et cela suffit pour des exemples raisonnables).

Format d'entrée des graphes manipulés

Le format d'entrée texte des graphes manipulés est une suite d'entiers.

- Un premier entier (noté n dans ce qui suit) qui est la valeur de l'ordre du graphe.
- Après ce premier entier, on trouve une suite (de longueur a priori inconnue) d'entiers positifs ou nuls, x y_1 c_1 y_2 c_2 ... y_k c_k 0 séparés par des espaces (attention au 0 qui termine la suite) ; cette suite est structurée de la manière suivante :
  - un premier entier, x, qui est le numéro d'un sommet ;

- puis une suite de couples d'entiers y c où y est un autre sommet, voisin de x, et c est le coût de l'arête (x, y) ; on suppose tous les $c > 0$;
- cette suite de couples se termine dès que l'on rencontre l'entier 0 en lecture.

Les n sommets devront avoir été décrits de cette manière, avec leurs successeurs et les arcs correspondants, y compris les sommets sans successeurs.

Comme les graphes sont non orientés, pour ne pas représenter deux fois une arête (x, y), on la représentera seulement dans les voisins du sommet du plus grand numéro ; par exemple l'arête entre les sommets 4 et 6 sera définie uniquement dans les voisins du sommet 6.

Voici un exemple de graphe, avec sa description dans le format précédent ; il contient (entre autres) une arête de coût 34 entre les sommets 5 et 6.

```
6
1 0
2 1 61 0
3 1 50 2 1 0
4 3 68 0
5 4 84 0
6 1 28 5 34 0
```

Il est important de noter que la forme particulière de présentation que j'ai utilisée est sans aucune importance, on pourrait tout à fait présenter ce graphe de la manière suivante : `6 1 0 2 1 61 0 3 1 50 2 1 0 4 3 68 0 5 4 84 0 6 1 28 5 34 0`

La seule chose imposée concerne les blancs séparant les entiers. Votre outil devra permettre de lire tout fichier texte respectant cette présentation.

Fonctionnement demandé pour le programme

Les deux programmes demandés, de noms `KruskalL` (où le graphe est représenté par liste d'adjacence) et `KruskalM` , auront deux ou trois arguments sur la ligne de commande.

- le premier argument est le nom d'un fichier qui contient la description du graphe, au format explicité précédemment ;
- le second argument est le numéro d'un sommet s, qui est le sommet de départ à partir du quel on calcule un arbre recouvrant de poids minimum ;
- le troisième argument optionnel est le nom d'un fichier dans lequel – une fois le calcul d'un plus petit arbre recouvrant est terminé – on écrit le résultat du calcul du plus petit arbre recouvrant sous la forme suivante.
  - Une ligne indiquant `LE GRAPHE EST CONNEXE` ou `LE GRAPHE N'EST PAS CONNEXE`.
  - Une ligne avec un entier indiquant le coût de l'arbre (ou des arbres si le graphe n'était pas connexe) recouvrant que vous avez trouvé.
  - Puis toutes les arêtes de l'arbre (ou des arbres si le graphe n'était pas connexe), énumérées comme suit.

    Chaque arête x est représenté comme `(x -> y :  c)` avec c est le coût de l'arête (x, y)

    Comme les graphes sont non orientés, pour ne pas représenter deux fois une arête (x, y), on la représentera seulement une fois, et dans l'ordre des sommets croissants. Par exemple l'arête $(9, 4)$ (ou $(4, 9)$ puisque les graphes ne sont pas orientés) de coût 123 sera affichée une seule fois comme `(4 -> 9 :  123)`, et apparaîtrait avant les arêtes $(4, 10)$ et $(5, 6)$.

– En dernier lieu, vous indiquerez le temps CPU utilisé par le calcul de votre arbre recouvrant. Attention, ce temps CPU ne devra prendre en compte que le temps de calcul CPU pour le calcul de l'arbre recouvrant (hors saisie du graphe, et affichage de l'arbre recouvrant résultat.

si le nom de fichier de sortie n'est pas indiqué, alors il est remplacé par un simple affichage à l'écran.

Le rapport du projet

Le rapport doit rappeler, de manière très courte, les éléments importants de l'algorithme de Kruskal et de son implémentation.

Et doit également contenir les points suivants.

- Le détail des différentes structures de données utilisées pour votre application.
- Les explications nécessaires pour comprendre votre programmation : architecture et organisation de vos sources, subtilités (s'il y en a). **Dans tous les cas, votre rapport doit permettre de se retrouver en quelques instants dans l'organisation de votre programme et dans sa conception**.
- Une partie détaillant les instructions à exécuter pour compiler, et exécuter votre programme, ainsi qu'un bref mode d'emploi, le cas échéant. Cette partie est à mettre également dans un fichier README.txt séparé.
- Une étude de la complexité, vous devrez en particulier étudier le comportement des deux versions (liste d'adjacence ou matrice et expliquer les différences s'il y en a) sur des graphes de tailles et densités différente.

Le travail à rendre

Tout le travail devra être rendu dans une archive ZIP (exclusivement ce format d'archive, pas de 7z, ou rar...) organisée de la manière suivante :

- Un fichier `.zip` ayant pour nom les noms des personnes ayant réalisé le projet par ordre alphabétique séparés par un blanc souligné, les noms étant en minuscules sans accents ; par exemple `mangore_rameau.zip` si le binôme est formé de Agustín Barrios Mangoré et Jean-Philippe Rameau. L'archive doit se décompresser dans un répertoire ayant le même nom, `mangore_rameau` dans mon exemple (dans la suite, j'appelle ce répertoire, le « répertoire racine »).
- Dans ce répertoire racine, on trouvera le fichier contenant le rapport, dont le nom sera `rapport_barrios_mangore.pdf` (**n'oubliez pas de mettre les noms des programmeurs sur la page de garde du rapport**), au format PDF à l'exclusion de tout autre format.
- On trouvera également dans ce répertoire un autre répertoire, nommé `src` contenant la hiérarchie des fichiers sources, organisée comme il le faut de sorte que l'on puisse compiler et exécuter dans ce répertoire, votre réalisation.
- N'oubliez pas que votre programme doit pouvoir être compilé et exécuté en dehors de tout IDE, dans un bête terminal, et les instructions que vous mettrez dans votre rapport et dans le fichier `README.txt` doivent être adaptées à cette situation.
- Dans le répertoire racine, on trouvera également un fichier `README.txt` qui contiendra essentiellement les instructions à exécuter pour compiler et exécuter votre programme, plus quelques autres informations qui vous sembleront utiles à cet endroit. Ce fichier `README.txt` ne remplace pas le rapport, et vice-versa.

- si vous avez d'autres données à fournir (on ne sait jamais) vous les placerez dans un autre sous-répertoire du répertoire racine de votre archive.

Evaluation

Vous serez évalué sur :

- La propreté et modularité de votre code. Attention, cette partie est un prérequis. Nous ne voulons pas voir de code illisible, non réutilisable, non modulaire, incomprehensible. Je le redis chaque année, si ce point est en dessous de tout, cela impliquerait une très mauvaise note globale, même si les autres points étaient parfaits.
- L'implémentation de votre algorithme et de votre gestion de graphe.
- Le respect des consignes dans l'exécution.
- La partie rapport, concernant votre votre implémentation et le manuel d'installation et d'execution..
- La partie rapport concernant l'étude des performances de votre programme.