Baribeaud Arthur
BDIA3
N° NAQ503552

Rapport TP noté

1. L'objectif de la première question était d'instancier les classes nécessaires à la création d'un graphe ainsi que de permettre à l'utilisateur de renseigner à la main, les arcs et sommets le composant.
   Pour ce faire, j'ai implémenté 3 classes : graphe, sommet et arc.

   Graphe : Il s'agit de la classe principale qui représente un ensemble d'objets (sommet) dans lequel certains de ces objets ont des relations (arc).
   Pour ce faire, la classe graphe possède deux champs : une liste de sommets et une liste d'arcs.

   En plus des champs renseignés dans le constructeur, on rajoutera une méthode *saisieUser()* dans notre classe afin de permettre à notre utilisateur de saisir sur l'entrée standard du clavier le nombre et le nom des arcs et des sommets afin de les enregistrer dans les champs de type liste : sommet[] et arc[].

   Sommet : La classe sommet représente ici l'un des objets de notre graphe, et possède deux champs : nom et degré
   Le champ nom est utilisé comme identifiant de notre sommet et est par conséquent unique. Le champ degré, quant à lui, représente la somme des degrés entrant et sortant de notre sommet.

   On rajoute ensuite la méthode *saisie()* dans notre classe afin que l'utilisateur puisse saisir sur l'entrée standard le nom du sommet. C'est d'ailleurs cette méthode qui est appelé dans la méthode *saisieUser()* de la classe graphe afin de créer des instances de notre objet sommet.

   Arc : Enfin, la classe arc représente ici les relations entre les objets de notre graphe, et possède quatre champs : un nom, un prédécesseur, un successeur et un poids.
   Le champ nom sert comme pour la classe sommet d'identifiant unique pour un arc. Les champs prédécesseur et successeur comme leur nom l'indique servent à définir les relations entre deux sommets : le prédécesseur représentant le sommet initial et le successeur le sommet final. Enfin, puisque nous travaillons avec des graphes pondérés, le dernier champ représente le poids entre le successeur et le prédécesseur

   On rajoute également la méthode *saisie()* dans notre classe afin que l'utilisateur puisse saisir sur l'entrée standard le nom de l'arc, le nom (l'identifiant) du sommet initial (prédécesseur) et final (successeur). De la même manière que dans la classe sommet, la méthode *saisie()* est notamment utilisée dans la méthode *saisieUser()* de la classe Graphe.

2. La deuxième question consiste à saisir un graphe mais, à l'inverse de la première question, la saisie du graphe doit être automatique et le choix des sommets et arcs est prédéfini dans un fichier txt.

Baribeaud Arthur
BDIA3
N° NAQ503552

Pour ce faire, on implémente la fonction constructFromFile() comme méthode de la classe Graphe.
Cette méthode va dans un premier temps ouvrir en lecture le fichier txt qui contient le schéma du graphe à construire. Une fois le fichier ouvert, il ne reste plus qu'à nettoyer le fichier afin d'extraire les tokens nous intéressants.
On vient ainsi extraire la première ligne du fichier pour la nettoyer afin de récupérer les identifiants de nos sommets.
Une fois les sommets récupérés, on vient extraire le reste des lignes qui correspondent à un arc par ligne. De la même manière que pour extraire nos sommets, on vient nettoyer chaque ligne afin d'y extraire le nom de l'arc ainsi que son prédécesseur et son successeur.

3. La troisième question nous demande d'implémenter une méthode *calculeDegre()* qui permet de calculer les degrés d'un sommet.

   Pour ce faire, on passe dans un premier temps l'identifiant du sommet en paramètre de notre méthode. Une fois l'identifiant récupéré, on vient boucler sur l'ensemble des arcs de notre graphe et pour chaque arc, on vérifie que notre sommet apparait comme successeur ou prédécesseur. En fonction de la réponse de la clause if, on incrémente ou non une variable degré qui représente, une fois la boucle finie, le nombre de fois où notre sommet est apparu comme étant successeur ou prédécesseur d'un arc, autrement dit le degré de notre sommet.

4. Afin de mettre en place un algorithme de calcul du plus court chemin, j'ai dans un premier temps implémenté des méthodes utilitaires :

   getSommetParNom(self, nom) : méthode qui retourne une instance d'un objet sommet en passant l'identifiant du sommet en paramètre.

   getSuccesseur(self, sommet) : méthode qui permet de récupérer l'ensemble des sommets qui sont successeur du sommet passé en paramètre.

   Une fois les méthodes utilitaires instanciées, on peut lancer notre méthode plusCourtChemin() qui se base sur l'algorithme de Dijkstra.

   Initialisation : L'algorithme prend tout d'abord en paramètre les identifiants du sommet de départ et du sommet d'arrivée. On vient ensuite récupérer l'instance de notre sommet de départ à l'aide de la fonction *getSommetParNom()* et on initialise un dictionnaire qui contiendra les chemins possibles mis à jour tout au long de l'algorithme.

   Boucle : Une fois cela fait, on fait tourner notre algorithme dans une boucle while qui vient vérifier que l'élément sur lequel on boucle n'est pas le sommet d'arrivée. En effet, si il s'agit du sommet d'arrivée, cela veut dire que l'ensemble des chemins a été parcouru et donc que l'on connait le plus court chemin.

   La boucle commence donc avec le sommet de départ sur lequel on va récupérer ses sommets successeurs. Et pour chaque sommet successeur, on va noter dans notre dictionnaire chemin{} le

Baribeaud Arthur
BDIA3
N° NAQ503552

nom des nouveaux chemins possibles ainsi que leur poids (ex : pour le sommet de départ A -> {A : 0, AB : 5, AC : 4}). Une fois les nouveaux chemins répertoriés, on supprime le précèdent chemin (dans notre exemple pop(A) -> {AB : 5, AC : 4}).

Puis on extrait de notre dictionnaire le nouveau chemin le plus court (ex : {AC : 4}) ainsi que le dernier sommet visité de ce chemin (ex : {AC : 4} -> C). Une fois cela fait, on remonte notre boucle avec le nouveau sommet et on recommence (ex : {AB : 5, AC : 4, ACD : 7, ACV : 6}) et pop(AC) {AB : 5, ACD : 7, ACV : 6} …).

Ainsi on vient parcourir l'ensemble des possibilités en prenant le soin de supprimer à chaque fois les chemins déjà visités et considérés comme plus petits. Cela emmène petit à petit, notre algorithme à se diriger vers notre sommet d'arrivée qui une fois désigné comme étant le chemin le plus court s'arrête.

5. L'algorithme de plus court chemin avec obstacle est équivalent à l'algorithme de plus court chemin à l'exception prêt qu'il possède un sommet « obstacle » par lequel ne doit pas passer notre chemin.

   Pour réaliser ce deuxième algorithme, j'ai réutilisé la méthode du plusCourtChemin() en y rajoutant une subtilité lors de l'appel à la méthode getSuccesseur(). En effet, lors de l'appel, on passe en paramètre l'identifiant de notre obstacle getSuccesseur(self, sommet, obstacle). De ce fait, la méthode getSuccesseur() nous renvoie l'ensemble des successeurs du sommet en y excluant le sommet obstacle. On peut ainsi continuer l'exécution de notre algorithme en s'assurant que le sommet obstacle n'interviendra pas dans le calcul du plus court chemin.

6. La comparaison de deux graphes est faite en deux étapes : comparaison des sommets puis comparaison des arcs.

   Pour ce faire, on implémente la méthode compareGraphe() dans la classe graphe qui prend en paramètre un graphe à comparer compareGraphe(self, second_graphe)

   <u>Comparaison des sommets :</u> Pour comparer les sommets des deux graphes, on exécute une double boucle for pour venir comparer chaque sommet du premier graphe par rapport à chaque sommet du second graphe. Et pour chaque sommet équivalent, on le note dans une liste sommet_commun[]

   <u>Comparaison des arcs :</u> On reprend le même principe que pour la comparaison des sommets à l'exception près que l'on compare les arcs non pas uniquement sur leurs identifiants (nom), mais sur leurs successeurs AND prédécesseurs AND poids