Master1 IWOCS 2022/2023

Rapport sur l'algorithme de Dijkstra

Graphe

Algorithme de Dijkstra

Plus courts chemins

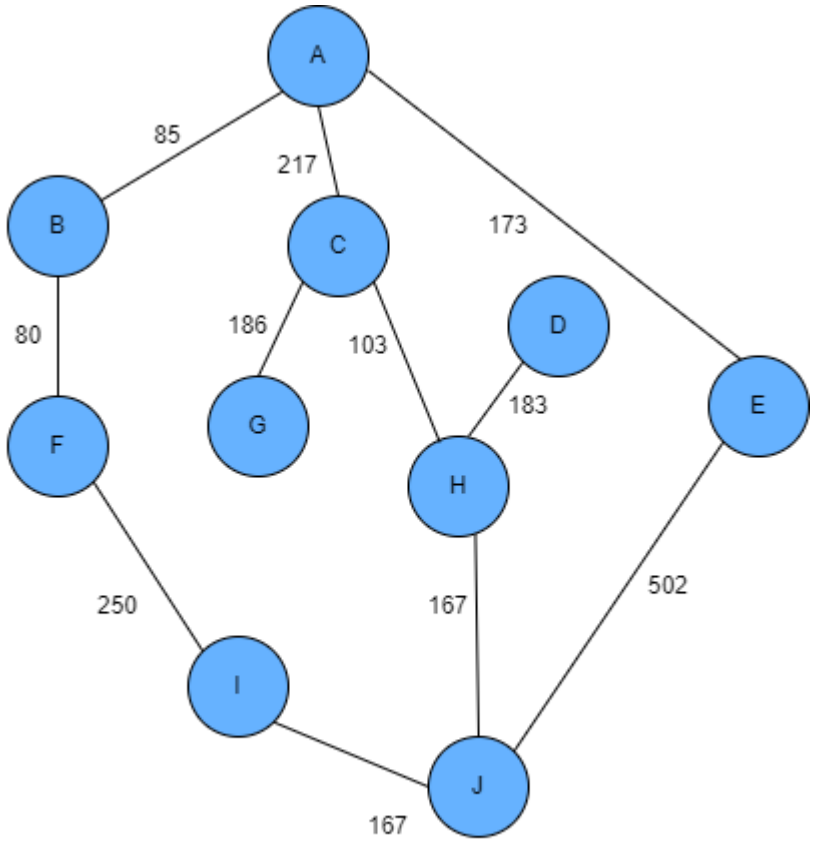
Réaliser par : Firdaous El Halafi

Table des matières

Introduction

La théorie des graphes consiste à modéliser différents problèmes de la vie réelle sous forme de graphes. L'une des utilisations les plus classiques est la modélisation d'un réseau routier entre différentes villes. L'une des problématiques principales étant l'optimisation des distances entre deux points. Pour trouver le plus court chemin, on utilise souvent l'algorithme de Dijkstra.

Dijkstra a proposé en 1959 un algorithme qui permet de calculer le plus court chemin entre un sommet particulier et tous les autres dans un graphe pondéré (orienté ou non) dont tous les poids sont positifs.



exemple d'un graph pondéré

Imaginons que l'on cherche à trouver le plus court chemin entre la ville A et la ville J.

Tout au long de l'algorithme on va garder en mémoire le chemin le plus court depuis A pour chacune des autres villes dans un tableau.

On répète toujours le même processus :

1. On choisit le sommet accessible de distance minimale comme sommet à explorer.

2. A partir de ce sommet, on explore ses voisins et on met à jour les distances pour chacun. On ne met à jour la distance que si elle est inférieure à celle que l'on avait auparavant
3. On répète jusqu'à ce qu'on arrive au point d'arrivée ou jusqu'à ce que tous les sommets aient été explorés.

Rappel du sujet

Dans ce premier TP, il était demandé d'implanter implémentez une version naïve de l'algorithme de Dijkstra, et de le tester sur un graph générer aléatoirement, puis comparer ses performances avec l'autre algorithme de Dijkstra (celui de GraphStream).

Definition d'un Generateur

Le module « gs-algo » propose un certain nombre de générateurs et d'algorithmes. Les générateur du GraphStream crée des graphiques aléatoires de n'importe quelle taille. L'appel begin() place un nœud unique dans le graphe, puis nextEvents() ajoutera un nouveau nœud à chaque fois qu'il est appelé et connectera ce nœud de manière aléatoire aux autres.

Le générateur essaie de générer des nœuds avec des connexions aléatoires, chaque nœud ayant en moyenne un degré donné.

`La méthode Generateur :`

La méthode **Generateur()** prend en paramètre deux entiers «taille » et « degré », elle fait appel au generateur du GraphStream, elle permet de créer un graph générer aléatoirement à partir de la taille (le nombre des nœuds du graph) et le degré moyen(le nombres d'arêtes moyennes de chaque noeud du graphe) passé en paramètre. On ajoute après des poids aléatoires générer par la méthode **Random()** sur nos arêtes sinon Dijkstra ne marche pas, puis nous attribuons une valeur numérique à chaque attribut « poid » des arêtes, et à chaque attribut « label » des noeuds.

Implémentation de notre algorithme Dijkstra

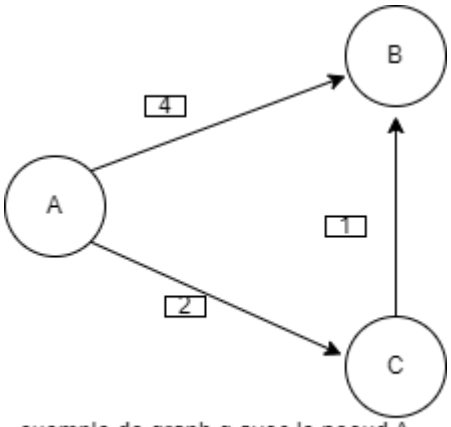
Dijkstra effectue un parcours en largeur (BFS) Avec des files à priorités.

L'algorithme de parcours en largeur (ou BFS, pour *Breadth-First Search* en anglais) permet le parcours d'un graphe de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les voisins non explorés des voisins, etc.

Les files à priorités sont des structures de données permettant de stocker des éléments triés en fonction de leur « priorité »

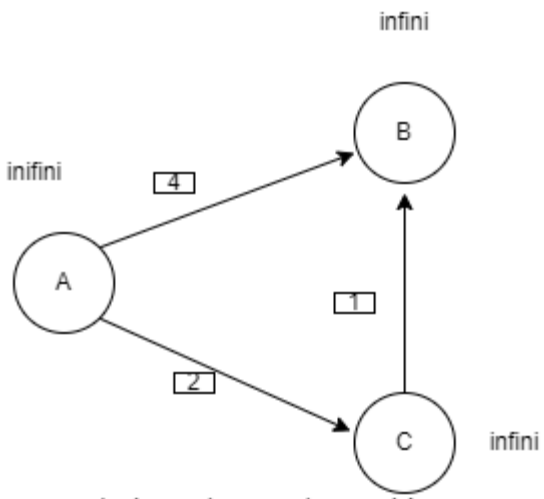
La méthode Dijkstra :

La méthode Dijkstra prend en paramètre deux attributs un graph à parcourir « g » et un nœud source (celui à partir de lequel on veut commencer notre parcours) « s ».



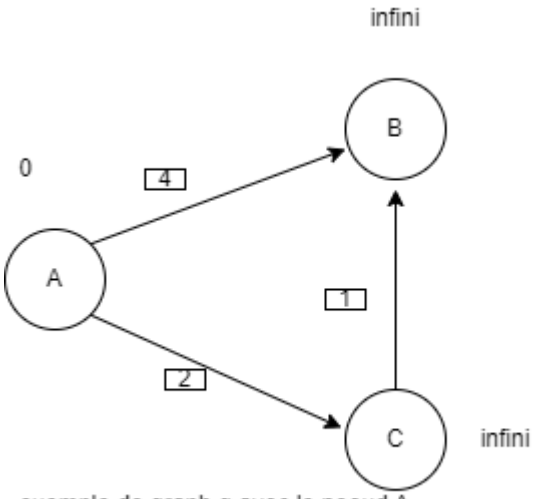
exemple de graph g avec le noeud A comme source

on commence d'abord à initialiser tout les nœuds avec une distance infinie, et leurs attributs parents à 0.



exemple de graph g avec le noeud A comme source

On attribue 0 à la distance de la source. On crée une liste et on ajoute la source puis on parcourt notre liste tans qu'il n'est pas vide.



exemple de graph g avec le noeud A comme source

On fait appel après à notre méthode extractMin

La méthode extractMin :

Le méthode **extractMin** prend en paramètre un nœud « n » (le nœud qu'on veut récupérer son élément à priorité minimum) et une liste à parcourir « list ».

On initialise nôtre élément à priorité minimum avec le premier élément de la List, et notre distance minimum à l'infinie.

On parcourt notre liste, et s'il y'a une arrête entre le nœud courant (nœud n) et le nœud actuel de la liste(list.get(i)), et que la distance entre notre nœud et l'élément de la liste (la distance ici

est équivalente à la distance du nœud actuel de la liste plus le poid de l'arrêtes entre les deux nœuds) est inférieur à notre distance minimum.

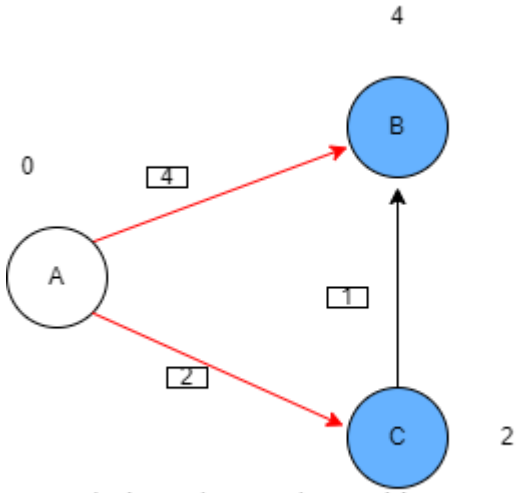
On met à jour notre minimum distance avec la distance entre notre nœud et l'élément de la liste, et notre nœud apriorité minimum au l'élément actuel de la liste (list.get(i)).

Notre fonction retourne à la fin après avoir fini la liste le nœud a priorité minimum.

On crée un itérateur qui point sur les voisins du nœud apriorité minimum.

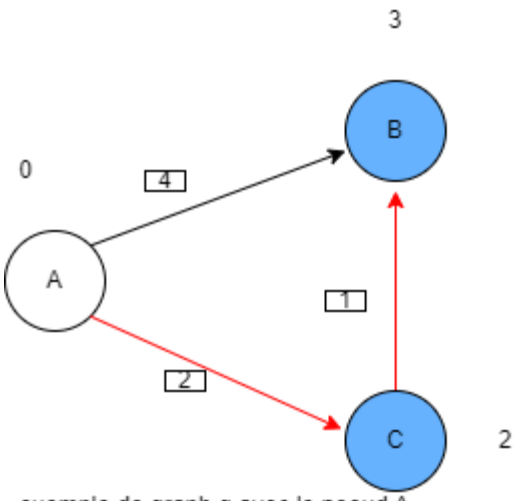
s'il y a une arrête dans le bon sens entre le nœud courant et son voisin dans le cas d'un graph orienté(on rencontre ce problème), et quand on trouve une distance entre notre nœud (le nœud à priorité minimum) et son voisin (la distance ici est équivalente à la distance du nœud courant plus le poid de l'arrêtes entre les deux nœuds) est inférieur à notre distance minimum un nœud du distance inferieur à la distance du noeud voisin.

On met à jour la distance du nœud voisin avec la distance du nœud courant plus le poid de l'arrêtes entre les deux nœuds.



exemple de graph g avec le noeud A comme source

On met à jours l'attribut parents du nœud voisin au nœud courant, on colorie le nœud voisin, et on l''ajout a note list la liste(list.add(voisin)), et on affiche le chemin entre la source et le voisin.



exemple de graph g avec le noeud A comme source

On supprime après le nœud de priorité minimum(list.remove(courant))

Dijkstra de GraphStream

L'utilisation classique de la classe de GrahStream se déroule en 4 étapes.

1. Définition d'une instance de Dijkstra avec les paramètres nécessaires à l'initialisation.

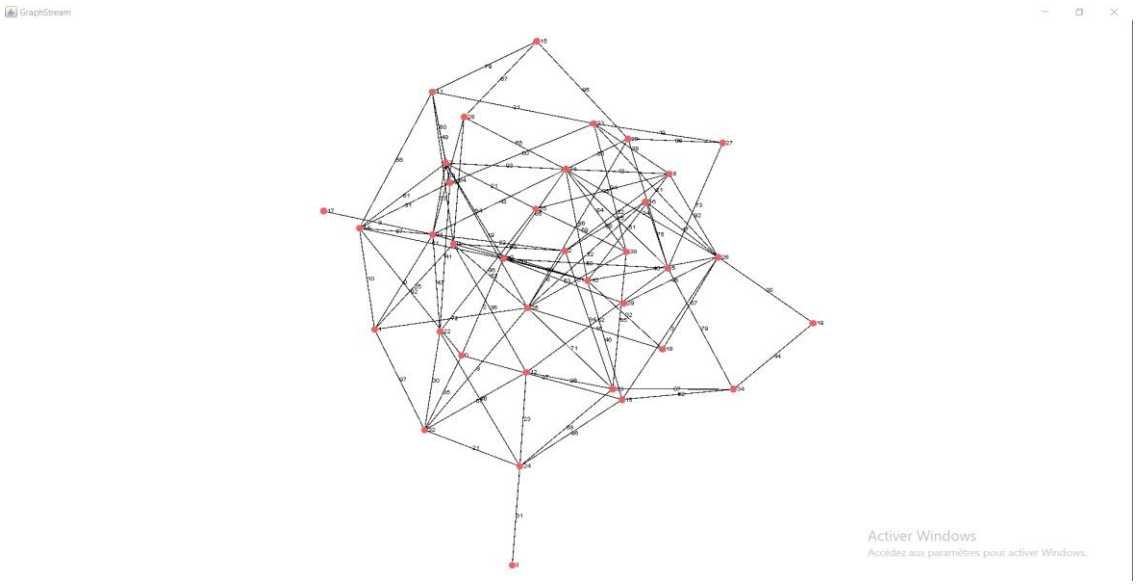
   La création de l'instance Dijkstra se fait avec le **Dijkstra(Element, String, String)** constructeur en donnant 3 paramètres :

   (I) Tout d'abord, le type d'élément à considérer pour le calcul des plus courts chemins (**Dijkstra.Element.edge** ou **Dijkstra.Element.node**).
   (II) Deuxièmement, la chaîne de clé de l'attribut utilisé pour le calcul du poids.
   (III) Le troisième paramètre est l'identifiant du nœud source pour lequel l'arbre le plus court sera construit.

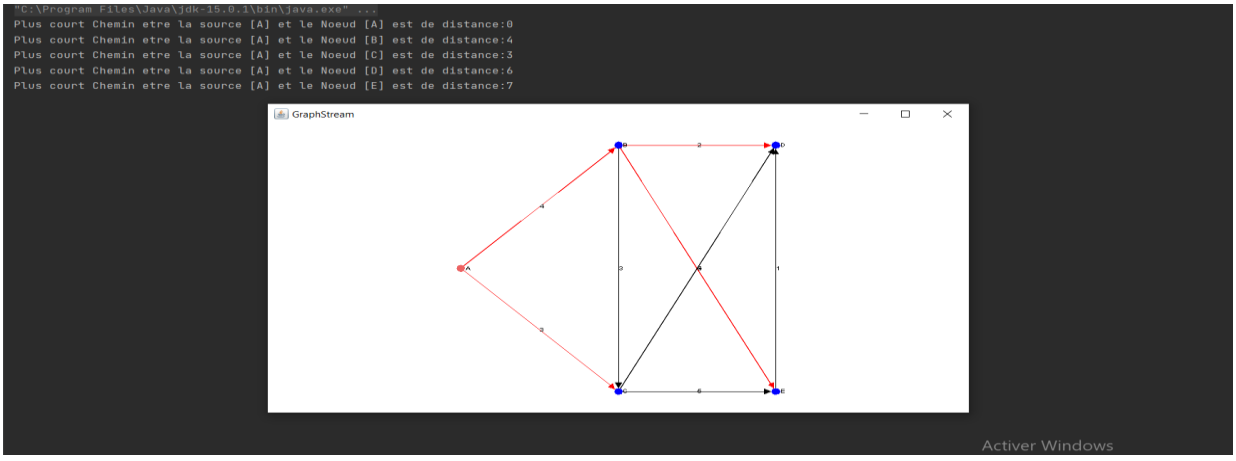
2. Initialisation de l'algorithme avec un graphe à travers la **init(Graph)** méthode.
3. Calcul de l'arbre des plus courts chemins avec la **compute()** méthode.
4. Récupération des chemins les plus courts pour des destinations données avec la **getShortestPath(Node)** méthode par exemple.

Les tests

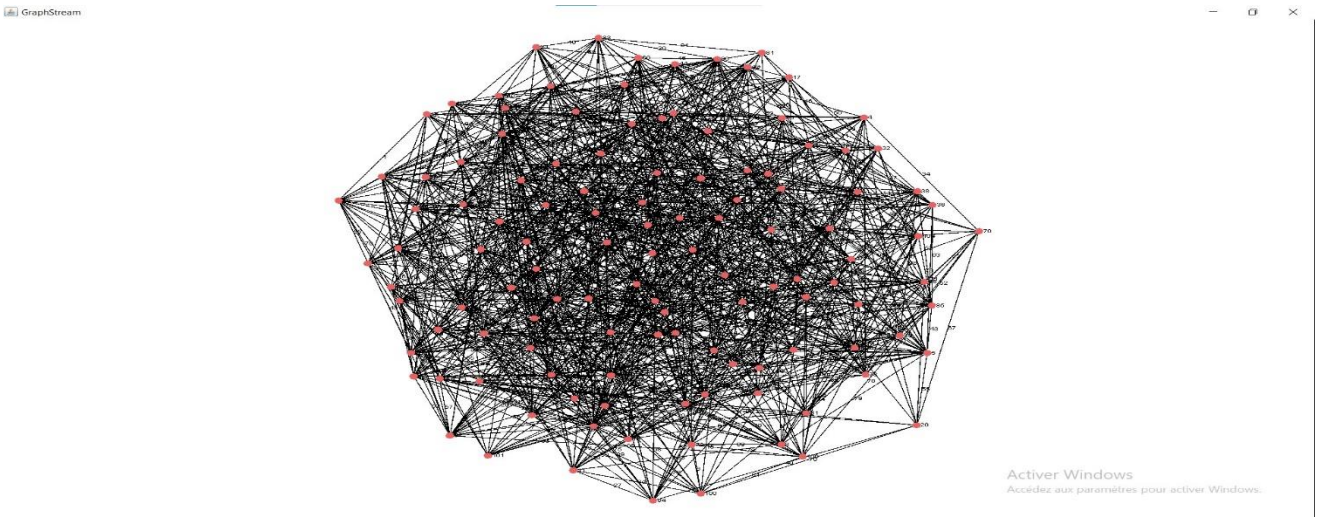
D'abord j'ai commencé par créer un graph aléatoire avec des poids sur les arrêts avec la méthode Generateur et j'obtiens le résultat suivant :



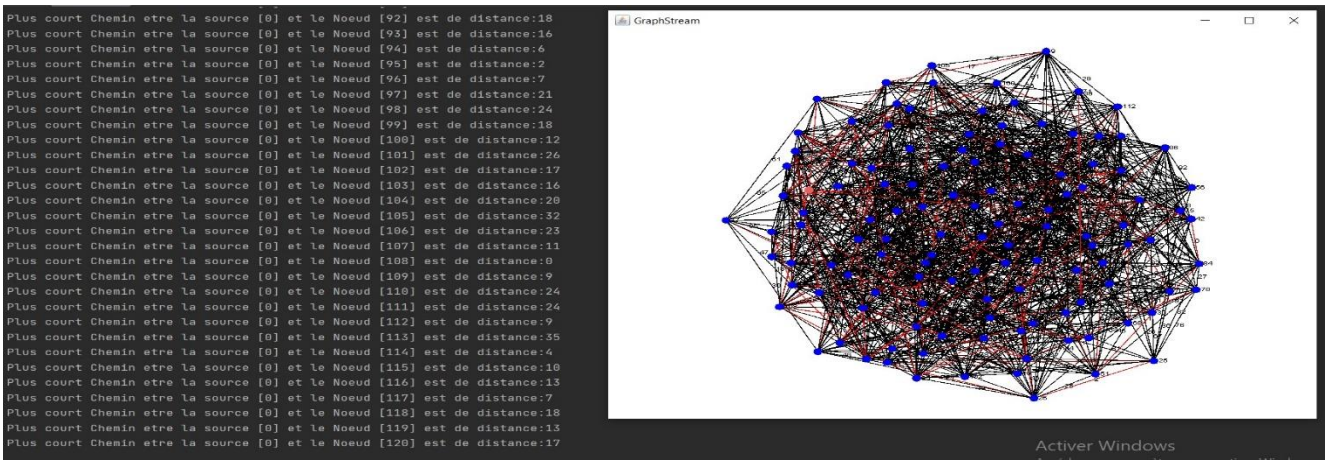
Après j'ai implémentez mon algo du dijkstra, Pour tester bien le fonctionnement de mon algo, j'ai commencé d'abord par créer un premier exemple d'un graph crée à partir du fichier DGS, il s'agit d'un graph oriente avec des poids sur les arrêtes, puis j'ai appliqué mon algo sur ce graph avec 'A' comme nœud source, on obtient les résultats suivants :



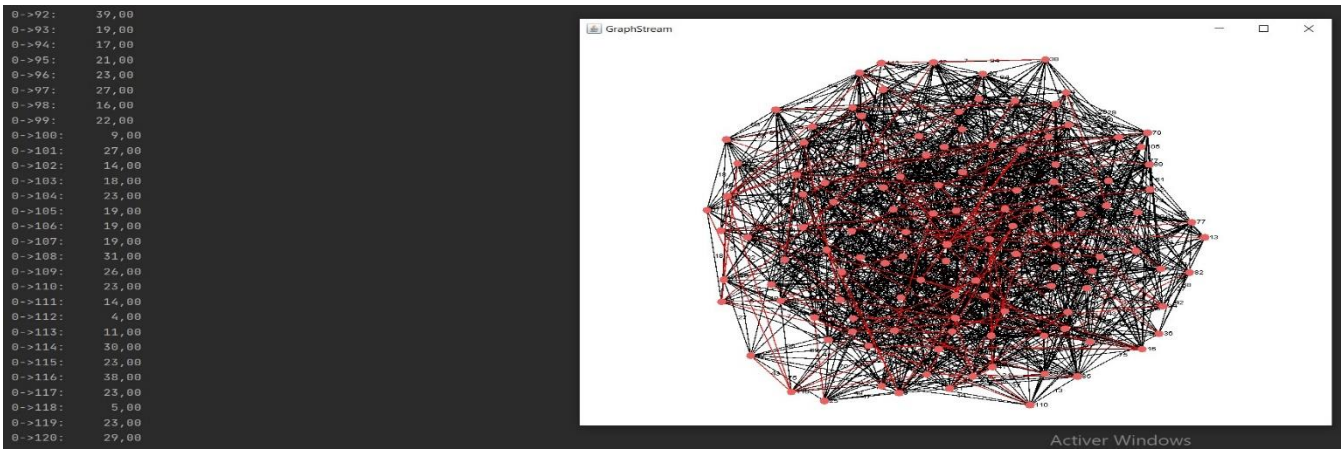
J'ai créé un graph de grand taille (taille 100 et degré 20) avec la méthode Generateur, j'obtiens le résultat suivant :



Après j'ai appliqué mon algo sur ce graph avec le nœud d'id 0 comme source et j'obtiens le résultat suivant :



Puis on applique le Dijkstra du GraphStream sur le même graph et le même nœud, j'obtiens le résultat suivant :



Pour tester la performance de mon algo, j'ai créé deux fonctions (**tempExecution, tempExecutionGraphStream** et qui prennent en paramètres un graph et un nœud source) pour mon algo et celui du GraphStream et qui permet de calculer le temps d'exécution de l'algo en milliSecond, puis j'ai comparé les deux avec le graph précèdent et j'obtiens le résultat suivant :

```
Temps d'execution de notre algo : 262ms.
Temps d'execution de l'algo du GraphStrea : 18ms.
```

Pour mieux comparer J'ai fait une fonction qui crées des graphs de plus en plus grands (***tempExecutionPlusieurGraphIdentiques*** qui prend en paramètre un entier début qui représente la taille de plus petit graph et un entier nbIterations qui représentes le nombre d'itérations et un entier degré qui representes le degre des arretes) avec une boucle de nbIterations itérations, et on applique nos deux algorithmes sur construits de la même manière. On appel nos fonctions de comparaisons et on stocke les résultats dans deux tableaux, on obtient les résultats suivants :

Comparatif de 100 résultats pour un graphe de taille entre [900,1000] identique :

```
Temps d'execution de notre algo :
        Minimal: 1598ms.
        Maximal: 4395ms.
        Moyenne: 2768.17ms.
Temps d'execution de l'algo du GraphStream :
        Minimal: 6ms.
        Maximal: 63ms.
        Moyenne: 11.81ms.
```

Comparatif de 10 résultats pour un graphe de taille entre [1990,2000] identique :

```
Temps d'execution de notre algo :
        Minimal: 17880ms.
        Maximal: 26653ms.
        Moyenne: 21171.8ms.
Temps d'execution de l'algo du GraphStream :
        Minimal: 32ms.
        Maximal: 101ms.
        Moyenne: 47.7ms.
```

J'ai fait une autre fonction qui crées des graphs de plus en plus grands (***tempExecutionPlusieurGraphDifferent*** qui prend en paramètre un entier nbIterations qui représentes le nombre d'itérations et un entier max et min qui representent le minimum et le maximum de la tailles du graph, et un entier degré qui representes le degre des arretes) avec une boucle de nbIterations itérations, et on applique nos deux algorithmes sur des graphes de tailles differents crée à partir du méthode **Random**() . On appel nos fonctions de comparaisons et on stocke les résultats dans deux tableaux, on obtient les résultats suivants :

Comparatif de 100 résultats pour un graphe de taille entre [900,1000] aléatoire :

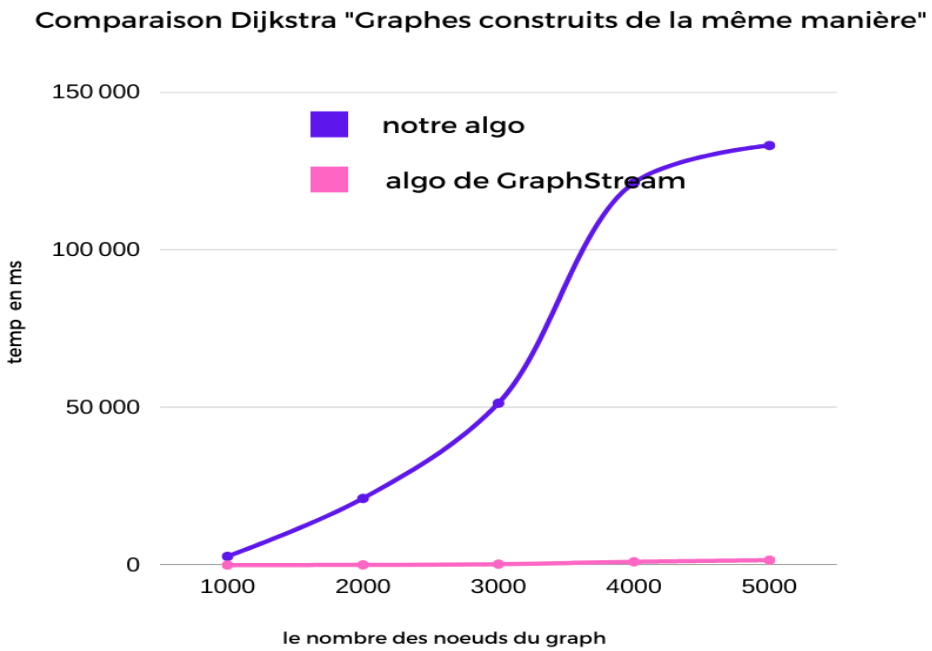
```
Temps d'execution de notre algo :
        Minimal: 1599ms.
        Maximal: 4527ms.
        Moyenne: 2751.62ms.
Temps d'execution de l'algo du GraphStream :
        Minimal: 6ms.
        Maximal: 80ms.
        Moyenne: 12.61ms.
```

Comparatif de 10 résultats pour un graphe de taille entre [1990,2000] aléatoire:

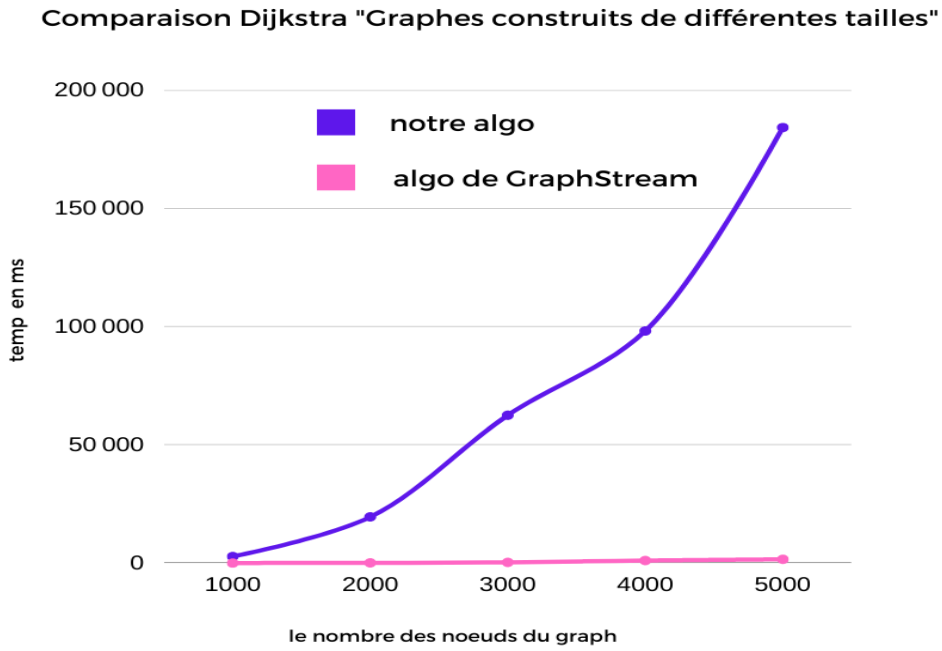
```
Temps d'execution de notre algo :
        Minimal: 16427ms.
        Maximal: 23686ms.
        Moyenne: 19481.3ms.
Temps d'execution de l'algo du GraphStream :
        Minimal: 28ms.
        Maximal: 117ms.
        Moyenne: 49.3ms.
```

Présentation des résultats obtenus

Pour les résultats des graphes construits de la même manière et pour plusieurs exécutions on obtient :



Pour les résultats des graphes de différentes tailles et pour plusieurs exécutions on obtient



Explications des résultats obtenus

Les courbes des comparaisons de Dijkstra (tailles identiques et tailles aléatoires) sont presque pareilles.

On constate que le temps d'exécution augmente avec la taille du Graph, et que notre algo est beaucoup moins rapide que celui du GraphStream.

La différence vient de l'optimisation lié au tas de Fibonnacci donnes à l'algo de GraphStream une meilleure performance. Et aussi liée à la complexité O(m + nlog n), tandis que notre complexité est O(n + m). On peut donc se dire que l'implémentation de l'algorithme sera toujours à peu près similaire mais peut-être optimisé via différent moyen (tas de Fibonnacci, etc.)

Conclusion

Pour conclure, les deux algorithmes n'utilisent pas les mêmes méthodes de résolution mais on peut également remarquer que l'algorithme de GraphStream est plus efficace et puissant pour des graphes de très grande taille.