Recherche de plus court chemin multimodal de point à point dépendant du temps

Arthur Finkelstein

▶ **To cite this version:**

Arthur Finkelstein. Recherche de plus court chemin multimodal de point à point dépendant du temps. Recherche opérationnelle [math.OC]. Université Côte d'Azur, 2022. Français. NNT : 2022COAZ4048 . tel-03851145

HAL Id: tel-03851145
https://theses.hal.science/tel-03851145v1

Submitted on 14 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Recherche de plus court chemin multimodal de point à point dépendant du temps

Arthur FINKELSTEIN

Laboratoire d'Informatique, de Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 UCA CNRS

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur

Dirigée par : Jean-Charles RÉGIN
Co-encadrée par : Mohammed-Amine AIT-OUAHMED
Soutenue le : 22 septembre 2022

Devant le jury, composé de :
David COUDERT, Directeur de recherche, Inria, Sophia-Antipolis
Leandro CALLIGARI COELHO, Professeur titulaire, Université Laval, Québec
Emmanuel NÉRON, Professeur, Université de Tours
Dorothea WAGNER, Professeur, Karlsruhe Institute of Technology
Yann HERVOUET, PDG, Instant System

RECHERCHE DE PLUS COURT CHEMIN MULTIMODAL DE POINT À POINT DÉPENDANT DU TEMPS

Time-dependent multimodal point to point shortest path

Arthur FINKELSTEIN

Jury :

Président du jury
David COUDERT, Directeur de recherche, Inria, Sophia-Antipolis

Rapporteurs
Leandro CALLIGARI COELHO, Professeur titulaire, Université Laval, Québec
Emmanuel NÉRON, Professeur, Université de Tours

Examinateurs
Dorothea WAGNER, Professeur, Karlsruhe Institute of Technology

Membres invités
Yann HERVOUET, PDG, Instant System

Université Côte d'Azur

Arthur FINKELSTEIN

Recherche de plus court chemin multimodal de point à point dépendant du temps

xiii+135 p.

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : sommaire.tex – 12/10/2022 – 11:04

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

À Marion

Recherche de plus court chemin multimodal de point à point dépendant du temps

Résumé

Calculer un itinéraire est un problème fondamental dans notre société. Le problème du plus court chemin est étudié depuis de nombreuses années et fait partie des problèmes les plus connus en théorie des graphes et en recherche opérationnelle. Une variante de ce problème existe pour les réseaux de transports en commun, avec la nécessité de modifier les algorithmes pour prendre en compte la temporalité des transports en commun. De nombreux algorithmes existent, mais de nouveaux algorithmes doivent être créés pour répondre aux besoins des utilisateurs.

Dans cette thèse, nous étudions le calcul d'itinéraires pour les transports en commun et son utilisation dans des applications de mobilité pour smartphone. La première contribution est un algorithme multiobjectif, pour répondre aux différents besoins des utilisateurs, qui utilise du précalcul afin de diminuer le temps de réponse. Cet algorithme est basé sur la recherche guidée par le but, ainsi pour un itinéraire entre Nice et Cannes, les itinéraires passant par Menton ne seront pas étudiés.

La deuxième contribution est l'application du problème des k plus courts chemins aux réseaux de transports en commun. Le problème des k plus courts chemins énumère les plus courts chemins entre un point de départ et un point d'arrivée. Leurs applications aux réseaux de transports en commun permettent de fournir un grand nombre de résultats avec un temps de réponse très rapide. En outre, les résultats produits s'avèrent bien souvent dissimilaires, ce qui permet de fournir aux utilisateurs des résultats potentiellement intéressants.

La dernière contribution est un algorithme intermodal combinant les transports en commun et le covoiturage. Un tel algorithme permet de pallier les désavantages de chaque réseau : pour les réseaux de transports en commun, les horaires de passages peuvent ne pas satisfaire les besoins de l'utilisateur et pour le covoiturage, trouver un conducteur qui passe par le point de départ et d'arrivée d'un utilisateur est très rare. Cet algorithme répond à une demande fréquente des acteurs du monde moderne.

Mots-clés : calcul d'itinéraires, plus court chemin, optimisation Pareto, chemins dissimilaires, covoiturage.

Time-dependent multimodal point to point shortest path

Abstract

Computing a route is a fundamental problem in our society. The shortest path problem has been studied for many years and is one of the best known problems in graph theory and operations research. A variant of this problem exists for public transit networks, with the need to modify the algorithms to take into account the temporality of public transit. Many algorithms exist, but new algorithms must be created to meet the needs of users.

In this thesis, we study public transit routing and its use in mobility applications for smart phones. The first contribution is a multi-objective algorithm, to meet different user needs, which uses precomputation to decrease the response time. This algorithm is based on goal-directed search, so for a route between Nice and Cannes, the routes passing through Menton will not be studied.

The second contribution is the application of the k shortest path problem to public transit networks. The k shortest paths problem enumerates the shortest paths between a starting point and an ending point. Their application to public transit networks provides a large number of results with a very fast response time. Moreover, the results produced are often dissimilar, which allows to provide users with potentially interesting results.

The last contribution is an intermodal algorithm combining public transport and carpooling. Such an algorithm overcomes the disadvantages of each network: for public transit networks, the transit schedules may not satisfy the user's needs, and for carpooling, finding a driver who passes by a user's departure and arrival point is very rare. This algorithm responds to a frequent request from the modern world.

Keywords: public transit routing, shortest path, pareto optimization, dissimilar paths, ridesharing.

Remerciements

Je tiens à remercier d'abord les membres de mon jury. Merci à Leandro Coelho et à Emmanuel Neron d'avoir été rapporteurs de ma thèse. Merci également à David Courdert, Yann Hervouet et Dorothea Wagner d'avoir accepté de faire partie du jury de ma thèse.

Je remercie infiniment Jean-Charles Régin d'avoir accepté d'encadrer ma thèse et d'avoir proposé une thèse CIFRE avec Instant System. Je remercie également toutes Instant System, Xavier et Yann d'avoir accepté de me prendre en stage de M2 qui est ensuite devenu cette thèse. Ainsi qu'à toute l'équipe R&D , Amine, Florian, Amosse et Ben pour leurs multiples conseils.

Je remercie aussi Carine Fedele pour toutes les relectures, sans elle cette thèse ne serait pas sous cette forme.

Merci aux membres du laboratoire I3S et de l'équipe MDSC tout particulièrement, mes cobureaux Rémi et Julia, mes anciens professeurs Enrico, Arnaud, Marie et Cinzia, tous les thésards Sara, Laetitia(s), Alexandre, Nicolas, Samvel, Loïc, Steve et François. Et merci à Ali pour tous ses efforts sur notre projet des k plus courts chemins.

Je remercie Marion, ma compagne, d'avoir su me motiver tout au long de cette thèse et ma famille pour son soutien.

Et finalement, je remercie Christophe, Sylvain et Wesley d'avoir toujours été là au fil des années malgré la distance.

Table des matières

Contributions

CHAPITRE 1

Introduction

1.1 Les transports en France

Un besoin fondamental est celui de pouvoir se déplacer librement et facilement. Depuis des dizaines d'années, des investissements colossaux sont faits dans le développement des réseaux routiers et de transports en commun. Cette évolution implique une augmentation de leurs utilisations et donc une augmentation des émissions de gaz à effet de serre. En 2019, le secteur des transports représente 31% des émissions françaises [4]. De plus, le parc automobile de la France est composé de 44,4 millions de véhicules dont 85% sont des véhicules privatifs et seulement 2% sont des poids lourds, dont les bus et cars font partie. Les émissions de gaz à effet de serre dues au déplacement des personnes en voiture particulière sont à l'origine de 51% des émissions du secteur des transports. Les poids lourds sont quant à eux à l'origine de 22% des émissions du secteur des transports.

Les émissions de gaz à effet de serre des véhicules privatifs représentent donc une part majoritaire des émissions du secteur des transports. Le nombre de personnes par voiture est extrêmement faible [Santos et al., 2011], ce qui entraîne des embouteillages et une utilisation peu efficace du réseau routier. À l'inverse, le réseau de transports en commun mutualise un moyen de transport (bus, car, tram, métro ...) pour utiliser soit le réseau routier, soit un réseau propre. La différence principale entre ces deux modes est que la voiture privative permet à son conducteur d'aller exactement de son point de départ à son point d'arrivée aux horaires qu'il souhaite (au moins pour le départ), alors que les transports en commun permettent uniquement d'aller de l'arrêt le plus proche du départ à l'arrêt le plus proche de l'arrivée, à des horaires prédéfinis.

Le très grand nombre de voitures sur la route a un impact, non pas seulement sur la santé des Français, avec la pollution de l'air, mais également sur leur temps libre. Ainsi, en 2021 les automobilistes français auraient passé plus de 140 heures dans des embouteillages [9]. Si le taux d'occupation des voitures privatives ou des transports en commun augmentait, alors le nombre de véhicules sur la route diminuerait. Ce qui permettrait de diminuer les embouteillages ainsi que les émissions de gaz à effet de serre.

La plupart des Français seraient prêts à utiliser les transports en commun, mais trouver son chemin dans un réseau de transports en commun est une tâche complexe. Les lignes ont des heures de passage fixes et prendre un moyen de transport qui se dirige vers l'arrivée n'est pas forcément la solution la plus rapide. Par exemple, partir dans la direction opposée à notre arrivée, afin de rejoindre un pôle de transports pour y prendre un bus direct vers notre arrivée. De plus, un « bon » itinéraire pour une personne ne l'est peut-être pas pour une autre personne. Par exemple, il existe deux itinéraires pour aller de Nice à Sophia Antipolis, soit prendre le 230 qui est un bus direct passant par l'autoroute, soit prendre un train puis le bus A. Certains utilisateurs préféreront rester

assis dans le 230 pour éviter un transfert même si le trajet est un peu plus long, tandis que d'autres préféreront prendre le train puis le bus, même s'il faut faire un transfert pour arriver plus vite.

De la même manière que pour les transports en commun, une partie des Français seraient prêts à covoiturer. Malheureusement, trouver plusieurs utilisateurs qui ont les mêmes points de départ et les mêmes points d'arrivée est une tâche complexe. Même si un covoiturage quotidien existe entre deux personnes, si l'un des deux doit exceptionnellement rester plus tard ou partir plus tôt, cela peut ne pas convenir. Il faut donc pouvoir facilement trouver d'autres offres de covoiturage.

Il existe des leviers utilisés par l'État français pour petit à petit changer les habitudes de déplacements vers des modes plus vertueux. Par exemple, en 2018 le « plan de mobilité » [7, 10] a été rendu obligatoire pour les entreprises de plus de 100 salariés. Ce plan vise à améliorer la mobilité du personnel en encourageant l'utilisation des transports en commun et le recours au covoiturage.

De plus, depuis le 10 mai 2020, afin d'encourager l'utilisation de mode de transports plus propre pour les trajets domicile-travail, l'État français a mis en place le « forfait mobilités durables » [12, 11]. Ce forfait permet la prise en charge par l'employeur d'une partie des frais de transports personnels, pour des modes permettant une réduction des émissions de gaz à effets de serre, tels que :

- les vélos ;
- la voiture dans le cadre du covoiturage ;
- les engins de déplacement personnels (trottinettes, scooters ...) ;
- tout autre service de mobilité partagée.

Ce forfait est une raison de plus pour les utilisateurs d'utiliser les transports en commun en conjonction avec d'autres modes de transports, soit « doux » comme le vélo ou la trottinette, soit le covoiturage.

Le travail de ma thèse s'inscrit dans cette même idée de démocratiser l'utilisation des transports en commun au plus grand nombre. Grâce à l'élaboration d'algorithmes de calculs d'itinéraires pour les transports en commun intégrés dans des applications de mobilités pour des agglomérations ou des régions [5, 3], mais également, en développant le covoiturage. Ce mode de transports est moins utilisé mais a un très fort potentiel si on l'utilise conjointement avec les transports en commun. En effet, cette union permet de pallier les désavantages de chacun de ces deux modes de transports.

1.2 Objectifs de cette thèse

L'objectif de cette thèse est dans un premier temps, de fournir aux utilisateurs de transports en commun, des algorithmes qui renvoient un ensemble de résultats « intéressants ». La définition d' « intéressant » est complexe, car elle change pour chaque utilisateur. Cependant, des critères universels peuvent être trouvés. Par exemple, le nombre de transferts et la distance de marche peuvent être vus de manière très différente d'une personne à une autre.

Par ailleurs, la très grande majorité des applications de calcul d'itinéraires pour les transports en commun sont accessibles soit sur Internet, soit sur des smartphones. Cela implique que le temps de calcul est très limité, pas plus de quelques secondes sinon les utilisateurs pensent que le programme ne fonctionne pas. Il en découle deux contraintes :

- fournir plusieurs résultats par rapport à des critères ;
- répondre en moins de quelques secondes.

Cela demande l'élaboration d'un algorithme spécifique pour répondre à ce problème.

Dans un deuxième temps, nous constatons qu'en utilisant uniquement les deux critères décrits précédemment (la distance de marche et le nombre de transferts), les résultats peuvent ne pas satisfaire tous les utilisateurs. En effet, les utilisateurs peuvent avoir d'autres besoins en termes de mobilité qui ne dépendent pas des deux critères. Par exemple, un utilisateur peut préférer une ligne par rapport à une autre car celle-ci est moins fréquentée qu'une autre, lui assurant une place assise. Il faudrait que les utilisateurs puissent ajouter leurs propres critères dans la recherche d'itinéraires. Mais, dans ce cas, la deuxième contrainte ne serait plus respectée, car le temps de réponse serait beaucoup trop long. Une solution à ce problème est de trouver un grand nombre de résultats, pour ensuite n'en garder qu'un sous-ensemble qui pourrait intéresser les utilisateurs.

Enfin, nous pouvons constater que les deux méthodes précédentes permettent de donner plus de résultats aux utilisateurs des transports en commun. Néanmoins, le réseau de transports en commun n'est pas toujours suffisant pour trouver des résultats satisfaisants pour un utilisateur. C'est pourquoi, le dernier objectif de cette thèse est de combiner le covoiturage et les transports en commun dans un même calculateur. Le but est d'utiliser le covoiturage avec une approche pragmatique et concrète. Ainsi, le premier utilisateur arrivé est le premier servi par un conducteur. Nous ne cherchons pas à couvrir l'ensemble des demandes de covoiturage comme c'est souvent le cas, mais nous considérons le covoiturage comme une sorte de transport en commun avec des arrêts variables.

Notations

Graphe

G	un graphe
X	l'ensemble des noeuds d'un graphe
A	l'ensemble des arêtes d'un graphe
ℓ	fonction de distance pour un arc
dist(s,t)	distance entre deux noeuds du graphe

Table horaire

S	l'ensemble des arrêts
T	l'ensemble des "trips"
C	l'ensemble des connexions
F	l'ensemble des chemins piétons

Calcul d'itinéraire pour les transports en commun

s	noeud ou arrêt source
t	noeud ou arrêt puits
τ_0	heure de départ
τ_{max}	heure maximale d'arrivée
$d_{PT}(s,t,\tau_s)$	durée entre deux arrêts en transports en commun
$\mathcal{B}(a)$	frontière d'une zone
$\overline{d_{PT}}(s,t,\tau_s)$	borne supérieure de la durée en transports en commun
$\underline{d_{PT}}(s,t,\tau_s)$	borne inférieure de la durée en transports en commun

K plus courts chemins

s	noeud ou arrêt source
t	noeud ou arrêt puits
τ_0	heure de départ
τ_{max}	heure maximale d'arrivée
P_i	ième plus court chemin
J_i	ième chemin d'arrivée au plus tôt
π	préfixe d'un chemin
dep$_t(J)$	fonction qui renvoie l'heure de départ de J
arr$_t(J)$	fonction qui renvoie l'heure d'arrivée de J
M	résultat de l'algorithme PCSA

CHAPITRE 2

Présentation d'algorithmes classiques de cheminement

Le calcul d'itinéraire est un problème fondamental dans notre société, aussi bien pour les réseaux routiers que pour les réseaux de transports en commun. L'ajout de plusieurs critères (le nombre de transferts, la distance de marche, le prix ...) complexifie grandement le calcul d'itinéraire, cependant la qualité des résultats fournis aux utilisateurs est sans pareil. Cela crée plusieurs problèmes majeurs en fonction de la qualité des résultats souhaités. Les premiers algorithmes datent des années 1950 et les travaux pour ce problème continuent encore maintenant. En plus des problèmes majeurs qui apparaissent, nous voyons plusieurs problématiques apparaitre en fonction des requêtes utilisateurs. Nous allons voir les différents algorithmes qui existent pour répondre aux problèmes majeurs et aux problématiques, aussi bien pour le routier que pour les réseaux de transports en commun.

Route planning is a fundamental problem in our society, both for road and public transportation networks. The addition of several criteria (number of transfers, walking distance, price ...) greatly complicates the route calculation, however the quality of the results provided to users is unparalleled. This creates several major problems depending on the quality of the desired results. The first algorithms date back to the 1950's and work on this problem continues even today. In addition to the major problems that appear, we see several issues appearing depending on the user requests. We will see the different algorithms that exist to answer the major problems and issues, both for road and transit networks.

2.1 Introduction

Calculer un itinéraire est un problème fondamental dans notre société, et ce depuis l'aube des temps. Aller de manière efficace d'un point A à un point B est un besoin de tous les jours, cependant trouver un tel chemin est une tâche complexe. On peut utiliser notre sens de l'orientation ainsi que les points cardinaux pour essayer de nous guider, mais dans des réseaux de grande taille ce genre d'approche peut produire des solutions très inefficaces alors que d'autres, plus contre-intuitives, mènent au résultat recherché [Bongiorno et al., 2021], par exemple suivre le périphérique pour traverser une ville alors que la solution naïve est de garder toujours la même direction. Ce problème d'apparence simple s'avère vite assez compliqué : il ne faut pas uniquement prendre en compte la topologie du réseau, mais également le temps pour traverser chaque rue, départementale et nationale, ainsi que le temps d'attente aux feux ou bien la circulation. Par ailleurs, nous n'avons pas tous la même définition de l'efficacité : un chemin efficace pour un automobiliste n'est pas le même que pour un cycliste ni le même que pour un autre automobiliste, sans parler des piétons.

Les modèles et abstractions du problème de la recherche d'un itinéraire ont donc fortement évolué au cours du temps. On a inventé les routes, les panneaux indicateurs, les cartes, les boussoles, puis les représentations mathématiques pour aboutir aux modèles informatiques. Ainsi nous avons maintenant l'habitude de représenter les réseaux et les cartes à l'aide de la notion de graphe. Un graphe est défini par un ensemble de sommets ou nœuds, généralement noté X et un ensemble d'arcs qui relient les sommets entre eux selon un sens défini, noté A. L'arc (x, y) relie le sommet x au sommet y et ne peut être traversé que de x vers y. On peut aussi avoir la notion non orientée dans laquelle ce sont des arêtes qui relient des nœuds. Dans ce cas il n'y a pas de sens et on peut aussi bien aller de x vers y, que de y vers x. Les rues d'une ville deviendront un graphe où les nœuds correspondant aux intersections/carrefours et fins d'impasses et les arcs aux rues reliant les nœuds entre eux.

Grâce à ce modèle, on peut alors définir très précisément et sans ambiguïté les problèmes que l'on cherche à résoudre. Aller d'un point A vers un point B dans un véhicule devient la recherche d'un chemin du nœud A vers le nœud B dans le graphe des rues empruntables en voiture.

De façon générale, plusieurs problèmes se dégagent comme la recherche d'un chemin d'un nœud source, noté s, vers un nœud puits, noté t. Bien entendu, on peut associer une valeur à chaque arête, représentant un coût, un poids ou une distance que nous appellerons longueur ce qui va permettre de définir les versions pondérées des problèmes précédents ; on parlera alors de plus court chemin au lieu de parler uniquement de chemin. La longueur d'un chemin est définie par la somme des longueurs des arcs traversés. On parle ainsi de problème de plus court chemin ou de la recherche d'un chemin de s à t, ou de recherche de *s-t path* en anglais. Une variante de ce problème est la recherche d'un chemin de s vers tous les autres sommets. Une autre est la recherche de chemins entre toutes les paires de sommets.

Il existe deux algorithmes très célèbres pour résoudre les problèmes de plus court chemin, abrégé problème de pcc : l'algorithme de Dijkstra [Dijkstra et al., 1959] et l'algorithme Bellman-Ford-Moore [Bellman, 1958, Ford Jr, 1956, Moore, 1959].

Le premier est un algorithme dit glouton, parce qu'il procède par étapes successives qu'il ne remet jamais en cause. À chaque étape il effectue un choix basé sur la recherche d'un optimum local en espérant que cela permettra d'aboutir à un optimum global. Cet algorithme est optimal dans le cas où le graphe ne contient pas d'arc de coût négatif. Il explore successivement les sommets les plus proches de la source jusqu'à atteindre le puits. C'est donc un algorithme plutôt simple et effi-

cace pour rechercher un plus court chemin de s vers tous les sommets. Mais lorsqu'on s'intéresse uniquement à un chemin de s vers t, alors il se montre assez peu efficace, car il n'exploite pas le fait que l'on cherche uniquement à rejoindre t et non pas tous les autres sommets. Par exemple pour un calcul de plus court chemin entre Nice et Brest, l'algorithme va regarder s'il existe des chemins passant par l'Italie, la Suisse ou l'Espagne alors qu'il est évident qu'aucun résultat efficace ne passe par là. Pour pallier ces faiblesses, de multiples améliorations ont été proposées. Par exemple, on pourra lancer l'algorithme de s vers tous les sommets et en même temps un autre de t vers tous les sommets. Quand certains sommets en commun seront explorés, alors on pourra arrêter l'algorithme et construire le plus court chemin de s à t. On appelle cette variante l'algorithme de Dijkstra bidirectionnel [Dantzig, 1963]. On peut aussi exploiter la connaissance d'un puits t pour guider l'exploration. L'algorithme A* [Hart et al., 1968] améliore l'exploration de l'algorithme de Dijkstra en utilisant une fonction qui minore la distance entre un sommet et le puits t (par exemple en utilisant la distance à vol d'oiseau dans les problèmes géographiques). Le découpage du graphe, qui a également beaucoup d'applications, va permettre aussi de réduire l'espace de recherche en permettant de travailler à un niveau de granularité plus gros. L'idée est de commencer par essayer de déterminer les régions qui peuvent être traversées avant de s'intéresser au niveau inférieur, cela peut être fait de plusieurs manières avec l'algorithme *Precomputed Cluster Distance* [Maue et al., 2010] ou avec l'algorithme *Contraction Hierarchies* [Geisberger et al., 2012] par exemple. Finalement, des algorithmes hybrides mélangeant différentes améliorations entre elles existent [Holzer et al., 2005, Goldberg et al., 2006, Bauer et al., 2010]. L'idée est d'essayer de pallier les faiblesses d'une méthode avec une idée (structure de données, calculs) d'une autre méthode, en espérant un gain de performance.

L'autre algorithme célèbre est celui de Bellman-Ford-Moore qui présente l'avantage de pouvoir résoudre le problème dans le cas où certains coûts sont négatifs. Il s'avère aussi souvent compétitif si l'on cherche réellement les chemins de s à tous les autres sommets.

Il existe donc de nombreux algorithmes efficaces aussi bien en complexité qu'en pratique pour résoudre les problèmes de pcc. Actuellement, la demande industrielle et des utilisateurs est très forte sur une variation du problème : celui de la recherche de chemin dans des réseaux de transports. La grande différence avec les problèmes classiques de pcc est que la notion de temps est introduite. En effet, un arc (x, y) ne peut plus être traversé n'importe quand, mais seulement si l'heure d'arrivée au nœud x est compatible avec une heure de départ pour un véhicule qui emprunte l'arc (x, y). Cependant, comme un arc (x, y) peut maintenant être emprunté par plusieurs véhicules à des heures différentes, le graphe doit être transformé pour faciliter le calcul d'un pcc. La première idée pour transformer le graphe [Berge, 1983] fut de l'expanser temporellement, c'est-à-dire plutôt que d'avoir un nœud avec un ensemble d'arcs sortants avec chacun une heure de départ, on a un nœud qui est démultiplié dans le temps et qui n'a qu'un arc, celui d'un véhicule à une heure donnée. Le but est de s'affranchir de la temporalité en augmentant le nombre de nœuds du graphe, afin de permettre l'utilisation d'un algorithme classique de pcc. Le revers de la médaille de cette représentation est le nombre conséquent de nœuds ajoutés : si un arrêt est desservi toutes les 15 minutes, pour une journée nous aurons plus de 50 nœuds pour représenter ce concept dans le graphe expansé temporellement.

Un réseau de transports en commun est fait de lignes qui sont desservies par des véhicules avec une temporalité extrêmement forte. Expanser le graphe de transports en commun fait disparaître sa structure et ralentit les algorithmes utilisés. De ce constat, plusieurs algorithmes qui exploitent la structure particulière des réseaux de transports en commun voient le jour : RAPTOR [Delling et al., 2015b] qui utilise uniquement les lignes pour explorer le réseau, *Connection Scan*

Algorithm (CSA) [Dibbelt et al., 2018] qui utilise uniquement la temporalité des arcs, *Trip Based Routing* [Witt, 2015] qui utilise les trajets des véhicules. Ces algorithmes sont détaillés dans la section 2.4.

La plupart des algorithmes présentés précédemment fournissent un unique résultat. Néanmoins, un même résultat peut ne pas convenir à deux personnes différentes, par exemple si l'une d'entre elles n'aime pas marcher. Ainsi, de la même manière que l'efficacité est différente pour deux automobilistes, deux utilisateurs de transports en commun peuvent avoir des besoins différents. Par exemple, un chemin en transports en commun qui arrive à 16h avec 4 transferts peut plaire à certains usagers, mais peut déplaire tout autant à d'autres usagers, pour des raisons personnelles (peur de rater un transfert ...). Fournir plusieurs résultats est une solution pour satisfaire les besoins des différents clients. Cependant il faut que les résultats soient suffisamment différents. Deux résultats qui sont identiques sur tout le trajet sauf le dernier arrêt, où un le fait avec un véhicule et l'autre à pied, ne sont pas utiles pour les utilisateurs. Pour pallier ce problème, nous allons utiliser une optimisation multiobjectif qui a pour but de trouver parmi un sous-ensemble de solutions celles qui minimisent (ou maximisent) autant que possible plusieurs objectifs. Ne connaissant pas les besoins des utilisateurs a priori, aucun objectif n'est plus important que les autres. Ce qui veut dire qu'un chemin qui arrive à 16h avec 4 transferts ne peut pas être comparé à un chemin arrivant à 16h10 avec 2 transferts. Malheureusement, trouver plusieurs résultats grâce à l'optimisation multiobjectif est un problème complexe. Chaque ajout d'un objectif augmente considérablement la taille de l'espace de recherche.

Dans un premier temps, nous allons développer les différents problèmes considérés dans cette thèse. Ensuite, nous ferons un tour d'horizon des différents algorithmes de plus court chemin, puis ceux pour le calcul d'itinéraire pour les transports en commun et finalement ceux pour le calcul d'itinéraire pour les transports en commun multiobjectif.

2.2 Problèmes considérés

Les trois problèmes majeurs considérés dans ce chapitre sont le calcul d'un plus court chemin, le calcul d'un itinéraire pour les transports en commun et le calcul d'un itinéraire pour les transports en commun multiobjectif. Le temps de réponse est très important pour ces trois problèmes, car l'utilisation sur un smartphone ou un ordinateur d'un calculateur d'itinéraire force une réponse en moins d'une seconde, sous peine de voir l'utilisateur se rabattre sur une autre alternative. Pour les deux premiers problèmes, ce critère est respecté même pour de très grands graphes. Cependant pour le troisième problème, surtout si on augmente le nombre de critères, le temps de réponse dépasse très facilement la seconde, ce qui est inacceptable pour des utilisateurs.

Le calcul d'itinéraire, qu'il soit routier, pour les transports en commun ou multiobjectif, ne se résume pas à une unique classe de problème : celle de calculer un plus court chemin entre un point de départ et un point d'arrivée. Il y en a quatre qui apparaissent :

1. Un calcul de chemin d'un point de départ vers un point d'arrivée.
2. Un calcul de chemin d'un point de départ vers tous les autres points du graphe.
3. Un calcul de chemin d'un point de départ vers un point d'arrivée, avec le point de départ et d'arrivée qui changent.
4. Un calcul de chemin de tous les points du graphe vers tous les points du graphe.

Les classes de problèmes 1 et 2 sont très souvent décrites dans les articles théoriques, mais sont très loin de ce qui est nécessaire en pratique. Pour un calculateur d'itinéraire utilisé par de nombreux utilisateurs, les points de départs et d'arrivée de toutes ces demandes sont différents, ce qui est décrit par la classe de problème 3. Finalement, nous avons la classe de problème 4 qui calcule un plus court chemin à partir de tous les points du graphe vers tous les points du graphe, qui est une solution intéressante, mais intenable, car l'espace mémoire nécessaire pour cette solution est beaucoup trop grand. Une approche qui va permettre de diminuer le temps de réponse, pour les classes de problèmes 3 et 4, est d'accepter des prétraitements afin de répondre à la contrainte qu'un calcul de chemin ne doit pas dépasser un certain temps. Nous pouvons différencier deux types de prétraitements : ceux qui utilisent des valeurs exactes et ceux qui utilisent des bornes inférieures. Ceux qui utilisent des valeurs exactes sont plus performants, mais sont très sensibles aux changements et donc peu robustes, tandis que ceux qui utilisent des bornes inférieures sont moins performants, mais beaucoup plus robustes aux changements.

2.2.1 Plus court chemin

Le problème du plus court chemin se décompose en plusieurs problèmes, le premier qui nous vient à l'esprit est celui du « plus court chemin de point à point ». On prends en entrée un graphe $G = (X, A, \ell)$, un nœud de départ $s \in X$ et un nœud d'arrivée $t \in X$ et on renvoie en sortie la longueur du plus court chemin entre s et t, ou $\text{dist}(s, t)$ la distance entre s et t. Les autres problèmes sont « le plus court chemin d'un point vers tous les points » qui va cette fois-ci calculer la longueur du plus court chemin entre le nœud de départ s et tous les autres nœuds du graphe, ou inversement « le plus court chemin de tous les points vers un point » qui va calculer la longueur du plus court chemin entre tous les nœuds du graphe et le nœud d'arrivée t. Pour finir, « le plus court chemin de plusieurs points vers plusieurs points » calcule la longueur du chemin entre un ensemble de nœuds S vers un ensemble de nœuds T et « le plus court chemin entre toutes les paires » lorsque S, T et X sont identiques.

Il est important de calculer non plus uniquement la longueur du plus court chemin, mais aussi le plus court chemin lui-même. Ce calcul du plus court chemin peut également s'appliquer au « plus court chemin d'un point vers tous les points » en créant un arbre expansé de plus court chemin (*out-shortest path tree*) qui en est une représentation compacte, de même pour « le plus court chemin de tous les points vers un point » avec l'arbre contracté de plus court chemin (*in-shortest path tree*). L'arbre a pour racine r, aussi bien pour l'expansé que pour le contracté, pour chaque nœud $u \in X$, le plus court chemin entre r et u dans G est le plus court chemin entre r et u dans l'arbre, comme nous pouvons le voir dans la figure 2.1.

De manière assez intuitive, nous savons qu'un plus court chemin n'est pas nécessairement unique. Nous pouvons avoir à faire un choix pendant notre itinéraire entre deux chemins qui sont aussi longs l'un que l'autre, par exemple contourné une ville par le nord ou le sud. Ces choix peuvent être multiples et amener à un très grand nombre de chemins avec les mêmes longueurs.

2.2.2 Calcul d'itinéraires pour les transports en commun

L'application aux transports en commun ajoute une dimension temporelle aux graphes, ainsi les arcs seront représentées par 3 paramètres (u, v, h) avec $u, v \in X$ et h un horodatage. De plus, le problème du plus court chemin va se transformer en problème d'arrivée au plus tôt qui va prendre, en plus du graphe G, du point de départ s et du point d'arrivée t, une heure de départ τ et

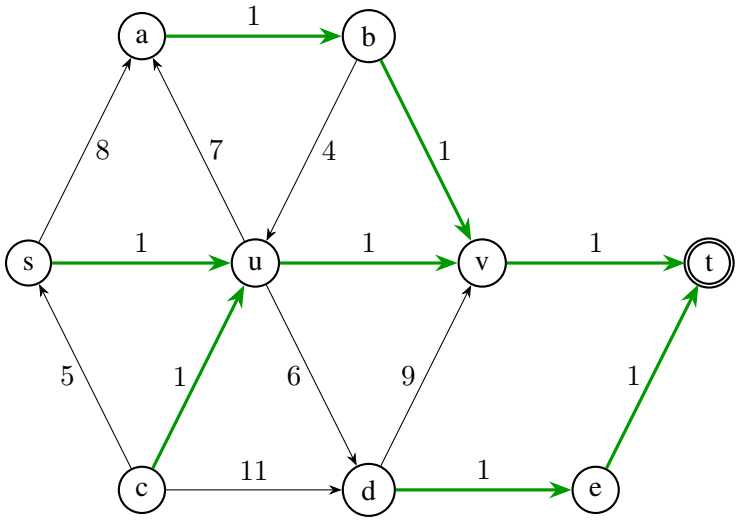


FIGURE 2.1 – Exemple d'un arbre de plus court chemin avec pour racine t.

ne calculera pas le plus court chemin qui peut partir bien après l'heure de départ τ mais le chemin qui arrive au plus tôt après l'heure de départ demandée.

Les représentations compactes pour les arbres de plus courts chemins existent également pour les arbres d'arrivées au plus tôt, aussi bien les expansés que les contractés.

2.2.3 Calcul d'itinéraires pour les transports en commun multiobjectif

Une autre problématique qui, contrairement aux problématiques précédentes, va donner plusieurs résultats « qualitatifs » est celle de l'optimisation Pareto.

Définition 2.2.1. Un ensemble d'itinéraires est Pareto optimal s'ils sont incomparables entre eux :

Soient 2 vecteurs $a = (a_1, \cdots, a_d)$ et $b = (b_1, \cdots, b_d)$, si $a \neq b$ et $a_i \leq b_i$ pour tout $1 \leq i \leq d$, alors a domine b.

Soit un ensemble V de vecteurs, $a \in V$ est Pareto optimal dans V s'il n'y a pas de $b \in V$ qui domine a.

Le front de Pareto est extrêmement dépendant du nombre de critères, car chaque critère augmente le nombre de résultats de manière exponentielle. Même si c'est rarement le cas en pratique, cette augmentation a un impact sur les performances, car chaque nœud du graphe stocke un front de Pareto qui doit être maintenu tout au long du calcul [Müller-Hannemann and Weihe, 2001]. Ainsi le choix des critères est important pour éviter de trop augmenter le nombre de résultats et donc le temps de calcul.

Il s'agit de trouver les critères qui vont satisfaire le plus grand nombre de clients, ainsi la stratégie est de choisir des critères communs et fréquemment utilisés. La distance de marche à pied dans un calcul d'itinéraire pour les transports en commun par exemple, peut ne pas être un soucis pour certains, mais le sera pour d'autres.

Comme nous pouvons le voir dans la figure 2.2, nous avons un exemple de plusieurs résultats Pareto optimaux dans le cadre des transports en commun sur le réseau de Bordeaux. Ici les critères utilisés pour le front de Pareto sont l'heure d'arrivée, l'heure de départ (qui est maximisée), le

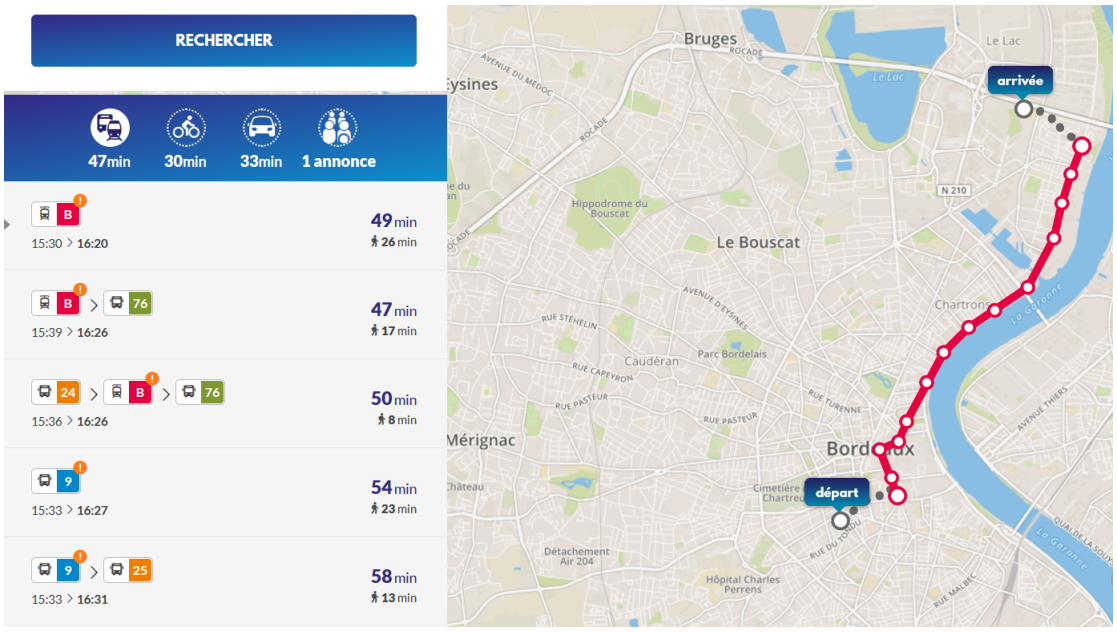


FIGURE 2.2 – Exemple d'un front de Pareto sur un réseau de transport.

nombre de transferts et la distance de marche. Le premier résultat est un itinéraire partant à 15h30, arrivant à 16h20 avec aucun transfert et 26 minutes de marche à pied. Le deuxième résultat est un itinéraire partant à 15h39, arrivant à 16h26 avec un transfert et 17 minutes de marche. Nous pouvons voir que ces deux résultats sont incomparables, car aucun des deux itinéraires n'est meilleur que l'autre sur tous les critères. Les 3 autres itinéraires sont également non dominés.

Problème d'arrivée au plus tôt par plage horaire Un sous problème intéressant pour le calcul d'itinéraire dans un réseau de transports en commun est le problème d'arrivée au plus tôt par plage horaire, appelé *profile* par les anglo-saxons, qui va simultanément résoudre le problème d'arrivée au plus tôt entre s et t pour chaque heure de départ, représentée par la figure 2.3. De manière plus théorique, cela revient à résoudre un problème Pareto optimal, qui va maximiser l'heure de départ et minimiser l'heure d'arrivée. Nous avons ainsi toutes les heures de départ des itinéraires en transports en commun intéressants, c'est-à-dire les plus rapides. Il existe des algorithmes qui sont très efficaces pour résoudre ce problème avec des temps de réponse rapide.

2.3 Algorithmes pour le calcul de plus courts chemins

Le calcul d'un plus court chemin dans un graphe quelconque est un problème étudié depuis très longtemps avec des algorithmes vus par la plupart des étudiants en informatique comme l'algorithme de Dijkstra ou le A*, en passant par des algorithmes plus complexes qui atteignent des temps de réponse extrêmement rapides, comme le *Hub Labeling*. Le nombre d'applications est vaste, avec une des plus évidentes qui est le calcul d'un plus court chemin appliqué à un réseau routier, mais il y a également des applications dans les télécommunications, la recherche opérationnelle. La représentation du réseau routier colle très fortement avec celle d'un graphe, les nœuds sont des intersections entre plusieurs routes et les arcs représentent des routes.

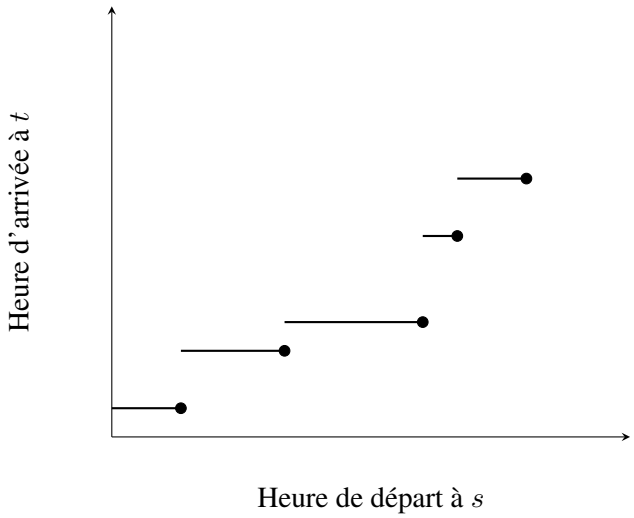


FIGURE 2.3 – Représentation d'une arrivée au plus tôt par plage horaire.

Les temps de réponses extrêmement rapides sont possibles, non seulement avec des algorithmes efficaces, mais également, grâce à du précalcul. Le but d'un précalcul est de sortir des informations importantes du réseau, comme sa structure ou des distances ou certaines propriétés. Ainsi quand nous avons un plus court chemin entre s et t qui passe par u, alors nous avons également un plus court chemin entre u et t.

Nous nous basons sur les travaux de [Bast et al., 2016] pour étudier l'état de l'art des calculs de plus court chemin. L'ensemble des algorithmes présenté dans ce chapitre sont des algorithmes exacts.

2.3.1 Algorithmes usuels

Algorithme de Dijkstra La solution standard pour le problème de plus court chemin d'un point vers tous les points est l'algorithme de Dijkstra [Dijkstra et al., 1959] (voir algorithme 2.1). L'algorithme maintient une file de priorité Q sur les distances des nœuds triés par leur distance courante par rapport à s, avec $v \in X, \text{dist}[v]$. Au début, toutes les distances ont une valeur mise à l'infini, sauf pour $\text{dist}(s) = 0$, puis s est ajouté à Q. Pour chaque itération, l'algorithme extrait de Q le nœud avec la plus petite distance à s et ce nœud est scanné, c'est-à-dire que pour chaque arc sortant (u, v) de u on regarde si passer par u permet de diminuer la distance de s vers v. Pour chaque arc $a = (u, v)$, on calcule la distance jusqu'à v en calculant $\text{dist}(u) + \ell(a)$, où $\ell(a)$ est la longueur de l'arc a : si cette valeur améliore $\text{dist}(v)$ l'algorithme met à jour $\text{dist}(v)$ et ajoute le nœud v avec comme clé $\text{dist}(v)$ à la file Q. Avec l'algorithme de Dijkstra une fois qu'un nœud $u \in X$ est scanné, la valeur de sa distance $\text{dist}(u)$ est exact. Ainsi, pour le problème de plus court chemin de point à point, l'algorithme peut s'arrêter une fois que le nœud d'arrivée t est scanné. L'ensemble des nœuds $S \subseteq X$ scannés par l'algorithme s'appelle son espace de recherche, la figure 2.4 illustre l'espace de recherche de l'algorithme de Dijkstra.

Le temps de réponse de l'algorithme de Dijkstra dépend de la file de priorité utilisée. La complexité est de $\mathcal{O}(|X|^2)$ en utilisant une liste, de $\mathcal{O}(|A| + |X|\log|X|)$ pour un tas de Fibonacci [Fredman and Tarjan, 1987] et de $\mathcal{O}(|A| + L|X|)$ en utilisant une file de priorité avec L *buckets*,

Algorithme 2.1 Algorithme de Dijkstra

```
function DIJKSTRA(G, s)
    for all vertex v ∈ X do
        dist[v] ← +∞
        prev[v] ← ∅
    end for
    dist[s] ← 0
    add u to Q
    while Q is not empty do
        u ← Q.pop()          ▷ On récupère le nœud avec la plus petite distance à partir de s
        for all neighbors v of u do
            if dist[u] + ℓ(u, v) < dist[v] then
                dist[v] ← dist[u] + ℓ(u, v)
                prev[v] ← u
                add v to Q
            end if
        end for
    end while
end function
```

appelé algorithme de Dial [Dial, 1969]. Les différentes applications sont plus ou moins efficaces en pratique, par exemple l'algorithme de Dial est borné par rapport au plus grand coût.

En pratique, l'espace de recherche peut être diminué en utilisant une recherche bidirectionnelle [Dantzig, 1963], qui lance simultanément une recherche à partir de s et une recherche en aval à partir de t. L'arrêt de l'algorithme devient plus complexe, pendant une recherche bidirectionnelle, il faut que la recherche à partir de s et la recherche à partir de t aient scanné le même nœud u qui appartient au plus court chemin entre s et t.Pour les réseaux routiers, la recherche bidirectionnelle diminue approximativement de moitié l'espace de recherche.

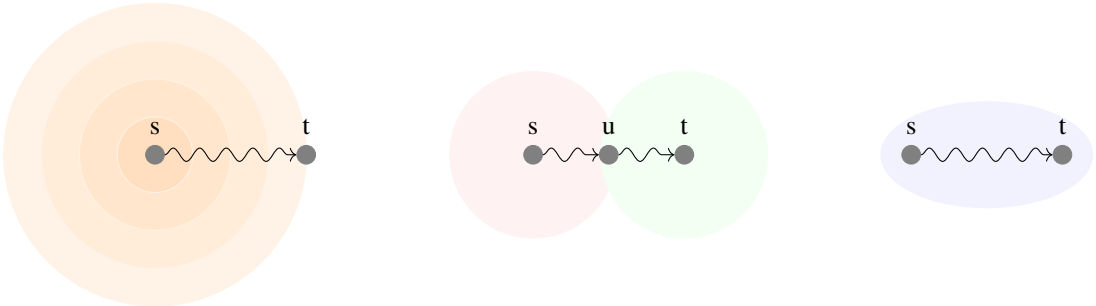


FIGURE 2.4 – Représentation schématisée de l'espace de recherche de l'algorithme de Dijkstra, du Dijkstra bidirectionnel et de l'algorithme A*.

Algorithme de Bellman-Ford-Moore Un autre algorithme classique est l'algorithme de Bellman-Ford-Moore [Bellman, 1958, Ford Jr, 1956, Moore, 1959] (voir algorithme 2.2). L'algorithme n'utilise pas de file de priorité, mais son but est que pour tout arc $a = (u, v)$ dans le graphe,

Algorithme 2.2 Algorithme de Bellman-Ford-Moore

```
function BELLMAN-FORD-MOORE(G, s)
    for all vertex v ∈ X do
        dist[v] ← +∞
        prev[v] ← ∅
    end for
    dist[s] ← 0

    for k = 0 until |X| − 1 do
        for all (u, v) ∈ A do
            if dist[u] + ℓ(u, v) < dist[v] then
                dist[v] ← dist[u] + ℓ(u, v)
                prev[v] ← u
            end if
        end for
    end for
end function
```

alors la propriété $\text{dist}[v] < \text{dist}[u] + \ell(a)$ est vraie, où dist est la distance depuis le nœud de départ. Le principe de l'algorithme est de mettre à jour les distances jusqu'à vérifier la propriété pour chaque arc. Quand un arc ne respecte pas la propriété, cela signifie qu'on peut réduire la distance du nœud de départ à v en passant par u : $d(v)$ est mis à jour avec $\text{dist}[v] = \text{dist}[u] + \ell(a)$. L'algorithme est initialisé avec la distance au nœud de départ à 0 et à $+\infty$ pour tous les autres nœuds. C'est un algorithme de type *label correcting* (correction d'étiquettes), contrairement à l'algorithme de Dijkstra qui est de type *label setting*, car chaque nœud peut être scanné de multiples fois. La complexité est de $\mathcal{O}(|X||A|)$ dans le pire des cas, mais l'algorithme de Bellman-Ford-Moore peut-être compétitif avec l'algorithme de Dijkstra dans certains scénarios. De plus, l'algorithme autorise des arcs avec des poids négatifs.

2.3.2 Algorithmes dirigés par le but

L'algorithme de Dijkstra scanne tous les nœuds qui ont une distance inférieure à $\text{dist}(s, t)$, ce qui donne un espace de recherche en cercle, pour les graphes géométriques, comme le montre la figure 2.4. Le but des algorithmes dirigés par le but est de diminuer l'espace de recherche en guidant la recherche vers le nœud d'arrivée, en évitant de scanner des nœuds qui ne sont pas dans la direction de t. Cela se fait en exploitant les propriétés du graphe ou en utilisant la structure (géométrique) du graphe, comme la structure des arbres de plus court chemin vers des zones du graphe.

A* Search L'algorithme A* [Hart et al., 1968] est un classique des algorithmes dirigés par le but. Il utilise une fonction de potentiel $\pi : X \to R$ sur les nœuds, qui est une borne inférieure sur la distance $\text{dist}(u, t)$ de u à t. Ensuite, une version modifiée du Dijkstra est lancée dans laquelle la valeur d'un nœud u dans la file de priorité Q (la distance courante par rapport à s) est changée pour devenir $\text{dist}(s, u) + \pi(u)$. L'effet de cette modification est que les nœuds proches de t sont scannés plus tôt dans l'algorithme. L'espace de recherche est ainsi diminué, comme le montre la figure 2.4. Plus particulièrement, si π est une borne inférieure exacte ($\pi(u) = \text{dist}(u, t)$), seuls

des nœuds sur les plus courts chemins entre s et t sont scannés. Cependant, avec la plupart des fonctions de potentiel le nombre de nœuds scannés est beaucoup plus grand, mais tant que cette fonction est faisable (c'est-à-dire si $\ell(v,w) - \pi(v) + \pi(w) \geq 0$ pour $(v,w) \in A$), l'algorithme peut s'arrêter quand t est sur le point d'être scanné.

L'algorithme A* peut être rendu bidirectionnel, mais il faut faire attention pour garantir son exactitude. Une approche possible est de s'assurer que la fonction de potentiel pour les deux recherches (en amont et en aval) est consistante. C'est-à-dire si pour tous les arcs $(u,v) \in A$, $\ell_{\pi_t}(u,v)$ dans le graphe d'origine est égal à $\ell_{\pi_s}(v,u)$ dans le graphe inversé, où $\ell_{\pi_t}(u,v) = \ell(u,v) - \pi_t(v) + \pi_t(u)$ et $\ell_{\pi_s}(v,u) = \ell(u,v) - \pi_s(v) + \pi_s(u)$ représentent le coût réduit de l'arc (u,v) par rapport à π_t et π_s respectivement [Goldberg and Harrelson, 2005]. Une autre approche, qui donne des résultats similaires en pratique, est de changer la condition d'arrêt.

Il est facile d'utiliser des fonctions π_t et π_s qui ne sont pas consistantes. Dans ce cas, les fonctions de distances pour les recherches en amont et en aval ne sont pas les mêmes, donc quand les recherches se rencontreront alors il n'y aura aucune garantie que le plus court chemin ait été trouvé.

2.3.3 Algorithmes avec prétraitement

Une autre approche est de faire des prétraitements afin de diminuer les temps de réponse des calculs de plus courts chemins entre s et t avec s et t variables. Ces prétraitements ont un coût, mais le but est de calculer des informations réutilisables par un très grand nombre de requêtes différentes. L'autre alternative est de calculer tous les chemins entre toutes les paires de nœuds. Cependant au moindre changement dans le graphe, les solutions ne sont plus valides. Cette deuxième solution est peu robuste, alors que les réseaux routiers ont très souvent des embouteillages ou des routes fermées, ce qui implique d'utiliser du prétraitement et de calculer des informations réutilisables.

2.3.3.1 ALT

L'espace de recherche peut être encore plus réduit avec l'utilisation de l'algorithme ALT [Goldberg and Harrelson, 2005]. Pendant une phase de précalcul, il choisit un petit ensemble $L \subseteq X$ de repères et stocke la distance entre eux et tous les nœuds du graphe. Pendant un calcul de plus court chemin entre s et t, l'algorithme utilise l'inégalité triangulaire avec les repères pour calculer une borne inférieure valide de $\text{dist}(u,t)$ pour n'importe quel nœud u. Plus précisément, pour n'importe quel repère l_i, $\text{dist}(u,t) \geq \text{dist}(u,l_i) - \text{dist}(t,l_i)$ et $\text{dist}(u,t) \geq \text{dist}(l_i,t) - \text{dist}(l_i,u)$ sont vrais. Si plusieurs repères sont disponibles, on peut prendre la meilleure borne inférieure.

La qualité des bornes inférieures (et donc des temps de réponse) dépend de quels nœuds sont choisis comme repère pendant la phase de précalcul. Pour des réseaux routiers, des repères bien espacés et près des « bords » du graphe permettent d'obtenir les meilleurs résultats.

2.3.3.2 Geometric Containers

Les *geometrics containers* sont une autre méthode restreinte aux graphes avec des coordonnées géographiques. Un précalcul est fait pour tous les arcs $a = (u,v) \in A$ pour calculer une étiquette $L(a)$ qui encode l'ensemble X_a des nœuds qui ont un plus court chemin qui commence par l'arc a. Plutôt que de stocker l'ensemble des nœuds explicitement, l'étiquette approxime cet ensemble en utilisant des informations géographiques, ici les coordonnées géographiques des arrêts. Ainsi,

pendant un calcul de plus court chemin si l'arrêt d'arrivée t n'est pas dans $L(a)$ alors a peut être supprimé de manière sûre.

[Schulz et al., 2000] stocke le label $L(a)$ avec un secteur angulaire, pour tout arc (u, v) un nœud g à gauche de (u, v) et un nœud d à droite de (u, v) sont choisis tels que tous les nœuds de l'ensemble V_a sont dans le secteur angulaire $\angle(g, u, d)$. Les nœuds d et g sont choisis de manière à minimiser l'angle $\angle(g, u, d)$. Une méthode pour déterminer comment modifier l'angle $\angle(g, u, d)$ si un nouveau nœud w n'est pas le secteur angulaire :

- Soit il est à gauche de l'arc (u, v) et le secteur angulaire devient $\angle(w, u, d)$.
- Soit il est à droite de l'arc (u, v) et le secteur angulaire devient $\angle(g, u, w)$.

[Wagner et al., 2005] utilise d'autres formes géométriques pour stocker le label $L(a)$ et montre que les *bounding boxes* ont de bonnes performances, sont simples avec seulement quatre données (les points supérieurs gauche et droit ainsi que les points inférieurs gauche et droit) et sont faciles à maintenir.

Cependant un des désavantages des *geometrics containers* est que le prétraitement consiste à faire un calcul de plus court chemin de tous les nœuds vers tous les nœuds, qui est très coûteux.

2.3.3.3 Arc Flags

Les *arc flags* [Lauther, 2006] sont similaires aux *geometric containers*, mais n'utilisent pas de géométrie. Pendant le précalcul, le graphe est partitionné en K cellules qui sont grossièrement équilibrées, c'est-à-dire qu'elles ont le même nombre de nœuds, et qui ont un petit nombre de nœuds frontières. Tous les arcs maintiennent un vecteur de K bits (*arc flags*), où le $i$ème bit est activé si l'arc est dans un plus court chemin qui a comme arrêt d'arrivée un nœud de la cellule i. Pendant la recherche d'itinéraire, l'algorithme peut supprimer les arcs qui n'ont pas le bit activé pour la cellule qui contient t.

Les *arc flags* d'une cellule i sont calculés avec un arbre de plus court chemin inversé à partir de chaque nœud frontière de i. Ensuite le $i$ème bit est activé pour chaque arc de l'arbre. Une manière alternative de le faire est d'utiliser un algorithme de type *label correcting* à partir de tous les arrêts frontières de la cellule i simultanément.

2.3.3.4 Precomputed Cluster Distances

Une autre méthode qui utilise elle aussi une partition (si possible équilibrée) est le *precomputed cluster distances* [Maue et al., 2010]. Soit une partition $\mathcal{C} = (C_1, C_2, \cdots, C_K)$ des nœuds du graphe avec K cellules, un précalcul va trouver la plus courte distance entre toutes les paires de cellules.

Le calcul d'itinéraire est une version modifiée de l'algorithme de Dijkstra. Pour tout nœud u scanné, une borne inférieure sur la distance à t est $\text{dist}(s, u) + \text{dist}(C(u), C(t)) + \text{dist}(v, t)$ où $C(u)$ est la cellule contenant u et v est le nœud frontière de $C(t)$ le plus proche de t. Si cette borne inférieure est plus grande que la meilleure borne inférieure actuelle de $\text{dist}(s, t)$ alors le nœud est écarté de la recherche.

Le *precomputed cluster distances* est une méthode robuste aux changements, ce qui prend tout son sens pour les transports en commun. Nous allons proposer un algorithme qui se base sur cette idée.

2.3.3.5 Compressed Path Databases

La méthode des *compressed path databases* stocke de manière implicite les informations des chemins entre toutes les paires de nœuds pour qu'on puisse extraire rapidement un plus court chemin pendant un calcul de plus court chemin. Tous les nœuds $u \in X$ maintiennent un label $L(u)$ qui stocke le premier arc du plus court chemin vers tous les autres nœuds v du graphe. Un calcul de plus court chemin à partir de s scanne $L(s)$ pour trouver le premier arc (s, u) du plus court chemin vers t. Ensuite on fait de même pour u jusqu'à atteindre t.

Stocker de manière explicite tous les premiers arcs des plus courts chemins prend trop d'espace, $\Theta(X^2)$. Une alternative est de grouper les plus courts chemins qui ont le même premier arc. Les temps de réponse sont rapides mais la consommation en mémoire est très grande.

2.3.4 Algorithmes basés sur des séparateurs

Les graphes planaires ont des petits séparateurs [Lipton and Tarjan, 1979]. Si les graphes routiers ne sont pas planaires à cause de la présence de ponts ou de tunnels, ils ont également de petits séparateurs qui sont facilement calculables [Eppstein and Goodrich, 2008].

2.3.4.1 Vertex Separators

Le but des séparateurs S est de faciliter le calcul de plus court chemin, par exemple pour un calcul de plus court chemin entre x et y, nous pouvons calculer en premier un plus court chemin entre x et S, puis un plus court chemin entre S et y, qui nous donne le plus court chemin entre x et y, comme le montre la figure 2.5. De manière un peu plus formelle $\text{pcc}(x, y) = \min_{v \in S}(\text{pcc}(x, v) + \text{pcc}(v, y))$.

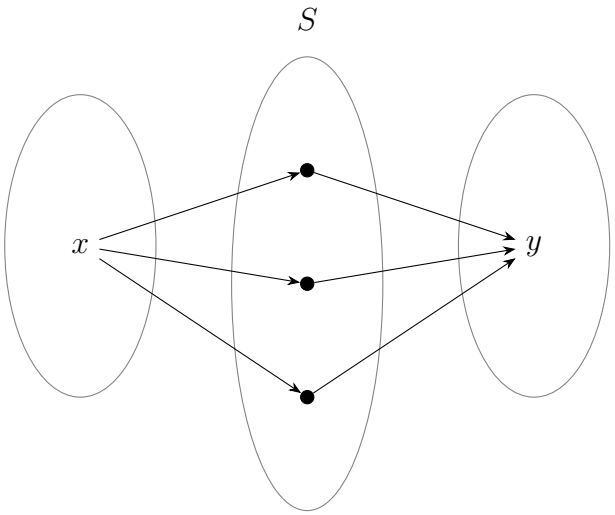


FIGURE 2.5 – Représentation schématique d'un séparateur.

Un *vertex separator* est un sous-ensemble de nœuds $S \in X$ qui une fois enlevé décompose le graphe en plusieurs régions. Ce séparateur peut être utilisé pour calculer un graphe de superposition G'. Des arcs raccourcis sont ajoutés à la superposition afin que les distances entre toutes les paires de nœuds de S soient préservées, c'est-à-dire qu'elles soient équivalentes à celles dans G.

Le graphe de superposition est beaucoup plus petit et peut être utilisé pour accélérer les calculs d'itinéraires.

[Schulz et al., 2000] utilise un graphe de superposition sur un sous-ensemble S soigneusement choisi, mais pas nécessairement séparateur, de nœuds « importants ». Pour chaque paire de nœuds $u, v \in S$, un arc (u, v) est ajouté à la superposition si le plus court chemin de u à v dans G ne contient pas d'autre nœud w dans S. Cette approche peut être étendue encore plus loin avec des hiérarchies multiniveaux, en plus d'arcs entre les nœuds séparateurs d'un même niveau, la superposition a pour chaque région au niveau i des arcs entre les séparateurs des niveaux supérieurs et inférieurs. Ainsi pour un calcul de plus court chemin entre s et t, on remonte dans le graphe de superposition jusqu'à ce que s et t soient contenues dans le même sous-ensemble ce qui nous donne le plus court chemin entre s et t ou si le plus haut niveau est atteint alors on calcule le plus court chemin dans ce graphe de superposition.

2.3.4.2 Arc Separators

Une autre méthode pour séparer le graphe se base sur les arcs pour construire un *overlay graph*. On partitionne d'abord les nœuds du graphe en un ensemble de cellules $\mathcal{C} = (C_1, C_2, \cdots, C_K)$ en essayant de minimiser le nombre d'arcs coupés, ceux qui sont entre les nœuds frontières de cellules différentes. Ensuite, des arcs sont ajoutés entre toutes les cellules de la partition et une matrice de distance est calculée entre toutes les cellules.

Une des premières versions de cette approche est la méthode Hierarchical Multi (HiTi) [Jung and Pramanik, 2002]. Un *overlay graph* est construit contenant tous les nœuds frontières des cellules et tous les arcs coupés. De plus, l'algorithme HiTi ajoute à l'*overlay graph* pour chaque paire de nœuds frontières u, v dans C_i, un arc raccourci qui représente le plus court chemin entre u et v dans le graphe original restreint à C_i. Pour un calcul de plus court chemin entre s et t, l'algorithme de Dijkstra est utilisé sur un graphe induit composé des cellules contenant s et t ainsi que l'*overlay graph*.

Plus récemment, l'algorithme *Customizable Route Planning* (CRP) [Delling et al., 2011a] utilise une approche similaire, mais est spécifiquement conçu pour répondre aux exigences de systèmes réels appliqués aux transports en commun. L'algorithme permet une gestion du coût des virages (un virage à gauche à une intersection est plus compliqué qu'un virage à droite) et est optimisé pour permettre une mise à jour rapide de la fonction de coût. De plus, l'utilisation de PUNCH [Delling et al., 2011b], un partitionneur de graphe spécialisé dans les graphes routiers permet d'obtenir de très bonnes partitions. L'algorithme CRP a un précalcul en deux phases : un précalcul indépendant de la métrique et de la configuration. La première précalcule, en plus du partitionnement multiniveaux, la topologie des superpositions. À noter que la partition ne dépend pas de la fonction de coût. La deuxième phase calcule les coûts des arcs entre tous les nœuds frontières, pour chaque niveau de la partition.

2.3.5 Hub Labeling

L'idée derrière l'algorithme *Hub Labeling* [Cohen et al., 2003] est de précalculer les distances entre des paires de nœuds, en ajoutant de manière implicite des arcs « virtuels » dans le graphe. Ainsi les calculs d'itinéraire peuvent renvoyer la longueur « virtuelle » en utilisant uniquement les distances précalculées entre les paires de nœuds et pas le graphe d'origine.

Pendant une étape de précalcul, un *label* $L(u)$ est calculé pour chaque nœud u du graphe, tel que, pour chaque paire de nœuds (u, v) la distance dist(uv) peut être déterminée en utilisant uniquement les *labels* $L(u)$ et $L(v)$. Les *labels* sont choisis de manière à ce qu'ils obéissent à la propriété de couverture : pour n'importe quelle paire de nœuds (s, t), $L(s) \cap L(t)$ doit contenir au moins un nœud, aussi appelé *hub*, du plus court chemin entre s et t. Ensuite, la distance dist(s, t) peut être déterminée en temps linéaire (en fonction de la taille du *label*) en calculant dist$(s, t) =$ min$\{$dist$(s, u) +$ dist$(u, v) \mid u \in L(s)$ et $u \in L(t)\}$. Chaque *hub* va aussi garder la distance entre u et v (ou v et u), ce qui implique de parcourir tous les hubs pour trouver celui avec la distance minimale.

Pour des graphes dirigés, les *labels* associés avec u sont divisés en 2 : les *forward labels* $L_f(u)$ qui ont les distances de u jusqu'aux *hubs*, et les *backward labels* $L_b(u)$ qui ont les distances des *hubs* jusqu'à u. Ainsi un plus court chemin entre s et t a un *hub* dans $L_f(u) \cap L_b(u)$.

Le nombre de moyens de *label* pour chaque noeud est en général $\Theta(|V|)$. Ce nombre peut être significativement réduit pour certaines classes de graphes. L'algorithme *Hub Labeling* est l'algorithme le plus rapide pour le calcul d'un plus court chemin dans un réseau routier, en moyenne le même temps que 5 accès mémoire. Son désavantage est une consommation mémoire supérieure à la moyenne, qui peut être réduite au prix d'un temps de réponse plus long.

2.3.6 Parallélisation du calcul de plus courts chemins

Il existe différentes méthodes pour diminuer le temps de réponse d'un calcul de plus court chemin en parallélisant le travail à faire. Le premier moyen est d'utiliser le processeur (CPU) en donnant des opérations complexes à un faible nombre de cœurs. Le deuxième moyen de le faire est d'utiliser le processeur graphique (GPU) en donnant des opérations simples à un très grand nombre de cœurs.

2.3.6.1 Utilisation du Central Processing Unit (CPU)

Une méthode pour paralléliser le calcul de plus court chemin est d'utiliser la librairie Boost [1] qui implémente plusieurs variantes de l'algorithme de Dijkstra. L'algorithme de Dsijktra peut être parallélisé en autorisant plusieurs nœuds à être scannés en même temps pendant une étape appelée *superstep*. Le potentiel souci de cette méthode est qu'un nœud scanné pendant une super étape peut avoir un voisin qui aurait du être scanné avant les autres nœuds ce qui peut amené a du travail supplémentaire.

La première variante est celle de [Crauser et al., 1998] qui permet de scanner plus de nœuds pendant un *superstep*. Le désavantage de cet algorithme est le fait de devoir maintenir trois files de priorité, ce qui crée beaucoup de travail à chaque nœud. Le nombre de *superstep* est de $\mathcal{O}(\sqrt[3]{n})$ pour un graphe généré de manière aléatoire.

La deuxième variante est celle de [Meyer and Sanders, 1998], elle permet de scanner tous les nœuds qui sont à une distance comprise entre une constante et la valeur de la plus petite distance. Cette constante s'appelle le *lookahead*. Cette variante utilise un δ, qui est un *lookahead*, qui va permettre de scanner des nœuds avant d'être sûr d'avoir trouvé leur plus courte distance, ce qui permet encore plus de parallélisme que la variante précédente. La variante introduit une structure de données multi niveaux de *buckets* qui a des contraintes d'ordre plus souple qu'une file de priorité utilisé par d'autres variantes, ce qui permet de réduire la complexité des insertions, mise

à jour et suppressions dans la structure de données. Cette variante est la plus efficace de toutes les variantes de Dijkstra.

2.3.6.2 Utilisation du Graphics Processing Unit (GPU)

Une autre méthode pour calculer un plus court chemin est d'abandonner l'exécution séquentielle avec un CPU et de se pencher sur une parallélisation de l'algorithme avec l'utilisation d'un GPU en utilisant le modèle de programmation CUDA. La difficulté vient de l'exécution de l'algorithme de Dijkstra qui est séquentiel et compliqué à paralléliser. L'algorithme a 2 boucles : une extérieure qui sélectionne un nœud à scanner, une intérieure qui itère sur les arcs et met à jour les distances.

En fonction de la boucle qui est parallélisée, les résultats sont très différents et la complexité du code change également. Quand la boucle intérieure est parallélisée, le temps de réponse diminue fortement quand le degré moyen des nœuds du graphe est élevé [Harish and Narayanan, 2007]. En revanche, quand la boucle extérieure est parallélisée, on peut paralléliser le calcul de tous les nœuds qui ont une distance à partir du point de départ qui est minimale [Crauser et al., 1998], cette méthode peut ensuite être améliorée pour fournir de meilleures performances [Ortega-Arranz et al., 2013]. Même des implémentations plus complexes [Davidson et al., 2014] ont du mal à avoir de bonnes performances sur les réseaux routiers comparé à un algorithme de Dijkstra séquentiel. Dans [Davidson et al., 2014], pour des graphes routiers, les méthodes utilisant des GPU ont un temps de réponse de 2 à 4 plus lent qu'un algorithme de Dijkstra. Cependant sur d'autres types de graphes, de gros gains apparaissent pouvant aller jusqu'à un facteur 64.

Un autre inconvénient de ces méthodes est leur sensibilité aux données. En effet, elle demande des données exprimées sous une forme simple (tableaux), ce qui n'est pas compatible avec un front de Pareto.

2.3.7 Performances des algorithmes de plus courts chemins

Nous allons comparer les temps de réponse des différents algorithmes dont nous avons parlé, sur des réseaux de transports routiers. Le but est de voir les gains fournis par les différents algorithmes, afin de mieux apprécier leurs avantages et leurs désavantages.

Nous pouvons voir dans le tableau 2.1 les performances des algorithmes pour le calcul d'itinéraire routier sur des requêtes générées de manière aléatoire. Les temps de réponse viennent d'expériences différentes, les résultats sont modifiés pour prendre en compte les différentes machines utilisées. Nous pouvons voir que l'algorithme de Dijkstra est le plus lent avec un temps de réponse autour de 2 s, ensuite vient l'algorithme de Dijkstra bidirectionnel avec un temps de réponse autour de 1 s, ce qui est compréhensible, car l'espace de recherche est divisé par 2. Ensuite, nous pouvons voir l'algorithme A* qui est plus lent que l'algorithme de Dijkstra en utilisant des bornes inférieures euclidiennes. Finalement, viennent les algorithmes avec du précalcul et nous pouvons voir que même de très faibles temps de précalcul peuvent très fortement améliorer les performances. L'algorithme ALT avec 0.15 heure de précalcul divise par quasiment 20 le temps de réponse de l'algorithme de Dijkstra, ensuite vient le CRP avec une heure de précalcul et un temps de réponse de 1 ms puis l'algorithme *Arc Flags* avec 0.3 h de précalcul et un temps de réponse de 0.4 ms. Tout en bas du tableau, nous pouvons voir les temps de précalcul et de réponse pour le calcul de la matrice entre tous les nœuds du graphe (l'algorithme PHAST), en plus des 145

Algorithme	Réseau	Million d'arcs	Pré-traitement (h)	Temps de réponse (ms)
Dijkstra	Europe	42,5	-	2195
Dijkstra bidirect.	Europe	42,5	-	1205
A*	Centre est des États-Unis	15,5	-	3987
ALT	Centre est des États-Unis	15,5	0,15	112
CRP	Europe	42,5	1	1,6
Arc Flags	Europe	42,5	0,33	0,4
PHAST	Europe	42,5	145	0,00006

TABLE 2.1 – Tableau récapitulatif des algorithmes pour le calcul d'un plus court chemin dans un réseau routier en se basant sur [Bast et al., 2016, Goldberg and Harrelson, 2005].

heures de précalcul, la taille nécessaire pour stocker la matrice est de plus d'un million de Go, ce qui la rend complètement inutilisable en pratique.

2.4 Algorithmes pour le calcul d'itinéraire pour les transports en commun

Le calcul d'itinéraire pour les transports en commun va utiliser une représentation assez similaire à celle pour les réseaux routiers, un graphe avec les nœuds qui représentent les arrêts de transport (les poteaux physiques), c'est-à-dire le seul endroit dans le réseau de transports en commun où un utilisateur peut passer d'un véhicule à un autre. Les arcs vont représenter le passage d'un nœud à un autre dans les moyens de transports. Contrairement au réseau routier, une composante temporelle très forte est ajoutée et un arc ne peut être traversé qu'à certains temps fixe. Par exemple : un bus ne passe à un arrêt qu'à 15h30 et 16h30.

Toutes ces informations temporelles et celles sur les véhicules sont stockées dans une structure de données appelée une table horaire. Son utilisation permet de calculer des itinéraires avec toutes les informations nécessaires à un usage par des utilisateurs, par exemple les numéros de lignes empruntés, les noms des directions affichés sur les véhicules, etc.

Les premiers algorithmes pour le calcul d'itinéraire pour les transports en commun sont très liés avec l'algorithme de Dijkstra. Récemment, un grand nombre d'algorithmes ont vu le jour. Cet essor a permis la création d'algorithmes extrêmement efficaces avec des idées très différentes les unes des autres.

Quand les algorithmes utilisent du prétraitement, la robustesse est extrêmement importante. Dans les réseaux routiers, un retard de quelques minutes se propage tout au long du trajet et peut augmenter l'heure d'arrivée d'un peu plus que le retard initial. Cependant pour les réseaux de transports en commun un retard de quelques minutes peut avoir un impact très fort sur l'heure d'arrivée, voir même rendre le trajet impossible si le retard fait rater le dernier moyen de transports vers l'arrivée.

Nous nous basons sur les travaux de [Bast et al., 2016] pour étudier l'état de l'art des calculs d'itinéraire pour les transports en commun.

2.4.1 Table horaire

Nous allons utiliser la même représentation que celle utilisée dans [Dibbelt et al., 2018]. Une table horaire, appelé *timetable* par les anglo-saxons, représente pour une journée spécifique, les arrêts et horaires de passage de l'ensemble des véhicules, ainsi que les chemins piétons. Une table horaire est définie par un quadruplet (S, C, T, F) d'arrêts S, de trajets T, de connexions C et de chemins piétons F :

- Un arrêt est une position en dehors d'un véhicule où un utilisateur peut attendre. C'est à un arrêt et seulement à un arrêt qu'un utilisateur peut descendre ou monter dans un véhicule.
- Un trajet est un véhicule qui passe par une suite d'arrêts à des horaires fixes. Formellement, un trajet est un véhicule unique qui va partir d'un arrêt de départ et qui va s'arrêter à un ensemble d'arrêts jusqu'à un terminus. Les départs et les arrivées ont lieu à des horaires fixes.
- Une connexion est un véhicule qui va d'un arrêt à un autre sans arrêts intermédiaires, c'est une sous-partie d'un trajet. Formellement, c'est un quintuplet $(c_{dep\_stop}, c_{arr\_stop}, c_{dep\_time}, c_{arr\_time}, c_{trip})$ avec pour attributs respectivement un arrêt de départ, un arrêt d'arrivée, une heure de départ, une heure d'arrivée et un identifiant de trajet. Chaque connexion doit respecter deux conditions : une connexion ne peut pas être une boucle ($c_{dep\_stop} \neq c_{arr\_stop}$) et une connexion ne peut pas remonter dans le temps ($c_{dep\_time} < c_{arr\_time}$).
- Un chemin piéton permet de modéliser les transferts, c'est-à-dire comment un utilisateur peut passer d'un trajet à un autre. Les chemins piétons ne sont ni des trajets ni des connexions. Formellement, un chemin piéton est un triplet $(f_{dep\_stop}, f_{arr\_stop}, f_{dur})$, avec pour attributs respectivement un arrêt de départ, un arrêt d'arrivée et une durée.

Passer d'une connexion c à une connexion c', tel que $c_{trip} \neq c'_{trip}$, n'est faisable que si et seulement si :

- un chemin piéton de c_{arr_stop} vers c'_{dep_stop} existe.
- la condition $c'_{dep\_time} - c_{arr\_time} > f_{dur}$ est vrai, c'est-à-dire qu'un utilisateur puisse descendre à l'arrêt d'arrivée de la connexion c et aller à l'arrêt de départ de la connexion c' avant son heure de départ en empruntant le chemin piéton f.

Une boucle est créée pour chaque arrêt afin de permettre à un utilisateur de descendre et de prendre un autre trajet qui passe par ce même arrêt.

Exemple Nous illustrons ces notions pour le réseau de transports en commun de Nice.

Les trajets peuvent être des trains, des bus, des trams, des ferrys ou n'importe quel autre mode de transport qui a des heures de départs et d'arrivées fixes. Soit t un trajet de la ligne 1 du tram qui va en direction de Pasteur avec 08/09/2021 10h00 comme heure de départ. Un trajet sans sa date de départ n'est pas un identifiant unique, car ce trajet existe tous les jours de la semaine.

Si nous prenons 3 arrêts consécutifs de t, Gare Thiers, Nice étoile et Masséna. Il existe une connexion valide avec Gare Thiers comme arrêt de départ, Nice étoile comme arrêt d'arrivée et t comme trajet. Une autre connexion valide a comme arrêt de départ Nice étoile, Masséna comme arrêt d'arrivée et t comme trajet. Cependant, la connexion avec Gare Thiers comme arrêt de départ,

Masséna comme arrêt d'arrivée et t comme trajet, n'est pas une connexion valide, car il existe un arrêt intermédiaire entre Gare Thiers et Masséna.

Soit Jean Médécin l'arrêt de la ligne 2 de tram à Nice. Il existe un chemin piéton entre l'arrêt Jean Médécin et Nice étoile pour permettre à un utilisateur de passer de la ligne 1 à la ligne 2 du tram.

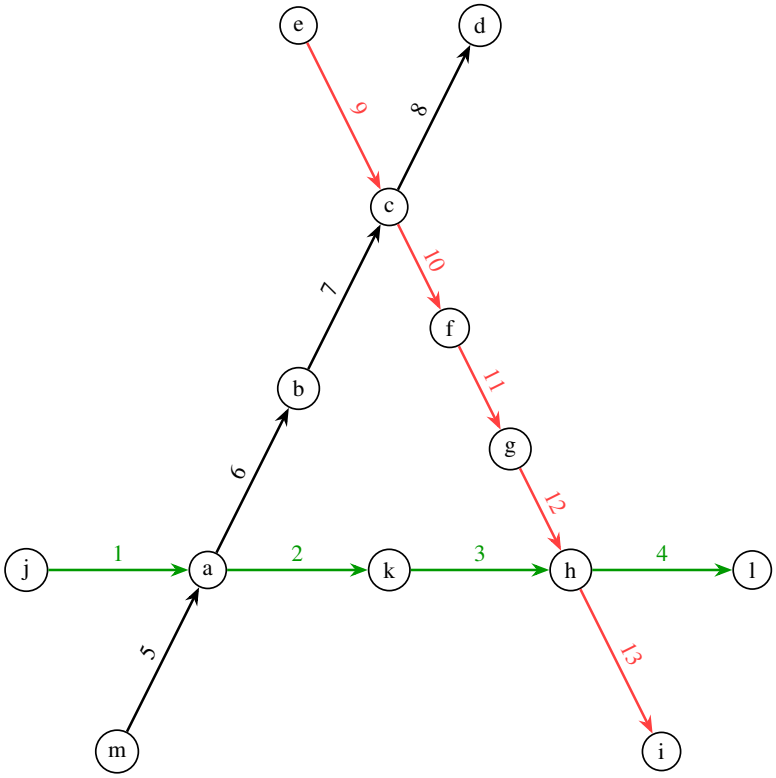


FIGURE 2.6 – Exemple sur un petit réseau.

Exemple. La figure 2.6 est un exemple d'un petit réseau qui sera appliquer sur pour chaque algorithme.

Les tableau 2.2, tableau 2.3 et tableau 2.4 décrivent respectivement les connexions des lignes verte, noire et rouge dans l'ordre chronologique.

	Arc	Heure de départ	Heure d'arrivée
1	j $\longrightarrow$ a	9h40	9h50
2	a $\longrightarrow$ k	9h50	10h00
3	k $\longrightarrow$ h	10h00	10h10
4	h $\longrightarrow$ l	10h10	10h20

TABLE 2.2 – Détails des connexions de la ligne verte

	Arc	Heure de départ	Heure d'arrivée
5	m ⟶ a	9h50	10h00
6	a ⟶ b	10h00	10h10
7	b ⟶ c	10h10	10h30
8	c ⟶ d	10h20	10h30

TABLE 2.3 – Détails des connexions de la ligne noire

	Arc	Heure de départ	Heure d'arrivée
9	e ⟶ c	10h20	10h30
10	c ⟶ f	10h30	10h40
11	f ⟶ g	10h40	10h50
12	g ⟶ h	10h50	11h00
13	h ⟶ i	11h00	11h10

TABLE 2.4 – Détails des connexions de la ligne rouge

2.4.2 Itinéraire

Un itinéraire décrit comment un passager peut voyager à travers un réseau de transports en commun. Il est composé de ce qu'appellent les anglo-saxons des *legs* qui sont une paire de connexions $(l^i_{enter}, l^i_{exit})$ du même trajet. l^i_{enter} doit apparaître avant l^i_{exit} dans le trajet. On peut avoir $l^i_{enter} = l^i_{exit}$ si le *leg* n'a qu'une connexion.

Formellement, un itinéraire est composé alternativement de connexions et de $f^0, l^0, f^1, l^1, \cdots, f^k, l^k$. Un itinéraire doit commencer et finir par un chemin piéton, qui peut être une boucle.

2.4.3 Modélisation

Les réseaux de transports en commun sont dépendants du temps en raison du fait que les véhicules ont un temps de passage à un arrêt. Ainsi un arc n'est plus seulement défini par une paire de nœuds, mais également par une paire d'heures, une heure de départ et une heure d'arrivée, c'est-à-dire $a \in A, u, v \in V, \tau_0, \tau_1 \in M \; t.q. \; a = (u, v, \tau_0, \tau_1)$. Ce qui implique qu'un arc ne peut être traversé que quand une condition est remplie : un utilisateur doit être à l'arrêt u à un temps $\tau \leq \tau_0$ pour pouvoir emprunter l'arc a et ainsi se retrouver à l'arrêt v au temps τ_1.

L'ajout de la temporalité va avoir de fortes implications dans la modélisation des réseaux [Müller-Hannemann et al., 2007] et a un impact sur les algorithmes, tant sur les performances que sur leur élaboration. Certaines modélisations vont changer la structure du graphe pour y intégrer les contraintes de temps, en ajoutant des nœuds et des arcs ou bien en ajoutant une fonction sur les arcs.

2.4.3.1 Graphe expansé dans le temps (time expanded)

Une des premières modélisations par [Pallottino and Scutella, 1998, Schulz et al., 2000], se base sur le fait que les événements de transport passent à un moment discret à un nœud. Nous pouvons donc ajouter une dimension temporelle au graphe pour chaque nœud. La modélisation va

dérouler les nœuds par rapport au temps : un nœud sera dupliqué pour chaque départ ou arrivée d'un véhicule passant par ce même nœud. Tous les événements de départ et d'arrivée desservis par un même véhicule seront reliés par un arc de « voyage », représenté par des arcs horizontaux dans la figure 2.7. Cela implique de créer de nouveaux arcs pour pouvoir utiliser d'autres événements (ou véhicule) passant par un même arrêt physique qui est devenu plusieurs nœuds dans le graphe. Des arcs verticaux, dits de « transferts », sont créés entre les nœuds du même arrêt physique de manière chronologique, représenté par des arcs verticaux dans la figure 2.7.

Le défaut de cette modélisation est sa taille. Un nœud se verra dupliquer autant de fois qu'un véhicule passe par ce même nœud, ce qui augmente considérablement le nombre de nœuds et d'arcs du graphe.

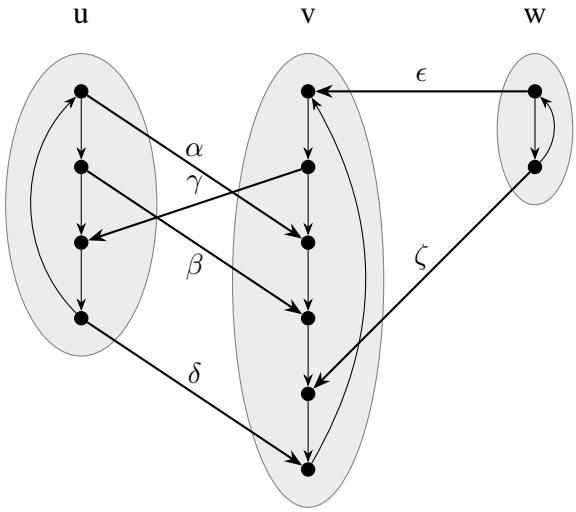


FIGURE 2.7 – Représentation d'un graphe expansé dans le temps.

La figure 2.7 montre un graphe expansé dans le temps avec 3 arrêts de transports en commun u, v et w qui sont transformés en de multiples nœuds dans le graphe. Ici 4 nœuds pour u, 6 nœuds pour v et 2 nœuds pour w. De plus, nous pouvons voir que le nombre d'arcs entre les arrêts sont multiples. Chaque arc représente un unique véhicule qui va aller d'un arrêt à un autre.

Une variation de ce modèle [Müller-Hannemann and Schnee, 2007, Pyrga et al., 2008, Pyrga et al., 2004] permet l'ajout de temps de transfert minimum qui sont utilisés dans un itinéraire pour éviter de calculer des itinéraires impossibles en pratique. Ce modèle plus réaliste ajoute de nouveaux nœuds ce qui augmente encore une fois la taille du graphe.

2.4.3.2 Graphe dépendant du temps (time dependent)

Une autre modélisation par [Brodal and Jacob, 2004] va cette fois-ci garder un nœud pour chaque arrêt, et ajouter pour chaque connexion élémentaire entre un nœud u et un nœud v, un arc (u, v) avec une heure de départ et une heure d'arrivée. Quand un utilisateur est au nœud u et veut atteindre le nœud v, alors plusieurs arcs (u, v) peuvent exister avec des heures de départ et d'arrivée différentes. Pour savoir laquelle emprunter, ou même si nous pouvons en emprunter une, nous utilisons une fonction qui prend en entrée une heure de départ et qui renvoie une durée de voyage.

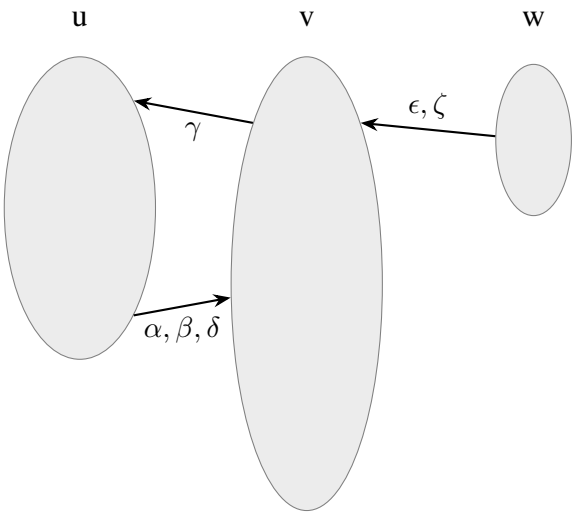


FIGURE 2.8 – Représentation d'un graphe dépendant du temps.

La figure 2.8 montre un graphe dépendant du temps, ici les 3 arrêts de transports u, v et w sont représentés par 3 nœuds et nous n'avons qu'un seul arc entre les nœuds. Cependant chaque arc peut représenter plusieurs évènements. L'arc entre u et v représente les évènements α, β et δ. Quand l'arc doit être parcouru, l'évènement est choisi en fonction de l'heure d'arrivée à l'arrêt et l'heure de départ des évènements de l'arc.

Une variation de ce modèle [Pyrga et al., 2008] permet une nouvelle fois l'ajout de temps de transfert minimum. Une autre variation [Disser et al., 2008] va permettre de connecter les nœuds proches par des chemins piétons.

2.4.3.3 Modélisation par fréquence

Une autre modélisation par [Bast and Storandt, 2014] se base sur le fait que, dans les réseaux de transports, les véhicules desservent les arrêts à intervalles réguliers. Par exemple, pendant les heures de pointe, un tram va passer toutes les 3 minutes et pendant les autres périodes un tram ne passera que toutes les 6 minutes. La différence avec la modélisation précédente est que nous n'allons pas stocker chaque heure de départ ou d'arrivée d'un évènement, mais seulement un tuple qui va contenir une heure de départ, une heure d'arrivée et une fréquence. Toutes les heures de départ peuvent ainsi être reconstruites grâce au tuple. Cette modélisation permet de compresser les données utilisées par un calculateur, ce qui se traduit aussi bien par un gain en espace qu'un gain en temps sur certains algorithmes.

2.4.4 Algorithme de Dijkstra

Tout comme pour le calcul d'itinéraire routier, le premier algorithme pour le calcul d'itinéraire pour les transports en commun est une variante de l'algorithme de Dijkstra. La différence majeure est la modélisation utilisée pour représenter le graphe. L'algorithme reste très similaire avec l'utilisation de la file de priorité et le déroulement classique de l'algorithme.

2.4.4.1 Time Expanded Dijkstra

Cette variante de l'algorithme de Dijkstra se base sur la modélisation expansée dans le temps et s'appelle *Time Expanded Dijkstra* (TED) [Schulz et al., 2000].

Le calcul d'un itinéraire entre un arrêt s et t à un temps τ commence par l'initialisation du nœud de départ qui correspond au premier évènement après τ à l'arrêt s. L'heure d'arrivée au plus tôt à t est celle de l'évènement qui permet d'atteindre un des nœuds d'arrivée. Le déroulement de l'algorithme reste le même. Cependant le nombre de nœuds et d'arcs atteints est beaucoup plus conséquent que pour un Dijkstra classique, ce qui impacte fortement le temps d'exécution de l'algorithme, voir tableau 2.1.

Exemple. L'algorithme TED est appliqué au réseau de la figure 2.6.

Les arrêts qui ont plusieurs lignes en commun sont différenciés par une couleur. L'arrêt c de la ligne noir est différencié de l'arrêt c de la ligne rouge.

Le tableau 2.5 montre le déroulement de l'algorithme TED pour l'exemple pour une recherche d'itinéraire entre l'arrêt a et l'arrêt h avec une heure de départ à 10h00.

2.4.4.2 Time Dependent Dijkstra

Cette variante de l'algorithme de Dijkstra se base sur la modélisation dépendante du temps et s'appelle *Time Dependent Dijkstra* (TDD) [Cooke and Halsey, 1966, Dreyfus, 1969].

Cette fois-ci, pour un itinéraire entre s et t à un temps τ, chaque nœud correspond à un arrêt de transport. La difficulté est de savoir quel arc emprunté entre deux nœuds u et v. Il s'agit de prendre celle avec une heure de départ supérieure à $\tau + d(s, u)$, c'est-à-dire supérieure à l'heure de départ plus la plus courte durée pour rejoindre s à partir de u donc l'heure d'arrivée au plus tôt pour u. Le déroulement de l'algorithme est très similaire à celui de l'algorithme de Dijkstra routier et le résultat est stocké pour le nœud t.

L'algorithme TDD est plus rapide que l'algorithme TED, voir tableau 2.1. Cependant tous les deux utilisent une file de priorité qui est très coûteuse à maintenir. Récemment de nouveaux algorithmes qui laissent de côté la file de priorité, et pour certains même la représentation en graphe, ont été développés et ont de bien meilleures performances.

Exemple. L'algorithme TDD est appliqué au réseau de la figure 2.6.

Le tableau 2.6 montre le déroulement de l'algorithme TED pour l'exemple pour une recherche d'itinéraire entre l'arrêt a et l'arrêt h avec une heure de départ à 10h00.

2.4.5 Connection Scan Algorithm

Connection Scan Algorithm (CSA) [Dibbelt et al., 2018] est un algorithme qui n'utilise pas de graphe, mais une liste de toutes les connexions triées par heure de départ, ce qui en fait un algorithme réservé aux transports en commun. Le problème résolu par l'algorithme CSA est celui de l'arrivée au plus tôt. De multiples variantes existent pour résoudre d'autres problèmes.

L'algorithme se déroule de la manière suivante (voir algorithme 2.3 pour une variante optimisée qui prend en compte les chemins piétons) : une heure d'arrivée au plus tôt provisoire est stockée pour chaque arrêt et l'heure d'arrivée au plus tôt pour l'arrêt de départ s est initialisée à τ. Ensuite, l'algorithme itère sur toutes les connexions, si celle-ci est « utilisable », c'est-à-dire que l'heure d'arrivée au plus tôt pour l'arrêt de départ est inférieure à l'heure de départ de la connexion, alors l'heure d'arrivée au plus tôt pour l'arrêt d'arrivée de la connexion est mise à

	à a	à b	à c	à c	à d	à e	à f	à g	à h	à h	à i	à j	à k	à l
initialisation	10h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
a (10h00)	10h00	10h10	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
b (10h10)		10h10	10h20	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
c (10h20)			10h20	10h20	10h30	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
c (10h20)				10h20	10h30	$+\infty$	10h40	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
d (10h30)					10h30	$+\infty$	10h40	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
f (10h40)						$+\infty$	10h40	10h50	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
g (10h50)						$+\infty$		10h50	11h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
h (11h00)						$+\infty$			11h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Notes : Quand une heure est soulignée pour un nœud, cela signifie que le nœud vient d'être atteint et donc que l'heure soulignée est l'heure d'arrivée au plus tôt pour ce nœud.

TABLE 2.5 – Déroulé de l'algorithme TED sur l'exemple

	à a	à b	à c	à d	à e	à f	à g	à h	à i	à j	à k	à l
initialisation	10h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
a (10h00)	<u>10h00</u>	10h10	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
b (10h10)		<u>10h10</u>	10h20	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
c (10h20)			<u>10h20</u>	10h30	$+\infty$	10h40	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
d (10h30)				<u>10h30</u>	$+\infty$	10h40	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
f (10h40)					$+\infty$	<u>10h40</u>	10h50	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
g (10h50)					$+\infty$		<u>10h50</u>	11h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$
h (11h00)					$+\infty$			<u>11h00</u>	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Notes : Quand une heure est soulignée pour un nœud, cela signifie que le nœud vient d'être atteint et donc que l'heure soulignée est l'heure d'arrivée au plus tôt pour ce nœud.

TABLE 2.6 – Déroulé de l'algorithme TDD sur l'exemple

Algorithme 2.3 Algorithme CSA

function CSA(T, s, t, τ_s)
  for all stops $s \in T$ **do**
    $S[s] \leftarrow +\infty$
  end for
  for all trips $t \in T$ **do**
    reset $T[t]$
  end for
  for all footpaths f from s **do**
    $S[s] \leftarrow \tau_s + f_{dur}$
  end for
  for all connections c by increasing c_{dep_time} **do**
    if $T[c_{trip}]$ is set or $S[c_{dep\_stop}] \leq c_{dep\_time}$ **then**
      raise $T[c_{trip}]$
      for all footpaths f from c_{arr_stop} **do**
        $S[f_{arr\_stop}] \leftarrow \min\{S[f_{arr\_stop}], c_{arr\_time} + f_{dur}\}$
      end for
    end if
  end for
end function

jour, et de même pour chaque chemin piétons partant de l'arrêt d'arrivée de la connexion. Après avoir itéré sur chaque connexion, l'heure d'arrivée provisoire stockée à l'arrêt d'arrivée est l'heure d'arrivée au plus tôt pour l'itinéraire allant de s à t partant à l'heure τ.

Plusieurs optimisations existent, la première consiste à scanner uniquement les connexions qui ont une heure de départ supérieure à l'heure de départ demandée dans le calcul d'arrivée au plus tôt. La seconde consiste à arrêter l'algorithme CSA quand les connexions scannées ont une heure de départ supérieure à l'heure d'arrivée au plus tôt stockée à l'arrêt d'arrivée. La dernière va permettre de ne pas regarder les chemins piétons partant de l'arrêt d'arrivée de la connexion si l'heure d'arrivée au plus tôt pour l'arrêt d'arrivée de la connexion n'est pas améliorée. La troisième amélioration nécessite que le graphe des chemins piétons soit transitivement fermé, c'est-à-dire que s'il existe un chemin piéton entre x et y et un chemin piéton entre y et z alors il existe un chemin piéton entre x et z. Si on utilise un modèle de chemin piéton avec une distance de marche à pied maximum, alors le contre-exemple suivant peut se produire (voir figure 2.9) : soient A, B et C trois arrêts, soient un chemin piéton entre A et B d'une durée de 7 minutes et un chemin piéton entre B et C d'une durée de 7 minutes, soient une connexion qui arrive qui arrive à A à 8h00 et une connexion qui arrive à B à 9h00 et les deux connexions sont utilisables. Avec la dernière amélioration, quand la connexion qui arrive à A est scannée, l'heure d'arrivée au plus tôt à A est mise à jour à 8h00 et donc l'heure d'arrivée à B est également mise à jour à 8h07, quand la connexion qui arrive à B doit être scanné, l'amélioration nous dit que l'heure d'arrivée au plus tôt à l'arrivée n'étant pas améliorée alors nous n'avons pas besoin de mettre à jour l'heure d'arrivée des chemins piétons, ce qui veut dire que C n'a pas d'heure d'arrivée alors que la connexion qui arrive à B est utilisable.

Le temps de réponse de l'algorithme CSA est beaucoup plus rapide que TED ou TDD qui maintiennent une file de priorité. Ainsi le nombre d'opérations faites par l'algorithme CSA est

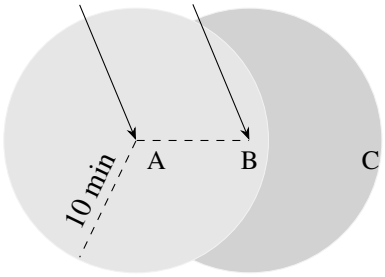


FIGURE 2.9 – Contre-exemple pour l'optimisation du CSA avec un modèle de chemin piétons avec une distance maximale.

supérieur au nombre d'opérations faites par TDD (ou TED), mais les opérations de file de priorité sont plus complexes et coûteuses que celle effectuées pour l'algorithme CSA (des comparaisons et des minimums).

Exemple. L'algorithme CSA est appliqué au réseau de la figure 2.6.

Les connexions sont triées par heure de départ et sont scannées dans cet ordre. Le tableau 2.7 montre le déroulement de l'algorithme CSA pour l'exemple pour une recherche d'itinéraire entre l'arrêt a et l'arrêt h avec une heure de départ à 10h00.

Variante de l'algorithme CSA Une variante pour améliorer le temps de réponse pour le problème d'arrivée au plus tôt existe, appelé *Accelerated Connection Scan Algorithm* (ACSA) [Strasser and Wagner, 2014]. L'algorithme partitionne le graphe en plusieurs niveaux et l'idée générale est d'éviter de regarder les bus ruraux qui ne sont ni autour de l'arrêt de départ ni autour de l'arrêt d'arrivée, afin de ne garder qu'un sous-ensemble des connexions entre des villes. Cela permet de réduire le nombre de connexions scannées et donc le temps de réponse. On observe de bonnes améliorations des performances pour les réseaux nationaux, cependant on remarque qu'il n'améliore pas les performances pour de grands réseaux métropolitains. Enfin l'algorithme est complexe à implémenter.

2.4.6 Transfer Patterns

Une autre possibilité pour améliorer les performances d'un algorithme est l'utilisation de précalculs, comme le *Transfer Patterns* [Bast et al., 2010]. Le principe est d'utiliser des « schémas de transfert », qui sont l'ensemble des transferts optimaux entre deux arrêts du réseau de transports en commun. Par exemple, entre Nice et Lyon, il existe deux trajets optimaux un passant par Marseille et un par Aix-en-Provence. Ensuite, une fois que les trajets optimaux entre le point de départ et le point d'arrivée sont trouvés, un graphe de requête est créé et peut être parcouru très rapidement grâce à sa taille extrêmement réduite comparée au graphe du réseau complet.

La phase de précalcul lance une recherche multicritère à partir de toutes les paires de points afin d'en sortir les connexions optimales qui forment des « schémas de transfert » et qui sont stockées dans un graphe dirigé acyclique pour chaque nœud. Cette phase de précalcul est très longue car elle revient à calculer un chemin d'arrivée au plus tôt multiobjectif à partir de tous les nœuds vers tous les autres nœuds. Une amélioration existe pour calculer seulement une sous-partie des « schémas de transfert » et ensuite recréer les « schémas de transfert » complets pendant une requête.

	à a	à b	à c	à d	à e	à f	à g	à h	à i	à j	à k	à l
initialisation	10h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(j, a, 9\text{h}40, 9\text{h}50)$	10h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(a, k, 9\text{h}50, 10\text{h}00)$	10h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(m, a, 9\text{h}50, 10\text{h}00)$	10h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(k, h, 10\text{h}00, 10\text{h}10)$	10h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(a, b, 10\text{h}00, 10\text{h}10)$	10h00	10h10	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(h, l, 10\text{h}00, 10\text{h}10)$	10h00	10h10	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(b, c, 10\text{h}10, 10\text{h}20)$		10h10	10h20	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(c, d, 10\text{h}20, 10\text{h}30)$			10h20	10h30	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(e, c, 10\text{h}20, 10\text{h}30)$			10h20	10h30	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(c, f, 10\text{h}30, 10\text{h}40)$				10h30	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(f, g, 10\text{h}40, 10\text{h}50)$					$+\infty$	10h40	10h50	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(g, h, 10\text{h}50, 11\text{h}00)$					$+\infty$		10h50	11h00	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$(h, i, 11\text{h}00, 11\text{h}10)$					$+\infty$			11h00	11h10	$+\infty$	$+\infty$	$+\infty$

TABLE 2.7 – Déroulé de l'algorithme CSA sur l'exemple

Le précalcul des « schémas de transferts » est fait une unique fois avec les heures de passages théoriques des moyens de transport. Si par exemple un train a du retard ou est annulé alors les « schémas de transferts » calculés pendant la phase de prétraitements peuvent être maintenant faux. En effet, supposé qu'un réseau de transports en commun ne change pas est une hypothèse peu réaliste qui a un impact sur les résultats calculés par l'algorithme.

Le temps de réponse est extrêmement faible cependant le temps de précalcul est énorme, en moyenne plusieurs jours, comme le montre le tableau 2.9.

2.4.7 Public Transit Labeling

Un algorithme avec du précalcul plus récent, avec de bien meilleures performances que le *Transfer Patterns* pour un temps de précalcul beaucoup plus rapide, est *Public Transit Labeling* [Delling et al., 2015a]. L'algorithme se base sur le principe de *2-hop labeling*, c'est-à-dire qu'on associe pour chaque nœud v des *forward labels* $L_f(v)$ et des *backward labels* $L_b(v)$.

L'application aux transports en commun se fait avec l'utilisation d'un graphe expansé dans le temps, qui est plus facile pour adapter le *2-hop labelling*. Dans un graphe expansé dans le temps, les arcs ont des coûts comparés au graphe dépendant du temps où les arcs ont une fonction qui renvoie une durée. Le graphe est généré de la manière la plus simple possible : un arrêt physique est représenté par de multiples nœuds, un pour chaque connexion passant par cet arrêt et ensuite des arcs sont ajoutés entre tous les nœuds pour représenter les chemins piétons et la possibilité d'attendre à l'arrêt la prochaine connexion. Ensuite n'importe quel algorithme de génération de *labels* peut être utilisé. Pour le *Public Transit Labeling* l'algorithme utilisé est RXL [Delling et al., 2014], ce qui va créer des *forward labels* et des *backward labels* pour chaque nœud.

Pour le calcul d'un chemin d'arrivée au plus tôt entre s et t avec une heure de départ τ_0, le but est de trouver le premier nœud e_s de s avec une heure de départ supérieure ou égale à τ_0 puis de trouver le premier nœud de t qui est atteignable à partir de e_s. Ce qui nous donne l'itinéraire entre s et t avec l'heure d'arrivée au plus tôt.

De la même manière que pour l'algorithme *Transfer Patterns*, l'algorithme *Public Transit Labeling* ne prend pas en compte les possibles changements du réseau (retards, annulation ...), ce qui force à refaire le prétraitement ou renvoyer des résultats possiblement incorrects.

2.4.8 Performances des algorithmes de calcul d'itinéraire pour les transports en commun

De nombreux algorithmes existent pour résoudre le problème de plus court chemin dans un réseau de transports en commun. Au fur et à mesure des années, les algorithmes sont devenus de plus en plus performants, en utilisant différentes représentations et en empruntant les avancées faites par le calcul d'itinéraire routier. Le tableau suivant récapitule les performances et les temps de précalcul des différents algorithmes présentés dans cette partie.

Nous pouvons voir dans le tableau 2.8 les performances des algorithmes pour le calcul d'itinéraire monobjectif, c'est-à-dire l'arrivée au plus tôt, sur des requêtes générées de manière aléatoire. Les temps de réponse viennent d'expériences différentes, les résultats sont modifiés pour prendre en compte les différentes machines utilisées. Nous remarquons que les algorithmes TED et TDD qui utilisent une représentation avec un graphe sont les plus lents avec un temps d'exécution de 44 ms et 11 ms, ensuite vient le CSA qui laisse de côté la représentation avec un graphe avec un temps d'exécution beaucoup plus rapide aux alentours de 2 ms.

Algorithme	Réseau	Million de connexions	Pré-traitement (h)	Temps de réponse (ms)
Time Expanded Dijkstra	Londres	5,1	-	44,8
Time Dependent Dijkstra	Londres	5,1	-	11,0
RAPTOR	-	-	-	-
Connection Scan Algorithm	Londres	5,1	-	2,0
Trip Based Routing	-	-	-	-
Transfer Patterns	Allemagne	13,9	249	0,2
Public Transit Labeling	Londres	5,1	0.9	0,0019

TABLE 2.8 – Tableau récapitulatif des algorithmes pour le calcul d'itinéraire pour les transports en commun monobjectif en se basant sur [Bast et al., 2016, Delling et al., 2015a].

Ensuite viennent les algorithmes avec du précalcul, *Transfer Patterns* avec un temps de précalcul extrêmement long, a un temps de réponse 10 fois plus rapide que CSA, de 0.2 ms. L'algorithme *Public Transit Labeling* quant à lui a un temps de précalcul moindre, aux alentours d'une heure, et a un temps de réponse extrêmement rapide de 2 μs. Bien que les temps de réponse soient très rapides pour les algorithmes avec du prétraitement, l'algorithme *Transfer Patterns* et l'algorithme *Public Transit Labeling* ne peuvent pas prendre en compte des possibles changements du réseau comme les retards ou annulation. De plus ces changements arrivent en temps réels, jusqu'à plusieurs fois par minute, ce qui rend la stratégie de refaire les prétraitements à chaque changement extrêmement coûteuse.

2.5 Algorithmes pour le calcul d'itinéraire pour les transports en commun multiobjectif

Le calcul d'itinéraire pour les transports en commun ne se résume pas à fournir un unique résultat pour une requête utilisateur. Il faut en fournir de multiples et s'assurer de leur qualité, ce qui pour nous signifie l'utilisation d'un front de Pareto avec plusieurs critères : l'heure d'arrivée, l'heure de départ qui en conjonction avec l'heure d'arrivée permet de fournir des résultats intéressants sur une plage horaire, le nombre de transferts et la distance de marche. Les deux derniers critères sont très importants, car une grande majorité des préférences des utilisateurs peuvent être exprimées avec un nombre de transferts et une distance de marche. Ainsi certains utilisateurs préfèrent arriver plus tôt au prix de multiples transferts alors qu'à l'inverse certains utilisateurs préfèrent rester assis dans un train plus longtemps même si l'heure d'arrivée sera plus tardive. D'autres critères existent, le prix par exemple, mais son calcul est complexe ce qui peut avoir un impact très fort sur les performances, nous pouvons avoir des prix différents en fonction des modes empruntés, des zones de départ ou d'arrivée, des distances parcourues, des profils des utilisateurs et beaucoup d'autres possibilités. Cependant, le prix peut être écarté des critères du calcul d'itinéraire, car les usagers des transports en commun ont très souvent un abonnement et ne se soucient donc pas du prix de leurs trajets. Plusieurs algorithmes existent pour répondre au problème du cal-

cul d'itinéraire pour les transports en commun multiobjectif, qui sont pour la plupart des variantes d'algorithmes déjà existants.

2.5.1 Dijkstra

L'algorithme *Multicriteria Label-Setting* (MLS) [Müller-Hannemann and Schnee, 2007] est une extension de l'algorithme TED pour résoudre le problème multicritère. L'algorithme MLS maintient un front de Pareto pour chaque arrêt, chaque élément dans le front de Pareto est appelé une étiquette et représenté par un tuple, où chaque élément du tuple représente un critère à optimiser. La file de priorité maintient des étiquettes plutôt que des arrêts. À chaque itération, l'étiquette minimale est extraite (en utilisant un ordre lexicographique par exemple) et les arcs sortants de l'arrêt sont scannés. Le coût de l'arc est ajouté à l'étiquette puis cette nouvelle étiquette est insérée dans le front de Pareto de l'arrêt d'arrivée de l'arc et dans la file de priorité. L'algorithme MLS peut être optimisé pour réduire le nombre d'opérations de la file de priorité et ainsi diminuer le temps de réponse [Disser et al., 2008].

Des variantes du *Time Dependent Dijkstra* résolvent le problème d'arrivée au plus tôt par plage horaire. La première [Dean, 1999, Nachtigall, 1995] maintient une fonction de temps de voyage à chaque arrêt u, qui représente le temps de voyage optimal de s à u pour la fenêtre de temps considéré. Quand l'algorithme relaxe un arc (u, v), il « lie » la fonction de temps de voyage de u avec la fonction de coût dépendante du temps associé à v. L'algorithme n'est plus un algorithme *label-setting*, car la fonction de temps de voyage ne peut pas être complètement triée. Ce qui fait que l'algorithme peut réinsérer un arrêt dans la file de priorité quand il trouve un itinéraire qui améliore la fonction de temps de voyage d'un arrêt déjà scanné, l'algorithme n'est plus glouton. Une deuxième variante est l'algorithme de *Self Pruning Connection Setting* (SPCS) [Delling et al., 2012] qui se base sur le fait que n'importe quel itinéraire optimal de s à t doit commencer par un des trajets qui passe par s. Ainsi, l'algorithme lance pour chaque trajet un algorithme de Dijkstra à partir de s avec son heure de départ. SPCS lance ses algorithmes d'arrivée au plus tôt en utilisant une file de priorité partagée avec les valeurs triées par heure de départ. Quand l'algorithme scanne un arrêt u, il vérifie si u a déjà été scanné pour un autre trajet avec une heure de départ plus tardive, dans ce cas u est élagué. De plus, SPCS peut être parallélisé.

2.5.2 Round Based Public Transit Optimized Router

L'algorithme *Round Based Public Transit Optimized Router* (RAPTOR) [Delling et al., 2015b] n'utilise pas une représentation avec un graphe, mais représente le réseau de transports en commun en utilisant un tableau de trajet et de parcours, qui sont l'ensemble des trajets qui partagent les mêmes arrêts. L'algorithme parcourt exclusivement les tableaux de trajets et de parcours ainsi que leurs horaires, ce qui est en fait un algorithme exclusivement réservé aux transports en commun.

L'algorithme se déroule de la manière suivante : il itère un nombre fixe de fois, appelé *rounds* et ici défini par K, et pour chaque nœud l'algorithme stocke une heure d'arrivée pour chaque *round*. L'algorithme est initialisé avec $k = 0$ et puis marque tous les trajets passant par l'arrêt de départ s, ensuite :

- On itère sur tous les trajets marqués pour mettre à jour l'heure d'arrivée au plus tôt stockée pour le $k$ème *round* des arrêts après le point de départ du trajet.
- En même temps, quand on itère sur un arrêt qui a un parcours que l'algorithme n'a jamais touché alors ce parcours est marqué. Ensuite, l'algorithme itère sur les trajets des routes

marqués afin de trouver celui qui part juste après l'heure d'arrivée au plus tôt stocké pour le point de départ du parcours, puis ajoute ce trajet à la liste des trajets pour le prochain *round*.

– Si k est égal à K alors on renvoie les heures d'arrivée au plus tôt, sinon les horaires d'arrivée au plus tôt du round k sont copiés dans ceux du round $k+1$ et ensuite on incrémente k.

Exemple. L'algorithme RAPTOR est appliqué au réseau de la figure 2.6.

Pour une recherche d'itinéraire entre l'arrêt a et l'arrêt h avec une heure de départ à 10h00, RAPTOR commence par trouver les lignes passant par a, qui sont la ligne noire et la ligne verte. L'heure de départ à l'arrêt a est à 10h00, la ligne verte part de l'arrêt a à 9h50 ce qui implique que RAPTOR ne peut pas la scanner. Pour la ligne noire, l'heure de départ à l'arrêt a est à 10h00 ce qui fait que RAPTOR scanne la ligne noire.

L'heure d'arrivée pour le premier *round* pour les arrêts de la ligne noire après a, donc les arrêts b, c et d, sont mis à jour avec comme valeur respective 10h10, 10h20 et 10h30. Toutes les lignes accessibles par RAPTOR sans aucun transfert sont scannées et l'on peut mettre à jour les horaires pour le *round* 2.

La ligne rouge est la seule ligne que RAPTOR peut scanner parmi les nouveaux arrêts atteints. L'heure d'arrivée au plus tôt au premier *round* de c est de 10h20 et la ligne rouge part de c à 10h30, ce qui implique que RAPTOR peut scanner cette ligne. L'heure d'arrivée pour le deuxième *round* pour les arrêts de la ligne rouge après le c, donc les arrêts f, g, h et i, sont mis à jour avec comme valeur respective 10h30, 10h40, 10h50 et 11h00. Toutes les lignes accessibles par RAPTOR en un transfert sont scannées et l'on peut passer au *round* 3, mais aucune autre ligne ne peut être atteinte avec les nouveaux arrêts mis à jour, nous pouvons donc regarder si nous avons un résultat.

Nous pouvons voir qu'il n'y a pas d'itinéraire entre l'arrêt a et l'arrêt h en 0 transfert, mais nous avons bien un résultat avec un transfert qui arrive à 11h00.

Variante de l'algorithme RAPTOR Une variante améliore le temps de réponse de l'algorithme RAPTOR pour des réseaux de transports trop grands [Delling et al., 2017]. L'idée se base sur du précalcul en partitionnant un hypergraphe en cellules, dans lesquels les nœuds sont des parcours et les hyperarcs sont des arrêts, puis marquer les parcours qui permettent un voyage optimal entre les cellules. Pour un calcul d'itinéraire, l'algorithme RAPTOR peut donc être restreint aux parcours marqués et ceux dans les cellules de départ et d'arrivée.

Il existe d'autres variantes. Une première permet d'augmenter le nombre de critères du front de Pareto. Ainsi maintenant plutôt que de garder uniquement une heure d'arrivée pour chaque *round* à chaque arrêt, l'algorithme va garder un ensemble d'étiquettes incomparables (un front de Pareto). Une autre variante permet de résoudre le problème d'arrivée au plus tôt par plage horaire. L'algorithme prend toutes les heures de départ à l'arrêt (pour chaque trajet passant par l'arrêt de départ) puis les trie par heure de départ décroissante avant de lancer RAPTOR pour chaque heure de départ.

2.5.3 CSA

Des variantes pour résoudre le problème d'arrivée au plus tôt par plage horaire et le problème multiobjectif existent pour l'algorithme CSA. La variante pour le problème d'arrivée au plus tôt par plage horaire modifie un peu l'algorithme CSA : les connexions sont triées par heure de départ

décroissante et la manière dont une connexion est « utilisable » va quelque peu changer. Avec la version classique de l'algorithme CSA, une connexion est utilisable si l'heure d'arrivée au plus tôt à son arrêt de départ permettait de monter dans le véhicule pour l'emprunter. Maintenant la logique est inversée. Quand une connexion est scannée, nous voulons savoir l'heure d'arrivée au plus tôt à l'arrêt d'arrivée. Trois cas apparaissent :

— le premier est le plus simple, la connexion permet d'atteindre l'arrêt d'arrivée.
— le deuxième est que le trajet de la connexion permet d'atteindre un trajet qui atteint l'arrivée.
— le troisième est qu'un chemin piéton permet de prendre un trajet qui atteint l'arrêt d'arrivée.

Ensuite, on prend le minimum entre ces 3 cas et l'on met à jour le front de Pareto de l'arrêt de départ de la connexion. La variante qui permet de résoudre le problème multiobjectif, prend en compte également le nombre de transferts en gardant le même principe que la version de l'algorithme CSA pour le problème d'arrivée au plus tôt par plage horaire.

2.5.4 Trip Based Routing

De la même manière que pour les algorithmes de calcul d'itinéraires pour les transports en commun, ceux qui résolvent le problème multiobjectif vont également utiliser du prétraitement pour améliorer les performances. L'algorithme *Trip Based Routing* (TBR) [Witt, 2015] calcule un front de Pareto avec comme critère l'heure d'arrivée et le nombre de transferts.

L'algorithme TBR utilise les trajets, qui sont des moyens de transport, et les transferts entre les trajets comme ses éléments de base. Contrairement à l'algorithme CSA, TED ou TDD, TBR ne garde pas d'étiquettes pour chaque arrêt. À la place, les trajets sont étiquetés avec l'arrêt par lequel le voyageur est monté. Ensuite, une liste précalculée de transferts vers d'autres trajets est parcourue et les nouveaux trajets atteints sont étiquetés. Quand un trajet atteint l'arrivée, un itinéraire est rajouté à l'ensemble des résultats. L'algorithme se termine quand tous les itinéraires optimaux ont été trouvés.

L'algorithme se base sur le fait qu'une fois qu'un voyageur est monté dans un trajet ses futures options sont clairement définies :

– Soit il utilise un transfert pour rejoindre un autre trajet en utilisant la liste précalculée des transferts.
– Soit l'arrivée est atteinte, dans ce cas-là, l'heure d'arrivée est retrouvée dans la table horaire.

Nous pouvons donc voir qu'il n'est pas nécessaire de garder les heures d'arrivée pour les arrêts intermédiaires des trajets. L'algorithme se déroule de manière très similaire à un parcours en largeur, où les niveaux sont le nombre de transferts. TBR est donc multiobjectif par nature.

Même si une structure en graphe est utilisée, il n'y a pas besoin d'une file de priorité. L'étape de précalcul est nécessaire pour calculer les transferts entre les trajets, mais peut être parallélisée.

L'algorithme TB a un temps d'exécution très faible (voir tableau 2.9) avec un temps de précalcul allant de quelques minutes à une heure en fonction de la taille du réseau. L'algorithme TB est plus rapide que le RAPTOR.

Variante du Trip Based Routing Une variante pour réduire le temps de réponse de l'algorithme *Trip Based Routing* existe [Witt, 2016]. Cette variante se base sur le même principe que l'algorithme *Transfer Patterns* : le but est de calculer l'ensemble des lignes qui correspond à l'itinéraire

optimal, afin de n'utiliser qu'un sous-ensemble des lignes pendant l'exécution de l'algorithme *Trip Based Routing*.

2.5.5 Public Transit Labeling

Il existe une variante pour le problème d'arrivée au plus tôt par plage horaire [Delling et al., 2015a]. La variante est assez facile à implémenter, il faut parcourir de manière coordonnée les *forwards labels* et les *backwards labels* et pour chaque *hub* h, nous considérons l'itinéraire avec son heure de départ et son heure d'arrivée. Comme le problème d'arrivée au plus tôt par plage horaire ne cherche que les itinéraires qui partent le plus tard et arrivent le plus tôt, alors un front de Pareto est maintenu pour supprimer les itinéraires inutiles.

Il existe aussi une variante multiobjectif qui modifie le graphe expansé dans le temps : un arc entre deux connexions (u, w) est transformé en un arc (u, v) de coût 0 et un arc (v, w) de coût 1. Le coût de 1 est interprété comme le fait de changer de véhicule, nous permettant de compter le nombre de transferts. Pour modéliser le fait de rester dans un même véhicule, les nœuds de connexions consécutives sont liés avec des arcs avec un coût nul. La valeur stockée par les *labels* de ce graphe est le nombre de transferts pour aller de l'arrêt de départ à l'arrêt d'arrivée, la durée de l'itinéraire peut être calculée avec l'heure d'arrivée et l'heure de départ. Pour calculer un front de Pareto, un premier itinéraire d'arrivée au plus tôt est calculé, puis on itère sur les événements à l'arrêt d'arrivée pour essayer de trouver un itinéraire avec moins de transferts, si c'est le cas on l'ajoute aux résultats, puis on répète l'opération jusqu'à avoir scanné tous les événements de l'arrêt d'arrivée.

2.5.6 Performances des algorithmes de calcul d'itinéraire pour les transports en commun multiobjectif

Nous avons deux tableaux pour comparer les différents algorithmes présentés pour le problème d'arrivée au plus tôt par plage horaire et le problème multiobjectif.

Algorithme	Réseau	Million de connexions	Pré-traitement (h)	Temps de réponse (ms)
Time Expanded Dijkstra (MLS)	Londres	5,1	-	50
Time Dependent Dijkstra	-	-	-	-
RAPTOR	Londres	5,1	-	5,4
Connection Scan Algorithm	Londres	5,1	-	7,5
Trip Based Routing	Londres	5,1	0,1	1,2
Transfer Patterns	Allemagne	13,9	372	0,2
Public Transit Labeling	Londres	5,1	49	0,018

TABLE 2.9 – Tableau récapitulatif des algorithmes pour le calcul d'itinéraire pour les transports en commun multiobjectif (heure d'arrivée et nombre de transferts) en se basant sur [Bast et al., 2016, Dibbelt et al., 2018, Delling et al., 2015a, Witt, 2015].

Nous pouvons voir dans le tableau 2.9 les performances des algorithmes pour le problème multiobjectif, dans ce cas bi-objectif avec l'heure d'arrivée et le nombre de transferts, sur des requêtes générées de manière aléatoire. Les temps de réponse viennent d'expériences différentes, les résultats sont modifiés pour prendre en compte les différentes machines utilisées. Encore une fois, l'algorithme le plus lent est une adaptation de l'algorithme de Dijkstra, l'algorithme *Multi-criteria Label Setting*, avec un temps de réponse de 50 ms. Au niveau des algorithmes mettant de côté la représentation des graphes, RAPTOR est plus rapide que CSA avec des temps de réponse respectivement de 5 ms et 10 ms

Ensuite viennent les algorithmes avec du précalcul, avec le *Trip Based Routing* et son temps de précalcul extrêmement faible qui lui permet de gagner un facteur quasiment égal à 5 comparé au RAPTOR. En ce qui concerne les algorithmes avec des temps de précalcul plus long, nous avons encore une fois le *Transfer Patterns* avec un temps de précalcul conséquent, de 372 heures, pour un temps de réponse 4 fois plus rapide que le *Trip Based Routing*. Le *Public Transit Labeling* a un temps de précalcul beaucoup plus long que sa version monobjectif, passant d'approximativement une heure à 49 heures, mais le temps de réponse reste toujours extrêmement rapide, plus de dix fois inférieure à l'algorithme *Transfer Patterns*.

Algorithme	**Réseau**	**Million de connexions**	**Pré-traitement (h)**	**Temps de réponse (ms)**
Time Expanded Dijkstra	-	-	-	-
Time Dependent Dijkstra (SPCS)	Londres	5,1	-	843
RAPTOR	-	-	-	-
Connection Scan Algorithm	Londres	5,1	-	161
Trip Based Routing	-	-	-	-
Transfer Patterns	Allemagne	13,9	249	2,2
Public Transit Labeling	Londres	5,1	0,9	0,05

TABLE 2.10 – Tableau récapitulatif des algorithmes pour le calcul d'itinéraire pour les transports en commun pour le problème d'arrivée au plus tôt par plage horaire en se basant sur [Bast et al., 2016, Delling et al., 2015a].

Nous pouvons voir dans le tableau 2.10 les performances des algorithmes pour le problème d'arrivée au plus tôt par plage horaire sur des requêtes générées de manière aléatoire. Ce problème revient à résoudre un problème biobjectif avec comme critère l'heure d'arrivée et l'heure de départ. Les temps de réponse viennent d'expériences différentes, les résultats sont modifiés pour prendre en compte les différentes machines utilisées. Le tableau des résultats pour le problème d'arrivée au plus tôt par plage horaire ressemble au tableau du problème d'itinéraire monobjectif avec l'absence de l'algorithme RAPTOR et *Trip Based Routing*. L'algorithme SPCS, qui est une variante de l'algorithme de Dijkstra, est le plus lent avec 843 ms de temps de réponse, ensuite le CSA est beaucoup plus rapide avec 161 ms de temps de réponse.

Pour les algorithmes avec du précalcul, les temps de précalcul sont similaires, mais les temps de réponse sont beaucoup plus lents pour les algorithmes *Transfer Patterns* et *Public Transit Labeling*, comparé au problème monobjectif et ainsi qu'au problème multiobjectif.

Les temps de réponse des algorithmes pour le problème d'arrivée au plus tôt par plage horaire sont beaucoup plus longs, car le nombre de résultats renvoyé est très grand, en moyenne une centaine, ce qui explique que les temps de réponse soient beaucoup plus lents.

Nous voyons grâce au tableau 2.9 et au tableau 2.10 que le problème du calcul d'itinéraire pour les transports en commun biobjectif est résolu, mais l'augmentation des critères dégradent fortement les performances. Cependant, le multicritère, au-delà du biobjectif, est nécessaire pour fournir aux utilisateurs des résultats qualitatifs.

2.6 Synthèse

Nous avons pu voir dans ce chapitre trois problèmes majeurs :

1. le calcul d'un plus court chemin.
2. le calcul d'itinéraire pour les transports en commun.
3. le calcul d'itinéraire pour les transports en commun multiobjectif.

De plus, pour chacun de ces problèmes nous avons identifié quatre classes de problèmes :

1. un calcul de chemin d'un point de départ vers un point d'arrivée.
2. un calcul de chemin d'un point de départ vers tous les autres points du graphe.
3. un calcul d'un chemin d'un point de départ vers un point d'arrivée avec le point de départ et d'arrivée qui changent.
4. un calcul de chemin de tous les points du graphe vers tous les points du graphe.

Les classes de problèmes 1 et 2 sont résolues pour les trois problèmes majeurs. Nous avons des algorithmes avec des temps de réponse extrêmement rapides au prix de précalcul d'environ une heure même pour des réseaux immenses. La classe de problème 3 est elle aussi résolue pour les trois problèmes majeurs. Cependant pour le calcul d'itinéraire pour les transports en commun multiobjectif nous pouvons voir que les temps de réponse restent rapides, mais que l'ajout de critère dans le front de Pareto augmente fortement les temps de réponse et de précalcul. De plus, les algorithmes utilisant du prétraitement doivent être robustes, car les réseaux de transports en commun changent souvent. Néanmoins aucun des algorithmes avec prétraitement présentés n'est robuste. Il reste beaucoup de travail à faire pour le problème multiobjectif, au-delà du biobjectif, et utilisant du précalcul robuste. La classe de problème 4 ne sert pas dans la pratique, car elle est beaucoup trop sensible aux changements, tous les chemins possibles sont calculés, mais le moindre changement du réseau implique de nombreux calculs.

Cette focalisation sur le multiobjectif se base sur les besoins des utilisateurs, à qui on ne peut pas proposer un seul et unique résultat. Plusieurs résultats sont nécessaires et ces résultats doivent représenter au mieux l'ensemble des possibilités, ce qui est faisable avec l'utilisation d'un front de Pareto. Pour avoir plus de résultats, il faut augmenter le nombre de critères, mais cela diminue très fortement les performances, alors que les utilisateurs veulent un grand nombre de résultats « qualitatifs » et que le calcul d'itinéraire se fasse extrêmement rapidement, ce qui implique de créer de nouveaux algorithmes.

Contributions

CHAPITRE 3

Recherche multiobjectif de plus court chemin dans un réseau de transports en commun

Un des problèmes principaux d'un calcul d'itinéraire de transports en commun réaliste est le besoin de donner aux utilisateurs plusieurs résultats qualitatifs. Usuellement, les calculs d'itinéraires de transports en commun impliquent quatre critères principaux : l'heure de départ, l'heure d'arrivée, le nombre de transferts et la distance de marche. Le problème du calcul d'un front de Pareto avec ces critères est appelé le problème Pareto range. Ce problème est complexe et difficile à résoudre avec les contraintes du monde industriel des applications pour smartphone, par exemple un temps de réponse de l'ordre de la seconde. Dans ce chapitre, nous présentons le Goal Directed Connection Scan Algorithm (GDCSA), un algorithme exact qui permet, pour la première fois, de résoudre ce problème avec des temps de réponse proche de la seconde sur la plupart des réseaux européens ou des réseaux de transports ferroviaires nationaux, comme Berlin ou la Suisse. De plus, le GDCSA satisfait d'autres besoins industriels : il est conceptuellement simple et facile à mettre en œuvre. Il partitionne le graphe en de petites régions géographiques et précalcule des bornes inférieures sur la durée d'un itinéraire afin de sélectionner pour chaque calcul d'itinéraire un sous-ensemble des régions pour diminuer le nombre de connexions à scanner. En combinant ce sous-ensemble de connexions et un calculateur d'itinéraire utilisant ces quatre critères, le nombre de connexions est divisé par un facteur allant jusqu'à 7,6 par rapport au meilleur algorithme actuel. Le nombre de nœuds touchés durant le calcul d'itinéraire est divisé par un facteur allant jusqu'à 2 et le temps de réponse est divisé par un facteur allant jusqu'à 7 sur des réseaux métropolitains. L'intégration du GDCSA dans un serveur backend pour une application de smartphone améliore le temps de réponse d'un facteur 5 tout en gardant l'intégralité des solutions Pareto optimales.

One of the main problems for a realistic journey planning in public transit is the need to give the user multiple qualitative choices. Usually, public transit journeys involve four main criteria : the departure time, the arrival time, the number of transfers and the walking distance. The problem of computing Pareto sets with these criteria is called the Pareto range query problem. This problem is complex and difficult to solve within the constraints of the industrial world of smartphone applications, such as a response time of the order of a second. In this chapter, we present the Goal Directed Connection Scan Algorithm (GDCSA), an exact algorithm that allows, for the first time, to solve this problem with run times close to one second on most European cities or country-wide networks, like Germany or Switzerland. In addition, GDCSA satisfies other industrial needs : it is conceptually simple and easy to implement. It partitions the graph in geographically small areas and precomputes some lower bounds on the duration of a trip in order to select for each itinerary a subset of these areas to decrease the number of scanned connections. Combining this subset and a journey planning using four criteria, the number of scanned connections is lowered by a factor of up to 7.6 times compared to the best algorithms. The number of nodes opened during the search is lowered by a factor of up to 2 and the query times are lowered by a factor of up to 7 on metropolitan networks. The integration of GDCSA in a smartphone application backend server led to an improvement in results by a factor of 5, while keeping all the Pareto optimal solutions.

3.1 Introduction

Répondre au besoin d'un utilisateur qui veut aller d'un arrêt A à un arrêt B dans un réseau de transport n'est pas un problème aisé. Sans aucune information préalable, il faut répondre aux besoins d'utilisateurs tous différents les uns des autres, ce qui interdit le calcul d'un seul et unique chemin d'arrivée au plus tôt. Un utilisateur peut en effet préférer arriver au plus tôt à sa destination, ou minimiser le nombre de transferts afin de rester assis plus longtemps dans le véhicule, ou même minimiser la distance de marche, car il voyage avec des bagages, ou n'importe quelle autre raison imaginable. Une solution à ce problème est de renvoyer plusieurs résultats devant être incomparables entre eux par rapport à des critères afin que tous les résultats soient utiles pour les utilisateurs. Cet ensemble de résultats est appelé Pareto optimal. Beaucoup de travaux existent sur le problème de calcul de chemin d'arrivée au plus tôt dans un réseau de transports en commun, mais très peu intègrent les besoins réels des utilisateurs, qui ne sont pas seulement bi-objectif et nécessitent de nombreux critères.

Il existe de multiples variantes du problème de calcul d'itinéraire pour les transports en commun :

- La variante d'arrivée au plus tôt par plage horaire qui résout simultanément le problème d'arrivée au plus tôt pour toutes les heures de départ.
- La variante dite *range* qui résout simultanément le problème d'arrivée au plus tôt pour les itinéraires qui sont au plus deux fois plus longs que l'itinéraire le plus rapide.

Pour chacune de ces variantes, nous pouvons ajouter un ensemble de Pareto pour résoudre un problème multicritères qui amène à la variante de Pareto par plage horaire et la variante de Pareto *range*.

Quatre critères se détachent clairement par rapport à ce que les utilisateurs veulent : l'heure d'arrivée, l'heure de départ, le nombre de transferts et la distance de marche. La variante de Pareto *range*, plus spécifiquement avec les quatre critères décrits précédemment, est utile pour les utilisateurs, car ils ne veulent pas arriver beaucoup plus tard que la plus petite heure d'arrivée et peuvent avoir des préférences par rapport au nombre de transferts ou à la distance de marche. L'heure de départ et l'heure d'arrivée sont les deux critères de la variante *range*. Tandis que les deux autres critères offrent un large choix pour les utilisateurs, ce qui leur permet de choisir le résultat approprié pour leur mobilité, leur aversion pour les changements ou n'importe quel autre besoin. À noter que le prix n'est pas utilisé comme critère, car la majorité des utilisateurs ont des abonnements et donc le prix d'un trajet n'est pas important.

L'énorme désavantage avec l'utilisation d'un front de Pareto pour la variante *range* du calcul d'itinéraire est l'ajout d'un très grand nombre d'opérations. Ainsi chaque ajout d'un critère dans un front de Pareto fait augmenter de manière exponentielle la taille des solutions. Ce qui se traduit par un temps de réponse de plus en plus lent. Or le temps de réponse ne peut pas être supérieur à 2 secondes lors d'une utilisation interactive, sinon les utilisateurs abandonnent. Toutes les entreprises qui travaillent sur ce genre d'applications remarquent ce phénomène [8, 6].

Pour résoudre ce problème, il existe principalement deux types de méthode. La première précalcule des solutions pendant plusieurs heures (par exemple la nuit) afin de permettre une réponse rapide aux utilisateurs. La seconde calcule à la demande les solutions. Le principal désavantage des méthodes de précalcul est qu'elles ne peuvent pas prendre en compte de manière fluide les modifications qui apparaîtront de manière inévitable sur le réseau. Quant aux méthodes de calcul à la demande (appelé méthode de recherche), elles n'ont pas ce problème, mais sont trop lentes pour être utilisées dans un calculateur d'itinéraires pour les transports en commun moderne.

On propose de combiner adroitement ces deux méthodes, de manière similaire aux travaux de [Maue et al., 2010] en les appliquant cependant aux réseaux de transports en commun.

L'algorithme GDCSA a été présenté au *1st International Workshop on Heuristic Search in Industry* (HSI) ainsi que publié et présenté à la conférence *10th International Conference on Operations Research and Enterprise Systems* (ICORES) [Finkelstein and Régin, 2020].

3.2 État de l'art

Différents algorithmes existent pour résoudre le problème de recherche d'itinéraire pour la variante Pareto par plage horaire. Nous avons regroupé dans le tableau 3.1 les plus utilisés. À noter que les temps de réponse sont pour un front de Pareto avec trois critères (heure de départ, heure d'arrivée et nombre de transferts). Nous pouvons voir que les algorithmes sans précalcul ont des temps de réponse assez lents, sachant qu'il faut encore ajouter un critère au front de Pareto ce qui va encore grandement ralentir le temps de réponse. Pour les algorithmes avec précalcul, le *Transfer Patterns* a un temps de précalcul très long donc très coûteux en infrastructure. Quant au *Trip Based Routing*, il ne permet pas l'ajout de critères à son front de Pareto. Ainsi, aucun algorithme existant ne permet de résoudre ce problème avec un temps de précalcul raisonnable et des temps de réponse satisfaisants pour les utilisateurs. Il existe cependant des variantes des algorithmes RAPTOR et CSA qui ont pour but de faire diminuer le temps de réponse avec un peu de précalculs.

Algorithme	**Réseau**	**Million de connexions**	**Pré-traitement (h)**	**Temps de réponse (ms)**
Time Expanded Dijkstra	-	-	-	-
Time Dependent Dijkstra	-	-	-	-
RAPTOR	Londres	5,1	-	922
Connection Scan Algorithm	Londres	5,1	-	466
Trip Based Routing	Londres	5,1	0.1	70
Transfer Patterns	Allemagne	13,9	372	27,1
Public Transit Labeling	-	-	-	-

TABLE 3.1 – Tableau récapitulatif des algorithmes pour le calcul d'itinéraire pour les transports en commun multiobjectif (heure d'arrivée, heure de départ et nombre de transferts) par plage horaire en se basant sur [Bast et al., 2016, Witt, 2015].

Une alternative utilisée par [Delling et al., 2019] pour l'algorithme RAPTOR est de ne plus calculer le front de Pareto complet, mais uniquement une sous-partie. Le but de l'algorithme est d'utiliser des itinéraires « ancres » (*anchor journeys*) qui sont les résultats du front de Pareto à deux critères (heure d'arrivée et nombre de transferts), ainsi que des valeurs d'écarts par rapport à l'heure d'arrivée et au nombre de transferts. Pour un front de Pareto avec plus de critères, l'algorithme garde les résultats avec une heure d'arrivée et un nombre de transferts inférieurs aux heures d'arrivées et nombre de transferts des résultats « ancres » auxquels on ajoute les valeurs

d'écarts. Cependant, cette approche ne convient pas, car le fait de ne pas avoir toutes les solutions est extrêmement problématique dans un contexte industriel où l'exactitude est demandée.

Une variante du *Connection Scan Algorithm* a été développée afin d'améliorer le temps de réponse au prix de quelques précalculs. L'idée du *Accelerated Connection Scan Algorithm* (ACSA) est d'utiliser des techniques dirigées par le but sous la forme de graphes multiniveaux. Une partition multiniveaux hiérarchisée de l'ensemble des arrêts est faite, ce qui est facile à trouver entre des villes dans un réseau national. Cela devient beaucoup plus complexe dans une ville spécifique. L'algorithme ACSA utilise un graphe hiérarchique organisé de manière géographique, de grandes zones en haut et des petites en bas. Un niveau va correspondre à une abstraction géographique de la réalité : le but sera de ne pas scanner toutes les connexions d'une cellule. On peut précalculer un sous-ensemble de connexions dans une cellule qui va nous permettre de traverser cette cellule. Ce sous-ensemble s'appelle les connexions de transit. L'utilité des connexions de transit est que les connexions dans un itinéraire où l'utilisateur ne monte pas ou ne descend pas d'un véhicule n'ont pas besoin d'être scannées. Cela permet à l'algorithme ACSA de scanner moins de connexions en identifiant les zones importantes pour un calcul d'itinéraire spécifique et en utilisant leurs connexions de transit. Le ACSA est très dépendant de son partitionnement hiérarchique complexe et a une augmentation significative de sa complexité algorithmique comparativement au CSA, comme dit par ses auteurs [Dibbelt et al., 2018]. De plus, la prise en compte des modifications du réseau prend en moyenne 2 minutes avec 16 cœurs, alors que ces modifications peuvent arriver plusieurs fois par minute, et l'algorithme ACSA requiert que la table horaire modifiée soit similaire à la table horaire d'origine. Toutes ces raisons font que le ACSA n'est pas un candidat, car l'algorithme doit rester simple et maintenable.

Ma contribution, qui est aussi l'approche développée dans ce chapitre, est de partitionner le graphe d'arrêts géographiquement. Puis avec l'aide de bornes inférieures et supérieures on ne garde qu'un sous-ensemble de zones « utiles » pour un calcul d'itinéraire spécifique. Le partitionnement du graphe ainsi que les bornes inférieures sont calculés à l'aide d'une phase de précalcul, tandis que les bornes supérieures et les zones candidates sont calculées à la volée pour chaque calcul d'itinéraire spécifique.

3.3 Partitionnement du graphe

Le but est de partitionner le graphe de manière géographique et de faire en sorte que chaque partie soit d'une taille similaire afin de permettre à la recherche d'être guidée vers les zones les plus intéressantes. La taille de chaque partie ainsi que l'équilibre entre les tailles des différentes parties vont avoir un impact très fort sur les performances de l'algorithme.

Pour partitionner le graphe de manière « géographique », nous commençons par décrire certains concepts et certaines notations, comme les zones et frontières de zone.

3.3.1 Graphe géographique

Tous les nœuds d'un graphe géographique ont une latitude et une longitude. Ce graphe géographique permet de guider la recherche d'itinéraire. Nous allons construire un graphe à partir de l'ensemble d'arrêts et de l'ensemble des connexions. Nous avons $G_g = (S, E)$ le graphe géographique extrait à partir de la table horaire, où S est l'ensemble des arrêts de la table horaire T et E est l'ensemble des arcs. L'ensemble des arcs est défini comme suit :

$$(u,v) \in E \Rightarrow \begin{cases} \exists t_1, t_2 \in M,\ z \in T \mid (u,v,t_1,t_2,z) \in C & \text{ou} \\ \exists d \in M \mid (u,v,d) \in F \end{cases} \tag{3.1}$$

avec M l'ensemble des secondes d'une journée, T la table horaire, C l'ensemble des connexions et F l'ensemble des chemins piétons. Ainsi un arc est créé entre un arrêt u et un arrêt v s'il existe une connexion entre u et v ou s'il existe un chemin piéton entre u et v.

3.3.2 Zone

Définition 3.3.1. Une zone est un sous-graphe connecté de G_g, tel que l'ensemble des zones forme une partition de G, c'est-à-dire qu'elles sont disjointes deux à deux et l'union des différentes parties nous donne l'ensemble des nœuds.

Nous introduisons quelques notations pour une zone.

Notation 3.3.1. Une connexion c appartient à une zone a si et seulement si $c_{dep\_stop} \in a$.

Notation 3.3.2. Pour tout $u \in S,\ A(u)$ est la zone contenant u.

Notation 3.3.3. $\text{connections}(a)$ est la liste des connexions d'une zone a.

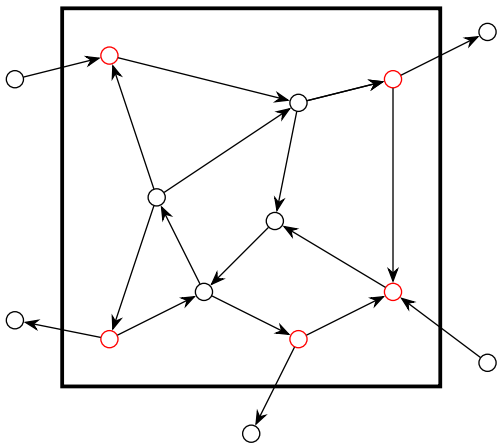


FIGURE 3.1 – Exemple d'une zone.

Dans la figure 3.1, nous pouvons voir que la zone est le carré noir, les nœuds du graphe G_g sont les cercles noir ou rouge et que les arcs sont les flèches entre les nœuds. De plus, la zone encapsule certains des nœuds du graphe et coupe certains arcs, ce qui pose la question de l'appartenance des connexions à une zone, répondue par la notation 3.3.1.

3.3.3 Frontière d'une zone

Définition 3.3.2. La frontière d'une zone est l'ensemble des arrêts $\mathcal{B}(a) \subseteq a$ qui ont au moins un arc en dehors de la zone a.

La frontière $\mathcal{B}(a)$ contient les arrêts qui permettent de sortir de la zone a pour en atteindre une autre et donc d'atteindre les arrêts internes d'autres zones. Ce sont les arrêts en rouge dans la figure 3.1.

3.3.4 Partitionnement en zones du graphe

Le partitionnement peut se faire de multiples manières, mais le but reste le même : faire un ensemble de zones équilibrées, c'est-à-dire avec un nombre de nœuds similaires, et avec un nombre d'arcs coupés minimum, aussi appelé *min-cut*. Nous imposons une contrainte sur la taille maximale d'une zone égale à $(1 + \epsilon)$ fois la taille moyenne des zones et on essaye juste de minimiser la taille totale de la coupe.

Définition 3.3.3. Soit $G = (X, A)$ un graphe non dirigé, nous recherchons des zones $A_0, \cdots, A_k$ qui partitionnent X, c'est-à-dire $A_0 \cup \cdots \cup A_k = X$ et $A_i \cap A_j = \emptyset$ pour $i \neq j$. La contrainte d'équilibre fait en sorte que $\forall i \in \{0 \cdots k\} : |V_i| \leq (1+\epsilon)\frac{|X|}{(k+1)}$. C'est-à-dire que toutes les zones ont le même nombre de nœuds, à un ϵ près.

Grâce à nos définitions, un passager pourra traverser une zone en utilisant uniquement ces connexions. Une fois que le graphe est partitionné, nous pouvons précalculer des bornes inférieures entre les zones.

Il existe plusieurs algorithmes de partitionnement du graphe :

3.3.4.1 Inertial Flow

L'algorithme *Inertial Flow* [Schild and Sommer, 2015] utilise une méthode simple, mais efficace. Les arrêts sont triés par rapport à leurs latitudes ou longitudes (ou bien les deux). Ensuite, un certain pourcentage de nœuds au début et à la fin de la liste sont choisis (en dessous de 50%) pour devenir respectivement des « sources » et des « puits ». Puis un algorithme de flots est utilisé pour en sortir la *min-cut* qui nous donnera la plus petite frontière qui divise en deux notre ensemble d'arrêts. Ces étapes sont répétées pour chaque nouvel ensemble d'arrêts, jusqu'à ce qu'une profondeur maximale ou qu'une taille minimale de zone soit atteinte.

3.3.4.2 KaHiP

La solution *Karlsruhe High Quality Partitioning* (KaHiP) [Sanders and Schulz, 2013] est un ensemble de programmes fourni par l'Université de Karlsruhe. Le partitionnement se base sur une heuristique qui fonctionne très bien pour de grands graphes, le *multilevel graph partitioning*. Le partitionnement se fait en trois étapes (voir figure 3.2) : une étape de contraction, une étape de partitionnement et une étape de décontraction.

- La phase de contraction du graphe se fait en trouvant un ensemble d'arcs qui n'ont aucun nœud en commun avant de « contracter » chaque arc (u, v), ce qui revient à créer un nouveau nœud w connecté aux voisins de u et v. Le nœud a un poids égal à la somme des poids des nœuds u et v. Si la contraction de plusieurs arcs amène à la génération de deux arcs parallèles, alors le poids de l'arc est égal à la somme des arcs parallèles. Les arcs à contracter sont choisis avec une fonction d'évaluation et une fois que le graphe a une taille raisonnable pour être partitionné avec un autre algorithme, la phase de contraction s'arrête.
- Une partition sur le graphe contracté est une partition sur le graphe en entrée, cependant la *min-cut* dans le graphe contracté n'est pas la *min-cut* dans le graphe d'entrée. En effet, si un arc contracté est dans la *min-cut*, alors lors de la décontraction les nœuds peuvent se placer d'un côté ou de l'autre de la partition.

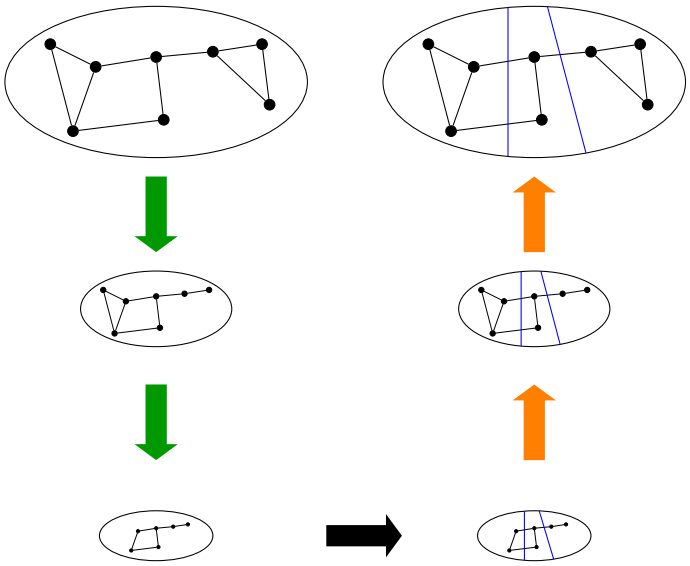


FIGURE 3.2 – Schéma de partitionnement multiniveaux d'un graphe.

– Pendant la phase de décontraction, les contractions sont défaites de manière itérative. Un algorithme de raffinement va faire passer les nœuds d'une région à une autre afin d'améliorer la *min-cut* ou l'équilibre entre les régions. Il existe deux heuristiques de recherche locale pour améliorer la *min-cut* durant la décontraction : une recherche *k-way* ou une recherche *two-way*. La recherche *two-way* permet d'échanger des blocs entre une paire de régions tandis que la recherche *k-way* permet d'échanger un nœud d'une région avec n'importe quelle autre région.

3.3.4.3 METIS

La solution *Metis* [Karypis and Kumar, 1998] se base sur la même méthode que KaHiP, c'est-à-dire du *multilevel graph partitioning*. Les différences entre METIS et KaHiP viennent des méthodes utilisées pour le partitionnement et des heuristiques lors de la phase de décontraction.

3.4 Gestion des zones candidates

Une fois le graphe partitionné, reste la question des zones à ouvrir ou à fermer pendant une recherche d'itinéraire. Tout se basera sur des bornes inférieures et supérieures sur la durée dans les transports en commun, avec une partie des calculs qui pourra être réalisée à l'avance (i.e. précalcul) et une autre qui se fera lors de la recherche d'itinéraire. Le but est de diminuer le nombre de zones considérées et donc le nombre de nœuds avec des résultats intermédiaires, sans toutefois perdre des résultats du front de Pareto à l'arrêt d'arrivée.

3.4.1 Ouverture d'une zone candidate

Nous avons deux bornes pour la durée dans les transports en commun.

Définition 3.4.1. Une borne supérieure sur la durée dans les transports en commun entre deux arrêts s et $t \in S$, avec une heure de départ τ_s s'écrit $\overline{d_{PT}}(s, t, \tau_s)$.

Définition 3.4.2. Une borne inférieure sur la durée dans les transports en commun entre deux arrêts s et $t \in S$, avec une heure de départ τ_s s'écrit $\underline{d_{PT}}(s, t, \tau_s)$.

Les techniques dirigées par le but, comme leurs noms l'indiquent, ont pour but d'aider la recherche vers l'arrivée en évitant de scanner des arrêts inutiles. Nous utilisons les mêmes techniques, mais appliquées aux zones du graphe G afin d'éviter de scanner des connexions qui nous éloigneraient de l'arrivée. Nous allons utiliser des bornes supérieures et inférieures sur la durée entre des arrêts et plus spécifiquement entre les zones.

Une zone candidate ouverte ajoute ses connections à l'espace de recherche du calcul d'itinéraire pour les transports en commun. De la même manière, si la zone candidate n'est pas ouverte alors ses connexions ne sont pas ajoutées à l'espace de recherche du calcul d'itinéraire pour les transports en commun.

Définition 3.4.3. Soit $s, t \in S$. Une zone candidate a est ouverte quand on calcule un itinéraire entre s et t à un temps τ_s si et seulement si

$$\underline{d_{PT}}(A(s), a) + \underline{d_{PT}}(a, A(t)) \leq \overline{d_{PT}}(s, t, \tau_s) \tag{3.2}$$

On ne prend pas en compte le temps de traversée de la zone afin de garder les calculs très simples. Nous faisons cela car utiliser un temps de traversée égal à 0 est une borne inférieure valide et n'affecte pas les résultats de l'algorithme GDCSA.

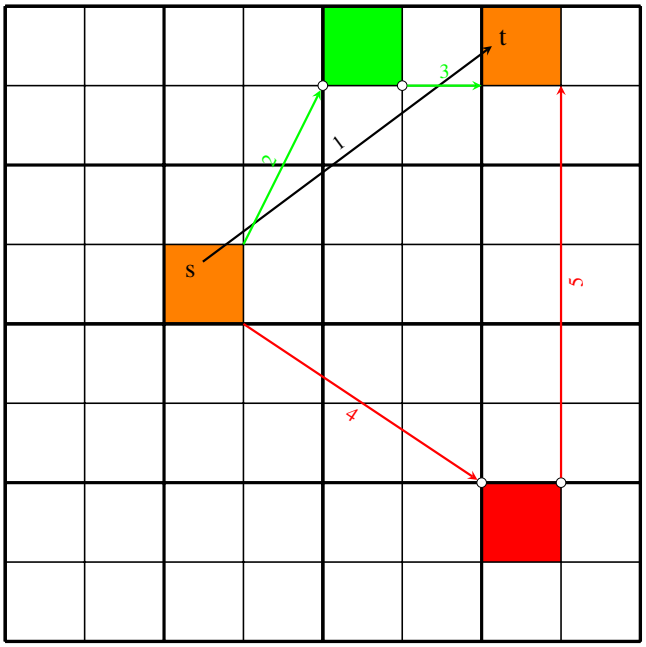


FIGURE 3.3 – Exemple d'ouverture de zones.

L'ouverture de zones est schématisée dans la figure 3.3. Supposons qu'un plus court chemin entre deux points est une ligne droite. Les flèches vertes et rouges représentent les bornes inférieures des distances entre les zones et la flèche orange représente la borne supérieure entre s et t.

Deux zones sont visibles, la verte qui est ouverte, car la somme de la longueur de la flèche 2 (entre la zone contenant s jusqu'à la zone verte) et la flèche 3 (entre la zone verte et la zone contenant t) est inférieure à la longueur de la flèche 1 (entre la zone contenant s et la zone contenant t). La zone rouge n'est pas ouverte, car la somme de la longueur de la flèche 4 (entre la zone contenant s et la zone rouge) et la flèche 5 (entre la zone rouge et la zone contenant t) est supérieure à la longueur de la flèche 1 (entre la zone contenant s et la zone contenant t).

3.4.2 Borne supérieure sur la durée

De manière intuitive, une borne supérieure sur la durée peut être trouvée en calculant rapidement la taille de l'intervalle de recherche, car un utilisateur n'est pas intéressé par des itinéraires qui arrivent significativement plus tard que l'heure d'arrivée au plus tôt.

Soit x l'heure d'arrivée au plus tôt, nous utilisons l'heure maximale d'arrivée τ_t comme définie par [Dibbelt et al., 2018], qui est égale à

$$\tau_t = \tau_s + 2 \times (x - \tau_s) \tag{3.3}$$

Cette borne supérieure sur la durée est avantageuse, car elle est réaliste. Par exemple, considérons un utilisateur qui part à 8h00 d'un point de départ et arrive à 9h00 à un point d'arrivée, alors des itinéraires qui arrivent après 10h00 peuvent être ignorés, car ils ne sont pas pertinents.

La borne supérieure sur la durée que nous utilisons est

$$\overline{d_{PT}}(s, t, \tau_s) = \tau_t - \tau_s \tag{3.4}$$

C'est-à-dire l'intervalle de temps qui peut satisfaire une demande d'itinéraire. Nous pouvons facilement estimer $\overline{d_{PT}}(s, t, \tau_s)$:

$$\begin{aligned} \overline{d_{PT}}(s, t, \tau_s) &= \tau_t - \tau_s \\ &= \tau_s + 2 \times (x - \tau_s) - \tau_s \\ &= 2 \times (x - \tau_s) \end{aligned} \tag{3.5}$$

La seule inconnue est x qui est l'heure d'arrivée au plus tôt. Elle peut être facilement calculée avec un appel à l'algorithme CSA en un temps extrêmement faible.

3.4.3 Borne inférieure sur la durée

De manière intuitive, la borne inférieure sur la durée entre deux arrêts s et t est la durée minimum sur tous les itinéraires d'une journée qui vont de la frontière de la zone contenant s à la frontière de la zone contenant t.

Les bornes inférieures d'une zone a_s sont calculées avec un PCSA, la variante avec une fenêtre de temps du CSA. Les arrêts de départs sont ceux de la frontière de la zone, supposant ainsi que nous pouvons les atteindre instantanément. Ensuite, nous lançons le PCSA sur toute la journée pour finir par itérer sur toutes les autres zones et prendre la durée minimale pour rejoindre la frontière de la zone a_t depuis la frontière de la zone a_s. L'algorithme 3.1 est une implémentation possible.

Les bornes inférieures peuvent être calculées une fois pour toutes pendant une étape de précalcul car les bornes inférieures sont valides pour une journée complète du fait que les véhicules

Algorithme 3.1 Algorithme de calcul des bornes inférieures

function LOWER BOUND(G)
  $dur \leftarrow [G.\text{areas}()][G.\text{areas}()]$ ▷ Initialisation d'une matrice de durée
  for all $a_s \in G.\text{areas}()$ **do**
    $r \leftarrow \text{PCSA}(\mathcal{B}(a_s))$ ▷ On lance un PCSA en utilisant les frontières comme sources
    for all $a_t \in G.\text{areas}()$ **do**
      $min\_dur \leftarrow +\infty$ ▷ Durée minimum pour atteindre a_t à partir de a_s
      for all $b \in \mathcal{B}(a_t)$ **do**
        $min\_dur \leftarrow \min(min\_dur, r[b])$
      end for
      $dur[a_s][a_t] \leftarrow min\_dur$
    end for
  end for
  return dur
end function

de transports en commun peuvent uniquement avoir du retard et pas d'avance. De plus, cette étape de précalcul peut être parallélisée ce qui veut dire que le temps de précalcul peut être diminué.

3.5 Goal Directed Connection Scan Algorithm

L'algorithme GDCSA se déroule en quatre phases. La première calcule une borne supérieure de la durée entre l'arrêt de départ et l'arrêt d'arrivée de la requête. La deuxième itère sur chaque zone pour ne garder que celles qui seront utiles pour répondre à la requête d'itinéraire. La troisième fusionne toutes les connexions des zones ouvertes et la dernière lance un *Pareto Range Variant Connection Scan Algorithm* (PRVCSA) en utilisant ce sous-ensemble de connexions.

Algorithme 3.2 Algorithme GDCSA

function GDCSA(G, s, t, τ_s)
  $L_a \leftarrow$ empty list
  $ub \leftarrow \overline{d_{PT}}(s, t, \tau_s)$ ▷ On itère sur toutes les zones du graphe
  for all $a \in G.\text{areas}()$ **do**
    if $\underline{d_{PT}}(A(s), a) + \underline{d_{PT}}(a, A(t)) \leq ub$ **then**
      L_a.insert(a)
    end if
  end for
  $L_c \leftarrow$ empty list ▷ On itère sur toutes les zones ouvertes du graphe
  for all $a \in L_a$ **do**
    L_c.insertAll(connections(a))
  end for
  $L_c \leftarrow \text{sort}(L_c)$ ▷ On utilise uniquement les connexions des zones ouvertes
  return PRVCSA(G, s, t, τ_s, L_c) ▷ Renvoie l'ensemble de Pareto de t
end function

GDCSA est décrit par l'algorithme 3.2. Les quatre phases sont mises en évidence dans le pseudo-code. Les deux premières parties (le calcul de la borne supérieure et l'ouverture des zones) sont faciles à coder. Il nous faut précalculer les bornes inférieures : nous le faisons avec la variante d'arrivée au plus tôt par plage horaire du CSA qui est rapide et simple à coder.

3.5.1 Optimisations

Plusieurs améliorations existent pour diminuer le nombre de connexions utilisées dans chaque zone et donc diminuer le temps d'exécution de l'algorithme.

3.5.1.1 Utiliser l'arrivée au plus tôt à la place de la borne inférieure

Plutôt que d'utiliser l'équation 3.2 (page 57) pour ouvrir une zone candidate, nous pouvons utiliser :

$$d_{PT}(s, r, \tau_s) + \underline{d_{PT}}(r, A(t)) \leq \overline{d_{PT}}(s, t, \tau_s) \tag{3.6}$$

La borne inférieure entre l'arrêt de départ de la requête et une zone candidate est remplacée par l'heure d'arrivée au plus tôt calculée par le CSA.

Cela nous permet d'éliminer un bon nombre de zones candidates, et d'en avoir une gestion plus fine.

3.5.1.2 Optimisation de l'heure d'ouverture

Toutes les connexions d'une zone ne sont pas utiles, car plus nous sommes éloignés de l'arrêt de départ s, plus le nombre de connexions atteignables diminue. Ainsi, pour une zone a, la première connexion qui peut être scannée par PRVCSA doit satisfaire

$$c_{dep\_time} \geq \tau_s + \underline{d_{PT}}(A(s), a) \tag{3.7}$$

Nous gardons ainsi uniquement les connexions qui ont une heure de départ supérieure à celle de la première connexion atteignable.

Par exemple, en considérant un itinéraire entre Nice et Cannes avec une heure de départ à 15h00. Une zone près de Cannes n'a besoin de scanner que des connexions avec une heure de départ supérieure à 15h20, car la borne inférieure entre Nice et Cannes est de 20 minutes. Cette amélioration diminue le nombre de connexions scannées.

3.5.1.3 Optimisation de l'heure de fermeture

La même optimisation peut être faite pour les dernières connexions atteignables. La dernière connexion d'une zone a qui peut être scannée par PRVCSA doit satisfaire

$$c_{dep\_time} \leq \tau_t - \underline{d_{PT}}(a, A(t)) \tag{3.8}$$

Ainsi, nous ne gardons que les connexions qui ont une date de départ inférieure à celle de la dernière connexion atteignable.

3.5.2 Temps réel

La prise en compte du temps réel dans l'algorithme GDCSA (retards, grèves ...) est garantie par le fait que les véhicules ont un contrat social et ne peuvent pas partir en avance. Ainsi le temps réel ne peut être qu'un retard, un moyen de transport ne peut partir qu'après l'heure de départ théorique. Les bornes inférieures n'ont pas besoin d'être modifiées pour prendre en compte le retard. La borne supérieure utilise un CSA qui prend en compte la mise à jour des connexions avec le retard.

3.6 Expérimentations

Nous évaluons GDCSA et le comparons à la variante Pareto *range* à quatre critères de l'algorithme CSA. En plus de mesurer les temps de réponse, nous rapportons les temps de précalcul pour chacune des différentes méthodes présentées, ainsi que d'autres métriques en rapport avec la taille de l'espace de recherche (c'est-à-dire le nombre de connexions scannées, le nombre d'arrêts avec au moins un *label* et le nombre de *labels*).

3.6.1 Instances

Réseau	Arrêts	Connexions	Lignes	Voyages	Chemins piéton
Paris	44534	3209401	1864	150963	502291
Berlin	28651	1379755	1296	63569	62456
Stockholm	14258	703326	664	34799	22138
Allemagne	74398	3601420	3599	168024	599284
Suisse	29844	2599675	5645	248826	27202

TABLE 3.2 – Taille des différents réseaux.

Nos instances de tests sont basées sur les données des réseaux de transports en commun de trois villes (Paris, Berlin et Stockholm) et de deux réseaux de train nationaux (Allemagne et Suisse). Les données sont accessibles de manière libre via un flux GTFS (https://transitfeeds.com/) qui a été téléchargé en octobre 2019. Les données de 2019 sont tout à fait comparables avec celles de 2022, les réseaux de transports en commun changent très peu. Le détail des tailles des différents réseaux est disponible dans le tableau 3.2.

Nous pouvons voir que les réseaux de transports métropolitains sont de tailles croissantes, allant de Stockholm à Paris. Le but de ces réseaux est de montrer les performances de l'algorithme GDCSA sur des réseaux denses. Quant aux réseaux de trains nationaux, leur but est de montrer les performances de l'algorithme GDCSA sur des réseaux épars.

Les chemins piétons sont donnés dans le flux GTFS, mais le graphe n'est pas transitivement fermé : les chemins manquants sont engendrés explicitement.

Les algorithmes sont implémentés en Java 11 et exécutés avec OpenJDK 11. Toutes nos expérimentations sont lancées sur un processeur Intel Core i7-1185G7 avec 32 Go de RAM.

Pour notre évaluation, nous avons lancé tous les algorithmes avec le même jeu de 1000 requêtes généré aléatoirement. L'arrêt de départ et l'arrêt d'arrivée sont choisis uniformément au hasard. L'heure de départ est choisie uniformément au hasard dans la journée.

3.6.2 Résultats de l'algorithme GDCSA

Réseau	Algorithme	Temps de réponse (ms)	# Connexions scannées	# Arrêts màj	# Labels
Paris	PRVCSA	34982	572271	22583	1041274
	GDCSA	4827	111465	14928	290151
Berlin	PRVCSA	1555	350359	16080	296856
	GDCSA	313	61547	8406	82501
Stockholm	PRVCSA	769	162497	7251	128596
	GDCSA	101	25913	3894	33385
Allemagne	PRVCSA	10301	1254153	22469	723809
	GDCSA	2490	164641	17216	333064
Suisse	PRVCSA	1180	906216	14703	199303
	GDCSA	250	171349	10119	87152

TABLE 3.3 – Détails de l'algorithme GDCSA et des variantes de l'algorithme CSA sur des réseaux de transports métropolitain et de train nationaux.

Nous présentons les résultats de l'algorithme GDCSA et de l'algorithme PRVCSA, qui sont identiques pour les deux algorithmes. L'algorithme GDCSA utilise pour le partitionnement, l'algorithme KaHiP avec un nombre de zones égal à $2^{12} = 4096$.

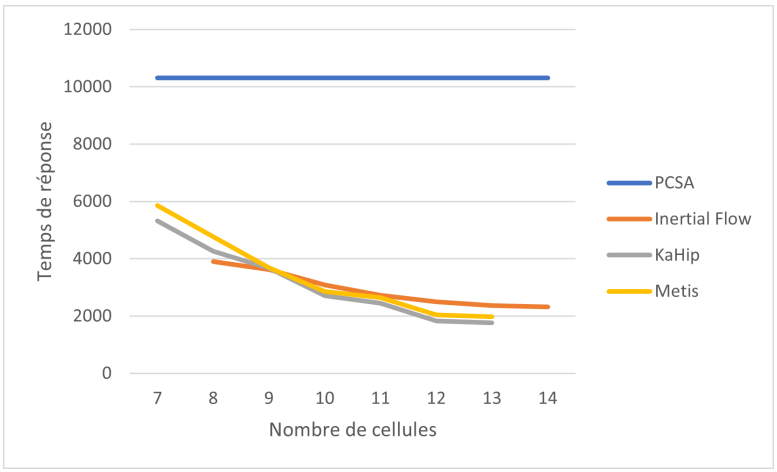
Le tableau 3.3 donne la moyenne des temps d'exécution, du nombre de connexions scannées, du nombre d'arrêts mis à jour et du nombre de *labels*.

Nous voyons que le temps d'exécution de l'algorithme GDCSA est de 4 à 7 fois plus rapide que celui de l'algorithme PRVCSA. L'algorithme GDCSA a de plus gros gains sur des réseaux métropolitains que sur des réseaux de trains nationaux. Les gains de l'algorithme ACSA sont toujours bons même pour des réseaux denses. Pour tous les réseaux, nous avons une diminution considérable du nombre d'arrêts mis à jour (avec au moins un *label*), du nombre de connexions scannées et du nombre de *labels* entre l'algorithme GDCSA et l'algorithme PRVCSA.

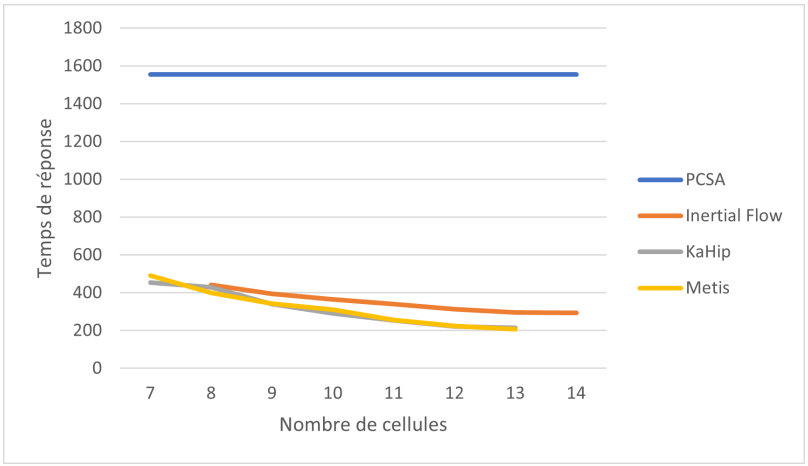
Nous pouvons voir dans la figure 3.4 et la figure A.1 que le temps de réponse diminue quand le nombre de zones augmente, et ce quel que soit l'algorithme de partitionnement (KaHiP, Metis ou Inertial Flow). Cependant, nous pouvons voir que les temps de réponses diminuent beaucoup plus lentement aux alentours de 2^{12} ou 2^{13} zones. De plus, nous pouvons voir que les temps de réponse sont très similaires pour les trois algorithmes, avec un léger avantage pour les algorithmes KaHiP et Metis, mais qui vient avec le fait que ce sont des algorithmes qui sont compilés et lancés de manière externe au code du GDCSA.

3.6.2.1 Rang géographique

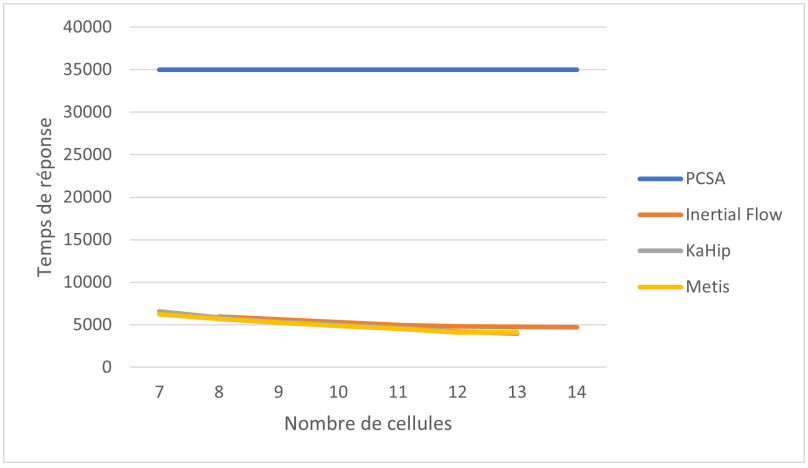
En choisissant les arrêts de départ et d'arrivée des requêtes de manière aléatoire uniforme, il y a une très forte probabilité que l'itinéraire traverse tout le réseau. Le rang géographique d'un itinéraire entre un arrêt de départ s et un arrêt d'arrivée t est le logarithme du rang du tri. C'est-à-dire les arrêts sont triés par rapport à leur distance à vol d'oiseau de l'arrêt s, soit i la position de



(a) Temps de réponse en fonction du nombre de zones pour le réseau de train de l'Allemagne

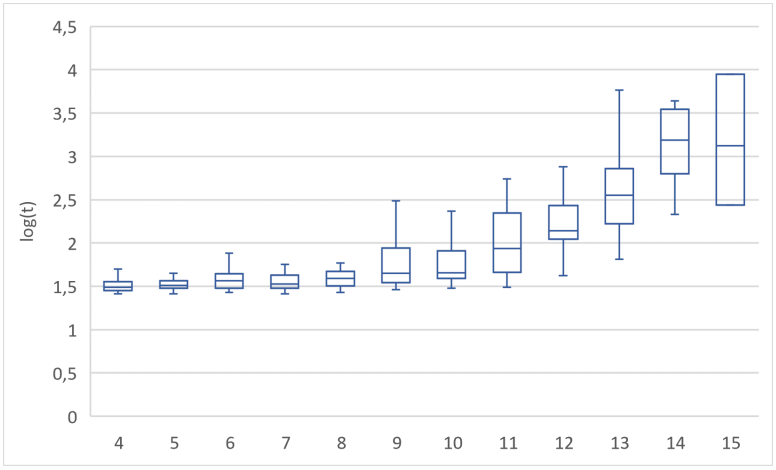


(b) Temps de réponse en fonction du nombre de zones pour la ville de Berlin

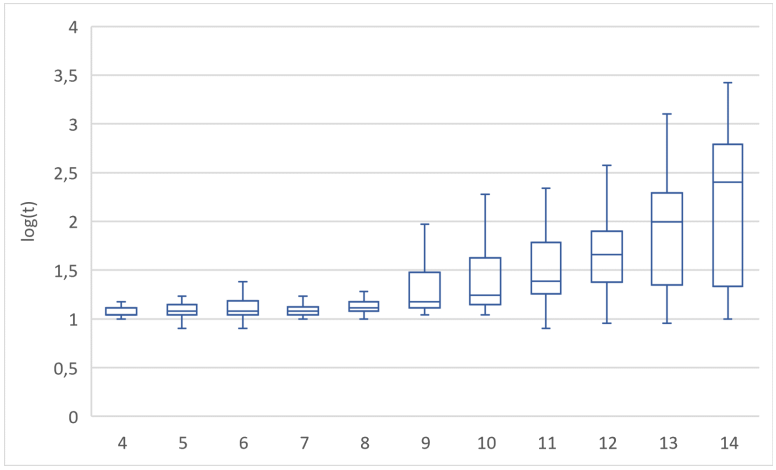


(c) Temps de réponse en fonction du nombre de zones pour la ville de Paris

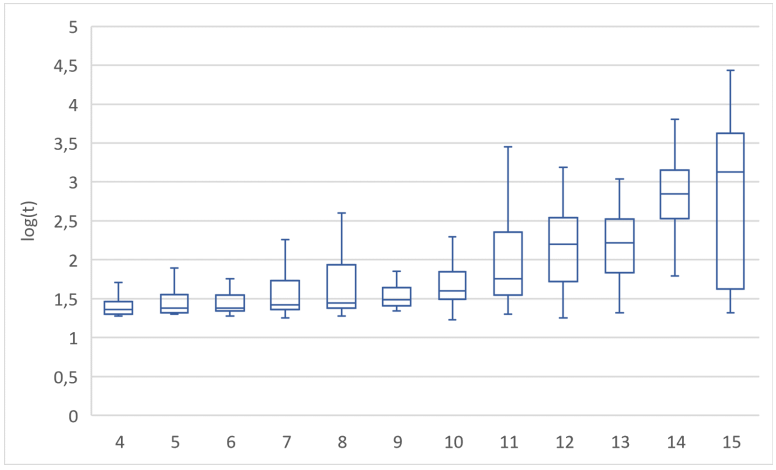
FIGURE 3.4 – Temps de réponse en fonction du nombre de zones pour les réseaux de Paris, Berlin et l'Allemagne.



(a) Temps de réponse en fonction du rang géographique pour le réseau de train de l'Allemagne



(b) Temps de réponse en fonction du rang géographique pour la ville de Berlin



(c) Temps de réponse en fonction du rang géographique pour la ville de Paris

FIGURE 3.5 – Temps de réponse en fonction du rang géographique pour les réseaux de Paris, Berlin et l'Allemagne.

l'arrêt t dans la liste des arrêts triés. Soit $\log_2(i)$ le rang géographique de l'itinéraire entre l'arrêt s et l'arrêt t.

La figure 3.5 et la figure A.2 montre les temps de réponse en fonction du rang géographique avec des diagrammes en boîtes pour tous les réseaux avec l'algorithme GDCSA. Nous pouvons voir que les temps de réponse augmentent avec le rang géographique pour tous les réseaux sauf pour le réseau national allemand où nous voyons les temps de réponse diminuer pour les derniers rangs. Cela est dû au fait que beaucoup de requêtes avec des rangs géographiques élevés n'ont pas de résultats, ce qui réduit fortement les temps de réponse pour ces rangs.

3.6.3 Précalcul de la borne inférieure

	Paris		Berlin	
Nombre de zones	**Temps de réponse (ms)**	**Précalcul (s)**	**Temps de réponse (ms)**	**Précalcul (s)**
8	5945	1221	440	153
9	5628	2092	395	272
10	5302	3597	365	469
11	4970	6202	340	774
12	**4827**	**9837**	**313**	**1152**
13	4756	14124	295	1537
14	4685	18369	292	1890

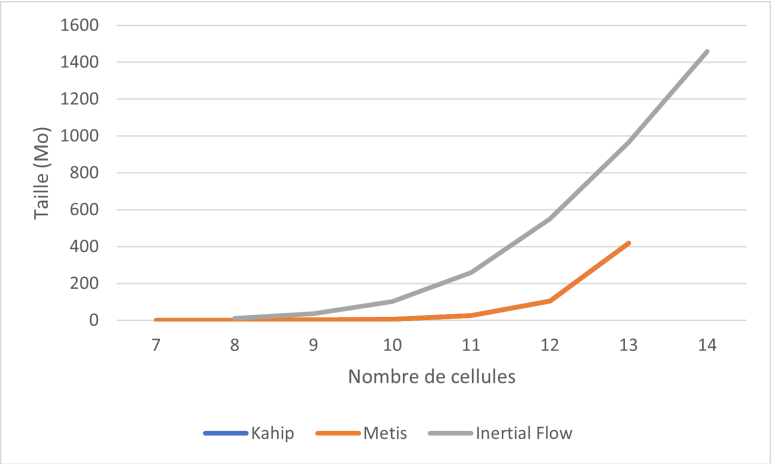
	Stockholm	
Nombre de zones	**Temps de réponse (ms)**	**Précalcul (s)**
8	125	75
9	118	119
10	108	196
11	103	296
12	**101**	**414**
13	99	537
14	99	620

TABLE 3.4 – Performance du précalcul sur les réseaux métropolitains de transports en commun.

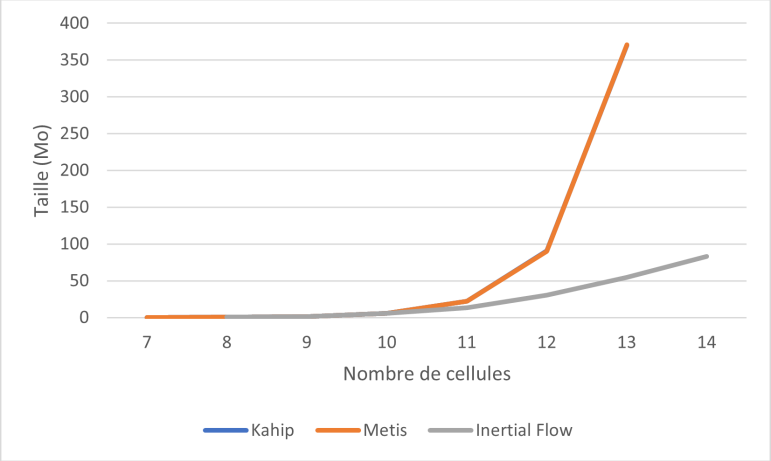
Le précalcul de la borne inférieure se fait en lançant un variante d'arrivée au plus tôt par plage horaire de l'algorithme CSA pour chaque région.

Nous évaluons le partitionnement et le précalcul sur les cinq mêmes réseaux de transports en commun, avec un nombre de zones allant de $2^7 = 128$ à $2^{13} = 8192$ pour les algorithmes KaHiP et Metis, et de $2^8 = 256$ à $2^{14} = 16384$ pour l'algorithme *Inertial Flow*.

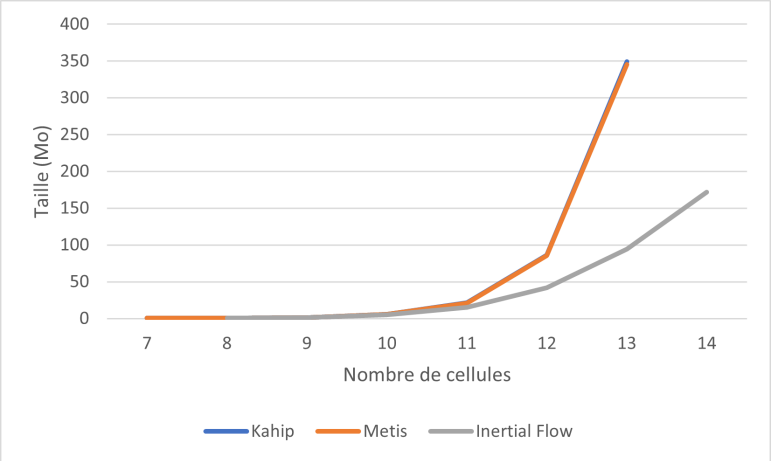
Comme nous pouvons le voir avec la figure 3.6 et la figure A.3, l'espace disque pris par le stockage du partitionnement et de la borne inférieure de la distance entre les régions est assez faible pour tous les réseaux, en moyenne une centaine de Mo. L'espace disque utilisé est plus important par les algorithmes KaHiP et Metis, sauf pour le réseaux de l'Allemagne.



(a) Taille de la partition en fonction du nombre de zones pour le réseau de train de l'Allemagne

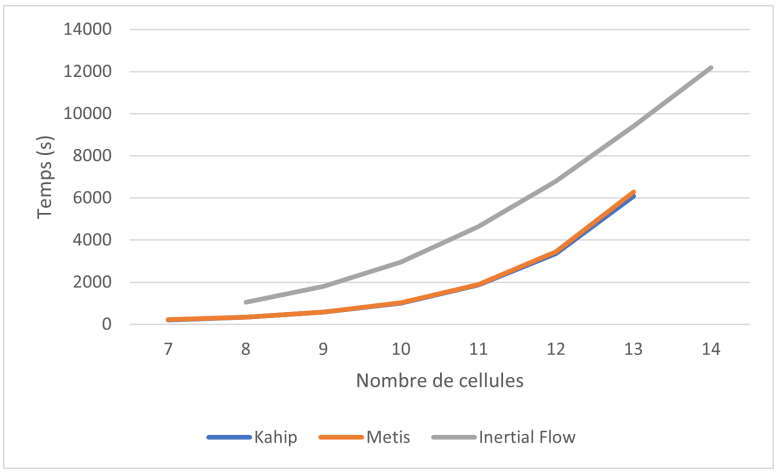


(b) Taille de la partition en fonction du nombre de zones pour la ville de Berlin

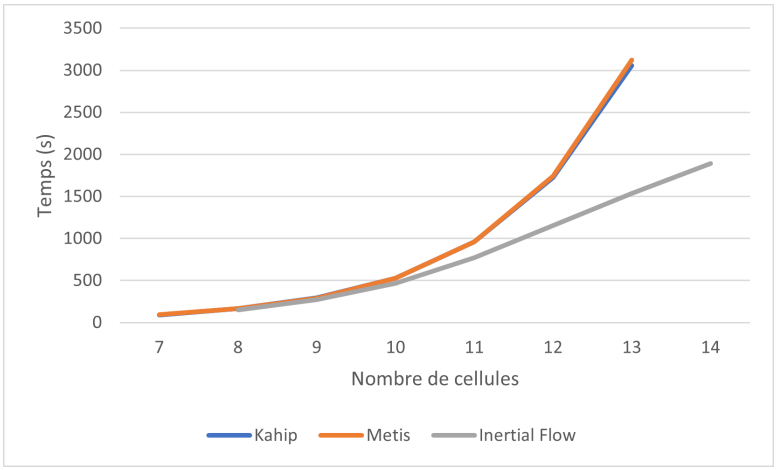


(c) Taille de la partition en fonction du nombre de zones pour la ville de Paris

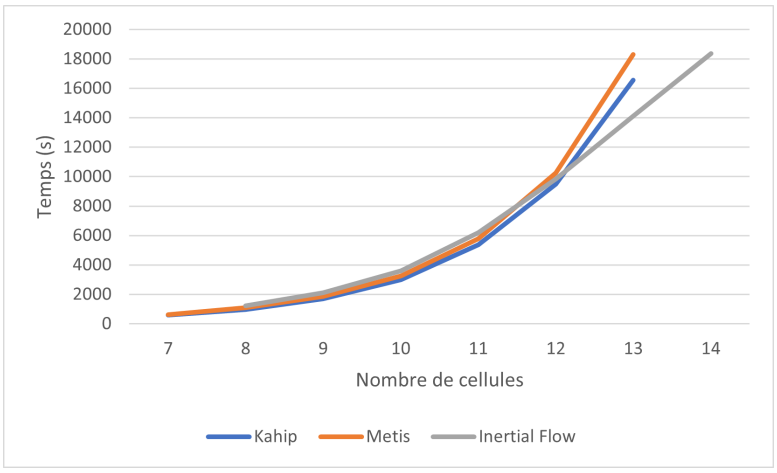
FIGURE 3.6 – Taille de la partition en fonction du nombre de zones pour les réseaux de Paris, Berlin et l'Allemagne.



(a) Temps de précalcul en fonction du nombre de zones pour le réseau de train de l'Allemagne



(b) Temps de précalcul en fonction du nombre de zones pour la ville de Berlin



(c) Temps de précalcul en fonction du nombre de zones pour la ville de Paris

FIGURE 3.7 – Temps de précalcul en fonction du nombre de zones pour les réseaux de Paris, Berlin et l'Allemagne.

	Allemagne		Suisse	
Nombre de zones	**Temps de réponse (ms)**	**Précalcul (s)**	**Temps de réponse (ms)**	**Précalcul (s)**
8	3900	1058	358	352
9	3623	1812	329	481
10	3087	2957	302	723
11	2728	4639	285	1123
12	**2490**	**6786**	**250**	**1640**
13	2372	9409	233	2246
14	2326	12176	227	2720

TABLE 3.5 – Performance du précalcul sur les réseaux nationaux de trains.

Pour ce qui est des temps de précalcul, nous pouvons voir sur la figure 3.7 et la figure A.4 que le temps de calcul est assez rapide, en moyenne une vingtaine de minutes, sauf pour les plus gros réseaux où le temps de précalcul avoisine les 3 heures. Tous les calculs sont séquentiels, et la partie la plus longue du précalcul peut être très facilement parallélisée (l'appel à la variante d'arrivée au plus tôt par plage horaire de l'algorithme CSA pour chaque région), ce qui implique que de gros gains sont possibles. On retourne pour l'algorithme *Inertial Flow* des temps de précalcul et une consommation de l'espace disque similaire aux algorithmes METIS et KaHiP.

Le choix du nombre de zones fixé avec le meilleur gain en temps de réponse comparé au temps de précalcul est de $2^{12} = 4096$, comme le montre les tableau 3.4 et tableau 3.5. Quand le nombre de zones augmente trop, vers $2^{14} = 16384$, le temps de précalcul ainsi que l'espace disque augmente trop, mais le temps de réponse baisse à peine, ce qui amène sur une valeur intermédiaire de zones.

3.7 Intégration dans un produit industriel

Le développement de l'algorithme GDCSA vient du besoin pour *Instant System* de pouvoir répondre à des appels d'offres pour des calculateurs régionaux. Le but est de pouvoir fournir de multiples résultats optimaux dans toute une région, c'est-à-dire un grand nombre de villes plus ou moins grandes et un réseau de trains ou de bus longue distance. L'algorithme GDCSA est intégré dans un serveur industriel et utilisé en production pour le réseau Modalis (région Nouvelle-Aquitaine), pour le réseau SYTRAL (agglomération lyonnaise), pour une expérimentation sur le Mobility-as-a-Service sur le réseau de Bruxelles et pour une expérimentation sur le réseau de Lausanne.

En plus d'un calculateur d'itinéraires, de nombreuses fonctionnalités sont nécessaires pour répondre aux besoins des utilisateurs. Premièrement, les points de départs et d'arrivées peuvent être des coordonnées et ne sont pas obligatoirement des arrêts de transports en commun, ce qui implique de calculer des plus courts chemins piétons qui sont pris en entrée du calculateur d'itinéraires. Deuxièmement, l'ajout de multimodalité pour permettre l'utilisation de P+R en conjonction avec un trajet en voiture, ou l'utilisation de mode dit doux pour permettre un rabattement ou une fin de trajets en utilisant du vélo ou des trottinettes électriques. Troisièmement, l'ajout d'améliorations diverses et variées, l'ajout des horaires de prochains passages sur les moyens de transport

renvoyés, l'affichage de perturbations sur la ligne (arrêts non desservis, travaux ...), des filtres pour rendre les résultats plus cohérents entre eux (comparaison entre résultats en transports en commun et résultats multimodaux), et beaucoup d'autres améliorations.

Malgré l'ajout de toutes ces fonctionnalités, les résultats obtenus dans la sous-section 3.6.2 pour certains réseaux de transports européens sont retrouvés pour les réseaux en production chez *Instant System*. De plus, un des avantages de l'algorithme CSA que l'on retrouve également dans l'algorithme GDCSA est une certaine lisibilité. Le code du calculateur est fluide à la lecture et facilement évolutif, un autre ingénieur peut travailler dessus sans connaissance de la théorie sur les calculs d'itinéraires pour les transports en commun.

3.8 Synthèse

Nous avons développé une amélioration de l'algorithme CSA pour calculer la variante Pareto *range* du calcul d'itinéraires pour les transports en commun en découpant le graphe en une partition de zone, en ajoutant une borne supérieure et inférieure pour guider la recherche et éviter d'étudier des zones qui ne sont pas nécessaires pendant la recherche. L'algorithme GDCSA utilise des bornes inférieures précalculées entre les zones, et des bornes supérieures qui sont calculées pendant le calcul d'itinéraire. Il utilise l'algorithme PRVCSA comme pièce maîtresse qui prend en entrée la liste triée des connexions des zones ouvertes. Ainsi l'implémentation de l'algorithme GDCSA nécessite seulement une variante Pareto *range* de l'algorithme CSA ainsi qu'une variante d'arrivée au plus tôt de l'algorithme CSA. De plus, l'algorithme est facilement extensible, car l'ajout ou la suppression de critères est simplifié par le fait que les calculs de bornes supérieures et inférieures ne sont pas affectées par les critères Pareto.

Nos expériences montrent que sur de grands réseaux métropolitains réalistes et des réseaux ferroviaires nationaux, GDCSA est au plus 7 fois plus rapide que PRVCSA en calculant un front de Pareto à quatre critères. Pour avoir ces gains, les temps de précalcul sont raisonnables avec en moyenne 1 heure de précalcul et l'espace mémoire est en moyenne de 150 Mo. De plus, le prétraitement est robuste car nous calculons des bornes inférieures, ce qui nous évite de recalculer à chaque changement du réseau et le prétraitement est réutilisable pour plusieurs requêtes. Une utilisation en temps réel peut être envisagée pour la plupart des réseaux avec des temps de réponse satisfaisants et même si le réseau de Paris est encore trop lent, des progrès ont été réalisés.

Pour nos futurs travaux, une direction intéressante à poursuivre serait dans le calcul de la borne inférieure afin de les rapprocher de l'optimal, en y ajoutant une dimension temporelle. Par exemple en calculant une borne inférieure pour chaque intervalle de temps de 4 heures, afin d'obtenir des bornes inférieures plus proches des temps de trajet réel.

Du côté du partitionnement, une alternative est d'utiliser d'autres algorithmes de partitionnement ou de continuer d'utiliser METIS ou KaHiP, mais avec une configuration différente de celle de base. Comme montré dans [Maue et al., 2010], d'autres méthodes de partitionnement existent et certaines ont de meilleurs résultats que METIS sur les réseaux routiers. La caractéristique principale d'un partitionnement qui fournit de bons résultats est un diamètre faible pour chaque zone : cette piste est toutefois à vérifier pour des réseaux de transports en commun.

Un autre axe d'amélioration est l'utilisation d'une structure de données pour approximer le front de Pareto, afin de minimiser le nombre d'étiquettes stockées pour chaque arrêt, ce qui va diminuer le temps nécessaire pour maintenir le front de Pareto et donc diminuer le temps de réponse. Cependant le choix des solutions intermédiaires à écarter est difficile pour éviter de perdre

des résultats qui peuvent faire partie du front de Pareto à l'arrêt d'arrivée, un algorithme similaire existe, mais il utilise l'algorithme RAPTOR [Delling et al., 2019].

CHAPITRE 4

k plus courts chemins simples appliqué aux réseaux de transports en commun

Le calcul d'itinéraire dans un réseau de transports en commun a intéressé de nombreux chercheurs pendant la dernière décennie et plusieurs algorithmes ont été développés. Cependant, la plupart des méthodes ne retournent qu'un seul ou un nombre limité de résultats, ce qui n'est pas souhaitable dans un contexte de transports. Dans ce chapitre, nous considérons le problème de trouver k chemins d'arrivée au plus tôt dans un réseau de transports en commun à partir d'un départ jusqu'à une arrivée, c'est-à-dire, un chemin d'arrivée au plus tôt du départ à l'arrivée, un deuxième chemin d'arrivée au plus tôt, etc., jusqu'au $k$ème chemin d'arrivée au plus tôt. Pour ce faire, nous proposons un algorithme, appelé Yen - Public Transport, qui étends au réseau de transports en commun l'algorithme de Yen pour trouvé les "top-k" chemins simples dans un graphe. De plus, nous proposons un algorithme amélioré, appelé Postponed Yen - Public Transport (PY-PT), permettant un gain de temps considérable en pratique. Nos expériences sur plusieurs réseaux de transports en commun montrent qu'en pratique, PY-PT est plus rapide que Y-PT d'un ordre de grandeur.

Journey planning in a public transportation network has interested many researchers during the last decade and several algorithms have been developed. However, most methods return only one or a limited number of results, which is undesirable in a transportation context. In this chapter, we consider the problem of finding k earliest arrival paths in a transit network from a departure to an arrival, i.e., an earliest arrival path from departure to arrival, a second earliest arrival path, etc., up to the kth earliest arrival path. For this purpose, we propose an algorithm, called Yen - Public Transport, which extends Yen's algorithm for finding the "top-k" simple paths in a graph to the public transportation network. In addition, we propose an improved algorithm, called Postponed Yen - Public Transport (PY-PT), which saves considerable time in practice. Our experiments on several public transport networks show that in practice, PY-PT is one order of magnitude faster than Y-PT.

4.1 Introduction

Comme nous l'avons vu dans le chapitre précédent, le calcul d'itinéraire pour les transports en commun multiobjectif est un problème complexe. L'utilisation d'un front de Pareto permet de répondre à ce problème. Cependant le maintien d'un front de Pareto avec de multiples critères pour chaque arrêt du graphe est très coûteux en temps. Une alternative est donc nécessaire. Si on considère uniquement le problème de la recherche d'un ensemble de plus courts chemins, alors il existe d'autres méthodes. La plus populaire, le problème des k plus courts chemins, énumère les plus courts chemins entre un point de départ et un point d'arrivée.

On peut considérer deux types de variantes : la première considère tous les types de chemins, alors que la deuxième considère uniquement les chemins simples, c'est-à-dire sans boucle. Les algorithmes pour résoudre ces deux variantes utilisent la notion de détour, voir la figure 4.1 comme exemple. Les algorithmes sont amorcés avec un premier plus court chemin, ici $C = (s, u, v, w, t)$. Un détour D de C est un chemin qui partage une première partie des nœuds de P appelé préfixe, ici (s, u). L'arc (v, x) est appelé l'arc source du détour, avec v le point interne du détour et x le point externe du détour. Ainsi un détour de C est $D = (s, u, v, x, t)$.

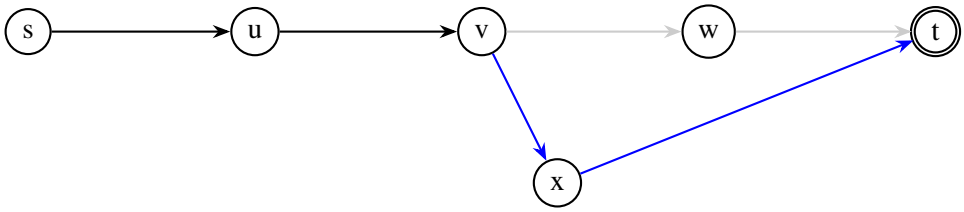


FIGURE 4.1 – Exemple d'un détour.

La première variante est bien résolue avec l'algorithme d'Eppstein [Eppstein, 1998]. L'algorithme se déroule en 3 étapes :

1. Il commence par calculer un arbre de plus court chemin dans le transposé qui a pour racine t et donc en même temps un plus court chemin C entre s et t.
2. Pour chaque arc (v, w) du chemin C, un détour est calculé avec v comme point interne du détour et comme point externe du détour un nœud x différent de w, puis le chemin de x vers t est trouvé avec l'arbre de plus court chemin déjà calculé.
3. Ainsi, le 2e plus court chemin est un détour de C avec la plus petite distance et ainsi de suite jusqu'à atteindre k plus courts chemins.

Le calcul des k plus courts chemins se fait en un temps $\mathcal{O}(k + |A| + |X| \log |X|)$.

La deuxième variante peut être résolue en utilisant l'algorithme d'Eppstein et en ne gardant que les chemins simples. Cependant cette méthode est peu efficace. Une autre méthode est l'algorithme de Yen [Yen, 1971], qui est très similaire à la première méthode, mais remplace l'utilisation d'un arbre de plus court chemin par des appels à un algorithme de plus court chemin. L'étape numéro 2 est quelque peu modifiée pour obtenir un chemin simple : avant l'appel à l'algorithme de plus court chemin, tous les nœuds du préfixe sont supprimés du graphe, garantissant ainsi que le détour soit simple. Le calcul des k plus courts chemins se fait en un temps $\mathcal{O}(k|X|(|A| + |X| \log |X|))$.

Puisque cette complexité en temps est la meilleure connue pour ce problème, des efforts significatifs sont faits sur la conception d'algorithmes pour résoudre de manière efficace le

problème des k plus courts chemins simples lors de problème réels [Kurz and Mutzel, 2016, Al Zoobi et al., 2020, Al Zoobi et al., 2021a].

Dans la pratique, le problème des k plus courts chemins doit renvoyer uniquement des chemins simples, car le graphe n'a pas de poids négatif et une boucle ne peut que faire perdre du temps. À partir de maintenant, nous considérons que tout plus court chemin est simple.

Un réseau routier peut être facilement modélisé par un graphe orienté. Ainsi, trouver k « bons » (plus court, plus rapide ...) chemins dans un réseau routier se fait en utilisant n'importe quel algorithme de k plus courts chemins. Malheureusement, ce problème est plus complexe pour un réseau de transports en commun.

- Les transports en commun sont extrêmement dépendants du temps, c'est-à-dire que certains arcs du réseau ne peuvent être traversés qu'à certains moments spécifiques dans le temps.
- Plusieurs autres critères d'optimisation doivent être considérés pour les réseaux de transports en commun comme l'heure de départ, le nombre de transferts, la distance de marche et beaucoup d'autres.

Vo et al. [Vo et al., 2015] proposent un graphe dépendant du temps pour modéliser un réseau de bus. Ensuite, ils modifient l'algorithme de Yen pour trouver des chemins alternatifs dans ce réseau. Plus précisément, ils choisissent un ensemble de chemins alternatifs, des chemins qui partagent une sous-partie de leurs arcs, parmi ceux renvoyés par l'algorithme de Yen modifié. Ils utilisent la variante Time Dependent de l'algorithme de Dijkstra (TDD) [Schulz et al., 2000] pour calculer les détours arrivant le plus tôt d'un chemin dans le réseau de transports en commun. Ils évaluent leur méthode sur un unique réseau de 4 000 nœuds et 8 000 connexions, avec en moyenne un temps de réponse de 1 seconde pour trouver 5 chemins.

D'une autre manière, Scano et al. [Scano et al., 2015] modélisent un réseau de transports comme un graphe dirigé avec une étiquette sur les arcs, où une étiquette est un objet composé du mode de transports (piéton, voiture, bus ...) et un temps de trajet. Ce modèle fusionne le réseau routier et le réseau de transports en commun. Ensuite, les algorithmes de Yen et de Eppstein sont modifiés pour les adapter à leur modèle. Dans les deux algorithmes, un algorithme ressemblant au Dijkstra appelé *Dijkstra Regular Language Constraint* (DRegLC) [Barrett et al., 2008] est utilisé pour répondre au problème d'arrivée au plus tôt. De plus, l'algorithme *Iterative Enumeration* (IEA) est proposé pour extraire uniquement des chemins simples en utilisant l'algorithme d'Eppstein : c'est-à-dire en utilisant l'algorithme d'Eppstein de k plus courts chemins comme itérateur et en ne sélectionnant que des chemins simples, des chemins qui ne visitent pas deux fois le même nœud.

De manière expérimentale, Scano et al. montrent que leur algorithme IEA est plus rapide qu'un algorithme de Yen pour le réseau de transport de Toulouse, qui a 75 000 nœuds, 500 000 arcs routiers et 43 000 arcs de transports en commun. Sur ce réseau, le temps de réponse moyen de l'algorithme de Yen pour trouver 100 chemins est de 250 secondes, alors qu'il n'est que de 0.6 seconde pour l'algorithme IEA. Cependant, l'algorithme IEA est une heuristique qui ne renvoie pas forcément l'ensemble complet des résultats. De plus, une duplication des parties en transports en commun dans les résultats est possible, car en utilisant leur modèle de graphe avec des étiquettes, une partie des résultats peut ne différer que par des arcs piétons tout en partageant les mêmes parties en transports en commun. Ce qui est indésirable pour une utilisation qui doit renvoyer des chemins en transports en commun diversifiés.

Pour répondre au problème des k plus courts chemins dans un réseau de transports en commun, nous avons créé un algorithme qui utilise les avancées récentes dans le calcul d'itinéraire pour

les transports en commun et les caractéristiques des réseaux de transports en commun. Ainsi les chemins dans un réseau de transports en commun ont un nombre de nœuds très inférieur au nombre de nœuds d'un chemin dans un réseau routier, car le nombre de nœuds dans un réseau de transports en commun est beaucoup plus faible que dans un réseau routier. Par exemple, le réseau routier de l'Allemagne a plus 1 000 000 de nœuds, tandis que son réseau ferroviaire en a moins de 100 000. Nous utilisons le modèle de table horaire des réseaux de transports en commun pour proposer une adaptation performante de l'algorithme de Yen, appelé *Yen - Public Transport* (Y-PT), qui utilise le CSA et est beaucoup plus rapide que n'importe quelle adaptation du Dijkstra, en opposition à [Scano et al., 2015, Vo et al., 2015]. Puis, pour notre contribution principale, nous avons proposé un nouvel algorithme, appelé *Postponed Yen - Public Transport* (PY-PT), qui utilise une borne inférieure sur l'heure d'arrivée d'un détour d'un des k plus courts chemins simples, et exploite plus les calculs intelligents, en retardant le calcul de ce détour. Le but final étant de ne pas le faire.

Des résultats expérimentaux sur plusieurs réseaux de transports en commun et de trains montrent que les temps de réponse de notre adaptation de l'algorithme sont acceptables en pratique. De plus, sur ce même jeu de données, l'algorithme PY-PT est plus rapide d'un facteur de 10 à 30 en moyenne comparé à l'algorithme Y-PT.

Finalement, nous évaluons la similarité, et plus intéressant encore la dissimilarité, des chemins renvoyés par nos algorithmes. Nous pouvons montrer de manière expérimentale que nos algorithmes peuvent extraire des plus courts chemins simples qui sont mutuellement dissimilaires.

L'algorithme PY-PT a été publié et présenté à la conférence *11th International Conference on Operations Research and Enterprise Systems* (ICORES) [Al-Zoobi et al., 2022] et a gagné le *Best Paper Award*.

4.2 Préliminaires

Dans cette section, nous formalisons les données nécessaires ainsi que les algorithmes utilisés dans ce chapitre. Nous utilisons la même formalisation que dans [Dibbelt et al., 2018] pour le CSA et dans [Al Zoobi et al., 2020] pour l'algorithme de Yen.

4.2.1 Définitions et notations sur les graphes

Soit $G = (X, A)$ un graphe dirigé avec $n = |X|$ nœuds et $m = |A|$ arcs.
Soit $N^+(v) = \{w \in V \mid (v, w) \in A\}$ l'ensemble des *out-neighbors* d'un nœud.
Soit $\ell : A \to \mathcal{R}^+$ la fonction de distance pour les arcs.

Définition 4.2.1. Pour tout $s, t \in X$, un chemin d'une source s à un puits t dans G est une séquence $P = (s = v_0, v_1, \cdots, v_r = t)$ de nœuds avec $(v_i, v_{i+1}) \in A$ pour tout $0 \leq i < r$.

Définition 4.2.2. Un arc (u, v) appartient à un chemin P, que l'on note $(u, v) \in P$, si et seulement si u et v sont deux nœuds consécutifs de P, c'est-à-dire, il y a $0 \leq i < l$ tel que $u_i = u$ et $u_{i+1} = v$.

Définition 4.2.3. Un chemin est simple si tous les nœuds qu'il contient sont distincts, c'est-à-dire, $v_i \neq v_j$ pour tout $0 \leq i < j \leq l$.

Définition 4.2.4. La longueur d'un chemin P est la somme de la longueur des arcs de P, $\ell(P) = \sum_{0 \leq i < l} \ell(v_i, v_{i+1})$.

Définition 4.2.5. La distance $d(s,t)$ entre deux nœuds $s,t \in X$ est la longueur d'un plus court $s-t$ chemin, c'est-à-dire, un chemin avec la plus courte distance entre tous les s-t chemins.

Définition 4.2.6. Soit deux chemins $P = (v_0, \cdots, v_r)$ et $Q = (w_0, \cdots, w_p)$, nous notons par $P.Q$ le v_0-w_p chemin qui est le résultat de la concaténation de P et Q. C'est-à-dire :

$$P.Q = (v_0, \cdots, v_r, w_0, \cdots, w_p) = (v_0, \cdots, v_r, Q) = (P, w_0, \cdots, w_p)$$

4.2.2 Définitions et notations pour les k plus courts chemins

Voici des définitions et notations propres au problème des k plus courts chemins.

Définition 4.2.7. Soit $s,t \in X$, un ensemble de *top-k* plus courts s-t chemins simples est un ensemble S de s-t chemins simples tels que $|S| = k$ et $\ell(P) \leq \ell(P')$ pour tout s-t chemin $P \in S$ et s-t chemin $P' \notin S$.

Le problème des k plus courts chemins simples prend en entrée un graphe dirigé $G = (X, A)$, une fonction de distance pour les arcs $\ell : A \to \mathcal{R}^+$ et une paire de nœuds $s,t \in V$ et demande de trouver un ensemble de *top-k* plus courts s-t chemins simples.

Définition 4.2.8. Soit $P = (v_0, v_1, \cdots, v_i, v_{i+1}, \cdots, v_l)$ un chemin de G. Soit $0 \leq i < l$, un chemin $P' = (v_0, \cdots, v_i, v', v'_1, \cdots, v'_r = v_l)$ tel que $v' \neq v_{i+1}$ est appelé un détour de P à v_i.

À noter que ni P ni P' n'ont besoin d'être simple. Cependant, si P' est simple, alors il sera appelé détour simple de P à v_i. De plus, P' est appelé un plus court détour (simple) à v_i si et seulement si P' est un détour avec la plus courte distance parmi tous les détours (simples) de P à v_i.

Définition 4.2.9. Le sous-chemin $\pi_i = (v_0, \cdots, v_{i-1})$ de P qui commence à s et qui finit à v_{i-1} avec $0 \leq i \leq l$ est appelé le chemin i-préfixe de P (le chemin 0-préfixe de n'importe quel chemin est un chemin vide).

Un détour avec interdiction est défini par : soient $P = (v_0, v_1, \cdots, v_l)$ un chemin, S un ensemble de v_0-v_l chemins, avec $P \notin S$, $\pi_i = (v_0, \cdots, v_{i-1})$ le i-préfixe de P. Un détour avec interdiction de P à i se calcule en enlevant les nœuds de π_i de G, en enlevant tous les arcs (v_i, w) tel que S contient un chemin avec $(v_0, \cdots, v_i, w)$ comme $i+1$-préfixe et en enlevant l'arc (v_i, v_{i+1}). Ainsi $P' = (v_0, \cdots, v_i, v', v'_1, \cdots, v'_r = v_l)$ est un détour de P à v_i et $P' \notin S$.

Si P est un chemin simple, alors P' est un détour simple avec interdiction, car le v_i-v_t chemin est calculé après que π_i ait été enlevé. De plus, $(v_i, v_{i+1}) \notin P'$, car l'arc (v_i, v_{i+1}) de P est enlevé avant le calcul du v_i-v_t chemin et la construction de P'. Ainsi, P' est un plus court détour simple avec interdiction de P à i.

4.2.3 L'algorithme de Yen

Nous allons maintenant décrire l'algorithme de Yen, en utilisant la même formalisation que dans [Al Zoobi et al., 2020], pour calculer un ensemble de *top-k* plus courts chemins simples dans G. Par souci de simplicité, nous supposons qu'il y a au moins k s-t chemins simples dans G.

L'algorithme de Yen commence par calculer un plus court s-t chemin $P_0 = (s = v_0, v_1, \cdots, v_l = t)$ en utilisant l'algorithme de Dijkstra. À noter que P_0 est simple, car les poids

des arcs de G sont strictement positifs. Intuitivement, un deuxième plus court s-t chemin simple est un plus court détour simple avec interdiction de P_0 à un de ces nœuds.

L'algorithme de Yen calcule un deuxième plus courts chemins en 3 étapes :

- L'algorithme de Yen commence par calculer pour chaque nœud v_i dans P_0, un plus court détour simple de P_0 à v_i avec interdiction, appelé C_i. À noter que l'index i, appelé ensuite index de déviation, où le chemin $(v_0, \cdots, v_{i-1}, v_i, v', v'_1, \cdots, v'_r = v_l)$ dévie de P_0 à i est gardé de manière explicite, c'est-à-dire qu'un chemin est stocké avec son index de déviation.
- Ensuite, C_i est ajouté à l'ensemble des candidats C, initialement vide, pour chaque $0 \leq i < l$.
- Une fois que C_i est ajouté à l'ensemble des candidats pour tout $0 \leq i < l$, alors le chemin avec la plus courte distance dans l'ensemble des candidats est un deuxième plus court s-t chemin simple.

Maintenant, supposons qu'un ensemble S de top-k' (avec $0 < k' < k$) plus courts s-t chemins simples ait été calculé et que l'ensemble des candidats contiennent un ensemble de plus court s-t chemins simples tel qu'il existe un plus court chemin $Q \in C$ avec $S \cup \{Q\}$ un ensemble de top-$(k'+1)$ plus courts s-t chemins simples. Soit $R = (v_0 = s, \cdots v_j, \cdots, v_r = t)$ un chemin dans l'ensemble de candidats avec une distance minimale et soit j son index de déviation. De manière similaire à la méthode pour trouver un deuxième plus court chemin, l'algorithme de Yen itère sur les nœuds v_i ($j \leq i < r$) de R. Pour chaque nœud v_i, un plus court détour simple avec interdiction de R à v_i est ajouté à l'ensemble des candidats, car un de ces détours est peut être un $k'+1$ème plus court s-t chemins simples. Ce processus est répété jusqu'à ce que k chemins soient trouvés, c'est-à-dire quand $k' = k$.

Ainsi, pour chaque chemin R qui est extrait de l'ensemble des candidats, $\mathcal{O}(|X(R)|)$ appels à l'algorithme Dijkstra sont faits. Cela donne une complexité en temps de $\mathcal{O}(kn(m + n \log n))$.

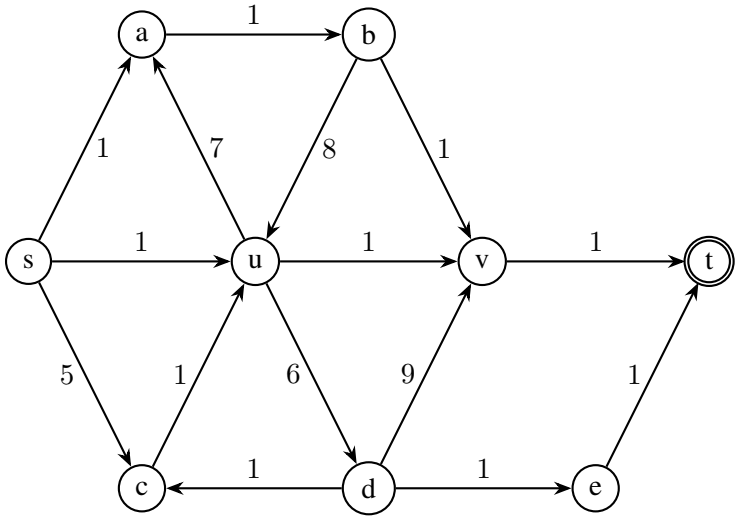


FIGURE 4.2 – Exemple pour l'algorithme de Yen.

Exemple Nous allons utiliser la figure 4.2 pour calculer 3 plus courts chemins. L'algorithme de Yen commence par calculer un premier plus court chemin (s, u, v, t) avec une longueur de 3,

l'ajoute à l'ensemble des solutions. Le chemin (s, u, v, t) est ajouté à l'ensemble des solutions S et ensuite l'algorithme de Yen calcule les plus courts détours simples avec interdiction du chemin (s, u, v, t) :

1. Pour le nœud s, l'algorithme enlève du graphe le préfixe du chemin, qui est vide, et enlève l'arc (s, u), puis calcule un plus court chemin dans le graphe modifié qui est concaténé au préfixe, ce qui nous renvoie le chemin (s, a, b, v, t) d'une longueur de 7.
2. Pour le nœud u, l'algorithme enlève du graphe le préfixe du chemin, qui est s, et enlève l'arc (u, v), puis calcule un plus court chemin dans le graphe modifié qui est concaténé au préfixe, ce qui nous renvoie le chemin (s, u, d, e, t) d'une longueur de 9.
3. Pour le nœud v, l'algorithme enlève du graphe le préfixe du chemin, qui est s, u, et enlève l'arc (v, t), puis calcule un plus court chemin dans le graphe modifié qui est concaténé au préfixe, ce qui n'est pas possible donc aucun chemin n'est renvoyé.

Les chemins (s, a, b, v, t) et (s, u, d, e, t) sont ajoutés à l'ensemble des candidats C. L'algorithme de Yen récupère de C le chemin avec la plus petite longueur, le chemin (s, a, b, v, t), qui est le 2e plus court chemin entre s et t et l'ajoute à l'ensemble des solutions. Ensuite l'algorithme de Yen calcule les plus courts détours simples avec interdiction du chemin (s, a, b, v, t) :

1. Pour le nœud s, l'algorithme enlève du graphe le préfixe du chemin, qui est vide, enlève l'arc (s, a) et enlève également l'arc (s, u) pour ne pas recalculer un chemin déjà dans S. Puis l'algorithme calcule un plus court chemin dans le graphe modifié qui est concaténé au préfixe, ce qui nous renvoie le chemin (s, c, u, v, t) d'une longueur de 8.
2. Pour le nœud a, l'algorithme enlève du graphe le préfixe du chemin, qui est s, enlève l'arc (a, b). Puis l'algorithme calcule un plus court chemin dans le graphe modifié qui est concaténé au préfixe, ce qui n'est pas possible donc aucun chemin n'est renvoyé.
3. Pour le nœud b, l'algorithme enlève du graphe le préfixe du chemin, qui est s, a, enlève l'arc (b, v). Puis l'algorithme calcule un plus court chemin dans le graphe modifié qui est concaténé au préfixe, ce qui nous renvoie le chemin (s, a, b, u, v, t) d'une longueur de 12.
4. Pour le nœud v, l'algorithme enlève du graphe le préfixe du chemin, qui est s, a, b, enlève l'arc (v, t). Puis l'algorithme calcule un plus court chemin dans le graphe modifié qui est concaténé au préfixe, ce qui n'est pas possible donc aucun chemin n'est renvoyé.

Les chemins (s, c, u, v, t) et (s, a, b, u, v, t) sont ajoutés à l'ensemble des candidats C. L'algorithme de Yen récupère de C le chemin avec la plus petite longueur, le chemin (s, c, u, v, t), qui est le 3e plus court chemin entre s et t et l'ajoute à l'ensemble des solutions.

Quand l'algorithme calcule tous les plus courts détours du $i$ème plus court chemin :

- soit un $i + 1$ème plus court chemin fait partie des détours calculés ;
- soit un $i + 1$ème plus court chemin est dans l'ensemble des candidats.

Nous pouvons voir le défaut principal de l'algorithme de Yen dans cet exemple. Pour calculer 3 plus courts chemins, il faut 8 appels à un algorithme de plus courts chemins, ce qui implique que plus les chemins sont longs, plus le nombre d'appels est élevé.

4.2.4 Table horaire et itinéraires

Dans cette partie, nous utilisons les mêmes structures de données que dans la sous-section 2.4.1. La table horaire utilisée est toujours la même, la différence vient principalement de la définition des itinéraires.

De la même manière que précédemment (sous-section 2.4.2), les itinéraires vont être définis plus en détail afin de permettre une meilleure utilisation des connexions qui les composent. Ainsi un itinéraire est composé alternativement de chemins piétons et de *legs*, $J = (f^0, l^0, f^1, l^1 \dots f^{r-1}, l^r, f^r)$ où $l^i = (c^i_0, \cdots, c^i_{\delta_i})$. Dans cette section, nous allons décrire les itinéraires de cette manière $J = (f^0, c^0, c^1, \cdots, c^\alpha, f^1, c^{\alpha+1}, \cdots, f^{r-1}, c^{\gamma+1}, \cdots c^\phi, f^r)$ où $c^0 = c^0_0, c^1 = c^0_1, \cdots, c^\phi = c^r_{\delta_r}$.

Définition 4.2.10. Soient deux arrêts s et t dans S, un s-t itinéraire J est un itinéraire $(f^0, c^0, \cdots, c^\phi, f^r)$ tel que f^0 commence à s et f^r finit à t.

Définition 4.2.11. Nous définissons l'heure de départ d'un itinéraire $dep_t(J)$ comme l'heure de départ de son premier chemin piéton, formellement $dep_t(J) = c^0_{dep\_time} - f^0_{dur}$.

Définition 4.2.12. De manière similaire, l'heure d'arrivée d'un itinéraire $arr_t(J)$ est l'heure d'arrivée de son dernier chemin piéton, c'est-à-dire $arr_t(J) = c^\phi_{arr\_time} + f^r_{dur}$.

Définition 4.2.13. Un itinéraire est appelé simple, s'il ne visite pas deux fois le même arrêt.

Définition 4.2.14. La concaténation de deux itinéraires $J = (f^0, l^0, \cdots, l^r, f^r)$ et $J' = (f'^0 = f^r, l'^0, \cdots, l'^\ell, f'^\ell)$, tel que $f^r = f'^0$ et $arr_t(J) \leq dep_t(J')$, est un itinéraire qui commence par f^0 suit J jusqu'à f^r et ensuite J' jusqu'à f'^ℓ. Nous notons $J" = J.J' = (f^0, l^0, \cdots, f^r = f'^0, \cdots, l'^\ell, f'^\ell)$.

Définition 4.2.15. Soit un itinéraire $J = (f^0, c^0, \cdots, c^i, \cdots, c^\phi, f^r)$, un itinéraire $Q = (f'^0, c'^0, \cdots, c'^i, \cdots, c'^w, f'^\ell)$ est appelé un *détour* de J à i si $f'^0 = f^0, c'^0 = c^0, \cdots, c'^{i-1} = c^{i-1}$ mais $c'^i \neq c^i$ et $f'^\ell_{arr\_stop} = f^r_{arr\_stop}$.

Définition 4.2.16. Si Q est simple, il est appelé un détour simple de J à i, et Q est appelé un détour d'arrivée au plus tôt (simple) de J à i, si $arr_t(Q) \leq arr_t(Q')$ pour chaque détour (simple) Q' de J à i.

Définition 4.2.17. Deux itinéraires sont égaux si et seulement si tous leurs attributs sont identiques.

Notation 4.2.1. Nous notons par $\mathcal{J}^{\tau_0,\tau_{max}}_{s,t}$ l'ensemble des s-t itinéraires simples qui commencent de s après t_0 et atteignant t avant t_{max}, c'est-à-dire $\mathcal{J}^{\tau_0,\tau_{max}}_{s,t} = \{J$ tel que J est un s-t itinéraire simple avec $dep_t(J) \geq \tau_0$ et $arr_t(J) \leq \tau_{max}\}$.

4.2.5 Profile Connection Scan Algorithm

Dans cette partie, nous utilisons la même définition que dans la sous-section 2.4.5. L'algorithme CSA résout le problème d'arrivée au plus tôt avec un arrêt de départ s, un arrêt d'arrivée t, une heure de départ τ_0 et une heure d'arrivée τ_{max}.

Le *Profile Connection Scan Algorithm* (PCSA) calcule un *mapping* entre l'heure de départ en partant d'un arrêt sur l'heure d'arrivée à l'arrêt d'arrivée.En effet, le problème d'arrivée au plus tôt par plage horaire résout simultanément le problème d'arrivée au plus tôt pour toutes les heures de départ.

Comparé au CSA, l'algorithme PCSA itère sur les connexions triées par heure de départ décroissante, ce qui amène à la résolution du problème de calcul d'itinéraires de tous les points du

graphe vers un point d'arrivée. L'algorithme PCSA construit les itinéraires du plus tard au plus tôt et exploite le fait qu'un itinéraire arrivant plus tôt ne peut avoir que des itinéraires arrivant tard comme prochaine partie d'itinéraire. L'algorithme PCSA a un temps de réponse qui est une cinquantaine de fois plus lent que l'algorithme CSA [Dibbelt et al., 2018], ce qui est acceptable, car il résout le problème de calcul d'itinéraires de tous les points du graphe vers un point d'arrivée pour toutes les heures possibles de départ.

À noter, que l'algorithme PCSA donne, pour chaque arrêt d jusqu'à l'arrêt t, au moins un s-t itinéraire d'arrivée au plus tôt qui part après τ_0 et atteint t avant τ_{max}.

Soit M le résultat du PCSA, $M_{s,t}^{\tau_0,\tau_{max}}$ un résultat d'arrivée au plus tôt qui commence à s et finit à t avec une heure de départ après τ_0 et arrivant avant τ_{max}.

4.3 Problème des k chemins simples d'arrivée au plus tôt

Dans cette section, notre but est de trouver k chemins simples d'arrivée au plus tôt à partir d'une source vers un puits. Formellement, le problème prend en entrée une table horaire $\mathcal{T} = (S, T, C, F)$, un arrêt source et un arrêt puits $s, t \in S$, une heure de départ τ_0, une heure maximale d'arrivée τ_{max} et un entier k. Et veut en réponse un ensemble $\mathcal{J}^* = \{J_1, J_2, \cdots, J_k\}$ de top-k s-t chemins simples d'arrivée au plus tôt, c'est-à-dire $J_i \neq J_j$ pour $0 \leq i < j \leq k$, et pour tout $J \in \mathcal{J}^*$, $J' \in \mathcal{J}_{s,t}^{\tau_0,\tau_{max}}$, $arr_\tau(J) \leq arr_\tau(J')$.

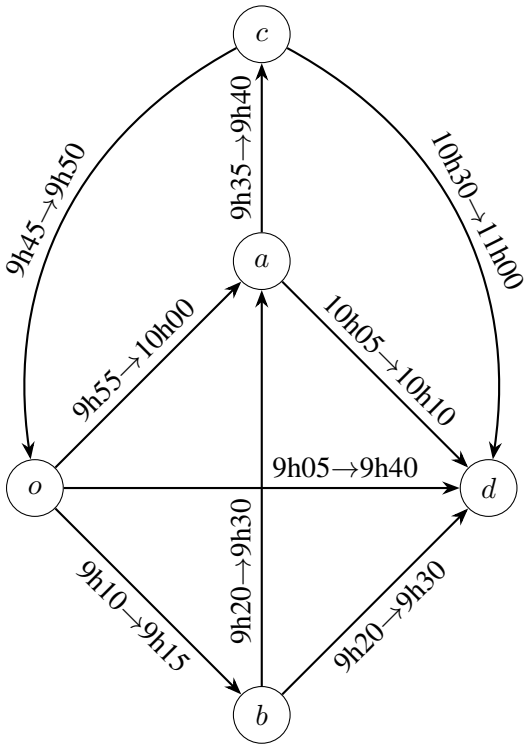


FIGURE 4.3 – Réseau exemple pour les k chemins d'arrivée au plus tôt.

4.3.1 Exemple

Dans l'exemple de la figure 4.3, nous recherchons les 4 premiers chemins d'arrivée au plus tôt de o à d partant après 9h00.

- Le chemin qui arrive le plus tôt $J_0 = (o, b, d)$ arrive à d à 9h30. Il atteint d en passant par b, le passager arrive à b à 9h15 et attend 5 minutes avant de prendre la connexion qui va de b à d.
- Le deuxième chemin $J_1 = (o, d)$ arrive à 9h40 et va directement de o à d.
- Le troisième chemin $J_2 = (o, b, a, d)$ arrive à 10h10 et va de b à a et finalement de a à d, le passager arrive à b à 9h15, attend 10 minutes puis prend la connexion qui va de b à a, arrive à 9h30 et attend 35 minutes avant de monter dans la connexion qui va de a à d.
- Le quatrième chemin $J_3 = (o, a, d)$ arrive à 10h10 et va de o à a puis à d.

À noter que le chemin $J_{ns} = (o, b, a, c, o, a, d)$ qui arrive à 10h10 ne fait pas partie des solutions, car il n'est pas simple, puisqu'il visite l'arrêt o deux fois. À noter également, qu'il y a d'autres o-d chemins dans cet exemple, qui arrivent après 10h10. À la fin, $\{J_0, J_1, J_2, J_3\}$ sont les 4 o-d chemins simples d'arrivée au plus tôt.

4.4 L'algorithme de Yen appliqué aux transports en commun (Y-PT)

L'algorithme Y-PT résout le problème des k chemins simples d'arrivée au plus tôt. C'est pourquoi il prend en entrée une table horaire $\mathcal{T} = (S, T, C, F)$, un arrêt source et un arrêt de destination $s, t \in S$, une heure de départ τ_0, une heure maximale d'arrivée τ_{max} et un entier k. Ici $\tau_{max} = \tau_0 + 36$h car un calcul d'itinéraire renvoie un résultat qui doit arriver dans la même journée, ainsi 36 heures est une borne supérieure pertinente qui peut prendre en compte les trains de nuit par exemple. Il renvoie un ensemble de résultats $\{J_1, J_2, \cdots, J_k\}$ de *top-k s-t* chemins simples d'arrivée au plus tôt dans $\mathcal{T}$.

De manière grossière, l'algorithme Y-PT commence par calculer un premier chemin d'arrivée au plus tôt, itère sur l'ensemble de ces connexions pour en calculer les détours simples d'arrivée au plus tôt avec interdiction et ajoute le détour avec la plus petite heure d'arrivée dans l'ensemble des résultats. Ensuite, l'algorithme Y-PT répète ce processus jusqu'à ce que k chemins soient ajoutés à l'ensemble de résultats.

Nous présentons maintenant l'algorithme Y-PT avec son pseudo-code (cf. algorithme 4.1). De manière analogue à l'algorithme de Yen, l'algorithme Y-PT commence par calculer un chemin d'arrivée au plus tôt J_0 et l'ajoute à l'ensemble des candidats. L'algorithme initialise l'ensemble des résultats par l'ensemble vide. Ensuite l'algorithme va itérer jusqu'à ce que k chemins d'arrivée au plus tôt soient extraits ou qu'il n'y ait plus de chemins candidats. L'algorithme extrait de l'ensemble des candidats un élément J avec une heure d'arrivée minimale, puis l'ajoute à l'ensemble des résultats.

Soit C_J la séquence des connexions du chemin J. L'algorithme itère sur les connexions de C_J en commençant à l'index de déviation de J, enlève le i-préfixe du chemin, enlève également la connexion c^i pour tous les chemins, dans l'ensemble des résultats, qui commencent avec les connexions $(c^0, c^1, \cdots, c^{i-1})$ et enlève la connexion c^i du chemin C_J. Ensuite, en utilisant un CSA, l'algorithme Y-PT calcule un chemin d'arrivée au plus tôt Q à partir de $c^{i-1}_{arr\_stop}$ jusqu'à t avec $c^{i-1}_{arr\_time}$ comme heure de départ. Soit J_{new} la concaténation du préfixe de J et de Q, le chemin J_{new} est ajouté à l'ensemble des candidats avec i comme index de déviation.

Algorithme 4.1 Algorithme de Yen appliqué aux transports en commun (Y-PT)

$J_0 \leftarrow CSA(\mathcal{T}, s, t, \tau_{dep}, \tau_{max})$
$Candidates \leftarrow \{(J_0, 0)\}$
$Output \leftarrow \emptyset$
while $|Output| < k$ and $Candidates \neq \emptyset$ **do**
  $\varepsilon = (J, j) \leftarrow extractMin(Candidates)$
  Let $C_J = (c^0, \cdots, c^\phi)$ be the sequence of connections of J
  add J to $Output$
  for each connection c^i with $j \leq i \leq \phi$ in C_J **do**
    $c_{arr_stop} \leftarrow$the arrival stop of c^{i-1}
    $c_{arr_time} \leftarrow$the arrival time of c^{i-1}
    $\pi = (f^0, c^0, \cdots, c^{i-1})$
    $S_\pi \leftarrow$ the set of stations visited by the connections $(c^0, \cdots, c^{i-1})$
    $C_{dev} \leftarrow \{c'$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c')\}$
    $\mathcal{T}' = (S \setminus S_\pi, T, C \setminus C_{dev}, F)$
    $Q \leftarrow CSA(\mathcal{T}', c_{arr\_stop}, d, c_{arr\_time}, \tau_{max})$
    $J_{new} \leftarrow \pi.Q$
    add (J_{new}, i) to $Candidates$
  end for
end while
Return $Output$

De la même manière que pour l'algorithme de Yen, l'algorithme Y-PT a un gros défaut qui est le nombre d'appels à l'algorithme de calcul de chemin d'arrivée au plus tôt. Ainsi pour calculer un chemin d'arrivée au plus tôt supplémentaire, il faut calculer tous ces détours d'arrivée au plus tôt avec interdiction, ce qui revient à faire autant d'appels que le nombre de connexions du chemin.

4.5 L'algorithme de Yen reporté appliqué aux transports en commun (PY-PT)

Nous présentons maintenant l'algorithme *Postponed Yen* appliqué aux transports en commun (PY-PT) avec son pseudo-code (cf. algorithme 4.2). Il est inspiré du *Postponed Node Classification Algorithm* (PNC) [Al Zoobi et al., 2021a] pour le problème des k plus courts chemins simples.

4.5.1 Idée générale

L'algorithme PY-PT prend les mêmes entrées que l'algorithme Y-PT et renvoie un ensemble de *top-k* chemins simples d'arrivée au plus tôt d'un arrêt source à un arrêt puits dans une table horaire. Cependant, l'ordre d'extraction des chemins n'est pas forcément le même, c'est-à-dire que les chemins renvoyés par Y-PT ne sont pas forcément les mêmes que ceux renvoyés par PY-PT. Ce scénario apparaît quand plusieurs chemins entre la source et le puits ont la même heure d'arrivée.

Le principal inconvénient de l'algorithme Y-PT est le nombre surdimensionné d'appels à l'algorithme CSA. L'algorithme PY-PT, avec l'utilisation d'une borne inférieure sur l'heure d'arrivée de détours simples, repousse les appels à l'algorithme CSA dans le but d'en éviter une partie.

Cette borne inférieure est calculée par un Profile CSA. Au contraire de l'algorithme Y-PT où tous les chemins dans l'ensemble candidats sont simples, ici l'algorithme PY-PT peut ajouter des chemins non simples. Comme montré plus bas, cela correspond à des détours dont le calcul (l'appel à l'algorithme CSA) effectif est reporté.

Nous décrivons maintenant l'algorithme PY-PT en détail. Pour un chemin entre la source s et le puits t avec une heure de départ τ_0, l'algorithme PY-PT utilise en premier le Profile CSA (PCSA). Soit M le *mapping* renvoyé par le PCSA. Le *mapping* M associe à chaque arrêt $s \in S$ et à chaque heure de départ $\tau' \geq \tau_0$ un s-t chemin d'arrivée au plus tôt, s'il est possible d'atteindre t à partir de s en arrivant avant τ_{max} et en partant après τ' (nous mettons $\tau_{max} = \tau_0 + 36$h dans nos expériences).

Algorithme 4.2 Algorithme de Yen reporté appliqué aux transports en commun (PY-PT)

$M \leftarrow PCSA(\mathcal{T}, o, d, \tau_{dep}, \tau_{max})$
$J_0 \leftarrow M_{o,d}^{\tau_{dep},\tau_{max}}$
$Candidates \leftarrow \{(J_0, 0, \zeta = 1)\}$
$Output \leftarrow \emptyset$
while $Candidates \neq \emptyset$ and $|Output| < k$ **do**
  $\varepsilon = (J, j, \zeta) \leftarrow extractMin(Candidates)$
  Let $C_J = (c^0, \cdots, c^\phi)$ be the sequence of connections of J
  if $\zeta = 1$ (J is simple) **then**
    add J to $Output$
    for each connection c^i in C_J $(c_j, \cdots, c^\phi)$ **do**
      $J_{new} \leftarrow$ EarliestArrivalDetour(J, i, M)
      $\zeta' \leftarrow 0$
      if J_{new} is simple **then**
        $\zeta' \leftarrow 1$
      end if
      add (J_{new}, i, ζ') to $Candidates$
    end for
  else
    $S_\pi \leftarrow$ the set of stations visited by one of the connections $(c^0, \cdots, c^{j-1})$
    $C_{dev} \leftarrow \{c$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c)\}$
    $\mathcal{T}' = (S \setminus S_\pi, T, C \setminus C_{dev}, F)$
    $Q \leftarrow CSA(\mathcal{T}', c_{arr\_stop}, d, c_{arr\_time}, \tau_{max})$
    if Q exists **then**
      $J_{new} \leftarrow (f^0, c^0, \cdots, c^j, Q)$
      add $(J_{new}, j, \zeta = 1)$ to $Candidates$
    end if
  end if
end while
return $Output$

4.5.2 Différentiation entre chemin simple et non simple

La propriété clé sur laquelle se base l'algorithme PY-PT est :

Propriété 4.5.1. *Le calcul d'un détour à i ne nécessite pas de relancer un CSA à partir de i, car le mapping du PCSA contient déjà toutes les informations nécessaires. Cependant le détour d'arrivée au plus tôt peut être non simple mais l'heure d'arrivée est une borne inférieure.*

Démonstration.

L'algorithme PCSA calcule le front de Pareto qui maximise l'heure de départ et minimise l'heure d'arrivée de tous les arrêts vers l'arrêt d'arrivée. Ce qui veut dire que l'algorithme PCSA calcule l'heure d'arrivée au plus tôt pour chaque heure de départ pour chaque arrêt. Ainsi, aucun itinéraire vers l'arrêt d'arrivée ne peut être raté.

□

De manière similaire à l'algorithme Y-PT, l'algorithme PY-PT commence par ajouter un chemin d'arrivée au plus tôt J_0 à l'ensemble des candidats. Un élément ε dans l'ensemble candidat a trois attributs : le chemin J, son index de déviation j et un indicateur booléen ζ qui indique si le chemin J est simple ou non. Ainsi, l'élément $\varepsilon_0 = (J_0, 0, 1)$ est ajouté à l'ensemble des candidats. Contrairement avec l'algorithme Y-PT où un appel à l'algorithme CSA est fait pour calculer J_0, l'algorithme PY-PT extrait J_0 du *mapping* M déjà calculé, plus précisément $J_0 = M_{s,t}^{\tau_0,\tau_{max}}$. De la même manière que pour l'algorithme Y-PT, l'ensemble des résultats est initialisé avec un ensemble vide. Après ces étapes d'initialisation, l'algorithme commence par extraire un chemin d'arrivée au plus tôt (J, j, ζ) parmi ceux dans l'ensemble des candidats. Deux cas sont à distinguer :

si $\zeta = 1$ (J est simple) : J est ajouté à l'ensemble des résultats, ensuite tous les détours d'arrivée au plus tôt de J sont ajoutés à l'ensemble de candidats. Cette opération est faite comme suit, soit $C_J = (c^0, c^1, \cdots, c^\phi)$ la séquence de connexions de J, pour chaque connexion c^i (pour $j \leq i < \phi$) dans C_J, un détour d'arrivée au plus tôt J_{new} de J à i est extrait. Cette opération utilise M comme décrit plus tard.

Le chemin J_{new} peut ne pas être simple (aussi décrit plus tard). Cependant, J_{new} est ajouté à l'ensemble des candidats avec i comme index de déviation et $\zeta = 1$ si J_{new} est simple (et $\zeta = 0$ sinon).

si $\zeta = 0$ (J n'est pas simple) : J est « réparé », c'est-à-dire, il est remplacé (si possible) par son chemin simple d'arrivée au plus tôt correspondant. Pour ce faire, un algorithme applique la même routine que l'algorithme Y-PT. Plus précisément, soit c_j la connexion à l'index de déviation, l'algorithme retire les arrêts dans le préfixe, c'est-à-dire, chaque arrêt visité par une des connexions dans $c^0, \cdots, c^{j-1}$, de $\mathcal{T}$. De plus, pour chaque chemin J' dans l'ensemble des résultats qui commence par les connexions $c^0, c^1, \cdots, c^{j-1}, c^j$, la connexion c^j est retirée de $\mathcal{T}$. Ensuite, en utilisant l'algorithme CSA, l'algorithme PY-PT calcule un chemin d'arrivée au plus tôt Q à partir de l'arrêt d'arrivée de c^j ($c^{j-1}_{arr\_stop}$) jusqu'au puits t avec $c^{j-1}_{arr\_time}$ comme heure de départ. Soit J_{new} la concaténation du préfixe de J et de Q. Le chemin J_{new} est ajouté à l'ensemble des candidats avec j comme index de déviation et avec $\zeta = 1$ (car J_{new} est simple).

L'algorithme PY-PT répète ce processus jusqu'à ce que k chemins soient ajoutés dans l'ensemble des résultats.

Exemple Nous allons utiliser la figure 4.4 pour calculer 3 plus courts chemins. Chaque arc sur la figure a une heure de départ et une heure d'arrivée. L'algorithme PT-Y commence par calculer un

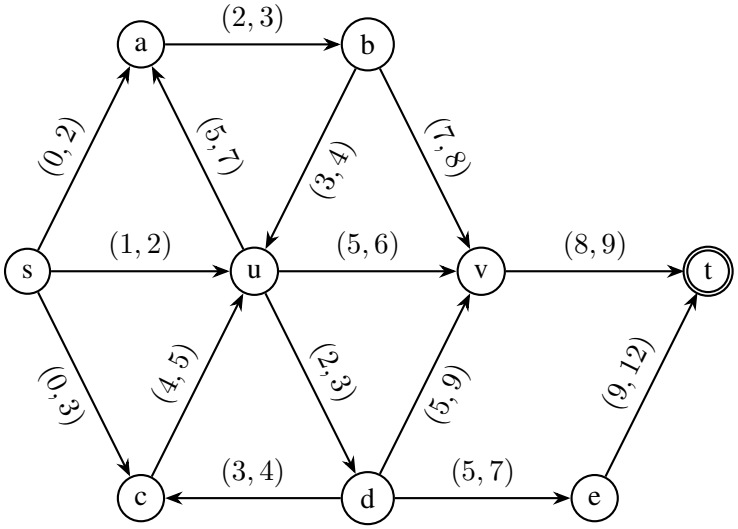


FIGURE 4.4 – Exemple pour l'algorithme PY-PT.

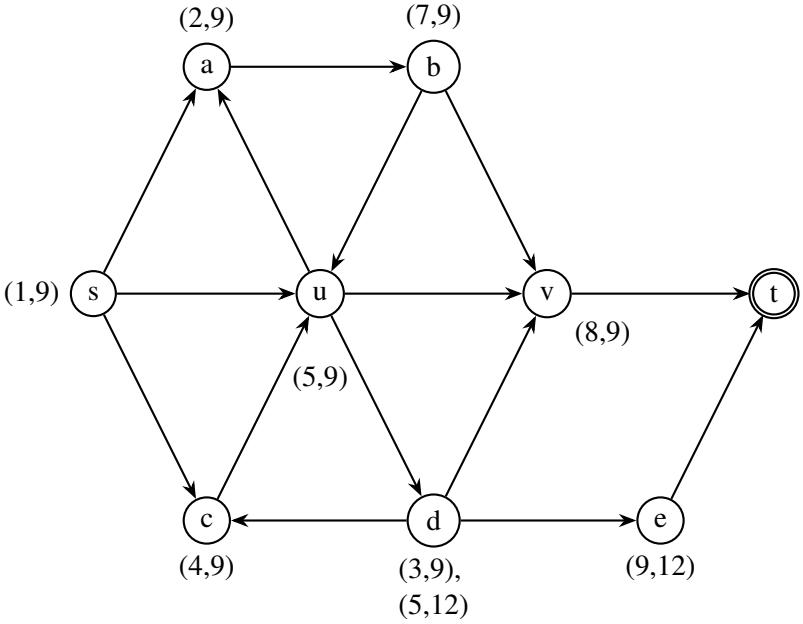


FIGURE 4.5 – *Mapping* de l'exemple pour l'algorithme PY-PT.

PCSA avec t comme arrivée, qui maximise l'heure de départ et minimise l'heure d'arrivée pour chaque nœud. Le *mapping* pour la figure 4.4 est la figure 4.5.

L'algorithme Y-PT commence par calculer un premier chemin d'arrivée au plus tôt en utilisant le *mapping* à partir de s, ce qui nous renvoie le chemin (s, u, v, t) avec une heure d'arrivée de 9 et l'ajoute à l'ensemble des solutions. Ensuite l'algorithme calcule les détours simples d'arrivée au plus tôt de ce chemin :

1. Pour le nœud s, l'algorithme regarde tous les voisins ainsi que leurs heures d'arrivée dans le *mapping*, ce qui nous donne un chemin partant de a arrivant à un temps de 9 et un chemin partant de c arrivant à un temps de 9. Le chemin choisi est celui passant par a de manière arbitraire, car ils avaient la même heure d'arrivée, l'algorithme renvoie le chemin (s, a, b, v, t).
2. Pour le nœud u, l'algorithme regarde tous les voisins ainsi que leurs heures d'arrivée dans le *mapping*, ce qui nous donne un chemin partant de d arrivant à un temps de 9 et le chemin partant de a n'est pas valide, car l'heure d'arrivée à a est supérieure à l'heure de départ du chemin qui amène à t. Le chemin choisi est celui passant par d, l'algorithme renvoie le chemin (s, u, d, c, u, v, t), qui est un chemin non simple.
3. Pour le nœud v, l'algorithme regarde tous les voisins ainsi que leurs heures d'arrivée dans le *mapping*, aucun nœud n'est choisi, car l'arc (v, t) est déjà utilisé par un chemin dans les résultats.

Les chemins (s, a, b, v, t) et (s, u, d, c, u, v, t) sont ajoutés à l'ensemble des candidats C. L'algorithme de Yen récupère de C le chemin avec la plus petite heure d'arrivée, le chemin (s, u, d, c, u, v, t) est choisi de manière arbitraire. Cependant, il est non simple donc l'algorithme le « répare ». Les étapes sont les mêmes que pour l'algorithme Y-PT ou de Yen : l'algorithme enlève le préfixe s ainsi que l'arc (u, v) pour avoir un chemin simple et qui n'a pas été déjà calculé. Ensuite un CSA est utilisé pour avoir un chemin d'arrivée au plus tôt dans la table horaire modifiée, qui nous renvoie le chemin (u, d, e, t). Le chemin « réparé » est donc (s, u, d, e, t) avec une heure d'arrivée de 12 et il est inséré dans l'ensemble des candidats. Une fois de plus, l'algorithme de Yen récupère de C le chemin avec la plus petite heure d'arrivée, le chemin (s, a, b, v, t) est choisi avec une heure d'arrivée de 9. Ensuite l'algorithme calcule les détours simples d'arrivée au plus tôt de ce chemin :

1. Pour le nœud s, l'algorithme regarde tous les voisins ainsi que leurs heures d'arrivée dans le *mapping*, ce qui nous donne un chemin partant de c arrivant à un temps de 9. L'algorithme renvoie le chemin (s, c, u, v, t).
2. Pour le nœud a, l'algorithme regarde tous les voisins ainsi que leurs heures d'arrivée dans le *mapping*, aucun nœud n'est choisi, car l'arc (a, b) est déjà utilisé par un chemin dans les résultats.
3. Pour le nœud b, l'algorithme regarde tous les voisins ainsi que leurs heures d'arrivée dans le *mapping*, ce qui nous donne un chemin partant de u arrivant à un temps de 9. L'algorithme renvoie le chemin (s, a, b, u, v, t).
4. Pour le nœud v, l'algorithme regarde tous les voisins ainsi que leurs heures d'arrivée dans le *mapping*, aucun nœud n'est choisi, car l'arc (v, t) est déjà utilisé par un chemin dans les résultats.

Les chemins (s, c, u, v, t) et (s, a, b, u, v, t) sont ajoutés à l'ensemble des candidats C. L'algorithme de Yen récupère de C le chemin avec la plus petite heure d'arrivée, le chemin (s, c, u, v, t) est choisi de manière arbitraire, qui est le 3e plus court chemin entre s et t et l'ajoute à l'ensemble des solutions.

Nous pouvons voir que le principal défaut de l'algorithme Y-PT ne l'est pas pour l'algorithme PY-PT. Pour calculer 3 chemins d'arrivée au plus tôt, un seul appel à l'algorithme CSA est fait,

pendant la « réparation » du chemin non simple. Chaque calcul de déviation n'utilise que le *mapping* précalculé par le PCSA. La question est de savoir si le surcoût du temps de calcul du PCSA est amorti par un faible nombre de « réparation » de chemins non simples.

4.5.3 Calcul des déviations

Algorithme 4.3 Fonction EarliestArrivalDetour(J, i, M)

$c^i \leftarrow$ the i^{th} connection of J
$c_{arr\_stop} \leftarrow$ the arrival stop of c^{i-1}
$c_{arr\_time} \leftarrow$ the arrival time of c^{i-1}
$C_{dev} \leftarrow \{c'$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c')\}$
$C^N = \{c' \in C$ s.t. $c'_{dep\_stop} = c_{arr\_stop}$, $c'_{dep\_time} \geq c_{arr\_time}$ and $c' \notin C_{dev}\}$
$c^{LB} \leftarrow$ a connection in C^N leading to a minimum arrival time from c_{arr_stop} to d after c_{arr_time} following M
$F_{dev} \leftarrow \{f$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, f)\}$
$F^N = \{f \in F$ s.t. $f_{dep\_stop} = c_{arr\_stop}$ and $f \notin F_{dev}\}$
$f^{LB} \leftarrow$ a footpath in F^N leading to a minimum arrival time from c_{arr_stop} to d following M
$J_{c^{LB}} \leftarrow c^{LB}.M^{c^{LB}_{arr\_time}, \tau_{max}}_{c^{LB}_{arr\_stop}, d}$
$J_{f^{LB}} \leftarrow f^{LB}.M^{c_{arr\_time}+f^{LB}_{dur}, \tau_{max}}_{f^{LB}_{arr\_stop}, d}$
$J_{min} \leftarrow$ the earliest arrival journey among $J_{c^{LB}}$ and $J_{f^{LB}}$
$\pi = (f^0, c^0, \cdots, c^{i-1})$
$J_{new} \leftarrow \pi.J_{min}$
return J_{new}

Maintenant, expliquons comment le chemin J_{new} est calculé (dans le cas où $\zeta = 1$). Le pseudo-code de ce processus est décrit dans algorithme 4.3. Soit c^{i-1} la $i-1^{ème}$ connexion de C_J (pour $j \leq i < \phi$), le processus suivant est appliqué :

- Dans un premier temps, l'algorithme scanne les connexions qui ont comme arrêt de départ $c^{i-1}_{arr\_stop}$, une heure de départ après $c^{i-1}_{arr\_time}$ et qui amène sur de nouveaux chemins, c'est-à-dire, différent de ceux dans l'ensemble des résultats. Plus précisément, soit $C_{dev} = \{c_{old} \in C$ tel qu'il y a un chemin dans l'ensemble des résultats qui commence avec les connexions $c^0, \cdots, c^{i-1}, c_{old}\}$, soit $C^N = \{c \in C$ tel que $c_{dep\_stop} = c^{i-1}_{arr\_stop}$, $c_{dep\_time} \geq c^{i-1}_{arr\_time}$ et $c \notin C_{dev}\}$ est l'ensemble des nouvelles connexions de déviations. L'algorithme scanne les connexions de C^N. Soit c^{LB} une connexion de C^N qui amène à un chemin d'arrivée au plus tôt de $c^{i-1}_{arr\_stop}$ à t en utilisant M. Formellement, pour chaque $c \in C^N$, soit J_c le chemin qui passe par c en suivant M, c'est-à-dire, soit $J_c = c.M^{c_{arr\_time}, \tau_{max}}_{c_{arr\_stop}, t}$, alors c^{LB} est une connexion dans C^N telle que $arr_\tau(J_{c^{LB}}) \leq arr_\tau(J_c)$ pour chaque $c \in C^N$.

  Remarque 4.5.1 – Si l'élément juste avant c^i est un chemin piéton, il est possible d'avoir un chemin avec deux chemins piétons consécutifs. Pour éviter ce scénario, les chemins du *mapping* pour $c^{i-1}_{arr\_stop}$ qui commencent par des chemins piétons ne sont pas scannés.

- Dans un deuxième temps, l'algorithme scanne les chemins piétons qui ont comme arrêt de départ $c^{i-1}_{arr\_stop}$ et qui amène à de nouveaux chemins, c'est-à-dire, différent de ceux

dans l'ensemble des résultats. De la même manière que précédemment, soit $F_{dev} = \{f_{old}$ tel qu'il y a un chemin dans l'ensemble des résultats qui commence avec les connexions $c^0, \cdots, c^{i-1}$ suivi de $f_{old}\}$, soit $F^N = \{f \in F$ tel que $f_{dep\_stop} = c^{i-1}_{arr\_stop}$ et $f \notin F_{dev}\}$ est l'ensemble des nouveaux chemins piétons de déviation et soit f^{LB} un chemin piéton de F^N qui amène à un chemin d'arrivée au plus tôt de $c^{i-1}_{arr\_stop}$ à t en utilisant M. Plus précisément, pour chaque $f \in F^N$, soit J_f le chemin qui passe par f en suivant M, c'est-à-dire, $J_f = f.M^{c^{i-1}_{arr\_time}+f_{dur},\tau_{max}}_{farr\_stop,d}$, alors f^{LB} est un chemin piéton dans F^N tel que $arr_\tau(J_{f^{LB}}) \leq arr_\tau(J_f)$ pour chaque $f \in F^N$.

Soit Q_{min} le chemin avec l'heure d'arrivée minimum parmi $J_{c^{LB}}$ et $J_{f^{LB}}$ et soit J_{min} le chemin créé par la concaténation du préfixe du chemin de J et de Q_{min}, c'est-à-dire, $J_{min} = (f^0, c^0, \cdots, c^{i-1}, Q_{min})$. À noter que J_{min} peut ne pas être simple, car le chemin extrait de M peut repasser par un des arrêts du préfixe de J. Par exemple, un arrêt visité par c^0 ou $c^1, \cdots$, ou c^{i-1} peut être à nouveau visité par $J_{c^{LB}}$ (ou par $J_{f^{LB}}$).

Remarque 4.5.2 – Quand l'algorithme scanne des connexions avec comme arrêt de départ $c^{i-1}_{arr\_stop}$ et une heure de départ supérieure à $c^{i-1}_{arr\_time}$, le chemin $M^{c_{arr\_time},\tau_{max}}_{c_{arr\_stop},d}$ peut soit commencer par un chemin piéton ou une boucle piétonne. En revanche, quand l'algorithme scanne des chemins piétons avec comme arrêt de départ $c^{i-1}_{arr\_stop}$, le chemin $M^{c^{i-1}_{arr\_time}+f_{dur},\tau_{max}}_{farr\_stop,d}$ ne peut commencer que par une boucle piétonne, sinon le chemin serait composé de deux chemins piétons consécutifs. Pour éviter cela, l'algorithme PCSA stocke les itinéraires dans deux structures de données séparées, une pour les chemins qui commencent avec une boucle piétonne et une pour les chemins qui commencent par un chemin piéton.

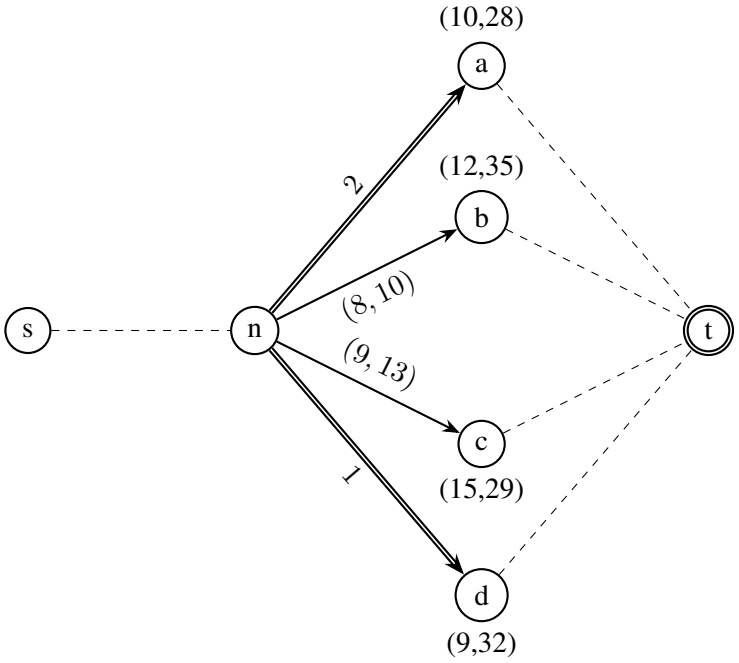


FIGURE 4.6 – Exemple d'un calcul de déviation pour l'algorithme PY-PT.

Exemple Nous allons utiliser la figure 4.6 pour illustrer le calcul des déviations dans l'algorithme PY-PT. Nous avons un calcul des déviations à partir de n avec un temps d'arrivée de 7. Les arcs doubles représentent des chemins piétons, ici (n, a) et (n, d). L'arc (n, a) est déjà utilisé par un autre chemin dans l'ensemble des résultats.

Le calcul des déviations se fait en deux phases. La première identifie toutes les connexions qui ne sont pas utilisées par d'autres chemins dans l'ensemble des résultats, ici (n, b) et (n, c). Ensuite grâce au *mapping*, la connexion avec le temps d'arrivée le plus tôt est identifiée :

- l'arc (n, b) permet d'atteindre le nœud b avec un temps d'arrivée inférieur au temps de départ du *mapping*, ce qui permet d'atteindre l'arrivée avec un temps d'arrivée de 35 ;
- l'arc (n, c) permet d'atteindre le nœud c avec un temps d'arrivée inférieur au temps de départ du *mapping*, ce qui permet d'atteindre l'arrivée avec un temps d'arrivée de 29.

La connexion avec le temps d'arrivée le plus tôt est (n, c). La deuxième phase fait la même chose pour les chemins piétons. Un des chemins piétons est déjà utilisé par un chemin de l'ensemble des résultats donc un seul chemin piéton est à considérer. L'arc (n, b) est un chemin piéton que l'on traverse en 2 unités de temps, avec un temps d'arrivée de 7 au nœud n, le temps d'arrivée au nœud n est de 9 et permet d'utiliser le résultat du *mapping* pour atteindre l'arrivée avec un temps d'arrivée de 32.

La dernière phase compare la meilleure connexion avec le meilleur chemin piéton pour prendre celui qui a le temps d'arrivée le plus tôt, ici la connexion avec un temps d'arrivée de 29. Puis on concatène le préfixe $(s, \cdots, n)$, la connexion (n, c) et le chemin contenu dans le mapping qui amène à t.

Pour conclure, à la différence de l'algorithme Y-PT où un détour simple d'arrivée au plus tôt est calculé à partir des chemins extraits en utilisant l'algorithme CSA, l'algorithme PY-PT envisage un détour d'arrivée au plus tôt (pas nécessairement simple) donné par l'algorithme PCSA, et deux cas sont à distinguer :

- Soit le détour d'arrivée au plus tôt est simple, un appel à l'algorithme CSA est évité et un détour simple d'arrivée au plus tôt est ajouté à l'ensemble de candidats.
- Soit le détour d'arrivée au plus tôt n'est pas simple et l'algorithme PY-PT ajoute ce détour d'arrivée au plus tôt dans l'ensemble des candidats, mais avec un indicateur booléen explicitant qu'il n'est pas simple.

Les chemins dans l'ensemble des candidats sont stockés par heure d'arrivée croissante. Ainsi, seulement quand ce détour non simple est extrait de l'ensemble des candidats, sa version simple est calculée avec un appel à l'algorithme CSA. En d'autres mots, le véritable calcul de ce détour simple est reporté. Ce report permet d'éviter certains appels à l'algorithme CSA, par exemple quand k chemins d'arrivée au plus tôt sont ajoutés dans l'ensemble des résultats, il peut rester un certain nombre de chemins non simple dans l'ensemble de candidats et leur processus de « réparation » est évité.

À noter que malgré les reports, l'ordre d'extraction des chemins simples de l'ensemble des candidats reste valide. C'est dû au fait qu'un chemin J dans l'ensemble de candidats est soit inséré avec sa vraie heure d'arrivée (dans le cas où J est simple), soit avec une borne inférieure sur son heure d'arrivée (dans le cas où J est non simple).

4.6 Expérimentations

Dans cette partie, nous décrivons nos expériences. Tout d'abord, nous commençons par décrire notre implémentation et les paramètres expérimentaux, ensuite nous discutons de nos résultats sur les réseaux de transports en commun.

Réseau	Arrêts	Connexions	Lignes	Voyages	Chemins piéton
Paris	44534	3209401	1864	150963	502291
Berlin	28651	1379755	1296	63569	62456
Stockholm	14258	703326	664	34799	22138
Allemagne	74398	3601420	3599	168024	599284
Suisse	29844	2599675	5645	248826	27202

TABLE 4.1 – Taille des différents réseaux.

4.6.1 Paramètres expérimentaux

Nous allons décrire les détails de notre implémentation et des paramètres expérimentaux utilisés dans nos expériences.

Nous avons implémenté l'algorithme Y-PT et PY-PT en Java et notre code est disponible publiquement [2].

À noter que dans nos implémentations le paramètre k ne fait pas partie des entrées. Cela permet l'utilisation de nos méthodes comme des itérateurs, avec la possibilité de renvoyer un chemin d'arrivée au plus tôt suivant tant qu'un existe, malgré le fait que certaines optimisations supplémentaires peuvent être ajoutées si k fait partie des entrées.

4.6.1.1 Instances

Réseau	Algorithme	Temps de réponse (ms)	# d'appel au CSA
Paris	Y-PT	52306	1186
	PY-PT	4728	34
Berlin	Y-PT	18206	1500
	PY-PT	726	16
Stockholm	Y-PT	5984	1818
	PY-PT	166	7
Allemagne	Y-PT	30810	684
	PY-PT	1479	11
Suisse	Y-PT	18779	1181
	PY-PT	1264	42

TABLE 4.2 – Détails du Y-PT et du PY-PT sur des réseaux de transports métropolitain et de trains nationaux.

Nous avons évalué les performances de nos algorithmes sur deux réseaux de trains nationaux (Allemagne et Suisse) et trois réseaux de transports en commun métropolitains (Paris, Berlin et Stockholm). Les caractéristiques de ces réseaux sont présentées dans le tableau 4.1. Ces jeux de données sont accessibles publiquement à travers un flux GTFS (https://transitfeeds.com/), nous avons téléchargé ce jeu de données en octobre 2019. Les données de 2019 sont tout à fait comparables avec celles de 2022, les réseaux de transports en commun changent très peu.

Les réseaux de transports en commun métropolitains sont plus denses que les réseaux de trains nationaux, c'est-à-dire, que le rapport de connexions par rapport aux nœuds est plus faible pour les réseaux de trains que sur les réseaux métropolitains. L'explication de ce phénomène est simple : les réseaux de trains peuvent seulement utiliser des trains, tandis que les réseaux métropolitains peuvent utiliser les bus, trains, ferry et n'importe quel autre mode de transport. Ainsi, nous allons montrer les performances de nos algorithmes sur ces deux types de réseaux.

Nous avons choisi au hasard 1000 requêtes (une paire origine-puits d'arrêts) pour chaque réseau de transports en commun, et nous lançons chaque algorithme pour chaque paire d'arrêts avec k allant d'une valeur de 2 à 100.

Nous considérons le temps d'exécution et le nombre d'appels à l'algorithme CSA. À noter que le nombre d'appels à l'algorithme CSA est une indication du temps de réponse qui est indépendant de l'implémentation et de l'architecture de la machine.

Tous les calculs sont faits sur une machine équipée d'un Intel(R) Core(TM) i7-1185G7 à 3.00GHz et 32 GB de RAM.

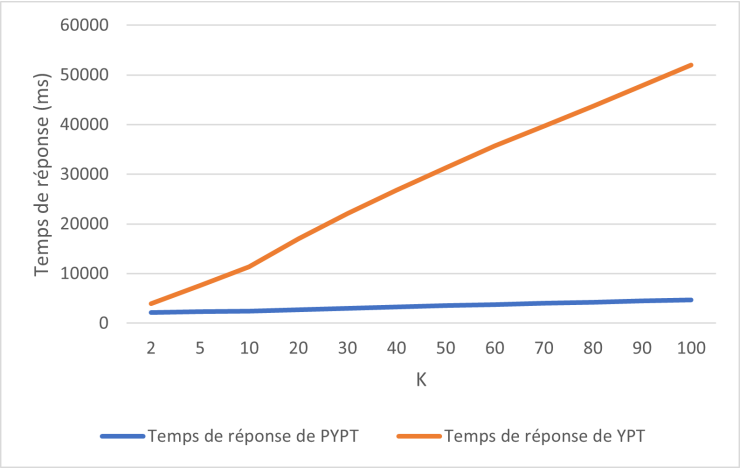
4.6.2 Résultats expérimentaux

Dans cette partie, nous décrivons nos résultats expérimentaux sur les réseaux de transports en commun.

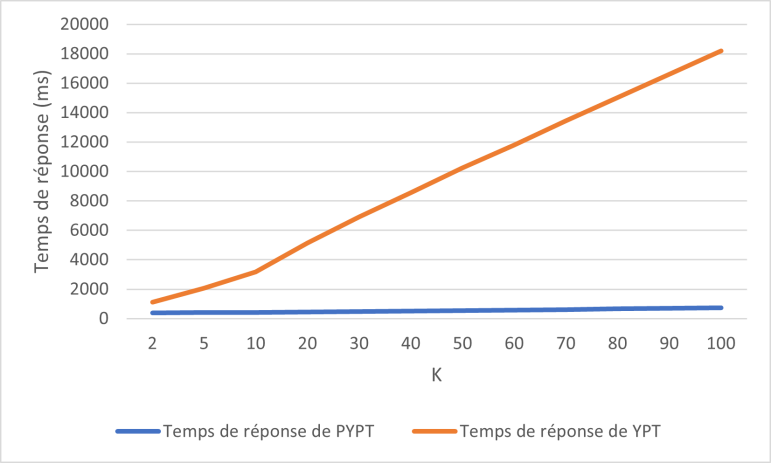
Nous avons mesuré le temps moyen de réponse des algorithmes pour les réseaux considérés. Les données (le temps de réponse et le nombre d'appels à l'algorithme CSA) dans le tableau 4.2 correspond à la plus grande valeur testée de k ($k = 100$). Tandis que les données de la figure 4.7, la figure B.5, la figure 4.9 et la figure B.7 correspond à l'évolution du temps de réponse et du nombre d'appels à l'algorithme CSA en fonction des valeurs de k.

Le temps moyen de réponse rapporté dans le tableau 4.2 montre que l'algorithme PY-PT est significativement plus rapide que l'algorithme Y-PT pour chaque réseau considéré (le temps de réponse est plus rapide d'au moins un facteur 10 pour $k = 100$). À noter que pour une requête donnée, de très nombreux chemins ont la même heure d'arrivée. De plus, une comparaison plus poussée sur les différents réseaux (figure 4.8 et figure B.6) montre que l'algorithme PY-PT est plus rapide que l'algorithme Y-PT sur quasiment toutes les requêtes. Également, la figure 4.7 et la figure B.5 montre que ce gain de temps de réponse reste considérable même pour des petites valeurs de k (même pour $k = 2$). Ce qui veut dire que le temps pris pour le calcul du PCSA est compensé par l'extraction des détours simples, même pour $k = 2$.

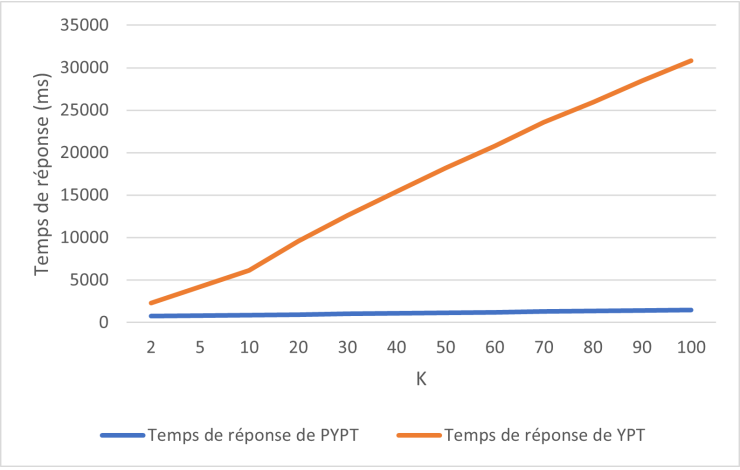
Nous pouvons donc conclure qu'en pratique, l'algorithme PY-PT est plus rapide que l'algorithme Y-PT pour tous nos scénarios. De plus, la figure 4.9 et la figure B.7 montre que le nombre d'appels à l'algorithme CSA est significativement réduit en utilisant l'algorithme PY-PT et cela garantit les gains pour n'importe quels paramètres expérimentaux [Johnson, 2002].



(a) Temps de réponse du YPT en fonction du k pour la ville de Paris

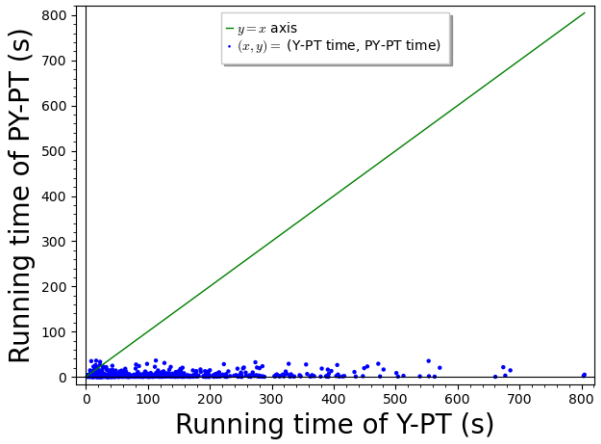


(b) Temps de réponse du YPT en fonction du k pour la ville de Berlin

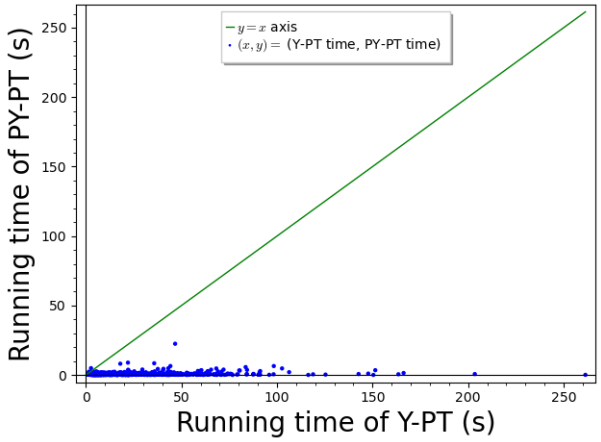


(c) Temps de réponse du YPT en fonction du k pour le réseau de trains de l'Allemagne

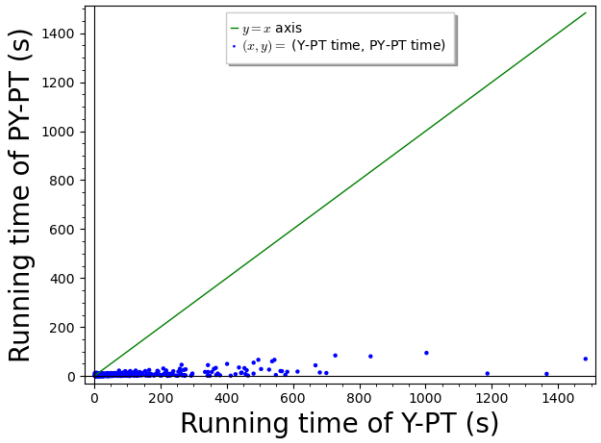
FIGURE 4.7 – Temps de réponse du YPT en fonction du k pour les réseaux de Paris, Berlin et l'Allemagne.



(a) Temps de réponse du YPT en fonction du temps de réponse du PYPT pour le réseau de trains de l'Allemagne

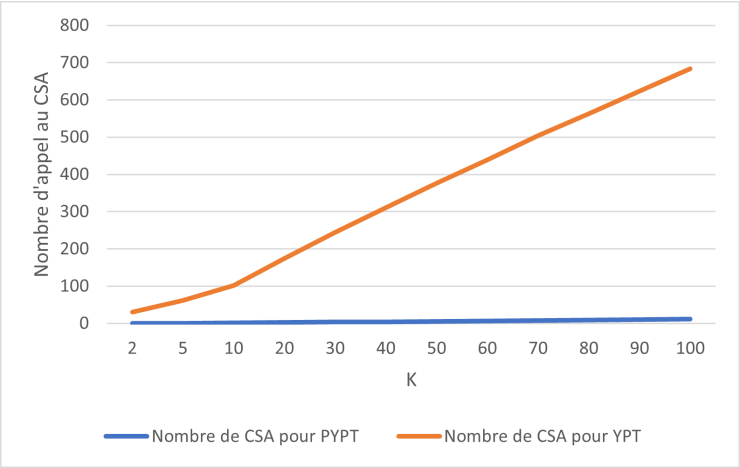


(b) Temps de réponse du YPT en fonction du temps de réponse du PYPT pour la ville de Berlin

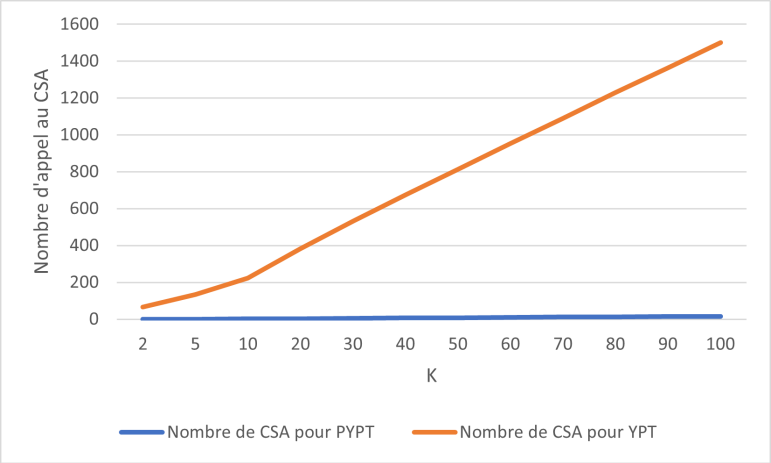


(c) Temps de réponse du YPT en fonction du temps de réponse du PYPT pour la ville de Paris

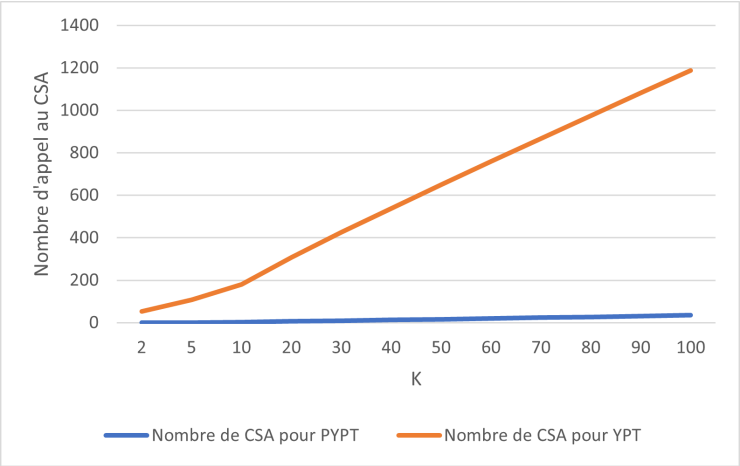
FIGURE 4.8 – Temps de réponse du YPT en fonction du temps de réponse du PYPT pour les réseaux de Paris, Berlin et l'Allemagne.



(a) Nombre d'appels au CSA en fonction du k pour le réseau de trains de l'Allemagne



(b) Nombre d'appels au CSA en fonction du k pour la ville de Berlin



(c) Nombre d'appels au CSA en fonction du k pour la ville de Paris

FIGURE 4.9 – Nombre d'appels au CSA en fonction du k pour les réseaux de Paris, Berlin et l'Allemagne.

4.7 (Dis)similarités des chemins renvoyés

Deux chemins sont similaires s'ils partagent une majeure partie de leurs connexions et chemins piétons. Dans les graphes, une mesure bien connue est celle de Jacquard. Elle mesure la similarité de deux chemins en utilisant le rapport entre la longueur des arcs qu'ils ont en commun et la longueur de leur union (équation 4.1) [Chondrogiannis et al., 2017, Al Zoobi et al., 2021b].

$$S(P,Q) = \frac{\ell(P \cap Q)}{\ell(P \cup Q)} \tag{4.1}$$

Nous proposons de l'adapter au problème de calcul de chemin d'arrivée au plus tôt dans un réseau de transports en commun. Ainsi, nous définissons la similarité de deux chemins comme le rapport entre le temps de voyage de leurs connexions et les chemins piétons en commun et le temps de voyage de l'union de leurs connexions et leurs chemins piétons. Plus précisément, nous définissons la similarité de deux chemins comme suit :

$$S_{PT}(J_1, J_2) = \frac{\sum_{c \in J_1 \cap J_2} dur(c) + \sum_{f \in J_1 \cap J_2} f_{dur}}{\sum_{c \in J_1 \cup J_2} dur(c) + \sum_{f \in J_1 \cup J_2} f_{dur}} \tag{4.2}$$

Avec :

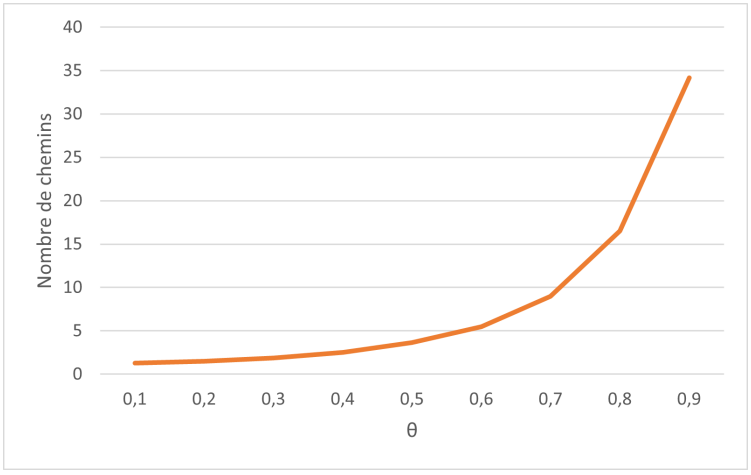
- $dur(c)$ est le temps de voyage d'une connexion c, et est défini comme le temps passé dans le véhicule de c, c'est-à-dire, $dur(c) = c_{arr\_time} - c_{dep\_time}$.
- f_{dur} est la durée du chemin piéton f.
- J_1, J_2 sont deux chemins de transports en commun composés de connexions et de chemins piétons.

Soit une valeur seuil $\theta \in [0,1]$, deux chemins J_1 et J_2 sont appelés θ dissimilaire si leur similarité ne dépasse pas θ, c'est-à-dire, si $S_{PT}(J_1, J_2) \leq \theta$. Ainsi, J_1 et J_2 sont dits θ dissimilaire si J_1 ne possède pas plus de θ connexions et chemins piétons en commun avec J_2. Pour une utilisation en pratique, une valeur de θ fixée à 0,5 est judicieuse, nous avons ainsi des chemins qui partagent la moitié de leurs connexions et de leurs chemins piétons.

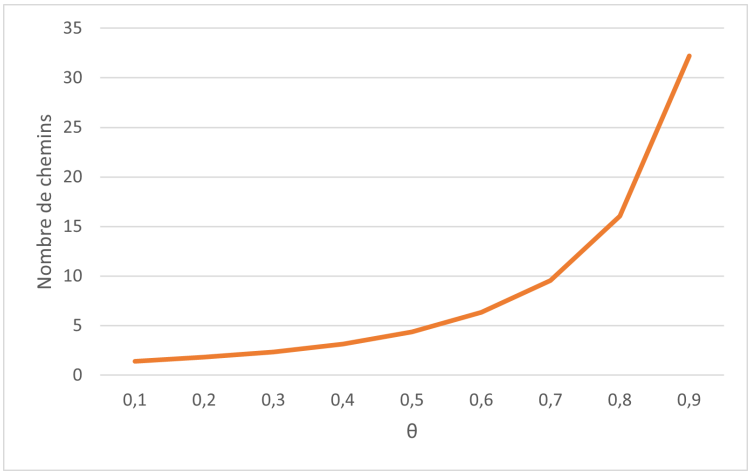
Pour évaluer le nombre de chemins dissimilaires renvoyés par nos algorithmes, nous mesurons le nombre de chemins qui sont θ dissimilaires parmi les 100 premiers chemins renvoyés par nos algorithmes. Comme montré dans la figure 4.10 et la figure B.8, nos algorithmes peuvent être utilisés pour extraire, en moyenne, de 3 à 5 chemins qui sont 0,5 dissimilaires, avec plusieurs chemins qui ont la même heure d'arrivée. De plus, la figure 4.11 et la figure B.9 montre le nombre de chemins dissimilaires parmi les 100 chemins renvoyés par nos algorithmes par rapport à la mesure de similarité θ. Nous pouvons voir que plus la valeur de θ augmente et plus le nombre de chemins dissimilaires augmentent. Cependant la qualité des résultats baisse également, car les résultats auront une très grande majorité de connexions et de chemins piétons en commun.

4.8 Synthèse

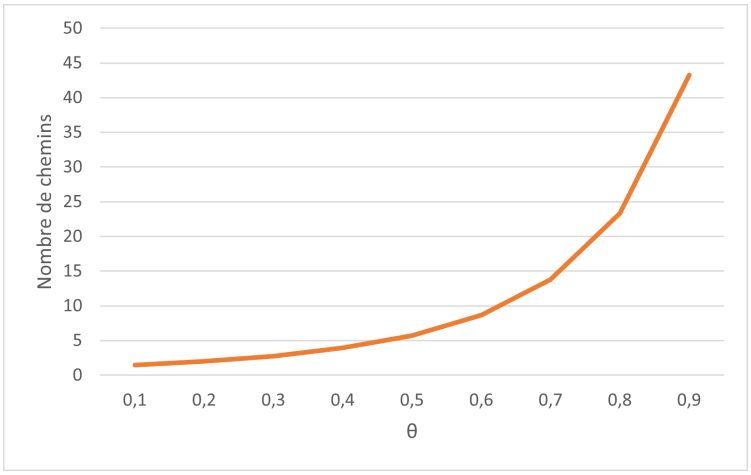
Nous avons développé une alternative au calcul d'itinéraire pour les transports en commun, qui offre un vaste ensemble de résultats intéressants. Grâce à une adaptation du problème de k plus courts chemins pour les transports en commun, en utilisant l'algorithme de Yen et l'algorithme PNC. L'algorithme Y-PT est une variante de l'algorithme de Yen pour les réseaux de transports en commun et utilise comme algorithme de calcul de chemin d'arrivée au plus tôt le CSA. L'algorithme CSA est le plus rapide pour le calcul monobjectif parmi les algorithmes qui n'utilisent pas



(a) Nombre de chemins dissimilaires en fonction du θ pour le réseau de trains de l'Allemagne

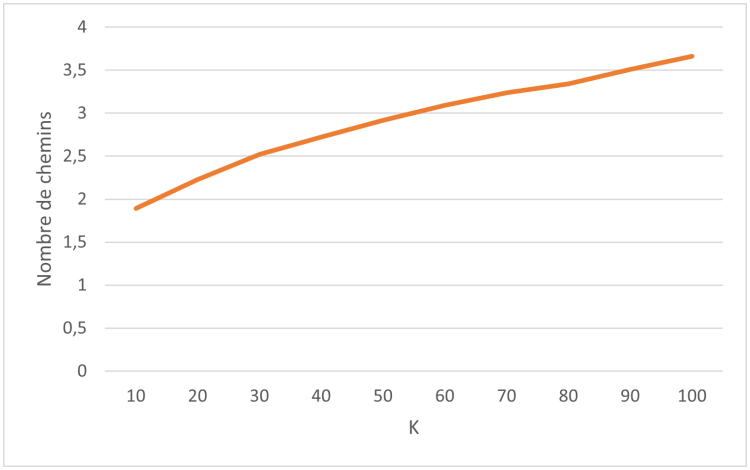


(b) Nombre de chemins dissimilaires en fonction du θ pour la ville de Berlin

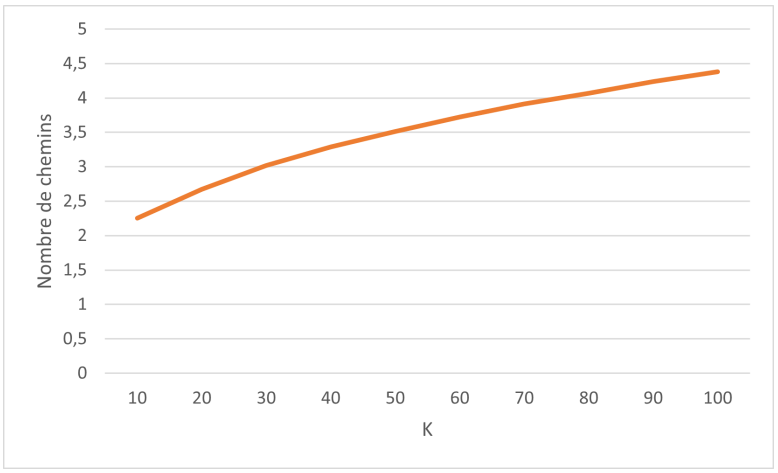


(c) Nombre de chemins dissimilaires en fonction du θ pour la ville de Paris

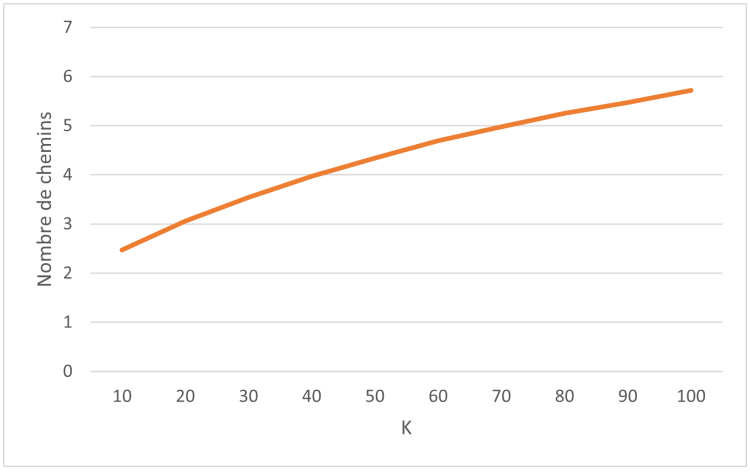
FIGURE 4.10 – Nombre de chemins dissimilaires en fonction du θ pour les réseaux de Paris, Berlin et l'Allemagne.



(a) Nombre de chemins 0,5 dissimilaires en fonction du k pour le réseau de trains de l'Allemagne



(b) Nombre de chemins 0,5 dissimilaires en fonction du k pour la ville de Berlin



(c) Nombre de chemins 0,5 dissimilaires en fonction du k pour la ville de Paris

FIGURE 4.11 – Nombre de chemins 0,5 dissimilaires en fonction du k pour les réseaux de Paris, Berlin et l'Allemagne.

de précalcul. Des précalculs qui sont contre-productifs dans le cas de l'algorithme de Yen, car de multiples arcs doivent être supprimés tout au long du déroulement de l'algorithme. L'algorithme de Yen calcule un chemin d'arrivée au plus tôt et calcule des détours d'arrivée au plus tôt simple avec interdiction pour chaque nœud de ce chemin, puis fait de même pour le détour d'arrivée au plus tôt avec la plus petite heure d'arrivée et ainsi de suite jusqu'à ce que k chemins soient trouvés.

Le gros défaut de l'algorithme Y-PT, qui est le même que pour l'algorithme de Yen, est le très grand nombre d'appels à l'algorithme de plus court chemin. L'algorithme PNC permet de résoudre ce problème et l'algorithme PY-PT est une variante de l'algorithme PNC pour les réseaux de transports en commun. Il utilise également l'algorithme CSA, mais aussi l'algorithme PCSA pour calculer un arbre d'arrivée au plus tôt afin d'essayer d'éviter des calculs de chemin d'arrivée au plus tôt. L'algorithme PY-PT fonctionne de la même manière que l'algorithme Y-PT cependant pour le calcul du détour d'arrivée au plus tôt avec interdiction, il utilise le *mapping* du PCSA pour calculer une borne inférieure de l'heure d'arrivée. Quand un chemin est extrait de l'ensemble des candidats :

- s'il est simple l'algorithme a évité un appel du CSA ;
- s'il est non simple alors l'algorithme le « répare » en utilisant la même procédure que l'algorithme Y-PT.

Nos expériences montrent que sur tous les réseaux considérés le temps de réponse de l'algorithme PY-PT est plus de 10 fois plus rapide que l'algorithme Y-PT, pour une valeur de $k = 100$. De plus, l'algorithme PY-PT est plus rapide même pour la plus petite valeur de $k = 2$, grâce au fait que le calcul du PCSA permet de retarder un très grand nombre d'appels au CSA lors du calcul de détours. De plus, nous avons également vu que les résultats fournis par les algorithmes Y-PT et PY-PT sont dissimilaires. Nous avons en moyenne 3 à 5 chemins qui ne partagent au maximum que 50% de leurs connexions et chemins piétons, ce qui permet de fournir des résultats intéressants à des utilisateurs.

Pour nos futurs travaux, une direction intéressante à poursuivre serait dans l'algorithme de chemin d'arrivée au plus tôt. Pour l'instant nous utilisons l'algorithme CSA, mais de très gros gains sont possibles en utilisant du précalcul, voir le tableau 2.8. Cependant le précalcul reste un obstacle majeur, car le préfixe ainsi que les arcs déjà utilisés par les chemins de l'ensemble des résultats doivent être supprimés du graphe.

Du côté de la dissimilarité, une piste très intéressante serait de comparer les résultats θ dissimilaires avec un front de Pareto. Le front de Pareto est une structure de données coûteuse à maintenir et peut-être que des résultats dissimilaires avec un certain θ peuvent être identiques au front de Pareto avec un certain nombre de critères.

Une autre question intéressante sur la dissimilarité serait de comparer nos résultats obtenus avec l'algorithme PY-PT à un algorithme qui renvoie des chemins dissimilaires dans un graphe, une adaptation de [Chondrogiannis et al., 2017] par exemple.

CHAPITRE 5

Connection Scan Algorithm et covoiturage

Le réseau routier et le réseau de transports en commun ont chacun des avantages. Le réseau de transports en commun possède des modes a haute fréquence de passage, qui peuvent avoir des voies dédiées leur permettant de desservir très vite des stations et assurant en général une grande robustesse pour les horaires. Le réseau routier donne une liberté totale à ces usagers, mais le très grand nombre de voitures et le nombre d'utilisateurs moyen dans une voiture qui est très proche de 1 provoquent de nombreux embouteillages. Le covoiturage se présente comme une solution utilisant les avantages de ces deux modes de transports tout en diminuant le nombre de voitures dans le réseau routier. Cependant, très peu d'algorithmes existent pour permettre la fusion du covoiturage et du transport en commun. Nous avons développé un algorithme appelé Multimodal Connection Scan Algorithm (MCSA) qui permet de fournir en moyenne une dizaine de résultats multimodaux à des utilisateurs avec des temps de réponse inférieurs à 2 secondes, permettant une utilisation interactive dans une application pour smart phone.

The road network and the public transit network both have advantages. The public transit network has high frequency modes, that use dedicated travel lanes, allowing them to serve stations very quickly and generally ensuring robust schedules. The road network gives total control to its users, but the very large number of cars and the average number of users in a car, which is very close to 1, causes many traffic jams. Carpooling is a solution that uses the advantages of these two modes of transportation while reducing the number of cars on the road network. However, very few algorithms exist to allow the fusion of carpooling and public transit. We have developed an algorithm called Multimodal Connection Scan Algorithm (MCSA) that provides an average of 10 multimodal results to users with response times of less than 2 seconds, allowing interactive use in a smart phone application.

5.1 Introduction

La mobilité est un axe important pour le développement des villes et a un impact sociétal et environnemental [Dustdar et al., 2017]. Deux réseaux existent uniquement pour la mobilité : le réseau de transports en commun et le réseau routier. Le réseau de transports en commun est composé de plusieurs modes de transports où un transporteur peut prendre de multiples usagers sur une ligne qui passe à des arrêts à des heures fixes. Quant au réseau routier, il est composé de deux modes : les voitures et les motos, les deux ayant le même rôle « emmener un utilisateur à sa destination », mais la voiture permet de prendre de multiples utilisateurs simultanément. En ce moment, le réseau le plus utilisé est le routier avec une très forte préférence pour la voiture privée [Sierpiński, 2013]. Cependant, le nombre d'utilisateurs moyen dans une voiture est très proche de 1 [Santos et al., 2011], ce qui veut dire que les embouteillages causés par les voitures peuvent être fortement réduits en augmentant le nombre d'usagers dans chaque voiture.

Chaque réseau a ses avantages et ses inconvénients. Pour le réseau de transports en commun, de multiples modes n'utilisent pas le réseau routier et sont ainsi moins touchés par des embouteillages (métro, train, tram ...) et les horaires de passage fixes sont très pratiques quand on planifie un itinéraire. Mais les arrêts sont placés par l'urbanisme pour bénéficier au plus grand nombre et non tout le monde. En outre ces horaires sont en général bien respectés, mais les horaires de passage impliquent qu'attendre est quasiment obligatoire. Pour le réseau routier, le conducteur a un plus grand contrôle sur son itinéraire avec une heure de départ qu'il choisit ainsi qu'une destination qui lui convient complètement. Cependant, le réseau routier est partagé par un très grand nombre d'utilisateurs et avec les villes qui deviennent de plus en plus grandes, les routes sont encore plus susceptibles de devenir embouteillées.

Une solution qui peut améliorer à la fois le réseau de transports en commun et le réseau routier est d'augmenter le nombre d'utilisateurs dans les voitures personnelles grâce au covoiturage. L'impact du covoiturage diminue le nombre de voitures sur le réseau routier et assure la fluidité du réseau de transports en commun, qui est composé de nombreux bus qui partagent le réseau routier avec les voitures. Cependant, les itinéraires uniquement en covoiturage ne sont pas la solution, car trouver un ou plusieurs utilisateurs qui ont simultanément les mêmes arrêts de départ et d'arrivée (ou aux alentours), ainsi que les mêmes heures de départ et d'arrivée est quasiment impossible. Nous ne pouvons pas non plus nous attendre à ce que le conducteur fasse de longs crochets pour prendre et déposer les utilisateurs. La solution est d'utiliser les deux réseaux à la fois dans un itinéraire, afin de pallier les soucis de chaque réseau tout en utilisant leurs points forts, en fusionnant le covoiturage avec le réseau routier et donner aux utilisateurs des itinéraires multimodaux.

Nous proposons une nouvelle méthode qui a pour but de calculer des itinéraires multimodaux impliquant à la fois du covoiturage et des transports en commun. Ceci est fait en fusionnant le graphe de covoiturage avec le graphe de transports en commun, en se basant sur l'observation qu'un covoiturage ressemble très fortement à une ligne de transports en commun passant par chaque arrêt une fois. Nous introduisons l'algorithme *Multimodal Connection Scan Algorithm* (MCSA) qui combine de manière transparente les itinéraires de covoiturage et les itinéraires de transports en commun.

La mise en œuvre d'un système de covoiturage utilisant l'algorithme CSA est en cours d'élaboration chez Instant System. Un travail de l'état de l'art a été fait et un travail d'expérimentation est en cours. Nous présentons ici un *proof of concept* sur des données générées aléatoirement.

5.2 État de l'art

Le problème de covoiturage a de multiples définitions, par exemple quand on connaît tous les conducteurs, tous les passagers au début de la journée et qu'on doit optimiser le remplissage des voitures. Le transport à la demande (TAD ou *Dial A Ride Problem* ou DARP) est l'équivalent dynamique du problème précédent. Le DARP est un problème très étudié, il existe de nombreux algorithmes pour le résoudre. Les algorithmes de type *branch and cut* sont très efficaces [Ropke et al., 2007, Cordeau, 2006], des variantes du problème DARP existent avec l'utilisation de VTC (voiture de transport avec chauffeur) si un utilisateur ne peut être récupéré par le fournisseur de service [Schenekemberg et al., 2022]. Il existe même des variantes du DARP qui utilise également des transports en commun [Posada et al., 2017]. De nombreuses études existent sur le covoiturage au sens large, [Ho et al., 2018] sur le problème de transport à la demande, [Furuhata et al., 2013] sur le problème du covoiturage et [Agatz et al., 2012] sur le problème de covoiturage dynamique.

On s'intéresse au covoiturage du point de vue de l'utilisateur, comme une requête, et nous ne considérons pas le problème de recouvrement des demandes. De notre point de vue pragmatique, une voiture est une offre de covoiturage pour le premier qui en fait la demande. Une fois que quelqu'un a pris le covoiturage il est éventuellement disponible pour un autre passager : soit en autorisant la prise d'un autre passager uniquement après avoir déposé le premier pour éviter d'influer sur l'heure d'arrivée du premier passager, soit en autorisant le prise d'un autre passager à n'importe quel moment, ce qui peut impacter l'heure d'arrivée du premier passager, si les points de prises et de déposes ne sont pas les mêmes.

Le problème que nous souhaitons résoudre est le covoiturage dynamique et multimodal, où un utilisateur utilise à la fois le réseau routier et le réseau de transports en commun pour atteindre sa destination. Il existe plusieurs algorithmes mélangeant transports en commun et réseau routier, cependant tous ne vont pas répondre au même problème. Soit la partie en covoiturage est fixé avant le calcul de l'itinéraire en transports commun, soit l'algorithme trouve l'itinéraire en covoiturage optimal pendant le calcul de l'itinéraire en transports en commun.

Quand l'itinéraire en covoiturage est fixé, la solution de [Bit-Monnot et al., 2013] découpe l'itinéraire multimodal en 5 parties : l'itinéraire du conducteur entre son point de départ et le point de ramassage, l'itinéraire du passager entre son point de départ et le point de ramassage, l'itinéraire que le conducteur et le passager partagent entre le point de ramassage et le point de dépose, l'itinéraire du conducteur entre le point de dépose et son arrivée et l'itinéraire du passager entre le point de dépose et son arrivée. Le but de l'algorithme est de minimiser le temps de trajet pour le conducteur et le passager. Ce but est atteint en lançant de multiples algorithmes de Dijkstra, avec des recherches en amont et en aval, pour optimiser chaque partie de l'itinéraire. Ce qui veut dire que pour répondre à une requête par un utilisateur, il faut lancer 4 fois l'algorithme de Dijkstra pour chaque annonce de covoiturage correspondant aux besoins de l'utilisateur, ce qui le rend peu pratique pour une utilisation en temps réel.

Quand l'itinéraire de covoiturage est choisi pendant le calcul de l'itinéraire multimodal, [Aissat and Varone, 2015] calculent un premier itinéraire en utilisant uniquement le réseau de transports en commun et ensuite sont calculés tous les potentiels itinéraires en voiture que l'on pourrait substituer à un sous-chemin de l'itinéraire en transports en commun. Le covoiturage prend en compte un temps maximum de crochet et devrait renvoyer un itinéraire qui arrive plus tôt que l'itinéraire en transports en commun. Un désavantage de cette méthode est que les itinéraires de covoiturage ont forcément un point de ramassage et de dépose près de l'itinéraire de transports

en commun calculé au début, alors que potentiellement un bon itinéraire de covoiturage pourrait déposer le passager plus loin que son arrêt d'arrivée puis le passager reviendrait vers sa destination en utilisant les transports en commun.

Le travail de [Masri et al., 2017] utilise une approche similaire, mais le calcul de l'itinéraire de transports en commun utilise un autre algorithme. La première partie est récursive et cherche un itinéraire en transports en commun entre le point de départ et le point d'arrivée : si on en trouve un, l'algorithme s'arrête, sinon, nous prenons le milieu entre le point de départ et le point d'arrivée et on calcule un itinéraire entre le point de départ et le milieu ainsi qu'entre le milieu et le point d'arrivée. Finalement, l'algorithme substitue un itinéraire de covoiturage avec une sous-partie de l'itinéraire en transports en commun. Cette approche a le même problème que [Aissat and Varone, 2015] même si plusieurs points sont pris comme départs et arrivées autour du point de départ et d'arrivée, respectivement.

Un projet de recherche, appelé SocialCar, [Jamal et al., 2017] mêle le réseau de transports en commun avec le réseau routier. L'algorithme utilise pour la recherche d'itinéraire un algorithme de Dijkstra modifié qui utilise une fonction multiobjectif pondérée, ce qui permet de fournir de bons résultats, mais ne permet pas de représenter le front de Pareto. Les expérimentations sont faites sur un unique réseau de petite taille, le réseau routier a 200 000 arcs et le réseau de transports en commun a 66 000 connexions. Cependant, même sur un réseau de petite taille, les temps de réponse sont pour les points les plus éloignés aux alentours de 1 seconde, ce qui en fait un algorithme difficilement adaptable sur de plus grands réseaux.

Finalement, [Huang et al., 2018] mélangent les 2 réseaux, celui de covoiturage ayant des points de prises et de déposes flous et celui de transports en commun, dans un seul réseau. Le but est de lier les arrêts du covoiturage et les arrêts de transports en commun qui sont proches les uns des autres. Cependant quand on fusionne les arrêts du réseau de transports en commun qui est un réseau rigide et du réseau de covoiturage qui est plus flou, le réseau fusionné doit garder ces deux propriétés. La fusion des réseaux est faite avec des *Drive Time Area* (DTA), qui est la zone qu'un véhicule peut atteindre en un certain temps à partir d'un certain arrêt. Quand un arrêt de transports en commun comprend dans son *Drive Time Area* un arrêt de covoiturage, cela veut dire que cet arrêt de transports en commun peut être atteint en un certain temps en conduisant et ces arrêts peuvent être utilisés pour passer d'un covoiturage à du transport en commun (ou vice-versa). Pour éviter de regarder trop d'arrêts de covoiturage, ils introduisent le concept de *Point of Action* (POA), qui limite les arrêts à considérer dans un itinéraire de covoiturage à ceux où un conducteur doit prendre une décision (c'est-à-dire prendre une sortie, tourner dans une rue, continuer tout droit ...), car ces points peuvent être le début d'un crochet pour prendre ou déposer un passager. Il n'est pas fait mention du temps de calcul d'un tel itinéraire multimodal. Une des hypothèses faites pour cette modélisation est qu'un conducteur de covoiturage ne fera qu'un seul et unique crochet pour récupérer un passager. Ce qui veut dire, qu'à moins de mutualiser le point de prise et de dépose, ce qui est extrêmement complexe, le conducteur ne remplira sa voiture qu'avec une autre personne alors qu'il pourrait en prendre encore 2 autres.

5.3 Multimodal Connection Scan Algorithm

Le but de l'algorithme MCSA est de fusionner les itinéraires de covoiturage avec les itinéraires de transports en commun. Cependant l'itinéraire de covoiturage (en voiture) autour duquel va s'articuler l'itinéraire de transports en commun n'est pas fixé à l'avance et l'itinéraire de covoi-

turage optimal est calculé par l'algorithme MCSA. De plus, un autre critère important lors d'un covoiturage est de fournir un grand nombre de résultats afin de laisser le choix au passager (par exemple sur la taille de la voiture, la durée du covoiturage, les lignes de transports en commun utilisé avant ou après le covoiturage ...). Du côté du conducteur, il n'est pas réaliste de lui imposer de prendre quelqu'un dans sa voiture. Ainsi, le covoiturage doit être accepté par le conducteur, ce qui renforce notre vision du covoiturage « dynamique » comparé à une méthode de couverture. L'algorithme MCSA permet à un conducteur de covoiturage de prendre plusieurs utilisateurs, avec un très faible surcoût en temps de calcul.

5.3.1 Transformer un itinéraire de covoiturage en connexions

Un itinéraire en covoiturage a un point de départ, un point d'arrivée et une heure de départ, de plus en utilisant les *Points of Action* de [Huang et al., 2018] nous allons avoir tous les points possibles pour un début de crochet. La figure 5.1, montre en blanc et en jaune des rues et avenues de Nice, l'itinéraire en voiture est en vert avec la pastille verte qui est le point de départ et la pastille rouge qui est le point d'arrivée. Les points en bleu représentent les *points of action*, qui sont un possible départ pour un crochet. Pour les points de départ des crochets, nous avons également tous les arrêts de transports en commun accessibles à pied à partir de ces points. Avec l'heure de départ et un calculateur d'itinéraire routier, nous pouvons calculer une heure d'arrivée pour chaque point de départ d'un crochet. Avec toutes ces informations, nous pouvons créer une séquence de connexions pour représenter l'itinéraire de covoiturage.

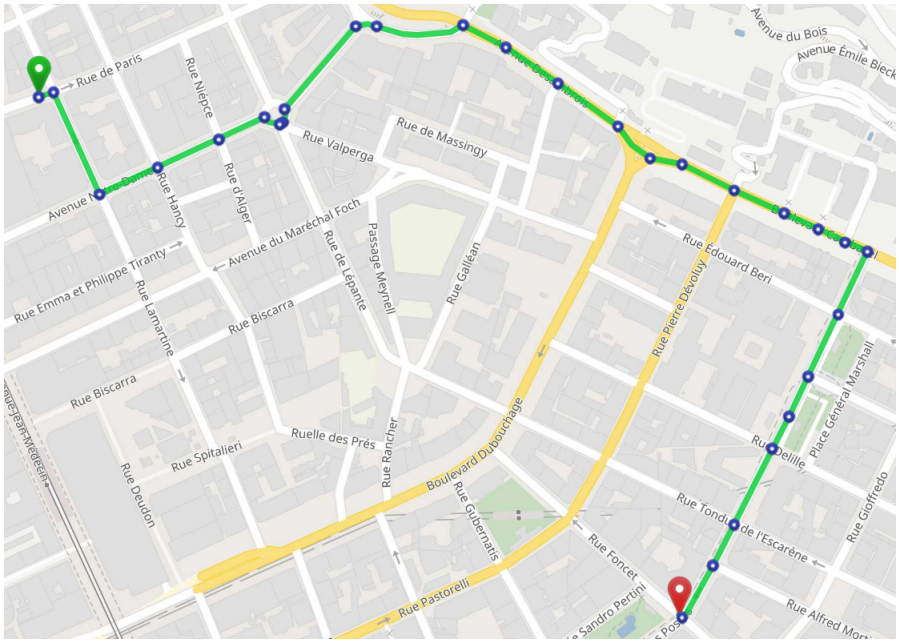


FIGURE 5.1 – Exemple d'un itinéraire en voiture avec ses *points of action*.

Pour un itinéraire de covoiturage quelconque, la première connexion créée a pour arrêt de départ le point de départ du covoiturage, avec comme heure de départ l'heure de départ du covoiturage, comme arrêt d'arrivée le premier *point of action*, avec comme heure d'arrivée l'heure de départ du covoiturage plus la durée pour rejoindre le premier *point of action*, et comme identifiant

de voyage un identifiant créé pour cet itinéraire de covoiturage. Nous faisons ensuite de même pour tous les *points of action* jusqu'à atteindre l'arrivée de l'itinéraire de covoiturage. Grâce à la création de cette séquence de connexions, les itinéraires de covoiturage sont maintenant connectés au réseau de transports en commun et lisibles par un *Connection Scan Algorithm*.

5.3.2 Calculer un itinéraire multimodal

Algorithme 5.1 Pseudocode pour la fonction de domination pour le MCSA

```
function DOMINATION(X, Y)
    if X_rs_id == Y_rs_id and
    X_dep_time ≥ Y_dep_time and
    X_arr_time ≥ Y_arr_time then
        return True
    else
        return False
    end if
end function
```

Avec l'intégration des itinéraires de covoiturage dans la table horaire, en lançant un *Connection Scan Algorithm* il est maintenant possible de renvoyer des résultats intermodaux. Cependant certaines contraintes doivent être ajoutées : un itinéraire ne peut avoir qu'un seul et unique covoiturage parmi ces legs, car dans la pratique aucun utilisateur ne fait un trajet contenant du transport en commun et deux covoiturages différents. Cette contrainte se matérialise par un identifiant de covoiturage a stocké dans chaque étiquette. Quand une connexion de covoiturage est sur le point d'être scannée, si l'identifiant de covoiturage de l'étiquette est identique à celui de la connexion ou si l'identifiant de covoiturage de l'étiquette est nul, alors la connexion est scannée, sinon on passe à la connexion suivante. De plus, plusieurs résultats sont gardés pour chaque arrêt, ce qui veut dire plusieurs étiquettes pour chaque arrêt, et l'algorithme garde une étiquette avec un chemin 100% transports en commun et une étiquette pour chaque chemin avec un covoiturage différent, comme le montre le pseudocode dans l'algorithme 5.1. Le but d'avoir pour un arrêt une étiquette par covoiturage est de ne pas proposer un unique résultat en covoiturage, mais un ensemble de résultats afin de donner à l'utilisateur le choix.

L'algorithme *Multimodal Connection Scan Algorithm* (MCSA) est très similaire au CSA :

- On commence par initialiser les ensembles d'étiquettes pour chaque nœud, les booléens pour savoir si un véhicule a déjà été emprunté et on initialise les nœuds autour du départ.
- Ensuite, la boucle principale va itérer sur les connexions triées par ordre croissant par rapport à leur heure de départ.
- Puis, on itère sur les étiquettes stockées pour l'arrêt de départ de la connexion, si la connexion est un mode de transports en commun ou si l'étiquette est un chemin 100% transports en commun ou si l'étiquette et la connexion sont du même covoiturage alors on passe le booléen du véhicule à True, si c'est une connexion avec du transport en commun.
- Après, on itère sur tous les chemins piétons qui partent de l'arrêt d'arrivée de la connexion et nous mettons à jour les fronts de Pareto avec les nouvelles étiquettes crées, en utilisant la fonction de domination.

L'algorithme MCSA est en pseudocode dans l'algorithme 5.2.

Algorithme 5.2 Pseudocode pour l'algorithme MCSA

```
function MCSA(T, s, t, τ_s)
    for all stops s ∈ T do
        S[s] ← (−∞, +∞, −1)
    end for
    for all footpaths f from s do
        S[s] ← (τ_s, τ_s + f_dur, −1)
    end for

    for all connections c by increasing c_dep_time do
        for all labels ℓ in S[c_dep_stop] do
            if ℓ_arr_time ≤ c_dep_time and
            (c_mode = PT or ℓ_rs_id = −1 or c_rs_id = ℓ_rs_id) then
                for all footpaths f from c_arr_stop do
                    X ← (ℓ_dep_time, c_arr_time + f_dur, c_rs_id)
                    updateParetoFront(S[f_arr_stop], X)
                end for
            end if
        end for
    end for
end function
```

L'algorithme MCSA permet également à un conducteur de covoiturage de prendre plusieurs utilisateurs à des points de prises différents et de les déposer à des endroits différents. Les liens entre le réseau de transports en commun et le réseau de covoiturage se font grâce aux chemins piétons qui n'ont pas d'heure de départ et d'arrivée fixe, contrairement à [Huang et al., 2018], mais ont uniquement une durée pour les traverser. Ainsi, quand un premier utilisateur choisit de prendre un covoiturage, les heures de passages du covoiturage sont mises à jour, c'est-à-dire que ses connexions sont modifiées puis la liste des connexions est triée, ce qui prend à peu près une seconde.

Nous avons un algorithme qui est facile à comprendre, avec peu de lignes de code ainsi que peu d'actions complexes. Ici la complexité vient du maintien du front de Pareto. Les résultats renvoyés sont de multiples itinéraires, un itinéraire utilise uniquement du transport en commun, tandis que tous les autres sont des itinéraires multimodaux mélangeant transports en commun et covoiturage. Contrairement à certains algorithmes présentés dans la partie état de l'art, l'algorithme MCSA ne prend pas en entrée un itinéraire de covoiturage, en fonction de la requête de calcul d'itinéraire c'est l'algorithme lui-même qui trouve le ou les covoiturages optimaux pour l'utilisateur.

L'algorithme MCSA doit équilibrer les avantages et les désavantages pour le conducteur et le passager. Pour le conducteur la durée maximale du crochet du conducteur est fixée à 10 minutes (5 minutes aller et 5 minutes retour), pour éviter de lui faire perdre trop de temps et il est rémunéré en retour. Pour le passager, le covoiturage est payant, mais lui fait gagner du temps sur son trajet et surtout l'algorithme propose plusieurs conducteurs différents afin de donner du choix à l'utilisateur.

5.4 Expérimentations

Dans cette partie, nous évaluons l'implémentation de l'algorithme MCSA fait pour Instant System pour un *proof of concept*, c'est-à-dire pour savoir si l'algorithme MCSA permet bien de faire fusionner les réseaux de transports en commun et de covoiturage. Les expériences sont minimalistes, car les données de covoiturage sont générées aléatoirement et donc l'impact réel sur le calculateur ne peut être qu'entraperçu.

5.4.1 Instances

Réseau	Arrêts	Connexions	Lignes	Voyages	Chemins piéton
Paris	44534	3209401	1864	150963	502291
Berlin	28651	1379755	1296	63569	62456
Stockholm	14258	703326	664	34799	22138
Allemagne	74398	3601420	3599	168024	599284
Suisse	29844	2599675	5645	248826	27202

TABLE 5.1 – Taille des différents réseaux.

De la même manière que précédemment, nos instances de tests sont basées sur les données des réseaux de transports en commun de trois villes (Paris, Berlin et Stockholm) et de deux réseaux de train nationaux (Allemagne et Suisse). Les données sont accessibles de manière libre via un flux GTFS (`https://transitfeeds.com/`) qui a été téléchargé en octobre 2019. Les données de 2019 sont tout à fait comparables avec celles de 2022, les réseaux de transports en commun changent très peu. Le détail des tailles des différents réseaux est disponible dans le tableau 5.1.

Les chemins piétons sont donnés dans le flux GTFS, mais le graphe n'est pas transitivement fermé : les chemins manquants sont engendrés explicitement.

Les algorithmes sont implémentés en Java 11 et exécutés avec OpenJDK 11. Toutes nos expérimentations sont lancées sur un processeur Intel Core i7-1185G7 avec 32 Go de RAM.

Pour notre évaluation, nous avons lancé tous les algorithmes avec le même jeu de 1000 requêtes généré aléatoirement. L'arrêt de départ et l'arrêt d'arrivée sont choisis uniformément au hasard. L'heure de départ est choisie de manière aléatoire entre 6h30 et 7h30. Les itinéraires de covoiturage sont aussi générés de manière aléatoire, l'heure de départ du covoiturage est fixée entre 7h et 10h aussi de manière aléatoire.

5.4.2 Résultats de l'algorithme MCSA

Nous présentons les résultats des algorithmes MCSA et CSA. Le tableau 5.2 donne la moyenne des temps d'exécution, du temps de trajet et du nombre de résultats.

Nous voyons que le temps de l'algorithme MCSA est de 13 à 35 fois plus lent que l'algorithme CSA. Cependant l'algorithme MCSA renvoie de 7 à 15 résultats en moyenne avec un temps moyen de trajet qui est réduit de 86 à 20 minutes. Nous pouvons voir que le temps de trajet sur le réseau de l'Allemagne est plus long pour l'algorithme MCSA que pour l'algorithme CSA. Cela vient du fait que 573 résultats n'ont pas de solution en utilisant l'algorithme CSA et que seulement 36 résultats n'en ont pas en utilisant l'algorithme MCSA. L'algorithme MCSA a permis de trouver des

Réseau	Algorithme	Temps de réponse (ms)	Temps de trajet (min)	# Résultats
Paris	CSA	24,4	169,8	0,99
	MCSA	861,9	146,1	15,1
Berlin	CSA	9,5	227,7	0,99
	MCSA	338,5	207,4	12,5
Stockholm	CSA	4,5	194,2	0,95
	MCSA	155,7	154,1	12,8
Allemagne	CSA	49,9	299,1	0,42
	MCSA	1524,9	362,3	7,1
Suisse	CSA	44,5	353,8	0,81
	MCSA	603,6	267,3	9,7

TABLE 5.2 – Détails de l'algorithme CSA et MCSA sur des réseaux de transports métropolitain et de train nationaux.

résultats pour une très grande partie des itinéraires qui n'ont pas de résultat en utilisant uniquement l'algorithme CSA.

Nous pouvons également voir que, pour les réseaux métropolitains, l'algorithme MCSA a un nombre de résultats qui est toujours supérieur à 10 et dépasse même les 15 pour le réseau de Paris. Tandis que, pour les réseaux de trains nationaux, le nombre de résultats pour l'algorithme MCSA est en dessous de 10. Cela peut être dû au fait que les réseaux métropolitains sont beaucoup plus denses et que des covoiturages peuvent être empruntés par un très grand nombre d'itinéraires. Au contraire des réseaux nationaux, qui sont extrêmement épars ce qui rend plus difficile l'utilisation d'un covoiturage par de multiples itinéraires.

Cependant, un atout de l'algorithme MCSA pour les réseaux nationaux est la possibilité de renvoyer un itinéraire que l'algorithme CSA ne trouve pas. L'exemple le plus flagrant est l'Allemagne avec 573 itinéraires sans résultat avec l'algorithme CSA mais uniquement 36 itinéraires sans résultat pour l'algorithme MCSA.

Pour conclure, nous pouvons voir que l'algorithme MCSA permet de renvoyer des résultats multimodaux, les temps de réponse sont plus lents que l'algorithme CSA mais le nombre de résultats est multiplié par 10 en moyenne. Même si les itinéraires de covoiturages sont générés aléatoirement, l'algorithme MCSA permet l'utilisation du covoiturage dans un algorithme de calcul d'itinéraires pour les transports en commun avec un temps de réponse inférieur à 1 seconde pour des réseaux métropolitains et moins de 2 secondes pour des réseaux nationaux de trains.

5.5 Synthèse

Nous avons commencé le développement d'un algorithme qui mêle le covoiturage et l'algorithme CSA, l'algorithme MCSA. Le covoiturage peut être intégré dans un réseau de transports en commun, car les deux sont extrêmement similaires, un véhicule qui passe à des arrêts à un horaire. La découpe des chemins de covoiturage est facilité par l'utilisation des *Points of Action* de [Huang et al., 2018]. Les résultats de nos expérimentations, pour le *proof of concept* d'*Instant*

System, sur des réseaux métropolitains et des réseaux nationaux de trains montrent un temps de réponse inférieure à 2 secondes et plus de 10 résultats en moyenne. De plus, pour des réseaux épars, l'algorithme MCSA permet de trouver des résultats intermodaux, alors que l'algorithme CSA ne permet pas de trouver de résultats monomodaux.

Pour nos futurs travaux, les jeux de données générés aléatoirement pour les itinéraires de covoiturage pourront être remplacés par de vrais itinéraires de covoiturage, utilisés sur des réseaux en production d'*Instant System*, anonymisés pour répondre au besoin du RGPD. Les requêtes d'itinéraire générées elles aussi aléatoirement pourront être remplacées par de vraies requêtes d'itinéraire tirées de l'historique d'*Instant System*. De plus, l'étude des résultats pourra être plus poussée, car les requêtes et itinéraires de covoiturage ne seront plus aléatoires.

CHAPITRE 6

Conclusion et Perspectives

6.1 Conclusion

Le problème spécifique qui nous intéresse est le calcul d'itinéraires pour les transports en commun multiobjectif avec un point de départ et un point d'arrivée qui change. Ce choix est motivé par les besoins des utilisateurs qui veulent de multiples résultats pour représenter au mieux l'ensemble des possibilités. Tous les algorithmes présentés dans le chapitre 2 utilisent des fronts de Pareto avec au maximum trois critères : l'heure d'arrivée, l'heure de départ et le nombre de transferts. Cependant, pour fournir les résultats les plus qualitatifs, un critère supplémentaire est nécessaire : la distance de marche. L'ajout d'un critère supplémentaire dans le front de Pareto a pour conséquence d'augmenter fortement les temps de réponse. Mais les utilisateurs, en plus de résultats qualitatifs, attendent un temps de réponse extrêmement faible, en dessous de 2 secondes, ce qui est en opposition complète avec un front de Pareto à 4 critères. L'algorithme GDCSA a été créé en partant de ces besoins, un calculateur d'itinéraires rapide pour des fronts de Pareto avec de nombreux critères. GDCSA se base sur une idée simple qui découpe un réseau en un ensemble de zones et précalcule des bornes inférieures des durées de trajet entre chaque zone, ce qui est en fait un précalcul robuste. Puis lors d'un calcul d'itinéraires entre s et t, en utilisant les bornes inférieures des durées de trajet entre les zones et la borne supérieure de la durée du trajet entre s et t, les zones par lesquelles passent les chemins entre s et t appartenant au front de Pareto sont gardées tandis que les autres sont éliminées. Cette diminution de l'espace de recherche a pour conséquence la baisse du temps de réponse d'un facteur d'au plus 7 par rapport à l'algorithme CSA pour un front de Pareto à 4 critères.

Une autre approche développée pour proposer une alternative au calcul d'itinéraires pour les transports en commun multiobjectif est l'utilisation du problème des k plus courts chemins. Nous proposons deux algorithmes :

- l'algorithme Y-PT est une variante de l'algorithme de Yen pour les réseaux de transports en commun ;
- l'algorithme PY-PT est une variante de l'algorithme PNC pour les réseaux de transports en commun.

L'algorithme Y-PT remplace l'algorithme de Dijkstra par l'algorithme CSA, qui est le plus rapide pour calculer un itinéraire monobjectif sans précalcul. Cependant, le défaut principal de l'algorithme de Yen qui est le grand nombre d'appels à l'algorithme de plus court chemin ou de calcul d'itinéraire d'arrivée au plus tôt, se retrouve dans l'algorithme Y-PT. L'algorithme PY-PT résout ce problème, de la même manière que l'algorithme PNC, en calculant un arbre d'arrivée

au plus tôt avec l'algorithme PCSA. Comme pour l'algorithme Y-PT, l'algorithme PY-PT calcule les détours avec interdiction pour chaque nœud de l'itinéraire, mais utilise l'arbre de plus court chemin plutôt qu'un CSA, à la différence que l'itinéraire peut être non simple. Quand un itinéraire non simple est extrait de l'ensemble des candidats, il est « réparé » en recalculant son détour avec l'algorithme CSA. Le but est d'avoir le moins de détours non simples, donc moins de réparations à faire pour que le surcoût de l'algorithme PCSA soit amorti. Nos expériences ont montré que c'était bien le cas : l'algorithme PY-PT est 10 fois plus rapide que l'algorithme Y-PT pour $k = 100$ et même pour $k = 2$ car très peu de « réparations » sont faites. De plus, les résultats fournis par les algorithmes Y-PT et PY-PT sont dissimilaires, ce qui permet de fournir des résultats intéressants aux utilisateurs.

Le covoiturage et les réseaux de transports en commun bien qu'utilisant deux modes de transports très différents, la voiture et les transports en commun, ont tout de même un aspect très similaire avec un véhicule qui passe à des arrêts à certains horaires. Nous avons commencé le développement d'un algorithme qui mêle le covoiturage et l'algorithme CSA, l'algorithme MCSA. Les réseaux routiers et de transports en commun sont liés en utilisant les *points of action* de [Huang et al., 2018], mais en s'éloignant de l'utilisation d'un réseau expansé dans le temps au profit des connexions de l'algorithme CSA. Les résultats de nos expériences, sur des données générées aléatoirement, pour le *proof of concept* d'*Instant System* sur des réseaux métropolitains et nationaux de trains permettent de montrer qu'avec des temps de réponse inférieure à 2 secondes, l'algorithme MCSA permet de trouver en moyenne plus de dix résultats. De plus, il permet également sur des réseaux épars de trouver des résultats intermodaux pour des requêtes alors qu'aucun résultat monomodal n'existe.

6.2 Perspectives

L'élaboration de l'algorithme GDCSA avait pour but de créer un calcul d'itinéraires multiobjectifs rapide même sur de très grands réseaux de transports en commun. Comme le montrent les expériences du chapitre 3, les performances sur le réseau de Paris ne sont toujours pas à la hauteur de la demande des utilisateurs, c'est-à-dire un temps de réponse inférieur à 2 secondes. Même si nous avons de très gros gains comparés à une version sans amélioration de l'algorithme CSA, il reste encore plusieurs pistes à explorer pour diminuer encore les temps de réponse.

La première piste pour améliorer les performances de l'algorithme GDCSA serait d'améliorer la borne inférieure. La borne inférieure actuelle calcule la durée minimum de trajet dans la journée pour passer d'une région à une autre, une possibilité d'amélioration serait d'ajouter de la temporalité. Par exemple, une durée minimum de trajet pour aller d'une région a à une région b entre 2 heures et 6 heures du matin, puis entre 6 heures et 10 heures du matin ... La granularité reste à définir en fonction du gain pour le temps de réponse et la quantité de stockage nécessaire pour le prétraitement. Le but est d'avoir la borne inférieure la plus précise, ce qui diminue le nombre de régions considérées et ce qui diminue le temps de réponse.

Une autre piste est la modification de l'algorithme de partitionnement, soit en utilisant d'autres configurations pour METIS ou KaHiP, soit en utilisant des heuristiques. Pour les heuristiques, [Maue et al., 2010] ont de bons résultats sur les réseaux routiers en essayant de partitionner le graphe en un ensemble de zones de tailles similaires et avec un diamètre faible. Il faut donc vérifier si cette heuristique permet d'avoir de bons résultats aussi sur les réseaux de transports en commun.

La dernière piste à explorer est l'approximation du front de Pareto afin de diminuer le nombre d'étiquettes stockées pendant un calcul d'itinéraire. Par exemple, si une étiquette qui est sur le point d'être ajoutée au front de Pareto a plus de 3 transferts qu'un itinéraire « ancre », alors cette étiquette est supprimée, car son résultat ne devrait pas être intéressant. Cette idée est déjà utilisée pour l'algorithme RAPTOR [Delling et al., 2019]. Une possibilité serait de transposer cette idée et les valeurs d'écarts à l'algorithme GDCSA, si la transposition est possible. Cependant, approximer le front de Pareto peut empêcher de trouver des résultats optimaux, ce qui dans un contexte industriel et commercial peut coûter très cher quand un client ne trouve pas un résultat attendu.

Une perspective pour les algorithmes Y-PT et PY-PT est d'étudier le remplacement de l'algorithme CSA par un algorithme utilisant du précalcul, par exemple l'algorithme *Transfer Patterns*, l'algorithme *Trip Based Routing* ou l'algorithme *Public Transit Labeling*. La difficulté étant que pendant le calcul des k plus courts chemins, nous devons supprimer des arcs du graphe ce qui peut être une tâche complexe pour des algorithmes utilisant du précalcul.

Une autre perspective serait de poursuivre l'évaluation des algorithmes Y-PT et PY-PT. Nous avons étudié la dissimilarité des résultats. Un axe intéressant à poursuivre serait de comparer la valeur de k nécessaire pour retrouver les résultats d'un front de Pareto à 2, 3 ou 4 critères puis de comparer les temps de réponse de chaque méthode. Le maintien du front de Pareto est une opération coûteuse tandis que l'algorithme PY-PT peut renvoyer un chemin pour un coût minime. Cependant, un grand nombre des chemins renvoyés par l'algorithme PY-PT sont de simple substitution d'arcs. Ainsi, il serait intéressant de savoir quelle méthode est la plus rapide.

Une autre question intéressante sur la dissimilarité serait de comparer les résultats obtenus avec les algorithmes Y-PT et PY-PT aux résultats d'algorithmes dont le but est de renvoyer des chemins dissimilaires. Pour les réseaux routiers, il existe [Chondrogiannis et al., 2017] ce qui implique d'en faire une adaptation pour les réseaux de transports en commun.

Une autre perspective est d'évaluer les performances de l'algorithme MCSA sur des jeux de données réels, récupérés chez *Instant System* et anonymisés pour répondre au besoin du RGPD. Ce jeu de données sera en libre accès avec les données du réseau de transports en commun de la ville en question. Le but sera de quantifier le temps de création d'une requête, qui doit être transformée en connexions en utilisant ses *points of action*, ainsi que le temps de réponse moyen d'un calcul d'itinéraires multimodal. Une autre comparaison intéressante serait de regarder la différence des durées entre les itinéraires monomodaux et les itinéraires multimodaux, mais également, en comparaison avec d'autres algorithmes qui mêlent covoiturage et transports en commun.

Et finalement, la dernière perspective serait d'étudier la prise en compte d'un temps de passage du covoiturage un peu plus flou, afin de permettre une correspondance plus facile entre utilisateur et conducteur. Dans la version présentée de l'algorithme MCSA, seul l'utilisateur peut attendre le covoiturage. Si le conducteur passe une minute après l'heure d'arrivée de l'utilisateur à un *point of action* alors le covoiturage ne peut pas se faire. Il faut permettre à l'algorithme de prendre en compte l'attente du conducteur, cela implique la modification des heures de départ et d'arrivée des connexions de covoiturage.

Bibliographie

[Agatz et al., 2012] Agatz, N., Erera, A., Savelsbergh, M., and Wang, X. (2012). Optimization for dynamic ride-sharing : A review. *European Journal of Operational Research*, 223(2) :295–303.

[Aissat and Varone, 2015] Aissat, K. and Varone, S. (2015). Carpooling as complement to multimodal transportation. In *International Conference on Enterprise Information Systems*, pages 236–255. Springer.

[Al-Zoobi et al., 2022] Al-Zoobi, A., Coudert, D., Finkelstein, A., and Régin, J.-C. (2022). On finding k earliest arrival time journeys in public transit networks. In *11th International Conference on Operations Research and Enterprise Systems (ICORES)*, pages 314–325.

[Al Zoobi et al., 2020] Al Zoobi, A., Coudert, D., and Nisse, N. (2020). Space and time trade-off for the k shortest simple paths problem. In *18th International Symposium on Experimental Algorithms (SEA)*, volume 160, page 13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Al Zoobi et al., 2021a] Al Zoobi, A., Coudert, D., and Nisse, N. (2021a). Finding the k Shortest Simple Paths : Time and Space trade-offs. Research report, Inria ; I3S, Université Côte d'Azur.

[Al Zoobi et al., 2021b] Al Zoobi, A., Coudert, D., and Nisse, N. (2021b). On the complexity of finding k shortest dissimilar paths in a graph. Research report, Inria ; CNRS ; I3S ; UCA.

[Barrett et al., 2008] Barrett, C., Bisset, K., Holzer, M., Konjevod, G., Marathe, M., and Wagner, D. (2008). Engineering label-constrained shortest-path algorithms. In *International conference on algorithmic applications in management*, pages 27–37. Springer.

[Bast et al., 2010] Bast, H., Carlsson, E., Eigenwillig, A., Geisberger, R., Harrelson, C., Raychev, V., and Viger, F. (2010). Fast routing in very large public transportation networks using transfer patterns. In *European Symposium on Algorithms*, pages 290–301. Springer.

[Bast et al., 2016] Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. (2016). Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer.

[Bast and Storandt, 2014] Bast, H. and Storandt, S. (2014). Frequency-based search for public transit. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22.

[Bauer et al., 2010] Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., and Wagner, D. (2010). Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 15 :2–1.

[Bellman, 1958] Bellman, R. (1958). On a routing problem. *Quarterly of applied mathematics*, 16(1) :87–90.

[Berge, 1983] Berge, C. (1983). *Graphes*, volume 2. Gauthier-villars Paris.

[Bit-Monnot et al., 2013] Bit-Monnot, A., Artigues, C., Huguet, M.-J., and Killijian, M.-O. (2013). Carpooling : the 2 synchronization points shortest paths problem. In *13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Bongiorno et al., 2021] Bongiorno, C., Zhou, Y., Kryven, M., Theurel, D., Rizzo, A., Santi, P., Tenenbaum, J., and Ratti, C. (2021). Vector-based pedestrian navigation in cities. *arXiv preprint arXiv :2103.07104*.

[Brodal and Jacob, 2004] Brodal, G. S. and Jacob, R. (2004). Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92 :3–15.

[Chondrogiannis et al., 2017] Chondrogiannis, T., Bouros, P., Gamper, J., and Leser, U. (2017). Exact and approximate algorithms for finding k-shortest paths with limited overlap. In *Proceedings of the 20th International Conference on Extending Database Technology, (EDBT)*, pages 414–425, Venice, Italy. OpenProceedings.org.

[Cohen et al., 2003] Cohen, E., Halperin, E., Kaplan, H., and Zwick, U. (2003). Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5) :1338–1355.

[Cooke and Halsey, 1966] Cooke, K. L. and Halsey, E. (1966). The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications*, 14(3) :493–498.

[Cordeau, 2006] Cordeau, J.-F. (2006). A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3) :573–586.

[Crauser et al., 1998] Crauser, A., Mehlhorn, K., Meyer, U., and Sanders, P. (1998). A parallelization of dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer.

[Dantzig, 1963] Dantzig, G. (1963). *Linear programming and extensions*. Princeton university press.

[Davidson et al., 2014] Davidson, A., Baxter, S., Garland, M., and Owens, J. D. (2014). Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE.

[Dean, 1999] Dean, B. C. (1999). *Continuous-time dynamics shortest path algorithms*. PhD thesis, Massachusetts Institute of Technology.

[Delling et al., 2019] Delling, D., Dibbelt, J., and Pajor, T. (2019). Fast and exact public transit routing with restricted pareto sets. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 54–65. SIAM.

[Delling et al., 2015a] Delling, D., Dibbelt, J., Pajor, T., and Werneck, R. F. (2015a). Public transit labeling. In *International Symposium on Experimental Algorithms*, pages 273–285. Springer.

[Delling et al., 2017] Delling, D., Dibbelt, J., Pajor, T., and Zündorf, T. (2017). Faster transit routing by hyper partitioning. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Delling et al., 2011a] Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. (2011a). Customizable route planning. In *International Symposium on Experimental Algorithms*, pages 376–387. Springer.

[Delling et al., 2014] Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. (2014). Robust distance queries on massive networks. In *European Symposium on Algorithms*, pages 321–333. Springer.

[Delling et al., 2011b] Delling, D., Goldberg, A. V., Razenshteyn, I., and Werneck, R. F. (2011b). Graph partitioning with natural cuts. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 1135–1146. IEEE.

[Delling et al., 2012] Delling, D., Katz, B., and Pajor, T. (2012). Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics (JEA)*, 17 :4–1.

[Delling et al., 2015b] Delling, D., Pajor, T., and Werneck, R. F. (2015b). Round-based public transit routing. *Transportation Science*, 49(3) :591–604.

[Dial, 1969] Dial, R. B. (1969). Algorithm 360 : Shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11) :632–633.

[Dibbelt et al., 2018] Dibbelt, J., Pajor, T., Strasser, B., and Wagner, D. (2018). Connection scan algorithm. *Journal of Experimental Algorithmics (JEA)*, 23 :1–56.

[Dijkstra et al., 1959] Dijkstra, E. W. et al. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1) :269–271.

[Disser et al., 2008] Disser, Y., Müller-Hannemann, M., and Schnee, M. (2008). Multi-criteria shortest paths in time-dependent train networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 347–361. Springer.

[Dreyfus, 1969] Dreyfus, S. E. (1969). An appraisal of some shortest-path algorithms. *Operations research*, 17(3) :395–412.

[Dustdar et al., 2017] Dustdar, S., Nastić, S., and Šćekić, O. (2017). Smart cities. In *The Internet of Things, People and Systems*. Springer.

[Eppstein, 1998] Eppstein, D. (1998). Finding the k shortest paths. *SIAM Journal on Computing*, 28(2) :652–673.

[Eppstein and Goodrich, 2008] Eppstein, D. and Goodrich, M. T. (2008). Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10.

[Finkelstein and Régin, 2020] Finkelstein, A. and Régin, J.-C. (2020). Using goal directed techniques for journey planning with multi-criteria range queries in public transit. *SCITEPRESS Digital Library*.

[Ford Jr, 1956] Ford Jr, L. R. (1956). Network flow theory. Technical report, Rand Corp Santa Monica Ca.

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3) :596–615.

[Furuhata et al., 2013] Furuhata, M., Dessouky, M., Ordóñez, F., Brunet, M.-E., Wang, X., and Koenig, S. (2013). Ridesharing : The state-of-the-art and future directions. *Transportation Research Part B : Methodological*, 57 :28–46.

[Geisberger et al., 2012] Geisberger, R., Sanders, P., Schultes, D., and Vetter, C. (2012). Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3) :388–404.

[Goldberg and Harrelson, 2005] Goldberg, A. V. and Harrelson, C. (2005). Computing the shortest path : A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM*

symposium on Discrete algorithms, pages 156–165. Society for Industrial and Applied Mathematics.

[Goldberg et al., 2006] Goldberg, A. V., Kaplan, H., and Werneck, R. F. (2006). Reach for a* : Shortest path algorithms with preprocessing. In *The shortest path problem*, pages 93–139. Citeseer.

[Harish and Narayanan, 2007] Harish, P. and Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer.

[Hart et al., 1968] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2) :100–107.

[Ho et al., 2018] Ho, S. C., Szeto, W. Y., Kuo, Y.-H., Leung, J. M., Petering, M., and Tou, T. W. (2018). A survey of dial-a-ride problems : Literature review and recent developments. *Transportation Research Part B : Methodological*, 111 :395–421.

[Holzer et al., 2005] Holzer, M., Schulz, F., Wagner, D., and Willhalm, T. (2005). Combining speed-up techniques for shortest-path computations. *Journal of Experimental Algorithmics (JEA)*, 10 :2–5.

[Huang et al., 2018] Huang, H., Bucher, D., Kissling, J., Weibel, R., and Raubal, M. (2018). Multimodal route planning with public transport and carpooling. *IEEE Transactions on Intelligent Transportation Systems*, 20(9) :3513–3525.

[Jamal et al., 2017] Jamal, J., Montemanni, R., Huber, D., Derboni, M., and Rizzoli, A. E. (2017). A multi-modal and multi-objective journey planner for integrating carpooling and public transport. *Journal of Traffic and Logistics Engineering Vol*, 5(2) :68–72.

[Johnson, 2002] Johnson, D. S. (2002). A theoretician's guide to the experimental analysis of algorithms. *Data structures, near neighbor searches, and methodology : fifth and sixth DIMACS implementation challenges*, 59 :215–250.

[Jung and Pramanik, 2002] Jung, S. and Pramanik, S. (2002). An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5) :1029–1046.

[Karypis and Kumar, 1998] Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1) :359–392.

[Kurz and Mutzel, 2016] Kurz, D. and Mutzel, P. (2016). A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. *arXiv preprint arXiv :1601.02867*.

[Lauther, 2006] Lauther, U. (2006). An experimental evaluation of point-to-point shortest path calculation on road networks with precalculated edge-flags. In *The Shortest Path Problem*, pages 19–39.

[Lipton and Tarjan, 1979] Lipton, R. J. and Tarjan, R. E. (1979). A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2) :177–189.

[Masri et al., 2017] Masri, A., Zeitouni, K., and Kedad, Z. (2017). Retry : Integrating ridesharing with existing trip planners. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10.

[Maue et al., 2010] Maue, J., Sanders, P., and Matijevic, D. (2010). Goal-directed shortest-path queries using precomputed cluster distances. *Journal of Experimental Algorithmics (JEA)*, 14 :3–2.

[Meyer and Sanders, 1998] Meyer, U. and Sanders, P. (1998). Delta-stepping : A parallel single source shortest path algorithm. In *European symposium on algorithms*, pages 393–404. Springer.

[Moore, 1959] Moore, E. F. (1959). The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292.

[Müller-Hannemann and Schnee, 2007] Müller-Hannemann, M. and Schnee, M. (2007). Finding all attractive train connections by multi-criteria pareto search. In *Algorithmic Methods for Railway Optimization*, pages 246–263. Springer.

[Müller-Hannemann et al., 2007] Müller-Hannemann, M., Schulz, F., Wagner, D., and Zaroliagis, C. (2007). Timetable information : Models and algorithms. In *Algorithmic Methods for Railway Optimization*, pages 67–90. Springer.

[Müller-Hannemann and Weihe, 2001] Müller-Hannemann, M. and Weihe, K. (2001). Pareto shortest paths is often feasible in practice. In *International Workshop on Algorithm Engineering*, pages 185–197. Springer.

[Nachtigall, 1995] Nachtigall, K. (1995). Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1) :154–166.

[Ortega-Arranz et al., 2013] Ortega-Arranz, H., Torres, Y., Llanos, D. R., and Gonzalez-Escribano, A. (2013). A new gpu-based approach to the shortest path problem. In *2013 International Conference on High Performance Computing & Simulation (HPCS)*, pages 505–511. IEEE.

[Pallottino and Scutella, 1998] Pallottino, S. and Scutella, M. G. (1998). Shortest path algorithms in transportation models : classical and innovative aspects. In *Equilibrium and advanced transportation modelling*, pages 245–281. Springer.

[Posada et al., 2017] Posada, M., Andersson, H., and Häll, C. H. (2017). The integrated dial-a-ride problem with timetabled fixed route service. *Public Transport*, 9(1) :217–241.

[Pyrga et al., 2008] Pyrga, E., Schulz, F., Wagner, D., and Zaroliagis, C. (2008). Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12 :1–39.

[Pyrga et al., 2004] Pyrga, E., Schulz, F., Wagner, D., and Zaroliagis, C. D. (2004). Experimental comparison of shortest path approaches for timetable information. In *ALENEX/ANALC*, pages 88–99. Citeseer.

[Ropke et al., 2007] Ropke, S., Cordeau, J.-F., and Laporte, G. (2007). Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks : An International Journal*, 49(4) :258–272.

[Sanders and Schulz, 2013] Sanders, P. and Schulz, C. (2013). Think locally, act globally : Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933, pages 164–175. Springer.

[Santos et al., 2011] Santos, A., McGuckin, N., Nakamoto, H. Y., Gray, D., Liss, S., et al. (2011). Summary of travel trends : 2009 national household travel survey. Technical report, United States. Federal Highway Administration.

[Scano et al., 2015] Scano, G., Huguet, M.-J., and Ngueveu, S. U. (2015). Adaptations of k-shortest path algorithms for transportation networks. In *2015 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 663–669. IEEE.

[Schenekemberg et al., 2022] Schenekemberg, C. M., Chaves, A. A., Coelho, L. C., Guimarães, T. A., and Avelino, G. G. (2022). The dial-a-ride problem with private fleet and common carrier. *Computers & Operations Research*, 147 :105933.

[Schild and Sommer, 2015] Schild, A. and Sommer, C. (2015). On balanced separators in road networks. In *International Symposium on Experimental Algorithms*, pages 286–297. Springer.

[Schulz et al., 2000] Schulz, F., Wagner, D., and Weihe, K. (2000). Dijkstra's algorithm on-line : An empirical case study from public railroad transport. *Journal of Experimental Algorithmics (JEA)*, 5 :12–es.

[Sierpiński, 2013] Sierpiński, G. (2013). Changes of the modal split of traffic in europe. *Archives of Transport System Telematics*, 6.

[Strasser and Wagner, 2014] Strasser, B. and Wagner, D. (2014). Connection scan accelerated. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 125–137. SIAM.

[Vo et al., 2015] Vo, K. D., Pham, T. V., Nguyen, H. T., Nguyen, N., and Van Hoai, T. (2015). Finding alternative paths in city bus networks. In *2015 International Conference on Computer, Control, Informatics and its Applications (IC3INA)*, pages 34–39. IEEE.

[Wagner et al., 2005] Wagner, D., Willhalm, T., and Zaroliagis, C. (2005). Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics (JEA)*, 10 :1–3.

[Witt, 2015] Witt, S. (2015). Trip-based public transit routing. In *Algorithms-ESA 2015*, pages 1025–1036. Springer.

[Witt, 2016] Witt, S. (2016). Trip-based public transit routing using condensed search trees. *arXiv preprint arXiv :1607.01299*.

[Yen, 1971] Yen, J. Y. (1971). Finding the k shortest loopless paths in a network. *Management Science*, 17(11) :712–716.

Pages web

[1] Boost c++ libraries. https://www.boost.org/doc/libs/1_80_0/libs/graph_parallel/doc/html/dijkstra_shortest_paths.html.

[2] Dépôt github de l'algorithme pt-kssp. https://github.com/fink-arthur/PT-KSSP.

[3] Jvmalin. https://www.services.jvmalin.fr/fr/.

[4] Les émissions de gaz à effet de serre du secteur des transports. https://www.notre-environnement.gouv.fr/rapport-sur-l-etat-de-l-environnement/themes-ree/defis-environnementaux/changement-climatique/emissions-de-gaz-a-effet-de-serre/article/les-emissions-de-gaz-a-effet-de-serre-du-secteur-des-transports.

[5] Modalis. https://modalis.fr/fr/.

[6] Akamai. Akamai online retail performance report : Milliseconds are critical. https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report.

[7] ADEME Expertises. Le plan de mobilité. https://expertises.ademe.fr/professionnels/entreprises/reduire-impacts/optimiser-mobilite-salaries/dossier/plan-mobilite/plan-mobilite-quest-cest.

[8] Google. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/.

[9] INRIX. Inrix 2021 global traffic scorecard. https://inrix.com/scorecard/.

[10] Légifrance. Code des transports. https://www.legifrance.gouv.fr/codes/section_lc/LEGITEXT000023086525/LEGISCTA000023069061/.

[11] France Mobilités. Le « forfait mobilités durables » créé par la lom est entré en vigueur et voit une revalorisation de son montant depuis le début de l'année ! https://www.francemobilites.fr/actualites/forfait-mobilites-durables-cree-la-lom-entre-en-vigueur-et-voit-une-revalorisation.

[12] Service Public. Entrée en vigueur du forfait mobilités durables. https://www.service-public.fr/particuliers/actualites/A14046.

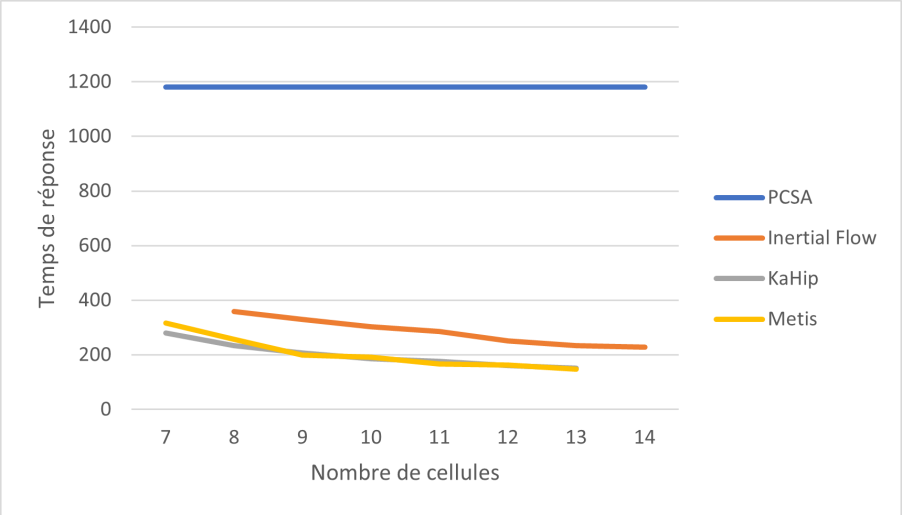
Liste des figures

Liste des tableaux

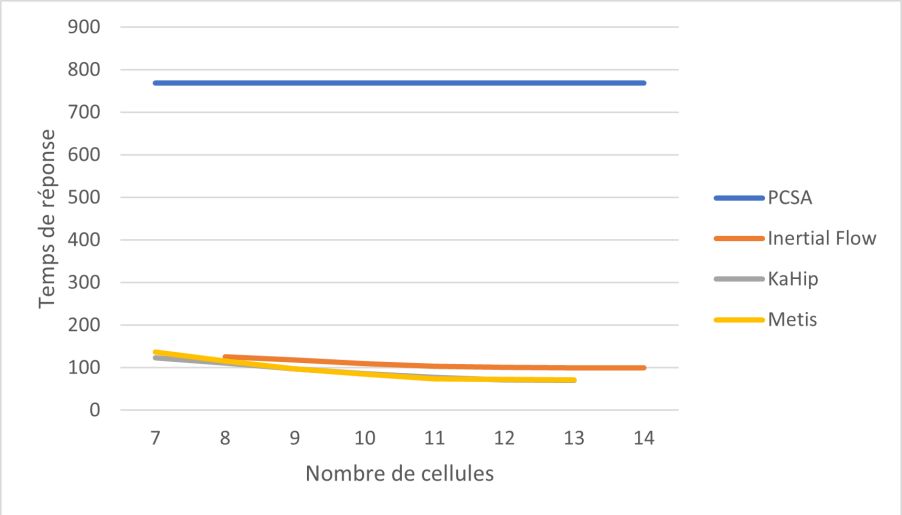
Listes des algorithmes

Annexes

A Figures pour le chapitre 3

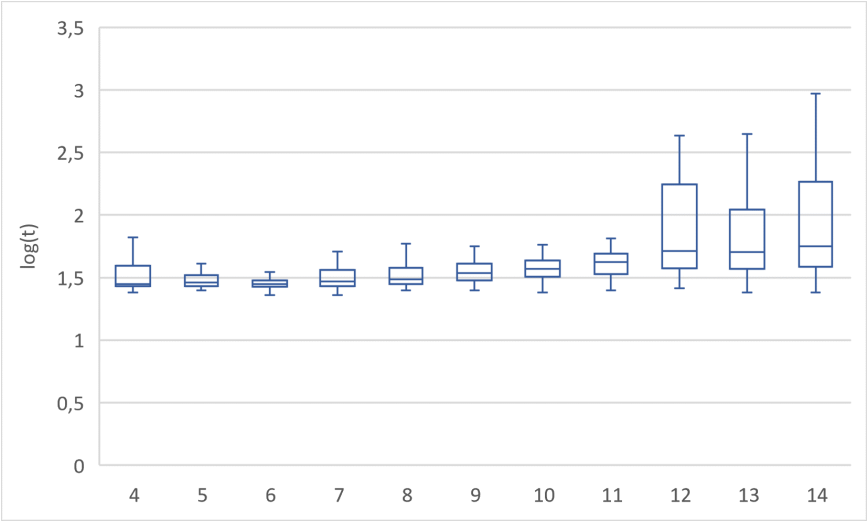


(a) Temps de réponse en fonction du nombre de zones pour le réseau de train de la Suisse

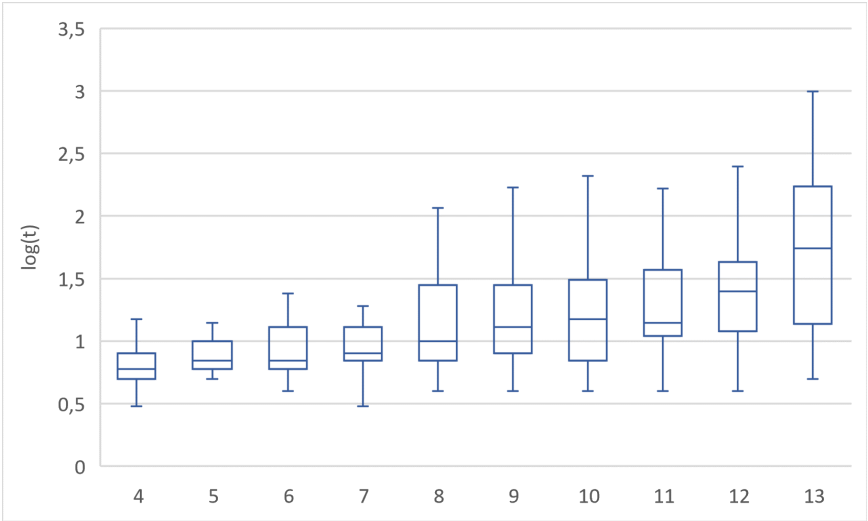


(b) Temps de réponse en fonction du nombre de zones pour la ville de Stockholm

FIGURE A.1 – Temps de réponse en fonction du nombre de zones pour les réseaux de Stockholm et de la Suisse.

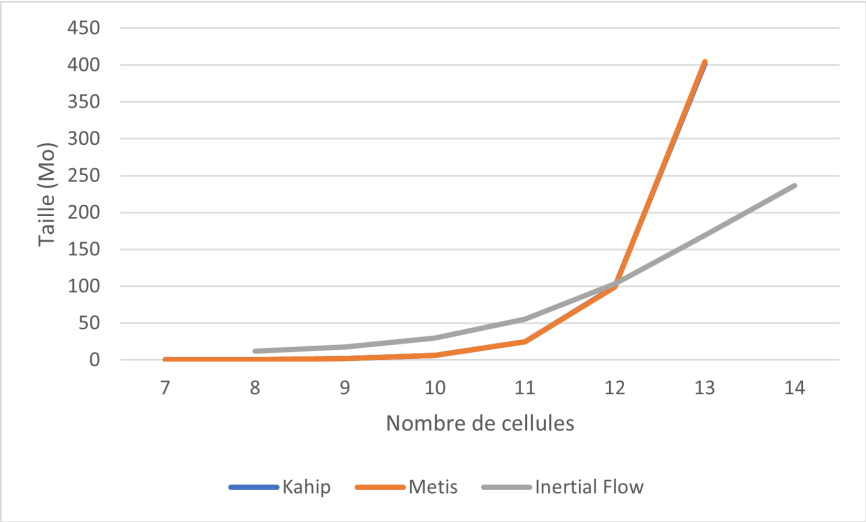


(a) Temps de réponse en fonction du rang géographique pour le réseau de train de la Suisse

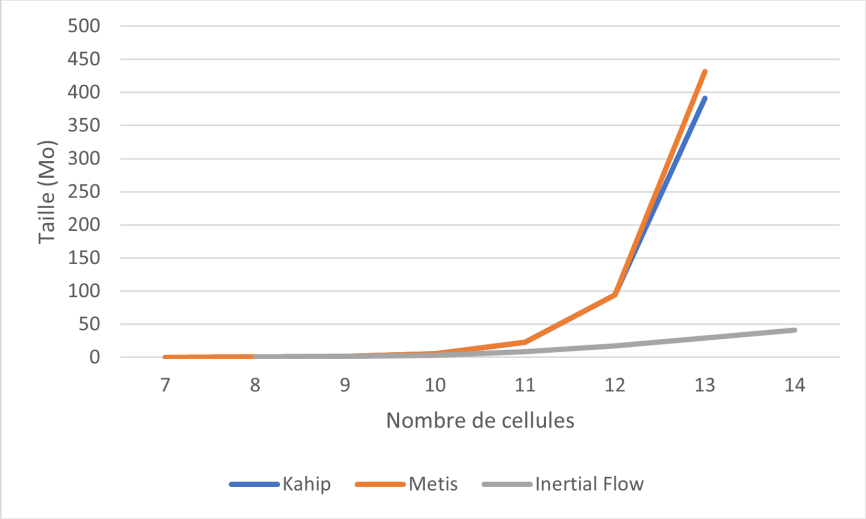


(b) Temps de réponse en fonction du rang géographique pour la ville de Stockholm

FIGURE A.2 – Temps de réponse en fonction du rang géographique pour les réseaux de Stockholm et de la Suisse.

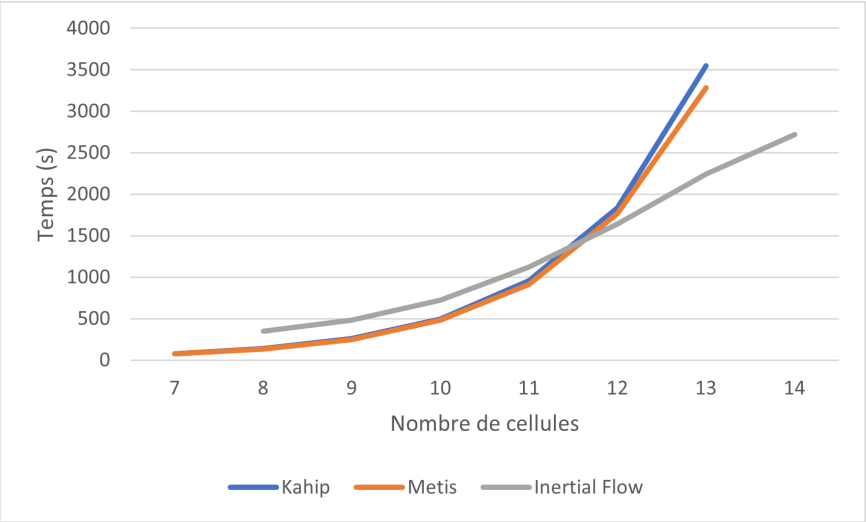


(a) Taille de la partition en fonction du nombre de zones pour le réseau de train de la Suisse

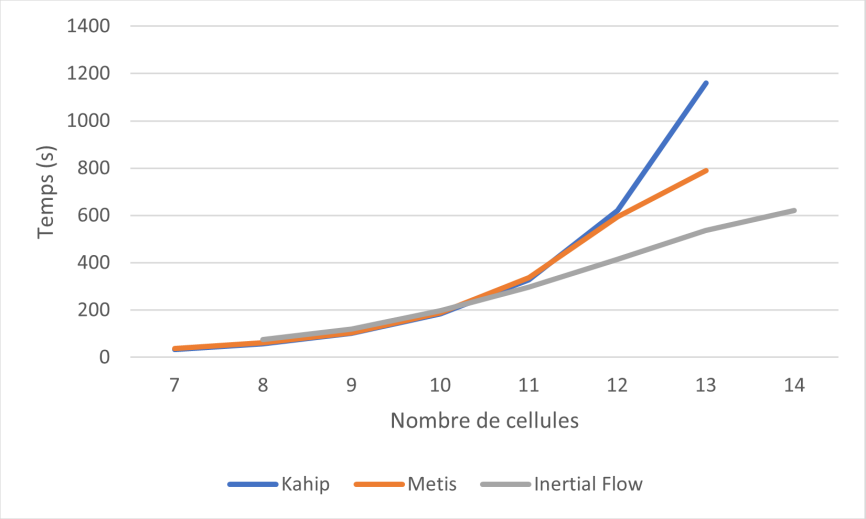


(b) Taille de la partition en fonction du nombre de zones pour la ville de Stockholm

FIGURE A.3 – Taille de la partition en fonction du nombre de zones pour les réseaux de Stockholm et de la Suisse.



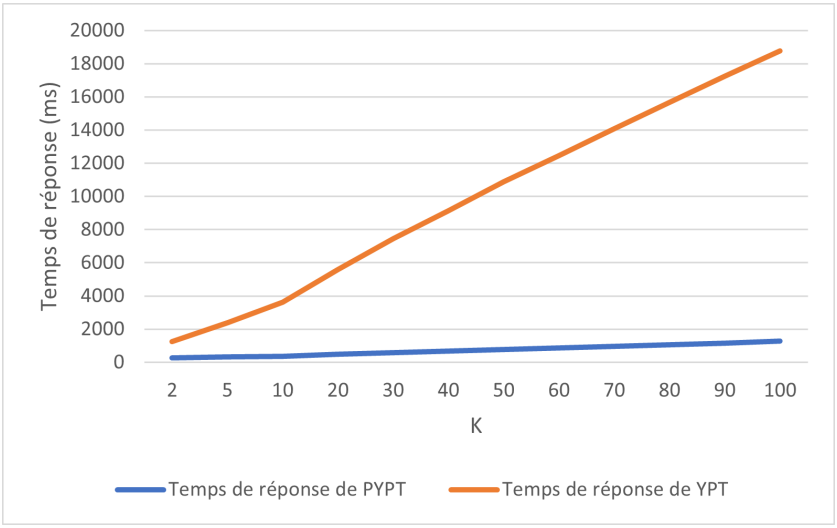
(a) Temps de précalcul en fonction du nombre de zones pour le réseau de train de la Suisse



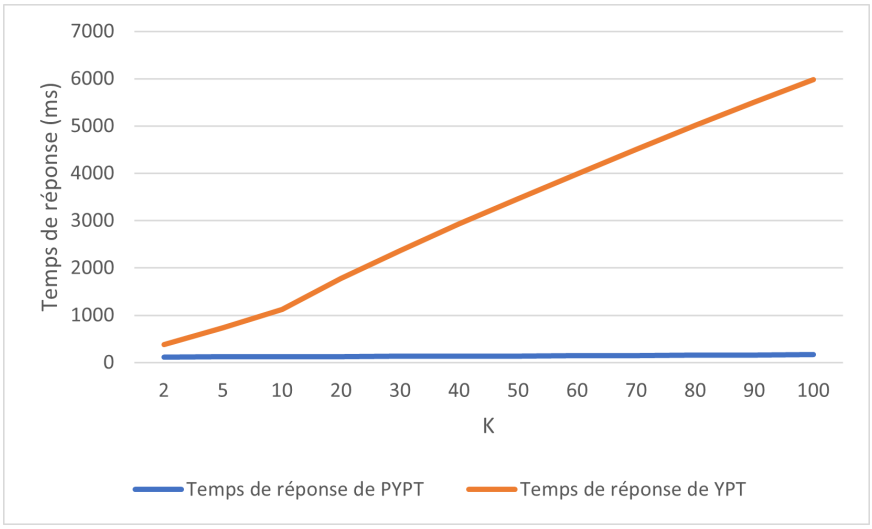
(b) Temps de précalcul en fonction du nombre de zones pour la ville de Stockholm

FIGURE A.4 – Temps de précalcul en fonction du nombre de zones pour les réseaux de Stockholm et de la Suisse.

B Figures pour le chapitre 4

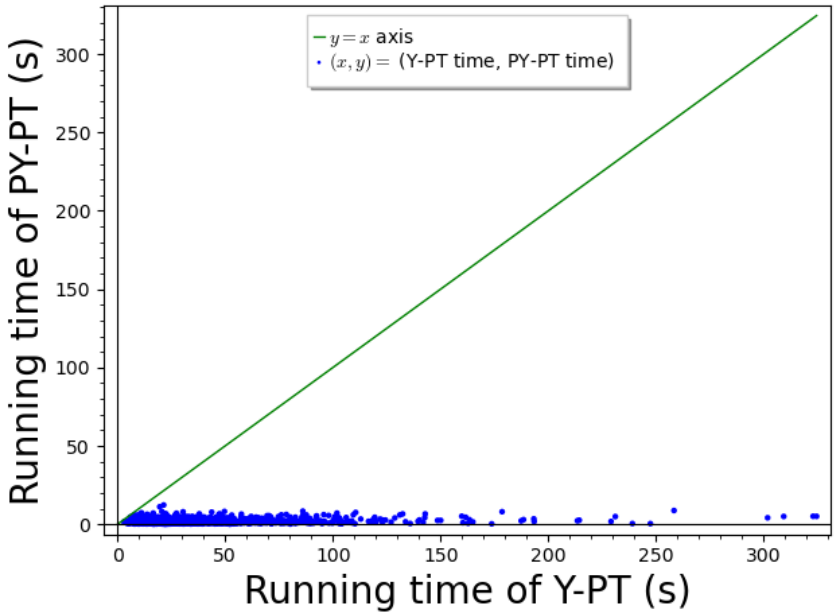


(a) Temps de réponse du YPT en fonction du k pour le réseau de trains de la Suisse

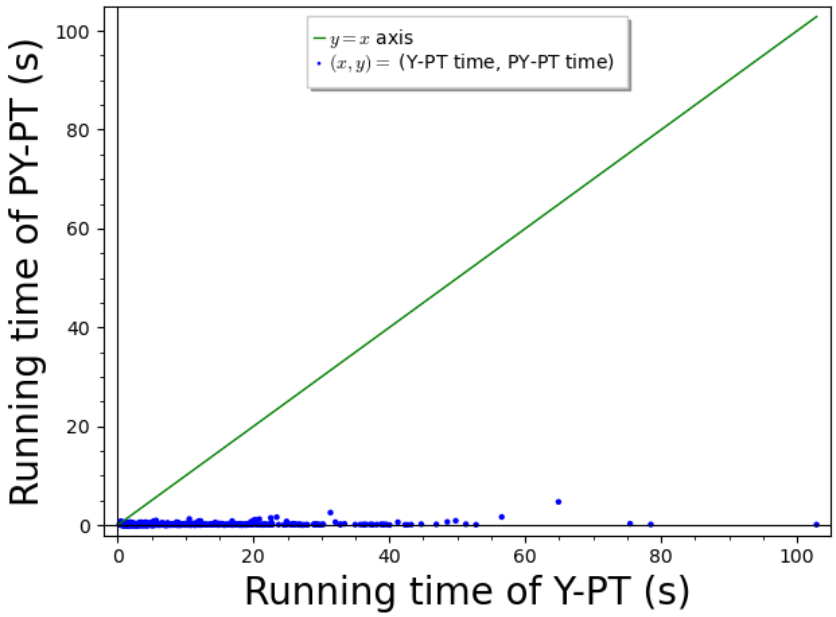


(b) Temps de réponse du YPT en fonction du k pour la ville de Stockholm

FIGURE B.5 – Temps de réponse du YPT en fonction du k pour les réseaux de Stockholm et de la Suisse.

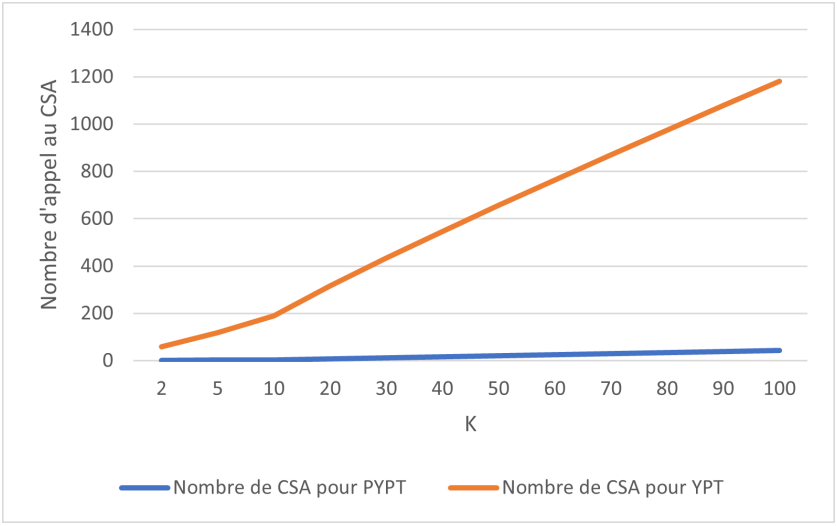


(a) Temps de réponse du YPT en fonction du temps de réponse du PYPT pour le réseau de trains de la Suisse

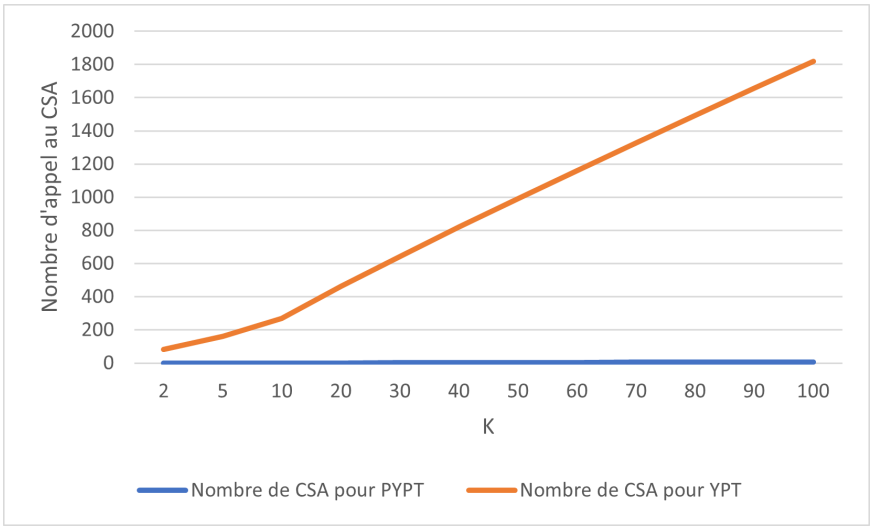


(b) Temps de réponse du YPT en fonction du temps de réponse du PYPT pour la ville de Stockholm

FIGURE B.6 – Temps de réponse du YPT en fonction du temps de réponse du PYPT pour les réseaux de Stockholm et de la Suisse.

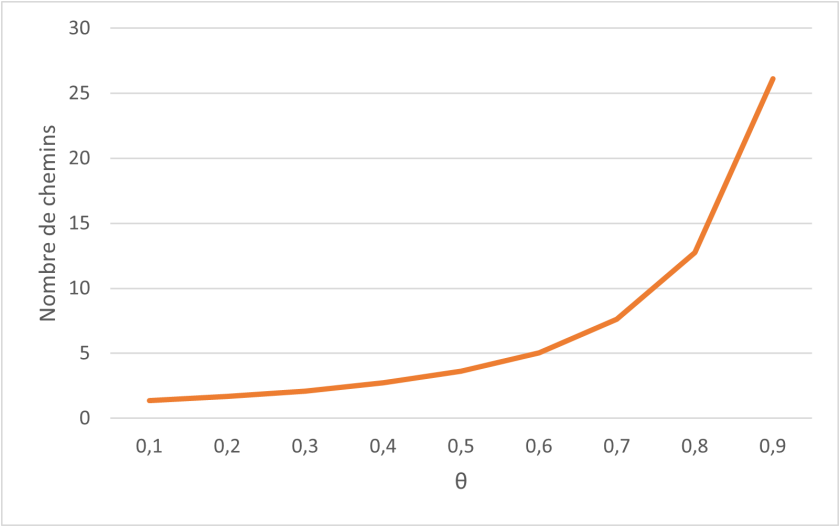


(a) Nombre d'appels au CSA en fonction du k pour le réseau de trains de la Suisse

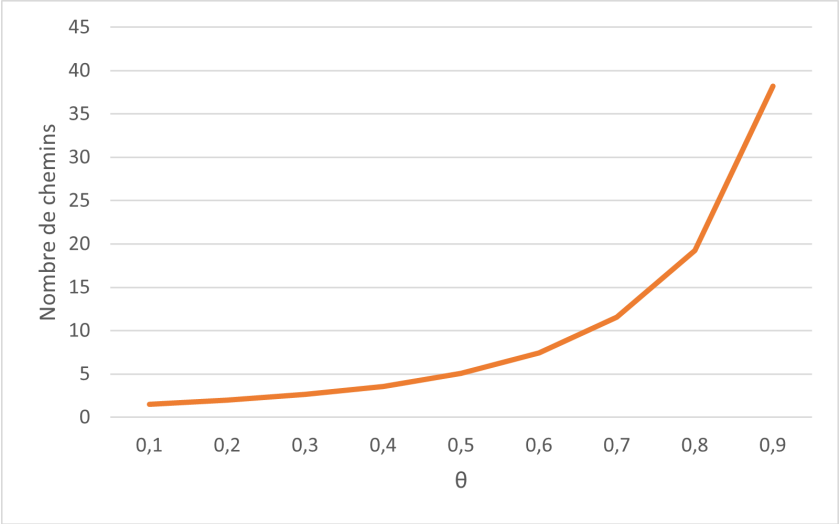


(b) Nombre d'appels au CSA en fonction du k pour la ville de Stockholm

FIGURE B.7 – Nombre d'appels au CSA en fonction du k pour les réseaux de Stockholm et de la Suisse.

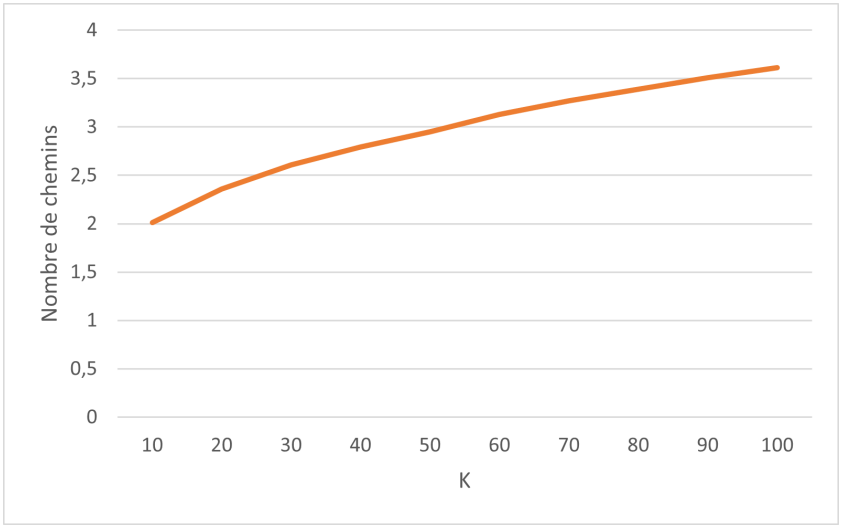


(a) Nombre de chemins dissimilaires en fonction du θ pour le réseau de trains de la Suisse

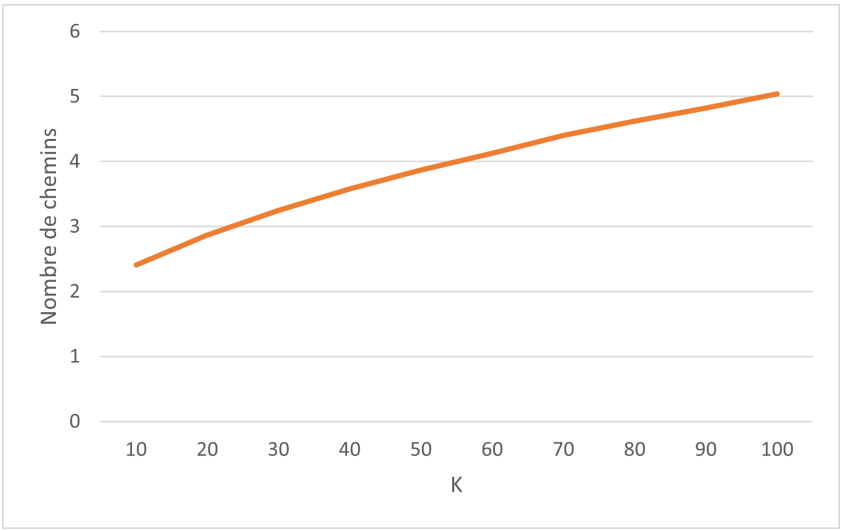


(b) Nombre de chemins dissimilaires en fonction du θ pour la ville de Stockholm

FIGURE B.8 – Nombre de chemins dissimilaires en fonction du θ pour les réseaux de Stockholm et de la Suisse.



(a) Nombre de chemins 0,5 dissimilaires en fonction du k pour le réseau de trains de la Suisse



(b) Nombre de chemins 0,5 dissimilaires en fonction du k pour la ville de Stockholm

FIGURE B.9 – Nombre de chemins 0,5 dissimilaires en fonction du k pour les réseaux de Stockholm et de la Suisse.