

SAE 2.02
Exploration Algorithmique
Recherche du plus court chemin dans un graphe

Représentation d'un graphe	1
Calcul du plus court chemin par point fixe (Bellman-Ford)	1
Calcul du meilleur chemin par Dijkstra	3
Validation et expérimentation	3
Extension : Intelligence Artificielle et labyrinthe	7
Bilan de la SAE	7

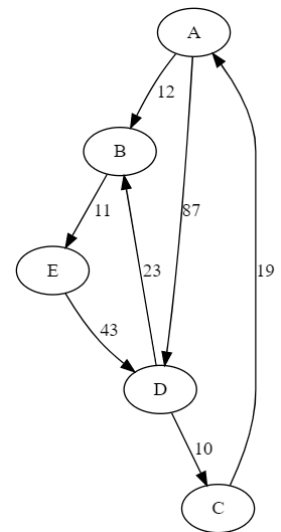
I. Représentation d'un graphe

Dans le cadre de la SAE S2.02, il nous a été demandé d'implémenter des algorithmes de recherches de chemins minimaux dans un graphe en langage Java, afin de pouvoir appliquer ces méthodes sur des objets de type Labyrinthe par exemple.

Cela nous a d'abord amené à représenter un graphe en Java. Pour cela, nous avons créé une interface Graphe (qui sert à leur représentation) ainsi que plusieurs classes : la classe Noeud (qui représente les différents nœuds, ou sommets, d'un graphe), la classe Arc (qui représente les différents arcs d'un graphe reliant les nœuds), et la classe GrapheListe implémentant l'interface Graphe (qui représente les données d'un graphe).

Nous avons donc programmé les méthodes demandées dans ces classes (notamment dans GrapheListe, la méthode suivant permettant de retourner la liste des arcs, c'est-à-dire les chemins menant à d'autres nœuds, partant de ce nœud) tout en réalisant des tests lors de la conception.

Par ailleurs, nous nous sommes aussi occupés des méthodes toString et toGraphviz dans GrapheListe, permettant l'affichage de graphes. toGraphviz renvoie une String encodée de manière à pouvoir être interprétée par Graphviz, ce qui permet d'obtenir un affichage du graphe. Nous nous sommes par ailleurs occupés de la génération d'objets Graphe à partir de fichiers.



[Question 10] Graphe créé avec le résultat de la méthode toGraphviz() →

II. Calcul du plus court chemin par point fixe (Bellman-Ford)

Par la suite, nous avons commencé à réaliser les algorithmes. Nous devons tout d'abord réfléchir à la manière de structurer l'algorithme de point fixe. Cette méthode consiste à calculer le chemin minimal pour tous les sommets, puis à recommencer jusqu'à arriver à un "point fixe", c'est-à-dire un état stable qui établit les chemins minimaux entre les sommets. Cet algorithme est aussi connu sous le nom d'algorithme de Bellman-Ford.

Il nous a été demandé en premier lieu d'écrire l'algorithme (cf. page suivante). Nous l'avons ensuite implémenté en Java, comme méthode dans une classe dédiée, en utilisant la classe Valeur fournie.

Nous avons ensuite utilisé cette méthode dans un main et nous avons réalisé les tests correspondants. Nous avons finalement codé la méthode suivant permettant de retourner la liste des arcs, c'est-à-dire les chemins menant à d'autres nœuds, partant de ce nœud.

[Question 13]

fonction pointFixe (Graphe g InOut, Noeud depart)

début

// initialisation

L(depart) <- 0

pour i allant de 0 à g.listeNoeuds().size() faire

si g.listeNoeuds().get(i) != depart faire

L(g.ListeNoeuds().get(i)) <- + infini

fin si

fin pour

// fin initialisation

// étapes

boolean pointFixe = faux

tant que (!pointFixe) faire

pointFixe <- vrai

pour i allant de 0 à g.ListeNoeuds().size() faire

pour j allant de 0 à g.suivants(g.ListeNoeuds().get(i)).size faire

tmpNoeud <- g.suivants(g.listeNoeuds().get(i)).get(j).getDest()

tmpActualVal <- L(tmpNoeud)

tmpNewVal <- L((g.listeNoeuds().get(i)) +

g.suivants(g.listeNoeuds().get(i)).get(j).getCout()

si (tmpNewVal < tmpActualVal) faire

L(tmpNoeud) <- tmpNewVal

L(tmpNoeud).setParent <- g.listeNoeuds().get(i)

pointFixe <- faux

fsi

fpour

fpour

ftantque

fin

Lexique:

g : Graphe, graphe étudié

depart : Noeud, noeud d'origine du chemin recherché

L : valeur de la distance entre le noeud d'origine et le noeud en paramètre de L

i : entier, itération

pointFixe : booléen, vrai si deux étapes successives se répètent (c'est-à-dire il n'y a eu aucun changement entre deux étapes). Il correspond à l'état de point fixe.

j : entier, itération

tmpNoeud : Noeud, valeur courante du Noeud de destination par rapport au Noeud étudié dans la boucle.

tmpActualVal : réel, valeur courante de tmpNoeud (L(tmpNoeud))

tmpNewVal : réel, valeur courante du Noeud étudié dans la boucle additionnée au coût de l'arc le reliant à tmpNoeud

III. Calcul du meilleur chemin par Dijkstra

Il nous restait alors à implémenter le deuxième algorithme de recherche de chemin dans un graphe, l'algorithme de Dijkstra. Étant donné qu'on nous fournissait l'algorithme, nous avons dû le retranscrire en code Java dans une classe dédiée, avec le tests unitaire de la méthode résoudre et le main permettant de l'utiliser.

Cette partie était globalement similaire à la partie 2 sur l'algorithme de Bellman-Ford.

IV. Validation et expérimentation

Dans cette partie, il nous a été demandé d'évaluer les performances des algorithmes implémentés précédemment et de les comparer (comprendre leurs différences et en quoi ils apportent quelque chose par rapport à l'autre).

[Question 21] Nous avons pu observer que :

- L'algorithme de Bellman-Ford applique le calcul du plus court chemin sur tous les sommets du graphes à chaque itération, jusqu'à ce que 2 itérations donnent des résultats identiques.
- L'algorithme de Dijkstra applique le calcul du plus court chemin pour un nœud jusqu'à trouver le plus court chemin pour ce nœud, avant de passer au suivant.

[Question 22] On peut ainsi conclure qu'avec l'algorithme de Bellman-Ford, on peut espérer un nombre plus faibles ou plus élevé d'itérations pour trouver le chemin le plus court, tandis qu'avec l'algorithme de Dijkstra, on a besoin d'effectuer autant d'itérations que de noeuds dans le graphe.

[Question 23] Nous avons mesuré le temps d'exécution des deux algorithmes sur des graphes de différentes tailles, avec comme point de départ le sommet "A" et comme point d'arrivée le sommet "E".

D'après les résultats obtenus (cf. ci-derrière), l'algorithme de Dijkstra est plus efficace que celui de Bellman-Ford (en termes de rapidité de temps d'exécution).

L'algorithme de Bellman-Ford est plus facile à appréhender et à implémenter, mais il a des performances moindres à celui de Dijkstra.

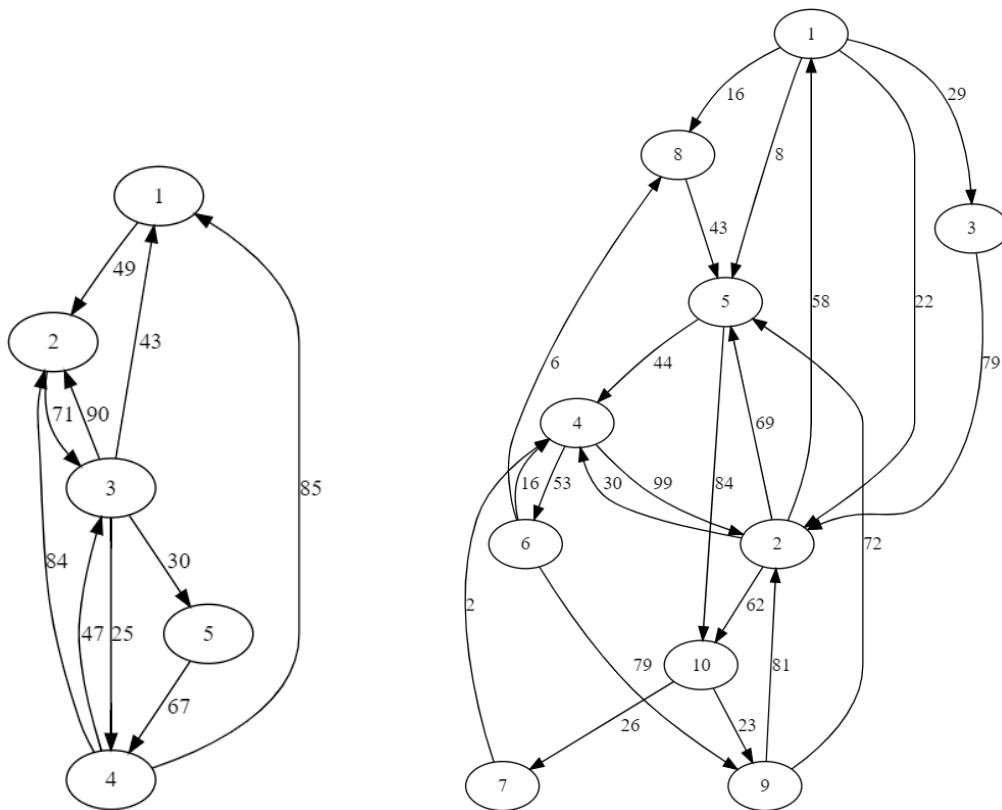
	bF:E	Dj:E		bF:E	Dj:E		bF:E	Dj:E
Graphe1.txt	107300	9100	Graphe81.txt	17800	17900	Graphe605.txt	4482100	
Graphe2.txt	4600	3800	Graphe82.txt	14000	16500	19600		
Graphe4.txt	11100	3200	Graphe83.txt	10400	17000			
Graphe5.txt	9300	3200	Graphe84.txt	16700	15700	Graphe701.txt	18400	19900
			Graphe85.txt	11800	16400	Graphe702.txt	19700	19400
Graphe11.txt	4800	2700				Graphe703.txt	18100	18300
Graphe12.txt	5100	3200	Graphe91.txt	12000	16900	Graphe704.txt	17300	19500
Graphe13.txt	6800	2700	Graphe92.txt	10700	16500	Graphe705.txt	17800	17400
Graphe14.txt	7800	5500	Graphe93.txt	16700	17500			
Graphe15.txt	6600	3900	Graphe94.txt	10300	16200	Graphe801.txt	24000	17800
			Graphe95.txt	18100	16100	Graphe802.txt	23500	19100
Graphe21.txt	6800	3700				Graphe803.txt	24900	19000
Graphe22.txt	6700	4800	Graphe101.txt	9600	8800	Graphe804.txt	21400	19600
Graphe23.txt	9700	11600	Graphe102.txt	9900	8200	Graphe805.txt	22100	17500
Graphe24.txt	9000	12200	Graphe103.txt	11400	9300			
Graphe25.txt	9000	15900	Graphe104.txt	10400	22200	Graphe901.txt	16500	20300
			Graphe105.txt	13000	11000	Graphe902.txt	20400	17400
Graphe31.txt	9600	9800				Graphe903.txt	20100	19900
Graphe32.txt	6000	6000	Graphe201.txt	16800	19700	Graphe904.txt	20500	21000
Graphe33.txt	6900	6000	Graphe202.txt	16900	19800	Graphe905.txt	18800	17400
Graphe34.txt	17300	17400	Graphe203.txt	18100	20300			
Graphe35.txt	7800	14700	Graphe204.txt	14000	18500			
			Graphe205.txt	17400	17700			
Graphe41.txt	9900	17300						
Graphe42.txt	9600	18200	Graphe301.txt	20300	19900			
Graphe43.txt	10600	19300	Graphe302.txt	17000	18500			
Graphe44.txt	9900	17500	Graphe303.txt	17900	20200			
Graphe45.txt	10800	17500	Graphe304.txt	23100	17900			
			Graphe305.txt	21700	21100			
Graphe51.txt	9700	21400						
Graphe52.txt	17800	17900	Graphe401.txt	22500	21300			
Graphe53.txt	9800	18300	Graphe402.txt	21200	21800			
Graphe54.txt	9500	21100	Graphe403.txt	26800	23000			
Graphe55.txt	10200	17700	Graphe404.txt	21300	21100			
			Graphe405.txt	23900	22800			
Graphe61.txt	12400	17900						
Graphe62.txt	8700	16900	Graphe501.txt	22300	21400			
Graphe63.txt	8300		Graphe502.txt	23900	22282200			
3712300								
Graphe64.txt	13400	16900	Graphe503.txt	20500	22200			
Graphe65.txt	18600	19500	Graphe504.txt	21600	22100			
			Graphe505.txt	24700	21700			
Graphe71.txt	9300	16400						
Graphe72.txt	10700	16700	Graphe601.txt	30000	22100			
Graphe73.txt	8900	16000	Graphe602.txt	26000	21800			
Graphe74.txt	9600	16600	Graphe603.txt	24100	22500			
Graphe75.txt	15000	16700	Graphe604.txt	28200	21900			

[Question 24] Nous avons ensuite réalisé une méthode dans `GrapheListe` permettant de générer des graphes. Elle commence par ajouter tous les nœuds au graphe. Ensuite, tant que chaque nœud n'est pas antécédent ou successeur d'un autre nœud, elle sélectionne au hasard un nœud de départ et un nœud d'arrivée qui ne sont pas identiques et pas encore reliés par un arc, elle génère un poids pour l'arc puis elle ajoute l'arc correspondant au graphe.

Ainsi, on obtient des graphes fortement connectés de taille souhaitée (la méthode possède un paramètre `taille` entier permettant de choisir la taille du graphe à générer).

[Question 25]

Voici ci-dessous quelques rendus de graphes avec `GraphViz`, grâce à la classe `MainExperimentation` :



A partir de notre générateur de graphes, nous avons mesuré les temps d'exécution en fonction de la taille des graphes, grâce à la classe `MainExperimentationGenere`. Cette classe permet de choisir le nombre de noeuds des graphes à générer ainsi que le nombre de graphes à créer, puis calcule le temps moyen d'exécution des 2 algorithmes ainsi que le ratio Dijkstra par rapport à Bellman-Ford.

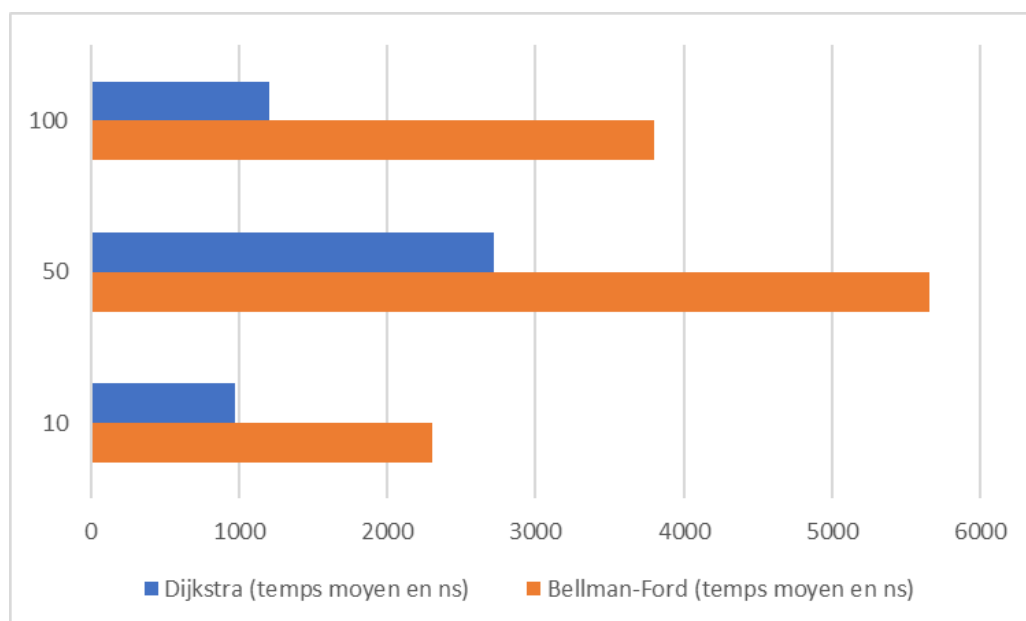
Voici les résultats pour différentes expérimentations :

Conditions	Bellman-Ford (temps moyen en ns)	Dijkstra (temps moyen en ns)	Ratio Dijkstra / Bellman-Ford
10 noeuds, 1000 essais	2303.9	965.6	0.8057325152387392
50 noeuds, 1000 essais	5662.7	2720.9	0.5310066551301743
100 noeuds, 1000 essais	3802.9	1198.4	0.4218661877897214

[Questions 26 et 27] D'après les résultats, l'algorithme de Dijkstra semble être le plus efficace en étant 20% plus rapide que Bellman-Ford pour 10 nœuds, 50% plus rapide pour 50 nœuds et 60% plus rapide pour 100 nœuds.

Le ratio varie donc selon le nombre de nœuds.

[Question 28] Ainsi, nous pouvons conclure que pour optimiser les performances d'un programme qui parcourt des graphes, il faut privilégier l'utilisation de l'algorithme de Dijkstra.



V. Extension : Intelligence Artificielle et labyrinthe

Pour cette dernière partie, on nous demandait d'utiliser ces algorithmes sur des objets de type Labyrinthe, c'est-à-dire rechercher le plus court chemin dans un labyrinthe.

[Question 30] La première solution consistait à créer une méthode `genererGraphe` dans `Labyrinthe`, permettant, à partir du tableau de murs du labyrinthe, de construire un graphe où chaque noeud est une case qui n'est pas un mur, nommée par un couple correspondant aux coordonnées de la case dans le labyrinthe. A partir de labyrinthes générés d'après des fichiers, la recherche du plus court chemin donnait effectivement le plus court chemin pour chacun des labyrinthes.

[Question 31] Nous n'avons pas eu le temps de terminer la deuxième solution, qui consistait à utiliser un pattern `Adapter` pour générer un graphe à partir d'un labyrinthe, en créant une nouvelle classe implémentant directement l'interface `Graphe`.

VI. Bilan de la SAE

Comme toujours, c'est le cheminement intellectuel (sans jeu de mot) qu'il est important de retenir de cette SAE, cheminement qui nous permet de développer notre logique et nos compétences en programmation.

Plus particulièrement, ce projet nous a permis de comprendre le fonctionnement des algorithmes de Bellman-Ford et de Dijkstra et de savoir comment les appliquer en algorithmique puis en programmation, et c'est ce sur quoi nous avons eu le plus de difficultés. De plus, nous connaissons désormais les performances de ces 2 approches, comment les comparer et la manière de les utiliser dans n'importe quel projet faisant intervenir la notion de cheminement.