

## SAE S2.02 RAPPORT

### Représentation d'un graphe

#### Question 1 : écriture de la classe Arc

Après s'être approprié le sujet, l'écriture de la classe Arc ne nous a pas posé de problème. Les méthodes `getDest` et `getCout` sont de simples getters. Nous n'avons pas ressenti de difficulté particulière.

#### Question 2 : écriture de la classe Arcs

De même que pour la classe Arc, la classe Arcs s'est écrite toute seule. Cette dernière est une liste d'Arc. Afin d'écrire sa méthode `ajouterArc`, nous avons utilisé la méthode `add`.

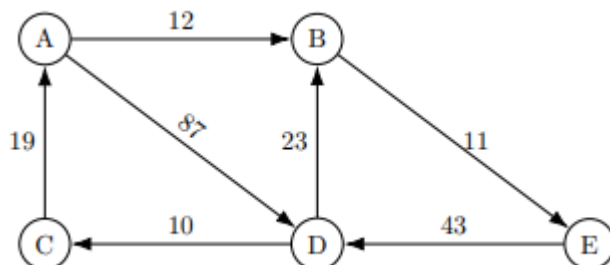
#### Question 3 : écriture de l'interface Graphe

Cette interface possède deux méthodes : `listeNoeuds` et `suivants`, qui seront utilisés par la suite par la classe `GrapheListe`. Elle nous a fait une petite piqure de rappel sur la façon d'écrire une interface.

#### Question 4 : écriture de la classe GrapheListe

La classe `GrapheListe` était plus longue à écrire que les autres. Cependant, elle ne nous a toujours pas posé problème. Cette dernière regroupe les classes que nous avons écrites précédemment, et utilise l'interface `Graphe`. Nous avons alors codé les méthodes `listeNoeuds` et `suivants`, en plus de la méthode `ajouterArc`. C'est cette dernière méthode qui nous a fait nous poser sur sa réalisation. Nous devons étudier les cas dans lesquels le noeud de départ et/ou d'arrivée existaient ou non, et les créer si besoin. Afin d'organiser notre code, nous avons initialement mis des commentaires.

#### Question 5 : écriture du main



Pour le Main, nous avons repris le graphe donné dans le sujet et nous l'avons bien récréée.

#### Question 6 : écriture du toString

Afin d'écrire le toString, nous avons utilisé un forEach que nous avons réutilisé par la suite dans nos programmes. Cette méthode est très pratique car elle nous rend plus facile la compréhension et l'écriture du code.

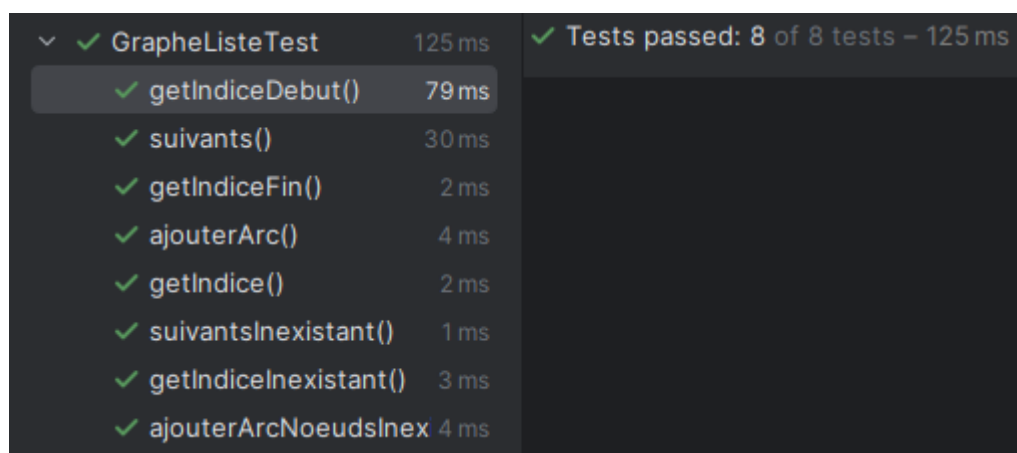
#### Question 7 : écriture des tests unitaires

Nous avons rencontré quelques problèmes pour écrire la forme des tests : parfois la méthode assertEquals comparait les références et non le contenu. Nous avons donc décidé de contourner ce problème en comparant les String (grâce à notre méthode toString), notamment dans les tests ajouterArc, suivants et suivantsInexistant.

Nous avons donc écrit en tout 8 tests :

- getIndex, getIndexDebut, getIndexFin et getIndexInexistant pour la méthode getIndex;
- suivants et suivantsInexistant pour la méthode correspondante;
- ajouterArcNoeudsInexistants et ajouterArc pour la méthode correspondante.

Résultat des tests : aucune erreur.



✓ GrapheListeTest	125 ms	✓ Tests passed: 8 of 8 tests – 125 ms
✓ getIndexDebut()	79 ms	
✓ suivants()	30 ms	
✓ getIndexFin()	2 ms	
✓ ajouterArc()	4 ms	
✓ getIndex()	2 ms	
✓ suivantsInexistant()	1 ms	
✓ getIndexInexistant()	3 ms	
✓ ajouterArcNoeudsInex	4 ms	

#### Calcul du plus court chemin par point fixe

##### Question 8 : écriture de l'algorithme de Bellman-Ford

Afin d'écrire cet algorithme, nous nous sommes aidé de l'algorithme de Dijkstra présent dans la partie "Calcul du meilleur chemin par Dijkstra". Cela nous a beaucoup aidé face à la difficulté de reprendre une écriture algorithmique.

##### Question 9 : écriture du programme correspondant

L'algorithme de Bellman-Ford nécessite pour son bon fonctionnement, de connaître les antécédents de chaque sommets, or notre implémentation nous donne seulement les successeurs. Nous avons donc dû chercher les antécédents pour chaque sommet. Un sommet a pour antécédent les sommets qui l'ont en successeur. Ainsi nous avons pu déterminer les antécédents et réussir à implémenter l'algorithme de Bellman-Ford. Ceci était la plus grosse difficulté de ce programme, nous nous perdions un peu dans les boucles nécessaires, nos commentaires nous ont aidés à rendre cela plus clair. Cependant, le calcul des antécédents que nous avons implémenté, est amené à calculer plusieurs fois les antécédents du même sommet, notre algorithme va parcourir les noeuds, calculer leurs antécédents, vérifier si un chemin plus court existe et recommencer, le calcul des antécédents va donc à chaque étape se relancer alors qu'il n'est pas amené à changer.

Question 10 : écriture du main

Pour le main, une fois toutes les méthodes ajoutées, nous avons créé le graphe présenté dans la figure 1, et testé la méthode résoudre, aucune difficultés, les méthodes et le constructeur fonctionnaient.

Question 11 : écriture des tests unitaires

✓ BellmanFordTest	86 ms	✓ Tests passed: 1 of 1 test – 86 ms
✓ résoudre()	86 ms	

Pour le test du programme de l'algorithme de Bellman-Ford, nous avons été au plus simple, nous avons déroulé à la main l'algorithme du point fixe, rentrer les résultats dans les assertEquals, en tant qu'argument "actual", et utiliser la méthode résoudre sur le même graphes, aucune erreur, donc l'ordinateur a trouvé le bon résultat.

Question 12 : écriture de calculerChemin

Pour calculerChemin, on prend en argument le sommet dont on veut le chemin, on prend le nœud parent avec .getParent que l'on ajoute dans une liste, on répète l'opération sur ce nœud puis sur le suivant... Une fois que le nœud parent est égal à nul on arrête, on inverse la liste, car le chemin a été fait en sens inverse, à l'aide de la méthode Collection.reverse(List l) on obtient alors la liste des sommets, menant jusqu'au nœud demandé en argument, avec la distance minimale.

## Calcul du meilleur chemin par Dijkstra

Question 13 : écriture de la méthode résoudre

Pour cette méthode, nous avons pris la liste des sommets du graphes (nous avons copié la liste à l'aide d'une boucle car nous sommes amenés à enlever des nœuds au fur et à mesure de la progression de l'algorithme). Tant que tous les sommets ne sont pas traités, on continue. On commence par prendre le sommet qui a la valeur la plus petite, en premier lieu c'est le sommet de début qui aura la valeur la plus petite, il est à 0 tandis que les autres sont à Double.MAX\_VALUE, on va donc retirer ce sommet de la liste, et calculer le chemin de chacun de ses successeurs, si la nouvelle distance trouvée est plus petite pour ses successeurs en partant de lui, on la remplace. Une fois tous les successeurs étudiés, on répète cette opération, chercher la valeur min, ses successeurs... Jusqu'à ce que la liste des nœuds soit vide. Nous sommes très satisfait du rendu final de cette méthode, nous avons essayé d'aller au plus simple et pensons avoir réussi

Question 14 : écriture des tests unitaires

✓ DijkstraTest	123 ms	✓ Tests passed: 2 of 2 tests – 123 ms
✓ résoudreDistance()	103 ms	"C:\Program Files\Java\jdk-21\bin\java.exe" ...
✓ résoudreParents()	20 ms	Process finished with exit code 0

Nous avons été au plus simple pour les tests, ils fonctionnent correctement et prouvent le bon fonctionnement de notre méthode

### Question 15 : écriture du MainDijkstra

Une fois que la méthode resoudre de la classe Dijkstra est correct, le main ne pose aucun problème, on utilise seulement un objet de la classe Dijkstra pour appeler la fonction resoudre, le mainDijkstra est quasiment identique à notre Main, seulement la méthode de calcul n'est pas la même

## Validation et expérimentation

### Question 16 : présentation des résultats obtenus

```
public static void main(String[] args) {
    GrapheListe g = new GrapheListe();
    g.ajouterArc( depart: "A", destination: "B", cout: 12);
    g.ajouterArc( depart: "C", destination: "A", cout: 19);
    g.ajouterArc( depart: "A", destination: "D", cout: 87);
    g.ajouterArc( depart: "D", destination: "C", cout: 10);
    g.ajouterArc( depart: "D", destination: "B", cout: 23);
    g.ajouterArc( depart: "B", destination: "E", cout: 11);
    g.ajouterArc( depart: "E", destination: "D", cout: 43);

    BellmanFord b = new BellmanFord();
    Dijkstra d = new Dijkstra();
    long date_debutB = System.nanoTime () ;
    Valeur v = b.resoudre(g, depart: "A");
    long date_finB = System.nanoTime () ;

    long date_debutD = System.nanoTime () ;
    Valeur v2 = d.resoudre(g, depart: "A");
    long date_finD = System.nanoTime () ;

    long totalD = date_finD - date_debutD;
    long totalB = date_finB - date_debutB;
    System.out.println("Pour Dijkstra : "+totalD);
    System.out.println("Pour Bellman-Ford : "+totalB);
}
```

Nous avons créé nous même le graphe ci-dessus sur lequel nous avons calculé le temps en utilisant l'algorithme de Dijkstra et celui de Bellman-Ford. Nous avons également fait des tests avec les fichiers donnés. Les résultats obtenus en globalité sont les suivants :

```
"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Algorithme de Dijkstra sur petit graphe a fini en : 418100
Algorithme de Bellman-Ford sur petit graphe a fini en : 1390000
Algorithme de dijkstra sur grand graphe a fini en : 51751500
Algorithme de Bellmon-Ford sur grand graphe a fini en : 3180324100

Process finished with exit code 0
```

#### Question 17 : analyse des résultats obtenus

Nous pouvons remarquer que l'algorithme de Dijkstra est plus rapide que celui de Bellman-Ford (1 284 700ms contre 8 962 700ms). Ceci semble logique au vu de la complexité de ces deux algorithmes, l'algorithme de Dijkstra est bien plus performant car il ne nécessite pas autant de calcul que l'algorithme de Bellman-Ford. On voit d'ailleurs que plus le graphe est grand, contient beaucoup de noeuds et beaucoup d'arc, plus l'écart entre les deux algorithmes se creuse.

#### Question 18 : algorithme le plus efficace

Pour conclure, nous pensons que l'algorithme qui fonctionne le mieux expérimentalement est celui de Dijkstra car il est bien plus rapide, et donc plus efficace. Cela est dû à sa complexité qui est moindre que l'algorithme de Bellman-Ford (il y a moins d'itérations, moins de calculs).

#### Question 19 : constructeur

Pour le constructeur, le format des fichiers graphes données nous a vraiment facilité la tâche. Nous avons créé un reader sur le nom du fichier entré en paramètre, puis à partir de cela, un BufferedReader nous permettant de lire le fichier ligne par ligne

### **CONCLUSION**

Pour conclure, cette SAE nous a permis de mieux cerner l'implémentation d'algorithmes mathématiques dans le langage java. Nous avons pu découvrir l'importance des mathématiques dans l'informatique, et l'importance de toujours revoir ses méthodes afin de les rendre plus efficaces. Nous avons rencontré des difficultés au niveau de la compréhension de l'algorithme de Dijkstra et de certains tests. Néanmoins nous avons réussi à surmonter ces difficultés en découpant le problème pour mieux l'analyser. Cela nous a permis de rendre un travail sans erreurs.