Isochrones in Multimodal Spatial Networks

Ph.D. Thesis in Computer Science

Doktorarbeit in Informatik

Tesi di Dottorato di Ricerca in Informatica

Markus Innerebner

Relatore - *Doktorvater* - Faculty Advisor: Prof. Dr. Johann Gamper

Approved by: Prof. Günther Specht Full Professor University of Innsbruck (chair), and Dr. Markus Schedl, assistant professor at the University of Linz, and Prof. Paolo Ciancarini, full professor at the University of Bologna, and Dr. Domenico Lembo, assistant professor at the University "La Sapienza" of Rome, and Prof. Alessandro Artale, internal member of FUB, Faculty of CS Bozen/Bolzano

Keywords: Isochrones, Isochrone Queries, Spatial Network Queries, WebGIS, Multimodal Networks

ACM categories: DATA STRUCTURES, Algorithms, Spatial databases and GIS

Copyright © 2013 by Markus Innerebner

Acknowledgments

First and foremost, I want to thank Prof. Dr. Johann Gamper for assuming the supervision after the exit of my previous supervisor. He has taught me, both consciously and unconsciously, how deepened research in the field of computer science is done. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. The joy and enthusiasm he has for his research was contagious and motivational for me, even during tough times in the Ph.D. pursuit.

Second, I would like to acknowledge honorary my previous supervisor Prof. Dr. Michael Böhlen who assisted me the first two years. He has instructed me intensively in the field of scientific research by emphasizing in particular to perform precise and autonomous research.

Third, I would like to thank all off my group members for their helpful inputs during the inspirational discussion in our seminars or in the coffee breaks. In particular I am especially grateful to Dr. Nikolaus Augsten for his valuable contribution.

I am grateful to Roberto Loperfido from the Municipality of Bozen/Bozen for providing the urban data and Tobias Ebner from the transportation agency SASA for delivering me the schedules and the stop stations. I would like to express thanks to Wolfgang Moser and Ivo Planötscher from the Department 9.6 of the Autonomous Province of Bozen/Bozen for providing me the entire street network of the region South Tyrol and Markus Lauer from the transportation agency SII for serving me the schedules of all transportation systems in the entire region.

Furthermore, I want to express gratitude to Marc Zebisch, head of the Institute of Applied Remote Sensing from EURAC research, for offering me flexibility in the working hours of my side job in the years 2012/2013.

Special thanks go to my girlfriend Monika for encouraging me over the final Ph.D. period and for her indefatigable effort in correcting and improving my written English.

Lastly, I would like to thank my family for all their love and encouragement. For my parents, Margaret and Martin, who raised me with a love of science and supported me in all my pursuits in my entire life. Thank you.

Abstract

The aim of this thesis is to design and develop a system for the computation of isochrones in multimodal spatial networks that can be used to conduct various types of geospatial reachability analysis. Such kind of analysis provide an important tool in many application areas, including environmental and life sciences, epidemiology, social sciences, medicine, emergency management or city planning.

We introduce and formally define isochrones for multimodal networks, which can be classified as continuous or discrete along the space and the time dimension respectively. An isochrone in a multimodal spatial network is defined as a possibly disconnected subgraph that covers all space points in the network from where a query point q is reachable within a given time span and by a given arrival time at q. Isochrones are a new query type in spatial network databases and provide a useful instrument to perform reachability analysis.

We propose efficient and scalable algorithms that mainly differ in the trade-off between memory consumption and runtime performance. The completely main-memory based algorithm MDijkstra loads the entire network in memory and is therefore limited by the available memory. To make the computation scalable in terms of memory usage, we propose the incremental network expansion algorithm MINE. The space complexity of MINE is independent of the network size and depends only on the size of the isochrone. The runtime is determined by the incremental loading of the relevant network portions. In terms of data transfer, MINE is optimal because only those network portions are loaded in memory that eventually will be part of the isochrone. To further reduce the memory requirements, we introduce the concept of vertex expiration, which eagerly expires the isochrone and keeps in memory only the minimal set of expanded vertices that is necessary to avoid cyclic expansions. Vertex expiration is implemented in the algorithm MINEX, which reduces the memory requirements from $\mathcal{O}(|V^{iso}|)$ to $\mathcal{O}(\sqrt{|V^{iso}|})$

for grid and $\mathcal{O}(1)$ for spider networks, respectively. To improve the runtime efficiency of MINEX, we propose a hybrid approach, termed MRNEX. Instead of loading the network edge by edge during network expansion, the algorithm reads small chunks of the network and performs network expansion in main memory. MRNEX significantly reduces the I/O costs, whereas the memory usage is thanks to vertex expiration only slightly higher than for MINEX. Thus, MRNEX is a good trade-off between memory complexity and runtime efficiency.

We conduct a detailed empirical evaluation of our algorithms using both synthetic data and real-world data with different network topologies, covering city networks that have a more regular structure as well as skewed regional networks. The experiments confirm the analytical results on synthetic data and show that for real-world data the memory requirements are very small indeed.

Finally, we implemented a client-server system, termed *ISOGA*, which uses the isochrone algorithms in combination with a statistical component to conduct various types of geospatial analysis. The use of *ISOGA* is illustrated in various real-world application scenarios, such as in urban planning to analyze the coverage of a city with public services. We plan to deploy the *ISOGA* system to local institutions with the aim to get feedback for further improvements.

Zusammenfassung

Das Ziel dieser Arbeit ist die Konzeption und Entwicklung eines Systems für die Berechnung von Isochronen in multimodalen räumlichen Netzwerken, welche für verschiedene räumlichen Erreichbarkeitsanalysen eingesetzt werden können. Solche Analysen bieten ein wichtiges Anwendungsinstrument in vielen Bereichen, wie etwa in Umwelt-und Biowissenschaften, Epidemiologie, soziale Naturwissenschaften, Medizin, Zivilschutz oder Städteplanung.

Zuerst werden Isochrone in multimodalen Netzwerken eingeführt, welche kontinuierlich oder diskret bezüglich der Raum-Zeit Dimensionen sein können. Eine Isochrone in einem multimodalen räumlichen Netzwerk ist definiert als ein möglicherweise entkoppelter Untergraph, der jene räumlichen Punkte beinhaltet, die einen gewissen Anfragepunkt q, innerhalb einer gewissen Zeitspanne und zu einem bestimmtem Zeitpunkt erreichen. Eine Isochronenanfrage kann als eine neue Art von Anfrage in räumlichen Datenbanken angesehen werden. Nebenbei bietet sie ein nützliches Instrument für Erreichbarkeitsanalysen.

Wir präsentieren effiziente und skalierbare Algorithmen, welche sich hauptsächlich im Trade-off zwischen Speicherbedarf und Laufzeit unterscheiden. Der Algorithmus MDijkstra, eine auf den Hauptspeicher basierte Implementierung, lädt das gesamte Netzwerk in den Hauptspeicher und wird daher durch den zu verfügbaren Speicher begrenzt. Um die Berechnung von Isochronen skalierbar im Speicherverbrauch zu machen, wurde der netzwerk-basierte, inkrementelle Expansionsalgorithmus MINE entwickelt, dessen Speicherbedarf unabhängig vom Netzwerk ist und nur von der Größe der Isochrone abhängt. MINE ist optimal, dadurch dass nur jener Bereich des Netzwerkes in den Speicher geladen wird, der die Isochrone repräsentiert.

Zur weiteren Reduzierung des Speicherbedarfs führen wir das Konzept der Vertex Expiration (Erlöschen eines Knotens) ein, in welchem eifrig untersucht wird, welche Knoten aus der Isochrone in zukünftige Expansionsschritten nicht mehr

erreicht werden können und infolgedessen als erloschen markiert werden. Dadurch wird im Hauptspeicher nur eine minimale Anzahl von Knoten gehalten, um zu garantieren, dass keine Expansionszyklen auftreten können, ohne dabei die Korrektheit des Ergebnisses zu verletzen. Die Speicherkomplexität des Algorithmus (MINEX) entspricht $\mathcal{O}(|V^{iso}|)$ bis $\mathcal{O}(\sqrt{|V^{iso}|})$ für gitter-förmige Netzwerke und $\mathcal{O}(\sqrt{|V^{iso}|})$ bis $\mathcal{O}(1)$ für spinnen-förmige Netzwerke, wobei $V^{iso}|$ die Anzahl der Knoten in der Isochronen ist.

Um die Laufzeit des Algorithmus von MINEX zu verbessern, wird ein hybrider Ansatz vorgeschlagen namens MRNEX. Anstatt in jedem Expansionschritt nur die Netzwerkinformationen der benachbarten Knoten zu laden, lädt MRNEX mit einem einzigen Datenbankzugriff grössere Teile des Netzwerks, um diese dann im Hauptspeicher zu expandieren. Dadurch reduzieren sich drastisch die I/O Kosten, während die Speicheranforderungen nur geringfügig höher ausfallen. Deswegen ist MrneX ein guter Kompromiss zwischen Speicher Komplexität und Laufzeit Performance.

Es wurde eine detaillierte empirische Studie durchgeführt, in welcher die unterschiedlichen Algorithmen mit synthetischen und Realdaten hinsichtlich Laufzeitverhalten und Speicherbedarf ausgewertet wurden. Die Experimente, welche auf den synthetischen Daten ausgeführt worden sind, bestätigen die analytischen Ergebnisse über den Speicherbedarf. Auch der in den Realdaten erforderte Speicherbedarf ist nur geringfühgig höher.

Schlussendlich wird eine Client-Server Anwendung mit der Bezeichnung *ISOGA* entwickelt, in welchem diese Algorithmen gemeinsam mit einer statistischen Komponente für die Analyse von räumlichen Erreichbarkeitssuchen eingesetzt werden können. Die Verwendung des Systems *ISOGA* kann in verschieden Anwendungsszenarien eingesetzt werden, so z.B. in der Städteplanung, um die Abdeckung von wichtigen öffentlichen Gebäuden zu analysieren. In Zukunft soll *ISOGA* lokalen Körperschaften, wie Gemeinde oder Provinz zu Verfügung gestellt werden, um Feedback über Verbesserungsvorschläge und eventuell neue Anwendungsszenarien zu erhalten.

Riassunto

Lo scopo di questa tesi è di progettare e sviluppare un sistema per il calcolo di isocrone in reti spaziali multimodali che possa essere utilizzato per effettuare diversi tipi di analisi di raggiungibilità geospaziale. Tale tipo di analisi fornisce un importante strumento per molti settori applicativi, tra cui le scienze ambientali e della vita, l'epidemiologia, le scienze sociali, la medicina e la gestione delle emergenze nell'ambito dell'urbanistica. In questa tesi sono introdotte e definite in modo formale le isocrone per reti multimodali, che vengono classificate come continue o discrete lungo le dimensioni spaziali e temporali. Una isocrona in una rete multimodale spaziale è definita come un sottografo possibilmente disconesso che copre tutti i punti nello spazio della rete da cui un punto di interesse q è raggiungibile in un dato lasso temporale e non oltre ad un orario di arrivo specificato. Le isocrone possono essere viste come un nuovo tipo di query nell'ambito delle banche dati spaziali; in quanto esse forniscono uno strumento utile per eseguire analisi di raggiungibilità. Questo tesi propone diversi algoritmi, efficienti e scalabili, che si differenziano nel trade-off tra il consumo di memoria e le prestazioni di runtime.

L'algoritmo MDijkstra carica l'intera rete in memoria ed è quindi limitato dalla memoria disponibile. Per rendere il calcolo scalabile in termini di utilizzo di memoria, viene introdotto l'algoritmo MINE il quale carica ed espande la rete in memory in modo incrementale.

La complessità di memoria di MINE è indipendente dalla dimensione della rete e dipende esclusivamente dalla dimensione dell'isocrona. La complessità di runtime è determinata dal caricamento incrementale delle porzioni di rete rilevanti. MINE è ottimale nel senso che vengono caricate in memoria solo le porzioni di rete che alla fine formeranno l'isocrona.

Per ridurre ulteriormente i requisiti di memoria, si introduce il concetto di vertex expiration (scadenza dei vertici), implementato nell'algoritmo MINEX. Questo

algoritmo identifica e marca come scaduti i vertici che in future iterazioni non possono essere più raggiunti e mantiene in memoria solamente il minimo insieme di vertici necessario per evitare espansioni cicliche. L'algoritmo MINEX riduce la memoria richiesta da $\mathcal{O}(|V^{iso}|)$ a $\mathcal{O}(\sqrt{|V^{iso}|})$ per reti di tipologia in forma a griglia e da $\mathcal{O}(\sqrt{|V^{iso}|})$ a $\mathcal{O}(1)$ per reti di forma di ragnatela.

Per un miglioramento della runtime performance di MINEX, viene proposto l'approccio ibrido MRNEX, nel quale invece di caricare in memoria la rete arco per arco durante l'espansione, vengono caricate piccole porzioni di rete e l'espansione è eseguita in memoria principale. Con questo approccio si riesce a ridurre significativamente i costi di I/O, mentre il consumo della memoria grazie alla vertex expiration è solo leggermente superiore. Quindi l'algoritmo MRNEX è un buon compromesso tra complessità di memoria ed efficienza di runtime. In questa tesi viene eseguita una dettagliata valutazione empirica nella quale vengono analizzati l'utilizzo di memoria e i tempi di esecuzione dei vari algoritmi, sia con dati sintetici sia con dati reali. Gli esperimenti confermano i risultati analitici sui dati di sintesi e dimostrano che con i dati reali l'utilizzo di memoria è molto efficiente. Infine, viene presentato *ISOGA*, un sistema client-server che è stato implementato. *ISOGA* utilizza gli algoritmi sviluppati in combinazione con una componente statistica per effettuare diversi tipi di analisi geospaziale. L'uso di *ISOGA* è illustrato in vari campi d'applicazione, come ad esempio nella pianificazione urbanistica dove è stato usato per analizzare la copertura di una città con i servizi pubblici. Abbiamo in programma di distribuire il nostro sistema *ISOGA* ad enti locali altoatesini, con l'obiettivo di ottenere un feedback per ulteriori miglioramenti e per utilizzarlo in nuovi campi applicativi.

Contents

Acknowledgments **v**

Abstract **vii**

Zusammenfassung **ix**

Riassunto **xi**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**
1.1 Introductory Examples 3
1.2 Contributions 5
1.3 Publications 7
1.4 Thesis Organizations 7

2 Related Work **9**
2.1 Spatial Network Queries 10
2.2 Time-dependent Networks 13
2.3 Multimodal Networks 15
2.4 Routing Applications and Systems 16
2.5 Summary 17

3 Isochrones in Multimodal Spatial Networks **19**
3.1 Multimodal Spatial Networks 20
3.2 Definition of Isochrones 25
3.3 Summary 26

4 Algorithms for the Computation of Isochrones **29**
4.1 Memory-based Algorithm MDijkstra 30
4.1.1 Extensions 32
4.2 Multimodal Incremental Network Expansion 33
4.2.1 Algorithm MINE 33
4.2.2 Algorithm MINEX 35
4.2.3 Vertex Expiration 37
4.2.4 Properties 41
4.3 A Hybrid Approach 43
4.3.1 Multimodal Range Network Expansion 44
4.3.2 Setting an Upper Bound for Range Queries 48
4.4 Summary 51

5 Experimental Evaluation **53**
5.1 Data Sets 54
5.2 Setup 55
5.3 Memory Experiments 56
5.3.1 Synthetic Data 56
5.3.2 Real World Data 58
5.4 Runtime Experiments 59
5.4.1 Varying the Location of the Query Points 63
5.4.2 Varying the Arrival Time at the Query Point 64
5.5 Summary 65

6 A System for Geographic Reachability Analysis **67**
6.1 System Components 68
6.1.1 Computing Isochrones 68
6.1.2 Creating a Surface Around Isochrones 69
6.1.3 Computing Statistics with Isochrones 73
6.2 Architecture 74
6.2.1 Presentation Tier 74
6.2.2 Logical Tier 76
6.2.3 Data Tier and Data Model 77
6.3 Conceptual Design for Mobile Devices 79
6.4 Application Scenarios 81
6.4.1 Vertex Expiration. 83

6.5 Summary 84

7 Conclusions **85**
7.1 Summary 85
7.2 Future Research Directions 86

Bibliography **87**

List of Figures

1.1 Five Isochrones in Edinburgh. 2
1.2 Reachability Analysis of Primary Schools. 3
1.3 Apartments within a 25 minutes Isochrone. 4
1.4 Evacuation Scenario for Natural Disasters. 5

3.1 Multimodal Network. 21
3.2 Multimodal Path. 23
3.3 Isochrone with d_{max}=5min, s=2m/s and t = 06:06:00 26

4.1 Schedule Representations. 32
4.2 Network Expansion in MINE. 35
4.3 Step-by-step Network Expansion of MINEX for $d_{max} = 5$ min, $s =$ 2 m/s, and t_q = 06:06:00. 38
4.4 LRU Strategy. 41
4.5 Network Expiration. 41
4.6 Multimodal Range Network Expansion. 45
4.7 Multimodal Range Network Expansion. 46
4.8 Illustrating Range Query Expansion on two Real-world Data Sets. . 48
4.9 Precomputed Query Ranges for Vertices. 49
4.10 Precomputing the 4-NN for Vertex v_0. 51

5.1 Network Topologies Used in the Experiments. 55
5.2 Memory Requirements in Synthetic Networks with Central q. 57
5.3 Memory Depending on the Location of q. 57
5.4 Memory Requirements on Real World Data. 59
5.5 Vertex Expiration — 10/20/30 min. 60

5.6 Number of Loaded Tuples. 61
5.7 False Positive Edges in Ranges Queries. 61
5.8 Runtime on Real World Data. 62
5.9 Break-even Point and Network Independence. 62
5.10 Central Query Points. 63
5.11 Peripheral Query Points. 63
5.12 Frequency Histograms of Transportation Systems. 64
5.13 Low-frequent Time Intervals. 65
5.14 Medium-frequent Time Intervals. 65
5.15 High-frequent Time Intervals. 65

6.1 Isochrone Area Generation Using *LBA* Approach. 69
6.2 Intialization: visit edge (v_0, v_1). 72
6.3 Illustration of SBA^* algorithm (1). 72
6.4 Illustration of SBA^* algorithm (2). 72
6.5 Isochrone Area Generation Using SBA^* Approach. 73
6.6 Query Tree with Workflow. 73
6.7 Architecture. 74
6.8 Isochrone Request. 75
6.9 Map(WMS) Request. 75
6.10 Feature(WFS) Request. 76
6.11 Analysis Request. 77
6.12 Design of a Mobile Architecture. 80
6.13 Using Isochrones in a Flat Search Scenario. 81
6.14 Joining Flats and Isochrones. 82
6.15 Reachability of Primary Schools. 82
6.16 Percentage of Schools Kids in a 15 min. Isochrone. 83
6.17 Vertex Expiration in MINEX. 84

List of Tables

3.1 Example of a Schedule. 21

5.1 Real-World Data Sets: Italy (IT), South Tyrol (ST), and San Francisco (SF), and Bozen-Bolzano (BZ). 54
5.2 Clustering of Arrival Time Intervals. 64

6.1 Vertex Table. 77
6.2 Edge Table. 77
6.3 Schedule Table. 78
6.4 Daymarker Table. 78
6.5 Transportation System Table. 78
6.6 Spatial SQL Operations. 79

List of Algorithms

1 $MDijkstra(q, d_{max}, s, t, \mathsf{N})$ 31
2 $MINE(q, t, d_{max}, s, \mathsf{N})$. 34
3 $MINEX(q, t, d_{max}, s, \mathsf{N})$ 36
4 $MRNEX(q, t, d_{max}, s, \mathsf{N})$. 47
5 Precomputation of the Query Ranges. 50

6 $SBA^*(E^{iso}, size)$ 70
- Function DFS($root, e, E^{iso}, P, l$) 71

CHAPTER 1

Introduction

The growing popularity of online mapping services such as Google Maps, Bing Maps, or Microsoft MapPoint has led to an increasing interest in offering various types of queries on spatial networks in real time. Typical examples are finding the shortest route between locations or finding the nearest neighbor objects for important facilities (hospitals, schools, etc.). More recently, routing in so-called multimodal networks has received much attention, where multiple means of transportation (e.g. walking in combination with buses, cars, trains, etc.) are considered. While computing the shortest path is a basic building block in almost all kinds of queries in spatial networks, with the broader availability of data and technologies more advanced queries have been studied. In particular, there is a growing focus on reachability queries and analysis.

In this thesis we study *isochrones* as a powerful instrument to conduct various types of reachability analysis in spatial network databases. The term isochrone derives from the Greek words iso ($\iota\sigma o$) = same and chronos ($\chi\rho\acute{o}\nu o\varsigma$) = time. Generally, an isochrone is defined as a line drawn on a map that connects all points at which something occurs or arrives at the same time [65, 67]. In the context of reachability analysis or transportation geography, an isochrone is defined as the set of all space points (i.e. an area) that are reachable from a query point in a given timespan, or vice versa all points from where the query point is reachable in a given timespan. When schedule-based networks are taken into account, such as the public transport system, the arrival/departure time becomes important, and the shape of an isochrone is a possibly disconnected set of areas around the query point (reachable by walking) and around the public transport stops (reachable by a combination of walking and using public transport). Isochrones as an instrument

for reachability analysis were first discussed by Armstrong [1] in 1972 in a study about the accessibility of the airport Hampshire in South East England, where he created time-based travel maps for private motor transport.

Figure 1.1 shows five isochrones (0-15 min, ..., 60-75 min) at 09:00 am with query point q in the center of Edinburgh (black circle)[1]. The red area represents an isochrone in which every location reaches q in less than 15 minutes. The turquoise areas represent peripheral regions from which it takes 60-75 minutes to reach q. There exist also areas that are part of a smaller isochrone, though they are geographically more distant from q. For instance, the town Inverkeithing north-west of Edinburgh belongs to the $15-30$ min isochrone, because there is a train connection at 08:35 am that arrives in Edinburgh at 08:59 am. In contrast, the district Buckstone south of Edinburgh, though geographically closer to q, belongs to the $30-45$ min isochrone; the area is not well covered by public transport systems.

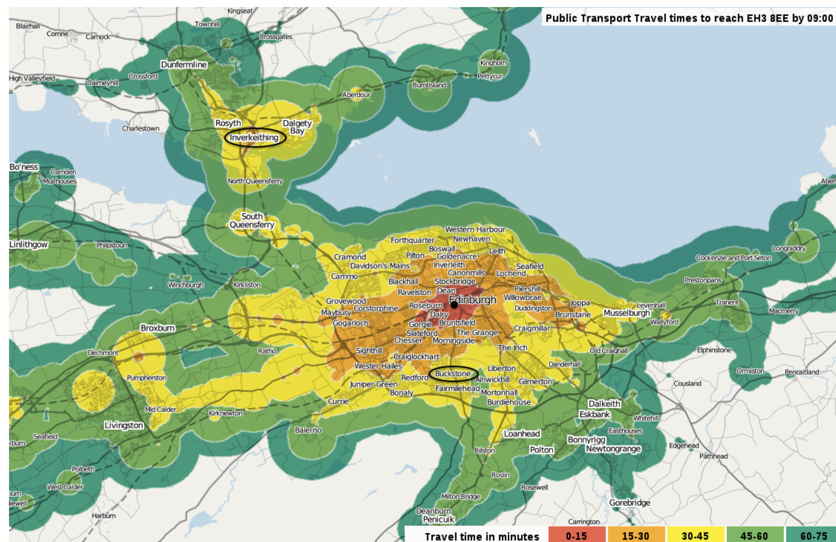


Figure 1.1: Five Isochrones in Edinburgh.

The goal of this thesis is to study isochrones and their computation in multi-modal spatial network databases. In particular, we will formally define isochrones and devise efficient algorithms for the computation of isochrones.

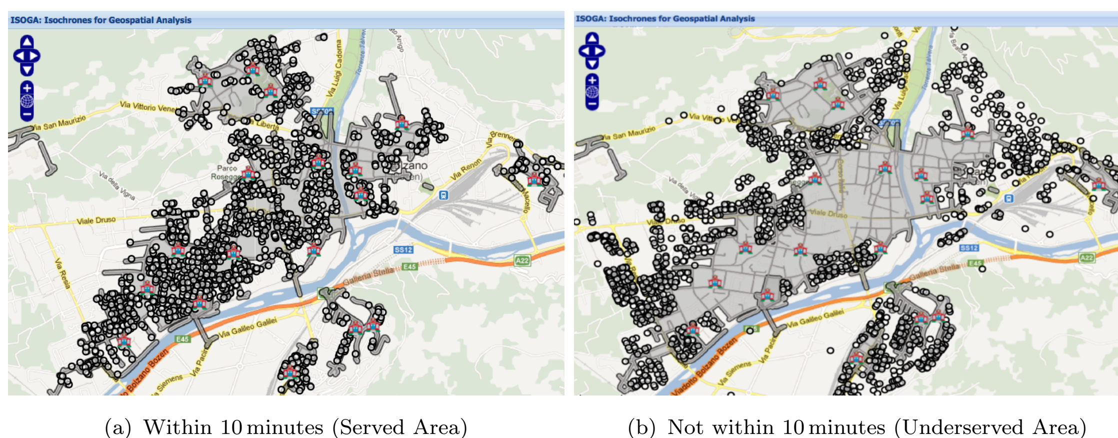
[1] source: http://mapumental.com

1.1 Introductory Examples

We motivate the thesis by discussing three real-world scenarios that emerged from the *eBZ 2015* umbrella initiative in collaboration with the local municipality. More specifically, the project *"Bolzano – 10 minuti da tutto"* had the high level objective that citizens should reach all important services (schools, hospitals, etc.) in less than ten minutes using the public transport system. In this project, isochrones have been used to perform various kinds of reachability analysis.

Isochrones for Accessibility Analysis. In this example, we analyze how well the primary schools in Bozen/Bolzano are reachable by walking in combination with city buses. Figure 1.2(a) illustrates such a scenario. The query points are formed by all primary schools in the city. The gray shaded area represents the 10-minutes isochrone for these query points, i.e. the parts of the city from where the kids reach the closest school in ten minutes or less. By joining the isochrone with the inhabitants database and filtering out the schoolchilds, we can determine the number of kids living in this area, and hence to what degree certain areas are served or not. The small circles in Figure 1.2(a) indicate the buildings in which schoolchilds live who reach their nearby school in less than ten minutes.

Figure 1.2(b) illustrates the result of the opposite query, i.e. areas in which kids do not reach the closest school in less than ten minutes, hence areas that are not well served.



(a) Within 10 minutes (Served Area) (b) Not within 10 minutes (Underserved Area)

Figure 1.2: Reachability Analysis of Primary Schools.

The benefit of such a tool is to easily identify areas with low reachability, where the public transport system needs to be improved, e.g. by incrementing the bus frequency, modifying the bus schedules, or adding additional facilities.

Isochrones for House Hunting Services. In this example in Figure 1.3 we consider a person who is looking for a suitable apartment that is in a specific price range and less than 25 minutes away from his/her working place by using the public transport system. The query point (symbol '*') represents the working place, and the gray area represents the isochrone, where he/she should look for an apartment.

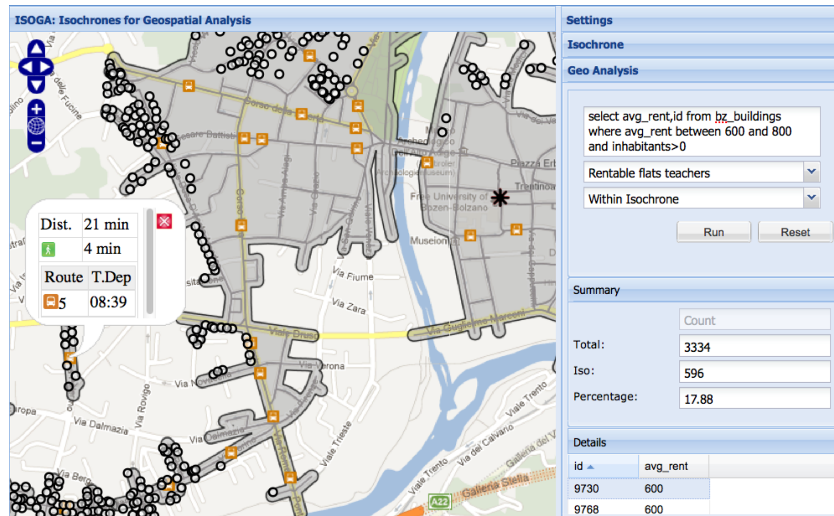


Figure 1.3: Apartments within a 25 minutes Isochrone.

The right-hand side of the screenshot shows the interface that allows the user to formulate an SQL query to select objects that shall be joined with the isochrone. In this example, the query retrieves all apartments/houses with a rent between 600 and 800 Euros (small circles inside the isochrone). It shows also a summary statistics, i.e. the number of apartments that are located in the isochrone, the total number of apartments in the specified price range, and the resulting percentage. The popup on the left-hand side shows additional information about one of the qualifying apartments, such as the indication on how to reach the working place: walk on food for four minutes to the closest bus station and then take the bus line 5 at 8:39 am; the total distance is 21 minutes.

Isochrones for the Analysis of Evacuation Scenarios. The third application scenario emerged from a collaboration with the Civil Defence Department of the Autonomous Province of Bozen/Bolzano. In the case of a natural disaster in urban areas, it is very important to be able to carry a large number of people in a short time to the closest safe refuge place. To develop an emergency plan for such situations, the following aspects need to be considered: Which type of buildings can be used as refuge places? Where should they be located? How many new

refuges must additionally be set up in order to guarantee a 100% or close to 100% coverage?

Figure 1.4 illustrates such a scenario, where the buildings in the map acts as refuge places. The gray area denotes a 15 minute isochrone in which 67% of the total inhabitants live. To guarantee a higher coverage additional buildings in the upper left and in the lower part of the map must be set up to approach a maximal coverage.

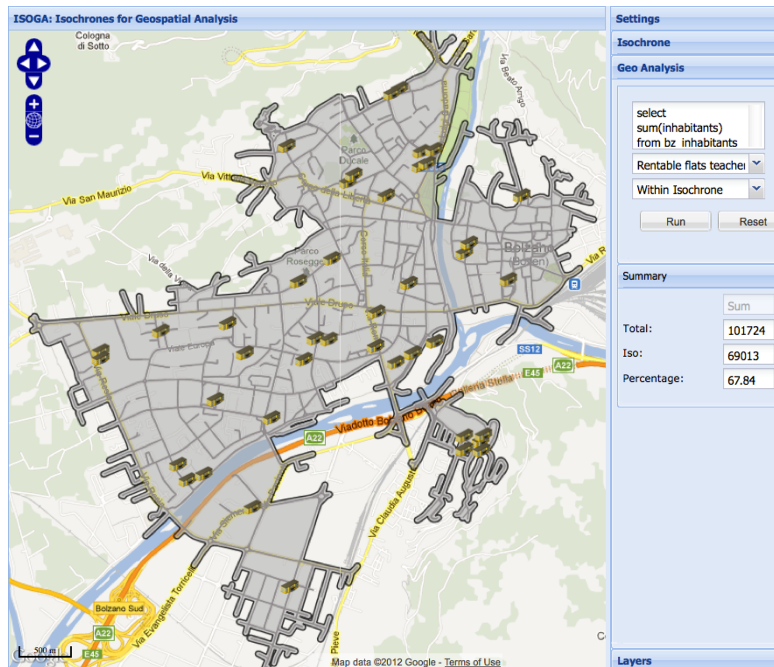


Figure 1.4: Evacuation Scenario for Natural Disasters.

1.2 Contributions

The overall goal of this thesis is to design and develop a system for the computation of isochrones in multimodal spatial networks, which can be used for various types of geospatial reachability analysis. More specifically, the technical contributions are as follows:

- We provide a *formal definition of isochrones* for multimodal spatial networks with different transport modes that can be discrete or continuous in space and time, respectively. Isochrone queries form a new query type in spatial network databases.

- We developed three different *algorithms for the computation of isochrones.* They all use an incremental network expansion strategy, but differ in the memory structure (disk vs. memory based). The first algorithm, *MDijkstra* (*Multimodal Dijkstra Network Expansion*), requires $\mathcal{O}(|V|)$ memory since the entire network is loaded in memory.

- To reduce the memory requirements, the second algorithm *MINEX* (*Multimodal Incremental Network Expansion*) incrementally loads only those parts of the network that will eventually form the isochrone. The algorithm is independent of the network size and depends only on the size of the isochrone, i.e. requires $\mathcal{O}(\sqrt{|V^{iso}|})$ memory, where $|V^{iso}|$ is the number of vertices in the isochrone. To avoid to keep the entire isochrone in memory, we propose *vertex expiration* that identifies a minimal set of vertices that need to be kept in memory in order to avoid cyclic expansions. Vertex expiration reduces the memory requirements from $\mathcal{O}(|V^{iso}|)$ to $\mathcal{O}(\sqrt{|V^{iso}|})$ for grid and $\mathcal{O}(1)$ for spider networks, respectively.

- To improve the runtime efficiency, the algorithm *MRNEX* (*Multimodal Range Network Expansion*) adopts a hybrid approach. Instead of loading the network edge by edge during network expansion, the algorithm reads small chunks of the network and performs network expansion in memory. Whenever during network expansion a vertex is not in memory, a small chunk of the network around the expanding vertex is loaded. MRNEX significantly reduces the I/O costs, while the memory usage is only slightly higher than in MINEX, and it can be controlled by specifying the size of the chunks in combination with vertex expiration. MRNEX provides a good tradeoff between memory complexity and runtime efficiency.

- We conducted detailed *empirical evaluations* both on synthetic as well as real-world datasets, representing urban, regional, and national networks. The experiments confirm the analytical results about the scalability of our solutions in terms of memory and runtime complexity.

- We implemented a Web-based *prototype system* which combines the computation of isochrones with a statistics component and provides a useful instrument to perform various kinds of geospatial reachability analysis.

1.3 Publications

Most of the scientific results presented in this thesis have been published in the following conferences:

M. Innerebner, M. H. Böhlen, and I. Timko. A Web-enabled Extension of a Spatio-temporal DBMS. In *Proc. of the 15th ACM International Symposium on Geographic Information Systems (ACM-GIS 2007)*, pages 34–41. ACM, 2007.

J. Gamper, M. H. Böhlen, W. Cometti, and M. Innerebner. Defining Isochrones in Multimodal Spatial Networks. In *Proc. of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011)*, pages 2381–2384, 2011.

J. Gamper, M. H. Böhlen, and M. Innerebner. Scalable Computation of Isochrones with Network Expiration. In *Proc. of the 24th International Conference on Scientific and Statistical Database Management (SSDBM 2012)*, pages 526–543, 2012.

M. Innerebner, M. H. Böhlen, and J. Gamper. *ISOGA*: A System for Geographical Reachability Analysis Enhanced with Statistics. In *Proc. of the 12th International Symposium on Web and Wireless Geographical Information Systems (W2GIS 2013)*, pages 180–189, 2013.

1.4 Thesis Organizations

Chapter 2. This chapter discusses related research work in the area of query processing in spatial, time-dependent, and multimodal networks. Since our prototype system is an essential part of the thesis, we investigate also state-of-the-art implementations with similar functionalities.

Chapter 3: In this chapter, we first introduce basic concepts about multimodal networks which allow to represent several transport systems with different transportation modes in a single network. Then, we provide a formal definition of isochrones in such networks.

Chapter 4: This chapter presents three algorithms for the computation of isochrones in multimodal networks: the memory-based MDijkstra algorithm, the disk-based MINEX algorithm with vertex expiration, and the hybrid MRNEX algorithm. The concept of vertex expiration is described in detail, and we provide analytical results about the memory complexity for various types of network structures.

Chapter 5: In this chapter we perform a detailed empirical evaluation on synthetic and real-world data. We measure the memory and runtime requirements of our algorithms. The experiments confirm the analytical results and show the scalability of the proposed algorithms.

Chapter 6: This chapter presents a prototype implementation, termed *ISOGA*, which combines the computation of isochrones with a statistical component to provide a tool for geospatial analysis. We describe the system architecture of the Web-based client-server system and discuss several application scenarios.

Chapter 7: This chapter summarizes the achieved results and points out some interesting directions for future research work.

CHAPTER 2

Related Work

In this chapter we discuss related research work, which we divide in four parts. The first part in Section 2.1 reviews research about query processing in spatial network databases, where different types of queries have been proposed and studied. The second part in Section 2.2 is dedicated to time-dependent networks, where the edge costs depend on the time. While most of the research in the past has been done on unimodal networks, the third part in Section 2.3 discusses work on routing algorithms that consider different transportation modalities. Finally, Section 2.4 provides an overview about open source and commercial routing applications.

2.1 Spatial Network Queries

Different query types have been studied for spatial network databases, in which the network distance is used as metric instead of the Euclidean distance [15, 16, 49]. The most studied problem is the shortest path problem (SP), with Dijkstra's influencing SP algorithm [16]. The algorithm finds the shortest path from a given source vertex v_s to a given destination vertex v_d and works as follows: each vertex is assigned a tentative cost, which is initially zero for the source vertex and ∞ for all other vertices in the network. Starting from the source vertex, the algorithm visits all outgoing edges. For each outgoing edge, it checks weather by traversing the current edge the adjacent vertex can be reached with a lower cost. If this is the case, the cost computed so far is updated to the new cost. Once all outgoing edges are visited, the vertex is called expanded. In the next round, denoted expansion phase, the vertex with the smallest tentative cost that has not yet been expanded is examined. The expansion terminates when the target vertex is encountered.

There exist different search techniques to accelerate Dijkstra's SP algorithm. The most straightforward optimization is the *bidirectional search* [12]. An additional search is started from the destination vertex that can be run in parallel. The algorithm stops as soon as both searches meet each other.

A^*-search [26] is a *goal directed search* strategy that improves over Dijkstra's algorithm by using a lower bound estimate of the shortest path, which is computed as the actual network distance from v_s to the current vertex plus the Euclidean distance from the current vertex to v_d. This yields a more informed and directed search towards the target vertex.

A very efficient form of goal directed search is the *Arc-Flags* technique [64, 37, 43] that partitions the graph into cells and attaches a label to each edge. A label contains a flag for each cell. The flag indicates whether a shortest path starting with this edge exists to the corresponding cell. As a result, a bidirectional Arc-Flags-Dijkstra visits only those edges that lie on the SP of a long-range query.

Approximation techniques based on precomputed distances to so-called *landmarks* [62, 20, 21, 52, 23] have been widely studied. The ALT algorithm introduced by Goldberg et al. [20, 21] precomputes the distances using the A^*-algorithm (A) from a set of selected landmark vertices (L) to all other vertices and uses then the triangle inequality (T) to estimate the SP between two queried vertices. Tretyakov et al. [63] improved the accuracy of the SP landmark estimation by introducing shortest path trees for each landmark, instead of simply storing the distances to landmarks.

Contraction hierarchies [19, 5, 55] are an extension of Arc-Flags. This approach uses a routine that iteratively removes unimportant nodes and adds new edges to preserve the correct distances between the remaining nodes. By visiting a much smaller amount of vertices, fast results are achieved in directional SP searches.

Hierarchical search methods [33, 34, 35] structure a large network into smaller subgraphs. To reduce the memory consumption of storing the shortest path between all pairs of vertices, the network is divided in partitions and only the shortest path between the boundary vertices of each partition are stored. Based on these shortcuts, a second subnetwork is built in which the cardinality of connections is much smaller. It consists of the shortcuts and edges that connect vertices of different hierarchies. However, finding an optimal network partitioning is known to be an NP-hard problem [18]. Lee et al. [40] improve the space complexity by storing the subnetworks in a flattened structure rather than storing them in a hierarchical structure.

Other techniques rely on space-driven SP precomputation. Examples are the partitioning into Voronoi regions and precomputing distances within and across regions. A Voronoi diagram divides the space into a number of regions such that in each region there exist a specific point, called generator point, which is the nearest neighbor for all other points in that region. VN3 [38] computes a Voronoi diagram that consists of several Voronoi cells, where each of them represents the region of the nearest neighbor in the network. Xuan et al. [71, 70] extend previous work Voronoi diagrams to answer range queries.

For the computation of k-nearest neighbor queries in a spatial network, Samet et al. [56] use a space-driven materialization technique for storing the shortest path in quadtrees. In a quadtree the search space is recursively divided into quadrants until the number of data points in each quadrant fits in a single page. The algorithm is based on precomputing the shortest paths between all possible vertices in the network and using an encoding that takes advantage of the fact that the shortest paths from vertex u to all of the remaining vertices can be decomposed into subsets, based on the first edges on the shortest paths to them from u. By taking advantage of their spatial coherence, which is captured by the aid of a shortest path quadtree, the amount of storage to keep track of it is reduced to $\mathcal{O}(N)$, where N is the number of vertices.

Papadias et al. [49] present a storage model for spatial network databases together with a number of evaluation algorithms for the most popular queries, including nearest neighbor, range, and closest pair queries. Given a source point q and an entity dataset S, a k nearest neighbor (kNN) query retrieves the k objects of S that are closest to q according to the network distance, e.g. find the ten closest restaurants. A range query retrieves all objects of S that are within distance d from q. A closest-pairs query finds for two given datasets S and T the k pairs (s, t), $s \in S, t \in T$, that are closest in terms of network distance. For instance, find the hotel, restaurant pair with the smallest driving distance.

For *kNN* queries, two different solutions are proposed [49]. The Incremental Euclidean Restriction (IER) algorithm uses the Euclidean distance to iteratively

prune the search space for candidate objects, for which then the network distance can be computed using a more directed search. The Incremental Network Expansion (INE) algorithm is an adaptation of Dijkstra's algorithm and performs network expansion in all directions starting from the query point q.

For range queries, the Range Network Expansion (RNE) algorithm starts from q and computes first the set of all qualifying segments (QS) within the network range d. Then, the data objects are retrieved that are located on these segments. RNE uses two disk-based index structures (R-tree [28]) for indexing edges and data objects. After identifying the qualifying segments, an intersection join is performed that returns all objects in S that are within distance d. The Range Euclidean Restriction (RER) algorithm retrieves first a set S' of candidate objects that are within Euclidean distance d. Next, a network expansion is performed starting from q by examining all edges that are within the network distance d. Objects in S' that are within d are removed from S' and collected in the result set. The process terminates, when all edges in the range are exhausted or when S' becomes empty. The data structure for accessing the network is a disk-based extension of the connectivity-clustered access method CCAM [58] that stores the lists of neighbor nodes together in the same cluster. This search pruning strategy is not applicable in multimodal networks, where the Euclidean distance does not determine an upper bound of the network distance.

Deng et al. [15] improve over the work in Papadias et al. [49] by performing less network distance calculations and therefore accessing less network data. A major problem of INE is that the search expands in all directions, hence it requires many data accesses. Though IER improves the strategy using directional search, the network distance is still computed for all candidates, including those which will not appear in the final result. The new algorithm maintains a separate heap for each candidate and adopts the A^* search strategy. Expanding the vertex v with the smallest distance lower bound over all heaps, the 1st NN is computed first, then the 2nd NN, etc. Thus, the network distance is only computed for candidates that appear in the final result.

Almeida and Güting [13] present a storage schema with index structures together with a modified version of Dijkstra's algorithm for the incremental computation of kNN queries to allow a one-by-one retrieval of the objects on an edge. The algorithm can stop before retrieving k objects or continue to retrieve the k+1th NN. The algorithm is shown to outperform INE [49] and the VN3 [38] approach.

Jensen et al. [31] include in spatial networks the temporal dimension for computing spatial queries between moving objects. The distance between two moving points depends on several characteristics of a road (length, maximum speed, etc.). Since the provided abstract data model is specifically designed for moving objects in networks, it does not directly apply in multimodal networks.

Conceptually, isochrone queries are closest to range queries. A range query retrieves all objects within a given network distance d. In contrast, an isochrone query retrieves all network points within network distance d, thus the result is a portion of the network covering all space points within d rather than all data objects within d. Another difference concerns the algorithmic solution. The algorithms for the computation of isochrones presented in this thesis adapt Dijkstra's incremental network expansion strategy and are similar in spirit to INE, but they are generalized to switch between and to expand in different networks. Third, pruning techniques such as the Euclidean restriction can not be easily adopted and applied for isochrones in multimodal networks due to the schedule-based networks and the need to explore each individual pedestrian edge. Isochrone queries are more flexible than range queries: by intersecting the area covered by an isochrone with an object relation, all objects within an isochrone can be determined without the need to compute the distances to the individual objects.

2.2 Time-dependent Networks

Time dependency in spatial networks has been introduced to handle settings when the cost to traverse an edge changes over time, e.g. changing traffic conditions in road networks. Most existing work on time-dependent networks is on the shortest path problem, where two classes of solutions have been proposed. *Discrete time* models discretize the timeline and compute an instance of the traditional shortest path for each possible starting time [10, 47]. This approach effectively captures networks with few scheduled start times for which the cost of each edge is then fixed. It falls short in networks, where the starting time can be any time in a given interval and the edge cost is therefore a function of time. Since a discretization approach introduces inaccuracy and is inefficient if many possible starting times are considered, solutions based on a *continuous time* model have been proposed [17, 36]. Kanoulas et al. [36] adopt a continuous time model and investigate the problem of finding the fastest path (FP) from a source vertex v_s to a destination vertex v_d, when the starting time at v_s can be any time point in a given interval I and the travel time on an edge is specified as a piece-wise constant function over time, termed speed pattern. The result is a starting interval with a single fastest path (single FP) or a set of starting intervals covering I with the corresponding fastest path (all FP). The proposed solution is a novel extension of the A^* algorithm. Ding et al.[17] present a novel solution for the single FP problem, which they call time-dependent shortest path (TDSP) problem. An edge-delay function specifies how much time it takes to traverse an edge from u to v depending on the starting time at u, and waiting on vertices is allowed to minimize the travel time. The proposed algorithm is based on Dijkstra and requires less time and space than

previous work [36].

In the field of public transportation networks, researchers have recently turned the attention to schedule-based systems in multimodal transportation networks [68, 45]. Public transportation networks can be modeled as graphs just like road networks, except that besides spatial information also schedules have to be considered.

Müller-Hannemann et al. [45] summarize three different models for modeling time dependency in schedule-based networks. In the *condensed model* a node is introduced for each station and an edge is inserted if there is a direct connection between two stations. The weight of the edge is set to be the minimum travel time over all possible connections between these two stations. The benefit of this model is the relatively small size of the resulting graph. There are also several drawbacks. The model does not incorporate the actual departure time from a given station. Travel times highly depend on the time of the day, and the time needed to change to another type of transportation system is not taken into account. As as result, the calculated travel time between two arbitrary stations is only a lower bound on the real travel time.

The *time-dependent-model* [48, 47, 9] is an extension of the condensed model and uses time-dependent edges. Each station is modeled by a single node, and an edge between two vertices exists if there is a direct connection. Several weights are assigned to each edge. Each weight represents a travel time for a mean of transport running from one station to another. The specific edge that is used in query answering is then picked according to the departure time from the station. The advantages of this model are its small size, delay times can easily be incorporated, and the obtained travel time is feasible. However, adapting speed up techniques to time-dependent graphs is more complicated [14, 53].

In the *time-expanded-model* [46, 44] a vertex is used for each departure and arrival event. An edge is inserted for each connection between two events. The main drawback of this approach is that the resulting graphs are much bigger than for the time-dependent models. The benefit is the large flexibility in adding additional constraints.

Based on the time-expanded model, Huang [30] suggests a schedule-based shortest path finding algorithm for transit networks. The algorithm investigates the network by following route patterns based upon a pattern-centered spatio-temporal transit network model, which extends the time-expanded-model. A pattern represents a specific route. The backward PFS algorithm computes the fastest path from a destination vertex with an expected arrival time back to the origin vertex. The forward PFS algorithm does the opposite.

Bast [3] presents an overview of public transportation networks. The work highlights that the algorithmic problem of computing the shortest path between a source and a destination point is surprisingly different between road networks

and public transportation networks. Bauer et al. [6] show in their experimental study that speed up techniques, developed for static road networks [20, 37], perform significantly worse on time-expanded models. As a consequence running a simple time-dependent Dijkstra is faster than any speed up technique on the corresponding time-expanded graph.

Google has released a specification named *General Transit Feed Specification* (GTFS) [22] that proposes a general model designed for public transportation systems. The most relevant entities are schedules, trips, routes, and stops. Since many other routing applications have started to use this model, GTFS tends to become a de facto standard for modeling multimodal transportation systems.

2.3 Multimodal Networks

Florian et al. [68] investigate multimodal networks. They provide a general framework for a multimodal network in which events act as space-time elements. An event represents the arrival at a vertex and the corresponding time. The transition from one event to another is achieved by executing an activity (e.g. walking, waiting and boarding, riding). The presented algorithm computes the shortest path from a source vertex to a destination vertex within a given time range. Events are stored in a priority queue. The work is kept general and there is no formalization of events and transitions. Furthermore, the switching between different networks is not discussed.

Trip planning algorithms have been intensively investigated both in Euclidean space and road networks [32, 29, 57, 41, 61]. The aim is to provide an optimal and precise route according to user defined constraints. In the work of Li et al. [41], the user specifies a set of points of interest that belong to a specific category. Then the system computes the best route from the source to the destination location. Several approximation techniques are suggested to provide near-optimal answers with approximation ratios that depend on either the number of categories or the maximum number of points per category. The k-stops shortest path problem [57] seeks for the optimal path between two locations passing through exactly k intermediate points in the Euclidean space.

A data model presented by Booth et al. [8] introduces a framework of an advanced transportation system. The model provides a trip consisting of several transportation modes that are applied for multimodal shortest path queries. In a graph model each vertex represents a place in a transportation network. Places are annotated with a label and a geometric representation, i.e. point, line or polygon. Every edge is associated with a particular transportation mode. A vertex that consists of different adjacent edges is denoted as a transfer station. A trip is defined as a sequence of legs, where each leg represents a path in the same trans-

portation mode. Besides returning a path with minimum costs of time or distance, respectively, the result can have more constraints and choices, e.g. different motion modes, specifying the maximal number of transfers, etc.

Multi-geography route panning is investigated by Balasubramanian et al. [2] with the goal to determine the least cost weighted paths from source to destination locations residing in different geographies. A least cost path from a source to a destination retrieves that path with minimum resources, e.g. mileage. Geographies are described in multiple representations, e.g. raster vs. vector data or data with different projection systems. The primary motivation of this paper is to study real-time and safely routing through unknown spaces where an emergency event occurs.

Xu [69] proposes a system for managing moving objects with multiple transportation modes build on top of the SECONDO database management system [24].

2.4 Routing Applications and Systems

The growing popularity of online mapping services has activated a large development in online routing services for answering shortest path queries in multimodal networks.

OpenTripPlanner[1], as an alternative to Google Transit[2] is an open source multimodal trip planner that uses SP algorithms to compute the shortest trip between a source and a destination location. The routing is based on the directed search algorithms A^* [26] and contraction hierarchies [19]. The implementation is designed and optimized for SP queries. In June 2012, OpenTripPlanner released an extension, named *OpenTripPlanner Analyst*, that provides a service to compute isochrones in multimodal networks. The basic routing functionality was extended to enable one-to-many and many-to-one routing.

The project *pgRouting*[51] (PGR) extends the spatial DBMS PostGIS with network routing functionalities that are implemented as user-defined functions on top of PostgreSQL. The *driving distance* (isoline) SQL operator expects as input an edge relation, a query point that must be a vertex, and a maximal range. The edge relation must project a vertex identifier, the start vertex, the end vertex and the cost of an edge. Dynamic costs can only be specified before the computation starts. Therefore PGR can not be used for multimodal routing characterized by time-dependent costs, in which the transfer time to traverse an edge depends on the time t when the start vertex is visited. Since t is not know at the beginning of the computation, PGR can not manage time-dependent costs.

Oracle Spatial provides similar methods [66, 39] in Java, termed *withinCost*, as an alternative to the Euclidean distance operator *SDO_WITHIN_DISTANCE*.

[1] www.opentripplanner.org

[2] www.google.com/transit

The *withinCost* method returns a set of paths, from where each vertex in the path is reachable from q within the given costs.

Similar functionalities are provided also by other commercial GIS software. ESRI offers *Network Analyst* as an extension of the ARCGIS software product, and ERDAS provides the product *RouteFinder*[3] as an extension of the desktop product Intergraph. All these products are restricted to perform reachability analysis in a single, unimodal network. The company *Hacon* has implemented for the German railway agency "Deutsche Bahn" the timetable information system. Their product *Hafas* [25] – primarily used for trip advisory services – uses an efficient algorithm for range queries applied on the railway transportation network.

There exist a few applications that compute isochrones in multimodal networks. *Mapnificient*[4] uses a simple heuristic based on the Euclidean distance to determine the closure of an isochrone. The expansion is only performed in the transportation networks, while the distances covered in walking mode are approximated with a circle having as radius the remaining distance. The British company Mapumental[5] developed a similar commercial product that uses isochrones in house hunter services.

2.5 Summary

There has been a lot of research work in the area of spatial network databases, investigating various types of queries, different networks such as time-dependent networks, as well as different transportation modalities. Conceptually, isochrones are closest to range queries. While range queries retrieve all objects within a given network distance, isochrone queries return all network points within this distance. Thus, the result of an isochrone query is a portion of the network rather than a set of data objects. Another difference concerns the algorithmic solutions. For the computation of isochrones the network needs to be explored in all possible directions to identify all qualified space points, whereas the computation of range queries can be optimized by search space pruning techniques and driving the search towards candidate objects. Existing systems to compute isochrones either use approximation techniques or work only for unimodal networks.

[3] http://www.routeware.dk/routefinder/
[4] www.mapnificent.net
[5] http://mapumental.com/

CHAPTER 3

Isochrones in Multimodal Spatial Networks

In this chapter we first introduce and define basic concepts about multimodal networks and routing. We define a multimodal path that is characterized by a composition of edges that represent different transport networks. We propose a generic cost function that computes the cost for traversing an arbitrary edge in a multimodal network. We define a multimodal path characterized by its time-dependency. Given a multimodal network, we finally introduce and formally define an isochrone as the minimal and possible disconnected subgraph that covers exactly those locations from where a query point is reachable within given time constraints. We provide also a second definition of isochrones, which cover all the locations that are reachable from the query points.

3.1 Multimodal Spatial Networks

We begin with the definition of a multimodal network that allows to represent several transport systems in a single network.

Definition 3.1.1. (*Multimodal Network*) A *multimodal network* is a eight-tuple $\mathsf{N} = (G, \Theta, S, \theta, \mu, \lambda, \tau, \omega)$. $G = (V, E)$ is a directed multigraph with a set V of vertices and a multiset E of ordered pairs of vertices, termed edges. Θ is a set of transport systems. $S = (\Theta, TID, W, \sigma_a, \sigma_d)$ is a schedule, where TID is a set of trip identifiers, $W \subseteq V$, and $\sigma_a : \Theta \times TID \times W \mapsto \mathbb{T}$ and $\sigma_d : \Theta \times TID \times W \mapsto \mathbb{T}$ determine arrival and departure time, respectively ($\mathbb{T}$ is the time domain). Function $\mu : \Theta \mapsto \{'csct', 'csdt', 'dsct', 'dsdt'\}$ assigns to each transport system a transport mode, and the functions $\theta : E \mapsto \Theta$, $\lambda : E \mapsto \mathbb{R}^+$, $\tau : E \times \mathbb{T} \mapsto \mathbb{R}^+$, and $\omega : E \times \mathbb{T} \mapsto \{(0, 1], \infty\}$ assign to each edge transport system, length, transfer time, and weight, respectively.

A multimodal network allows to represent several transport systems, Θ, with different modalities in a single network. For the transport modalities we distinguish between continuous and discrete in space and time, respectively. This gives the following four different transport modes:

- *continuous space and time mode* ($\mu(.) = 'csct'$), e.g. pedestrian network;
- *discrete space and time mode* ($\mu(.) = 'dsdt'$), e.g. the public transport system, such as trains and buses;
- *discrete space continuous time mode* ($\mu(.) = 'dsct'$), e.g. moving walkways or moving stairs;
- *continuous space discrete time mode* ($\mu(.) = 'csdt'$), e.g. regions or streets that can be passed by pedestrians or cars only in specific time slots.

Vertices represent crossroads of the street network and/or stops of the public transport system. Edges represent street segments, transport routes, moving walkways, etc. For an edge (u, v) we denote u as the start vertex and v as the end vertex.

Example 3.1.1. Figure 3.1 shows a multimodal network with two transport systems, $\Theta = \{'P', 'B'\}$, representing the pedestrian network with mode $\mu('P') = 'csct'$ and bus network with route number 'B' with mode $\mu('B') = 'dsdt'$, respectively. Solid lines are street segments of the pedestrian network, e.g. edge $e = (v_1, v_2)$ with $\theta(e) = 'P'$. Pedestrian edges are annotated with the edge length, which is the same in both directions, e.g. $\lambda((v_1, v_2)) = \lambda((v_2, v_1)) = 300$. Dashed lines represent edges of the bus network.

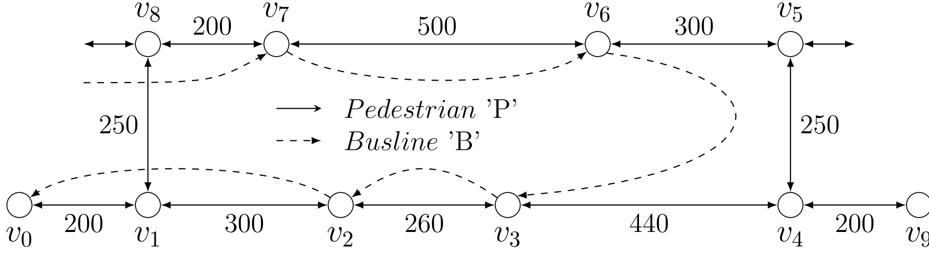


Figure 3.1: Multimodal Network.

The schedule stores for discrete time ('*dsdt*', '*csdt*') transport systems(T-Sys) the arrival and departure time at the stop stations for the individual trips of a certain route. An excerpt of a schedule is shown in Table 3.1, e.g. $TID = \{1, 2, \dots\}$, $\sigma_a(\text{'B'}, 1, v_6) = \sigma_d(\text{'B'}, 1, v_6) =$ 05:33:00.

T-Sys (Θ)	Trip (TID)	Stop (V)	Arrival (σ_a)	Departure (σ_d)
B	1	v_7	05:31:30	05:32:00
B	1	v_6	05:33:00	05:33:00
B	1	v_3	05:34:00	05:34:30
	⋮	⋮	⋮	⋮
B	2	v_7	06:01:30	06:02:00
B	2	v_6	06:03:00	06:03:00
B	2	v_3	06:05:00	06:05:30

Table 3.1: Example of a Schedule.

The schedule follows a time-dependent model [14, 54, 4], where the cost of an edge is not just a scalar value, but a piece-wise linear function that maps each possible arrival time from the start vertex of the edge to a travel cost. For an edge $e = (u, v)$, the function $\tau(e, t)$ computes the time-dependent transfer time that is required to traverse e, when starting from u as late as possible and arriving at v no later than time t. Pyrga et al. [54] term this the *latest-departure problem* (LDB), in which the optimization criterion is to maximize the actual departure time at the departure station among all connections that arrive at the arrival station by the given arrival time. This problem occurs when we compute an isochrone as the set of points from where the query point is reachable within the given time constraints. The opposite problem is termed *earliest-arrival problem* (EAP) [54], in which the optimization consists in minimizing the difference between the arrival time and the given departure time. This problem occurs when isochrones are defined as the set of all space points that are reachable from the query point.

In order to define a multimodal path, which is an essential part of an isochrone, we first introduce the transfer time for an arbitrary edge in a multimodal network.

Definition 3.1.2 (Transfer time)**.** Given a multimodal network N with edges E and time domain $\mathbb{T}$. Furthermore, let $t \in \mathbb{T}$ be either the arrival time at v or the departure time at u. The *transfer time* of an edge (u, v) for the transport modes '*dsdt*', '*dsct*', and '*csct*' is determined by a function $\tau : E \times \mathbb{T} \to \mathbb{R}^+$ that is defined as follows:

$$\tau((u,v),t) = \begin{cases} \frac{\lambda((u,v))}{s} * \omega(t) & \mu(\theta((u,v))) \in \{\text{'}csct\text{'}, \text{'}dsct\text{'}\}, \\ t - t' & \mu(\theta((u,v))) = \text{'}dsdt\text{'} \wedge \\ & t \text{ is arrival time at } v \wedge \\ & t' = \max\{\sigma_d(r, tid, u) \mid \sigma_a(r, tid, v) \leq t\}, \\ t' - t & \mu(\theta((u,v))) = \text{'}dsdt\text{'} \wedge \\ & t \text{ is departure time at } u \wedge \\ & t' = \min\{\sigma_a(r, tid, u) \mid \sigma_d(r, tid, v) \geq t\}. \end{cases}$$

For continuous space and time edges ($\mu((u, v)) =$ '*csct*'), the transfer time is modeled as a time-dependent function specified with a weight function w. The weight is a function that assigns to each edge in dependency of the time a value from the interval $(0, 1]$, e.g. to afford for different traffic conditions during rush hours, or ∞ if the edge cannot be traversed during some time periods. For discrete time edges ($\mu((u, v)) =$ '*dsdt*') the arrival or departure time must be considered. If the departure time is given, the transfer time is difference between the earliest arrival time t' at v minus t. If the arrival time at v is given, the transfer time is the difference between the current time t and the latest departure time t' at vertex u. For the transport modes '*csdt*' the transfer time can be specified in a similar way.

Example 3.1.2. We assume a constant walking speed of $s = 2\,\text{m/s}$ and a constant weight $\omega(t) = 1$, which yields a transfer time $\tau(e, t) = \frac{\lambda(e)}{2\,\text{m/s}}$ for pedestrian edges. In Figure 3.2 with a given arrival time 06:04:00, the transfer time on the pedestrian edge (v_7, v_6) is $\frac{500\text{m}}{2\,\text{m/s}} = 250\,\text{s}$. The transfer time on the bus edge (v_7, v_6) is 06:04:00 − 06:02:00 = 120 s, including a 60 s waiting time for the next bus.

A continuous in space edge allows to start or stop a trip not only at the vertices but also at any point along the edge. We refer to such positions as locations.

Definition 3.1.3 (Location)**.** A *location* in a multimodal network N is any point on an edge $e = (u, v) \in E$ that is accessible. We represent it as $l = (e, o)$, where $0 \leq o \leq \lambda(e)$ is an offset that determines the relative position of l from u on edge e.

A location represents vertex u if $o = 0$ and vertex v if $o = \lambda(e)$; any other offset refers to an intermediate point on edge e. In continuous space networks all points on the edges are accessible. Since a pedestrian segment is modeled as a pair of directed edges in the opposite direction, any point on it can be represented by two locations, $((u, v), o)$ and $((v, u), \lambda((u, v)) - o)$, respectively. For instance, in Figure 3.2 the location of q is $l_q = ((v_2, v_3), 180) = ((v_3, v_2), 80)$. In discrete space networks only vertices are accessible, thus $o \in \{0, \lambda(e)\}$ and locations coincide with vertices. An *edge segment*, (e, o_1, o_2), with $0 \leq o_1 \leq o_2 \leq \lambda(e)$ represents the contiguous set of space points between the two locations (e, o_1) and (e, o_2) on edge e. We generalize the length function for edge segments to $\lambda((e, o_1, o_2)) = o_2 - o_1$.

After having introduced the basic components of a multimodal network, we are now ready to define a multimodal path and its costs.

Definition 3.1.4 (Path)**.** A *path* from a source location $l_s = ((v_1, v_2), o_s)$ to a destination location $l_d = ((v_k, v_{k+1}), o_d)$ is defined as a sequence of connected edges and edge segments, $p(l_s, l_d) = \langle x_1, \ldots, x_k \rangle$, where $x_1 = ((v_1, v_2), o_s, \lambda((v_1, v_2)))$, $x_i = (v_i, v_{i+1})$ for $1 < i < k$, and $x_k = ((v_k, v_{k+1}), 0, o_d))$.

The first and the last element in a path can be edge segments, whereas all other elements are entire edges. Edges along a path may belong to different transport systems. This implies a switch into a different transport system.

Example 3.1.3. In Figure 3.2, a path from v_7 to q is to take bus 'B' from v_7 to v_3 and then walk to q, i.e. $p(v_7, l_q) = \langle x_1, x_2, x_3 \rangle$, where $x_1 = (v_7, v_6)$ and $x_2 = (v_6, v_3)$ are complete edges and $x_3 = ((v_3, v_2), 0, 80)$ is an edge segment.

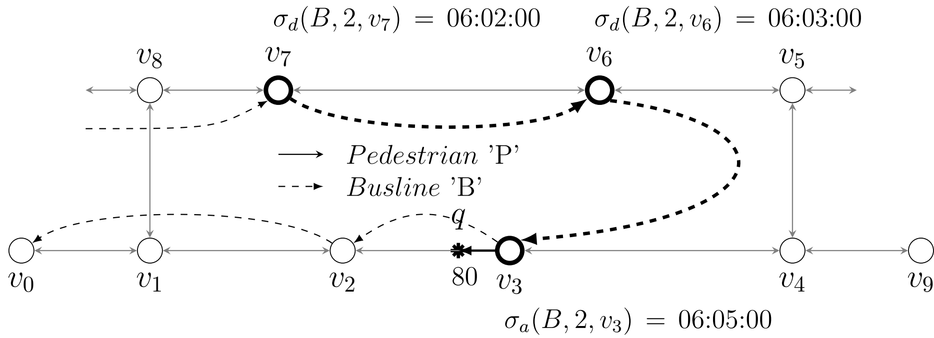


Figure 3.2: Multimodal Path.

Definition 3.1.5 (Path Cost)**.** The *cost* of a path $p(l_s, l_d) = \langle x_1, \ldots, x_k \rangle$ with t as arrival time at location l_d or departure time at location l_s is the sum of the

individual transfer times of all edges/edge segments in the path, i.e.

$$\gamma(\langle x_1,\dots,x_k\rangle,t) = \begin{cases} \tau(x_k,t) & k=1, \\ \tau(x_k,t)+\gamma(\langle x_1,\dots,x_{k-1}\rangle, t-\tau(x_k,t)) & k>1 \wedge \\ & t \text{ is arr. time at } l_d, \\ \tau(x_1,t)+\gamma(\langle x_2,\dots,x_k\rangle, t+\tau(x_1,t)) & k>1 \wedge \\ & t \text{ is dep. time at } l_s. \end{cases}$$

The above definition is recursive. If the path consists of a single edge or edge segment ($k = 1$), function τ computes the cost of traversing this edge, depending on whether t is arrival or departure time. If the path, $\langle x_1,\dots,x_k\rangle$, contains more than one edge or edge segments, the transfer time for the last edge (segment), x_k, is determined if t is arrival time at l_d and the cost of the first edge (segment), x_1, if t is departure time at l_s. For the remaining path, function γ is called in a recursive way with a new arrival/departure time. The recursion terminates when the path contains a single edge (segment).

Example 3.1.4. We continue the previous example and consider path $p(v_7,l_q) = \langle(v_7,v_6),(v_6,v_3),((v_3,v_2),0,80)\rangle$ with arrival time $t = $ 06:06:00 at location l_q and a constant walking speed of $2\,m/s$. Then, the cost of traversing this path is

$$\gamma(p(v_7,l_q), 06{:}06{:}00) = \tau(((v_3,v_2),0,80), 06{:}06{:}00) + \gamma(p(v_7,v_3),t'),$$

where t' is the latest arrival time at v_3 to reach l_q in time. The transfer time on the last edge segment is determined as $\tau(((v_3,v_2),0,80), 06{:}06{:}00) = \frac{80m}{2m/s} = 40\,\mathrm{s}$, which gives

$$\begin{aligned} \gamma(p(v_7,l_q), 06{:}06{:}00) &= 40\,s + \gamma(p(v_7,v_3), 06{:}06{:}00 - 40\,s) \\ &= 40\,s + \gamma(p(v_7,v_3), 06{:}05{:}20) \\ &= 40\,s + \tau((v_6,v_3), 06{:}05{:}20) + \gamma(p(v_7,v_6),t'). \end{aligned}$$

We calculate now the transfer time on the discrete edge (v_6,v_3) with arrival time at v_3 no later than 06:05:20. Since the latest bus that matches this constraint departs at 06:03:00 and arrives at 06:05:00, we have a waiting time of 20 s at v_3, and the transfer time is $\tau((v_6,v_3), 06{:}05{:}20) = 06{:}05{:}20 - 06{:}03{:}00 = 140\,\mathrm{s}$. This gives

$$\begin{aligned} \gamma(p(v_7,l_q), 06{:}06{:}00) &= 40\,s + 140\,s + \tau((v_7,v_6), 06{:}05{:}20 - 140\,s) \\ &= 40\,s + 140\,s + \tau((v_7,v_6), 06{:}03{:}00). \end{aligned}$$

We compute the transfer time on the bus edge (v_7, v_6) in a similar way as above and get the final path cost as:

$$\gamma(p(v_7, l_q), t) = 40\,s + 140\,s + 40\,s = 220\,s.$$

Notice that with a different arrival time at q, e.g. $t = 06{:}05{:}00$, the path cost might be significantly different.

Next, we introduce network distance as the cost of the shortest path from a source location to a destination location.

Definition 3.1.6 (Network distance). The *network distance*, $d(l_s, l_d, t)$, from a source location, l_s, to a destination location, l_d, with t being the arrival time at l_d (the departure time at l_s), is defined as the minimum cost of any path from l_s to l_d with arrival time t at l_d (departure time t at l_s) if such a path exists, and ∞ otherwise.

We measure the network distance in terms of transfer time that is required from a source to a destination.

3.2 Definition of Isochrones

We proceed with the definition of an isochrone as the minimal subgraph of a multimodal spatial network that covers all locations from where a query point q is reachable under the given time constraints if the direction of the reachability is incoming. Otherwise the isochrone covers all locations reachable from q.

Definition 3.2.1 (Isochrone). Let $\mathsf{N} = (G, \Theta, S, \mu, \theta, \lambda, \tau, \omega)$ with $G = (V, E)$ be a multimodal network, Q be a set of query points with arrival time (departure time) t, and $d_{max} > 0$ be a maximum time span. An *isochrone*, $N^{iso} = (V^{iso}, E^{iso})$, is defined as the minimum and possibly disconnected subgraph of G that satisfies the following conditions:

- $V^{iso} \subseteq V$,
- $\forall l(l = (e, o) \wedge e \in E \wedge \exists q \in Q(\hat{d}(l, q, t) \leq d_{max})$
  $\Leftrightarrow \exists x \in E^{iso}(x = (e, o_1, o_2) \wedge o_1 \leq o \leq o_2))$,

where $\hat{d}(l, q, t) = d(l, q, t)$ is the network distance from l to q if t is the arrival time at q, and $\hat{d}(l, q, t) = d(q, l, t)$ is the network distance from q to l if t is the departure time at q.

The first condition requires the vertices of the isochrone to be a subset of the vertices of the multimodal network. The second condition constrains an isochrone to cover exactly those locations l with a network distance $d(l, q, t)$ to its closest $q \in Q$ (from its closest $q \in Q$) that is smaller than or equal to d_{max}. Notice the usage of edge segments in E^{iso} for representing partially reachable edges. Whenever an edge e is entirely covered by an isochrone, we use e instead of $(e, 0, \lambda(e))$.

Example 3.2.1. In Figure 3.3, the subgraph in bold represents the isochrone for $d_{max} = 5\,\text{min}$ and arrival time $t =$ 06:06:00 at q. The numbers in parentheses are the network distance to q. Edges close to q are entirely reachable, whereas edges on the isochrone border are only partially reachable. Partially reachable edges are labeled with the offset of the reachable portion from the edge's start vertex. For instance, on edge (v_0, v_1) only locations after an offset of 80 meters reach q within the given time constraints.

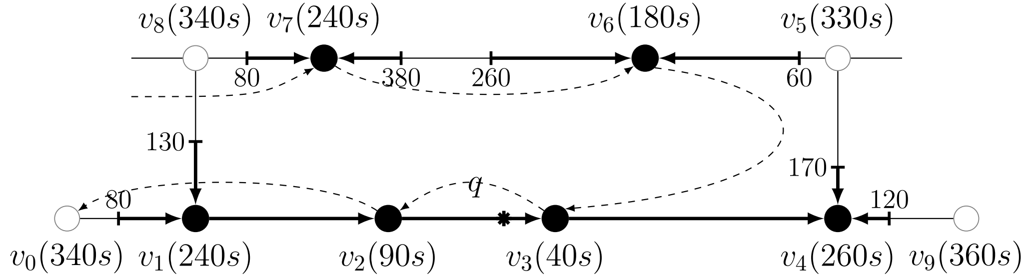


Figure 3.3: Isochrone with d_{max}=5min, s=2m/s and $t =$ 06:06:00 .

More formally, the isochrone subgraph is represented by the following set of vertices and edges:

$$
\begin{aligned}
V^{iso} &= \{v_3, v_2, v_6, v_1, v_7, v_4\}, \\
E^{iso} &= \{((v_0, v_1), 80, 200), ((v_8, v_1), 130, 250), ((v_2, v_1), 180, 300), \\
&\quad ((v_1, v_2), 0, 300), ((v_3, v_2), 0, 260), \\
&\quad ((v_2, v_3), 0, 260), ((v_4, v_3), 0, 440), \\
&\quad ((v_3, v_4), 360, 440), ((v_5, v_4), 170, 250), ((v_9, v_4), 120, 200), \\
&\quad ((v_5, v_6), 60, 300), ((v_7, v_6), 260, 500), \\
&\quad ((v_6, v_7), 380, 500), ((v_8, v_7), 80, 200)\}.
\end{aligned}
$$

3.3 Summary

In this section we introduced and formally defined isochrones in multimodal spatial networks. We begun with the definition of basic concepts of multimodal spatial

networks that can be continuous or discrete along, respectively, the time and space dimensions. More specifically, we introduced a general time-dependent cost function to compute the time-dependent transfer time of an edge. Thereby, the time can be specified as starting time at the source location or as arrival time at the destination location.

CHAPTER 4

Algorithms for the Computation of Isochrones

In this chapter we propose three different algorithms for the computation of isochrones in multimodal spatial networks. While all three algorithms apply an incremental network expansion strategy, they differ mainly in the way they load the network in main memory during network expansion and in the overall memory requirements. The first algorithm, *MDijkstra*, is completely memory-based and initially loads the entire network in memory, where network expansion is then performed. The second algorithm, *MINEX*, loads the network from disk during the network expansion process — one node with incoming edges on each expansion step. Thus, the algorithm keeps only the isochrone in memory. To further reduce the memory requirements, we introduce the concept of *vertex expiration*, which allows to keep in memory only a minimal portion of the isochrone that is required to avoid cyclic network expansions. The third algorithm, *MRNEX*, adopts a hybrid approach. The network is loaded in small chunks, which significantly reduces the number of DB accesses. With vertex expiration, the memory requirements can be minimized similar to the previous algorithm.

4.1 Memory-based Algorithm MDijkstra

A memory-based algorithm for the computation of isochrones has first been introduced in [7]. We improved this algorithm in several directions as will be described below.

The overall strategy of computing isochrones using network expansion a la Dijkstra works as follows. Consider a multimodal network N, query point q with arrival time t at q, maximal duration d_{max}, and walking speed s. The network expansion starts from query point q and propagates backwards along the incoming edges in all directions. When a vertex v is expanded, all incoming edges $e = (u, v)$ are considered. The network distance of vertex u to query point q, when traversing e is computed incrementally as the network distance of v to q plus the transfer time to traverse edge e. The expansion terminates when all locations with a network distance to q that is smaller than d_{max} have been visited. In order to avoid cyclic expansion, two sets of vertices are maintained:

- *Closed vertices*, C: They have already been expanded and the network distance to q is known. C is maintained in a hash table.
- *Open vertices*, O: They have been encountered, but are not yet expanded. The set O is maintained in a priority queue.

For each vertex $v \in O \cup C$, we keep the network distance, d_v (abbrev. for $d(v, q, t)$) to q. On each iteration of network expansion, the vertex in O with the smallest distance to q is expanded.

Algorithm 1 shows the pseudocode of algorithm MDijkstra, which implements this strategy. First the entire network is loaded in main memory, and the set O of open vertices is initialized to the set of all vertices together with an initial distance of ∞. The set C of closed vertices is initialized to the empty set. Next, the position of q in the network is located. If q coincides with a vertex v, the distance of v in O is updated to 0. Otherwise, q is on an edge (u, v). The distance of vertex u and v in O is updated by considering the walking distance from q to these vertices, i.e. the time it takes to reach q. The traversed edge segments are output.

After this initialization, vertex expansion starts (line 10). The vertex v with the smallest distance is dequeued from O and marked as closed by moving it to the set C. Then, all incoming edges, $e = (u, v)$, to v are examined. Since a discrete time edge always implies a disk access to query the schedule, we do this only if it is really required, i.e. if vertex u is not yet closed. In this case its distance to q when traversing edge e is incrementally computed as the distance of v plus the time to traverse e starting from the end vertex v. If this distance, d'_u, is smaller then the one in O, d_u is updated (lines 16). Otherwise, when u is already closed and the edge is a discrete edge, there is no need to compute the distance of u via

Algorithm 1: $MDijkstra(q, d_{max}, s, t, \mathsf{N})$

input : $q, d_{max}, s, t, \mathsf{N}$
output: E^{iso}, V^{iso}

1 load network in main memory;
2 $O \leftarrow \{(v_i, \infty) \mid v_i \in V\}$;
3 $C \leftarrow \emptyset$;
4 **if** *q coincides with v* **then**
5 | Update the distance of v in O to 0 ;
6 **else**// $q = ((u, v), o) = ((v, u), o')$
7 | Update the distance of u in O to o/s ;
8 | Update the distance of v in O to o'/s ;
9 | Output $((u, v), \max(0, (o - d_{max}/s), o)$ and $((v, u), \max(0, (o' - d_{max}/s), o')$;
10 **while** $O \neq \emptyset \wedge (v, d_v) \leftarrow dequeue(O) \wedge d_v \leq d_{max}$ **do**
11 | $O \leftarrow O \setminus \{v\}$;
12 | $C \leftarrow C \cup \{v\}$;
13 | **foreach** $e = (u, v) \in E$ **do**
14 | | **if** $u \notin C$ **then**
15 | | | $d'_u \leftarrow \tau(e, t - d_v) + d_v$;
16 | | | $d_u \leftarrow \min(d_u, d'_u)$;
17 | | **else**
18 | | | **if** $\mu(\theta(e)) \in \{$'csct', 'csdt'$\}$ **then**
19 | | | | $d'_u \leftarrow \tau(e, t - d_v) + d_v$;
20 | | **if** $\mu(\theta(e)) \in \{$'csct', 'csdt'$\}$ **then**
21 | | | **if** $d'_u \leq d_{max}$ **then**
22 | | | | Output $(e, 0, \lambda(e))$;
23 | | | **else**
24 | | | | Output $(e, o, \lambda(e))$, where $d((e, o), q, t) = d_{max}$;
25 | $V^{iso} \leftarrow V^{iso} \cup \{(v, d_v))\}$;

e. If, however, e is continuous in space (e.g. a pedestrian edge), the distance d'_u is computed to determine the segment of e that is included in the isochrone. The last step on each iteration adds the reachable portions of continuous space edges to the result. Discrete space edges do not produce an output, since they have no accessible locations except their start and end vertices, which are added when the incoming continuous space edges are processed. The algorithm terminates when O is empty or the network distance of the first vertex in O exceeds d_{max}.

Complexity. Cormen et al. [11] suggest for sufficiently sparse networks, where the number of edges is $O(|V|^2/\log(|V|))$, to implement the min-priority queue O with a binary min-heap. Building the binary heap takes $\mathcal{O}(|V|)$ time. The dequeue operation requires $\mathcal{O}(\log |V|)$ time. Updating a vertex in the binary heap takes $\mathcal{O}(\log |V|)$, and there are at most $|E|$ such operations. The total running time is therefore $\mathcal{O}((|V| + |E|) \log |V|)$.

Since the schedule is kept on disk, each traversal of a discrete edge requires a database lookup on the schedule to determine the latest departure time of the start vertex. Overall, this algorithm suffers from the high initialization (loading)

cost and the memory requirements, which makes the algorithm not scalable (in terms of memory usage) for very large networks.

4.1.1 Extensions

As already mentioned, we improved the basic isochrone algorithm in [7] in several directions. First, we did a complete reengineering and made the implementation independent from the underlying spatial database. The Oracle specific Java classes [66] were replaced by more general classes that support all features of a multimodal network as described in Chapter 3 and makes the algorithm independent of the underlying database system. Our classes provide a clear separation of the logical network from the physical information such as geometries. For the computation of the isochrones the algorithm needs only the logical network, which is comparably small. The geometries, which can be very large, are only needed for the rendering of the result. This separation reduced the initial loading time of MDijkstra by more than a factor of two.

Second, we changed the underlying model of the schedule from an instant-based representation to an interval-based representation. The instant representation stores in every tuple a single stop with arrival and departure time (see Table 4.1(a)). In contrast, in the interval representation in Table 4.1(b), each entry stores an edge of the network with departure time at the source vertex an arrival time at the target vertex. The interval representation is more efficient since less database accesses are required. For instance, to determine the transfer time along the discrete edge (v_7, v_6) in our running example, we first have to calculate the latest arrival time $\sigma_a(\text{'B'}, 1, v_6)$ at v_6. In a second database access we can then determine the corresponding departure time at vertex v_7 (marked in boldface). Even if this operation is implemented with a self-join, it has first to read the tuple of vertex v_6 and then the tuple of v_7. In contrast, in the interval representation only one tuple needs to be fetched (marked in boldface). The change from an instant to an interval representation brought performance improvements of up to 30% in networks with a large public transport system.

Θ	TID	V	σ_a	σ_d
	⋮	⋮	⋮	⋮
B	1	v_7	05:31:30	**05:32:00**
B	1	v_6	**05:33:00**	05:33:00
B	1	v_3	05:34:00	05:34:30
	⋮	⋮	⋮	⋮

(a) Instant-based

Θ	TID	V_s	V_e	$\sigma_d(V_s)$	$\sigma_a(V_e)$
	⋮	⋮	⋮	⋮	⋮
B	1	...	v_7	...	05:31:30
B	1	v_7	v_6	**05:32:00**	**05:33:00**
B	1	v_6	v_3	05:33:00	05:34:00
B	1	v_3	...	05:34:30	...

(b) Interval-based

Figure 4.1: Schedule Representations.

Third, when using the interval-based representation, we can additionally reduce the number of database lookups. Instead of determining for each single discrete edge the latest departure time for the currently expanded vertex v, it is sufficient to determine for all adjacent vertices that are connected to v the latest departure time over all edges that share the same start vertex. This modification improved the runtime for the schedule lookups by another 20-25% in networks with dense and large schedules.

4.2 Multimodal Incremental Network Expansion

To overcome MDijkstra's memory limitation, we propose a multimodal incremental network expansion strategy. Similar to the INE algorithm of Papadias et al. [49], we adopt an on-demand main-memory loading strategy. We present two algorithms. The algorithm MINE (*Multimodal Incremental Network Expansion*) loads the relevant edges incrementally from the database in every expansion cycle and keeps only the actual isochrone in memory. The algorithm MINEX (*Multimodal Incremental Network Expansion with network Expiration*) additionally applies vertex expiration to remove from memory network portions that are no longer needed to avoid cyclic expansion.

4.2.1 Algorithm MINE

Algorithm 2 shows the MINE algorithm. The major difference to the memory-based MDijkstra is that the network is not initially loaded in memory. Instead, during each step of network expansion the incoming edges of the expanded vertex are loaded from disk. Thus, only the open and closed vertices, O and C, are stored in main memory.

First, the closed vertices C are initialized to the empty set. O is initialized to v with $d_v = 0$ if q coincides with vertex v. Otherwise, $q = ((u, v), o) = ((v, u), \lambda(v)u - o)$ is an intermediate location on a pedestrian edge, and C is initialized with vertices u and v in the same way as in MDijkstra. The reachable edge segments are output to the result.

During the expansion phase, vertex v with the smallest network distance is dequeued from O and added to C. All incoming edges, $e = (u, v)$, are retrieved with a multi-point query from the database and considered in turn. If vertex u is visited for the first time, it is added to O with a distance set to ∞. Then, the distance d'_u of u when traversing e is computed according to the edge type, and the distance d_u is updated. If e is a '*csct*' or '*csdt*' edge, the reachable part of e is added to the result.

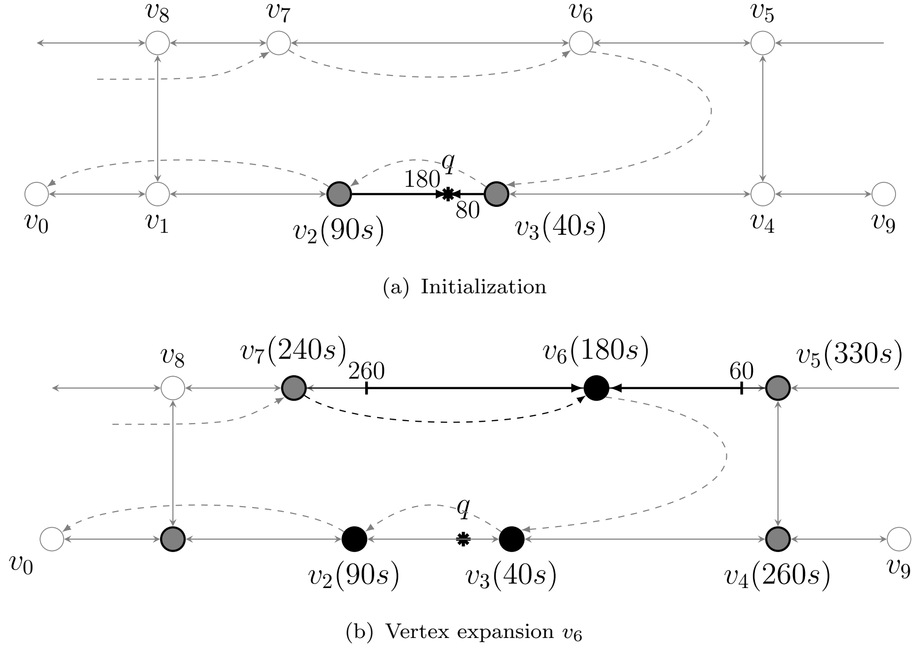
Algorithm 2: $MINE(q, t, d_{max}, s, \mathsf{N})$.

```
   input : q, d_max, s, t, N
   output: E^iso, V^iso
1  C ← ∅;
2  if q coincides with v then
3  |  O ← {(v, 0)};
4  else// q = ((u, v), o) = ((v, u), o′)
5  |  O ← {(u, o/s), (v, o′/s)};
6  |  Output ((u, v), max(0, (o − d_max/s), o) and ((v, u), max(0, (o′ − d_max/s), o′);
7  while O ≠ ∅ ∧ (v, d_v) ← dequeue(O) ∧ d_v ≤ d_max do
8  |  O ← O \ {v};
9  |  C ← C ∪ {v};
10 |  foreach e = (u, v) ∈ E do
11 |  |  if u ∉ O ∪ C then
12 |  |  |  O ← O ∪ {(u, ∞, cnt_u)};
13 |  |  if u ∉ C then
14 |  |  |  d′_u ← τ(e, t − d_v) + d_v;
15 |  |  |  d_u ← min(d_u, d′_u);
16 |  |  else
17 |  |  |  if μ(θ(e)) ∈ {'csct', 'csdt'} then
18 |  |  |  |  d′_u ← τ(e, t − d_v) + d_v;
19 |  |  if μ(θ(e)) ∈ {'csct', 'csdt'} then
20 |  |  |  if d′_u ≤ d_max then
21 |  |  |  |  Output (e, 0, λ(e));
22 |  |  |  else
23 |  |  |  |  Output (e, o, λ(e)), where d((e, o), q, t) = d_max;
24 |  V^iso ← V^iso ∪ {(v, d_v))};
```

Example 4.2.1. Figure 4.2(a) shows the situation after the initialization phase. The bold edge segments, $((v_2, v_3), 0, 188)$ and $((v_3, v_2), 0, 80)$, have been traversed so far. The gray vertices are open, i.e. $O = \{(v_3, 40), (v_2, 90)\}$. Only the open vertices, v_2 and v_3, and the connecting edges, (v_2, v_3) and (v_3, v_2) have been loaded and are stored in memory.

Network expansion starts, and Figure 4.2(b) illustrates the situation after expanding vertex v_6, which has a network distance of $180s$. The incoming edges are loaded and traversed. After traversing the pedestrian edge (v_7, v_6), the vertex v_7 gets assigned the network distance $d_{v_7} = 430s$. Only a part of the edge is reachable within the given time constraints (edge segment in bold). Next, we traverse the bus edge (v_7, v_6) with a smaller transfer time that updates the distance of v_7 to $240s$. Finally, after examining the pedestrian edge (v_5, v_6), the distance of v_5 is updated to $d_{v_5} = 330s$.

Complexity. The runtime complexity of MINE due to network expansion is the same as in MDijkstra. The only difference is that the size of the min-priority queue grows with the size of the isochrone and is independent of the network size. The total runtime is dominated by the number of database accesses during



(a) Initialization

(b) Vertex expansion v_6

Figure 4.2: Network Expansion in MINE.

the expansion, which is linear in the isochrone size, i.e. $\mathcal{O}(V^{iso})$. The memory complexity, however, is different. MINE holds in main memory only the isochrone, yielding a complexity of $\mathcal{O}(|V^{iso}| + k + l)$. V^{iso} is the size of the isochrone, k is the number of open vertices in memory that have not been expanded, and the constant l is the maximum out-degree of all closed vertices.

4.2.2 Algorithm MINEX

Though MINE does not load the entire network at the beginning, the memory requirements grow with the size of the isochrone, which in the worst case can be the entire network for very large isochrones. In this section we present an improved algorithm, termed MINEX, which applies vertex expiration to eagerly remove vertices from memory when they are no longer needed to avoid cyclic expansions. We present here the algorithm and leave a more detailed discussion for the next section.

Intuitively, *vertex expiration* works as follows. Whenever a vertex is expanded and will not be visited again during network expansion, it can be removed from memory. To keep the memory requirements low, we eagerly expire the isochrone and hold in memory only the minimal set of expanded vertices that is necessary to avoid cyclic expansions. A cyclic expansion occurs when an already expanded vertex is visited for a second time. The isochrone, which is stored in memory,

does not contain sufficient information to identify expired vertices. Therefore we introduce a strategy that counts for each vertex the outgoing edges that have not yet been traversed.

Algorithm 3 shows the algorithm MINEX which applies vertex expiration. It works similar to MINE, except that every vertex is annotated with a counter $cnt_.$, which keeps track of the number of outgoing edges that have not yet been traversed. When a vertex u is added to O, its counter cnt_u is initialized with the number of outgoing edges (lines 3,5 and 12). Whenever an edge $e = (u, v)$ is examined, the counter of its start vertex u (of which e is an outgoing edge) is decremented by 1 (line 18). If u is closed and $cnt_u = 0$, vertex u is expired and removed from C (line 20). Once all incoming edges of v are processed, the expiration and removal of v is checked (line 26).

Algorithm 3: $MINEX(q, t, d_{max}, s, \mathsf{N})$

```
   input  : q, d_max, s, t, N
   output: E^iso, V^iso
1  C ← ∅;
2  if q coincides with v then
3  |   O ← {(v, 0, cnt_v)} ;
4  else// q = ((u, v), o) = ((v, u), o')
5  |   O ← {(u, o/s, cnt_u), (v, o'/s, cnt_v)} ;
6  |   Output ((u, v), max(0, (o−d_max/s), o) and ((v, u), max(0, (o'−d_max/s), o');
7  while O ≠ ∅ ∧ (v, d_v, cnt_v) ← dequeue(O) ∧ d_v ≤ d_max do
8  |   O ← O \ {v};
9  |   C ← C ∪ {v};
10 |   foreach e = (u, v) ∈ E do
11 |   |   if u ∉ O ∪ C then
12 |   |   |   O ← O ∪ {(u, ∞, cnt_u)};
13 |   |   if u ∉ C then
14 |   |   |   d'_u ← τ(e, t − d_v) + d_v;
15 |   |   |   d_u ← min(d_u, d'_u);
16 |   |   if μ(θ(e)) ∈ {'csct', 'csdt'} then
17 |   |   |   d'_u ← τ(e, t − d_v) + d_v;
18 |   |   cnt_u ← cnt_u − 1 ;
19 |   |   if u ∈ C ∧ cnt_u = 0 then
20 |   |   |   C ← C \ {u} ;
21 |   |   if μ(θ(e)) ∈ {'csct', 'csdt'} then
22 |   |   |   if d'_u ≤ d_max then
23 |   |   |   |   Output (e, 0, λ(e)) ;
24 |   |   |   else
25 |   |   |   |   Output (e, o, λ(e)), where d((e, o), q, t) = d_max ;
26 |   if cnt_v = 0 then
27 |   |   C ← C \ {v} ;
28 |   V^iso ← V^iso ∪ {(v, d_v))};
```

Example 4.2.2. Figure 4.3 shows the complete network expansion of MINEX using our running example, where $d_{max} = 5\,\text{min}$, $t_q = $ 06:06:00, and $s = 2\,\text{m/s}$. Bold

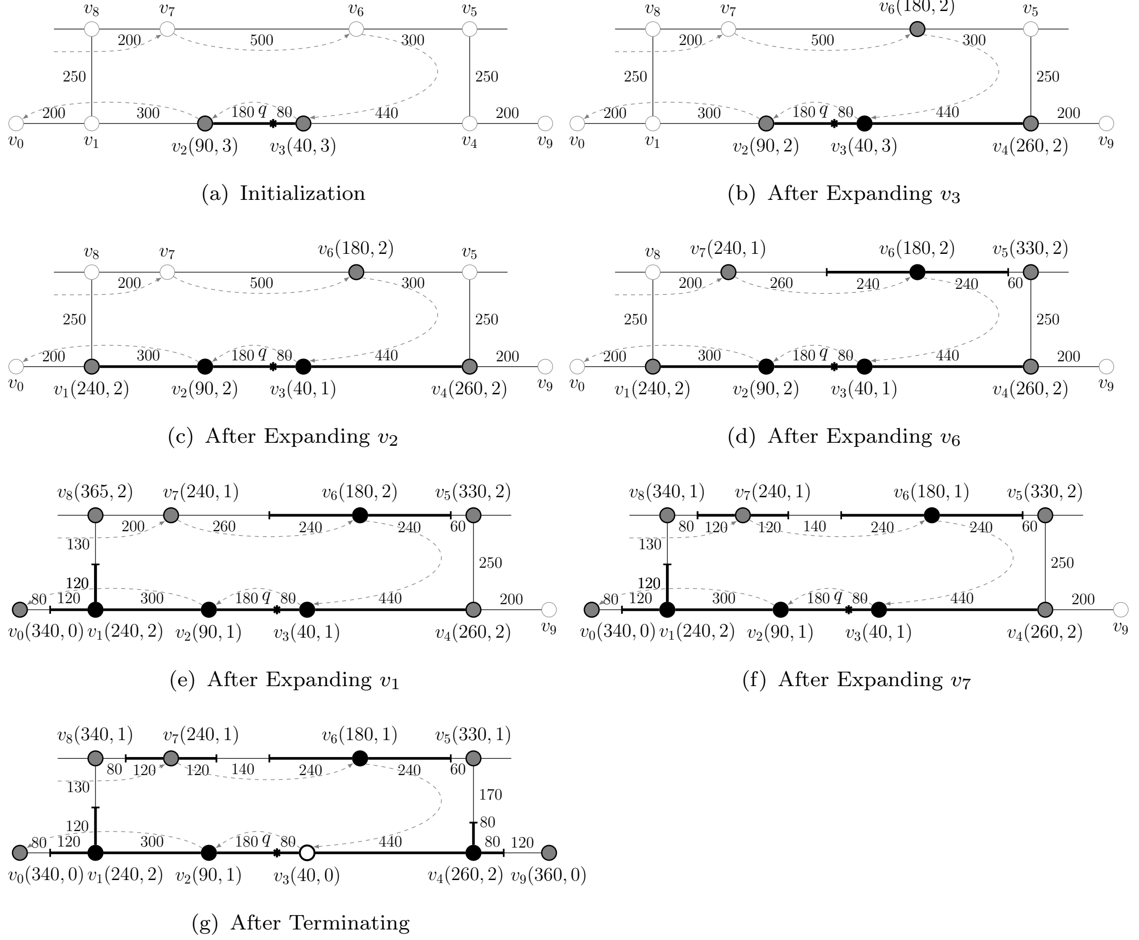
lines indicate reachable network portions which at the end make up the isochrone. Solid black nodes are closed, gray nodes are open, bold white nodes are expired, and white nodes with a thin border are not yet loaded. Only open and closed vertices are actually kept in memory. The numbers in parentheses are the network distance and the counter of a vertex. To simplify the illustration, pedestrian edges are represented by undirected lines, where each such line represents a pair of edges in opposite directions.

Figure 4.3(a) shows the isochrone after the initialization step with $C = \{\}$ and $O = \{(v_3, 40, 3), (v_2, 90, 3)\}$. Vertex v_3 has the smallest distance to q and is expanded next (Figure 4.3(b)). The distance of the visited vertices is $d_{v_4} = 40\,\mathrm{s} + \frac{440m}{2m/s} = 260\,\mathrm{s}$ and $d'_{v_2} = 40\,\mathrm{s} + \frac{260m}{2m/s} = 170\,\mathrm{s}$, which does not improve the old value $d_{v_2} = 90\,\mathrm{s}$. For the distance of v_6, we determine the required arrival time at v_3 as $t = t_q - d_{v_3} = 06{:}06{:}00 - 40\,\mathrm{s} = 06{:}05{:}20$ and the latest bus departure at v_6 as 06:03:00, yielding $d_{v_6} = 40 + (06{:}05{:}20 - 06{:}03{:}00) = 180\,\mathrm{s}$. After updating the counters (v_2, v_4, v_6), the new vertex sets are $C = \{(v_3, 40, 3)\}$ and $O = \{(v_2, 90, 2), (v_6, 180, 2), (v_4, 260, 2)\}$. Next, v_2 is expanded as shown in Figure 4.3(c), where $(v_1, 240, 2)$ is added to O. Since vertex v_3 is closed, we output the reachable (continuous) edge (v_3, v_2). In the successive iteration shown in Figure 4.3(d) vertex $(v_6, 180, 1)$ is expanded. Its neighbors are updated with their distances to $(v_7, 240, 1)$ and $(v_5, 330, 2)$. Next, we expand in Figure 4.3(e) vertex v_1, add $(v_8, 365, 2)$ and $(v_0, 340, 2)$ to O, and decrement the counter of the closed neighbor v_2 by 1. The next vertex to expand is v_7 as illustrated in Figure 4.3(f). The network distance of the neighbor v_8 is updated to $(v_8, 340, 1)$ and the counter of v_6 is decremented by 1. Finally, Figure 4.3(g) shows the isochrone after expanding vertex v_6, when the algorithm terminates. The gray vertex v_3 is expired, since the counter of outgoing edges is zero. In fact, it is easy to see that vertex v_3 would not be visited anymore in future expansion steps, since all adjacent vertices are already closed. They only way to visit v_3 again would be to expand an adjacent vertex.

Notice that an algorithm that alternates between first (completely) expanding in the discrete network and then from each reachable vertex start a complete expansion in the continuous network is sub-optimal in most of the cases. The reason is that portions of the network might be loaded and expanded several times.

4.2.3 Vertex Expiration

Closed vertices are needed to avoid cyclic network expansion. In this section we introduce *expired vertices* (Definition 4.2.1) which allow us to limit the number of closed vertices that need to be kept in memory. Expired vertices are never revisited



(a) Initialization

(b) After Expanding v_3

(c) After Expanding v_2

(d) After Expanding v_6

(e) After Expanding v_1

(f) After Expanding v_7

(g) After Terminating

Figure 4.3: Step-by-step Network Expansion of MINEX for $d_{max} = 5\,\text{min}$, $s = 2\,\text{m/s}$, and $t_q = $ 06:06:00.

in future expansion steps, hence they are not needed to prevent cyclic expansions and can be removed (Lemma 1). Isochrones contain insufficient information to handle vertex expiration (Lemma 2). Therefore, MINEX uses a counter-based solution to correctly identify expired vertices and to eagerly expire vertices during the expansion (Lemma 3).

To facilitate the discussion we introduce a couple of auxiliary terms. For a vertex v, the term *in-neighbor* refers to a vertex u with an edge (u, v). The term *out-neighbor* refers to a vertex w with an edge (v, w). The term *in-degree* denotes the number of incoming edge and the term *out-degree* represents the number of outgoing edges. Recall that the status of vertices changes from open (O) when they are encountered first to closed (C) when they are expanded, and finally to

expired (X) when they are expired; the sets O, C, and X are pairwise disjoint.

Definition 4.2.1 (Expired Vertex)**.** A closed vertex, $u \in C$, is *expired* if all its out-neighbors are either closed or expired, i.e. $\forall v((u, v) \in E \Rightarrow v \in C \cup X)$.

Example 4.2.3. Consider the isochrone in Figure 4.3(g). Vertex v_3 is expired since v_2 and v_4 are closed, and v_3 has no other out-neighbors. In contrast, v_2 is not yet expired since the out-neighbor v_0 is not yet closed (and the expansion of v_0 leads back to v_2).

Lemma 1. An expired vertex u will never be revisited during the computation of the isochrone and can be removed from C without affecting the correctness of MINEX.

Proof. There is only one way to visit a vertex u during network expansion: u has an out-neighbor v (that is connected via an edge $(u, v) \in E$) and $v \in O$; the expansion of v visits u. Since according to Definition 4.2.1 all of u's out-neighbors are closed or expired, and closed and expired vertices are not expanded (line 11 in Algorithm 3), u cannot be revisited. □

The identification of expired vertices according to Definition 4.2.1 has two drawbacks: (1) it requires a database access to determine all out-neighbors since not all of them might already have been loaded, and (2) the set X of expired vertices must be kept in memory, which we want to avoid.

Lemma 2. If the isochrone is used to determine the expiration of a closed vertex, $u \in C$, the database must be accessed to retrieve all of u's out-neighbors, and X needs to be stored in memory.

Proof. According to Definition 4.2.1, for a closed vertex u to expire we have to check that all out-neighbors v are closed or expired. The expansion of u loaded all out-neighbors v that have also an inverse edge, $(v, u) \in E$. For out-neighbors v that are not connected by an inverse edge, $(v, u) \notin E$, we have no guarantee that they are loaded. Therefore, we need to access the database to get *all* adjacent vertices. Next, suppose that X is not maintained in memory and there exists an out-neighbor v of u without an inverse edge, i.e. $(v, u) \notin E$. If v is in memory, its status is known. Otherwise, either is v already expired and has been removed, or it has not yet been visited. In the former case, u shall expire, but not in the latter case, since the expansion of v (re)visits u. However, with the removal of X we loose the information that these vertices already expired, and we cannot distinguish anymore between not yet visited and expired vertices. □

Example 4.2.4. The isochrone does not contain sufficient information to determine the expiration of v_2 in Figure 4.3(c). While v_1 and v_3 are loaded and their status is known, the out-neighbor v_0 is not yet loaded (and actually violates the condition for v_2 to expire). To ensure that *all* out-neighbors are closed, a database access is needed. Next, consider Figure 4.3(g), where v_3 is expired, i.e. $X = \{v_3\}$. To determine the expiration of v_2, we need to ensure that $v_3 \in C \cup X$. If X is removed from memory, the information that v_3 is already expired is lost. Since v_3 will never be revisited, v_2 will never expire.

To correctly identify and remove all expired vertices without the need to access the database and explicitly store X, MINEX maintains for each vertex, u, a counter, cnt_u that keeps track of the number of outgoing edges of u that have not yet been traversed.

Lemma 3. Let cnt_u be a counter associated with vertex $u \in V$. The counter is initialized to the number of outgoing edges, $cnt_u = |\{(u, v) \mid (u, v) \in E\}|$, when u is encountered for the first time. Whenever an out-neighbor v of u is expanded, cnt_u is decremented by 1. Vertex u is expired iff $u \in C$ and $cnt_u = 0$.

Proof. Each vertex v expands at most once (when it is dequeued from O), and the expansion of v traverses all incoming edges (u, v) and decrements the counter cnt_u of vertex u by 1. Thus, each edge in the network is traversed at most once. When $cnt_u = 0$, vertex u must have been visited via all of its outgoing edges. From this we can conclude that all out-neighbors have been expanded and are closed, which satisfies the condition for vertex expiration in Definition 4.2.1. □

Example 4.2.5. In the isochrone in Figure 4.3(g), vertex v_3 is expired and can be removed since $cnt_{v_3} = 0$ and $v_3 \in C$. Vertex v_2 expires only when v_0 is expanded and counter cnt_{v_2} is decremented to 0. Similar, vertex v_6 expires when v_5 is expanded.

Lemma 4. Vertices cannot be expired according to a Last Recently Used (LRU) strategy.

Proof. We show a counter-example in Figure 4.4, which illustrates a multimodal network expansion that started at q. Although q has been expanded and closed first, it cannot be expired because an edge from vertex v, which will be expanded later, leads back to q and would lead to cyclic expansions. In contrast, the white vertices that are expanded and closed after q can be expired safely. □

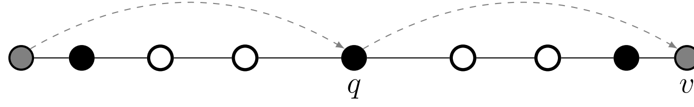
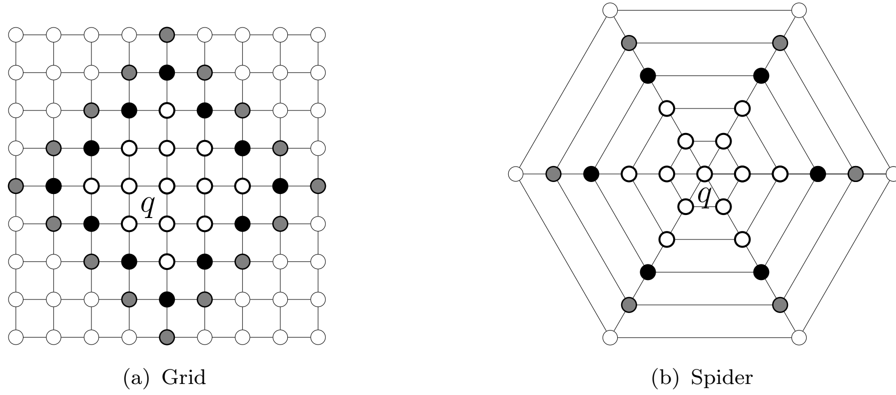


Figure 4.4: LRU Strategy.

4.2.4 Properties

Vertex expiration ensures that the memory requirements of the algorithm MINEX are reduced to a tiny fraction of the isochrone. Figures 4.5(a) and 4.5(b) illustrate the isochrone size and MINEX's memory complexity for grid and spider networks, respectively. Solid black vertices (C) and gray (O) are stored in memory, whereas white vertices with vertices with a bold border are expired (X) and removed from memory. The other vertices are not yet visited and loaded.



(a) Grid (b) Spider

Figure 4.5: Network Expiration.

The following two lemmas provide a bound for the isochrone size and MINEX's memory complexity for grid and spider networks. Only the pedestrian mode is considered, though the results can easily be extended to multimodal networks.

Lemma 5. The size of an isochrone, $|V^{iso}|$, is $\mathcal{O}(d_{max}^2)$ for a grid network and $\mathcal{O}(d_{max})$ for a spider network and a central query point q.

Proof. Consider the grid network in Figure 4.5(a). Without loss of generality, we measure the size of an isochrone as the number of its vertices (i.e. open, closed, and expired vertices), and we assume a uniform distance of 1 between connected vertices. The size of an isochrone with distance $d = 1, 2, \ldots$ is given by the recursive formula $|V^{iso}|_d = |V^{iso}|_{d-1} + 4d$ with $|V^{iso}|_0 = 1$; $4d$ is the number of new vertices that are visited when transitioning from distance $d-1$ to d (i.e. the number of vertices at distance d that are visited when all vertices at distance $d-1$

are expanded). This forms an arithmetic series of second order (1, 5, 13, 25, 41, 61, ...) and can also be written as $|V^{iso}|_d = 1 + \sum_{i=0}^{d} 4i = 2d^2 + 2d + 1$, which yields $|V^{iso}| = \mathcal{O}(d_{max}^2)$.

Next, consider the spider network in Figure 4.5(b). Without loss of generality, we assume a uniform distance of 1 between all adjacent vertices along the same outgoing edge from q. It is straightforward to see that the size of the isochrone is $|V^{iso}| = deg(q) \cdot d_{max} + 1 = \mathcal{O}(d_{max})$, where $deg(q)$ is the degree of vertex q. □

Lemma 6. The memory complexity of MINEX is $|O \cup C| = \mathcal{O}(d_{max}) = \mathcal{O}(\sqrt{|V^{iso}|})$ for a grid network and $\mathcal{O}(1)$ for a spider network with a central located query point q.

Proof. Recall that MINEX keeps in memory only the open and closed vertices, $O \cup C$. Consider the grid network in Figure 4.5(a). By referring to the proof of Lemma 5, the cardinality of the open vertices at distance d can be determined as $|O|_d = 4d$ and the cardinality of the closed vertices as $|C|_d = 4(d-1)$. Thus, the memory requirements in terms of d_{max} are $|O \cup C| = \mathcal{O}(d_{max})$.

To determine the memory requirements depending on the size of the isochrone, $|V^{iso}|$, we use the formula for the size of an isochrone from the proof of Lemma 5 and solve the quadratic equation $2d^2 + 2d + 1 - |V^{iso}|_d = 0$, which has the following two solutions:

$$d_{1,2} = \frac{-2 \pm \sqrt{2^2 - 4 \cdot 2 \cdot (1 - |V^{iso}|_d)}}{2 \cdot 2} = \frac{-1 \pm \sqrt{2|V^{iso}|_d - 1}}{2}.$$

Since the result must be positive,

$$d = \frac{-1 + \sqrt{2|V^{iso}|_d - 1}}{2}$$

is the only solution. By substituting d in the above formulas for open and closed vertices we get, respectively,

$$|O|_d = 4d = 4\frac{-1 + \sqrt{2|V^{iso}|_d - 1}}{2}$$

and

$$|C|_d = 4(d-1) = 4(\frac{-1 + \sqrt{2|V^{iso}|_d - 1}}{2} - 1),$$

which proves $|O \cup C| = \mathcal{O}(\sqrt{V^{iso}})$.

Next, we consider the spider network in Figure 4.5(b) with the query point q in the center. It is straightforward to see that the cardinality of the open and closed vertices is $|O \cup C| = 2 \cdot deg(q) = \mathcal{O}(1)$, where $deg(q)$ is the degree of vertex q. □

Theorem 4.2.1. Algorithm MINEX is optimal in the sense that all loaded vertices and '*csct*'/'*csdt*' edges are part of the isochrone, and each of these edges is loaded and traversed only once.

Proof. When a vertex v is expanded, all incoming edges $e = (u, v)$ are loaded and processed (Algorithm 3, line 11). If e is a '*csct*'/'*csdt*' edge, the reachable portion of e (including the end vertices u and v) is added to the isochrone (line 25). While u might not be reachable, v is guaranteed to be reachable since $d_v \leq d_{max}$. In contrast, '*dsct*'/'*dsdt*' edges are not added since they are not part of the isochrone; only the end vertices u and v are accessible, which are added when the incoming '*csct*'/'*csdt*' edges are processed. Therefore, since each vertex is expanded at most once each edge is loaded at most once, and all loaded edges except '*dsct*'/'*dsdt*' edges are part of the isochrone. □

From this theorem we can conclude that search space pruning techniques as applied for shortest path computation are not applicable for isochrones.

Complexity. The runtime complexity of MINEX is the same as MINE. However, thanks to the vertex expiration the size of the min-heap does not grow with the size of the isochrone, but remains a small fraction of the isochrone.

4.3 A Hybrid Approach

Thanks to vertex expiration, MINEX has a very small memory footprint which is almost constant even for very large isochrones. However, it is not scalable in terms of runtime for several reasons. First, the number database accesses is very large, since each expansion step loads only the incoming vertices to the expanded vertex. Second, since the algorithm follows a greedy expansion strategy, in which the vertex with the shortest distance is expanded next, space locality is generally not guaranteed. That is, two consecutive expanded vertices may reside on totally different places on the disk. Third, the full capacity of a disk block is not utilized since the incoming edges do not fill up the entire transferred block, and caching might not be very effective due the lack of spatial locality during the expansion. As a result, the algorithm has large I/O costs, which significantly decrease the runtime of MINEX for large isochrones.

To improve the runtime scalability, we propose a solution, termed *Multimodal Range Network Expansion with network eXpiration* (MRNEX), which decreases the I/O costs by taking advantage of spatial locality on the disk and reducing the number of disk accesses in combination with filling up transferred blocks with data that is needed next. Spatial locality on disk can be obtained by adding a primary index on the table based on the spatial property. A maximal filling of the

blocks can be gained by transferring the data as chunks clustered on their spatial property. We propose two algorithms that differ mainly in the way how the size of the network chunks are determined that are loaded in each database access. In the first solution, the size of the junk is determined by the radius of the residual distance (assuming that there is sufficient memory). The second solution allows to control the size of the chunks depending on the available memory capacity.

4.3.1 Multimodal Range Network Expansion

During the network expansion, the vertices pass through different states. An open vertex is loaded in memory, but not yet expanded. A open vertex changes to closed, when it is expanded. Finally, a closed vertex expires, when it is guaranteed that it cannot be revisited again during network expansion. For range network expansion an additional state is needed for open vertices, which we called *stalled*.

Before we describe the multimodal range expansion we introduce the concept of *stalled vertices* and *range queries*.

Definition 4.3.1 (Stalled Vertex)**.** A open vertex, $v \in O$, is *stalled* if none of its incoming edges is loaded in memory.

A stalled vertex essentially blocks network expansion in main memory and signals that the next chunk of the network needs to be loaded.

Definition 4.3.2 (Range query)**.** Let N be a multimodal network, $v_q \in V$ be a vertex in the network and R be a range that covers v_q. A *range query* Q with vertex v_q and query range R retrieves all edges (u, v) such that v is located inside R, i.e.

$$Q = \{(u, v) \in E \mid v \in R\}.$$

Multimodal range network expansion adopts the same incremental expansion strategy as MINEX with the important difference that each database access is a range query that loads a larger chunk of the network rather than only the incoming edges of the expanded vertex. Let N be a multimodal network with query point q, arrival time t at q, maximal duration d_{max}, and walking speed s. Network expansion starts at the query point q, which either coincides with a vertex or is mapped to the closest edge.

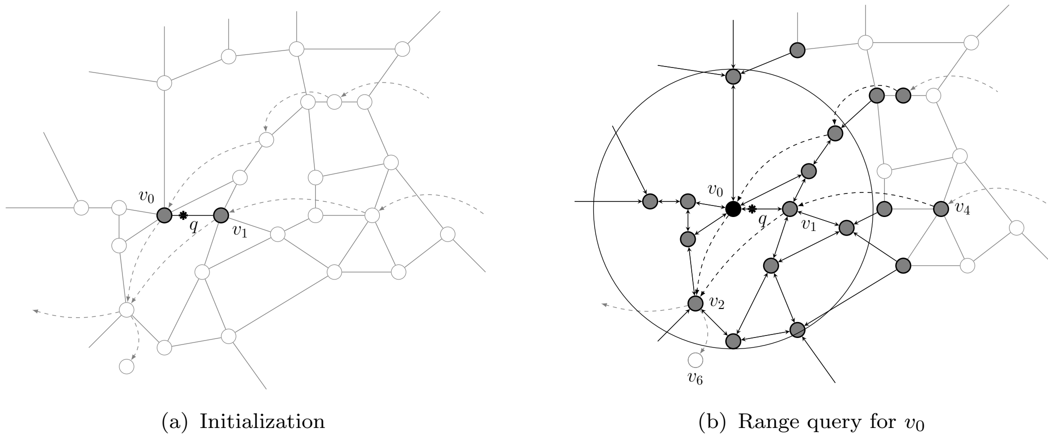
Then, network expansion starts with expanding a stalled vertex v. A range query with the center point v and a radius that is determined as $(d_{max} - d_v) * s$ is issued to load the network portion that is within walking distance to v.

If q coincides with a vertex and only the pedestrian mode is considered or there are no public transport stops in this area, the query range is an upper bound for

the isochrone. If public transport systems are present, network expansion might exceed the query range and encounter a stalled vertex, which blocks the expansion since no incoming edges are yet loaded. Therefore, a new range query is needed to load the next chunk from disk in order to continue the expansion process. The range is determined by the time that is still available, i.e. $d_{max} - d(v, q, tq)$. In order to avoid overlapping between range query and hence multiple loading of the same network portions, we intersect each new query range with the ranges of the previous queries.

Every range query requires a DB lookup using a spatial operator that retrieves all edges whose target vertex is inside the query range. The range is defined as a geometry of type polygon having initially the shape of a circle. In order to avoid that edge are loaded multiple times, the intersected areas from previous loaded ranges are not considered.

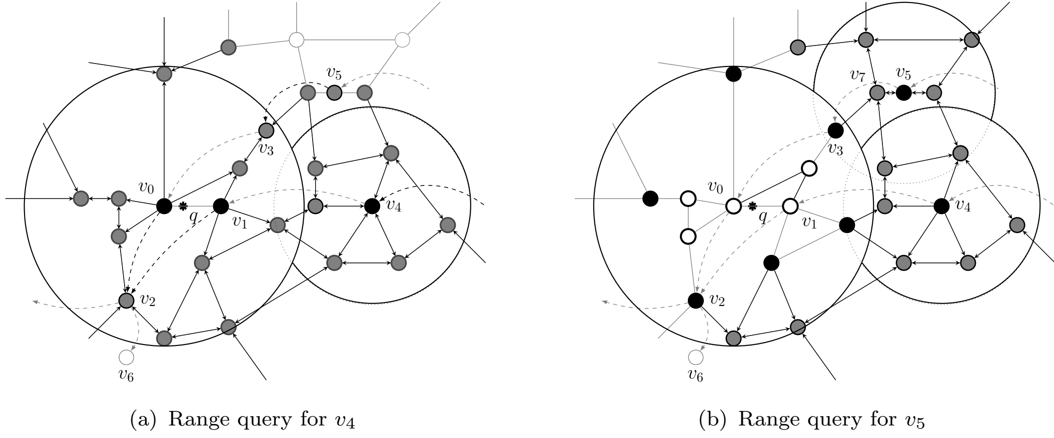
Example 4.3.1. Figure 4.6(a) illustrates the initial situation after mapping the query point q to the edges (v_0, v_1) and (v_1, v_0). The two edges are loaded in memory and vertices v_0 and v_1 are marked as stalled.



(a) Initialization (b) Range query for v_0

Figure 4.6: Multimodal Range Network Expansion.

Vertex v_0 has the smaller distance to q and is expanded first. Since v_0 is stalled, a range query is performed. Figure 4.6(b) illustrates the result of the network after executing the range query with radius $r_{v_0} = (d_{max} - d_{v_0}) * s$, i.e. the remaining time multiplied with the walking speed. Black lines represent edges that have been loaded, since their end vertex is inside the query range. Gray vertices represent the loaded vertices, which are all marked as open. Notice the vertices outside the query range, which are connected to a target vertex which is inside the query range. For instance, vertex v_4 is loaded because v_1 (i.e. the target of the bus edge



(a) Range query for v_4 (b) Range query for v_5

Figure 4.7: Multimodal Range Network Expansion.

$(v_4, v_1))$ is in the query range. In contrast, v_6 is not loaded, because it is outside the query range and has no outgoing edge to a vertex inside the range.

After completing the range query, network expansion proceeds with the expansion of vertex v_0. The expansion of v_1 drives the expansion outside the query range to vertex v_4, which is marked as stalled. When v_4 becomes the vertex to be expanded next, a new range query is executed as illustrated in Figure 4.7(a). The radius is set on the remaining distance of v_4 multiplied with s and the intersecting part with the range of the previous range query is subtracted.

Figures 4.7(b) shows the situation after a range query for the stalled vertex v_5 has been executed. For instance, edge (v_3, v_7) is loaded because v_7 is within the query range, but the closed vertex v_3 is not added to O. Note that since v_3 is not in the loaded range, none of its incoming edges are loaded. This guarantees that every edge is loaded at most once. The vertices in the query range of v_0 that are represented by white circles are expired and have already been removed from memory.

Algorithm 4 shows the algorithm MRNEX. The initialization part until line 5 is identical to MINEX and maps the query point to a vertex or an edge in the network. The auxiliary set R^{ier} tracks the areas of the range queries by storing the center and the radius of each query range. R^{ier} is initialized to the empty set. At line 7 the network expansion starts by retrieving the vertex v with the shortest distance from the set O of open vertices. If v is a stalled vertex, i.e. the in-degree is zero, a new range query needs to be issued. The query range is determined as a circle R with centre v and radius r_v (line 10), which is the maximal walking distance with the available time. To avoid that network portions are loaded multiple times, the overlapping parts with previous query ranges in R^{ier}

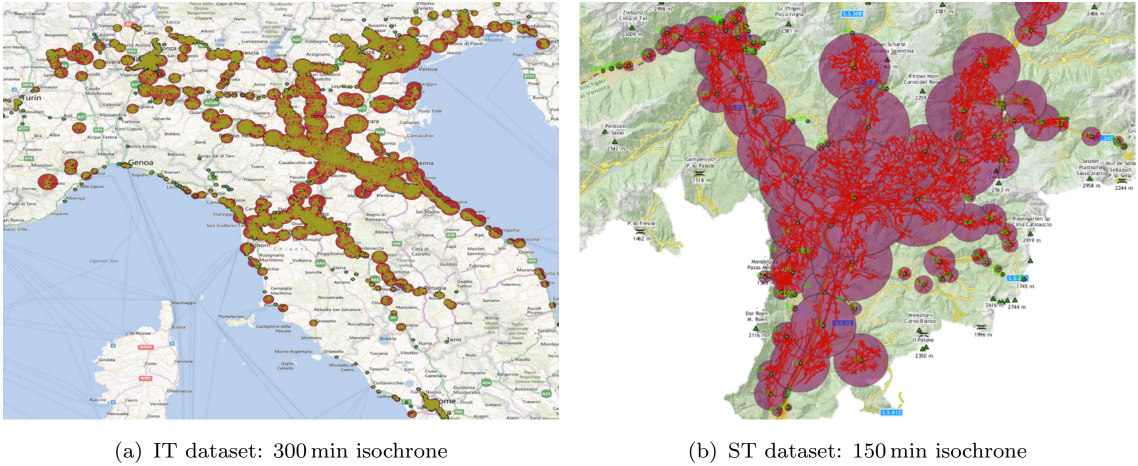
are subtracted from R. The resulting range R that forms either circle or a polygon is passed to the range query (line 15). The returned edges are added in memory and only vertices that are neither open nor closed are inserted in O (lines 18 and 20). Since only incoming edges of the vertices that reside in that range are loaded in memory, this guarantees that expired vertices are not reloaded. The rest of the algorithm is identical to MINEX, including vertex expiration.

Algorithm 4: $MRNEX(q, t, d_{max}, s, \mathsf{N})$.

```
   input  : q, d_max, s, t, N
   output: E^iso, V^iso
1  if q coincides with v then
2  |   O ← {(v, 0, cnt_v)};
3  else// q = ((u, v), o) = ((v, u), o')
4  |   O ← {(u, o/s, cnt_u), (v, o'/s, cnt_v)};
5  |   Output ((u, v), max(0, (o−d_max/s), o) and ((v, u), max(0, (o'−d_max/s), o');
6  R^ier ← ∅;
7  while O ≠ ∅ ∧ (v, d_v, cnt_v) ← dequeue(O) ∧ d_v ≤ d_max do
8  |   if indegree(v) = 0 // v is stalled then
9  |   |   r_v ← (d_max − d_v) * s;
10 |   |   R ← create circle with center v and radius r_v ;
11 |   |   foreach (v', r_v') ∈ R^ier do
12 |   |   |   if d_ε(v, v') < r_v + r_v' then
13 |   |   |   |   R ← R \ R ∩ ρ(v', r_v');
14 |   |   R^ier ← R^ier ∪ {(v, r_v)};
15 |   |   Q ← issue range query on R ;
16 |   |   foreach (u', v') ∈ Q do
17 |   |   |   if u' ∉ O ∪ C then
18 |   |   |   |   O ← ∪{(u', ∞, cnt_u')} ;
19 |   |   |   if v' ∉ O ∪ C then
20 |   |   |   |   O ← ∪{(v', ∞, cnt_v')} ;
   |   // ...proceed expansion as MINEX
```

Example 4.3.2. Figure 4.8 illustrates the expansion strategy of MRNEX for two isochrones on the two datasets Italy (IT) and South Tyrol (ST). The query range is decreasing as network expansion proceeds. It is easy to see that not all vertices/edges that are loaded in the range queries are also part of the final isochrone. For instance, dataset IT contains 13% false positives and ST has 18%.

In the experimental evaluation in Chapter 5 we show that MRNEX significantly improves the runtime, but the memory consumption is also slightly higher. That is, MRNEX is not space optimal, because the range queries may retrieve false positive vertices, i.e. vertices that are loaded in main memory, but are not part of the isochrone. In the experimental evaluation we analyze in detail the ratio of false positive vertices.



(a) IT dataset: 300 min isochrone

(b) ST dataset: 150 min isochrone

Figure 4.8: Illustrating Range Query Expansion on two Real-world Data Sets.

Complexity. MRNEX has the runtime complexity $\mathcal{O}((|V|+|E|)\log|V|)+klog|V|$, where k is the number of stalled expanded vertices. However, the loading of small chunks reduces the number of database accesses, which significantly improves the runtime. The memory complexity is larger than for MRNEX since the adjacency list is in memory and O is larger because of the false positives.

4.3.2 Setting an Upper Bound for Range Queries

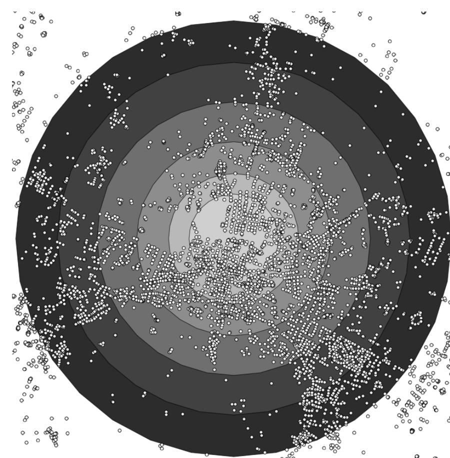
The eager loading strategy of MRNEX, i.e. to load network chunks that are as large as possible, determined only by the available timespan, turned out to perform bad in some situations. We observed in experiments that for large isochrones the memory consumption can become quite high. Already the first range query can load a very large portion of the network. Moreover, if high speed public transport systems are present, the expansion along these lines is done very early and exceeds the network portions that have already been loaded. As a consequence, many range queries are issued very early in the expansion process, although large parts of the loaded network portions are only visited later on.

To tackle this problem, we propose to consider the available memory in addition to the available timespan for the calculation of the query range. Instead of eagerly loading maximal ranges, the loaded network chunks should be upper bounded by a percentage of the available memory. Such a strategy allows to better use the memory for range queries in areas where the expansion is actually active.

To implement this strategy efficiently, we precompute for each vertex different ranges that include a different number of vertices. That is, we precompute different query ranges for different sets of k vertices (or network portions). Figure 4.9

V	k	Radius
v_1	500	1.7
v_1	1000	2.5
v_1	2000	3.7
v_1	3000	5.3
v_1	4000	6.8
v_1	5000	8.5
⋮	⋮	⋮

(a)



(b)

Figure 4.9: Precomputed Query Ranges for Vertices.

illustrates this idea. In the table in Figure 4.9(a), each entry is a precomputed range. The first column contains the vertex identifier, the second column the number of vertices k in the range, and the third column the radius of the range. When a range query is issued, the radius is determined by the largest k that fits in memory. More specifically, let v be a stalled vertex, m be the maximum size for the result of a range query, and k with radius r_k be the maximum range (i.e. the maximum number of vertices) that fit in m. Then, the radius of the range query is determined as

$$r_v = min(r_k, (d_{max} - d_v) * s).$$

Efficient Preprocessing

A naive algorithm to precompute the query range for a junk of size k is to scan the entire table and sort the vertices according to the Euclidean distance in order to determine the distance of the k-th vertex. This naive approach is very time consuming. For example, the computation of all query ranges for the IT dataset takes approximately 54 days. By starting with a predefined, small range and adapting the range until it contains at least k elements, we were able to reduce the precomputation time to 14 days, which is still not acceptable.

To improve the precomputation time, we developed a more efficient solution shown in Algorithm 5 that computes all query ranges for the IT dataset in less than five hours. The algorithm works as follows. The vertices are loaded into an array V and sorted by the x-coordinate. To determine the query ranges for a vertex $V[i]$, the Euclidean distance needs to be computed only for a reduced set

of vertices. Assume that we want to compute the query range for the k closest vertices to the first vertex $V[0]$. We use a max-priority queue H of fixed size k to keep the k-nearest neighbors. The Euclidean distance of the first element in the queue determines the query range. We fill H with the first k elements of V. To guarantee that there are no other vertices that are closer than the top element in H, the array V is scanned until a vertex is encountered that has an x-coordinate which is larger than the Euclidean distance of the top element in H. If during this scan a vertex v is found that is closer to $V[0]$ than the top element in H, the top element is removed and v inserted in the priority queue.

To determine the query range for an intermediate vertex $V[i]$, $i > 0$, we have to scan both forward and backward until we encounter vertices that are more distant from $V[i]$ (along the x-axis) than the Euclidean distance of the top element in H. This strategy is implemented in the algorithm in while loop at line 5, which iterates as long as the x-range between the outermost vertices $V[l]$ and $V[r]$ becomes smaller than the diameter of the circle with center $V[i]$. When the loop terminates, the elements in the queue are the k closest elements to $V[i]$, and the query radius corresponds to the Euclidean distance of the top element in the queue.

Algorithm 5: Precomputation of the Query Ranges.

```
   input  : V, K[]
   output: M
1  V[] ← sort vertices by x coordinate;
2  H ← ∅; i ← 1; M ← ∅;
3  foreach v ∈ V do
4      k ← ∞; l ← i ; r ← i; idx ← 1;
5      while V[r].x − V[l].x ≤ min(2k, k + min(V[i].x − V[1].x, V[|V|].x − V[i].x)) ∧ idx ≤ |V|  do
6          if l = 0 then // left out of range check
7              idx ← ++r;
8          else if r = |V| then // right out of range check
9              idx ← --l;
10         else // select next closest element
11             if V[i].x − V[l − 1].x < V[r + 1].x − V[i].x then
12                 idx ← --l;
13             else
14                 idx ← ++r;
15         if |H| < K[|K|] // fill heap until its size reaches largest measurement point then
16             insert(H, d_ε(V[i], V[idx]));
17         else if findmax(H) > d_ε(V[i], V[idx]) then
18             deletemax(H);
19             insert(H, d_ε(V[i], V[idx]));
20     D[] ← ∅; j ← |H|;
21     while H ≠ ∅ do
22         D[j--] ← findmax(H); deletemax(H);
23     foreach k ∈ K do
24         M ← M ∪ {(v, k, D[k])};
25     i++;
26 return M;
```

Example 4.3.3. The algorithm is illustrated in Figure 4.10. Assume that we want to compute the 4-NN for vertex v_0, which is the first vertex in the array. First, the queue is filled with the first four vertices, i.e. $H = \langle v_1, v_2, v_4, v_3 \rangle$. Then, the scan of the array continues, and v_5 is encountered. Since v_5 is closer to v_0 than the top element v_1, vertex v_5 is inserted in the queue and the top element is removed, yielding $H = \langle v_5, v_2, v_4, v_3 \rangle$. The next vertex in the array is v_6, which along the x-axis is more distant from v_0 than the Euclidean distance of v_5. The while loop terminates, and $\langle v_3, v_4, v_2, v_5 \rangle$ are the four nearest neighbors to v_0.

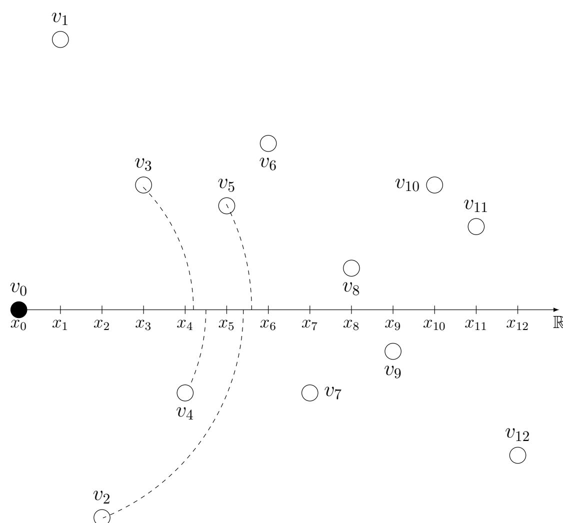


Figure 4.10: Precomputing the 4-NN for Vertex v_0.

Note that in the neighbor search both predecessors and successors have to be considered. The x-difference is at most twice the largest Euclidean distance in the queue. The dimension along which the array V shall be sorted depends on the density of the data. That is, if the vertices are closer along the y-axis, the array should be sorted along the y-coordinate of the vertices.

4.4 Summary

In this chapter we introduced three implementations for computing isochrones in a multimodal network. MDijkstra algorithm works well for large d_{max} where the size of the isochrone becomes equal to the network. However it suffers in the high initializations costs to load the entire network in main memory. We proposed the MINEX algorithm, which is independent of the actual network size and depends only on the size of the isochrone. MINEX is optimal regarding that

only those network portions are loaded that eventually will be part of the isochrone. The concept of expired vertices reduces MINEX's memory requirements to keep in memory only the minimal set of expanded vertices that is necessary to avoid cyclic expansions. To identify expired vertices, we proposed an efficient solution based on counting the number of outgoing edges that have not yet been traversed. However, MINEX is not scalable in runtime because of the large number of DB lookups $\mathcal{O}(V^{iso})$, which depends on the vertex size of the isochrones. MRNEX and its extension MRNEXIM drastically reduced the number of DB lookups by loading the network in main memory in chunks. This approach is a good trade-off between scalability in runtime and scalability in memory. The empirical evaluation in the following chapter confirms this statement.

CHAPTER 5

Experimental Evaluation

In this chapter we describe the results of a detailed empirical evaluation of our algorithms both on synthetic as well as on real-world datasets. We analyze the runtime and the memory consumption of the algorithms presented in the previous section. The evaluation confirms the analytical results and the scalability in terms of runtime and memory consumption.

5.1 Data Sets

In our evaluation we use four real-world datasets that vary in the network topologies, in the network density, in the number of transportation modes, and in the frequency of active transportation systems. These datasets are summarized in Table 5.1. The second column (Size) denotes the network size in Megabytes, whereas the other columns are using as cardinality the number of tuples in the corresponding relation. $|V|$ represents the total number of vertices, $|E|$ the total number of edges, $|E_{'csct'}|$ the number of continuous (pedestrian) edges, $|E_{'dsdt'}|$ the number of discrete edges, and $|S|$ the size of the schedule table.

Data	Size	$\|V\|$	$\|E\|$	$\|E_{'csct'}\|$	$\|E_{'dsdt'}\|$	$\|S\|$
IT	2,128	1,372.0	3,633.7	3,633.1	0.6	1.3
ST	137	77.7	197.8	182.4	9.4	179
SF	138	33.6	96.4	90.0	6.4	1,112
BZ	21	3.2	8.3	6.5	1.9	118

Table 5.1: Real-World Data Sets: Italy (IT), South Tyrol (ST), and San Francisco (SF), and Bozen-Bolzano (BZ).

The BZ dataset represents a small urban network and contains the street network in combination with the public transport network of Bozen-Bolzano, by courtesy of the Municipality of Bozen-Bolzano and the local transportation company SASA. If not specified otherwise, the default query point is a central square in the city with arrival time 10:10 am on a weekday. At that time the frequency of the means of transport is high: in general every 10-15 minutes a bus of the same line passes to a certain stop station.

The ST dataset represents a regional network and contains the South-Tyrolean street network in combination with different means of transport (train, bus, funicular and cable car), by courtesy of the Province of Bozen-Bolzano and the local transportation companies SAD and SII. The default query point is close to the railway station of Bozen-Bolzano with arrival time 10:30 am on a weekday, which allows to catch a large number of transportation systems.

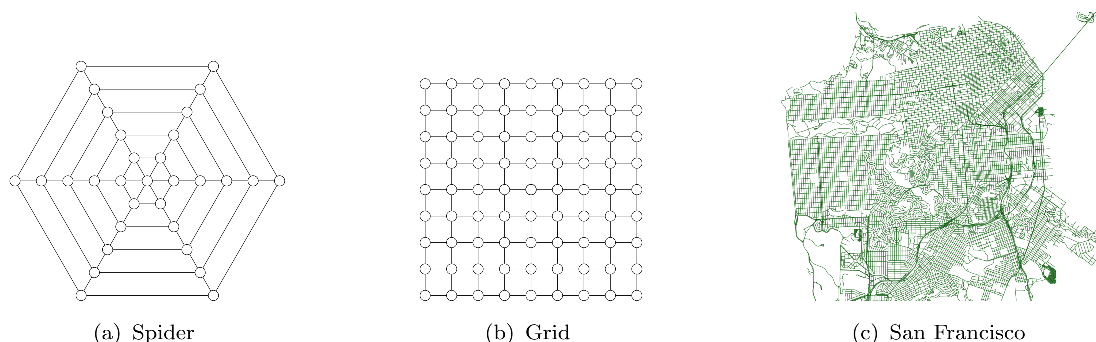
The SF dataset represents an urban network and contains the street network of San Francisco (imported as sample data from NAVTEQ[1]) and the schedules of all public means of transport that are available in Google Transit format from SFMTA. The default query point is the San Francisco public library with arrival time 09:15 pm on a weekday. The library is located in the center of the city.

The IT data set represents a large skewed network and contains the Italian street network in combination with the national train network that connects more

[1] http://sampledata.navteq.com

than 2200 cities. The street data were imported from OpenstreetMaps[2], and the train schedules were extracted automatically from the web page of Trenitalia[3]. The default query point is close to the railway station of Bologna with arrival time 3:00 pm on a weekday. The chosen query points and arrival times ensure a high frequency of the public transport systems.

For the evaluation we also generated two categories of synthetic data. One category has a grid topology (Figure 5.1(b)), the other a spider topology (Figure 5.1(a)). The grid relation consists of 10.000 vertices (100×100 matrix), 39.600 edges and every edge has a length of 60 meters. The spider relation has 6 outgoing axis starting from the innermost vertex and 1000 rings, which results in a total of 6001 vertices and 24.000 edges. The length of those edges that connects two rings is set to 60 meters too. The length of the edges that form a ring grows with the distance from the central vertex. All synthetic datasets represent a single continuous (pedestrian) network without any public transport systems.



(a) Spider (b) Grid (c) San Francisco

Figure 5.1: Network Topologies Used in the Experiments.

Figure 5.1(c) illustrates the network topology of the San Francisco (SF) dataset which resembles a grid network. The authors in [50] denote grid networks as raster road pattern and spider networks as radial/concentric pattern. The topology of the datasets IT, ST, and BZ resemble a branching road pattern [50].

5.2 Setup

All experiments were performed in the same environment: a 64bits virtual machine with a Intel Dual Core Xeon processor with 2.67 GHz and 3 GB RAM memory running under Ubuntu Linux (Kernel 2.6.32). All algorithms were implemented in Java, version 6.0. The communication between application and the database is established with a JDBC 4.0 driver. We use as relational database PostgreSQL 8.4

[2]hosted on http://download.gfoss.it/osm

[3]http://www.viaggiatreno.it

with the spatial extension PostGIS, version 2.0. We configure the DB as follows: *shared_buffers* that represents the reserved memory for data caching is set to 650MB, *work_mem* that is the reserved memory for complex queries and sorting is set to 512MB, and *effective_cache_size* that is reserved memory for disk caching by the operating system is set to 500MB.

Non-spatial attributes are indexed with a B-Tree [11] and spatial data with a generalized search tree (GiST) [27]. In the edge and vertex tables a primary (clustered) index is set on the spatial attribute.

To generate stable measurements for the runtime experiments, each experiment was run 50 times, over which we compute the average runtime.

5.3 Memory Experiments

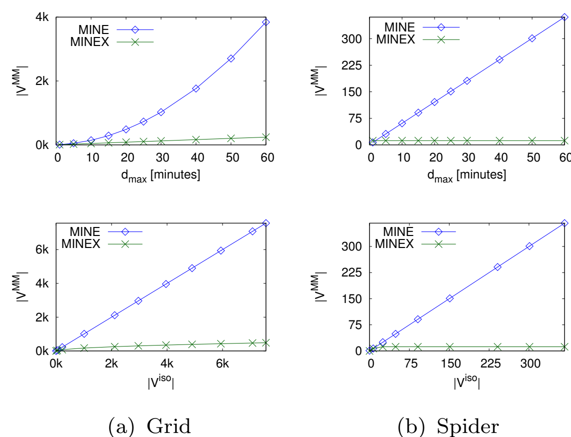
In this set of experiments, we measure first the memory consumption by varying the duration d_{max} and then the memory consumption by varying the size of the isochrone. As measurement criterion, the number of vertices in main memory is used.

5.3.1 Synthetic Data

The first experiment evaluates the memory consumption in synthetic networks, when varying d_{max} and the size of the isochrone, respectively. To better illustrate the benefit of vertex expiration, we compare only the algorithms MINE and MINEX. The space complexity of MDijkstra is independent of d_{max} and the isochrone size, since the entire network is loaded. MRNEX in a unimodal network behaves similar to MINE, and we analyze the difference in a later experiment but considering the multimodal network.

The results are shown in Figure 5.2 and confirm Lemma 5 and 6. For grid networks, MINEX's memory requirements grow linearly in d_{max} and with square root in $|V^{iso}|$. MINE's memory consumption corresponds to the isochrone size, i.e. $|V^{MM}| = |V^{iso}|$, and grows quadratically in d_{max}. In spider networks, the memory complexity is constant for MINEX and linear in d_{max} for MINE for a central query point.

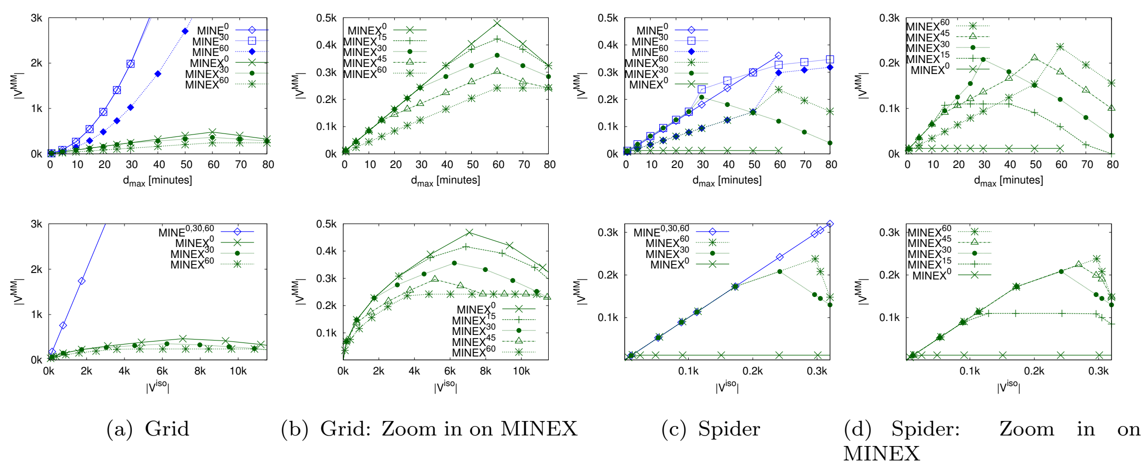
In the second memory experiment in Figure 5.3 the query point q is varied, starting from the central vertex and moving outwards. The upper index on the name of the algorithm indicates the number of rings distant from the central vertex. For example, MINEX^{60} means that q is located on an edge that is 60 rings away from the central vertex. Figure 5.3(a) shows that MINEX's memory requirements are only a small fraction of the memory requirements of MINE (which is equal to the isochrone size). Moreover, the variation of the query point has only a minor



(a) Grid (b) Spider

Figure 5.2: Memory Requirements in Synthetic Networks with Central q.

impact on the memory. In Figure 5.3(b) we zoom in on MINEX. It shows the linear dependency from d_{max} and the square root dependency from $|V^{iso}|$. As expected, the memory consumption slows down or even decreases when the isochrone reaches the network border. Consequently, for the outermost query point (i.e. MINEX^{60}) the memory consumption is least, whereas the central query point (MINEX^{0}) consumes most memory.



(a) Grid (b) Grid: Zoom in on MINEX (c) Spider (d) Spider: Zoom in on MINEX

Figure 5.3: Memory Depending on the Location of q.

Figures 5.3(c)-5.3(d) show the results for the spider network. While the memory is constant for the central query point (i.e. MINEX^{0}), for decentral query points the memory requirements initially grow. The closer the expiration front reaches the innermost vertex, the more the memory size converges to this constant

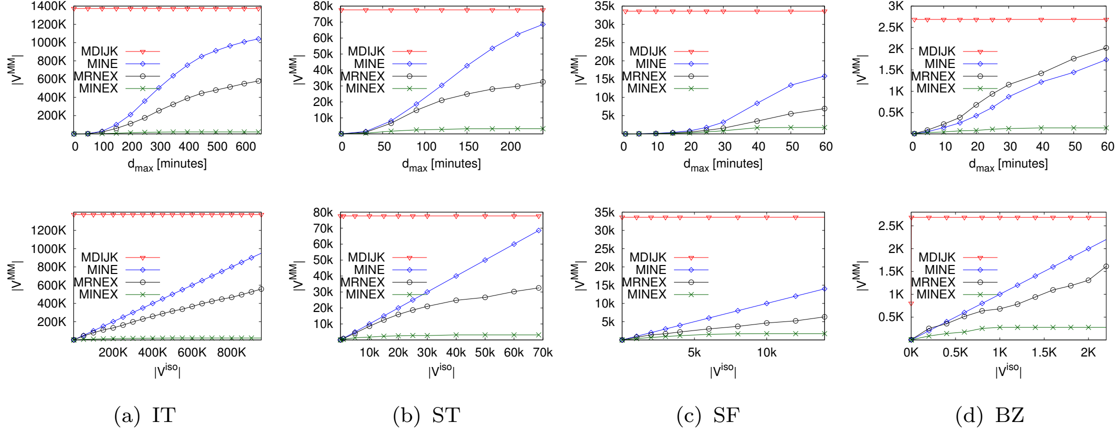
factor. For example, in the experiment MINEX^{30} the expiration front reaches after 30 minutes the central vertex with the maximal memory consumption; then the memory requirements decrease.

5.3.2 Real World Data

In the following experiments we measure the memory complexity of the algorithms MDijkstra, MINE, MINEX, and MRNEX using the four real-world datasets. As expected, MINEX's memory consumption is only a tiny fraction of the isochrone size, and it further decreases when the isochrone reaches the sparse network boundary. This behaviour is not visible in the figures, because we measure the maximal memory usage for each individual isochrone.

Memory Usage. Figure 5.4 illustrates the memory consumption for the real-world datasets. The memory requirement of MDijkstra is equal to the size of the entire network, and is independent of the actual size of the isochrone. Notice that the algorithm needs to keep in memory both the vertices and the edges. The memory of MINE grows quadratically in d_{max} until the isochrone approaches the network border, where the growing begins to slow down. Thus, the memory is independent of the network size and depends only on the isochrone size. Thanks to vertex expiration, MRNEX grows much slower than MINE. The reason for the comparably smaller difference between MINE and MRNEX for the BZ dataset in Figure 5.4(d) is that for larger d_{max} the initial range query loads a major part of the entire isochrone. MINEX shows almost constant memory requirements, independent of the size of the isochrone. Vertex expiration is much more effective in MINEX than in MRNEX, since the latter tends to load large areas at the beginning, although they are examined only later on. By loading minimal chunks (i.e. only the incoming vertices to the current vertex), MINEX keeps locality and loads only those edges which are immediately processed.

Vertex Expiration. Figure 5.5 illustrates vertex expiration using three subsequent screenshots of our prototype system *ISOGA* that are taken after 10, 20, and 30 minutes, respectively. Gray vertices are open, black vertices are closed, and white vertices are expired. Only gray and black vertices are stored in memory, while the white vertices have already been removed. In Figure 5.5(a), only a pedestrian network is used. The expansion front starts from q and grows in all directions. Hence, expired vertices are surrounded by a ring of closed vertices in main memory. In a multimodal network, as illustrated in Figure 5.5(b), every disconnected subgraph around a bus stop has its own expansion front with an internal area of expired vertices. Moreover, we have sometimes a few interior vertices with



(a) IT (b) ST (c) SF (d) BZ

Figure 5.4: Memory Requirements on Real World Data.

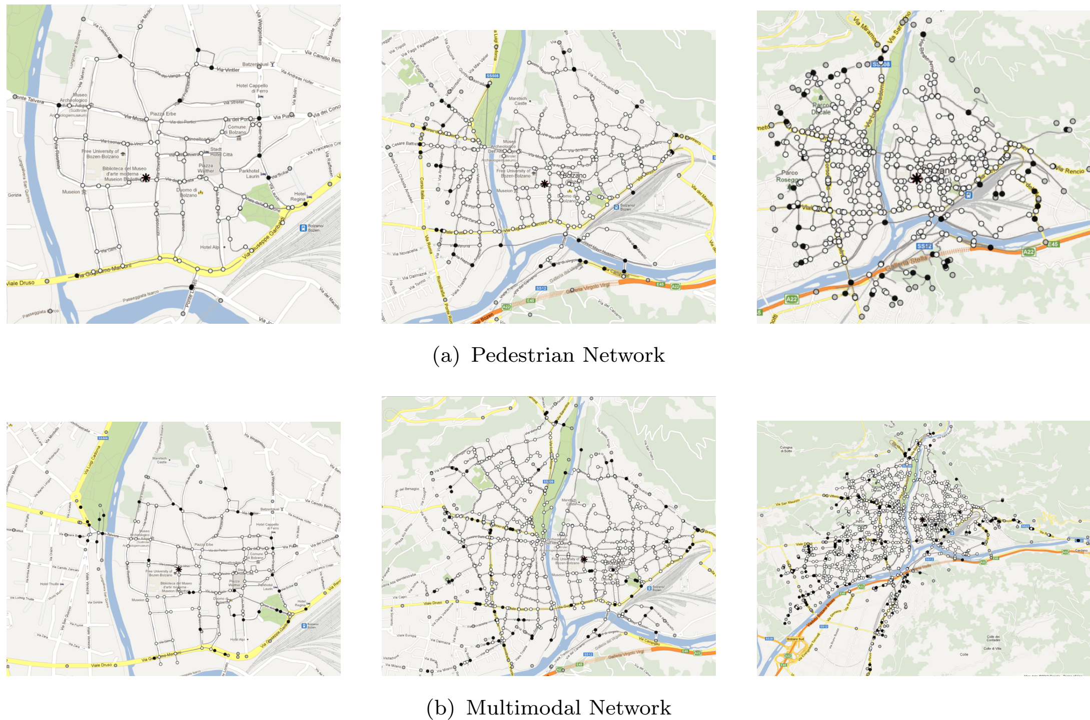
incoming edges from vertices that are far away, which hampers such vertices to expire (or they expire only very late in the expansion process). This happens, for example, if a high speed public transport systems are present.

Number of Loaded Tuples. The experiment in Figure 5.6 measures the total number of tuples (edges) that are transferred from disk (database) into main memory. MINEX and MINE load the same minimal number of tuples, which corresponds exactly to the size of the isochrone. In other words, all edges that are loaded will eventually be part of the isochrone. In MRNEX the number of loaded tuples exceeds the size of the isochrone, since the range queries load false positive tuples that will not be part of the isochrone. For the national and regional networks IT and ST with a sparse transportation network and a skewed network topology, the number of false positives is slightly higher than for the urban networks SF and BZ. In urban networks with a dense transportation network and a regular network topology, the difference is rather small.

Figure 5.7 illustrates the false positive edges that are loaded by MRNEX but are not part of the isochrone for the datasets IT and ST. The totally loaded network portions are colored in yellow, the actual isochrone is in green, and the false positive edges are in red. Both figures show that the number of false positives in MRNEX is rather small.

5.4 Runtime Experiments

In this series of experiments we analyze in detail the runtime of MDijkstra, MINEX, and MRNEX. MINE is not included since its runtime is essentially the same as



(a) Pedestrian Network

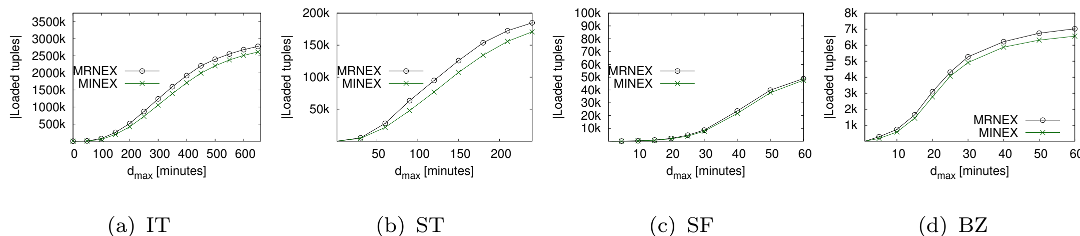
(b) Multimodal Network

Figure 5.5: Vertex Expiration — 10/20/30 min.

for MINEX.

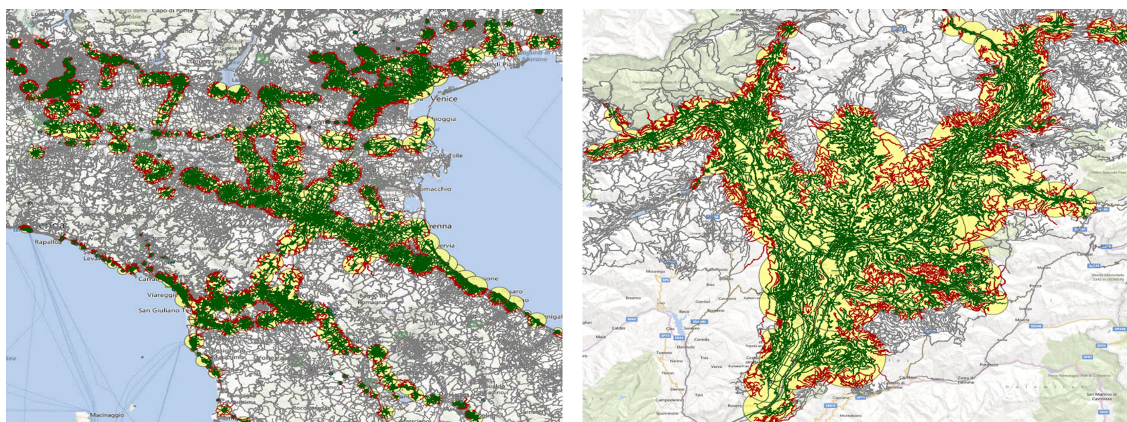
Varying the Size of the Isochrone. Figure 5.8 shows the runtime depending on the maximum duration d_{max} and the isochrone size. For small d_{max} values and small isochrones, MDijkstra has the worst performance due to the initial loading of the entire network which can be very expensive. For large d_{max} and isochrones, MDijkstra (though limited by the available memory) is more efficient since the initial loading of the network using a full table scan is faster than the incremental loading in MINEX and MRNEX. Comparing MINEX with MRNEX, we see that the latter is much faster. The break-even point between MDijkstra and the other two algorithms is smaller for city networks and higher for regional networks.

Figure 5.8(a) shows the result for the IT dataset, which is a large skewed network with few and distantly located train stations. The break-even point between MRNEX and MDijkstra is rather high at a d_{max} that generates isochrones which cover large parts of the network. MINEX is four times slower than MRNEX because of the larger number of DB accesses (one access for each vertex expansion), but still more efficient than MDijkstra for small isochrones. Figure 5.8(b) shows the runtime in the regional network ST. MDijkstra outperforms MINEX after a d_{max} of 45 minutes and MRNEX after 100 minutes. Figure 5.8(c) and 5.8(d) show



(a) IT (b) ST (c) SF (d) BZ

Figure 5.6: Number of Loaded Tuples.



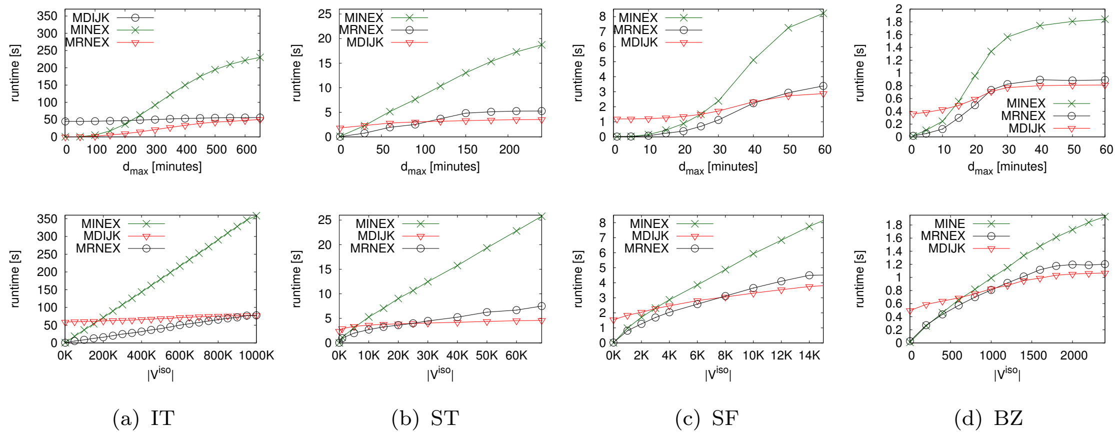
(a) IT: 300 min Isochrone. (b) ST: 150 min Isochrone.

Figure 5.7: False Positive Edges in Ranges Queries.

the runtime in urban networks with a duration of one hour. Both figures show a similar runtime behavior, with a quadratic increase in the beginning and a more moderate increase when the isochrone size approaches the network size.

Break-even Points. In Figure 5.9(a) we analyze the break-even point between the three algorithms. The break-even point occurs when the runtime of MINEX (MRNEX) exceeds the runtime of MDijkstra.

The break-even point depends mainly on the size of the network. The larger the network is, the more expensive is the initial loading in MDijkstra. In the large dataset IT, the break even point for MINEX occurs when the size of the isochrone is equal to 12% of the network size. For MRNEX, the break-even point is around 70% of the total network size. In the second largest network, ST, the break-even point for MINEX is at 7%, whereas for MRNEX it is at 31% of the total network size. In the urban network SF, MINEX is faster than MDijkstra as long as the size of the isochrone is smaller then 8% of the network size, whereas MRNEX outperforms MDijkstra with isochrones smaller than 23%. For the dataset BZ, the

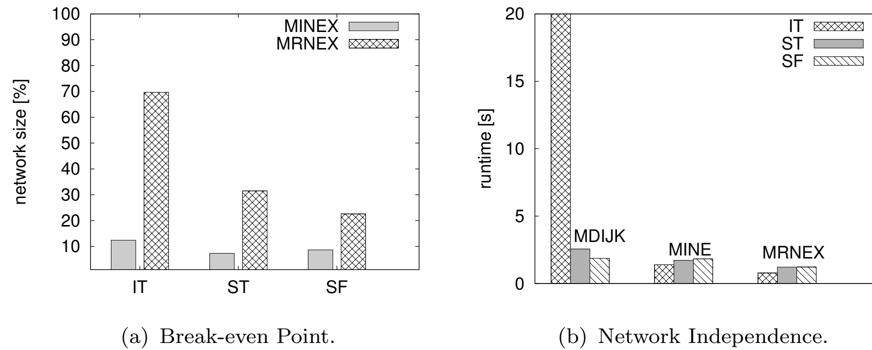


(a) IT (b) ST (c) SF (d) BZ

Figure 5.8: Runtime on Real World Data.

break-even point for MINEX and MRNEX is at 20% and 30%, respectively.

We learn from this experiment that in large datasets the break-even point moves towards the size of the isochrone. This is because the data loaded initially in MDijkstra do not fit into the cache of the database, whereas for smaller datasets they do.



(a) Break-even Point. (b) Network Independence.

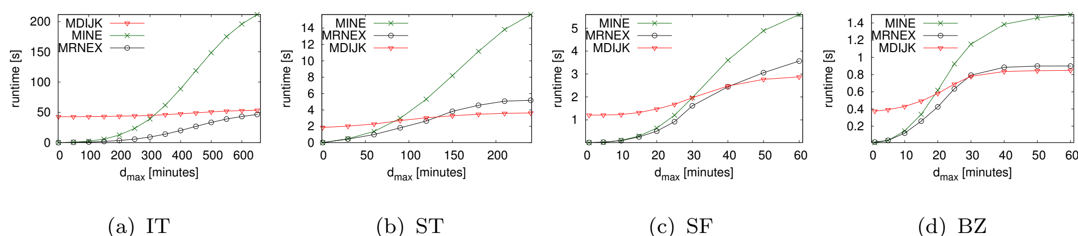
Figure 5.9: Break-even Point and Network Independence.

Figure 5.9(b) confirms that MINEX and MRNEX do not depend on the network size. We compute an isochrone of a fixed size $|V^{iso}| = 3.000$ for the different real-world data sets. The runtime of MINEX and MRNEX is almost the same for all data sets. In contrast, the runtime of MDijkstra depends directly on the network size: for the IT dataset the runtime is 54 s, in ST after 2.6 s , and in the urban networks SF and BZ after 1.9 s and 0.85 s, respectively.

5.4.1 Varying the Location of the Query Points

In this section we analyze the impact of varying locations of the query point on the runtime behavior. For each dataset, we randomly generated 20 central and 50 peripheral query points. In urban networks, a central query point is located in the central area of a city, whereas a peripheral query point is located in the peripheral areas with a minor appearance of transportation systems. In regional networks, central query points are set in major cities, whereas peripheral query points are positioned in smaller towns and villages with some distance to large cities. In the IT dataset, railway stations of the large cities belong to central points and small town railway stations to peripheral points. The runtime depending on d_{max} is measured, and we take the average over all central query points and the average over all peripheral query points.

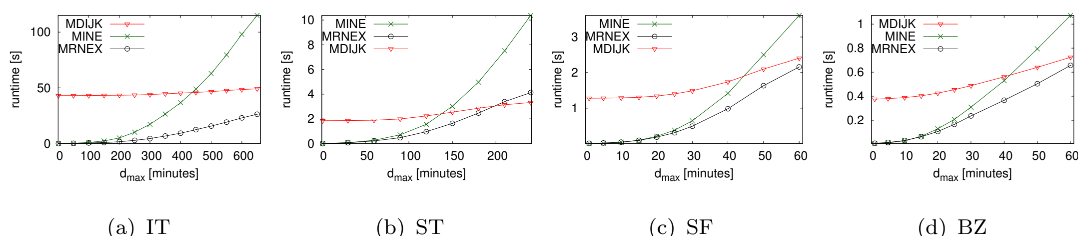
Figure 5.10 shows the results for the central query points. As expected, there is not much difference to Figure 5.8, where the default query point in the centre of the network is used.



(a) IT (b) ST (c) SF (d) BZ

Figure 5.10: Central Query Points.

In contrast, in the experiments with peripheral query points in Figure 5.11, the runtime grows slower. The reason is in the lower frequency of the transportation systems and in the sparseness of the network. MRNEX outperforms MDijkstra for all but very large isochrones. As usual, MINEX initially behaves like MRNEX, but becomes slower for larger isochrones.



(a) IT (b) ST (c) SF (d) BZ

Figure 5.11: Peripheral Query Points.

5.4.2 Varying the Arrival Time at the Query Point

In this experiment we analyze the runtime when varying the arrival time at the query point. We grouped the arrival times in three different clusters (low, medium, high), depending on the frequency of the available transport systems. Figure 5.12 shows the histograms that measure the frequency of transport systems in different time slots.

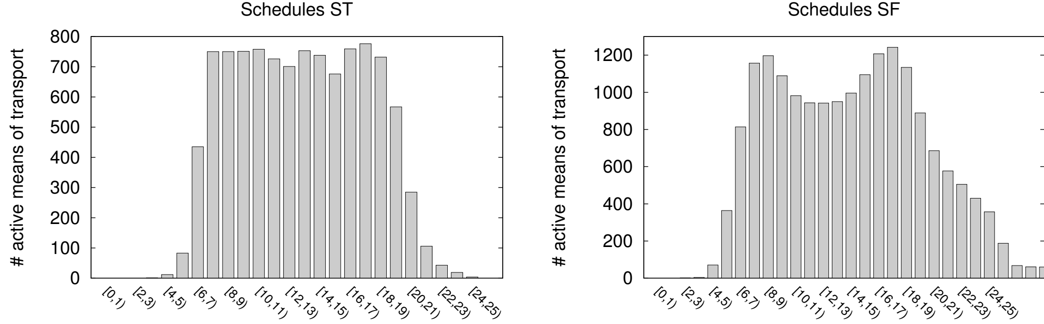


Figure 5.12: Frequency Histograms of Transportation Systems.

Table 5.2 shows the three interval clusters with the chosen time intervals. For example, the column low lists the time intervals, where the frequency of public transport systems is low.

Data	Low	Medium	High
IT	[02 : 00 – 04 : 00] am	[07 : 00 – 09 : 00] am	[09 : 30 – 11 : 00] pm
ST	[02 : 30 – 04 : 00] am	[10 : 00 – 12 : 00] pm	[08 : 00 – 10 : 00] am
SF	[02 : 00 – 03 : 00] am	[10 : 00 – 00 : 00] pm	[04 : 30 – 07 : 00] pm
BZ	[02 : 30 – 04 : 30] am	[09 : 30 – 10 : 30] pm	[08 : 00 – 10 : 00] am

Table 5.2: Clustering of Arrival Time Intervals.

The first experiment in Figure 5.13 measures the runtime when the arrival time is set to a time point in the interval with a low frequency of the public transport systems. In all datasets, MRNEX outperforms the competing algorithms. The reason for the huge difference for the IT dataset is the very small number of active public transport systems in this time range. As a consequence of this, the isochrones are very small compared to the whole network, which needs to be loaded by MDijkstra.

Figure 5.14 shows the results for arrival times that are in the medium range. MRNEX outperforms MDijkstra for the large IT dataset. For the smaller datasets, the break-even is approximately the same as in previous experiments.

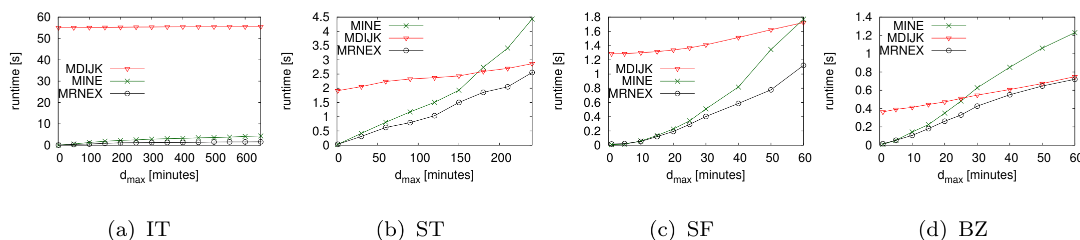


Figure 5.13: Low-frequent Time Intervals.

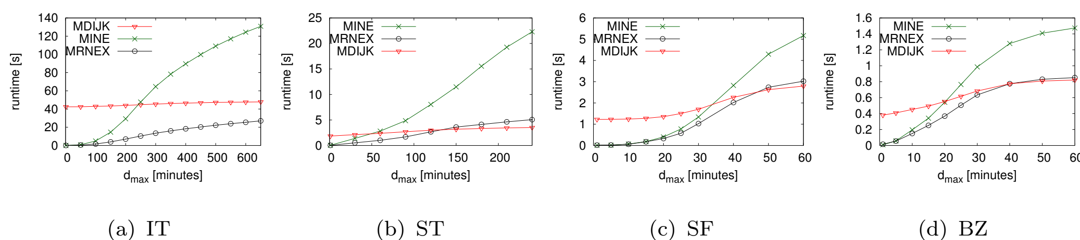


Figure 5.14: Medium-frequent Time Intervals.

Finally, Figure 5.8 shows the results when the arrival time at the query point is in the range of a high frequency of public transport systems. The runtime for all datasets is similar to previous experiments.

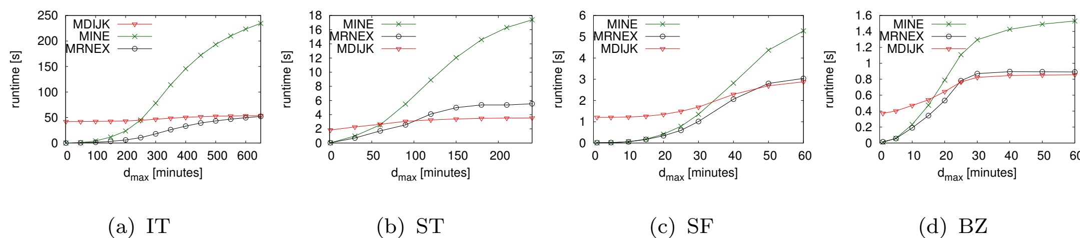


Figure 5.15: High-frequent Time Intervals.

Overall, the break-even point is lower when a dense transportation system with high frequencies is present, and it increases with a sparse public transport system and/or low frequencies.

5.5 Summary

The memory experiments in Section 5.3 measured the memory requirements and confirm that vertex expiration is effective and leads to memory requirements that

are only a tiny fraction of the isochrone size. MDijkstra keeps the entire network in memory. MRNEX's memory requirements primarily depend on the initial loaded range. MINEX depends only on a minimal set of expanded vertices that is necessary to avoid cyclic network expansions.

The runtime experiments in Section 5.4 show that for large datasets, MRNEX outperforms all other algorithms for medium as well as for large isochrones. MINEX performs well on small isochrones, but since the number of DB lookups depends on the size of the isochrone, it is not scalable in terms of runtime. Thus, MRNEX is a good trade-off between memory usage and runtime performance.

CHAPTER 6

A System for Geographic Reachability Analysis

Geospatial analysis covers various approaches to perform statistical analysis on data with a geographical or geospatial dimension and provides an important tool in many application areas, including environmental and life sciences, epidemiology, social sciences, medicine, emergency management or city planning.

This chapter presents *ISOGA* (ISOchrones for Geospatial Analysis), a system for reachability analysis enhanced with statistical analysis in spatial networks. The basis for computing geographical reachability are isochrones. *ISOGA* adopts a service-oriented, three-tier architecture and uses components that are compliant with standards from the Open Geographic Consortium (OGC). The client, a Web application, allows to compute and to visualize isochrones within a graphical user interface and to perform geospatial analysis on it. The server embeds the algorithms for the computation of isochrones presented in Chapter 4. After computing the isochrone, the system allows to join it with an arbitrary relation that contains geo-referenced objects, e.g. people, houses or hotels. The set of geo-referenced objects can be specified by the user as a general SQL query. As a result, the system shows a simple summary statistics together with a list of all objects that are located within the isochrone. The objects can be visualized on the interactive map and by clicking on an object a popup shows additional information.

6.1 System Components

The core components of the *ISOGA* system are the following:

1. The algorithms presented in Chapter 4 for the efficient computation of isochrones, where an isochrone is represented as a subgraph of the network consisting only of logical (non spatial) data.

2. An algorithm for calculating the surface of each disconnected subgraph that belongs to the isochrone. The surface representation provides the basis for joining it with object relations for the geospatial analysis.

3. A module for the statistical analysis which joins the isochrone surface with the user-specified relation of geo-referenced objects and computes the statistics.

6.1.1 Computing Isochrones

The algorithms MDijkstra, MINEX and MRNEX for the computation of isochrones have been described in detail in Chapter 4. Here, we discuss how to assign the logical representation of an isochrone with the real geometries.

The algorithms we presented produce as output an isochrone that is represented as a logical network. For visualization purposes and in order to do statistics, we need to extend the logical model with the real geometries that represent the vertices and edges of the isochrone, i.e. assign the geometric points of the vertices and the (poly)lines representing the geometry of the edges. This process is executed when the isochrone is inserted and stored in the database. When the vertices of the isochrone are inserted, the geometry is fetched from the vertex and edge tables of the network. If edges are only partially reached, we apply a clipping operation to subtract from the complete geometry of an edge only the part that is not reachable. For this we use the spatial function $ST_Line_Substring(geometry, start_fraction, end_fraction)$. The function expects as input parameters a linestring geometry and the start-offset and end-offset specified as a fraction of the edge length.

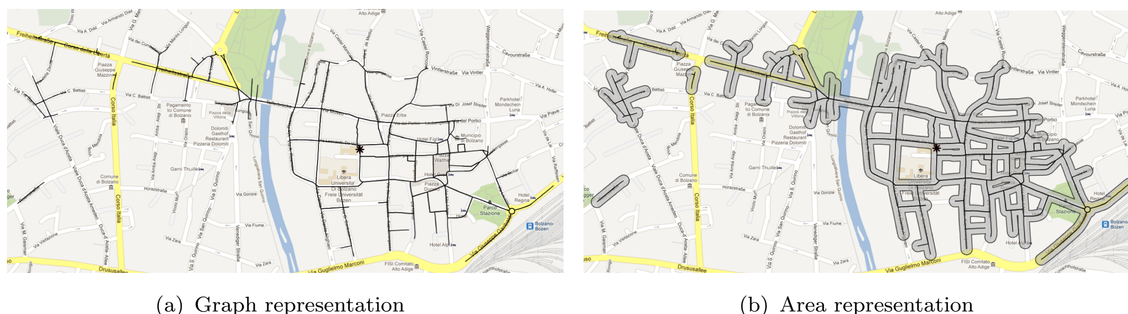
Since every insert operation of an edge or a vertex requires a lookup in the original network tables to retrieve the geometry, the writing of the output is implemented in a separate thread that uses a bulk insert strategy. A bulk-insert process inserts several SQL-insert statements in a single block, i.e. sends them from the client to the database server.

6.1.2 Creating a Surface Around Isochrones

An isochrone with associated geometries as described in the previous subsection covers all space points on the edges and vertices that are included in the isochrone. We call this a network representation. Such a network representation is not always sufficient to determine all objects that are within an isochrone. Many application are not only interested in the objects that are not *on* the edges, but also in objects that are in the immediate vicinity of edges or vertices, e.g. houses that are along the streets but not on the streets. Thus, the spatial areas that are covered by an isochrone need to be considered. We call this an area representation of isochrones.

Marciuska and Gamper [42] propose two algorithms to construct isochrone areas, but they are limited to work only for isochrones in a pedestrian network. The *Link-Based Approach* (*LBA*) simply computes a buffer around each edge in the isochrone, while the *Surface-Based Approach* approach (*SBA*) computes a minimum bounding polygon around the outermost edges in the isochrone. We improved these two approaches to become more efficient and applicable for general isochrones.

In *LBA* [42], the creation of a separate buffer around each edge in the isochrone produces a large number of intersecting areas, one for each edge. Thus, joining an isochrone with an object relation required to join each of these buffers with the relation, which produces a large overhead, and due to the overlapping, a single object may fall into several buffers. To remedy this problem, we aggregated all overlapping buffers into a single geometry. This modification improved the runtime for the join even in the small dataset BZ by 20%. For larger isochrones and a larger size of spatial objects it will be considerably more.



(a) Graph representation (b) Area representation

Figure 6.1: Isochrone Area Generation Using *LBA* Approach.

Figure 6.1(b) shows the screenshot of an isochrone area that is generated by the *LBA* approach from the isochrone in Figure 6.1(a). The isochrone consists of five disconnected subgraphs.

Algorithm 6 shows SBA^*, which extends *SBA* [42] for isochrones that are

composed of disconnected subgraphs, which is typically the case if public transport systems are present in addition to the pedestrian network. The input parameters to the algorithm are a set of edges E^{iso} that represents an isochrone network and a parameter $size$ that represents the margin of the buffer that is created around the outermost edges. SBA^* iterates over the set E^{iso}, and in each iteration it computes an area around a single subgraph of the isochrone. It determines the leftmost edge in E^{iso} and calls the function DFS which traverses the outermost edges of the subgraph in a recursive way. The parameter P returns the ordered list of points that represent the geometry of the outermost edges. Then, P is transformed into a polygon and a buffer with a margin of size $size$ is created around the polygon. Finally, all edges that are within the isochrone area are removed from E^{iso}.

Once the leftmost edge of a subgraph is found, function DFS recursively computes the outermost edges of that subgraph and collects their geometry in an ordered set of points P. DFS has as input the leftmost edge $root$, the current visited edge e, the isochrone edges E^{iso}, the ordered set of points P, and the recursion level l. DFS examines in a depth-first-search the subgraph until it either returns to the root edge or when no other edges are found. If $e = root$ and $l > 0$, the edge traversal is returned to the root edge. The geometry of the current edge is added to P (line 2 after inverting the order of the points) and the recursion terminates (line 3). If the root edge e is visited the first time ($l = 0$), we first add its geometry to P. Then, we consider all incoming edges to v, ordered by the angle to e, by recursively calling DFS with that edge and a recursion level that is incremented by one. If $l > 0$, the edges are processed in a similar way (line 11). The only difference is that we have to consider the case that an edge is only partially reachable; $o(e)$ is the reachable segment from the start vertex of e and $o(e^{-1})$ is the reachable edge segment from the end vertex of e. If $o(e) + o(e^{-1}) \leq \lambda(e)$, the two edge segments cover the entire edge, hence the recursive traversal of the edges continues (line 15). If this is not the case, only the inverted points current edge segment are added (line 18), the recursion stops (line 19) by returning false.

Algorithm 6: $SBA^*(E^{iso}, size)$

```
  input: E^iso, size
1 while E^iso ≠ ∅ do
2 |   P ← ∅;
3 |   e = (u, v) ← leftmost edge in E^iso;
4 |   DFS(e, e, E^iso, P, 0);
5 |   g ← ST_Buffer(ST_MakePolygon(P), size) ;
6 |   E^iso ← E^iso \ ST_Within(E^iso.geo, g);
7 |_  Output ST_Multi(g);
```

If `DFS` returns true, we stop the recursion; otherwise, we add the geometry of e to P and consider the next incoming edge to v. Once the root edge is reached

Function DFS($root, e, E^{iso}, P, l$)

```
1  if e = root ∧ l > 0 then
2  |   P ← P ∪ geom(e)^{-1} ;
3  |_  return true ;
4  (u, v) ← e;
5  if l = 0 then
6  |   P ← P ∪ geom(e);
7  |   foreach (u', v) ∈ E^{iso} sorted by angle α((u, v), (u', v)) do
8  |   |   if DFS(root, (u, v), E^{iso}, P, l + 1) then  return true;
9  |   |_  else  P ← P ∪ geom(e);
10 |   return false;
11 else
12 |   if o(e) + o(e^{-1}) ≤ λ(e) then
13 |   |   P ← P ∪ geom(e)^{-1} ∪ geom(e^{-1}));
14 |   |   foreach (t, u) ∈ E^{iso} sorted by angle α((u, v), (t, u)) do
15 |   |   |   if DFS(root, (t, u), E^{iso}, P, l + 1) then  return true ;
16 |   |   |_  else  P ← P ∪ geom(e) ;
17 |   else
18 |   |_  P ← P ∪ geom(e)^{-1} ;
19 |_  return false ;
```

the recursion terminates and by returning true `DFS` prevents that any other edge is examined. Next, in the main algorithm the points in P are transformed into a polygon, around which a buffer with a margin of size *size* is created (line 5). A polygon is valid, if it shares in common only the first and the last point. Since during the backtracking `DFS` (line 18) adds same points of the geometry in reversed order, the final constructed polygon becomes invalid. The SQL function $ST_MakeValid(geometry)$ remedies this misbehaviour by transforming an invalid polygon geometry into a valid geometry collection, on which a buffer is created.

Next, we illustrate the algorithm SBA^* using a small example and compute the area step-by-step.

Example 6.1.1. The leftmost edge (v_0, v_1) in Figure 6.2 is passed as root edge to the function `DFS`. Since the level of recursion is zero, the geometry is added to P and all incoming edges to the end vertex v_1 are determined and sorted by the angle relative to the current visited edge. This results the order set $\{((v_2, v_1), \alpha_{21}),((v_8, v_1), \alpha_{81}),\ ((v_0, v_1), \alpha_{01})\}$. Note the angle to the edge itself is set to 360°. As next edge DFS is called with (v_2, v_1).

Since we are examining edge (v_2, v_1) in the first recursive call ($l > 0$) we have to determine the incoming edge relating to the start vertex v_2. This returns the set $\{((v_3, v_2), \alpha_{32}),((v_7, v_2), \alpha_{72}),\ ((v_1, v_2), \alpha_{12})\}$. And we proceed the recursion with edge (v_3, v_2) until examining edge (v_4, v_3) as shown in Figure 6.3(a). At this point, when entering into the next level of recursion with edge (v_9, v_4) and traversing a partial edge, the inverted geometry is added to P and the recursion is interrupted

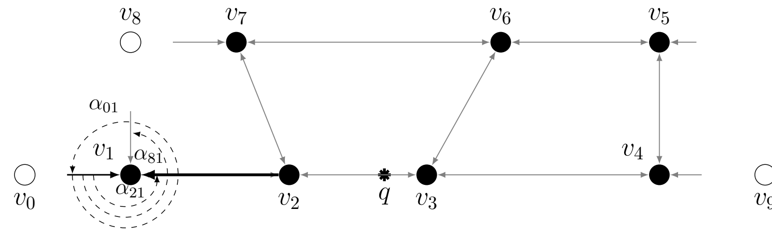
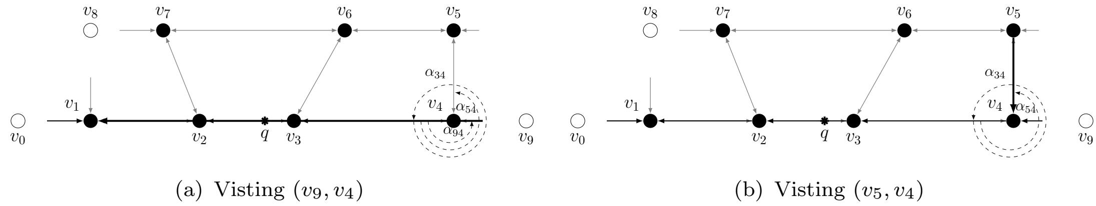


Figure 6.2: Intialization: visit edge (v_0, v_1).

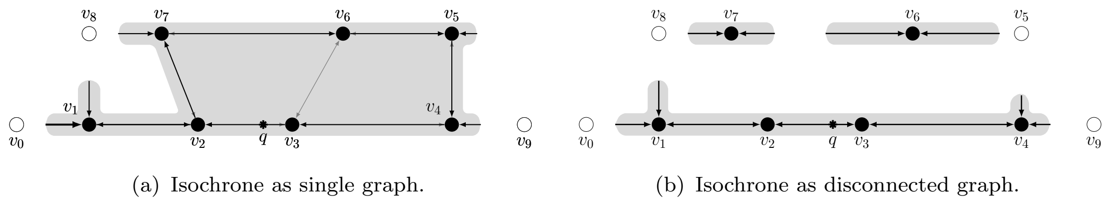
by returning false. Returning false indicates that the root edge was not found. Next, after appending the geometry or edge (v_9, v_4), the recursion proceeds with (v_5, v_4) as illustrated in Figure 6.3(b).



(a) Visting (v_9, v_4) (b) Visting (v_5, v_4)

Figure 6.3: Illustration of SBA^* algorithm (1).

The final surface is shown in Figure 6.4(a) as the light-gray area. Note that the edges $(v_3, v_6), (v_6, v_3)$ do not belong to border edges and were consequentially not examined in DFS. Finally all edges that are enclosed by the buffer are extracted from E^{iso}.

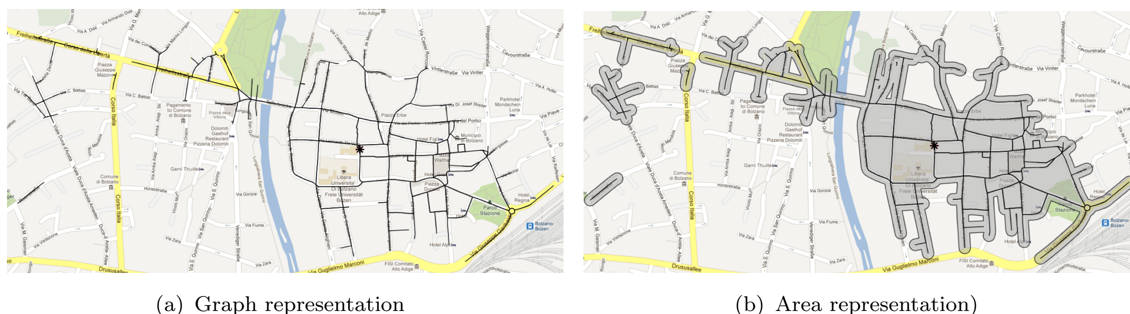


(a) Isochrone as single graph. (b) Isochrone as disconnected graph.

Figure 6.4: Illustration of SBA^* algorithm (2).

Typically in multimodal networks an isochrone consists of several disconnected graphs. This implies that after the first iteration of the while loop, E^{iso} is not empty and DFS is launched again with the leftmost edge of the next subgraph. In Figure 6.4(b), that represents a different isochrone consisting of three subgraphs, after creating the polygon with root edge (v_0, v_1), DFS is called with edge (v_8, v_7) and later with (v_7, v_6). DFS terminates when no more edges reside in E^{iso}.

Since the algorithm performs a network examination to identify the outermost edges, the requirement is that the isochrone represents a planar graph. That means an edge can cross another edge only at the start or at the end vertex.



(a) Graph representation (b) Area representation)

Figure 6.5: Isochrone Area Generation Using SBA^* Approach.

Figure 6.5(b) shows the screenshot of an isochrone area that is generated by the SBA^* approach from an isochrone consisting of multiple subgraphs.

6.1.3 Computing Statistics with Isochrones

The area representation of an isochrone allows us to perform spatial operations such as joining the isochrone with other geo-referenced objects and to compute statistics about the reachability. Figure 6.6 illustrates the workflow of the statistical component. The user creates an arbitrary SQL expression Q, where one

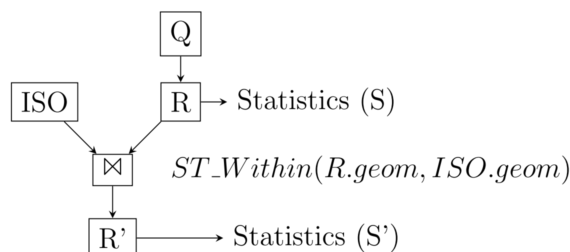


Figure 6.6: Query Tree with Workflow.

can specify the objects of interest for the analysis (e.g. houses, people, museums) and a predicate that defines the spatial relationship (intersection, disjoint, ...) between these objects and the isochrone. In order to join the selected relation with the isochrone, one of the projected attributes must have a spatial property. That can be a geometry of type point, linestring, or polygon. The query expression Q is parsed, and the spatial attribute is extracted. Then, Q is passed to

the DB and executed. This returns an intermediate result R, from which the total number of tuples and/or an aggregated attribute value (e.g. SUM) are computed. Next, the intermediate relation R is joined with the isochrone area ISO, i.e. $R' = \pi_{R.*}(\sigma_{ST_Within(R.geom,ISO.geom)=true}(R \bowtie ISO))$. Relation R' contains all tuples with a geometry that is inside the area of the isochrone. Again, on R' basic statistics are computed. As a final result, the user gets relation R' and a summary statistics about the reachability (see Sec. 6.4 for some examples).

6.2 Architecture

From a technical perspective, *ISOGA* adopts a service-oriented, three-tier architecture composed of a presentation tier, logical tier, and data tier, and it uses standardized OGC services for exchanging spatial data between client and server (see Figure 6.7). The system can be accessed at `www.isochrones.inf.unibz.it/isoga`.

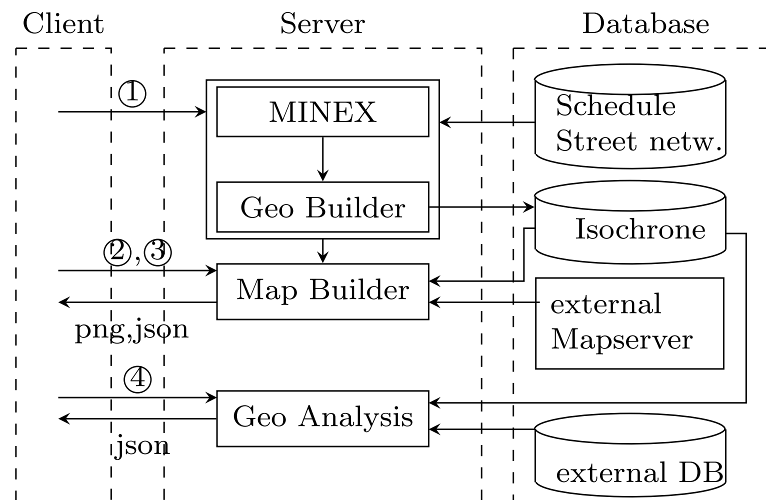


Figure 6.7: Architecture.

6.2.1 Presentation Tier

The presentation tier is a WebGIS client, implemented in JSP and JavaScript. It uses *Comet*[1] as Web application model for asynchronous data sending and for managing long polling requests. *Openlayers*[2] is used as Web mapping framework and *GeoExt*[3] as framework for building interactive WebGIS applications. A long

[1] `www.cometd.org`
[2] `www.openlayers.org`
[3] `www.geoext.org`

polling request is an Internet-based communication that allows the emulation of an information push from a server to a client.

The main tasks of the client are the interaction with the map, the input of the query parameters, and the visualization of the results. The client communicates with the server over the HTTP protocol using the following standardized OGC[4] services:

- Web Map Service (*WMS*) for serving geo-referenced map images;
- Web Feature Service (*WFS*) for requesting geographical features.

The client submits asynchronously four different types of HTTP requests. An *isochrone request* ① invokes MINEX for the computation of an isochrone with the user defined parameters. Figure 6.8 exemplifies such a request.

```
http://../isoServlet?request=GetIsochrone
&arrivalTime='2010-09-01T21:20'&poi='1263595,5860641'
&speed=2&duration=20&networks='P,B'
```

Figure 6.8: Isochrone Request.

A *map request* ② (see Figure 6.9) retrieves the isochrone in form of a binary image format and includes it as a separate layer in the map. Vertices, edges and areas are added as three different layers in the map. Similar, the base layer images (e.g.

```
http://../isoServlet?request=GetMap
&format=image:png&layers=edges&width=800&height=600
&bbox=1263657,5860274,1263963,5860580'
```

Figure 6.9: Map(WMS) Request.

the street network or orthophoto) that come from different external geographic map providers (for instance, Google Maps, OpenStreetMaps or Microsoft Bing) are fetched via WMS requests. The request is triggered during the initialization of the map, or after that the algorithms have produced their output or whenever there is an interaction with the map (zoom, pan, identify).

A *feature request* ③ retrieves detailed information about a selected feature and returns as response the information in textual format. An example is shown in Figure 6.10: The parameters specify the selected bounding box that represents the spatial range in which the feature is located, the name of the layer, and the format of the response. For instance, in Figure 6.14 the returned feature is visualized in

[4] http://www.opengeospatial.org/

```
http://../isoServlet?request=GetFeature
    &format=text:json&typeName=vertices
    &bbox=1263657,5860274,1263659,5860275
```

Figure 6.10: Feature(WFS) Request.

a pop up and represents a vertex that is annotated with information on how to reach the query point (e.g. distance, walking time, bus line, and departure time). The request is triggered by clicking on an object on the map.

6.2.2 Logical Tier

The logical tier (i.e. the server in Figure 6.7) accepts requests via a Java Servlet. During an isochrone request first the coordinate of the query point is mapped to the closest continuous edge and its offset and then invokes the algorithms presented in Section 6.1.

In order to enable the client to access the isochrone via WMS and WFS, the three output tables, i.e. (iso)vertices, (iso)edges and (iso)areas are registered as vector layers in the map builder module. As map builder we use the rendering engine *Geoserver*[5] which provides standardized OGC services to access to the spatial information of these layers and acts as a rendering engine. For a WMS request, Geoserver reads the spatial data from the DB and creates an image that is sent back to the client. For a WFS request, the information is retrieved from the DB and sent to the client as a feature in textual format. The map builder serves also as proxy for providing base layers from external maps servers.

The network representation of the isochrone is passed to the *GeoBuilder* module, which performs two tasks. First, the isochrone is annotated with geometry information and stored in vector format in a spatial relation in the DB. Second, the network representation of the isochrone is transformed into a spatial area (polygon).

The third main task of the logical tier is to perform the geospatial analysis as described in the previous section. Once an isochrone is computed, the user can specify an arbitrary SQL expression that is sent to the server via a *geoAnalysis request* ④. The query is processed as described in Section 6.1.3.

Figure 6.11 illustrates an analysis request that returns a relation with all buildings in the specified price range, that reside in the previous computed isochrone.

The final result is converted from a relational format into a Java Script Object Notation (JSON) format and sent back to the client.

[5] www.geoserver.org

```
http://../isoServlet?request=GeoAnalysis
  &format=text:json&typeName=area
  &query='select id,avg_rent,geom from buildings
          where avg_rent between 600 and 800'
```

Figure 6.11: Analysis Request.

6.2.3 Data Tier and Data Model

The data tier uses a relational DBMS with a spatial extension to perform spatial operations, such as edge clipping if an edge is only partially reached, mapping the query point to the closest edge, area buffering, spatial intersection, or other spatial operations.

The multimodal network is stored in five different tables. The vertex table contains all crossroads of a street network and all stop stations belonging to public transport systems. It consists of a unique identifier and a spatial attribute of type point. This geometry is used for rendering the vertices as a geographical point in the map. In addition this attribute is required in the range queries of the MRNEX algorithm. Table 6.1 shows an excerpt of the vertex table.

id	*geometry*
5025	POINT(680386 5152168)
5027	POINT(680455 5152044)
2000542	POINT(680393 5152167)
⋮	⋮

Table 6.1: Vertex Table.

The edge table contains a unique identifier, the id of the start vertex, the id of the end vertex, the length of the edge (e.g. in terms of meters), a transport system to which the edge is associated, the in-degree of the start vertex, the out-degree of the end vertex and the geometry of type linestring. The in- and out-degree are used in the algorithms MINEX and MRNEX. Table 6.2 shows an excerpt of the edge table. Note that discrete edges are not annotated with a geometry and the edge length is undefined.

id	**start**	**end**	**length**	**t_sys**	**t_mode**	**s_indeg**	**e_outdeg**	**geometry**
270	5027	5025	null	28	'*dsdt*'	12	14	null
270	5027	5025	null	32	'*dsdt*'	12	14	null
2006436	5025	2000542	8	0	'*csct*'	12	4	LINESTRING(680386 5152168,680393 5152167)
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.2: Edge Table.

The table of the schedules represents the interval representation described in Chapter 4. It consists of a trip identifier, an identifier of the transport system, a start vertex, departure time at the start vertex, an end vertex, arrival time at the end vertex, and an id of the day when the transport system is active. Departure and arrival time are expressed in seconds after midnight. An example relation is illustrated in Table 6.3.

trip_id	**t_sys**	**start**	**departure**	**end**	**arrival**	**active_day**
4559	28	5027	17820	5025	17880	11
4563	28	5027	20220	5025	20280	11
4572	28	5027	23220	5025	23280	11
...	...	...	...	...	...	...
475	32	5027	21420	5025	21480	24
482	32	5027	24720	5025	24780	24
485	32	5027	25620	5025	25680	24
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.3: Schedule Table.

The daymarker table specifies codes for groups of days on which a trip is active. For instance, the first tuple in Table 6.4 specifies the code for trips of a mean of transport that is available on weekdays only, while the second tuple specifies the code for trips that run only on the weekend.

day_id	**mo**	**tue**	**wed**	**thu**	**fri**	**sa**	**su**
11	1	1	1	1	1	0	0
24	0	0	0	0	0	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.4: Daymarker Table.

Finally, the transport system table illustrated in Table 6.5 contains all types of means of transport that are available in the multimodal network.

t_sys	**short_name**	**long_name**	**agency**
0	'Ped'	'Pedestrian Network BZ'	'Municipality BZ'
28	'3 BZ'	'Bus Route Nr 3 Area BZ'	'SASA'
34	'5 BZ'	'Bus Route Nr 5 Area BZ'	'SASA'
⋮	⋮	⋮	⋮

Table 6.5: Transportation System Table.

The *ISOGA* system works with PostGIS 2.0 as well as with Oracle Spatial 11g. Only OGC-standardized spatial operators (OGC/SQL-MM [60]) are used on the DB level. This implies an easy migration to any other OGC-compliant spatial DB, such as SpatiaLite[6]. Figure 6.6 lists the spatial SQL operations used in the system *ISOGA*.

[6] `www.gaia-gis.it/spatialite/`

Functionality	**Operator**	**Description**
Mapping	$ST_Line_Locate_Point$ $(E.geom, Q.geom)$	Maps the coordinate of q to its closest edge in E
Clipping	$ST_Line_Substring$ $(geom, from, to)$	Creates a line segment from the origin geometry determined by the two fraction offsets (from,to)
Buffering	ST_Buffer $(geom, size)$	Constructs around a geometry (point,line,polygon) a buffer of type polygon with a specific size
Casting	$ST_Multi(geom)$	Casts a single geometry (point,line,polygon) to a collection of geometry from the same type. Used for creating the isochrone area formed by the outermost edges.
Conversion	$ST_MakeValid(geom)$	Converts an invalid polygon into a valid one
Inclusion	ST_Within $(R.geom, S.geom)$	Spatial predicate returns all tuples from relation R whose geometry resides within the geometry of S

Table 6.6: Spatial SQL Operations.

6.3 Conceptual Design for Mobile Devices

The current version of *ISOGA* is deployed as a client-server Web application. In principle, the client would run on a mobile device within a Web browser, but the graphical components are not designed and optimized for such small screens. Moreover, running the client on such devices with a small bandwidth may cause long response times, and when no network connection is available the client will not work. Also, high roaming costs might occur if the user accesses the service from a foreign country. For all these reasons, *ISOGA* has to be reengineered to work in offline mode and as a native mobile application. In this section, we propose an adaption of the *ISOGA* architecture that is suitable for mobile applications.

In typical client-server applications the server has a large amount of memory and a fast connection with the client, so the data can often completely be loaded in main memory and the major workload is performed by the server. Contrary, in mobile phones memory is still a critical issue. For instance, a state of the art mobile smart phone, such as the Android device *Sony Ericsson XPERIA arcS [59]*, has a limited amount of RAM ($512MB$), where at least half of the memory is reserved for the operating system or other internal services. However, these devices have a large capacity in secondary memory. For example, the model XPeriaARCS has a

phone storage of $1GB$ and an expansion slot for a microSD card up to $32GB$ [59].

Figure 6.12 shows a preliminary design of an architecture for mobile devices. The system is divided in two blocks: the mobile application and the download server. The first block is an application that works independent from the server. A communication between these blocks is only established, when spatial data is synchronized (e.g. a new schedule is published) or when new data is added.

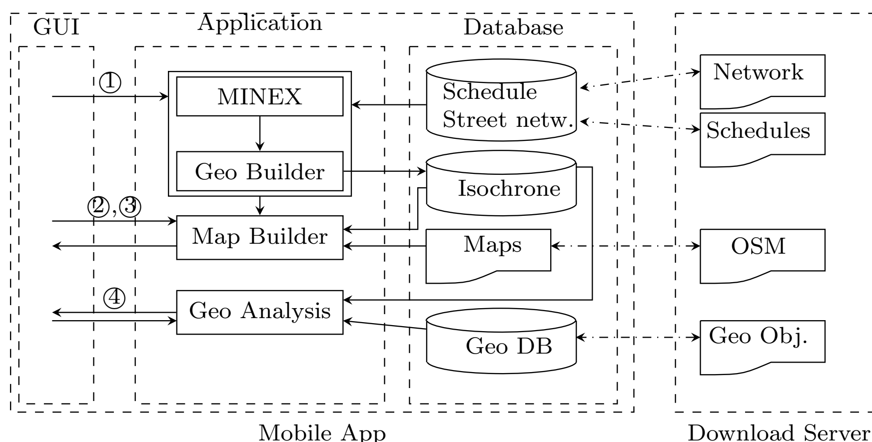


Figure 6.12: Design of a Mobile Architecture.

The presentation layer is a GUI developed with the Android User Interface Widgets. It communicates with the application layer over service methods to invoke the requests that we described in the previous section. The computation of the isochrone will be done by the MRNEX algorithm, since it offers the best trade-off between memory usage and runtime performance. The code in GeoBuilder remains untouched, since the DB calls are identical for a database on the mobile device. The MapBuilder part will be modified by replacing Geoserver with Tilemill [7]. Tilemill uses Mapnik as a renderer and supports various spatial formats, such as shapefiles, rasters, etc. Since Tilemill is an open source product, it can be extended to use the light-weighted DBMS SQLite with its spatial extension SpatiaLite. SpatiaLite provides a limited set of spatial data types and operators, but all the spatial operators used in *ISOGA* listed in Table 6.6 are implemented. Therefore the data tier loads the used tables from the secondary memory (MMC card), where space is not a problem.

[7] http://mapbox.com/tilemill/

6.4 Application Scenarios

We present two application scenarios that emerged from a collaboration with the Municipality of Bolzano-Bozen and one example that shows network expiration to illustrate the low memory requirements of MINEX. All of these application scenarios are online available under the following URL: `www.isochrones.inf.unibz.it`. The client was tested with the Web browsers Mozilla, Chrome, and Safari.

Flat Search. In this use case we want to seek cheap apartments that are close to the working place. As illustrated in Figure 6.13, the user specifies a single query point by clicking on the map that represents her/his working place and the other input parameters shown on the right-hand side of the figure. As a result, the isochrone together with the vertices that are visited with public means of transport, the edges, and the isochrone area are added as overlays to the map.

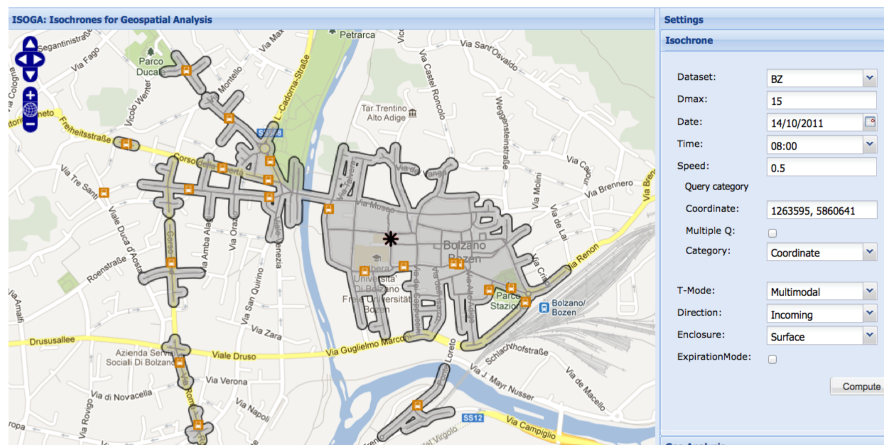


Figure 6.13: Using Isochrones in a Flat Search Scenario.

In the second step, the user specified SQL query shown in the upper right corner in Figure 6.14 retrieves a subrelation with all flats in the specified price range that is joined with the isochrone using the spatial relationship *within*. All available flats that are located inside the isochrone are visualized as circles on the map. On the right-hand side some statistics about the flats that are located inside the isochrone are shown, and the table in the lower right corner shows the corresponding prices. By clicking on the bus icon, the path and the traveling time to the working place is shown.

Reachability of Schools. In this use case we want to analyze how well the primary schools in the city Bozen-Bolzano are reachable in less than 15 minutes

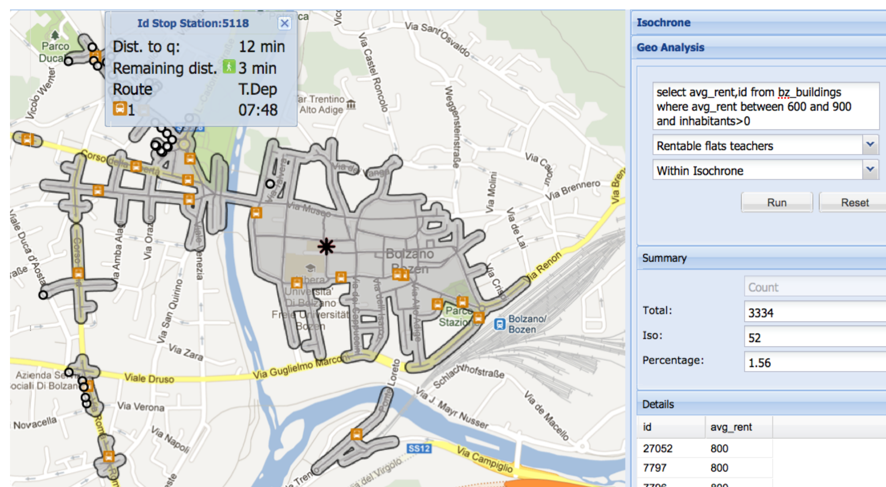


Figure 6.14: Joining Flats and Isochrones.

by using public transport systems in addition to walking. For this we issue an isochrone query with multiple query points as shown in Figure 6.15. More specifically, we select the category *primary school* which selects and displays the locations of all urban primary school buildings together with the beginning of the school, e.g. 08:00 am.

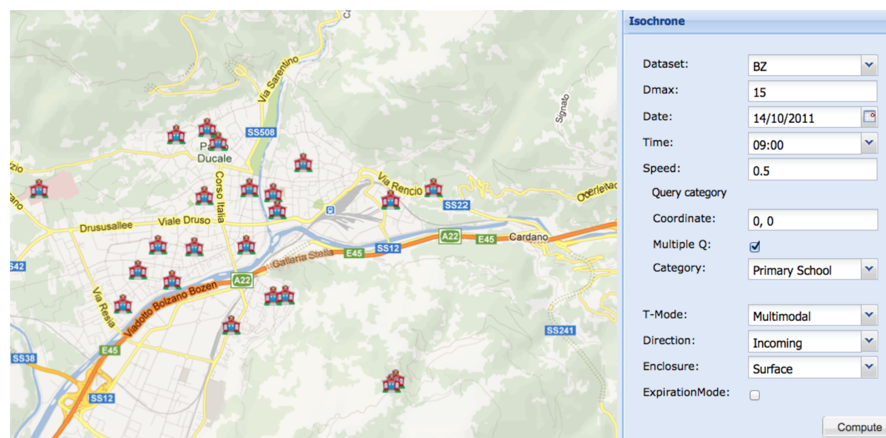


Figure 6.15: Reachability of Primary Schools.

Next, we specify an SQL query that retrieves from the inhabitants database all houses in which school kids with an age between 6 and 11 years live. This relation is joined with the isochrone. By specifying in the projection the *SUM* aggregation function the statistics shows in the lower right side of Figure 6.16 the number and the percentage of school kids that reach the closest school in less than 15 minutes.

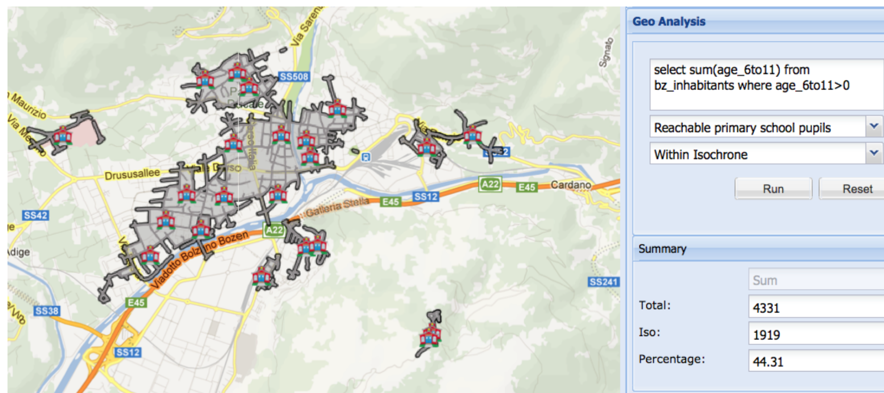


Figure 6.16: Percentage of Schools Kids in a 15 min. Isochrone.

If we change in the input parameters the time to 01:00 pm and the direction from the query points outwards, we get the number of kids who reach their home after school ending in less than 15 minutes.

6.4.1 Vertex Expiration.

The last scenario illustrates network expiration of MINEX (see Figure 6.17). The user can select different datasets. Currently, the cities of Bolzano-Bozen, and San Francisco as well as the regional networks of South Tyrol and Italy are available. All datasets have different network topologies and different types of transport systems. After computing an isochrone, the user can open the Layers panel and activate the visualization of the status of the vertices during the computation of the isochrone. Black circles represent the vertices that are kept in memory to avoid cyclic expansions (the expansion frontier). Gray circles represent vertices in memory that were not expanded). White circles represent expired vertices that have already been removed from memory in order to minimize the memory requirements of the algorithm.

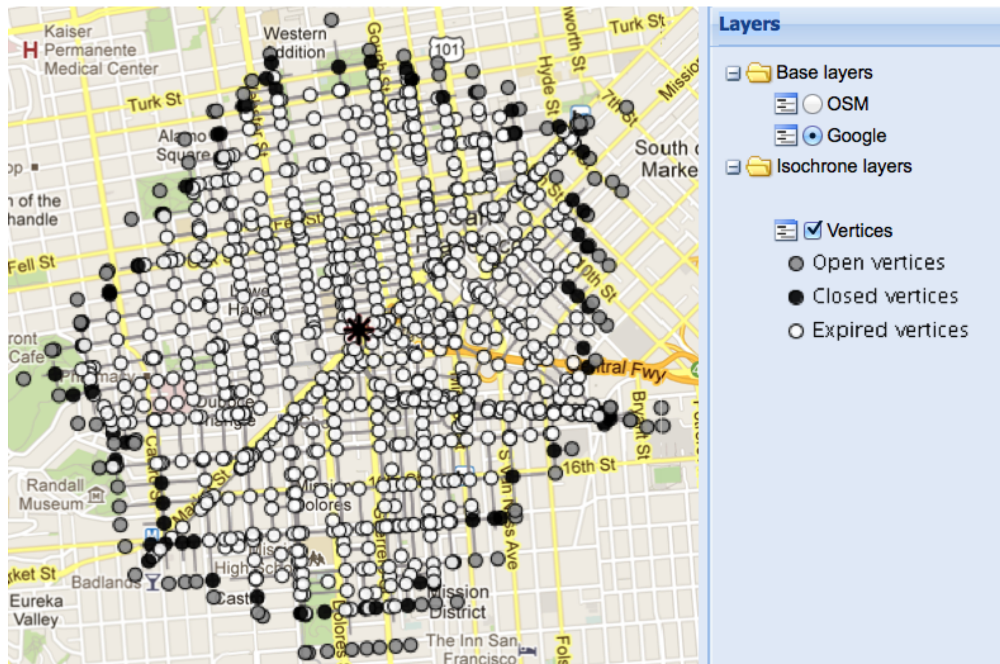


Figure 6.17: Vertex Expiration in MINEX.

6.5 Summary

In this chapter, we presented *ISOGA*, a prototype system for computing, querying, and visualizing isochrones in combination with statistical analysis. The system allows to join an isochrone with an arbitrary relation that contains geo-referenced objects, e.g. people, houses, or hotels. The set of geo-referenced objects can be specified by the user as a general SQL query. As result of a query, the system shows a simple summary statistics together with a list of all objects that are located within the isochrone. The system is subdivided in three major components: the first component is responsible for an efficient computation of the isochrone that forms a disconnected subgraph of the origin network. The second component cares about the correct surfacing of the isochrone in such a way to make it queryable with other spatial objects. The third component is responsible for joining an isochrone with an arbitrary, user defined spatial relation with the aim to perform geospatial reachability analysis. From a technical perspective, *ISOGA* adopts a service-oriented three-tier architecture and uses standardized OGC services for exchanging spatial data.

CHAPTER 7

Conclusions

7.1 Summary

In this thesis we introduced, defined, and provided algorithmic solutions for the computation of isochrones in multimodal spatial networks. We further investigated the use of isochrones for geospatial analysis and developed a prototype system that shows the usefulness of such analysis in different real-world application scenarios.

We begun with the definition of multimodal spatial networks that can be continuous or discrete along the time and space dimensions, respectively. We introduced a general time-dependent cost function to compute the time-dependent transfer time on network edges. We then defined an isochrone as a possibly disconnected subgraph that covers all space points in the network from where a query point is reachable within a given time span and by a given arrival time.

We developed three different algorithms for the computation of isochrones, where each of them is suitable in different computational environments. The first algorithm MDijkstra, though it is very efficient, has a high memory cost and shall be used if a powerful server with a large memory is available. The second algorithm MINEX has a very low memory footprint and is suitable in a computational environment where memory is more important than runtime. The third algorithm MRNEX is a trade-off between memory and runtime and is the best choice for native mobile applications.

An in-depth empirical evaluation confirms the scalability of the proposed algorithms. While MDijkstra is fast once the entire network is loaded in memory, vertex expiration that is implemented in MINEX and MRNEX is an effective way

to minimize the memory requirements. Indeed, the memory requirements of the latter two algorithms are only a tiny fraction of the isochrone size. That is, only a minimal set of expanded vertices is kept in memory that is necessary to avoid cyclic network expansions. The runtime experiments show that for large datasets, MRNEX, which loads the network in small chunks outperforms all other algorithms for medium and large isochrones. MINEX performs well on small isochrones, but since the number of DB lookups depends on V^{iso}, it is not scalable in terms of runtime. Overall, MRNEX is a good trade-off between memory and runtime.

We implemented a Web-based *prototype system* which combines the computation of isochrones with a statistics component to provide a useful instrument to perform various kinds of geospatial analysis, in particular reachability analysis. The system is subdivided in three components. The first component is responsible for an efficient computation of isochrones. The second component transforms a network representation of isochrones in an area representation. The third component joins an isochrone with an user-defined spatial relation that contains geo-referenced objects. From a technical perspective, *ISOGA* adopts a service-oriented three-tier architecture and uses standardized OGC services for exchanging spatial data. The system can be accessed at `www.isochrones.inf.unibz.it/isoga`.

7.2 Future Research Directions

Future work points in different directions. First, we want to complete the implementation to cover all possible combinations of transportation modes, including discrete space continuous time networks and continuous space discrete time networks, such as the use of cars.

Second, we will study approximation techniques to further improve the runtime efficiency. The goal is to find a very efficient solution that provides a good approximation of the exact isochrone.

Third, we plan to migrate *ISOGA* into a native mobile application that computes isochrones offline. The major challenges of this task are how to embed a light-weighted spatial DBMS (for instance SpatiaLite) in mobile devices and how to adopt the expensive rendering process of raster and vector data to devices with less computational power.

Fourth, we aim to standardize isochrone queries as a Web Processing Service (WPS). The OGC provides rules for standardizing inputs and outputs (i.e. requests and responses) for invoking geospatial processing services as a Web service.

Finally, we will deploy our prototype system *ISOGA* as a full operational system with the aim to use it as an instrument for various kinds of reachability analysis.

Bibliography

[1] H. W. Armstrong. A network analysis of airport accessibility in south hampshire. *Journal of Transport Economics and Policy*, 6(3):294–307, 1972.

[2] V. Balasubramanian, D. V. Kalashnikov, S. Mehrotra, and N. Venkatasubramanian. Efficient and scalable multi-geography route planning. In *EDBT*, pages 394–405, 2010.

[3] H. Bast. Car or public transport – two worlds. In *Efficient Algorithms*, volume 5760 of *LNCS*, pages 355–367, 2009.

[4] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *In proc. 11th workshop on algorithm engineering and experiments (alenex)*, pages 97–105. SIAM, 2009.

[5] R. Bauer and D. Delling. Sharc: Fast and robust unidirectional routing. In *ALENEX*, pages 13–26, 2008.

[6] R. Bauer, D. Delling, and D. Wagner. Experimental study of speed up techniques for timetable information systems. *Networks*, 57(1):38–52, 2011.

[7] V. Bauer, J. Gamper, R. Loperfido, S. Profanter, S. Putzer, and I. Timko. Computing isochrones in multi-modal, schedule-based transport networks. In *GIS*, pages 1–2. ACM, 2008.

[8] J. Booth, A. P. Sistla, O. Wolfson, and I. F. Cruz. A data model for trip planning in multimodal transportation systems. In *EDBT*, pages 994–1005, 2009.

[9] G. S. Brodal and R. Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electr. Notes Theor. Comput. Sci.*, 92:3–15, 2004.

[10] I. Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Records*, 1645:170–175, 1998.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[12] G. B. Dantzig. *Linear programming and extensions.* Princeton University Press, Princeton, N.J.,, 1963.

[13] V. T. de Almeida and R. H. Gueting. Using dijkstra's algorithm to incrementally find the k-nearest neighbors in spatial network databases. In *SAC*, pages 58–62, 2006.

[14] D. Delling. Time-dependent sharc-routing. *Algorithmica*, 60(1):60–94, 2011.

[15] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li. Instance optimal query processing in spatial networks. *VLDB J.*, 18(3):675–693, 2009.

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

[17] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *EDBT*, pages 205–216, 2008.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[19] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[20] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.

[21] A. V. Goldberg and R. F. F. Werneck. Computing point-to-point shortest paths from external memory. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *ALENEX/ANALCO*, pages 26–40. SIAM, 2005.

[22] Google. General transit feed specification reference, 2011.

[23] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, pages 499–508, New York, NY, USA, 2010. ACM.

[24] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. SECONDO: An extensible DBMS platform for research prototyping and teaching. In K. Aberer, M. J. Franklin, and S. Nishio, editors, *ICDE*, pages 1115–1116. IEEE Computer Society, 2005.

[25] HAFAS.

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

[27] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *VLDB*, pages 562–573. Morgan Kaufmann, 1995.

[28] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[29] M. Hua and J. Pei. Probabilistic path queries in road networks: traffic uncertainty aware path selection. In *EDBT*, pages 347–358, 2010.

[30] R. Huang. A schedule-based pathfinding algorithm for transit networks using pattern first search. *GeoInformatica*, 11(2):269–285, 2007.

[31] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, pages 1–8, 2003.

[32] H. Jeung, M. L. Yiu, X. Zhou, and C. S. Jensen. Path prediction and predictive range querying in road network databases. *VLDB J.*, 19(4):585–602, 2010.

[33] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical optimization of optimal path finding for transportation applications. In *CIKM*, pages 261–268, 1996.

[34] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 10(3):409–432, 1998.

[35] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans. Knowl. Data Eng.*, 14(5):1029–1046, 2002.

[36] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE*, page 10, 2006.

[37] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *WEA*, pages 126–138, 2005.

[38] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.

[39] R. Kothuri, A. Godfrind, and E. Beinat. *Pro Oracle spatial - the essential guide to developing spatially enabled business applications.* Apress, 2004.

[40] K. C. K. Lee, W.-C. Lee, and B. Zheng. Fast object search on road networks. In *EDBT*, pages 1018–1029, 2009.

[41] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[42] S. Marciuska and J. Gamper. Determining objects within isochrones in spatial network databases. In *ADBIS*, pages 392–405. Springer, 2010.

[43] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In *WEA*, pages 189–202, 2005.

[44] M. Müller-Hannemann, M. Schnee, and K. Weihe. Getting train timetables into the main storage. *Electr. Notes Theor. Comput. Sci.*, 66(6):8–17, 2002.

[45] M. Müller-Hannemann, F. Schulz, D. Wagner, and C. D. Zaroliagis. Timetable information: Models and algorithms. In *ATMOS*, pages 67–90, 2004.

[46] M. Müller-Hannemann and K. Weihe. Pareto shortest paths is often feasible in practice. In *Algorithm Engineering*, pages 185–198, 2001.

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

[48] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *J. ACM*, 37(3):607–625, 1990.

[49] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.

[50] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[51] PGRouting. pgrouting project, an open source routing library.

[52] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[53] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *J. Exp. Algorithmics*, 12:2.4:1–2.4:39, June 2008.

[54] E. Pyrga, F. Schulz, D. Wagner, and C. D. Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12, 2007.

[55] M. N. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *PVLDB*, 4(2):69–80, 2010.

[56] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD Conference*, pages 43–54, 2008.

[57] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[58] S. Shekhar and D.-R. Liu. Ccam: A connectivity-clustered access method for networks and network computations. *IEEE Trans. Knowl. Data Eng.*, 9(1):102–119, 1997.

[59] Sony Ericsson. *XPERIA ARCS LT18i,LT18a white paper.*

[60] K. Stolze. SQL/MM spatial - the standard to manage spatial data in a relational database system. In *BTW*, pages 247–264, 2003.

[61] M. Terrovitis, S. Bakiras, D. Papadias, and K. Mouratidis. Constrained shortest path computation. In *SSTD*, pages 181–199, 2005.

[62] M. Thorup and U. Zwick. Approximate distance oracles. In *STOC*, pages 183–192, New York, NY, USA, 2001. ACM.

[63] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In C. Macdonald, I. Ounis, and I. Ruthven, editors, *CIKM*, pages 1785–1794. ACM, 2011.

[64] L. Ulrich. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilitäet: Von der Forschung zur praktischen Anwendung*, pages 219–230, 2004.

[65] various. Babylon dictionary.

[66] Various. Oracle spatial network data model. *An Oracle Technical White Paper*. Technical report, Oracle, 2005.

[67] Wikipedia. Isochrone map.

[68] N. H. M. Wilson and A. Nuzzolo, editors. *Schedule-Based Dynamic Transit Modeling: Theory and Applications*. Operations Research/Computer Science Interface Series. Kluwer Academic Publishers, 2004.

[69] J. Xu. *Moving Objects with Multiple Transportation Modes*. PhD thesis, FernUniversitaet in Hagen, 2012.

[70] K. Xuan, G. Zhao, D. Taniar, M. Safar, and B. Srinivasan. Voronoi-based multi-level range search in mobile navigation. *Multimedia Tools Appl.*, 53(2):459–479, 2011.

[71] K. Xuan, G. Zhao, D. Taniar, B. Srinivasan, M. Safar, and M. L. Gavrilova. Network voronoi diagram based range search. In *AINA*, pages 741–748, 2009.

Copyright © 2013 by Markus Innerebner

Cover design by Markus Innerebner.
Printed and bound by your printer.

ISBN: 90–XXXX–XXX–X