HAYE ARTHUR

# PROJET – Algorithme de Floyd-Warshall

Et dans ce cas, quel documents ou informations suis-je en mesure de communiquer à l'entreprise ?

Dans le cas contraire, que pourrais-je communiquer par rapport à l'Université de Poitiers.

cuments ou informations suis-je en mesure de communier à l'entreprise ?

Sommaire :



## PRESENTATION :

L'algorithme de FloydWarshall

Prend en entrée un graphe orienté et valué, sous la forme d'une matrice d'adjacence donnant le poids d'un arc lorsqu'il existe et la valeur ∞ sinon. Le poids d'un chemin entre deux sommets est la somme des poids sur les arcs constituant ce chemin. Les arcs du graphe peuvent avoir des poids négatifs, mais le graphe ne doit pas posséder de cycle de poids strictement négatif. L'algorithme calcule, pour chaque paire de sommets, le poids minimal parmi tous les chemins entre ces deux sommets.

Complexité : Θ(n3).

## ENSEMBLE DES CLASSES :

Le Fichier Data présent dans Projet Math contient le 'graph-001.alists' initial, et la matrice des couts initial renommée 'data.data' . Les Classes non implémentées de ma part présente : 'fr_departments'.

### -Class Main :

La classe est l'exécutable de notre programme, pas d'attributs, juste une méthode

**public static void main(String[] args**)

et une exception pour crée les fichiers des matrices “java.io.IOException e2”.

Un graph initial sera initialisé dans le Main puis transférer à la Class Floyd-Warshall pour y être utilisé.

## -Class Dialogue :

La classe contient les différents commentaires durant le programme, et permet

*L’affichage des noms des départements choisi au début, à l’aide des classes du package ‘f**r_departments**’ ,

*notamment la condition d’arrêt du programme :

**public static Boolean stop()throws FileNotFoundException**

* et un affichage sommaire des matrices d’initialisation :

**public static void choose(GraphLinearDirected g)throws FileNotFoundException**

## -Class Floyd_Warshall :

La Classe contient l’algorithme ainsi que diverses méthodes spécifiques à l’algorithme.

Elle possède deux attributs et un constructeur :

**private int [][]matrice_chemin ;**

**private int [][]matrice_adjacence ;**

**public Floyd_Warshall (GraphLinearDirected g , int Dpt_debut , int Dpt_arriver ) throws FileNotFoundException**

Voici quelques méthodes présente dans la classe essentielle à notre programme :

**public int getPoid( int source ,int target , int [][] cout)**

Qui récupère depuis la matrice des “couts des arcs” un poids.

**public int[] remplie_tab_chemin ( int [] chemin , int Dpt_debut ,GraphLinearDirected g, int[][] cout, List<Integer> check )**

Qui remplira ligne par ligne, la matrice d’adjacence.

**public int [][] init_chemin ( int [][] chemin , int ordre, int [][] cout, GraphLinearDirected g )**

Qui initialise la “matrice chemin” des plus court chemins et la “matrice adjacence”.

La méthode suivante est le cœur de notre classe, notre algorithme :

**public int [][] algo ( int [][] chemin , int [][] matrice_adj )**

## -Class Matrice:

Cette classe regroupes des fonctions coder durant les différents TP et que j’ai jugé possiblement utile lors du Projet.

**public static byte[][] getAdjacencyMatrix(Scanner input , int n)**

**public static int[][] getAdjacencyList(Scanner input , int n)**

**public static int[] computeAdjacencyList( byte[][] m , int x)**

**public static byte[][] computeAdjacencyMatrice( int[][] list )**

Et

**public static int [][] Set_arete_dpt() throws FileNotFoundException**

Qui convertis le fichier du cout des département en matrice utilisable.

Avec différentes fonctions d'affichage :

**public static void affiiche(byte[][] m)**

**public static void affiiche2(int[][] m)**

**public static void tab(int [] m)**

### -Class RecupGraph :

Comme son nom l'indique, la classe sert essentiellement à convertir les matrices en fichier Texte d'extension "amatrix".

Elle possède un constructeur et une seule méthode.

**public RecupGraph(int [][] matrice_adjacence, int [][] matrice_chemin) throws java.io.IOException**

**public  void Retranscrit_info( int [][] matrice_adjacence , File file_matrice_adj )throws IOException**

Les autres Classes myException et GraphLinearDirected qui sont liée, ne seront pas détaillé comme demander, et font évidemment partie du programme.

## MODE D'EMPLOIE ET EXECUTION :

Le programme utilise un version Java non récente.

Le Fichier contient de prime abord, le fichier source nommé "src" que vous avez trouvé au côté de ce rapport.

Pour exécuter le programme,

Vous pouvez ouvrir le fichier "src",

Une fois dedans vous avez l'ensemble des différents packages/fichiers,

Pour compiler le programme, il vous suffit de vous placer dans le fichier "src" puis de lancer la commande suivante :

- **javac Main.java**

Vous pouvez ensuite le lancer avec la commande suivante :

- **java Main**

Les fichiers graph-011.paths et graph-001.costs contiendront les matrices, en format amatrix, demandées, une fois seulement que vous mettrez fin au programme, c'est à dire que presserez la touche zéro '0', quand le programme vous le proposera.

Le fichier Projet_Math, contient l'ensemble des classes coder et décrites précédemment,

-Dans ce fichier, vous trouverez le fichier Data, qui contient les matrices d'initialisation données.

Le fichier fr_departments contient l'ensemble des classes donné sur les départements.

Une fois le programme lancer, vous aurez la possibilité de visionner les matrices d'initialisations,

En tapant les mots clés suivant :

**DATA_ARC_POID_DPT_FR** : Donne la matrice des couts entre chaque département.

**LISTE_ADJACENCE** : Donne la liste d'adjacence, les liens qu'ils existent entre chaque vertex.

**MATRICE_ADJACENCE** : Donne la matrice en bit des liens qu'ils existent entre chaque vertex.

Si vous souhaitez passer directement à l'algorithme taper :

**NON :** non en majuscule.

Une fois dans l'algorithme vous devrez entrer un **numero** département de départ puis un autre d'arriver.

Le programme vous affiche le chemin le plus courts qu'il existe entre les deux, puis affiche également son cout.

Enfin il vous sera demander, au choix, de continuer vers une saisie différente en rentrant le mot clé :

**NON :** non en majuscule.

Ou bien de mettre fin au programme en rentrant le mot clé suivant :

**0 :** zero.

## COMPTE-RENDU DE L'EXERCICE :

Question a) On note les sommets de la matrice **Δ**, k ∈ {1, 2, 3, …, k} en fonction des coordonnées i et j de la matrice :

**Δk(i,j) = min(Δk-1(i,j) , Δ k-1(i,k) , Δ k-1(k,j)**

**Fonction FloydWarshall (G)**

**Δ0 = matrice d'adjacence de G (matrice n x n)**

**For( k = 1 to n ){**

**For ( i = 1 to n){**

**For ( j = 1 to n){**

**Δk(i,j) = min(Δk-1(i,j) , Δ k-1(i,k) , Δ k-1(k,j)**

**}}}**

**Renvoyer Δn .**

Question b)

On traite une matrice[n][n], n fois, donc ça Complexité est de : Θ(n3). Cela se retrouve dans notre algorithme avec les 3 boucles for.

Question c)

L'Algorithme de Bellman-Ford serait plus pertinent a utilisé dans ce cadre-là, puisqu'il est adapté à cette perspective.

Question d)

Nous avons étudié l'algorithme pour trouver le chemin le plus court chemin d'un sommet a un autre, pour que l'algorithme puisse déterminer le chemin le plus court qu'un sommet possède par rapport à un autre, en étudiant les autres chemins, il faut établir une matrice des chemins.

Qui utilisera des prédécesseurs, c'est à dire en s'aidant des autres chemins existant afin d'établir le chemin le plus optimal, de cette façon en chaque chemin est optimisé par l'intermédiaire des autres chemins déjà trouvé.

Ainsi si un chemin plus court est trouvé de par cet algorithme :

**Δk(i,j) = min(Δk-1(i,j) , Δ k-1(i,k) , Δ k-1(k,j)**

On l'affecte alors dans notre matrice des chemins, comme un chemin existant pour le sommet, là où il n'avait pas de lien auparavant,

De cette façons le chemin vers ce sommet destination, est le chemin trouvé par un autre sommets indépendant mais connecté à notre sommet initial.

Nous ajouterons alors à note algorithme l'opérations suivante :

**matrice_des_chemins[i][j] = matrice_des_chemins[k][j]**