GROUPE 2

Penot Paul // Bories Thomas

CIR 2

ISEN Lille

Institut Supérieur de l'électronique et du Numérique

PROJET PLURIDISCIPLINAIRE

Projet C++

Professeur référent : Mme Zgaya M. Soulignac

Table des matières

[1-Introduction 3](#_Toc262501731)

[1.1-Projet 3](#_Toc262501732)

[1.2-plan du rapport 3](#_Toc262501733)

[1.3-Organisation 3](#_Toc262501734)

[2-Mise en place des Graphes 5](#_Toc262501735)

[2.1-Le type Matrix 5](#_Toc262501736)

[2.1.1-Introduction 5](#_Toc262501737)

[2.1.2-Conception 5](#_Toc262501738)

[2.1.3-Implémentation 6](#_Toc262501739)

[2.1.5-Conclusion 6](#_Toc262501740)

[2.2-Le type Graphe *(basée sur les Matrices)* 6](#_Toc262501741)

[2.2.1-Introduction 6](#_Toc262501742)

[2.2.2-Conception et implémentation 6](#_Toc262501743)

[2.2.3-Conclusion 7](#_Toc262501744)

[3-Algoritmes 8](#_Toc262501745)

[3.1-Welsh-Powell 8](#_Toc262501746)

[3.1.1-Introduction 8](#_Toc262501747)

[3.1.2-conception 8](#_Toc262501748)

[3.1.3-Implémentation 9](#_Toc262501749)

[3.1.4-Conclusion 10](#_Toc262501750)

[3.2-Dijkstra 11](#_Toc262501751)

[3.2.1-Introduction 11](#_Toc262501752)

[3.2.2-Conception 12](#_Toc262501753)

[3.2.3-Implémentation 12](#_Toc262501754)

[3.2.3-Conclusion 12](#_Toc262501755)

[3.3-Little 13](#_Toc262501756)

[3.3.1-Introduction 13](#_Toc262501757)

[3.3.2-conception 13](#_Toc262501758)

[3.3.3-Implémentation 14](#_Toc262501759)

[3.3.4-Conclusion 15](#_Toc262501760)

[4-Conclusion 15](#_Toc262501761)

# 1-Introduction

## 1.1-Projet

Dans le cadre de notre deuxième année à l'ISEN, nous devons réaliser un projet pluridisciplinaire par binôme. Celui-ci devra proposer à l'utilisateur une application permettant de manipuler des graphes.

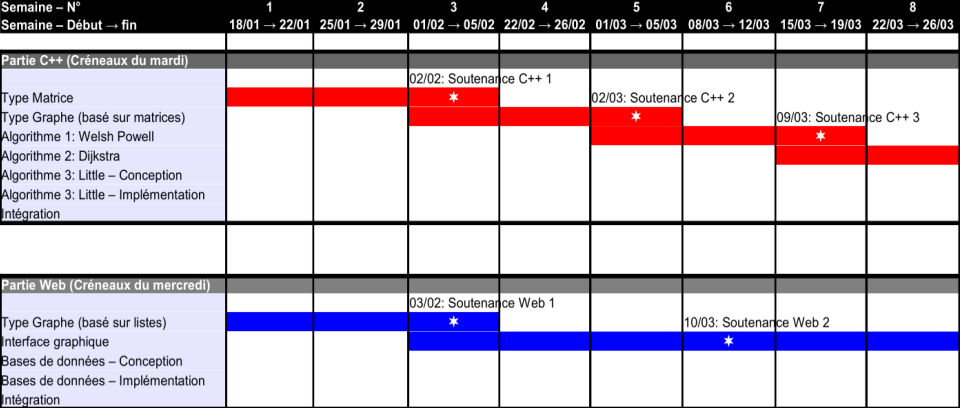
Pour mener à bien ce projet nous allons devoir implémenter différents algorithmes capables d'intervenir sur des graphes que nous aurons modélisé précédemment grâce la puissance de la programmation orienté objet (POO) que nous offre le C++. Nous allons d'ailleurs utiliser l'une des méthodes de modélisation vue en cours pour concevoir nos propre matrices. Une fois tous les algorithmes fonctionnels ils pourront être mis en relation avec une interface WEB développée en parallèle.

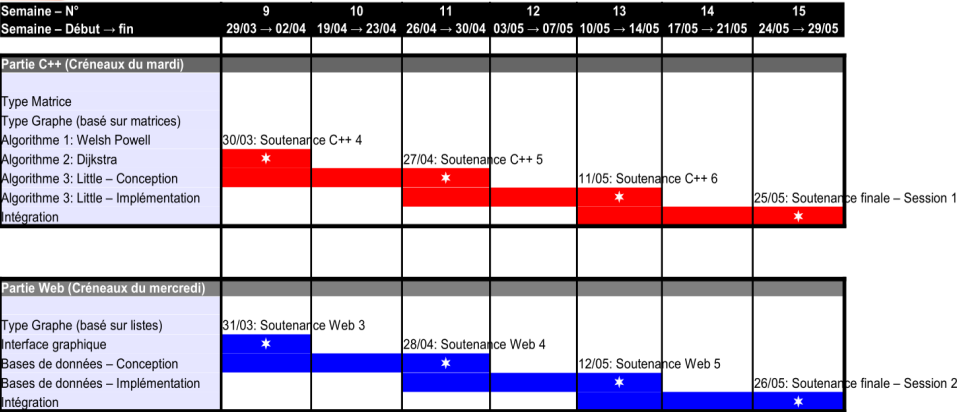
## 1.2-plan du rapport

Le rapport s'organisera en sous-partie résumant les différentes étapes du projet C++. Il doit permettre de suivre l'évolution du projet, chacune des sous partie donnera les objectifs à réaliser pour chaque soutenance, notre choix de conception, implémentation, conclusion et divers éléments permettant la description approfondie du travail réalisé.

## 1.3-Organisation

La réalisation de ce projet s'étend sur 7 étapes qui doivent être atteintes a intervalle régulier pour nous permettre de ne pas prendre de retard et nous mener jusqu’à un projet fonctionnel respectant les spécifications données.





# 2-Mise en place des Graphes

## 2.1-Le type Matrix

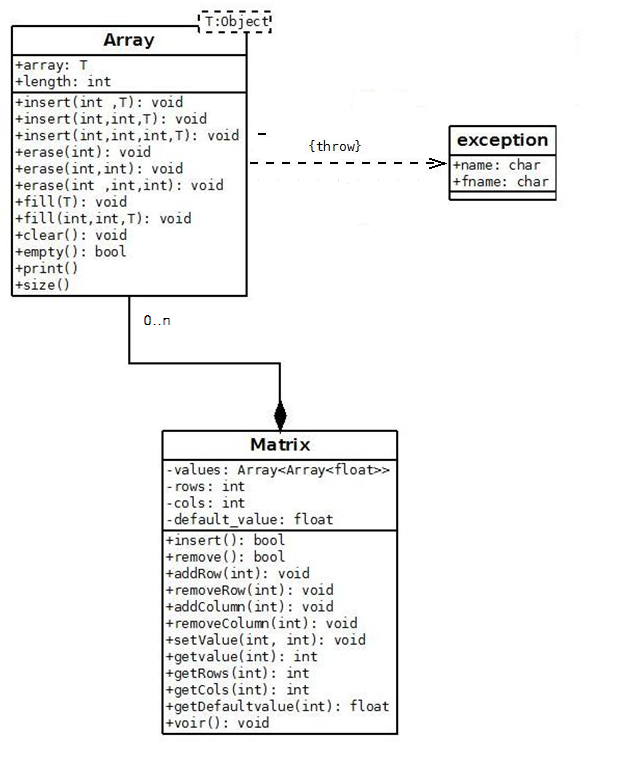
### 2.1.1-Introduction

Pour pouvoir créer nos futurs graphes grâce aux méthodes que nous connaissons, il nous faut commencer par implémenter une classe Matrix. Nous utilisons donc l'avantage de la poo pour pouvoir évoluer dans notre projet. Cette classe Matrix sera l'une de nos fondation pour le type graphe.

### 2.1.2-Conception

Nous avons tout d'abord pensé créer une classe Array, permettant de manipuler des tableaux unidimensionnels. Par manque de temps, nous avons délibérément choisi d'utiliser la classe vector de la librairie standard. Nous sommes conscient de ne pas respecter une des spécification, mais nous avons préféré ne pas perdre de temps sur la création d'une classe, d'autant plus que nous manipulons des tableaux depuis l'année dernière, cela ne nous aurait donc rien appris au niveau algorithmique. Quant à l'héritage & cie, nous avons eu des TP dessus, et aurons l'occasion de réutiliser ces concepts dans la suite du projet.

Nous créons aussi une classe exception qui hérite de la classe exception standard, nous permettant de renvoyer des exceptions sur n'importe quels types d'erreurs.



La classe Matrix prend comme paramètre une entité de la classe Array contenant les valeurs de la matrice.

rows et cols sont le nombres de ligne et colonne de la matrice .

### 2.1.3-Implémentation

Nous avons choisi de pouvoir stocker n'importe quel entité dans la classe Array en prévision de la création de graphe à partir de matrice car nos graphes seront représentés par des matrices d'objet tel que nœud ou arc.

### 2.1.5-Conclusion

Le nouveau concept de Template vue en cour nous a permis de réaliser une classe Matrice construite sur des grâce a une classe Array capable de contenir des objets come nous l'avions voulu il ne reste plus qu'à créer le type graphe à partir de ces fondations

## 2.2-Le type Graphe (basée sur les Matrices)

### 2.2.1-Introduction

Cette étape du projet doit permettre une implémentation des graphes basés sur les matrices, celle-ci est permise grâce a l'implémentation des matrices lors de la première période du projet. Cette représentation des graphes nous donnera une base pour appliquer les algorithmes de Welsh Powell, Dijkstra, Little.

### 2.2.2-Conception et implémentation

Pour construire un graphe nous avons besoin de différents éléments qui sont des nœuds et des arcs, pour les représenter nous utiliseront 2 instances du type Matrix la première matrice contenant les id et les valeurs des nœuds et la deuxième matrice contenant les nœuds de départ d'arriver pour un arc et sa valeur.

Nous aurons donc seulement besoin de créer une classe graphe ayant comme élément 2 matrices pour représenter ces arcs et nœuds et deux booleans pour donner sont statut de pondération et d'orientation. Pour la création de graphe nus reprenons les fonction vue en cour en les adaptant au langage.

Les différentes fonctions permettant de créer les graphes sont :

-addNode(int i) :Crée un nœud d'indice i dans le graphe.

-removeNode(int i):Supprime le nœud d'indice i du graphe ainsi que les arrêtes lui étant liées.

-addEdge(int i,int j):Crée un arc entre les nœuds i et j.

-removeEdge(int i, int j):Supprime un arc .

-getNodeValue(int i): Retourne le poids du nœud i.

-setNodeValue(int i, float v):Permet de modifier la valeur du nœud i par v

-getEdgeValue(int i, int j):Retourne le poids de l'arc

-setEdgeValue(int i, int j, float v):Permet de modifier la valeur de l'arc entre les nœuds i et j par v.

-getNeighborhoodSize(int i):Retourne le nombre d'arcs partant du nœud i.

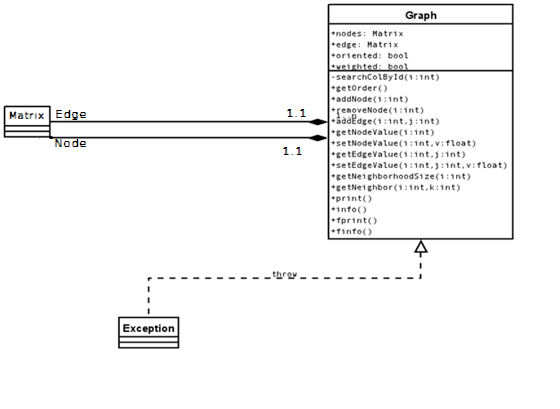
-getNeighbor(int i, int k):Retourne le k-ieme voisin du nœud i.

-print():Affiche les nœuds avec leur arcs associer.

-info():Affiche les informations sur le graphe ,s'il est pondéré et orienté .

-fprint():retourne les même informations que print() mais les écrit dans une fichier texte

En plus de ces fonctions une fonction test vas être créé qui effectuera les test nécessaire pour vérifier le bon fonctionnement de notre classe graphe .Elle écrira les résultats des tests directement dans un ficher texte plus pratique à utiliser.



### 2.2.3-Conclusion

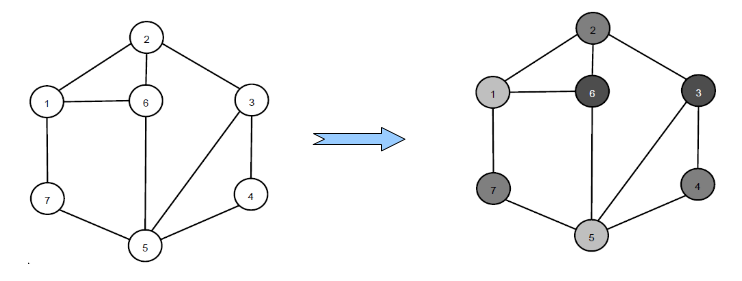
Pour cette nouvelle étape nous avons terminé l'implémentation du type graphe nous permettant maintenant de créer des graphes selon nos envies. Nous avons aussi pu ajouter une fonction générant des graphes aléatoire pour simplifier la création des cas de tests .l'ensemble nous permet donc de continuer sur une nouvelle partie qui implique de l'algoritmie.

# 3-Algoritmes

## 3.1-Welsh-Powell

### 3.1.1-Introduction

L'algorithme de Welsh-Powell est utilisé dans le cas d'une coloration de graphe, en utilisant le moins de couleurs possible, il doit attribuer une couleur à chaque nœuds de telle façon qu'aucun nœud n'est la même couleur que ses voisins. L'algorithme implémenté ici n'assurera pas que le nombre de couleurs utilisées soit le minimum. Pour tester notre algorithme nous disposons maintenant de notre classe Graphe.



### 3.1.2-conception

Nous choisissons d'implémenter notre Welsh-Powell comme une fonction membre de la classe Graphe, en effet elle ne nous servira normalement que pour être appliquée sur des graphes. Nous ajoutons donc seulement 3 fonctions a la classe Graphe :

-sortNodeByDegreeAndInitColor(m:Matrix):

-canBeColored(id:int,color:int):bool

-WelshPowell()

Pour représenter la couleur des nœuds dans notre rendu final nous choisissons de retourné un nombre de 1 à n pour chaque couleur différente et 0 si le nœud n'est pas coloré.



### 3.1.3-Implémentation

Les fonctions sont créées pour respecter certaines étapes.

Ainsi la fonction « sortNodeByDegreeAndInitColor » permet de classer les nœuds selon l'ordre décroissant de leurs degrés dans une matrice et initialise la couleur courante pour tous les nœuds à 0 représentant l'absence de couleur.

La matrice a la forme :

sommet | couleur | degré

----------+---------+------

s1 | c1 | d1

... | ... | ...

sn | cn | dn

La colonne degré ne sert qu'à la fonction, et est donc enlevé à la fin.

La fonction « canBeColored » nous sert à tester si le nœud appelé n'est pas déjà coloré et si aucun des ses voisins n'est de la même couleur.

Vient enfin la fonction « WelshPowell » qui fonctionne en 3 étapes tant qu'il reste des nœuds non colorés :

1. Incrémentation de l'indice i de la couleur courante.

2. Parcours de la liste des noeuds dans l'ordre jusqu'à trouver le premier non coloré, et lui attribue la couleur i

3. Attribution de cette même couleur à chaque noeud non encore coloré et non adjacent à un noeud de cette couleur grâce à la fonction canBeColored.

### 

### 3.1.4-Conclusion

Ce premier algorithme nous a permis de tester et de corriger les quelques défauts qu'il pouvait exister sur notre classe Graphe, ayant déjà bien détaillé l'algorithme en cour nous n'avons pas éprouvé de vraie difficulté, le seul vrai travail a été d'adapter la version vue en cour en simple langage en C++++ et trouver comment retourner le résultat de l'algorithme de façons à pouvoir l'interpréter.

## 3.2-Dijkstra

### 3.2.1-Introduction

L'algorithme de Dijkstra doit nous permettre de résoudre le problème du plus court chemin entre deux nœuds, start et goal Il s'applique à un graphe connexe dont le poids lié aux arêtes est positif ou nul. L'algorithme devra utiliser la représentation des graphes sous forme de matrices déjà implémentées auparavant et devra être soumis a un panel de test pour vérifier son fonctionnement.

### 

### 3.2.2-Conception

Pour la conception, quatre nouvelles fonctions vont être ajoutées à la classe Graph :

-insertNode(open: Matrix, id: int, float: cost)

-int searchNode(open: Matrix, id: int):int

-addNode(open: Matrix, id: int, newcost: float ):bool

-Dijkstra(start: int, goal: int)

-Ajout de cas de test.

Dijkstra étant un algorithme seulement applicable sur des graphes nous choisissons de l'inclure dans les fonctions de la classe graph.

### 3.2.3-Implémentation

Pour pouvoir parcourir tous les nœuds en stockant leurs antécédents et le coût pour atteindre celui-ci, nous créons une matrice open.

Les nœuds se divisent en trois catégories:

-Les nœuds non explorés :Nœuds dont le coût d'atteinte sont inconnus.(Représenté par leurs id négatifs durant le traitement.

-Les nœuds ouverts : Nœuds en cours de traitement pour lesquels une estimation du coût d'atteinte a été calculé.

-Les nœuds fermés : Nœuds pour lesquels le coût d'atteinte a été validé comme optimal.

La construction du chemin le plus court va être implémenté pour suivre 2 phases :

-Phase 1:Propagation des coûts:

L'algorithme de Dijkstra va se propager dans le graphe sur un front formé par les nœuds ouverts, sa condition d'arrêt est émise par la rencontre de ce front avec le nœud goal ou qu'il n'y a plus de nœuds ouverts (Dans quel cas on ne procède pas à la phase 2).

-Phase 2:Propagation des coûts :

Nous créons maintenant une matrice path pour stocker les nœuds par lesquels nous passons en partant du nœud goal jusqu'au nœud Start en utilisant les antécédents.

### 3.2.3-Conclusion

Comme précédemment ayant déjà bien détaillé l'algorithme en cour nous n'avons pas éprouvé de vraie difficulté, une fois encore nous avons dû transformer le code en C++ même si cela était une partie plutôt simple étant donné que l'algorithme n'utiliser pas de concept complexe.

## 3.3-Little

### 3.3.1-Introduction

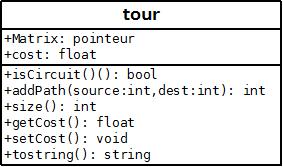
L'algorithme de Little va nous servir à résoudre le problème du voyageur de commerce soit l'énoncer suivant :

Étant donné *n* points (des « villes ») et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point et revienne au point de départ.

Celui-ci nous contraint à limiter notre utilisation de Little sur certain graphe pour lesquels nous allons détailler les contraintes.

Cet algorithme a été décomposer en 2 étape pour nous permettre de finir correctement la conception avant l'implémentation.

### 3.3.2-conception



Une classe Tour construit la tournée au fur et à mesure que l'on y rajoute des chemins, grâce à la fonction addPath, qui de plus renvoie le chemin parasite.

Le cycle se construit de lui-même. En effet, lorsque l'on rajoute le chemin (A--B), la fonction parcourt la matrice et cherche la fin d'un chemin qui correspond au début de celui qui est à rajouter. S'il le trouve, il le rajoute après, sinon, à la fin.

À la fin de l'algorithme, si tout s'est bien passé (vérification grâce à la fonction isCircuit), le circuit est correctement construit, et le coût total de la tournée est stocké dans la propriété cost.

Le retour de la fonction permet de supprimer les chemins parasites. Sous la forme d'un tableau [source, destination], il servira à la fonction realLittle pour l'enlever.

Par défaut, le chemin parasite du morceau de circuit (A,B) est (B,A).

La fonction realLittle prend des matrices en paramètre ainsi que la tournée. C'est sur ces paramètres que la fonction va faire des modifications, en les sauvegardant avant, pour une réutilisation éventuelle (ie visite d'une autre branche de l'arbre).

### 

### 3.3.3-Implémentation

La première question à traiter et sur quel type de graphe cet algorithme peut-il être appliqué et qu'allons nous devoir retourner. L'algorithme peut être appliqué à n'importe quels graphes mais il ne retournera pas forcement une tournée, en effet selon si le graphe est orienté ou non les condition nécessaire et suffisante pour pouvoir retourné un tourné change et est assez complexe à tester.

Dans le cas du retour nous avons choisi de créer une classe «tour» détailler ci-après.

« tour » comprend comme paramètres:

-Un pointeur sur Matrix représentant la tournée.

-Un float donnant le coût total de la tournée.

Et comme fonctions :

-Une fonction qui renvoie true si le chemin est un circuit, false sinon.

-Une fonction pour ajouter un morceau de chemin et nous renvoyer le chemin parasite.

-Une fonction qui retourne la taille de la tournée.

Ensuite pour pouvoir effectuer l'algorithme de Little nous avons suivi les différentes étapes que suit notre conception. Nous créons une matrice contenant le coût des différents chemins pouvant être réalisés entre chaque noeuds. Cette matrice est généré à partir du graphe de départ pour ensuite être traité.

Pour effectuer les réductions lignes et colonnes de la matrice nous avons ajouté à la classe Matrix des fonctions permettant de trouver la valeur minimal de chaque colonnes//lignes ainsi que les fonctions permettant de soustraire à une ligne//colonne une valeur, celle-ci permettront aussi le calcul des regrets.

Pour pouvoir construire le chemin nous utilisons notre classe « tour » dans la classe « graphe » où celle-ci va être construite par récursivité.

Notre fonction Little va nous permettre d'effectuer les appel récursif pour la construction de l'arbre de recherche à chaque appel elle va stocker une copie des matrice précédemment parcourue aux cas ou le choix d'un chemin s'avère être une erreur.

Elle fonctionne en plusieurs étapes utilisant les fonction créé pour la manipulation de matrice.

Soit :

- Réduction de la matrice en ligne, puis en colonne.

- Ajout de la somme des minimums ligne et colonne au coût actuel de la tournée.

- Calcul des regrets.

- Mise à jour du cycle, et suppression du chemin parasite

- Suppression des lignes ("on ne peut plus partir du noeud de départ") et des colonnes ("on ne peut plus arriver au noeud d'arrivée") correspondant à celles du trajet choisi (i.e. Si l'on choisit 3-1, on va supprimer la ligne 3 et la colonne 1 de la matrice des coûts).

- Appel récursif pour établir la suite du trajet, jusqu'à ce que la matrice des coûts soit vide.

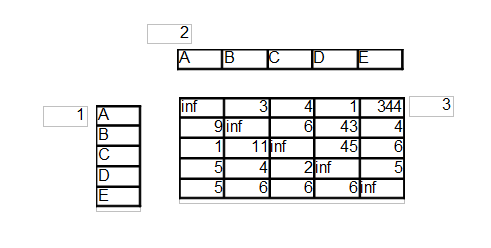
Nous avons ajouté une nouvelle classe, Tour, qui servira à modéliser la tournée. La tournée en elle-même est représentée par une matrice 2xN, où N est la taille de la tournée. La première ligne correspond aux nœuds de départ, et la deuxième ceux d'arrivée :



Ici, la tournée est donc (X1,X2)-(X2,X3)-...-(XN-1,XN).

La propriété cost est le coût total de la tournée.

Pour réaliser l'algorithme de little, nous avons choisi de représenter la matrice des coûts par une matrice. Les identifiants des noeuds sont stockés dans deux matrices 1xN.



1. Nœuds de départ des arcs

2. Nœuds d'arrivé des arcs

3. Matrice des coûts

L'exemple ci-dessus est celui d'un graphe complet d'ordre 5.

### 3.3.4-Conclusion

Pour arriver à la mise en place de cet algorithme nous avons dû cette fois-ci trouver les différentes étapes à réaliser et les tests à effectuer pour trouver un résultat correct. L'implémentation sur 2 semaine nous a permis de trouver comment décomposer en différentes étapes le déroulement de l'algorithme beaucoup d'exemple sont facilement trouvable dans des livres où internet mais ne résout.

# 4-Conclusion

Grâce à ce projet nous avons pu explorer de nombreux langage de façons assez avancer tout en profitant des méthodes et concept vue en début d'année tel que la conception orientée Objet.

Ce projet à était construit sous forme d'étape qui nous on aidait à nous calibrer et dans le même temps nous ont fait découvrir les conditions dans lesquels peuvent se réaliser un projet dans une entreprise. Les rapport et soutenance nous ont aidés dans la mise en place de notre implémentation pour correspondre aux exigence attendue.