IMAC
INGÉNIEUR
ESIPE

Algorithmique Avancé

TP 6

Graphes

Beaucoup de problèmes ne peuvent pas être résolu que par des arbres ou des listes. Il arrive souvent qu'un élément ne soit pas uniquement le parent d'un autre mais qu'il soit connecté à plusieurs autres éléments. Un cas basique est les relations d'amitiés dans un réseau social, une personne est connectée à d'autre et ces autres personnes sont eux-même connectés à d'autres. Mais il existe plein d'autre problème comme trouver un itinéraire ou détecter un objet sur une image.

1 Graphe

Un **Graphe** est une structure de données défini par un ensemble de **sommets** S et en ensemble d'**arêtes** (couple de sommets) A.
Exemple de graphe pour $S = \{1, 4, 5, 6, 7, 9, 10\}$,
$A = \{(1, 4), (4, 6), (6, 7), (5, 6), (7, 4), (10, 7), (4, 10), (9, 10), (9, 4)\}$:

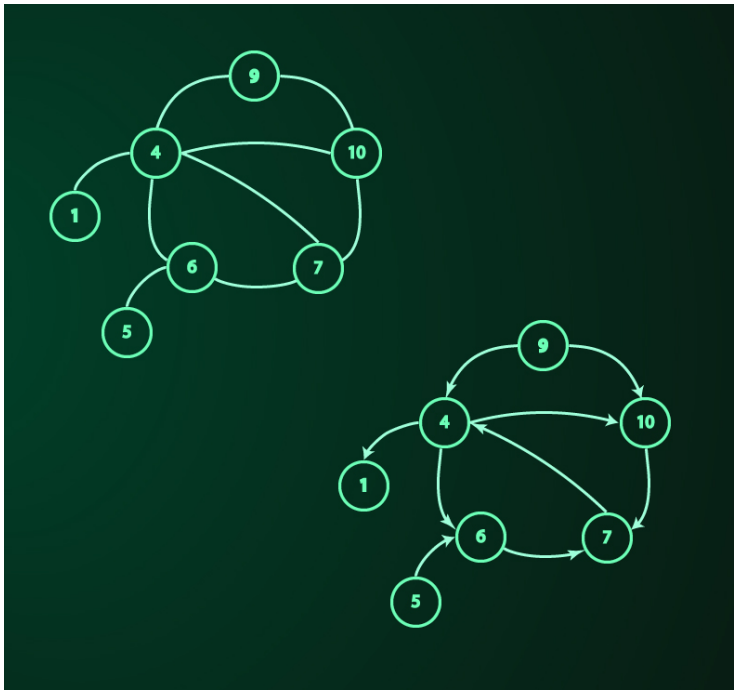

FIGURE 1 – Exemple de Graphe

Il existe deux grands types de graphes, les graphes **non-orientés** et les graphes **orientés** qui possèdent respectivement des arêtes **bidirectionnelles** et **unidirectionnelles**. Une arête unidirectionnelle signifie qu'un sommet pointe vers un autre mais que ce dernier ne pointe pas forcément vers lui. Dans le cas des relations amicales on va considérer un graphe non-orienté, l'ami d'une personne a pour ami cette même personne (la relation va dans les deux sens). Dans un réseau d'écoulement d'eau, une jointure mène l'eau vers une autre mais cette dernière ne peut faire couler l'eau dans le sens inverse.

1.1 Graphe pondéré

En gardant l'exemple d'un réseau d'écoulement d'eau, on peut ajouter des informations en plus sur les arêtes pour par exemple représenter la capacités des tuyaux. Ce genre de graphes qui stockent des valeurs pour chaque arrête sont des graphes pondérés.

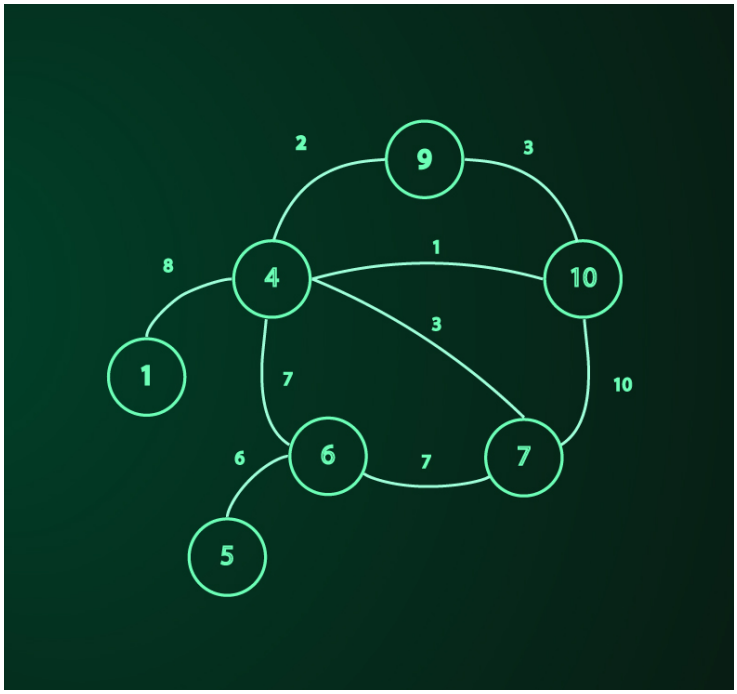

FIGURE 2 – Exemple de Graphe pondéré

1.2 Chemin et Circuit

Un **chemin** est un ensemble d'arête consécutif permettant d'atteindre un sommet en partant d'un autre. Par exemple, dans le graphe précédent, le chemin $(9, 4), (4, 6), (6, 7)$ permet d'atteindre 7 en partant de 9.
Un **circuit** est un chemin permettant d'atteindre un sommet en partant de ce même sommet. Par exemple, dans le graphe précédent, le circuit $(4, 6), (6, 7), (7, 4)$ permet d'atteindre 4 en partant de 4.

1.3 Graphe Connexe et fortement connexe

Un graphe connexe est un graphe dont chaque sommet peut trouver un chemin vers n'importe quel autre sommet du graphe. Le graphe est fortement connexe s'il s'agit d'un graphe orienté. Par exemple le sous-graphe $S = \{4, 6, 7, 10\}$ du graphe est fortement connexe, il existe au moins un chemin entre chaque sommet.

2 Algorithme

2.1 Représentation

Un graphe peut être représenté de deux manières différentes.

2.1.1 Matrice d'adjacence

Une matrice d'adjacence est une matrice carrée dont la taille est égale au nombre de sommets dans le graphe. Le nombre de la ligne i et de la colonne j représente la relation entre le sommet i et le sommet j, s'il est égale à 1, il existe une arête entre ces deux éléments (de i vers j).
Exemple pour le graphe précédent.

$$S = \begin{bmatrix} 1 & 4 & 5 & 6 & 7 & 9 & 10 \end{bmatrix}$$

$$\left[\begin{array}{c|ccccccc} & 1 & 4 & 5 & 6 & 7 & 9 & 10 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 7 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} \\ 10 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \end{array}\right]$$

Note : Un graphe non-orienté aura pour matrice d'adjacence une matrice symétrique.

Note : Un graphe pondéré peut être représenté par une matrice d'adjacence dont chaque valeur représente la valeur de l'arête entre i et j.

2.1.2 Liste d'adjacence

On peut aussi représenter un graphe stockant les arrêtes dans une liste chainée pour chaque sommet.

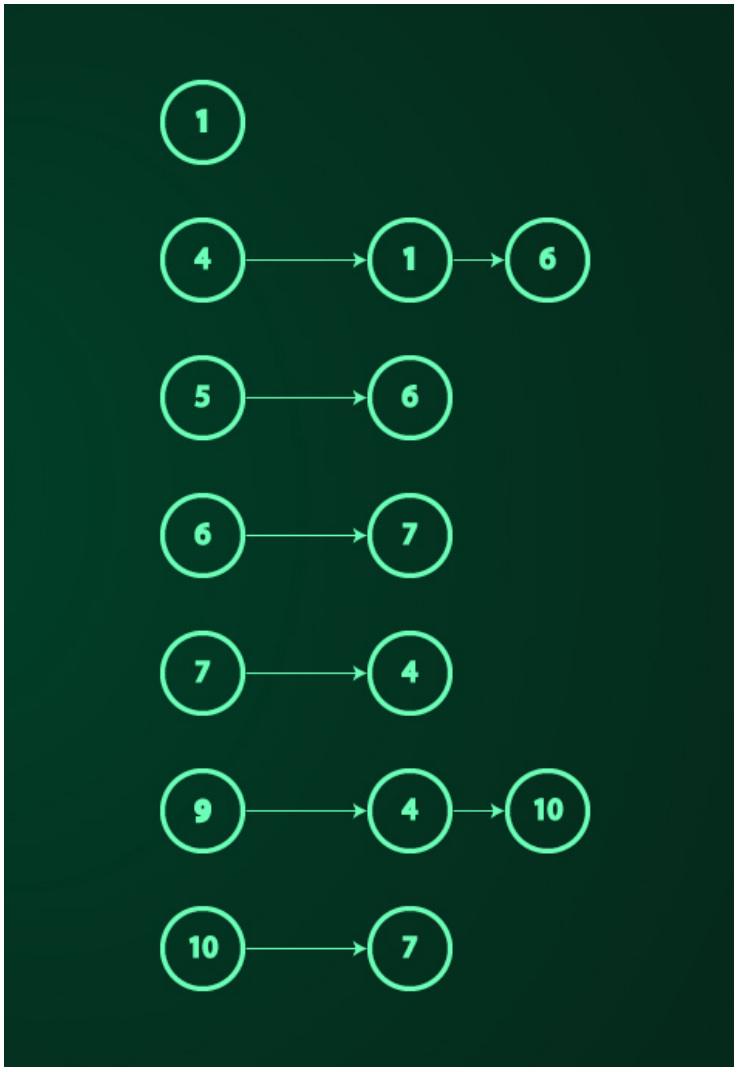

FIGURE 3 – Représentation d'un graphe par listes

2.2 Parcours en profondeur

Le parcours d'un graphe peut poser un problème lorsqu'on arrive dans un circuit. En suivant un circuit, on risque d'emprunter toujours les mêmes arrêtes. Pour éviter ce problème, il faut définir un tableau listant l'ensemble des sommets parcourus.
Le parcours en profondeur se base sur un appel récursif du parcours sur les différents sommets qu'on rencontre.

```
void deep_travel(Vertex v, bool visited[])
{
  print(v);
  visited[v.index]=true;
  for (Edge e=v.first_edge; e != NULL; e=e.next)
  {
    if (!visited[e.second_vertex.index])
      deep_travel(e.second_vertex, visited);
  }
}
```

2.3 Parcours en largeur

Le parcours en largeur est semblable au parcours en profondeur, mais évite la récursion en utilisant une File.

```
void wide_travel(Vertex vertices[])
{
  Queue q;
  bool visited[vertices.size()];
  \\ init all visited to false
  q.push(vertices[0]);
  while(!q.empty())
  {
    Vertex v = q.pop();
    print(v);
    visited[v.index] = true;
    for (Edge e=v.first_edge; e != NULL; e=e.next)
    {
      if (!visited[e.second_vertex.index])
        q.push(e.second_vertex);
    }
  }
}
```

3 TP

Implémenter les fonctions d'un graphe en utilisant la représentation par listes.

- **buildFromAdjenciesMatrix**(int** $adjencies$, int $nodeCount$) : Construit un graphe à partir d'une matrice d'adjacence. La valeur d'un sommet représente son indice dans le tableau. $adjencies$ est tableau 2D de taille $nodeCount$.
- **deepTravel**(GraphNode* $first$, GraphNode* $nodes[\,]$, int& $nodesSize$, bool $visited[\,]$) : rempli le tableau $nodes$ en parcourant le graphe en profondeur à partir de $first$. $nodeSize$ est le nombre de noeud dans $nodes$ et est donc égale à 0 lors du premier appel de fonction. $visited$ est un tableau de booléen rempli de $false$ lors du premier appel.
- **wideTravel**(GraphNode* $first$, GraphNode* $nodes[\,]$, int& $nodesSize$, bool $visited[\,]$) : rempli le tableau $nodes$ en parcourant le graphe en largeur à partir de $first$. $nodeSize$ est le nombre de noeud dans $nodes$ et est donc égale à 0 lors du premier appel de fonction. $visited$ est un tableau de booléen rempli de $false$ lors du premier appel.
- **detectCycle**(GraphNode* $first$, bool $visited[\,]$) : Retourne Vrai si le graphe possède un circuit commençant par $first$.

3.1 C++

Le dossier *Algorithme_TP6/TP* contient un dossier *C++*. Vous trouverez dans ce dossier des fichiers *exo<i>.pro* à ouvrir avec *QtCreator*, chacun de ces fichiers projets sont associés à un fichier *exo<i>.cpp* à compléter pour implémenter les différentes fonctions ci-dessus.
L'exercice *exo1.cpp* implémente une structure *Graph* définissant un tableau de *GraphNode* (de sommets). Chaque *GraphNode* possède une liste chainée de *Edge* reliant un *GraphNode* source et un *GraphNode* destination. Ces trois classes permettent de décrire un graphe.

```
struct Edge
{
  Edge(GraphNode* source, GraphNode* destination, int distance=0);

  GraphNode* source;
  GraphNode* destination;

  int distance;

  Edge* next; // allow chaining
};

struct GraphNode
{
  GraphNode(int value);

  void appendNewEdge(GraphNode* destination, int distance=0); // add a new edge
       to destination

  int value;
  Edge* edges;
};

class Graph
{
public:
  Graph(int size=20); // size = allocated array size

  int nodesCount(); // real number of nodes
  void appendNewNode(GraphNode* node); // add a new node in the nodes array

  GraphNode& operator[](int index); // get the GraphNode& from the given index
};
```

Notes :

- Dans une fonction C_{++}, si le type d'un paramètre est accompagné d'un '&' alors il s'agit d'un paramètre entré/sortie. La modification du paramètre se répercute en dehors de la fonction.
- Lorsque vous faites appel à $this$ dans une méthode d'une structure (ou d'une classe), vous pouvez accéder aux attributs de la structure en question, comme dans l'exemple ci-dessus.