El Bouroumi Anas
Pinchon Théo
S2D

RAPPORT:

1) Représentation d'un graphe:

Pour commencer nous devions implémenter un graphe en java, un graphe est un objet informatique qui est représenté comme un ensemble de nœud (qui est un nom) et une liste de ses nœuds adjacents.

Dans cette première partie nous devions alors pouvoir construire un graphe et surtout l'afficher, l'afficher en deux formats.

-Le première étant une format basique,
exemple: A -> B(12) D(87) B -> E(11) C -> A(19) D -> B(23) C(10) E -> D(43)

-Le deuxième format étant en format graphiz,
exemple : digraph G { A -> B [label = 12] A -> D [label = 87] B -> E [label = 11] C -> A [label = 19] D -> B [label = 23] D -> C [label = 10] E -> D [label = 43] }

Pour ce faire nous avons développer plusieurs classe, une classe Nœud, une classe Arc, une interface Graphe pour ensuite faire une classe GrapheListe.

L'implémentation de ces classes n'était pas très compliqué, il fallait pour terminer écrire un constructeur dans GraphListe qui permettrait de lancer un Graphe à partir d'un fichier texte fournis.

2) Point Fixe:

L'algorithme du point fixe ou encore appelé algorithme de Bellman-Ford consiste à initier les valeurs de tous les nœuds à +∞, puis, pour chaque nœud X, à estimer la valeur de la fonction L(N) de ses nœuds adjacents N à partir de L(X) et du coût de l'arc.

Nous avons alors l'algorithme suivant:
fonction Bellman-Ford(G,d)

```
//initialisation:

    pour n dans G faire:
    L(n)<-+∞
    parent(n) = null
  l(d)<-0
  //boucle principale :
  continuer <- vrai
  tant que continuer, faire :
    continuer <- faux
    pour chaque arc(u,v,poids) de G faire :
      Si L(v) > L (u) + poids alors
        l(v)<-l(u)+poids
        parent(v)<-u
        continuer <-vrai
      fsi
    fpour
  ftantque
```

Lexique :

G : Graphe
continuer : booleen
arc: Arc
u : noeud
v: noeud
poids : double
d: depart : Chaine de caractère

Grâce à la classe Valeur, nous avons alors implémenté cet algorithme en java dans la méthode **public Valeur resoudre(Graphe g, String depart)** , ce qui nous a permis de pouvoir utiliser la technique du point fixe.

Nous avons alors réalisé quelques tests sur cette méthode, comme par exemple si il est fonctionnel, si la valeur de départ est la bonne et si les calculs de parents sont les bons.
Après avoir réaliser tout cela nous devions afficher le chemin le plus court qui mène un nœud donné, nous avons alors écris la méthode
public List calculerChemin(String destination).
Cette méthode permet de retourner une liste de nœud correspondant au chemin menant au nœud passé en paramètre.

3) L'algorithme de Dijkstra:

L'algorithme de Dijkstra permet de construire le plus court chemin en utilisant la même fonction L(X) que celle vue dans l'algorithme du point fixe. L'algorithme de Dijkstra consiste `a partir d'un nœud de départ et à calculer de proche en proche la

valeur de L(X), la distance du plus court chemin qui passe par X, et le parent de X dans ce chemin.
Cependant, au lieu d'appliquer un calcul sur chaque nœud jusqu'à atteindre un point fixe, l'algorithme de Dijkstra consiste à choisir un ordre de calcul adapté pour propager les valeurs calculées sans avoir besoin de mettre à jour des valeurs une fois qu'elles ont été validées.

Nous avons l'algorithme suivant:

```
fonction Dijkstra:
Début
Q <- {} // utilisation d'une liste de nœuds à traiter
Pour chaque sommet v de G faire
        v.distance <- Infini
        v.parent <- Indéfini
        Q <- Q U {v} // ajouter le sommet v à 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
```

```
Lexique :
        G un graphe orientée avec une pondération (poids) positive des arcs
        A un sommet (départ) de G
```

Nous avons alors implémenté cet algorithme en java dans la méthode:
public Valeur resoudre(Graphe g, String depart).
Comme pour la méthode du point fixe nous avons réalisé des tests unitaires et écris un main qui pouvait utilisé cet algorithme pour calculer le chemin le plus court pour un nœud donné.

4) Validation et expérimentation:

21) La différence entre ces deux algorithme est très simple,
Le principe de l'algorithme de Bellman-Ford est d'initier les valeurs de tous les nœuds a +∞ et de mettre la valeur de départ à 0, puis pour chaque nœud d'estimer L(N) de ses nœuds adjacents N à partir de L(X) et de répéter cela jusqu'à trouver le point fixe (quand aucun L(N) n'évolue).

L'algorithme de Dijkstra consiste aussi à mettre tout ses nœuds a +∞, et sa valeur de départ à 0 et à calculer de proche en proche la valeur L(N) des nœuds N.
La subtilité de cette algorithme consiste à considérer la liste de tous les nœuds du graphe et de choisir comme nœud à développer le nœud X dans cette liste avec la plus petite distance L(X). Une fois qu'un nœud à été changer sa valeur ne bouge plus.

22) On remarque qu'avec les deux algorithme on obtient le même résultat, on comprend donc que les deux algorithmes nous renvoient bel et bien le chemin le plus court du graphe, il nous reste alors à savoir qu'elle algorithme est le plus optimal et le plus rapide.

23) Tableau synthétique représentant la comparaison des deux algorithmes:
(En Nano Seconde)

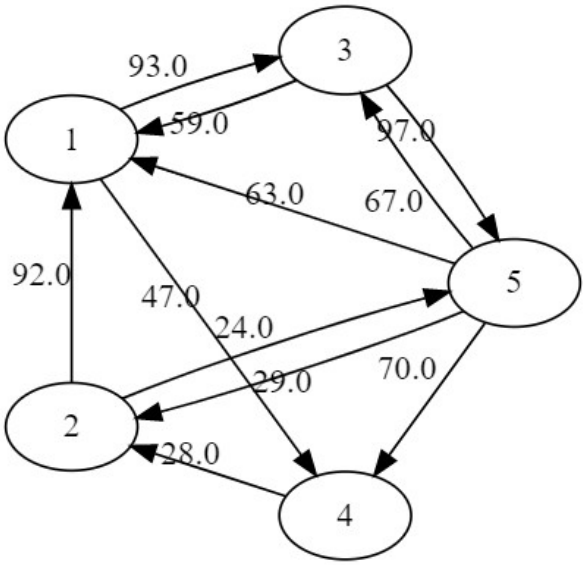
nom du fichier	Bellman-Ford	Dijkstra	nombre nœud	Temps / nœud 1	Temps / nœud 2
Graphe1.txt	6999900	1001200	10	699990	100120
Graphe101.txt	71000000	8999900	100	710000	89999
Graphe102.txt	16000700	22000200	100	160007	220002
Graphe103.txt	10000300	5000200	100	100003	50002
Graphe104.txt	9999600	4000200	100	99996	40002
Graphe105.txt	84000300	3999600	100	840003	39996
Graphe11.txt	0	0	10	0	0
Graphe12.txt	0	0	10	0	0
Graphe13.txt	0	1000100	10	0	100010
Graphe14.txt	0	0	10	0	0
Graphe15.txt	0	0	10	0	0
Graphe2.txt	0	0	10	0	0
Graphe201.txt	251999200	19001900	200	1259996	95009,5
Graphe202.txt	19000400	110000200	200	95002	550001
Graphe203.txt	150999400	10000700	200	754997	50003,5
Graphe204.txt	12000600	19999400	200	60003	99997
Graphe205.txt	10999900	329999900	200	54999,5	1649999,5
Graphe21.txt	0	0	20	0	0
Graphe22.txt	0	999600	20	0	49980

Ce tableau représente la comparaison entre l'algorithme de Bellman-Ford et de Dijkstra (voir dans le dépôt Git pour avoir le tableau entier, tableau.xlsx).
On peut alors remarquer grâce à ce tableau que l'algorithme de Dijkstra est plus efficace que celui du point fixe, car par exemple dans le graphe 1, le temps émis par l'algorithme point fixe divisé par le nombre de nœud est de 699 990 tandis que celui de l'algorithme Dijkstra est de 100 120 ce qui est une grande différence.

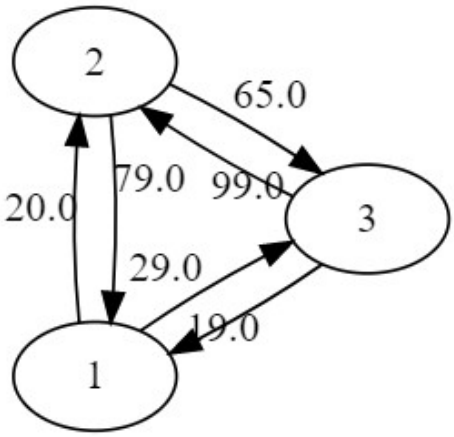
Nous avons d'ailleurs calculé une moyenne qui est de (Gauche= Point fixe , Droite= Dijkstra):

moyenne	47600091,5789474	40903981,0526316

25) Un graphe de taille 5 générer automatiquement:



Un graphe générer automatiquement de taille 3:



26) Nous allons alors comparer les deux algorithme par rapport à des graphes avec énormément de nœuds, on a alors pour un graphe qui possède **1000 nœuds:**
-Bellman-Ford: 66 821 100
-Dijkstra: 109 706 200

Pour un graphe qui possède **9000 nœuds :**
-Bellman-Ford: 2 990 394 800
-Dijkstra: 5 979 593 900

on peut donc remarqué que plus le graphe est grand, plus l'algorithme le plus efficace est celui du point fixe.

27) Nous allons alors calculer le ratio de performance, on a alors:
5 979 593 900 / 2 990 394 800 =1,999, on comprend donc que l'algorithme du point fixe est approximativement 2 fois plus efficace que celui de Dijkstra pour les grands graphe.

28) Nous en tirons alors en conclusion que chaque algorithme est utile, mais cela dépend du cas d'utilisation, par exemple si l'ont veut déterminer le chemin le plus court dans un graphe avec pas beaucoup de nœud la méthode la plus rapide est celle de l'algorithme Dijkstra cependant si le nombre de nœud devient de plus en plus excessif l'algorithme du point fixe est alors préférable.
Dans le cas d'une intelligence artificielle possédant un grand nombre de possibilité et donc de nœud, l'algorithme de Bellman-Ford est le plus adapté.

4) Extension : Intelligence Artificielle et labyrinthe :

Dans cette partie, on propose d'appliquer ces algorithmes à la recherche de chemin dans des labyrinthes.
Le but était alors d'utiliser le travail que nous avions fournis dans les parties au dessus pour pouvoir, trouver dans un labyrinthe le chemin le plus court, nous devions alors par exemple implémenter une méthode genererGraphe qui retourne le graphe associé au labyrinthe, pour ensuite faire les 2 solutions proposés pour trouver le chemin le plus court. (VOIR LE CODE)

Pour conclure, cette SAE nous a prouvé une fois de plus que la façon de s'attaquer à un problème tel que le choix d'un chemin critique dans un graphe est d'une grande importance dans le bon et rapide déroulement d'un programme.
En général cette SAE a été très enrichissante et intéressante, elle nous a montré que l'informatique est une science qui repose sur d'autre science comme les mathématiques avant tout.
L'intelligence artificielle étant de plus en plus présente dans notre vie de tous les jours, la portée de cette SAE est à but éducatif mais également prévisionnel en vue des futurs métier de ce domaine.