Eyer Nathan - Bena Hugo

SAE 2.2 - EXPLORATION ALGORITHMIQUE

Recherche de plus court chemin dans un graphe

SOMMAIRE:

REPRÉSENTATION D'UN GRAPHE:

- Cette première partie du projet constitue le pilier du projet, les fondations du code qui nous ont par la suite permis de construire et étudier des graphes orientés. Il était donc primordial de correctement la réaliser sans oublier les tests adéquats. Le programme est constitué de 3 classes et 1 interface, et nous allons détailler le fonctionnement de chacuns.

- Nous avons commencé par rédiger la classe Arc qui représente les arcs existants entre les nœuds du graphe. Cette classe posséde 2 attributs: *dest* et *cout* qui représente respectivement le nœud de destination et le coût de l'arc créé, ceux-ci sont initialisés dans le constructeur. La classe posséde notamment des *getter* pour les deux attributs et un *toString* pour faciliter l'affichage des objets Arc. Pour vérifier le fonctionnement de la classe Arc, nous avons créé une classe de tests pour la valider.

- Nous avons par la suite réalisé la classe Arcs qui gère simplement une liste d'*Arc* en attributs initialisée par un constructeur. La classe possède notamment une méthode *ajouterArc*, un *getter* et un *toString* pour faciliter tous les appels et l'affichage.

- L'étape suivante consistait en l'écriture de l'interface Graphe, un type abstrait de données qui contient uniquement l'entête de deux méthodes: *listeNoeud* et *suivants* qui sont définis dans GrapheListe.

- GrapheListe implémente donc l'interface Graphe pour représenter les données associées à un graphe. Cette classe posséde deux attributs: *noeuds* représentant une liste de String correspondant aux nœuds et *adjacence* représentant une liste d'Arcs qui sont les deux initialisés dans le constructeur de la classe. Nous avons notamment dû remplir la liste *adjacence* dans le constructeur avec des objets Arcs vides à l'aide d'une boucle pour éviter les problèmes de liste <nulle>. La classe possède de nombreuses méthodes: elle implémente déjà forcément *listeNoeuds* ainsi que *suivants* mais intègre aussi une méthode *getIndice* qui retourne l'indice du noeud donné en paramètre. GrapheListe possède notamment une méthode ajouterArc comme la classe Arcs où nous devions vérifier l'existence du nœud de départ et de destination avant d'ajouter un nouvel Arc. Pour finir, comme dans les autres classes, nous avons écrit un toString pour avoir un affichage concis notamment en appelant les toString des autres classes. GrapheListe étant la classe la plus longue à écrire, nous avons évidemment écrit des tests pour chaque méthode pour la valider.

CALCUL PAR POINT FIXE:

Question 8 : Ecriture de l'algorithme du point fixe

→ Avant l'écriture de la fonction du point fixe, nous avons avant tout dû conceptualiser celle-ci à l'aide d'un algorithme. Cela nous a permis de programmer plus efficacement la méthode résoudre. Cette fonction modifie les valeurs associées aux nœuds du graphe pour trouver le chemin le plus court en partant du nœud de départ passé en paramètre. Une fois cela fait, à l'aide d'une classe Valeur fournie, nous avons pu commencer l'écriture de la méthode du point fixe dans une classe BellmanFord. Celle-ci prend un graphe et un objet String en paramètre et retourne un objet Valeur contenant les distances et les parents de chaque nœud. Puis nous avons testé cet algorithme dans une classe Main qui affiche pour chaque nœud leur valeur de distance. Et vient

enfin la création d'une classe test: TestBellmanFord pour la validation de cet algorithme.

```
Fonction pointFixe(graphe g, Noeud départ)
    Pour chaque nœud dans le graphe faire
        N.valeur ← +∞
        N.parent ← null
    Fin Pour
    départ.valeur ← 0.0
    modif ← vrai

    Tant que modif est vrai faire
        modif ← faux

        Pour chaque nœud dans le graphe faire
                Pour chaque noeud Y à X parent faire
                        val ← val(X) + coût de l'arc entre X et Y
                        Si val < val(Y) alors
                                val(Y) ← val
                                Parent de Y ← X
                                modif ← vrai
                        Fin Si
                        Si val < minimum alors
                                minimum ← val
                                parent ← Y
                        Fin Si
                Fin Pour

                val ← min
                X.parent ← parent
        Fin Pour
    Fin Tant que

    Retourner minimum
Fin


LEXIQUE

Graphe orienté (g) : (GrapheListe) représente un ensemble de noeuds connectés par des arcs orientés, où chaque arc a un cout associé.

Départ (départ) : (Noeud) départ à partir duquel on souhaite trouver le chemin le plus court

Valeur (valeur) : Valeur associée à chaque nœud dans la fonction, représentant la cout le plus petit trouvé jusqu'à ce noeud.

parent (parent) : (Noeud) parent associé à chaque noeud dans la fonction, représentant le noeud précédent sur le chemin le plus court pour atteindre ce même noeud.

Min (minimum) : (int) Valeur minimale utilisée pour trouver la valeur minimale parmi les valeurs des noeuds.

Parent (parent) : (Noeud) Noeud parent temporaire utilisé pour stocker le noeud associé à la valeur minimale pendant la recherche du minimum.
```

CALCUL PAR DIJKSTRA:

Pour la méthode de Dijkstra, cette fois-ci l'algorithme nous était fourni, notre mission était alors de le retranscrire en java. Dans une classe Dijkstra, nous avons donc écrit la méthode resoudre provenant de l'interface Algorithme. Tout comme l'algorithme de BellmanFord, L'algorithme de Dijkstra trouve le chemin minimal. Cependant ici il consiste en la sélection du nœud à chaque étape avec la plus petite valeur actuelle, et en mettant à jour les valeurs de ses nœuds adjacents si une valeur plus courte est trouvée. Ce processus se répète jusqu'à ce que tous les nœuds soient traités, assurant ainsi que les valeurs finales représentent les chemins les plus courts depuis le nœud de départ.

Ensuite, nous avons testé cet algorithme dans une classe `MainDijkstra` qui affiche pour chaque nœud leur valeur de distance et les nœuds du trajet parcouru. Enfin, nous avons créé une classe de test TestDijkstra pour la validation de cet algorithme.

VALIDATION / EXPÉRIMENTATION:

Question 16: Présentation des résultats
→ Pour expérimenter la comparaison entre les deux algorithmes. Nous avons choisi de tester la rapidité des deux algorithmes à l'aide des fichiers graphes.txt fournis.

Nous avons créé la classe LectureFichier qui s'occupe de récupérer les valeurs des lignes de chaque fichier pour rendre le traitement plus simple et comprehensible. Au départ, étant donné qu'il n'y avait que des nombres dans les fichiers, nous pensions qu'il fallait retranscrire les deux premiers nombres de chaque ligne en lettre alphabétique pour les utiliser en tant que nœuds du graphe, ce qui n'était pas le cas. La classe s'occupe donc simplement de renvoyer un tableau à double entrée avec les donnés du fichier donné en paramètre du constructeur.

Dans la classe Comparaison nous avons choisi de mesurer la rapidité de chaque algorithme sur tous les fichiers donnés, et sur 5 fichiers au hasard dont voici les résultats:

Test sur tous les graphes:

Temps moyen d'exécution pour chaque graphe avec la méthode BellmanFord: 0.014506901063829788 secondes
Temps total: 1,364 secondes
Temps moyen d'exécution pour chaque graphe avec la méthode Dijkstra: 0.025734058510638296 secondes
Temps total: 2,419 secondes

Test: sur 5 graphes:

Temps moyen d'exécution pour chaque graphe avec la méthode BellmanFord: 1.968E-4 secondes
Temps total: 0,000984 secondes
Temps moyen d'exécution pour chaque graphe avec la méthode Dijkstra: 1.2242E-4 secondes
Temps total: 0,000612 secondes.

Question 17: Analyse des résultats
→ On constate que l'algorithme de Dijkstra est clairement moins rapide que BellmanFord sur tous les graphes mais est légèrement plus rapide sur un nombre de graphes très réduit. Expérimentalement, la méthode du point fixe est donc plus efficace d'après ces résultats étant donné qu'elle permet de trouver le plus court chemin entre un sommet source et tous les autres sommets du graphe en une seule fois, ce qui peut être plus rapide que de chercher le plus court chemin entre deux sommets à la fois avec d'autres algorithmes. D'autres part, la méthode de Dijkstra possède une complexité supérieure à celle de BellmanFord ce qui explique un temps de calcul plus long. Dans ce cas précis, plus le graphe sera grand et plus la méthode de Dijsktra

prendra de temps à s'exécuter en appelant les différentes méthodes des autres classes.

Question 18: Point de vue personnelle
→ D'après nous, c'est l'algorithme de Dijkstra qui devrait être le plus efficace car il évite d'explorer des sommets qui ne font pas partie du plus court chemin, ce qui permet de considérablement diminuer la charge de travail. Il demande beaucoup moins de calcul au format papier lorsque le graphe est dessiné. Cela ne veut pas dire qu'il est plus efficace dans un programme informatique étant donné que nous réfléchissons différemment d'un ordinateur.
Les résultats ne jouent pas en notre faveur mais nous trouvons que Dijsktra est plus simple à mettre en place et à exploiter. Toutefois l'algorithme de BellmanFord pouvant être appliqué sur des graphes orientées avec des arcs au coût négatif, nous pensons que celui-ci est exploitable dans plus de cas.

CONCLUSION:

Cette saé et ce long programme Java composé dans notre cas de 17 classes, nous ont permis non seulement de progresser en programmation objet notamment sur l'utilisation d'interfaces, mais aussi de réfléchir à comment traduire des fonctionnements mathématiques en langage machine.
Dans ce projet, nous avons implémenté deux algorithmes de recherche de plus court chemin dans un graphe : l'algorithme du point fixe (ou de Bellman-Ford) et l'algorithme de Dijkstra.
Nous avons donc constaté que l'efficacité d'un algorithme de recherche de chemin dans un graphe dépend de la taille du graphe et de la manière dont ce chemin est recherché.