UNIVERSITY OF STRASBOURG

DOCTORAL SCHOOL MSII

MATHEMATIQUES, SCIENCES DE L'INFORMATION ET DE L'INGENIEUR

P H D T H E S I S

to obtain the title of

PhD of Science

of the University of Strasbourg

Specialty : Computer Science

Defended by

Amine Mohamed Falek

Efficient Route Planning for Dynamic & Multimodal Transportation Networks

prepared at ICUBE, Network Team

defended on 11th December, 2020

Jury :

Advisors :	Dr. Fabrice Théoleyre	-	CNRS, France
	Prof. Cristel Pelsser	-	Université de Strasbourg, France
Co-advisor :	Prof. Antoine Gallais	-	Université Polytechnique Hauts-de-France, France
Reviewers :	Prof. Marco Fiore	-	IMDEA Networks Institute
	Prof. Hervé Rivano	-	INSA Lyon
Examinators :	Prof. Houda Labiod	-	Telecom ParisTech
	Dr. Diego Cattaruzza	-	Ecole Centrale Lille

Résumé

Les performances du monde socio-économique sont étroitement liées à notre capacité à déplacer les personnes et les marchandises. Ainsi, la mobilité a toujours été un enjeu crucial pour bâtir des sociétés prospères. Un essor important des technologies de l'information et de la communication à permis d'adresser divers problèmes de mobilité tels que la gestion de trafic et la prévention d'accidents. La *planification d'itinéraire* désigne un calcul optimisant une ou plusieurs métriques à la fois tel que la durée du trajet, la distance parcourue, et les dépenses encourues. Sur un réseau routier *dynamique*, il est nécessaire de prendre en compte la congestion pour adapter l'itinéraire en fonction du trafic. Ceci requiert donc la collecte ainsi que le traitement des données de trafic, souvent en temps-réel.

Les données de trafic sont principalement obtenues via des dispositifs de positionnement GPS, sujets à des erreurs de localisation importantes [Von Watzdorf & Michahelles 2010]. Pour répondre à des besoins, tels que le guidage de véhicules en temps réel, divers algorithmes probabilistes permettent le traitement de *traces* GPS, émises par le véhicule en question, afin de le positionner sur une carte digitale. Toutefois, il est important de corriger toute erreur et d'écarter les mesures ambiguës. Nous nous sommes intéressés à ce problème, une étape essentielle pour la suite de nos travaux.

Le calcul d'itinéraire dynamique consiste à calculer et mettre à jour en continu le meilleur itinéraire entre un point de départ et d'arrivée fixes. Il correspond à un vaste sujet de recherche et les algorithmes proposées sont divers. Néanmoins, en termes d'évaluations, l'état de l'art contient principalement des simulations basées sur des données de trafic synthétiques, ou sinon, les études ne se focalisent que sur de petites régions géographiques. Nous avons donc exploité des données de trafic *réelles*, collectées sur une durée de plusieurs mois sur plusieurs réseaux routiers. Notre objectif étant de comprendre l'impact qu'a la congestion sur diverses stratégies de routage.

En réalité, les trajets combines souvent plusieurs modes de transports. Un réseau, dit *multimodal*, incorpore divers modes tel que les voitures, les transports publics, la marche à pied, et les vélos. La planification d'itinéraires sur des réseaux multimodaux a reçu peu de contributions en recherche. Ceci est dû notamment au fait que les contraintes utilisateur spécifiant les modes souhaités ne sont connus que lorsque sa requête est émise. De ce fait, les techniques de pré calcul appliqués aux réseaux monomodaux ne sont pas facilement adaptables. À cela s'ajoute le fait que les applications de calcul d'itinéraires sont en pratique interactive et nécessitent des temps de requêtes rapides, de l'ordre de millisecondes. De surcroît, la prise en compte de la dynamique du trafic requiert un prétraitement algorithmique facilement adaptable. Ainsi, un bon compromis entre le temps de requête et de prétraitement est aussi un facteur de performance important.

Contributions

Nos contributions concernent le traitement de données de trafic, la planification d'itinéraires dynamiques, et la planification d'itinéraires multimodaux. Nous détaillons ci-dessous nos principales contributions :

Map Matching: La généralisation globale des dispositifs de positionnement a révolutionné le domaine du transport. La géolocalisation a en effet permis une multitude d'applications telles que la détection d'incidents, le suivi de flotte et la surveillance en temps-réel du trafic routier. Néanmoins, en fonction de facteurs environnementaux, la localisation GPS est souvent imprécise [Von Watzdorf & Michahelles 2010]. Par conséquent, pour efficacement exploiter un ensemble de données GPS, il est crucial, au préalable, de corriger les données recueillies en identifiant la localisation réelle associée à chaque mesure. Ce processus est connu sous le nom de *Map Matching*.

Il existe divers algorithmes de map matching destinés à des applications variées telles que les systèmes de guidage en temps réel, l'analyse du trafic et le suivi du fret. Pour la plupart des algorithmes proposés, le but est d'identifier le chemin dans le graphe qui correspond *au mieux* à une trace de mesures GPS. Cependant, pour certaines applications, un map matching exact, plutôt que probable, est essentiel. Par exemple, l'association d'une vitesse mesurée au mauvais tronçon de route impacte négativement la précision des mesures sur le réseau routier. Cela a suscité notre intérêt pour le développement d'un algorithme de map matching *non ambigu*. Essentiellement, notre approche consiste à calculer des sous-chemins entre la première paire de mesures de la trace que nous étendons itérativement pour chaque mesure supplémentaire. Nous montrons que la contrainte temporelle utilisée pour écarter les sous-chemins "impossibles" se renforce à chaque mesure supplémentaire. Grâce à des évaluations expérimentales basées sur des traces GPS simulées et réelles, nous sommes en mesure de faire un matching exact de plus de 90% de nos traces GPS.

Stratégies de Routage pour Réseaux Routiers Dynamiques: La planification d'itinéraire dynamique consiste en général à résoudre le problème du chemin le plus court en tenant compte de la congestion du trafic en tant que phénomène évolutif. La littérature est abondante sur ce sujet en terme d'algorithmes d'évitement de la congestion. Néanmoins, les évaluations à grande échelle des stratégies de re-routage des véhicules sont rares. Plus précisément, la plupart des études reposent sur des ensembles de données synthétiques ou se concentrent sur de petites régions géographiques et négligent donc les effets de congestion d'un point de vue global. Pour combler ce manque, nous avons collecté un vaste ensemble de données de profils de vitesses de trafic réels pour différents réseaux routiers sur une période de 3 mois. Nos données ont été finement échantillonnées et nous ont permis de mener une évaluation approfondie. Nous avons comparé quatre stratégies de routage : statique, sans re-routage, re-routage continu et routage optimal. Cette dernière stratégie correspond à une borne inférieure du temps de parcours, émulant des prédictions idéales.

Elle constitue une référence de comparaison. Notre évaluation expérimentale prouve qu'un échantillonnage de 5 à 10 min est suffisant pour obtenir un routage *quasi* optimal, en utilisant la stratégie de re-routage continu. De plus, nous avons identifié des points précis, dits de divergence, indiquant les emplacements géographiques où le re-routage continu diverge souvent de l'itinéraire optimal en raison d'un réacheminement inexact. Plus important encore, nous montrons les écarts entre les données de trafic réelles et synthétiques, souvent utilisées en recherche.

Planification des Itinéraires pour Réseaux Multimodaux: De nombreux algorithmes permettent un calcul d'itinéraire en quelques microsecondes seulement, sur des réseaux à échelle continentale. La plupart des solutions ne sont toutefois adaptées, qu'aux réseaux routiers ou bien aux transport en commun [Bast 2009]. Les algorithmes de calcul d'itinéraires multimodaux sont donc nécessaires pour exploiter la diversité de l'infrastructure des réseaux de transport. Néanmoins, les solutions actuelles manquent encore en performances pour gérer efficacement des requêtes interactives dans des conditions réalistes incluant les embouteillages et les délais de transit, souvent imprévisibles. Nous avons proposé MUSE, un nouvel algorithme multimodal basé sur les séparateurs de graphes. Il partitionne le réseau en plusieurs régions indépendantes, permettant ainsi un prétraitement rapide et parallèle. La partition est indépendante des conditions de trafic de sorte que le prétraitement n'est exécuté qu'une seule fois. MUSE considère également la séquence de modes fournie par l'utilisateur pour contraindre l'itinéraire. Nous augmentons également notre algorithme avec des heuristiques pendant la phase de requête pour obtenir de meilleurs temps de calcul sans compromettre l'exactitude de l'itinéraire. Nous fournissons des résultats expérimentaux sur le réseau multimodal Français, comprenant les réseaux piétons, routiers, cyclables et divers transports en commun.

Conclusion et Travaux Futurs

Nous avons exploré au cours de cette thèse plusieurs sujets liés à la planification d'itinéraires pour les réseaux de transport dynamiques et multimodaux. Nous avons basé nos recherches autant que possible sur des mesures de trafic réels afin d'éviter les biais et les hypothèses inhérents à de nombreux ensemble de données synthétiques. Pour améliorer la précision de notre algorithme de map matching, nous prévoyons d'analyser l'impact de la topographie locale. Dans les réseaux routiers urbains à topologie dense, il peut exister plusieurs itinéraires alternatifs difficiles à distinguer de l'itinéraire réel. C'est généralement le cas pour certains réseaux tels que celui de Manhattan. Idéalement, nous cherchons à fournir une technique adaptative, capable d'ajuster la fréquence d'échantillonnage de manière dynamique en fonction d'un ensemble de métriques locales. Quant au routage dynamique, nous prévoyons de proposer une stratégie pouvant prendre en compte le gain de temps de trajet, en fonction des caractéristiques de l'itinéraire et de la zone locale. De plus, il serait possible d'adapter le temps d'échantillonnage GPS suivant la localisation du véhicule. Pour améliorer les temps de requête de MUSE, le partitionnement multiniveau pour graphes multimodaux est une prochaine étape intéressante. Ceci permet de réduire la zone de recherche sur le graphe en explorant les cellules source et cible via les niveaux élevés de la partition.

Liste de Publications

Cette thèse s'est traduite académiquement par deux conférences francophones, une conférence internationale (rang B), un article de journal (Journal of ITS), et un article soumis dans un autre journal (Transportation Science).

- "MUSE: Multimodal Separators for Efficient Route Planning." Transportation Science (soumis). Impact Factor de 4,6.
- "MUSE: une planification d'itinéraires inspirée de Séparateurs Multimodaux." ALGOTEL 2020–22èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications. 2020.
- "To Re-Route, or not to Re-Route: Impact of Real-Time Re-Routing in Urban Road Networks." Journal of Intelligent Transportation Systems: Technology, Planning, and Operations (2020). Impact Factor de 3,2.
- "De l'(in) utilité du temps-réel pour le calcul d'itinéraire dans les réseaux routiers." Algotel 2019. 2019.
- "Unambiguous, Real-Time and Accurate Map Matching for Multiple Sensing Sources." WiMob 2018-14th International Conference on Wireless and Mobile Computing, Networking and Communications. 2018.

Abstract

The performance of the socio-economic world is closely linked to our ability to move people and goods. Thus, mobility has always been a crucial requirement for building prosperous societies. A significant boom in information and communication technologies has made it possible to address various mobility problems such as traffic management and accident prevention. *Route planning* designates a system that optimizes one or a combination of metrics simultaneously, such as the duration of the travel time, the travel distance, and the incurred expenses. On a *dynamic* road network, it is necessary to take congestion into account to adapt the route according to the traffic conditions. This, therefore, requires the collection and processing of traffic data, often in real-time.

Traffic data is mainly obtained via GPS positioning devices, subject to significant localization errors. To address applications such as real-time vehicle guidance, various probabilistic algorithms allow the processing of GPS *traces*, emitted by the vehicle in question, to locate it on a digital map. However, it is often required to guarantee correctness and, therefore, correct any errors and rule out ambiguous measurements. This has sparked our interest in the map matching problem, an essential step to pursue our research on route planning.

Dynamic route calculation consists of computing and continuously updating the best route between a departure and arrival locations. It corresponds to a vast research topic, and the proposed algorithms are diverse. However, in terms of experimental evaluations, the state-of-the-art mainly consists of simulations based on synthetic traffic data or only focuses on small geographical areas. We instead focused on a data-driven approach that relies mainly on *real* traffic data, collected over several months for several road networks. Our goal is to understand the impact that congestion has on various routing strategies.

In practice, journeys often combine several modes of transportation. A *multimodal* network incorporates various modes such as cars, public transit, walking, and bicycles. Route planning on multimodal networks has received few contributions. This is due in particular to the fact that user constraints specifying the desired modes are only known during query-time. As a result, the preprocessing techniques applied to unimodal networks are not easily adaptable.

Acknowledgments

Terminus. I embarked on this incredible journey on January's 1st 2017. Back then, I had no idea what I had gotten myself into. I am astonished how time flies when I recall that this thesis represents a significant part of my life now, 14% to be precise.

Manelle, my love, I would certainly have gone mad without your infallible support. Thank you for having to put up with me throughout all those years. Having you by my side has given meaning to it all.

I had the pleasure of being surrounded by incredible people during this thesis. My dear supervisors: Fabrice Théoleyre, Cristel Pelsser, and Antoine Gallais, thank you. Your guidance and benevolence were precious. It truly is a unique experience getting my defective ideas destroyed during our meetings. I learned so much from you, I cannot thank you enough.

This CIFRE thesis would not have been possible without my supervisors at Technology & Strategy. Stéphane Klein, who recruited and supervised me early during the thesis, and Sébastien Julien, thank you for your counsel. I could always count on your support.

To my wonderful parents, Salima and Salah, look, I have made it! You instilled the love of science in me and sacrificed so much for us. I owe you my successes and always keep you in my thoughts. My sister, Lina, you have always been my best friend. You are a constant source of inspiration, even when you wake me up at 4 in the morning to admire Venus in the freezing cold weather. I extend my thanks to all my family members in France and Khenchela, who never ceased to support me.

I thank all my colleagues who lifted my spirits during the precious *coffee* moments. My friends: Ahmed Krizou, Rodrigo, Julian, Sebastian, Andréas, Jean-Romain, Renato, Loic, and Odnan, thank you for the debates, the laughs, and the chess games. I wish you all success and happiness.

Contents

Chapter 1

Introduction

1.1 Context

Transportation systems fulfill two fundamental functions that enable societies to exist: moving people and moving goods. Thus, mobility has a far-reaching impact on socio-economic welfare, and thereby, transportation efficiency has always been a crucial human endeavor to travel faster, cheaper, and safer. However, growing metropolitan areas accelerated the urbanization rate, which grew from 39% to 52% over the past 30 years and is expected to reach 66% by 2050 [Shi *et al.* 2019]. Consequently, traffic congestion became a global challenge as a natural outcome of the overload of the transport infrastructure. Traffic congestion is an emergent phenomenon resulting from unique and often unpredictable interactions between travelers. It can be classified as either recurring or non-recurring congestion. Recurring congestion corresponds mainly to rush-hour traffic, while non-recurring congestion results from a variety of factors such as accidents and incidents (bad weather conditions, road repairs, vehicle stalling), and accounts for more than 50% of the total congestion [Afrin & Yodo 2020].

Worldwide, the total number of personal and commercial vehicles in use grew from 892 million in 2005 to 1.3 billion by 2015 [OICA 2018]. In the United States, while the population size doubled over the past 70 years, the number of vehicles in operation had grown six times fold during the same period [Boundy 2019]. Increasing infrastructure capacity has been undertaken to curb congestion by widening and building new highways costing hundreds of billions annually. Nonetheless, congestion levels are still worsening. In essence, the average drivers are traveling twice as much since 1980 because of the overall distance increase between housing and other locations [TFA 2020].

Intelligent Transportation Systems (ITS) designate the application of Information and Communication Technology (ICT) to the transport sector to improve safety and mobility [Sładkowski & Pamuła 2016]. It represents an ecosystem relying on information exchange between people, vehicles, and the network, which benefits various applications such as traffic management and congestion forecast. The United States Department of Transportation (USDOT) created the ITS Joint Program Office (ITSJPO) in 1991 to research, develop, and test ITS. Currently, ITSJPO serves millions of users and achieved substantial economic benefits that are estimated to $2.3 billion annually [ITSJPO 2020]. A fundamental characteristic of ITS is monitoring the network to acquire road segment speeds, vehicle counts, and accident detection. This data is processed to subsequently serve various applications

[ITSJPO 2020], such as traffic forecasting and route planning systems, to inform users and help coordinate traffic.

Smart Roadside is a mobility system part of the ITS plan that aims at enhancing truck transport efficiency. It targets commercial routes to suppress long queues of commercial vehicles that build-up at inspection stations. Instead, it relies on real-time electronic screening technology through wireless communication between the trucks and the infrastructure. Mobility-On-Demand program is another ITS project that focuses on the need for alternative forms of transportation. The goal is to develop flexible transportation systems that incorporate shared-use and multimodal integration. The New York City Connected Vehicle Pilot program was initiated in 2015 as a measure to reduce traffic incidents in the Manhattan area where 73% of all fatalities involved pedestrians compared to 14% nationwide. It implements Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) safety applications to assist the driver in several scenarios such as alert in case of an imminent crash, detect blind-spot adjacent lane vehicles, assess if crossing an intersection is safe, and alert upon reaching a designated work or school area with high speed. ITS is, however, a broad topic comprising a variety of open problems: *how to faithfully model traffic conditions through large volumes of acquired traffic data? how to efficiently exploit such data to avoid congestion? how to incorporate user constraints to achieve route planning across several modes of transportation?*

These are the questions we tackle in this thesis. As entitled, our focus lies on efficient route planning strategies for realistic use-case scenarios that consider traffic congestion and exploit transportation networks' diversity. Nonetheless, route planning is also contingent on accurate traffic information. Therefore, we also address the Map Matching problem [Chao *et al.* 2020]: processing raw GPS measurements to accurately aggregate the paths of vehicles in the network onto a common digital map. Our map matching approach is particularly useful for research applications as it guarantees correctness, and therefore, provides a strong foundation for the subsequent route planning problems we addressed that required experimental evaluation based on real traffic data.

The term *route planning* refers to identifying the most cost-effective route in the network where *cost* is either a unique metric or a combination of several metrics such as travel time, distance, number of transfers, and monetary expense. Historically, Dijkstra's algorithm [Dijkstra 1959], designed to compute shortest-paths on graph structures, has become a fundamental building block to modern route planning techniques. The movement of public access to transportation data over the past 20 years has further catalyzed the field, especially for routing in road networks [Sanders & Schultes 2007]. Consequently, current algorithms are operated on continental-scale networks, computing a route in mere microseconds. Applications such as Google Maps, TomTom, HERE, PTV, and countless other industries serve millions of users daily, a testament to remarkable progress.

However, the vast majority of contributions to the field of route planning are disproportionately tailored to road networks compared to public transit networks. Although road network route planning techniques can often be adapted and applied to public transit networks, their performance drops significantly because of the structural differences between these distinct networks [Bast 2009]. For instance, hierarchy is a strong feature observed in road networks and is a consequence of the infrastructure's intentional design. In contrast, public transit networks are inherently time-dependent and operate according to a predefined schedule. They do not present any notable hierarchy, and therefore, hierarchical speedup techniques are significantly less effective when applied to public transit networks.

Nonetheless, in practice, trips are multimodal: they consist of a combination of walking, riding a bicycle, driving a car, and using public transit. According to the U.S Department of Transportation [DOT 2015], 40% of all traveled passenger-kilometers consist of commuting. While commuting distances are increasing, average speeds are, however, decreasing due to steadily rising congestion. By 2015, the average commuter typically spent 40 hours every year stuck in traffic, costing $121 billion annually. For decades, personal-vehicle travel has been, and remains, the dominating trend: 105 million American commuters rely mostly on driving while the remaining 32 million depend on all other transportation modes. Nevertheless, public transit, although accounting for only 5% of the overall commute trips, is vital to alleviating congestion. The same report indicates that if public transit users in major metropolitan areas of the united states suddenly reverted to driving, congestion is estimated to increase by 24% and cost an additional $17 billion annually. Fortunately, over the past two decades, public transportation thrived with a registered 25% increase in public transit ridership. Enhanced access to information significantly improved transit networks, and thus, steered urban populations to consider better alternatives to their private cars.

A personal use-case scenario depicted my multimodal journey upon arriving in France a few years ago. Mainly, I had to plan every step of my trip carefully: I first booked my Algiers-Paris flight ticket and matched the correct Paris-Strasbourg train with enough transfer time in case of any delays. Considering the heavy traffic congestion during the first part of my Boumerdes-Algiers trip, I drove to the airport earlier than usual to not miss my flight. To achieve such planning, one must refer to several applications, each dedicated to a unique transportation network, and manually construct the trip. Because of the computational complexity that arises from combining several transportation networks, the field of multimodal shortest path algorithms has been neglected for decades. In recent years, few contributions were put forth but often only evaluated on small scale networks.

1.2 Contribution

The need for more efficient and practical multimodal algorithms has initially motivated this work both in research and industry (CIFRE thesis). Nonetheless, route planning is a broad subject, and over the course of the thesis, we identified specific topics to contribute to the literature of route planning in transportation networks. Specifically, this thesis took part in a time interval during which access to transport data has become less secretive and, therefore, providing an opportunity to conduct data-driven research. Consequently, our main contributions are:

- a map matching algorithm that can be used to construct an accurate model of traffic conditions based on GPS traces;
- an extensive evaluation of the dynamic routing problem based on real traffic data;
- a multimodal route planning algorithm called MUSE that relies on graph separators to achieve fast preprocessing times and therefore handle unpredictable traffic congestion and transit delays.

We further detail each contribution in the following sections.

1.2.1 Map Matching:

The global spread of positioning devices has revolutionized the field of ITS. Geolocation has indeed enabled a multitude of applications such as incident detection, fleet monitoring, and real-time road traffic surveillance. GPS measurements are, however, not accurate and follow a Gaussian error distribution model. The problem of *map matching* is, therefore, the task of associating each GPS-trace to a correct sequence of road segments. Nonetheless, most of the literature's map matching algorithms are designed to compute the path that matches *best* a given GPS trace. However, for some applications, exact map matching is required. This sparked our interest in developing a new map matching algorithm.

We follow a topological approach that relies on the timestamps of the GPS-trace to prune inconsistent matchings. Mainly, for each GPS measurement, we compute a set of edge candidates in its vicinity. There is a single edge among such candidates, denoting the actual road segment the measurement should be matched to. Using Breadth-First-Search, we iteratively construct subroutes that fit the time measurement constraints of the GPS-trace. Lastly, we discard all ambiguous paths denoting two or more valid matchings for a given measurement. We conduct an experimental evaluation on both a simulated dataset (synthetic measurements) and real GPS measurements and successfully match 90% of the traces.

1.2.2 Dynamic re-Routing:

Route planning represents a major challenge with a substantial impact on safety, economy, and even climate. An ever-growing urban population caused a significant increase in commuting times, therefore, stressing the prominence of efficient real-time route planning. In essence, the goal is to compute the fastest route to reach the target location in a realistic environment where traffic conditions are time-evolving. Consequently, a large volume of traffic data is potentially required, and the route is continuously updated. We base our study on a real dataset, comprising the travel times of the road segments of New York, London, and Chicago, collected over three months. We implement an optimal algorithm that assesses future traffic conditions with regard to departure time to compute the lower bound of travel time. It depicts an *ideal* routing algorithm with exact predictions and serves as a reference to compare a static, no re-routing, and continuous re-routing strategies based on their achieved travel time gains. We show that a continuous re-routing strategy with a low sampling rate (5 to 10 min intervals) provides enough accuracy to compute the best route while reducing the computational cost. Furthermore, we identify a small number of crossroads where re-routing often occurs but leads to sub-optimal routes. Ultimately, we determine *when*, *how often*, and *where* is re-routing worthwhile.

1.2.3 Multimodal Route Planning:

Many algorithms compute shortest-path queries in mere microseconds on continental-scale networks. Most solutions are, however, tailored to either road or public transit networks in isolation. To fully exploit the transportation infrastructure, multimodal algorithms are sought to compute shortest-paths combining various modes of transportation. Nonetheless, current solutions still lack performance to efficiently handle interactive queries under realistic network conditions where traffic jams, public transit cancelations, or delays often occur. We present MUSE, a new multimodal algorithm based on graph separators. It partitions the network into independent, smaller regions, enabling a fast and scalable preprocessing. The partition is common to all modes and independent of traffic conditions so that the preprocessing is only executed once. MUSE also considers the sequence of modes provided by the user to constrain the shortest path during the online phase accordingly. We also augment our algorithm with heuristics during the query phase to achieve further speedups without trading-off correctness. We provide experimental results on France's multimodal network containing the pedestrian, road, bicycle, and public transit networks.

1.3 Overview

The thesis is organized into six chapters. Initially, in chapter 2, we provide the necessary background and related work to support the subsequent chapters. We begin by presenting the fundamental graph theory concepts and the recurring notation throughout the thesis. We discuss techniques to model time-independent, time-dependent, and multimodal transportation networks with varying degrees of realism. We then focus on the Shortest Path Problem (SPP), a pivotal notion in this thesis, and describe several strategies and speedup techniques that address it. Then, we review the relevant associated literature. We begin by introducing state-of-the-art map matching algorithms. We grouped a multitude of techniques into three categories: geometrical, topological, and statistical map matching. After that, we focus on the dynamic route planning problem. We review several re-routing strategies and the few studies aimed at evaluating the efficiency of each technique. In the last section, we review multimodal shortest path algorithms.

In chapter 3, we present our unambiguous map matching algorithm. We describe our model and the four stages of the algorithm, mainly: preprocessing the raw data, selecting edge candidates for each GPS measurement, computing valid subroutes for the trace, and finally computing the matching path. We provide the results of an experimental evaluation on simulated and real GPS measurements.

In chapter 4, we discuss the dynamic routing problem. We compare four routing strategies using extensive traffic datasets for several road networks. Throughout the experimental evaluation, we systematically compare the results obtained from real measurements to a simulated dataset for the same road network.

In chapter 5, we explain our approach to solving the multimodal route planning problem. We explain how we assemble a multimodal network as a layered graph of unimodal networks. We describe the three stages of our algorithm, MUSE, which consist of graph partitioning, a preprocessing to compute an overlay, and a query phase. We conduct an experimental evaluation on a country scale network and discuss our results.

Finally, in chapter 6, we summarize our results and discuss possible future work, either extending our current contributions or relying on them to address other interesting problems. We also provide links to our repositories containing useful tools developed throughout the thesis to run new experiments or replicate some of our results.

bonne lecture.

Chapter 2

Related Work

Contents

We review in this chapter all relevant literature to clearly put in context our contributions in the upcoming chapters. We initially provide the necessary background covering the fundamental aspects of route planning. We review in section 2.1 relevant graph theory definitions as well as the recurring terms and notation throughout the remaining chapters. In section 2.2, we detail the different techniques to model transportation networks. We present different versions of the time-independent model, which is often used to represent road networks, and the time-dependent model, specifically tailored to public transit networks. We further explain how to construct a multimodal network as a layered graph structure composed of several unimodal networks. In section 2.4, we review the Shortest Path Problem (SPP) and detail a variety of algorithms and speedup techniques to solve it. In section 2.3, we explain the map matching problem and its applications. We classify map matching algorithms into three categories: similarity-based, statistical models, and machine learning techniques. In section 2.5, we review dynamic routing strategies for road networks. Mainly, our goal is to present studies that assess the performance of such strategies either based on simulated or real traffic data. In section 2.6, we detail the challenges of multimodal route planning and describe the proposed state-of-the-art solutions. Finally, section 2.7 concludes the chapter.

2.1 Graph Theory Concepts

A significant number of contributions to route planning rely on graph theory to model transportation networks [Dijkstra 1959, Thomson & Richardson 1995, Cherkassky *et al.* 1996]. Mainly because graphs are simple and intuitive structures to abstract various types of networks. Nevertheless, mostly, graph theory offers a large panoply of theorems and algorithms that can be leveraged to devise efficient route planning solutions.

A Graph $G(V, E)$ consists of a set of vertices $v \in V$, and a set of edges $(v, w) \in E$, such that each edge connects two vertices $v, w \in V$. An *undirected graph* is a graph whose edges consists of a pair of unordered vertices, that is, edges (v, w) and (w, v) are the same. In contrast, a *directed* graph, or digraph, has directed edges. An edge (v, w) is an ordered pair of vertices v, w referred to as the *tail* and *head*, respectively. Two vertices v, w are said to be *adjacent* if they form an edge (v, w) and the *degree* of a vertex v is the number of its adjacent vertices. In a digraph, we distinguish the in-degree and out-degree of v, denoting the number of incoming (v is the head) and outgoing (v is the tail) edges. A *simple* graph is a graph with no parallel edges or loops i.e., edges of the form (v, v). Throughout the remainder of the dissertation, all graphs are considered directed and not simple, unless otherwise specified.

The Edge Cost The set of edges in a graph are labeled with *weights* $c(v, w, \tau)$ denoting the cost associated to (v, w) at time τ. The edge cost $c(v, w, \tau)$ is modeled using a positive piece-wise linear function $f_{vw} : \Pi \to \mathbb{R}^+$ mapping the cost for all times $\tau \in \Pi$. If f_{vw} is constant, then (v, w) is a *time-independent* edge, otherwise, it is *time-dependent*.

A Path $P = \{v_0, v_1, .., v_k\}$, also written $P_{v_0 v_k}$, is an ordered sequence of vertices. In a time-independent graph, its cost $c(P) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$ is evaluated as the sum of the costs of the individual edges that make up the path. If the graph is time-dependent, the cost is recursively evaluated based on the departure time τ at vertex v_0, and is given by $c(P, \tau) = c(v_0, v_1, \tau) + c\big(\{v_1, .., v_k\}, \tau + c(v_0, v_1, \tau)\big)$. Let $\mathcal{P}$ be the set of all P_{st} paths with a departure time τ. $P^*_{st} \subseteq \mathcal{P}$ denotes the paths of smallest cost, that is, $c(P^*_{st}, \tau) \leq c(P_{st}, \tau) \forall P_{st} \in \mathcal{P}$. We call P_{st}, a *shortest path* and use the notation $d(s, t, \tau)$ to denote its cost.

2.2 Modeling Transportation Networks

Graphs are useful to encode the structural and topological properties of transportation networks. The set of vertices V abstracts physical entities such as road junctions or public transit stations. The set of edges E represents road segments or public transit routes such as bus or train lines. In that context, the cost associated with each edge often denotes travel-time or other metrics such as distance or fare expenses. Figure 2.1 illustrates the graph denoting the road network of the Illkirch region in Strasbourg, France.

We distinguish two categories of transportation networks: *time-independent networks* such as road and pedestrian networks and *time-dependent networks* consisting of all types of public transit networks, which are schedule-based in nature. Although the road network is subject to traffic congestion, which impacts edge costs, the road network can still be modeled using time-independent graphs, albeit inaccurately. In contrast, public transit networks are inherently time-dependent: one cannot traverse an edge except at specific discrete times denoting shuttle departures. We review in the following sections 2.2.2 and 2.2.3 the details of transportation networks and their graph-based models ranging from the most basic to more realistic.

2.2.1 The FIFO property

Route planning in transportation networks often consists of computing the shortest path between two given locations [Bast *et al.* 2016b]. In time-dependent networks, it is known as the Time-Dependent Shortest Path (TDSP) problem. The algorithmic complexity of TDSP depends on the cost functions used to model the time-dependency of the network. Kaufman and Smith [Kaufman & Smith 1993] investigated this problem and showed that the network must be a *FIFO*-network to solve TDSP in polynomial time. The FIFO property, also known as the non-overtaking property, states that for any edge (v, w) in the network, a vehicle A departing from v toward w at any time τ_1 cannot be overtaken by another vehicle B departing from v at a later time $\tau_2 \geq \tau_1$. Formally, for all times $\tau_1, \tau_2 \in \Pi$ such that $\tau_1 \leq \tau_2$ then $\tau_1 + f_{vw}(\tau_1) \leq \tau_2 + f_{vw}(\tau_2)$. Hence, waiting at a vertex never pays-off.

In practice, transportation networks are not necessarily FIFO-networks, but the underlying graph can be adapted to adhere to the FIFO property without loss of accuracy. For instance, a separate parallel edge can be inserted into the graph to account for a high-speed train that may be scheduled to overtake another train. Therefore, we can search for edges that exhibit non-FIFO behavior and correct the graph accordingly.

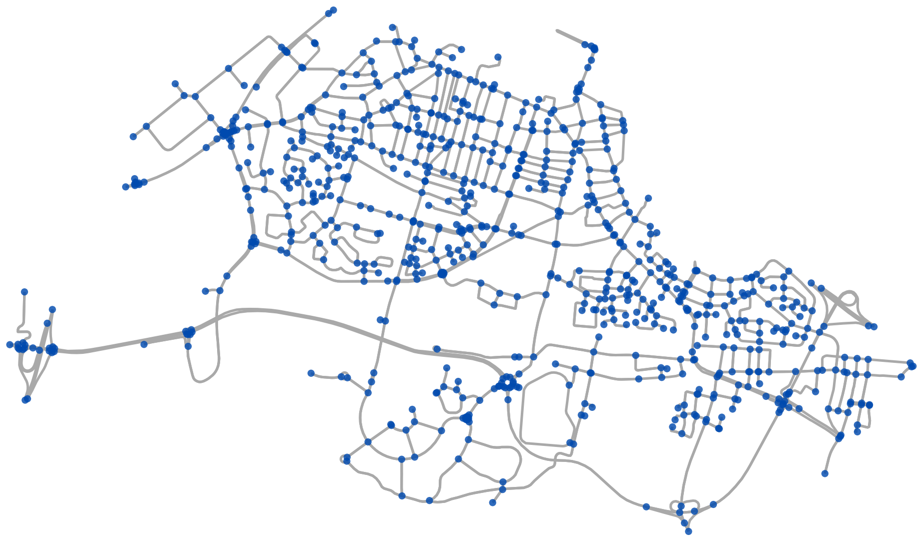


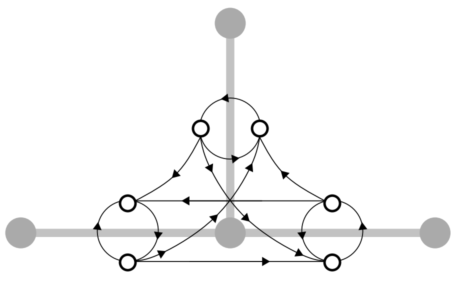
Figure 2.1: Graph representation of Illkirch's road network in Strasbourg, France. The dots are the graph's vertices and the segments are the edges, drawn to preserve the original shape of the road segments.

2.2.2 Time-Independent Models

A directed graph $G(V, E)$ with constant edge weights is often used as a basic structure to model time-independent networks [Sanders & Schultes 2007]. Each edge $(v, w) \in E$ represents a segment connecting vertices v and w and allowing traffic to flow from v towards w.

More realistic approaches [Geisberger & Vetter 2011] tend to incorporate *turn penalties* to account for the additional time required to perform a turn based on the speed and turn angle. Furthermore, *turn restrictions* must be enforced to abide by traffic laws. The *pseudo-dual* graph [Winter 2002] is derived by transforming the edges in G into vertices, and then, the edge-set is built by adding an edge for each valid turn. However, as depicted in figure 2.2, this approach results in a much larger graph as a single edge in the original graph, embeds several turns. Another, more compact approach [Delling *et al.* 2011a], consists in associating a *turn table* to each vertex $v \in V$. The turntable stores the permutations of all v's incoming and outgoing edges and is used to check for valid turns. The advantage of this approach is that turn restrictions are, most of the time, the same across similar junctions. Hence, a single turntable can be shared by a subset of vertices.

The time-independent graph is often used to model the road, cycling, and pedestrian networks. To account for traffic fluctuations on the road network, edge weights, instead of scalars, are augmented into time-dependent cost functions without any additional alteration to the time-independent graph structure [Zhao *et al.* 2008].



(a) Without turn restrictions (b) Enforcing turn restrictions

Figure 2.2: Pseudo-dual graph derived from the original graph (in bold) denoting all possible turns (left) and its reduced version containing only valid turns (right).

2.2.3 Time-Dependent Models

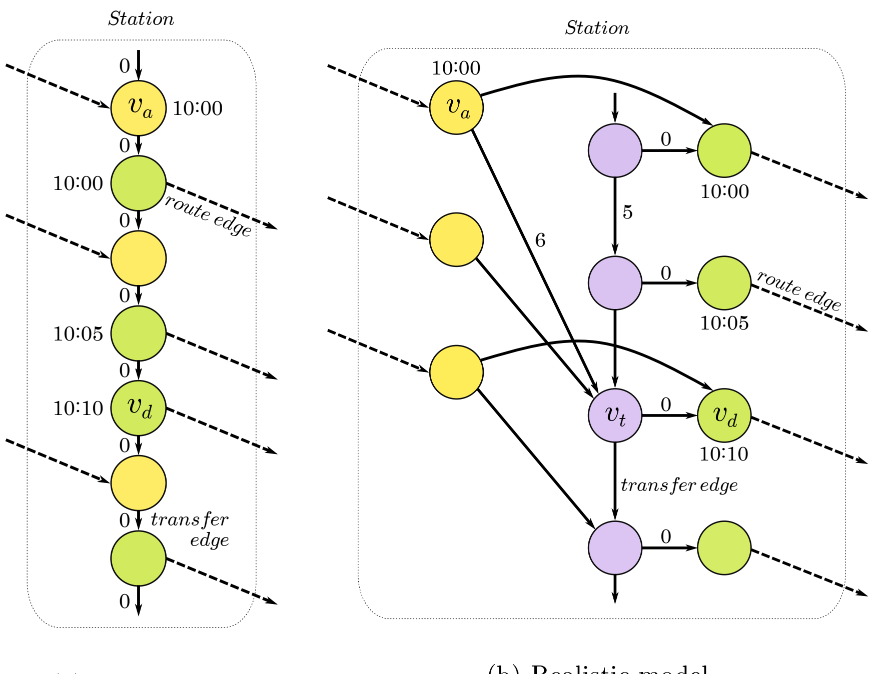
Time-dependent networks are modeled according to timetable information [Müller-Hannemann *et al.* 2007]. The timetable is periodic and consists of a set of shuttles $\mathcal{Z}$, a set of stations $\mathcal{B}$, and *elementary connections* $\mathcal{C}$. Given two stations S and S', an elementary connection is a 5-tuple $c = \{Z, S, S', t_d, t_a\}$ and defines a shuttle Z departing from S at time t_d and arriving at S' at time t_a without a stop at any intermediate station. A ***trip*** is an ordered set of elementary connections and defines the route of a unique shuttle Z. Typical graph representations for such networks are the *time-expanded* and *time-dependent* graph models.

The *time-expanded* model is illustrated in figure 2.3 and built as a directed graph with each vertex denoting a specific *event* occurring at a particular time. In the basic version of the model (figure 2.3a), an event is either a *departure* or an *arrival*. To construct the graph, we iterate over all elementary connections $\{Z, S, S', t_d, t_a\} \in \mathcal{C}$ and add a departure vertex v assigned to station S with timestamp t_d and an arrival vertex w assigned to station S' with timestamp t_a. Each station S is subsequently formed by a set of departure and arrival vertices that are chronologically sorted, as shown in figure 2.3a. All vertices within the station are connected using internal edges with a Zero-weight, and the last vertex is connected to the first to preserve connectivity.

Route edges model connections, their weight denotes the travel-time between two consecutive stations. Given a route edge (v, w), its weight is given by the function in equation (2.1), where Π is the period of the timetable:

$$f_{vw}(t_d, t_a) = \begin{cases} t_a - t_d & \text{if } t_a \geq t_d, \\ \Pi + t_a - t_d & \text{otherwise} \end{cases} \tag{2.1}$$

Although the basic time-expanded model is useful in describing the structural aspects of the network, it assumes, however, that all transfers are feasible. As illustrated in figure 2.3a, upon arriving at vertex v_a at 10h00, the basic model allows for



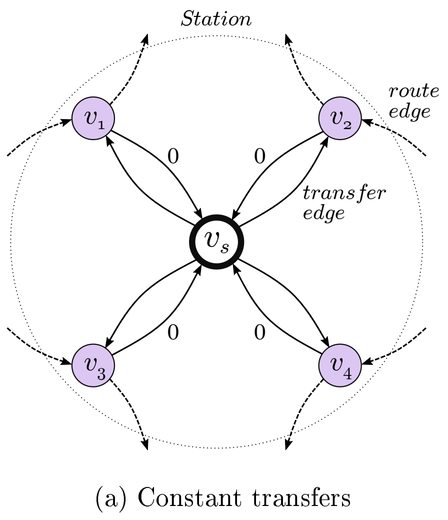
(a) Basic model (b) Realistic model

Figure 2.3: Time-expanded model of a public transit station. Vertices colored in yellow, green, and purple denote arrivals, departures, and transfers, respectively.

catching the departure scheduled at 10h05 even though 5 minutes are not enough to achieve the transfer.

For that matter, the realistic model [Pyrga *et al.* 2004, Pyrga *et al.* 2008] incorporates additional, *transfer* vertices, to account for transfer times. The realistic model is shown in figure 2.3b. Each departure vertex v_d is paired to a transfer vertex v_t such that $c(v_t, v_d) = 0$. Furthermore, each arrival vertex v_a is either directly connected to a departure vertex denoting no transfer with no additional cost $c(v_a, v_d) = 0$, otherwise, it is connected to the first transfer vertex v_t such that the transfer is feasible and its cost is given by $c(v_a, v_t)$. In figure 2.3b, upon arriving at vertex v_a at 10:00, the realistic model restricts the transfers by allowing only those that are feasible: the earliest transfer (v_a, v_t) costs 6 minutes and allows to catch the departure from v_d, scheduled at 10:10.

The *time-dependent model* is a *compact* representation of the transit network [Brodal & Jacob 2004]. A station $S \in \mathcal{B}$ is abstracted with a single station vertex $s \in V$, and a set of route vertices are added to the station to account for the various routes transiting through S. Hence, edges are considered as either *transfer edges* or *route edges*. A transfer edge connects two vertices belonging to the same station, and its associated weight denotes the required transfer time. A route edge connects two route vertices belonging to two distinct stations. Figure 2.4a illustrates the described model. As depicted, transfer costs are attached to outgoing edges from the station



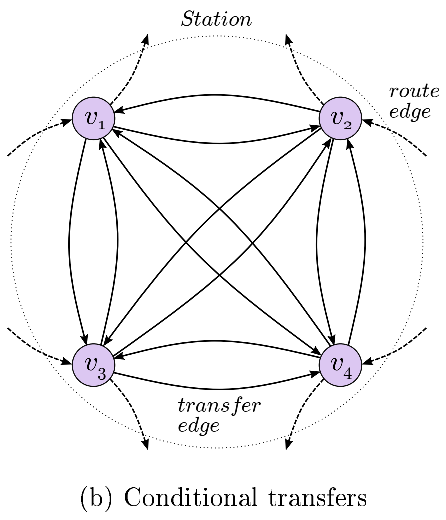


Figure 2.4: Time-dependent graph representation of a public transit station using either constant (left) or conditional (right) transfer times.

vertex, while incoming transfer edges have a zero cost and preserve connectivity. This depiction assumes constant transfers throughout the whole station, which is not necessarily realistic. Indeed, in a train station, for instance, platforms that are distant from each other incur additional time to achieve the transfer. The more realistic model shown in figure 2.4b employs conditional transfers but also requires additional edges. The weight of a route edge (v, w) is modeled with a periodic piecewise linear function $f_{vw} : \Pi \to \mathbb{R}^+$ with a slope $df_{vw}/dt = -1$ for all the segments of the function, as shown in figure 2.5. At any given time $t \in \Pi$, $f_{vw}(t)$ maps the associated travel time, taking into account the necessary waiting time to catch the earliest departure. Footpaths are also added via *foot edges* to model transfers between nearby stations [Pyrga *et al.* 2008].

2.2.4 Multimodal Network Model

Routing, in practice, often involves combining several means of transportation such as walking, driving, riding a train, flying, and cycling. Networks that enable such combinations are modeled using *multimodal* graphs. A multimodal graph embeds a set of unimodal graphs, one for each transportation mode. Figure 2.6 illustrates this layered structure containing, for instance, the road, foot, cycling, and public transit networks. The multimodal graph is, in fact, a directed labeled graph denoted $G(V, E, \Sigma)$. Every edge $(v, w) \in E$ holds a unique label $\sigma \in \Sigma$ denoting the transportation mode it carries. We denote by $G_\sigma(V_\sigma, E_\sigma)$ the unimodal graph labeled $\sigma \in \Sigma$. The vertex-set of the multi-modal graph G is given by $V = \cup_{\sigma \in \Sigma} V_\sigma$ and its edge-set $E = \cup_{\sigma \in \Sigma} E_\sigma \cup E^{\text{link}}$ where E^{link} denotes a set of *link* edges allowing to transfer from on transportation mode to another.

Intuitively, any transition from one network to another should be mediated

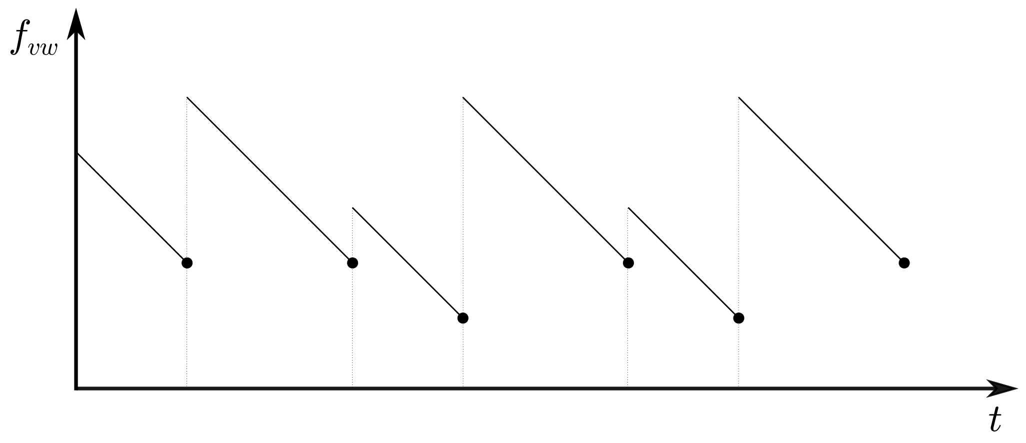


Figure 2.5: Piecewise linear function representing the travel time associated to a route edge (v, w) in a time-dependent graph.

via the foot network, as *some* walking is usually required. We find in the literature, however, multimodal graph structures that do not satisfy this condition [Delling *et al.* 2009, Dibbelt *et al.* 2015]. In that case, the authors insert link edges directly between the road and the public transit network. In the remainder of the dissertation, we stick to the former definition to explicitly account for walking during transfers. Nonetheless, transitions to and from the foot network are only allowed at specific locations, and thus, we select a subset of *link* vertices $V_\sigma^{\text{link}} \subseteq V_\sigma$ from each graph G_σ:

Foot $\leftrightarrow$ Road: The road network is accessible everywhere a car is allowed to park. Furthermore, rental vehicles are accessible at rental stations and thus all parking spots and rental stations are marked as link vertices ${V_c}^{\text{link}} = {V_c}^{\text{park}} \cup {V_c}^{\text{rent}}$.

Foot $\leftrightarrow$ Bicycle: Considering that bicycles can be used almost everywhere walking is possible (except on stairways, for instance), every vertex $v \in V_b$ is a link vertex from which we can access the foot network. Hence, ${V_b}^{\text{link}} = V_b \cup {V_b}^{\text{rent}}$ which includes rental stations as well.

Foot $\leftrightarrow$ Public Transit: Public transit stations are accessible via station vertices. Hence, $\forall v_S \in V_p$ where v_S is a station vertex, we label it as a link vertex and add it to ${V_p}^{\text{link}}$.

Then, for each vertex $v \in V_{\sigma \in \Sigma}^{\text{link}}$, we must compute the closest vertex $w \in V_f$ in the foot network and add the link edges $(v, w), (w, v) \in E^{\text{link}}$. Additional labels are added to Σ as we label link edges according to the type of transfer:

- link edges $(v, w) | v \in V_f$ and $w \in {V_c}^{\text{rent}}$ denote transfers to car-rental stations are labeled with c_r.
- link edges (v, w), $(w, v) | v \in V_f$ and $w \in {V_b}^{\text{rent}}$ are labeled with $lab(v, w) = b_r$ and $lab(w, v) = b_s$ which imply *renting* and *restoring* the bicycle respectively.

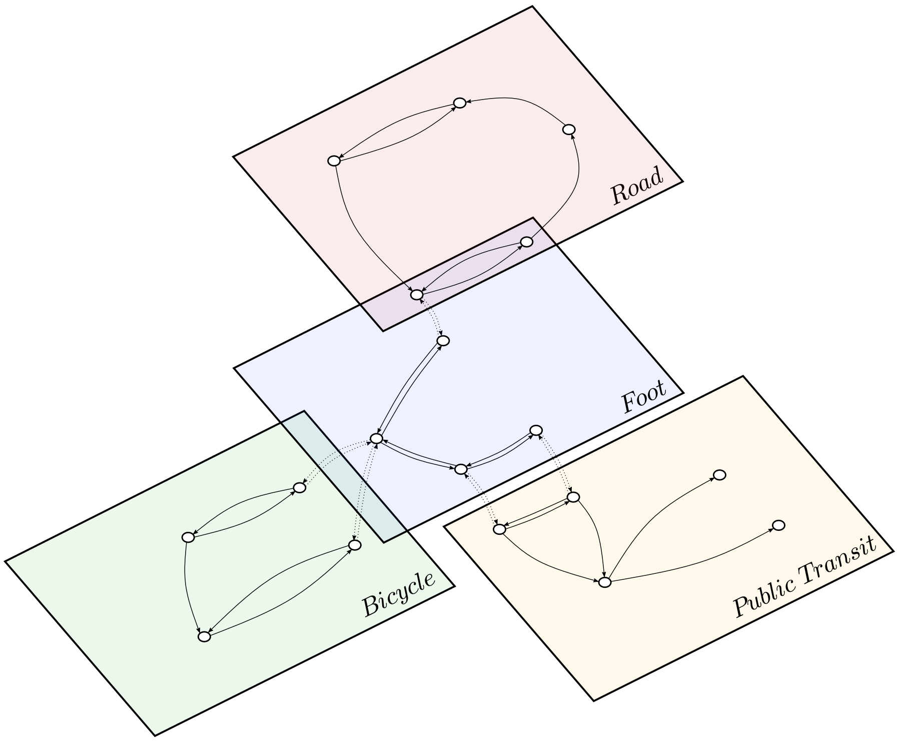


Figure 2.6: The multimodal graph consists of individual unimodal networks that are combined together using link edges (dashed arrows) transiting through the foot network.

- the remaining link edges (v, w) are labeled $lab(v, w) = t$ denoting a regular modal transfer.

Relying on a brute force approach to compute link edges is costly: we have to scan the whole foot network to identify the closest foot-vertex for each vertex in the bicycle network, leading to a quadratic complexity of $\mathcal{O}(V_b \times V_f)$. A better approach consists of clustering the foot vertices using a 2d-tree [Bhatia & Others 2010] based on latitude and longitude, which is a suitable data structure for solving the *nearest neighbor* problem in logarithmic time.

The cost of a link edge $c(v, w)$ is fixed and depends on the type of transfer. It includes the required walking-time to transfer to or from the foot network and an additional cost to consider either parking-time, processing at a rental station, or for instance, the time it takes to secure a bicycle. Nonetheless, we must also ensure path *feasibility*. That is, if the private car (bicycle) is left behind at some point during the trip in favor of using the bus, we would not be able to use our private car (bicycle) again. Similarly, a scenario in which the private car is used after taking a train is not valid. Nevertheless, the road (bicycle) network remains accessible via other means such as a taxi or from a rental station. Such constraints are not embedded within the graph but, rather, must be dealt with separately.

2.3 Map Matching

The global widespread of positioning devices revolutionized transportation systems by enabling a plethora of applications [Chao *et al.* 2020], such as incident detection, fleet tracking, and traffic sensing. Nonetheless, depending on environmental factors, GPS localization is not accurate [Von Watzdorf & Michahelles 2010]. For instance, GPS-enabled mobile phone accuracy drops from a few meters to hundreds of meters depending on the number of visible satellites and the surrounding landscape. Consequently, in order to benefit from the large data sets of raw GPS information, we must accurately map each measurement to a correct associated geographical location. This process is known as *Map Matching*.

In the context of route planning applications, map matching is often used in a preprocessing phase during which individual GPS *traces* are matched onto a common digital map. A trace defines a sequence of measurements sampled from the trajectory of a unique vehicle. The sampling frequency ranges from a few seconds to several minutes as a tradeoff of memory requirements and targeted matching accuracy. Map matching is also extensively used for crowdsensing applications in smart cities. For instance, Li et al. [Li *et al.* 2017a] infer traffic conditions from map matching on a large dataset of GPS traces. Lehmann et al. [Lehmann & Gross 2016] exploit crowd-sensing to predict traffic, and thus the pollution peaks. Map matching can also serve in merging different data sources: Aly et al. [Aly *et al.* 2017] proposed to use individual traces to enrich a base map with semantic information.

2.3.1 Similarity-based Map Matching

Early map matching techniques [Bernstein *et al.* 1996, White *et al.* 2000, Greenfeld 2002] rely on the geometric similarities of the GPS-trace and the underlying graph structure to identify the road segment every GPS measurement should be matched to. Often referred to as *point-to-point* or *point-to-curve* map matching, the intuition is to match each measurement to the closest vertex, or alternatively edge, in the graph. These techniques are, however, not robust in dense graph regions with close parallel road segments.

In contrast, topological-based algorithms [Yin & Wolfson 2004, Yu *et al.* 2010] rely on the structural configuration of the trace with regard to the graph: in addition to geometrical similarities, the road segment's connectivity and contiguity are also considered. Other topological approaches [Quddus *et al.* 2003] additionally assess the speed and heading of the vehicle, inferred from the timestamps of the GPS measurements, to prune unfeasible routes. Zhu et al. [Zhu *et al.* 2017] compute the shortest path between the trace's endpoints and a similarity score of the trace and the obtained shortest path. If the score is lower than a predefined threshold, the trace is subdivided into several smaller GPS sequence measurements, and the same process is repeated for each sequence. The overall matching is then obtained by concatenating the shortest paths that maximize the similarity score of the trace.

2.3.2 Probabilistic-based Map Matching

GPS localization is subject to erroneous measurements, for instance, due to coverage and weather conditions. Therefore, each GPS measurement is projected onto neighboring road segments denoting the geographical locations that the measurement can potentially be matched to. Each projection is referred to as a *state* and is marked with the measurement's timestamp. The emission probability of a given state represents its likelihood of being a correct match. It follows a gaussian distribution centered around the measurement, and therefore, states that are closer to the measurement receive a higher probability. Furthermore, considering two successive measurements, we, therefore, can compute several paths between each pair of their respective states. A path represents a *transition* and is assigned a transition probability. The Hidden Markov Model (HMM) constructs a temporal graph whose vertices and edges represent states and transitions, respectively, and embeds all the possible paths that might be matched to the trace.

In general, HMM map matching algorithms differ in the way transition probabilities are evaluated. Newson and Krumm [Newson & Krumm 2009a] observed that the shortest path distance for successive GPS measurements is close to the great circle distance between them, i.e., the beeline distance embedded on the earth's surface. They subsequently derive an exponential probability distribution function to compute the transition between successive states in the HMM. Using dynamic programming, notably the Viterbi algorithm [Forney 1973], they compute the matching path in the HMM that maximizes the joint probability of the path's states and transitions. Depending on the probability transition model, other techniques make use of Kalman [Ochieng *et al.* 2003] or particle [Gustafsson *et al.* 2002] filters.

Nonetheless, for low-frequency sampling, HMM techniques suffer from the selection bias problem [Hunter *et al.* 2013], which causes the HMM to assign a strong weight to long road segments with low network connectivity (such as highways) compared to the surrounding segments. The Conditional random field model (CRF) [Xu *et al.* 2015] addresses this problem and produces a better matching, although at the cost of a higher computational effort.

2.3.3 Machine Learning-based Map Matching

HMM-based algorithms are subject to latency issues when augmented for real-time applications, the reason being that future measurements need to be accounted for to identify the best transition. Therefore, another approach consists of training algorithms to learn matching from historical data sets. Osogami et al. [Osogami & Raymond 2013] use inverse reinforcement learning to model a transition probability distribution for the HMM that incorporates additional metrics such as vehicle turns. Goh et al. [Goh *et al.* 2012] rely on Support Vector Machine (SVM) to derive the emission and transition probability functions based on several features such as measured speed, speed limit, vehicle heading direction and, road width.

There exist a wide variety of map matching algorithms [Chao *et al.* 2020, Hashemi & Karimi 2014], and the presented classification is mostly meant to underline the dominant characteristic common to many map matching algorithms. Nonetheless, most techniques favor a probabilistic approach because the targeted application is often real-time navigation. In that context, correctness is not necessarily the ultimate objective, but rather, the goal is to match efficiently and as accurately as permissible.

2.4 The Shortest Path Problem

The *Shortest Path Problem* (SPP) is a famed topic in graph theory [Madkour *et al.* 2017] with a wide range of applications, especially useful for route planning. In essence, given a graph $G(V, E)$, solving the *one-to-one* SPP consists of computing the path with minimum *cost* from a single source vertex $s \in V$ to a single target vertex $t \in V$. The cost represents any chosen metric attached to the set of edges E and often denotes travel time or traveled distance. Other metrics such as travel fare or the number of transfers are particularly valuable if the graph models a public transit network. There exist other variants of the problem, mainly, the *one-to-many* SPP requires computing a set of shortest paths between a single source s and a set of target vertices $T \subseteq V$. Lastly, the *many-to-many* SPP generalizes the problem to a set of source and target vertices $S, T \subseteq V$, where the goal is to solve the one-to-many SPP for each source $s \in S$. For time-dependent networks, the solution to the SPP further depends on departure time. In that context, the *earliest arrival problem* consists of computing the path departing at a time τ from the source and reaching the target as early as possible. In contrast, the *profile problem* consists of computing the shortest path as a function of time $\tau \in [\tau_a, \tau_b]$ for a range of potential departures.

Dijkstra's algorithm [Dijkstra 1959] is, historically, the classical solution to the shortest path problem. Also known as a *label-setting* algorithm, it greedily explores the graph but settles each vertex only once. That is, when the shortest path to a given vertex v is identified, v in never processed again, and hence the complexity of the algorithm is bounded by the size of the input graph. For large networks, however, with up to millions of vertices and edges, Dijkstra's algorithm is too slow for practical applications. Therefore, over the course of the past decades, a race toward faster, more efficient algorithms lead to *speedup* techniques that are several orders of magnitude faster than Dijkstra's. Notably, the 9^{th} Dimacs Challenge [Demetrescu *et al.* 2009] fueled the research by providing a common infrastructure for experimentation on large scale road networks. To run faster, most speedup techniques operate in two distinct stages: in an offline *preprocessing* step, specific information is extracted from the graph and subsequently leveraged to accelerate the online *query* step. This additional information is derived from specific features that transportation networks possess.

In the following sections, we review several shortest path algorithms, grouped by their common underlying speedup technique. In each section, we first present the original version of the algorithm, designed for time-independent networks. We then discuss the necessary alterations to adapt it to time-dependent networks. Although most algorithms can be adapted to handle time-dependency, their performance often drops significantly due to the discrepancies observed in time-independent and time-dependent models [Bast 2009]. For that matter, the last section of this chapter is dedicated to algorithms that were specifically designed for time-dependent networks and rely on other data structures than graphs.

2.4.1 Dijkstra's algorithm

Given a time-independent graph $G(V, E)$ and a source vertex $s \in V$, Dijkstra's algorithm [Dijkstra 1959] solves, by design, the one-to-many SPP. It assigns a distance label $k(v)$ denoting the tentative cost to reach vertex v from the source vertex s. All vertices are inserted into a priority queue (PQ) and their labels are initialized such that $k(s) = 0$ and $k(v) = +\infty$ for all remaining vertices. Then, at each iteration, the algorithm extract from the PQ the vertex v with the smallest distance label $k(v)$. From that vertex, it *scans* each neighboring vertex w, and assesses if the tentative cost $k(w)$ can be improved via the outgoing edge (v, w). That is, if $k(v) + c(v, w) < k(w)$, the tentative cost is updated to $k(w) = k(v) + c(v, w)$. Updating the tentative cost is referred to as *relaxing the edge*. After scanning all of v's outgoing edges, vertex v becomes *settled*, and its tentative cost cannot be improved anymore. Hence, it corresponds to the actual cost of the shortest path $P(s, v)$. The algorithm then extracts another vertex from the PQ, and the whole procedure is repeated.

When a vertex is settled, it is never scanned again and therefore, the algorithm may terminate as soon as the target vertex t is settled (in the case of the one-to-one SPP) or until the PQ is emptied (one-to-all SPP), at which point the shortest paths to all vertices in the graph are known. Retrieving the actual shortest path is known as *path unpacking*. In the case of Dijkstra's algorithm, each vertex is also assigned another label $p(v)$ denoting its *parent* vertex, i.e., when an edge (u, v) is relaxed, it sets $p(v) = u$. Hence, the shortest path $P(s, t)$ is obtained through a recursive call to parent vertices from the target t.

There exists a multitude of priority queue implementations that support the basic operations needed to implement Dijkstra's algorithm [Williams 1964, Fredman *et al.* 1986, Fredman & Tarjan 1987]. Choosing the best implementation is, however, not straightforward. The performance of the algorithm is affected by the structure of the graph itself (sparse, dense, planar), the additional overhead, and the individual complexities of the priority queue operations [Chen *et al.* 2007]. With a binary heap (most widely used), the *deleteMin* operation allows extracting the vertex with the smallest distance label from the queue in constant time $\mathcal{O}(1)$. Extracting a vertex requires, however, rearranging the structure of the queue by *floating* the next vertex with the smallest distance label to the top of the queue. This is done in $\mathcal{O}(\log n)$, where n is the number of vertices in the queue. Furthermore, relaxing an edge consists of updating the distance label of a vertex inside the queue. This is known as a *decrease-key*, which is an amortized constant time operation with complexity $\mathcal{O}(m)$ where m denotes the number of times the decrease-key operation is performed. Consequently, the overall complexity of the algorithm is $\mathcal{O}(n \log n + m)$, where n is the number of settled vertices and m the sum of outgoing edges of all settled vertices. For the one-to-all SPP or a worst-case scenario of the one-to-one SPP, all the vertices in the graph are settled, and therefore n is bounded by $|V|$, the size of the vertex-set.

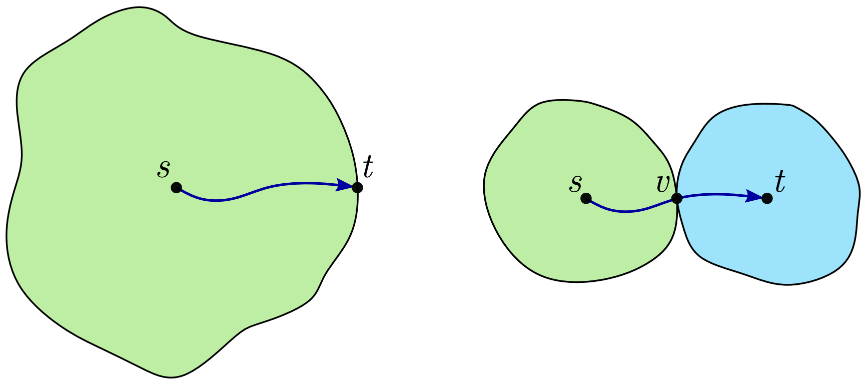
Time-dependent Dijkstra. Augmenting Dijkstra for time-dependent networks is straightforward [Cooke & Halsey 1966]. The goal is to compute $d(s,t,\tau)$ denoting the cost of the shortest path from the source s to the target t when departing at time τ. The algorithm proceeds as previously described, except that relaxing an edge (v,w), requires evaluating its weight at time $k(w) = \tau + d(s,v,\tau)$.

2.4.2 Bidirectional search

Bidirectional search [Dantzig & Thapa 2006] is a fundamental strategy, often coupled to other speedup techniques. As depicted in figure 2.7a, the plain version of Dijkstra's algorithm, also known as *unidirectional* Dijkstra, scans and then settles the nearest vertices to the source until it reaches the target vertex. Therefore, its *search space* representing the set of settled vertices is a "disk-like" shape centered at the source.

Bidirectional search improves upon unidirectional Dijkstra by running a *forward* search at the source, and a *backward* search from the target. In essence, it consists of two simultaneous Dijkstra algorithms, each with their own priority queues where the forward search is run on the graph $G(V,E)$ and the backward search is run on the reverse graph $\overleftarrow{G}(V,\overleftarrow{E})$, obtained by flipping the direction of all edges $(v,w) \in E$. When the forward (alternatively the backward) search settles a vertex v that has already been settled by the backward (alternatively the forward) search, the algorithm is stopped, and the shortest path $P(s,t) = (s,..,v,..,t)$ is obtained by concatenating the subpaths $P(s,v)$ and $P(v,t)$. As illustrated in figure 2.7b, the search space of bidirectional Dijkstra is two smaller disks with overall half the size of that of unidirectional Dijkstra, and therefore, yields queries that are approximately twice as fast.

Time-dependent Bidirectional Dijkstra. Intuitively, the analogous implementation of bidirectional search on time-dependent networks would suggest running both a forward and backward time-dependent searches to compute $d(s,t,\tau_d)$ and $d(t,s,\tau_a)$ respectively, where τ_d is the departure time and τ_a is the arrival time. Evidently, the arrival time τ_a is not known beforehand, and thus, the backward search requires further refinement. Instead of a time-dependent Dijkstra, the backward search is run on the reverse time-independent *lower bound graph* $\underline{\overleftarrow{G}}$ [Nannicini *et al.* 2008]. To construct such graph, each edge $(v,w) \in E$ is reversed, and its weight is set to $\underline{c}(w,v) = min\{f_{vw}(\tau) \mid \tau \in \Pi\}$ denoting the lower bound



(a) Unidirectional Dijkstra (b) Bidirectional Dijkstra

Figure 2.7: The search space of Dijkstra's algorithm in time-independent networks

of the time-dependent cost function $f_{vw}(\tau)$. The algorithm is summarized in three phases:

1. During the first phase, the forward search runs as a regular time-dependent Dijkstra at the source and the backward search as a time-independent Dijkstra from the target. Let M denote the set of vertices settled by the backward search. Phase 1 ends when the search spaces intersect at a given vertex v.

2. In the second phase, the algorithm evaluates the cost μ of the preliminary shortest path $P^*(s, t, \tau_d)$ passing through vertex v. Although P^* is not necessarily the shortest path we seek to compute, its cost $\mu \geq d(s, t, \tau_d)$ is an upper bound of the actual shortest path. The algorithm resumes until the backward search settles a vertex w such that $\overleftarrow{d}(w) \geq \mu$. That is, the cost to reach all the remaining vertices from the target exceeds the cost of the preliminary shortest path. At that point, the backward search terminates, and the additional vertices that it settled are subsequently added to M.

3. The forward search resumes but is restricted to only scan vertices belonging to M. The algorithm terminates when the forward search settles the target t.

In practice, however, bidirectional Dijkstra is even slower than unidirectional Dijkstra [Nannicini *et al.* 2008]. Nonetheless, it achieves speedups if we are willing to trade correctness and accept approximate solutions.

2.4.3 Goal Direction

Knowing the location of the target allows for making informed decisions while searching for the shortest path. Instead of "blindly" expanding from the source, the intuition behind goal direction is to use heuristics to guide the search by prioritizing vertices that are closer to the target.

When settling a vertex v, unlike Dijkstra which only considers the shortest distance between the source s and v, $A^\star$ (pronounced 'A-star') [Appi 1966] is a goal directed algorithm that estimates the total cost of the path P_{st} constrained to go through vertex v, as given by equation (2.2):

$$k(v) = g(v) + \pi(v) \tag{2.2}$$

The vertices in the queue are extracted in non-decreasing order based on their key value $k(v)$. The first part $g(v)$ denotes the cost of the currently known shortest path P_{sv} from the source s to vertex v. The second part $\pi(v)$ is called a potential function and estimates the cost of the shortest path P_{vt}, to reach the target t from v. To guarantee correctness, the heuristic must be *admissible*, meaning that it must not overestimate the cost of the shortest path and hence, satisfies $\pi(v) \leq d(v,t)$. A heuristic is said to be *consistent* if $\pi(v) \leq c(v,w) + \pi(w)$ for all $v, w \in V$ where $c(v,w)$ is the cost of edge (v,w). Consistency implies admissibility and guarantees that $g(v) = d(s,v)$ for each settled vertex v. Therefore, the algorithm settles each vertex only once, yielding faster execution times.

Setting, for instance, $\pi(v) = 0\ \forall v \in V$ is both consistent and admissible and is a special case where $A^\star$ behaves exactly like Dijkstra's algorithm. Often, when dealing with physical distances, $\pi(v)$ is set as a measure of the Euclidean distance $d_{Euc}(v,t)$, which is by definition consistent in the case of transportation networks as a consequence of the triangle inequality. If the cost represents travel time, we can set $\pi(v) = d_{Euc}(v,t)\ /\ speed_{max}$, where $speed_{max}$ denotes the upper bound of the speed in the network.

2.4.3.1 A*, Landmarks, Triangle inequality (ALT)

ALT [Goldberg & Harrelson 2005] uses the same approach as A*; however, it achieves faster speedups by computing better lower bounds for the potential function. In a preprocessing step, a small set of vertices called landmarks $L \subseteq V$ are selected. Then, for each landmak $l \in L$, the algorithm computes the cost of all shortest paths P_{lv} and P_{vl} for all vertices $v \in V$. This is accomplished using two independent instances of Dijkstra's algorithm with l being the source vertex: the first is executed on the regular graph G and computes the forward shortest paths P_{lv} while the second is run on the reverse graph $\overleftarrow{G}$ to compute the backward shortest paths P_{vl}.

During the query, the triangle inequality property guarantees that:

$$d(v,t) + d(v,l) \geq d(t,l) \tag{2.3}$$
$$d(l,v) + d(v,t) \geq d(l,t) \tag{2.4}$$

therefore, the potential function of vertex v with respect to landmark l can be set to the best lower bound of $d(v,t)$ as defined in equation 2.5. The overall best potential function is then obtained by scanning over the set of landmarks, that is, $\pi(v) = max\{\pi_l(v) \mid \forall l \in L\}$.

$$\pi_l(v) = max\{d(t,l) - d(v,l), d(l,t) - d(l,v)\} \tag{2.5}$$

The performance of the algorithm relies significantly on the quality of the landmarks L. A *good* landmark provides better lower bounds and consequently restricts the search space during the query. For road networks, landmarks that are geometrically located "before" the source vertex or "behind" the target vertex tend to be good landmarks [Goldberg & Harrelson 2005]. Consider an *st* query using a landmark l that lies beyond the target t from the perspective of the source s. In such configuration, the left side of the triangle inequality $d(s,l) - d(t,l) \leq d(s,t)$ is a good estimation of the true $d(s,t)$ distance.

Even though finding optimal landmarks is NP-hard [Fuchs 2010], there exist several landmark selection strategies [Goldberg & Harrelson 2005], and their performance is a tradeoff between the quality of the produced lower bounds and the required preprocessing time. Randomly selecting a set of landmarks is the fastest strategy but provides the worst results in terms of query times. Even then, random landmark selection is still significantly faster than a plain A* algorithm. The *farthest* landmark selection strategy consists of identifying a set of k landmarks such that the distance between any pair of landmarks $\{l_i, l_j\}$ is maximized. It proceeds iteratively and evaluates at each step the vertex in the graph that is farthest from the set of already selected landmarks. In general, good landmarks tend to be located at the periphery of the graph, and therefore, the *planar* landmark selection strategy splits the graph into k pie-like slices such that all regions are balanced in size. Then, it picks one landmark located at the outer border of each region such that, overall, all landmarks are mutually far one from another. There exist several other techniques and optimizations for landmark selection but often at the cost of a significant increase in preprocessing time.

Time-dependent ALT. In a time-dependent network, we must ensure that the potential function π remains admissible for any departure time τ. Therefore, one approach consists of precomputing the landmark distances on the lower bound time-independent graph $\underline{G}$. Therefore $d(v,l) = d(v,l,\tau_l) \leq d(v,l,\tau)$, where τ_l is the time when $d(v,l)$ is minimum, compared to all other times τ.

2.4.3.2 Arc Flags (AF)

Arc Flags [Hilger *et al.* 2009] is one of the fastest goal directed algorithms [Delling *et al.* 2013]. In a preprocessing step, it computes a partition of the vertex-set V denoted $\mathcal{C} = \{C_0, C_1, .., C_k\}$, where each element C_i is referred to as a *cell*. The intuition behind Arc Flags is to label all edges to assess, during query time, if an edge (v,w) is worth exploring to reach the intended target vertex. To do so, every edge (v,w) is labeled with a $k-$bit vector denoted $AF(v,w)$ such that the i^{th} bit is set $(AF_i(v,w) = 1)$ only if (v,w) lies on a shortest path leading to some vertex belonging to cell C_i. Initially, all *own-cell* flags are set: for all edges $(v,w) \in E$ such that $v, w \in C_i$, the cell's arc flag is set, that is, $AF_i(v,w) = 1$. Furthermore,

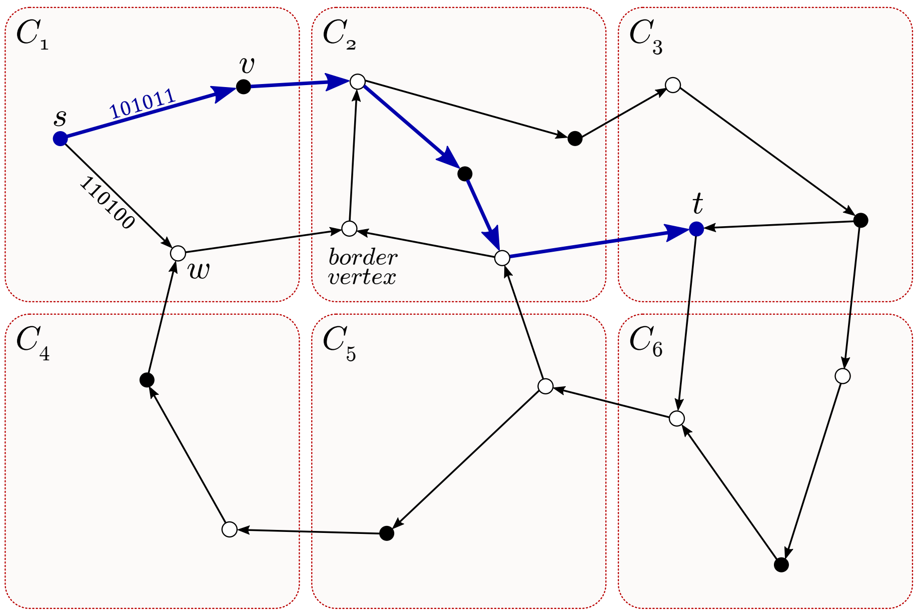


Figure 2.8: The graph is partitioned into six regions and their border vertices are outlined. The shortest *st* path is shown in bold blue edges. During the query, Arc Flags algorithm prunes edge (s, w) because the third bit-flag is not set.

intuitively, any shortest path terminating inside a cell C_i must cross its borders. Given a cell C_i, a vertex $b \in C_i$ is called a boundary vertex if there exists an edge (v, b) such that $v \in C_j \neq C_i$. Therefore, the remaining arc flags are computed by growing, from each boundary vertex b, a shortest-path tree T_b rooted at b by running Dijkstra's algorithm on the reverse graph $\overleftarrow{G}$. The arc flag of an edge (v, w) is set for a cell C_i if (v, w) is a tree edge for at least one shortest-path tree, grown from a given boundary vertex b of cell C_i.

During the query phase, a Dijkstra-like algorithm works by pruning all edges, not leading to the cell containing the target vertex, significantly shrinking the search space compared to plain Dijkstra. This process is illustrated in figure 2.8: the third bit-flag of all the edges along the shortest *st*-path (shown in bold) is set because $t \in C_3$. Therefore, initially at vertex s, Arc Flags only scans the neighboring vertex v but skips vertex w. However, once the target cell is reached, arc flags are not beneficial anymore, and AF behaves exactly like Dijkstra. Therefore, as an improvement, multilevel partitions [Möhring *et al.* 2007] are used to split the graph into finer cells further. Upon reaching the target cell, the algorithm accesses arc flags of a higher level in the partition until the target is reached. Although currently among the fastest goal-directed algorithms, Arc Flags' major drawback is its preprocessing time, which requires up to several hours on large graph instances.

Time-dependent AF. To address time-dependency, the simplest approach is to set the arc flag $AF_i(v, w) = 1$ of an edge (v, w) if it lies on a shortest path toward cell C_i for at least one departure time $\tau \in \Pi$, where Π denotes the time period of the graph. Although significantly less effective than in the time-independent scenario, this approach still allows edge pruning during the query, based on the departure time. To fully incorporate time-dependency to arc flags, instead of computing reverse shortest-path trees, we compute a reverse *profile* graph for each boundary vertex b of every cell. The reverse profile graph $\overleftarrow{G}_b$ for the boundary vertex b encodes all shortest paths $P_{vb}(\tau)$ for all vertices $v \in V$ and times $\tau \in \Pi$. Therefore, during the query, time-dependent AF consists of a time-dependent Dijkstra with the additional feature of pruning non-beneficial edges evaluated using the precomputed profile graphs. Nonetheless, running profile queries is time-expensive, and therefore, preprocessing time increases significantly, rendering time-dependent AF not practical for large graph instances. Another approach consists of computing approximate profiles but trades off correctness for the sake of preprocessing time [Delling & Wagner 2009].

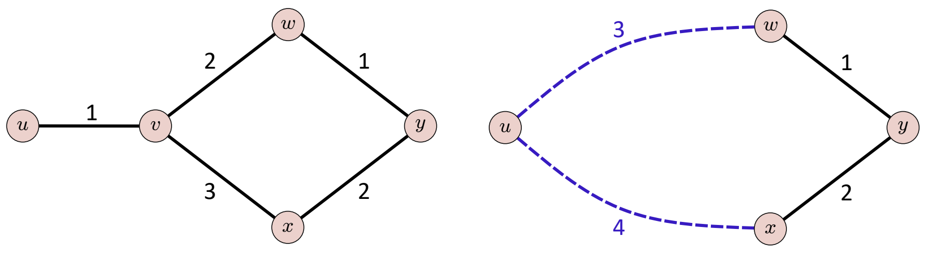
2.4.4 Hierarchy

Road networks are intently designed to maximize traffic flow: interstate highways are built to support a high load and accommodate fast traffic in order to minimize travel-time for long-distance trips. In contrast, urban roads are smaller and much more restrictive in terms of speed. They tolerate dense traffic and are suitable for navigating the city. This hierarchy, inherent to road networks, can be exploited to significantly speedup shortest path queries. Typically, if the source and target vertices are *sufficiently* far from each other, the shortest path, starting at the departure location, gradually converges into more *important* roads. Similarly, once close to the target location, the shortest path merges back to smaller roads.

2.4.4.1 Contraction Hierarchies (CH)

The intuition behind Contraction Hierarchies [Geisberger *et al.* 2008] algorithm is to remove unimportant vertices from the graph and insert shortcut edges that preserve shortest path distances. Hence, during a query, the added shortcuts allow to skip over most of the network resulting in a significant speedup. This vertex removal process is called a *contraction*: a vertex v is temporarily removed from the graph and for each pair of edges (u, v), (v, w), incident to v, such that (u, v, w) is the shortest uw-path, a shortcut edge (u, w) is added. Figure 2.9 illustrates the process of a vertex contraction and the resulting contraction hierarchy. During preprocessing, the algorithm contracts, in some predefined order, all of the graph's vertices. The obtained shortcut edges together with the original graph form a *contraction hierarchy.*

Nonetheless, the order of contraction significantly impacts the performance of the algorithm. Therefore, each vertex is assigned an *importance* (a rank), and subsequently, all vertices are contracted from least to most important. Geisberger et al. [Geisberger *et al.* 2008] list several criteria to evaluate the importance of a



(a) Before contracting vertex v (b) After contracting vertex v

Figure 2.9: Contracting vertex v: two shortcut edges (u, w) and (u, x) are inserted. However, no shortcut is added between vertices w and x because (w, v, x) is not a shortest path.

vertex:

1. *Edge difference:* when a vertex is contracted, its incident edges are removed, and new shortcut edges are inserted. This edge difference is a metric that captures the density of the hierarchy as a consequence of the contraction order. The goal is to achieve a vertex order that minimizes edge difference to produce a sparse contraction hierarchy. The vertices are scanned twice: during the first pass, the edge difference of each vertex is evaluated, then, during the second pass, vertices are contracted from smallest to largest edge difference.

2. *Uniformity:* relying on a single criterion such as edge difference is not efficient. In that case, the first vertex to be contracted corresponds to the tip of a dead-end road because no additional shortcut edges are required. This triggers a linear contraction of the whole road segment, and although edge difference is minimized, no speedup will be achieved during the query because of the lack of shortcuts. One approach consists of counting for each vertex the number of its neighbors that have already been contracted to achieve a uniform contraction throughout the graph.

3. *Contraction cost:* to contract a vertex v with incident edges (u, v) and (v, w), we must verify if (u, v, w) is indeed the shortest uw-path in order to insert a shortcut edge (u, w). This verification consists of running a local shortest-path search. Depending on the local topology in the neighborhood of v, this shortest-path search may be expensive. Therefore, a good strategy to reduce preprocessing time is to contract expensive vertices last because the graph gets smaller (as contracted vertices are removed), and therefore, the shortest-path search is faster.

During the query phase, a bidirectional variant of Dijkstra's algorithm is employed: both the forward and backward search process vertices that are higher-up in the hierarchy until they settle a common vertex (the shortest path vertex with the highest rank).

Time-dependent CH. Contraction Hierarchies is correct, no matter the vertex ordering. Therefore, we can order vertices as in the time-independent case. Furthermore, considering that the query phase is a bidirectional search, we augment it as explained in section 2.4.2, except that both forward and backward searches must always process vertices that higher up in the hierarchy.

2.4.5 Hybrid Algorithms

Each shortest path speedup technique relies on particular graph properties, themselves derived from the fundamental characteristics of transportation networks such as topology, planarity, and hierarchy. Therefore, it is often possible to combine several shortest path techniques to achieve even better speedups [Goldberg *et al.* 2006, Goldberg *et al.* 2007]. This opens a new realm of possibilities considering the already great variety of shortest path algorithms. Individual speedup techniques can be thought of as "flavors" that can be manipulated together to derive better recipes for the shortest path problem.

CHASE algorithm [Bauer *et al.* 2010] combines the principles of contraction hierarchies and arc flags. During preprocessing, it computes the graph's contraction G', then, a subgraph $H \subset G'$ is extracted and consists of the V_H vertices of the highest rank. Finally, it computes the arc flags of the partition on the subgraph H. The size of V_H is a tuning parameter and was set to 5% as computing arc flags is time-consuming. During a query, the bidirectional search is halted when a vertex v belonging to H is reached. When both the forward and the backward searches stop, the search is resumed using the precomputed arc flags.

2.4.6 Journey Planning in Time-Dependent Networks

Route planning in public transit networks is often referred to as *journey planning*: routes between stations are usually fixed (bus lines, train rails), and therefore, the goal is to compute the most cost-effective sequence of trips. In this context, a trip denotes part of a journey from the moment a passenger embarks onto the bus, for instance, until the moment he disembarks at a given stop. Furthermore, unlike in road networks where travel time is often the single most important metric to optimize for, journey planning is usually a multicriteria problem. Not only are passengers concerned with travel time, but the number of transfers as well as the monetary cost are important factors to consider. Consequently, optimizing several criteria simultaneously consists of computing a Pareto set of non-dominating journeys. A journey J_1 dominates another journey J_2 if J_1 is either equal or better than J_2 for all the considered criteria.

Dijkstra can in fact be augmented into the *Layered Dijkstra* (LD) algorithm [Brodal & Jacob 2004] to optimize both travel time and number of transfers or the *Multi-Label Correcting* (MLC) algorithm [Pyrga *et al.* 2008] for arbitrary criteria. Nonetheless, non-graph based methods that rely on dynamic programming to directly parse the public transit timetable data are usually significantly faster.

2.4.6.1 Round Based Public Transit Routing (RAPTOR)

RAPTOR [Delling *et al.* 2015] is a journey planning algorithm that optimizes, in the simplest version of the algorithm, both travel time and the number of transfers. Each stop p is labeled with a vector $(\tau_0(p), \tau_1(p), .., \tau_K(p))$ where $\tau_i(p)$ designates the earliest arrival time at p after i transfers at most and K is the predefined upper bound on the number of rounds. For a given round k, RAPTOR computes $\tau_k(p)$ for all stops $p \in \mathcal{P}$. A public transit route is a sequence of stops traversed by several shuttles according to a timetable schedule. Hence, RAPTOR scans each route, exactly once, to extend the journeys evaluated during the previous round. At the end of each round, footpaths connecting nearby stations are considered: in case walking up to a given stop improves its arrival time.

Other versions of RAPTOR address the range problem denoting a window of potential departure times (rRAPTOR) and multicriteria queries that further optimize monetary cost and walking time (McRAPTOR). Considering that RAPTOR does not require preprocessing, it is suitable for dynamic queries where unpredictable delays can occur. Furthermore, route scanning can be parallelized across many CPU cores for non-conflicting routes.

2.4.6.2 Connection Scan Algorithm (CSA)

The time-expanded graph model seen in section 2.2.3 is acyclic. In fact, instead of adding an edge between the last and first vertex at each station, we can *unwrap* the timetable to add additional vertices for repeated events beyond the timetable's period Π. Connection Scan Algorithm [Dibbelt *et al.* 2013] exploits this property: instead of using a graph model, it sorts all connections into an array data structure denoted C, based on departure time. Consequently, given a source stop p_s and a departure time τ, CSA initializes the earliest arrival time to all other stops in the network to $\tau(p) = \infty \mid \forall p \in \mathcal{P}$. It then scans C, in order, starting from the connection c_i whose departure time $\tau_i \geq \tau$. It subsequently evaluates for each connection $c \in C$ if it is reachable and, in that case, if it improves the arrival time of the connection's target stop. Once the array is scanned, the computed $\tau(p)$ is correct for all stops.

2.5 Dynamic Route Planning

The ubiquity of traffic data accelerated the development of Intelligent Transportation Systems (ITS). Large volumes of information such as road segment speeds, traffic counts, lane density, and accident detection are compiled into datasets to enable informed navigation and traffic forecasting. At regular time intervals, sometimes in seconds, a global *snapshot* of the up-to-date traffic conditions in the network is recorded. This data often benefits route planning systems that help commuters minimize their end-to-end travel time. Nonetheless, congestion is still a contemporary problem: the 2015 urban mobility scorecard [Schrank *et al.* 2015] reports that urban Americans traveled an extra 6.9 billion hours and had to purchase an extra 3.1 billion gallons of fuel. To reliably arrive on time, travelers had to plan 48 minutes for a trip that should last only 20 minutes in light traffic. The estimated cost of congestion is steeply increasing since reported in 1982, with a cost of $42 billion against a total of $160 billion in 2015.

To alleviate congestion, several strategies are undertaken, for instance, increasing the network's capacity by redesigning bottlenecks [Systematics 2005], managing traffic flow through dynamic control of traffic lights [Li & Sun 2019], and relaying traffic data to travelers. Nonetheless, congestion is an emergent phenomenon due to complex and often unpredictable interactions in the network. It is estimated that 50% of travel time delays are linked to non-recurring congestion [Güner *et al.* 2012]; thus, accurate traffic forecast remains a challenging problem.

Route planning systems address this problem, often selfishly, by continuously re-routing each user to avoid congested areas [Cabannes *et al.* 2017]. The continuous assessment of each query as traffic data evolve in real-time requires significant processing resources. Effective routing in the dynamic context is, therefore, contingent on efficient management of traffic information.

2.5.1 Re-routing in road networks

The field of shortest path algorithms flourished in the last decade, with continuous improvement in execution times. Pre-processing consists of transforming the initial graph to reply efficiently to route queries [Bast *et al.* 2016a]. However, using real-time information implies that the preprocessing phase has to be re-executed.

[Gmira *et al.* 2019] proposes to construct a delivery plan for the Dynamic Vehicle Routing Problem (DVRP). Their solution collects speed values in real-time and update the path for a vehicle only if it becomes infeasible. However, they do not investigate the sub-optimality cost, *i.e.*, a route is not updated if no constraint is violated, even if its travel time is not minimal. Understanding when to re-route vehicles is critical to reducing the re-computation cost. [Pan *et al.* 2013] propose an infrastructure-based approach: when congestion is detected, the system asks nearby vehicles to re-compute their shortest route. Congestion threshold, as well as the set of vehicles to re-route, impact the efficiency of this proposal significantly.

In the Dynamic Shortest Path problem (DSP), a re-routing algorithm tries to update the shortest path to handle multiple edge weight updates [Chan & Yang 2009]. Such an approach is much more efficient than re-executing the shortest path algorithm from scratch.

2.5.2 Real-time data sources

Collecting travel-time measurements of each road segment in real-time is a challenging task. One approach is to rely on the GPS trajectories of a collection of vehicles to deduce the specific travel time of each road segment [Sanaullah *et al.* 2016, Duan *et al.* 2018]. For this purpose, each trajectory has to be mapped to a set of road segments while minimizing the mismatch ratio [Falek *et al.* 2018]. [Ladino *et al.* 2016] propose to merge heterogeneous data sources (e.g., cameras, induction loops) to create a combined, more accurate dataset. Furthermore, imputation techniques [Chen *et al.* 2018] help to reconstruct missing data (e.g., a road segment is not crossed for a short time period).

Distributed ITS infrastructures try to estimate the level of congestion of each area locally. [Wang *et al.* 2016] use, for instance, a multi-agent system, where each vehicle is an agent, exchanging its measurements locally via Vehicle-to-Vehicle communications. However, this approach is less accurate than collecting all the travel times in real-time. The accuracy depends on the strategies to collect real-time data [Mathew & Xavier 2014] relying on: i) RFID tags, ii) image processing, iii) an extensive collection of wireless sensors, iv) instrumented connected vehicles, v) cellular networks, vi) GPS. [Lai & Kuo 2016] proposes to exploit the cellular network statistics to estimate the position and the speed of each vehicle. [D'Andrea *et al.* 2015] even infer the traffic conditions from a social network application (Twitter). With machine learning, they try to identify tweets related to traffic incidents. However, all these solutions relying on indirect data are unreliable by nature and cannot accurately forecast congestion.

Recent techniques propose to adopt a synthetic model. Typically, the SUMO simulator [Behrisch *et al.* 2011] is used to generate the mobility pattern of a large number of devices, using realistic urban points of interest. Travel and Activity PAtterns Simulation (TAPAS) was used to generate the well-known TAPAS-Cologne dataset [Uppoor *et al.* 2014]. However, to the best of our knowledge, no study has been conducted to verify the accuracy of these simulations. In [Kamga *et al.* 2011], the authors use simulated data from VISTA [Ziliaskopoulos *et al.* 1999] to determine the impact of incidents on travel times. Other traffic simulation software such as Dynameq [dyn 2020] and TRANSIMS [Barrett *et al.* 2002] provide simulated datasets of the travel times in a road network.

2.5.3 Traffic prediction

Recently, traffic prediction has received much attention in order to provide *prediction as a service* [Liebig *et al.* 2017]. These techniques try to consider the inherent characteristics of road networks (e.g., flow conservation) to predict future trends accurately. Predictions rely on computationally intensive techniques such as, e.g., bee colony optimization [Dell'Orco *et al.* 2016], or spatiotemporal random field [Liebig *et al.* 2017]. Alternatively, [Wang *et al.* 2019] propose to predict end-to-end travel-times directly but limiting its practical interest for route computation.

However, congestion is highly variable: [Coifman & Mallika 2007] highlight that 48% of the congestion is difficult to predict. [Li *et al.* 2014] concludes that accidents are impossible to predict and even complicated to detect early.

2.5.4 Experimental evaluations on dynamic networks

Congestion is complicated to model; however, the vast majority of experimental evaluations in the literature rely on simulation models to study the impact that congestion has on routing performance in dynamic networks. Mostly, this is due to the secrecy surrounding access to traffic information. [Smith *et al.* 2014] evaluates the impact of accidents on traffic congestion using a vehicular simulation and highlights the need for using actual traffic conditions to predict the travel time. Similarly, [Wang *et al.* 2013] provides a performance analysis of different route planning algorithms (i.e., Dijkstra, static A*, dynamic Dijkstra, and dynamic A*) in smart cities. Their experiments use the TAPAS-Cologne dataset [Uppoor *et al.* 2014], which was built by generating traffic demand using TAPAS and Gawron's algorithm for traffic assignment. They consider a rush hour during a weekday to evaluate the impact of traffic jams. [McArdle *et al.* 2012] attempts to simulate traffic in the Greater Dublin region. Typically, each vehicle selects its destination according to a radiation model that estimates the probabilities of interactions between different regions.

In recent years, more access has been gradually granted to the research community. Nonetheless, most studies are conducted on small geographical areas, focusing on the traffic patterns of a few selected highways only. [Güner *et al.* 2012] proposes dynamic routing models to improve delivery performance. They rely on a simulated network of South-East Michigan freeways. Although they incorporate historical traffic data, the overall network consists of 30 vertices and 98 edges only, which is too small to capture the impact of congestion. [Liang *et al.* 2018] model a trip assignment system for automated taxis to assess its impact on congestion. They develop a case-study city for Delft, Netherlands, using a survey dataset comprising origin and destination trends and typical departure and arrival times. Nonetheless, the road network they model consists of 46 vertices and 66 edges only.

As shown throughout this section, the dynamic routing problem is a vast topic and a longstanding problem in research. Although access to traffic information has significantly benefited the research community, efficient traffic management to mitigate congestion remains a complex task. Furthermore, so far, we only focused on road networks. Nonetheless, means of transportation are diverse, and thus, augmenting the dynamic routing problem to multimodal networks is even more difficult.

2.6 Multimodal Route Planning

Many algorithms compute shortest-path queries in mere microseconds on continental-scale networks. Most solutions are, however, tailored to either road or public transit networks in isolation. To fully exploit the transportation infrastructure, multimodal algorithms are sought to compute shortest-paths combining various modes of transportation.

Transit [Antsfeld & Walsh 2012] operates on a graph combining different public transit networks and evaluates the shortest path based on the associated risk of transfers (probability to miss one transfer and its impact on travel time) on a combination of transit networks. We believe, however, that *true* multimodal networks must combine both unrestricted networks such as road and pedestrian networks, and schedule-based networks such as trains, trams, and buses. Otherwise, a classical time-dependent variant of Dijkstra's algorithm [Bauer *et al.* 2011] would suffice to solve the problem.

2.6.1 Access-Node Routing

ANR [Delling *et al.* 2009] was designed for long-range trips with the assumption that the road network is only used at the beginning and end of the trip while most of the distance in-between is covered using some type of public transportation. Considering that the subgraph designating the road network in a multimodal graph is usually far bigger (and denser) than the public transit network, ANR relies on table lookups to skip the road network and focus the search on the much smaller, public transit network. It precomputes *access-nodes*: a set of vertices $A \subset V$ in the multimodal graph forming a boundary between the road and the public transit network. Then, for each vertex v in the road network, it precomputes all shortest paths P_{va} for all access nodes $a \in A$. Therefore, during the query phase, it skips the road network by performing table lookups and runs a many-to-many version of Dijkstra's algorithm on the public transit network. For each vertex in the road network, a profile-graph consisting of the shortest paths to all access-nodes is precomputed, requiring significant additional memory. An alternative technique, core-based ANR, hierarchically contracts the road network first. Subsequently, access-nodes are only evaluated for the *core*, i.e., the contracted graph, with significantly fewer vertices.

2.6.2 State-Dependent ALT (SDALT)

State-Dependent ALT (SDALT) [Kirchler *et al.* 2011] solves the Label Constrained Shortest Path Problem (LCSPP) by adapting ALT [Goldberg & Harrelson 2005], a popular goal directed algorithm (discussed in section 2.4.3.1), to solve multimodal queries. The intuition is to speed up Dijkstra with a heuristic based on the triangle inequality observed in transportation networks. In essence, it computes during preprocessing, a set of landmark vertices, then, for each landmark, it constructs a constrained shortest-path tree to (and from) all other vertices in the graph. During query time, the preprocessed shortest path costs are used to assign a potential to each vertex representing the tentative distance to reach the target vertex. It relies on D_{RegLC} [Barrett *et al.* 2000], a multimodal version of dijkstra, constrained with predefined automata. The complexity of the automaton impacts both preprocessing and query times. The main drawback of this approach is scalability: computing landmark distances becomes too costly for large graph instances.

2.6.3 User Constrained Contraction Hierarchies (UCCH)

User Constrained Contraction Hierarchies (UCCH) [Dibbelt *et al.* 2015] is a hierarchical technique originally designed for road networks. The main idea is to contract during preprocessing the graph vertices ordered by their measured *importance*. For each contracted vertex, a shortcut edge is inserted between its neighboring vertices and whose weight preserves the cost of the shortest path containing the contracted vertex. To solve a query, the algorithm runs a bidirectional Dijkstra, only scanning vertices with higher importance, until the forward and the backward search meet. To separate modal constraints from preprocessing, the authors initially split all vertices into different sets based on their labels. They contract each set of vertices separately to make sure that all shortcut edges have a unique label.

2.7 Conclusion

We reviewed in this chapter all the background we deemed necessary to support the material in the upcoming chapters. In essence, depending on the type of network, certain graph representations are more suitable to capture the network's structural characteristics. Nonetheless, most speedup techniques that perform well in time-independent networks are often several orders of magnitude slower when adapted to time-dependent networks. Because of the strong structural discrepancies between the road and public transit networks, algorithms that are based on bidirectional search, goal direction, and hierarchy, for instance, are significantly slower on public transit networks. Instead, specialized algorithms that are usually non-graph based are better at solving the shortest path problem in public transit networks. Furthermore, shortest path algorithms often consist of a preprocessing, offline phase, prior to resolving queries. The performance of the algorithm then becomes a tradeoff between preprocessing times (and memory requirements) and the resulting query times. Developing new algorithms is, therefore, contingent upon all the stated considerations.

The ubiquity of traffic information and the movement toward open access to mobility data over the recent years enabled further research into the map matching and the dynamic routing problems. Nonetheless, most contributions are based on simulations with inaccurate assumptions about the nature of traffic congestion. Furthermore, augmenting route planning solutions to handle multimodal networks is not trivial due to the structural discrepancies of time-dependent and time-independent networks. We researched over the course of the thesis these topics and provided in the upcoming chapters a detailed description for each of our contributions.

Chapter 3

Unambiguous Map Matching

Contents

The concept of Smart City refers to modern cities relying on ICT for increased efficiency [Gharaibeh *et al.* 2017]. A myriad of deployed devices allow cities to perform traffic light management, traffic congestion avoidance and smart transportation. Based on the produced data, the smart city services can exploit a real time road map (e.g. speed, congestion level). In particular, dynamic route planning needs real time traffic conditions to guide vehicles and users through the optimal route [Liebig *et al.* 2017]. Yet, deploying thousands of traffic sensors is particularly expensive.

Participatory sensing [Guo *et al.* 2016] relies on individual bodies to collect a large dataset of measurements. Unfortunately, each participant provides independent sequences of measurements, which have to be merged to construct a consistent, real time view of the whole road or street network. A map matching algorithm aims to map all these distinct traces onto a common base map, so that we create a global dataset from a set of individual traces [Ahmed *et al.* 2015]. Openstreetmap[1] is commonly used as a base map [Jokar Arsanjani *et al.* 2015].

Map matching algorithms exploit an ordered sequence of waypoints (geographical coordinates) obtained with a cellular or GPS positioning system. Thus, each point is riddled with measurement errors. Typically, GPS-enabled smartphones provide an accuracy of 4.9 m in ideal conditions (open sky) [van Diggelen & Enge 2015]. Most map matching algorithms identify the most probable path corresponding to a

[1]`http://www.openstreetmap.org`

given individual sequence. However, trajectory planning requires a precise mapping of congestion information on the map for highly accurate travel time estimation.

Because acquiring a GPS location is very energy consuming, participatory sensing shall benefit from a low sampling rate GPS trace. However, reducing this sampling rate negatively impacts the accuracy [Quddus & Washington 2015], thus jeopardizing the identification of the most probable route for a given trace. For a dense map, multiple *similar* paths may exist between two data points, and no argument can fairly differentiate the actual path from its alternative.

In this chapter, we propose an **unambiguous map matching**, to identify all the possible paths corresponding to a sequence of waypoints. Indeed, a probabilistic method is to our mind insufficient: identifying the wrong route implies that a bias is created. For instance, an urban planner may need to count the actual number of vehicles for a specific street. Here we choose not to resolve ambiguities. We rather not consider segments between waypoints that we can not match definitely.

To the best of our knowledge, we propose the first unambiguous map matching method. The contributions of this chapter are:

1. we propose a map matching algorithm for sparse traces, enabling real-time mapping with a low computation time;

2. instead of a probabilistic method, we adopt here an unambiguous approach: we consider that the map matching has partially failed when several path candidates are obtained, and we thus identify the subroute which was used *for sure* by the trace ;

3. we thoroughly evaluate our solution, on emulated GPS traces on the London, Paris and Luxembourg maps. We always identify the initial path (no false negative), while reaching a small set of candidates (false positive), with a reasonable sampling period (under 50s).

4. we illustrate the performance of our algorithm on real GPS traces. We show its robustness to increasing sampling rates in the face of real-life GPS measurement errors.

3.1 Models and Assumptions

We analyze the case of a vehicle equipped with a GPS measuring device and moving on a given road network. The sampling frequency of the GPS can be adjusted to generate these measurements (i.e. latitude and longitude coordinates) at a constant rate T. A GPS trace is a sequence of such measurements with T seconds intervals from the start until the end of the journey. We use the term *true route* to refer to the real route used by the vehicle during its journey. Our goal is to reconstruct the *true route* from the issued trace.

Let us model the road network with a directed graph $G(V, E)$, V denoting the set of vertices, and E the set of edges. Each road intersection corresponds to a vertex

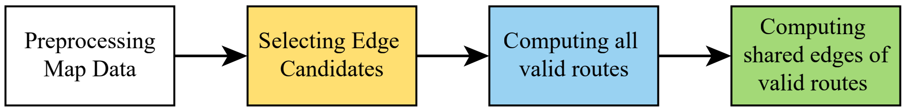

Preprocessing Map Data
Selecting Edge Candidates
Computing all valid routes
Computing shared edges of valid routes

Figure 3.1: Map Matching algorithm process

in the graph. Each road segment is an edge between the corresponding vertices. To each vertex is associated a geographical coordinate, and a road segment may be subdivided into small segments to preserve the geographical *shape* of the road. The weight of an edge is the time (in seconds) elapsed to join the two corresponding vertices, at the speed limit of the road section. We call a couple of two consecutive data points, the tail and the head vertices, respectively. Besides, we denote by `path` a sequence of connected vertices and edges in the map.

We use here Openstreetmap as base map, since this opendata project provides a free access to a huge and accurate collection of road networks.

3.2 Map Matching Algorithm

Our objective is to map a `trace` to a set of *valid routes* that are proven to be feasible for the journey. Then, we extract the set S of all shared edges among valid routes. We prove that $\forall e \in S,\ e \in \textit{true route}$. We use the notation described in table 3.1. Our map matching algorithm relies on four pipelined steps (Figure 3.1):

1. **Preprocessing map data:** we first organize the map into a regular grid to accelerate the identification of edges close to a given location;

2. **Selecting edge candidates for each measurement:** each measurement Z_i in the trace allows us to approximate the actual position P_i to a disk centered on Z_i with a radius $\mathcal{E}_{gps}$ (which is fixed to consider the measurement error). Then, we identify all the *edge candidates* ($Cand(Z_i)$) that correspond to Z_i. More precisely, any edge that crosses the boundary of the disk is an edge candidate. We make the distinction between the incoming (their tail is outside the disk) and the outgoing (their head is outside the disk) edges.

3. **Identifying valid routes from the set of edge candidates:** we then have to compute routes, which include one edge candidate for each Z_i, and which respect a time delay constraint. For each pair of consecutive measurements Z_i and Z_{i+1}, a Breadth First Search (BFS) calculates all the possible subroutes i.e a sequence of edges that start with an incoming edge candidate for Z_i and finish with an outgoing edge candidate of Z_{i+1}. The cost of a subroute must respect the elapsed time between measurements Z_i and Z_{i+1}. Finally, the different subroutes are assembled and we verify the end-to-end time delay constraint.

4. **Extracting common segments for all the possible routes:** we identify the

edges which are **for sure** on the journey, constituting a set of *common segments*. The longer the segments, the more accurate our algorithm: ideally one single segment would cover the whole route meaning that a single option matches our constraints.

Table 3.1: Notation used in this chapter

notation	**definition**
$\mathcal{E}_{gps}$	maximum GPS error for a data measurement
σ_{gps}	standard deviation of the GPS error (Gaussian distribution)
Δ_{speed}	security coefficient of speed excess authorized for the trace
$Z = \{Z_i\}_{i \in [0,\|Z\|]}$	GPS trace, with $\|Z\|$ data points
$tail(e)$ / $head(e)$	tail (respectively head) of the edge e
$C(e)$	cost (in seconds) to join $tail(e)$ and $head(e)$
$Cand(Z_i)$	set of candidate edges for the measurement point Z_i
$Outgoing(Z_i)$	outgoing edge candidates for the measurement point Z_i
$Incoming(Z_i)$	incoming edge candidates for the measurement point Z_i
$\mathcal{R} = \{e_i\}_{i \in [0,k]}$	a valid route from the edge e_0 to e_k
$subRoute(e_i, e_j)$	valid subroute between the candidate edges e_i and e_j
$D(Z_i)$	disk centered on Z_i with a radius $\mathcal{E}_{gps}$
δ_{cell}	size of a cell in the grid in degrees of latitude and longitude

3.2.1 Pre-processing

Identifying the edges close to a given location would require to fetch the whole graph, and to compute the distance of each of them. Thus, to accelerate the computation, we organized the map into geographical cells, i.e. rectangles of δ_{cell} degrees of width and height. We compute for each edge its set of cells by using a modified version of the Bresenham's algorithm [Bresenham 1965]. More precisely, an edge belongs to a cell if both of its end-vertices lie inside the cell, or if the edge crosses its boundaries. When searching for all the edges at a maximum distance of a specific location, we just have to scan all the edges belonging to the cells around the location.

3.2.2 Selecting edge candidates for each measurement

GPS typically follows a Gaussian distribution with standard deviation σ_{gps} [Heng *et al.* 2011]. Thus, the true position of the vehicle may be located anywhere in a disk centered on the measurement (Z_i) with the radius:

$$\mathcal{E}_{gps} = 3 \times \sigma_{gps} \tag{3.1}$$

which accounts for a 99.9 % certainty that the disk area contains the true position P_i.

An edge candidate is any edge that allows accessing and/or exiting the area that contains the true position of the vehicle. In other words, any edge that crosses the disk centered at Z_i. Thus, Z_i and the closest point of an edge candidate are separated by a distance smaller than $\mathcal{E}_{gps}$. Let us denote by $d(u, v)$ the Euclidean distance between the points u and v. An edge e is a candidate if one of the following conditions holds:

1. the head is outside the disk: e in an outgoing edge candidate;

$$(d(tail(e), Z_i) \leq \mathcal{E}_{gps}) \;\; \wedge \;\; (d(head(e), Z_i) > \mathcal{E}_{gps}) \tag{3.2}$$

2. the tail is outside the disk: e is an incoming edge candidate;

$$(d(head(e), Z_i) \leq \mathcal{E}_{gps}) \;\; \wedge \;\; (d(tail(e), Z_i) > \mathcal{E}_{gps}) \tag{3.3}$$

3. both the head and tails are outside the disk but the edge intersects the disk: e is both an incoming and outgoing edge candidate.

$$\begin{aligned}(d(proj(e, Z_i), Z_i) < \mathcal{E}_{gps}) \;\; \wedge \;\; (d(tail(e), Z_i) > \mathcal{E}_{gps}) \\ \wedge \;\; (d(head(e), Z_i) > \mathcal{E}_{gps})\end{aligned} \tag{3.4}$$

where $tail(e)$ and $head(e)$ denote the tail and head of the edge e respectively, and $proj(e, Z_i)$ denotes the projection of Z_i on the edge e.

3.2.3 Constructing valid routes from a list of edge candidates

We have now to infer the global route which corresponds to a sequence of edge candidates, and which also respect the time delay constraint. Formally, a route $\mathcal{R}$ is an ordered sequence of edges $E = \{e_i\}_{i \in [0,k]}$ such that $\forall i \in [0, k-1], head(e_i) = tail(e_{i+1})$.

We make a distinction between:

- A True route: this corresponds to the set of edges which were followed during the journey;
- Valid routes: all the possible routes which respect the constraints. The true route is included in the set of valid routes, thus preventing us from a false negative.

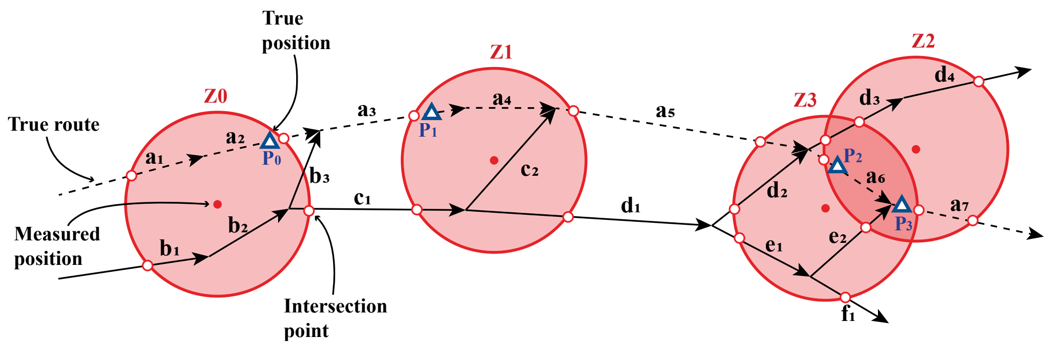


Figure 3.2: Example illustrating the process of appending subroutes.

We propose an iterative approach, where the route is grown step by step, appending subroutes from one measurement to the next. More formally, a subroute is a sequence of consecutive edges such that:

$$subRoute(e_i, e_{i+1}) = \{e_k \quad | \, head(e_k) = tail(e_{k+1})\} \\ s.t. \; e_i \in Incoming(Z_i) \wedge e_{i+1} \in Outgoing(Z_{i+1}) \quad (3.5)$$

Remark 1 *For the first pair of measurements* (Z_0, Z_1)*, a subroute begins with an outgoing edge candidate of* Z_0*. This exception is due to the fact that the first measurement* Z_0 *might not possess any incoming edge candidate, since the journey has started inside the disk, possibly in a dead-end street.*

To compute all the *valid routes*, we propose an iterative two step approach:

1. calculating subroutes, for a pair of consecutive measurements. We also verify that they are valid concerning the timestamps (time delay constraint). Typically, in Figure 3.2, the subroute $(a_3, a_4, a_5, a_6, a_7)$ is a valid subroute from Z_1 to Z_2.

2. appending the corresponding subroute to all the already computed valid routes, and then verifying the end-to-end delay constraint. In Figure 3.2, the subroute $(a_3, a_4, a_5, a_6, a_7)$ can be combined to the valid route (a_2, a_3), creating a new route $(a_2, a_3, a_4, a_5, a_6, a_7)$ from Z_0 to Z_2.

Remark 2 *We later retract the new route until we reach an incoming edge of measurement* Z_2*. In our example we would obtain* $(a_2, a_3, a_4, a_5, a_6)$ *after retracting the route. We explain why in the next subsection.*

When the algorithm reaches the last measurement Z_N, all the valid routes are identified.

3.2.3.1 Constructing the subroutes

The objective now is to construct all the possible subroutes between two edge candidates. We have to verify that a subroute is *valid*, i.e. the vehicle is able to join the two measurement points without exceeding the speed limit.

To identify all the valid subroutes, we apply a Breadth First Search (BFS) approach starting from each incoming edge candidate for the measurement Z_i, and stopping after either of the two conditions:

1. an outgoing edge candidate for the measurement Z_{i+1} is visited (a valid subroute is discovered). We will explain below why we have to stop at an *outgoing* edge candidate;

2. the cost of the subroute exceeds the delay constraint (the exploration has to stop, backtracking to the other edges to discover). We will explain in the next subsection how the cost of a subroute is actually computed.

The BFS algorithm evaluates all the subroutes in the graph starting from every incoming edge e of Z_i, such that e corresponds to the end of a route that was calculated earlier. All the neighbors are recursively explored, their *id* being pushed on top of a stack. In parallel, our algorithm maintains the minimum time cost to walk the explored subroute to verify the delay constraint. As soon as an outgoing edge for the next measurement point is scanned, the stack is saved: a new subroute was discovered.

When either a subroute has been discovered, or the delay constraint is violated, the BFS backtracks to the previous neighbor to explore. We implement a simple stack of unexplored neighbors since we have to make a complete exploration to detect *all* the possible subroutes.

Let us consider the example in Figure 3.2. A valid route has been constructed up to the head of the edge c_1. We apply the BFS strategy, and we obtain recursively the subroutes $(c_1, c_2, a_5, a_6, a_7)$ and $(c_1, d_1, e_1, e_2, a_7)$. Assuming that the subroutes $(c_1, c_2, a_5, d_3, d_4)$, $(c_1, d_1, d_2, d_3, d_4)$ and $(c_1, d_1, d_2, a_6, a_7)$ have been discarded because they violated the time constraints.

Remark 3 *Please note that we stop the BFS only when scanning an* ***outgoing*** *edge for Z_{i+1}. If the two measurement points are very close because of a small sampling period or large measurement inaccuracies, the two corresponding disks may overlap, preventing the algorithm from finding a subroute that ends with an incoming edge at Z_{i+1}.In Figure 3.2, the disks of Z_2 and Z_3 overlap. In particular, the incoming edge a_6 of Z_2 for the valid route is* after *the incoming edge a_5 of Z_3. Thus, we cannot compute a subroute that begins and ends with incoming edges at Z_2 and Z_3 respectively. On the contrary, we can safely stop at the first outgoing edge of Z_3, which is always located* after *the incoming edge of Z_2, as demonstrated below.*

Let Z_{min} be the minimum index such that the disks centered on Z_0 and Z_{min} respectively do not overlap. Similarly, let Z_{max} be the maximum index such that the disks centered on $Z_{|\{Z_i\}|-1}$ and Z_{max} do not overlap. For example, in figure 3.2, Z_{min} corresponds to Z_1 and Z_{max} corresponds to Z_1 aswell.

Lemma 1 *For any pair of consecutive measurements $(Z_i, Z_{i+1})_{i\in[Z_{min},Z_{max}]}$, there exists a sequence of connected edges such that the first is an incoming edge candidate of Z_i and the last one is an outgoing edge candidate of Z_{i+1}*

Proof 1 *The true route corresponds to a sequence of edges $\{e_k\}_{k\in[0,|\mathcal{R}|-1]}$ in the graph. By construction, at least one edge of the true route is at a maximum distance of $\mathcal{E}_{gps}$ from Z_i ($min < i < max$). Let $E_i = \{e_k\}$ denote this set, and kmin be the minimum index. e_{kmin} is an incoming edge for Z_i.*

Let us prove it by contradiction. If e_{kmin} is not an incoming edge, e_{kmin-1} is also in the set E_i, which is impossible since kmin is minimum. Besides, $kmin > 0$, since at least one edge is in the disk centered on Z_0 and not in the set E_i, by construction of Z_{min}. Thus, e_{kmin} has to be an incoming edge for Z_i.

We demonstrate similarly that an edge e_{kmax} of the true route is an outgoing edge for Z_{i+1}. Moreover, the incoming edge for Z_i has been traversed before the timestamp T_i corresponding to the measurement Z_i (first arrival in the disk). Inversely, the outgoing edge has been visited after T_{i+1}, and by definition, $T_i < T_{i+1}$, thus $kmin \leq kmax$.

The sequence of the edges in the true route, from e_{kmin} to e_{kmax}, forms a valid subroute, which will be scanned by the BFS.

Remark 4 *We cannot assume that the true route* enters *in the first disk, and* exits *from the last one. Thus, we scan all the edges inside or crossing the first and last disks. This exhaustive search increases slightly the computation time, but only for the first and last measurement points.*

3.2.3.2 Verifying the cost of a subroute

We have to compute the minimum time to travel a corresponding subroute. Thus, we define the cost for an edge e as the time required to join the tail and the head vertices with the speed limit of the corresponding edge.

$$\forall e \in E, C(e) = C\left(head(e), tail(e)\right) = \frac{d(head(e), tail(e))}{speed(e) * \Delta_{speed}} \tag{3.6}$$

with $speed(e)$ the speed limit associated to the edge e in the map, and Δ_{speed} a safety margin to take into account speed excesses. By extension, the cost of a subroute corresponds to the sum of the costs of its edges.

Because the exact location corresponding to a measurement is by definition imprecise, we do not know the exact distance travelled between two GPS measurements. We know that the minimum distance is the distance between the intersection points of the outgoing edge candidate of Z_i and the incoming edge candidate of Z_{i+1} with the disks $D(Z_i)$ and $D(Z_{i+1})$ respectively (positions P_0 and P_1 in Figure 3.2). Thus, we need to calculate the cost for the section of the subroute that lies outside the area enclosed by the disks $D(Z_i)$ and $D(Z_{i+1})$. This cost has to be below the time between two GPS samples ($T_{sampling}$) for the subroute to be feasible.

In figure 3.2, for the subroute (a_2, a_3, a_4, a_5), this cost corresponds to the sum of costs from the intersection of a_2 with $D(Z_0)$ to the intersection of a_3 with $D(Z_1)$.

Lemma 2 *The true route always corresponds to at least one valid subroute.*

Proof 2 *Let us consider the true route defined by a sequence of edges $\mathcal{R} = \{e_i\}_{i\in[0,|\mathcal{R}|]}$. Let P_{out} (resp. P_{in}) be the intersection of the last outgoing (resp. incoming) edge in $\mathcal{R}$ with the disk centered on Z_i (resp. Z_{i+1}). P_{out} is by definition the latest possible position of the vehicle along the true route at the timestamp T_i, which leads to the measurement P_i. Similarly, P_{in} is the earliest possible position for the timestamp T_{i+1}. Thus, $\sum_{e_k\in(P_{in},P_{out})} C(e) \leq T_{i+1} - T_i = T_{sampling}$. In conclusion, the true route will not be discarded by this condition, and is discovered with the BFS (lemma 1).*

3.2.3.3 Appending a subroute to a valid route

We now have to assemble the subroute with the set of valid routes to extend them until the last measurement point. Because we start the next subroute from an incoming edge (sequentiality constraint), we cannot assemble the subroute directly. We thus propose to:

1. prune all the edges in the subroute until (i) an incoming edge for Z_{i+1} is scanned, **or** (ii) the end of a valid route is detected. The second condition holds to still consider the case for which the two disks overlap, and stopping earlier the back-tracking is more efficient;

2. append this pruned subroute at the end of all the existing valid routes stopping at the corresponding edge candidate;

3. verify the end-to-end delay constraint for each valid route created in this way.

Then, the next measurement point will be considered.

We have to verify that the subroute respects the time constraint:

$$\sum_{e\in s} C(e) \leq i * T_{sampling} \tag{3.7}$$

with s being the sequence of edges in the valid route, from the last outgoing edge of Z_0 until the first incoming edge of Z_i.

Remark 5 *The first and last measurements in the trace Z_0 and $Z_{|Z|}$ respectively, are treated separately because neither can we guarantee that Z_0 possesses incoming edge candidates nor $Z_{|Z|}$ possesses outgoing edge candidates. Indeed, the journey could start and terminate at a dead-end street.*

3.2.4 Computing shared road segments in valid routes

Once all valid routes for the journey are established, we aim to compute the subset which is common to all the routes: we are sure that the vehicle has followed these specific edges.

More formally, let VR denote the set of valid routes. We compute the set of edges $\mathcal{S}$ such that:

$$\forall e \in \mathcal{S}, \quad \forall r \in VR, \quad e \in r \tag{3.8}$$

We use the term *correct matching ratio* (*cmr*) to define the ratio of all edges in $\mathcal{S}$ (weighed by their length) to the total length of the *true route*:

$$cmr = \frac{\sum_{e \in \mathcal{S}} d(e)}{\sum_{e \in \mathcal{T}} d(e)} \tag{3.9}$$

with $\mathcal{T}$ being the *true route*.

3.3 Experimental Evaluation

We carried out two different experiments to evaluate our algorithm. In the first experiment, we emulated GPS traces and used OpenstreetMap[2] as a common map. In the second experiment, we used a real GPS trace from a publicly available dataset [Newson & Krumm 2009b].

3.3.1 Emulated GPS traces

We first evaluate the accuracy of our unambiguous map matching method by emulating GPS traces. Prior to using the OSM data for our experiment, we cleaned it from duplicate entries and missing information. Specifically, we remove all the isolated vertices, i.e. not part of the largest connected component. These private or disconnected roads constitute less than 5% of the vertices of the graph. Finally, we precompute the grids for the OSM map (a tile corresponds to 0.001°).

Table 3.2: Dataset

Road Network	**London**	**Paris**	**Luxembourg**
Number of vertices	836,271	169,879	27,306
Number of edges	1,637,300	284,246	50,607
Degree	8	6	5

We used 3 different cities (Tab. 3.2) with different characteristics to evaluate the robustness of our method. For each city, we generated 500 GPS traces in the following way:

1. we pick randomly two waypoints (A and B), located approximatively 8km apart;

[2] https://www.openstreetmap.org/

Table 3.3: Evaluation setup

σ_{gps}	4.07 m (standard deviation of the GPS chipset)
Δ_{speed}	1.2 (safety margin for the speed limit, cf. section 3.2.3.2)
δ_{cell}	0.001° (size of the grid for the precomputation)
T_{samp}	$\in [1, 50]$ (sampling rate of the GPS trace)
CPU	Core i7-4600U CPU @ 2.10 GHz
memory	16 GB RAM
software	Java 8, Eclipse IDE Oxygen.2 Release (4.7.2)

2. we compute the shortest path from A to B. We emulate the movement of a vehicle, by following the path with the speed limit of each segment, as given by openstreetmap. A sample is saved every 1 second;

3. for each waypoint, we emulate a real GPS measurement, i.e. the actual location with an additionnal error caused by the GPS system. The location inaccuracy follows a Gaussian distribution with a standard deviation of 4.07m [Heng *et al.* 2011].

4. we finally subsample the GPS trace with a period comprised between 1 and 50 seconds. We can thus compare the different sampling rates: they correspond to the same journey.

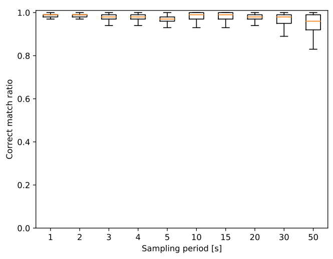
We consider here only regular sampling periods to more easily interpret the results. Indeed, an irregular sampling would correspond to a mix of different cases, making the results more difficult to interpret. Tab. 3.3 summarizes our evaluation setup.

We measured the following metrics:

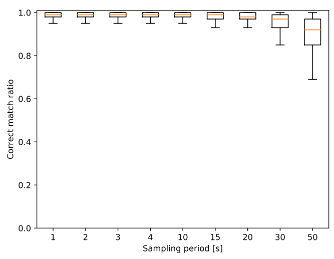
- **correct matching ratio:** we compute the weighted subset of edges which are common to all the valid routes (cf. section 3.2.4). $cmr = 1$ corresponds to a perfect match (i.e. there is only one valid route which happens to be the true route). Similarly, $cmr = 0$ means that even though we did not discard the *true route*, there exists at least one valid route that does not share any edge with the rest of the valid routes.

- **computation time:** time required to compute the set of valid routes.

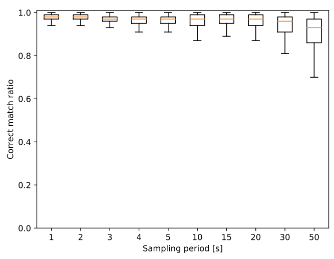
We then analysed the effect of road junctions density (i.e. number of edges) around a specific waypoint on the correct matching ratio. Figure 3.3 illustrates the correct matching ratio of London, Paris and Luxembourg, for sampling periods ranging from 1 to 50 seconds. The true route is always in the set of valid routes. In addition, for very small sampling periods, it is often the only valid route. Consequently, we are able to identify the whole true route. The accuracy decreases for higher sampling periods. Above a sampling of 15 to 20 seconds (depending on the city), some measurement points are far from each other, and several valid subroutes exist to join them.



(a) London

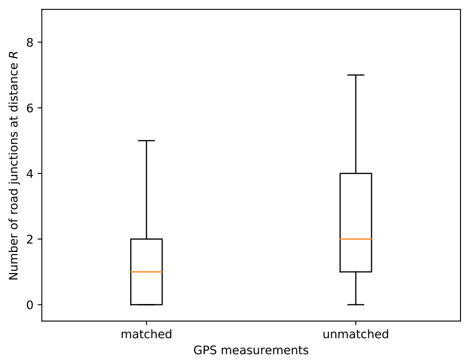


(b) Paris

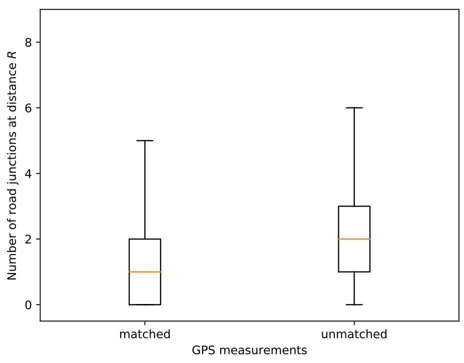


(c) Luxembourg

Figure 3.3: Correct matching ratio for London, Paris, Luxembourg road networks.



(a) London



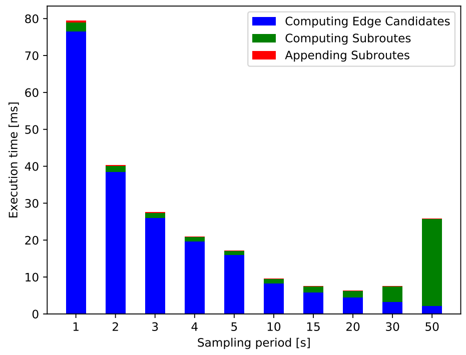
(b) Luxembourg

Figure 3.4: Road junction vertices density near matched and unmatched measurements ($R = 4 \times \mathcal{E}_{gps}$).

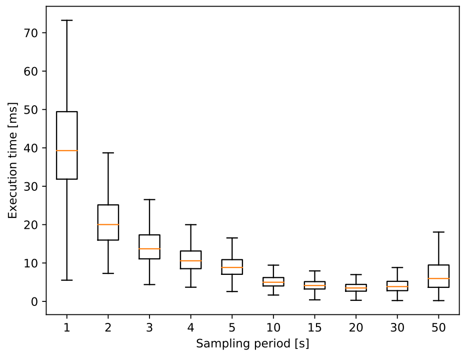
Our algorithm is unambiguous and does not pick randomly one of these possible routes. However, even for very large sampling rates (one GPS measurement every 50 s), more than 95% of the true route is accurately identified. In the worst case, for some cities, such as London, we are able to identify 80% of the true route. In conclusion, our unambiguous algorithm is very efficient to exploit sparse GPS measurements.

We also identify the metrics that affect the success or failure of the matching process. We suspect that the number of road junctions near a given measurement Z_i, increases the odds of computing more subroutes which decreases the probability of matching Z_i to a single route i.e the *true route*. For London and Luxembourg experiments, we measured the number of road junctions at a distance $d < 4 \times \mathcal{E}_{gps}$ of matched and unmatched measurements. For each experiment we matched approximately 12,000 measurements distributed over a dataset of 500 journeys. We observe in figure 3.4 that there is indeed more road junctions near unmatched measurements. Though, we still need to thouroughly test this hypothesis to be sure it holds for various road networks.

Figure 3.5b illustrates the computation time (for London experiment) when varying the sampling period. Whatever the conditions, we identify accurately the true



(a) Average execution time of main methods of map matching algorithm



(b) Execution time distribution

Figure 3.5: Performance of map matching algorithm based on execution time.

route in less than 80ms. Our algorithm is thus efficient for online strategies, where the reactiveness is of primary importance. We can observe in figure 3.5a the average execution time for the three main methods of the algorithm. For low sampling periods, we have many measurement points to consider, which increases the computation time for computing all edge candidates. Inversely, large sampling periods imply a larger computation time for the BFS, to identify all the possible subroutes, and verifying their cost. The computation time remains below 20ms for a sampling period of 5 s, which corresponds also to a close to perfect correct matching ratio.

These results were obtained with emulated GPS traces, whose parameters (e.g., measurement uncertainty, car speed) remained fixed throughout the emulation. We thus aimed at validating our proposition on real-world traces, whose dynamics may impact our results.

3.3.2 Real GPS trace (Seattle)

We here evaluate our algorithm in a real case scenario, by using data sampled at 1 Hz using a RoyalTek RBT-2300 GPS logger during a road trip in Seattle, WA [Newson & Krumm 2009b]:

- **Road Network:** The 2009 road network representation of Seattle, Washington, USA area.
- **GPS trace:** 7531 GPS points collected over 2 hours of driving (80 km)[3].
- **Ground Truth:** The true sequence of road segments that correspond to the GPS trace.

The error in the GPS measurements is not a metric we control (as for the emulated trace) but is rather due to the GPS measuring device and environment.

[3]The details of the trace are available on: `https://www.microsoft.com/en-us/research/publication/hidden-markov-map-matching-noise-sparseness/`

Such real-world measures allowed us to take into account the dynamics of the road network which would cause the vehicle to slow down or even halt instead of traveling at the allowed speed limit for the whole trip.

Figure 3.6 depicts the performance results of our algorithm for matching a real GPS trace. The results corroborate the first part of our evaluation using emulated GPS data. At small sampling periods (less than 5 seconds), we are almost able to reconstruct the true route in its entirety. The correct match ratio decreases to 85% with a sampling period of 50 seconds.

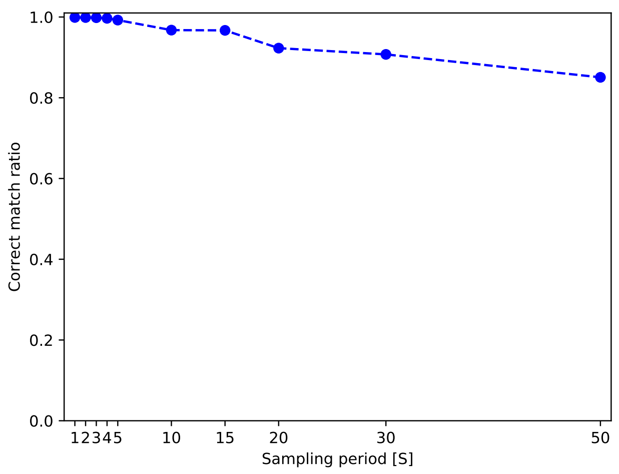


Figure 3.6: Correct matching ratio for Seattle network using real GPS trace data.

3.4 Conclusion

We presented a time-efficient map matching algorithm, to reconstruct the route from a sequence of data points. We proposed an unambiguous method: we do not aim to find the most probable route, but rather to identify the road segments that were used for *sure*. This way, we can merge heterogeneous traces, extracted for instance from a crowd-sensing application. Thus, our algorithm relies on identifying candidate edges for each data point, and computing routes among them which respect a time delay constraint. Our algorithm is time-efficient, and can be used for real-time situations. Our performance evaluation shows that our map matching technique unambiguously identifies more than 85% of the true route for a sampling period up to 50 seconds. When the data points are too inter-spaced, our algorithm terminates with several routes that cannot be differentiated.

Now that we are capable of correctly matching a significant proportion of real GPS traces, we can construct a dynamic representation of the network by matching measurements of a predefined area at regular time intervals. We address the problem of dynamic route planning in the following chapter.

Chapter 4

Dynamic Route Planning

Contents

Route planning algorithms [Bast *et al.* 2016a] compute a *route*, which consists of a sequence of road segments between a departure and arrival locations, and a *travel-time*, denoting its associated end-to-end cost. Historically, mobile navigation devices were used to guide the driver by autonomously computing a suitable route. With the wide adoption of smartphones, though, many applications nowadays benefit from the cloud where all computations are performed [Li *et al.* 2017c].

Route planning requires traffic information (estimated speed, vehicle counts) gathered from various sensors and probes within the network. Some mobile navigation devices rely on the median recorded travel-times of a set of monitored road segments to predict congestion. Smartphones and vehicles which usually embed GPS devices are also exploited by regularly relaying traffic measurements to the cloud so that a real-time model of the road network can be constructed [Ahsani *et al.* 2019]. Real-time traffic information is particularly valuable to reduce the cost of urban freight transportation [Flamini *et al.* 2018].

Prediction as a service aims at predicting the travel time of each road segment, based on past measurements [Wu *et al.* 2016]. That way, a vehicle would use an alternative, faster route if congestion has been predicted along its primary route. Uncertainty may also be considered to compute a route with a given latest time of arrival [Lee *et al.* 2019]. While long-term prediction algorithms exist in the literature [Chen *et al.* 2019], short-term predictions are still challenging. Traffic jams are complicated to predict [Hu *et al.* 2017], and their impact on travel time depends on various local factors (e.g., speed limit, type of street, neighboring local roads).

While vehicle re-routing may reduce the impact of congestion, relying on real-time data can be computationally expensive. Computing a route, directly on a small embedded device is inconvenient as traffic data must be downloaded first, which may correspond to a large volume of data when the target location is geographically far away. Practically, this means that hundreds of weights have to be collected periodically, even if the target is in the same urban area. This represents a massive amount of data when considering a large collection of users. Alternatively, the route computation can be executed directly in the cloud: centralized servers reply in real-time to queries. Thereby continuously re-evaluating the route of each vehicle until it reaches its destination. Consequently, the load in the cloud is roughly proportional to the sampling rate used to refresh the real-time data. In particular, it becomes challenging to exploit route planning algorithms that rely on pre-processed data because of the continuous traffic changes measured on the network.

In this chapter, we quantify the benefit of using dynamic shortest path algorithms that re-route vehicles when congestion increases. Indeed, alternative paths may provide a faster route, but can only be identified when using real-time information. For this purpose, we exploit a real dataset comprising the travel times of the road segments of several cities (i.e., New-York, London, Chicago, and Cologne). To compare simulations with these real datasets, we also used the TAPAS (simulated) dataset [Uppoor *et al.* 2014]. The contributions of this chapter are as follows:

1. we quantify the travel-time gain when using real-time data to re-route vehicles, compared to a no re-routing approach that prohibits diverting from the path evaluated at the departure. Our dataset highlights that re-routing is very seldom required real conditions, even during rush hours;

2. we provide an optimal algorithm to compute a lower bound for the travel time between any pair of locations. The algorithm replays the real dataset to mimic ideal predictions. The gain of using ideal predictions is below 10% compared with a continuous re-routing solution, even for the worst route;

3. we compare the impact of re-routing based on two types of datasets (a real dataset vs. the simulated TAPAS dataset). Surprisingly, and contrary to current belief, we obtain very different behaviors when studying re-routing strategies. Simulations tend to exacerbate the randomness of travel time. Thus, simulations seem to not accurately capture the complexity of urban road network dynamics, proving the relevance of exploiting real datasets;

The remainder of this chapter is organized as follows. We present the methodology in section 4.1. We introduce our assumptions and model in section 4.1.1. Then, we describe four route planning strategies in section 4.1.2. In section 4.1.3, we explain our evaluation workflow for each routing strategy. We detail our datasets in section 4.1.4 and present a set of metrics to assess the performance of each routing strategy in section 4.1.5. Section 4.2 summarizes our experimental evaluation results with regard to each performance metric. We conclude and discuss future work in section 4.3.

4.1 Methodology

A route planning solution identifies a path to follow for a vehicle. Most solutions can be classified into:

1. **embedded devices** help to compute a route, sometimes after having downloaded real-time data about the congestion of the road network [Wang *et al.* 2015];

2. **cloud-based** infrastructures receive a collection of queries from the vehicles that they have to handle in real-time [Li *et al.* 2017b]. Continuously reconsidering the route because of real-time data is also expensive in a cloud serving a large number of customers, where additional resources have to be provisioned.

Each route planning strategy has to solve individual queries, returning a route with minimal travel time. Formally, we define a query as $q = (s, d, t_s) | s, d \in V$ and $t_s \in T$, where s, d and t_s represent the departure location, the destination location and the departure time respectively. Likewise, a route is defined as an ordered list of vertices (road segments) in the graph.

In this chapter, we consider the following strategies, ordered by the volume of resources (bandwidth and computation) they require:

1. **static:** we do not have any knowledge of the actual traffic conditions (i.e., no data is exchanged);

2. **no re-routing:** we know the travel time of each road segment precisely just before the vehicle leaves its starting point. For the sake of limiting computational cost, the vehicle does not reconsider its decision after departure;

3. **continuous re-routing:** we have continuous access to the most recent real-time data. A vehicle may be redirected to a different (shorter) route as soon as traffic conditions change;

4. **prediction based routes:** we are able to predict the traffic conditions perfectly. Consequently, we can identify the shortest ideal route, which constitutes our lower bound;

4.1.1 Assumptions and Model

A road network is defined by a list of road segments. Each of them consists of a set of consecutive coordinates (latitude and longitude) that define its shape. We use a dynamic directed graph $G_T = (V, E, W_T)$ to represent the road network, where $v \in V$ is a vertex representing a physical intersection of two or more road segments, $e_{ij} \in E$ a directed edge from vertex i to j, and $w_{ij}(t) \in W_T$ the weight assigned

Table 4.1: Keywords & symbols definitions.

Keywords & symbols	Definition
e_{ij}	Directed edge from vertex i to j
d_{ij}	Length $[m]$ of e_{ij}
$s_{ij}(t)$	Speed $[m/s]$ of e_{ij} at time t
$w_{ij}(t)$	Travel time $[s]$ of e_{ij} at time t
T	Number of timeslots in a given dataset
$\delta t = 60\ sec$	Sampling rate of the datasets
$\Delta t \geq \delta t$	Variable sampling rate
$r_i = \{u, v, .., w\}$	Defines a road segment
$R = q(s, d, t_s)$	Route in reply to the query from vertex s to d at timeslot t_s
$R = \{u, v, w, .., z\}$	A route is an ordered set of vertices
$TTime[q \text{ or } R]$	Travel time of a query q (or a route R)
$algo$	Designates one of four algorithms: static/no re-routing/continuous re-routing/ideal
$TTime[q, algo]$	Travel time of q using algorithm $algo$
$t \geq 0 \in \mathcal{R}$	Represents the time in seconds
$0 \leq t_k \leq T \in \mathcal{N}$	Timeslot index of a real-time data update

to e_{ij}, as a function of time $t \in T$. Table 5.1 contains definitions of the recurring keywords and symbols used throughout the chapter.

For the sake of clarification, we use the term *dynamic* to refer to a graph whose structure remains the same (no edges are deleted or inserted), but edge weights change over the period T. We consider road networks with traffic congestion, but we neglect the new roads that may appear.

We exploit a dataset where the speed of each road segment is monitored periodically and synchronously. Thus, we denote by *timeslot* the discretized time, during which the speeds for all the road segments remain unchanged. An edge weight $w_{ij}(t) = d_{ij}/s_{ij}(t)$ represents the travel time in seconds required to join the vertices i and j, where d_{ij} and $s_{ij}(t)$ represent the length of the edge e_{ij} and its corresponding speed at time t respectively.

A road segment $r_i = \{u, v, w, .., z\} \in V$ consists of an ordered list of vertices in the graph. When the speed on the road segment r_i changes, it affects the weights of all the edges associated with that road segment. Hence, in our implementation, every edge in the adjacency list points to a specific road segment in the road segments map.

4.1.2 Route Planning Strategies

We now describe in more detail the route computation algorithms we use in the rest of the chapter to evaluate the importance of real-time speed data along with road segments. Real-time information implies that vehicles may change their route when congestion occurs. Practically, drivers may be progressively aware of current incidents because they use different data sources [RafałKucharski & Gentile 2019]. We neglect here the mutual impact of their decisions, *i.e.*, travel-time may increase if all the drivers take the same decision. We model in the rest of this section different families of routing algorithms, taking into consideration traffic information.

Each strategy corresponds to a given travel time formulation. The here presented route planning strategies (*i.e.*, static, no re-routing, continuous re-routing, and ideal prediction based) respectively deal with travel times that are either time-independent (*i.e.*, no knowledge of traffic conditions), evolution-independent (*i.e.*, knowledge of traffic conditions at the initial computation time only), time-aware (*i.e.*, knowledge of initial traffic conditions and regular updates throughout the journey) or ideal (perfect forecast of traffic conditions).

4.1.2.1 Static Route Planning

We assume here the system has no knowledge about traffic conditions (e.g., an embedded device disconnected from the Internet). It represents our worst but thrifty strategy and provides a baseline for comparison. Thus, it will use the maximum speed for each road segment to compute the fastest route. The weights are time-independent, and hence w_{ij} represents the travel time from vertex i to j, where s_{ij} is the speed limit in this particular case. For a given query $q(s, d, t_s)$, the departure time t_s is meaningless since the algorithm would always return the same route for a departure from s toward the destination d. We use Dijkstra's algorithm [Dijkstra 1959] to compute the route with the shortest travel time.

4.1.2.2 No re-Routing Route Planning

The system has here a complete knowledge of the travel times but never reconsiders its decision. It mimics an embedded navigation system that is disconnected after departure or cloud infrastructure that executes the query only once. Thus, we apply the same strategy as previously, just executing Dijkstra's algorithm, with the travel time of each road segment at departure. This way, we can quantify the gain of using up-to-date information *before* departure.

4.1.2.3 Continuous re-Routing Route Planning

We continuously reconsider the routing decision by trying to compute a better route with shorter travel time, modeling the approach proposed by [Chen *et al.* 2010]. This strategy helps to bypass congested areas when they appear, but it also consumes more resources. If the computation is delocalized, the device has to retrieve all the

Algorithm 1: Fastest Route with continuous re-Routing.

```
Data: departure vertex ($s$), destination vertex ($d$), departure time ($t_s$),
      sampling rate ($\Delta t$)
Result: shortest route as an ordered list of vertices ($route$)
/* Set current position to vertex $s$ and current time to $t_s$ */
1  $here \leftarrow s$;
2  $t_{now} \leftarrow t_s$;
3  do
       /* updates the route, from here to the destination */
4      $route \leftarrow dijkstra(here, d, t_{now})$ ;
       /* true if a new data sample occurs */
5      $isNewSample \leftarrow false$;
       /* traverse the route until a new data sample occurs */
6      while $!isNewSample$ do
           /* next edge in shortest route */
7          $e_{vw} \leftarrow getNextEdge(route)$ ;
8          $lg \leftarrow getLength(e_{vw})$          /* get length of edge $e_{vw}$ */
9          while $lg > 0$ do
               /* maximum distance that can be covered by next speed change */
10             $lg_{max} \leftarrow speed(e_{vw}, t_{now}) * (\Delta t - t_{now} \mod \Delta t)$;
11             if $lg < lg_{max}$ then     /* crossroad reached before next sample */
12                 $lg \leftarrow 0$ ;
13                 $t_{now} \leftarrow t_{now} + \frac{lg}{getSpeed(e_{vw}, t_{now})}$;
14             else                        /* crossroad not reached before next sample */
15                 $isNewSample \leftarrow true$;
16                 $lg \leftarrow lg - lg_{max}$;
17                 $t_{now} \leftarrow t_{now} + \frac{lg_{max}}{getSpeed(e_{vw}, t_{now})}$;
18         $here \leftarrow getHeadVertex(e_{vw})$;
19 while $here \neq d$;
   /* we reached $d$ */
20 return $(route)$
```

travel times periodically for its area. In a cloud, this means that the query has to be re-executed continuously, consuming computational resources.

Let Δt be the sampling rate of the real-time traffic data. The route planning algorithm will re-compute the optimal route toward the destination at each crossroad. We use a dynamic version of Dijkstra's algorithm as detailed in algorithm 1:

1. we use a time-dependent graph model, where the weight of each edge is time-variant. When executing Dijkstra's algorithm, we pick the most recent weights;

2. the route is reconsidered when a new data sample occurs, i.e., it corresponds to an update of speed data. In that case, the shortest route to the destination from

the current position is computed (line 4);

3. we traverse the graph, following the current route (lines 7-17). We have to verify if we can reach the next crossroad before the next speed sample occurs (line 11);
4. if we cannot, we have also to consider the upcoming speed updates to reflect the actual travel time (line 14).
5. when a new sample occurs, we set the upcoming crossroad as the new position (line 18) from which we re-evaluate the route (step 1).

This version accommodates any sampling rate. This way, we can compare the impact of the data accuracy on the travel time.

4.1.2.4 Ideal Prediction Based Route Planning

As mentioned earlier, traffic congestion forecasting is a technique used by most advanced route planning algorithms. By incorporating predictions, one could predict recurrent traffic jams, and thus, plan the route accordingly at departure time.

We propose here to model such prediction-based routing algorithm [Liu 2017]. More precisely, our goal is to quantify the maximal benefit attainable when using a perfect forecast. Thus, we use an **ideal** prediction algorithm to compute the maximum gain achieved by **any** prediction-based routing algorithm. To do so, we depart a vehicle in the past by *replaying* the recorded measurements *a posteriori*.

We propose to use Algorithm 2:

1. we insert all vertices into a priority queue keeping the vertex with earliest arrival time at the head (line 2). The arrival time of the source vertex s is set to the departure time t_s. All other vertices are initially considered as unreachable and assigned an infinite arrival time.
2. at each iteration, we poll a vertex v from the queue and insert it into the settled set (line 5);
3. we compute the travel time required to reach each of its neighbors w (lines 7-18). In particular, we traverse each $edge_{vw}$, and we update its speed when a new sample occurs before reaching the next crossroad (lines 15-17);
4. we re-insert w into the queue with its corresponding parent vertex v and the updated arrival time t_w (line 18);
5. when the destination vertex is settled, we construct the route by browsing backward all the parent vertices starting at the destination until we reach the departure vertex (lines 20-23).

4.1.3 Evaluation Workflow

To fairly compare different route planning strategies, we need to compute enough routes using each strategy to cover most roads in the road network. To do so, we

Algorithm 2: Optimal Route with ideal predicion routing.

```
Data: departure vertex ($s$), destination vertex ($d$), departure time ($t_s$)
Result: shortest route as an ordered list of vertices ($route$)
/* settled is the set of vertices for which the shortest route was found */
1  $settled \leftarrow \{\}$;
/* priority queue contains triplets of the form (a vertex, its parent, its
   arrival time). The source's arrival time is set to $t_s$ while all other
   vertices are initially unreachable and with unknown parents */
2  $queue \leftarrow \{(s, s, t_s)\} \cup \{(v, \varnothing, \infty) | v \neq s \in V\}$;
3  $route \leftarrow \{d\}$; /* add destination to the route */
4  do
       /* extract vertex $v$ with smallest arrival time from the queue and insert
          it into the settled list with departure time $t_v$ */
5      $settled.add((v, u, t_v) \leftarrow queue.poll())$;
       /* iterate over all outgoing edges from v */
6      for $e \in getOutgoingEdges(v)$ do
7          $w \leftarrow getHeadVertex(e_{vw})$;
8          $t_w \leftarrow t_v$;  /* initialize arrival time at $w$ */
9          $lg \leftarrow getLength(e_{vw})$;  /* get length of $e_{vw}$ as $lg$ */
           /* compute travel time of edge $e_{vw}$ */
10         while $lg > 0$ do
               /* maximum distance which can be covered by next speed change */
11             $lg_{max} \leftarrow speed(e_{vw}, t_w) * (\Delta t - t_w \mod \Delta t)$;
12             if $lg < lg_{max}$ then  /* $w$ is reached before next sample */
13                 $lg \leftarrow 0$ ;
14                 $t_w \leftarrow t_w + \frac{lg}{getSpeed(e_{vw}, t_w)}$;
15             else  /* next sample occurs before reaching $w$ */
16                 $lg \leftarrow lg - lg_{max}$;
17                 $t_w \leftarrow t_w + \frac{lg_{max}}{getSpeed(e_{vw}, t_w)}$ ;
18         $queue.push((w, v, t_w))$;
19 while $d \notin settled$;
/* browse parent vertices starting at $d$ until $s$ is reached */
20 $parent \leftarrow getParentVertex(d)$;
21 while $parent \neq s$ do
22     $route.add(parent)$; /* insert into head of route */
23     $parent \leftarrow getParentVertex(parent)$;
24 return $(route)$
```

generate a massive number of queries and solve each of them with our four routing strategies. Hence, for a given route $R = q(s, d, t_s)$, by varying the departure time t_s, we can track the changes of R as a *vehicle* experiences congestion along the route

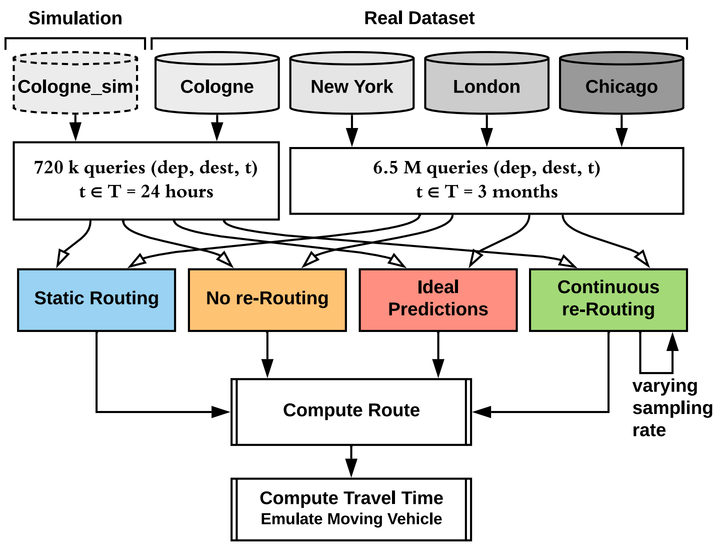


Figure 4.1: Workflow for the stretch factor quantification of not using real-time data.

from s to d, at different times of the day.

We insist on the fact that we do not simulate the *deployment* of multiple vehicles on the road network at once. Indeed, we do not have any means to incorporate the added congestion due to those vehicles, as one might expect in a mobility simulation tool. Instead, we consider each query as representing a *probe vehicle*, to measure the impact of the measured traffic (from our datasets) on its route (and not the inverse).

We apply here the following workflow to quantify the interest in exploiting real-time data (Figure 4.1):

1. we use real vs. simulated (TAPAS) datasets, and emulate different sampling rates (from 1 to 30 min) by subsampling the datasets;

2. we randomly select 1000 pairs of source and destination vertices in each road network;

3. for every (source, destination) pair, we generate a broad set of queries at different timestamps (every 1 min for the simulation and every 10 min on the real dataset);

4. for each query, we execute each route planning strategy to extract the route to follow;

5. we then emulate a vehicle moving along the route returned by the algorithm in the previous step. This way, we can accurately evaluate the actual end-to-end travel time for each strategy at different times of the day.

Finally, we use the previous results to compare the static, no re-routing, continuous re-routing and ideal prediction strategies, detailed in sections 4.1.2.1, 4.1.2.2,

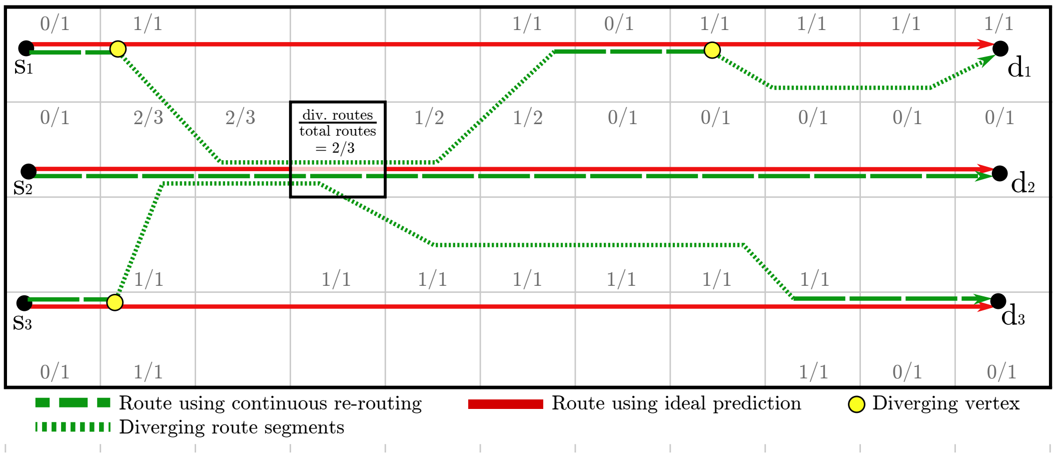


Figure 4.2: Diagram illustrating the process of computing the divergence ratio of a cell.

4.1.2.3 and 4.1.2.4 respectively.

We ran all our experiments on the High-Performance Computing (HPC) at the University of Strasbourg. We generated thousands of jobs that were executed on Intel Xeon Sandy-Bridge nodes with 16 cores and 64GO of RAM. The combined computation effort required approximately 50,000 CPU-hours.

4.1.4 Datasets

To evaluate the impact of the different route planning strategies, we use real travel times for a broad set of road segments. Additionally, we also use a simulated dataset for comparison purposes. We rely upon:

1. **the simulated TAPAS dataset:** [Uppoor *et al.* 2014] focuses on a small German City (Cologne), during a weekday (24 hours). It relies on the simulator SUMO to simulate the traffic from a large set of emulated vehicles, pseudorandomly selecting pairs of sources and destinations (e.g., home, office). We run the simulation using SUMO to generate a dump file specifying the speed along every road segment at 1-sec intervals. We subsample the dataset to get the same sampling rate used in the real dataset.

2. **a real dataset:** we use the dataset of three major cities (New York, London, and Chicago) for which HERE [HERE Technologies 2018] provides a fine-grain estimation of the speeds for the most important road segments. We collected three months (i.e., from September 21st to December 18th 2017) worth of data at a sampling rate of 1 min, which allows an accurate estimation of the actual travel time experienced by users. Additionally, we also use the city of Cologne, for the same geographical area and during the same time window as the TAPAS dataset for a fair comparison.

4.1.5 Metrics for the Performance Evaluation

We now detail the metrics we used to compare the different routing strategies.

4.1.5.1 Identification of congestion

To identify the rush hour period, we compute the Congestion Factor (CF) of each query as the ratio of the optimal travel time (using the ideal routing strategy) to the free-flow travel time. In the TAPAS dataset, the free-flow speed of a road segment corresponds to its speed limit. In the real dataset, the free-flow speed was provided as the average measured speed during low-volume periods and depends on the road characteristics such as lane width.

Hence, given a query $R = q(s, d, t_s)$, the congestion factor of the end-to-end route R is defined as:

$$CF[R] = \frac{TTime[R, ideal]}{TTime[R, freeflow]} \tag{4.1}$$

To analyze more clearly the behavior of each routing strategy, we need to classify the routes according to their congestion level. For this purpose, we define for each route its *traffic flow* (TF) metric, which represents the congestion of its most congested road segment. For each route $R = q(s, d, t_s)$, we measure the level of congestion of each road segment $r_i \in R$. More precisely, the congestion level corresponds to the relative speed decrease compared with free-flow conditions:

$$TF[r] = max\left(\frac{speed(r_i, t)}{speed_{ff}(r_i)}\right)_{r_i \in R} \tag{4.2}$$

where $speed(r_i, t)$ corresponds to the speed at time t for the road segment r_i, and $speed_{ff}(r_i)$ to its speed in ideal conditions (*i.e.*, free flow). $TF = 1$ corresponds to free-flow conditions, and $TF = 0$ corresponds to a complete halt of all vehicles on one of its road segments.

4.1.5.2 Travel Time Stretch and Gain Factors

We will use the *stretch factor* in travel time to compare the different strategies. The ideal strategy provides, by definition, the lowest end-to-end travel time and represents our lower bound. The stretch factor for a query $q(s, d, t_s)$ is defined as:

$$SF[q] = \frac{TTime[q, algo]}{TTime[q, ideal])} \tag{4.3}$$

where $TTime[q, algo]$ represents the end-to-end travel time for the route returned by the algorithm $algo$ (static/no re-routing/continuous re-routing/ideal) for the query q. Notice that $SF[q] \geq 1$, and hence, the higher the stretch factor gets, the worse is the performance of algorithm $algo$ compared to the ideal.

To specifically focus on the gain achieved by re-routing vehicles after their departure, we also compute the *gain factor*. It corresponds to the relative gain in travel time through the path selected by the prediction-based and the redirecting strategies compared with the static one (*i.e.*, without redirection after the departure).

4.1.5.3 Identification of divergences

We focus on the continuous re-routing strategy to precisely determine *where* a vehicle is practically re-routed because the congestion has changed. Practically, we identify the *divergence vertices*, *i.e.*, geographical locations where the next edge is different with or without rerouting. Formally, given two routes R_1 and R_2 that share at least the same first and last vertices, we define their diverging vertices as the set $\{j \in V \mid \exists i, k, w \in V \mid (e_{ij}, e_{jk}) \in R_1 \wedge (e_{ij}, e_{jw}) \in R_2 \wedge k \neq w\}$.

We construct a grid overlay where each cell is a 10×10 square meters. We consider a large number of route queries (pairs of sources/destinations). We execute the route computation for each query at different instants covering to cover the whole dataset. For each cell $cell_i$, we consider all the routes that cross this cell. We compute its *divergence ratio* ($0 \leq div[cell_i] \leq 1$) as the ratio of the number of diverging routes to the number of total routes. A route is diverging if the routes with and without the re-routing strategy are different for *at least one instant* of the dataset. Hence, a cell with a high divergence ratio means that more vehicles are re-routed when crossing this cell.

Some cells may be traversed by only a few routes, leading to statistically meaningless results. Thus, we only consider cells traversed by a significant number of routes (300 in this case).

4.2 Experimental Evaluation

In a road network with no congestion, all routing strategies should return optimal routes. Hence, our first goal is to distinguish the critical parts of the dataset by identifying the rush hours in each city. The remainder of the chapter will solely focus on evaluating the presented routing strategies during rush hours only.

We also provide a visual, interactive interface to showcase a sample of the NYC dataset and obtained results at `http://its-icube.com/`. Each query is represented individually, to visually represent the temporal characteristics of each route all along the week. In particular, we can identify the diverging vertices, as well as the evolution of the congestion.

4.2.1 Absolute Travel Time and Rush Hours

Figure 4.3 illustrates the *CF* in the Cologne TAPAS dataset. As expected, we identify the rush hours in the morning (6:00-9:00) and afternoon (15:00-19:00) of the working day. The optimal travel time is almost six times longer than the free-flow travel time during these two periods.

Figure 4.4 illustrates the congestion factor for the experimental datasets. We focused uniquely on Thursday, Friday, Saturday, and Sunday to provide readable charts (the remaining weekdays display similar characteristics to Thursday). We can derive the following observations:

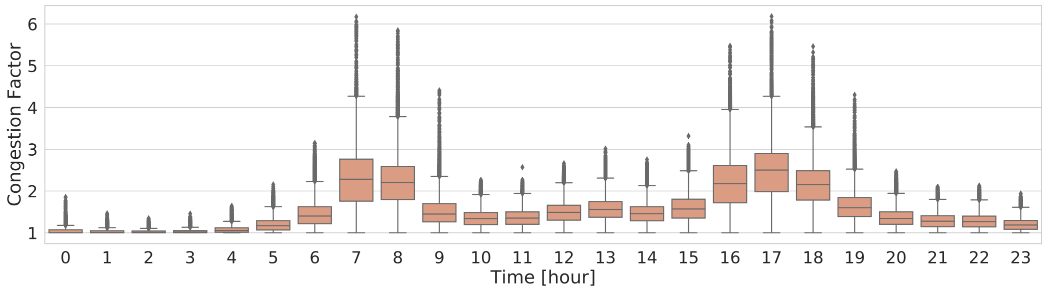


Figure 4.3: Congestion Factor distribution in Cologne simulation, Germany, over a period of 24 hours.

1. some queries benefit from travel times smaller than the free-flow. They correspond to very short distances (a few hundred meters) when streets are empty;
2. we observe the rise in congestion starting at 6:00 and intensifying in the afternoon with a peak at 17:00.
3. we can easily make a distinction between working days and weekends, which are much less congested.

Similar observations hold for London and Chicago.

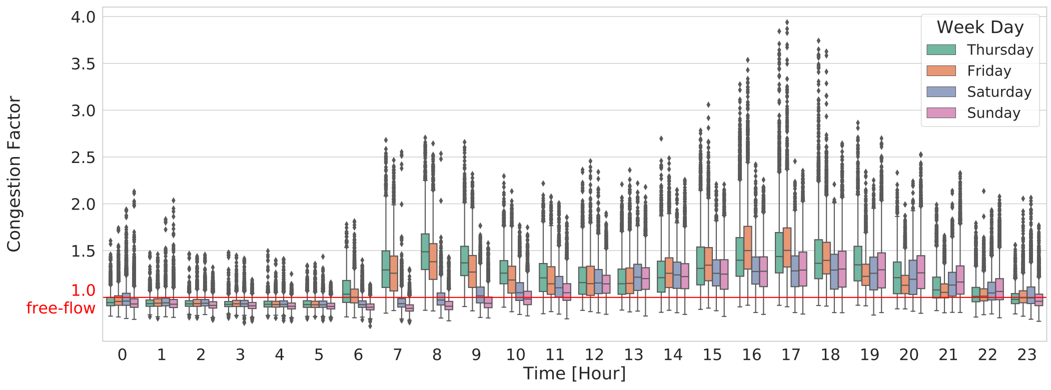


Figure 4.4: Congestion Factor distribution in New York, based on weekday and daytime over a period of 3 months.

4.2.2 Travel Time Stretch and Gain Factors

We now compare the achieved travel time for each routing strategy. In particular, Fig. 4.5 illustrates the stretch factor for the TAPAS dataset (left) vs. the real datasets (right). The stretch factor denotes the travel time increase compared with the shortest path, with an ideal dataset (*i.e.*, with ideal predictions). We clipped the plot at $SF = 3$, as it reaches ≈ 6 in the simulation.

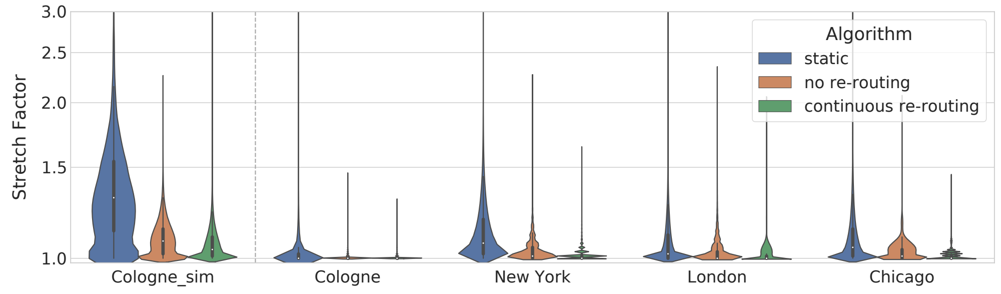


Figure 4.5: Stretch factor of the static, no re-routing and continuous re-routing algorithms on both the TAPAS (left) and the real (right) datasets.

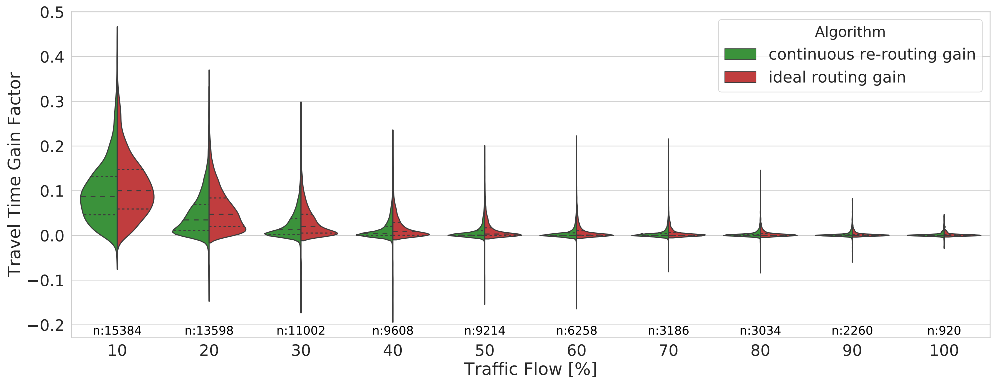


Figure 4.6: Travel time gain of continuous re-routing and ideal routing algorithms over the no re-routing algorithm as function of traffic flow in New York

The static routing strategy provides, as expected, the longest travel time. However, the distribution is significantly different between the simulation and the real dataset. With TAPAS, 28% of the queries have an SF greater than 1.5. In contrast, we only report 1.6% of the queries in Cologne and approximately 4% in New York, London, and Chicago. In Cologne, the SF using either the no re-routing or continuous re-routing strategies yields optimal travel times most of the time. Only ≈ 12% of the queries are characterized with a stretch factor greater than 1. In TAPAS, though, more than 75% of the queries have a $SF > 1$ even when using continuous re-routing. Moreover, for several queries in the simulation, travel time is worse when using continuous re-routing rather than no re-routing. The high variability of congestion profoundly affects re-routing in the simulation as a vehicle might engage in a seemingly faster alternative route only to become more congested than the initial route. In New York, London, and Chicago, approximately 75% of the queries benefit from ideal travel times when using continuous re-routing.

To further comprehend when continuous re-routing is advantageous, we pinpoint the travel time gain it offers (compared to no re-routing) based on the congestion level (Fig. 4.6). We normalize the travel time difference of using continuous re-routing (and ideal routing for reference) over the no re-routing strategy.

As expected, re-routing is relevant only when congestion occurs. The gain is, however, often negligible when the actual speed is close to 50% the speed limit. For very congested routes ($TF \leq 10\%$), we reduce the travel time significantly by re-routing the vehicle. Exploiting ideal predictions allows the gain to be even higher, to choose a better route: the best route is selected, before the congestion's increase.

Conclusion: continuous re-routing significantly reduces travel time for most queries. In New York City, we can reduce by 15% (1st quartile), which seems to justify the usage of a smart route planning strategy. However, predictions only marginally decrease the travel time for all the measured datasets. The traffic conditions seem to evolve smoothly, and redirecting the vehicle when the congestion occurs appears as a sufficient strategy. Clearly, continuous re-routing is essential to re-route around highly congested road segments. When the road network is less congested ($TF \geq 40\%$), the gain factor quickly converges to zero.

4.2.3 Impact of Sampling Rate on Travel Time

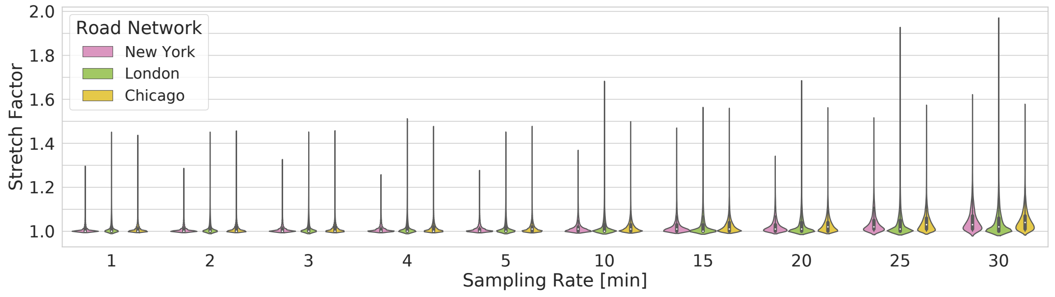
For the continuous re-routing algorithm, the sampling rate Δt impacts both travel and execution times. For fine-grained values, the algorithm is aware of the latest changes in traffic congestion and can re-route accordingly. However, it may also re-compute unnecessarily the routes, increasing execution time significantly. Our goal is thus to determine the right sampling rate for a good travel/execution time trade-off.

By varying the sampling rate Δt from 1 to 30 minutes, we re-compute the queries we generated using continuous re-routing. We only consider queries with a travel time $TTime[q] \geq 60min$ to make sure re-routing has potentially occurred for the largest sampling rate $\Delta t = 30$.

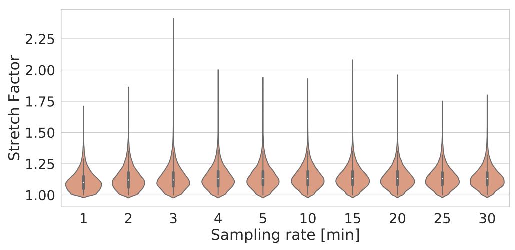
Figure 4.7a illustrates the distribution of the stretch factor as a function of the sampling rate. Obviously, a higher sampling rate means inaccurate speed values. Thus, the route planning strategy will take sub-optimal decisions. However, the re-routing strategy performs well, even for average sampling rates. Surprisingly, in New York, for instance, the road network variations can be efficiently handled even if measurements are reported every 10 minutes.

Figure 4.7c illustrates the execution time for New York. At $\Delta t = 10min$, the execution time drops from $\approx 500ms$ to less than $50ms$, which provides a good trade-off between travel time and execution time. Of course, there exists a myriad of algorithmic techniques in the literature, capable of significantly reducing the execution time. Considering that Δt dictates the number of times we have to re-compute the route, we are only interested here in the rate of the change of execution time as Δt increases.

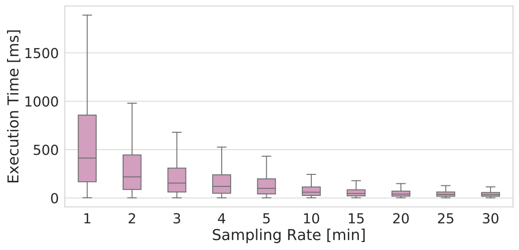
Figure 4.7b represents the execution time for the TAPAS dataset. In complete contradiction with the real dataset results, the stretch factor distribution is almost the same regardless of the sampling rate value. Again, we observe that congestion changes too fast throughout the whole road network (even at $\Delta t = 1min$), causing the algorithm to inevitably re-route through highly congested road segments.



(a) Travel time stretch factor in New York, London and Chicago



(b) Cologne simulation stretch factor



(c) New York execution time

Figure 4.7: Impact of sampling rate on travel time in Cologne simulation, New York and London (a, c and d) and execution time in Cologne simulation (b) using the continuous re-routing algorithm.

Conclusion: we can accommodate average sampling rates when using real-time data. Five minutes provide enough accuracy (for all our road networks) to identify the best routes while reducing the computational or bandwidth cost.

4.2.4 Route Divergence Patterns

Figure 4.8 illustrates the distribution of the diverging vertices in each road network, as defined in section 4.1.5.3. A divergence vertex corresponds to a crossroad where at least one vehicle has been re-routed to reduce travel time (*i.e.*, traffic congestion has changed since its departure). The divergence ratio counts the ratio of routes (source/destination) for which the path diverges when crossing the cell, with and without the re-routing strategy.

The road networks in the real dataset exhibit a small set of diverging vertices with a high divergence ratio, typically almost all diverging vertices have a divergence ratio ≤ 0.5. This means that only 50% of the routes that cross these cells are re-routed at least once during the whole duration of the dataset.

Inversely, the simulated dataset (with TAPAS) exhibits a significant number of diverging vertices. TAPAS generates pseudorandomly the traffic and estimates the level of congestion, and the speed for each road segment. It seems that TAPAS exhibits a very different pattern compared with the real datasets.

Figure 4.9 summarizes the obtained divergence patterns for both the TAPAS

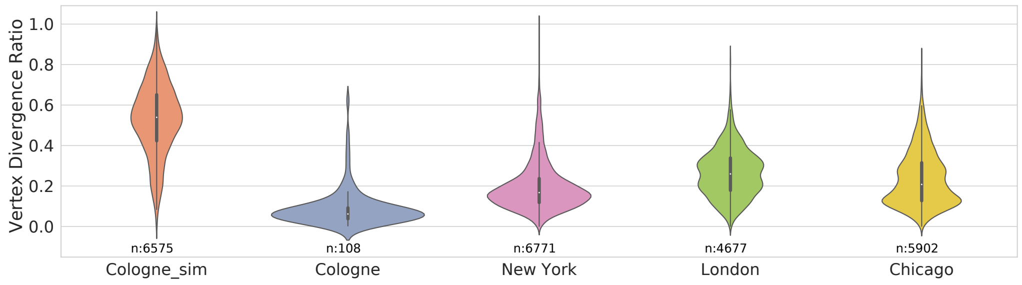
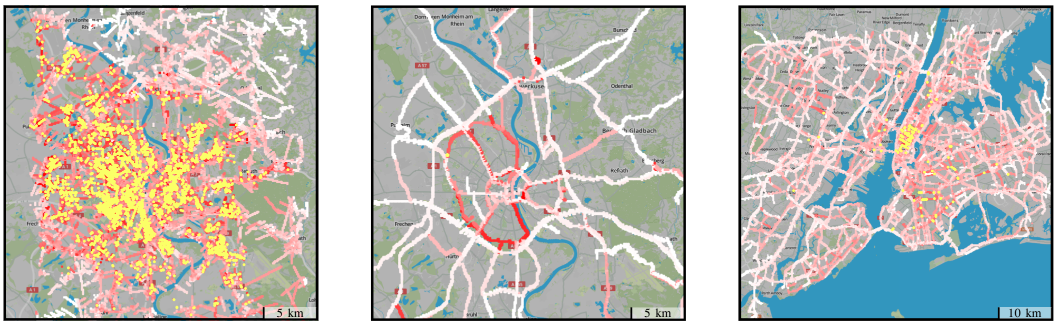


Figure 4.8: Diverging vertices divergence ratio distribution.



(a) Cologne simulation (b) Cologne (c) New York

Figure 4.9: Diplaying the divergence ratio level (proportional to the red shade intensity). The yellow dots depict diverging vertices with a divergence ratio ≥ 0.5.

and real datasets. We only represent here New-York City since other real datasets behave similarly. Each graph corresponds to a heat map (a red cell corresponds to a cell with a high divergence ratio). We also highlighted the divergence vertices with a divergence ratio ≥ 0.5 (yellow dots).

Remark: while we identify many cells with a high divergence ratio in all regions, we notice in the TAPAS dataset that the divergence seems present everywhere. We assume that this unrealistic behavior is a consequence of Gawron's traffic assignment algorithm [Gawron 1998] used to generate the dataset. In fact, the raw dataset required several adjustments [Uppoor *et al.* 2013] to *repair* inconsistent traffic behavior. It includes, for instance, adjusting the O/D matrix to only account for the estimated vehicular traffic in the region and adapting the rate of injected vehicles during the simulation to reduce excessive congestion. In practice, routing strategies seem more complex, and the trips seem to follow a uniform distribution.

Conclusion: many cells exhibit a large divergence ratio, and we cannot directly use this metric to trigger the route re-computation efficiently. However, we identified only a small number of diverging vertices. Thus, we would be able to execute the computation only when the route crosses these specific points, making the computation much more efficient. We would reduce the processing load without increasing the end-to-end travel time.

4.3 Conclusion

Modern intelligent systems guide vehicles through the fastest routes to avoid congested areas. However, they need to exploit real-time data, where the speed of each road segment has to be known precisely. Even worse, this real-time feature has a cost, in bandwidth (data collection), and computation (re-computation of the route to the destination). Interestingly, the no re-routing strategy provides close to ideal travel times most of the time. Using the continuous re-routing strategy can further improve travel time by avoiding very congested areas when they appear.

We also used a simulated dataset (TAPAS) that leads to very different results concerning the travel time and the re-rerouting gain. TAPAS seems not able to capture the road network characteristics accurately, and particularly its dynamics. This observation speaks in favor of working with real datasets to model realistic environments.

Addressing the problem of route planning in dynamic road networks is crucial for developing practical applications based on realistic features of the network. Nonetheless, most trips often combine several means of transportation such as cycling and public transit. In the following chapter, we extend the problem of route planning to multimodal networks.

Chapter 5

Multimodal Route Planning

Contents

Multimodal algorithms are of increasing importance for route planning to exploit the diversity of transportation infrastructure fully. The goal here is to compute a route that combines several modes of transportation according to predefined user constraints. Transportation modes that are typically considered are walking, cycling, driving, and public transit. We also consider rental vehicles and bicycles. A crucial aspect of multimodal routing is the route *feasibility*, that is, we must exclude impossible trips such as driving the private car between two bus rides. Efficient multimodal route planning a difficult task considering the large size of multimodal networks, the structural discrepencies between the road and public transit networks, the user modal constraints, and path feasibility.

Customizable Route Planning (CRP) [Delling *et al.* 2011a] is a road network algorithm based on multilevel separators. Initially, the graph is partitioned, aiming to minimize the number of boundary vertices. Then, an overlay is constructed by computing full-cliques across all boundary vertices within each cell in the partition. Finally, during a query, bidirectional-Dijkstra is run on the query graph combining the source and target cells and the overlay, allowing to skip most of the vertices in the underlying graph.

We detail here MUSE, a MUltimodal SEparators-based algorithm, extending CRP to handle the multimodal travel computation. By combining label constraints with a profile label correcting algorithm during preprocessing, we can handle modal constraints and time-dependency efficiently. Our main contributions are the following:

- We present a multimodal graph partitioning approach, with a label correcting algorithm to compute multimodal time-dependent cliques efficiently;
- We associate the multimodal graph to a labeled automaton, to reduce the number of vertices in the product graph. It reduces the memory footprint, and achieves faster preprocessing times;
- We experimentally evaluate our algorithm on a country-scale network with different heuristics for faster queries.
- We provide an open-source tool to construct multimodal networks and a unified dataset to serve as a benchmark for multimodal route planning algorithms.

We first detail in section 5.1 the model we used to represent a realistic multimodal network. In section 5.2, we explain each stage of our solution. Mainly, we discuss partitioning in section 5.2.1, the process of computing a multimodal overlay in section 5.2.2, and the algorithm and heuristics used to answer queries in section 5.2.3. The experimental setup, the results, and associated discussion are available in section 5.3. Finally, in section 5.4, we detail our conclusion and possible future work.

5.1 Model and Assumptions

Transportation networks are usually modeled with *graph* structures for their intuitiveness and the extensive algorithmic toolbox of graph theory [Thomson & Richardson 1995]. Mainly, we model a multimodal network using a labeled directed graph with time-dependent edge costs. It consists of multiple *layers* of unimodal networks that are interconnected via *link* edges.

We detail in sections 5.1.1 through 5.1.4 each unimodal network followed by the process of computing link edges in section 5.1.5 to obtain the multimodal graph. For better readability, we summarize recurring notation in table 5.1. Following are definitions consistently used throughout the chapter:

A directed Graph $G(V, E)$ consists of a set of vertices $v \in V$, and directed edges $(v, w) \in E$ connecting vertices $v, w \in V$. A vertex is an abstraction of a physical entity in the network: an intersection of road segments in the road network or a station in the public transit network. Throughout the chapter, all graphs are considered directed unless otherwise specified.

The Edge Cost Function $c(v, w, \tau)$ (also written $f_{vw}(\tau)$ in the literature) represents the time required to travel when departing at time τ from vertex v to w,

Table 5.1: keywords and symbol definitions

notation	**definition**
$G(V, E)$ or G	Directed graph with V vertices and E edges.
$G(V, E, \Sigma)$ or G^{Σ}	Directed graph, labeled with alphabet set Σ.
$(v, w) \in E$	Directed edge from vertex v to w.
$c(v, w)$	The cost, represents travel time $[s]$ of (v, w).
$c(v, w, \tau)$	Time dependent cost of (v, w) at time τ.
$len(v, w)$	Physical length $[m]$ of (v, w).
$lab(v, w)$	Label attached to (v, w).
$P = \{v_0, v_1, .., v_k\}$ or $P_{v_0 v_k}$	A path is an ordered set of vertices $v_i \in V(G)$.
$c(P, \tau)$	The cost of path P when departing at time τ.
$d(r, t, \tau)$	The cost of the shortest path P_{rt} departing at τ.
$word(P) = \cup_{i=0}^{k-1} lab(v_i, v_{i+1})$	The sequence of edge labels associated to path P.
$\mathcal{A} = \{S, \Sigma, \delta, s_0, F\}$	Non-deterministic Finite Automaton (NFA) consists of a set of states S and alphabet Σ, a transition function δ, an initial state s_0, and a set of final states F. See section 5.2.2.
$G(V^{\times}, E^{\times})$ or $G^{\times} = G^{\Sigma} \times G^{\mathcal{A}}$	Product graph merging graph G^{Σ} and graph automaton $G^{\mathcal{A}}$.
$\langle v, s\rangle \in V^{\times}$	Product vertex is a pair of vertex $v \in V(G^{\Sigma})$ and a state $s \in S(\mathcal{A})$.
$\big(\langle v, s\rangle, \langle w, s'\rangle\big) \in E^{\times}$	Product edge, requires a valid transition $s' \in \delta(s, lab(v, w))$.

which is referred to as the travel-time. The cost is a periodic positive piece-wise linear function $f : \Pi \to \mathbb{R}^+$ where $\Pi = [0, p] \subset \mathbb{R}$ with a period $p \in \mathbb{N}$. If f is constant, then the edge belongs to a time-independent network such as the foot or bicycle networks; otherwise, the edge is said to be time-dependent. In public transportation, the period is typically one week. In any case, the *FIFO* property is maintained to ensure polynomial complexity [Kaufman & Smith 1993]. Also known as the non-overtaking property, it holds that for all $\tau_1, \tau_2 \in \Pi$ such that $\tau_1 \leq \tau_2$ then $\tau_1 + f(\tau_1) \leq \tau_2 + f(\tau_2)$. In other words, waiting at a vertex never pays-off.

A Path $P = \{v_0, v_1, .., v_k\}$, also written $P_{v_0 v_k}$, is an ordered sequence of vertices $v_i \in V$. Its associated cost is recursively evaluated with the edge cost of its edges by $c(v_0, v_1, \tau) + c\big(\{v_1, .., v_k\}, \tau + c(v_0, v_1, \tau)\big)$ when departing from v_0 at time τ. For a *query* $q(r, t, \tau)$ with $r, t \in V$, our goal is to compute the *shortest path* P with the smallest cost denoted by $d(r, t, \tau)$.

5.1.1 Road Network (private cars, taxis, and rental vehicles

Structurally, road networks consist of intersecting road segments. Each segment is characterized by its length and a specific speed-limit, which can be used to derive the cost function. In its graph representation $G(V, E)$, each edge $(v, w) \in E$ represents a road segment, and the vertices $v, w \in V$ mark the junction of two or more road segments.

For a realistic model, though, dynamic traffic conditions must be taken into account as speed can unpredictably vary. Thus, we rely on speed measurements to construct the cost function. For each segment, we collect a set of speed values over a time window Π sampled at a fine-grained rate Δt (we detail our dataset in the experimental evaluation of section 5.3). Then, for each edge (v, w) we construct its speed profile as a piece-wise linear function f_{vw}. During query time, we compute the cost $c(v, w, \tau_1)$ of departing from v at time τ_1 by evaluating the area under the speed-curve, adjusting the arrival time τ_2 such that: $\int_{\tau_1}^{\tau_2} f_{vw} dt = len(v, w)$ where $len(v, w)$ is the length of the edge. This is a trivial geometric computation considering the function is piecewise linear. Solving the integral for τ_2, travel-time is given by $c(v, w, \tau_1) = \tau_2 - \tau_1$.

Moreover, getting onto or off of a car is only permissible where parking is possible. Hence, for each vertex belonging to a road segment where parking is authorized (typically excludes highways, tunnels, bridges, and sidewalks), we label it as an eligible parking spot $v \in V^{\text{park}} \subset V$. Furthermore, instead of restricting the road network to private driving only, we consider alternative options, including rental vehicles and on-demand services such as Uber and taxis. While on-demand vehicles are typically accessible at every vertex $v \in V^{\text{park}}$ (we discuss the additional incurred waiting cost in section 5.3), rental vehicles are only available at rental stations. Thus, we add a vertex $v \in V^{\text{rent}} \subset V$ for each rental station and insert an edge $(v, w) \in E$ to connect the station to the closest junction w in the road network.

5.1.2 Foot Network

The foot network is represented by a time-independent graph $G(V, E)$. Vertices V represent junctions and edges E are added for each footpath including sidewalks, bridges, and stairs. The cost of an edge (v, w) is thereby a constant given by $c(v, w) = len(v, w)/S_{walk}$ where S_{walk} represents the average pedestrian walking speed.

5.1.3 Bicycle Network

Similar to a foot network, the bicycle network is based on a time-independent graph $G(V, E)$ where vertices V represent junctions and edges E depict either cycling lanes or road segments where biking is allowed (typically non-motorway road segments). The cost $c(v, w) = len(v, w)/S_{bicycle}$ is evaluated based on average cycling speed $S_{bicycle}$.

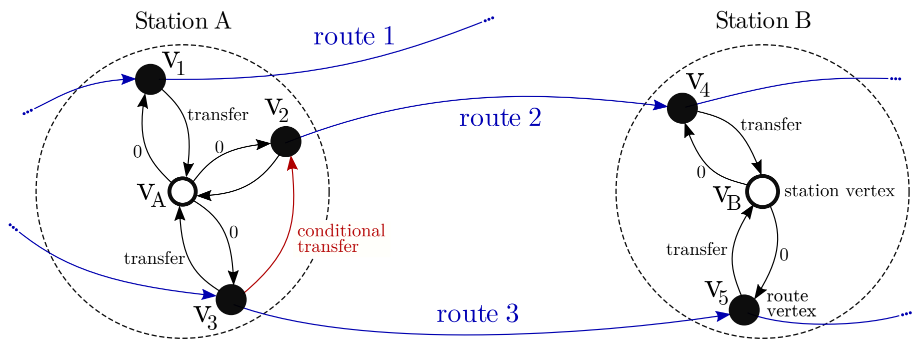


Figure 5.1: Time-dependent graph representing the public transit network

In addition to private bicycles though, bicycle-sharing systems are very common in urban cities and are efficient for fast transfers between nearby public transit stations. Rental bicycles must, however, often be picked up and returned at specific locations (bicycle stations). Thus, we add a vertex $v \in V^{\text{rent}} \in V$ for each rental station and an edge (v, w) between the station and its closest junction in the bicycle network.

5.1.4 Public Transit Network

The public transit network is based on a timetable $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$ which consists of a set of shuttle vehicles $\mathcal{Z}$, a set of stations $\mathcal{S}$, and a set of *elementary connections* $\mathcal{C}$. An elementary connection is a 5-tuple $c = \{Z, S_d, S_a, t_d, t_a\} \in \mathcal{C}$ and represents a unique shuttle $Z \in \mathcal{Z}$ departing from a station $S_d \in \mathcal{S}$ at time t_d and arriving at another station $S_a \in \mathcal{S}$ at time t_a without a stop at any intermediate station.

We define a *trip* as a sequence of elementary connections $\{c_0, c_1, .., c_k\}$ fulfilled by a single shuttle such that $\forall i = 1, 2, .., k$ if $c_i = \{Z(i), S_d(i), S_a(i), t_d(i), t_a(i)\}$ then $Z(i) = Z(i-1)$ and $S_d(i) = S_a(i-1)$. A trip typically denotes the itinerary of a single vehicle scheduled at a particular time. We group multiple trips into a single *route*, which is time-independent and denotes an ordered sequence of stations. To obtain the set of routes $\mathcal{R}$, we iterate over all trips and extract for each of them, a route $r = \{S_0, S_1, .., S_k\}$ that we add to $\mathcal{R}$ *iff* $r \notin \mathcal{R}$. We denote by $R_S \subset \mathcal{R}$ the subset of routes passing through station $S \in \mathcal{S}$, that is $\forall r \in R_S, S \in r$.

We have to model the transfers between different trips and routes, to construct a realistic time-dependent graph $G(V, E)$ from the timetable $\mathcal{T}$. We proceed as follows:

- $\forall s \in \mathcal{S}$ we add a *station vertex* $v_s \in V_S \subset V$. Let us denote by $f_{sta}(\mathcal{S}) \rightarrow V_S$ the bijective function connecting each station $s \in \mathcal{S}$ to its *station vertex* $v_s \in V_S$.

- $\forall s \in \mathcal{S}$, $\forall r \in R_S$, we add a *route vertex* $v_r \in V_S$, modeling the *platform* in the station for the corresponding route. Let us denote by $f_{route} : (\mathcal{S}, R_S) \rightarrow V_S$ the bijective function connecting each pair of station and route to its *route vertex*.

- $\forall s \in \mathcal{S}$, we add *conditional transfer edges* (v_{r_i}, v_{r_j}) between route vertices $v_{r_i}, v_{r_j} \in V_S$ of the same station. Their cost is the transfer time (walking time) from one platform to another;

- $\forall s \in \mathcal{S}, \forall r \in R_S$, we add a *transfer edge* $(f_{sta}(s), f_{route}(s)) \in E$ between the route vertex and its corresponding station vertex. If the transfer time is not known exactly, we cannot create *conditional transfer edges.* Thus, we compute the average walking time to go to a central point in the station, denoting a fixed transfer cost $c(f_{route}(s), f_{sta}(s))$ toward any other platform. As $c(f_{route}(s), f_{sta}(s))$ includes the full transfer cost, we add an edge $(f_{sta}(s), f_{route}(s))$ with $c(f_{sta}(s), f_{route}(s)) = 0$ to preserve the connectivity: we consider that the transfer time has already been considered when the passenger arrives at the station vertex.

  These transfer edges are also used when a passenger exits the transit network.

- $\forall c \in \mathcal{C}$, we add a *connection edge* representing one "edge" of a route (served by a collection of shuttles). A *connection edge* (v_{r_i}, v_{r_j}) is inserted between the route vertices $v_{r_i} \in V_I$ and $v_{r_j} \in V_J$ where $I = S_d(c)$ and $J = S_a(c)$ (a shuttle leaves $S_d(c)$ and arrives at $S_a(c)$). Its cost $c(v_{r_i}, v_{r_j}, \tau) = t_a(c) - t_d(c)$ is time dependent, and depends on the timetable.

The resulting graph is illustrated in figure 5.1. A hypothetical journey may begin at station A at $\tau = $ 8:30. There, we wait for the next train departing from the second platform (vertex v_2) towards station B via route 2. The cost function of edge (v_2, v_4) denoted $f_{v_2 v_4}$ is depicted in figure 5.2. Hence, the total cost for the traversal of (v_2, v_4) at $t = \tau$ is $c(v_2, v_4, \tau) = f_{v_2 v_4}(t = \text{8:30}) = 135 min$, comprising both the waiting and travel time. Because the slope $df_{v_2 v_4}(t)/dt = -1$, waiting time is given by the elapsed time between arrival and the next departure (8:30 $\rightarrow$ 9:45). Upon arriving to station B at 10:45, we disembark on the platform denoted by vertex v_4 and proceed to transfer to another train traveling along route 3, costing us $c\big(v_4, v_B, \tau + c(v_2, v_4, \tau)\big)$ additional time.

5.1.5 Assembling the Multi-modal Network

The multimodal network combines all of the road, foot, bicycle, and public transit networks within a single data structure: a labeled directed graph $G(V, E, \Sigma)$. To distinguish the networks, we attach a unique *label* $\sigma \in \Sigma = \{c, f, b, p\}$ to each edge, where c, f, b, and p stand for *car*, *foot*, *bicycle*, and *public* respectively. Let us denote by $G_\sigma(V_\sigma, E_\sigma)$ the uni-modal graph labeled $\sigma \in \Sigma$. The vertex-set of the multi-modal graph G is given by $V = \cup_{\sigma \in \Sigma} V_\sigma$ and its edge-set $E = \cup_{\sigma \in \Sigma} E_\sigma \cup E^{\text{link}}$ where E^{link} contains the set of *link* edges allowing modal changes in G, such as taking a bus after a short walk, by linking the different uni-modal networks together.

Intuitively, any transition from one network to another should be mediated via the foot network, as *some* walking is usually required for any modal change. Depending on the network however, transitions to and from the foot network are only

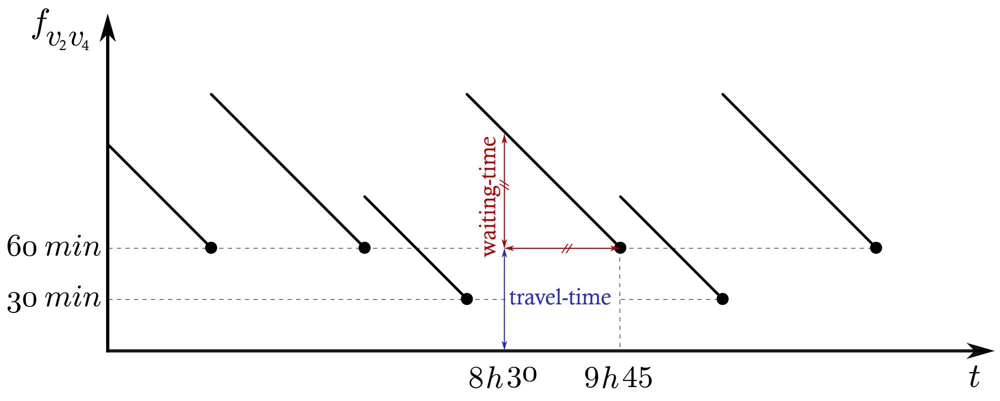


Figure 5.2: Piecewise linear function describing the cost of edge (v_2, v_4) from figure 5.1. Six trains are scheduled: four regular trains of 60 min travel-time and two fast trains of 30 min travel-time. Upon arriving to the station at 8:30, the next shuttle is a regular train scheduled to depart at 9:45.

allowed at specific locations, and thus, we select a subset of *link* vertices $V_\sigma^{\text{link}} \subseteq V_\sigma$ from each graph G_σ:

Foot $\leftrightarrow$ Road: The road network is accessible everywhere a car is allowed to park. Furthermore, rental vehicles are accessible at rental stations and. Thus, all parking spots and rental stations are marked as link vertices $V_c{}^{\text{link}} = V_c{}^{\text{park}} \cup V_c{}^{\text{rent}}$.

Foot $\leftrightarrow$ Bicycle: Considering that bicycles can be used almost everywhere walking is possible (except on stairways, for instance), every vertex $v \in V_b$ is a link vertex from which we can access the foot network. Hence, $V_b{}^{\text{link}} = V_b \cup V_b{}^{\text{rent}}$ which includes rental stations as well.

Foot $\leftrightarrow$ Public Transit: Public transit stations are accessible via station vertices. Hence, $\forall v_S \in V_p$ where v_S is a station vertex, we label it as a link vertex and add it to $V_p{}^{\text{link}}$.

Then, for each vertex $v \in V_{\sigma\in\Sigma}^{\text{link}}$, we must compute the closest vertex $w \in V_f$ in the foot network and add the link edges $(v, w), (w, v) \in E^{\text{link}}$. Additional labels are added to Σ as we label link edges according to the type of transfer:

- link edges $(v, w) | v \in V_f$ and $w \in V_c{}^{\text{rent}}$ denote transfers to car-rental stations, and are labeled with the label c_r.
- link edges (v, w), $(w, v) | v \in V_f$ and $w \in V_b{}^{\text{rent}}$ are labeled with $lab(v, w) = b_r$ and $lab(w, v) = b_s$ which imply *renting* and *restoring* the bicycle respectively.
- the remaining link edges (v, w) are labeled $lab(v, w) = t$ denoting a regular modal transfer.

Relying on a brute force approach to compute link edges is costly: we have to scan the whole foot network to identify the closest foot-vertex for each vertex in the bicycle network, leading to a quadratic complexity of $\mathcal{O}(V_b \times V_f)$. A better approach relies on clustering the foot vertices using a 2d-tree [Bhatia & Others 2010] based on latitude and longitude, which is a suitable data structure for solving the *nearest neighbor* problem in logarithmic time.

It is worth noting that this preprocessing is executed once, even if the traffic congestion evolves later: multimodal links are not time-dependent. Thus, the cost of a link edge $c(v, w)$ is fixed and depends on the type of transfer. It includes the required walking-time to transfer to or from the foot network and an additional cost to consider either parking-time, processing at a rental station, or for instance, the time it takes to secure a bicycle. Nonetheless, we must also ensure path *feasibility*. That is, if the private car (bicycle) is left behind at some point during the trip in favor of using the bus, we would not be able to use our private car (bicycle) again. Similarly, a scenario in which the private car is used after taking a train is not valid. However, the road (bicycle) network remains accessible via other means such as a taxi or from a rental station. Such constraints are not embedded within the graph but rather, are dealt with using an automaton, as detailed in the upcoming section.

5.2 MUSE: The Algorithm

We detail here MUSE, a speedup technique to *Regular Language Constrained Dijkstra* (D_{RegLC}) based on graph separators, to solve the multimodal shortest path problem. By dividing the multimodal graph G^{Σ} into multiple smaller regions, we can precompute an overlay graph H significantly smaller than G^{Σ} and use it to compute queries much faster. MUSE runs in three stages: (i) *partitioning the graph*, (ii) *computing the overlay*, and (iii) *solving queries*. The first two stages belong to the preprocessing phase.

In the first stage of the preprocessing, we partition the graph G^{Σ} to split it into k cells $\{C_0, C_1, .., C_k\}$ so that we minimize the average number of boundary vertices per cell (section 5.2.1). Regardless of the time-dependent nature of G^{Σ}, partitioning is run only once, as it only depends on the topology of the graph. In the second stage of preprocessing, we compute for each cell a *full clique*, essentially adding a directed edge for each pair of boundary vertices (section 5.2.2). It is worth noting that the full clique has to be recomputed when the weights change. The overlay graph consists of all cliques and *cut edges*, i.e., edges connecting two different cells. The cliques are constrained by an automaton, that defines the sequences of modes that are allowed by the user. Furthermore, considering that the graph contains time-dependent edges, the weight of clique edges can also, possibly, be time-dependent. Therefore, we run *profile queries* to compute the cost function of each clique edge. Even though profile queries are costly, cliques are independent, and the computations are parallelized, allowing to compute the whole overlay H for large instances in a few minutes only.

During the query phase, we compute the shortest path $P = q(r, t, \tau)$ by running

D_{RegLC} on the query graph $G^q = G_r \cup H \cup G_t$ which consists of the subgraphs G_r and G_t, induced by the cells containing the source and target vertices r and t, respectively and the overlay H. We can improve query times with a goal-directed version of the algorithm called MUSE*. Better speed ups are possible using heuristics (MUSE^{SV}) with a tradeoff on correctness (section 5.2.3).

5.2.1 Stage 1: Partitioning The Graph

Planar graphs (i.e., graphs that can be drawn on a flat surface without any intersecting edges) belong to a class of sparse graphs with valuable topological properties. Most importantly, planar graphs can be partitioned in linear time with small separators [Djidjev 1982].

Given an undirected graph $G(V, E)$, a partition on G is a collection $\{C_1, C_2, .., C_k\}$ where each element $C_i \subseteq V$ is referred to as a *cell*. Partitioning breaks apart the vertex-set V into k cells $V = \cup_{i=1}^{k} C_i$ with no overlap $C_i \cap C_j = \varnothing | i \neq j$. Most importantly, the goal is to compute a partition such that the number of boundary vertices is minimized. An edge (v, w) is a *cut edge* if both v and w are boundary vertices belonging to two different cells.

Road networks, although not planar (due to overpasses and tunnels), can also be efficiently partitioned [Eppstein & Goodrich 2008, Sanders & Schulz 2012]. In a multimodal graph, the vast majority of the vertices belong to the road network. In fact, most foot and bicycle vertices are duplicates of road vertices (think of sidewalks along a road segment, for instance); therefore, partitioning remains viable.

Computing ideal partitions is, however, NP-hard [Garey & Johnson 2002]; thus, we recourse to approximations that have exhibited good results for transportation networks. METIS [Karypis & Kumar 1998] is a Multilevel Graph Partitioning algorithm that runs in three stages, as depicted in figure 5.3. The first, *coarsening* stage, aims at reducing the size of the graph by repeatedly merging neighboring vertices and collapsing edges. Each iteration i, produces a coarser graph G_i than its predecessor G_{i-1}. The second stage, *partitioning*, is run on the coarsest graph. Finally, during the *uncoarsening and refining* stage, the graph is expanded at each iteration i, and the partition is refined until the graph reaches its initial size.

Setup: our objective is to minimize the overall number of boundary vertices, regardless of the number of incoming and outgoing cut edges at each cell. Therefore, we transform the directed labeled graph G^Σ into an undirected graph $G(V, E)$ with no labels. Furthermore, considering that the public transit network is modeled as a time-dependent graph, the partitioning process might break apart the route vertices of a single station across multiple cells. To avoid this, we contract all the substations to a single vertex $v_s \in V$, corresponding to a station. We also keep an undirected edge $(v_s, v_{s'}) \in E$ if there exists at least one elementary connection between any substation of v_s and any substation of v'_s.

Coarsening is an iterative process where pairs of vertices are contracted together to reduce the size of the graph. At each iteration i we obtain a coarser graph

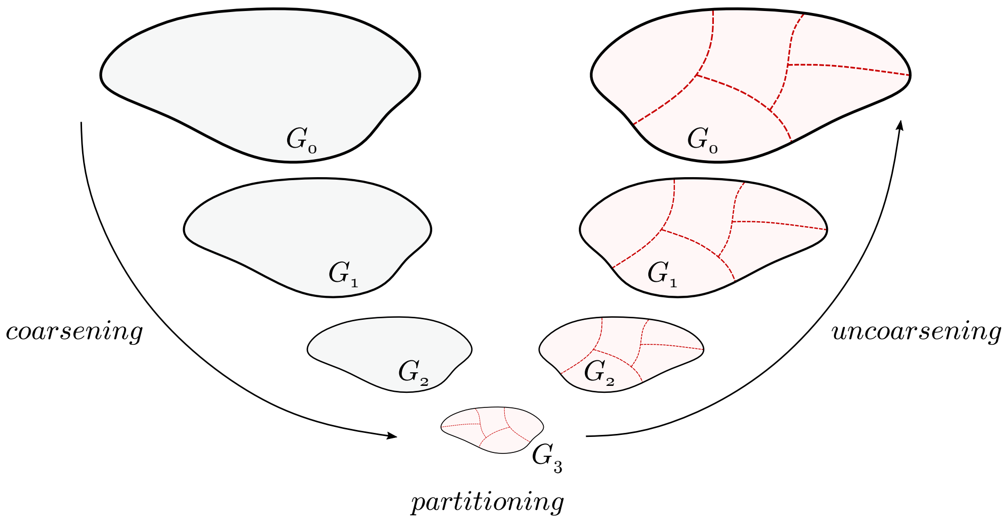


Figure 5.3: Multilevel graph partitioning.

$G_i(V_i, E_i)$, so that $|V_i| < |V_{i-1}|$. Initially, $\forall v \in G$ we attribute a size label $s(v) = 1$. Contracting two vertices v, w means replacing them with a new vertex x such that $s(x) = s(v) + s(w)$. To contract multiple vertices simultaneously, a set of edges $M \subset E_i$ is selected, such that no two edges share the same vertex, also known as a *matching*. Thus, at each iteration, we compute the maximal matching, i.e., a matching such that no additional edge can be added, and contract all vertices $v, w|(v, w) \in M$. Coarsening is halted when the size of the graph is sufficiently small. Most importantly, we must ensure that $|V_i| \geq k$ where k represents the number of desired cells in the partition.

Partitioning: the coarsest graph $G_c(V_c, E_c)$ can be partitioned using Breadth-First-Search (BFS) starting from a random vertex $v \in V_c$ and growing a tree $T \subset V_c$ until $|T| \simeq 1/2|V_c|$. To obtain k partitions, the initial partitions are then recursively partitioned $log_2(k)$ times. [Kernighan & Lin 1970] use a heuristic to polish the partitioning by exchanging vertices between cells producing a smaller cut-size.

Uncoarsening and Refining: at each iteration i, a less coarse graph G_i is obtained by expanding G_{i-1}. For all vertices $x \in V_{i-1}$, we expand x to obtain vertices $v, w \in V_i$, and assign them to the same cell x belonged to. Kernighan Lin's heuristic is then run again to refine the partition. After the last iteration, we expand the station vertices to retrieve the public transit network and transform the undirected partitioned graph G back to the multimodal graph G^{Σ}.

We obtain a partitioned multimodal graph, as illustrated in figure 5.4, where each layer corresponds to a specific type of transportation. The cells of the partition are, therefore, multimodal cells with boundary vertices located in different layers.

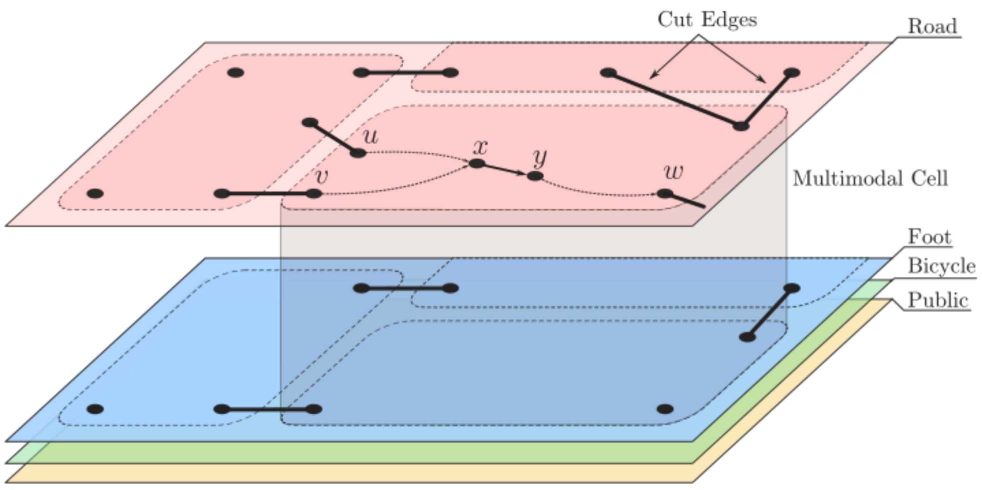


Figure 5.4: Multimodal graph partition with cut edges shown in bold segments. The multimodal cell spans across all unimodal networks. The vertices u, v, and w are boundary vertices and the shortest paths P_{uw} and P_{vw} share the subpath $\{x, y, ..w\}$.

5.2.2 Stage 2: Computing The Overlay

The partition produces a set of cells, where each cell C_i contains a set of boundary vertices $V_i^b \subset C_i$. The overlay graph $H(V^H, E^H)$ of G^Σ has the vertex-set $V^H = \cup_{i=1}^k V_i^b$ which consists of all boundary vertices. The goal is to compute for each pair of boundary vertices $v, w \in V_i^b$ belonging to the same cell C_i, a shortcut edge (v, w) denoting the shortest path P_{vw}.

Computing all shortcuts within a cell produces a *clique*. After computing all cliques, the edge-set of the overlay $E^H = E_{cut} \cup_{i=1}^k (v, w) \mid v, w \in V_i^b$ consists of all edge cuts E_{cut} together with the clique edges. This explains why we minimize the number of boundary vertices when partitioning the graph. Computing a clique edge (v, w) requires, however, solving the two following problems:

- The shortest path P_{vw} between v and w is multimodal. Therefore, we must solve the Label Constrained Shortest Path Problem [Barrett *et al.* 2000] (LCSPP) to restrict the modal sequence of P_{vw}.

- Edges can be time-dependent, hence, the cost of P_{vw} varies based on departure time. Therefore, we must compute the *profile* of (v, w) denoting its associated cost for all departure times.

We solve both problems simultaneously using a label correcting algorithm, which produces a Constrained Profile Clique (CPC) for each cell in the partition. We first detail each problem separately, then delve into the details of the algorithm.

Solving the LCSPP: Using formal languages [Harrison 1978], we can define modal constraints using *regular expressions* to prune prohibited edge transitions while computing clique edges. In language theory, an *alphabet* Σ is a finite set of letters $\sigma \in \Sigma$. A *word*, is thereby any sequence of letters over Σ and the collection of all possible words is the set Σ^*. A *language* $L \subseteq \Sigma^*$ is a subset of words that comply to a set of specific rules and is considered *regular* if there exists a regular expression R whose language $\mathcal{L}(R) = L$.

Definition 1 *Regular Expression. A regular expression R is a set of algebraic rules denoting a regular language $\mathcal{L}(R)$. Given an alphabet Σ:*

1. *the empty set $\varnothing$ is a regular expression.*
2. *any letter $\sigma \in \Sigma$ is a regular expression.*
3. *if R_1 and R_2 are regular expressions, then $(R_1 \cup R_2)$, $(R_1.R_2)$, and $(R_1)^*$ are also regular expressions, where $\cup$, $.$, and * denote the or, and, and kleene operators, respectively.*

Moreover, any regular expression can be written in the form of a non-deterministic finite state automaton (NFA) [Brüggemann-Klein 1993]. Formally, an NFA is a 5-tuple $\mathcal{A} = \{S, \Sigma, \delta, s_0, F\}$ which consists of a finite number of states S, an alphabet Σ, a transition function $\delta : S \times \Sigma \to 2^S$, an initial state s_0, and a set of accepting states $F \subseteq S$. Therefore, given a directed labeled graph G^Σ and an NFA $\mathcal{A}$, MUSE solves the LCSPP by computing a path in G^Σ such that (i) its cost is minimum and (ii) the *word* obtained by concatenating its edge labels (sequence of modes) is accepted by $\mathcal{A}$ (*i.e.*, it corresponds to an acceptable sequence of modes).

Conveniently, $\mathcal{A}$ can be implemented as a directed labeled graph $G^\mathcal{A}$. Thus, we can solve the LCSPP with D_{RegLC} [Barrett *et al.* 2000], a regular language constrained Dijkstra algorithm, deployed on the *product graph* $G(V^\times, E^\times)$ where $V^\times$ are vertices and $E^\times$ edges. The product graph $G^\times = G^\Sigma \times G^\mathcal{A}$ is a composition of the underlying graph G^Σ and the automaton graph $G^\mathcal{A}$. A product vertex $\langle v, s\rangle \in V^\times$ is a pair of a vertex $v \in V(G^\Sigma)$ and a state $s \in S(\mathcal{A})$. A product edge $(\langle v, s_i\rangle, \langle w, s_j\rangle) \in E^\times$ is added *iff* there exists an edge $(v, w) \in E(G^\Sigma)$ such that $s_i \times lab(v, w) \to s_j$ is a valid transition of $\delta \in A$.

Computing a shortest path profile (D_p): In this section we use the term *distance label* of a vertex v, written $l(v)$, to designate the tentative cost to reach v from a source vertex r (not to be confused with multimodal labels). In a unimodal time-dependent graph G, we call D_p a *label correcting* version of Dijkstra's algorithm [Orda & Rom 1990] allowing to compute the *profile* function $d_*(r, t)$ denoting the minimum travel time of the shortest path P_{rt} for all departure times τ. Because of time-dependency, the algorithm does not necessarily settle a vertex v after it is extracted from the queue. Rather, it propagates its profile $d_*(r, v)$ to its neighbors and potentially, re-inserts v into the queue (label correction) if $d_*(r, v)$ is improved for some departure time τ.

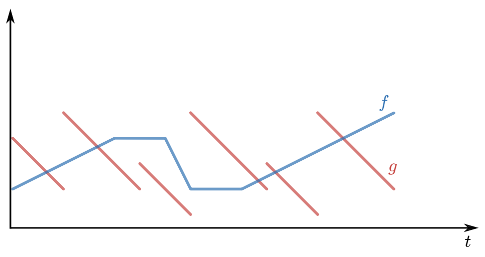
In that context, performing edge relaxations requires additional operations to manipulate functions, rather than scalars. Therefore, we define the following operations:

- **evaluate:** given a piece-wise linear function f and a time τ, evaluation returns $f(\tau)$ in $\mathcal{O}(log|f|)$.
- **link:** given two edges (v, w) and (w, x) with cost functions f and g, respectively, linking returns the cost function $f * g = f + g \circ (f + \tau)$ of the path (v, w, x) with complexity $\mathcal{O}(|f| + |g|)$. The operator $\circ$ designates the composition of two functions.
- **merge:** given two parallel edges (v, w) and $(v, w)'$, with cost functions f and g, respectively, merging returns the cost function $h = min(f, g)$, with minimal travel time from u to v for any time τ. That is, $merge(f, g) = min\{f(\tau), g(\tau) \mid \forall \tau \in \Pi\}$ with complexity $\mathcal{O}(|f| + |g|)$.

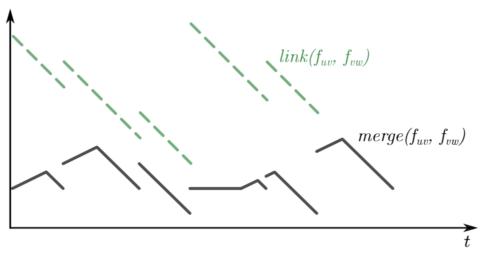
Performing the merge operation, in particular, is computationally expensive because we have to compute all segment intersections of the input functions f and g. This is done using a line sweep algorithm [Bentley & Ottmann 1979] over the breakpoints of f and g. Computing intersections is required because the merged functions are not necessarily homogeneous, that is, the slope of the piecewise linear functions f and g can be different, rendering the merge operation non-trivial. Figure 5.5 illustrates the outcome of the link and merge operations. In this example, we are merging (and linking) the cost function f of a road network edge with the cost function g of a public transit edge. As illustrated, the merged function in figure 5.5b contains breakpoints belonging to either f or g (for instance the first and last breakpoint) but also breakpoints corresponding to the segment-intersections of f and g. Furthermore, the resulting function has more breakpoints than either f or g, and thus, memory requirements become a concern after successive merge-operations.

Nonetheless, merging can be avoided if one of the functions dominates the other; that is, if $\underline{f} \geq \overline{g}$, then g is kept as the result of the merge. Similarly, explicit linking can be avoided if either function is constant. If both f and g are constant, linking is a trivial addition. If only one function is constant, we distinguish two cases: if f is constant, we simply translate all g segments upwards by the constant. If otherwise, g is constant, we additionally translate all f segments left by the constant. Only if f and g are non-constant functions, we scan their breakpoints and perform the link operation.

Algorithm 3 details the steps of D_p, which is very analogous to the classic label setting Dijkstra algorithm with a few adaptations to handle vertex labels which are piecewise linear functions. At each iteration, the vertex v with the smallest key $key(v) = \underline{d_*}(r, v)$ (*i.e.*, it has the highest priority) is extracted from the queue (line 4) and all of its neighboring vertices are scanned (line 7). For each outgoing edge (v, w), we compute the distance label $l(w) = d_*(r, v) * f_{vw}$ (link operation) and check if it improves $d_*(r, w)$ for some departure time. In that case, we update the distance label, with the merge operation: $d_*(r, w) = merge(d_*(r, w), l(w))$ and compute $key(w) = \underline{d_*}(r, w)$ (line 8-10). If w is scanned for the first time, we insert it into the queue; otherwise, we just update its key inside the queue (11-15). This whole process is repeated until a vertex v is extracted from the queue such that



(a) cost functions f_{uv} of a road network edge, and f_{vw} of a public transit edge.



(b) the resulting functions after linking or merging f_{uv} and f_{vw}.

Figure 5.5: Illustrating the outcome of the linking and merging non-homogeneous edge cost functions f and g.

$\underline{d_*}(r,v) > \overline{d_*}(r,t)$ holds (line 5). Therefore, the target's distance label cannot be improved anymore, the algorithm is stopped, and $d_*(r,t)$ is correct. Similarly, for a set of targets T, the same condition must hold for each $t \in T$ for the algorithm to stop.

Algorithm 3: D_p (Profile Dijkstra)

```
Input  : graph $G(V,E)$, source vertex $r$, set of target vertices $T$
Output: profile function $d_*(r,t)$ for all $t \in T$
1  $d_*(r,v) \equiv \infty \mid \forall v \in V$ and $scanned(v)$ = false;
2  $d_*(r,r) \equiv 0$, set $key(r) = 0$ and insert $r$ into priority queue $Q$;
3  while $Q$ not empty do
4      extract vertex $v$ with smallest key from $Q$;
5      if $\underline{d_*}(r,v) > \overline{d_*}(r,t) \forall t \in T$ then // stop when $v$ cannot improve any target $t \in T$
6          return
7      foreach outgoing edge $(v,w)$ do
8          if $d_*(r,v) * f_{vw} < d_*(r,w)$ then // $(u,w)$ yields an improvement for some departure $\tau$
9              $d_*(r,w) = merge(d_*(r,w),\ d_*(r,v) * f_{vw})$; // update profile of $w$
10             $key(w) = \underline{d_*}(r,v)$; // set key to global minimum of the profile function
11         if not $scanned(w)$ then
12             insert $w$ into $Q$;
13             $scanned(w) = true$;
14         else
15             update $key(w)$ inside $Q$;
16     end
17 end
```

Constrained Profile Clique Algorithm (lcD_{pRegLC}): Given a cell C_i, we have to compute the shortest path profiles from each boundary vertex $v \in V_i^b$ toward all other boundary vertices. Since G^Σ is a labeled graph, we must also ensure that all shortest paths abide by the regular language of automaton $\mathcal{A}$. Let us call D_{pRegLC} the regular language constrained version of D_p, which computes profiles of constrained shortest paths. Hence, one approach is to run D_{pRegLC}, $|V_i^b| \times |S|$ times, from each product vertex $\langle v, s \rangle$ where $v \in V_i^b$ and $s \in S(\mathcal{A})$. In other words, the algorithm would compute a path from any boundary vertex, the path being possibly different for any state of the NFA. However, this approach is wasteful in terms of memory usage since a vertex v cannot necessarily be combined to any state s of the NFA.

Consider $\mathcal{A}_2$, the automaton shown in figure 5.6b. A vertex v whose label $lab(v) = r$ (corresponds to the road network) is not compliant with the state s_2 that corresponds to the public transit network, and that is only attainable from either itself or s_1 (the foot network). Therefore, we design the automaton such that each state is attributed a unique label $\{f, r, b, p\}$ designating a specific type of network. Hence, the product vertex $\langle v, s \rangle$ is *valid* only if $lab(v) = lab(s)$. This way, we obtain a smaller set $V_i^{\times b}$ of boundary vertices in the product graph.

Moreover, instead of running D_{pRegLC} a number of $|V_i^{\times b}|$ times (once for each valid product boundary vertex), we are better off using a *multiple source* approach to compute the whole clique in a single run. To understand the appeal of using a multiple-source approach over multiple calls to single-source shortest path algorithm, consider the diagram of figure 5.4. Vertices u, v, and w are boundary vertices belonging to the same cell and the edge (x, y) is common to both shortest paths P_{uw} and P_{vw}. With a multiple source approach, each vertex is attributed one distance label for each source vertex. Hence, when scanning vertex y, we can update both $d_*(u, y)$ and $d_*(v, y)$ at once. This approach is favored when the set of source vertices are close to one another, increasing the likelihood of overlapping shortest paths. Thereby, particularly effective to compute our constrained cliques considering that in practice, boundary vertices are tightly packed, outlining the boundary of the cell they belong to, as illustrated in figure 5.11e.

We call our multiple-source algorithm lcD_{pRegLC}, which stands for *label-correcting Regular Language Constrained profile Dijkstra*. As shown in Algorithm 4, lcD_{pRegLC} takes as input the multimodal graph G^Σ, the automaton $\mathcal{A}$ and a set of source vertices R. Since we are manipulating product vertices, the distance label of vertex $\langle v, s' \rangle$ with respect to a source vertex $\langle r, s \rangle$ is written $d_*^{r,s}(v, s')$.

The first block (line 1-5) initializes, for each source vertex $\langle r, s \rangle$, its own distance label to $d_*^{r,s}(r, s) \equiv 0$ and that of all other vertices $d_*^{r,s}(v, s') \equiv \infty$ (with respect to the source $\langle r, s \rangle$). Then, all source vertices are added to the queue. At each iteration, the vertex $\langle v, s' \rangle$ with the smallest key is extracted from the queue (line 7). Then, for each outgoing product edge $(\langle v, s' \rangle, \langle w, s'' \rangle)$ (line 8-9), we check if the distance label of $\langle w, s'' \rangle$ can be improved with respect to each source vertex $\langle r, s \rangle$. That is, if $d_*^{r,s}(v, s') * f_{vw} < d_*^{r,s}(w, s'')$ holds for any departure time τ, then the

Algorithm 4: *lcD_{pRegLC}* (label correcting D_{pRegLC})

```
Input  : Graph $G(V, E, \Sigma)$, Automaton $\mathcal{A}(S, \Sigma, \delta, s_0, F)$, and Source set
         $R \subseteq V$
Output: Constrained shortest path profiles $d_*^{r,s}(v, s') \mid \forall\langle r, s\rangle \in R \times S$ and
         $\forall\langle v, s'\rangle \in V \times S$

1  foreach source $\langle r, s\rangle \in R \times S$ do
2      $d_*^{r,s}(r, s) \equiv 0$;                      // set distance label of $\langle r, s\rangle$ to 0
3      $d_*^{r,s}(v, s') \equiv \infty \mid \forall\langle v, s'\rangle \neq \langle r, s\rangle$;   // set distance label of all other
        vertices to infinity
4      set $key(r, s) = 0$ and add $\langle r, s\rangle$ to priority queue $Q$;
5  end
6  while $Q$ not empty do
7      extract product vertex $\langle v, s'\rangle$ with smallest key from $Q$;
8      foreach outgoing edge $(v, w)$ do
9          foreach transition $s' \times lab(v, w) \rightarrow s''$ do
10             $updated = false$;
11             foreach source $\langle r, s\rangle$ do  // compute distance labels of $\langle w, s''\rangle$ for
                each source $\langle r, s\rangle$
12                 if $d_*^{r,s}(v, s') * f_{vw} < d_*^{r,s}(w, s'')$ then      // relax product edge
                    $(\langle v, s'\rangle, \langle w, s''\rangle)$
13                     $d_*^{r,s}(w, s'') = merge(d_*^{r,s}(w, s''),\ d_*^{r,s}(v, s') * f_{vw})$;
14                     $updated = true$;
15             end
16             if $updated$ then
                   // set key to the global minimum among all distance labels of
                   $\langle w, s''\rangle$
17                 $key(w, s'') = min\{\underline{d_*^{r,s}}(w, s'') \mid \forall\langle r, s\rangle \in R \times S\}$;
18                 add $\langle w, s'\rangle$ to $Q$ if not in $Q$, otherwise update its key;
19         end
20     end
21 end
```

edge is relaxed (line 13). In that case, we update the key $key(w, s'')$ to the smallest $\underline{d_*^{r,s}}(w, s'')$ among all source vertices $\langle r, s\rangle$ and add $\langle w, s''\rangle$ to the queue (line 16-19).

To compute the clique of a cell C_i, we define the set of source vertices as the boundary vertices $R = V_i^b$. We also define a set of targets $T = V_i^b$ and stop the algorithm when the next extracted vertex $\langle v, s'\rangle$ respects rule $key(v, s') > \overline{d_*^{r,s}}(t, s'')$ for all targets $\langle t, s''\rangle \in T \times S$. This means that we cannot find a shorter path for the most distant boundary vertex.

5.2.3 Stage 3: Computing Queries

Algorithm 5: MUSE (query algorithm)

```
Input  : graph $G(V, E, \Sigma)$, partition $\{C_1, C_2, .., C_k\}$, overlay $H$, NFA
         $\mathcal{A} = \{S, \Sigma, \delta, s_0, F\}$, source $r \in V$, target $t \in V$, departure time $\tau$
Output: cost of shortest path $P_{rt}(\tau)$
1  Let $G^q(V^q, E^q) = C_r \cup H \cup C_t$; // $G^q$ is the query graph and $C_r$ and $C_t$ are the source and target subgraphs induced by their respective cells
2  $d(v, s) = \infty \mid \forall \langle v, s \rangle \in V^q \times S$; // set distance label of all vertices to infinity
3  $d(r, s_0) = 0$; // set distance label of source vertex to 0
4  add source $\langle r, s_0 \rangle$ to priority queue $Q$
5  while $Q$ not empty do
6      extract product vertex $\langle v, s \rangle$ with smallest key $d(v, s)$ from $Q$;
7      if $v = t$ and $s \in F(\mathcal{A})$ then
8          return $d(v, s)$; // $v$ is the target and $s$ is a final state
9      foreach outgoing edge $(v, w) \in E^q$ do
10         foreach transition $s \times lab(v, w) \rightarrow s'$ do
11             if $d(v, s) + f_{vw}(\tau + d(v, s)) < d(w, s')$ then
12                 $d(w, s') = d(v, s) + f_{vw}(\tau + d(v, s))$; // relax the product edge $(\langle v, s \rangle, \langle w, s' \rangle)$
13                 $key(w, s') = d(w, s') + \pi(w)$;
14                 add $\langle w, s' \rangle$ to $Q$ if not in $Q$, otherwise update its key;
15         end
16     end
17 end
```

During the online phase, MUSE solves a query $q(r, t, \tau)$ by running D_{RegLC} on the query graph $G^q = G_r \cup H \cup G_t$ which consists of the overlay H (precomputed in the previous stage) and the subgraphs G_r and G_t, induced by the cells containing the source and target vertices r and t, respectively. As G^q is significantly smaller than the original multimodal graph, MUSE computes the shortest path $P_{rt}(\tau)$ fast enough for interactive queries. Algorithm 5 details the flow of execution.

Nonetheless, knowing the location of the target allows for making informed decisions while searching for the shortest path. Instead of "blindly" expanding from the source, the intuition behind goal direction is to use heuristics to guide the search by prioritizing vertices that are closer to the target. We call MUSE* (pronounced 'MUSE-star') the goal directed version of the algorithm (analogous to $A^\star$ [Appi 1966]) which uses Euclidean distances to compute the *potential function*. The potential function $\pi(v)$ of a vertex v represents an estimate of the remaining cost to reach the target vertex t. To guarantee correctness however, the potential function must be admissible, which means that it must underestimates the true cost, that is, $\pi(v) \leq cost(P_{vt})$ where P_{vt} denotes the shortest path between v and t.

Another variant called MUSE^{SV} applies the same strategy but relies on the *Sedgewick-Vitter* heuristic [Sedgewick & Vitter 1984], which trades off correctness for the sake of speed. The potential function is evaluated as $\pi(v) = \alpha \times d_{Euc}(v,t) \,/\, Speed_{max}$ where $d_{Euc}(v,t)$ denotes the Euclidean distance between v and the target t, $Speed_{max}$ the highest speed in the network and α a tuning parameter. Setting $\alpha > 1$ may potentially overestimate the cost of the shortest path, which impacts correctness, but results in a smaller search space overall.

5.3 Experimental Evaluation

We report in this section, the performance evaluation of MUSE and its associated heuristic-based variants, namely MUSE* and MUSE^{SV}.

5.3.1 Evaluation Setup

To assess scalability, we run all algorithms on two graph instances: the Ile-de-France region denoted G_{idf} and a country-size graph G_{fr} representing France. Table 5.2 summarizes the graph characteristics based on each transportation layer. The graphs were modeled by combining:

1. a topological dataset from **OpenstreetMap** [Mooney *et al.* 2017], an open-access dataset built through the effort of crowd-sensing and accessible via the GeoFabric [Geofabrik 2019] online platform. We use it to construct the road, cycling, and pedestrian graphs. It consists of latitude and longitude coordinates denoting roads, parking spots, and even bicycle and car rental service stations.

2. a *General Transit Feed Specification* (**GTFS**) dataset represents the standard format used to encode public transit schedule information. For the Ile-de-France graph instance, we rely on the *Ile-de-France mobilités* (IdFm) dataset [de France Mobilités 2020], an online platform providing up-to-date data in GTFS format combining train, RER, subway, tramways, and bus networks in the region. For the public transit network of France, we combine four GTFS datasets covering the whole country and obtained via the open-source platform Navitia [Data 2020].

To model common use case multimodal trips, we define several automata that are depicted in figure 5.6:

$\mathcal{A}_1$ depicts the combination of walking and all types of public transportation.

$\mathcal{A}_2$ depicts trips that rely on the private car only, for the whole trip;

$\mathcal{A}_3$ extends $\mathcal{A}_1$ with faster transfers, using rental bicycles, mostly available in the city center at specific locations. An additional state s_2 is reachable via link edges labeled l_b to either retrieve or return the rental bicycle before pursuing the trip;

Table 5.2: Graph characteristics. For the road and bicycle networks, the stations column designates the number of rental stations.

Transportation Network	Vertices		edges		stations	
	G_{idf}	G_{fr}	G_{idf}	G_{fr}	G_{idf}	G_{fr}
Foot (G_f)	519 558	8 048 695	1 363 995	20 157 922	-	-
Road (G_r)	457 406	7 252 489	1 018 814	16 712 036	249	899
Bicycle (G_b)	406 711	7 354 519	1 000 591	18 292 964	800	3 324
Public Transit (G_p)	60 959	255 872	351 551	1 089 911	18 836	251 685
Foot-Road edges (E_r^{link})	-	-	44 270	431 470	-	-
Foot-Bicycle edges (E_b^{link})	-	-	813 422	14 665 980	-	-
Foot-Public edges (E_p^{link})	-	-	37 672	192 886	-	-
Multimodal (G^{Σ})	1 444 634	22 911 575	4 630 315	71 543 169	-	-

$\mathcal{A}_4$ excludes the public transit network. It allows using the private bicycle initially followed by a rental car for the remaining part of the trip;

$\mathcal{A}_5$ combines all means of transportation. The private car is used only initially, followed by any combination of walking, public transportation, and rental bicycles.

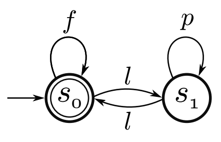
We ran all our experiments on the High-Performance Computing (HPC) of the University of Strasbourg. We generated a unique *job* for each partition and automaton that was executed on an Intel Haswell node totaling 24 cores and 32 GB of RAM.

5.3.2 Preprocessing

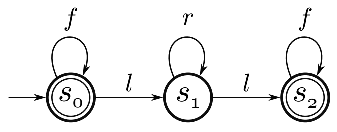
Graph partitioning results are reported in table 5.3. We tested various partition sizes ranging from 100 - 500 cells for G_{idf} and 800 - 6 000 cells for G_{fr}. The size of the partition, i.e., the number of cells, impacts both preprocessing and query times. Increasing the partition size yields smaller cells with fewer border vertices but produces a larger overlay graph, which slows the queries. In contrast, a smaller partitioning results in large cells, which increases the preprocessing times. Consequently, selecting the *adequate* partition is a tradeoff between preprocessing and query times.

Once the graph is partitioned, the cells are independent of each other and therefore preprocessing benefits from parallelism. For instance, figure 5.11c depicts a partitioning of the G_{fr} graph into 100 cells. As illustrated, the number of nodes of each cell varies according to the density of the graph. Given a cell with N border product vertices, we evaluate two strategies to compute its clique:

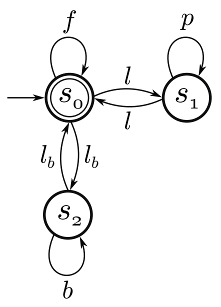
One-to-Many Strategy: runs a profile regular language constrained Dijkstra D_{pRegLC} algorithm from each border product vertex. The algorithm is run N times to compute the clique.



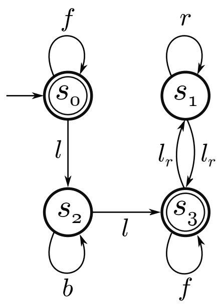
(a) $\mathcal{A}_1$: walking and public transportation.



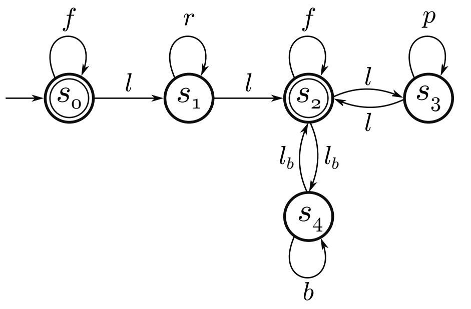
(b) $\mathcal{A}_2$: walking and private car.



(c) $\mathcal{A}_3$: walking and public transportation with the possibility of using rental bicycles.



(d) $\mathcal{A}_4$: walking and private bicycle followed by a rental car for the last part of the trip.



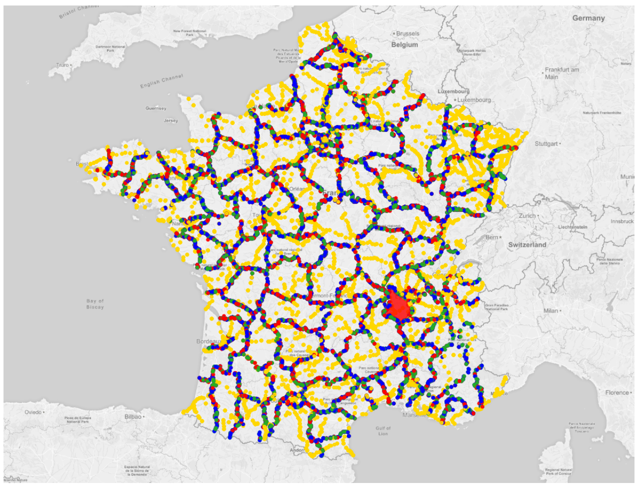
(e) $\mathcal{A}_5$: the private car is used for the first part of the trip, followed by any combination of walking, public transportation and rental bicycles.

Figure 5.6: Set of Automata depicting four scenarios used to constrain preprocessing and queries.

Many-to-Many Strategy: runs lcD_{pRegLC} only once. It simultaneously computes all the shortest path profiles of the clique in a single call.

Given that the density of the graph is not uniform, preprocessing time varies significantly (few milliseconds to seconds) from one cell to another within the same partition. Using the one-to-many strategy, we make the computation in parallel for each border vertex, *i.e.*, we do not need to allocate all the border vertices of a cell to the same CPU. However, the one-to-many strategy does not leverage the fact that the shortest paths of a clique are likely to share common subpaths. Therefore, vertices that are shared by several shortest paths are processed several times.

In contrast, the many-to-many strategy is more efficient as it simultaneously processes all the shortest paths of the clique. However, it prohibits load balancing


(a) France multimodal network partition

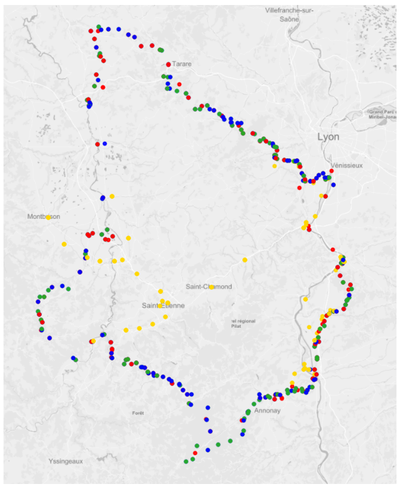

(b) boundary vertices of a single cell

Figure 5.7: A multimodal partition of France separating the underlying graph into 100 cells. (a) Boundary vertices belonging to the foot, bicycle, road and public transit networks are displayed in blue, green, red, and yellow dots, respectively. (b) boundary vertices of the area highlighted in red in figure (a)

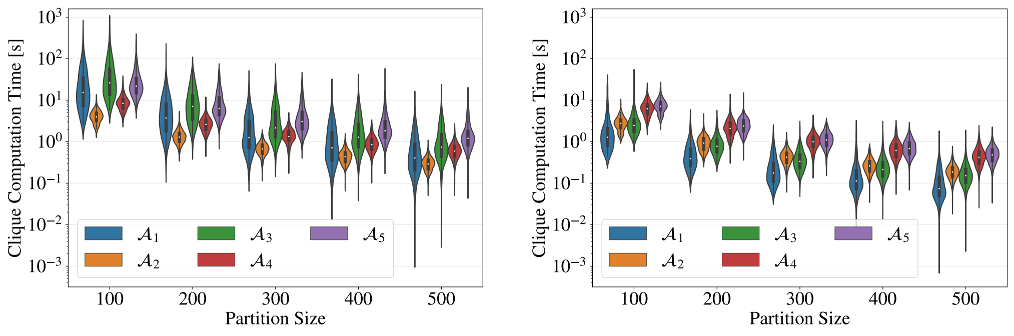
because it allocates a whole cell to a single CPU.

Figure 5.8 depicts the clique computation time distribution in the Ile-de-France region using the one-to-many (left) and many-to-many (right) strategies for each partition and automaton. As expected, preprocessing time decreases when the partition size gets larger. Additionally, larger automata require more processing time (for instance, $\mathcal{A}_5$ compared to $\mathcal{A}_1$) because it increases the size of the product graph, in addition to computing one shortest path for each available state. An interesting result is that lcD_{pRegLC} is an order of magnitude faster compared to running D_{pRegLC} multiple times. This confirms our previous hypothesis and shows that within a cell, there is indeed a significant number of shortest paths of the clique that do share common subpaths.

For a clearer comparison, we compute the *Preprocessing Gain Factor* (PGF) for each clique, denoting the ratio of computation time using the one-to-many strategy to that of using the many-to-many strategy. The results are shown in figure 5.9 and suggest that the gain depends mainly on the type of automaton used to constrain the preprocessing. The gain is smaller on automata that rely, partially, on using the road network ($\mathcal{A}_5$) and smallest for automata that exclude the public transit network ($\mathcal{A}_2$ and $\mathcal{A}_4$). In our setup, the road and public transit networks are the dominating means of transportation, that is, if all means are allowed, the shortest path between two locations is comprised, mostly, of road network and public transit edges (fastest alternatives). Furthermore, since the road network is much denser and larger than the public transit network, the shortest paths that make up a clique overlap less when constrained by the road network compared to the public transit

Table 5.3: Computational time for different partition sizes and the associated number of border vertices.

Graph	**Partition size**	**# Border vertices**				**Time [s]**
		min	med	max	tot	
G_{idf}	100	124	239	428	24 518	1.023
	200	65	180	344	36 975	1.278
	300	52	159	407	49 102	1.474
	400	56	134	263	55 303	1.673
	500	29	122	308	63 081	1.874
G_{fr}	800	13	91	317	94 897	16.471
	1 000	13	91	317	94 897	18.351
	2 000	3	67	331	141 104	19.742
	4 000	4	47	191	194 352	24.598
	6 000	2	38	183	239 639	33.366



(a) one-to-many algorithm (D_{pRegLC}) (b) many-to-many algorithm (lcD_{pRegLC})

Figure 5.8: Computational time for the profile cliques of G_{idf} as a function of partition size and automaton.

network. Hence, the gain factor is significant for automata $\mathcal{A}_1$ and $\mathcal{A}_3$.

For the France region, we experimentally selected the partitions such that the cell sizes are equivalent to those of the Ile-de-France region, and therefore, preprocessing times are similar when one CPU is dedicated to each cell in both cases.

5.3.3 Queries

After precomputing an overlay graph will all the partitions and automata, we generate 10 000 random queries $q(r, t, \tau)$ where r, t are the source and target vertices respectively, and τ is the departure time. We compare MUSE with D_{RegLC}

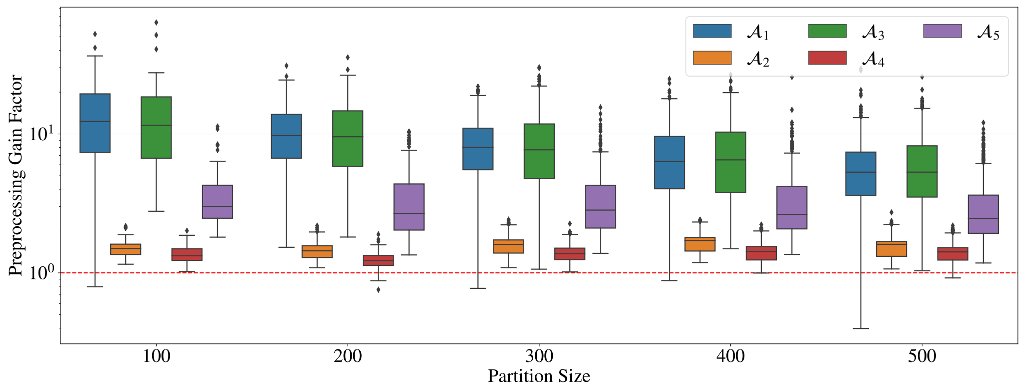


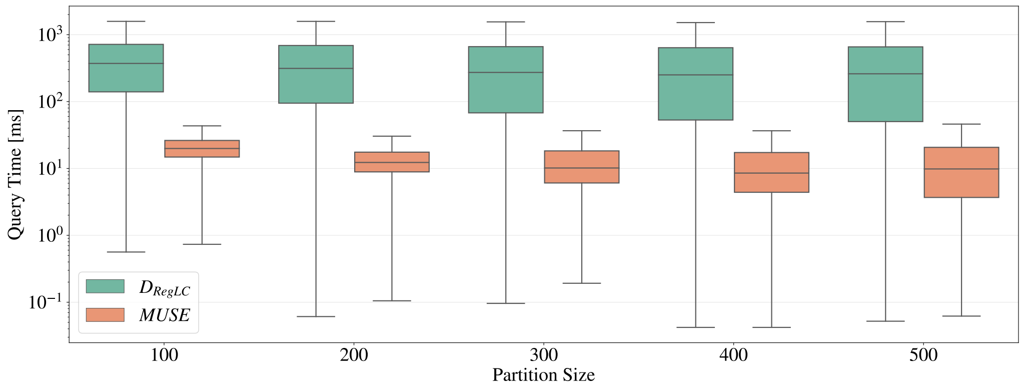
Figure 5.9: The preprocessing Gain Factor denoting the speedup of the many-to-many (lcD_{pRegLC}) over the one-to-many (D_{pRegLC}) strategy for each automaton and partition size. The horizontal red line corresponds to a gain of 1 and imply no speedup.

Table 5.4: List of algorithms used in the query phase of the experimental evaluation.

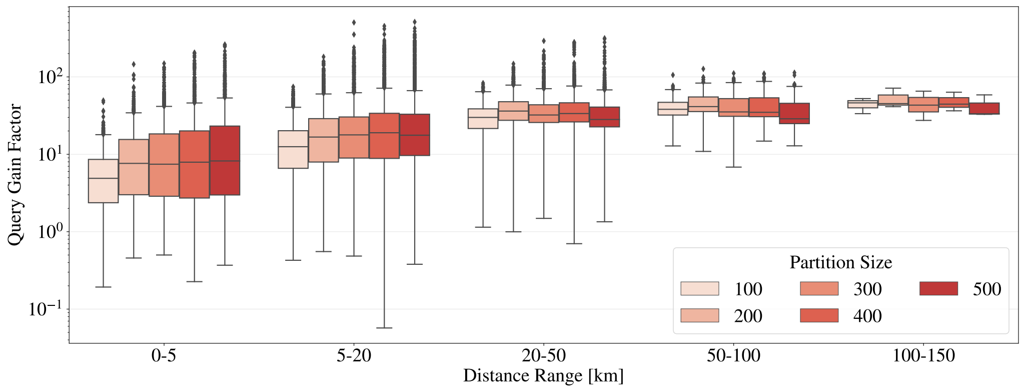
D_{RegLC}	Regular Language Constrained Dijkstra [Barrett *et al.* 2000].
MUSE	similar to D_{RegLC} but runs on the query graph $G^q = G_r \cup H \cup G_t$, formed by the overlay H and the source and target induced subgraphs G_r and G_t, respectively.
MUSE*	augments MUSE to obey Goal-Direction principles, similar to A* [Appi 1966].
MUSE^{SV}	similar to MUSE* but implements the *Sedgewick-Vitter* heuristic [Sedgewick & Vitter 1984] to tradeoff correctness for the sake of speed.

[Barrett *et al.* 2000], a multimodal version of Dijkstra (Table 5.4). Figure 5.10a depicts query times of both algorithms on the Ile-de-France region, for all queries, regardless of the constraining automaton. MUSE achieves a speedup of almost two orders of magnitude over D_{RegLC}.

By organizing the queries according to their range (Euclidean distance between the source and the target), we observe that the speedup increases with the distance range (figure 5.10b). This is expected because the query time not only depends on the partition size but also on the location of the source and target vertices. Recalling that the query graph $G^q = G_r \cup H \cup G_t$ combines the subgraphs G_r and G_t, induced by the cells containing the source and target vertices r and t, respectively and the overlay H, MUSE always explores G_r and G_t. Therefore, if the cells are large or if the source and target are either located inside the same cell or within adjacent cells, then MUSE does not benefit from the overlay to skip over the rest of the graph. Hence, MUSE performs best for long-range queries, as shown in figure 5.10b.



(a) Query times distribution of D_{RegLC} and MUSE for all automata.

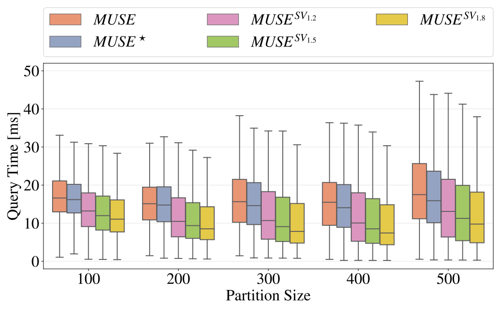


(b) The Query Gain Factor represents the achieved speedup of MUSE over D_{RegLC} for different partitions and query distance ranges (Euclidean distance).

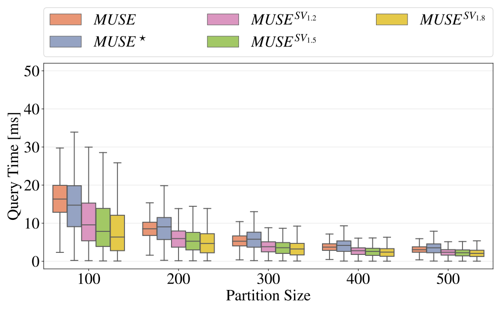
Figure 5.10: Performance comparison of D_{RegLC} and MUSE on the Ile-de-France region graph.

To achieve further speedups during the query step, MUSE can be augmented using any technique from the literature that does not require a preprocessing. We considered bi-directional search, that is, simultaneously running a forward-search (from the source) and backward-search (from the target) to reduce the search space. Although bi-directional search usually reduces the query time by half on static graphs, it was shown to be inefficient (and often results in speed-downs) on time-dependent graphs [Nannicini *et al.* 2008].

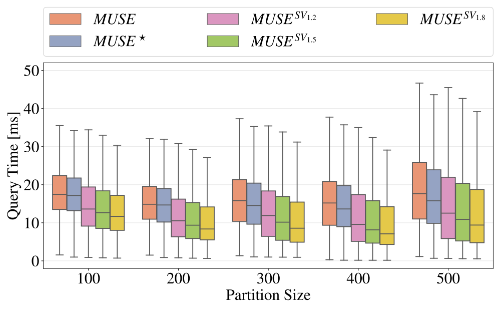
Instead, we focus here on goal direction and implemented MUSE* and MUSE^{SV} (Table 5.4), as already discussed in section 5.2.3. For the MUSE^{SV} variant, we tested three values for the heuristic tuning parameter $\alpha_1 = 1.2$, $\alpha_2 = 1.5$, and $\alpha_3 = 1.8$. We provide the results in figure 5.11, depicting the query time distribution of all MUSE variants for each automaton and partition size.



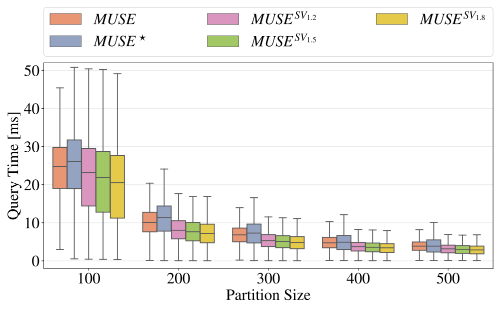
(a) Automaton $\mathcal{A}_1$



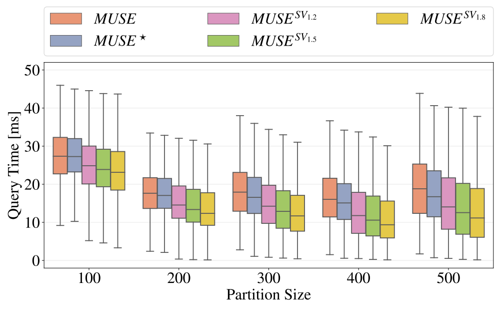
(b) Automaton $\mathcal{A}_2$



(c) Automaton $\mathcal{A}_3$



(d) Automaton $\mathcal{A}_4$



(e) Automaton $\mathcal{A}_5$

Figure 5.11: Query times distribution for the Ile-de-France region for each automaton of figure 5.6. Each plot depicts the running times of MUSE, MUSE*, and MUSE^{SV} (with a heuristic factor of 1.2, 1.5, and 1.8) for each partition size of the preprocessing.

We notice that query times on $\mathcal{A}_2$ and $\mathcal{A}_4$ are faster and keep decreasing when the partition size increases. In contrast, an optimal partition size (200 cells) exists for the automata $\mathcal{A}_1$, $\mathcal{A}_3$, and $\mathcal{A}_5$. Ultimately, query times are bound to increase again beyond a certain threshold of the partition size. The largest possible partition is, in fact, $|V|$ (the size of the graph) with each cell containing a single vertex. Hence, the query graph is the whole graph, and MUSE degrades to D_{RegLC}. Considering that $\mathcal{A}_2$ and $\mathcal{A}_4$ exclude the public transit network, the link and merge operations, which are the bottleneck of the algorithm, are trivial, which explains why the queries are significantly faster.

Goal direction does not seem to be a good strategy to accelerate queries in the Ile-de-France region. In fact, [Delling *et al.* 2011a] seem to obtain similar results for small partitions on unimodal road networks using Precomputed Cluster Distances (PCD) [Maue *et al.* 2010], another goal directed algorithm.

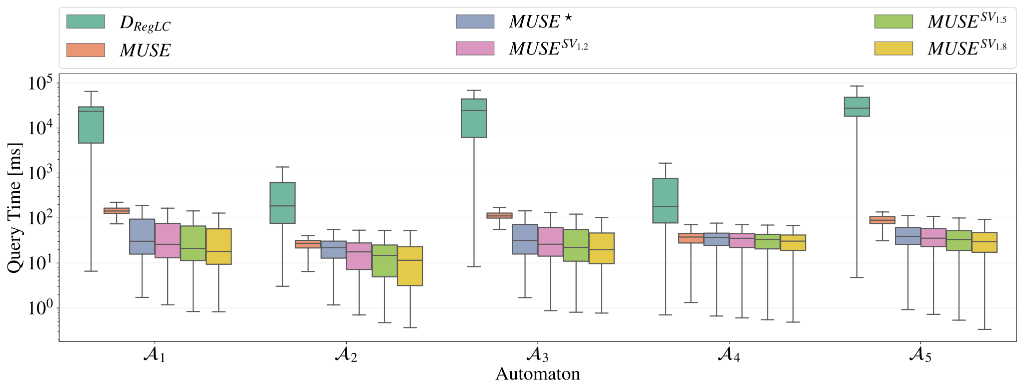
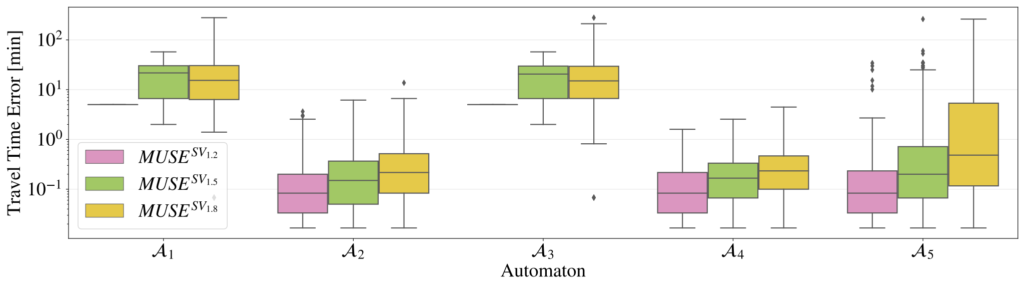


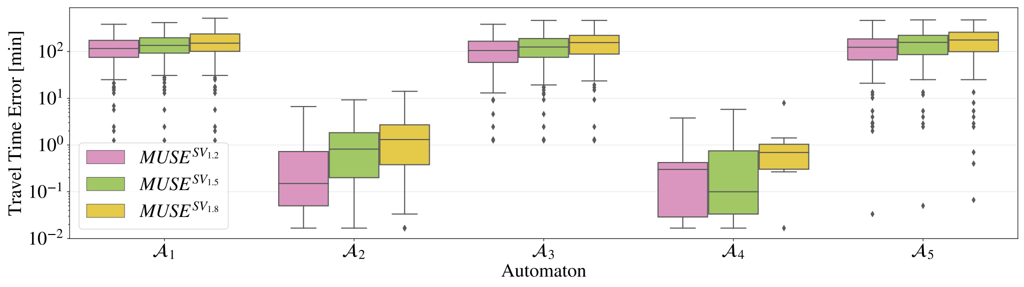
Figure 5.12: On the France region, we fix the partition size to 1 000 cells and report the query times distribution of D_{RegLC} versus MUSE and all of its variants, for each automaton.

In the France region, we fix the partition size to 1 000 cells and provide the query times in figure 5.12. As query the distance range increases significantly (up to 1 000 km), the computational gain of MUSE over D_{RegLC} reaches up to 3 orders of magnitude, especially on automata that include the public transit network ($\mathcal{A}_1$, $\mathcal{A}_3$, and $\mathcal{A}_5$). Furthermore, on the same automata, MUSE* provides a significant speedup.

Using the *Sedgewick-Vitter* heuristic allows us to reduce query times by half in Ile-de-France and up to an order of magnitude in France depending on the automaton and partition size (Fig. 5.12). This is, however, at the cost of sacrificing the travel time correctness. We report in figure 5.13 the Travel Time Error (in minutes) for each MUSE^{SV} variant, across all automata. MUSE$^{SV_{1.2}}$, the less aggressive overestimating variant, provides paths that are 99% of the times only a few seconds longer than the shortest paths. In the worst case, we measure a travel time error of 5 minutes. Interestingly, even MUSE$^{SV_{1.5}}$ and MUSE$^{SV_{1.8}}$ behave well if the automata do not allow using the public transit network ($\mathcal{A}_2$ and $\mathcal{A}_4$). Since the travel time function of public transit graph edges is non-continuous (figure 5.2), missing a connection results in a sudden increase of travel time, which explains the large errors observed on the remaining automata. Thus, this variant should be carefully exploited.



(a) Ile-de-France region



(b) France region

Figure 5.13: Travel Time Error distribution on the Ile-de-France and France regions for each automaton based on the heuristic factor setting of MUSE^{SV}.

5.4 Conclusion

We presented MUSE, a new approach to solve the multimodal shortest path problem via graph partitioning and language theory. Its main advantage is scalability, as it splits large graph instances into several independent cells that can be processed in parallel. We also provide different strategies to improve the performance of both the preprocessing and the query phases. We achieve preprocessing and query times that are fast enough to process a cell as well as resolve a query in only a few milliseconds. Hence, MUSE is a viable solution for real-time multimodal route planning. We experimentally tested our implementation and achieved a speedup of up to two orders of magnitude compared to D_{RegLC}, the label constrained version of Dijkstra's algorithm for multimodal networks. Using goal direction and heuristics, we further accelerate the queries and show that the obtained paths are most of the time, only a few seconds longer than the shortest paths.

Chapter 6

Conclusion

We explored over the course of this thesis several topics related to routing in transportation networks. Specifically, we based our research as much as possible on real traffic information to avoid the inherent bias and assumptions of many simulated datasets. Our main contributions relate to the map matching problem (chapter 3), an extensive evaluation of re-routing strategies in dynamic road networks (chapter 4), and efficient multimodal route planning (chapter 5) based on graph separators and language theory.

Map Matching is a long-standing problem and a crucial component of most route planning systems. Consequently, the literature is abundant in algorithms for various applications such as real-time guiding systems, traffic analytic, and freight tracking. For most of the proposed algorithms, the aim is to identify the graph's path that *best* matches a raw GPS-trace. However, for some applications such as research, correct, rather than probable, matching is critical. This prompted the need for a *unambiguous* map matching algorithm. In essence, our approach consists of computing candidate subpaths between the first pair of measurements in the trace that we iteratively extend for each subsequent GPS measurement. We show that the temporal constraint used to prune non-feasible subpaths gets stronger for each additional measurement. Through experimental evaluations using both simulated and real GPS traces, we are able to match 90% of the traces correctly.

Dynamic Route Planning attempts at solving the Shortest Path Problem (SPP) by taking into account traffic congestion as a time-evolving phenomenon. The literature is abundant in traffic-aware algorithms and congestion avoidance techniques. Nonetheless, large scale evaluations of vehicle re-routing strategies are not common. Specifically, most studies either rely on simulated datasets with synthetic traffic information or focus on small geographical regions and therefore overlook congestion patterns at the scale of a whole city. To address this gap, we collected a large dataset of real traffic speed profiles for various road networks for a period of 3 months. Our data was finely sampled and allowed us to conduct a thorough evaluation. We compared four routing strategies: static, no re-routing, continuous re-routing, and optimal routing. The latter technique is not realistic but emulates ideal predictions and was used as a reference to compute the absolute travel time gain factor for the remaining strategies. Our experimental evaluation showed that a sparse sampling of 5-10 min was enough to achieve almost optimal routing using the continuous re-routing strategy. Furthermore, we identified divergence vertices for each graph, denoting the geographical locations where continuous re-routing often diverges from the optimal route due to inaccurate re-routing. Most importantly, we show the discrepancies between real and synthetic traffic data.

Multimodal Route Planning adapts the SPP to networks that combine several modes of transportation. Consequently, traditional preprocessing techniques are not straightforward to adapt since the user modal constraints are only known at query time. Additionally, the size of the multimodal graphs is significantly larger than the unimodal graphs, which slows queries. Our goal was to devise a scalable preprocessing to address multimodal queries on large scale graphs. We, therefore, developed MUSE, a label constrained shortest path algorithm based on graph separators. Using graph separators allows us to take advantage of parallelism. We use a many-to-many label correcting shortest path algorithm to accelerate the preprocessing of each cell in the partition. To solve a query, we use the precomputed overlay to construct, on the fly, a query graph significantly smaller than the original graph. We experimentally evaluate our technique and provide several heuristic-based query algorithms to achieve further speedups. On the French multimodal network combining road, foot, bicycle, and all types of public transportation, MUSE achieves a speedup of up to 3 orders of magnitude over D_{RegLC}.

Throughout the thesis, we also reviewed many shortest path techniques and often had to implement several state-of-the-art algorithms to fairly compare their performance and gain a better insight on their advantages and disadvantages. Subsequently, we developed a graphical desktop application to visually inspect, on a digital map, shortest path queries, and their associated statistics based on the used algorithms. This tool became essential throughout our work, and we, therefore, extended it for other purposes: to display interactive results for the dynamic re-routing problem or to automatically construct multimodal graphs from publicly accessible datasets and visualize multimodal queries. We believe these tools could benefit other researchers that seek to replicate some of our results or run their own experiments. We summarize the list of the developed tools in table 6.1.

6.1 Future Work

The research related to route planning in transportation networks is vast and covers a multitude of problems according to the type of networks being considered, their size, the nature of the optimized metrics, and the intended applications. Our main goal is to develop a route planning algorithm that enables three fundamental features for practical applications: multimodality, multicriteria, and real-time traffic updates. Ultimately, all of these requirements should be integrated efficiently to answer queries on a global scale.

In the following sections, we discuss our perspectives, first, considering goals that expand on the contributions presented in this thesis, then, we discuss broader ideas, extending our gained knowledge in route planning to a new territory: *Space*.

6.1.1 Short term perspectives

First, to further improve the accuracy of our map matching algorithm, we plan to analyze the impact of the local topography. In urban road networks with a dense topology, there might exist several alternative routes that cannot be distinguished from the actual route. This is usually the case in grid-like networks such as in Manhattan. Ideally, we seek to provide an adaptive technique, able to tune the sampling rate dynamically depending on a set of local metrics, to reduce the energy consumption of many crowd-sensing applications. The sampling frequency could, for instance, be increased for parts of the network where matching is ambiguous.

For the dynamic routing problem, we plan to incorporate historical traffic patterns to further improve the continuous re-routing strategy. Our goal is to adapt the sampling rate for each query, based on the route evaluated at departure time. In essence, by considering traffic congestion and keeping track of diverging vertices, we plan to propose a technique that automatically adapts the GPS sampling rate based on the location of the vehicle. An interesting approach might be a machine learning model that leverages the computed query factors (travel time gain, divergence ratio, stretch factor) to automatically trigger re-routing.

Lastly, on the multimodal route planning problem, we recognized that the partitioning is critical for the performance of both the preprocessing and the query. We first tested our own implementation of the PUNCH algorithm [Delling *et al.* 2011b] but switched later on to METIS, the general graph partitioning algorithm, which was more reliable and provided better results. Therefore, a formal analysis of graph partitioning algorithms tailored to transportation networks, and especially to multimodal networks, is essential to improve our current results. Furthermore, to improve query times, multilevel partitioning for multimodal graphs is an interesting next step. It allows skipping over most of the network using the overlay edges on the low level of the partition while reducing the search space in the source and target cells by exploring the high levels of the partition. During preprocessing, we also need to compute multimodal profile cliques. We observed that the *merge* and *link* operations are the bottleneck of lcD_{pRegLC} due to a rapidly increasing number of breakpoints in the travel time functions. An interesting work perspective would be to rely on heuristics [Strasser 2017] to simplify these operations and evaluate the tradeoff between computational speed and impact on correctness.

Furthermore, considering the schedule-based nature of public transit networks, missing a departure because of a limited transfer time can result in a sudden significant increase in the overall travel time. Therefore evaluating the *risk* associated with a route is an important factor, and in practice, the user should be able to select a route from a few alternatives. Additionally, we plan to develop an efficient technique to monitor cells and correct the preprocessing according to traffic updates.

SNCF, the French National Railway Company, seeks to develop a new application that enables multimodal route planning by coordinating the efforts of different agencies. They, in fact, showed great interest in our work, and thanks to the CIFRE aspect of this thesis, we discussed potential collaboration in the upcoming months.

So far, in this thesis, we reviewed the characteristics of dynamic routing and proposed an efficient multimodal routing algorithm capable of handling traffic variations. However, we currently only optimize a single metric: travel time. From a practical point of view, users may be inclined to also minimize monetary costs, walking time, and public transit transfers. Our approach, MUSE, is currently capable of handling certain constraints such as the maximum number of allowed transfers. We manage this by adding transitions in the automaton. Nonetheless, Incorporating several metrics is not trivial because paths that minimize different criteria are not comparable, and therefore we must compute Pareto sets of non-dominated paths. Such paths are, however, numerous, especially in a multimodal network. [Müller-Hannemann & Weihe 2001] study the problem of multicriteria single-source shortest path problem and show that there exist certain characteristics in practical scenarios related to railway networks that limit the number of Pareto optimal sets. We believe the next exciting step is to further augment this study to the multimodal problem in a dynamic environment.

6.1.2 Long term perspectives

Accurate traffic prediction is a long-standing problem of paramount importance for several applications such as traffic flow management, transport navigation, and taxi dispatch. In urban areas, road networks are often actively monitored using a variety of dedicated sensors. Fixed sensors are embedded within the network and consist of, for instance, inductive loops, radars, and surveillance cameras. Depending on the type of sensors, they can provide traffic volumes and density, road segment speeds, and even vehicle category. Nonetheless, their deployment and maintenance are not sustainable due to the high monetary cost.

We are interested in traffic inference [Lin *et al.* 2017] as a process of estimating traffic conditions for non-monitored road segments based on known traffic information for a subset of road segments in the network. Considering the road segment's Spatio-temporal dependencies, we would like to answer the following question: *which segments should we monitor to infer traffic globally?* We already ran a few experiments based on Dynamic Time Warping (DTW) and machine learning to cluster road segments based on their speed profiles as time series functions. Currently, the number of clusters we obtain is too high due to the stochastic nature of traffic data. We believe that prior to clustering, we should develop a method to first identify recurring vs. non-recurring congestion.

Route planning is evidently not exclusive to transportation networks. In fact, solving the shortest path problem is required in a plethora of applications such as the game industry, social networks, and IP routing. Being inclined to space technology, we believe we could contribute to the problem of multilayer satellite routing [Xiaogang *et al.* 2016], which is rather analogous to the multimodal shortest path problem.

Currently, there are approximately 2 000 satellites orbiting Earth. In the upcoming years, however, tens of thousands of additional satellites are planned for launch. For the ambitious goal of providing global internet access, the Starlink project [McDowell 2020], for instance, will deploy 12 000 satellites, a number that will potentially be extended to 42 000 satellites. In fact, a space-race toward the same objective is currently taking place and includes other projects such as Kuiper and OneWeb [Winslett 2019], which will deploy their own satellite networks.

In that context, inter-satellite routing is required to efficiently transmit information in a highly dynamic environment, subject to transmission delays and limited on-board processing power. A multilayer satellite network is composed of satellites with different orbits, mainly: Low Earth Orbit (LEO), Medium Earth Orbit (MEO), and Geostationary Earth Orbit (GEO) satellites. In such networks, data can be transmitted either through intra-plane (same layer satellites) or inter-layer (satellites on different layers) links. Nonetheless, satellites are orbiting at different speeds and inclinations, and therefore, the overall topology of the network is constantly changing.

Our goal is to partition space into cells that are fixed with respect to Earth's rotation. For each cell, we can compute the different topologies of satellites it contains at different time intervals during which the topology is maintained. We can then adapt the ideas behind the MUSE algorithm to compute constrained shortest paths in each cell to achieve load-balancing.

Table 6.1: List of useful applications developed throughout the thesis.

Application	Description
SPAGUI	Shortest Path Algorithms Graphical User Interface (SPAGUI) is a desktop application that enables downloading maps from OpenStreetMap (.osm files) and automatically generates a directed graph representing the road network of the chosen region. SPAGUI also implements several shortest path algorithms and therefore serves as a benchmark. https://github.com/aminefalek/spagui
DYNAMO	a graphical interface allows the user to construct a dynamic road network model by combining OpenStreetMap data for the topology and traffic data (road segment speed profiles) from a third-party API. It allows the user to generate and resolve shortest path queries using the routing strategies seen in chapter 4. The results are interactively displayed and shared with a common cloud infrastructure to collect and exploit experimental results from other researchers. https://github.com/aminefalek/dynamo
ICube-ITS	a web application used to display some of our results for the dynamic re-routing problem. http://its-icube.com/
MUSE	extends SPAGUI to create multimodal graphs from OpenStreetMap and GTFS datasets automatically. https://github.com/aminefalek/muse
GTA	Graph Theory App was developed to complement the teaching material of a graph theory class. It provides means to draw arbitrary graphs and visualize classical algorithms such as Dijkstra, Kosaraju, and Kruskal. It also enables the students to code each algorithm themselves and visually debug their implementation. https://github.com/aminefalek/gta

List of Figures

List of Tables

Bibliography

[Afrin & Yodo 2020] Tanzina Afrin and Nita Yodo. *A Survey of Road Traffic Congestion Measures towards a Sustainable and Resilient Transportation System*. Sustainability, vol. 12, no. 11, page 4660, 2020. 1

[Ahmed *et al.* 2015] Mahmuda Ahmed, Sophia Karagiorgou, Dieter Pfoser and Carola Wenk. *A comparison and evaluation of map construction algorithms using vehicle tracking data*. GeoInformatica, vol. 19, no. 3, pages 601–632, jul 2015. 35

[Ahsani *et al.* 2019] Vesal Ahsani, Mostafa Amin-Naseri, Skylar Knickerbocker and Anuj Sharma. *Quantitative analysis of probe data characteristics: Coverage, speed bias and congestion detection precision*. Journal of Intelligent Transportation Systems, vol. 23, no. 2, pages 103–119, 2019. 49

[Aly *et al.* 2017] H Aly, A Basalamah and M Youssef. *Automatic Rich Map Semantics Identification Through Smartphone-Based Crowd-Sensing*. IEEE Transactions on Mobile Computing, vol. 16, no. 10, pages 2712–2725, oct 2017. 16

[Antsfeld & Walsh 2012] Leonid Antsfeld and Toby Walsh. *Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm*. In Proceedings of the 19th ITS World Congress, page 32, 2012. 31

[Appi 1966] J Math Anal Appi. *A formal basis for the heuristic determination of minimum cost paths*. 1966. 22, 83, 89

[Barrett *et al.* 2000] Chris Barrett, Riko Jacob and Madhav Marathe. *Formal-language-constrained path problems*. SIAM Journal on Computing, vol. 30, no. 3, pages 809–837, 2000. 32, 77, 78, 89

[Barrett *et al.* 2002] Chris Barrett, Keith Bisset, Riko Jacob, Goran Konjevod and Madhav Marathe. *Classical and Contemporary Shortest Path Problems in Road Networks: Implementation and Experimental Analysis of the TRANSIMS Router*. In European Symposium on Algorithms (ESA), pages 126–138, 2002. 30

[Bast *et al.* 2016a] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner and Renato F Werneck. *Algorithm engineering*. chapter Route Plan, pages 19–80. Springer, 2016. 29, 49

[Bast *et al.* 2016b] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner and

Renato F Werneck. *Route planning in transportation networks.* In Algorithm engineering, pages 19–80. Springer, 2016. 9

[Bast 2009] Hannah Bast. *Car or public transport—two worlds.* In Efficient Algorithms, pages 355–367. Springer, 2009. iii, 3, 19

[Bauer *et al.* 2010] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes and Dorothea Wagner. *Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm.* Journal of Experimental Algorithmics (JEA), vol. 15, pages 2–3, 2010. 27

[Bauer *et al.* 2011] Reinhard Bauer, Daniel Delling and Dorothea Wagner. *Experimental study of speed up techniques for timetable information systems.* Networks, vol. 57, no. 1, pages 38–52, 2011. 32

[Behrisch *et al.* 2011] Michael Behrisch, Laura Bieker, Jakob Erdmann and Daniel Krajzewicz. *Sumo–simulation of urban mobility.* In The Third International Conference on Advances in System Simulation (SIMUL 2011), Barcelona, Spain, volume 42, 2011. 30

[Bentley & Ottmann 1979] Jon Louis Bentley and Thomas A Ottmann. *Algorithms for reporting and counting geometric intersections.* IEEE Transactions on computers, no. 9, pages 643–647, 1979. 79

[Bernstein *et al.* 1996] David Bernstein, Alain Kornhauser*et al.* *An introduction to map matching for personal navigation assistants.* 1996. 16

[Bhatia & Others 2010] Nitin Bhatia and Others. *Survey of nearest neighbor techniques.* arXiv preprint arXiv:1007.0085, 2010. 15, 74

[Boundy 2019] Robert Gary Boundy. *Transportation Energy Data Book: Edition 37.* Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2019. 1

[Bresenham 1965] J E Bresenham. *Algorithm for computer control of a digital plotter.* IBM systems journal, vol. 4, no. 1, pages 25–30, 1965. 38

[Brodal & Jacob 2004] Gerth Stølting Brodal and Riko Jacob. *Time-dependent networks as models to achieve fast exact time-table queries.* Electronic Notes in Theoretical Computer Science, vol. 92, pages 3–15, 2004. 12, 27

[Brüggemann-Klein 1993] Anne Brüggemann-Klein. *Regular expressions into finite automata.* Theoretical Computer Science, vol. 120, no. 2, pages 197–213, 1993. 78

[Cabannes *et al.* 2017] Théophile Cabannes, Marco Antonio Sangiovanni Vincentelli, Alexander Sundt, Hippolyte Signargout, Emily Porter, Vincent Fighiera, Juliette Ugirumurera and Alexandre M Bayen. *The impact of*

GPS-enabled shortest path routing on mobility: a game theoretic approach 2. University of California, Berkeley, 2017. 29

[Chan & Yang 2009] E P F Chan and Y Yang. *Shortest Path Tree Computation in Dynamic Graphs*. IEEE Transactions on Computers, vol. 58, no. 4, pages 541–557, apr 2009. 29

[Chao *et al.* 2020] Pingfu Chao, Yehong Xu, Wen Hua and Xiaofang Zhou. *A Survey on Map-Matching Algorithms*. In Australasian Database Conference, pages 121–133. Springer, 2020. 2, 16, 18

[Chen *et al.* 2007] Mo Chen, Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche and Lingling Tong. Priority queues and dijkstra's algorithm. Computer Science Department, University of Texas at Austin, 2007. 19

[Chen *et al.* 2010] Yanyan Chen, Michael G H Bell and Klaus Bogenberger. *Risk-Averse Autonomous Route Guidance by a Constrained A* Search*. Journal of Intelligent Transportation Systems, vol. 14, no. 3, pages 188–196, 2010. 53

[Chen *et al.* 2018] C Chen, S Jiao, S Zhang, W Liu, L Feng and Y Wang. *TripImputor: Real-Time Imputing Taxi Trip Purpose Leveraging Multi-Sourced Urban Data*. IEEE Transactions on Intelligent Transportation Systems, vol. 19, no. 10, pages 3292–3304, oct 2018. 30

[Chen *et al.* 2019] Che-Ming Chen, Chia-Ching Liang and Chih-Peng Chu. *Long-term travel time prediction using gradient boosting*. Journal of Intelligent Transportation Systems, vol. 0, no. 0, pages 1–16, 2019. 49

[Cherkassky *et al.* 1996] Boris V Cherkassky, Andrew V Goldberg and Tomasz Radzik. *Shortest paths algorithms: Theory and experimental evaluation*. Mathematical programming, vol. 73, no. 2, pages 129–174, 1996. 8

[Coifman & Mallika 2007] Benjamin A Coifman and Ramachandran Mallika. *Distributed surveillance on freeways emphasizing incident detection and verification*. Transportation research part A: policy and practice, vol. 41, no. 8, pages 750–767, 2007. 30

[Cooke & Halsey 1966] Kenneth L Cooke and Eric Halsey. *The shortest route through a network with time-dependent internodal transit times*. Journal of mathematical analysis and applications, vol. 14, no. 3, pages 493–498, 1966. 20

[D'Andrea *et al.* 2015] Eleonora D'Andrea, Pietro Ducange, Beatrice Lazzerini and Francesco Marcelloni. *Real-time detection of traffic from twitter stream analysis*. IEEE Transactions on Intelligent Transportation Systems, vol. 16, no. 4, pages 2269–2283, 2015. 30

[Dantzig & Thapa 2006] George B Dantzig and Mukund N Thapa. Linear programming 2: theory and extensions. Springer Science & Business Media, 2006. 20

[Data 2020] Navitia Open Data. *France public transport data*, 2020. 84

[de France Mobilités 2020] Ile de France Mobilités. *Données Offre de transport Ile-de-France Mobilités au format GTFS*, 2020. 84

[Delling & Wagner 2009] Daniel Delling and Dorothea Wagner. *Time-dependent route planning*. In Robust and online large-scale optimization, pages 207–230. Springer, 2009. 25

[Delling *et al.* 2009] Daniel Delling, Thomas Pajor and Dorothea Wagner. *Accelerating multi-modal route planning by access-nodes*. In European Symposium on Algorithms, pages 587–598. Springer, 2009. 14, 32

[Delling *et al.* 2011a] Daniel Delling, Andrew V Goldberg, Thomas Pajor and Renato F Werneck. *Customizable route planning*. In International Symposium on Experimental Algorithms, pages 376–387. Springer, 2011. 10, 67, 92

[Delling *et al.* 2011b] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn and Renato F Werneck. *Graph partitioning with natural cuts*. In 2011 IEEE International Parallel & Distributed Processing Symposium, pages 1135–1146. IEEE, 2011. 97

[Delling *et al.* 2013] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk and Renato F Werneck. *PHAST: Hardware-accelerated shortest path trees*. Journal of Parallel and Distributed Computing, vol. 73, no. 7, pages 940–952, 2013. 23

[Delling *et al.* 2015] Daniel Delling, Thomas Pajor and Renato F Werneck. *Round-based public transit routing*. Transportation Science, vol. 49, no. 3, pages 591–604, 2015. 28

[Dell'Orco *et al.* 2016] Mauro Dell'Orco, Mario Marinelli and Mehmet Ali Silgu. *Bee Colony Optimization for innovative travel time estimation, based on a mesoscopic traffic assignment model*. Transportation Research Part C: Emerging Technologies, vol. 66, pages 48–60, 2016. 30

[Demetrescu *et al.* 2009] Camil Demetrescu, Andrew V Goldberg and David S Johnson. The Shortest Path Problem: Ninth DIMACS Implementation Challenge, volume 74. American Mathematical Soc., 2009. 18

[Dibbelt *et al.* 2013] Julian Dibbelt, Thomas Pajor, Ben Strasser and Dorothea Wagner. *Intriguingly simple and fast transit routing*. In International Symposium on Experimental Algorithms, pages 43–54. Springer, 2013. 28

[Dibbelt *et al.* 2015] Julian Dibbelt, Thomas Pajor and Dorothea Wagner. *User-constrained multimodal route planning.* Journal of Experimental Algorithmics (JEA), vol. 19, pages 1–19, 2015. 14, 33

[Dijkstra 1959] Edsger W Dijkstra. *A note on two problems in connexion with graphs.* Numerische mathematik, vol. 1, no. 1, pages 269–271, 1959. 2, 8, 18, 19, 53

[Djidjev 1982] Hristo Nicolov Djidjev. *On the problem of partitioning planar graphs.* SIAM Journal on Algebraic Discrete Methods, vol. 3, no. 2, pages 229–240, 1982. 75

[DOT 2015] U S DOT. *Beyond traffic 2045: Trends and choices.* US: DOT, 2015. 3

[Duan *et al.* 2018] Z Duan, Y Yang, K Zhang, Y Ni and S Bajgain. *Improved Deep Hybrid Networks for Urban Traffic Flow Prediction Using Trajectory Data.* IEEE Access, vol. 6, pages 31820–31827, 2018. 29

[dyn 2020] *Traffic simulation software your city can plan on*, 2020. 30

[Eppstein & Goodrich 2008] David Eppstein and Michael T Goodrich. *Studying (non-planar) road networks through an algorithmic lens.* In Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems, page 16. ACM, 2008. 75

[Falek *et al.* 2018] M Falek, C Pelsser, A Gallais, S Julien and F Theoleyre. *Unambiguous, Real-Time and Accurate Map Matching for Multiple Sensing Sources.* In International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob). IEE, nov 2018. 30

[Flamini *et al.* 2018] Marta Flamini, Marialisa Nigro and Dario Pacciarelli. *The value of real-time traffic information in urban freight distribution.* Journal of Intelligent Transportation Systems, vol. 22, no. 1, pages 26–39, 2018. 49

[Forney 1973] G David Forney. *The viterbi algorithm.* Proceedings of the IEEE, vol. 61, no. 3, pages 268–278, 1973. 17

[Fredman & Tarjan 1987] Michael L Fredman and Robert Endre Tarjan. *Fibonacci heaps and their uses in improved network optimization algorithms.* Journal of the ACM (JACM), vol. 34, no. 3, pages 596–615, 1987. 19

[Fredman *et al.* 1986] Michael L Fredman, Robert Sedgewick, Daniel D Sleator and Robert E Tarjan. *The pairing heap: A new form of self-adjusting heap.* Algorithmica, vol. 1, no. 1-4, pages 111–129, 1986. 19

[Fuchs 2010] Fabian Fuchs. *On Preprocessing the ALT-Algorithm.* Student thesis, Faculty of Computer Science, Institut for Theoretical Informatics (ITI), Karlsruhe Institute of Technology (KIT), 2010. 23

[Garey & Johnson 2002] Michael R Garey and David S Johnson. Computers and intractability, volume 29. wh freeman New York, 2002. 75

[Gawron 1998] Christian Gawron. *An iterative algorithm to determine the dynamic user equilibrium in a traffic simulation model.* International Journal of Modern Physics C, vol. 9, no. 03, pages 393–407, 1998. 65

[Geisberger & Vetter 2011] Robert Geisberger and Christian Vetter. *Efficient routing in road networks with turn costs.* In International Symposium on Experimental Algorithms, pages 100–111. Springer, 2011. 10

[Geisberger *et al.* 2008] Robert Geisberger, Peter Sanders, Dominik Schultes and Daniel Delling. *Contraction hierarchies: Faster and simpler hierarchical routing in road networks.* In International Workshop on Experimental and Efficient Algorithms, pages 319–333. Springer, 2008. 25

[Geofabrik 2019] OpenStreetMap Geofabrik. *OpenStreetMap data extract*, 2019. 84

[Gharaibeh *et al.* 2017] A Gharaibeh, M A Salahuddin, S J Hussini, A Khreishah, I Khalil, M Guizani and A Al-Fuqaha. *Smart Cities: A Survey on Data Management, Security, and Enabling Technologies.* IEEE Communications Surveys Tutorials, vol. 19, no. 4, pages 2456–2501, 2017. 35

[Gmira *et al.* 2019] M Gmira, M Gendreau, A Lodi and J.-Y. Potvin. *Managing in real-time a vehicle routing plan with time-dependent travel times on a road network.* Technical report 2019-4, CIRRELT, 2019. 29

[Goh *et al.* 2012] Chong Yang Goh, Justin Dauwels, Nikola Mitrovic, Muhammad Tayyab Asif, Ali Oran and Patrick Jaillet. *Online map-matching based on hidden markov model for real-time traffic sensing applications.* In 2012 15th International IEEE Conference on Intelligent Transportation Systems, pages 776–781. IEEE, 2012. 17

[Goldberg & Harrelson 2005] Andrew V Goldberg and Chris Harrelson. *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, 2005. 22, 23, 32

[Goldberg *et al.* 2006] Andrew V Goldberg, Haim Kaplan and Renato F Werneck. *Reach for A*: Shortest Path Algorithms with Preprocessing.* In The Shortest Path Problem, pages 93–140, 2006. 27

[Goldberg *et al.* 2007] Andrew V Goldberg, Haim Kaplan and Renato F Werneck. *Better landmarks within reach.* In International Workshop on Experimental and Efficient Algorithms, pages 38–51. Springer, 2007. 27

[Greenfeld 2002] Joshua S Greenfeld. *Matching GPS observations to locations on a digital map.* In 81th annual meeting of the transportation research board, volume 1, pages 164–173. Washington, DC, 2002. 16

[Güner *et al.* 2012] Ali R Güner, Alper Murat and Ratna Babu Chinnam. *Dynamic routing under recurrent and non-recurrent congestion using real-time ITS information*. Computers & Operations Research, vol. 39, no. 2, pages 358–373, 2012. 29, 31

[Guo *et al.* 2016] B Guo, C Chen, D Zhang, Z Yu and A Chin. *Mobile crowd sensing and computing: when participatory sensing meets participatory social media*. IEEE Communications Magazine, vol. 54, no. 2, pages 131–137, feb 2016. 35

[Gustafsson *et al.* 2002] Fredrik Gustafsson, Fredrik Gunnarsson, Niclas Bergman, Urban Forssell, Jonas Jansson, Rickard Karlsson and P-J Nordlund. *Particle filters for positioning, navigation, and tracking*. IEEE Transactions on signal processing, vol. 50, no. 2, pages 425–437, 2002. 17

[Harrison 1978] Michael A Harrison. Introduction to formal language theory. Addison-Wesley Longman Publishing Co., Inc., 1978. 77

[Hashemi & Karimi 2014] Mahdi Hashemi and Hassan A Karimi. *A critical review of real-time map-matching algorithms: Current issues and future directions*. Computers, Environment and Urban Systems, vol. 48, pages 153–165, 2014. 18

[Heng *et al.* 2011] Liang Heng, Grace Xingxin Gao, Todd Walter and Per Enge. *Statistical characterization of GPS signal-in-space errors*. In International Technical Meeting of the Institute of Navigation (ION ITM), pages 312–319, San Diego, CA, 2011. 39, 45

[HERE Technologies 2018] HERE Technologies. *Real-Time Traffic*. \url{https://www.here.com}, 2018. 58

[Hilger *et al.* 2009] Moritz Hilger, Ekkehard Köhler, Rolf H Möhring and Heiko Schilling. *Fast point-to-point shortest path computations with arc-flags*. The Shortest Path Problem: Ninth DIMACS Implementation Challenge, vol. 74, pages 41–72, 2009. 23

[Hu *et al.* 2017] Wenbin Hu, Liping Yan, Huan Wang, Bo Du and Dacheng Tao. *Real-time traffic jams prediction inspired by Biham, Middleton and Levine (BML) model*. Information Sciences, vol. 381, pages 209–228, 2017. 50

[Hunter *et al.* 2013] Timothy Hunter, Pieter Abbeel and Alexandre Bayen. *The path inference filter: model-based low-latency map matching of probe vehicle data*. IEEE Transactions on Intelligent Transportation Systems, vol. 15, no. 2, pages 507–529, 2013. 17

[ITSJPO 2020] Intelligent Transportation Systems Joint Program Office ITSJPO. *ITS Research Fact Sheets*, 2020. 1, 2

[Jokar Arsanjani *et al.* 2015] Jamal Jokar Arsanjani, Peter Mooney, Alexander Zipf and Anne Schauss. Quality Assessment of the Contributed Land Use Information from OpenStreetMap Versus Authoritative Datasets, pages 37–58. Springer International Publishing, Cham, 2015. 35

[Kamga *et al.* 2011] Camille Kamga, Kyriacos Mouskos and Robert Paaswell. *A methodology to estimate travel time using dynamic traffic assignment {(DTA)} under incident conditions.* Transportation Research Part C: Emerging Technologies, vol. 19, no. 6, pages 1215–1224, dec 2011. 30

[Karypis & Kumar 1998] George Karypis and Vipin Kumar. *A fast and high quality multilevel scheme for partitioning irregular graphs.* SIAM Journal on scientific Computing, vol. 20, no. 1, pages 359–392, 1998. 75

[Kaufman & Smith 1993] David E Kaufman and Robert L Smith. *Fastest paths in time-dependent networks for intelligent vehicle-highway systems application.* Journal of Intelligent Transportation Systems, vol. 1, no. 1, pages 1–11, 1993. 9, 69

[Kernighan & Lin 1970] Brian W Kernighan and Shen Lin. *An efficient heuristic procedure for partitioning graphs.* Bell system technical journal, vol. 49, no. 2, pages 291–307, 1970. 76

[Kirchler *et al.* 2011] Dominik Kirchler, Leo Liberti, Thomas Pajor and Roberto Wolfler Calvo. *UniALT for regular language contrained shortest paths on a multi-modal transportation network.* In 11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011. 32

[Ladino *et al.* 2016] A Ladino, A Kibangou, H Fourati and C C de Wit. *Travel time forecasting from clustered time series via optimal fusion strategy.* In European Control Conference (ECC), pages 2234–2239, 2016. 30

[Lai & Kuo 2016] Wei-Kuang Lai and Ting-Huan Kuo. *Vehicle Positioning and Speed Estimation Based on Cellular Network Signals for Urban Roads.* ISPRS International Journal of Geo-Information, vol. 5, no. 10, page 181, 2016. 30

[Lee *et al.* 2019] Haengju Lee, Saerona Choi, Hojin Jung, Byungkyu Brian Park and Sang H Son. *A route guidance system considering travel time unreliability.* Journal of Intelligent Transportation Systems, vol. 23, no. 3, pages 282–299, 2019. 49

[Lehmann & Gross 2016] A Lehmann and A Gross. *Using crowd sensed data as input to congestion model.* In International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops), pages 1–6. IEEE, mar 2016. 16

[Li & Sun 2019] Xiang Li and Jian-Qiao Sun. *Multi-objective optimal predictive control of signals in urban traffic network*. Journal of Intelligent Transportation Systems, vol. 23, no. 4, pages 370–388, 2019. 29

[Li *et al.* 2014] Y Li, D Jin, P Hui, Z Wang and S Chen. *Limits of Predictability for Large-Scale Urban Vehicular Mobility*. IEEE Transactions on Intelligent Transportation Systems, vol. 15, no. 6, pages 2671–2682, 2014. 30

[Li *et al.* 2017a] Weizi Li, Dong Nie, David Wilkie and Ming C Lin. *Citywide estimation of traffic dynamics via sparse GPS traces*. IEEE Transactions on Intelligent Transportation Systems, jul 2017. 16

[Li *et al.* 2017b] Z Li, R Al Hassan, M Shahidehpour, S Bahramirad and A Khodaei. *A Hierarchical Framework for Intelligent Traffic Management in Smart Cities*. IEEE Transactions on Smart Grid, vol. PP, no. 99, page 1, 2017. 51

[Li *et al.* 2017c] Z Li, I V Kolmanovsky, E M Atkins, J Lu, D P Filev and Y Bai. *Road Disturbance Estimation and Cloud-Aided Comfort-Based Route Planning*. IEEE Transactions on Cybernetics, vol. 47, no. 11, pages 3879–3891, nov 2017. 49

[Liang *et al.* 2018] Xiao Liang, Gonçalo Homem de Almeida Correia and Bart van Arem. *Applying a model for trip assignment and dynamic routing of automated taxis with congestion: system performance in the City of Delft, The Netherlands*. Transportation Research Record, vol. 2672, no. 8, pages 588–598, 2018. 31

[Liebig *et al.* 2017] Thomas Liebig, Nico Piatkowski, Christian Bockermann and Katharina Morik. *Dynamic route planning with real-time traffic predictions*. Information Systems, vol. 64, pages 258–265, 2017. 30, 35

[Lin *et al.* 2017] Lu Lin, Jianxin Li, Feng Chen, Jieping Ye and Jinpeng Huai. *Road traffic speed prediction: a probabilistic model fusing multi-source data*. IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 7, pages 1310–1323, 2017. 98

[Liu 2017] Jingmeng Liu. *LSTM network: a deep learning approach for short-term traffic forecast*. IET Intelligent Transport Systems, vol. 11, pages 68–75(7), 2017. 55

[Madkour *et al.* 2017] Amgad Madkour, Walid G Aref, Faizan Ur Rehman, Mohamed Abdur Rahman and Saleh Basalamah. *A survey of shortest-path algorithms*. arXiv preprint arXiv:1705.02044, 2017. 18

[Mathew & Xavier 2014] Joseph Mathew and P M Xavier. *A survey on using wireless signals for road traffic detection*. International Journal of Research in Engineering and Technology, vol. 3, no. 1, 2014. 30

[Maue *et al.* 2010] Jens Maue, Peter Sanders and Domagoj Matijevic. *Goal-directed shortest-path queries using precomputed cluster distances.* Journal of Experimental Algorithmics (JEA), vol. 14, pages 2–3, 2010. 92

[McArdle *et al.* 2012] Gavin McArdle, Aonghus Lawlor, Eoghan Furey and Alexei Pozdnoukhov. *City-scale Traffic Simulation from Digital Footprints.* In ACM SIGKDD International Workshop on Urban Computing, pages 47–54, 2012. 31

[McDowell 2020] Jonathan C McDowell. *The Low Earth Orbit Satellite Population and Impacts of the SpaceX Starlink Constellation.* The Astrophysical Journal Letters, vol. 892, no. 2, page L36, 2020. 99

[Möhring *et al.* 2007] Rolf H Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner and Thomas Willhalm. *Partitioning graphs to speedup Dijkstra's algorithm.* Journal of Experimental Algorithmics (JEA), vol. 11, pages 2–8, 2007. 24

[Mooney *et al.* 2017] Peter Mooney, Marco Minghini and Others. *A review of OpenStreetMap data.* 2017. 84

[Müller-Hannemann & Weihe 2001] Matthias Müller-Hannemann and Karsten Weihe. *Pareto shortest paths is often feasible in practice.* In International Workshop on Algorithm Engineering, pages 185–197. Springer, 2001. 98

[Müller-Hannemann *et al.* 2007] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner and Christos Zaroliagis. *Timetable information: Models and algorithms.* In Algorithmic Methods for Railway Optimization, pages 67–90. Springer, 2007. 11

[Nannicini *et al.* 2008] Giacomo Nannicini, Daniel Delling, Leo Liberti and Dominik Schultes. *Bidirectional A* search for time-dependent fast paths.* In International Workshop on Experimental and Efficient Algorithms, pages 334–346. Springer, 2008. 20, 21, 90

[Newson & Krumm 2009a] Paul Newson and John Krumm. *Hidden Markov map matching through noise and sparseness.* In Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems, pages 336–343, 2009. 17

[Newson & Krumm 2009b] Paul Newson and John Krumm. *Hidden Markov Map Matching Through Noise and Sparseness.* In 17th ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL GIS), pages 336–343, Seattle, WA, nov 2009. 44, 47

[Ochieng *et al.* 2003] Washington Y Ochieng, Mohammed Quddus and Robert B Noland. *Map-matching in complex urban road networks.* Revista Brasileira de Cartografia, vol. 55, no. 2, 2003. 17

[OICA 2018] Organisation Internationale des Constructeurs d' OICA. *Automobiles.(2018b). World vehicles in use–all vehicles. OICA*, 2018. 1

[Orda & Rom 1990] Ariel Orda and Raphael Rom. *Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length*. Journal of the ACM (JACM), vol. 37, no. 3, pages 607–625, 1990. 78

[Osogami & Raymond 2013] Takayuki Osogami and Rudy Raymond. *Map matching with inverse reinforcement learning*. In Twenty-Third International Joint Conference on Artificial Intelligence, 2013. 17

[Pan *et al.* 2013] J Pan, I S Popa, K Zeitouni and C Borcea. *Proactive Vehicular Traffic Rerouting for Lower Travel Time*. IEEE Transactions on Vehicular Technology, vol. 62, no. 8, pages 3551–3568, oct 2013. 29

[Pyrga *et al.* 2004] Evangelia Pyrga, Frank Schulz, Dorothea Wagner and Christos D Zaroliagis. *Experimental Comparison of Shortest Path Approaches for Timetable Information*. In ALENEX/ANALC, pages 88–99. Citeseer, 2004. 12

[Pyrga *et al.* 2008] Evangelia Pyrga, Frank Schulz, Dorothea Wagner and Christos Zaroliagis. *Efficient models for timetable information in public transportation systems*. Journal of Experimental Algorithmics (JEA), vol. 12, pages 2–4, 2008. 12, 13, 27

[Quddus & Washington 2015] Mohammed Quddus and Simon Washington. *Shortest path and vehicle trajectory aided map-matching for low frequency GPS data*. Transportation Research Part C: Emerging Technologies, vol. 55, pages 328–339, 2015. 36

[Quddus *et al.* 2003] Mohammed A Quddus, Washington Yotto Ochieng, Lin Zhao and Robert B Noland. *A general map matching algorithm for transport telematics applications*. GPS solutions, vol. 7, no. 3, pages 157–167, 2003. 16

[RafałKucharski & Gentile 2019] RafałKucharski and Guido Gentile. *Simulation of rerouting phenomena in Dynamic Traffic Assignment with the Information Comply Model*. Transportation Research Part B: Methodological, vol. 126, pages 414–441, 2019. 53

[Sanaullah *et al.* 2016] Irum Sanaullah, Mohammed Quddus and Marcus Enoch. *Developing travel time estimation methods using sparse GPS data*. Journal of Intelligent Transportation Systems, vol. 20, no. 6, pages 532–544, 2016. 29

[Sanders & Schultes 2007] Peter Sanders and Dominik Schultes. *Engineering fast route planning algorithms*. In International Workshop on Experimental and Efficient Algorithms, pages 23–36. Springer, 2007. 2, 10

[Sanders & Schulz 2012] Peter Sanders and Christian Schulz. *Distributed evolutionary graph partitioning.* In 2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX), pages 16–29. SIAM, 2012. 75

[Schrank *et al.* 2015] David Schrank, Bill Eisele, Tim Lomax and Jim Bak. *Scorecard, Urban Mobility.* Technical report, The Texas A&M Transportation Institute and Inrix, \url{https://tti.tamu.edu/documents/mobility-scorecard-2015.pdf}, 2015. 28

[Sedgewick & Vitter 1984] R Sedgewick and J S Vitter. *Shortest Paths In Euclidean Graphs.* In 25th Annual Symposium onFoundations of Computer Science, 1984., pages 417–424, 1984. 84, 89

[Shi *et al.* 2019] Ge Shi, Jie Shan, Liang Ding, Peng Ye, Yang Li and Nan Jiang. *Urban road network expansion and its driving variables: a case study of Nanjing City.* International journal of environmental research and public health, vol. 16, no. 13, page 2318, 2019. 1

[Sładkowski & Pamuła 2016] Aleksander Sładkowski and Wiesław Pamuła. Intelligent transportation systems-problems and perspectives, volume 303. Springer, 2016. 1

[Smith *et al.* 2014] David Smith, Soufiene Djahel and John Murphy. *A SUMO based evaluation of road incidents' impact on traffic congestion level in smart cities.* Proceedings - Conference on Local Computer Networks, LCN, vol. 2014-Novem, no. November, pages 702–710, 2014. 31

[Strasser 2017] Ben Strasser. *Dynamic time-dependent routing in road networks through sampling.* In 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. 97

[Systematics 2005] Cambridge Systematics. *Traffic congestion and reliability: Trends and advanced strategies for congestion mitigation.* Technical report, United States. Federal Highway Administration, 2005. 29

[TFA 2020] Transportation for America TFA. *The Congestion Con. How more lanes and more money equals more traffic*, 2020. 1

[Thomson & Richardson 1995] Robert C Thomson and Dianne E Richardson. *A graph theory approach to road network generalisation.* In Proceeding of the 17th international cartographic conference, pages 1871–1880, 1995. 8, 68

[Uppoor *et al.* 2013] Sandesh Uppoor, Oscar Trullols-Cruces, Marco Fiore and Jose M Barcelo-Ordinas. *Generation and analysis of a large-scale urban vehicular mobility dataset.* IEEE Transactions on Mobile Computing, vol. 13, no. 5, pages 1061–1075, 2013. 65

[Uppoor *et al.* 2014] Sandesh Uppoor, Oscar Trullols-Cruces, Marco Fiore and Jose M Barcelo-Ordinas. *Generation and analysis of a large-scale urban vehicular mobility dataset.* IEEE Transactions on Mobile Computing, vol. 13, no. 5, pages 1061–1075, 2014. 30, 31, 50, 58

[van Diggelen & Enge 2015] Frank van Diggelen and Per Enge. *The World's first GPS MOOC and Worldwide Laboratory using Smartphones* . In International Technical Meeting of The Satellite Division of the Institute of Navigation (GNSS), pages 361–369. ION, 2015. 35

[Von Watzdorf & Michahelles 2010] Stephan Von Watzdorf and Florian Michahelles. *Accuracy of positioning data on smartphones.* In Proceedings of the 3rd International Workshop on Location and the Web, pages 1–4, 2010. i, ii, 16

[Wang *et al.* 2013] Shen Wang, Soufiene Djahel, Jennifer Mcmanis, Cormac Mckenna and Liam Murphy. *Comprehensive Performance Analysis and Comparison of Vehicles Routing Algorithms in Smart Cities.* In Global Information Infrastructure Symposium (GIIS), pages 1–8, Trento, Italy, 2013. 31

[Wang *et al.* 2015] C Wang, J Pan, H Xu, J Jia and Z Meng. *An Improved A* Algorithm for Traffic Navigation in Real-Time Environment.* In International Conference on Robot, Vision and Signal Processing (RVSP), pages 47–50, nov 2015. 51

[Wang *et al.* 2016] Shen Wang, Soufiene Djahel, Zonghua Zhang and Jennifer McManis. *Next Road Rerouting: A Multiagent System for Mitigating Unexpected Urban Traffic Congestion.* IEEE Transactions on Intelligent Transportation Systems, vol. 17, no. 10, pages 2888–2899, 2016. 30

[Wang *et al.* 2019] Hongjian Wang, Xianfeng Tang, Yu-Hsuan Kuo, Daniel Kifer and Zhenhui Li. *A Simple Baseline for Travel Time Estimation Using Large-scale Trip Data.* ACM Trans. Intell. Syst. Technol., vol. 10, no. 2, pages 19:1—-19:22, jan 2019. 30

[White *et al.* 2000] Christopher E White, David Bernstein and Alain L Kornhauser. *Some map matching algorithms for personal navigation assistants.* Transportation research part c: emerging technologies, vol. 8, no. 1-6, pages 91–108, 2000. 16

[Williams 1964] John William Joseph Williams. *Algorithm 232: heapsort.* Commun. ACM, vol. 7, pages 347–348, 1964. 19

[Winslett 2019] Christopher Winslett. *OneWeb Satellites.* 2019. 99

[Winter 2002] Stephan Winter. *Modeling costs of turns in route planning.* GeoInformatica, vol. 6, no. 4, pages 345–361, 2002. 10

[Wu *et al.* 2016] Yao-Jan Wu, Feng Chen, Chang-Tien Lu and Shu Yang. *Urban Traffic Flow Prediction Using a Spatio-Temporal Random Effects Model.* Journal of Intelligent Transportation Systems, vol. 20, no. 3, pages 282–293, 2016. 49

[Xiaogang *et al.* 2016] QI Xiaogang, MA Jiulong, WU Dan, LIU Lifang and HU Shaolin. *A survey of routing techniques for satellite networks.* 2016. 98

[Xu *et al.* 2015] Ming Xu, Yiman Du, Jianping Wu and Yang Zhou. *Map matching based on conditional random fields and route preference mining for uncertain trajectories.* Mathematical Problems in Engineering, vol. 2015, 2015. 17

[Yin & Wolfson 2004] Huabei Yin and Ouri Wolfson. *A weight-based map matching method in moving objects databases.* In Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004., pages 437–438. IEEE, 2004. 16

[Yu *et al.* 2010] Meng Yu*et al.* *Improved positioning of land vehicle in ITS using digital map and other accessory information.* 2010. 16

[Zhao *et al.* 2008] Liang Zhao, Tatsuya Ohshima and Hiroshi Nagamochi. *A* Algorithm for the time-dependent shortest path problem.* In WAAC08: The 11th Japan-Korea Joint Workshop on Algorithms and Computation, 2008. 10

[Zhu *et al.* 2017] Lei Zhu, Jacob R Holden and Jeffrey D Gonder. *Trajectory segmentation map-matching approach for large-scale, high-resolution GPS data.* Transportation Research Record, vol. 2645, no. 1, pages 67–75, 2017. 16

[Ziliaskopoulos *et al.* 1999] Athanasios Ziliaskopoulos, Steven Waller and Curtis Barrett. *VISTA, Visual Interactive System for Transportation Algorithms.* Technical report, UC Berkeley Transportation Library, 1999. 30