Rapport de mini-projet graphes
Mise en oeuvre des algorithmes de Dijkstra et de Warshall

Maëlig NANTEL
ENSSAT, IMR 2ème année

Introduction

Un graphe est un ensemble de sommets reliés ensemble par des arcs. Ils modélisent de nombreuses situations où interviennent des objets en interactions. Par exemple les interconnexions routières entre différentes agglomérations. En mathématiques la théorie des graphes propose un grand nombre d'algorithmes permettant de résoudre des problèmes connus pouvant être modélisés par des graphes.

L'objectif de ce mini-projet académique est de mettre en œuvre, dans le langage de notre choix, les algorithmes de Dijkstra et de Warshall. Ces algorithmes ont été préalablement étudiés en cours théoriques.

Rappel de notions fondamentales de la théorie des graphes :

- Un graphe est dit connexe s'il existe un chemin entre chaque sommet, c'est-à-dire que l'on peut atteindre n'importe quel sommet du graphe à partir de n'importe quel autre.

- Un graphe est dit valué si les arcs reliant deux sommets sont porteurs d'un poids. Ce poids sera ici un entier, son sens dépend entièrement du contexte d'étude du graphe. Dans le cas d'un réseau de transport entre deux villes, il peut par exemple s'agir de la distance entre les deux agglomérations. Un graphe non valué peut être vu comme un graphe valué avec tous les poids égaux.

- Un graphe est dit orienté si un arc qui relie deux sommets a un sens de parcours imposé. On peut donc dire qu'un graphe non orienté est un graphe orienté avec un arc de parcours à deux sens (donc deux arcs A → B et B→ A).

- Un cycle dans un graphe est un chemin, dont le sommet de départ, et d'arriver sont identiques

Pour ce mini-projet, j'ai choisi d'implémenter mes solutions en programmation-objet, avec le langage Java. Les scénarios de tests seront effectués avec la librairie JUnit. J'ai acquis de l'expérience en Java durant mes périodes en entreprises. J'ai donc préféré utiliser un langage connu afin de me concentrer sur un seul point d'apprentissage à la fois (apprendre l'algorithmique sur les graphes séparément d'un nouveau langage). De plus, j'ai déjà été amené à travailler avec le langage Python, j'en maîtrise donc les bases.

Partie 1 - Modélisation de la structure graphe

Avant de chercher à développer les algorithmes, il est nécessaire de réfléchir à la mise en œuvre de la structure de donnée permettant de stocker un graphe. Ici l'objectif n'est pas de chercher une bibliothèque existante permettant de répondre à notre besoin, mais bien de créer notre propre solution.

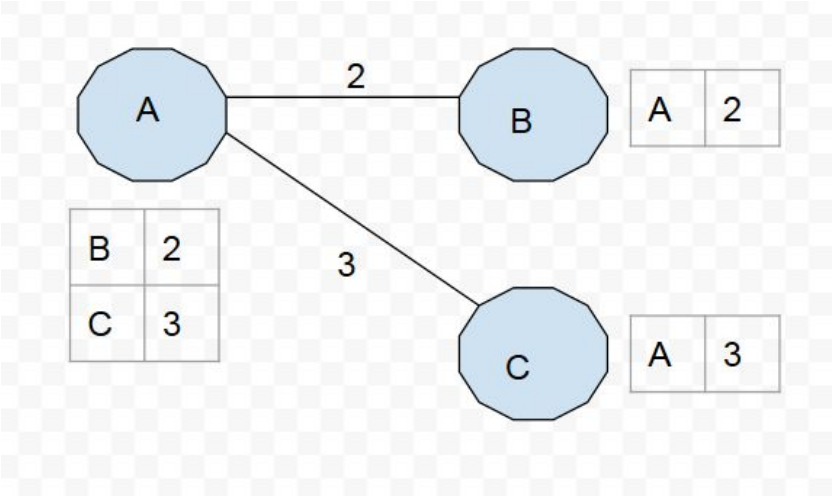
Le type abstrait (c'est-à-dire les fonctionnalités visibles depuis l'extérieur) d'un graphe est le suivant :

```java
/** Retourne vrai si le graphe est orienté. */
boolean estOriente();

/** Retourne vrai si le graphe est valué. */
boolean estValue();

/** Retourne vrai si le graphe est vide (ne comporte aucun sommet). */
boolean estVide();

/** Retourne la taille du graphe (nombre total de sommets). */
int taille();

// GESTION DES SOMMETS

/** Ajoute un sommet au graphe. Pas de doublons possibles. */
void ajouterSommet(Sommet sommet);

/** Supprime un sommet existant du graphe. Supprime également toutes les liaisons associées à ce sommet. */
void retirerSommet(Sommet sommet);

/** Retourne l'ensemble des sommets du graphe. */
Set<Sommet> getSommets();

/** Retourne l'enssemble des sommets directement accéssibles depuis le sommet 'source' */
Set<Sommet> getSommetsAdjacents(Sommet source);

/** Retourne le degré d'un sommet. Degré = Degré entrant + degré sortant */
int getDegre(Sommet sommet);

/** Retourne le degré sortant d'un sommet */
int getDegreSortant(Sommet sommet);

/** Retourne le degré entrant d'un sommet */
int getDegreEntrant(Sommet sommet);

/** Retourne vrai si le sommet éxiste dans le graphe */
boolean existeSommet(Sommet sommet);

/** Retourne l'enssemble des sommets atteignables depuis le sommet 's' */
Set<Sommet> getSuccesseurs(Sommet s);

/** Retourne l'ensemble des sommets permettant d'atteindre le sommet 's'. */
Set<Sommet> getPredecesseurs(Sommet s);

// GESTION DES ARCS

/** Retourne le poids de la liaison entre 'source' et 'destination' */
int getPoids(Sommet source, Sommet destination);

/** Ajoute une liaison entre 'source' et 'destination' (deux liaisons si le graphe est non orienté) */
void ajouterArc(Sommet source, Sommet destination, int poids);

/** Supprime la liaison existante entre 'source' et 'destination' */
void retirerArc(Sommet source, Sommet destination);

/** Retourne vrai si il éxiste un arc entre 'source' et 'destination'. */
boolean existeArc(Sommet source, Sommet destination);
```

Pour ce projet nous n'avons pas besoin de l'ensemble de ces fonctionnalités, elles ne seront donc pas toute implémentées. La modélisation retenue devra toutefois permettre de les développer facilement a posteriori.

À première vue, on peut être tenté de créer deux objets « Sommet » et « Arc » et de les stocker dans des listes. Nous aurons effectivement bien accès à l'ensemble des sommets et à l'ensemble des arcs. Cependant, on peut vite se rendre compte que nous ne mémorisons aucun lien entre les sommets, ce qui impose donc de parcourir très souvent l'ensemble de ces listes lors d'opérations sur le graphe. Ce n'est pas la proposition à retenir.

Une meilleure solution est d'utiliser la notion de liste d'adjacence pour représenter le graphe. Pour chaque sommet, on mémorise ses sommets adjacents (c'est à dire directement accessibles). Dans le cas d'un graphe valué, on mémorise également le poids associé.



Les sommets adjacents de A étant B et C, dans la liste d'adjacence on mémorise les couples (B,2) et (C,3). Cela indique qu'il existe un arc reliant A à B avec un poids de 2 et un arc reliant A à C avec un poids de 3. Cette modélisation fonctionne également pour un graphe orienté sans demander de modifications.

La modélisation sous forme de liste d'adjacence est bien adaptée pour les graphes ayant un faible nombre d'arcs (c'est-à-dire un graphe peu dense, ou creux). Dans le cas contraire, il serait plus intéressant d'utiliser une matrice d'adjacence, comme étudier en cours.

Partie 2 - Algorithme de Dijkstra

1) Principe

Un problème récurent dans la théorie des graphes et de trouver le meilleur chemin à emprunter pour se rendre d'un point à un autre. Sur l'exemple du réseau routier, l'algorithme de Dijkstra permet de trouver le chemin le plus court entre deux villes.

2) Algorithme

Les pré-conditions de l'algorithme sont:

- Disposer d'un graphe valué non vide (orienté ou non)
- Toutes les valuations doivent être positives ou nulles (pas de poids négatifs)
- Disposer d'un sommet source présent dans le graphe

L'algorithme de Dijkstra est un algorithme dit "glouton", c'est-à-dire qu'il calcule des optimums locaux successifs pour finir par avoir le résultat optimum global. On traite chaque sommet du graphe, et on n'y reviendra jamais par la suite. A chaque étape, on connaît la meilleure distance connue pour se rendre du sommet source à un autre. S'il existe un meilleur chemin, il n'est pas encore connu.

En pseudo-code, l'algorithme tel que vu en cours est:

X = {1,...,n} l'ensemble des sommets du graphe
S = 1 le sommet de départ
P[i,j] = poids de l'arc (i,j)
D[i] = plus courte distance courante de s(i) à i

```
Debut
   E = {1}
   pour i de 2 à n faire
      D[i] <- P[1,i]
   fin

   pour i de 2 à n faire
      choisir t parmi X-E tel que D[t] soit min
      ajouter t à E
      pour chaque S sommet adjacent de t faire
         D[x] = min( D[x] , D[t] + P[x,t] )
      fin
   fin
fin
```

J'ai choisi de modifier légèrement le déroulement. Au lieu de mémoriser les sommets explorés, on part de l'ensemble des sommets du graphe et on retire ceux que l'on a traités. Le principe reste strictement identique.

3) Complexité

Dans l'algorithme de Dijkstra, l'opération coûteuse est la comparaison des distances. On cherche donc à analyser combien de fois maximum on va effectuer cette opération pour déterminer la complexité globale de l'algorithme. On ne s'occupe pas des autres opérations qui sont considérées comme négligeables. On signalera tout de même que la structure de Map Java est indexée et permet donc un accès aux éléments en temps quasi constant.

Soit n le nombre total de sommets du graphe. On exécute n-1 fois (on retire la source) la boucle principale de l'algorithme.

```
while ( !nonExplore.isEmpty() ) {
    // Récupère le sommet le plus proche et n'ayant pas encore été exploré
    Sommet sommet = getSommetLePlusProche(nonExplore);
    nonExplore.remove(sommet);
    parcourirSommetsAdjacents(sommet);
}
```

Le nombre de comparaisons total sera donc (n-1) * le nombre d'opérations maximales à l'intérieur de la boucle.
On analyse donc les deux méthodes appelées :

```
private void parcourirSommetsAdjacents(Sommet sommet)
{
    for ( Sommet adjacent : graphe.getSommetsAdjacents(sommet) ) {

        // distance = distance déjà parcourue pour arriver à 'sommet' + la distance pour se rendre de 'sommet' à 'adjacent'
        int distanceDepuisAdjacent = distance.get(sommet) + graphe.getPoids(sommet, adjacent);
        if ( distanceDepuisAdjacent < distance.get(adjacent) ) {
            // Signifie qu'on a trouvé un meilleur chemin
            distance.put(adjacent, distanceDepuisAdjacent);
            predecesseurs.put(adjacent, sommet);
        }
    }
}
```

Ici, dans le "pire" (dans le sens où il y aura le plus de sommets à parcourir) cas, chaque sommet est relié à tous les autres sommets du graphe. Il y aurait donc n-1 sommets adjacents, donc autant de comparaisons à effectuer.

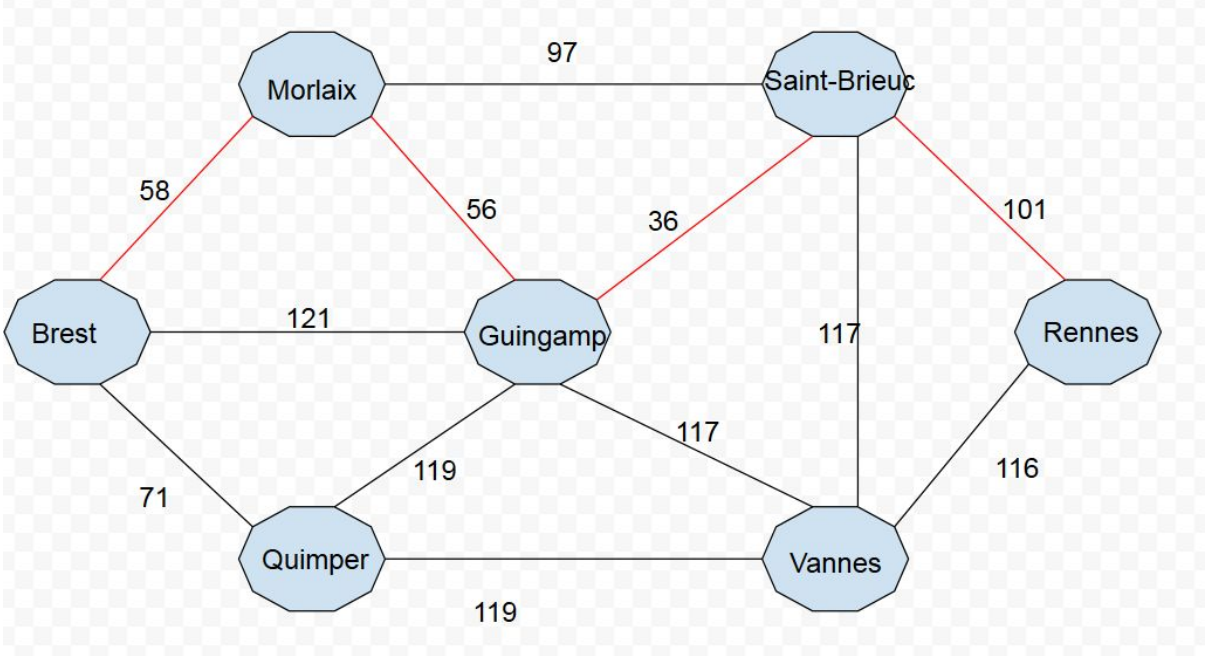
```java
private Sommet getSommetLePlusProche(List<Sommet> sommets)
{
    assert !sommets.isEmpty(); // précondition
    Iterator<Sommet> i = sommets.iterator();
    Sommet plusProche = i.next();
    while ( i.hasNext() ) {
        // INVARIANT: plusProche est le proche sommet de "sommets" entre le début (inclus) et i (exclus)
        Sommet candidat = i.next();
        if ( distance.get(candidat) < distance.get(plusProche) ) {
            plusProche = candidat;
        }
    }
    return plusProche;
}
```

Ici on parcourt au plus n-1 sommets restants.

On arrive donc à un nombre maximal d'opérations de $(n-1) * ((n-1)+(n-1))$ ce qui donne une complexité en $O(n^2)$, les constantes étant sorties.

4) Tests

On considère le graphe suivant; les sommets représentent des villes Bretonnes, les arcs des liaisons routières et les valuations la distance en km séparant les deux villes.



On souhaite trouver le plus court chemin pour se rendre de Brest à Rennes. La première étape est de calculer les plus courts chemins depuis Brest pour joindre toutes les autres villes (exécuter l'algorithme de Dijkstra). Ensuite, il sera possible de calculer le chemin à emprunter pour aller de Brest à Rennes.

Pour tester efficacement, une fois que le cas nominal d'exécution est validé, il est intéressant de chercher les cas limites pouvant poser problème à l'algorithme. Ici on identifie :

- Le chemin pour aller de la source vers la source
- Le chemin pour atteindre une destination qui ne se trouve pas dans le graphe

Les tests ont été implémentés avec la librairie Java JUnit et sont disponibles dans le fichier « DijkstraTest.java ».

Partie 3 - Algorithme de Warshall

1) Principe

L'objectif de l'algorithme de Warshall est de construire la fermeture transitive d'un graphe, orienté ou non. Si dans le graphe il existe un chemin pour se rendre d'un sommet A à un sommet B, alors on ajoute un arc (s'il n'existe pas déjà) entre A et B. Le calcul de la fermeture transitive d'un graphe permet donc de savoir s'il existe au moins un chemin entre deux sommets du graphe.

Une variante permet d'obtenir le routage entre deux sommets, c'est-à-dire de trouver un chemin pour se rendre entre deux sommets. Contrairement à l'algorithme de Dijkstra, ici il ne s'agit pas forcément du chemin le plus court, mais « d'un chemin ».

2) Algorithme

Soit n le nombre de sommets du graphe. Le pseudo-code de l'algorithme de Warshall est le suivant :

```
Pour k de 1 à n faire
   Pour i de 1 à n faire
      Si il existe un arc(G+,x,i) alors
         Pour j de 1 à n faire
            Si il existe un arc(G+,i,y) alors
               Ajouter un arc de i à j si il n'existe pas
            Fsi
         Fin
      Fsi
   Fin
Fin
```

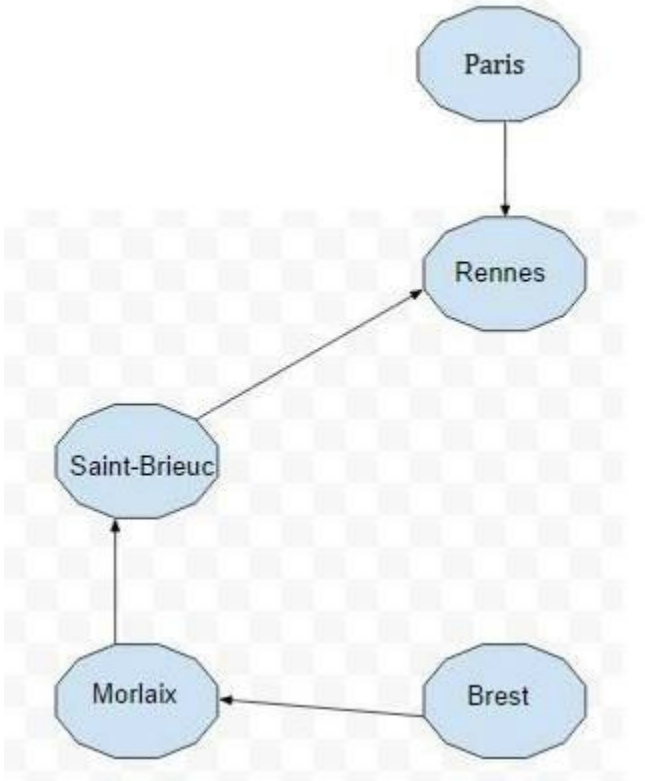
Pour la variante avec routage, on ajoutera alors le prédécesseur au moment de l'ajout de l'arc. On ajoute donc un arc de i à j avec un prédécesseur k.

3) Complexité

Ici, l'opération coûteuse est de regarder s'il existe un arc entre deux sommets. On identifie facilement que cette opération sera effectuée au plus une fois dans chacune des boucles, on est donc avec une complexité d'ordre O(n3).

4) Tests

Soit le graphe suivant:

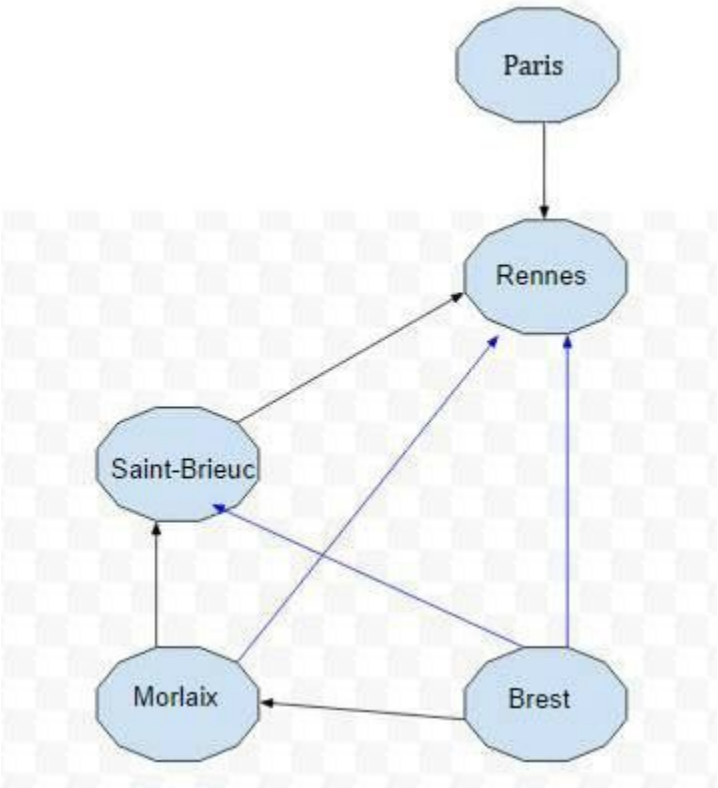


On y applique la fermeture transitive grâce à l'algorithme de Warshall. On va donc déterminer, à partir de chaque ville, où il est possible de se rendre.

On attend bien que l'algorithme rajoute les chemins :

- Brest vers Saint-Brieuc (puisqu'atteignable en passant par Morlaix)
- Brest vers Rennes (puisqu'atteignable en passant par Morlaix puis Saint-Brieuc)
- Morlaix vers Rennes (puisqu'atteignable en passant par Saint-Brieuc)

Aucun arc au départ de Paris ne doit être ajouté, étant donné qu'il n'existe aucune route depuis Rennes pour se rendre à Saint-Brieuc, Morlaix ou Brest (il s'agit bien évidemment d'un exemple non réaliste !).



Mon implémentation ne créer pas un nouveau graphe représentant la fermeture transitive, mais modifie le graphe passé en paramètre. C'est un choix personnel. Renvoyer un nouveau graphe ne serait pas très compliqué, pour cela il faut rajouter un constructeur de Graphe par copie. Pour vérifier le résultat, il est nécessaire de créer un graphe « résultat attendu » et de le comparer au graphe modifié après l'application de l'algorithme de Warshall.

On cherche là encore à identifier les cas un peu particuliers pouvant poser problème à l'algorithme. Je n'ai identifié que le cas où tous les sommets du graphe sont directement joignables entre eux. L'algorithme ne doit alors rien modifier au graphe.

Les tests implémentés sont disponibles dans le fichier « WarshallAlgorithmTest.java ».