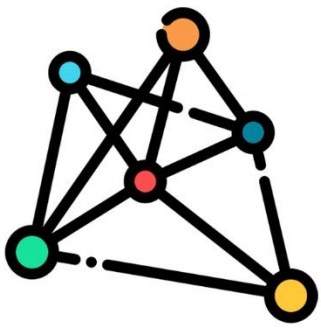
UNIVERSITÉ DE LORRAINE

S2.02 Exploration algorithmique d'un problème

2024

BABACHANAKH Kateryna

groupe S2D



2) Représentation d'un graphe

Dans cette partie, j'ai créé plusieurs classes et interfaces pour représenter un graphe orienté. Voici les détails du travail réalisé :

1. Classe *Arc*

   Description: La classe *Arc* représente une arc partant d'un nœud et contenant un coût (ou poids).

   Travail réalisé: Ajout des paramètres et des méthodes de classe, et définition du constructeur pour initialiser les arcs avec un nœud de destination et un coût.

2. Classe *Arcs*

   Description: La classe *Arcs* gère une collection d'arcs partant d'un nœud.

   Travail réalisé: Création de la classe avec un constructeur qui initialise une liste vide d'arcs. Ajout des méthodes pour ajouter un arc à la liste et pour récupérer la liste des arcs.

3. Interface *Graphe*

   Description: L'interface *Graphe* définit les méthodes pour manipuler des graphes.

   Travail réalisé: Définition des méthodes *listeNoeuds* et *suivants* pour retourner respectivement la liste des nœuds du graphe et la liste des arcs partant d'un nœud spécifique.

4. Classe *GrapheListe*

   Description: La classe *GrapheListe* implémente l'interface *Graphe* et permet de représenter les données associées à un graphe.

   Travail réalisé:

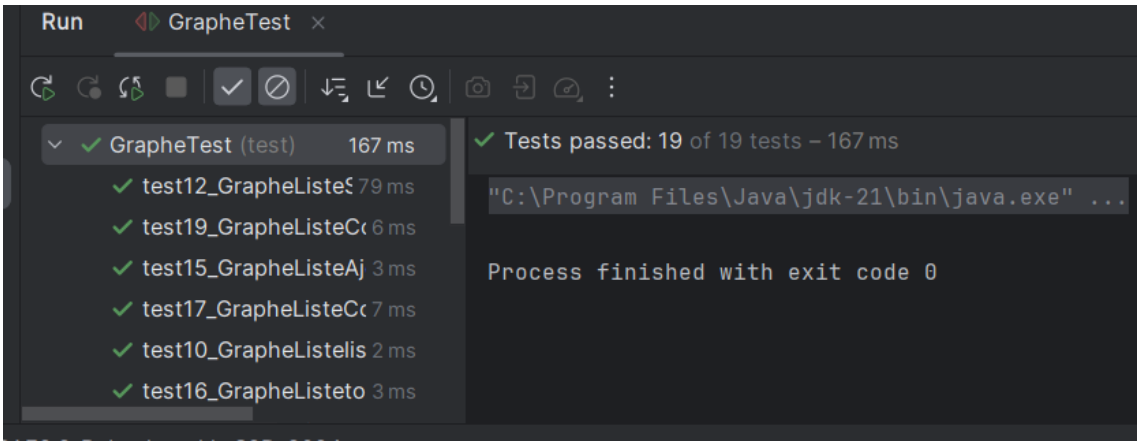
   - Ajout des attributs pour stocker les nœuds et les arcs.
   - Création de la méthode *ajouterArc* pour ajouter des arcs au graphe. J'ai décidé d'ajouter une vérification en double avant d'ajouter des arcs au graphe. Car sans cette vérification nous nous retrouverons avec des doublons et cela conduit à l'ajout d'enregistrements supplémentaires pouvant produire une erreur.
   - Création de la méthode *getIndice* pour retourner l'indice du noeud n de la liste noeuds.
   - Création de la méthode *toString* pour afficher les nœuds du graphe et les arcs liés à chaque nœud.

- J'ai ajouté un répertoire au projet avec des fichiers txt avec des exemples de graphe du arche. Après cela, j'ai décidé d'ajouter un constructeur pour initialiser le graphe à partir d'un fichier contenant les descriptions des arcs. Et j'ai ajouté des lignes de code à main et tests pour vérifier la bonne utilisation des fichiers.

Pour vérifier la validité des classes et méthodes, j'ai écrit des tests unitaires (*GrapheTest*) en utilisant Junit. Les tests suivants ont été réalisés :

1. Test d'un Arc avec un coût négatif pour vérifier que le coût est initialisé à 0 lorsque le coût fourni est négatif;
2. Test du constructeur de graphe vide pour vérifier que le constructeur initialise correctement un graphe vide;
3. Test de la méthode *getIndice* pour verifier que la méthode retourne les indices corrects des nœuds dans la liste;
4. Test de la liste des nœuds pour verifier que la méthode *listeNoeuds* retourne correctement tous les nœuds du graphe;
5. Test de la liste des arcs pour verifier que la *suivants* retourne correctement la liste des arcs partant d'un nœud spécifié;
6. Test de l'ajout des arcs pour verifier que la méthode *ajouterArc* ajoute correctement les arcs au graphe;
7. Tests du constructeur *GrapheListe* avec un fichier valide, un fichier invalide, et un fichier vide;
8. etc.

Tous les tests ont réussi sans erreurs:



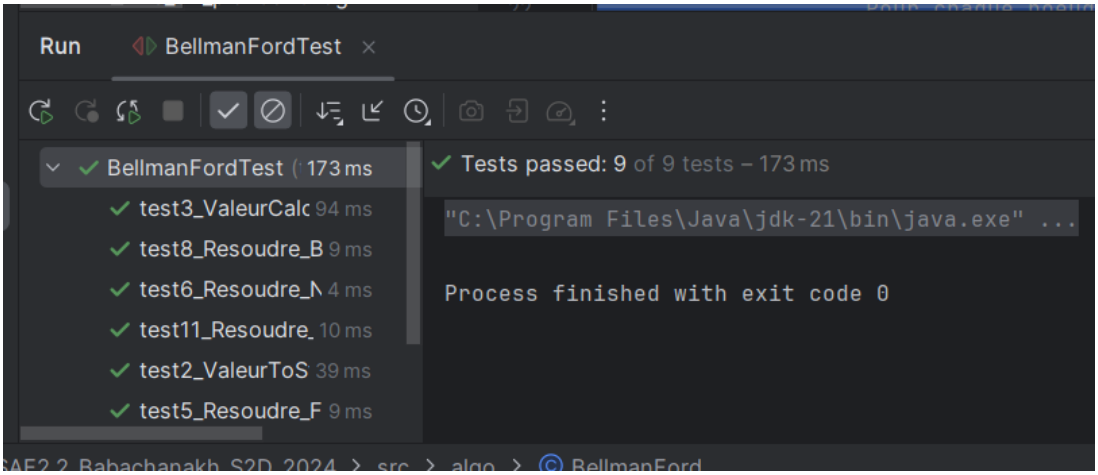
3) Calcul du plus court chemin par point fixe

Dans cette section, j'ai implémenté l'algorithme du point fixe, aussi appelé algorithme de Bellman-Ford, pour trouver les chemins les plus courts dans un graphe orienté. L'objectif était de calculer les valeurs des nœuds du graphe en fonction de leur distance depuis un nœud de départ et de déterminer les parents des nœuds. Voici les détails du travail réalisé :

1. Classe *Valeur*
   Description: La classe *Valeur* permet de stocker et manipuler les distances et les parents des nœuds.
   Travail réalisé: J'ai ajouté un fichier *Valeur.java* sur arche au projet. J'ai ajouté une méthode *calculerChemin* pour pouvoir afficher le chemin menant à un nœud donné en remontant les parents de ce nœud à partir du point de départ.
2. Classe *BellmanFord*
   Description: La classe *BellmanFord* contient la méthode resoudre qui implémente l'algorithme de Bellman-Ford.
   Travail réalisé: Ecriture de l'algorithme de la fonction *pointFixe(Graphe g InOut, Noeud depart)* dans les commentaires. Puis en utilisant la classe *Valeur* et mon implémentation de l'algorithme de Bellman-Ford, j'ai programmé l'algorithme du point fixe dans la classe *BellmanFord* en ajoutant la méthode *resoudre*.

Pour vérifier la validité de mon algorithme, j'ai écrit des tests unitaires et un programme principal pour appliquer l'algorithme sur un graphe fourni et afficher les valeurs de distance pour chaque nœud.

Tous les tests ont réussi sans erreurs:



4) Calcul du meilleur chemin par Dijkstra

Dans cette section, j'ai implémenté l'algorithme de Dijkstra pour trouver les chemins les plus courts dans un graphe orienté. L'objectif était de calculer les valeurs des nœuds du graphe en fonction de leur distance depuis un nœud de départ et de déterminer les parents des nœuds. Voici les détails du travail réalisé :

1. Classe *Dijkstra*
   Description: La classe *Dijkstra* contient la méthode resoudre qui implémente l'algorithme de *Dijkstra*.
   Travail réalisé: En utilisant la classe *Valeur* et l'algorithme de Dijkstra, j'ai programmé l'algorithme du point fixe dans la classe *Dijkstra* en ajoutant la méthode *resoudre*.

2. Interface *Algo*

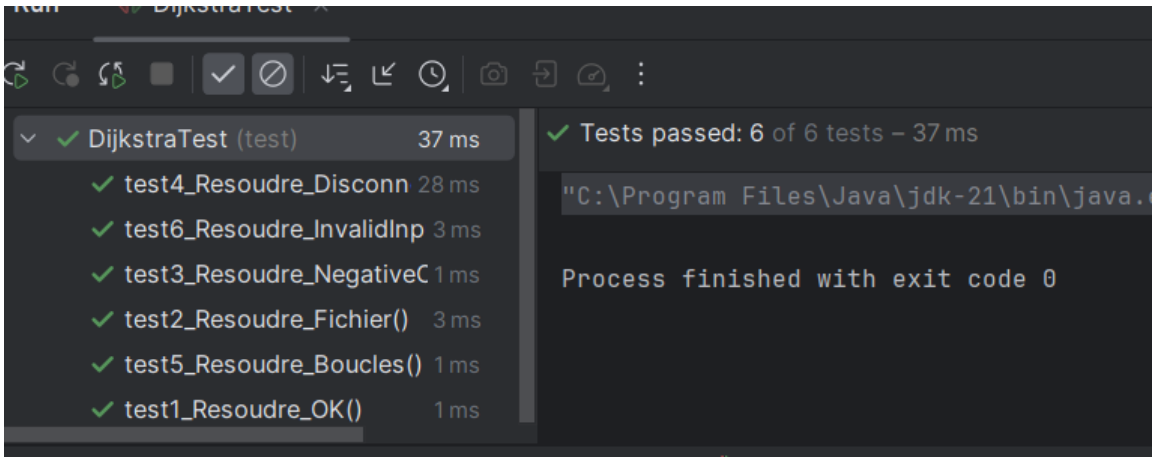
   Description: L'interface *Algo* permet de définir une méthode commune pour résoudre le problème du plus court chemin.

   Travail réalisé: J'ai remarqué que les classes BellmanFord et Dijkstra ont la même méthode. J'ai donc décidé de créer une interface Algo qui sert de base aux algorithmes des classes Bellman-Ford et Dijkstra. J'ai écrit de la méthode *resoudre* pour résoudre le problème du plus court chemin.

3. Programme principal *MainDijkstra*
   Description: J'ai écrit un programme principal *MainDijkstra* pour appliquer l'algorithme sur un graphe fourni et afficher des chemins pour des nœuds donnees.

Pour vérifier l'implémentation de l'algorithme de Dijkstra, j'ai écrit des tests unitaires similaires à ceux de l'algorithme de Bellman-Ford.

Tous les tests ont réussi sans erreurs:



5) Validation et experimentation

Les résultats des tests unitaires pour les algorithmes de Dijkstra et de Bellman-Ford sont présentés ci-dessous :

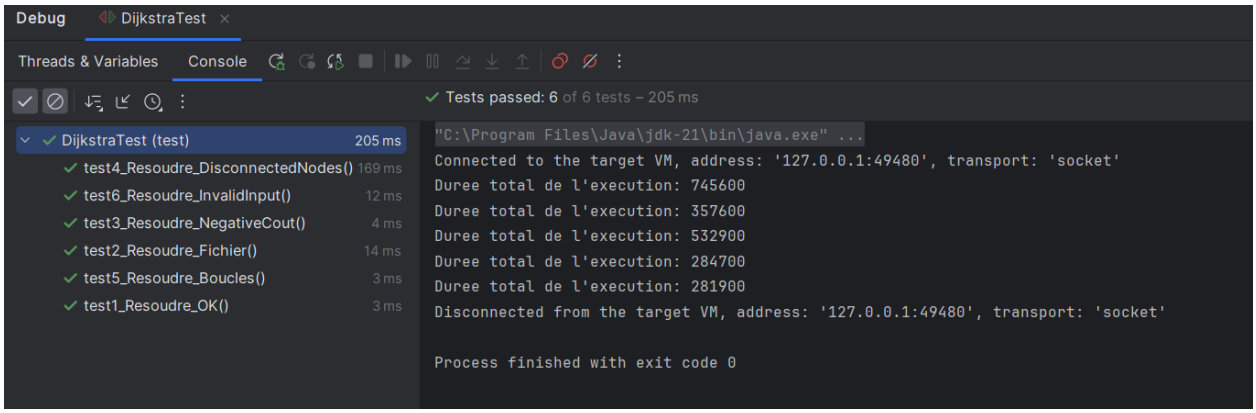


Figure 1 Algorithme de Dijkstra

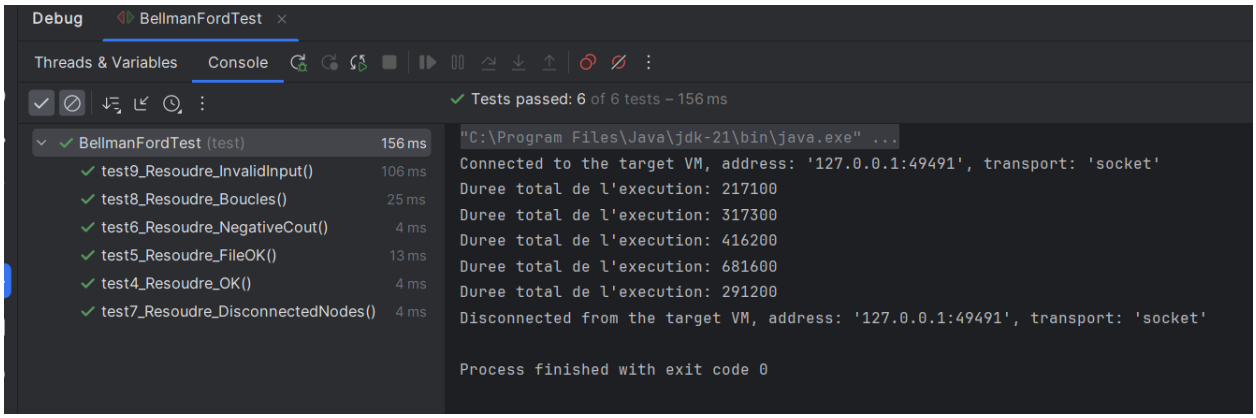


Figure 2 Algorithme de Bellman-Ford

Question 17 : Proposer une analyse des résultats obtenus. Quel algorithme fonctionne le mieux expérimentalement et pourquoi ?

Les tests montrent que l'algorithme de Bellman-Ford tend à être plus rapide dans la plupart des cas, avec des temps d'exécution inférieurs à ceux de Dijkstra.

L'algorithme de Dijkstra fournit des performances stables et optimales pour les graphes sans arêtes avec un coût négatif en raison de sa stratégie de propagation des valeurs.

Bellman-Ford a l'avantage de gérer des graphiques avec des bords de coûts négatifs, ce qui le rend plus polyvalent.

Question 18 : Quel est selon vous l'algorithme qui devrait être le plus efficace et pourquoi ?

L'algorithme de Bellman-Ford devrait être considéré comme le plus efficace pour les raisons suivantes :

- Bellman-Ford est capable de gérer des graphes avec des arêtes à coût négatif, ce qui n'est pas possible avec Dijkstra.
- Les tests montrent que Bellman-Ford est rapide et efficace même pour des graphes standards sans arêtes à coût négatif.
- Simplicité algorithmique et capacité à fonctionner dans des scénarios complexes sans modification majeure.

5.2 Chargement de Graphe

Quand j'ai réalisé la deuxième partie de ce SAE, j'ai ajouté un constructeur pour le chargement des fichiers contenant des graphes. De plus, tous les tests sont écrits pour vérifier le chargement des données à partir d'un fichier.

Conclusion générale

Ce projet m'a permis d'approfondir mes connaissances sur les algorithmes de recherche de chemins et sur la façon de représenter les graphes en programmation. J'ai appris à implémenter deux algorithmes importants, Bellman-Ford et Dijkstra, et à comparer leurs performances.

Les principales difficultés rencontrées concernaient la gestion des graphes avec des arêtes à coût négatif et l'optimisation des performances des algorithmes. La mise en œuvre de l'algorithme de Bellman-Ford était particulièrement complexe en raison de ses nombreuses étapes de calcul.

Ce travail m'a permis de comprendre l'importance des tests unitaires et de la validation expérimentale pour garantir la fiabilité des algorithmes.