Pellizzari Théo
Planade Remi

Rapport SAE 2.2 : Exploration Algorithmique

I)Présentation de la SAE

II) Représentation d'un graphe :

1)2)3) Au début, on crée une classe Nœud avec comme attributs un nom et une liste d'arc adjacents adj. Le constructeur de Nœud prend en paramètre un String nom. Cette classe possède 5 méthodes :

- Une méthode public boolean equals qui prend un objet en paramètre et qui spécifie que deux nœuds sont égaux si et seulement si leurs noms sont égaux
- Méthode publique void ajouterArc(String destination,doublecout) qui ajoute un arc allant vers le nœud destination avec un coût de cout à la liste adj.
- 2 méthode de guetteur pour le nom et la liste d'arc adj
- Une méthode toString()
-

4) Ensuite, création d'une classe Arc qui prend en attributs une destination de type String et un cout de type double. Le constructeur prend ses 2 attributs en paramètres et cette classe possède deux méthode de guetter pour les attributs.

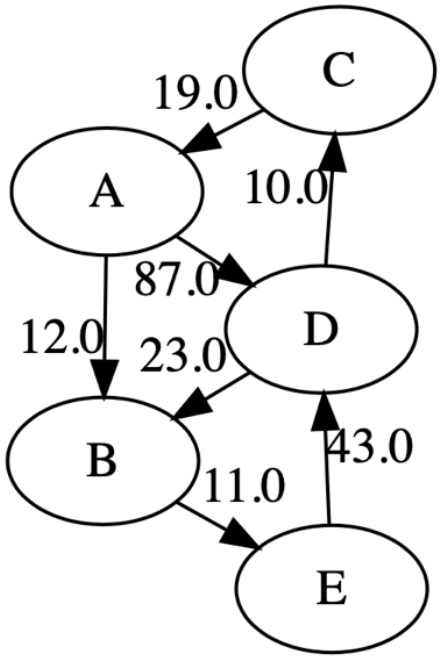
5)Création d'une interface Graphe avec comme méthodes public :

- listeNoeuds() qui renvoie tous les nœuds du graphe et renvoie un résultat du type List<String>
- suivants(String n) qui retourne la liste des arcs partant du noeud n passé en paramètre et renvoie un résultat du type List<Arc>

6)8)9) Création d'une classe GrapheListe qui implémente Graphe, prend 2 attributs : ensNom de type List<String> et ensNoeuds de type List<Nœud>. Il y a deux constructeurs : un constructeur vide et un constructeur qui prend un nom de fichier en paramètre. Il y a une méthode public void ajouterArc(String depart, String destination, double cout) permettant d'ajouter des nœuds et des arcs à un objet de type GrapheListe. Il y a les deux méthodes de Graphe à implémenter, une méthode ToString, une méthode toGraphViz() qui retourne une chaîne représentant le graphe en respectant le format GraphViz et enfin un guetter de ensNoeud.

7) Ecriture de la classe Main pour créer la figure 1 avec graphViz

10)



11) Écriture de la classe Test_Graphe

12) Ecriture d'un constructeur prenant en paramètre le nom du fichier contenant le descriptif du graphe et construisant l'objet graphe correspondant => public GraphListe(String nomdufichier)

III) Calcul du plus court chemin par point fixe

Insertion de la classe Valeur fournie sur Arche

13) fonction pointFixe(g : Graphe InOut, depart : Noeud)

pour i allant de 0 à longueur(listeNoeuds) faire

longueur[i] = null

fin pour

stop=faux

tant que non fixe faire

```
        stop = vrai
        pour i de 0 à longueur(listeNoeuds) faire
                pour chaque arc(u,v) du graphe faire
                        si longueur[v] > longueur[u] + cout(u,v) alors
                                longueur[v] = longueur[u] + cout(u,v)
                                precedent[v] = u
                                fixe = faux
                        fin si
        fin pour
    fin tant que
  retourner longueur[], precedent[]
fin
```

Lexique :

g : Graphe, graphe sans cycle de poids négatif

depart : Noeud, noeud de depart de g

listeNoeuds : List<Noeud>, liste des noeuds du graphe

longueur[i] : int; predecesseur

stop : boolean; booléen qui va changer de valeur lorsqu'il y aura un point fixe

u : String; départ de l'arc

v: String; destination de l'arc

longueur[u] : Int; longueur du plus court chemin de depart à u

longueur[v] : Int; longueur du plus court chemin de depart à v

14) En utilisant la classe Valeur fournie et notre algorithme, nous avons implémenté l'algorithme du point fixe dans une classe nommée BellmanFord en programmant la méthode Valeur resoudre(Graphe g, String départ). Cette méthode prend un graphe g et un String représentant le nœud de départ en paramètre et retourne un objet valeur correctement construit contenant les distances et les parents de chaque nœud.

15) Création de la classe MainBellmanFord

16) Création de la classe TestBellmanFord

17) Pour calculer le meilleur chemin, on a écrit une méthode calculerChemin(String destination) qui renvoie une Liste de nœud dans la classe Valeur à partir d'un point de départ donné lors de la construction de l'objet Valeur vers un nœud de destination donné en paramètre

IV) Calcul du meilleur chemin par Dijkstra

Algo pseudo-code

Entrées :
G un graphe orient´e avec une pondération (poids) positive des arcs A un sommet (de´part) de G

Début
Q <- {} // utilisation d'une liste de noeuds à traiter Pour chaque sommet v de G faire

v.distance <- Infini
v.parent <- Inde´fini
Q <- Q U {v} // ajouter le sommet v `a la liste Q

Fin Pour

A.distance <- 0
Tant que Q est un ensemble non vide faire
  u <- un sommet de Q telle que u.distance est minimale
  Q <- Q \ {u} // enlever le sommet u de la liste Q
  Pour chaque sommet v de Q tel que l'arc (u,v) existe faire
    D <- u.distance + poids(u,v)
    Si D < v.distance
      Alors v.distance <- D
        v.parent <- u
    Fin Si
  Fin Pour
Fin Tant que
Fin

18) Dans une classe Dijkstra, on recopie l'algorithme ci-dessus en commentaire puis on traduit ces lignes en java pour écrire la méthode Valeur resoudre(Graphe g, String départ) qui prend en paramètre le nom du nœud de départ pour calculer les plus courts chemins vers les autres nœuds du graphe avec l'algorithme de Dijkstra.

19) Ecriture des tests jUnit pour cette classe Dijkstra

20) Écriture d'un programme principal MainDijkstra permettant :

• la lecture des graphes à partir de fichiers texte ;
• le calcul des chemins les plus courts pour des nœuds donnés et l'affichage des chemins pour des nœuds données.

V) Validation et expérimentation

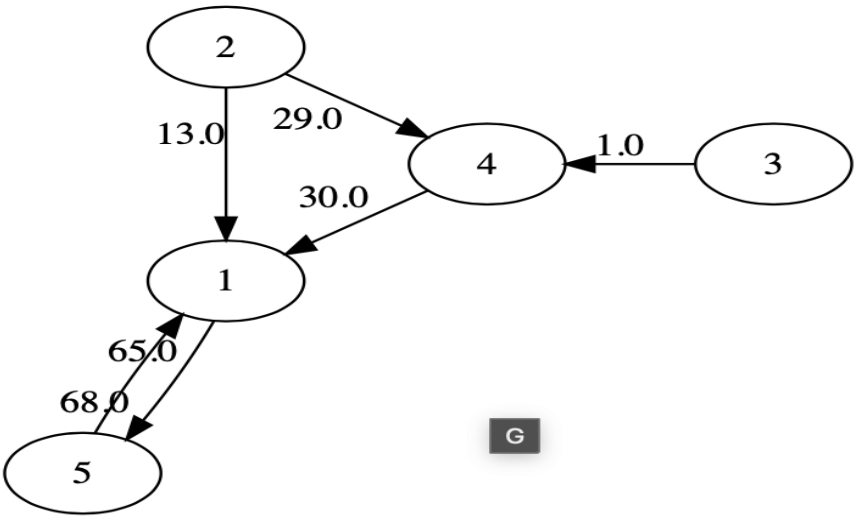
21) On a rajouter un System.out.println(valeur) Afficher le contenu de l'objet valeur après chaque itération de chacun des algorithmes. Ainsi, on peut différencier ces deux algorithmes : Contrairement à l'algorithme de Dijkstra, l'algorithme de Bellman-Ford autorise la présence de certains arcs de poids négatif et permet de détecter l'existence d'un circuit absorbant, c'est-à-dire de poids total strictement négatif, accessible depuis le sommet source.
22) L'algorithme de Bellman-Ford permet de calculer le plus court chemin depuis un sommet source donné dans un graphe orienté pondéré. L'algorithme de Dijkstra sert à trouver le meilleur chemin. Plus précisément, il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs. On peut l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

23)

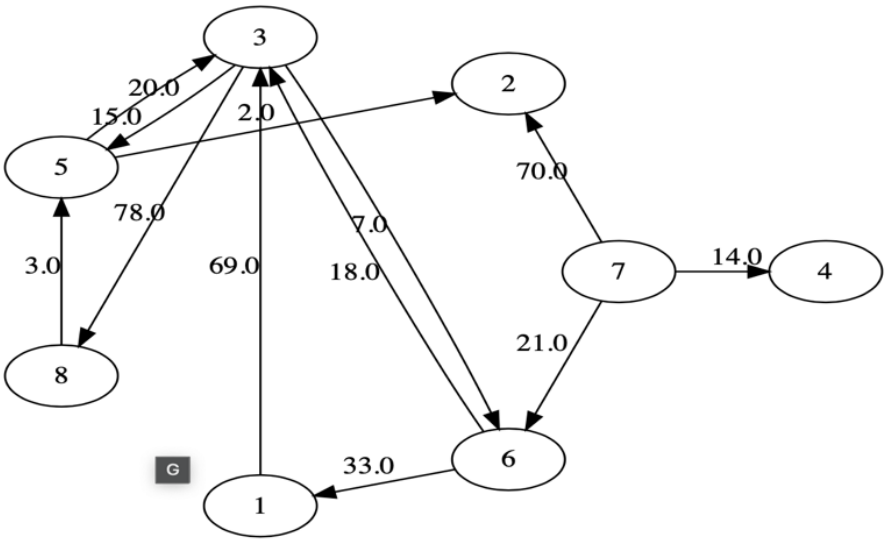
	A	B	C	D
1		Bellman	Dijkstra	
2	Graphe5.txt	19	18	
3	Graphe11.txt	20	18	
4	Graphe21.txt	23	21	
5	Graphe31.txt	28	22	
6	Graphe41.txt	30	28	
7	Graphe51.txt	30	29	
8	Graphe61.txt	47	29	
9	Graphe71.txt	53	35	
10	Graphe81.txt	60	31	
11	Graphe91.txt	63	30	
12	Graphe505.t	1336	115	
13	Graphe705.t	3488	200	
14	Graphe905.t	6696	310	
15				
16			Temps en millisecondes	
17				
18				

Nous avons testé l'efficacité de nos 2 algorithme de Bellman-Ford et Dijkstra grâce à un tableau excel qui reprend les résultat testés dans chaque Main(), le principe est d'initialiser un juste avant d'exécuter la méthode résoudre() et un timer de fin juste après. Le résultat sera la différence entre le timer de fin et de début. On a comparé quelques résultats de temps à partir de fichier Txt pour la création de graphe et on peut remarquer que l'algorithme Dijkstra a un temps d'exécution moins élevé que Bellman-Ford lorsqu'on crée le graphe à partir d'un fichier txt. Cela s'explique par le principe de Bellman-Ford qui nécessite de passer obligatoirement par tous les nœuds pour trouver un chemin minimal alors que dijkstra va simplement éliminer le noeud minimal à chaque itération pour trouver le plus court chemin.
24) Ajout d'un constructeur GraphListe(int taille, String depart, String arrivee) pour générer des graphes automatiquement d'une taille donnée (par exemple 1000 nœuds) en choisissant un départ et une arrivée. Les graphes sont fortement connectés en ajoutant des connections entre chaque nœud et le nœud suivant créé pour avoir au moins un chemin qui relie tous les nœuds.

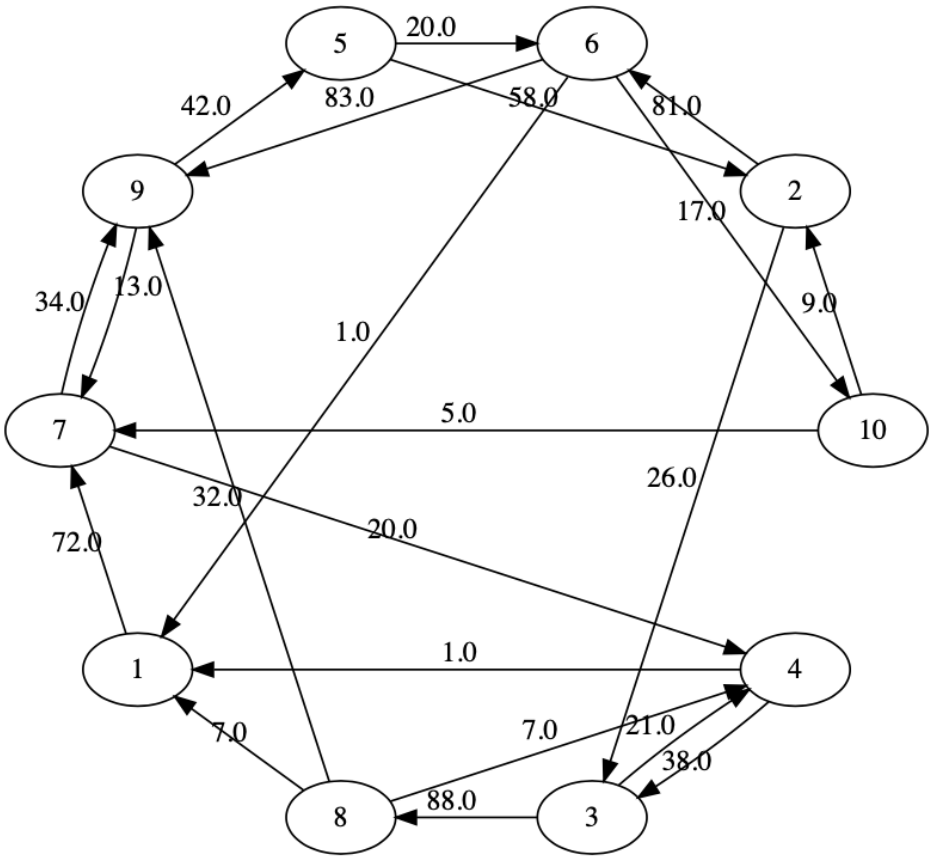
25) rendu GraphViz de quelques graphes générés :



Taille 5



Taille 8



Taille 10

26) Création d'une classe MainGraphAuto pour pouvoir tester quel algorithme est le plus efficace par rapport à un graphe généré automatiquement avec des tailles différentes.
-utilisation d'une ArrayList pour stocker les tailles de graphes, initialisation de moyennes, construction du graphe avec les index de la liste, timer sur la fonction résoudre que l'on met dans une boucle for pour pouvoir faire une moyenne(On a pris 10 comme exemple). A la fin, on affiche la moyenne pour les deux algorithmes que l'on va diviser par 10 car on l'a réalisé dans une boucle de 10 fois. On constate alors que l'algorithme de Dijkstra reste plus efficace en matière d'exécution par rapport a Bellman-Ford pour les valeurs de taille de graphe testés, voici le résultat de l'exécution du main() :

Taille| BellmanFord| Dijkstra
10| 369692ns | 153116ns

Taille| BellmanFord| Dijkstra
20| 227275ns | 123987ns

Taille| BellmanFord| Dijkstra
50| 846837ns | 307230ns

Taille| BellmanFord| Dijkstra
100| 4015104ns | 958312ns

Taille| BellmanFord| Dijkstra
500| 178509118ns | 21339446ns

Taille| BellmanFord| Dijkstra
1000| 518461813ns | 69150328ns

Taille| BellmanFord| Dijkstra
2000| 3553851864ns | 371042480ns

Taille| BellmanFord| Dijkstra
5000| 18796449055ns | 2861493735ns

27) Plus le nombre de nœuds augmente, plus le ratio de performance de l'algorithme de Dijkstra va augmenter car par exemple :
pour 10 nœuds : Dijkstra 2 fois plus rapide
pour 100 nœuds : Dijkstra 4 fois plus rapide
pour 1000 nœuds : Dijkstra 6 fois plus rapide

28)
L'algorithme de Bellman Ford fonctionne lorsqu'il y a un bord de poids négatif, il détecte également le cycle de poids négatif. Le résultat contient les sommets qui contiennent les informations sur les autres sommets auxquels ils sont connectés. Il peut facilement être mis en œuvre de manière distribuée et il prend plus de temps que l'algorithme de Dijkstra.

L'algorithme de Dijkstra ne fonctionne pas lorsqu'il y a un avantage de poids négatif. Le résultat contient les sommets contenant des informations complètes sur le réseau, pas seulement les sommets auxquels ils sont connectés. Il ne peut pas être mis en œuvre facilement de manière distribuée et prend moins de temps que l'algorithme de Bellman-Ford.

VI) Extension : Intelligence Artificielle et labyrinthe

Insertion de la classe Labyrinthe donnée pour pouvoir générer des graphes

29) Ajout de la méthode genererGraphe qui retourne le graphe associé au labyrinthe.

• les nœuds sont nommées selon les coordonnées de la case (par exemple ≪ (1,1) ≫)

• les arcs correspondent aux différents déplacements possibles et mènent à la case d'arrivée du déplacement (on ne stockera pas les actions associées aux arcs, mais cela pourrait constituer une autre extension à votre classe Graphe).

30) Création de la classe MainLaby => Graphe généré à partir du fichier laby1.txt

On essaie de trouver un des chemins les plus petits reliant 1.1 à 8.1.
On peut voir les chemins les plus courts sur le graphe ci-dessus, ils ont une longueur de 9.
Lorsque l'on utilise la méthode calculer chemin, on nous donne ce chemin :
[1.1, 2.1, 3.1, 4.1, 4.2, 5.2, 6.2, 7.2, 8.2, 8.1]
Il correspond bien à l'un des chemins les plus courts.

CONCLUSION SAE :

Cette SAE nous aura permis d'acquérir des compétences supplémentaires :

Analyser un problème avec plus de méthode, comparer des algorithmes différents pour voir leur efficacité et leurs différences et enfin pouvoir implémenter des méthodes, utiliser et tester des outils mathématiques pour l'informatique.

Les difficultés que nous avons rencontrées ce sont situés au niveau de l'implémentation de la classe Dijkstra, la génération de graphes automatiques car le chemin utilisé devait être aléatoire, la comparaison d'efficacité des deux algorithmes car il faut trouver des résultats cohérents et enfin l'acheminement à faire dans la classe labyrinthe.

Dans l'ensemble, nous sommes plutôt satisfaits du travail fourni car cette étude nous aura permis de mieux comprendre l'utilisations des algorithmes de Bellman-Ford et Dijkstra et surtout quel rôle chacun joue-t-il dans la résolution du plus court chemin.