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

FENARD Dorian ---- YAX Cyril

https://github.com/DorianFenard/SAE-2.2-FENARD_YAX

- Introduction

Ce rapport présente le travail réalisé dans le cadre de la SAE 2.02 sur l'exploration algorithmique et la recherche de chemin dans un graphe. Les algorithmes de Bellman-Ford et de Dijkstra ont été implémentés et comparés pour déterminer leur efficacité respective. Nous détaillerons les différentes parties du projet, les résultats obtenus et l'analyse de ces résultats.

- Représentation d’un graphe
- Algorithme du point fixe (Bellman-Ford)
- Algorithme de Dijkstra
- Validation
- Conclusion (Question 16/17/18)

Représentation d’un graphe

Dans cette partie, nous avons créé les classes Arc, Arcs, Graphe ainsi que GrapheListe, afin de pouvoir représenter des graphes et de pouvoir y appliquer par la suite les algorithmes.

Classe GrapheListe :

- Travail réalisé : Cette classe permet de représenter un graphe sous forme de liste d'adjacence. Elle inclut des méthodes pour ajouter des arcs, obtenir les successeurs d'un nœud et lister tous les nœuds du graphe.

Classe Arc :

- Travail réalisé : Cette classe représente un arc entre deux nœuds avec un coût associé. Elle inclut des méthodes pour obtenir la destination de l'arc et son coût.

Algorithme du point fixe

Algorithme du point fixe (Question 8)

```
Fonction pointFixe(Graphe g, String depart)
  v ← nouvelle Valeur()
  nonVisites ← nouvel Ensemble()

  Pour chaque noeud dans g.listeNoeuds() faire
    Si noeud == depart alors
      v.setValeur(noeud, 0)
    Sinon
      v.setValeur(noeud, +∞)
    Fin Si
    v.setParent(noeud, null)
    nonVisites.ajouter(noeud)
  Fin Pour

  Tant que nonVisites n'est pas vide faire
    min ← null
    Pour chaque noeud dans nonVisites faire
      Si min est null ou v.getValeur(noeud) < v.getValeur(min) alors
        min ← noeud
      Fin Si
    Fin Pour

    Si min est null alors
      Sortir de la boucle
    Fin Si

    Pour chaque arc dans g.suivants(min) faire
      voisin ← arc.getDest()
      nouvelleDistance ← v.getValeur(min) + arc.getCout()
      Si nouvelleDistance < v.getValeur(voisin) alors
        v.setValeur(voisin, nouvelleDistance)
        v.setParent(voisin, min)
```

```
        Fin Si
      Fin Pour

      nonVisites.supprimer(min)
   Fin Tant Que

   Retourner v
Fin Fonction    Fin Tant que
Fin
```

Variables

- g: Graphe, représenté par une structure contenant les noeuds et les arcs.
- depart: String, le nom du noeud de départ.
- v: Valeur, une structure qui stocke les distances minimales et les parents des noeuds.
- nonVisites: Ensemble<String>, un ensemble contenant les noms des noeuds non visités.
- min: String, le nom du noeud avec la distance minimale actuellement.
- noeud: String, le nom d'un noeud.
- arc: Arc, une structure contenant l'arc et son coût.
- voisin: String, le nom du noeud voisin.
- nouvelleDistance: Double, la nouvelle distance calculée pour un voisin.

Classe BellmanFord :

Travail réalisé : Cette classe implémente l'algorithme de Bellman-Ford pour trouver les plus courts chemins dans un graphe à partir d'un nœud de départ donné. L'algorithme met à jour les coûts des chemins et les parents des nœuds de manière itérative.

Tests effectués :

Vérification des résultats de l'algorithme sur différents graphes pour s'assurer de la correction des chemins calculés et des coûts associés dans une classe BellmanFordTest.

Algorithme de Dijkstra

L'algorithme de Dijkstra a été implémenté pour trouver les plus courts chemins de manière efficace à partir d'un nœud de départ donné.

Classe Dijkstra :

Travail réalisé : Cette classe implémente l'algorithme de Dijkstra en utilisant une file de priorité pour sélectionner le nœud avec la distance minimale à chaque étape.

L'algorithme met à jour les coûts des chemins et les parents des nœuds de manière efficace.

Tests effectués :

Vérification des résultats de l'algorithme sur différents graphes pour s'assurer de la correction des chemins calculés et des coûts associés dans une classe DijkstraTest .

Validation

Pour valider les algorithmes, nous avons comparé les performances des algorithmes de Bellman-Ford et de Dijkstra sur différents graphes grâce à la classe ComparaisonAlgo, qui calcule grâce à « nanoTime() » la durée de l'exécution de chaque algorithme, et l'insère dans un fichier csv afin de pouvoir comparer les différents résultats.

Classe ComparaisonAlgo :

Travail réalisé :

Travail réalisé : Cette classe compare les performances des algorithmes de Bellman-Ford et de Dijkstra sur différents graphes. Elle mesure le temps d'exécution de chaque algorithme et vérifie que les résultats des deux algorithmes sont identiques en termes de coûts des chemins calculés. Les résultats des performances sont ensuite affichés en nanosecondes, millisecondes et secondes pour une meilleure compréhension.

Tests effectués :

Comparaison des temps d'exécution des algorithmes de Bellman-Ford et de Dijkstra sur des graphes de tailles variées.

Vérification de l'égalité des résultats produits par les deux algorithmes pour s'assurer de leur correction.

Résultats :

Question 16.
Nous avons réalisé différentes comparaisons sur différents graphes de différentes tailles allant de 11 éléments à 905.

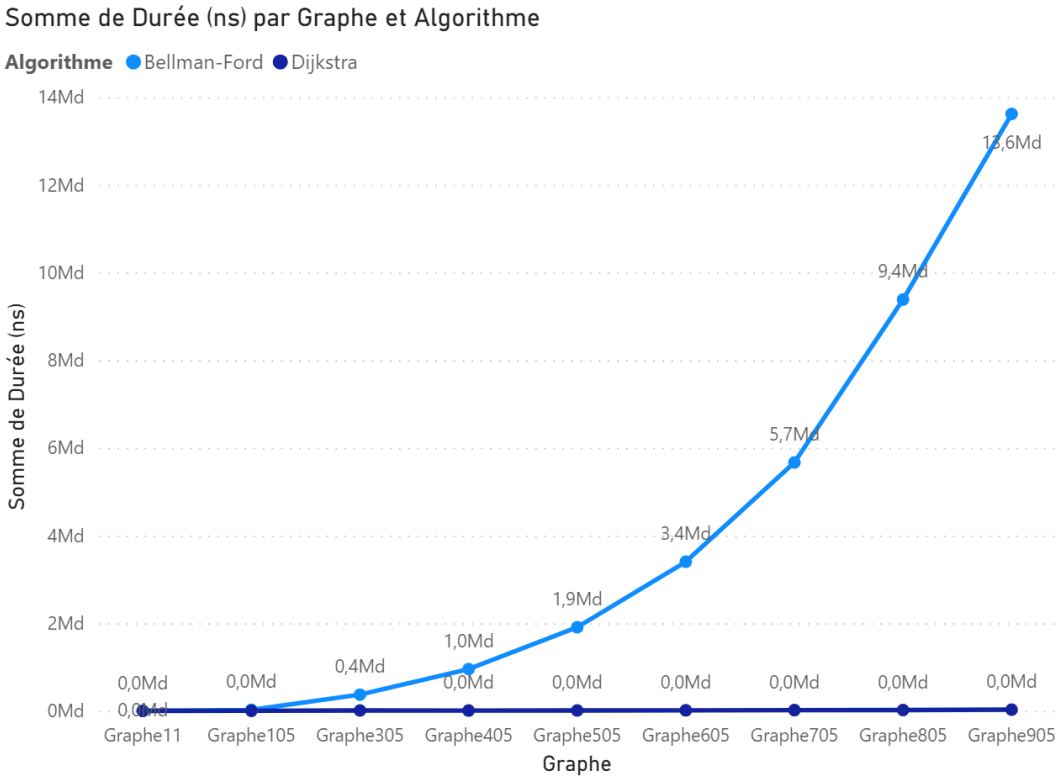
Graphe	Algorithme	Durée (ns)	Durée (ms)	Durée (s)	Résultats Identiques
Graphe11	Bellman-Ford	864916	0.864916	8.64916E-4	Oui
Graphe11	Dijkstra	217041	0.217041	2.17041E-4	Oui
Graphe105	Bellman-Ford	22659125	22.659125	0.022659125	Oui
Graphe105	Dijkstra	1790875	1.790875	0.001790875	Oui
Graphe305	Bellman-Ford	370945042	370.945042	0.370945042	Oui
Graphe305	Dijkstra	10049167	10.049167	0.010049167	Oui
Graphe405	Bellman-Ford	954112542	954.112542	0.954112542	Oui
Graphe405	Dijkstra	7838208	7.838208	0.007838208	Oui
Graphe505	Bellman-Ford	1907618375	1907.618375	1.907618375	Oui
Graphe505	Dijkstra	9978916	9.978916	0.009978916	Oui
Graphe605	Bellman-Ford	3402177833	3402.177833	3.402177833	Oui
Graphe605	Dijkstra	13012750	13.01275	0.01301275	Oui
Graphe705	Bellman-Ford	5668572417	5668.572417	5.668572417	Oui
Graphe705	Dijkstra	17938583	17.938583	0.017938583	Oui
Graphe805	Bellman-Ford	9382642583	9382.642583	9.382642583	Oui
Graphe805	Dijkstra	18728042	18.728042	0.018728042	Oui
Graphe905	Bellman-Ford	1,3616E+10	13615.83575	13.61583575	Oui
Graphe905	Dijkstra	28253417	28.253417	0.028253417	Oui

Question 17/18

Les deux algorithmes trouvent les mêmes résultats, mais dans des temps bien différents.

On voit que l'algorithme de Dijkstra est bien plus rapide que celui de Bellman-Ford, et cette différence se fait encore plus ressentir dans les grands graphes complexes, car Dijkstra est plus rapide de 18 secondes par rapport à Bellman-Ford (sachant qu'il n'a pris que 43 millisecondes pour le faire).

Il est le plus rapide car il dispose d'une file de priorité, ce qui est beaucoup plus rapide, car l'autre doit relâcher chaque arrête plusieurs fois, ce qui, dans un grand graphe, prend beaucoup de temps.



Graphique représentant la durée par Graphe de chaque Algo en nanosecondes (L'algorithme de Dijkstra n'atteint jamais une seconde de durée, alors que Bellman-Ford 13)

Conclusion Générale

Ce projet nous a permis d'implémenter et de comparer deux algorithmes classiques de recherche de plus courts chemins : Bellman-Ford et Dijkstra. Nous avons appris à représenter des graphes de manière efficace et à utiliser des structures de données adaptées pour optimiser les algorithmes.

Difficultés rencontrées :

Comprendre les différences entre les algorithmes et leurs conditions d'application.

Réussir à comprendre le fonctionnement de ces algorithmes, et à les retranscrire sous forme de code java (surtout l'algorithme de Dijkstra, qui nous était inconnu avant).

Bilan :

Cette SAE nous a permis de renforcer nos compétences en algorithmique et en structures de données. Nous avons également acquis une meilleure compréhension des critères de performance des algorithmes et de leur application pratique à des problèmes de graphes.