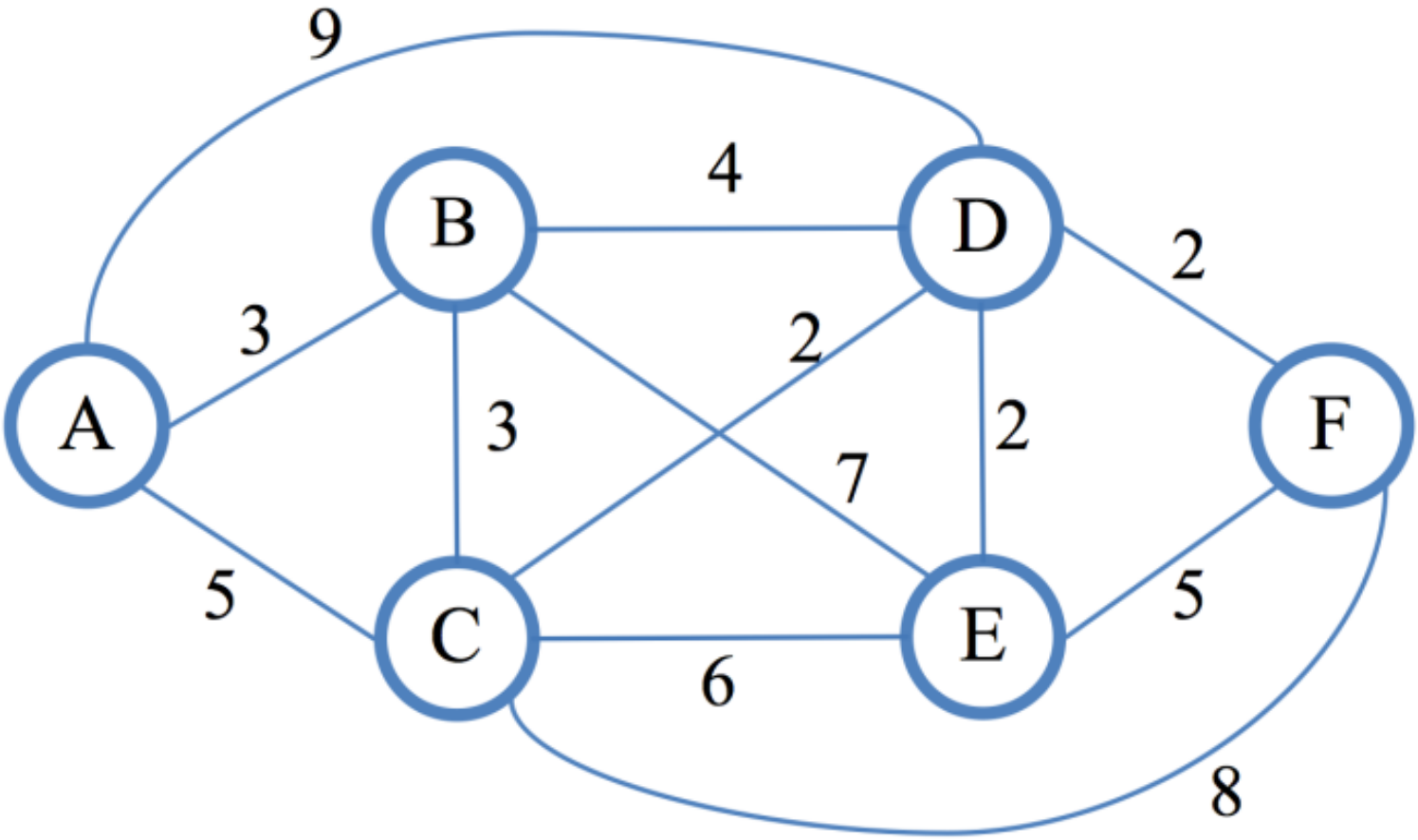
BUT INFORMATIQUE

MAI 2023

SAE 2.2 – Exploration Algorithmique Recherche de plus court chemin dans un graphe



PAR

HAROUNA Laeticia
TOUBHANS Fabien
S2E

Sommaire

Représentation d’un graphe

L'objectif de cette SAE est de mettre en œuvre des solutions algorithmiques pour résoudre le problème de recherche du plus court chemin dans un graphe.

Dans le cadre de la SAE S2.02, il nous a été demandé de mettre en place deux algorithmes de recherche de chemins minimaux dans un graphe en utilisant le langage Java. Ces deux algorithmes sont l’algorithme du point fixe (ou de Bellman-Ford) et l'algorithme de Dijkstra.

Pour commencer, nous avons dû 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.

Nous n'avons pas rencontré de difficultés particulières pour les classes Noeud, Arc et l'interface Graphe. Le seul challenge était de bien saisir le fonctionnement et l'intêret de chacunes pour la suite de la SAE. Nous avons donc pris du temps afin de relire et approfondir notre compréhension du sujet pour bien visualiser l'architecture globale de ce qui nous était demandé.

Représentation d’un graphe

La classe GrapheListe demandait plus de reflexion, nous avons dû reprendre exemple sur des précédents TP et SAE afin d'aborder la conception des méthodes et constructeurs facilement et efficacement.

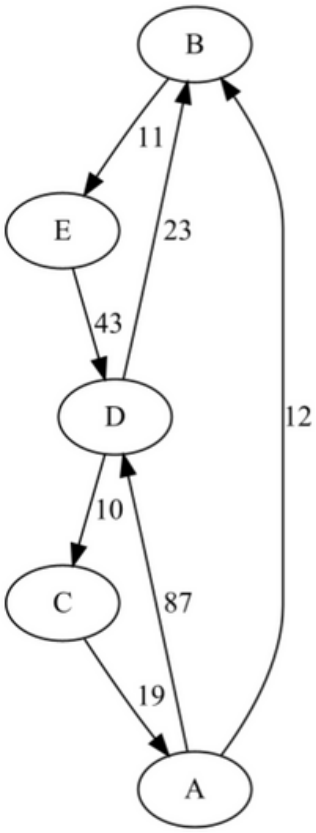
Lors de l'implémentation de la méthode ajouterArc dans notre projet, nous avons rencontré une petite erreur de conception qui nous a fait perdre du temps. La méthode ajouterArc permet de parcourir la liste actuelle de nœuds du Graphe et de vérifier si le nœud de départ et de destination existe à l'aide de deux booléens. Si ces nœuds existent, l'arc est créé entre eux avec un coût prédéfini. En revanche, si les nœuds n'existent pas, ils sont créés, puis l'arc est ajouté entre eux.
L'enregistrement dans ensNom était mal réalisée, ce qui nous a causé des erreurs lors des tests des deux algorithmes puisque nous pensions que cette méthode était pleinement fonctionnelle.

Malgré le petit problème de conception initiale, nous avons pu rectifier l'enregistrement des noms dans ensNom, ce qui nous a permis de poursuivre les tests des deux algorithmes sans aucun souci, sachant que la méthode ajouterArc fonctionnait désormais correctement.

Représentation d'un graphe

En plus de cela, nous nous sommes occupés des méthodes toString et toGraphviz dans la classe GrapheListe, qui permettent l'affichage des graphes. La méthode toString renvoie une représentation sous forme de chaîne de caractères du graphe, tandis que la méthode toGraphviz renvoie une chaîne encodée de manière à pouvoir être interprétée par Graphviz, un outil qui permet d'obtenir une visualisation graphique du graphe.

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



Pour créer ce graphe nous exécutons main puis nous copions le résultat dans un fichier PlantUML sur IntelliJ.

Représentation d’un graphe

Dans la classe GrapheListe, lors de la mise en place du constructeur à partir d'un fichier dans notre projet, nous avons utilisé un BufferedReader pour parcourir l'intégralité du fichier. Nous avons ensuite utilisé la méthode split avec les tabulations comme séparateurs pour diviser chaque élément du fichier.
La gestion de l'ajout des noeuds s'ils existent se fait dans la méthode ajouterArc que nous appelons à chaque ligne du fichier afin d'ajouter les éléments du graphe. La petite difficulté ici était la réflexion afin de déléguer la gestion de l'existence des noeuds dans la méthode ajouterArc.

La méthode convertirMatriceEnListe a constitué une part importante de notre travail lors de la réalisation de cette classe.
La récupération des données de la matrice et leur organisation n'étaient pas particulièrement compliquées. Cependant, la séparation des données des noms de nœuds en début de chaque colonne et de chaque ligne a été un défi qui nous a demandé beaucoup de réflexion et de travail.

Calcul du plus court chemin par point fixe

[Question 14]

```
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
         fsi
     fpour
// 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
```

Calcul du plus court chemin par point fixe

[Question 14]
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

L'algorithme de BellmanFord était simple à implémenter mais l'erreur de la méthode ajouterArc dans GrapheListe a causé des erreurs lors de l'exécution de l'algorithme ce qui nous a induit en erreur sur notre code qui était bon. La recherche du problème est ce qui a mis le plus de temps pour cette partie là.

Calcul du meilleur chemin par Dijkstra

La classe Dijkstra a été l'élément sur lequel nous avons consacré le plus de temps pendant cette SAE.
L'algorithme que nous avons écrit s'est révélé très utile et nous a permis de structurer rapidement la méthode. Cependant, la véritable difficulté que nous avons rencontrée résidait dans les différents tests, qui ont révélé à chaque fois un problème dans l'organisation de la méthode, faussant ainsi les résultats ou empêchant son exécution.
L'initialisation de la liste des noms de nœuds et de leurs données dans la classe "Valeur" était simple. C'est surtout lors de la comparaison des coûts de parcours des nœuds que nous avons perdu du temps. Nous étions quelque peu perdus dans les correspondances avec les parents et les valeurs de la classe "Valeur".
Afin de factoriser le code, nous avons ajouté la méthode "noeudMin" dans la même classe, ce qui nous a permis de récupérer le nœud ayant le coût de parcours le plus faible.
Cela nous a ensuite permis de clarifier notre méthode et d'adopter une approche plus efficace, ce qui a abouti à la version finale de la méthode qui fonctionne correctement et a validé tous nos tests.

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 22] Lors de nos observations, nous avons noté les différences suivantes entre les deux algorithmes :

- L'algorithme de Bellman-Ford calcule le plus court chemin pour tous les sommets du graphe à chaque itération, jusqu'à ce que deux itérations successives produisent les mêmes résultats. Il examine donc toutes les arêtes à chaque fois.
- L'algorithme de Dijkstra quant à lui calcule le plus court chemin pour un nœud spécifique avant de passer au nœud suivant. Il ne regarde pas les nœuds déjà traités.

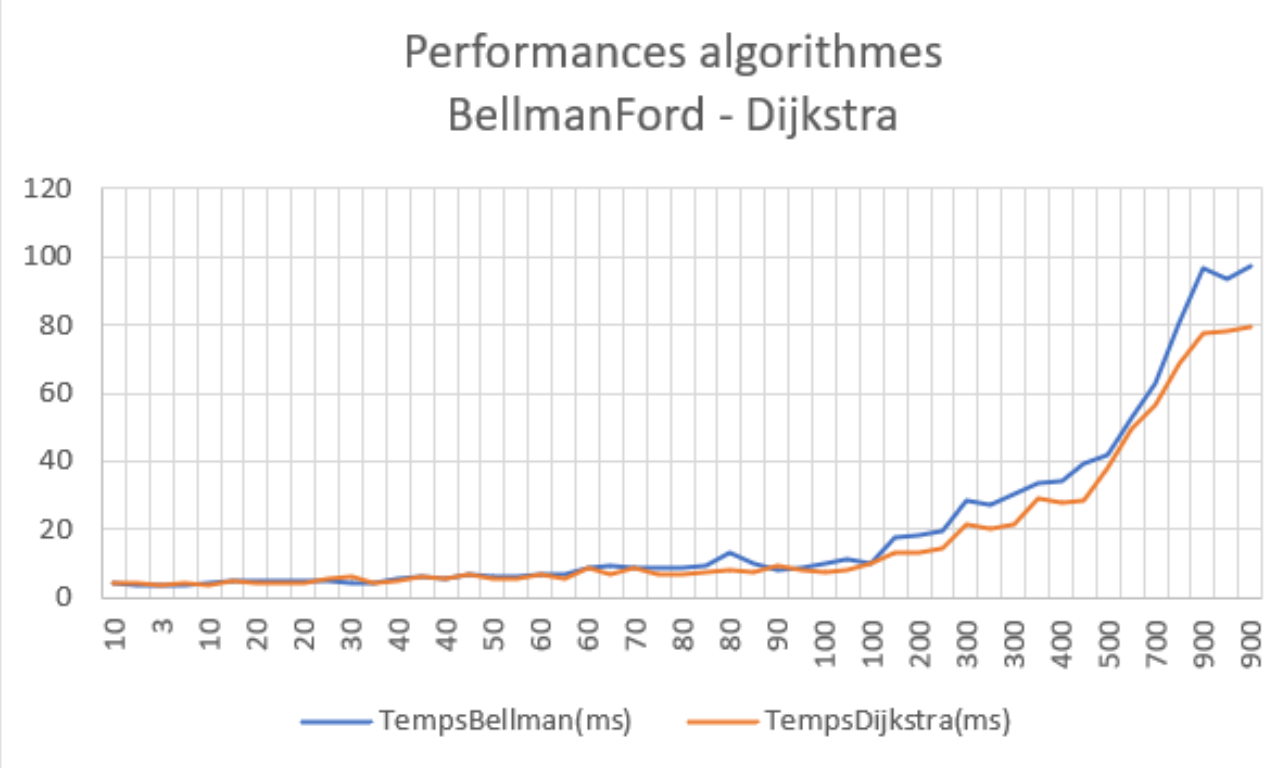
[Question 23] En conclusion, on peut constater que l'algorithme de Bellman-Ford peut nécessiter un nombre variable d'itérations pour trouver le chemin le plus court, tandis que l'algorithme de Dijkstra nécessite autant d'itérations que le nombre de nœuds dans le graphe.

[Question 24]] Nous avons mesuré le temps d'exécution des deux algorithmes sur des graphes de différentes tailles, avec comme point de départ le sommet "1" et comme point d'arrivée le sommet "10".

D'après les résultats obtenus (cf. ComparaisonAlgo), l'algorithme de Dijkstra est plus efficace que celui de Bellman-Ford (en terme 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.

Validation et expérimentation

[Question 24]



De plus, d'après le diagramme on peut observer que Dijkstra est plus rapide que Bellman-Ford dans les graphes denses avec de nombreux nœuds. Mais on remarque que pour des graphes avec un petit nombre de nœuds, il peut arriver que Bellman-Ford soit plus rapide même si c'est très peu visible ici. En conclusion, en terme d'efficacité globale, Dijkstra est préférable pour sa meilleure performance dans la plupart des cas.
(voir resultatsDijkstra.csv et resultatsBellmanFord.csv pour savoir quels fichiers - chemins - temps ont été effectués).

[Question 25]On a ajouté quelques méthodes dans la classe GrapheListe pour pouvoir générer des graphes automatiquement. C'était plus pratique pour créer des graphes qui sont bien connectés et sans doublons d'arcs, en plus de s'assurer que chaque nœud est lié à au moins un autre nœud.

Validation et expérimentation

[**Question 25]**On a commencé par ajouter la méthode arcContient(String noeudP) qui vérifie si un nœud donné est le successeur d'un arc existant dans le graphe. On fait une boucle pour parcourir tous les nœuds du graphe et on regarde leurs arcs pour voir si la destination de l'arc correspond au nœud donné.

Ensuite, on a fait la méthode arcVide(String noeudP) pour vérifier si un nœud a des arcs ou pas. On utilise la méthode suivants(String nom) pour récupérer les arcs suivants du nœud, et puis on regarde si la taille de la liste des arcs est zéro.

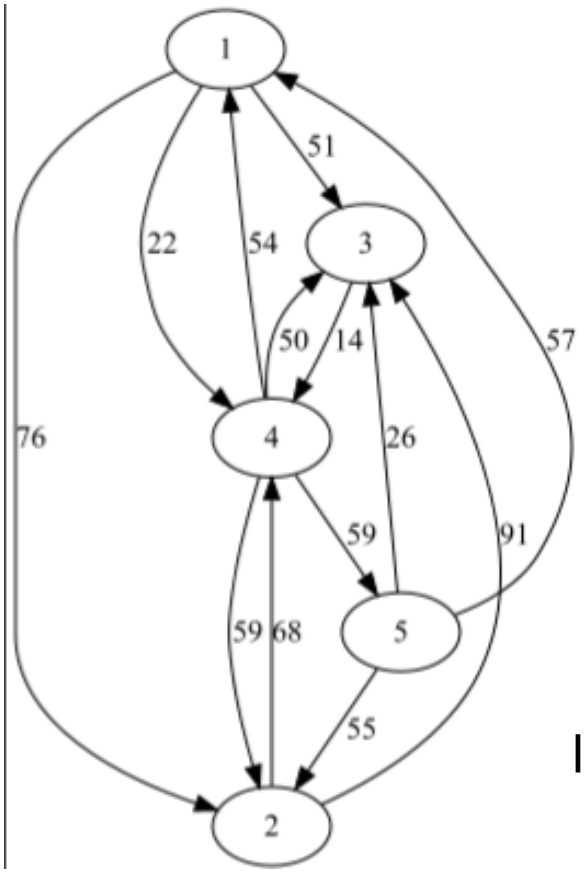
On a aussi ajouté la méthode relie(String nomNoeudDepartP, String nomNoeudArriveeP) pour vérifier si un arc existe déjà entre deux nœuds spécifiés. On fait une boucle pour parcourir tous les nœuds du graphe, on cherche le nœud de départ donné, et puis on regarde si l'arc vers le nœud d'arrivée existe déjà.

Une autre méthode ajoutée est ajouterNoeud(String noeudP) qui vérifie si un nœud avec le même nom existe déjà dans le graphe. Si ce n'est pas le cas, on l'ajoute à la liste des noms de nœuds (ensNom) et on crée un nouvel objet Noeud avec ce nom pour l'ajouter à la liste des nœuds (ensNoeuds).

Enfin, on a fait la méthode genererGraphe(int taille) pour générer un graphe aléatoire de la taille qu'on veut. On commence par ajouter tous les nœuds au graphe en utilisant ajouterNoeud(String noeudP). Ensuite, on choisit aléatoirement un nœud de départ et un nœud d'arrivée qui sont pas les mêmes et pas encore reliés par un arc. Ces méthodes nous ont bien aidées à générer automatiquement des graphes bien connectés, sans doublons d'arcs. On voulait simplifier le processus de génération de graphes et s'assurer que les données générées soient cohérentes, donc on a préféré les utiliser dans notre projet même si on peut très sans passer car elle reprend les méthodes déjà créées.

Validation et expérimentation

[**Question 26]**
Voici ci-dessous deux graphes générés par mainGenerer, le premier est de taille 5 (figure 1) et le deuxième de taille 10 (figure 2) :



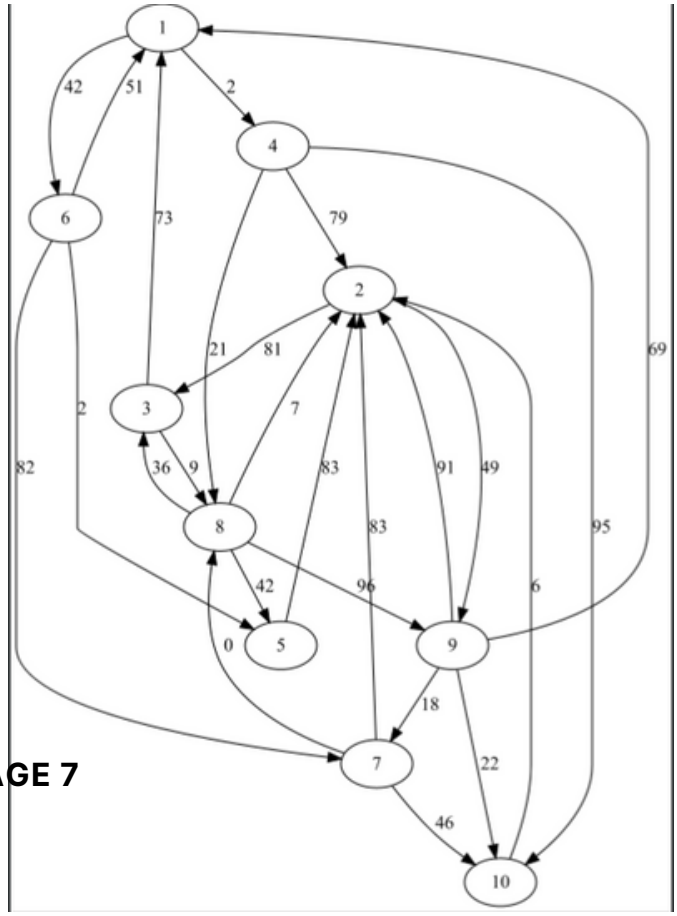


figure 1 - GrapheAuto1

figure 2 - GrapheAuto2

[Question 27] 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 mainGenereGraphe. 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 de chaque algorithme et enfin le temps moyen d'exécution des 2 algorithmes ainsi que le ratio Dijkstra par rapport à Bellman-Ford.

Nous avons testé les performances des différents algorithmes et avons obtenu les résultats ci-contre :

Validation et expérimentation

[Question 27]

Conditions	Bellman-Ford (temps moyen en ns)	Dijkstra (temps moyen en ns)	Ratio Dijkstra / Bellman-Ford
10 noeuds, 1000 essais	26895.8 ns	20315.4 ns	1.0238003821387078
50 noeuds, 1000 essais	118186.0 ns	105292.2 ns	1.005055331817948
100 noeuds, 1000 essais	330340.0 ns	287394.2 ns	0.9276838112918887

Selon les résultats Dijkstra est plus efficace en terme de performance, et cela est cohérent avec ce que l'on a dit précédemment.

[Question 28] Le ratio montre que Dijkstra est plus rapide quasiment à chaque fois.
Cela montre aussi que le ratio varie en fonction du nombre d'essais.

[Question 29] 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.

Bilan :
Nous avons longuement peiné lors des tests de performances des deux différents algorithmes car nous pensions que Dijkstra devait être plus rapide sur des graphes plus grand ce qui était effectivement censé être le cas mais n'était pas le nôtre. Il s'est avéré que les prints à chaque itération de l'algorithme ralentissait énormément celui ci, nous avons donc remplacer les print par un seul print final à la fin de l'algorithme qui indique le nombre d'itération totale.

Extension : Intelligence Artificielle et labyrinthe

[Question 30]La méthode genererGraphe n'était pas dur à réaliser, par contre elle nous a prit beaucoup de temps de réflexion. En effet, nous avons dû nous approprier les méthodes déjà écrites de la classe afin de les utiliser et afin de générer le graphe, juxtaposer le fait de pouvoir faire un mouvement avec la création d'un arc. Une fois que nous avions compris cela la réalisation de la méthode fut assez rapide et nous avons rencontré peu de problèmes, seulement quelques oublis lors des vérifications des conditions de déplacements.

[Question 32] Lorsque nous avons aborder cette question, elle semblait très flou et nous ne voyions pas comment adapter le labyrinthe vers un graphe. En reprenant l'exemple de la méthode genererGraphe nous avons vite compris que le plus facile était d'utiliser toutes les coordonnées comme noeuds possibles afin de générer la liste de noeuds du "graphe" pour implémenter la méthode listeNoeuds(). Pour la méthode suivants c'était plus complexe, le fait de récupérer les coordonnées depuis le nom du noeud n'était pas évident mais une fois que nous y avions pensé et réussi ce fût plus simple. La suite de cette méthode reprend une partie de la méthode genererGraphe où nous testons chaque possibilité de déplacement et en fonction des possibilité de réalisation de chaque nous créons directement les arcs correspondant depuis le noeud entré en paramètre.

Bilan de la SAE

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.