

INF2010 – Structures de données et algorithmes

Automne 2021 Travail Pratique 5

Graphes



Objectifs du laboratoire :

1. Comprendre le fonctionnement d'un graphe dirigé
2. Comprendre le fonctionnement de l'algorithme de Dijkstra
3. Utiliser et appliquer les connaissances de graphe acquises afin de résoudre un problème complexe

Ce laboratoire aura pour objectif de vous familiariser avec **les graphes** et la matière du **cours 8**. Le travail à effectuer sera identifier dans le code avec les « TODO ». Les « TODO » peuvent indiquer deux situations :

1. Compléter du code existant
2. Faire l'implémentation de la méthode entière

Le laboratoire sera évalué à l'aide de tests ainsi qu'un survol du code par le chargé. Veuillez-vous référer à la section « barème de correction » pour de l'information additionnel sur la correction.

Format de la remise :

Remettre dans un dossier nommé : *tp5_MatriculeX_MatriculeY.zip* avec tous les fichiers du laboratoire à remettre.

Il est seulement nécessaire de remettre les fichiers avec du code modifier

Important : Les travaux mal nommés ou avec des fichiers supplémentaires auront une pénalité de 20% sur le travail. Les travaux en retard seront pénalisés de 20% par jour de retard

Partie 1 : Graphe Dirigé

Cette partie du travail pratique portera sur la matière du Cours 8.

Voici un lien qui pourrait vous être utile pour visualiser pendant que vous complétez le travail pratique : https://csacademy.com/app/graph_editor/

Dans les classes suivantes, apportez les modifications nécessaires aux méthodes contenues dans l'énoncé :

Vertex :

compareTo :

Vertex surcharge la méthode compareTo de la classe Comparable. Vous devez compléter l'implémentation qui compare le coût (cost) de deux connexions à un sommet. Soit le coût présent au sommet et le nouveau coût en paramètre.

UndirectedGraph :

connect :

Ajouter et justifier les conditions de return dans connect. Utilisez les notes de cours si nécessaire pour l'implémentation et ensuite justifier la raison pour laquelle leur présence est nécessaire.

DirectedGraphWeighted :

initialize :

En vous référant à l'implémentation de DirectedGraph, ajouter le code nécessaire dans initialize pour que :

- neighbours : l'initialiser et allouer la mémoire
- nodeQuantity représente le nombre de sommet dans le graphe
- graphConnections soit initialisé

connect :

Ajouter les conditions nécessaires pour la connexion de nœud. Connecter les nœuds avec le hashmap **neighbours**.

toString :

Implémenter le toString pour qu'il affiche chaque arc entre les sommets ainsi que le coût associé pour l'arc.

adj :

Retourner un hashmap des éléments adjacents au sommet.

Partie 2 : Algorithme Dijkstra

Voici une visualisation de l'algorithme en action :

<https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html>

Dans la classe *Heap* :

DecreaseKey :

Diminuer le cout du sommet qui est passé en paramètre avec le nouveau cout.

findSmallestUnknown:

Trouver le sommet avec le cout le moins élevé qui n'est pas encore connu (known) et le retourner.

Dans la classe *DirectedGraphWeighted* :

findLowestCost :

Premier TODO :

Ajout des sommets (vertex en anglais) dans le Heap.

Deuxième TODO :

Évaluer chaque arc adjacent à un sommet afin de trouver si le coût est inférieur à celui qui se retrouve dans le HashMap nodes.

Si le coût est inférieur :

Changer le path dans **w** et modifier le coût du sommet avec decreaseKey.

Troisième TODO :

Ajouter le coût total du graphe en itérant sur le Heap.

Partie 3 : Analyse de l'implémentation de findLowestCost

1. Quel sera le nombre d'itération maximale et minimale pour la boucle suivante que vous avez implémenté dans le deuxième TODO:

```
while(true){
    Vertex v = vertices.findSmallestUnknown();
    if(v == null) break;
    v.known = true;
    for(Vertex w: adj(v.index)){
        /* TODO Evaluate each edge to see if the total cost is less than the cost contained in nodes. */
        /* TODO Decrease the cost of the vertex in the Heap using decreaseKey if conditions are met */
    }
}

/* TODO Add up the total cost of the elements in the Heap */
```

2. Dans le pire cas, quel sera le nombre de modification du coût pour **un sommet**? Le pire cas étant le cas qui cause le nombre de modifications au cout le plus élevé.

```
while(true){
    Vertex v = vertices.findSmallestUnknown();
    if(v == null) break;
    v.known = true;
    for(Vertex w: adj(v.index)){
        /* TODO Evaluate each edge to see if the total cost is less than the cost contained in nodes. */
        /* TODO Decrease the cost of the vertex in the Heap using decreaseKey if conditions are met */
    }
}

/* TODO Add up the total cost of the elements in the Heap */
```

3. Quel sera le nombre d'itération pour la boucle que vous avez implémenté dans le troisième TODO selon le nombre de sommet suivant:
- a. 10 sommets
 - b. 100 sommets
 - c. 1000 sommets

```
public int findLowestCost() {
    /* NE PAS MODIFIER CE CODE */
    int totalCost = 0;

    Heap vertices = new Heap( maxsize: nodeCapacity + 1);
    /* TODO Add all of the vertices to the Heap start at Index 1. The default cost should be the largest possible value */
    /* NE PAS MODIFIER CE CODE */

    while(true){
        Vertex v = vertices.findSmallestUnknown();
        if(v == null) break;
        v.known = true;
        for(Vertex w: adj(v.index)){
            /* TODO Evaluate each edge to see if the total cost is less than the cost contained in nodes. */
            /* TODO Decrease the cost of the vertex in the Heap using decreaseKey if conditions are met */
        }
    }

    /*TODO Add up the total cost of the elements in the Heap */

    return totalCost;
}
```

Partie 4 : Question Entrevue

Le coût minimum pour créer un chemin dans une matrice

On vous partage une matrice avec les dimensions $m \times n$. Chaque case à une direction associé :



Exemple d'une case avec une direction

Il y a quatre directions possibles :

- Droite – valeur numérique : 1
- Gauche – valeur numérique : 2
- Vers le bas – valeur numérique : 3
- Vers le haut – valeur numérique : 4

Vous allez recevoir en paramètre une matrice (*grid*) et dans chaque case, il y aura une valeur numérique de [1,4] pour indiquer la direction à prendre.

Par exemple :

Si on se déplace vers la droite, de $grid[i][j]$ à $grid[i][j+1]$

Si on se déplace vers le haut, de $grid[i][j]$ à $grid[i-1][j]$

Le problème :

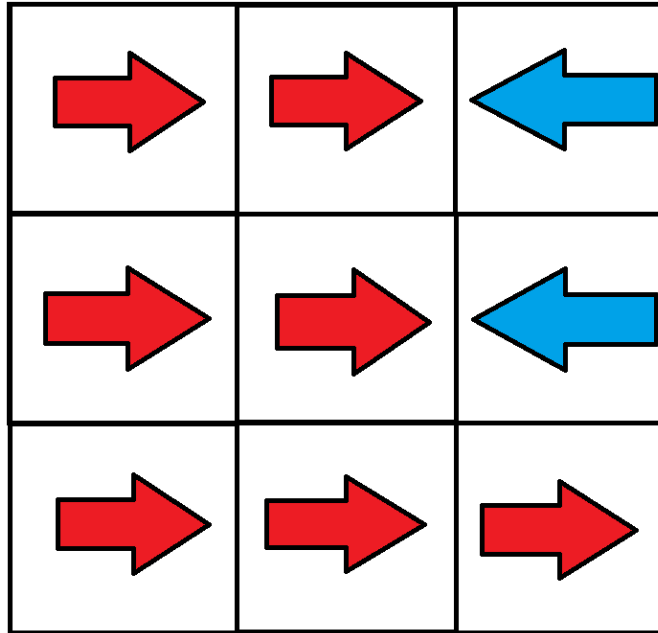
Vous commencez à la case (0,0). On veut trouver un chemin qui commence à (0,0) et qui se termine à ($m - 1$, $n - 1$) en suivant les directions des cases dans la matrice.

Il est possible de modifier la direction d'une case de matrice pour un cout de 1. Une case peut seulement être **modifiée une fois**.

Retournez le coût minimum pour créer un chemin valide dans la matrice.

Exemples :

1.



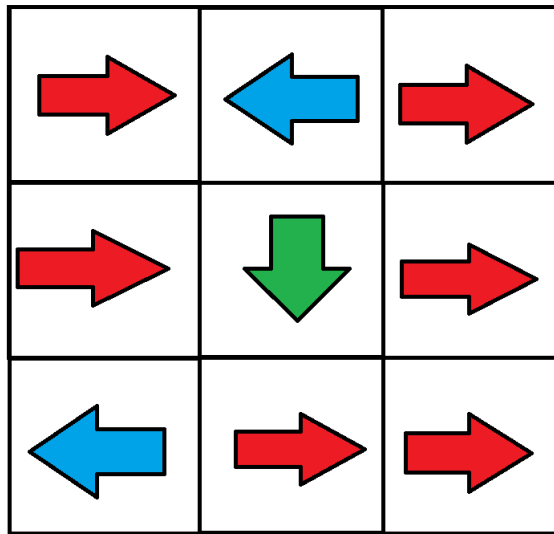
La matrice sera dans l'état suivante :

grid = [[1,1,2], [1,1,2], [1,1,1]]

La valeur de retour sera 2.

Nous avons modifié à deux reprises la direction afin de se rendre à la fin de la matrice.

2.











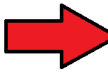
La matrice sera dans l'état suivante :

grid = [[1,2,1], [1,3,1], [2,1,1]]

La valeur de retour sera 1.

Nous avons modifié seulement une fois la direction afin de se rendre à la fin de la matrice.

3.

La matrice sera dans l'état suivante :

grid = [[1,1,3], [3,2,2], [1,1,1]]

La valeur de retour sera 0.

La direction n'a pas été modifié pour se rendre à la fin de la matrice.

Date de remise :

Groupe B1: 7 décembre à 23h55

Groupe B2: 7 décembre à 23h55

Barème de correction :

Partie 1 : 10 points

Tests partie 1 - 2 + correction manuelle : / 8

Qualité du code : /2

Partie 2 : 3 points

Correction manuelle : /3

Partie 3 : 7 Points

Tests partie 3 : /6

Qualité du code : /1

Note : - /20