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

Ce sujet présente la SAE d'exploration algorithmique. Il se base sur les CM/TD de graphes vus dans les ressources mathématiques (Sujet version 3.1 - (2023-05-16 11:10)).

Consigne

- **Auteurs du sujet** : Abdessamad Imine, Vincent Thomas.
- **Durée** : 16h de travail étudiant.
- **Format** : travail en binôme.
- **Rendus attendus** :
  - un dépôt `git` correctement structuré ;
  - un code java qui compile, clair, commenté avec une javadoc ;
  - un rapport qui présente le travail réalisé et les réponses aux questions abordées ;
  - des tests unitaires qui valident le travail réalisé.
- La section 7 présente en détail le contenu des éléments attendus.

1 Présentation de la SAE

Étant donné un graphe orienté, composé de nœuds et d'arcs étiquetés par des coûts (ou poids) positifs, le problème du plus court chemin consiste à trouver un chemin d'un nœud à un autre de sorte que la somme des coûts soit minimale.

Ce problème a plusieurs applications. L'exemple typique est l'utilisation du GPS lorsque l'on cherche un trajet entre deux villes sur une carte (représentée comme un graphe pondéré) en fonction de plusieurs critères (le chemin le plus rapide en temps, le chemin le plus court en distance, ...). Par exemple, dans le graphe de la figure 1, le plus court chemin entre `A` et `C` est `[A, B, E, D, C]` (ayant un coût de 76). Par ailleurs, dans un graphe orienté, on dit que le nœud `v` est *adjacent* au nœud `u` si et seulement si l'arc `(u, v)` existe. Les nœuds `B` et `D` de la figure 1 sont adjacents à `A`.

Le but de cette SAE est d'implémenter des solutions algorithmiques pour le problème de recherche du plus court chemin dans un graphe. Elle se présente comme suit :

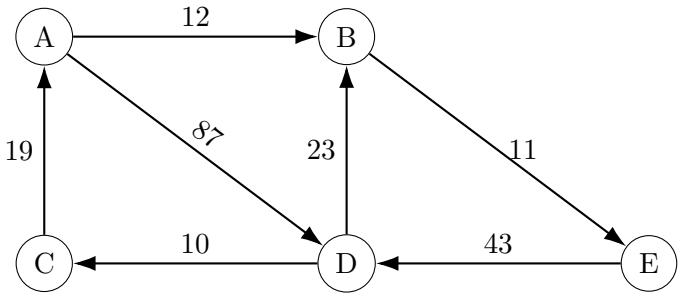

FIGURE 1 – Exemple de graphe pondéré

- Dans la section 3, vous devrez écrire puis implémenter **l'algorithme du point fixe** (ou de **Bellman-Ford**) que vous avez déjà utilisé dans les ressources de mathématiques.
- Dans la section 4, vous devrez aussi implémenter la solution algorithmique proposée par l'informaticien **Edsger Wybe Dijkstra (1930-2002)**.
- La section 6 est optionnelle (même si des points lui sont réservés dans le barème) et permet d'utiliser la recherche de chemins dans un labyrinthe fourni en utilisant des solutions de conception vues en COO.

Les graphes seront représentés comme un ensemble de nœuds où chaque nœud est décrit par un nom et la liste de ses nœuds adjacents. Pour ce faire, vous allez devoir développer dans la section 2 trois classes permettant de représenter un graphe :

- une classe `Noeud` pour décrire les nœuds d'un graphe ;
- une classe `Arc` pour représenter les arcs reliant les nœuds ;
- une interface `Graphe` et des sous-classes pour manipuler des graphes.

De plus, nous souhaitons pouvoir construire un graphe **à partir d'un fichier texte** où chaque ligne représente un arc pondéré entre deux nœuds.

2 Représentation d'un graphe (4h)

2.1 TAD Graphe

Un TAD [1] `Graphe(N)` est un objet informatique qui repose sur un type de nœuds `N`. Un arc partant d'un nœud `n` est défini par un couple `<suivant: N, cout: Réel>` où `suivant` désigne le nœud atteignable en suivant cet arc et `cout`, le coût associé à cet arc.

Pour pouvoir écrire l'algorithme du point fixe, on considérera les opérations suivantes sur les graphes :

- La fonction `listeNoeuds : Graphe(N)` $\rightarrow$ `Liste(N)`

1. Type abstrait de données - cf cours de structures de données

— Cette fonction `listeNoeuds` fournit la liste des nœuds d'un graphe donné.

— Sur l'exemple de la figure 1, en supposant que le graphe est nommé `G`, `listeNoeud(G)` retourne la liste des nœuds `[A,B,C,D,E]` (l'ordre des nœuds dans la liste n'a pas d'importance)

- La fonction `suivants : Graphe(N)` $\times$ `N` $\rightarrow$ `Liste(<noeud:N, cout:Reel>)`
  — Cette fonction `suivants` prend en paramètre un graphe `G` et un nœud `n` et fournit la liste des arcs partant du nœud `n` passé en paramètre.
  — Sur l'exemple de la figure 1, `suivants(G,A)` retourne la liste `[<B,12> , <D,87>]` car les nœuds `B` et `D` sont adjacents au nœud `A` et les arcs correspondants ont des coûts respectifs de 12 et 87.

2.2 Implémentation

En Java, on représentera les nœuds du graphe simplement par la classe `String` et les arcs par une classe `Arc`. En interne, les nœuds du graphe auront une structure plus complexe qu'un simple `String` car chaque nœud contient en outre la liste des arcs vers les nœuds adjacents. Ils seront représentés par la classe `Noeud`.

2.2.1 Classe `Noeud`

La classe `Noeud` représentera les nœuds du graphe. Un objet `Noeud` est décrit par deux attributs privés :

- `String nom` correspondant au nom du nœud qui permet de l'identifier ;
- `List<Arc> adj` contenant la liste des arcs reliant ce nœud à ses nœuds adjacents.

La classe `Arc` est décrite dans la section 2.2.2.

Question 1

Écrire la classe `Noeud` et un constructeur prenant en paramètre le nom du nœud et initialisant la liste `adj` à une liste vide.

Question 2

Afin de pouvoir chercher des nœuds facilement, redéfinir la méthode `boolean equals(Object o)` dans la classe `Noeud` qui spécifie que deux nœuds sont égaux si et seulement si leurs noms sont égaux (on supposera ainsi que le nom d'un nœud identifie le nœud de manière unique).

Question 3

Écrire la méthode publique `void ajouterArc(String destination, double cout)` qui ajoute un arc allant vers le nœud `destination` avec un coût de `cout` à la liste `adj`.

2.2.2 Classe `Arc`

Un objet de la classe `Arc` représente un arc partant d'un noeud. Il est décrit par deux attributs privés :

- `String dest` qui représente le nom du nœud destination de l'arc ;
- `double cout` correspondant au coût (ou poids) de l'arc.

Question 4

Écrire la classe `Arc` et un constructeur prenant en paramètres le nœud de destination et le coût (valeur strictement positive) de l'arc créé.

2.2.3 Interface Graphe

L'interface `Graphe` proposée est fondée sur le TAD graphe de la section 2.1 et possède donc les méthodes suivantes :

- `public List<String> listeNoeuds()` qui retourne tous les nœuds du graphe (sous la forme de `String` - c'est à dire leur nom).
- `public List<Arc> suivants(String n)` qui retourne la liste des arcs partant du nœud `n` passé en paramètre.

Question 5

Écrire l'interface `Graphe`

2.3 Classe GrapheListe

2.3.1 Implémentation de l'interface

La classe `GrapheListe` implémente l'interface `Graphe` et permet de représenter les données associées à un graphe. Cette classe est défini par deux attributs privés :

- un attribut `ensNom` de type `List<String>` contenant les noms des objets nœuds stockés (pour facilement écrire la méthode `List<String> listeNoeuds()`).

- un attribut `ensNoeuds` qui est une liste d'objet `Noeud` permettant de stocker les arcs.

Cette classe possède en outre une méthode publique `void ajouterArc(String depart, String destination, double cout)` permettant d'ajouter des nœuds et des arcs à un objet de type `GrapheListe`. L'ajout d'un arc se fera en précisant le nom du nœud de départ, le nom du nœud d'arrivée et le coût de l'arc. Vous penserez à vérifier que les nœuds existent bien dans le graphe lors de l'ajout et vous mettrez à jour les listes `ensNom`, `ensNoeuds` et la liste d'adjacence du nœud de départ.

Question 6

Écrire la classe `GrapheListe`

Question 7

Écrire un main qui crée le graphe présenté dans la figure 1.

2.3.2 Affichage d'un graphe

On souhaite pouvoir afficher le contenu d'un objet sous la forme d'une chaîne de caractères structurée de la manière suivante :

FICHIER

```
A -> B(12) D(87)
B -> E(11)
C -> A(19)
D -> B(23) C(10)
E -> D(43)
```

Question 8

Écrire une méthode `toString` permettant d'afficher le graphe sous cette forme.

On souhaite aussi pouvoir visualiser rapidement un graphe avec des outils adaptés. `GraphViz` (`https://graphviz.org/`) est une application permettant de générer automatiquement une image représentant un graphe à partir de sa description [2].

`GraphViz` est un outil complexe avec de très nombreuses options. On se limitera à générer une simple chaine de la forme suivante :

2. Pour information, `plantUML` utilise `Graphviz` pour positionner les éléments d'un diagramme de classe.

FICHIER

```
digraph {
  [...]
  <DEPART> -> <DESTINATION> [label = <COUT>]
  [...]
}
```

où chaque ligne représente un arc avec `<DEPART>` le nœud de départ, `<DESTINATION>` le nœud de destination et `<COUT>` le coût de l'arc (les autres informations sont liées à la structure des fichiers GraphViz). Pour l'exemple de la figure 1, la chaîne générée doit être :

FICHIER

```
digraph G {
A -> B [label = 12]
A -> D [label = 87]
B -> E [label = 11]
C -> A [label = 19]
D -> B [label = 23]
D -> C [label = 10]
E -> D [label = 43]
}
```

Question 9

Écrire une méthode `toGraphviz` qui retourne une chaîne représentant le graphe en respectant le format `GraphViz`.

A partir de cette chaîne, il est possible de générer automatiquement des images représentant les graphes comme la figure 2

- soit en utilisant directement le plugin `plantUml` sous `IntelliJ` ;
- soit en utilisant une version web de `GraphViz` comme `https://dreampuf.github.io/GraphvizOnline/` ;
- soit en utilisant directement `GraphViz` sur votre machine via la commande `dot fichier.txt -T pdf -o resultat.pdf` où `fichier.txt` désigne le fichier contenant le descriptif du graphe.

Question 10

Générer le graphe résultant et vérifier visuellement qu'il est correctement structuré.

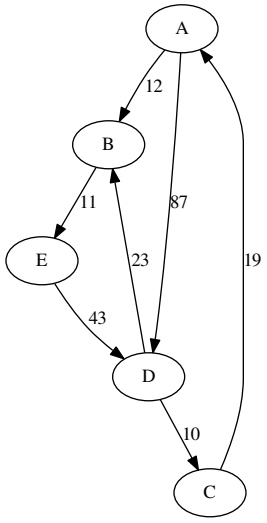

FIGURE 2 – Graphe généré par `GraphViz`

2.3.3 Validation du graphe

Question 11

Écrire des tests unitaires vérifiant que le graphe est bien construit

2.3.4 Chargement de Graphe

On utilisera des graphes représentés dans des fichiers texte. Le texte suivant représente le graphe de la figure 1. À titre d'exemple, le nœud `D` possède un arc de coût 23 vers le nœud `B` et un arc de coût 10 vers le nœud `C`.

FICHIER

```
D C 10
A B 12
D B 23
A D 87
E D 43
B E 11
C A 19
```

Le texte suivant représente le graphe de la figure 1 sous forme de matrice d'adjacence :

FICHIER

	A	B	C	D	E
A	0.	12	0.	87	0.
B	0.	0.	0.	0.	11
C	19	0.	0.	0.	0.
D	0.	23	10	0.	0.
E	0.	0.	0.	43	0.

Consigne

Attention, dans les fichiers fournis, les colonnes sont séparées par des tabulations. Il faut donc faire un `split` sur le caractère '`\t`'.

Question 12

Écrire un constructeur prenant en paramètre le nom du fichier contenant le descriptif (sous forme de liste d'arcs) du graphe et construisant l'objet graphe correspondant.

Question 13

Écrire une méthode prenant en paramètre le nom du fichier contenant le descriptif d'un graphe sous forme de matrice d'adjacence et produisant un fichier contenant une liste d'arcs.

3 Calcul du plus court chemin par point fixe (2h)

3.1 Algorithme du point fixe

Dans la ressource de « Graphe », vous avez déjà appliqué l'algorithme du point fixe (aussi appelé algorithme de « Bellman-Ford ») pour construire les chemins les plus courts dans des graphes (cf. vos notes de CM/TD sur les graphes prises en mathématiques).

Le principe de l'algorithme du point fixe consiste à initier les valeurs de tous les nœuds à $+\infty$, puis, pour chaque nœud `X`, à estimer la valeur de la fonction `L(N)` de ses nœuds adjacents `N` à partir de `L(X)` et du coût de l'arc. On modifie la valeur du voisin (et son nœud parent) si et seulement si la nouvelle valeur estimée est meilleure que la valeur estimée précédemment. L'algorithme converge vers un point fixe, c'est-à-dire une situation pour laquelle la fonction `L(X)` n'évolue plus au cours d'une itération sur tous les nœuds du graphe. Il suffit donc de répéter ces opérations jusqu'à atteindre le point fixe.

Question 14

En utilisant le TAD fourni, écrire **l'algorithme** de la fonction `pointFixe(Graphe g InOut, Noeud depart)` qui modifie les valeurs (parent et distance) associées aux nœuds du graphe pour trouver le chemin le plus court partant du nœud de départ passé en paramètre (en n'oubliant pas le lexique).

On supposera qu'on peut directement changer les valeurs de `L(X)` et `parent(X)` par une simple affectation comme `L(X)` $\leftarrow +\infty$.

3.2 Classe `Valeur`

Afin de simplifier votre code, on vous fournit la classe `Valeur` destinée à représenter les fonctions `L(X)` et `parent(X)` où X est un nœud. Cette classe permet de stocker des valeurs de type réel (pour la valeur) et de type `String` (pour le parent) en lien avec des String (représentant le nœud courant). Cela permet par exemple de stocker la valeur `L(X)` pour un nœud nommé `X` et de stocker un parent `pX` pour ce même nœud.

Cette classe sera utile pour les calculs nécessaires dans les algorithmes de recherche dans des graphes (ici l'algorithme du point fixe). Par exemple, le code java suivant construit une fonction de valeur qui stocke 10 dans `A`, et affecte `B` comme parent de `A`, et stocke 20 dans `B` et associe `C` comme parent de `B`.

```
1 Valeur v = new Valeur() ;
2 v.setValeur("A",10);  // valeur 10 dans L(A)
3 v.setParent("A","B"); // B comme parent de A
4 v.setValeur("B",20);  // valeur 20 dans L(B)
5 v.setParent("B","C"); // C comme parent de B
6 // etc ...
7 int valA = v.getValeur("A"); //retourne la valeur de A (ici 10)
8 String parentA = v.getParent("A"); //retourne le parent de A (ici B)
```

3.3 Programmation de l'algorithme du point fixe

Dans le cadre de la programmation de l'algorithme du point fixe, on remplacera la valeur $+\infty$ par la constante `Double.MAX_DOUBLE` qui est la valeur maximale que peut prendre un `double` en java.

Question 15

En utilisant la classe `Valeur` fournie et votre algorithme, implémenter l'algorithme du point fixe dans une classe nommée `BellmanFord` en programmant la méthode `Valeur resoudre(Graphe g, String depart)`.

Cette méthode prend un graphe `g` et un `String` représentant le nœud de départ en paramètre et retourne un objet valeur correctement construit contenant les

- distances et les parents de chaque nœud (après convergence de l'algorithme).

3.4 Test de l'algorithme du point fixe

La classe `Valeur` contient une méthode `toString()` permettant d'afficher l'ensemble des valeurs et des parents stockés.

Question 16

Écrire un `main` dans la classe `Main` qui applique l'algorithme du point fixe sur le graphe fourni pour construire le chemin le plus court. Afficher les valeurs de distance pour chaque nœud et vérifier qu'elles correspondent aux valeurs que vous avez calculées dans le module « Graphe ».

Question 17

Écrire un test unitaire qui vérifie que l'algorithme du point fixe est correct et que les parents des nœuds sont bien calculés.

3.5 Calcul du meilleur chemin

Une fois qu'une fonction de valeur est construite, on souhaite pouvoir afficher le chemin qui mène à un nœud donné. Il faut pour cela remonter les parents de ce nœud donné pour obtenir le chemin à partir du nœud de départ.

Par exemple, si l'objet `valeur` a été obtenu à partir de l'algorithme du point fixe sur le graphe exemple de la figure 10 en prenant pour départ le nœud `(A)`, demander le chemin menant à `C` retourne la liste `[A,B,E,D,C]`.

Question 18

Écrire la méthode `List<String> calculerChemin(String destination)` dans la classe `Valeur`. Cette méthode retourne une liste de nœuds correspondant au chemin menant au nœud passé en paramètre à partir du point de départ donné lors de la construction de l'objet `Valeur`.

4 Calcul du meilleur chemin par Dijkstra (2h)

4.1 Principe de l'algorithme de Dijkstra

L'algorithme de Dijkstra permet de construire le plus court chemin en utilisant la même fonction L(X) que celle vue dans l'algorithme du point fixe. L'algorithme de Dijkstra consiste à partir d'un nœud de départ et à calculer de proche en proche la valeur de `L(X)`, la distance du plus court chemin qui passe par `X`, et le parent de `X` dans ce chemin.

Cependant, au lieu d'appliquer un calcul sur chaque nœud jusqu'à atteindre un point fixe, l'algorithme de Dijkstra consiste à choisir un ordre de calcul adapté pour propager les valeurs calculées sans avoir besoin de mettre à jour des valeurs une fois qu'elles ont été validées.

Le **principe général** consiste à partir d'une fonction de valeur avec $+\infty$ pour chaque nœud du graphe, à attribuer une valeur de 0 au nœud de départ et à calculer la valeur `L` des voisins de proche en proche.

L'astuce de cet algorithme consiste à considérer la liste de tous les nœuds du graphe et de choisir comme nœud à développer le nœud `X` dans cette liste avec la plus petite distance `L(X)`. Le développement du nœud consiste à estimer la valeur de ses voisins (en la mettant à jour si la nouvelle distance calculée est plus petite que la distance déjà stockée). Une fois qu'un nœud a été développé, sa valeur ne pourra plus changer et il est retiré de la liste des nœuds à traiter. En recommençant à nouveau l'opération, sur le prochain nœud non traité de valeur la plus faible, on traite les nœuds un à un sans avoir besoin de revenir sur un nœud déjà traité.

Le fait que l'algorithme de Dijkstra fournit bien les plus courts chemins **peut se prouver mathématiquement** et repose sur les éléments suivants

1. tous les coûts sont positifs, l'ajout d'un arc sur un chemin ne peut donc que faire croître la distance ;
2. lorsqu'on choisit le nœud avec la distance la plus faible dans la liste des nœuds à traiter, on peut être sûr que cette valeur est définitive et qu'il n'existe pas de chemin plus court que celui déjà trouvé (étant donné que tous les nœuds encore non explorés ont une distance plus grande et qu'ajouter un arc augmente cette distance) ;
3. ce nœud peut alors être développé sans risque de trouver un meilleur chemin. Il peut ensuite être retiré de la liste des nœuds à traiter.

Les diagrammes suivants (figures 3, 4, 5, 6, 7, 8, 9) illustrent le déroulement de l'algorithme. Les nœuds à traiter (c'st-à-dire ceux qui sont encore dans la liste des nœuds à traiter qu'on appellera `Q`) sont en vert, le nœud dont on traite la valeur actuellement est en rouge et les nœuds déjà traités sont représentés en blanc.

La page wikipedia française de l'algorithme de Dijkstra[3] présente une animation permettant de mieux comprendre comment l'algorithme peut fonctionner.

3. `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`

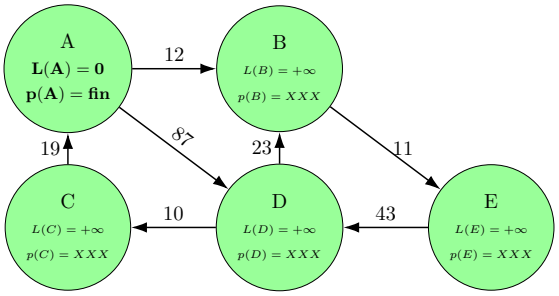

FIGURE 3 – Graphe initialisé - tous les nœuds sont à traiter (dans la liste `Q`) et `A` est le point de départ.

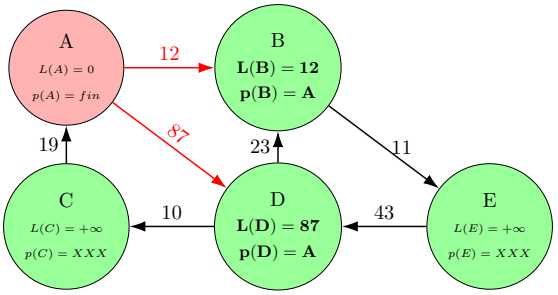

FIGURE 4 – Itération 1 - on développe le nœud de valeur minimale encore à traiter (dans la liste `Q`) (ici `A`) et on met à jour ses voisins `B` et `D`.

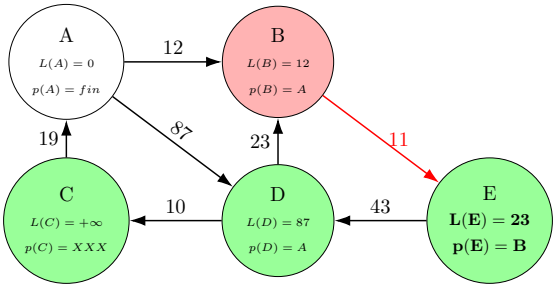

FIGURE 5 – Itération 2 - `A` a été traité, on développe le nœud de valeur minimale dans `Q` (ici `B`) et on met à jour ses voisins (ici `E`).

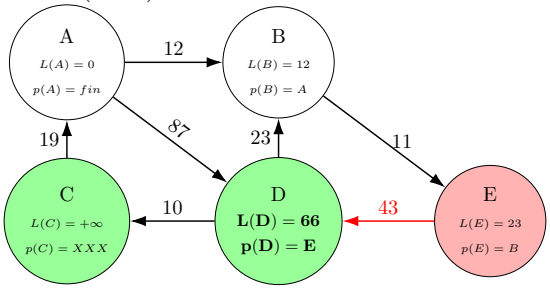

FIGURE 6 – Itération 3 - `B` a été traité, on développe le nœud de valeur minimale dans `Q` (ici `E`) et on met à jour ses voisins. Le nouveau chemin menant à `D` est meilleur et est mis à jour.

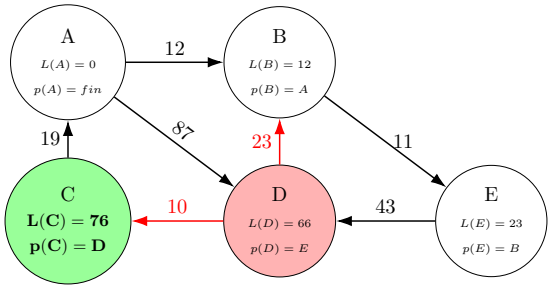

FIGURE 7 – Itération 4 - `E` a été traité, on développe le nœud de valeur minimale dans `Q` (ici `D`) et on met à jour ses voisins `C` et `B`. Le nouveau chemin menant à `B` n'est pas meilleur (cela n'est pas possible car `B` a déjà été traité).

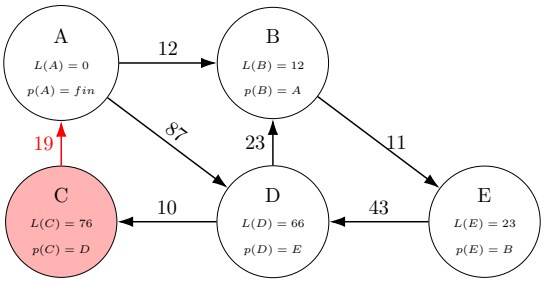

FIGURE 8 – Itération 5 - `C` est le dernier nœud à traiter et ne peut conduire à aucune amélioration.

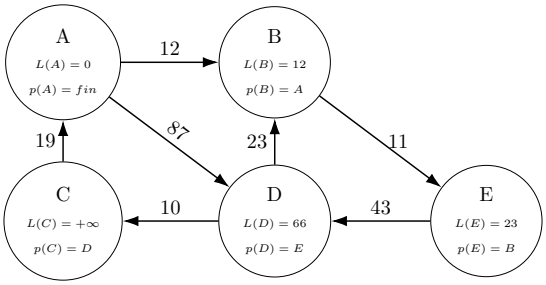

FIGURE 9 – Itération 6 - `Q` est vide, l'algorithme s'arrête. Les valeurs des parents des nœuds permettent de reconstruire le chemin pour aller vers n'importe quel nœud du graphe en partant de `A`. Par exemple pour `C` : `C` ← `D` ← `E` ← `B` ← `A` ← `fin`.

4.2 Algorithme de Dijkstra

On vous fournit ci-dessous une version de l'algorithme de Dijkstra tel que présenté sur la page wikipedia anglaise associée à cet algorithme [4].

Dans cet algorithme `Q` représente la liste des nœuds à traiter. La première partie de l'algorithme (boucle pour) construit la liste `Q` et une fonction de valeur initiale. La seconde partie (boucle tant que) traite les nœuds de cette liste un à un en choisissant à chaque fois le nœud avec la distance calculée minimale.

4. `https://en.wikipedia.org/wiki/Dijkstra's_algorithm`

```
Entrées :
G un graphe orienté avec une pondération (poids) positive des arcs
A un sommet (départ) de G

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

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

Question 19

Dans une classe `Dijkstra`, recopier cet algorithme en commentaire puis traduisez ces lignes en java pour écrire la méthode `Valeur resoudre(Graphe g, String depart)` qui prend en paramètre le nom du nœud de départ pour calculer les plus courts chemins vers les autres nœuds du graphe avec l'algorithme de Dijkstra.

Remarque : Vous noterez que les deux algorithmes implémentés (l'algorithme du point fixe et Dijkstra) possèdent une méthode de même nom, il est donc possible de définir une interface `Algorithme` qui permet de choisir l'algorithme à utiliser selon une approche `Strategie` tel que vu en COO. À partir de cette interface, de nouveaux algorithmes permettant d'effectuer une recherche de chemin pourront donc être facilement insérés dans une architecture logicielle en implémentant cette interface.

4.3 Validation Dijkstra

Question 20

De la même manière que pour l'algorithme du point fixe, écrire des tests unitaires pour vérifier le bon fonctionnement de votre algorithme.

4.4 Programme principal

Question 21

Écrire un programme principal `MainDijkstra` permettant :

- la lecture des graphes à partir de fichiers texte ;
- le calcul des chemins les plus courts pour des nœuds donnés ;
- l'affichage des chemins pour des nœuds données.

5 Validation et expérimentation (4h)

L'objectif de la partie suivante est d'étudier expérimentalement la complexité de ces deux algorithmes (comme fait dans la SAE 1.02 en structures de données). Vous lancerez pour cela plusieurs tests sur des graphes de taille variable pour générer des statistiques sur le temps nécessaire et en tirer des conclusion.

5.1 Comportement qualitatif des algorithmes

On vous donne le graphe `boucle` en format `GraphViz` (illustré par la figure 10) aussi disponible sur arche sous format texte pour chargement.

FICHIER

```
digraph{
rankdir ="LR"
A -> B [label = 20]
A -> D [label = 3]
D -> C [label = 4]
C -> B [label = 2]
B -> G [label = 10]
G -> F [label = 5]
F -> E [label = 3]
}
```

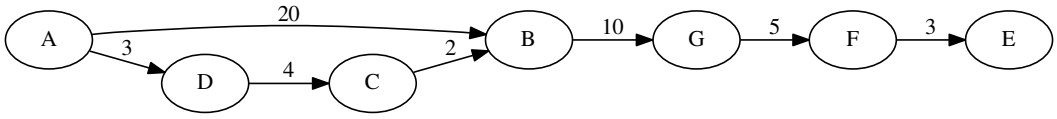

FIGURE 10 – représentation graphique du graphe `boucle`.

Question 22
Afficher le contenu de l'objet `valeur` après chaque itération de chacun des algorithmes. Expliquer la différence de comportement entre les deux algorithmes.

Question 23
Tirer des conclusions à partir de vos observations.

5.2 Graphes de tailles différentes

Tester vos algorithmes sur les graphes fournis sur arche et présenter un tableau synthétique présentant le temps de calcul pour chacune des approches (Point Fixe / Dijkstra).

Question 24
Lequel des deux algorithmes semble le plus efficace et pourquoi selon vous ?

5.3 Génération de graphes

Afin d'avoir une analyse plus fine, on souhaite générer des graphes automatiquement.

Question 25
Réfléchir et proposer une approche pour générer des graphes automatiquement d'une taille donnée (par exemple 1000 nœuds) en choisissant un départ et une arrivée.

Essayez de trouver un moyen pour avoir des graphes fortement connectés (éviter que de nombreux nœuds soient isolés) et avoir un chemin du nœud de départ au nœud d'arrivée (une manière de faire est d'ajouter au moins des connec-

tions entre chaque nœud et le nœud suivant créé pour avoir au moins un chemin qui relie tous les nœuds.).

Question 26
Ajouter dans votre rapport le rendu `GraphViz` de quelques graphes générés (de petite taille) pour présenter les structures de vos graphes.

5.4 Comparaison de résultats et conclusion

Question 27
A l'aide de votre générateur de graphes, comparer les résultats obtenus en fonction de la taille du graphe. Quel est l'algorithme le plus efficace selon ces résultats ?

Question 28
Quel est le ratio de performance entre les deux algorithmes en fonction du nombre de nœuds ? Ce rapport est-il constant ou dépend-il du nombre de nœuds ?.

Question 29
Quelle conclusion pouvez-vous tirer de cette étude ?

6 Extension : Intelligence Artificielle et labyrinthe

Les graphes que nous avons utilisés sont des graphes construits manuellement. Cependant, les algorithmes proposés sont à la base de nombreuses approches en intelligence artificielle (typiquement dans la construction de comportement de personnage dans les jeux vidéos, dans le cadre de la robotique – on parle de planification de trajectoire - et dans beaucoup d'autres domaines).

Dans cette partie, on propose d'appliquer ces algorithmes à la recherche de chemin dans des labyrinthes.

Cette partie est plus ouverte et bien que faisant partie de la SAE, elle n'est à faire que lorsque les parties précédentes auront été faites (une partie du barème sera dédiée à cette partie – mais il sera possible d'avoir une bonne note sans faire cette partie).

6.1 Classe Labyrinthe

On vous fournit la classe `Labyrinthe` proche de celle fournie pour Zeldiablo. Cette classe possède une méthode `int[] deplacerPerso(int i, int j,String action)` qui retourne les coordonnées de la case d'arrivée `(x,y)` sous la forme d'un tableau lorsqu'on part de la case de départ `(i,j)` et qu'on effectue l'acton `action`. Cette classe possède une liste de murs et considère la présence de mur lorsqu'on se déplace. Elle possède un constructeur permettant de construire un labyrinthe à partir d'un fichier texte.

Un objet `labyrinthe` représente ainsi **un graphe de manière implicite** (cf. figure 11).

- Un nœud de ce graphe correspond à une position dans le labyrinthe.
- Un arc correspond à l'exécution d'une action (ici un déplacement d'une case) et relie la case de départ avec la case d'arrivée après le déplacement.
- On considérera que les coûts sont tous les mêmes pour tous les déplacements (on mettra un coût de 1 par défaut - la distance correspondra donc au nombre d'actions à effectuer pour atteindre cette case).

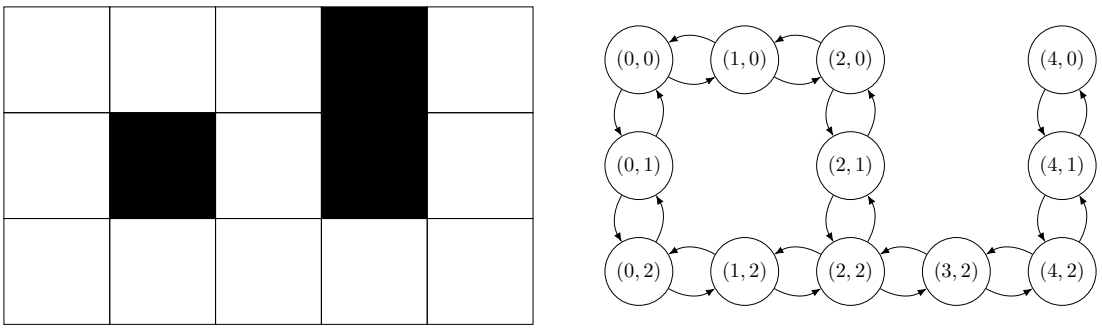

FIGURE 11 – Exemple de labyrinthe (à gauche) et graphe correspondant généré (à droite). Tous les coûts des arcs sont de 1 et ne sont pas représentés. Vous ferez attention aux noms des nœuds, ici représentés sur le graphe.

On souhaite pouvoir trouver le chemin le plus court dans le labyrinthe à l'aide des algorithmes développés sans modifier ces algorithmes et en tirant parti de la notion d'interface.

Il est possible d'effectuer cela de deux manières :

- utiliser la classe `Labyrinthe` pour générer un objet de type `GrapheListe` ;
- faire le lien entre l'interface `Graphe` et la classe `Labyrinthe` avec un `Adaptateur` comme vu en COO (c'est tout l'intérêt de l'adaptateur).

6.2 Génération de graphe

On souhaite générer un objet de type `GrapheListe` à partir d'un objet `Labyrinthe`.

- les nœuds sont nommés selon les coordonnées de la case (par exemple « `(1,1)` ») ;
- les arcs correspondent aux différents déplacements possibles et mènent à la case d'arrivée du déplacement (on ne stockera pas les actions associées aux arcs, mais cela pourrait constituer une autre extension à votre classe `Graphe`).

Question 30

Dans la classe `Labyrinthe` fournie, écrire une méthode `genererGraphe`qui retourne le graphe associé au labyrinthe.

Question 31

Effectuer une recherche de chemin sur le graphe généré et vérifier que ce chemin correspond bien à un des chemins les plus courts dans le labyrinthe pour relier l'entrée à la sortie.

6.3 Implémentation de Graphe

Une autre solution consiste à implémenter un objet `Graphe` particulier `GrapheLabyrinthe` qui utilise un objet `labyrinthe`. Lorsque la méthode `suivants(String n)` du graphe est appelée, l'objet `GrapheLabyrinthe` utilise le labyrinthe (et les actions possibles) pour retourner la liste des arcs utilisables.

Il est ainsi possible de **simuler un graphe** sans avoir besoin de générer un objet graphe mais en utilisant une solution de type **adaptateur** (cf cours de COO).

Question 32

À l'aide de ce second type d'approche et des algorithmes que vous avez implémentés, trouver le plus court chemin dans le labyrinthe.

6.4 Bonus : Recherche complexe et Intelligence artificielle

Ces deux approches s'appliquent sur le labyrinthe fourni mais aussi sur tout problème qu'on peut définir par des états et des actions : c'est **une des bases de l'intelligence artificielle** pour produire des agents intelligents capables de prendre des décisions de manière automatique.

Il est donc possible d'appliquer cette recherche de chemin à des problèmes beaucoup plus complexes. S'il vous reste du temps, vous pouvez tenter de trouver le chemin

le plus court lorsque le personnage se trouve dans un labyrinthe couvert de glace. Lorsqu'il se déplace, le personnage ne peut pas s'arrêter avant d'avoir rencontré un mur. Ce comportement correspond exactement au comportement développé lors du projet effectué dans le module `Qualité de développement`. Vous pouvez donc très facilement réutiliser ce que vous avez fait pour trouver le chemin le plus court (ce qui n'est pas toujours évident).

De la même manière, on peut facilement rendre le problème plus complexe :

- on peut ajouter des portes et des clefs dans le labyrinthe, l'état du jeu à un instant donné correspond à la position du personnage mais inclut aussi les clefs qu'il possède et qui lui permettent de traverser certaines portes.
- pour les étudiants qui ont programmé un jeu de type Sokoban dans la SAE Zeldiablo, vous pouvez utiliser ce jeu pour générer un graphe représentant les évolutions possibles du jeu (un nœud du graphe correspond à la position du personnage et des caisses à un instant donné). Trouver le meilleur chemin dans ce graphe produit un comportement intelligent consistant à pousser correctement les caisses pour les amener à leur emplacement.

Dès que vous pouvez programmer un problème de ce type, vous pouvez avec un adaptateur et les algorithmes de cette SAE construire une solution à ce problème. La difficulté vient du fait que votre graphe peut facilement atteindre un très grand nombre de noeuds (par exemple un jeu de sokoban sur un labyrinthe de 10x10 avec 2 caisses donne **un million de nœuds différents** : 100 possibilités pour la position du personnage fois 100 possibilités pour la position de la première caisse fois 100 possibilités pour la position de la seconde caisse). Très vite d'autres algorithmes sont nécessaires pour résoudre ces problèmes et c'est tout un pan de la recherche en Intelligence Artificielle que de proposer des algorithmes plus efficaces adaptés au problème spécifique à traiter.

7 Rendu attendu

Les **éléments suivants** sont attendus comme rendu

- un **dépôt** `git` dans lequel votre enseignant a été invité :
  - ce dépôt `git` doit être correctement structuré (par de fichier de configuration ni de fichier `.class`) ;
  - un fichier `README.md` à la racine du projet qui présente les noms des membres du binôme.
- un **code source** `Java` :
  - ce code source doit **impérativement** compiler ;
  - ce code doit être clair, correctement commenté, contenir une javadoc et contenir en commentaire les algorithmes à la base de votre programme ;
  - ce code doit aussi contenir des tests unitaires `JUnit` vous ayant permis de valider votre application.

- un **rapport (en pdf)** à la racine de votre dépôt :
  - ce rapport doit être structuré par partie en respectant les même noms de parties que le sujet (Représentation d'un graphe, point fixe, Dijkstra, Validation, Labyrinthe)
  - dans chaque partie, ce rapport doit faire ressortir le travail réalisé en précisant le code qui a été produit (quelle classe) et ce qui a été testé ;
  - le rapport doit aussi fournir dans chaque partie les réponses aux différentes questions qui ne correspondent pas à une production de code. A chaque fois n'oubliez pas de nommer la question à laquelle vous répondez (par exemple `[Question 13] ...`)
  - le rapport doit contenir une conclusion générale sur cette SAE qui mette en avant ce que vous avez appris, les difficultés rencontrées et le bilan à tirer de ce travail.