Travaux pratiques

Un graphe est une structure de données composée d'un ensemble de sommets, et d'un ensemble de relations entre ces sommets. Formellement, on note un graphe $G = (S, A)$ où S est l'ensemble des sommets et $A \subset S \times S$ l'ensemble des arêtes reliant ces sommets.
Une arête est donc une paire de sommets.Dans le cas d'un graphe orienté, le premier élément de l'arête représente la source, le deuxième la cible. Une autre information qui peut être importante est le poids de l'arête si le graphe est pondéré.
Par exemple, on peut voir le réseau des routes nationales comme un graphe où les villes sont des sommets et les routes sont des arêtes. Dans ce cas il y aura une arête entre deux sommets S_1 et S_2 si et seulement si il y a une route nationale directe de la ville représentée par S_1 et la ville représentée par S_2.
Dans cet exemple le graphe est orienté car on peut imaginer des routes à sens unique. Il est également pondéré, le poids d'une arête correspond à la longueur en kilomètre reliant deux villes.
Pour représenter un graphe, on a besoin de connaître le nombre de sommets S numérotés de 0 à $S-1$.Pour représenter les arêtes on peut utiliser des listes d'adjacences. Pour chaque sommet du graphe on a la liste des sommets qui lui sont directement reliés. Quand le graphe est pondéré, les éléments des listes d'adjacences contiennent aussi le poids de l'arête en plus du sommet cible.

Il est recommandé d'utiliser un vecteur de listes (ou directement un tableau de listes si on est sûr que le nombre de sommets ne changera pas) car le vecteur (ou tableau) permet d'accéder directement en temps constant l'ensemble des arêtes partant de n'importe quel sommet.
Pour gérer les listes d'adjacences, nous utiliserons les listes car elles permettent l'ajout en temps constant et le parcours en temps linéaire. Les éléments des listes peuvent être des *pairs* avec par exemple le premier élément qui désigne le sommet cible de l'arête et le second son poids. Nous utiliserons donc les structures suivantes pour gérer, nos sommets, arêtes et liste d'adjacences :

```
typedef std::pair<uint, uint> arete;
typedef std::list<arete> liste_adjacences;
std::vector<liste_adjacences> aretes;
```

Exercice 1
Créer une classe Graph, qui aura un constructeur qui prend en paramètre le nombre de sommets du graphe. Cette classe devra avoir une méthode ajouterArete pour ajouter des arêtes au graphe. Cette méthode prend trois arguments : le sommet de départ de la nouvelle arête, son sommet d'arrivée et le poids de l'arête.

Exercice 2
Créer une méthode d'affichage de votre graphe. On listera par sommet, la liste des arêtes avec leur poids.

Exercice 3
Créer une métode qui permettra de remplir votre graphe (saisie clavier ou lecture dans un fichier au choix).

Exercice 4
Coder une méthode qui permettra le parcours en profondeur de votre graphe, afin de connaîte pour deux sommets passés en argument la liste des chemins possibles entre ces deux sommets ainsi que leur poids.

Exercice 5
Idem avec un parcours en largeur.

Exercice 6
En vous aidant de la page Wikipedia, coder la méthode *plus_court_chemin* qui déterminera le chemin de poids le plus faible entre deux sommets. Cette méthode utilisera l'algorithme de Dijkstra.