Projet ALMF51 – Rapport

Algorithmes de graphes – Parcours, optimisation et applications

Introduction

Ce projet consiste en l'implémentation de plusieurs algorithmes classiques de théorie des graphes dans une application interactive. Chaque algorithme est utilisé pour répondre à un cas concret relatif à un réseau routier entre plusieurs villes françaises.

L'objectif est double :

- approfondir la compréhension des algorithmes de graphes (théorie),
- les mettre en pratique via une implémentation en langage de programmation (C#, Python, etc.).

Le rapport est structuré en quatre parties, correspondant aux quatre blocs d'algorithmes étudiés.

1 – Algorithmes de parcours (BFS et DFS)

1.1 - Parcours en largeur (BFS – Breadth-First Search)

1.1.1 - Théorie – Parcours en largeur (BFS)

Objectif

L'algorithme BFS (Breadth-First Search) a pour but de parcourir un graphe de manière systématique en explorant tous les sommets accessibles à une distance "1" du point de départ, puis ceux à distance "2", etc.
Il est particulièrement utilisé pour :

- trouver le plus court chemin (en termes de nombre d'arêtes) dans un graphe non pondéré
- tester la connexité d'un graphe
- explorer rapidement un graphe en couches.

Principe général

Initialisation

Chaque nœud (ou "sommet") du graphe est d'abord marqué comme "non vu".
On choisit un sommet de départ — ici, spécifié par l'utilisateur (ex : "Rennes").

Parcours en couches

On utilise une file, structure de données de type FIFO (First In, First Out), pour gérer l'ordre des visites.
Le sommet de départ est marqué comme "vu" et placé dans la file.

À chaque itération :

- On extrait le premier sommet de la file (le sommet le plus ancien à avoir été découvert).
- On explore tous ses voisins directs encore "non vus" (triés par ordre alphabétique si désiré).
- Chaque voisin découvert est marqué comme "vu", mémorisé dans l'ordre de visite, et ajouté à la file.

Fin de l'algorithme

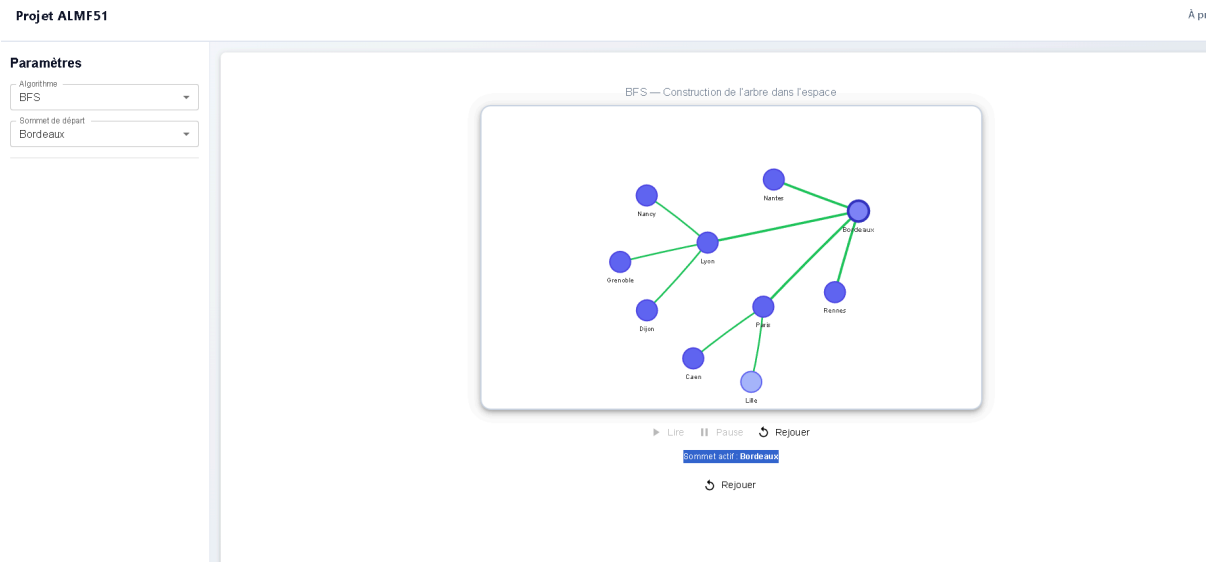
Le processus se termine quand la file est vide, c'est-à-dire lorsque tous les sommets accessibles depuis la source ont été explorés.

Complexité

L'algorithme parcourt chaque sommet et chaque arête au plus une fois.
Sa complexité est linéaire : O(V + E) où V = nombre de sommets, E = nombre d'arêtes du graphe.

L'algorithme BFS retourne :

- L'ordre de parcours des sommets
- Le tableau des parents de chaque sommet



Parcours en largeur depuis Bordeaux

1.2 - Parcours en profondeur (DFS – Depth-First Search)

1.2.1 - Théorie – Parcours en profondeur (DFS)

Objectif

L'algorithme DFS (Depth-First Search) a pour but de parcourir un graphe en explorant chaque chemin le plus en profondeur possible avant de revenir en arrière.
Il est particulièrement utile pour :

- détecter des cycles dans un graphe
- explorer toutes les composantes connexes
- tester s'il existe un chemin entre deux sommets
- générer des arbres de couvrance

Principe général

Initialisation

On part d'un sommet source donné.
On garde en mémoire un ensemble de sommets déjà visités pour éviter les boucles infinies.

Exploration en profondeur

On visite un sommet, puis, parmi ses voisins, on choisit le premier dans l'ordre alphabétique.
On poursuit la visite récursivement, en suivant le même principe.
Lorsque tous les voisins d'un sommet ont été visités, on revient en arrière pour revisiter ses autres voisins non encore explorés.

Fin de l'algorithme

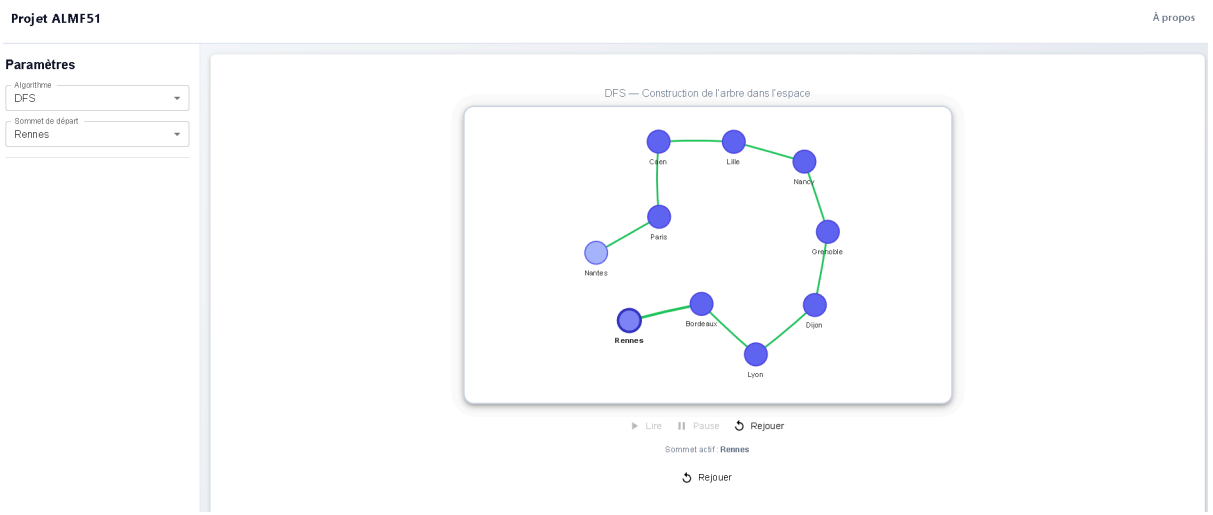
L'algorithme se termine lorsque tous les sommets atteignables depuis la source ont été visités.

Complexité

Complexité : O(V + E) où V = nombre de sommets, E = nombre d'arêtes du graphe.

L'algorithme DFS retourne :

- L'ordre de parcours des sommets
- Le tableau des parents de chaque sommet



Parcours en largeur depuis Rennes

2 – Arbre couvrant de poids minimum (Kruskal et Prim)

2.1 - Algorithme de Kruskal

2.1.1 - Théorie – Algorithme de Kruskal

Objectif

L'algorithme de Kruskal vise à construire un arbre couvrant de poids minimal (ACPM), c'est-à-dire un sous-ensemble d'arêtes qui :

- relie tous les sommets du graphe
- ne forme pas de cycle
- La somme des poids des arêtes est minimale parmi tous les arbres couvrants possibles

Principe général

Trier les arêtes

On commence par ordonner toutes les arêtes du graphe selon leur poids croissant.

Construire l'ACPM progressivement

On parcourt ensuite ces arêtes triées une par une.
On considère que le graphe est non orienté car Kruskal ne fonctionne pas sur graphe orienté.
À chaque étape, on ajoute l'arête au résultat si elle ne crée pas de cycle avec les arêtes déjà sélectionnées.
Pour cela, on vérifie si les deux extrémités de l'arête appartiennent déjà au même arbre.

Détection de cycles

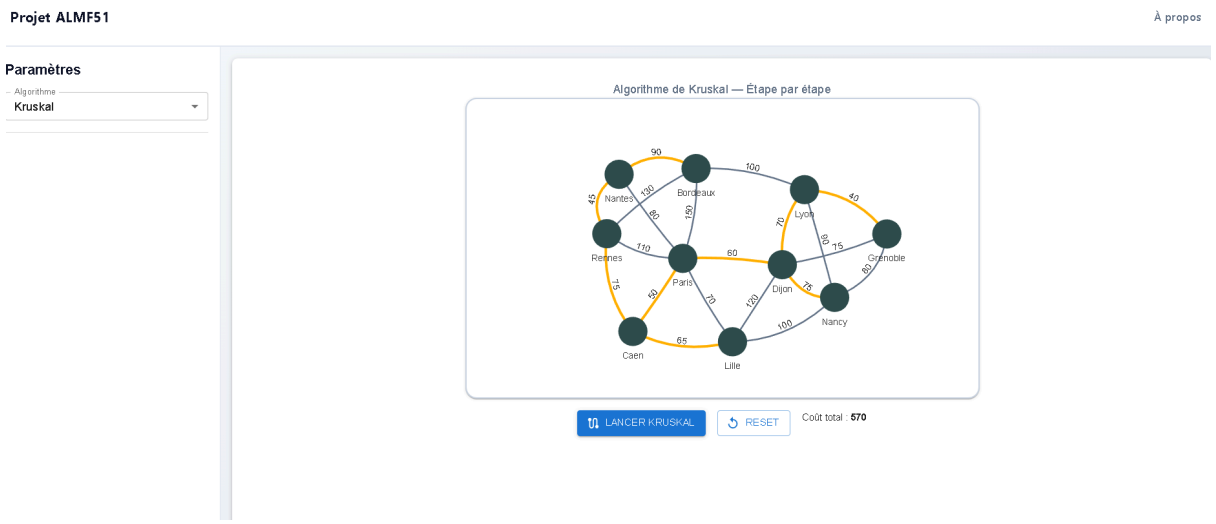
Pour chaque arête, on vérifie si les deux sommets appartiennent déjà à la même composante. Si c'est le cas, l'arête est ignorée pour éviter de former un cycle. Sinon, l'arête est ajoutée à l'arbre et les deux composantes sont fusionnées.

Complexité

Le tri initial des arêtes prend O(E log E), où E est le nombre d'arêtes.
La vérification des composantes peut nécessiter de remonter jusqu'aux racines des composantes pour chaque arête, ce qui donne une complexité globale O(E × V).

L'algorithme Kruskal retourne :

- La liste des arrêtes sélectionné
- Le coût total



Algorithme de Kruskal dans le graphe

2.2 Algorithme de Prim

2.2.1 - Théorie – Algorithme de Prim

Objectif

L'algorithme de Prim a pour objectif de construire un arbre couvrant de poids minimal (ACPM) dans un graphe pondéré et connecté.
Contrairement à Kruskal, qui sélectionne les arêtes les moins coûteuses globalement, Prim construit l'arbre progressivement à partir d'un sommet initial, en ajoutant à chaque étape l'arête de poids minimal qui relie un sommet déjà inclus à un sommet non encore visité.

Cet algorithme est particulièrement adapté aux graphes denses et aux applications nécessitant un arbre couvrant à partir d'un point de départ donné.

Principe général

Initialisation

On commence avec un sommet arbitraire (le sommet de départ).
Ce sommet est marqué comme visité et constitue le premier composant de l'arbre couvrant.

Sélection de l'arête minimale à chaque itération:

- On examine toutes les arêtes qui relient un sommet déjà inclus dans l'arbre à un sommet non encore visité.
- On sélectionne l'arête de poids minimal parmi celles-ci.
- Ajout au MST et mise à jour
- L'arête sélectionnée est ajoutée à l'arbre couvrant.
- Le sommet nouvellement connecté est marqué comme visité.

Le processus se répète jusqu'à ce que tous les sommets du graphe aient été inclus.

Détection de cycles

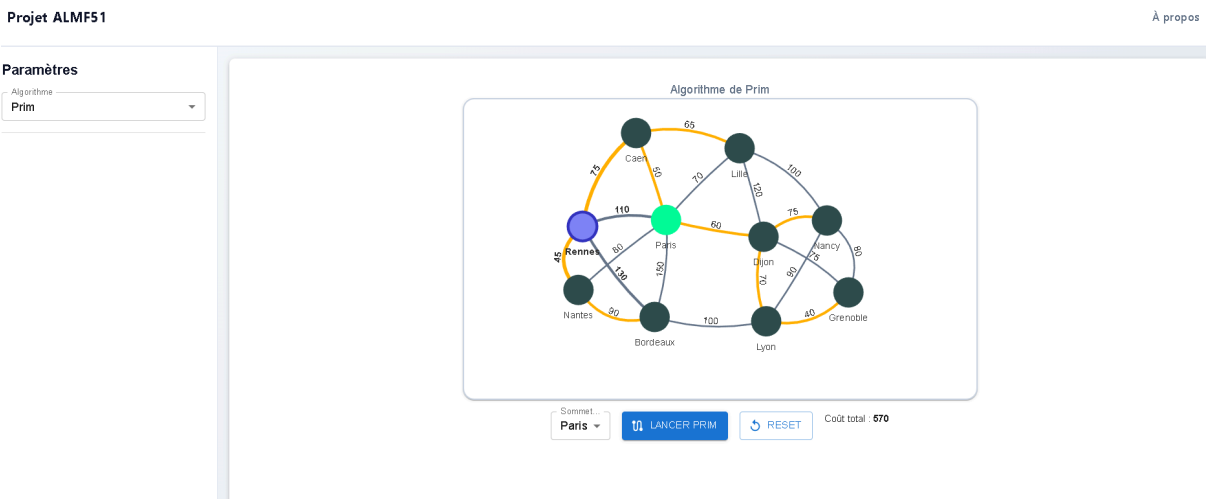
L'algorithme de Prim ne nécessite pas de structure spéciale pour détecter les cycles, car il n'ajoute jamais une arête reliant deux sommets déjà visités.
Cela garantit automatiquement que l'arbre reste acyclique tout au long de sa construction.

Complexité

Pour chaque sommet ajouté, on examine toutes les arêtes connectant l'arbre aux sommets non visités.
La complexité est O(V × E), où V est le nombre de sommets et E le nombre d'arêtes.

L'algorithme de Prim retourne :

- La liste des arrêtes sélectionné
- Le coût total



3 – Recherche du chemin optimal (Dijkstra)

3.1 - Théorie – Algorithme de Prim

Objectif

L'algorithme de Dijkstra a pour objectif de trouver le plus court chemin entre un sommet source et tous les autres sommets d'un graphe pondéré et sans poids négatifs.
Il est particulièrement utilisé pour :

- déterminer le chemin minimal dans un réseau routier ou de communication
- calculer la distance minimale entre deux points
- planifier des itinéraires optimisés dans des applications logistiques ou informatiques.

Principe général

Initialisation

On associe à chaque sommet une distance depuis la source.
La distance de la source à elle-même est initialisée à 0, et toutes les autres distances sont initialisées à l'infini.
On garde également un tableau parents pour mémoriser le chemin le plus court vers chaque sommet.

Exploration des sommets

On utilise une file de priorité (priority queue) pour toujours traiter le sommet non visité ayant la distance minimale depuis la source.
À chaque étape, on extrait le sommet current ayant la plus petite distance connue.

Relaxation des arêtes

Pour chaque voisin de current, on calcule la distance totale via current.
Si cette distance est plus courte que la distance déjà enregistrée pour ce voisin, on met à jour sa distance et on note current comme parent du voisin.

Répétition

L'algorithme continue jusqu'à ce que tous les sommets aient été visités ou que le sommet cible ait été atteint (si l'on cherche un chemin précis).

Détection et reconstruction du chemin

À la fin, la distance minimale de la source à chaque sommet est connue.

Le chemin le plus court vers un sommet donné peut être reconstruit en remontant le tableau des parents depuis ce sommet jusqu'à la source.

Si le sommet cible n'a pas de parent et n'est pas la source, cela signifie qu'il n'est pas atteignable depuis la source.

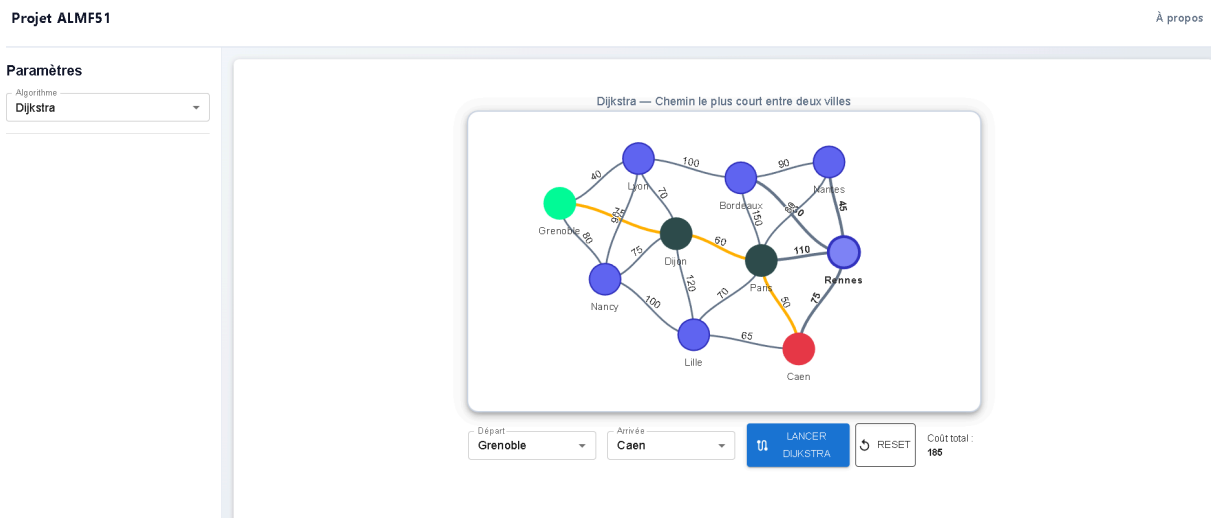
Complexité

L’algorithme visite chaque sommet et examine chaque arête au moins une fois.

Avec une priority queue, les opérations d’extraction et d’insertion prennent O(log V), ce qui donne une complexité globale O((V + E) log V), où V est le nombre de sommets et E le nombre d’arêtes.

L’algorithme Dijkstra retourne :

- Un chemin optimal
- Le coût total



Djikstra entre Grenoble et Caen

4 – Recherche du chemin optimal (Bellman-Ford et Floyd-Warshall)

4.1 - Théorie – Algorithme de Bellman-Ford

Objectif

L’algorithme de Bellman-Ford a pour objectif de calculer le plus court chemin depuis un sommet source vers tous les autres sommets dans un graphe pondéré, même si certaines arêtes ont un poids négatif.
Il est particulièrement utile pour :

gérer des graphes avec des poids négatifs,

détecter la présence de cycles de poids négatif,

calculer les distances minimales dans des réseaux où certaines transitions peuvent réduire le coût total.

Principe général

Initialisation

On associe à chaque sommet une distance depuis la source, initialisée à l'infini.

La distance de la source à elle-même est initialisée à 0.

Un tableau parents est utilisé pour mémoriser le chemin le plus court vers chaque sommet.

Relaxation répétée des arêtes

L'algorithme parcourt les arêtes du graphe et met à jour les distances si un chemin plus court est trouvé via une arête existante.

Cette opération est répétée jusqu'à ce qu'aucune distance ne puisse plus être réduite.

Propagation avec file de sommets actifs

On utilise un ensemble L pour contenir les sommets dont la distance a été modifiée.

Tant que cet ensemble n'est pas vide, on continue à propager les mises à jour vers les sommets voisins, en ajoutant à L tout sommet dont la distance est améliorée.

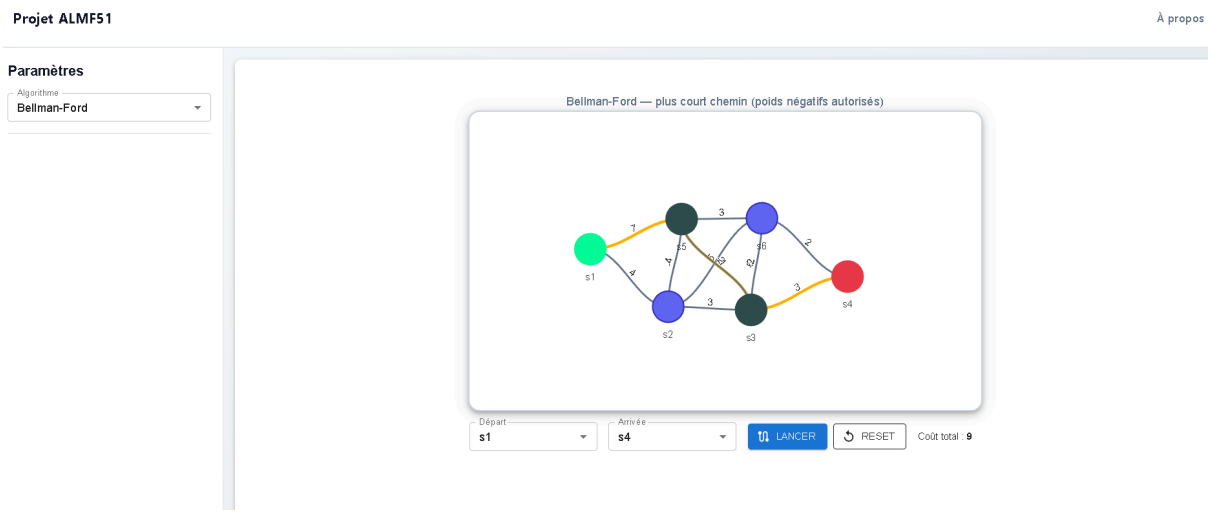
Complexité

Chaque arête est examinée plusieurs fois, et la propagation des distances se fait jusqu'à stabilisation.

Dans le pire cas, la complexité est $O(V \times E)$, où V est le nombre de sommets et E le nombre d'arêtes.

L'algorithme Bellman-Ford retourne :

- Les distances pour chaque sommet depuis le départ
- Les parents



Bellman Ford entre s1 et s4 avec des poids négatifs

4.2 - Théorie – Algorithme de Floyd-Warshall

Objectif

L'algorithme de Floyd-Warshall a pour objectif de calculer les plus courts chemins entre toutes les paires de sommets d'un graphe pondéré.
Il est particulièrement utile lorsque l'on a besoin de connaître la distance minimale entre tous les sommets et pas seulement depuis un sommet source unique.

Cet algorithme peut gérer :

les graphes avec des arêtes à poids positifs ou négatifs (mais sans cycle de poids négatif),

La reconstruction du chemin le plus court entre n'importe quelle paire de sommets.

Principe général

Initialisation

On représente les distances par une matrice dist, où dist[i, j] correspond à la distance minimale connue entre le sommet i et le sommet j.

La distance de chaque sommet à lui-même est initialisée à 0, et toutes les autres distances sont initialisées à l'infini.

Une matrice next permet de mémoriser le prochain sommet à visiter pour reconstruire le chemin le plus court.

Mise à jour itérative des distances

Pour chaque sommet k, considéré comme sommet intermédiaire potentiel, on met à jour toutes les distances dist[i, j] en testant si passer par k offre un chemin plus court :

$$\text{dist}[i,j] = \min(\text{dist}[i,j], \text{dist}[i,k] + \text{dist}[k,j])$$

Si un chemin plus court est trouvé via k, la matrice next[i,j] est également mise à jour pour mémoriser le sommet suivant sur ce chemin.

Propagation vers toutes les paires

Ce processus est répété pour tous les sommets k comme intermédiaires, ce qui permet de considérer tous les chemins possibles.

À la fin, la matrice dist contient les distances minimales entre toutes les paires de sommets, et la matrice next permet de reconstruire les chemins correspondants.

Complexité

L'algorithme effectue trois boucles imbriquées sur les sommets : pour chaque k, pour chaque i et pour chaque j.

Sa complexité est $O(V^3)$, où V est le nombre de sommets.

L'algorithme Floyd-Warshall retourne :

- Les distances entre chaque sommet
- Les parents

- Les sommets

	Rennes	Nantes	Bordeaux	Caen	Paris	Lille	Nancy	Dijon	Lyon	Grenoble
Rennes	0	45	130	75	110	140	240	170	230	245
Nantes	45	0	90	120	80	150	215	140	190	215
Bordeaux	130	90	0	200	150	220	190	170	100	140
Caen	75	120	200	0	50	65	165	110	180	185
Paris	110	80	150	50	0	70	135	60	130	135
Lille	140	150	220	65	70	0	100	120	190	180
Nancy	240	215	190	165	135	100	0	75	90	80
Dijon	170	140	170	110	60	120	75	0	70	75
Lyon	230	190	100	180	130	190	90	70	0	40
Grenoble	245	215	140	185	135	180	80	75	40	0

Floyd Warshall entre Lille et Bordeaux

Mickaël Céraline & Ghellab Ibrahim