Théorie des graphes

TD/TP 2 - TP noté

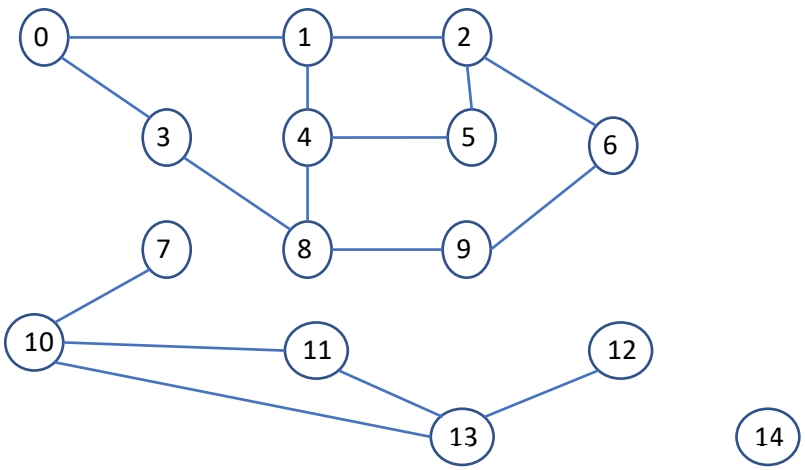
(à rendre le 17/03)

Parcourir un graphe (BFS – DFS)
Recherche des composantes connexes dans un graphe non orienté
Graphe eulérien
Implémentation (C++)

Vous devrez rendre un programme C++ respectant les consignes données (balisage **C++**) tout au long du présent sujet, au plus tard le 17/03/2019. Vous pouvez travailler en binôme. Déposez le code sous forme d'un dossier compressé *.zip* ou *.rar* (***Nom1_Nom2_numTD_TP2.zip***) contenant toutes les sources (*.h* et *.cpp*) et les fichiers d'entrées *.txt*.

Un code de base vous est proposé. Vous n'êtes pas obligé de l'utiliser. Vous pouvez juste vous en inspirer, le modifier ou l'utiliser tel quel.

Soit le graphe G dont la représentation sagittale est donnée ci-dessous :



- **Quel est son ordre ? quelle est sa taille ?**

I. Parcourir un graphe

1. Parcourir en largeur BFS

- **Effectuer un parcours en largeur à partir du sommet 1**. Dérouler l'algorithme en complétant le tableau ci-dessous comme vu en cours.
  Pour chaque sommet on indique sa couleur et son prédécesseur.

Sommets / étapes	0	1	…	Etat de la file
0	B ?	G -	…	1
1	G 1	N -	…	0 …
…				

A chaque étape, indiquer la couleur et le prédécesseur des sommets.

- **Quel est l'ordre de découverte des sommets ?**
- **Dessiner l'arborescence obtenue**

Rappel : le parcours BFS à partir d'un sommet initial s_0 donne les plus courts chemins de s_0 aux autres sommets.

2. Parcourir en profondeur DFS (avec une pile)

Rappel : avec un parcours en profondeur, le résultat obtenu dépend de l'ordre de visite des successeurs de chaque sommet.

- **Effectuer deux parcours en profondeur à partir du sommet 1** : **pour le premier parcours, les successeurs de chaque sommet seront visités suivant l'ordre de leurs numéros ; pour le deuxième, dans l'ordre inverse.**
  Dérouler l'algorithme en complétant le tableau ci-dessous comme vu en cours.

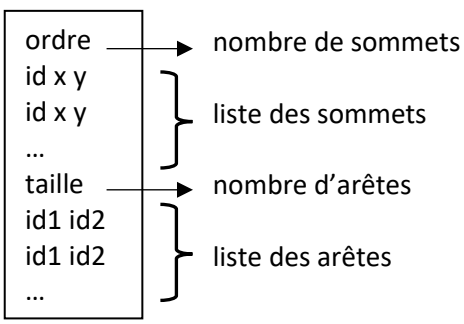
Sommets / étapes	0	1	…	Etat de la pile
0	B ?	G -	…	1
1	G 1	N -	…	…
…				

- **Pour chaque parcours, dessiner et comparer les arborescences obtenues**

3. Implémentation des algorithmes

Vous devez **rendre un programme c++ capable de :**

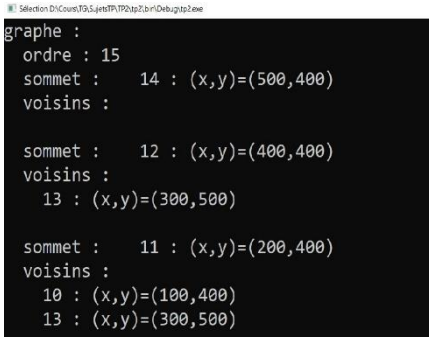
C++ • **Charger un graphe à partir d'un fichier texte au format suivant :**



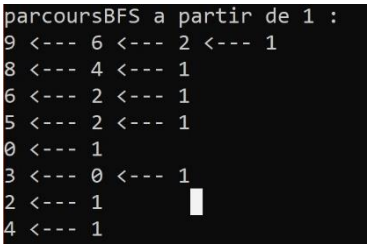
graphe.txt

Plusieurs fichiers d'entrée respectant ce format vous sont donnés pour tester vos algorithmes. Le fichier *graphe1.txt* correspond au graphe G de l'énoncé.

- **Afficher le graphe en console :**
  - L'ordre du graphe
  - Pour chaque sommet :
    - ses données
    - la liste de ses successeurs



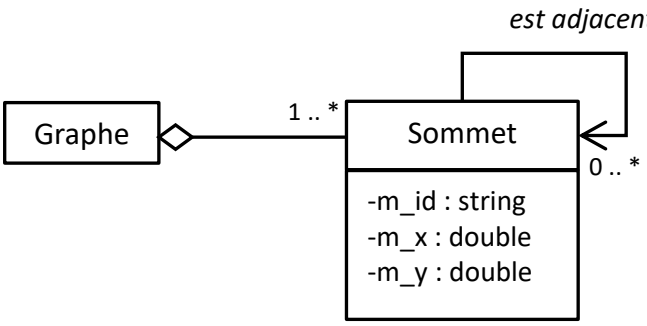
- Demander l'identifiant d'un sommet
- **Lancer des parcours BFS et DFS sur le sommet choisi et afficher le résultats des parcours** sous la forme id <-- id <-- id



Pour utiliser le code proposé :

- Télécharger *tp2.zip*, dézipper, ouvrir le projet, tester, coder les méthodes *parcoursBFS* et *parcoursDFS* de la classe Sommet, tester et comparer avec vos résultats obtenus sur papier.

Le modèle proposé est le suivant :



modélisation par liste de successeurs

La liste des sommets du graphe est stockée dans une *unordered_map*. Une map est un conteneur associatif qui contient des *paires* clé-valeur avec des clés uniques. La recherche, l'insertion et la suppression ont une complexité moyenne en temps constant. Dans le code proposé, l'identifiant m_id d'un sommet sert de clé. La valeur associée est un pointeur sur le sommet.

Les listes d'adjacence sont stockées dans des vecteurs de pointeurs sur sommet.

Le constructeur de la classe graphe charge le graphe à partir d'un fichier texte.

Pour coder les parcours, utilisez les classes *queue* (file) et *stack* (pile).

Pour le marquage des sommets, il suffit ici de savoir si le sommet a déjà été découvert ou pas : vous pouvez utiliser un *unordered_set*. Même principe qu'une *unordered_map* mais c'est directement la valeur qui sert de clé. A chaque fois qu'un sommet est découvert, il y est inséré (méthode *insert*). Pour vérifier si un sommet a déjà été découvert, il suffit de le chercher dans le set (méthode *find*).

A chaque fois qu'un sommet est découvert, il faut noter son prédécesseur. Dans le code proposé, on utilise une *unordered_map* avec comme clé l'id du sommet découvert, comme valeur l'id de son prédécesseur.

A la fin du parcours, cette liste contient toutes les informations pour retrouver l'arborescence obtenue. C'est le résultat retourné.

II. Recherche des composantes connexes

- **Revoir les définitions** : relation de connexité, graphe connexe, composante connexe.
- **Combien le graphe G possède-t-il de composantes connexes ?**

C++

- **Compléter le programme précédent pour qu'il recherche et affiche les composantes connexes du graphe.**
  - Pour chaque composante : les ids de ses sommets
  - Le nombre de composantes

```
composantes connexes :
   cc1
     14
   cc2
     10       7       11       12       13
   cc3
     2       0       6       5       8       9       1       3       4
nb cc :3
```

Rappel : un parcours BFS ou DFS à partir d'un sommet initial s_0 marque tous les sommets appartenant à la composante connexe de s_0. Utiliser judicieusement l'une des méthodes codées précédemment !

Pour trouver toutes les composantes connexes, il suffit de relancer un parcours sur un sommet non-marqué … tant qu'il reste des sommets non marqués.

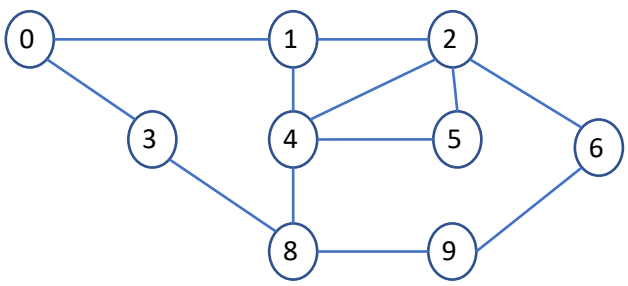
Pour utiliser le code proposé :

Coder la méthode *rechercherCC* de la classe Sommet. Elle retourne la liste des ids de la composante connexe du sommet sous forme d'un *unordered_set*.

Coder la méthode *rechercher_afficherToutesCC* de la classe graphe. Elle recherche toutes les composantes connexes, les affiche, et retourne leur nombre.

III. Graphe eulérien – recherche de chaines ou de cycles eulériens

- **Revoir les définitions** : chaines et cycles eulériens
- **Revoir le théorème d'Euler**
- **Le graphe G admet-il une chaine ou un cycle eulérien ? et ses composantes connexes ?**
- **Et le graphe G' ci-dessous ?**



Le fichier *graphe2.txt* correspond à ce graphe G'.

- **Quels sommets sont les extrémités d'une chaine eulérienne si elle existe ?**

C++

- **Compléter le programme précédent pour qu'il détermine et affiche si le graphe admet un cycle ou une chaine eulérienne :**
  - **Ajouter la méthode *getDegre* dans la classe sommet qui retourne le degré du sommet**
  - **Ajouter la méthode *isEulerien* dans la classe graphe qui détermine et affiche si le graphe admet une chaîne eulérienne (retourne 1), un cycle eulérien (retourne 2) ou rien d'eulérien (retourne 0).**

- **Chercher et coder un algorithme qui trouve et affiche une chaine ou un cycle eulérien s'ils existent.**