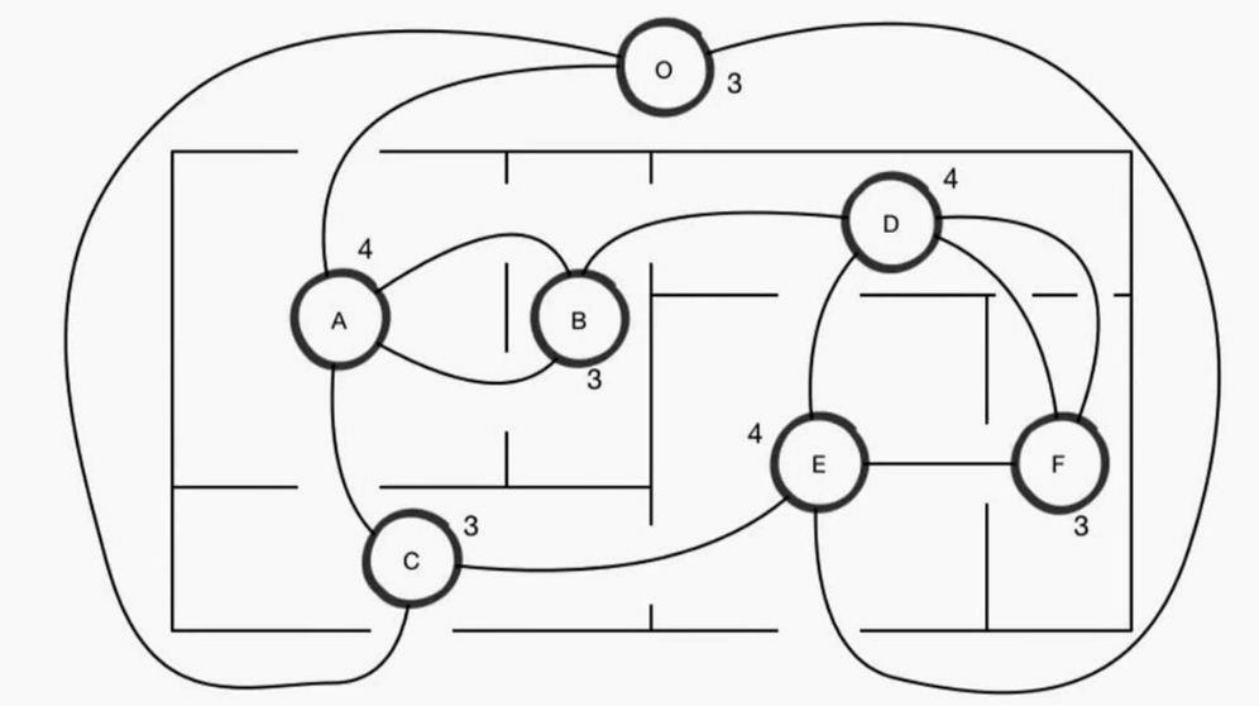
Projet d’algorithme :

Implémentation des graphes



Réalisé par :

Mohamed El Rahali

En cadre par :

M.Mestari

Remerciement

Je tiens à remercier dans un premier temps, toute l'équipe pédagogique de l'institut supérieure de génie appliqué.

Avant d'entamer ce rapport, nous profitons de l'occasion pour remercier tout d'abord nos professeurs M.Mestari qui n'a pas cessé de nous encourager pendant la durée du projet, ainsi pour sa générosité en matière de formation et d'encadrement. Nous le remercions également pour l'aide et les conseils concernant les missions évoquées dans ce rapport, qu'il nous a apporté lors des différents suivis, et la confiance qu'il nous a témoigné. Nous tenons à remercier nos professeurs de nous avoir incités à travailler en mettant à notre disposition leurs expériences et leurs compétences.

Sommaire

Introduction

Afin d'appliquer les méthodologies et les notions enseignées pendant cette année de L3 MIAGE à l'Institut supérieur de génie appliquée, je dois réaliser un projet qui me permet de m'initier au développement d'un projet plus conséquent que ceux que je l'ai pu déjà réalisé et d'appliquer les connaissances acquises durant notre scolarité.

Le projet que je dois réaliser est un programme à caractère didactique. Il doit permettre de tester différents algorithmes de parcours de graphe.

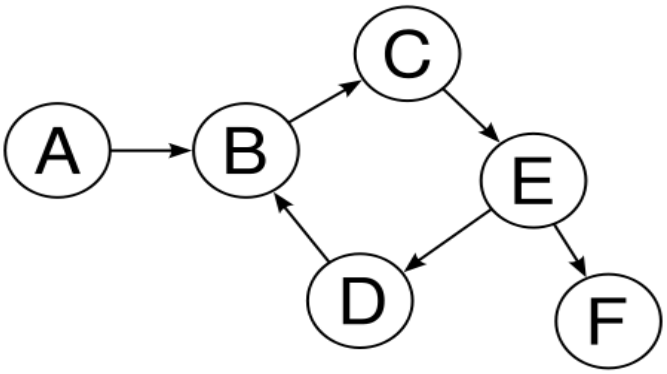
Afin de comprendre la démarche que j'ai utilisée pour mener ce projet à son terme, mon rapport se structure de la façon suivante : Tout d'abord, dans une première partie nous présentons le cadre général de notre projet, c'est-à- dire définir les différentes notions de graphe, les opérations qu'on pourrait effectuer sur ces graphes et aussi l'utilité des graphes pour résoudre les différents problématique. Ensuite dans une seconde partie, l'analyse de chaque algorithme, sa simulation et aussi sa complexité, avant que dans une troisième partie je décris les résultats obtenus.

Ce projet d'algorithmique va me permettre de concrétiser tous les algorithmes vus en cours. Le fait que les sommets soient déjà représentés par des nombres, va nous permettre de travailler directement avec les indices de tableau. Grâce à cela, je vais pouvoir me concentrer sur le fond (les algorithmes) plutôt que sur la forme (la perception des graphes par l'utilisateur).

I – Notions de graphe

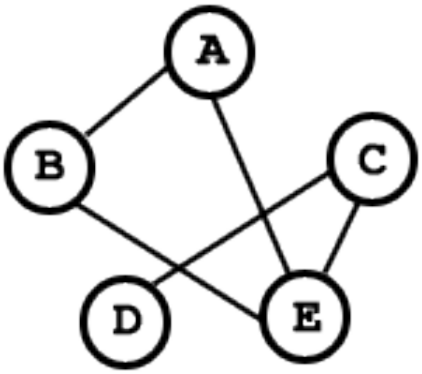
1-1 Graphes

1-1-1 Graphe orienté :



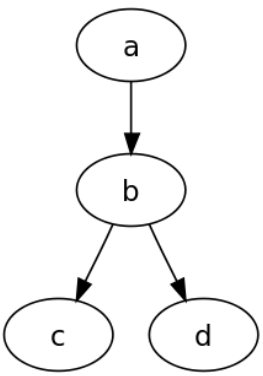
Un **graphe orienté** G = (V, A) est défini par la donnée d'un ensemble de sommets V et d'un ensemble d'**arcs** A, chaque arc étant un *couple* de sommets. Par exemple, si x et y sont des sommets, les couples (x,y) et (y,x) peuvent être des arcs du graphe G : dans ce cas, ils sont notés respectivement xy et yx.

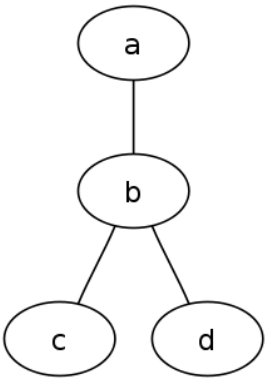
1-1-2 Graphe orienté :



Un **graphe non orienté** G = (V, E) est défini par la donnée d'un ensemble V de sommets et d'un ensemble E d'*arêtes*, chaque arête étant une paire de sommets. Par exemple, si x et y sont des sommets, la paire {x, y} peut être une arête du graphe G, auquel cas elle est notée xy.

1-1-3 Graphe simple :





Un **graphe** est dit **simple** s'il n'a pas de liens doubles ni de boucles. Dans un graphe simple, s'il existe un arc (pour un graphe orienté, une arête pour un graphe non orienté) du sommet x vers le sommet y, alors il n'existe aucun autre arc de x vers y (mais il peut exister un arc de y vers x si le graphe est orienté), et il n'existe aucun arc (aucune arête) d'un sommet vers lui-même.

II – Analyse des algorithmes

2-1 DFS

2-1-1 Définition

Lorsque l'on fait un parcours en largeur à partir d'un sommet x, on atteint d'abord les voisins de x, ensuite les voisins des voisins (sauf ceux qui sont déjà atteints) et ainsi de suite. C'est pourquoi le parcours en largeur est aussi appelé parcours concentrique. Il sert notamment à le recherche des plus courts chemins dans un graphe. Une description détaillée du parcours en largeur est donnée dans l'algorithme 1.

2-1-2 Exemple d'algorithme

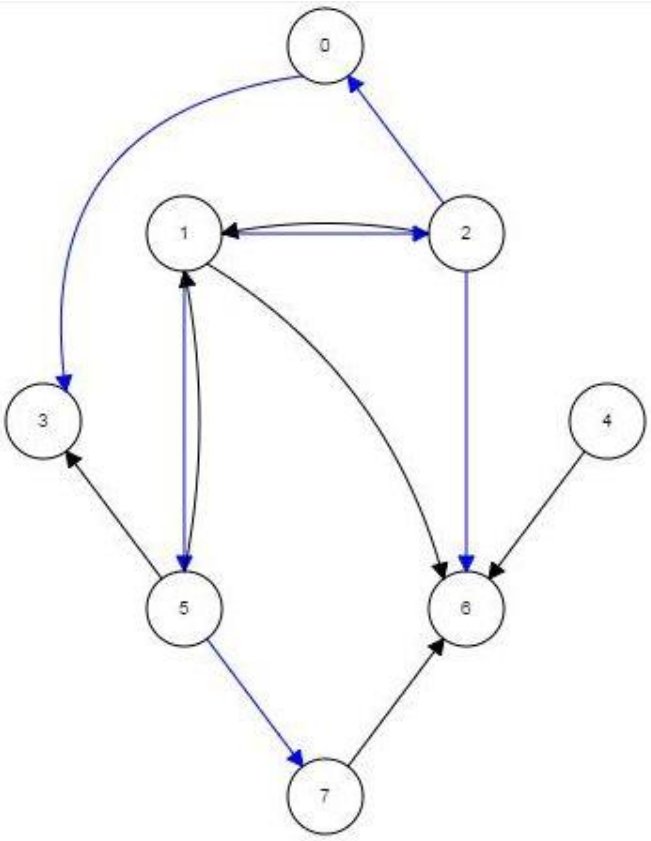
Algorithme 1 : Parcours en Largeur

```
Entrées : Un graphe G = (X, A)
Sorties : Un ordre σ sur les sommets et une arborescence
//init
pour chaque x ∈ X faire
    etat[x] ← non_atteint
index ← 1
a_Traiter ← ∅
//boucle principale
pour chaque z ∈ X faire
    si etat[z] = non_atteint alors
        //on lance un parcours à partir de z
        pere[z] ← nil
        σ[z] ← index
        index++
        a_traiter.AjouterEnQueue(z)
        tant que a_traiter ≠ ∅ faire
            x ← Tete(a_taiter)
            pour chaque y ∈ Γ(x) faire
                si etat[y] = non_atteint alors
                    etat[y] ← atteint
                    pere[y] ← x
                    σ[y] ← index
                    index++
                    a_traiter.AjouterEnQueue(y)
            a_traiter.EnleverTete()
            etat[x] ← traite
```

2-1-3 Simulation

	Parent
0	2
1	
2	1
3	0
4	
5	1
6	2
7	5

	Visited
0	T
1	T
2	T
3	T
4	f
5	T
6	T
7	T



2-1-4 Complexité

la complexité du parcours en largeur est O(n + m).

2-2 BFS

2-2-1 Définition

Contrairement au parcours en largeur, lorsque l'on fait un parcours en profondeur à partir d'un sommet x on tente d'avancer le plus loin possible dans le graphe, et ce n'est que lorsque toutes les possibilités de progression sont bloquées que l'on revient (étape de backtrack) pour explorer un nouveau chemin ou une nouvelle chaîne. Le parcours en profondeur correspond aussi à l'exploration d'un labyrinthe. L'algorithme 2 propose une implantation récursive du parcours en profondeur.

2-2-2 Exemple d'algorithme

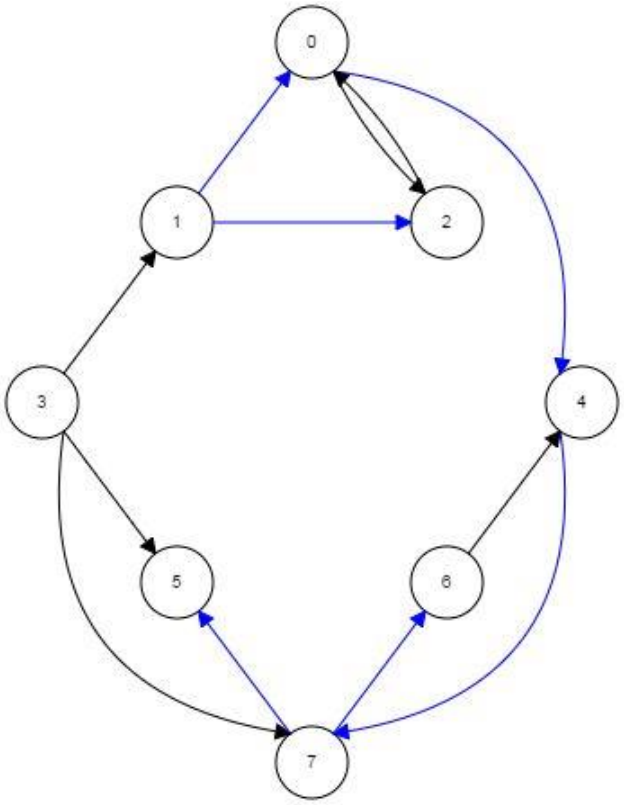
Algorithme 2 : Parcour en Profondeur

Entrées : Un graphe $G = (X, A)$
Sorties : Un ordre σ sur les sommets et une arborescence
//init ; **pour chaque** $x \in X$ **faire**
  $etat[x] \leftarrow non\_atteint$
$index \leftarrow 1$
//boucle principale **pour chaque** $x \in X$ **faire**
  si $etat[x] = non\_atteint$ **alors**
    //on lance un parcours à partir de x
    $pere[x] \leftarrow nil$
    Parcours_Prof_Recursif(x)

2-2-3 Simulation

	Parent
0	1
1	
2	1
3	
4	0
5	7
6	7
7	4

	Visited
0	T
1	f
2	T
3	f
4	T
5	T
6	T
7	T



2-2-4 Complexité

La complexité du parcours en profondeur est O(n + m).

2-3 Dijkstra

2-3-1 Définition

L'algorithme de Dijsktra calcule les plus courts chemins d'un sommet à tous les autres sous l'hypothèse où tous les coûts sont positifs (il n'est pas décile de trouver un exemple, avec coûts négatifs, où il donne une réponse erronée !). A chaque étape on met dans une liste des sommets traités les sommets dont la distance a été correctement calculée. Les sommets arrivent dans cette liste par distance croissante. Pour les sommets x de la liste A_traiter, le tableau d indique le plus court chemin de la source à x, dont tous les sommets sauf x sont déjà traités.

2-3-2 Exemple d'algorithme

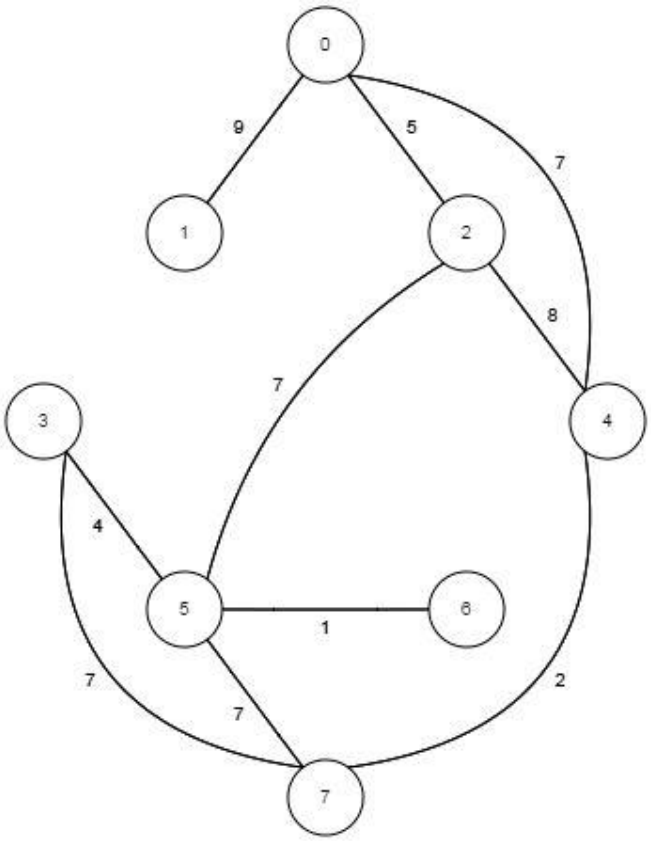
Algorithme 4 : Dijsktra

Entrées : Un graphe $G = (X, A)$, une fonction $w : U \to \mathbb{R}^+$, un sommet s
Sorties : Un tableau d tel que $d[x] = \sigma(s, x)$ pour tout x
Initpluscourtchemin (G, s)
A_Taiter $\leftarrow X$
Traités $\leftarrow \emptyset$
tant que $A\_Traiter \neq \emptyset$ **faire**
  choisir x dans A_Traiter tel que $d[x]$ soit minimum
  Traités $\leftarrow$ Traités $\cup \{x\}$
  A_Traiter $\leftarrow$ A_traiter - $\{x\}$
  pour tous les $y \in \Gamma^+(x)$ **faire**
    Relaxation (x, y)
retourner *le tableau d*

Procédure : Init_plus_court_chemin
Entrées : Un graphe $G = (X, A)$, un sommet s
Sorties : Un tableau d associant une valeur infinie à chaque sommet sauf s
pour tous les *sommet* x **faire**
  $d[x] \leftarrow \infty$;
$d[s] \leftarrow 0$;

Procédure : Relaxation
Entrées : un arc (x, y) de G
Sorties : Le tableau d mis à jour
si $d[y] > d[x] + w(x, y)$ **alors**
  $d[y] \leftarrow d[x] + w(x, y)$;

2-3-3 Simulation



Vertex	Known	Cost	Path	
0	T	0	-1	0
1	T	9	0	0 1
2	T	5	0	0 2
3	T	16	7	0 4 7 3
4	T	7	0	0 4
5	T	12	2	0 2 5
6	T	13	5	0 2 5 6
7	T	9	4	0 4 7

2-3-4 Complexité

La complexité du parcours en profondeur est O(n + m).

2-4 Bellman-Ford

2-4-1 Définition

Dans ce cas, il se peut que le graphe en entrée ait un circuit de coût strictement « négatif », appelé circuit absorbant. Dans cette situation on ne peut pas parler de chemin de coût minimum (il y aurait des chemins infinis, de coût $-\infty$. C'est pourquoi la sortie des algorithmes sera ou bien un tableau de coûts, ou bien une sortie d'erreur en cas de détection de circuit absorbant.

2-4-2 Exemple d'algorithme

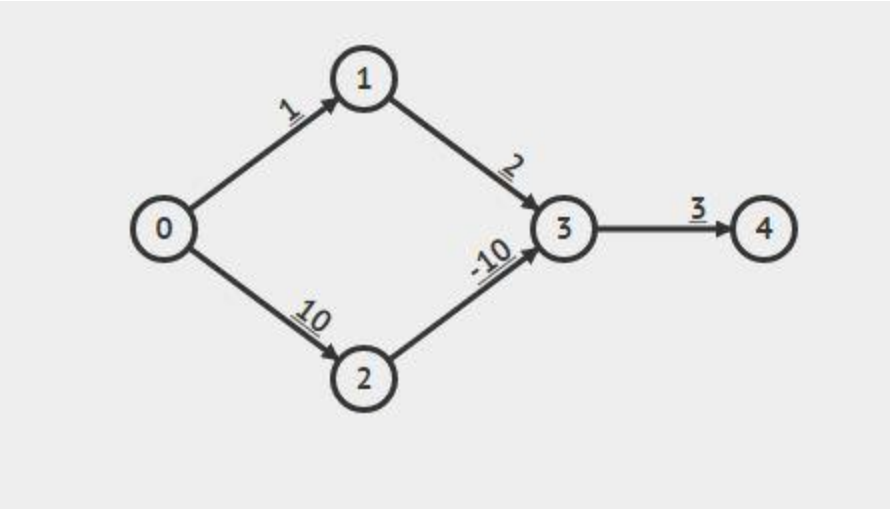
Algorithme 5 : Bellman

Entrées : Un graphe $G = (X, A)$, un sommet s.
Sorties : Un tableau D tel que $D[x] = \sigma(s, x)$ pour tout x, ou Détéction d'un circuit absorbant.

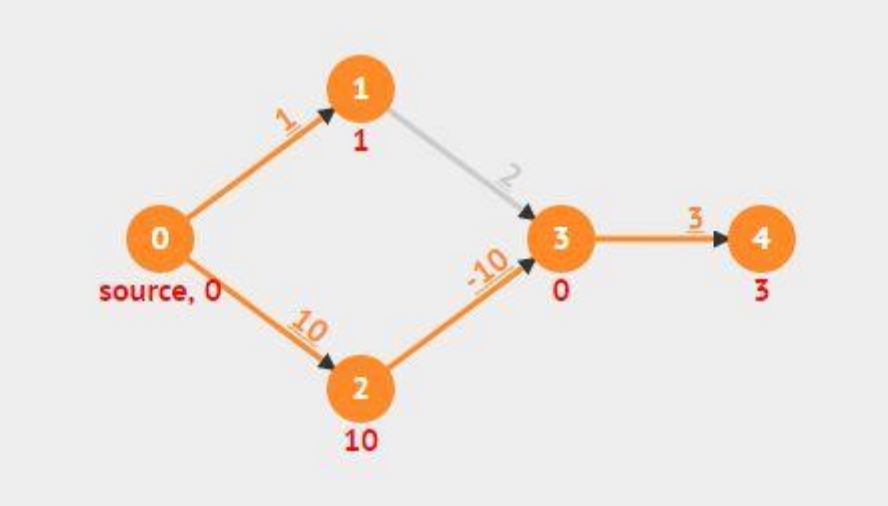
```
//init
D^0[x] ← 0
pour tous les sommet y ≠ x faire
    D^0[y] ← +∞
K ← 1
tant que K < n faire
    D^K[x] ← 0
    pour tous les sommet y faire
        pour chaque z ∈ Γ⁻[x] faire
            D^K[y] ← min(D^{K-1}[y], D^{K-1}[z] + w(x, y))
    si D^K = D^{K-1} alors
        retourner Arret de la boucle
    si K = n + 1 alors
        retourner il existe un circuit de poid négatif
```

2-4-3 Simulation

// le graphe d'initialisation



// le resultat



2-4-4 Complexité

La complexité de l'algorithme est en O(S || A) où S est le nombre de sommets A est le nombre d'arcs.

2-5 Floyd-Warshall

2-5-1 Définition

L'algorithme suivant dû à Floyd, très simple et de complexité O(n3) , calcule les plus courts chemins entre tout couple de sommets. A l'étape K, le tableau δk représente les longueurs des plus courts chemins de x à y dont les sommets internes sont uniquement des sommets numérotés entre 1 et K.

2-5-2 Exemple d'algorithme

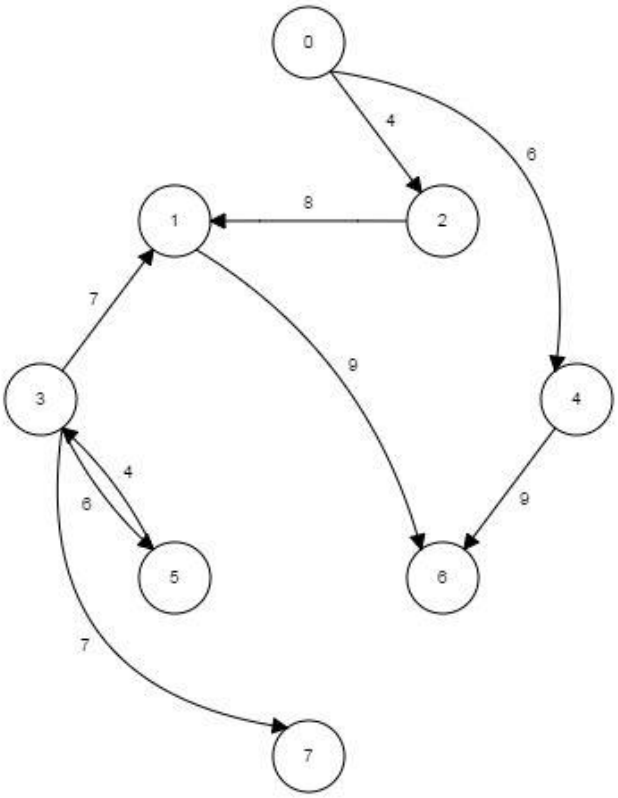
Algorithme 6 : Floyd

Entrées : Un graphe $G = (X, A)$, une application $w : A \leftarrow \mathbb{R}$
Sorties : Une application $\delta : X \times X \rightarrow \mathbb{R}$
//init
pour tous les $x, y \in X$ **faire**

$$\delta^0(x,y) \leftarrow \begin{cases} \infty & si \quad (x,y) \notin A \\ 0 & si \quad x = y \\ w(x,y) & si \quad (x,y) \in A \end{cases}$$

pour *K de 1 à n* **faire**
  pour tous les $x, y \in A$ **faire**
    $\delta^k(x,y) \leftarrow min(\delta^{k-1}(x,y), (\delta^{K-1}(x,x_k) + \delta^{K-1}(x_k,y)))$
si *Il existe* $x \in X$ *tel que* $\delta^n(x,x) < 0$ **alors**
  retourner circuit de cout négatif
sinon
  retourner δ^n (matrice nxn)

2-5-3 Simulation



Cost Table

	0	1	2	3	4	5	6	7
0	INF	12	4	INF	6	INF	15	INF
1	INF	INF	INF	INF	INF	INF	9	INF
2	INF	8	INF	INF	INF	INF	17	INF
3	INF	7	INF	INF	INF	6	16	7
4	INF	INF	INF	INF	INF	INF	9	INF
5	INF	11	INF	4	INF	INF	20	11
6	INF	INF	INF	INF	INF	INF	INF	INF
7	INF	INF	INF	INF	INF	INF	INF	INF

Path Table

	0	1	2	3	4	5	6	7
0	-1	2	0	-1	0	-1	4	-1
1	-1	-1	-1	-1	-1	-1	1	-1
2	-1	2	-1	-1	-1	-1	1	-1
3	-1	3	-1	-1	-1	3	1	3
4	-1	-1	-1	-1	-1	-1	4	-1
5	-1	3	-1	5	-1	-1	1	3
6	-1	-1	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1

2-5-4 Complexité

Nous cherchons un chemin pour chaque couple de sommets en passant par chaque sommet. Nous avons donc une complexité en $O(n^3)$ avec n le nombre de sommets dans le graphe.

2-6 Kruskal

2-6-1 Définition

L'algorithme de Kruskal calcule un arbre couvrant de poids minimal. On considère un graphe connexe non-orienté et pondéré : chaque arête possède un poids qui est un nombre qui représente le coût de cette arête. Dans un tel graphe, un arbre couvrant est un sous-graphe connexe sans cycle qui contient tous les sommets du graphe. Le poids d'un tel arbre est la somme des poids des arêtes qui le compose. Un arbre couvrant minimum est un arbre couvrant dont le poids est inférieur ou égal à celui de tous les autres arbres couvrants du graphe. L'objectif de l'algorithme de Kruskal est de calculer un tel arbre couvrant minimum.

2-6-2 Exemple d'algorithme

Algorithme 7 : Kruskal

Entrées : Un graphe $G = (X, A)$non orienté, une application $w : A \leftarrow \mathbb{R}$

Sorties : Un arbre$^a T = (X, A_T)$ recouvrant de poid minimum, $A_T \subseteq A$,$w(A_T)$ et minimum)

Trier les arrêtes par poids croissants dans L

$A \leftarrow \emptyset$

tant que $L \neq 0$ **faire**

  e $\leftarrow$ tête(L)

  $L \leftarrow L \backslash \{e\}$

  si $(X, A \cup \{e\})$ *n'a pas de cycle* **alors**

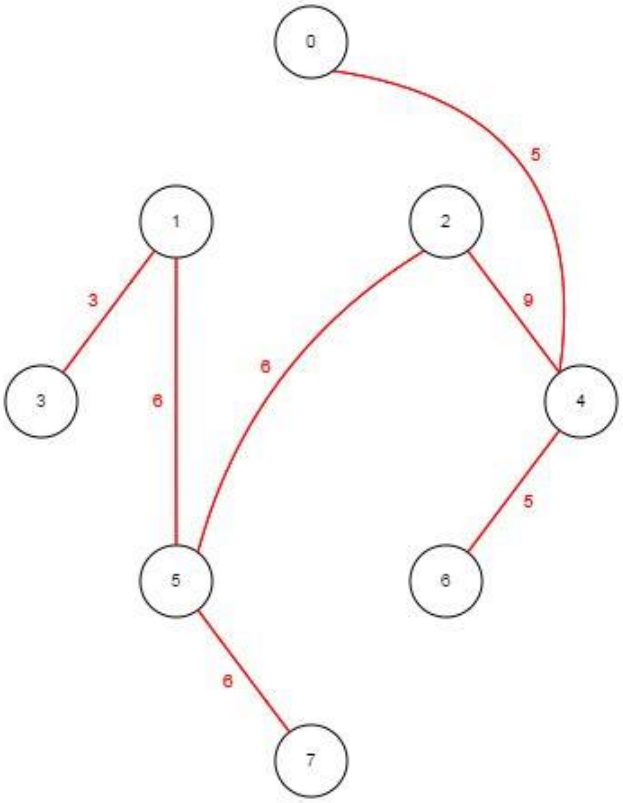
    $A \leftarrow A \cup \{e\}$

retourner (X, A)

2-6-3 Simulation

Disjoint Set

0	4
1	3
2	3
3	-8
4	3
5	3
6	4
7	3



2-6-4 Complexité

La complexité est en O(n) avec n la taille du tableau.

2-7 Prim

2-7-1 Définition

L'algorithme de prim est un algorithme qui calcule un arbre couvrant minimal dans un graphe connexe value et non orienté. En d'autres termes, cet algorithme trouve un sous-ensemble d'arêtes formant un arbre sur l'ensemble des sommets du graphe initial, et tel que la somme des poids de ces arêtes soit minimale. Si le graphe n'est pas connexe, alors l'algorithme détermine un arbre couvrant minimal d'une composante connexe du graphe.

L'algorithme consiste à faire croître un arbre depuis un sommet. On commence avec un seul sommet puis à chaque étape, on ajoute une arête de poids minimum ayant exactement une extrémité dans l'arbre en cours de construction. En effet, si ses deux extrémités appartenaient déjà à l'arbre, l'ajout de cette arête créerait un deuxième chemin entre les deux sommets dans l'arbre en cours de construction et le résultat contiendrait un cycle.

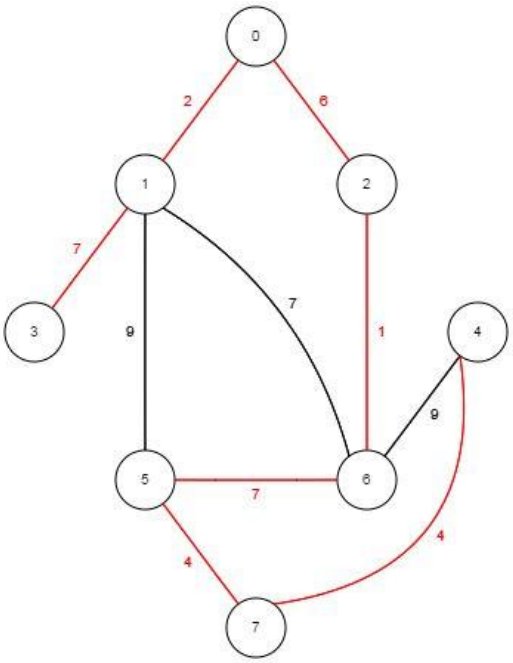
2-7-2 Exemple d'algorithme

Algorithme 8 : Prim

Entrées : Un graphe $G = (X, A)$non orienté, une application $w : A \leftarrow \mathbb{R}$
Sorties : Un arbre(Graphe connexe sans cycle $T = (X, A_T)$ recouvrant de poid minimum, $A_T \subseteq A$,$w(A_T)$ et minimum)

```
//init
T ← (a, ∅)
A ← ∅
pour tous les x ∈ X faire
    etat[x] ← non-atteint
etat[x] ← atteint
//boucle principale
pour i in 0..(n − 1) faire
    choisir l'arête e = (x, y) de poid minimum parmi celles ayant une extrêmité atteinte
    et une non atteinte
    A ← A ∪ {e}
    marquer comme atteinte l'extrêmité qui était non atteinte
```

2-7-3 Simulation



Vertex	Known	Cost	Path
0	T	0	-1
1	T	2	0
2	T	6	0
3	T	7	1
4	T	4	7
5	T	7	6
6	T	1	2
7	T	4	5

2-7-4 Complexité

la complexité de la fonction *Prim* est en $O(n^2)$ avec n le nombre de sommets dans le graphe.

III – Implémentation en java

3-1 Les outils utilises

3-1-1 Le langage JAVA

Java est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au *SunWorld*.

La société Sun a été ensuite rachetée en 2009 par la société Oracle qui détient et maintient désormais Java.

La particularité et l'objectif central de Java est que les logiciels écrits dans ce langage doivent être très facilement portables sur plusieurs systèmes d'exploitation tels que UNIX, Windows, Ma OS ou GNU/Linux, avec peu ou pas de modifications. Pour cela, divers plateformes et frameworks associés visent à guider, sinon garantir, cette portabilité des applications développées en Java.

3-1-2 Netbeans IDE

NetBeans est un environnement de développement intégré (EDI), placé en *open source* par Sun en juin 2000 sous licence CDDL (Commo Development and Distribution License) et GPLv2. En plus de Java, NetBeans permet la prise en charge native de divers langages tels le C, le C++, le JavaScript, le XML, le Groovy, le PHP et le HTML, ou d'autres (dont Python et Ruby) par l'ajout de *greffons*. Il offre toutes les facilités d'un IDE moderne (éditeur en couleurs, projets multi-langage, refactoring, éditeur graphique d'interfaces et de pages Web).

3-2 Les prototypes

3-2-1 La structure de donnée

```
public class Graph {
        Integer[][] adjMat;      // adjacency matrix
        int cardinality;         // number of Vertices in the graph
```

1- Attribues :

Matrice d'adjacence : [ ][ ] : entier

Le nombre de sommet : entier

```
static class Pair implements Comparable<Graph.Pair> {
        Integer vertex;
        Integer pathWt;
```

2- Attribues :

Sommet : entier

Arete : entier

3-2-2 Fonctions & procédures

```
public ArrayList<Integer> bfs(int start) ;

public ArrayList<Integer> dfs(int start) ;

public void kruskal() ;

public void mergeSets(ArrayList<HashSet<Integer>> sets,
HashSet<Integer> vert1Set,HashSet<Integer> vert2Set) ;

public void prim(int source) ;

public void modifiedDijkstra(int source) ;

public void bellmanFord(int source) ;

public void floydWarshall() ;

public ArrayList<Integer> getNeighbors(int vertex) ;

public ArrayList<Integer> getNeighbors(int vertex) ;

public Integer getEdgeWt(int outVert, int inVert) ;

public ArrayList<Edge> getEdges(boolean isDirected) ;

public void printMatrix(Integer[][] mat) ;

public int compareTo(Edge other) ;
```

Conclusion

Ce programme à nécessité de nombreuses heures de travail et, enfin, je suis parvenu à un résultat satisfaisant. Sa réalisation m'a permis non seulement de compléter mes connaissances en mathématiques mais aussi dans le langage de programmation Java pour lequel certaines recherches ont aussi été nécessaires pour parvenir au résultat final., Ceci reflète en effet ce qui ce passe en entreprise : le développement de logiciel conséquents. Enfin, vu l'étendu du sujet, ce programme pourra être, par la suite, complété par moi-même ou bien tout développeur le désirant.

Webographie

http://www-igm.univ-mlv.fr/~mac/ENS/01-projets/XMLV/pelleas/exemples/rapport.html

https://www.cs.usfca.edu/~galles/visualization/Prim.html

http://www.univ-orleans.fr/lifo/membres/todinca/Cours/Graphes/MonCours.pdf

http://perso.ens-lyon.fr/eric.thierry/Graphes2009/bouchitte.pdf

https://www.iut-info.univ-lille1.fr/~iovleff/pub/Teaching/Projets/Rapports/RapportAlgoGraphes.pdf

http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap10_slides.pdf

https://www.programming-algorithms.net/article/43764/Jarnik-Prim-algorithm

http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/

http://www.codebytes.in/2015/11/bellman-ford-single-sort-shortest-paths.html