Master 1 IWOCS

Compte rendu du TP 3
Plus courts chemins

Travail réalisé par :

Fouad TEKFA

Dirigé par :

Éric SANLAVILLE

Stefan BALEV

Louise PENZ

Table des matières

Introduction :

La théorie des graphes est une discipline mathématique et informatique qui consiste à formaliser plusieurs problèmes sous forme d'un graphe avec de différents paramètres selon le problème à résoudre.

Elle est utilisée dans plusieurs domaines comme les Sciences techniques (biologie, chimie, physique, informatique,..), Sciences humaines(géographie, histoire,...) ,monde économique (finance, logistique,…)….

Dans le cadre de notre TP nous allons implémenter l'algorithme Dijkstra qui a pour but l'optimisation d'un parcours entre deux points et la recherche du plus court chemin .

Rappel du sujet :

Dans ce TP il nous est demandé d'implémenter un programme JAVA permettant de donner le chemin le plus court à partir d'une source et tous les autres sommets en réalisant une version naïve de l'algorithme Dijkstra vu en cours et de lancer l'algorithme Dijkstra de GraphStream puis de faire une comparaison de temps d'exécution entre les deux versions en partant sur le même graphe qu'on va générer aléatoirement en utilisant RandomGenerator de GraphStream.

RandomGenerator (Générateurs de graphes) :

Ce générateur nous permet de créer des graphes aléatoires de n'importe quelle taille en commençant par créer un premier nœud en faisant appel à la fonction begin() puis on fait appel a la fonction nextEvents() qui ajoutera un nouveau nœud à chaque fois que on l'appel et connectera ce nœud aux autres nœud de manière aléatoire.

Ce générateur nous génère des nœuds qui ont des degrés en moyenne avec des connexions aléatoires qui utilise la loi de poisson.

Ce générateur en mesure d'ajouter des valeurs au hasard sur des attributs arbitraires sur les nœuds et les arêtes puis choisir leur direction au hasard.

Les arêtes par défaut ne sont pas orientées au moment où on demande l'orientation la direction sera choisie aléatoirement.

Après avoir lu la documentation et la javadoc de ce générateur j'ai crée une fonction GeneratorGraph qui nous génère des graphes en fonction des différents paramètres attribués.

Graph: le graphe qui sera attribué au générateur de type Graph.
NbNoeuds: nombre de nœuds du graphe de type entier.
averageDegree: le degré moyen du graphe de type entier.
directed: de type Boolean pour générer un graphe oriente ou pas.
poidsMax: la distance maximale à générer avec random de type entier.
display : de type Boolean pour afficher ou pas notre graphe.

```
private static void GeneratorGraph(Graph graph, int NbNoueds , int averageDegree, Boolean directed,   int poidsMax ,Boolean display ) {
    //généré un graphe d'un degree donner en paramètre
    RandomGenerator generator =new RandomGenerator(averageDegree);
    //graphe orienté ou pas aléatoirement
    generator.setDirectedEdges(directed, randomly: true);
    //récupérer tous les événements de notre graphe
    generator.addSink(graph);
    generator.addEdgeAttribute( name: "poids");
    //crrer un Node
    generator.begin();
    for(int i=0; i<NbNoueds; i++)
        //il rajoute un autre node a chaque fois puis il fait une
        generator.nextEvents();

    generator.end();

    graph.edges().forEach(e->{
        Random random =new Random();
        //Attribution des poids aléatoire pour chaque arête entre 0 et pointMax donner en paramètre
        e.setAttribute( s: "poids", random.nextInt(poidsMax));
        // affichage de distance entre les noeuds
        e.setAttribute( s: "ui.label",  ...objects: "" + e.getAttribute( s: "poids"));
        //style CSS pour les arêtes
        e.setAttribute( s: "ui.style",  ...objects: "text-size: 20px;");

    });
     //affichage des id des noeuds avec le style CSS
    graph.nodes().forEach(n -> {
        n.setAttribute( s: "label", n.getId());
        n.setAttribute( s: "ui.style", ...objects: "fill-color:Yellow; size:22; text-size:18px; ");

    });
    if (display==true){
    graph.display();}
    System.out.println("le graphe "+graph.getId()+" est générer ");

}
```

Figure 1: Pseudo code du générateur

En premier lieu j'ai passé un degré en paramètre pour générer le graphe qui sera à son tour orienté ou pas aléatoirement en utilisant la fontion SetDirectedEdges() puis récupérer tous les éventements en utilisant la fonction prédéfinie addSkin() qui prend le graphe en paramètre.

Ensuite on crée le premier nœud en utilisant la fonction begin() puis j'ai fais une boucle qui part du premier nœuds jusqu'à la taille du graphe (NbNoeuds) en rajoutant des nœuds (nextEvents()).

Puis pour chaque arête on attribue des poids aléatoires jusqu'au poidsMax passé en paramètres.

Enfin pour chaque nœuds j'ai affiché son identifiant, et pour donner du style au elements du graphe j'ai opté pour ui.style.

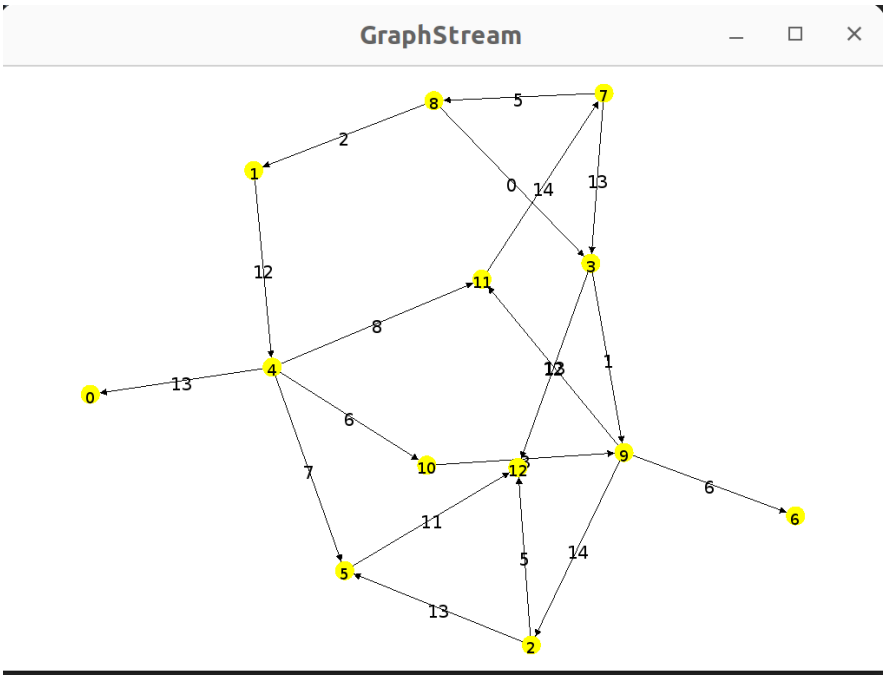

Figure 2: Execution de la fonction GeneratorGraph

Dijkstra

l'algorithme est divisé en trois phases comme vu en cours :

Phase de l'initialisation:

on initialise le nœud source à 0 et mettre la distance de ses nœuds voisins à l'infini finalement on ajoute le nœud source à la file de priorité.

Phase de l'extraction du minimum:

ici on extrait le minimum de la file de priorité.

Mise à jours des distances:

on mit à jours les distances à partir du nœud source vers ses voisins en calculant la distance puis elle renvoie le nœud qui a la distance la plus petite par rapport à la distance.
En dernier lieu on on groupe le tout dans la méthode DijkstraNaive().

Dijkstra GraphStream:

la classe de dijsktra dans GraphStream calcule le plus court chemin à partir d'un nœud source vers un autre nœud cela ce fait par:
Définition en instance à l'initialisation.
Initialisation de l'algorithme avec la fonction init().
Calcul du plus court chemin avec la méthode compute().
Récupération des chemins les plus courts pour des destination données avec la méthode getShortestPath() pour cette partie j'ai utilisé un Boolean pour avoir le choix d'afficher ou pas le plus court chemin entre la nœud de source et tous les autres nœud.
Enfin la création de l'instance Dijkstra avec le constructeur Dijkstra qui prend trois éléments en paramètres.

Tests et résultats:

Pour tout ce qui est test, c'est fait dans le «main».

Test 01:

J'ai effectué un test de calcule de la plus courte distance avec les deux algorithmes partant de même graphe et même nœud de source afin de comparer les résultat et d'être sûr que la versions native est fonctionnelle.

```
/home/tekfa/.jdks/corretto-1.8.0_352/bin/java ...
le plus court chemins entre le Node 0 et les autres
Noeud de source 0
========Dijkstra Naive =======
la distance la plus courte entre0 et 0 :  ====> 0
la distance la plus courte entre0 et 1 :  ====> 22
la distance la plus courte entre0 et 2 :  ====> 13
la distance la plus courte entre0 et3 :  ====> Infinity
la distance la plus courte entre0 et 4 :  ====> 16
la distance la plus courte entre0 et 5 :  ====> 2
la distance la plus courte entre0 et6 :  ====> Infinity
la distance la plus courte entre0 et 7 :  ====> 8
la distance la plus courte entre0 et 8 :  ====> 8
la distance la plus courte entre0 et 9 :  ====> 33
la distance la plus courte entre0 et 10 :  ====> 3
la distance la plus courte entre0 et11 :  ====> Infinity
la distance la plus courte entre0 et 12 :  ====> 8
 de temps d'exécution  =====>0.932755
```

Figure 3: Résultat d'exécution affichant la plus courte distance(Dijkstra naïf)

```
========Dijkstra graphstream===============
 la distance la plus courte entre 0 et 0 : ====>   0,00
 la distance la plus courte entre 0 et 1 : ====>  22,00
 la distance la plus courte entre 0 et 2 : ====>  13,00
 la distance la plus courte entre 0 et 3 : ====> Infinity
 la distance la plus courte entre 0 et 4 : ====>  16,00
 la distance la plus courte entre 0 et 5 : ====>   2,00
 la distance la plus courte entre 0 et 6 : ====> Infinity
 la distance la plus courte entre 0 et 7 : ====>   8,00
 la distance la plus courte entre 0 et 8 : ====>   8,00
 la distance la plus courte entre 0 et 9 : ====>  33,00
 la distance la plus courte entre 0 et 10 : ====>   3,00
 la distance la plus courte entre 0 et 11 : ====> Infinity
 la distance la plus courte entre 0 et 12 : ====>   8,00
de temps d'exécution 18.821706
```

Figure 4: Résultat d'exécution affichant la plus courte distance (Dijkstra GraphStream)

Test 02:

j’ai fait plusieurs test de temps d’exécution en fonction du degré du graphe en partant sur la même taille de graphe.

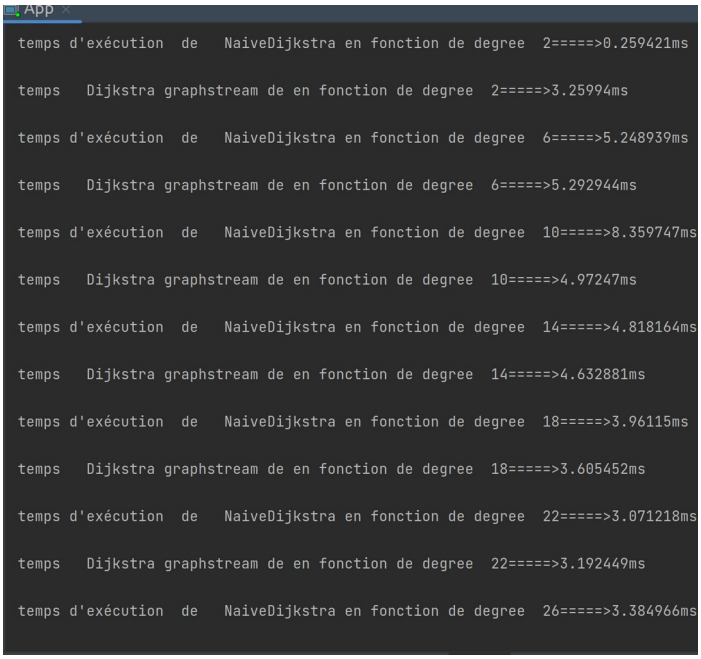

Figure 5: Temps d'exécution des deux versions d'algorithme en fonction de degré

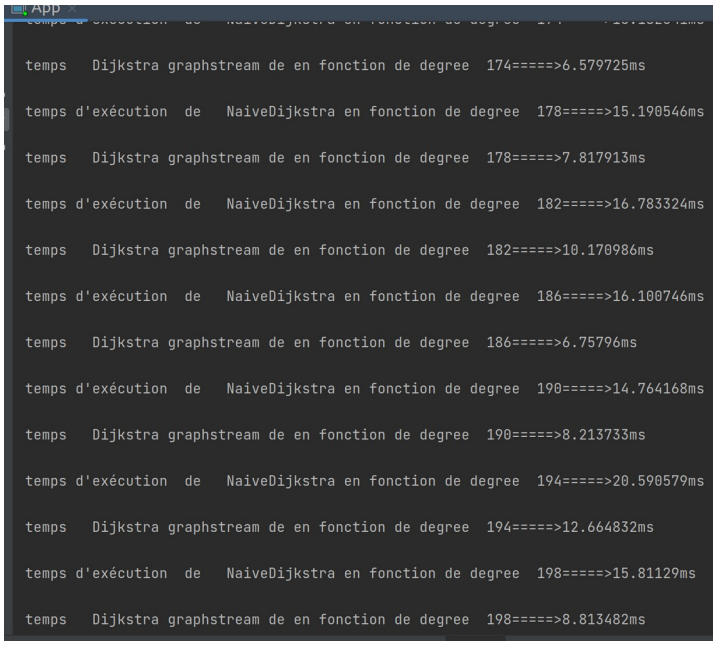

Figure 6: Temps d'exécution des deux versions d'algorithme en fonction de degré

Test 03:

le but de ce test est de faire une comparaison de temps d'exécution des deux versions en fonction de la taille du graphe, dans un premier temps l'extraction de deux fichiers où j'ai mis les résultats de temps d'exécution de chaque algorithme en fonction de leur taille sachant que le test est effectué sur le même graphe avec le même nœud source.
Une fois que les données sont bien extraites, j'ai opté pour l'outil gnuplot afin d'obtenir la courbe suivante:

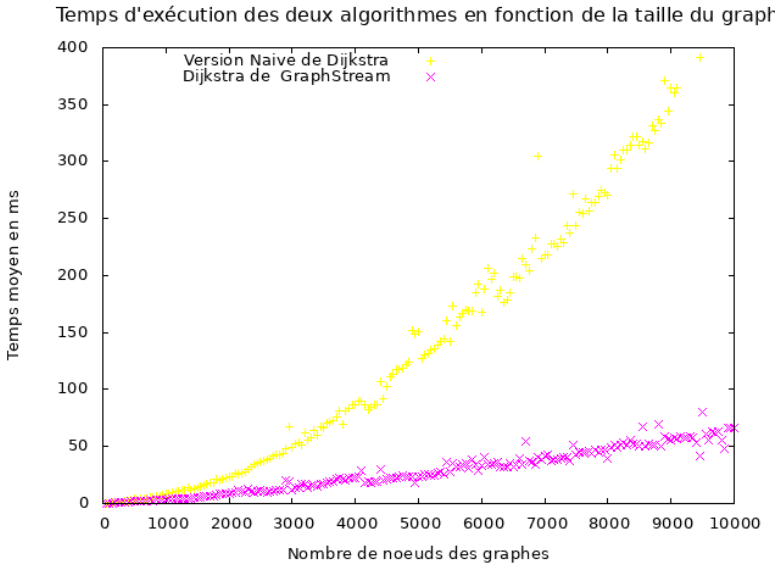

Figure 5: courbes montrant le temps d'exécution des deux algorithmes

D'après les résultats obtenues nous réalisons que le temps d'exécution est meilleur ou équivalent dans les deux cas d'algorithme pour un petit nombre de nœuds par contre pour un graphe de très grande taille l'algorithme Dijkstra de GraphStream est meilleur en terme du temps d'exécution .

Conclusion :

D'après ce que nous avons réaliser durant ce TP nous avons pu dégager la différence entre le temps d'exécution des deux algorithmes Dijkstra naïve et la version intégré dans GraphStream.