Année universitaire: 2024-2025

RAPPORT DE PROJET – PROJET GRAPHES ET ALGO

Implémentation Graphique et Algorithmique des Structures de Graphe

Projet réalisé par :

FERHANI Youssef

HOFFSTETTER Anthony

MAKHLOUF Sanad

BACHIRI Mehdi

Projet encadré par :

Elbaz Mounir

mounir.elbaz@uha.fr

Introduction

Dans le cadre de la Licence 3 Informatique, le projet « Graphes et Algorithmes » a pour objectif principal de mettre en pratique les notions étudiées en cours et en TD, notamment en matière de structures de données et d'algorithmes sur les graphes. Le projet consiste en la conception et la réalisation d'une application logicielle, dotée d'une interface graphique, permettant de manipuler différents types de graphes, ainsi que d'exécuter un ensemble d'algorithmes classiques.

L'ensemble du développement a été réalisé en C++ à l'aide du framework Qt, choisi pour sa robustesse, sa portabilité et sa richesse en composants graphiques. Le logiciel permet de créer, importer, visualiser, éditer et sauvegarder des graphes orientés ou non, pondérés ou non, selon plusieurs structures internes (matrice d'adjacence, FS/APS, listes dynamiques). Il propose en outre l'exécution d'algorithmes fondamentaux tels que Dijkstra, Kruskal, Tarjan, l'ordonnancement par chemin critique, entre autres.

Ce rapport présente dans un premier temps le cahier des charges du projet, suivi d'une description de la conception du logiciel, puis détaille l'interface graphique, les algorithmes implémentés, les formats de sauvegarde, et se termine par une réflexion sur les problèmes rencontrés et les perspectives d'amélioration.

Table des matières :

1. Cahier des charges

1.1 Contraintes techniques

Le projet devait être réalisé en langage C++ en respectant une architecture orientée objet. L'interface graphique devait être développée à l'aide d'une bibliothèque adaptée, parmi lesquelles Qt, TCL/TK, JavaFX ou OpenGL, et compatible avec les systèmes Windows ou Linux. Le choix s'est porté sur Qt, un framework mature et complet pour le développement multiplateforme d'interfaces graphiques. Qt offre des composants performants pour la gestion de la fenêtre principale, des dialogues, des événements utilisateurs, ainsi que pour le rendu graphique (QGraphicsScene, QGraphicsView).

L'application devait permettre :

- La manipulation de graphes orientés ou non orientés, pondérés ou non ;
- Le traitement d'un graphe courant (actif) avec la possibilité de le sauvegarder et de le recharger ;
- L'interaction avec l'utilisateur via des saisies multiples (clavier, fichiers, interface graphique) ;
- Un affichage graphique et alphanumérique du graphe ;
- L'exécution d'algorithmes spécifiques avec visualisation des résultats.

Enfin, un soin particulier devait être porté à la lisibilité du code, à sa modularité, et à la documentation des fonctions majeures.

1.2 Fonctionnalités attendues

L'application devait offrir les fonctionnalités suivantes :

➢ **Saisie du graphe :**

- **Saisie manuelle par clic** via une interface graphique ;
- **Saisie à partir de fichiers texte** (.txt), selon un format simple : une ligne par arc avec possibilité de poids ;
- **Saisie graphique**, optionnelle (non prioritaire).

➢ **Affichage du graphe :**

- **Affichage graphique interactif** : sommets représentés par des cercles, arcs par des segments fléchés, mise en évidence des poids et orientation ;
- **Affichage textuel** :
    - Matrice d'adjacence,
    - Structure FS/APS ou listes chaînées dynamiques (si applicable).

➢ **Édition :**

- Ajout, suppression et déplacement de sommets ;
- Ajout, suppression et modification d'arêtes (poids, direction) ;
- Modification des propriétés du graphe (orientation, pondération) ;
- Numérotation automatique des sommets à la création.

➢ **Algorithmes à implémenter :**

L'application devait intégrer les algorithmes suivants, exécutables sur le graphe courant :

- Calcul des distances (BFS pour non pondéré, Dijkstra et Dantzig pour pondéré) ;
- Calcul des rangs sur graphe orienté acyclique (tri topologique) ;
- Détermination des composantes fortement connexes (algorithme de Tarjan) et construction du graphe réduit ;
- Détection des points d'articulation et des isthmes dans les graphes non orientés;
- Algorithme de Kruskal pour l'arbre couvrant minimal ;
- Ordonnancement de tâches avec chemin critique.

Chaque algorithme doit afficher ses résultats sous forme exploitable : matrices, listes de sommets/arcs, durée totale, chemins, etc.

1.3 Structures de données à utiliser

Trois structures de représentation de graphe étaient imposées :

- **Matrice d'adjacence** : utilisée pour les graphes denses ou simples à parcourir ;
- **FS/APS (File de Successeurs / Adresses du Premier Successeur)** : représentation compacte bien adaptée aux graphes;
- **Listes dynamiques chaînées** : pour les graphes pondérés.

L'application devait permettre de convertir un graphe d'une structure à une autre. Ces conversions sont essentielles pour adapter la structure aux besoins de l'algorithme en cours d'exécution.

1.4 Algorithmes à implémenter

Les algorithmes suivants ont été spécifiés dans le cahier des charges, chacun devant être correctement implémenté, testé et documenté :

- Parcours en largeur (BFS)
- Dijkstra
- Dantzig (Floyd-Warshall)
- Tarjan
- Rang des sommets
- Points d'articulation et isthmes
- Kruskal
- Ordonnancement par chemin critique

Chaque algorithme devait respecter ses contraintes d'application (par exemple : poids positifs pour Dijkstra, graphe orienté acyclique pour le calcul de rangs, etc.), et être accompagné d'un exemple concret pour validation.

2. Conception du projet

2.1 Choix des structures de données

Afin de répondre aux exigences du cahier des charges et de garantir la compatibilité avec les différents algorithmes, plusieurs représentations de graphe ont été intégrées dans le projet. Le choix de la structure utilisée dépend du type de graphe manipulé et des algorithmes appliqués.

➢ Matrice d'adjacence

Cette structure a été utilisée pour sa simplicité d'implémentation et son accessibilité directe. Elle est particulièrement adaptée aux graphes denses et permet un accès rapide aux relations entre sommets. Elle a notamment été utilisée pour les algorithmes nécessitant des vérifications fréquentes de l'existence d'un arc entre deux sommets (ex : Dantzig, Kruskal).

➢ FS/APS (File de Successeurs / Adresses du Premier Successeur)

Cette représentation est adaptée aux graphes orientés. Elle permet un stockage mémoire plus compact qu'une matrice, surtout pour les graphes clairsemés. Elle a été utilisée notamment dans le cadre du tri topologique et du calcul des rangs. La structure est constituée :

- D'un tableau FS contenant les successeurs des sommets de manière concaténée ;
- D'un tableau APS indiquant les positions de début de chaque liste de successeurs dans FS.

➢ Listes dynamiques chaînées

Les listes dynamiques chaînées ont été choisies pour représenter les graphes valués car elles offrent une grande flexibilité et une gestion efficace de la mémoire. Dans un graphe valué, chaque arête possède un poids (ou coût) en plus de relier deux sommets.

Grâce aux listes chaînées, il est possible de représenter, pour chaque sommet, la liste de ses voisins directs avec le poids de l'arête correspondante. Chaque élément de la liste contient donc deux informations essentielles : le sommet d'arrivée et le poids de l'arête, ainsi qu'un pointeur vers l'élément suivant.

➢ Conversion entre structures

Des fonctions ont été implémentées pour assurer la conversion entre les différentes représentations. Cela permet à l'utilisateur de basculer d'un mode de traitement à un autre selon les algorithmes utilisés. Par exemple, il est possible de passer d'une matrice à une structure FS/APS avant d'exécuter un algorithme de tri topologique.

2.2 Architecture logicielle

Le projet a été conçu selon une architecture modulaire, facilitant la maintenance, l'évolutivité, et la séparation des responsabilités.

➢ Présentation des classes principales

- **Graphe**
  Classe principale représentant un graphe standard (non valué). Elle contient les structures internes du graphe (matrice, listes, FS/APS), ainsi que les fonctions de base : ajout de sommets et d'arcs, vérification de connexité, etc.

- **GrapheValue**
  Dérivée de la classe Graphe, elle permet la gestion des graphes pondérés. Les poids sont pris en compte dans les algorithmes tels que Dijkstra, Dantzig ou Kruskal. Cette classe ajoute la gestion des coûts associés à chaque arc.

- **Algorithms**
  Contient l'ensemble des algorithmes implémentés. Chaque algorithme est défini comme une fonction indépendante prenant un graphe et les éventuels paramètres nécessaires (sommet source, etc.). Les fonctions sont organisées pour respecter la séparation entre logique métier et interface graphique.

- **MainWindow et dialogues (ajoutergraphedialog, ajouterarcsdialog, etc.)**
  Ces classes gèrent l'interface graphique à l'aide de Qt. Elles permettent la création et l'édition de graphes, le lancement des algorithmes, et l'affichage des résultats (textuels et graphiques).

➢ Organisation générale

Le code est organisé selon un modèle MVC implicite :

- Le **modèle** correspond aux classes Graphe et GrapheValue qui manipulent les données du graphe ;

- La **vue** est assurée par les composants Qt (fenêtres, scènes graphiques, boîtes de dialogue) ;

- Le **contrôleur** est représenté par la classe MainWindow qui coordonne les actions utilisateur avec les traitements à effectuer.

➢ Modularité et réutilisabilité

Chaque module est conçu pour être réutilisable indépendamment. Les algorithmes sont séparés des données et de l'interface, ce qui permet de les tester en mode console ou d'intégrer de nouvelles structures sans modifier l'interface principale.

3. Interface graphique

3.1 Choix techniques

Pour le développement de notre application de manipulation de graphes, nous avons opté pour le framework Qt, un choix qui s'est avéré particulièrement pertinent pour plusieurs raisons :

Qt Framework : Nous avons choisi Qt pour son écosystème complet et sa maturité dans le développement d'interfaces graphiques. Qt offre une bibliothèque riche de widgets et d'outils de mise en page, ainsi qu'un système de signaux et slots qui facilite la programmation événementielle.

Avantages de Qt :

- Multiplateforme : Qt permet un développement cross-platform.
- Documentation complète et communauté active.
- Outils de développement intégrés (Qt Creator).
- Support natif des graphiques 2D via QGraphicsScene et QGraphicsView.
- Gestion simplifiée des événements utilisateur.

3.2 Fonctionnalités de l'interface

Saisie :

Notre application propose plusieurs méthodes de saisie pour la création de graphes :

- **Saisie manuelle :**
    - Interface graphique interactive pour ajouter des sommets et des arêtes.
    - Boîtes de dialogue pour la saisie des propriétés (nom, orientation, valuation).
    - Validation en temps réel des entrées.

- **Saisie par fichier :**
    - Support des fichiers texte (.txt, .dat).
    - Format simple : une ligne par arc (u v [poids]).
    - Gestion des commentaires (lignes commençant par #).

Affichage :

L'application offre plusieurs modes d'affichage :

- **Affichage graphique :**

    - Représentation visuelle des sommets et des arêtes.
    - Mise en page automatique des sommets.
    - Indication claire de l'orientation des arcs.
    - Affichage des poids pour les graphes valués.

- **Affichage alphanumérique :**

    - Liste des graphes avec leurs propriétés
    - Matrice d'adjacence
    - FS et APS

- **Affichage des résultats :**

    - Résultats des algorithmes sous forme de texte.
    - Matrices de distances.
    - Chemins et composantes.

Algorithmes :

L'application permet d'exécuter les différents algorithmes sur les graphes saisies, elle dispose également d'un onglet contenant une brève explication de chaque algorithme.

4. Implémentation des algorithmes

4.1 Calcul des distances

Le parcours en largeur (BFS – *Breadth-First Search*) est utilisé pour explorer un graphe de manière systématique à partir d'un sommet donné. Il permet notamment de déterminer la distance minimale (en nombre d'arcs) entre un sommet source et tous les autres dans un graphe non pondéré.
Dans le cadre de ce projet, on l'utilise pour calculer la matrice des distances entre tous les couples de sommets.

L'algorithme BFS fonctionne à l'aide d'une file FIFO :

1. On initialise la distance du sommet source à 0, les autres à ∞.

2. On place le sommet source dans une file.

3. Tant que la file n'est pas vide :
    - On dépile un sommet u.
    - Pour chaque voisin `v` non encore visité, on assigne `dist[v] = dist[u] + 1`, et on l'ajoute à la file.
4. Le processus continue jusqu'à ce que tous les sommets atteignables soient traités.

Pour obtenir la matrice des distances, on applique ce BFS pour chaque sommet du graphe (complexité : O(n·(n + m))).

4.2 Calcul des rangs

Le calcul des rangs permet d'attribuer à chaque sommet d'un graphe orienté acyclique un entier représentant la longueur du plus long chemin partant d'un sommet sans prédécesseur.

L'algorithme s'appuie sur un tri topologique réalisé via une propagation des rangs :

- On initialise le tableau `rang` à -1 pour tous les sommets.
- On commence par calculer le nombre de prédécesseurs de chaque sommet.
- Les sommets sans prédécesseurs (i.e. `d[i] == 0`) sont placés dans une file et leur rang est fixé à 0.
- Tant que la file n'est pas vide :
    - On dépile un sommet u.
    - Pour chacun de ses successeurs `v`, on décrémente `d[v]` (nombre de prédécesseurs restants).
    - Si `d[v] == 0`, alors tous les prédécesseurs de `v` ont été traités : on lui assigne le rang `rang[v] = rang[u] + 1` et on l'ajoute à la file.
- Si à la fin tous les sommets ont été traités, le graphe est acyclique et les rangs sont valides.
- Sinon, la présence d'un circuit est détectée.

Cet algorithme n'est applicable que sur les graphes orientés.

Complexité : O(n + m), avec n le nombre de sommets et m le nombre d'arcs.

4.3 Tarjan

L'algorithme de Tarjan permet de déterminer les composantes fortement connexes (CFC) d'un graphe orienté.
Deux sommets sont fortement connexes s'il existe un chemin de l'un vers l'autre dans les deux sens.
Cet algorithme est utile pour analyser la structure d'un graphe, identifier des sous-ensembles fortement liés, et simplifier le graphe en un graphe réduit.

Dans ce projet, on utilise Tarjan pour regrouper les sommets en composantes fortement connexes, puis pour construire le graphe réduit, où chaque CFC devient un sommet. A noter que lors de l'exécution de cet algorithme, le graphe réduit est dessiné et stocké dans la liste des graphes de notre logiciel.

L'algorithme est basé sur une exploration en profondeur (DFS) avec des indices d'exploration (num) et des valeurs basses (low) :

On initialise les tableaux num et low à 0.

On parcourt le graphe avec une fonction récursive DFS :

- À chaque sommet visité u, on affecte num[u] = low[u] = compteur++.
- On empile le sommet et on le marque comme étant en pile.
- Pour chaque successeur v :
    - - S'il n'a pas encore été visité, on lance un DFS(v) et on met à jour low[u] = min(low[u], low[v]).
    - - S'il est en pile, on met à jour low[u] = min(low[u], num[v]).

Quand low[u] == num[u], cela signifie que u est le sommet d'une nouvelle composante fortement connexe : on dépile tous les sommets jusqu'à revenir à u.

Complexité : O(n + m), avec n le nombre de sommets et m le nombre d'arcs.

4.4 Points d'articulation et isthmes

Les points d'articulation et les isthmes sont des éléments critiques dans un graphe non orienté.
Un point d'articulation est un sommet dont la suppression augmente le nombre de composantes connexes.
Un isthme (ou pont) est un arc dont la suppression déconnecte une partie du graphe.

L'algorithme repose sur une exploration en profondeur (DFS) avec des indices d'ordre et des valeurs basses (low) :

On initialise deux tableaux num et low à 0.
On parcourt chaque sommet avec un DFS en attribuant un numéro d'ordre d'exploration (num[u]) et en calculant low[u], la plus petite valeur de num atteignable depuis u ou ses descendants.

Lors du parcours :

- Si low[v] ≥ num[u] (et que u n'est pas la racine), alors u est un point d'articulation.
- Si low[v] > num[u], alors l'arête (u, v) est un isthme.
- Si u est la racine et possède plus d'un enfant dans l'arbre DFS, c'est aussi un point d'articulation.

Complexité : O(n + m), avec n le nombre de sommets et m le nombre d'arêtes.

4.5 Dijkstra

L'algorithme de Dijkstra permet de calculer les plus courts chemins depuis un sommet source vers tous les autres sommets d'un graphe pondéré, à condition que tous les poids soient positifs.
Il est utilisé ici pour déterminer les distances minimales dans un graphe pondéré orienté ou non orienté.

L'algorithme fonctionne de manière gloutonne :

On initialise toutes les distances à ∞ sauf celle du sommet source, fixée à 0.
On marque tous les sommets comme non visités.

À chaque itération :

- On sélectionne le sommet non visité ayant la plus petite distance temporaire.
- On le marque comme visité.
- On met à jour les distances de ses voisins si un chemin plus court est trouvé via ce sommet.

Le processus continue jusqu'à ce que tous les sommets soient visités ou que les distances restantes soient infinies.

Dans ce projet, le sommet source est fixé par défaut au sommet 1. Le programme affiche les distances vers tous les autres sommets.

Complexité : $O(n^2)$, avec n le nombre de sommets.

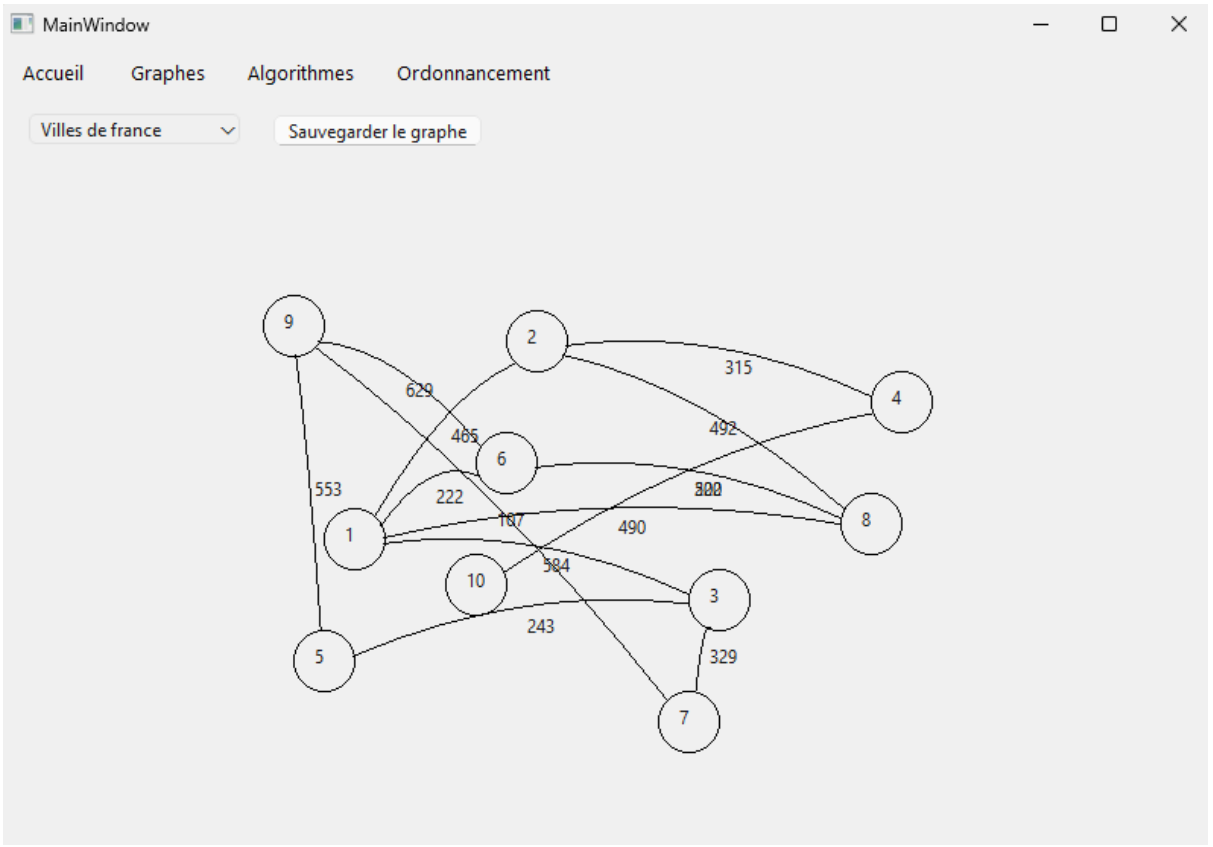
Exemple : Exécution de l'algorithme de Dijkstra sur 10 villes françaises avec leurs distances réelles

Fichier texte contenant le graphe :

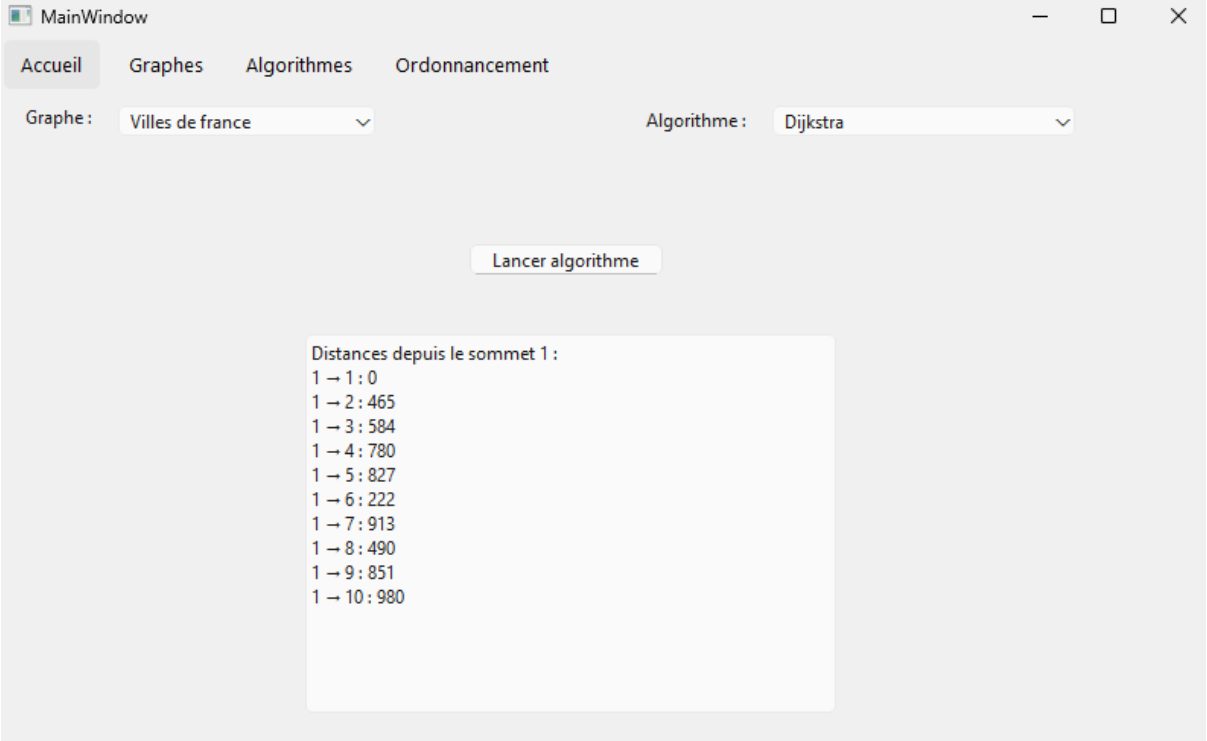
```
# 1 = Paris
# 2 = Lyon
# 3 = Bordeaux
# 4 = Marseille
# 5 = Toulouse
# 6 = Lille
# 7 = Nantes
# 8 = Strasbourg
# 9 = Rennes
# 10 = Nice

1 2 465
1 3 584
1 6 222
1 8 490
2 4 315
2 8 492
3 5 243
3 7 329
4 10 200
5 9 553
6 9 629
7 9 107
8 6 522
```

Graphe importé :



Exécution de l'algorithme et résultat :



Nous avons donc comme résultat, la distance la plus courte de Paris vers les 9 autres villes de France.

4.6 Dantzig

L'algorithme de Dantzig, aussi appelé Floyd-Warshall, permet de calculer les plus courts chemins entre tous les couples de sommets dans un graphe pondéré.
Il fonctionne aussi bien sur des graphes orientés que non orientés, tant qu'il n'y a pas de cycle de poids négatif.

Dans ce projet, l'algorithme est utilisé pour produire une matrice des plus courts chemins, où chaque case (i, j) représente la distance minimale entre les sommets i et j.

L'algorithme suit le principe suivant :

On initialise la matrice des distances avec les poids des arcs du graphe (et ∞ pour les couples non reliés).
La distance de chaque sommet à lui-même est fixée à 0.

On applique ensuite trois boucles imbriquées :

Pour chaque sommet intermédiaire k :
    Pour chaque sommet i :
        Pour chaque sommet j :
            Si le chemin i → k → j est plus court que le chemin actuel i → j, on le met à jour.

À la fin, si une case (i, i) est strictement négative, cela indique la présence d'un cycle de poids négatif.

Complexité : $O(n^3)$, avec n le nombre de sommets.

4.7 Kruskal

L'algorithme de Kruskal permet de construire un arbre recouvrant de poids minimal dans un graphe non orienté pondéré.
Il est utilisé pour connecter tous les sommets d'un graphe tout en minimisant le poids total, sans créer de cycle.

Dans ce projet, l'algorithme s'applique uniquement aux graphes non orientés pondérés, et renvoie un sous-ensemble d'arcs constituant un arbre couvrant minimal.

L'algorithme suit les étapes suivantes :

On récupère tous les arcs pondérés du graphe et on les trie par poids croissant.

On initialise une structure d'union-find pour gérer les composantes connexes.

On parcourt les arcs dans l'ordre croissant :

- Si les extrémités de l'arc appartiennent à deux composantes différentes, on les fusionne (union) et on ajoute l'arc à l'arbre.

- Si elles sont déjà connectées, on ignore l'arc pour éviter un cycle.

Le processus s'arrête dès que l'on a ajouté n – 1 arcs (avec n le nombre de sommets).

Complexité : O(m log m), avec m le nombre d'arcs.

4.8 Ordonnancement – Chemin critique

L'algorithme d'ordonnancement permet de planifier un ensemble de tâches avec contraintes, en identifiant la durée minimale d'un projet ainsi que son ou ses chemins critiques.
Un chemin critique est une séquence de tâches dont toute variation de durée retarde l'ensemble du projet.
Ce type d'analyse est essentiel dans la gestion de projets.

Dans ce projet, chaque tâche possède une durée et une liste de prédécesseurs. Le graphe est modélisé comme un graphe orienté pondéré, où :

- Chaque sommet représente une tâche.

- Les arcs représentent des dépendances avec une pondération égale à la durée de la tâche source.
- Deux sommets fictifs sont ajoutés :
    - DT (départ) : relié à toutes les tâches sans prédécesseur.
    - FT (fin) : relié à toutes les tâches sans successeur.

L'algorithme se décompose en plusieurs étapes :

1. Construction du graphe orienté pondéré à partir des tâches.
2. Calcul des dates au plus tôt en effectuant un tri topologique, en propageant la durée totale depuis DT.
3. Calcul des dates au plus tard en remontant depuis FT pour estimer les marges.
4. Identification des arcs critiques, c'est-à-dire ceux avec marge nulle.
5. Recherche de tous les chemins critiques de durée maximale reliant DT à FT.

Les résultats incluent :

- Les différents chemins critiques (en noms de tâches).
- La durée totale minimale du projet.

Complexité : O(n + m) pour le tri topologique et le calcul des dates.

5. Sauvegarde et chargement

- **Format de sauvegarde**

L'application utilise un format de fichier texte simple et lisible :

- Une ligne par arc.
- Format : sommet_départ sommet_arrivée [poids] .
- Commentaires supportés (lignes commençant par #) .
- Compatible avec les graphes orientés et valués.

- **Fonction de chargement**

Le chargement des graphes est géré de manière robuste :

- **Validation des données :**
    - Vérification du format des lignes
    - Détection des sommets manquants
    - Gestion des erreurs de syntaxe
- **Reconstruction du graphe :**
    - Création automatique des sommets nécessaires
    - Ajout des arcs avec leurs poids
    - Préservation des propriétés (orientation, valuation)

6. Extensibilité et évolutivité de l'application

L'architecture du projet a été pensée dès le départ pour permettre des évolutions simples et efficaces, que ce soit au niveau des structures de données, des algorithmes ou de l'interface utilisateur. Cette approche modulaire offre une grande souplesse pour intégrer de nouvelles fonctionnalités sans remettre en cause les bases existantes.

Une architecture conçue pour l'extensibilité :

Trois principes fondamentaux assurent la flexibilité de l'application :

- **Séparation claire des couches :**
    - La logique métier (classes `Graphe`, `GrapheValue`) est totalement découplée de l'interface utilisateur (Qt).
    - Les algorithmes (classe `Algorithms`) utilisent des interfaces génériques pour manipuler les graphes, sans dépendre de leur structure interne.
- **Modularité des composants :**
    - Chaque type de représentation de graphe (matrice, FS/APS, listes) implémente une interface commune avec les opérations de base (`ajouterArc()`, `supprimerSommet()`, etc.).
    - Les fonctions de conversion entre structures sont centralisées dans un module dédié.
- **Configuration centralisée :**

    - Les paramètres globaux (types de graphes, formats de fichiers, etc.) sont définis dans des fichiers de configuration facilement modifiables.

Exemple d'extension : ajout de l'algorithme de Bellman-Ford :

L'ajout de l'algorithme de Bellman-Ford (gestion des poids négatifs) illustre bien cette extensibilité. Voici les étapes nécessaires :

- **Implémentation de l'algorithme :**
    - Créer une méthode `bellmanFord()` dans la classe `Algorithms`, en s'inspirant de la structure de `dijkstra()`.
    - Exploiter les méthodes de `GrapheValue` pour parcourir les arcs et récupérer les poids.
- **Intégration à l'interface :**
    - Ajouter un bouton dans l'onglet *Distances*.
    - Connecter ce bouton à un slot dans `MainWindow` qui lance l'algorithme et affiche les résultats.
- **Tests et validation :**
    - Vérifier la compatibilité avec les différentes structures (matrices, listes dynamiques).
    - Ajouter un graphe avec poids négatifs aux fichiers de test.

Grâce à l'architecture actuelle, cette extension ne nécessiterait pas plus de deux heures de travail.

Perspectives d'évolution

Plusieurs pistes d'évolution peuvent être envisagées sans refonte majeure :

- **Nouveaux types de graphes :**
    - Graphes mixtes (orientés et non orientés) en adaptant la classe `Graphe`.
    - Graphes dynamiques avec gestion optimisée des insertions/suppressions via des listes chaînées.

- **Interopérabilité accrue :**
    - Possibilité de remplacer Qt par une autre bibliothèque graphique (comme ImGui) sans impacter la logique métier.
    - Prise en charge de formats standards (GraphML, DOT) pour l'import/export.
- **Optimisations :**
    - Intégration de structures plus performantes (matrices creuses, arbres de recherche) pour les grands graphes.

Cette architecture robuste et adaptable fait du projet une excellente base pour des développements futurs, aussi bien pédagogiques que professionnels.

7. Problèmes rencontrés et solutions

7.1 Aspects techniques

Pendant le développement de notre application, plusieurs défis techniques se sont présentés :

Conversion entre différentes structures de graphes :

Nous avons choisi de gérer les graphes via plusieurs représentations : matrice d'adjacence, listes FS/APS, et listes dynamiques. Passer de l'une à l'autre de manière fluide a demandé une organisation claire du code.

Affichage graphique dynamique :

L'un des plus gros enjeux a été de synchroniser en temps réel les données du graphe et leur représentation graphique. L'utilisation des classes `QGraphicsScene` et `QGraphicsItem` de Qt a facilité la création et la manipulation des éléments graphiques, mais a demandé un certain temps d'apprentissage. Nous avons finalement trouvé un bon équilibre entre logique métier et mise à jour visuelle.

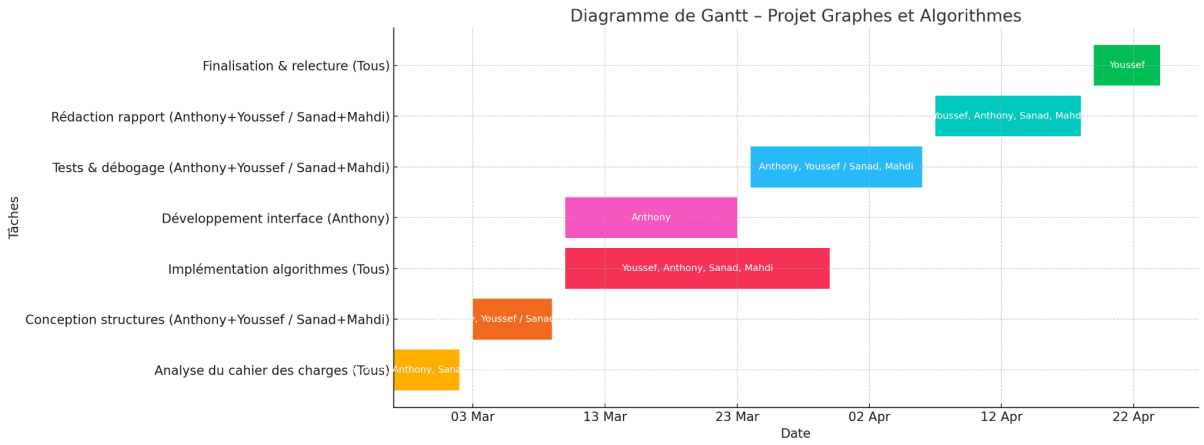
7.2 Aspects organisationnels

Répartition du travail :

Travailler en groupe nécessite une coordination constante. Pour éviter les conflits et optimiser l'efficacité, nous avons structuré le projet en modules (interface, algorithmes, structures) et utilisé Git pour gérer les versions et faciliter l'intégration continue. Chacun a pu avancer sur sa partie tout en restant aligné avec les autres.

Gestion du temps :

Comme souvent dans les projets universitaires, respecter les délais tout en maintenant la qualité n'a pas toujours été simple. Nous avons mis en place des réunions hebdomadaires pour faire le point sur l'avancement, redistribuer les tâches si nécessaire, et maintenir un bon rythme.



7.3 Limitations actuelles :

Même si l'application est fonctionnelle, certaines limites subsistent :

- L'édition graphique est encore basique : il manque des options comme annuler/revenir, ou le redimensionnement automatique de la scène.
- Certains algorithmes, comme celui de Dantzig, fonctionnent bien mais ne sont pas optimisés pour de très grands graphes, ce qui rend les affichages parfois lourds.
- L'interface utilisateur, bien que complète, pourrait être améliorée avec des retours plus clairs (pop-ups d'erreurs, messages de confirmation…).

8. Conclusion

8.1 Bilan du projet

Ce projet nous a permis d'allier théorie et pratique autour des structures et algorithmes de graphes. Nous avons non seulement implémenté des algorithmes complexes, mais aussi appris à les intégrer dans une application graphique complète et interactive.

Les principaux objectifs que nous nous étions fixés ont été atteints :

- Support de plusieurs structures de graphes (matrice, listes…)
- Intégration d'algorithmes fondamentaux (Dijkstra, Kruskal, Tarjan, etc.)
- Interface graphique interactive et intuitive
- Saisie, édition, affichage et sauvegarde de graphes

Ce projet constitue désormais une base solide qui pourrait facilement être enrichie.

8.2 Apports personnels et collectifs

Sur le plan personnel, chacun d'entre nous a renforcé des compétences spécifiques :

- Programmation en C++ avec Qt
- Manipulation avancée de structures de données
- Développement d'interfaces graphiques complexes
- Implémentation et compréhension profonde d'algorithmes

En tant qu'équipe, nous avons appris à travailler ensemble efficacement : partager du code proprement, résoudre les bugs en groupe, organiser notre travail, et surtout, nous adapter face aux imprévus.

8.3 Pistes d'amélioration

Le projet peut encore évoluer sur plusieurs aspects :

- Ajouter de nouveaux algorithmes (Bellman-Ford, Prim, A*, etc.)
- Améliorer l'expérience utilisateur : couleurs personnalisables, messages d'aide, meilleure gestion des erreurs
- Optimiser les performances pour les très grands graphes
- Ajouter de nouveaux formats d'import/export (JSON, GraphML…)

- Déployer l’application sur d’autres plateformes, ou même sur le web via Qt WebAssembly