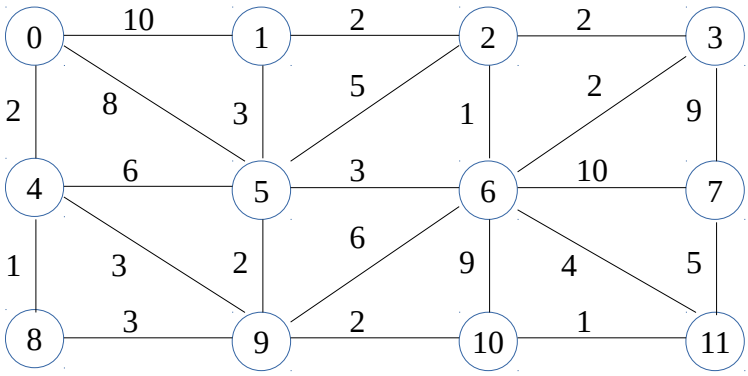
Université de Reims Champagne-Ardenne **Pierre Delisle**
Département de Math., Méca et Informatique
CHPS0742 - Optimisation
2018/2019

Travaux Pratiques 3
Graphes
Représentation – Arbres couvrants – Plus courts chemins - Parcours

Présentation du TP

L'objectif de ce TP est de programmer certains algorithmes de base sur les graphes : arbre couvrant de poids minimum, plus courts chemins et parcours. Pour ce faire, nous utiliserons le graphe suivant comme exemple :



Vous pouvez télécharger le fichier de données correspondant à ce graphe (**graphe1.txt**) sur le site Web du professeur (http://cosy.univ-reims.fr/~pdelisle/enseignement.php).

Sur la première ligne du fichier, on retrouve le nombre de sommets du graphe (dans la suite, les *n* sommets sont identifiés par un entier entre 0 et *n* – 1). Sur les 3 lignes suivantes, on retrouve les caractéristiques du graphe sous forme de booléens. Les valeurs indiquent donc que ce graphe est non orienté, valué et non complet. Dans la section délimitée par « debutDefAretes » et « finDefAretes », chaque arête est représentée par 3 nombres : le sommet d'origine, le sommet d'extrémité et la longueur (poids) de l'arête. La première étape est de représenter ce graphe en mémoire.

Représentation mémoire

On peut représenter ce graphe sous forme d'une matrice d'adjacence (matrice d'entiers). Dans cette dernière, la valeur à la position (*i*, *j*) (et aussi (*j*, *i*) pour les graphes non orientés) correspondra à la longueur de l'arête reliant le sommet *i* et le sommet *j*. Lorsqu'il n'y a pas d'arête entre 2 sommets, on retrouvera la valeur 0.

1) Écrivez une fonction **lireGraphe** permettant de lire les données du fichier **graphe1.txt** et de construire la matrice d'adjacence de ce graphe. Écrivez ensuite une fonction **afficherGraphe** permettant d'afficher cette matrice. Écrivez un main permettant de tester vos fonctions et vérifiez que vous obtenez bien un résultat semblable au suivant :

```
 0  10   0   0   2   8   0   0   0   0   0   0
10   0   2   0   0   3   0   0   0   0   0   0
 0   2   0   2   0   5   1   0   0   0   0   0
 0   0   2   0   0   0   2   9   0   0   0   0
 2   0   0   0   0   6   0   0   1   3   0   0
 8   3   5   0   6   0   3   0   0   2   0   0
 0   0   1   2   0   3   0  10   0   6   9   4
 0   0   0   9   0   0  10   0   0   0   0   5
 0   0   0   0   1   0   0   0   0   3   0   0
 0   0   0   0   3   2   6   0   3   0   2   0
 0   0   0   0   0   0   9   0   0   2   0   1
 0   0   0   0   0   0   4   5   0   0   1   0
```

Arbre couvrant de poids minimum

L'algorithme de Kruskal permet de déterminer un arbre couvrant de poids minimum d'un graphe :

```
Trier les arêtes par poids croissants et les ranger dans tabAretes
POUR i de 1 à n FAIRE
  tabCC[i] ← i
cptArbre ← 0
cptAretes ← 1
TANT QUE cptArbre < n − 1 FAIRE
  {x,y} ← tabAretes[cptAretes]
  cptAretes ← cptAretes + 1
  SI tabCC[x] != tabCC[y] ALORS
    cptArbre ← cptArbre + 1
    tabArbre[cptArbre] ← {x,y}
    indCC ← tabCC[y]
    POUR i de 1 à n FAIRE
      SI tabCC[i] = indCC ALORS
        tabCC[i] ← tabCC[x]
```

Le résultat de l'algorithme de Kruskal est un tableau d'arêtes où chaque arête est représentée par son sommet d'origine et son sommet d'extrémité.

2) Définissez une/des structure(s) de données appropriée(s) pour représenter les arêtes et le tableau d'arêtes définissant l'arbre couvrant de poids minimum.

Une implémentation de l'algorithme de Kruskal devrait comporter les principales étapes suivantes :

- construction d'un tableau de toutes les arêtes du graphe à partir de la matrice d'adjacence ;
- tri du tableau d'arêtes construit ;
- création d'un tableau d'entiers permettant de gérer les composantes connexes : l'indice *i* du tableau correspondra à l'indice de marquage de la composante connexe du sommet *i* ;
- création d'un tableau des arêtes qui seront retenues lors de l'exécution de l'algorithme ;
- construction de l'arbre couvrant de poids minimum (tableau des arêtes retenues) en choisissant les arêtes par ordre croissant de longueur selon l'appartenance ou non de ses sommets à la même composante connexe.

3) Écrivez une fonction **genererAcpmKruskal** permettant de générer un arbre couvrant de poids minimum du graphe d'exemple, puis une fonction **afficherAcpm** permettant d'afficher le tableau d'arêtes résultant. Testez vos fonctions dans le main et vérifiez que vous obtenez bien un résultat semblable au suivant :

```
Arbre couvrant de poids minimum :
2 6
4 8
10 11
1 2
2 3
0 4
5 9
9 10
4 9
5 6
7 11
Longueur de l'arbre : 24
```

Plus court chemin d'un sommet à tous les autres et de tout sommet à tout sommet

L'algorithme de Dijkstra permet de trouver le plus court chemin d'un sommet à tous les autres dans un graphe :

```
arbre ← {r} (r étant la racine de l'arborescence)
pivot ← r
distance(r) ← 0
POUR tout sommet x autre que r FAIRE
  distance(x) ← ∞
POUR i de 1 à n − 1 FAIRE
  POUR tout sommet y non dans arbre et successeur de pivot FAIRE
    SI distance(pivot) + p(pivot, y) < distance(y) ALORS
      distance(y) ← distance(pivot) + p(pivot,y)
      pere(y) ← pivot
  Chercher, parmi les sommets non dans arbre, un sommet y tel que
  distance(y) soit minimum
  pivot ← y
  arbre ← arbre ∪ pivot
```

Le résultat de l'algorithme est un tableau d'entiers représentant, pour chaque sommet, l'indice de son père dans l'arborescence des plus courts chemins, ainsi qu'un tableau d'entiers représentant la distance la plus courte entre le sommet racine et chaque autre sommet dans cette arborescence.

4) Définissez une/des structure(s) de données appropriée(s) pour représenter l'arborescence des plus courts chemins d'un sommet à tous les autres et les distances associées.

Une implémentation de l'algorithme de Dijkstra devrait comporter les principales étapes suivantes :

- construction d'un tableau permettant de déterminer si chaque sommet est couvert ou non dans l'arborescence ;
- construction de(s) structure(s) de données représentant les pères et les distances (initialisées à un majorant) ;
- construction itérative de l'arborescence des plus courts chemins en choisissant, à chaque itération, un sommet pivot le plus proche de l'arbre en construction et en mettant à jour les distances des sommets non encore couverts par rapport à ce pivot.

5) Écrivez une fonction **genererPccDijsktra** permettant de trouver les plus courts chemins du sommet 0 à tous les autres pour le graphe d'exemple, puis une fonction **afficherPcc** permettant d'afficher les données résultantes. Testez vos fonctions dans le main et vérifiez que vous obtenez bien un résultat semblable au suivant :

```
Plus courts chemins du sommet 0 a tout sommet :
Sommets  :   0   1   2   3   4   5   6   7   8   9  10  11
Distance :   0  10  11  12   2   7  10  13   3   5   7   8
Pere     :  -1   0   6   6   0   9   5  11   4   4   9  10
```

6) Généralisez la fonction de la question précédente pour qu'elle détermine le plus court chemin de tout sommet à tout sommet en appliquant l'algorithme de Dijkstra à chaque sommet du graphe.

Parcours

Le parcours en largeur d'un graphe permet de visiter les sommets « par niveaux » à partir d'une racine déterminée :

```
Mettre r dans la liste d'attente L
TANT QUE L n'est pas vide
  Retirer un sommet x
  POUR Tout arc (x, y)
    Traverser l'arc (x, y)
    SI y n'est par marqué ALORS
      Marquer y
      père(y) ← x
      Mettre y dans la liste d'attente
```

Pour effectuer un parcours en largeur, il faut gérer la liste d'attente comme une file : on insère en queue les sommets à mesure qu'on les marque et on les retire en tête pour ensuite parcourir leurs arcs. Pour gérer le marquage des sommets, on peut utiliser un tableau de booléens où une case d'indice *i* correspond au marquage du sommet *i*. Pour chaque sommet inséré dans la liste d'attente, on doit conserver son indice et l'indice de son père.

7) Définissez une/des structure(s) de données appropriée(s) pour effectuer le parcours en largeur, puis écrivez une fonction **parcoursLargeur** qui affiche les indices des sommets en ordre de leur marquage. Testez vos fonctions dans le main et vérifiez que vous obtenez bien un résultat semblable au suivant :

```
Ordre du parcours en largeur :
0 1 4 5 2 8 9 6 3 10 7 11
```

De son côté, le parcours en profondeur (DFS) permet de visiter les sommets en « avançant » sur le premier sommet non marqué du graphe à partir d'un sommet donné, puis de visiter les sommets restants lors du retour en arrière :

```
Marquer r
courant ← r
TANT QUE L'algorithme n'est pas terminé
  x ← courant
  SI Il existe un arc (x, y) non traversé ALORS
    Traverser l'arc (x, y)
    SI y n'est pas marqué ALORS
      Marquer y
      père(y) ← x
      courant ← y
  SINON
    SI x != r ALORS
      courant ← père(x)
    SINON
      L'algorithme est terminé
```

Cette fois-ci, il faudra être en mesure de « revenir en arrière » dans l'arborescence, ce qui suggère la définition, pour chaque sommet, d'un pointeur vers un sommet « père ».

7) Définissez une/des structure(s) de données appropriée(s) pour effectuer le parcours en profondeur, puis écrivez une fonction **parcoursDFS** qui affiche les indices des sommets en ordre de leur marquage. Testez vos fonctions dans le main et vérifiez que vous obtenez bien un résultat semblable au suivant :

```
Ordre du parcours en profondeur :
0 1 2 3 6 5 4 8 9 10 11 7
```

Questions supplémentaires pour les rapides et les motivés

- Écrivez le code permettant de déterminer un arbre couvrant de poids minimum par l'algorithme de Prim.
- Écrivez le code permettant de spécifier la numérotation préfixe ou postfixe lors du parcours en profondeur.
- Écrivez le code permettant de spécifier une opération différente de l'affichage lors des parcours de graphe.
- Écrivez le code permettant de représenter le graphe par un tableau de listes d'adjacence.
- Codez les algorithmes vus dans ce TP en utilisant un tableau de listes d'adjacence.