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

Korban Ryan, Aloïs Masson-Claudez

Représentation d'un graphe

Code Produit :

- La classe Arc pour représenter les arcs reliant les nœuds.
- La classe Arcs pour gérer un ensemble d'arcs partant d'un nœud.
- L'interface Graphe avec les méthodes listeNoeuds() et suivants() pour manipuler des graphes.
- La classe GrapheListe qui implémente l'interface Graphe et permet de représenter les données associées à un graphe.

Test Effectué :

La classe de test GrapheListTest test les méthodes de la classe GrapheListe.

- testListeNoeuds() : Vérifie si la méthode listeNoeuds() retourne correctement la liste des nœuds.
- testSuivants() : Vérifie si la méthode suivants() retourne correctement les arcs partant d'un nœud donné.
- testGetIndice() : Vérifie si la méthode getIndice() retourne correctement l'indice d'un nœud dans la liste des nœuds.
- testAjouterArc() : Vérifie si la méthode ajouterArc() ajoute correctement un nouvel arc à partir d'un nœud donné.

Point fixe

Code produit :

- La Classe BellmanFord représente l'algorithme de point fixe pour la recherche du chemin minimum
- L'interface Algorithme qui représente un algorithme à résoudre

Test Effectué :

- testResoudreGrapheVide() : Vérifie que le résultat est vide pour un graphe vide.

- testResoudreGrapheUnSeulNoeud() : Vérifie que la distance du nœud à lui-même est 0 pour un graphe avec un seul nœud.
- testResoudreGrapheUnSeulArc() : Vérifie que la distance entre deux nœuds est correcte pour un graphe avec un seul arc.
- testResoudreGrapheAvecCycleNegatif() : Vérifie que l'algorithme détecte correctement les cycles négatifs dans le graphe.
- testResoudreGrapheAvecArcsNegatifs() : Vérifie que l'algorithme gère correctement les arcs avec des coûts négatifs.
- testResoudreGrapheNonConnecte() : Vérifie que les nœuds inaccessibles renvoient une distance infinie.
- testResoudreGrapheAvecNomsNonAlphabetiques() : Vérifie que l'algorithme gère correctement les noms de nœuds non alphabétiques.
- testResoudreGrapheAvecArcsDeCoutZero() : Vérifie que l'algorithme gère correctement les arcs avec un coût de 0.
- testResoudreGrapheAvecNoeudsInaccessibles() : Vérifie que les nœuds inaccessibles renvoient une distance infinie.

Exercice :

```
début
    fonction pointFixe(g : Graphe, noeud : chaine)
    début
    lDistanceNoeuds
    lDistanceNoeuds[1] <- 0
    nbNoeuds <- g.listeNoeuds().size() - 1
    pour i de 2 à nbNoeuds :
    ajouterListe(lDistanceNoeuds, ∞)
    fin pour
    pareil <- faux
    pour i de 1 à nbNoeuds :
    noeudsSuivants <- lDistanceNoeuds.suivants()
    min <- noeudsSuivants[1]
    pour j de 2 à noeudsSuivants.size() :
    si noeudsSuivants[j] < min :
        min <- noeudsSuivants[j]
    fin si
    fin pour
    nvLDistanceNoeuds <- lDistanceNoeuds[i] + min
    fin pour
    tant que non pareil :
```

```
        n <- 1
        tant que lDistanceNoeuds[n] = nvLDistanceNoeuds[n] et n <=
lDistanceNoeuds.size() :
        n <- n+1
        fin tant
        si n > lDistanceNoeuds.size() :
        pareil <- vrai
        fin si
        pour i de 1 à nbNoeuds :
        lDistanceNoeuds <- nvLDistanceNoeuds
        noeudsSuivants <- lDistanceNoeuds.suivants()
        min <- noeudsSuivants.getFirst()
        pour j de 2 à noeudsSuivants.size() :
                si noeudsSuivants[j] < min :
                min <- noeudsSuivants[j]
                fin si
        fin pour
        nvLDistanceNoeuds <- lDistanceNoeuds[i] + min
        fin pour
        fin tant
        fin
 fin
```

Dijkstra

Code Produit :

- La classe MainDijkstra utilise les méthodes de la lasse Dijkstra
- La classe Dijkstra représente l'algorithme de Dijkstra pour la recherche du chemin minimum

Test Effectué :

- testResoudreCheminMin() : Vérifie si l'algorithme calcule correctement le chemin le plus court entre deux nœuds.
- testResoudreCheminMinVersE() : Vérifie si l'algorithme calcule correctement le chemin le plus court vers le nœud E.
- testNoeudNonConnecte() : Vérifie si l'algorithme attribue une distance infinie à un nœud non connecté.

- testGraphAvecCycles() : Vérifie si l'algorithme gère correctement les graphes contenant des cycles.

Exercice 13 :

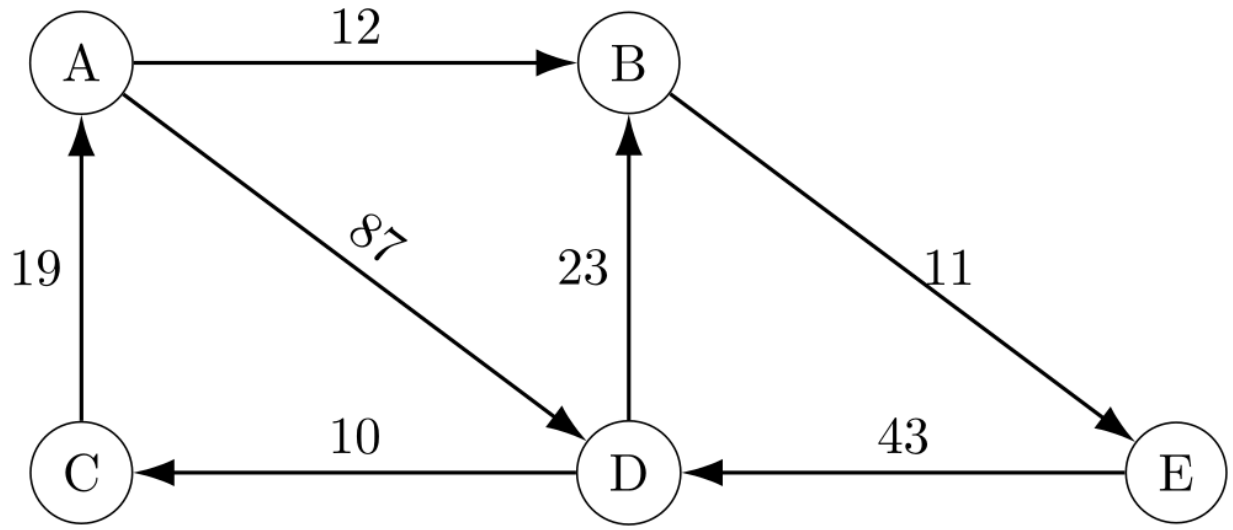
```
//Entrees :
//G un graphe orienté avec une pondération positive des arcs (coût ou poids)
//A un sommet (départ) de G

//Début
//Q <- {} // utilisation d'une liste de noeuds à traiter
//Pour chaque sommet v de G faire
//      v.valeur <- Infini
//      v.parent <- Indéfini
//      Q <- Q U {v} // ajouter le sommet v à la liste Q
//Fin Pour

//A.valeur <- 0
//Tant que Q est un ensemble non vide faire
//      u <- un sommet de Q telle que u.valeur est minimal
//      // enlever le sommet u de la liste Q
//     Q <- Q \ {u}
//      Pour chaque sommet v de Q tel que l'arc (u,v) existe faire
//              d <- u.valeur + poids(u,v)
//              Si d < v.valeur
//              // le chemin est plus interessant
//              Alors v.valeur <- d
//              v.parent <- u
//              Fin Si
//      Fin Pour
//Fin Tant que
// Fin
```

Validation

Code produit ;

- La classe comparerAlgo permet de comparer les différents algorithmes

Question 16 :
Avec le graphe utilisé ci-dessous, nous obtenons les résultats suivants :
Complexité Bellman-Ford : 24 itérations
Durée d'exécution Bellman-Ford : 3197.884 µs
Complexité Dijkstra : 6 itérations
Durée d'exécution Dijkstra : 318.57 µs



Question 17 :
On voit que l'algorithme de Dijkstra est plus rapide que celui de Bellman-Ford dans ce cas. Il permet de faire moins d'itérations pour trouver le résultat. Ce résultat est cohérent avec ce que nous avions vu en cours de mathématiques.

Question 18 :
Je pense que l'algorithme qui devrait être le plus efficace est l'algorithme de Dijkstra. Nous avons vu en cours que cet algorithme a une complexité plus faible dans le pire des cas $O((s + a)\ log(s))$ contre $O(|s|\times|a|)$ pour l'algorithme de Bellman-Ford. Cette affirmation est vraie pour le cas donné dans l'exemple de la question 16.

Conclusion :

Durant cette SAÉ, nous avons pu apprendre de nouvelles compétences, comme la distribution des tâches et la gestion d'un dépôt git, que nous avions abordées en TP d'ailleurs, mais jamais dans le cadre d'une SAÉ. Cette SAE nous a donc permis de améliorer nos compétences de collaboration et de l'organisation dans un projet.

Nous avons également implémenté des graphes, un concept que nous avons énormément étudié en math, en créant les classes Arc, Arcs et GrapheListe, nous avons pu mettre nos connaissances théoriques en pratique.

L'une des parties le plus important de cette SAÉ a été le codage des algorithmes de recherche du plus court chemin, Bellman-Ford et Dijkstra. Ces algorithmes, même s'il ont le même objectif, présentent. En les implémentant et en les testant, nous avons pu comparer leurs performances et déterminer lequel était le plus efficace dans divers scénarios. Nous avons découvert que l'algorithme de Dijkstra était plus performant pour les graphes sans arêtes de poids négatif, tandis que Bellman-Ford était plus efficace pour traiter les graphes avec des cycles de poids négatif.

Cependant, nous avons rencontré des difficultés, notamment avec l'algorithme de Bellman-Ford. La manière dont nous avions implémenté les graphes ne nous permettait pas d'accéder facilement aux antécédents des sommets, ce qui a complexifié l'implémentation. Cette difficulté nous a poussé à revoir notre approche et à améliorer notre compréhension de la manière dont on implémente des graphes en Java. Nous avons donc dû ajuster notre conception initiale.

Nous avons également réalisé des tests pour voir si nos implémentations marchaient bien. Ces tests nous ont permis de détecter et de corriger plusieurs erreurs, ce qui nous a permis d'améliorer notre maîtrise des tests et nous rendre compte de leurs importances.