TP N°02 : Théorie des graphes et algorithmes fondamentaux

Objectif TP : Créer et afficher un graphe de différentes manières et implémenter les algorithmes fondamentaux de la théorie des graphes.

Remarque : Ecrivez des commentaires dans vos programmes.

Partie 1

Exercice 2.1:

Question 1: (Graphe)

Écrivez un programme qui doit être capable de prendre en compte n’importe quel graphe orienté valué, sachant que :

- Les sommets sont des numéros entiers de ‘**0**’ à ‘**n-1**’ pour un graphe contenant **n** sommets ;
- L’ensemble des **m** arêtes de **G** est stocké dans un tableau **edge** de taille **m × 2** et dont les entrées appartiennent à **{0,…,n−1}**.
- L’ensemble des **m** arêtes de **G** est stocké dans un tableau **edge** de taille **m × 2** et dont les entrées appartiennent à **{0,…,n−1}**.
- Ainsi, si **xy** est l’arête de **G** d’indice **k**, on aura **edge[k][0]=x** et **edge[k][1]= y**.
- Les valeurs associées aux graphes sont des nombres entiers quelconques ;
- Il y a au plus un arc reliant deux sommets ;
- Il n’y a pas de boucle ;
- Il peut y avoir des sommets isolés (sans prédécesseur ni successeur).

Question 2: (Afficher un graphe)

Écrivez la fonction ***Affichage_Graphe(int n, int m, int edge[][2])***, qui afficher le graphe créé à la question précédente.

Question 3: (Création d’un graphe aléatoire)

Méthode 1:

Écrivez une fonction ***Graphe_Random(int n, int m, int edge[][2])*** qui engendre aléatoirement le tableau **edge** en tirant au hasard chacune de ses entrées On permettra la création d’arêtes multiples ou de boucles.

A la fin, afficher ce graphe par la fonction d’affichage précédente.

Méthode 2:

Écrivez la fonction ***Voisins_random(int n, int m, vector<int> voisins[ ])*** qui engendre aléatoirement les listes de voisins d'un graphe aléatoire avec **n** sommets et **m** arêtes. On prendra garde a :

- la symétrie: si **x** est voisin de **y**, alors **y** est voisin de **x**.
- ne pas créer de boucle.
- ne pas créer d'arête multiple.

A la fin, afficher ce graphe par la fonction d'affichage précédente.

Note : On pourra utiliser les appels :

- ***srand (time(NULL))*** // Initialise la graine (seed) de la fonction rand sur l'horloge.
- ***rand()%k*** // Retourne un entier entre **0** et **k - 1**.

Partie 2

Utilisez toutes les fonctions créées dans la partie 1 dans les exercices de la partie 2.

Exercice 2.2: (Parcours de graphes)

Q1) Implémenter l'algorithme de parcours en largeur (**Breadth First Search = BFS**) vu dans le cours **BFS(GRAPHE G, int S_0)**, sachant que **G** est graphe et le sommet de départ du parcours est la racine **S_0**.

- Utiliser une file d'attente (FIFO = First In First Out).
- Afficher la coloration des nœuds de cette file d'attente à chaque étape de l'algorithme.
- A la fin de cet algorithme, afficher l'ordre dans lequel les nœuds du graphe **G** sont parcourus en largeur et l'arborescence associée au parcours **BFS**.

Q2) Implémenter l'algorithme de parcours en profondeur (**Depth First Search = DFS**) vu dans le cours **DFS(GRAPHE G, int S_0)**, sachant que **G** est graphe et le sommet de départ du parcours est la racine **S_0**.

- Utiliser une pile (LIFO = Last In First Out).
- Afficher la coloration des nœuds de cette pile à chaque étape de l'algorithme.
- A la fin de cet algorithme, afficher l'ordre dans lequel les nœuds du graphe **G** sont parcourus en largeur et l'arborescence associée au parcours **DFS**.

Exercice 2.3: (Problème du plus court chemin)

Q1) Écrivez une fonction ***Dijkstra(*GRAPHE *G, int S₀, int π*)** sur le modèle de l'algorithme de ***Dijkstra*** que nous avons vu dans le cours, sachant que ***G*** est graphe, la racine de l'arbre des plus courts chemins est le sommet ***S_0*** et le tableau ***π*** représente l'arbre des plus courts chemins.

- Avant d'exécuter la fonction de **Dijkstra**, elle doit tester si la condition est vraie ou non (arcs avec des coûts négatifs) et ensuite l'exécuter.
- Afficher la constriction de l'arbre à chaque itération (étape par étape).
- A la fin de cet algorithme, afficher l'arbre des plus courts chemins et les coûts minimaux donnés pour chaque nœud par l'algorithme de **Dijkstra**.

Q2) Écrivez une fonction ***Bellman_Ford(*GRAPHE *G, int S_0, int π*)** sur le modèle de l'algorithme de ***Dijkstra*** que nous avons vu dans le cours, sachant que ***G*** est graphe, la racine de l'arbre des plus courts chemins est le sommet ***S_0*** et le tableau ***π*** représente l'arbre des plus courts chemins.

- Afficher la constriction de l'arbre à chaque itération (étape par étape).
- A la fin de cet algorithme, afficher qu'il existe un **circuit absorbant,** ou afficher l'arbre des plus courts chemins et les coûts minimaux donnés pour chaque nœud par l'algorithme de ***Bellman-Ford***.

Exercice 2.4: (Arbres couvrants minimaux)

Q1) L'existence d'un arbre couvrant implique que le graphe est connexe (et réciproquement) puisque qu'à partir de n'importe quel sommet il faut pouvoir accéder à tous les autres (il existe un chemin entre chaque paire de sommets du graphe).

Écrivez une fonction **Est_Connexe(GRAPHE *G*)** qui teste si un graphe est connexe.

Q2) Écrivez une fonction **Kruskal(GRAPHE *G*, int K)** sur le modèle de l'algorithme de **Kruskal** que nous avons vu dans le cours, sachant que ***G*** est graphe, et **K** représente le tableau de l'arbre couvrante de poids minimal.

- Utiliser la fonction **Est_Connexe** lors de la programmation de la fonction **Kruskal**.
- Afficher la constriction de l'arbre couvrante de poids minimal à chaque itération (étape par étape).
- A la fin de cet algorithme, afficher l'arbre couvrante de poids minimal et le coût minimal donné par l'algorithme de **Kruskal**.

Q3) Écrivez une fonction **Prim(GRAPHE G, int K)** sur le modèle de l'algorithme de **Prim** que nous avons vu dans le cours, sachant que **G** est graphe, et **K** représente le tableau de l'arbre couvrante de poids minimal.

- Utiliser la fonction **Est_Connexe** lors de la programmation de la fonction **Prim**.
- Afficher la constriction de l'arbre couvrante de poids minimal à chaque itération (étape par étape).
- A la fin de cet algorithme, afficher l'arbre couvrante de poids minimal et le coût minimal donné par l'algorithme de **Prim**.