Représentation d'un graphe

Ecriture de la classe Nœud et des méthodes de Noeud

Ecriture de la classe Arc

Ecriture de l'interface Graphe

Ecriture de la classe GrapheListe

Ecriture de test unitaire

Calcul du plus court chemin par point fixe

Ecriture de l'algorithme du point fixe

Ecriture de la classe Main

Ecriture de test unitaire

Calcul du meilleur chemin par Dijkstra

Ecriture de la classe Dijkstra

Ecriture de test unitaire

Ecriture de MainDijkstra

Validation et expérimentation

Question 22)

BellmanFord: Graphe1.txt	Dijkstra: Graphe1.txt
1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:1.7976931348623157E308 p:null 4 -> V:1.7976931348623157E308 p:null 5 -> V:1.7976931348623157E308 p:null 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null 1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:1.7976931348623157E308 p:null 4 -> V:15.0 p:1 5 -> V:1.7976931348623157E308 p:null 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null	1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:1.7976931348623157E308 p:null 4 -> V:1.7976931348623157E308 p:null 5 -> V:1.7976931348623157E308 p:null 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null 1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:1.7976931348623157E308 p:null 4 -> V:15.0 p:1 5 -> V:1.7976931348623157E308 p:null 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null

1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:1.7976931348623157E308 p:null 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null	1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:1.7976931348623157E308 p:null 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null
1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null	1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:1.7976931348623157E308 p:null 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null
1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:1.7976931348623157E308 p:null 7 -> V:32.0 p:3 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null	1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:1.7976931348623157E308 p:null 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null
1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:1.7976931348623157E308 p:null 7 -> V:32.0 p:3 8 -> V:21.0 p:3 9 -> V:1.7976931348623157E308 p:null	1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:22.0 p:5 8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null
1 -> V:0.0 p:null 10 -> V:1.7976931348623157E308 p:null 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:1.7976931348623157E308 p:null 7 -> V:32.0 p:3	1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:22.0 p:5

8 -> V:21.0 p:3 9 -> V:16.0 p:3	8 -> V:1.7976931348623157E308 p:null 9 -> V:1.7976931348623157E308 p:null
1 -> V:0.0 p:null 10 -> V:24.0 p:3 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:1.7976931348623157E308 p:null 7 -> V:32.0 p:3 8 -> V:21.0 p:3 9 -> V:16.0 p:3	1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:22.0 p:5 8 -> V:21.0 p:3 9 -> V:1.7976931348623157E308 p:null
1 -> V:0.0 p:null 10 -> V:24.0 p:3 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:1.7976931348623157E308 p:null 7 -> V:20.0 p:4 8 -> V:21.0 p:3 9 -> V:16.0 p:3	1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:22.0 p:5 8 -> V:21.0 p:3 9 -> V:16.0 p:3
1 -> V:0.0 p:null 10 -> V:24.0 p:3 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:20.0 p:4 8 -> V:21.0 p:3 9 -> V:16.0 p:3	1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:20.0 p:4 8 -> V:21.0 p:3 9 -> V:16.0 p:3
1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:25.0 p:5 7 -> V:20.0 p:4 8 -> V:21.0 p:3 9 -> V:16.0 p:3	1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:24.0 p:9 7 -> V:20.0 p:4 8 -> V:21.0 p:3 9 -> V:16.0 p:3
1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1	1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1

5 -> V:12.0 p:2 6 -> V:24.0 p:9 7 -> V:20.0 p:4 8 -> V:21.0 p:3 9 -> V:16.0 p:3 1 -> V:0.0 p:null 10 -> V:21.0 p:5 2 -> V:4.0 p:1 3 -> V:13.0 p:2 4 -> V:15.0 p:1 5 -> V:12.0 p:2 6 -> V:24.0 p:9 7 -> V:20.0 p:4 8 -> V:18.0 p:9 9 -> V:16.0 p:3	5 -> V:12.0 p:2 6 -> V:24.0 p:9 7 -> V:20.0 p:4 8 -> V:18.0 p:9 9 -> V:16.0 p:3

L'algorithme de BellmanFord parcours l'entièreté des Nœuds à chaque fois, tout en modifiant le chemin minimal menant à un Nœud si le chemin minimal déjà affecté est plus grand, alors que L'algorithme de Dijkstra lui affecte une valeur minimale aux parents du Nœud sur lequel il est actuellement, puis le parent avec le chemin minimal devient le Nœud actuelle et réitère les mêmes opérations sans jamais repassé par les Nœuds pour les lesquels il a déjà effectué cette opération.

Nœud = Sommet.

Question 23)

D'après l'observation de la question précédente, on peut en conclure que l'algorithme de BellmanFord est plus performant que celui de Dijkstra pour les graphes avec peu de chemins différents possibles, car il effectuera moins d'itérations, mais par conséquents pour les graphes avec beaucoup de chemins différents possibles, le second algorithme sera plus performant.

Par conséquent l'efficacité des algorithmes dépendent du nombre d'arc moyen par Sommet.

Question 24)

Nom du Graphe	Temps_BellmanFord	Temps_Dijkstra	Moy d'Arcs/Noeud	Ratio
Graphe1.txt	1573700	192700	7,8	177051,2821
Graphe101.txt	6760200	1906600	53,7	90383,6127
Graphe102.txt	2962500	1587400	52,1	26403,6098
Graphe103.txt	2273100	1670200	53,9	11181,3798
Graphe104.txt	2130900	1709000	51,3	8217,7639
Graphe105.txt	2607800	1837800	50,9	15115,8225
Graphe11.txt	37400	45100	9,0	-855,5556
Graphe12.txt	41900	40800	7,0	157,1429
Graphe13.txt	37700	27700	7,0	1428,5714
Graphe14.txt	40400	28900	8,2	1402,4390
Graphe15.txt	92000	27500	6,0	10750,0000
Graphe2.txt	58100	37700	8,4	2428,5714
Graphe201.txt	6555100	6339500	101,6	2123,0921
Graphe202.txt	5360900	3248500	103,4	20425,4496
Graphe203.txt	4660400	2873700	101,4	17629,0084
Graphe204.txt	5321600	3201100	100,5	21093,2060
Graphe205.txt	4333500	3029400	102,1	12774,0229
Graphe21.txt	46600	74700	13,0	-2161,5385
Graphe22.txt	38500	100000	12,1	-5082,6446
Graphe23.txt	36600	79700	11,5	-3747,8261
Graphe24.txt	40900	59800	13,6	-1389,7059
Graphe25.txt	47600	81400	11,7	-2888,8889
Graphe301.txt	11513900	7177300	152,1	28517,7554
Graphe302.txt	10451800	5355100	152,0	33536,8047
Graphe303.txt	7755500	5626000	153,1	13911,0269

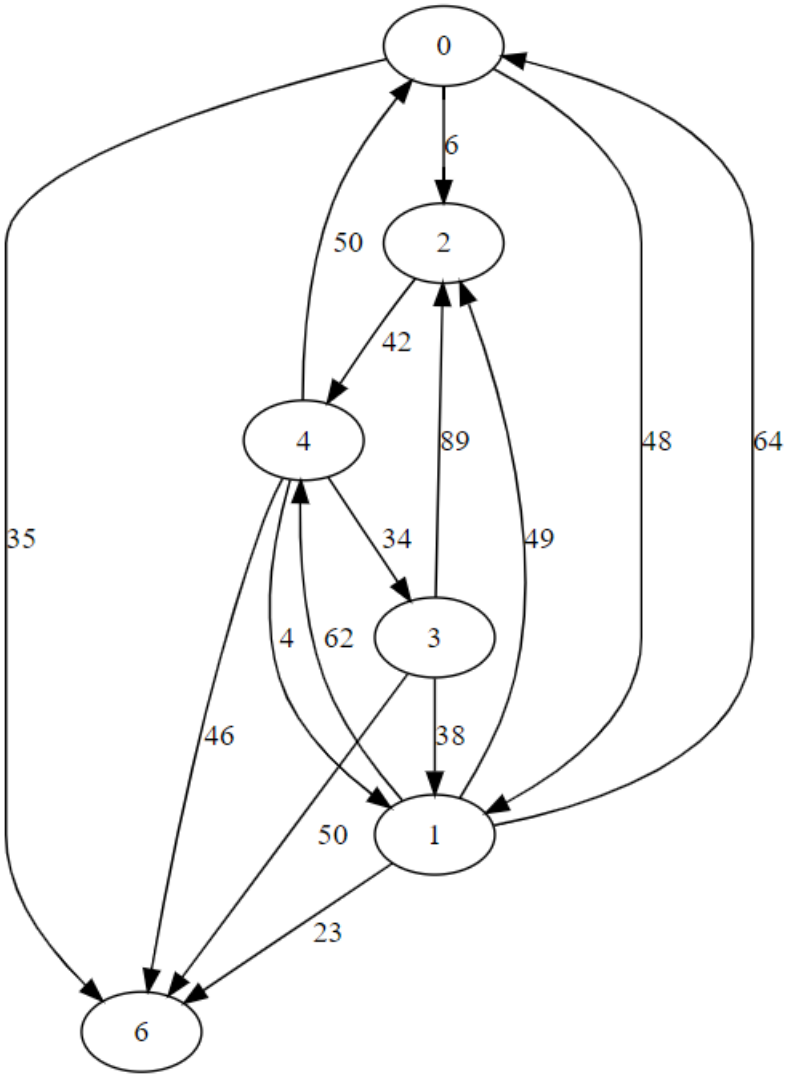
Graphe71.txt	421400	231900	36,7	5161,4786
Graphe72.txt	469400	248900	37,2	5927,4194
Graphe73.txt	471100	234500	37,6	6292,5532
Graphe74.txt	412600	234100	38,0	4700,9029
Graphe75.txt	402000	232500	36,5	4642,0188
Graphe801.txt	60240900	40467500	403,0	49060,6391
Graphe802.txt	52451400	38338200	401,9	35112,7034
Graphe803.txt	82944700	42923400	403,0	99320,7594
Graphe804.txt	75639700	38860100	402,5	91375,0505
Graphe805.txt	58171500	39996100	400,7	45356,5747
Graphe81.txt	557500	301800	41,9	6106,2687
Graphe82.txt	545700	304400	42,9	5631,2719
Graphe83.txt	429500	307500	42,3	2882,4572
Graphe84.txt	690100	302900	43,1	8983,7587
Graphe85.txt	645800	299500	41,7	8304,5564
Graphe901.txt	72968000	48897500	452,0	53258,8173
Graphe902.txt	68538600	47616900	450,4	46448,6265
Graphe903.txt	71258300	47893400	454,3	51427,5338
Graphe904.txt	81440100	56419500	450,8	55503,4827
Graphe905.txt	72885900	48117700	450,4	54996,9900
Graphe91.txt	679100	407600	46,8	5798,5287
Graphe92.txt	803400	377000	46,5	9163,3238
Graphe93.txt	668500	399500	46,7	5756,0628
Graphe94.txt	667300	376800	48,2	6032,5335
Graphe95.txt	680100	374700	48,1	6347,8060
Moy_Final	15865773	9722276	134,0	23805,0465

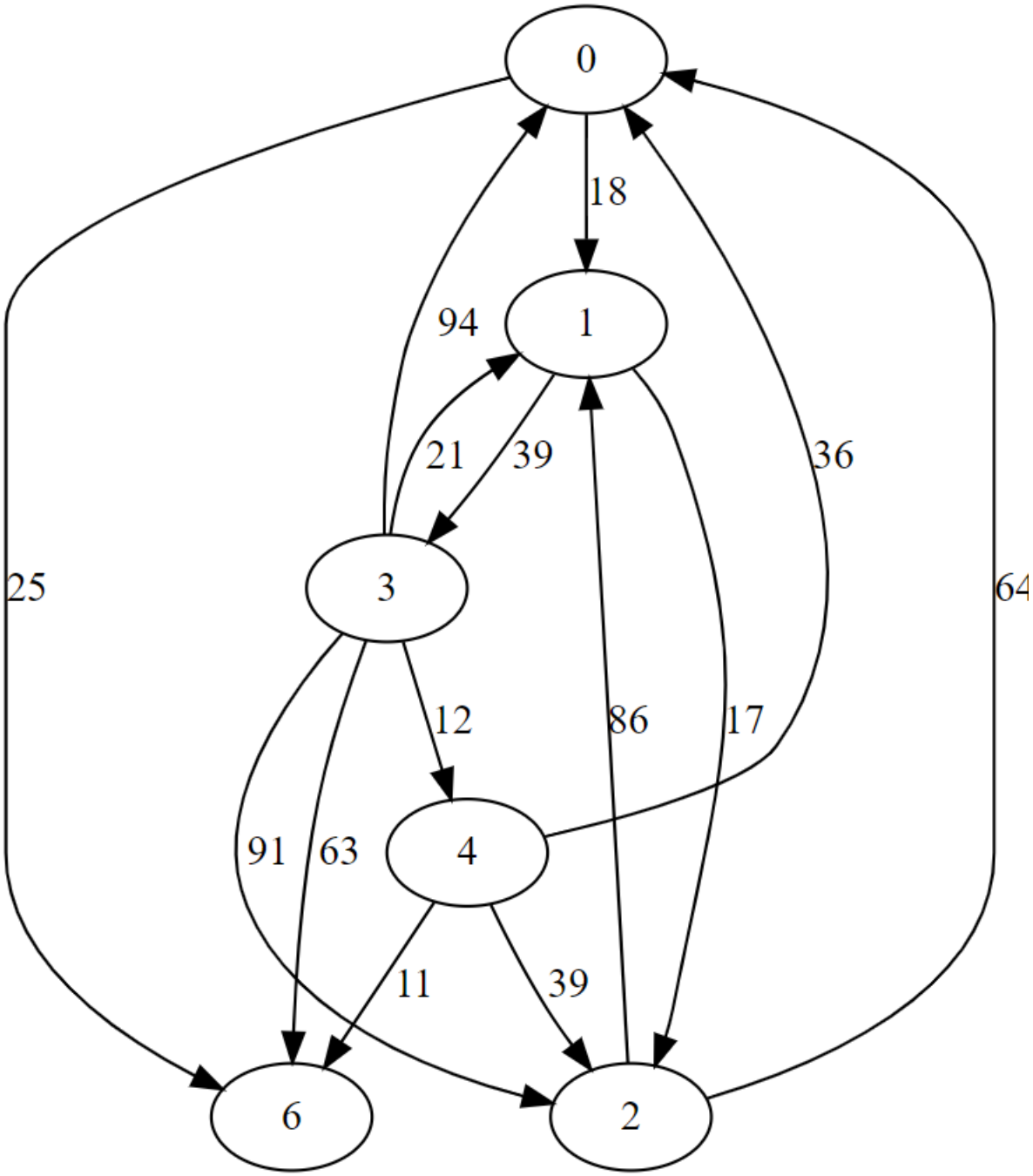
(Lorsque le ratio est positive cela signifie que l'algorithme de Dijkstra est plus performant)

Comme on peut le voir dans l'extrait des résultats obtenus, l'algorithme le plus efficace est très souvent celui de Dijkstra, avec une moyenne de 23,805µs de différences par nombre d'arc moyen par Nœud, les rares fois où l'algorithme de BellmanFord est plus performant sont observé lorsque le nombre d'arc moyen par nœud est faible, est que par conséquent peu de chemins sont possibles.

On en conclu donc que pour les graphes où de nombreux chemins possibles existent l'algorithme de Dijkstra est le plus performant, cependant lorsque le nombre de chemins est très faible l'algorithme de BellmanFord est plus performant, par conséquent il vaut mieux utiliser celui de Dijkstra.

Question 26)


0
6
50
2
42
4
89
48
64
35
34
49
4
62
3
46
38
50
1
23
6



Question 27)

Comme pour la question 24, l'algorithme le plus performant est celui de Dijkstra, l'algorithme de BellmanFord est le plus efficace parfois que pour des graphes de petites tailles, le graphe le plus grand pour lequel il a était plus performant comportait 55 Nœuds, mais après cette taille l'algorithme de Dijkstra est de plus en plus performant, voici ci-dessous les résultats moyens pour 1000 graphes allant de 3 Nœuds à 1003Noeuds

Moy_Final	134853943	29666381	251,0	297394,3834

Question 28)

Le ratio de performance par noeud est 209 120ns de différence en Moyenne totale avec une moyenne de 503 Nœud, ce ratio augmente de plus en plus avec le nombre de Nœud

Question 29)

On peut conclure de cette étude que plus un graphe est grand, plus la différence de temps entre l'algorithme de BellmanFord et de Dijkstra est élevée, par conséquent plus un graphe est grand est plus l'utilisation de l'algorithme de Dijkstra est recommandé. Cependant pour les graphes de petites tailles ce ratio est parfois négatif, ce qui signifie que l'algorithme de BellmanFord est plus performant dans ces conditions, mais comme ce n'est pas toujours le cas il vaut mieux toujours utiliser l'algorithme de Dijkstra.

Question 30)

Afin de pouvoir utiliser la classe GrapheListe dans la classe Labyrinthe j'ai du créer un nouveau package Graphe dans lequel j'ai déplacer toutes les Class en lien avec les graphes, car les classes présentes dans le src sont par défaut ajouter à un package non nommé, ce qui le rendait inaccessible pour des éléments d'autres packages puisqu'il fallait l'importer.

Question 31)

```
XX.XXXXX.XX
X..X..X...X
X.XXX...XXX
X..XX.XXXXX
X.....XXXXX
XXXXXXXXXXX
```

Résultat du chemins les plus courts pour le laby ci-dessus pour se rendre à la sortie.

1,1 -> V:2.0 p:2,1

1,2 -> V:3.0 p:1,1

1,3 -> V:4.0 p:1,2

1,4 -> V:5.0 p:1,3

2,0 -> V:0.0 p:null

2,1 -> V:1.0 p:2,0

2,3 -> V:5.0 p:1,3

2,4 -> V:6.0 p:1,4

3,4 -> V:7.0 p:2,4

4,1 -> V:13.0 p:5,1

4,4 -> V:8.0 p:3,4

5,1 -> V:12.0 p:5,2

5,2 -> V:11.0 p:5,3

5,3 -> V:10.0 p:5,4

5,4 -> V:9.0 p:4,4

6,2 -> V:12.0 p:5,2

7,1 -> V:14.0 p:7,2

7,2 -> V:13.0 p:6,2

8,0 -> V:16.0 p:8,1

8,1 -> V:15.0 p:7,1

9,1 -> V:16.0 p:8,1

Question 32)

J'ai commencé par restaurer la version initiale de la Classe Labyrinthe, puis j'ai créer une classe GrapheLabyrinthe qui implémente l'interface Graphe, les méthodes de Graphe dont elle à hérité ont était redéfinis par un appelle d'elle-même sur l'attribut de GrapheLabyrinthe qui est un GrapheListe, ainsi j'ai pu créer un graphe dans cette classe sans avoir à modifié le reste du code.

Avis sur la SAE)

Vincent : Cette SAE nous a appris l'utilisation des Map et TreeMap mais également à ne pas sous-estimer les tâches à faire. J'ai rencontré plusieurs problèmes, notamment sur les algorithmes comme des boucles infinies ou des résultats faux.

Mathis : Lors de cette SAE j'ai appris à utiliser les regex avec les classes Pattern et Matcher afin d'éviter d'exécuter les algorithme sur les graphes qui n'était pas fournis pas arche comme il était demander pour la question 24, j'ai également découvert l'existence d'un package prédéfinit pour les classes qui ne sont pas dans un packages à l'intérieur du src, qui est un package non nommé, à cause duquel il est impossible à une classe dans un package définit par nous-même d'accéder à ses classes.

J'ai également vu une erreur dans un commentaire de la classe Valeur qui était fournis, dans lequel il était écrit qu'une Map était une table et qu'une table c'était la même chose qu'un dictionnaire, or un dictionnaire est une table particulière, par conséquent une un dictionnaire est une table mais une table n'est pas forcément un dictionnaire, cela peut conduire à des erreurs sur l'interprétation du fonctionnement de la classe Valeur, qui a pour attribut deux TreeMap qui sont des arborescence de table, dans lesquels ont peut ajouter une clé et définir une seule valeur, ce qui constitue une table à chaque pair, un dictionnaire en java est un HashMap, où la seule différence avec le TreeMap est que à chaque clé on peut affecter plusieurs valeurs.

Lorsque j'ai utilisé cette classe pour créer l'algorithme de BellmanFord même si j'ai pas vu le commentaire j'ai confondu TreeMap et HashMap, ce qui a faussé totalement le code de mon algorithme.

La conclusion de ce pavé est simplement si une SAE où des classes fournit utilise des TreeMap, pour aiguiller les élèves sur ce que c'est n'écrivait pas « table = dictionnaire » mais simplement « Map = table » Merci de votre compréhension.