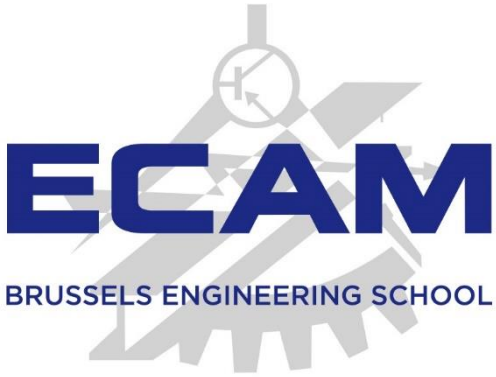
Haute Ecole ICHEC - ECAM - ISFSC



Etude algorithmique du calcul d'itinéraires sur un réseau routier dont les vitesses peuvent changer dynamiquement

Travail de fin d'études présenté par

Robert et FRANÇOIS

En vue de l'obtention du diplôme de

Master en Sciences de l'Ingénieur Industriel orientation Informatique

Année académique 2021-2022

Remerciements :

J’aimerais commencer par remercier mon promoteur de stage Sébastien Collette pour la supervision de mon travail ainsi que pour m’avoir donné les conseils qui m’ont permis de mener à bien les différents objectifs que j’ai atteint dans la partie pratique de ce TFE ainsi que pour la partie rédactionnelle.

Je remercie aussi mon superviseur de l’ECAM Philippe Dekimpe d’avoir été présent pour apporter des réponses à mes questions concernant l’organisation de ce TFE ainsi que ses conseils pour la préparation de la rédaction.

Enfin je remercie ma mère pour le travail qu’elle a effectué en relisant mon travail et en m’indiquant les endroits qui manquaient de clarté.

Résumé :

Ce TFE s'est déroulé dans une petite entreprise nommée Synapsis-group[1] et s'inscrit dans le cadre d'un projet[2] d'étude du trafic routier ainsi que des trajets des habitants de Bruxelles. Ce projet a été financé par la région Bruxelloise. Les objectifs principaux de ce projet sont d'étudier, d'optimiser, de modéliser et de récolter des informations sur les trajets des habitants, le trafic et tous ce qui concerne de près ou de loin ces domaines.

Le sujet de ce TFE concerne la partie du calcul d'itinéraires sur une carte de la Belgique avec la prise en compte de changements dynamiques du trafic routier. Le travail principal sera de développer la partie algorithmique qui gère les calculs d'itinéraires sur un graphe représentant la carte de la Belgique et dans lequel il est nécessaire de prendre en compte la dynamique des changements qui peuvent se produire sur le réseau routier d'une ville.

Il existe déjà plusieurs solutions pour résoudre la problématique du calcul d'itinéraires seulement elles sont soit trop lentes car elles nécessitent de faire beaucoup de calculs en amont d'un changement, soit elles ne permettent pas de prendre en compte certaines contraintes comme le changement dynamique du trafic sur une carte.

Pour atteindre les objectifs de ce TFE, les grandes étapes ont été les suivantes :

Dans un premier temps, les performances concernant le calcul d'itinéraires des principales technologies existantes seront d'évaluées à travers un benchmark.

Ensuite, sera abordée la réalisation d'une implémentation simple qui résolve la problématique et elle sera incorporée dans le benchmark précédemment réalisé afin de pouvoir évaluer où elle se situe d'un point de vue de ses performances par rapport aux autres technologies.

Finalement, les recherches dans différents domaines pour optimiser la solution en termes de temps de calcul afin de se rapprocher de la meilleure technologie existante seront expliquées.

[1] Site de l'entreprise : https://www.synapsis-group.com/
[2] Site du projet de départ : https://www.syty.io/

CAHIER DES CHARGES RELATIF
au TRAVAIL DE FIN D'ETUDES de

Robert FRANÇOIS inscrit en 5MIN

Année académique : 2ème Master

Titre provisoire : Etude algorithmique concernant le calcul d'itinéraires sur un réseau routier dont les vitesses peuvent changer dynamiquement.

Objectifs à atteindre : un itinéraire sur un réseau routier consiste à calculer un plus court chemin dans un graphe. Il existe de nombreux algorithmes pour résoudre ce problème, mais les implémentations les plus efficaces utilisent une approche de pré-calcul qui ne permet pas de changer dynamiquement le réseau routier.
À partir du benchmark réalisé sur les technologies principales ainsi que la nôtre, le travail consistera à optimiser notre solution afin de se rapprocher le plus possible, en termes de temps de calcul, de la meilleure des technologies avec pré-calcul, tout en autorisant des changements dynamiques fréquents du réseau routier.

Principales étapes :

1. Revue de la littérature, Dijkstra et analyse de la contraction de graphes.
2. Réalisation du benchmark des technologies principales.
3. Réalisation de la solution et comparaison par rapport au benchmark effectué.
4. Etude d'implémentation de la file d'attente prioritaire.
5. Etude de solutions pour minimiser le nombre de nœuds visités.
6. Implémentation du changement dynamique de coût des arêtes.
7. Objectif optionnel, implémentation de méthode de contraction de graphe pour minimiser le temps de calcul d'itinéraires.

Fait en trois exemplaires à Bruxelles, le 02/12/2021

L'Etudiant	Le Tuteur	Le Promoteur
Nom-prénom :	Nom-prénom :	Nom-prénom :
Robert FRANÇOIS	Philippe DEKIMPE	Sébastien Collette
	Département/Unité	Société
	Informatique	SYNOPSIS GROUP
Signature	Signature	Signature
François Robert		

Informations complémentaires :

FAP : file d'attente prioritaire = priority queue.

Tas : heap.

Vecteur : array à une dimension.

Nœud dans un arbre binaire : un nœud dans un arbre binaire est un élément contenant une valeur, ayant un parent (excepté pour le nœud racine) et au maximum deux nœuds enfants.

Nœud racine dans un arbre binaire : c'est le premier nœud dans la structure contenant les nœuds et qui, graphiquement, se trouve être tout en haut de l'arbre.

Nœud en théorie des graphes : un nœud en théorie des graphes est un élément constitutif du graphe généralement représenté par un point, à partir duquel zéro à plusieurs liaisons permettent de rejoindre un ou plusieurs autres nœuds.

Nœud source dans un graphe : nœud de départ à partir duquel l'algorithme qui est appliqué va commencer son travail. Par exemple dans la Figure 8*(a),* si l'on veut trouver un chemin entre le nœud A et D, alors A est le nœud source du chemin.

Nœud de destination dans un graphe : nœud vers lequel on veut trouver un chemin à partir d'un nœud source. Pareillement que pour le nœud source, par rapport à la Figure 8*(a),* si l'on veut trouver un chemin entre le nœud A et D, alors D est le nœud de destination du chemin.

Nœud courant dans un graphe : nœud qui a été sélectionné à partir de la liste de nœuds disponibles et sur lequel l'algorithme va commencer à travailler. Par exemple dans la Figure 8*(a),* en faisant l'hypothèse que l'on veuille trouver un chemin entre le nœud A et D, si l'algorithme qui pourcourt le graphe en est au nœuds C lors de la recherche d'un chemin et qu'il est en train de traiter celui-ci, alors C est le nœud courant.

Nœud sortant dans un graphe : nœud qui est atteignable via une des arêtes sortantes du nœud courant. Par exemple dans la Figure 8*(a),* les nœuds *F-B-C* sont des nœuds sortants de *A.*

$\lfloor x \rfloor$: équivalent de la fonction $floor(x)$ qui renvoie la valeur la plus grande étant inférieur ou égale à x, soit un arrondi à l'unité inférieur.

$\lceil x \rceil$: équivalent de la fonction $ceil(x)$ qui renvoie la valeur la plus petit étant supérieur ou égale à x, soit un arrondi à l'unité supérieur.

V : ensemble de nœuds.

E : ensemble d'arêtes.

$G = (V, E)$: Graphe G contenant un ensemble V et E.

s : nœud source où $s \in V$.

v ou u : nœud quelconque où $v, u \in V$.

$(u, v) \in E$: notation d'une arête entre deux nœuds u et v se trouvant dans l'ensemble E.

Table des matières

1 Introduction

1.1 Contexte :

Le cadre dans lequel j'ai effectué mon stage est une petite entreprise indépendante constituée de moins d'une dizaine d'employés et travaillant essentiellement sur des projets en lien avec des acteurs privés ou des institutions publiques. Les projets sur lesquels cette société travaille concerne principalement du front-end et du back-end. L'équipe utilise des technologies comme le react pour le coté front-end, et pour la partie back-end, elle emploie du dotnet à la fois pour la gestion de base de données ainsi que pour les parties algorithmiques.

Dans le monde actuel des grandes villes comme Bruxelles, l'analyse, la modélisation et la récolte d'information sur le trafic routier ainsi que les déplacements et habitudes des citoyens sont un point important pour pouvoir gérer et faciliter la vie de ceux-ci et peuvent permettre en plus, d'améliorer certaines décisions prises concernant la ville. C'est dans ce contexte là que la région Bruxelloise a décidé de lancer et financer un grand projet pour étudier, optimiser, modéliser et récolter des informations sur les trajets des habitants ainsi que le trafic en ville.

Ces informations permettront notamment de prédire en temps réel le trafic afin d'anticiper des embouteillages, de donner sur une carte les voies où la pollution est la plus présente afin de permettre aux cyclistes par exemple d'éviter certaines routes, d'indiquer l'affluence à certains endroits par lesquels passent fréquemment les personnes, ou permettre d'indiquer le degré de fréquentation d'un lieu. On pourrait par exemple déterminer si un bar ou un restaurant a une clientèle potentiellement élevée, étudier et modéliser des changements de comportements des citoyens à la suite de décisions concernant la ville et voir l'impact que cela aurait en général. Il existe outre ces exemples une multitude de choses qui pourrait être observées et analysées à des fins diverses et qui amélioreraient la vie des citoyens.

La problématique sur lequel ce TFE va se concentrer dans ce projet est une partie du projet qui concerne l'étude et l'optimisation de calcul d'itinéraires dans une ville où des changements peuvent fréquemment se produire. Pour aborder et résoudre ce problème, il existe déjà plusieurs technologies fonctionnelles. Cependant elles ont certaines limites, comme le fait qu'il soit impossible de changer dynamiquement le coût des routes constituant les différents chemins sans devoir recompiler l'ensemble de l'implémentation, ce qui peut alors prendre beaucoup de temps. Pour ces raisons, nous sommes partis sur le développement de notre propre solution.

Avant de passer à la suite, il y a quelques notions importantes à comprendre liées à cette problématique. Chacune des notions sera dans un premier temps abordées d'un point de vue général afin de cerner le problème et, dans un deuxième temps, un lien sera fait avec la théorie des graphes qui est la discipline utilisée pour résoudre ce genre de problèmes.

Premièrement qu'est-ce qu'un itinéraire ? Dans ce cadre-là, un itinéraire est une suite de points sur une carte, reliés par des traits joignant ces différents points et dont le premier et

le dernier sont définis par l'utilisateur au départ et entre lesquels il voudrait trouver un trajet. En théorie des graphes, un itinéraire peut donc être représenté comme un chemin où les points qui constituent des localisations sur une carte sont appelés nœuds, et les lignes qui relient ces nœuds entre eux des arêtes avec un coût pour les traverser. Un itinéraire est donc un chemin sur un graphe, pondéré avec un coût total qui est la somme de l'ensemble des coûts qui constituent ce chemin[3].

Deuxièmement, quel sont les principaux problèmes à résoudre ? Le premier défi consiste à trouver un itinéraire qui soit le plus court possible d'un point de vue temporel (ou de son temps de parcours) à partir d'un point A à une certaine localisation géographique, vers un point B à une autre localisation. En théorie des graphes, cela revient à trouver ou à calculer le chemin ayant le coût total le plus faible[3].

Cette tâche de recherche du plus court chemin à comme deuxième défi de trouver et calculer ce plus court chemin dans un temps qui soit lui aussi le plus court. C'est-à-dire que le temps mis pour trouver l'ensemble des nœuds et arêtes qui constitue un lien direct le plus court, entre le nœud de départ et celui d'arrivé, doit être calculé ou trouvé le plus rapidement possible.

Les différentes notions qui ont été brièvement expliquées ci-dessus permettent de comprendre la suite mais seront détaillées de manière plus formelle et mathématique à la section 2.1.

1.2 Objectifs :

Pour atteindre la partie du projet précédemment cité, voici en résumé une liste des différents grands objectifs :

1. Établir la performance entre les principales technologies existantes que sont itinero[4] et pgrouting[5] à travers un benchmark. Le but est de voir les performances qu'ont ces technologies et être capable de les comparer par rapport à notre solution.
2. Créer notre propre implémentation de la résolution du problème qui, dans notre cas sera un algorithme de Dijkstra pour effectuer des calculs d'itinéraires sur un graphe et comparer ces résultats aux autres technologies en l'ajoutant au benchmark.
3. Étudier différents domaines liés à l'algorithmique et à la théorie des graphes afin d'améliorer les performances de notre solution relative au problème dans le but de se rapprocher le plus possible de la meilleure des technologies.

1.3 Méthodologie :

Pour arriver à réaliser l'évaluation des technologies, il fallait évaluer les diverses solutions en commençant par étudier comment a été pensée leur utilisation et aussi chercher à comprendre leur fonctionnement. En effet, comme elles sont différentes, elles n'utilisent

[3] Les notions de chemin ainsi que le coût dans un graphe seront abordées de manière plus détaillée dans la section 2.1.
[4] http://www.itinero.tech/
[5] https://pgrouting.org/

pas les mêmes outils. Ensuite sur base d'une implémentation de celles-ci, il fallait réaliser un benchmark de leur performance et en faire un graphe pour pouvoir les comparer. Le benchmark consiste en l'évaluation du temps de calcul d'un ensemble d'itinéraires en en lançant un grand nombre d'affilée. Il sera décrit à la section 3.

Concernant la partie réalisation de notre solution, la première étape a été d'apprendre les bases théoriques nécessaires à la réalisation d'un algorithme de Dijkstra ainsi que certaines notions fondamentales de la théorie des graphes.

Une fois la théorie comprise, il a fallu ensuite implémenter de manière pratique cet algorithme et le tester en comparant les résultats qu'il donnait à une des technologies déjà existantes qui implémente aussi un algorithme de Dijkstra, afin de s'assurer que notre algorithme était correct.

Quand les divers tests de vérification de sa validité ont été réussis, il ne restait plus qu'à l'intégrer au benchmark à partir de données récupérées de l'exécution de l'algorithme sur un grand nombre d'itinéraire.

Pour la partie concernant l'amélioration de l'algorithme, le travail a consisté à étudier diverses pistes théoriques dans différents domaines pour essayer de réduire le temps de calcul de celui-ci afin de trouver le meilleur itinéraire, et donc le plus court chemin d'un point de vue de la théorie des graphes. Après un temps d'investissement pour analyser comment l'améliorer, il a été nécessaire de l'appliquer en pratique pour pouvoir évaluer l'impact des changements.

1.4 Structure du rapport :

Afin d'apporter un peu plus de clarté, l'organisation du document ainsi que ce qu'il contient dans les différentes sections vont être décrits de manière globale.

Premièrement, dans la section 3 on trouvera la première partie du travail qui a consisté en l'élaboration d'un benchmark des différentes technologies. Dans celui-ci nous verrons sur quel critère il a été conçu, quel sont les résultats obtenus ainsi que des explications pour pouvoir les interpréter correctement.

Deuxièmement, dans la section 4 nous aborderont le fonctionnement, l'implémentation ainsi que les spécificités de notre réalisation de l'algorithme de Dijkstra.

Troisièmement, dans la section 5 nous traiterons l'analyse théorique de la file d'attente prioritaire et de son rôle dans le projet.

Quatrièmement, dans la section 6 il sera question de la façon dont a été réalisé le changement dynamique de coût des arêtes, le contexte et le lien avec la suite du projet ainsi que les structures de données qui y sont liées.

Cinquièmement, dans la section 7 nous détaillerons les différentes voies pour améliorer les performances de l'algorithme avec, une explication de leur fonctionnement, leur implémentation et une conclusion sur leur faisabilité où les améliorations qu'elles ont apporté au projet.

Finalement, à la section 8 nous trouverons la conclusion qui résumera l'ensemble du projet et du travail accompli.

2 Contexte bibliographique

2.1 Notions fondamentales :

Cette section a pour but de décrire de manière plus formelle certaines notions abordées dans l'introduction afin de comprendre en pratique de quoi il s'agit et aussi d'exposer des notions théoriques indispensables à la bonne compréhension de la suite du travail. Les informations et notation utilisées dans cette section ont été tirées de cet ouvrage (1), p. 643-644.

2.1.1 Qu'est-ce qu'un chemin ? :

Un chemin est une séquence de nœuds ordonnés et reliés les uns à la suite des autres par une arête établissant un lien entre deux nœuds consécutifs, et dont l'ensemble de cette structure permet de joindre deux nœuds fournit au départ (un nœud source et un nœud de destination). Par exemple dans la Figure 8*(a)*, la suite de nœud {A, B, E} est un chemin, il peut être défini mathématiquement de la façon suivante :

$$p = \langle v_0, v_1, \dots, v_k \rangle$$

La lettre p (path) représente un chemin avec une suite ordonnée de nœuds commençant du nœud v_0 jusqu'au nœud v_k qui est le dernier de la séquence de nœuds.

2.1.2 Calcul du coût d'un chemin :

Le coût d'un chemin est la somme du coût de chaque arête qui le constitue et se calcule avec une fonction appelée ***weights*** qui est définie mathématiquement comme suit :

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \qquad (2.1)$$

La fonction $w(p)$ est une fonction qui permet de retourner la somme des coûts de l'ensemble des arêtes qui constitue un chemin p. Cette formule prend l'ensemble des coûts des arêtes récupérés entre deux nœuds successifs v_{i-1} et v_i, en commençant à 1 pour que v_{i-1} débute bien à 0 jusqu'au dernier nœud v_k de la séquence constituant le chemin.

2.1.3 Notion de plus court chemin :

Dans un graphe, le plus court chemin est un chemin entre deux nœuds u, v qui parmi l'ensemble des chemins qui existent, a comme coût total renvoyé par la fonction ***weights***, la valeur la plus faible.

Grâce à cette fonction qui permet de récupérer le poids total d'un chemin, le plus court chemin peut être défini de la façon suivante :

$$\delta(u,v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\rightsquigarrow} v\right\} & \text{Dans le cas ou un chemin existe entre u et v,} \\ \infty & \textit{dans les autres cas.} \end{cases} \qquad (2.2)$$

Ceci exprime le fait que si un chemin p existe entre deux nœuds u et v (symbolisé par $\overset{p}{\rightsquigarrow}$), alors la fonction $\delta(u, v)$ donnera, parmi l'ensemble des chemins qui existent, celui ayant la valeur renvoyée par la fonction ***weights*** la plus faible. Sinon dans le cas contraire cette fonction renverra une valeur infinie (ou très grande).

3 Benchmark

3.1 Objectifs :

Avant de de nous lancer dans l'élaboration de l'algorithme, il était nécessaire de réfléchir à un moyen d'évaluer l'efficacité de celui-ci. L'objectif du benchmark était de permettre dans un premier temps, de servir de référence en nous donnant la possibilité d'avoir une idée des performances de notre travail vis-à-vis des autres technologies.

Ensuite, le benchmark servirait aussi de point de repère lors des modifications du travail permettant de comparer la version modifiée avec la précédente et ainsi d'évaluer les bénéfices apportés.

3.2 Description des technologie étudiées :

Dans ce benchmark, les technologies étudiées sont itinero[6] et pgrouting[7]. Ce sont ces technologies qui ont été sélectionnées car elles sont parmi les plus performantes du marché dans ce domaine. En les prenant comme référence nous nous assurions ainsi que notre comparaison avait un sens et que ce que nous allions développer serait performant s'il égalait ou dépassait les performances de ces technologies.

La première technologie est itinero, qui est une librairie utilisable en dotnet permettant notamment de faire du calcul d'itinéraires entre deux coordonnées géographiques à partir de longitude/latitude.

La seconde est une extension à intégrer à une base de données postgreSQL qui est utilisée pour le calcul de plus court chemin sur un graphe. Elle propose l'usage de plusieurs algorithmes (notamment un Dijkstra qui a été utile pour évaluer l'efficacité de notre implémentation) qui peuvent travailler sur des latitudes/longitudes ou via certains types de données internes (comme des ID de nœuds ou d'arêtes) créées à partir d'une carte OSM[8].

3.3 Analyse des critères d'évaluation du benchmark :

Dans le but d'évaluer la performance de notre solution pour résoudre la problématique, il est nécessaire de la comparer aux solutions existantes sur base des critères suivants :

1. Le nombre d'itinéraires calculés est divisés par palier. La division par plusieurs paliers est faite afin de simplifier l'exécution du benchmark en pratique tout en ayant une bonne vision d'ensemble et avec un nombre assez grand d'itinéraires. Ces paliers sont les suivants 50, 100, 250, 500, 1000, 2500, 5000.

[6] http://www.itinero.tech/
[7] https://pgrouting.org/
[8] Le format de fichier OSM est un format très utilisé pour représenter des cartes et permet de rendre les données de ces cartes facilement exploitables/exportables dans divers domaines et applications. Des informations sur ce format sont disponibles à cette adresse : https://docs.fileformat.com/gis/osm/

2. Le temps total en seconde mis pour calculer ces itinéraires est le critère principal pour mesurer l'efficacité de ces technologies.

Ces critères nous permettent de comparer le temps de calcul de l'algorithme et donc son efficacité par rapport aux autres technologies.

3.4 Résultats et conclusions :

La figure suivante illustre les résultats obtenus après avoir effectué le benchmark des différentes technologies. Ensuite la méthodologie employée sera détaillée avec une explication du graphique et elle sera suivie de la conclusion.

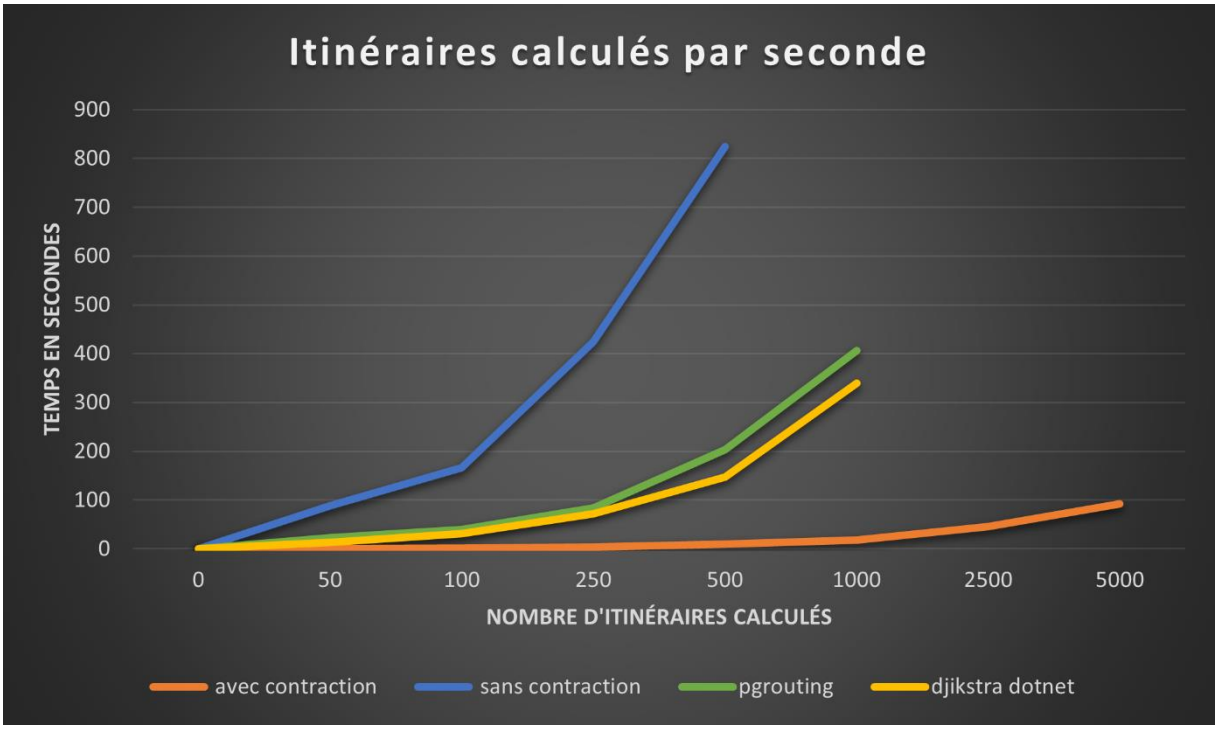


Figure 1 : Premier benchmark, effectué juste après l'implémentation de notre algorithme de Dijkstra.

Sur l'axe des abscisses est regroupé sous forme de paliers (50, 100…, comme expliqué dans la section 3.3 sur l'analyse des critères) le nombre de calculs d'itinéraires réalisés par chacune des technologies. Concrètement cela signifie par exemple que pour le palier à 50 itérations, un timer a été démarré avant le premier appel de l'algorithme, et qu'après cinquante appels d'affilée de celui-ci, le timer a été arrêté. Cette opération a été réalisée à l'identique pour tous les paliers.

Sur l'axe des ordonnées se trouve le temps mis par l'algorithme pour effectuer les calculs d'itinéraires pour chacun des paliers. En jaune on trouve la courbe de notre implémentation de l'algorithme de Dijkstra réalisé en dotnet, en vert on trouve l'algorithme de Dijkstra du module pgrouting, tandis que la courbe en bleu est la libraire dotnet itinero en version non contractée[9]. Enfin la dernière courbe en orange est la version contractée du graphe avec itinero.

[9] La contraction sera étudiée à la section 7.1.

Ce qui apparait très clairement dans un premier temps, est que, pour tous les outils utilisés, l'évolution du temps pour le calcul d'itinéraires est globalement linéaire par rapport au nombre d'itinéraires calculés. Une information importante à avoir à l'esprit lors de l'analyse de ce graphique est que cette linéarité ne se voit pas à cause d'une échelle spécifique. Ensuite, on peut constater que la version contracté/normal du calcul d'itinéraires d'itinero est bien d'un facteur x 100 et que notre implémentation de l'algorithme de Dijkstra se trouve être plus rapide que les autres technologies à l'exception de la version contracté d'itinero.

Le benchmark nous permet de voir quelle technologie est la plus performante en termes de temps de calcul et de savoir où se situe la nôtre par rapport aux autres. À partir de là, l'objectif est d'appliquer des optimisations pour réduire le plus possible le temps de calcul d'itinéraires (et donc de descendre la courbe de notre implémentation en jaune) afin de se rapprocher le plus possible de la meilleure technologie qui est représentée par la courbe orange.

4 Algorithme de Dijkstra :

L'algorithme de Dijkstra est un algorithme appartenant à la famille des algorithmes voraces (ou appelé aussi greedy) et a pour objectif de calculer le plus court chemin d'un nœud dans un graphe vers tous les autres nœuds ou entre un nœud de départ et un nœud d'arrivée.

Concernant la réalisation de l'algorithme de Dijkstra, avant d'énoncer les spécificités et les problèmes rencontrés lors de notre implémentation ainsi que les solutions qui y ont été apportées, son fonctionnement doit être détaillé avant d'aborder la suite pour mieux comprendre les étapes suivies par l'algorithme.

La section 4.1 décrit l'idée de l'algorithme général de manière théorique pour permettre de comprendre son fonctionnement par la suite dans le projet. La section 4.2 détaillera comment cet algorithme a été implémenté dans notre cas avec les spécificités liées à notre projet car certains détails concernant sa réalisation changent à cause d'un contexte de travail différent. La section 4.2.4 va appliquer la description théorique du chapitre précédent sur un exemple concret pour illustrer les étapes par lesquelles l'algorithme passe pour arriver au bon résultat.

Dans les explications qui seront exposées, la structure de données utilisée pour le stockage des nœuds est une FAP minimal, mais pour plus de simplicité elle ne sera pas développée dans cette section mais dans la section 5.2. La seule information pertinente pour la compréhension de l'algorithme est qu'il prend l'élément ayant la valeur minimale dans la structure de stockage des données.

4.1 Implémentation de manière générale :

L'algorithme de Dijkstra utilise deux fonctions principales qui seront décrites en premier lieu. Comme elles travaillent principalement sur des nœuds, il est nécessaire de définir certaines caractéristiques indispensables que celles-ci doivent posséder pour permettre par la suite à l'algorithme de fonctionner.

Un élément important à prendre en compte concernant le fonctionnement de l'algorithme concerne le fait que les arêtes qui constituent le graphe doivent toutes avoir des valeurs positives, dans le cas contraire l'algorithme se bloquera dans une boucle infinie.

Dans ce cadre théorique, un nœud possède au moins deux attributs, l'attribut d qui est la valeur du coût du plus court chemin pour arriver jusqu'à ce nœud, et pi qui est le nœud précédant dans le chemin. L'attribut d permet d'avoir le coût du plus court chemin total et l'attribut pi permet de remonter la succession de nœuds en partant du dernier jusqu'au premier afin de connaître les nœuds qui constituent ce chemin.

4.1.1 Fonction Initialize-Single-Source :

La première fonction est la fonction $Initialize - Single - Source$ qui permet d'initialiser certaines valeurs et attributs des éléments qui composent le graphe G à partir d'un nœud s fourni en entrée:

```
Initialize-Single-Source(G, s)
    for each vertex v in G.V
        v.d = infinity
        v.pi = nil
    s.d = 0
```

Figure 2 : Pseudo code de la fonction Initialize-Single-Source (1), p. 648.

Lorsque la fonction est appelée sur le nœud s, elle va appliquer sur chacun de ses nœuds sortants les modifications suivantes :

- Fixer à la valeur infini[10] le coût du chemin pour arriver au nœud v, soit $\forall v \in V, v.d = \infty$.
- Affecter la valeur du nœud précédent à nul car il n'existe pas au départ, soit $\forall v \in V, v.pi = nil$.
- Une fois sorti de la boucle for, le coût du nœud source est mis à 0 car étant le premier nœud sur lequel l'algorithme doit commencer, le chemin pour arriver jusqu'à lui est forcément nul.

Une fois la fonction terminée, le graphe sur lequel elle aura effectué ses opérations sera normalisé, ce qui permettra à l'algorithme de s'effectuer correctement et de fonctionner comme prévu de manière théorique.

4.1.2 Fonction Relax :

Cette fonction à la signature suivante $Relax(u, v, w)$ où u, v sont des nœuds entre lesquels il faut trouver le coût et w la fonction permettant de calculer ce coût. Elle permet de mettre à jour le chemin trouvé au niveau du coût actuel total pour arriver jusqu'au nœud courant :

```
Relax(u, v, w)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.pi = u
```

Figure 3 : Pseudo code de la fonction Relax (1), p. 649.

Le rôle de ces étapes est le suivant :

- La comparaison permet de vérifier si le coût du meilleur chemin pour arriver au nœud sortant v est supérieur à celui pour y arriver à partir du nœud courant u. Dans le cas où la comparaison est vraie, alors on a trouvé un meilleur chemin puisque le nouveau coût pour y arriver est inférieur à celui pour y arriver jusqu'à cette itération.
- Si la comparaison est fausse alors il ne se passe rien.
- Si la comparaison est vraie, la première ligne après la comparaison actualise la valeur du plus court chemin du nœud sortant v à la valeur du nœud courant u plus le coût de l'arête qui les relie.

[10] En pratique, c'est une valeur très grande qui est affectée.

- La dernière ligne est l'actualisation du nœud précédent faisant partie du plus court chemin. Dans le cas d'une comparaison vérifiée comme vrai, on actualise le nœud précédent constituant le plus court chemin du nœud v au nœud courant u.

Cette fonction a donc comme objectif de comparer un ancien et un nouveau plus court chemin sur chaque nœud et de les mettre à jour lorsqu'un chemin plus court est trouvé. Une fois ces opérations terminées, cela permet dans un premier temps d'avoir pour chaque nœud un coût représentant le chemin optimal pour les atteindre à partir d'un nœud de départ. Et deuxièmement, cela permet d'avoir pour chacun d'eux un lien vers un nœud précédant qui, placé les uns à suite des autres, ramène vers le nœud initial et donne de cette façon la séquence de nœuds qui constitue ce plus court chemin.

4.1.3 Fonctionnement de l'algorithme :

En ce qui concerne l'objectif de l'algorithme, il est de définir le plus court chemin à partir d'un nœud source pour chacun des nœuds qui se trouve dans un graphe G. Sa signature utilise les mêmes objets/fonctions que dans les sections précédentes :

```
Dijkstra(G, w, s)
    Initialize-Single-Source(G, s)
    S = []
    Q = G.V
    while Q =/= []
        u = Extract-Min(Q)
        S = S.append(u)
        for each vertex v in G.adj[u]
            Relax(u, v, w)
```

Figure 4 : Pseudo code de l'algorithme de Dijkstra (1), p. 658.

Voici l'explication des étapes de l'algorithme :

- En premier lieu l'appel de la fonction $Initialize - Single - Source$ sert à préparer le graphe comme expliqué dans la section 4.1.1 afin de permettre à l'algorithme de s'effectuer correctement.
- Les deux lignes suivantes initialisent les variables S, Q pour permettre leurs utilisations par la suite. La variable S contient l'ensemble des nœuds visités qui seront pris tour à tour au moment où leur coût est minimum, ce qui signifie qu'au moment où ces nœuds sont extraits, leur plus court chemin a été trouvé puisque l'algorithme prend toujours le nœud ayant le coût le plus faible et les nœuds sont visités suivant cet ordre. La variable Q contient l'ensemble des nœuds à visiter par l'algorithme.
- La boucle $while$ a pour but d'assurer que l'ensemble des nœuds appartenant au graphe soient visités. Elle continue tant que la variable Q qui contient le nombre de nœuds qui doivent être visités n'est pas vide.
- Une fois dans la boucle, la variable u récupère le nœud minimum contenu dans Q et ce nœud est ensuite ajouté dans S car il est visité et a donc son coût minimal trouvé (zéro dans le cas où il est le premier).

- Ensuite la boucle for a pour but de récupérer à partir des arêtes du nœud u, l'ensemble de ses nœuds sortants à partir d'un array[11] adj.
- Finalement l'algorithme appel la fonction $Relax$ afin d'effectuer la mise à jour du coût ainsi que des nœuds prédécesseurs du nœud u afin de trouver la valeur δ comme expliqué dans la section 4.1.2.

Une fois l'algorithme fini, l'ensemble des nœuds auront leur coût d'arrivé affecté à la valeur la plus faible possible pour les atteindre, ainsi que le nœud prédécesseur qui y mène. Cela permet d'avoir l'ensemble des plus courts chemins du graphe à partir d'un nœud source vers n'importe quel autre nœud.

4.2 Implémentation dans notre projet :

Dans cette section ce sera l'implémentation que nous avons réalisée qui sera décrite et elle abordera les spécificités et les problèmes liés à notre projet ainsi que la résolution des difficultés rencontrées. Un contexte de travail de l'algorithme important à prendre en compte concerne le fait qu'il sera appelé plusieurs fois d'affilée pour effectuer des calculs du plus court chemins et donc certaines variables et structure de données qu'il utilise devront être réinitialisées. Ci-dessous se trouvent son implémentation et la signature de la fonction de l'algorithme comme suit :

$Dijkstra(source,\ target)$ Où l'entrée $source$ est le nœud source de départ d'où l'algorithme débute et $target$ le nœud de destination, qui est le nœud de destination qu'on cherche à atteindre. Le but sera donc de trouver le chemin avec le coût le plus faible entre ses deux nœuds, soit $\delta(source, target)$ dans un délai qui soit minimal.

[11] Une manière de représenter un graphe est une liste de liste où chaque emplacement de la première liste est un nœud et le second emplacement est une liste faisant un lien vers ses nœuds sortant. Une explication plus détaillée de l'implémentation de notre graphe sera abordée dans la section 6.1.

```
1  Dijkstra(source, target) // Noeud de départ et de fin.
2      lv++                 // LastVisit soit nombre
3                           //   représentant le numéro passage.
4      pq = []              // pq -> priority queue ou FAP.
5      pq.AddNode(0, t)     // Ajout cu coût à 0 et du state t.
6      while pq.count =/= []
7          h = pq.Extract-Min()  // Extraction du head.
8          n = GetNode(h)        // Obtention du noeud "n".
9          if n == target
10             return <h.c, h.t> // clé-valeur du coût + state.
11         if n.lv == lv
12             continue
13
14         n.lv == lv // Actualisation de la dernière
15                    //   visite du noeud.
16
17         for each edge in NextEdges(n)
18             AddNode(C, t) // Coût du chemin "C" pour
19                           //    arriver au noeud n et
20                           //    nouvel état "t".
```

Figure 5 : Pseudo code de notre algorithme de Dijkstra.

Les grandes étapes de l'algorithme vont être expliquées ci-dessous afin de comprendre ce qu'il réalise et comment il le réalise. Certaines sections plus compliquées comme la gestion des nœuds déjà visités ou la gestion de la séquence de nœuds qui constitue un chemin seront détaillées à part dans la section 4.2.1 et 4.2.2.

Comme l'algorithme a besoin d'évaluer le coût des arêtes et que dans notre cas nous avons besoin de prendre certaines informations supplémentaires, cette partie sera aussi précisée à part dans la section 4.2.3.

- La ligne 2 consiste en l'incrémentation de la variable $lastVisit$ qui sera décrite à la section 4.2.1.
- À la ligne 4 est réinitialisé la FAP pour s'assurer qu'aucun nœud d'un FAP minimal précédent ne se trouve encore dedans.
- La ligne 5 a le même rôle que dans l'implémentation générale, on ajoute le nœud de départ avec un coût de zéro dans la FAP avec un objet $State$ qui est une structure dont l'objectif sera décrit dans la section 4.2.2.
- La boucle $while$ qui permet d'itérer jusqu'à trouver le meilleur chemin comporte plusieurs fins d'exécution qui seront expliquées le moment venu. La première rencontrée ici est l'absence de nœuds à extraire de la FAP. Comme l'algorithme part du nœud source fourni en entrée, c'est à partir de celui-ci et par la succession de nœuds sortant que le chemin optimal pourra être défini. Logiquement si la FAP est vide, c'est que soit le nœud source n'a pas de nœud sortant, soit que tous les nœuds qui ont été visités à la suite de ce nœud source ne donne pas de nœuds sortant vers le nœud final, ce qui nous amène à conclure que la destination n'a pas de chemin pour être atteinte.

- À la ligne 7 est extrait le nœud avec la valeur la plus faible de la FAP et à la ligne suivante est extrait le nœud correspondant à cet emplacement de la structure de données.
- La ligne 9-10 est la condition principale d'arrêt de la boucle, elle survient lorsque le nœud prélevé de la tête de la FAP est le même que celui que l'on cherche à atteindre et qui est fourni en entrée. Lorsque ces deux nœuds sont les mêmes, alors cela signifie que nous avons trouvé le plus court chemin et dans ce cas on peut retourner une paire de clé-valeur avec le coût $\delta(source, target)$ ainsi que le $State$ permettant de retrouver la séquence de nœuds qui constitue le chemin.
- Aux lignes 11-12 la structure de contrôle vérifie que la propriété $lastVisit$ du nœud extrait n'est pas égale à celle de la variable initialisée à la ligne 2 car si tel est le cas, cela signifie que le nœud a déjà été visité et qu'il est inutile de continuer sur ce nœud, il ne reste alors qu'à passer au nœud suivant. Cette partie sera expliquée plus en détails dans la section suivante.
- Une fois que les structures de contrôle ont vérifié d'une part la validité du nœud pris et d'autre part que l'algorithme doit continuer, alors, à la ligne 14, la propriété $lastVisit$ du nœud courant est actualisée à la valeur de l'itération actuelle. Cela permet de marquer le nœud et de ne plus le reprendre une deuxième fois. Le raisonnement est celui-ci : si nous sommes arrivés à ce nœud en prenant le chemin le plus court (certifié par le fait qu'il est en tête de la FAP) et qu'il est pris comme nœud courant, alors il n'y a pas de chemin plus court pour arriver jusqu'à lui et il est donc inutile de repasser par lui.
- Les deux dernières lignes concernent la boucle for qui va prendre chaque arête du nœud courant et ajouter ses nœuds sortants dans la FAP à travers un $State$ et à un emplacement déterminé par un coût défini par une fonction C.

Dans notre implémentation de l'algorithme, à la différence de l'implémentation traditionnelle, l'algorithme va continuer tant que la FAP n'est pas vide et à la fin de son travail, un seul chemin et coût $\delta(source, target)$ sera trouvé, celui entre le nœud source et le nœud de destination. Cela permet un grand gain de temps par rapport à un algorithme qui devrait trouver l'entièreté des chemins possibles sur un graphe contenant des millions de nœuds. De plus ce calcul serait inutile car notre l'algorithme est censé être appliqué à un graphe changeant et une fois ce calcul fait, aux moindres changements, il faudrait tout refaire.

4.2.1 Gestion des nœuds déjà visités :

Pour éviter d'être coincé dans une boucle infinie consistant à visiter consécutivement les mêmes nœuds ou simplement pour éviter de perdre du temps à traverser des chemins et des nœuds déjà parcourus, donc moins optimal que d'autre, il est nécessaire dans l'algorithme de s'assurer qu'un nœud qui a déjà été visité soit identifié comme tel et ignoré.

Pour s'assurer que cette assertion soit vraie, nous avons créé une variable $lastVisit$[12] (initialisé à la ligne 2 de la Figure 5) qui est un entier s'incrémentant d'une unité à chaque

[12] Pour éviter une confusion, le $lastVisit$ de l'algorithme sera qualifié préalablement de variable car il n'est lié à rien, et pour les nœuds, il sera qualifié de propriété car il propre au nœud depuis lequel il est appelé.

fois qu'un calcul du plus court chemin est effectué, soit à chaque fois que l'algorithme de Dijkstra est appelé. Parallèlement, les nœuds qui constituent le graphe ont une propriété du même type qui est mise à jour lorsque le nœud est visité.

Avec ces deux caractéristiques, nous avons comme première assurance qu'un nœud ne sera jamais visité plusieurs fois inutilement et cela sans casser l'intégrité du graphe, tout en permettant à l'algorithme de fonctionner correctement malgré ce changement. La seconde assurance que nous avons concerne le temps d'accès de cette information. En effet nous aurions pu stocker les nœuds déjà visités dans une liste, mais avec une telle solution, la vérification de l'appartenance ou non d'un nœud à cette liste se ferait en $O(n)$ alors que, grâce à une gestion des nœuds déjà visités gérée par les nœuds eux-mêmes, il est possible via cette propriété d'y accéder en une complexité en $O(1)$.

Par exemple, imaginons que dans la Figure 7 on veuille appeler trois fois l'algorithme à la suite pour trouver des plus courts chemins, le premier de D vers B, D vers E et de G vers C. Après chaque appel de l'algorithme, celui-ci va voir la valeur de sa variable $lastVisit$ s'incrémenter. Et dans cet appel, après qu'un nœud ait été visité, cette propriété $lastVisit$ va être mise à jour à la valeur de la variable de l'algorithme, permettant de savoir si ces nœuds ont déjà été visités dans cette appel.

Dans le tableau suivant, la première colonne indique le numéro correspondant à l'appel l'algorithme pour trouver un plus court chemin entre deux nœuds fournis tel qu'annoncé dans le paragraphe précédent. Au moment de cet appel, la valeur de la variable $lastVisit$ de l'algorithme est incrémentée de 1 par rapport à l'appel précédent (initialisé à 0).

La seconde colonne montre l'ensemble des nœuds qui seront visités ainsi que la valeur de leur propriété $lastVisit$ à la fin de l'appel de l'algorithme. Elle permet donc de comprendre comment ce processus s'effectue. Conformément à la description du fonctionnement de l'algorithme fait à la section précédente, seuls les nœuds qui seront pris de la FAP seront marqués, pas les nœuds qui y sont ajoutés en tant que nœud sortant.

Appel de l'algorithme	Nœuds ainsi que leur propriété $lastVisit$
$\mathbf{1 - D \rightarrow B}$	{D : 1, C : 1, A : 1, F : 0, G : 0, B : 1, E : 0, H : 0}
$\mathbf{2 - D \rightarrow E}$	{D : 2, C : 2, A : 2, F : 0, G : 2, B : 2, E : 2, H : 0}
$\mathbf{3 - G \rightarrow C}$	{D : 3, C : 3, A : 2, F : 0, G : 3, B : 2, E : 3, H : 0}

Tableau 1 : Actualisation des propriétés lastVisit en fonction des appels de l'algorithme avec en rouge les valeurs ayant changées à la suite de la prise d'un nœud de la liste des nœuds disponibles.

Dans l'appel numéro 1, la variable $lastVisit$ de l'algorithme est incrémentée pour la première fois et est donc égale à 1. Tous les nœuds qui seront alors pris pour trouver le plus court chemin auront leur propriété $lastVisit$ mise à 1 aussi. Comme la valeur $lastVisit$ de la variable de l'algorithme est invariante durant le calcul, tous les nœuds pris durant cet appel ayant une valeur de 1 seront ignorés, conformément à l'explication fournie par rapport à la ligne 11 de la Figure 5. Cette égalité signifie que dans le calcul actuel, le nœud a déjà été visité préalablement via un autre nœud, car sa propriété $lastVisit$ a été mise à la même valeur que celle de la variable de l'algorithme, et qu'il y a donc un chemin plus court pour y accéder qui a déjà été trouvé.

Lors de l'appel numéro 2, la variable de l'algorithme est à nouveau incrémentée d'une unité, avec pour conséquence de faire passer certains nœuds de 1 à 2, lorsqu'ils avaient déjà été visités dans le calcul précédent, et de 0 à 2 lorsque ces nœuds n'ont jamais[13] été visités puisqu'une unité manque entre les deux $lastVisit$.

À la dernière itération, les derniers nœuds qui seront visités seront actualisés à la valeur 3, qui est la valeur correspondant à ce dernier appel de l'algorithme. En actualisant la propriété $lastVisit$ des nœuds, nous nous assurons ici aussi que ces nœuds ne seront plus visités une deuxième fois lors de ce calcul du plus court chemin.

4.2.2 Gestion de la séquence de nœuds du chemin :

Un des problèmes à résoudre concernant l'algorithme est d'accéder à la séquence de nœuds qui conduit, à partir du nœud source, au nœud de destination. Pour résoudre ce problème, il faut être capable de trouver parmi l'ensemble des nœuds qui permettent d'arriver au nœud final et à ces prédécesseurs, ceux qui se trouvent dans le plus court chemin.

La solution qui a été apportée a été d'encapsuler les nœuds dans des objets nommés "State" qui contiennent l'ensemble des informations permettant à la fois de calculer le coût total d'un chemin et de retrouver la succession ordonnée de nœuds qui le constitue. Cet état à comme signature $State(cost, node, previousState)$ et garde les informations principales suivantes :

1. $cost$: c'est le coût total pour arriver jusqu'à un nœud soit le coût d'un chemin pour arriver jusqu'à ce nœud. Il permet de comparer les nœuds lors de leur ajout dans la FAP et de trouver leur place correcte. C'est aussi cette valeur qui est ajoutée dans la FAP et sert à donner une priorité au prochain nœud à prendre.
2. $node$: le nœud sortant sur lequel la boucle for est en train d'itérer. Lorsque la boucle prend le nœud courant, elle récupère les différentes arêtes de celui-ci et à partir de ces arêtes qui mènent vers un nœud sortant, elle ajoute ce nœud sortant dans l'objet $State$. Grâce à cela chaque objet $State$ a un nœud avec le coût total pour arriver jusqu'à lui.
3. $previousState$: cette dernière propriété est celle qui permet de retrouver la séquence entière d'un chemin. Elle contient l'état précédent, sauf le premier qui renvoie vers la valeur $null$. De cette façon, puisque chaque état contient le nœud plus l'état précédent, il est facile de remonter la séquence de nœuds en parcourant l'arborescence d'état jusqu'à arriver à la valeur $null$ qui est le premier nœud à avoir été créé.

[13] Dans ce cadre-ci, nous pouvons déduire que les nœuds n'ont jamais été visités car il passe de 0 à une valeur positive et comme 0 est la valeur affectée lors de leur création, cela signifie qu'ils n'ont pas été visités depuis cet création. Dans le cas où la valeur est un entier différent de 0, alors cela indiquera que les nœuds n'ont pas été pris dans la FAP depuis l'appel de l'algorithme correspondant à ce nombre.

La Figure 6 montre l'idée qui a été mise en œuvre pour résoudre le problème à l'aide d'un diagramme :

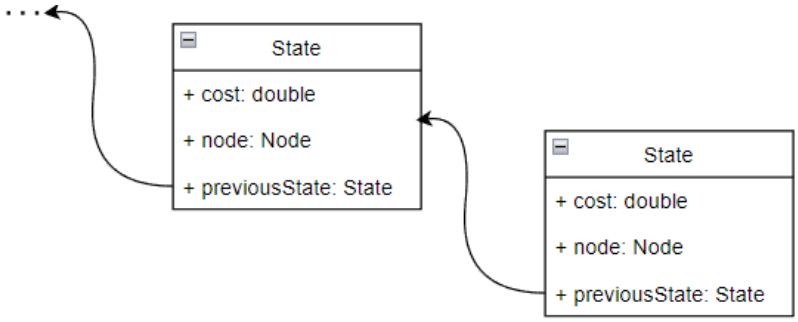


Figure 6 : Représentation sous forme de diagramme le système d'état qui permet de faire le lien de chaque état ver l'état précédent.

Chaque propriété $previousState$ permettra d'aller à l'état précédent et de récupérer via celui-ci le nœud qu'il contient, donnant ainsi la possibilité de trouver et remonter la chaîne de nœuds du plus court chemin. C'est grâce à cette méthode que dans la démonstration du fonctionnement de l'algorithme à la section 4.2.4, la suite de nœuds constituant le chemin sera trouvée.

4.2.3 Fonction coût :

Cette fonction est un élément clé de l'algorithme car c'est elle qui permet l'évaluation correcte du meilleur nœud à prendre pour poursuivre le chemin qui est en train d'être calculé.

Pour réaliser cette évaluation, elle utilise plusieurs variables qui sont nécessaires à la réalisation de ce calcul, sa signature est la suivante : $Cost(totalCostS, costToNextNode)$:

1. $totalCostS$: c'est le coût total du plus court chemin parcouru jusqu'à une itération n en partant du nœud source. Dans l'exemple de Figure 7, lorsque l'algorithme cherche un chemin entre le nœud *A* et *E*, à l'itération explorant le nœud *D* la valeur du $totalCostS$ sera de $\overbrace{0}^{A \to A} + \overbrace{1}^{A \to C} + \overbrace{3}^{C \to D} = 4$.
2. $costToNextNode$: c'est le coût de l'arête pour arriver au prochain nœud sortant à partir du nœud courant. Dans l'exemple de Figure 7, lorsque l'algorithme dans sa

recherche en est au nœud *D*, la valeur du $costToNextNode$ pour les nœuds sortant *B-G-H* sera celle des différentes arêtes $\overset{(D,B)}{\overset{\frown}{5}}$, $\overset{(D,G)}{\overset{\frown}{6}}$, $\overset{(D,H)}{\overset{\frown}{9}}$.

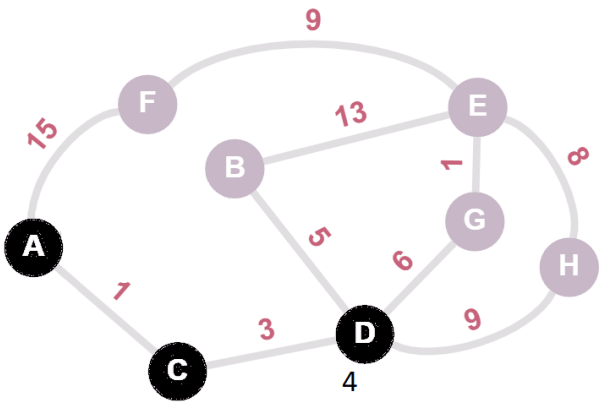


Figure 7 : Graphe avec en noir les nœuds visités qui font parties du plus court chemin.

La fonction retourne ainsi la somme de ces deux valeurs et permet d'avoir une méthode de calcul découplée de l'algorithme. En cas de modification de celle-ci, par exemple pour ajouter une heuristique, il suffira de modifier cette fonction.

4.2.4 Démonstration du fonctionnement de l'algorithme :

Cette section aborde une illustration de l'algorithme de Dijkstra. Ci-dessous est représenté à partir d'un graphe de base de la Figure 8(a) le parcours à chaque itérations successives de l'algorithme de Dijkstra. L'hypothèse de départ est que l'on cherche à trouver le plus court chemin entre le nœud *A* et le nœud *E*. Avant de rentrer dans l'explication de son fonctionnement, voici à quoi correspondent les couleurs utilisées sur le graphe ci-dessous Figure 8 :

1. Rouge : nœuds et arêtes entre les nœuds ayant déjà été visités.
2. Orange : nœud sortant atteignable à partir du nœud courant.
3. Brun : arêtes entre le nœud courant et les nœuds sortants.

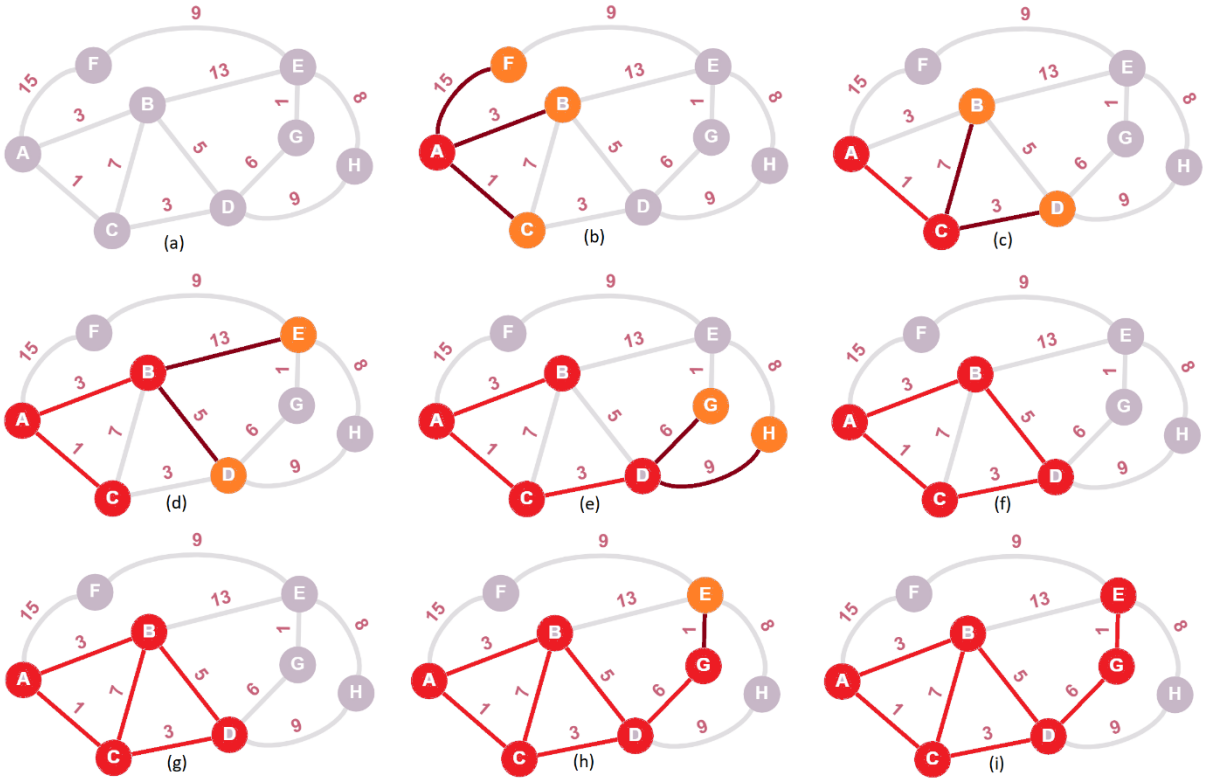


Figure 8 : Représentation de l'ensemble des états du graphe de départ en (a) après chaque itération de l'algorithme. Aux états (b)-(c)-(d)-(e)-(h), nous voyons que l'algorithme prend à partir d'un nœud courant ses nœuds sortant en orange pour les ajouter dans la FAP. Dans les états (f)-(g), il parcourt les nœuds mais ne les prend pas en compte car ils sont déjà visités. L'état final en (i) est l'itération où l'algorithme trouve le nœud fourni au départ dont on doit trouver le plus court chemin.

La Figure 8 est expliquée ci-dessous via une suite de tableau qui, à chaque instant, indiqueront dans l'ordre qui suit, l'index de l'étape avec sa lettre équivalente dans la figure, les nœuds déjà visités, les nœuds qui ont été ajoutés dans la file ainsi que le coût total pour arriver à chacun d'eux. Pour des raisons de simplicité, les tableaux ne représentent pas une FAP et les nœuds ne seront donc pas stocker par rapport à une règle, seul le nœud ayant le coût de chemin le plus faible sera mis en premier lieu pour faciliter la compréhension de son fonctionnement à chaque itération.

Étape	Partie	Nœuds déjà visités	Nœuds de la file
1	*(b)*	{}	{A : 0}

Cette étape correspond au point *(b)* où le seul nœud dans la file est celui indiqué au départ et à partir duquel il faut trouver un chemin. Il est donc mis à 0 et de là est pris ses nœuds sortants. Les trois nœuds sortants sont ajoutés avec comme coût la somme du coût pour arriver jusque-là (0 car *A* était le nœud de départ) et le coût de l'arête pour y arriver.

Étape	Partie	Nœuds déjà visités	Nœuds de la file
2	*(c)*	{A}	{C : 1, B : 3, F : 15}

Une fois le nœud *A* récupéré, il est marqué comme étant déjà visité et ne sera plus pris afin d'éviter des boucles infinies ou une perte de temps. Le second nœud à être pris est le *C* car il est le nœud avec le coût le plus faible directement lié au nœud de départ.

Étapes	Partie	Nœuds déjà visités	Nœuds de la file
3	*(d)*	{A, C}	{B : 3, D : 4, F : 15, B : 8}

À cette étape-ci, le nœud *B* qui part directement du *A* est celui qui est pris car il est celui ayant une valeur la plus faible et en même temps il existe un autre nœud *B* avec un coût de 8 qui est celui ajouté à partir du nœud *C*. Les autres nœuds qui viennent d'être ajoutés (*B* et *D*) ont leur coût qui a été calculé suivant la même formule, la somme du chemin passé pour arriver à eux ($A \to C = 1$) plus l'arêtes entre eux et le nœud précédent ($C \to B = 7$; $C \to D = 3$). Le résultat final donne bien : $\boxed{A \to C + C \to B = 1 + 7 = 8}$ et $\boxed{A \to C + C \to D = 1 + 3 = 4}$.

Étapes	Partie	Nœuds déjà visités	Nœuds de la file
4	*(e)*	{A, C, B}	{D : 4, D : 8, F : 15, B : 8, E : 16}

À cette étape, l'algorithme continue à faire la même opération qu'aux étapes précédentes et il n'y a rien de différent qui est effectué, le nœud *D* est pris.

Étapes	Partie	Nœuds déjà visités	Nœuds de la file
5	*(f)*	{A, C, B, D}	{D : 8, G : 10, F : 15, B : 8, E : 16, H : 13}

Dans cette étape-ci, le nœud ayant un chemin avec la valeur du plus court chemin est le *D,* seulement comme il a déjà été sélectionné à l'étape précédente (étape 4, voir Figure 8*(e)*), l'algorithme va continuer sur le prochain nœud et ignorer celui-ci. La raison logique est que si ce nœud a déjà été visité avant, alors c'était via un chemin plus court puisque c'est toujours le chemin avec le coût le plus faible qui est pris. Le nouveau chemin menant jusqu'à lui est forcément plus long et donc inutile.

Étapes	Partie	Nœuds déjà visités	Nœuds de la file
6	*(g)*	{A, C, B, D}	{B : 8, G : 10, F : 15, E : 16, H : 13}

Pareil que l'étape précédente, le nœud *B* a déjà été visité et n'est donc pas pris en compte, l'algorithme passe au nœud suivant de la liste.

Étapes	Partie	Nœuds déjà visités	Nœuds de la file
7	*(h)*	{A, C, B, D}	{G : 10, F : 15, E : 16, H : 13}

Le nœud *G* est le nœud suivant de la liste avec un coût cumulé depuis le début de 10, ce qui en fait le plus faible dans la liste et il n'a pas encore été visité, il est donc le suivant à être pris en compte par l'algorithme.

Étapes	Partie	Nœuds déjà visités	Nœuds de la file
8	*(i)*	{ A, C, B, D, G}	{E : 11, F : 15, E : 16, H : 13}

Cette étape-ci est la dernière car le prochain nœud de la liste ayant le plus court chemin est le nœud *E*, qui n'a pas encore été visité et qui est le nœud de départ qui devait être atteint. Comme c'est le premier nœud *E* à avoir été sélectionné et qu'on a successivement pris les chemins avec les valeurs les plus faibles, on sait lorsqu'on le sélectionne qu'il n'y a pas de plus court chemin pour l'atteindre.

L'algorithme s'arrête donc là car nous avons trouvé la valeur $\delta(A, E)$ (où A, E sont les nœuds spécifiés en hypothèse au début du chapitre). Le chemin p résultant de $\delta(A, E)$ peut-être trouvé en remontant la suite ordonnée de nœud ayant mené au nœud *E* de cette façon : $E \leftarrow G \leftarrow D \leftarrow C \leftarrow A$, qu'on peut écrire $p = \langle A, C, D, G, E \rangle$.

5 Étude de la file d'attente prioritaire

Cette section se concentre sur l'analyse théorique de la file d'attente prioritaire car elle est un élément clé de l'algorithme de Dijkstra. En effet, dans cet algorithme il est nécessaire de maintenir un ensemble de nœuds restant à visiter, et de toujours commencer par celui de plus petit poids. Pour rendre cette étape efficace, l'utilisation d'une structure de données adaptée nous permettra d'obtenir une approche optimale. La file d'attente prioritaire est une telle structure de donnée permettant :

a) L'ajout d'éléments avec priorité.
b) D'obtenir l'élément de priorité maximale et de le supprimer de l'ensemble. Il y a de nombreuses possibilités pour implémenter une file d'attente.

Il a été prouvé (1), p. 658 que l'implémentation sur base d'un tas est efficace, c'est pour cette raison que c'est cette structure de donnée qui sera étudiée et qui a été utilisée dans le projet.

Comme introduit dans la section 1 et la section 3.1, une fois la réalisation pratique de l'algorithme faite, un des objectifs était d'améliorer au maximum sa performance en réduisant son temps de calcul. Pour arriver à l'améliorer sur cette aspect-là, la FAP est un élément crucial car c'est elle qui se charge de gérer l'accès aux nœuds qui vont être utilisés tous le long de l'algorithme pour trouver le plus court chemin comme définis à l'équation de la section 2.1.3. Cette section va donc se charger de l'explication théorique concernant cette partie du travail.

Une FAP étant implémentée à partir d'un tas, il est important d'expliquer avant ce qu'est un tas ainsi que les fonctions qu'il utilise pour détailler par la suite le fonctionnement d'une FAP et comprendre la suite du projet.

5.1 Analyse théorique d'un tas :

Un tas est une structure de données sous forme d'arbre binaire où chaque nœud (excepté le nœud racine ou head, qui est le premier en partant du haut) est l'enfant d'un nœud parent et où chaque nœud parent se divise en deux nœuds enfant :

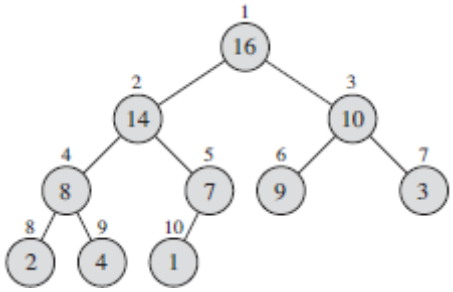


Figure 9 : Représentation graphique d'un tas sous forme arbre binaire (1), p. 152.

Dans une telle structure de données, chaque nœud est soumis à la règle suivante :

$$A[Parent(i)] \geq A[i]$$

Si la valeur du nœud racine du tas est maximal et qu'en descendant l'arbre binaire les valeurs diminues (aussi appelé ***max-heap***) et :

$$A[Parent(i)] \leq A[i]$$

Si la valeur racine est minimale et qu'en descendant les valeurs augmentent (aussi appelé ***min-heap***) (1), p. 151-153.

5.1.1 L'implémentation du tas :

L'implémentation d'un tas est très souvent faite dans un vecteur car elle permet un accès très simple et rapide à l'ensemble des nœuds et leurs voisins (enfants et parents).

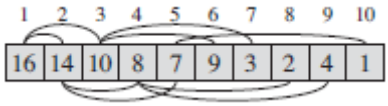


Figure 10 : Représentation de l'implémentation d'un tas de la Figure 9 dans un vecteur (1), p. 152.

Comme montré dans la Figure 10, on voit dans les emplacements du vecteur les nombres des différents nœuds correspondant au tas de Figure 9 et en haut de ces emplacements leurs ordres de rangements dans le vecteur.

Concernant les fonctions que le tas implémente, les principales seront décrites dans les prochains chapitres. La légende suivante permettra de mieux comprendre la correspondance des lettres avec ce qu'elles représentent :

- A : array contenant l'ensemble des nœuds de l'arbre, un vecteur dans notre cas.
- i : index d'un des nœuds du vecteur.
- h : hauteur de l'arbre.

5.1.2 Fonctions Parent, left-child, right-child :

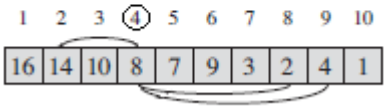
Avec la structure décrite plus haut, les trois fonctions les plus basiques sont l'accès au parent d'un nœud avec la fonction $Parent$, son enfant situé à gauche avec la fonction $Left - child$ et son enfant à droite avec $Right - child$. Chacune de ces fonctions prend un nœud i à partir duquel trouver les nœuds qui lui sont liés :

```
Parent(i)
    return ⌊i/2⌋

Left(i)
    return 2i

Right(i)
    return 2i + 1
```

(a)



(b)

Figure 11 : (a) Implémentation dans un pseudo code des fonctions d'obtention du parent, de l'enfant de gauche ainsi que celui de droite d'un nœud voulu. (b) représentation du tas dans un vecteur avec l'application des fonctions Parent, Left et Right sur le nœud 4 (1), p. 152.

Grâce à ce système, le tas permet un accès très rapide (en $O(1)$) au parent ainsi qu'aux enfants d'un nœud. Par exemple avec le point *(b)* de la Figure 11, si on prend le nœud 4 dans le vecteur contenant le tas, on voit qu'en appliquant la fonction $Parent$[14] on obtient le nombre 2 qui correspond au second emplacement du vecteur, ce qui conformément à sa représentation graphique (voir Figure 9) est bien son parent. Lorsqu'on applique la fonction $Left - child$ on obtient l'emplacement 8 et avec $Right - child$ on obtient le nœud 9, ce qui correspond bien à ses enfants gauche/droite (1), p. 152.

5.1.3 Max-Heapify :

Cette fonction a pour objectif de maintenir vraie la propriété consistant à garder chaque nœud parent comme supérieur ou inférieur à ses nœuds enfants selon le type de tas (***max-heap***, ***min-heap***). Sur un ***max-heap***, l'opération se définit de la manière suivante, $Max\_Heapify(A, i)$ où le premier paramètre est un vecteur sur lequel il faut vérifier la préservation de la relation hiérarchique entre les nœuds enfant / parents et le second est une valeur d'index à partir duquel il faut vérifier cette relation. La stratégie va donc être de partir de l'avant-dernier étage de l'arbre en allant de droite à gauche afin de vérifier tous les nœuds à travers un ensemble de sous-arbres. Ci-dessous l'implémentation de la fonction dans un pseudo langage :

```
Max-Heapify(A,i)
    l = Left(i)
    r = Right(i)
    if l <= A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r <= A.heap-size and A[r] > A[largest]
        largest = r
    if largest =/= i
        exchange A[i] with A[largest]
        Max-Heapify(A,largest)
```

Figure 12 : Pseudo code de la fonction Max-Heapify (1), p. 154.

La fonction suit les étapes suivantes :

- Premièrement, on prend les nœuds enfants gauche/droite du nœud à l'index i dans le vecteur A afin de descendre dans l'arbre et de pouvoir trouver la bonne place du nœud.
- Ensuite, on regarde si l'enfant de gauche est supérieur au nœud i et on enregistrer i comme ayant la valeur la plus grande si la condition est vraie afin de continuer avec ce nouveau nœud. Si ce n'est pas le cas, alors cela signifie que l'enfant de gauche est plus grand et que c'est lui qui doit être pris. Si jamais le nœud n'a pas d'enfant alors on est hors du vecteur et dans ce cas la première partie de la condition s'en charge en vérifiant que la valeur du nœud gauche ne dépasse pas la taille du vecteur.

[14] La valeur renvoyée comme l'indique les demi-crochets est bien la valeur arrondie à l'unité inférieure ce qui par exemple, dans le cas d'un nombre impair comme le neuf donne bien l'emplacement 4, vérifiant ainsi la formule.

- La deuxième structure conditionnelle compare le plus grand nœud trouvé précédemment à l'enfant droit du nœud i et enregistre le plus grand des deux.
- À cette étape, comme nous avons comparé le nœud i à ses enfants, nous avons l'assurance de savoir s'il est plus grand ou plus petit ainsi que l'endroit où il est nécessaire de se diriger dans l'arbre pour affecter le nœud au bon emplacement.
- Finalement on vérifie que le nœud i de départ est différent du nœud qui est le plus grand, si tel est le cas, on interchange ces deux nœuds et celui à l'index i prendra la place de son enfant qui est le plus grand. Ensuite on rappelle la fonction $Max\_Heapify(A, i)$ où i est le plus grand pour continuer la vérification sur les enfants suivant au cas où il violerait la propriété du ***max-heap*** et cette opération est faite jusqu'à trouver le bon emplacement du nœud i.

À la fin de l'appel récursif de la fonction, le nœud i aura voyagé par ses enfants et nous assure à la fin d'être inséré au bon endroit sans casser la validité de l'arbre et ce en une complexité algorithmique en $O(\lg n)$ ou $O(h)$ (1), p. 154-155.

5.1.4 Build-max-heap :

Cette fonction permet de créer un tas à partir d'un vecteur non-ordonné. Ci-dessous l'implémentation de la fonction :

```
Build-Max-Heap(A)
    A.heap-size = A.length
    for i =⌊A.length/2⌋ downto 1
        Max-Heapify(A,i)
```

Figure 13 : Pseudo code du Build-Max-Heap (1), p. 157.

- Pour cela, au départ on initialise le nombre d'éléments du tas (nœud valide inséré dans le vecteur de taille $A.length$) à la taille du vecteur pour son utilisation dans la fonction $Max - Heapify$.
- Ensuite, on appelle la fonction $Max - Heapify$ sur la première moitié des nœuds du vecteur afin de trier celui-ci et d'avoir un heap ordonné sans avoir à parcourir l'ensemble des nœuds. En partant du dernier nœud divisé par deux on prend le nœud le plus à droite ayant au moins un nœud enfants ce qui permet par la suite de parcourir tous les autres en remontant vers le nœud racine.

Cette opération se fait en une complexité de $O(n)$ et le coût total (Build-max-heap + Max-Heapify) est en $O(n.\lg n)$ (1), p. 156-157.

5.2 Analyse théorique de la FAP minimal :

Comme vu précédemment, une FAP minimal est un tas avec certaines fonctions supplémentaires et ayant certaines caractéristiques qui lui sont propre. La première est que le nœud racine est la valeur la plus petite de l'arbre. La seconde est que dans ce type de FAP, chaque nœud obéit à la règle suivante :

$$A[Parent(i)] \leq A[i]$$

Pour rappel, cette formule indique que n'importe quel nœud parent à une valeur inférieure ou égale à un nœud enfant, ce qui implique une augmentation des valeurs des nœuds en descendant dans l'arbre.

La troisième est l'implémentation des fonctions suivantes, car la FAP à priorité minimale permet quatre opérations en plus de celles d'un tas (1), p. 162 :

1. Insert (aussi appelé Enqueue) : $insert(S, x)$ insère un élément x dans un ensemble S, s'exécute en $O(\lg n)$.
2. Minimum : $Minimum(S)$ récupère la valeur minimum dans l'ensemble S, s'exécute en $O(1)$.
3. Extract-Min (aussi appelé Dequeue) : $Extract\_Min(S)$ retire et retourne l'élément avec le coût le plus faible de l'ensemble S, s'exécute en $O(\lg n)$.
4. Decrease-Key : $Decrease\_Key(S, x, k)$ réduit à la valeur k l'élément x où k ne doit pas être plus grand que x, s'exécute en $O(\lg n)$.

5.2.1 Informations sur l'implémentation de la FAP minimal :

La FAP minimal est un composant très important du projet car c'est elle qui gère les nœuds qui représente un chemin dans un graphe lors d'un calcul d'itinéraires, et les diverses opérations requises (ajouter, supprimer…) à la résolution du problème seront exécutées un grand nombre de fois puisqu'elles interviennent à chaque calcul d'itinéraires. Donc comme c'est un outil très important dans le projet, il fallait qu'elle soit optimale ce qui nécessitait qu'elle ait certaines caractéristiques.

Premièrement afin d'éviter du temps de calcul inutile, il faut que sa taille soit définissable à sa création afin d'éviter qu'elle se redimensionne une fois pleine et qu'un nœud doive être ajouté. Le problème avec cette opération de redimensionnement est qu'elle est assez chronophage et aurait été répétée beaucoup de fois lors d'un calcul d'itinéraires, ce qui peut être évité avec cette implémentation.

Deuxièmement il faut trouver le moyen le plus optimal pour stocker la FAP qui est de la gérer dans un vecteur qui représente la structure de données la plus rapide pour réaliser les fonctions dont nous avons besoin, comme cela a été vu dans le chapitre 5.1.1.

5.2.2 Fonction Insert et Decrease-key – Enqueue :

Cette fonction permet l'ajout d'un nouveau nœud dans la file avec une certaine priorité en s'assurant de l'insérer au bon endroit afin de garder la validité de la structure du tas. La signature de la fonction Enqueue est la suivante, $Enqueue(A, key)$ où A est le vecteur dans lequel ajouter la nouvelle valeur et key le nouveau nœud à placer au bon endroit :

```
Enqueue(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = infini
    Decrease-key(A, A.heap-size, key)
Decrease-key(A, i, key)
    A[i] = key
    while i > 1 and A[Parent(i)] > A[i]
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

Figure 14 : Pseudo-code de la fonction d'ajout d'un nœud Enqueue avec en haut l'ajout d'un élément dans un vecteur et en bas la fonction permettant d'amener un nœud à son bon emplacement (1), p. 164.

Les étapes de ces fonctions sont les suivantes :

- Dans $Enqueue$ comme on insère une nouvelle clé, on commence par augmenter la taille du vecteur qui devra accueillir un nouveau nœud et on lui affecte une valeur infinie (ou un nombre très grand).
- Ensuite on appelle la fonction $Decrease - key$ qui commence par affecter la nouvelle clé entrée à l'endroit du vecteur qui a été ajouté (soit à la fin de celui-ci).
- Puis, une boucle $while$ se charge d'échanger le nœud nouvellement ajouté avec son parent si celui-ci est plus petit.

Le procédé employé par ces deux fonctions est de faire remonter dans l'arbre le nœud qui vient d'être ajouté jusqu'à s'arrêter à un nœud qui lui soit inférieur, ou bien que la boucle atteigne le nœud racine en haut de l'arbre, qui sera nécessairement le plus petit. Nous avons donc l'assurance une fois l'algorithme fini que le nœud est placé aux bons endroits dans l'arbre.

5.2.3 Fonction Extract-Min – Dequeue :

Le but de cette fonction est de retirer et de renvoyer la valeur la plus petite contenue dans la FAP et donc celle d'un nœud du tas. Ce nœud se trouve toujours dans le premier emplacement du vecteur contenant le tas et se récupère instantanément. Cependant, une fois le nœud retiré, afin de garder un tas ordonné il est nécessaire de le restructurer en suivant les étapes décrites ci-dessous :

```
Dequeue(A)
    if A.heap-size < 1
        error "heap underflow"
    min = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    Max-Heapify(A,1)
    return min
```

Figure 15 : Pseudo-code de la fonction d'extraction de la valeur minimal (1), p. 163.

- La première structure de contrôle conditionnelle vérifie qu'il y a bien au moins un élément valide dans le tas à retourner, dans le cas contraire une erreur est retournée.

- Ensuite le premier nœud du tas est extrait et dans le cas d'une FAP minimal, il contient la valeur minimale. Cette valeur est affectée à une variable min qui sera retournée à la fin.
- Une fois le nœud extrait, on place le dernier élément du vecteur à l'emplacement où vient d'être pris le plus petit nœud, on réduit d'une unité le nombre de nœuds du tas car un nœud vient d'être enlevé et on appelle la fonction $Max - Heapify$ sur cet emplacement afin de faire redescendre ce nœud vers son emplacement correct car il risquerait de transgresser la condition de la FAP minimal exigeant que chaque nœud ait des enfants plus grands ou que le premier nœud soit le plus petit.
- Une fois que le nœud est déplacé à son bon emplacement, on retourne la valeur minimale qu'on avait stocké au début.

Cette opération permet donc à la fois de récupérer la valeur minimale d'un tas tout en s'assurant de garder la validité de celui-ci.

6 Changement dynamique du coût des arêtes

L'un des objectifs de ce travail a été de gérer les modifications du coût des arêtes (représentant les routes d'une ville) du graphe qui devront être effectuées lorsqu'un embouteillage, un problème sur une route ou une modification de celle-ci à des fins d'études du trafic, devra être réalisé.

En effet, comme expliqué dans l'introduction, le but est de pouvoir faire une étude et une modélisation approfondie du réseau routier qui soit dynamique, dans le sens où un évènement impactant en temps réel le réseau routier puisse être pris en compte dans le projet. Cela permet par exemple dans le cas d'un calcul d'itinéraires, de trouver celui qui est le plus efficient et à jour, et dans le cadre d'un changement appliqué via une modélisation du trafic, d'étudier l'impact qu'aurait ce changement sur celui-ci, ce qui n'est pas faisable si l'implémentation de la carte ne peut pas être changé dynamiquement.

Avant de donner plus d'explications sur la réalisation de cette fonctionnalité, il est d'abord nécessaire d'expliquer comment la carte a été implémentée dans le projet, à savoir la structure de données utilisée pour la représenter de manière "informatique".

6.1 Implémentation de la carte dans un graphe :

Pour permettre le calcul d'itinéraires sur une carte, il est nécessaire de réaliser une structure de données permettant d'accueillir et de représenter les informations relatives à cette carte (routes, coûts, nom de rue, id…) afin de permettre son utilisation dans le projet. Pour réaliser cette structure de données, les implémentations par listes et par matrices d'adjacences sont de bonnes approches (1), p. 589-592 pour représenter un graphe.

Cependant ces structures ne seront pas utilisées dans le projet car le graphe, dans notre cas, sera construit au lancement d'une application qui est censée tourner de manière continue, le temps gagné via une de ces méthodes n'est pas significatif et ne requiert pas d'implémentation spéciale, le gain de performance sera plutôt à rechercher au niveau de la partie algorithmique.

6.1.1 Diagramme de classe :

L'implémentation réalisée a été faite à partir d'un format de fichier créé pour représenter des cartes pour diverses applications, ce format est OSM (comme indiqué dans la section 3.2.). Les informations pertinentes pour notre projet que l'on trouve dans les fichiers OSM peuvent être résumées d'une part en une succession de nœuds qui représentent des points sur la carte et d'autre part à des arêtes pouvant être assimilées à des lignes qui les relient.

Les principales données qui ont été prises concernant les nœuds et les arêtes sont des données ayant trait à la géolocalisation ainsi qu'à des informations permettant de faire des calculs d'itinéraires (par ex. le coût pour atteindre un nœud à travers une arête, le nœud sortant…) ou bien des identifiants permettant de faire des liens entre les éléments de notre graphe et une carte basée sur le format OSM.

Sur base de ces données tirées d'un fichier OSM de la Belgique, le graphe est une classe regroupant dans un dictionnaire des nœuds et des arêtes qui, eux aussi sont représentés par

des classes. A cela s'ajoute diverses méthodes propres à la classe du graphe qui permettent d'accéder à diverses informations des nœuds et des arêtes (existence/ajout/récupération d'un nœud, récupération de nœuds sortant…).

Ci-dessous un diagramme de classe simplifié des différents objets ainsi que de leurs principales méthodes concernant la partie liée au graphe :

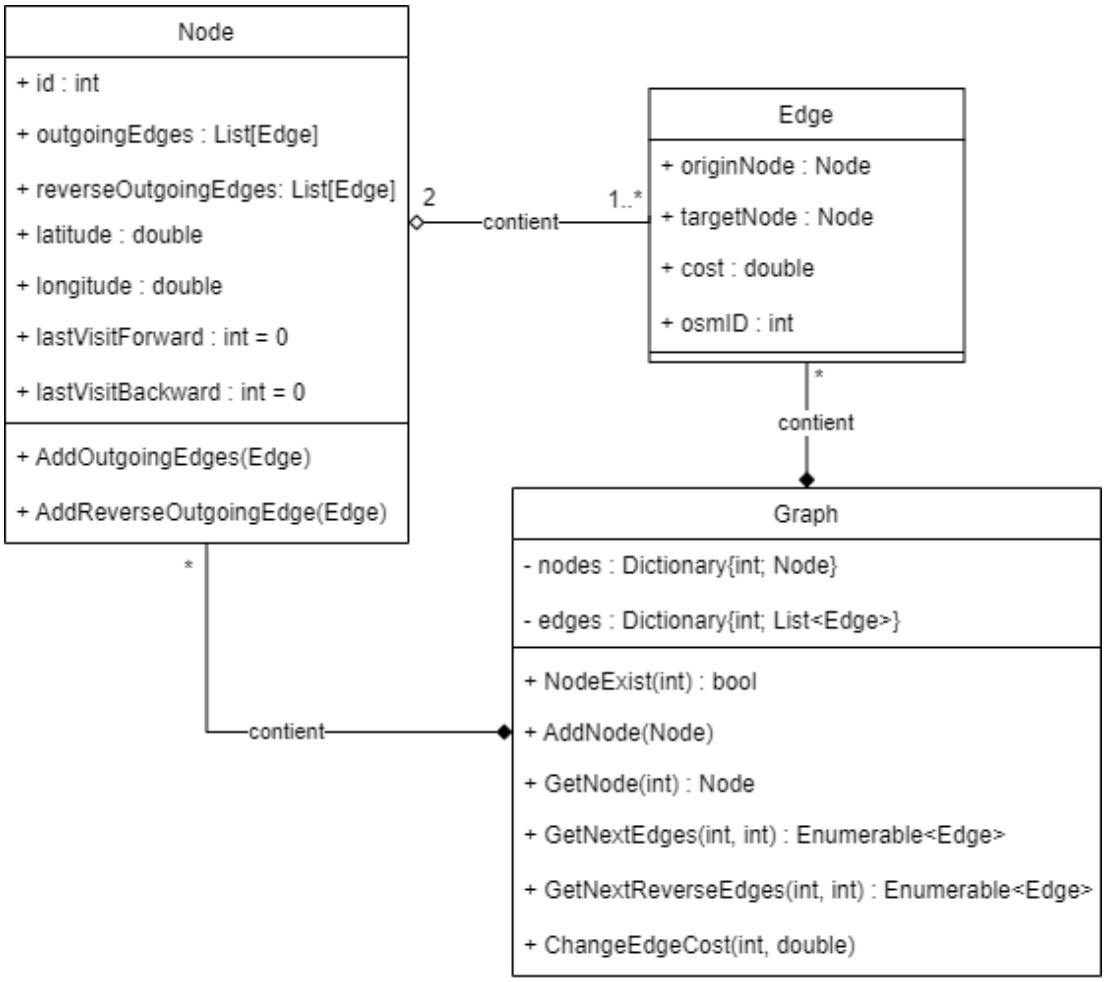


Figure 16 : Diagramme de classe simplifié du graphe ainsi que des objets qui y sont liés. L'objet "Node" correspond à un nœud du graphe et l'objet "Edge" est une arête qui sort d'un nœud et qui mène à un autre nœud.

Comme montré par le diagramme, les nœuds et les arêtes sont créés dans le graph et stockés dans un dictionnaire où ils pourront être récupérés via un identifiant tiré du format de carte OSM. Les arêtes seront aussi stockées dans les nœuds en fonction de ceux qu'elles relient ainsi que du sens dans lequel elles peuvent être parcourue (puisqu'elles représentent des arêtes dans la vrai vie).

Par exemple si une arête relie le nœud (u, v), elle sera ajoutée à la liste des arêtes sortantes ($outgoingEdges$) du nœud u et elle sera aussi ajoutée à la liste des arêtes sortantes dans le sens inverse ($reverseOutgoingEdges$) du nœud v. Dans le cas où la route qui est représentée par l'arête est bidirectionnelle, alors ce sera l'opération inverse qui sera effectuée en plus de la première. Dans ce cas, l'arête sera aussi ajoutée dans la liste des arêtes sortantes du nœud v et dans la liste des arêtes sortantes dans le sens inverse du nœud u.

La seconde information transmise par le diagramme concerne le nombre d'appartenance. Il y a un graph construit au départ qui créera et contiendra plusieurs nœuds et arêtes. Chacune des arêtes sera aussi ajoutée dans un nœud pour chaque direction[15], soit deux nœuds. Une arête est aussi liée à deux nœuds qu'elle relie (origin-target node), et chaque nœud contiendra plusieurs arêtes menant vers ses nœuds sortants.

Comme les arêtes et les nœuds sont créés dans le graphe, ils y sont reliés par une composition. Les arêtes sont en plus reliées par une association à un nœud puisqu'elles sont liées à celui-ci par l'ajout dans une liste après leur création.

Avec cette structure, pour parcourir le graphe il suffit d'appeler la méthode $GetNextEdges(int, int)$ en lui indiquant en premier paramètre l'identifiant (id) du nœud qu'on veut parcourir. Ensuite, puisque la méthode nous renvoie la liste des arêtes liées à ce nœud, il suffit d'appeler le nœud sortant (le $targetNode$ de la classe $Edge$) de l'arête pour accéder au nœud suivant de celui sur lequel on est en train d'itérer. Le deuxième paramètre de la méthode est la variable $lastVisit$ de l'algorithme qui sera expliqué dans la section suivante.

6.1.2 Explication des paramètres et méthodes des objets :

Cette section va se charger d'expliquer l'objectif des différentes propriétés et méthodes des objets en rapport avec le graphe ainsi que celles du graphe.

Classe $Node$:

- id : c'est l'identifiant qui permet de trouver un nœud et il correspond à un point dans OpenStreetMap (OSM).
- $outgoingEdges$: c'est la liste de toutes les arêtes qui sortent du nœud et qui se dirigent vers un des autres nœuds sortant (soit un des nœuds suivants).
- $outgoingEdges$: c'est la liste de toutes les arêtes qui sorte du nœud et qui se dirigent vers un des autres nœuds précédents (soit un des nœuds ayant menés jusqu'à celui-ci). Cette seconde liste d'arêtes permet de créer un graphe inverse parcourable pour la recherche bidirectionnelle15[15].
- $latitude$: latitude du nœud pour le retrouver sur une carte.
- $longitude$: longitude du nœud pour le retrouver sur une carte.
- $lastVisitForward$: c'est la variable qui permet de vérifier si le nœud a déjà été visité comme expliqué dans la section 4.2.1, et qui s'incrémentera à chaque fois que le nœud sera sorti de la FAP dans une itération de l'algorithme. Cette variable n'est utilisée que pour la recherche ***forward***[15].
- $lastVisitBackward$: c'est la variable qui permet de vérifier si le nœud a déjà été visité comme expliqué dans la section 4.2.1 mais pour la recherche ***backward***[15]***.***
- $AddOutgoingEdges(Edge)$: c'est une fonction qui permet d'ajouter une arête sortante au nœud, c'est à dire une arête menant vers un nœud sortant.

[15] Les informations sur les directions sont abordées au chapitre 7.2.1 au niveau de la recherche bidirectionnelle.

- $AddReverseOutgoingEdges(Edge)$: c'est la même fonction que $AddOutgoingEdges(Edge)$ appliquée à la création du graphe inverse pour la recherche bidirectionnel[15].

Classe $Edge$:

- $originNode$: nœud d'origine d'où l'arête part.
- $targetNode$: nœud sortant vers lequel l'arête mène.
- $costS$: coût de l'arête en seconde correspondant au coût pour parcourir la route qu'elle représente.
- $osmID$: identifiant OSM[16] permettant de faire le lien entre les arêtes de notre graphe et celles d'une carte OSM.

Classe $Graph$:

- $nodes$: c'est un dictionnaire regroupant l'entièreté des nœuds qui composent le graphe, l'entier en première valeur est l'id pour accéder au nœud et la seconde valeur est le nœud correspondant à cet id.
- $edges$: c'est un dictionnaire regroupant l'entièreté des arêtes du graphe. L'entier en première valeur est l'identifiant OSM pour accéder aux arêtes qui y correspondent et en second se trouve la liste d'arêtes qui correspondent à cet identifiant. En effet, un identifiant OSM permet parfois d'accéder à plusieurs arêtes, il est donc nécessaire de les stocker dans une liste.
- $NodeExist(int)$: méthode permettant de vérifier si un nœud existe ou pas dans le graphe à partir de son id. Elle renvoit un booléen en fonction de la présence de ce nœud ou de son absence.
- $AddNode(Node)$: ajoute le nœud fourni en paramètre dans le dictionnaire de nœuds.
- $GetNode(int)$: récupère le nœud ayant l'id fourni en entrée.
- $GetNextEdges(int, int)$: le premier paramètre est l'id du nœud à partir duquel on veut récupérer ses arêtes sortantes et le deuxième est son attribut $visitId$ qui permet de ne renvoyer ses arêtes que s'il n'a pas été visité. La logique derrière le renvoi d'une liste énumérable vide à cet endroit lorsqu'un nœud a déjà été visité est de gagner du temps. Sans vérification effectuées à cet endroit du code, des arêtes seraient retournées et il faudrait par la suite faire la vérification des valeurs $visitId$ des nœuds renvoyés, alors qu'à ce moment-là nous avons déjà l'information qui nous permettrait de savoir que le nœud ne doit pas être pris en compte.
- $GetNextReverseEdges(int, int)$: même chose que pour la fonction $GetNextEdges(int, int)$ mais pour les arêtes dans le sens inverse[15], soit celles menant vers le nœud précédant le nœud courant.
- $ChangeEdgeCost(int, double)$: cette fonction est celle permettant de changer le coût des arêtes. Elle prend comme premier paramètre l'id OSM qui est utilisé pour récupérer les arêtes qui y sont associées et comme second paramètre, la valeur de modification des arêtes qui multipliera le coût de toutes celles liées à cet identifiant.

[16] Informations liées au standard OpenStreetMap comme abordé à la section 3.2.

6.2 Implémentation du changement de coût :

Pour changer le coût d'une arête, il est nécessaire de la trouver dans le graphe par un moyen rapide à partir d'une donnée permettant de l'identifier de façon unique. C'est pour cette raison que le système permettant d'y accéder et de la trouver est le même que celui pour les nœuds. Les arêtes sont stockées dans un dictionnaire où leur identifiant OSM est placé comme clé pour y accéder en $O(1)$. Pour des raisons techniques propre au format OSM plusieurs arêtes peuvent avoir le même identifiant, pour ces raisons, toutes celles qui auront le même identifiant seront placé dans une liste attachée à cette clé.

Pour modéliser des cartes de pays, il existe un format déjà évoqué (voir à la section 3) qui est le format OSM et qui permet de modéliser dans des fichiers des cartes de pays. Ces cartes avec ces données peuvent aussi être visualisée sur un site, notamment la Belgique à cette adresse[17]. Comme les données que nous avons prises viennent d'une de ces cartes, l'identifiant des arêtes de notre graphe est identique à celui fourni par le fichier OSM. De ce fait, il permet à partir de n'importe quelle application externe qui aurait cet identifiant, de trouver dans notre graphe l'arête correspondante afin d'y apporter les modifications voulues.

Grâce à cette information faisant le lien entre, d'une part les arêtes et les nœuds de notre graphe et, d'autre part leurs équivalents dans OSM, l'objectif initial qui consiste à pouvoir modifier une arête est possible. En effet, une application externe se basant sur le format OSM pourra via cet identifiant trouver l'arête dans notre graphe et y apporter des modifications. La suite concernant cette partie du projet dépendra des applications qui auront besoin d'un outil de calcul d'itinéraires. C'est grâce à cette implémentation que les autres applications pourront modifier le graphe en changeant le coût des arêtes (qui représentent des routes dans la réalité) et faire des calculs d'itinéraire sur ce graphe venant d'être modifié.

Pour permettre la modification d'une arête dans le graphe, il a fallu ajouter une fonction qui a la signature suivante :

- $ChangeEdgeCost(int, double, double)$

Le premier entier est l'identifiant OSM, le second paramètre est la valeur du coût que l'on veut changer et le dernier est la valeur du coût en seconde que l'on veut changer. Les deux coûts viennent du fait qu'il existe un coût qui est une valeur interne à OSM et un coût en seconde qui est une estimation du temps de parcours d'une arête. À partir de là, une fois l'algorithme lancé, n'importe quelle application faisant une requête avec un identifiant OSM pourra modifier le graphe en modifiant le coût d'une arête.

[17] https://www.openstreetmap.org/#map=10/50.8424/4.5147

7 Étude d'améliorations applicables au projet

Comme mentionné dans l'introduction, dès que le développement du projet a été terminé (création du graphe, de l'algorithme de Dijkstra et du benchmark) il a fallu améliorer cette implémentation pour la rendre plus performante et se rapprocher de la meilleure des technologies, voir section 3.4.

Une des parties à améliorer était la FAP, qui a été étudiée à la section 5 et qui permet une gestion des nœuds vis-à-vis des opérations à effectuer sur celle-ci (ajout, suppression…) plus optimal qu'une simple liste. Cette section a pour but de présenter diverses autres méthodes qui peuvent être étudiées pour améliorer encore les performances de l'algorithme.

La première partie concerne l'étude de la méthode de contraction de graphe, qui est une technique applicable sur un graphe. L'objectif sera de voir comment fonctionne la méthode et s'il est possible de l'appliquer dans notre contexte. La seconde concerne le ou les moyens de minimiser le nombre de nœuds visités lors d'un calcul d'itinéraires.

7.1 Méthode de contraction hiérarchique de graphe :

Dans le cadre d'un travail sur les graphes, il existe un champ d'étude qui permet d'améliorer l'efficacité des algorithmes utilisés sur les graphes qui est labellisé sous le titre de ***contraction hiérarchique de graphe***. La contraction hiérarchique est une méthode ajoutant certaines informations sur le graphe afin d'en avoir une version contractée qui permet la réduction du temps de calcul pour trouver le plus court chemin des algorithmes s'exécutant sur celui-ci.

L'objectif de cette section sera de présenter l'idée derrière cette méthode pour comprendre en quoi elle est pertinente dans ce travail et de conclure sur la possibilité ou non possibilité de son utilisation dans le projet. Les informations récupérées pour écrire cette section peuvent être consultées à cette endroit (2) (3).

7.1.1 Explication de l'idée de cette méthode :

Le but de la contraction hiérarchique étant d'ajouter de l'information au graphe afin de le rendre plus rapidement parcourable, l'idée de cette méthode est d'introduire des arêtes ***short-cut*** dans le graphe qui créeront des raccourcis entre des nœuds et diminuerons le temps de parcours global du graphe. Ces arêtes ne seront ajoutées que si elles garantissent un raccourci du plus court chemin entre deux nœuds, c'est-à-dire qu'une arête ne sera pas créée s'il existe un autre chemin entre ces deux nœuds plus courts que le raccourci qui aurait été créé par l'arête.

Une première phase de pré-calcul se charge de contracter des séries de nœuds "à faible importance"[18] en les groupant dans des arêtes qui partiront du premier nœud de la série jusqu'au dernier de la série et auront comme coût la somme du coût des arêtes constituant les séries de nœuds. Cette étape de pré-calcul servant à réduire l'espace de recherche prend un certain temps de calcul et n'est donc pas une opération "gratuite" à réaliser, il est

[18] Ce sont des nœuds qui ont moins d'importance dans un calcul d'itinéraires, par exemple un nœud qui mène à une impasse, ou à contrario un nœud menant à une autoroute.

nécessaire de l'utiliser dans des contextes permettant, soit de tirer un gain de temps vis à vis de la contraction, soit de garder la structure contractée une fois celle-ci calculé, ce qui sera utile à avoir en mémoire dans la conclusion.

Par rapport à la Figure 17, la contraction d'un chemin entre les nœuds u_0 et u_4 sera une arête partant de u_0 et rejoignant u_4, où les nœuds allant du 1 au 3 sont enlevés et le coût de cette arête résultante est égal à la somme du coût des arêtes entre le nœud 0 et 4 :

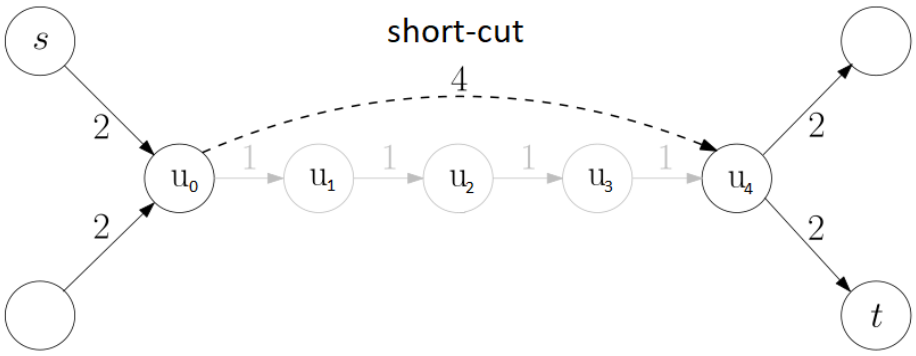


Figure 17 : Partie d'un graphe pris comme exemple pour visualiser la contraction hiérarchique (4). La contraction de la succession des nœuds de u_0 à u_4 mène à une arête rejoignant ces deux extrémités directement et avec un coût de 4 qui est bien la somme des quatre arêtes qui liaient les nœuds contractés.

On peut voir clairement que l'arête ***short-cut*** permettra d'accéder directement à la suite du graphe sans passer par des nœuds intermédiaires, raccourcissant de ce fait le temps de parcours d'un algorithme sur le graphe.

Afin de trouver quels sont les nœuds à contracter, il est nécessaire d'évaluer lesquels doivent l'être et dans quel ordre. Pour que l'opération de contraction soit correctement effectuée, il faut que le nombre d'arêtes ajoutées au graphe après cette opération soit le plus minimal possible. L'ordre dans lequel les nœuds doivent être pris pour effectuer une contraction dépend donc de l'heuristique qui sera utilisée. Par exemple, la Figure 18 montre le principe de la contraction hiérarchique selon l'ordre de prise de nœud :

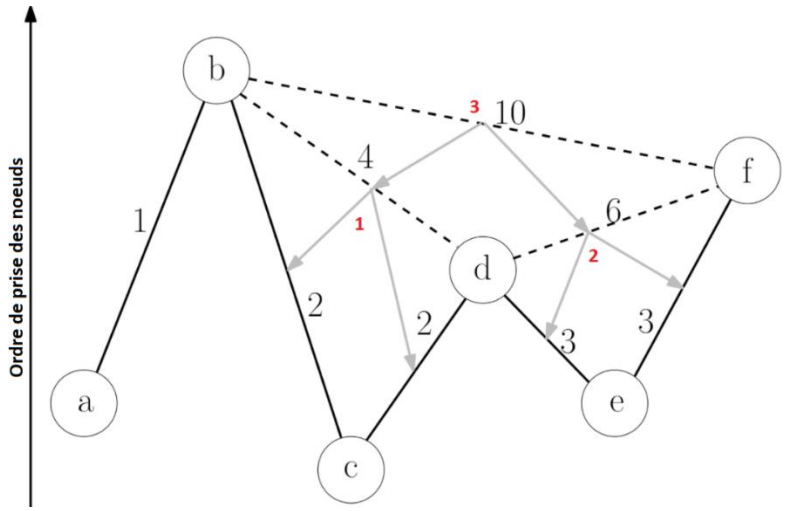


Figure 18 : Schéma représentant divers nœuds qui doivent être inspectés pour une contraction (4). Les nœuds sont alignés à un niveau en fonction de leur priorité (axe à gauche) et les numéros 1,2,3 en rouge représente l'ordre dans lequel les nœuds sont pris et donc l'ordre dans lequel leur contraction aura lieu.

Comme le montre la figure ci-dessus, le premier nœud à être contracté est le nœud c car il est celui le plus en bas. Une fois contracté, comme l'indique le numéro 1 en rouge, le résultat est une arête ***short-cut*** entre b et d de coût égal à la somme des deux arêtes qui liaient c, soit quatre.

Le second nœud est a seulement comme il n'introduit aucun chemin plus court, il n'est pas pris et l'algorithme passe au deuxième qui est e. Une fois arrivé à e, la même opération sera effectuée qu'à c et une deuxième arête ***short-cut*** sera ajoutée. Finalement à la dernière étape de cet exemple, c'est le nœud d qui sera choisi et le raccourci créé se fera entre les deux dernières arêtes ***short-cut*** créées à l'étape 1 et 2.

7.1.2 Conclusion sur la possibilité de son application :

La contraction hiérarchique est une très bonne méthode pour raccourcir le parcours d'un graphe et améliorer les performances des algorithmes opérant dessus, cependant le gros problème déjà esquisser dans le deuxième paragraphe de la section précédente est le fait que le pré-calcul pour trouver ces arêtes est long et que cela n'a de sens que dans un graphe qui ne change pas.

Or dans notre cadre de travail, puisque le graphe représente une carte avec un trafic censé évolué au fil du temps (impliquant un coût pour ses arêtes qui se modifient dynamiquement), le calcul qui nous donne un graphe modifié ne rendra ce nouveau graphe utilisable qu'entre deux changements consécutifs. Cela signifie qu'une fois la contraction du graphe effectuée, le moindre changement qui devra être appliqué sur le graphe obligera de réexécuter une nouvelle fois l'opération de contraction sur celui-ci afin de prendre en compte cette nouvelle information et d'avoir un graphe à jour. Comme la contraction est une opération couteuse en termes de temps de calcul, elle ralentirait fortement la solution de manière générale puisqu'au moindre changement sur le trafic dans le monde réel, il serait nécessaire de refaire de lourds calculs.

Le fait qu'un changement sur le trafic, qui est un évènement arrivant fréquemment, nécessite de refaire la contraction et que celle-ci soit une opération chronophage, a pour conséquence une forte perte de temps. Elle réduirait ainsi les performances de notre solution, la rendant inutilisable dans notre cas.

7.2 Moyen de minimisation des nœuds visités :

Une autre étape importante dans le projet et indiquée comme objectif dans le cahier des charges (page 5) est de trouver un moyen de minimiser le nombre de nœuds visités dans le graphe lors du parcours de celui-ci par l'algorithme.

En effet, comme le temps mis par l'algorithme pour calculer un itinéraire est directement lié au nombre de nœuds visités, réduire le nombre de nœuds visités doit permettre à celui-ci de trouver plus vite le meilleur itinéraire et donc permettre d'augmenter les performances de notre implémentation en général.

Un des moyens connus pour arriver à ce résultat est la recherche bidirectionnelle. Cette recherche consiste à commencer une seconde recherche en même temps que la première

(source -> destination) mais en partant dans le sens contraire (destination -> source). Cela permet, lorsque ces deux recherches se croisent, de réduire le temps pour trouver le meilleur chemin reliant la source et la destination. La section suivante va illustrer et expliquer son fonctionnement sur l'algorithme de Dijkstra.

7.2.1 Dijkstra bidirectionnelle :

Dans le cadre de Dijkstra, puisque son champ de recherche s'élargit en cercle, partir de la destination et faire la même opération dans le sens contraire permet de trouver le meilleur chemin à un des points de rencontre des deux cercles de recherche. Lorsque cela est fait, le nombre de nœuds visités est approximativement réduit de moitié par rapport au nombre de nœuds visités dans une recherche classique où l'on ne part que de la source. Un exemple ci-dessous entre les deux types de recherches concernant Dijkstra :

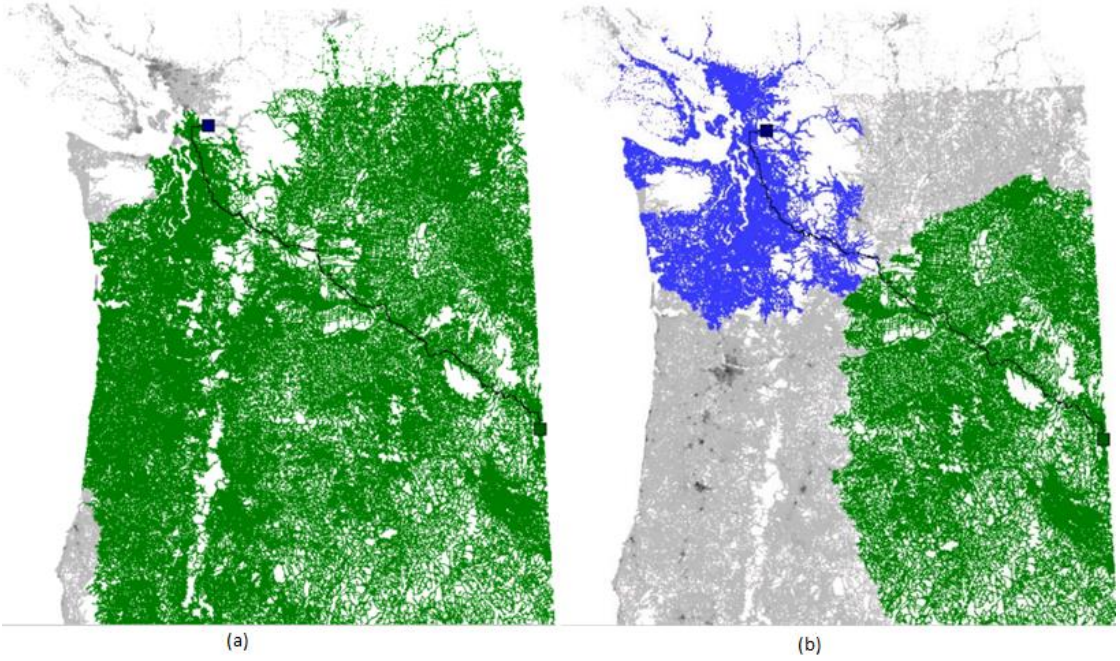


Figure 19 : L'image représente une comparaison entre deux cartes sur laquelle est lancé l'algorithme de Dijkstra classique et bidirectionnelle. En (a) se trouve l'évolution du Dijkstra classique, la couleur en vert représente les chemins visités, le carré bleu est un des deux points constituant la destination ou le départ et le carré en vert (au milieu à droite) et l'autre extrémité du chemin. En (b), toujours par rapport au même point de départ-destination, la couleur en bleu représente les chemins visités par la recherche "forward" et en vert par la recherche "backward" (5).

Ce qu'on peut voir à partir de ces deux figures est que la recherche bidirectionnelle réduit considérablement le nombre de nœuds visités (approximativement d'un facteur 2) et permet ainsi de trouver le plus court chemin plus rapidement. C'est deux résultats correspondent aux objectifs fixés puisque le premier concerne la réduction de nœuds visités et le deuxième, concerne la réduction du temps de calcul.

7.2.2 Fonctionnement de la recherche bidirectionnelle :

Avant de commencer, pour mieux comprendre la suite je préciserai ici la notion de plus court chemin total. Puisqu'il y a deux recherches (***forward*** et ***backward***), chacune d'elles donneront deux plus courts chemins correspondant à leur recherche, le plus court chemin total peut être décrit de la façon suivante :

L'union du plus court chemin appartenant aux deux recherches, qui relie le nœud source au nœud de destination à travers un chemin comme défini à la section 2.1. D'un point de vue mathématique, s'il existe sur un graphe un ou plusieurs chemins d'un nœud u à un nœud v, avec cette recherche bidirectionnelle ils peuvent être définis comme suit :

$$p_t = \langle v_0, v_1, \dots, v_k \rangle \rightarrow chemin\ total\ reliant\ u \text{ à } v.$$
$$= p_f + (v_{n-1}, v_n) + p_b$$
$$p_f = \langle v_0, v_1, \dots, v_{n-1} \rangle \rightarrow recherche \text{ forward.}$$
$$p_b = \langle v_k, v_{k-1}, \dots, v_n \rangle \rightarrow recherche \text{ backward.}$$

Où la recherche p_b est inversée car elle démarre du nœud final v_k vers le nœud reliant les deux recherches auquel il faut ajouter le coût de l'arête reliant ces deux chemins. Le poids d'un chemin dans ce cadre se trouve être :

$$w(p_t) = w(p_f) + \overbrace{w(v_{n-1}, v_n)}^{Arête\ liant\ p_f - p_b} + w(p_b)$$
$$= \sum_{i=1}^{n-1} w(v_{i-1}, v_i) + w(v_{n-1}, v_n) + \sum_{i=n+1}^{k} w(v_{i-1}, v_i)$$

Finalement le plus court chemin total sera :

$$\delta_t(u, v) = \begin{cases} \min\left\{w(p_f) + w(v_{n-1}, v_n) + w(p_b) : u \overset{p}{\rightsquigarrow} v\right\} \\ \infty \end{cases}$$

Pour trouver ce chemin final, il est nécessaire d'avoir une condition d'arrêt correcte pour s'assurer de stopper l'algorithme au moment où le plus court chemin est trouvé. Une méthode naïve serait de s'arrêter lorsqu'un nœud dans une des deux recherches a déjà été visité par l'autre. Cette méthode a été démontré comme fausse (6) et nécessite de chercher une alternative pour trouver la bonne condition d'arrêt permettant à l'algorithme de se terminer à la bonne itération.

L'idée derrière la nouvelle condition d'arrêt est de garder une variable μ qui représente le dernier plus court chemin total reliant la recherche ***forward*** et ***backward***. Comme il y a deux recherches, il y a deux FAP, une gardant les nœuds correspondant à la recherche ***forward*** et une pour la ***backward***. Lorsque les valeurs racines[19] des deux FAP sont supérieures à cette valeur μ, alors on arrête l'algorithme et on retourne les deux derniers chemins trouvés lors de cette vérification.

Cette nouvelle condition d'arrêt est la suivante (5), arrêter l'algorithme lorsque la première valeur de la somme de la recherche ***forward*** ou ***backward*** est supérieure à la dernière valeur de μ trouvée. Puisque dans le projet nous utilisons deux FAP pour chaque recherche, la condition d'arrêt s'exprime mathématiquement comme suit :

[19] Voir à la pages 6 qui explique la signification des différents termes.

$$cost\left(HFAP_{forward}\right) + cost(HFAP_{backward}) \geq \mu \rightarrow HFAP = heads.$$

L'explication de cette condition d'arrêt est la suivante.

Si :

- La valeur racine de la FAP ***forward*** et ***backward*** est le coût du chemin minimal, soit le plus court chemin total trouvé à chaque itération.
- En continuant d'explorer le graphe, les chemins ont plus de nœuds, donc plus d'arêtes et dans ce cas le coût des chemins et des valeurs racines augmentera forcément au fil de l'exploration (car il n'y a pas d'arêtes négatives), soit à chaque itération.

Alors comme conclusion 1) :

1) Les prochaines valeurs racines des deux FAP, qui représente les plus courts chemins actuels, ne feront qu'augmenter à chaque itération lorsque le graphe est exploré.

Si :

- La valeur de la variable μ est la somme des valeurs racines des deux FAP plus l'arête les reliant, elle est donc la valeur du dernier plus court chemin total trouvé entre les deux recherches.
- La variable μ n'est actualisée que lorsque la valeur du plus court chemin total donné par les deux FAP est inférieur à sa valeur actuelle.

Alors comme conclusion 2) :

2) La variable μ ne changera plus jamais si la valeur du plus court chemin total donné par les deux FAP à une itération n lui est supérieur.

Conclusion finale :

- Si à une itération n, μ est inférieur à la somme des valeurs racines des FAP, alors on a l'assurance d'avoir déjà trouvé le chemin ayant un coût plus faible, correspondant au plus court chemin total. En effet, comme tous les prochains chemins trouvés seront plus grands (conclusion 1)), que μ ne changera plus (conclusion 2)) et qu'elle est le coût du plus court chemin, alors à cette itération n le plus court chemin total a déjà été trouvé via les recherches ***forward*** et ***backward***.

La procédure à suivre pour garantir la conclusion finale et avoir une condition d'arrêt opérationnelle est la suivante :

1. Définir une variable d'arrêt μ avec comme valeur de départ $\mu = \infty$ qui contient la valeur du coût du dernier plus court chemin trouvé ainsi que deux variables contenant les ***états***[20] correspondant au dernier plus court chemin trouvé dans les deux recherches.

[20] Les états utilisés ici sont les mêmes que ceux utilisés dans la section 4.2.2.

2. Actualiser μ à chaque fois qu'un chemin total plus court est trouvé entre les deux recherches, soit que $\mu = \min\{\mu, w(p_f) + w(v_{n-1}, v_n) + w(p_b)\}$. Il faut effectuer cette vérification lorsqu'un nœud sortant de la recherche actuelle (***forward*** ou ***backward***) a déjà été visité par l'autre recherche (afin de s'assurer qu'on a bien un chemin complet) et effectuer cette opération dans chacune d'elle.
3. Actualiser des variables représentant l'***état***[20] du meilleur dernier chemin total trouvé par la recherche ***forward*** et ***backward*** afin d'avoir une référence vers celui-ci.
4. Lorsque la condition d'arrêt est validée, il faut reconstruire le meilleur chemin total sur base des deux ***états***[20] récupérés via les deux variables actualisées à l'étape 3.

Une fois ces opérations faites, l'algorithme s'arrêtera avec le meilleur chemin reliant les deux nœuds fournis en entrés.

7.2.3 Conclusion et résultats :

Cette section a pour but de montrer la différence une fois l'amélioration ajoutée sur le même graphe que celui utilisé dans la section 3.4.

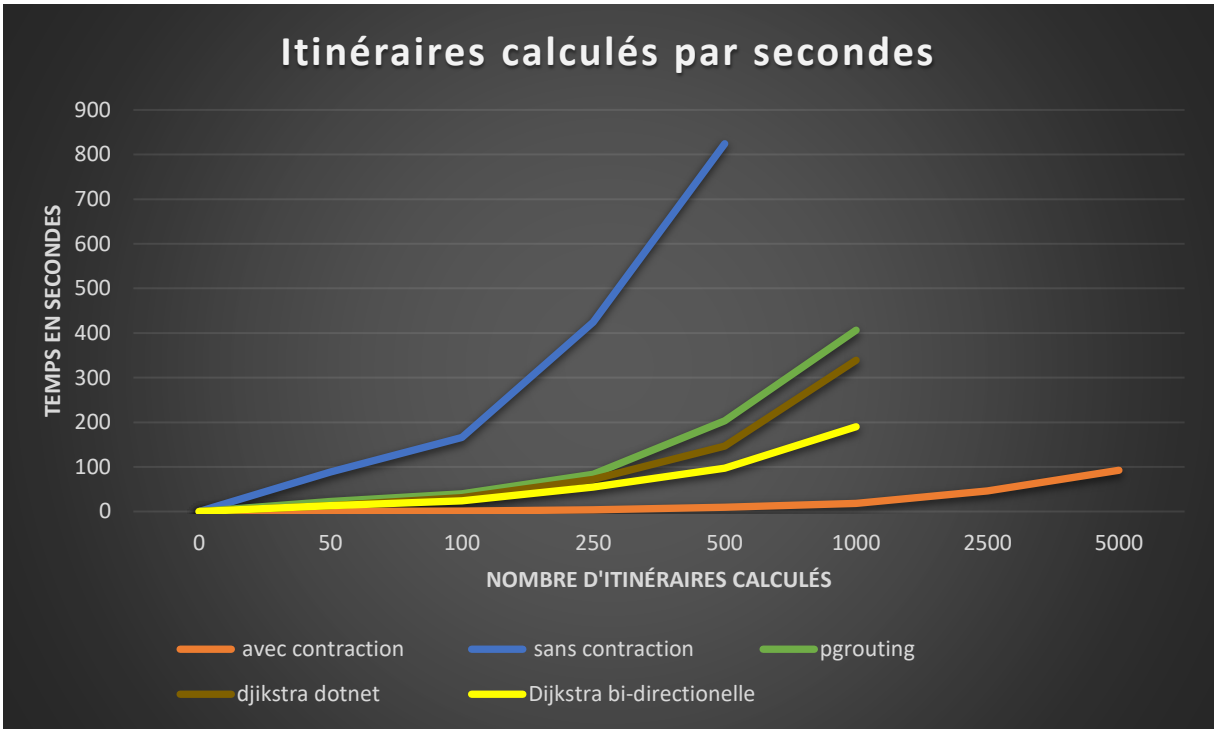


Figure 20 : résultat du temps de calcul pris une fois l'ajout de la recherche bidirectionnelle implémenté dans l'algorithme.

L'ancienne recherche correspondant à l'algorithme de Dijkstra traditionnelle a été recolorée en brun et la recherche bidirectionnelle en jaune. Comme nous pouvons le constater, l'ajout de la recherche bidirectionnelle a permis un gain assez intéressant correspondant en moyenne par rapport à l'ensemble des paliers de 25%.

Seulement pour l'instant le résultat est encore assez éloigné de la version contracté d'itinero qui reste à l'heure actuel, la meilleure des technologies.

8 Conclusion

La problématique principale de départ était de réaliser une solution qui regroupait à la fois l'implémentation du graphe modifiable dynamiquement et l'algorithme qui serait en mesure de faire des calculs d'itinéraires sur celui-ci. Pour atteindre cet objectif, nous avons réalisé un graphe représentant la Belgique via une carte OSM pour être en mesure par la suite de trouver des itinéraires entre deux localisations. Un algorithme de Dijkstra a été implémenté pour réaliser ces calculs d'itinéraires. Les résultats obtenus et présentés à la section 3.4 nous montrent qu'itinero est la meilleure technologie pour le calcul d'itinéraire et que notre algorithme est en seconde place sur le benchmark.

La deuxième problématique était d'améliorer l'implémentation que nous avons réalisée afin de la rendre plus performante. Le gain par rapport à l'implémentation de base est assez important (plus de 30% en terme vitesse de calcul) et nous a aidé à nous rapprocher de la version rapide d'itinero, cependant comme montré à la section 7, notre implémentation est dix fois plus lente qu'itinero qui reste donc la meilleure technologie. Notre algorithme comparativement aux autres technologies est assez efficace et il est sans doute possible d'améliorer encore un peu l'implémentation. Néanmoins il semble peu probable d'être en mesure d'atteindre les mêmes performances de la meilleure version d'itinero qui reste donc la meilleure solution.

Concernant les améliorations qui peuvent être apportées, nous pourrions ajouter une heuristique pour diriger la recherche vers la destination, comme l'incorporation d'une estimation à vol d'oiseau pouvant certainement assez bien améliorer les performances de l'algorithme, ou encore l'utilisation d'une FAP basée sur un tas de Fibonacci, qui pourrait donner de meilleurs résultats (1), p. 505-506. La suite du projet se poursuivra sur l'implémentation de l'algorithme sur une plateforme permettant de faire des calculs d'itinéraires à partir de requêtes venant de diverses applications.

9 Bibliographie

1. **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.** *Introduction to Algorithms, Third Edition.* Cambridge, Massachusetts : The MIT Press, 2009.

2. **Robert Geisberger, Peter Sanders, Dominik Schultes, Daniel Delling.** Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. *algo2.iti.kit.edu.* [En ligne] 2008. http://algo2.iti.kit.edu/schultes/hwy/contract.pdf.

3. **Geisberger, Robert.** algo2.iti.kit.edu. [En ligne] 2008. http://algo2.iti.kit.edu/documents/routeplanning/geisberger_dipl.pdf.

4. **_.** Contraction_hierarchies. *wikipedia.* [En ligne] https://en.wikipedia.org/wiki/Contraction_hierarchies.

5. **Andrew V. Goldberg, Chris Harrelson, Haim Kaplan, Renato F. Werneck.** Efficient Point-to-Point Shortest. *www.cs.princeton.edu.* [En ligne] https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf.

6. **_.** Single-Pair Shortest-Path using Dijkstra. *www.semanticscholar.org.* [En ligne] 04 2016. https://www.semanticscholar.org/paper/Single-Pair-Shortest-Path-using-Dijkstra/4599add6498fc5bb26260aa54c87bf7855a9f7f5?p2df.

7. **Patel, Amit.** Heuristics. *theory.stanford.edu.* [En ligne] http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#manhattan-distance.