ENSSAT
LANNION

Projet Graphes

Dijkstra

&

Warshall

---

me [at] colinleverger [dot] fr

Colin Leverger - ENSSAT Informatique, Multimédia et Réseaux
Promotion 2017

Destinataire : Daniel Rocacher

28 novembre 2015

Introduction

Dans le cadre de la formation Informatique, Multimédia et Réseaux dispensé à l'ENSSAT de Lannion, nous avons étudié la théorie des Graphes en seconde année. Il s'agit de réfléchir à la meilleure manière de rejoindre deux points en choisissant la distance la plus courte et le chemin le plus efficace. Le but est aussi d'écrire des algorithmes permettant d'appliquer ces méthodes via l'informatique, pour des traitements rapides et automatisés.

Différents algorithmes peuvent être utilisés pour ce faire. Les algorithmes que nous allons étudier dans ce mini projet seront ceux de Warshall et de Dijkstra. Ils permettent respectivement de calculer la fermeture transitive d'un graphe, et de calculer la plus courte distance et le chemin à emprunter pour se rendre à un point B depuis un point de départ A.

Ce rapport explicite la méthode employée pour développer les algorithmes. Je vais dans une première partie présenter succinctement Scala, le langage que je vais utiliser. La structure de donnée et les fonctions utilisées seront explicitées dans une deuxième partie. Je vais expliquer dans une troisième partie ma démarche pour coder Warshall, et dans une quatrième partie ma démarche pour coder Dijkstra. Je vais enfin conclure sur ce projet.

Table des matières

Partie 1

Scala

1.1 Rapide présentation

Scala est un langage inventé par Martin Odersky à l'École polytechnique fédérale de Lausanne (EPFL) en 2003. Scala est un dérivé du mot «Scalable» ; son nom peut signifier « langage qui peut être mis à l'échelle ». Il permet au programmeur de coder de manière conscice et élégante.

Scala s'exécute sur une JVM Java, c'est donc du bytecode qui est compilé. Scala utilise son propre compilateur (Scalac). Les principales différences entre Scala et Java sont les suivantes :

- Scala permet d'écrire du code succinct et élégant. Scala n'est pas verbeux, contrairement à son grand frère Java. Une «{» est égale à une «(« ; pas d'utilisation de points virgules.
- Scala est multi paradigmes.
- L'utilisation de variables immutables est très fortement encouragée.
- Les méthodes / fonctions définies en Scala sont vues comme des valeurs.
- L'héritage multiple est autorisé en Scala.

1.2 Les paradigmes et le workflow

Scala est un language impur. C'est un langage qui est à la fois :

- Fonctionnel.
- Object.

— Impératif.

Son typage est fort et inféré. Il est possible d'utiliser du Pattern Matching.

Scala laisse le choix au développeur du paradigme à utiliser en fonction du problème qui se pose. La plupart du temps, les problèmes peuvent se résoudre en codant purement fonctionnel. Cela rend souvent le développement plus lent dans mon cas, car je ne suis pas très familier avec le développement fonctionnel. Mais la qualité du code est supérieure, notamment grâce à l'utilisation de «valeurs». Ces valeurs, une fois attribuées, ne peuvent pas être modifiées, et cela apporte une grande sécurité lors de l'exécution.

Dans le cadre de ce TP, j'ai codé un maximum en fonctionnel (immutabilité), mais pour simplifier les modifications de structures, j'ai utilisé quelques éléments mutables. (Notamment des Maps mutables, notées MMap[...]).

Le workflow Scala est très simple : coder les tests pour chaque fonction, coder les fonctions elles-mêmes, tester, recommencer. Travailler en «continuous delivery» est aisé avec ce langage, grâce à l'outil «SBT» (Simple Build Tools). Cet outil permet de gérer les dépendances, tester et déployer de manière automatisée, à l'aide de commandes telles que :

— sbt test : exécute tous les tests unitaires présents dans le dossier test. C'est la commande que je vais utiliser pour valider mes fonctions.
— sbt run : run le main. Il n'y a pas de main dans ce projet (pas besoin !)
— sbt gen-idea : permets de récupérer et de mettre à disposition les dépendances utilisées dans le projet de manière automatique (et définie dans un fichier de build.sbt). Génère ensuite le workspace IntelliJ correspondant.

1.3 Les spécificités du langage utilisé dans ce TP

— Scala Collections : librairie native qui permet d'effectuer des traitements sur les structures de données (*List*, *Map*,...). Permet de récupérer toutes les clefs d'une Map, permet de transformer une List en Map... Les possibilités sont énormes, et l'utilisation simplifiée (codage concis et lisible).
— Scala test + Should matchers : tests unitaires Scala. C'est avec ceci que j'ai vérifié chacun des algorithmes codés. La sémantique est simple : «*l'exec de la fonction() should be (1)*», autrement dit «*le résultat renvoyé par cette fonction doit être (1)*».
— Case class : ce sont des classes java extended ; plus besoin de constructeur, plus besoin de getters ni de setters. (Note : une *case class* ne devrait pas contenir une référence vers une autre case class. C'est pour ça que ma classe «Graph» n'en est pas une.)
— For/yield : permet d'itérer sur une structure de donnée et d'en construire une autre à la volée, en «yieldant» chaque résultat.

Partie 2

Détails d'implémentation

2.1 Type Abstrait de Donnée

Il existe plusieurs manières de représenter un Graphe. J'ai choisi de fonctionner comme suis :

- Un graphe sera représenté par une classe «Graph», qui contiendra typiquement une liste de Nœuds. Il pourra ressembler à : *Graph(List[n1,n2])* dans le cas d'un graphe avec deux nœuds n1 et n2.
- Un nœud sera représenté par une classe «Node». Chaque nœud contiendra un numéro, et une liste de successeurs. Typiquement, si le nœud 1 est connecté avec le nœud 2, et que la distance les séparant est 3, on aura un nœud qui ressemblera à : *Node(1, MMap[2 ->3])*[1].

La structure de donnée (aka liste d'adjacence) que j'ai choisie est certainement moins simple que l'utilisation d'une matrice. En effet, on peut tout à fait représenter les graphes par des matrices d'adjacences, telles que :

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Note : ici, il y a une connexion entre les nœuds 1 et 1, les nœuds 2 et 2, 2 et 3...

Dans le cas de l'utilisation d'une matrice, pour le parcourt, on est en $\theta(n^2)$, car il suffit pour

1. À noter : une MMap est simplement une Map mutable (Hashmap de Java).

récupérer un élément de faire une double boucle (récupérer l'élément dans la bonne ligne et la bonne colonne). On a alors la valeur espérée directement. Dans le cas d'une multiplication entre deux matrices, on est en $\theta(n^3)$.

La solution que j'ai choisie est moins efficace, car il faudra itérer dans tout les successeurs d'un nœud pour trouver la valeur que l'on recherche ; si elle est située en tête de liste, c'est plus rapide, mais si elle est située en queue de liste, il faudra évidemment parcourir vainement toute la liste, ce qui peut être couteux. Dans le cas des exemples et des jeux de test fournis, on ne verra pas énormément de différence (les nœuds ont rarement plus de 3 successeurs...), mais avec des centaines de successeurs cela ferait certainement une grande différence au final.

2.2 Les fonctions utilitaires codées et testées

Pour les besoins du projet et pour simplifier la manipulation des graphes, j'ai codé (et testé, workflow DevOps oblige) un certain nombre de fonctions utilitaires. Ces fonctions permettront de créer un graphe vide, d'ajouter un nœud, de voir qu'il y a une connexion entre les deux nœuds... La liste de ces fonctions est disponible ci-dessous.

```
//FOR THE GRAPHS
def addNode(node: Node)
def delNode(node: Node)
def isEmpty: Boolean
def nbNodes: Int
def printGraph
def isArc(s1: Int, s2: Int): Boolean
def getNode(n: Int): Node
def getNodes: List[Node]
def getSuccessors(node: Node): Map[Int, Int]
def getNodesKeys: List[Int]
def dijkstraWeightBetweenTheseTwoNodes(node1: Int, node2: Int): Int

//FOR THE NODES
def addArc(key: Int, value: Int)
def delArc(keyToRemove: Int)
def getSuccessorsKeys: List[Int]
def getSuccessorsList(graph: Graph): List[Node]
def getSuccessors: Map[Int, Int]
def weightOfThisSuccessor(n: Int): Int
def numberOfSuccessor: Int
def printNode
```

À noter, la plupart des fonctions fournies ici ont été codées et testées, mais n'ont pas été utilisées dans les algorithmes (*addNode*, *delNode*...). Je les ai codés pour avoir un TAD le plus complet possible.

2.3 L'importance du test...

Je voudrais insister sur l'importance du test dans ce type de projet. Il ne s'agit pas uniquement de tester visuellement et de manière non approfondie si l'algorithme final testé renvoie la bonne valeur... Pour s'assurer d'obtenir toujours le résultat escompté, et ce dans n'importe quelles circonstances, les tests doivent être le plus exhaustifs possible sur toutes les fonctions développées, aussi petites soient-elles. Cela permet de s'assurer une non-régression du code, et surtout une maintenabilité améliorée. En effet, une fois testée, une fonction peut être utilisée à un niveau supérieur d'abstraction, et c'est très pratique : je sais que *fonction()* renvoie le bon résultat, je peux l'utiliser telle quelle, sans me soucier de ce qu'il y a derrière !

Un exemple de test qui passe pourra être vu en figure 2.1. Un exemple de test qui ne passe pas pourra être vu en figure 2.2 (j'ai simplement modifié une valeur par rapport aux testOK). L'explication de pourquoi le test ne passe plus est donnée par l'outil (le delta est assez précis : 1 n'est pas égal à 0 sur la ligne 16 du test dans ce cas.)

Le résultat des tests pour le projet tout entier sera fourni dans un fichier à la racine du dossier compressé comprenant ce rapport.

```
1  > testOnly GraphTests
2  [info] Updating {file:/Users/colinleverger/Google\%20Drive/ENSSAT/IMR2/ALGO/
       graphs-scala/}root...
3  [info] Resolving jline#jline;2.12.1 ...
4  [info] Done updating.
5  [info] GraphTests:
6  [info] - add node test
7  [info] - getNodesKeys test
8  [info] - del node test
9  [info] - connexion between nodes test
10 [info] - nbNode test
11 [info] ScalaTest
12 [info] Run completed in 652 milliseconds.
13 [info] Total number of tests run: 5
14 [info] Suites: completed 1, aborted 0
15 [info] Tests: succeeded 5, failed 0, canceled 0, ignored 0, pending 0
16 [info] All tests passed.
17 [info] Passed: Total 5, Failed 0, Errors 0, Passed 5
18 [success] Total time: 14 s, completed Nov 26, 2015 12:02:11 PM
```

Figure 2.1 – Test OK

2.4 Fonctionnel ou Impératif ?

Il est possible en Scala de naviguer entre les deux paradigmes Functionnels et Imparétif. Je n'ai pas de compétence particulière en programmation fonctionnelle, et je n'ai pas eu assez de temps

```
> testOnly GraphTests
[info] Compiling 1 Scala source to /Users/colinleverger/Google Drive/ENSSAT/IMR2
    /ALGO/graphs-scala/target/scala-2.11/test-classes ...
[info] GraphTests:
[info] - add node test *** FAILED ***
[info]   1 was not equal to 0 (GraphTests.scala:16)
[info] - getNodesKeys test
[info] - del node test
[info] - connexion between nodes test
[info] - nbNode test
[info] ScalaTest
[info] Run completed in 285 milliseconds.
[info] Total number of tests run: 5
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 4, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed: Total 5, Failed 1, Errors 0, Passed 4
[error] Failed tests:
[error]         GraphTests
[error] (test:testOnly) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 4 s, completed Nov 26, 2015 12:03:02 PM
```

Figure 2.2 – Test KO

pour décomposer les algorithmes sous cette approche. J'ai donc dans un souci de temps codé tous les algorithmes en Impératif. Je prendrai certainement le temps lors de la future période entreprise pour recoder l'algorithme de Dijkstra en fonctionnel (l'algorithme à l'air assez intéressant, et quand il est basé sur du pattern matching et dans une approche fonctionnelle, il ne fait qu'une cinquaine de lignes !).

Partie 3

Warshall

3.1 L'algorithme utilisé

L'algorithme de Warshall permet de calculer la fermeture transitive d'un graphe. Typiquement, tous les chemins d'une longueur 2 doivent être sous-tendus par un arc. Si a -> b et b -> c, on obtiendra au final a -> c.

L'algorithme utilisé est le suivant :

Algorithm 1: Algorithme de Warshall utilisé

```
Data: G : : Graph
Data: n : : size(Graph)
Result: G+
G+ = G;
for i de 1 à n do
    for x de 1 à n do
        if arc(G+,x,i) then
            for y de 1 à n do
                if arc(G+,i,y) && !(arc[G+,x,y] then
                    addArc[G+,x,y]
                end
            end
        end
    end
end
```

3.2 La complexité

Ici, il y a 3 boucles imbriquées ; ceci implique une complexité en $\theta(n^3)$.

3.3 Les jeux de test

Pour les jeux de tests, j'ai choisi d'utiliser notamment le graphe donné en cours.

Le graphe de départ (G) est visible en figure 3.1.

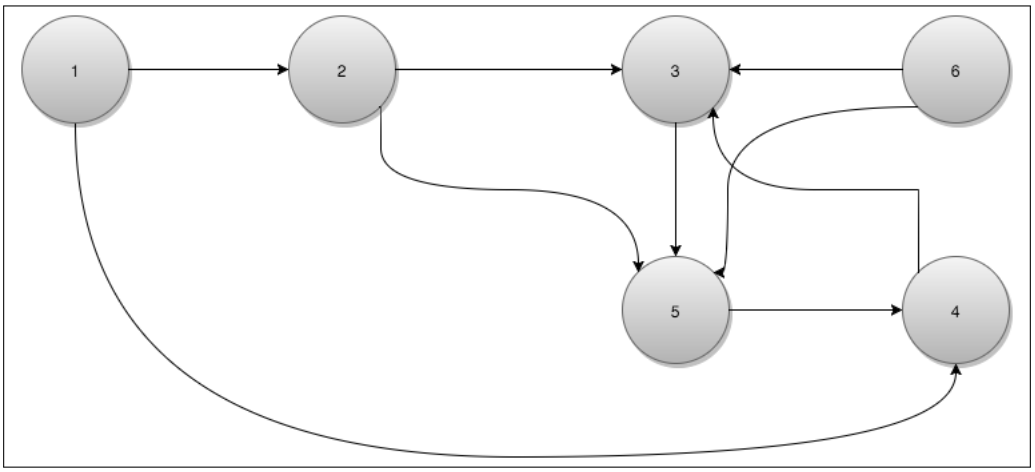

Figure 3.1 – Graphe G de départ

Le graphe d'arrivée (G^+) est visible en figure 3.2 après applications successives des θ(i) en suivant la méthode vue en cours. Les connexions rajoutées par l'algo sont visibles en rouge sur le schéma.

J'ai aussi utilisé le graphe donné en DS. C'est exactement le même principe qui sera appliqué. Je fournis ici les schémas pour le graphe du cours.

Note importante : l'exécution de l'algorithme va modifier le graphe initial donné.

Pour définir les tests, il ne faut pas tomber dans le piège de regarder le résultat de notre fonction et tester en conséquence. J'ai codé les tests avant même de coder Warshall, pour être sûr de récupérer les bons résultats.

Lors des tests, il m'a suffi de vérifier que l'algorithme vérifiait bien la liste des successeurs de chaque nœud. La fonction getSuccessorsKeys() me renvoie le numéro de tout les nodes successeurs

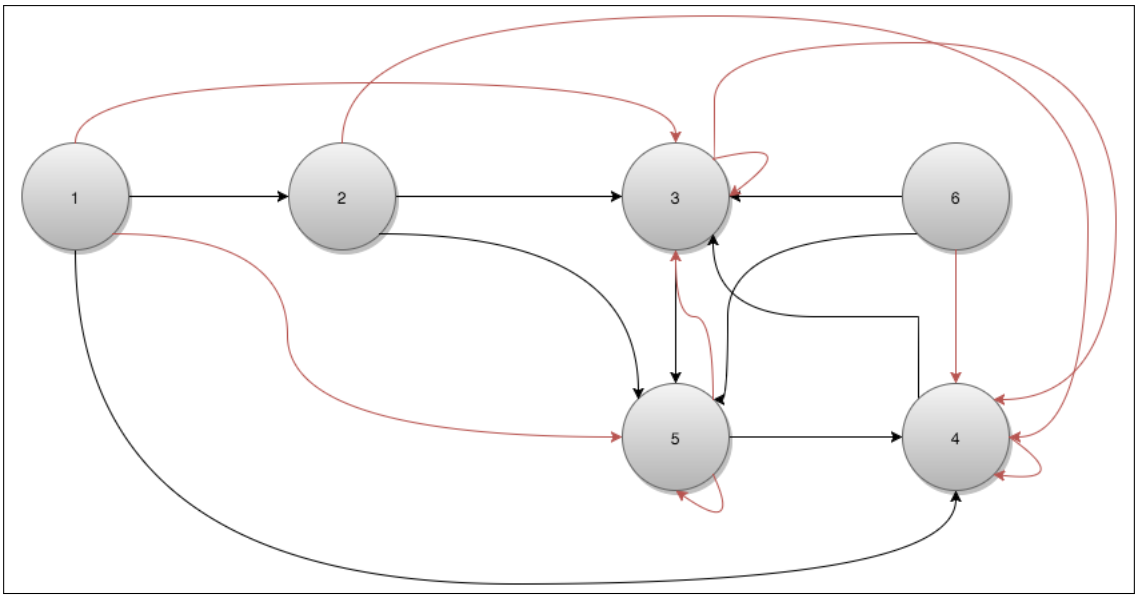

Figure 3.2 – Graphe G^+ d'arrivée

du node actuel, classés du plus petit au plus grand. (fonction testée et utilisable telle quelle).

```
1  class WarshallTests extends FunSuite with Matchers {
2
3    test("Warshall algorithm test - graph provided in the lesson") {
4      initialization();
5      [...]
6      g1.getNode(2).getSuccessorsKeys should be(List(3, 4, 5))
7      g1.getNode(3).getSuccessorsKeys should be(List(3, 4, 5))
8      g1.getNode(4).getSuccessorsKeys should be(List(3, 4, 5))
9      [...]
10   }
11 }
```

Si on compare le test ci-dessus (ligne 6) et le schéma 3.2, on observe qu'effectivement, le nœud 2 est connecté aux nœuds 3, 4 et 5.

Les résultats du test sont visibles dans le code suivant. On observe selon la syntaxe de sbt que tous les tests sont passés.

```
1  > testOnly WarshallTests
2  [info] WarshallTests:
3  [info] - Warshall algorithm test - graph provided in the lesson
4  [info] - Warshall algorithm test - graph provided in the exam
5  [info] ScalaTest
6  [info] Run completed in 159 milliseconds.
7  [info] Total number of tests run: 2
8  [info] Suites: completed 1, aborted 0
9  [info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
10 [info] All tests passed.
11 [info] Passed: Total 2, Failed 0, Errors 0, Passed 2
```

```
[success] Total time: 1 s, completed Nov 21, 2015 2:53:20 PM
```

J'ai aussi codé des tests pour pousser l'algorithme et tester les cas particuliers : et si G est déjà G^+ ? Et si aucun des nœuds du graphe ne sont connecté entre eux ? [1]

3.4 Question bonus : la matrice de routage

Pour la question bonus, il nous était demandé d'ajouter la possibilité de calculer la matrice de routage. Pour rappel, la matrice de routage permet de savoir le chemin parcouru pour arriver à un nœud.

Le principe est le suivant : on va associer au graphe G une matrice de routage R, telle que :

- R[x,y] = 0 si pas de chemin entre x et y.
- R[x,y] = z si z est le premier successeur de X.

La structure que j'ai utilisée est la suivante :

```
MMap[Map[Int, Int], Int]
```

La *Map* composée de deux *Int* à l'intérieur de la *MMap* représente la connexion entre deux nœuds : si j'ai une *Map(1->2)*, on parle de la connexion entre 1 et 2. Le Int vers lequel pointe cette connexion représente le chemin emprunté.

Là encore, la structure de donnée choisie n'est pas forcément la plus facile à utiliser. Lors de ce projet, j'ai eu quelques soucis avec les indices de tableau ; c'est pour m'affranchir d'indices que j'utilise ce type de structures, ou la clef est suffisamment explicite pour l'utilisateur.

1. Par souci de temps, les tests ne sont pas exhaustifs, mais seront tout de même les plus complets possible.

Partie 4

Dijkstra

4.1 L'algorithme utilisé

L'algorithme impératif utilisé pour développer Dijkstra sera le suivant :

Algorithm 2: Algorithme de Dijskstra utilisé (source : Wikipedia)

```
Data: G : : Graph
Data: S : : Node
Result: dist[ ], prev[ ]
create List Q;
foreach v noeud de G do
    dist[v] ← INFINIT
    prev[v] ← NONE
    Q ← Q + v
end
dist[s] ← 0
while Q n'est pas vide do
    u ← chooseMin(Q, dist)
    Q ← Q − u
    foreach v voisin de u do
        alt ← dist[u] + lenght(u,v)
        if alt < dist[v] then
            dist[v] ← alt
            prev[v] ← u
        end
    end
end
```

Note : None signifie «n'existe pas» en Scala. À ne pas confondre avec le Null, car le None peut être encapsulé et récupéré via les Options(...) ; les méthodes *get()* ou *getOrElse(v1,valeur par défaut)* sont très utiles pour éviter les plantages *NullPointer* au runtime.

4.2 La complexité

L'algorithme original de Dijkstra (celui étudié ici) est en complexité $\theta(n^2)$.

4.3 Les jeux de test

Des explications sous forme de schéma seront plus parlantes qu'un long discourt. De plus, j'insiste sur le fait que les méthodes utilisées sont évidemment celles vues en cours.

4.3.1 Premier exemple, premier graphe

Dans la figure 4.1, vous pouvez voir le premier graphe sur lequel je me suis appuyé pour tester l'algorithme.

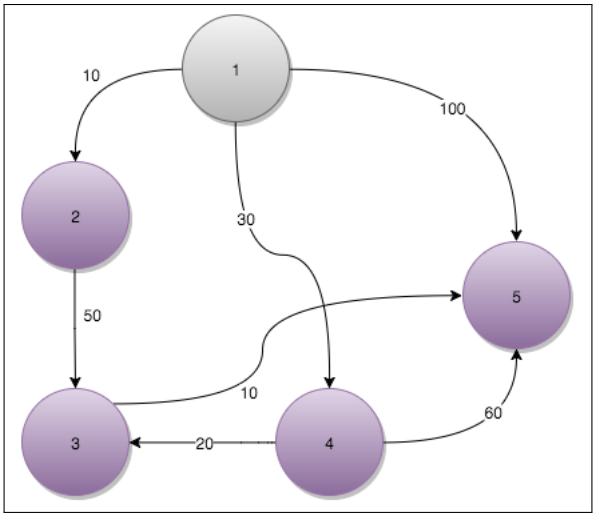

Figure 4.1 – Le premier graphe de test

Après exécution de l'algorithme, on observe que le chemin le plus court pour aller de 1 à 5 est le chemin rouge (1 -> 4 -> 3 -> 5) vu sur la figure 4.3 et que son poid est de 30 + 20 + 10 = 60. Les tests correspondants pourront être vu ci-dessous figure 4.2.

```
test("Dijktra algorithm test - first graph provided in the lesson") {
    /* INITIALISATION */

    // Check the output
    Dijkstra.giveShortestPathWeight(n5, dist) should be(60)

    [...]

    Dijkstra.giveShortestPath(n5.nodeNumber, prev) should be(List(4, 3, 5))

    [...]
  }
```

Figure 4.2 – Les tests écrits pour le premier graphe

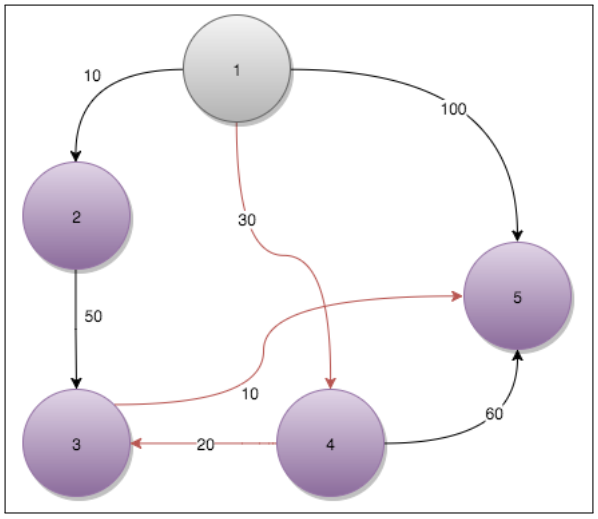

Figure 4.3 – Le premier graphe de test, chemin le plus court entre 1 et 5

4.3.2 Deuxième exemple, deuxième graphe

Le second graphe de test peut être vu en 4.4.

Le chemin le plus court entre 1 et 7 est mis en rouge sur la figure 4.5, et le chemin le plus court entre 1 et 6 en rouge également sur la figure 4.6.

Le code associé à ces cas d'utilisation / tests pourra être vu ci-dessous, en figure 4.7.

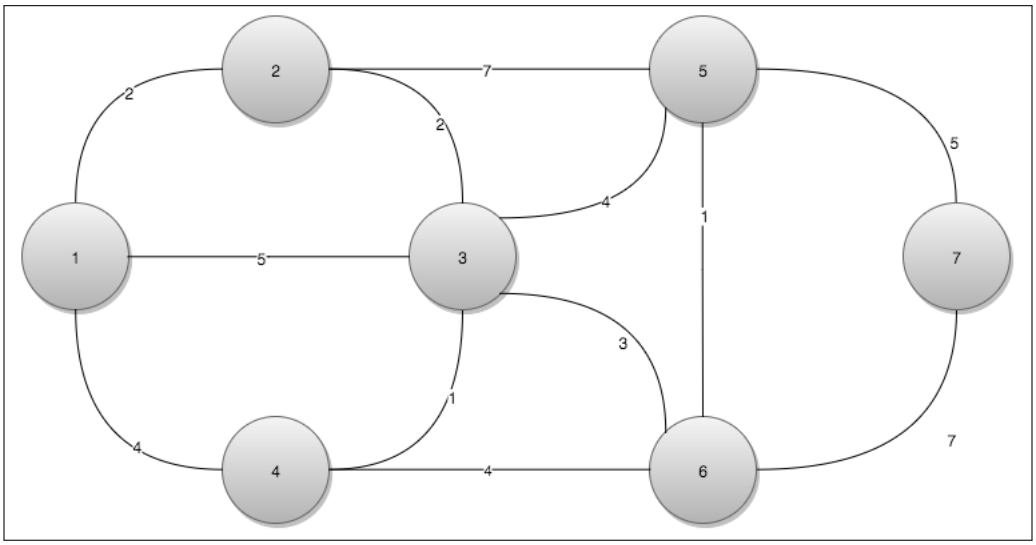

Figure 4.4 – Le second graphe de test

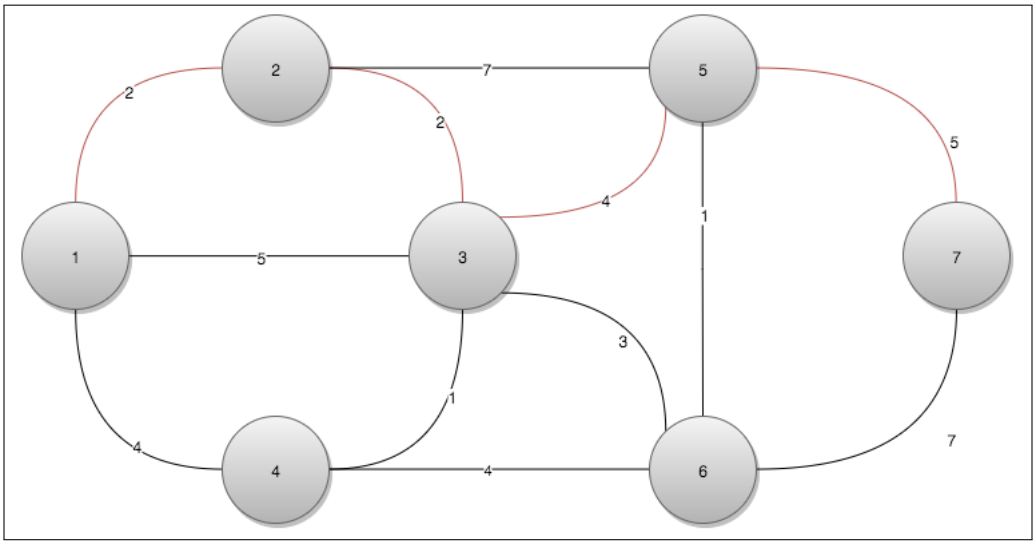

Figure 4.5 – Le second graphe de test, chemin le plus court entre 1 et 7

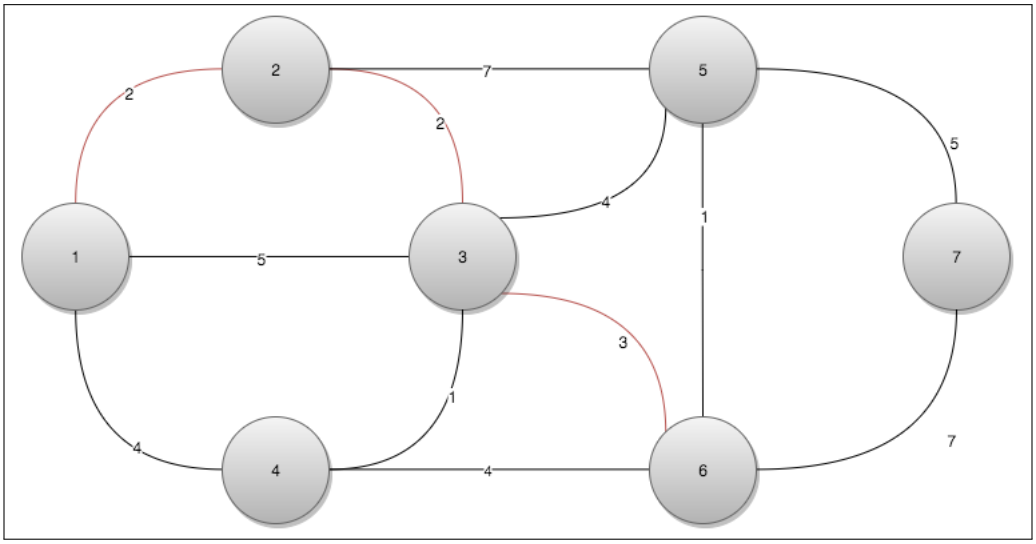

Figure 4.6 – Le second graphe de test, chemin le plus court entre 1 et 6

```
test("Dijktra algorithm test - second graph provided in the lesson") {
    /* NODE INITIALISATION */

    // Add the nodes inside a new graph
    val g1 = new Graph(List(n1, n2, n3, n4, n5, n6, n7))

    // Apply Dijktra on our graph
    val (dist, prev) = Dijkstra.applyDijkstra(g1, n1)

    // Check the output
    Dijkstra.giveShortestPathWeight(n7, dist) should be(13)
    [...]

    Dijkstra.giveShortestPath(n7.nodeNumber, prev) should be(List(2, 3, 5, 7))
    Dijkstra.giveShortestPath(n6.nodeNumber, prev) should be(List(2, 3, 6))
  }
```

Figure 4.7 – Les tests écrits pour le second graphe

Partie 5

Conclusion

Travailler sur ce projet m'a permis d'assoir mes compétences en algorithmie des graphes. L'algorithme de Warshall à été codé rapidement, mais l'algorithme de Dijkstra m'a posé un certain nombre de problèmes : dépassement d'indices, valeurs erronées ... C'est après un certain nombre d'heures que la situation s'est débloquée.

Développer en Scala m'a permis d'appréhender de manière plus complète l'utilisation des collections et la création de structures de données complexes à la volée grâce au for/yield. C'est une bonne expérience, car cela me servira certainement tous les jours en entreprise. Être fluent en Scala prend du temps, de par la syntaxe non verbeuse et la complexité du langage.

Si je devais refaire ce projet, je ferais certainement le choix dès le début d'utiliser des structures de données plus simplifiées. Les miennes sont parfois un peu complexes, bien que lisibles.

Mon regret est de ne pas avoir pu développer à cent pour cent fonctionnel pour ce projet faute de compétence.

Table des figures