National Technical University of Athens

School of Electrical
and Computer Engineering

Division of Computer Science

Shortest-path algorithms on road networks and real-world applications

PhD Thesis

of

Alexandros Efentakis

Diploma in Physics, University of Athens
MSc in Information Systems, Hellenic Open University

Athens, December 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αλγόριθμοι συντομότερης διαδρομής σε οδικά δίκτυα και εφαρμογές του πραγματικού κόσμου

Διδακτορική Διατριβή

του

Αλέξανδρου Εφεντάκη

Διπλωματούχου Φυσικού του Πανεπιστημίου Αθηνών
ΜΔΕ στα Πληροφοριακά Συστήματα, Ελληνικό Ανοικτό Πανεπιστήμιο

Συμβουλευτική Επιτροπή: Ι. Βασιλείου
D. Pfoser
Τ. Σελλής

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 19η Δεκεμβρίου 2014.

Ι. Βασιλείου Καθ. ΕΜΠ	D. Pfoser Assoc. Prof. George Mason Univ.	Τ. Σελλής Καθ. ΕΜΠ
Κ. Κοντογιάννης Αναπλ. Καθηγ. ΕΜΠ	Ι. Σταύρακας Ερευν. Β΄ Ερ. Κέντρου 'ΑΘΗΝΑ'	Ν. Κοζύρης Καθ. ΕΜΠ

Δ. Γουνόπουλος
Καθ. ΕΚΠΑ

Αθήνα, Δεκέμβριος 2014

. . .

Alexandros Efentakis
BSc in Physics, MSc in Informations Systems, PHD, N.T.U.A.

Contents

List of Figures

List of Tables

PREFACE

This thesis is submitted in fulfillment of the requirements for the degree of Doctor of Philosophy, in the School of Electrical and Computer Engineering, National Technical University of Athens (NTUA), Greece. The presented work describes efficient methods and algorithms for multiple variations of shortest-path queries on road networks, including several real-world applications that have been carried out during the last four years in the Knowledge and Database Systems Laboratory (KDBSL) of NTUA and in the Institute for the Management of Information Systems (IMIS) of Research Center "Athena".

I am very grateful to Prof. Yannis Vasilliou for his invaluable help and supervising this thesis and Prof. Timos Sellis for giving me a chance to work along him in RC "Athena". I would also like to thank all my co-authors and the people I share my office with, whose help and stimulating discussions has been invaluable, namely: Christodoulos, Giorgos, George, Kostas, Nikos, Sophia, Sotiris and Aris, along with the rest of my colleagues in RC "Athena". I also owe a very special thank you to my "mentor" Dieter Pfoser who has really taught me everything I know about research and his wife Nectaria for their amazing help and support. I consider both of them my family and I wish I could someday repay the kindness they have shown me. On a personal note, I would also like to thank: My now deceased parents Markos and Vasiliki (I miss you both very much), my brother Kostas, his great family and my sister Amalia for their constant love and support. My best friends Alexandros and Vasilis for the best fun times. Last but not least, I would like to thank my wife Katerina for persevering with me throughout this work and for simply not giving up on me. Without her endless love and support, I would have given up long ago.

The work presented in this dissertation was also partially funded by the EU 7th Framework Programme under the following projects: Initial Training Network "GEOCROWD" (`http://www.geocrowd.eu`, GA: FP7-PEOPLE-2010-ITN-264994), "SimpleFleet" (`http://www.simplefleet.eu`, GA: FP7-ICT-2011-SME-DCL-296423), "TALOS Mobile Guides" (`http://www.tmguide.eu`, GA: FP7-SME-2012-315333) and "GEOSTREAM" (`http://geocontentstream.eu`, GA: FP7-SME-2012-315631).

Alexandros Efentakis
Athens, December 2014

To Katerina.

ABSTRACT

During the last two decades, shortest-path algorithms in the context of road networks and mapping applications have been a very active area of research that has produced significant results used by thousand of users worldwide. Unfortunately, despite the plethora of efficient methods and algorithms, there has not been an one-size-fits-all solution that may universally handle all the different query variations and use-cases. To this end, this thesis focuses on advancing several aspects of the current state-of-the-art in this suggested area, by improving previous works regarding point-to-point queries and proposing novel algorithms addressing multiple shortest-path query variations, especially in the context of dynamic road-networks with frequent traffic updates. Those query variations include: (i) The single-source shortest-path problem (finding the shortest-path distances between a source vertex and all other graph vertices) (ii) The one-to-many variant (calculating distances between a source vertex and a set of target vertices). (iii) Range queries (find all vertices reachable from source vertex within a given timespan) and finally (iv) k-NN queries (finding the k-nearest objects to a given query location). For all those different query definitions on road networks, the methods and algorithms proposed in this dissertation provide excellent performance and very short preprocessing times, suitable for all types of practical use-cases and applications.

The second main focus of this dissertation is to bridge the gap between the different solutions and types of shortest-path queries on road networks and propose a unified framework of algorithms and data structures that may efficiently handle all the different variations of shortest-path problems, while requiring very short preprocessing times of a few seconds, so that it may also be used in the case of dynamic road networks with frequent traffic updates. Such a solution will be extremely useful to many practical problems and may be used in several real-world applications. Consequently, within this work we have also developed a real-time service that demonstrates the practical applications of our novel results by utilizing traffic-data originating from vehicle fleets. Moreover, the results of our efforts have been significantly extended to multiple domains, from geomarketing applications and map-updates to crowdsourcing sentiments of users in relation to space.

Chapter 1

Introduction

Shortest-path algorithms in the context of road networks and mapping applications have been a very active area of research during recent years that has produced significant results used by thousand of users worldwide. Unfortunately, despite the plethora of efficient methods and algorithms, there has not been an one-size-fits-all solution that may universally handle all the different query variations and use-cases. To this end, this thesis focuses on advancing the current state-of-the-art in this suggested area, by improving previous works and proposing new methods and algorithms for certain shortest-path query variations, especially in the context of dynamic road-networks with frequent traffic updates. Moreover, during this thesis we developed novel frameworks and real-time services that demonstrate the results of our algorithmic innovations in a real-world setting. In addition, we have significantly extended our results to multiple domains, from geomarketing applications and map-updates to crowdsourcing sentiments of users in relation to space. This chapter will present the overall scientific focus of our efforts and highlight the individual contributions of this dissertation.

1.1 Prologue

In the last two decades, there was a great number of publications focusing in answering point-to-point shortest-path queries in road networks. Although the classic Dijkstra algorithm [34] solves the single-pair shortest-path (SPSP) problem of finding an shortest-path of length $d(s, t)$ between an origin s and a destination t in a graph $G = (V, E, w)$ (where w is a positive weight function $E \rightarrow R^+$), more efficient algorithms typically use a two-stage approach: preprocessing requires a few minutes (or hours) and produces additional data that is subsequently used to accelerate shortest-path queries. The related research on this specific algorithmic area has evolved so rapidly that even recent overviews, such as [27, 12] had to be updated in subsequent publications [10].

Existing methods for solving the SPSP problem in road networks may be classified to three major categories. *Hierarchical Approaches* such as Transit Node Routing (TNR) [9], Contraction Hierarchies (CH) [49] or the Hub-based Labeling algorithm (HL) [1] exploit the inherent hierarchical structure of the given road network and build a *search Graph* which employs *shortcuts*, i.e., additional edges connecting important nodes. In contrast, *goal direction techniques* such as ALT [50] and Arc-flags [72, 86] direct the search towards the target by preferring edges that shorten the distance to the goal node and ignoring edges that cannot possibly belong to the shortest-path based on their preprocessed data. A third category is based on *graph separators* such as HiTi [65] and Customizable Route

Planning (CRP) [24, 30]. During preprocessing, one computes a multilevel partition of the graph to create a series of interconnected overlay graphs. A query starts at the lowest (local) level and moves to higher (global) levels as it progresses.

Many of those acceleration techniques have been significantly improved over the years. State-of-the-art methods such as TNR and HL originally required extensive preprocessing time of more than a few hours. Later works [2], [7] improved those preprocessing times to just a few minutes. Unfortunately those preprocessing times are still not fast enough for real-time mapping services, where edge weights change frequently due to traffic updates (typically every 15 - 30 minutes). Additionally, since they are based on Contraction Hierarchies (CH), whenever edge weights change new shortcuts must be added and others must be removed from the search graph, altering the search graph's entire structure, making path unpacking harder and slower to implement. Additionally, CH is very sensitive to the metric used, making the preprocessing significantly slower for (i) travel distances (ii) turn restrictions [24]. That is why mapping services such as Bing Maps prefer to use CRP which might be orders of magnitude slower than HL or TNR, but has faster preprocessing times (few secs) and uses always the same shortcuts regardless of the metric used [30, 24]. To sum up, hierarchical methods, although providing superior shortest-path computation speeds are not suited for a dynamic navigation scenarios where edge weights of the road network change based on actual traffic conditions.

In the case of the single-source shortest-path problem (SSSP) or the one-to-all problem, given a source vertex s, the goal is to find SP distances from s to *all other graph vertices*. Quite recently, Delling et al. [23] introduced the PHAST algorithm that, compared to Dijkstra, needs fewer operations, has better locality, and exploits parallelism at multicore and instruction levels. As a result, it is orders of magnitude faster. Later works [26] presented RPHAST for solving the one-to-many variant: given a set of targets T, compute the distances between s and all vertices in T. Since both PHAST and RPHAST are extensions of the Contraction Hierarchies algorithm, their preprocessing is slow, making those methods also practically unsuitable for dynamic scenarios. Moreover, since CH is essentially a bidirectional method, when applied to the SSSP problem, it may only be used if we know the target nodes in advance, as in the entire road network in PHAST, or a subset T of nodes in RPHAST. Therefore, these methods cannot be extended to *range queries*, i.e., find all nodes reachable from s within a given timespan) or *isochrone queries*, i.e., find all nodes AND edges reachable from s within a given timespan, since for those queries we do not know the target nodes in advance.

Another fundamental problem frequently encountered in location-based services is the k-NN query, i.e., given a query location and a set of objects on the road network, the k-NN search finds the k-nearest objects to the query location. This problem may be used in several contexts, such as tourists looking for k-nearest sights or dispatching nearby taxis to customer locations. Unfortunately, existing approaches, such as SILC [102] or ROAD [73, 74] cannot scale for continental road networks [126]. Even later attempts, such as G-tree [126] improving previous methods, still *require 16.8 hours of preprocessing for continental road networks*. In addition, for a large number of randomly distributed objects, an efficient Dijkstra implementation could answer k-NN queries (for small values of k) by settling a few hundreds nodes and requiring $< 1ms$. This performance benchmark is often overlooked when assessing novel k-NN methods. Moreover, most existing approaches require a *target-selection phase*, i.e., they need to mark the objects location within the underlying index. This phase takes a few seconds, hence having limited appeal for applications involving moving objects (e.g., vehicles). Therefore, it only makes sense

to use a complex (as in non-Dijkstra) *k*-NN processing framework in cases of either rather "small" numbers of objects or objects following skewed distributions (e.g., POIs located near the city center), i.e., for cases in which Dijkstra does not perform well.

Moreover, there is another significant limitation to previous approaches. Specialized indexes such as G-tree, may only answer k-NN queries. As a result, we require additional data structures for answering the most common type of queries, which is the typical point-to-point shortest-path queries. Having several data structures for different types of queries within the same application server, increases the complexity (and the required resources) and as a result it is not suitable for practical real-world applications. This particular problem has been tried to be addressed by an adaptation of CRP in [31] but unfortunately it shares the major limitation of previous methods: It performs well only for larger number of objects (where Dijkstra already provides excellent performance) and also needs a target-selection phase, which still requires a few seconds. As a result, Customizable Route Planning may only be applied for static objects.

Conclusively, there are many distinct research works addressing the different variants of shortest-path queries on road networks. This work aims to bridge the gap between the different solutions and propose a unified framework of algorithms and data structures that may efficiently handle all the different variations of shortest-path queries on road networks. The proposed solution provides excellent query performance and requires very short preprocessing times of a few seconds, so that it may also be used in dynamic road networks with frequent traffic updates. Such a solution will be extremely useful for multiple practical problems and may be used in several real-world applications.

In addition, within the scope of this work we have developed a system / service that demonstrates the practical test-cases of our novel results. The system built uses real (and not simulated) traffic data originating from vehicle fleets and covers the entire workflow of traffic related services from storing raw GPS-traces of fleet vehicles to live-traffic estimation and live-traffic shortest-path and isochrone computation. The interesting fact is that by using the created system in combination with additional demographics data, we can extract meaningful information regarding the impact of traffic to multiple domains from market research, urban planning to geomarketing. In addition, we may further exploit the results of the created infrastructure to infer information missing from the original road network dataset, in the form of turning restrictions. This way, we can evolutionary improve the quality of our provided services.

1.2 Contribution

The contribution of this dissertation may be classified in seven major categories. We will describe each category briefly along with the individual scientific contributions per category:

1.2.1 Single-pair shortest-path queries

The first contribution of our work is to significantly improve the performance of the ALT (A^* + Landmarks + Triangle equality) algorithm [50] and making it suitable for a dynamic navigation scenario. We focused our attention on the ALT algorithm, since it has none of the aforementioned disadvantages of hierarchical methods, i.e., (i) it is very robust with respect to the metric used [50] (ii) it does not requires any extra path-unpacking routine for producing the actual road network path of the shortest route (iii) its storage requirements

and auxiliary data structure size depend solely on the number of landmarks (and not on the utilized metric) and most importantly (iv) the typical landmarks-based preprocessing may also be used for estimating the graph distance between any two vertices. For that reason, it has been used in other contexts outside road networks (such as social networks) in cases when the actual distance between two nodes may be sufficiently replaced by the close estimation provided by the typical landmarks preprocessing [97, 115].

Our specific contributions as described in [43] are to improve (i) ALT's preprocessing time and (ii) its shortest-path query phase performance. By proposing a novel, simple, yet efficient landmark selection strategy and exploiting several optimization strategies, we managed *to lower the preprocessing time from several minutes to a few seconds*. Moreover, we also improved ALT's *query phase* and *tripled unidirectional ALT performance* while also *improving bidirectional performance by 44%*. Although we did not alter the actual algorithm (including memory requirements and time complexity) our efforts significantly broadened ALT's entire scope, since: (i) its preprocessing is now fast enough for supporting dynamic road networks with frequent traffic updates (ii) ALT algorithm is now fast enough to support real time SP queries for global scale mapping services.

Our second contribution in terms of SPSP queries, as described in [47], is to showcase a system which efficiently distributes typical graph separator shortest-path preprocessing to multiple web-clients. Instead of using a dedicated cluster of nodes connected to a network infrastructure, we used Web-browsers and Javascript as our computing platform. All the necessary computation work is distributed to the Web-browsers and the server just transmits cells and collects their results. Hence, not only the computation load on the server remained minimal but even the Web-clients hardly experience any load, since each cell's clique calculation takes less than 2s and memory usage remains below 150MB. Therefore, our client-side approach may work on any conventional workstation or current mobile device. Our extensive experimentation with a continental road network showed that the proposed approach is not only feasible but very fast and efficient as well. *With 8 Web-clients, preprocessing for a continent-sized road network requires 3min and we can answer SP queries in almost 2ms*. To the best of our knowledge, this is still the only work that uses Web-browsers and Javascript as a computing platform in the context of shortest-path computation.

1.2.2 Single-source shortest-path queries

Regarding single-source shortest-path (SSSP) queries, our contribution in [44], is to create novel graph-separator methods to efficiently handle all variations of the single-source shortest-path (SSSP) problem. The three proposed algorithms, GRASP, isoGRASP and reGRASP are each tailored to a specific SSSP problem (one-to-all, range, and one-to-many). They also require minimal preprocessing time (their preprocessing time is 1-2 orders of magnitude faster than PHAST and RPHAST) and, thus, are the only viable solution for handling dynamic road networks, i.e., road networks with changing edge weights due to traffic updates. Moreover, they provide excellent parallel performance for both travel times and travel distances, scale better on multicore processors and offer better or comparable performance to state-of-the-art approaches. But most of all, they may efficiently solve range / isochrone queries, not addressed by previous solutions. As recent works have suggested (cf. [38, 40]), this type of queries is very important for a spectrum of application contexts, including fleet management, urban planning and geomarketing. Moreover, all GRASP variants utilize the same graph structures and therefore can be used

within the same server infrastructure to concurrently answer all related queries. Thus, it is the only approach to offer this kind of simplicity and efficiency for all SSSP cases.

1.2.3 The SALT framework

Building on our previous contributions, namely the improvements of the ALT algorithm and the GRASP algorithms for handling all variations of the SSSP queries, we also propose a unified algorithmic solution that may be used in a *dynamic road network* context by having very short preprocessing times and competitive query times, while covering a *wide range of shortest-path problems*, such as (i) single-pair, (ii) one-to-all, (iii) one-to-many, (iv) range and (v) k-NN queries. Specifically, we aimed at combining the fragmented approaches related to the various shortest-path problem definitions and instead propose a unified framework that tackles all of them. Our proposed **SALT** (graph Separators + ALT) framework requires only seconds for preprocessing continental road networks and provides excellent query performance for a wide range of problems. Our experiments showed that SALT is (i) $3-4\times$ faster for point-to-point queries when compared to existing methods of similar preprocessing times, (ii) it answers one-to-all, one-to-many and range queries with comparable performance to state-of-the-art approaches and most importantly, (iii) it may also answer k-NN queries in $< 1ms$, for both, static or moving objects. As such, our SALT framework could be a *swiss-army-knife for tackling all shortest-path problem variants*, making it a serious contender for use in commercial applications.

1.2.4 Real-world applications

Although our aforementioned contributions are very interesting from a theoretical perspective, they may be also used in a range of practical real-world applications. Thus, we focused on creating a system that will showcase the entire workflow of traffic-related services, from vehicle GPS-traces' acquisition and storage to live-traffic estimation and live-traffic shortest-path and isochrone computation. To that purpose, we combined state-of-the-art research about road networks, Floating Car Data, map-matching, historic speed profile computation, live-traffic assessment and time-dependent shortest-path computation to provide an efficient, yet economical fleet management solution. This process was presented in [38] and its result, the fully functional SimpleFleet system and its accompanying demo [42] now cover the urban regions of three European metropolitan cities namely: Athens (Greece), Berlin (Germany) and Vienna (Austria). Creating the actual service required several intermediate steps such as: Creating road network graphs from OpenStreetMaps data, collecting a large amount of Floating Car Data (FCD) from fleet vehicles, applying state-of-the-art map-matching algorithms on this data for aligning the GPS traces to the road network graph and consequently producing high-quality historic speed profiles along with frequently updated live-traffic assessment. This combination of live-traffic information and speed profiles was subsequently used to provide up-to-date live-traffic shortest-path and isochrone computation (refreshed every 5 minutes). Although, global mapping services, such as Google or Bing Maps provide similar functionality, to the best of our knowledge this is the first work to combine the state-of-the-art isochrone computation of [82] with live-traffic data, in addition to providing this real-time system showcasing our results.

1.2.5 Real-world results

Although our main focus in our research was mainly towards shortest-path algorithms, the live-traffic assessment results produced by our fully functional SimpleFleet system in [38], was proven to be extremely beneficial to other contexts as well. Such a context is geomarketing, i.e., the integration of geographical information into various aspects of business intelligence, such as marketing, sales and distribution. A crucial component in the effort of integrating traffic information in a geomarketing context is the concept of *isochrones*, which are informally defined as the area from which a specific point of interest is reachable within a given time interval. In this sense, isochrones may provide accurate information about where to build a new franchise store in order to reach a larger pool of customers or identify areas less covered by existing stores. Since our work in [38] provided state-of-the-art isochrone computation based on live-traffic, this increased accuracy offers a unique advantage in comparison to typical static road-network approaches.

To this end, we have developed an additional geomarketing application that combines live-traffic, isochrones and demographic / business data for Berlin and Vienna in order to demonstrate the actual impact of traffic fluctuations to business intelligence decisions. Results were quite impressive: For both cities, *if we take traffic into account and for the 5 minute timespan - isochrone, we reach less than 15% of the potential customers we would have calculated on a static traffic-less road network graph*. Although this gap decreases for larger timespans, still for a timespan of 15 minutes the impact of traffic is more than 20-40%, i.e., the number of actual customers we reach within 15min is actually 20-40% smaller than the number calculated by the typical traffic-less scenario. In the case of congestion, the impact of traffic was shown to be even more dramatic, i.e., even for a timespan of 15 minutes the traffic's impact is more than 60% for both cities.

1.2.6 Turning restrictions

Turning restrictions on road networks are especially important for any routing / isochrone service. While a lot of scientific literature has focused on time-dependency on road networks (due to the traffic fluctuation), there is only a limited number of works that deal with turning restrictions. This is mostly due to the fact that *"no publicly-available realistic turn data exist"* [24]. In this dissertation, we propose a method to automatically identify / infer turning restrictions in the road-network dataset of OpenStreetMap by utilizing historic map-matched trajectories produced by our SimpleFleet system of [38], i.e., we crowdsourced the identification of turning restrictions to local vehicle drivers by mining the map-matched trajectories produced by them. This is also the true novelty of our contribution: instead of using the GPS trajectories directly, we use the map-matching results derived from them. Our approach makes sense: In comparison to raw GPS traces, map-matched trajectories are: (i) more condensed, since instead of random locations in the plane we have edge sequences and (ii) less error-prone (if an efficient map-matching algorithm is used) since they are interpolated with the actual road network. Therefore it made sense to utilize those historic results to extrapolate this additional meaningful information, instead of using raw FCD like most previous works. To the best of our knowledge, we are the first to utilize map-matched trajectories for such a task.

Our method for inferring turning-restrictions from map-matched trajectories as presented in [39] and [41] has proven solid: 66-74% of the turning restrictions we have extracted may be successfully verified. However, the most important result of is that we have identified and verified 2-18 times more turning restrictions than those existing in the

original datasets. This impressive feat proves the validity and credibility of our method.

1.2.7 Sentiment mapping travelblogs

One common theme in multiple aspects of our work is crowdsourcing, i.e., exploiting the results, processing power and data of individual users and online communities in order to further extract additional meaningful information, that in our case has also a prominent geographic footprint. In [47], we showcased a system, which efficiently distributes typical graph separator shortest-path preprocessing to multiple web-clients and took advantage of the processing power of potential users using the aforementioned infrastructure. In [38] we used the GPS-traces created by vehicle fleets to extract information about live-traffic and build historic speed-profiles for three European cities. In [39], we used the map-matched trajectories created by the service of [38] to infer information about existing turning-restrictions in the road network dataset. Finally, in [21] and [36] we focus on using travel blog entries to extract, aggregate and visualize the user sentiments in relation to geographic location. By crawling various travel related web sites (e.g. travelpod.com, travelblog.org), we created a corpus of 150,000 texts. Analyzing this text at the paragraph level for sentiment information, we essentially created a geospatial sentiment-map based on the user-contributed information contained in the articles of the respective travelblogs.

1.3 Outline

The outline of this work is as follows. Chapter 2 describes previous work related to all our contributions. The next eight chapters describe the individual scientific contribution of this dissertation. Chapter 3 describes our improvements to the ALT algorithm. Chapter 4 describes the architecture of a system which efficiently distributes typical graph separator shortest-path preprocessing to multiple Web-clients. Chapter 5 describes our novel family of GRASP algorithms for tackling multiple variations of the SSSP problem. Chapter 6 describes out SALT framework, which requires only seconds for preprocessing continental road networks and provides excellent query performance for a wide range of shortest-path problems, such as (i) single-pair, (ii) one-to-all, (iii) one-to-many, (iv) range and (v) k-NN queries. Chapter 7 describes our SimpleFleet system that showcases the entire workflow of traffic-related services, from vehicle GPS traces acquisition to live-traffic estimation and live-traffic shortest-path routing and isochrone computation. Chapter 8 exploits the results of the previous service to quantify the impact of live-traffic in the context of geomarketing decisions. Chapter 9 uses the map-matched trajectories produced by the SimpleFleet system to extract / infer turning restrictions information in the road-network dataset. Chapter 10 presents our effort for mapping the spatial distribution of the sentiments of travelblog users, in relation to the geographic areas visited by them. Finally, the last chapter of this dissertation provides conclusions and directions for future work.

Chapter 2

Related Work

In this chapter, we will introduce the basic notation and definitions that will be used throughout the individual chapters of this thesis. Moreover, we are also going to present and summarize the necessary information in terms of related work, regarding the discrete contributions of this dissertation. In detail, we will present basic information about the current state-of-the-art method and works, regarding single-pair, single-source and k-NN queries in road networks, as well as iscohrone computation and turning restrictions.

2.1 Preliminaries

In the discussion on related work that follows, we focus on directed weighted graphs $G(V, E, w)$, where V is a finite set of vertices, $E \subseteq V\text{x}V$ are the arcs of the graph and w is a positive weight function $E \rightarrow R^+$. Typically on road networks, weight w represents the travel time required to traverse the arc. For travel distances, w refers to the length of the arc in meters. The reverse graph $\bar{G} = (V, E)$ is the graph obtained from G by substituting each arc $(u, v) \in E$ by (v, u). A partition of V is a family of sets $C = \{c_0, c_1, \dots c_M\}$, such that each node $u \in V$ is contained in exactly one set c_i. An element of a partition is called a *cell*. A multilevel partition of V is a family of partitions $\{C^0, C^1, \dots C^L\}$ where ℓ denotes the level of a partition C^ℓ. Similar to [24], level 0 refers to the original graph, L is the highest partition level and in this work we use *nested multilevel partitions*, i.e., for each $\ell < L$ and each cell c_i^ℓ there exists a unique cell $c_j^{\ell+1}$ (called the supercell of c_i^ℓ) with $c_i^\ell \subseteq c_j^{\ell+1}$. Accordingly, c_i^ℓ is a subcell of $c_j^{\ell+1}$. In this notation, $c^\ell(v)$ is the cell containing the vertex v on level ℓ. Accordingly, the number of cells of the partition C^ℓ is denoted as $|C^\ell|$. For a boundary arc on level ℓ, the tail and head vertices are located in different level-ℓ cells; a boundary vertex on level ℓ is connected with at least one vertex in another level-ℓ cell. Note that for nested multilevel partitions, a boundary vertex/arc at level ℓ is also a boundary vertex/arc for all levels below.

2.2 Single-pair shortest-path queries

In the single-pair shortest-path (SPSP) problem we aim to find a shortest path of length $d(s, t)$ between an origin s and a destination t in the graph $G = (V, E, w)$. Although there is significant scientific literature, especially regarding SPSP queries on road networks, we will mainly focus on previous works that are more closely related to this dissertation.

$$d(u,v) <= \min_{Li}(d(u,L_i) + d(L_i,v))$$

$$d(u,v) >= \max_{Li}(\max\{d(u,L_i) - d(v,L_i), d(L_i,v) - d(L_i,u)\})$$

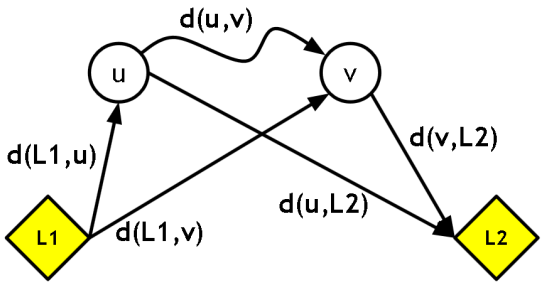


Figure 2.1: *Upper and lower bounds for graph distances, using landmarks*

2.2.1 The ALT algorithm

The concept of landmarks within the context of the single-pair shortest-path problem was officially introduced in [50]. In this work, a small set of vertices called landmarks is chosen and for each vertex, the authors precompute distances to and from every landmark. Given a set $S \subseteq V$ of landmarks and distances $d(L_i, v)$, $d(v, L_i)$ for all vertices $v \in V$ and landmarks $L_i \in S$, the following triangle inequalities hold: $d(u, v) + d(v, L_i) \geq d(u, L_i)$ and $d(L_i, u) + d(u, v) \geq d(L_i, v)$. Therefore, the function $\pi_f = max_{L_i} max\{d(u, L_i) - d(v, L_i), d(L_i, v) - d(L_i, u)\}$, where $0 \leq i \leq |S| - 1$, provides a lower bound for the graph distance $d(u, v)$.

Later works [97] expanded this concept and by using triangle equality again, showed that landmarks may also be used for providing upper bounds on the distance between any two graph vertices. Thus, the graph distance $d(u, v) \leq min_{L_i}(d(u, L_i) + d(L_i, v))$ for any graph vertices u and v. As a result, since landmarks preprocessing data provides both upper and lower bounds for the distance between any graph vertices pair, landmarks provide a very fast and efficient way to approximate graph distances. Conclusively for the graph distance $d(u, v)$ and for $0 \leq i \leq |S| - 1$ the following equations apply (see Fig. 2.1):

$$d(u, v) \geq max_{L_i} max\{d(u, L_i) - d(v, L_i), d(L_i, v) - d(L_i, u)\} \tag{2.1}$$

$$d(u, v) \leq min_{L_i}(d(u, L_i) + d(L_i, v)) \tag{2.2}$$

ALT is a bidirectional variant of the classic A* algorithm [57] using the aforementioned lower bounds. Since the combination of A* and bidirectional search is not trivial, correctness can only be guaranteed if π_f (the heuristic function for the forward search) and π_r (the heuristic function for the backward search) are consistent. This means $\pi_f(u, v)$ in G must be equal to $\pi_r(v, u)$ in the reverse graph. ALT typically uses the average potential function [61] defined as $p_f(v) = (\pi_f(v) - \pi_r(v))/2$ for the forward and $p_r(v) = (\pi_r(v) - \pi_f(v))/2 = -p_f(v)$ for the backward search.

The original implementation of ALT uses for each SP computation, only a subset of *h active landmarks*, which are those that provide the best lower bounds on the $s - t$ distance. Later works [51] update the set of active landmarks dynamically during the query phase. The computation starts using the initially best landmarks and as the algorithm progresses additional landmarks (which may provide better lower bounds) are brought into the active set. After every active landmark update, the potential functions change and therefore the priority queues must also be updated. Additionally the algorithm can

no longer terminate as soon as the two opposite searches meet. Instead the ALT algorithm may safely terminate only when the sum of minimum keys in the forward and the backward queue exceeds $\mu + p_f(s)$, where μ represents the tentative shortest path length.

Preprocessing. The *preprocessing stage* for ALT is divided in two phases, the landmarks selection process and the computation of distances of all other graph vertices from and to the landmarks. As far as the landmark selection process is concerned, many alternative strategies have been suggested in [50] and [51]. As Delling et al. suggest in [28], "*no technique picks landmarks that universally yield the smallest search space for random queries*" (although some methods, such as the Avoid and maxCover [51] typically perform better).

2.2.2 Contraction Hierarchies

The Contraction hierarchies (CH) algorithm [49] efficiently solves the SPSP problem on road networks. In the preprocessing phase, CH picks an ordering of the graph vertices and shortcuts them according to this specific order, i.e., for any pair u, y of neighbours of v such that $(u, v), (v, y)$ is the only shortest path between u and y in the current graph, CH adds a shortcut (u, y) with $w(u, y) = w(u, v) + w(v, y)$. To check if this added shortcut is actually needed, CH runs a local Dijkstra search between u and y. The final output of the CH preprocessing routine is the set E^+ of shortcut arcs and the position of each vertex v in the node ordering (denoted by $rank(v)$). Although any order provides a correct algorithm, best results are obtained by using on-line heuristics that select the next vertex to shortcut based on the number of arcs added and removed from the graph in each shortcut step [49].

The query phase of Contraction hierarchies runs a bidirectional Dijkstra algorithm in the graph $G^+ = (V, E \cup E^+)$, with one important difference: both forward and reverse searches only look at upward arcs, i.e., the ones leading to neighbours with higher rank. More precisely, let $E\uparrow = \{(v, w) \in E \cup E^+ : rank(v) < rank(w)\}$ and $E\downarrow = \{(v, w) \in E \cup E^+ : rank(v) > rank(w)\}$. During queries, the forward search is restricted to $G\uparrow = (V, E\uparrow)$ and the reverse search to $G\downarrow = (V, E\downarrow)$.

During the query phase, each vertex v maintains estimates on distances $d_s(v)$ from s (found by the forward search) and $d_t(v)$ from t (found by the reverse search). Initially, both values were set to infinity for all vertices. As the search progresses, the CH algorithm keeps track of the vertex u minimizing $\mu = d_s(u) + d_t(u)$ and the search terminates as soon as the minimum value in each priority queue is greater than μ. [49] showed that the maximum-rank vertex u on the shortest $s - t$ path minimizes $d_s(u) + d_t(u)$ and the actual shortest path from s to t is given by the concatenation of the $s - u$ path (belonging to $G\uparrow$) and $u - t$ path (belonging to to $G\downarrow$).

2.2.3 Graph separators

Another popular acceleration technique for the single pair shortest-path problem on road networks is based on *graph separators*. In graph-separator (GS) approaches, such as Customizable Route Planning (CRP) [24, 30] a partition C of the graph is computed. Then, the preprocessing stage builds a graph H containing all boundary nodes and boundary arcs of G. It also contains a clique for each cell c: for every pair (u, v) of boundary nodes in c, a clique arc (u, v) is created whose cost is the same as the shortest path (restricted to inner arcs of c) between u and v. The graph H is an overlay [108]: the distance between any two nodes in H is the same as in original graph G. In order to perform a SP query between s

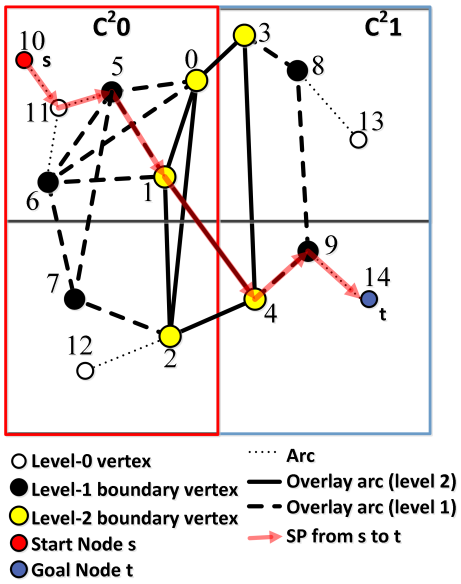


Figure 2.2: *A typical SP query for a 2-level graph-separator overlay graph*

and t, a unidirectional or bidirectional version of Dijkstra's algorithm must be run on the graph consisting of the union of H, $c_0(s)$ and $c_0(t)$. In order to accelerate queries, multiple levels of overlay graphs may be used as in [24, 30], which is a quite common accelerating technique for most partition-based approaches. For that purpose, CRP uses the PUNCH partitioning tool [25] to create multilevel, nested (i.e., a boundary vertex at level ℓ is also a boundary vertex at level $\ell - 1$, for $\ell \geq 1$) partitions of graph G, in top-down fashion.

Figure 2.2 shows a typical example of the construction of a multi-level overlay graph, connecting boundary vertices of partitions on different levels and how this overlay graph hierarchy is used to accelerate a SP query between a start node s and a goal node t.

Since each clique is calculated by using only the inner arcs of c, GS preprocessing may be easily parallelized, since each clique calculation may be assigned to a different process. Moreover, overlay graphs of higher level partitions may be computed by using the overlay graphs of lower levels to further reduce preprocessing time. By using those two optimizations, CRP is the most efficient SPSP algorithm in terms of preprocessing time (requiring few seconds for continental road networks) and is thus suitable for time dependent scenarios, i.e., road networks where arc weights change frequently due to traffic updates (usually every 15-30 minutes). Another advantage of graph separators methods is their robustness to the metric used. Therefore they provide stable performance and fast preprocessing times for either travel distances or turn costs, contrary to hierarchical methods such as CH where preprocessing time increases significantly when we switch to travel distances metric [24]. That is why global mapping services such as Bing Maps, prefer to use CRP, which although one order of magnitude slower than CH, it is still fast enough for real-time SP queries.

2.3 Single-source shortest-path queries

In the single-source shortest-path problem (SSSP) or the *one-to-all* problem, given the graph G and a source vertex s, the goal is to find shortest path distances from s to all other vertices in the graph. Delling et al. introduced the PHAST algorithm [23], which extended Contraction Hierarchies to efficiently answer SSSP queries. The preprocessing

phase of PHAST runs the exacts CH preprocessing, which provides a set of shortcuts E^+ and a vertex ordering. A PHAST query initially sets $d(v)=\infty$ for all $v \neq s$, and $d(s)=0$. It then executes the actual SSSP query in two subphases. First, it performs a simple forward CH search (denoted hereafter as the *upward phase*): it runs Dijkstra algorithm from s in $G\uparrow$, stopping when the priority queue becomes empty. This sets the distance labels $d(v)$ of all vertices visited by the search. The second (scanning) subphase scans all vertices in $G\downarrow$ in descending rank order, i.e., vertices on level l are only visited after all vertices on levels greater than l have been processed. To scan v, the PHAST algorithm examines each incoming arc $(u, v) \in E\downarrow$; if $d(v) > d(u)+w(u, v)$, PHAST sets $d(v) = d(u)+w(u, v)$. The main advantage of PHAST, is that only the (cheap) upward phase depends on the source s. The expensive scanning phase (i) visits all vertices and arcs in the same order regardless of the source and (ii) vertices belonging to the same CH level may be processed in parallel. As a result, the sequential PHAST algorithm is about 15 times faster than Dijkstra's algorithm and its parallelized CPU implementation requires *39-60ms* for solving the SSSP problem in continental road networks.

Later, the same authors introduced the RPHAST algorithm [26] for solving the *one-to-many* variant of finding shortest path distances from the source s to a non-empty set of targets $T \subseteq V$. RPHAST uses the exact same preprocessing as PHAST but contrary to PHAST, RPHAST has an additional target selection phase. Once T is known, RPHAST extracts from the Contraction Hierarchy only the information necessary to compute the distances from any source $s \in V$ to all targets T, creating a restricted downward graph denoted as $G_T^{\downarrow}$. RPHAST has the same query phase as PHAST but using $G_T^{\downarrow}$ instead of $G\downarrow$, during the scanning phase. Every time the set of target T changes, the target selection phase must be rerun as well. Still, the target selection phase takes less than $1s$ and therefore is significantly faster than running the entire CH preprocessing.

2.4 k-NN queries

There are many works on k-NN queries for static objects on road networks. ROAD [73, 74] extends the Dijkstra algorithm by using a hierarchical structure similar to graph-separators, by recursively partitioning a road network into sub-networks and precomputing shortest-path distances of "shortcuts" within a sub-network. By using Dijkstra-like network expansion, ROAD skips sub-networks which do not contain an object. Thus, it cannot efficiently prune subnetworks when the number of objects is quite large or when objects are uniformly distributed throughout the network. SILC [102] pre-computes all shortest paths between all vertex pairs and utilizes a quadtree-based encoding to store those precomputed shortest paths. Thus, SILC consumes $O(|V|^{1.5})$ storage space and requires extremely high preprocessing times which makes it unsuitable for continental road networks (SILC will require 618GB for the USA network [126]). Trying to address those drawbacks, G-tree [126] is a balanced tree structure, which is also constructed by recursively partitioning the road network into sub-networks. In this index structure, each G-tree node corresponds to a sub-network. Then the best-first algorithm is applied on the G-tree index structure to answer k-NN queries. Although building the G-tree index is faster than either SILC or ROAD, *it still requires 16.8 hours for the full USA network*. This time will double in case of directed networks. As a result, previous indexing solutions are totally unsuitable for dynamic road networks, i.e., road networks where edge weights change frequently due to traffic updates. In addition, all previous approaches require a *target selection phase* to index which tree-nodes or sub-networks contain objects (a process that

requires several seconds) and as a result they cannot be applied in case of moving objects.

There are also various works addressing k-NN queries for moving objects on road networks. Jensen et al. [63] formalized this problem and presented a system prototype for such queries. Mouratidis et al. [87] focused on continuously monitoring k-nearest neighbours of one moving object in comparison to the others (also moving objects). Later works [117] handle what they refer as *snapshot* (i.e., one-time) queries, where although the objects are moving, at the time of query we consider them static. Still, all those previous approaches are either disk-based [117], have not been tested on continental road networks [63, 87, 117] and they do not address dynamic road networks, where arc weights change frequently. As such, they are different in scope from our work.

Recently, CRP was expanded [31] to handle k-NN queries. Unfortunately, it shares some of the limitations of previous methods, since (i) it also requires a target selection phase (requiring from a few ms to a few sec) and therefore cannot applied to moving objects and (ii) it may only perform well when the objects are near the query location (otherwise the whole upper level of the overlay graph must be traversed). As a result, this solution is also far from optimal.

2.5 Isochrones

Isochrones are defined in [13] as the "set of all points from which a specific point of interest is reachable within a given time span". An important paper for the concept of isochrones and related to our work is [82], since it was the first to claim that the whole spatial area covered by an isochrone is important. In addition, that work introduced the "Edges' Hull" algorithm which creates a single area which is defined by a polygon composed of the outermost edges of the isochrone network. This approach offers increased accuracy in comparison to previous, typical convex hull approaches. Although isochrones have been utilized in public transport and walking combinations in [13] and [48], to the best of our knowledge, we are the first to combine the state-of-the-art isochrone computation of [82] with real live-traffic data in [38] (cf. Chapter 7) .

2.6 Turning-restrictions

Real-time Floating Car Data (FCD) collected by vehicles equipped with GPS-tracking devices has become the mainstream in traffic study because of its cost-effectiveness, flexibility and being the "*the only significant traffic data source with the prospect of global coverage in the future*" [59]. Typically, a GPS-trajectory describing a vehicle movement, consists of a sequence of measurements with latitude, longitude and timestamp information. However, this data is inherently imprecise *"due to measurement errors caused by the limited GPS accuracy and the sampling error caused by the sampling rate"* [96]. Therefore the observed GPS positions often need to be aligned with the road network graph. This process is called map-matching. Hence, a map-matching (MM) algorithm accepts as input a vehicle's GPS trajectory and outputs an ordered sequence of road network graph edges that this vehicle has traversed, along with travel time information, i.e., how long did it take for the specific vehicle to traverse the calculated path.

Despite their inherent imprecision and the low sampling rate of available datasets, latest years saw an explosion of research around GPS trajectories ([125] presents a partial overview of GPS research). Nevertheless, so far, only a limited portion of this research

focused on road network intersections. This is a major oversight, since intersections are important components of urban road networks and contribute much to the total travel time cost [90, 116]. [90] concludes that intersection delays i.e., the turn cost associated with the continuation of travel between edges via an intersection node [121] contribute to 17-35% of the total travel time, according to a survey in the Copenhagen urban area.

The few research works around road network intersections that actually exist, focused on estimating intersection delays based on the available Floating Car Data. Some researchers have utilized the historical mean method to calculate the intersection delays [111, 124], while other authors employ piecewise linear interpolation [8, 122]. Additional works employed the principal curves method [58] to overcome data sparseness of Floating Car Data and calculate turn delays tables for the region of Beijing [78].

Although turn costs / intersection delays are a generalization of turning restrictions (i.e., a turning restriction is a turn with delay set to ∞) existing works cannot extract information about turning restrictions because to calculate turn-cost for a specific turn, many vehicles need to actually use it. On the contrary, such data is missing for turning restrictions, because no vehicles actually traverse those prohibited turns. Thus, we could not find any previous works that infer turning restrictions from Floating Car Data.

In the remaining chapters we are going to present details about the individual contributions of this dissertation, starting from single-pair shortest-path queries and our engineering improvements of the ALT algorithm.

Chapter 3

Optimizing Landmark-Based Routing and Preprocessing

Many acceleration techniques exist for the single-pair shortest path problem on road networks. Most of them have been significantly improved over the years to achieve faster preprocessing times and superior performance. In this spirit, this chapter focuses on significantly improving the classic ALT (A* + Landmarks + Triangle equality) algorithm. By carefully optimizing both preprocessing and query phases, we managed to effectively minimize preprocessing time to a few seconds, making the ALT algorithm also suitable for dynamic scenarios, i.e., road networks with changing edge weights due to traffic updates. We also accelerated the query phase for both unidirectional and bidirectional versions of the ALT algorithm, providing fast enough query times (including full-path unpacking) suitable for real-time services and continental road networks. This chapter's content was initially presented on our [43] publication accepted in 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science, held in conjunction with the ACM SIGSPATIAL GIS 2013.

3.1 Introduction

Over the years there has been a great deal of research in finding point-to-point shortest paths in road networks. Although the classic Dijkstra algorithm [34] solves the single pair shortest path (SPSP) problem of finding an exact shortest path of length $d(s,t)$ between a given source s and target t in a graph $G = (V, E, w)$, it still requires a few seconds in continental-sized road networks. Faster alternative algorithms use a two-stage approach: preprocessing requires a few minutes (or hours) and produces a (linear) amount of additional data that is used to accelerate shortest path queries.

Existing methods for solving the SPSP problem in road networks may be classified to three major categories (see [10] for the most recent overview). *Hierarchical Approaches* such as Transit Node Routing (TNR) [9], Contraction Hierarchies (CH) [49] or Hub-based Labeling algorithm (HL) [1, 2] exploit the inherent hierarchical structure of the given road network and build a *search Graph* which includes *shortcuts*, i.e., additional edges connecting important nodes (those participating in many SP queries). In contrast, *goal direction techniques* such as ALT [50] and Arc-flags [72, 86] direct the search towards the target by preferring edges that shorten the distance to the goal node and ignoring edges that cannot possibly belong to the shortest path based on their preprocessed data. A third category is based on *graph separators* such as HiTi [65] and Customizable Route

Planning (CRP) [24, 30]. During preprocessing, one computes a multilevel partition of the graph to create a series of interconnected overlay graphs. A query starts at the lowest (local) level and moves to higher (global) levels as it progresses.

Many of those acceleration techniques have been significantly improved over the years. State-of-the-art methods, such as HL and TNR originally required preprocessing times of more than a few hours. Later works [2], [7] lowered those preprocessing times to just a few minutes. Unfortunately those preprocessing times are still not fast enough for real-time mapping services, where edge weights change frequently due to traffic updates (every 15 - 30 minutes). Additionally, since they are based on Contraction Hierarchies (CH), whenever edge weights change new shortcuts must be added and others must be removed from the search graph, altering the search graph's entire structure, making path unpacking harder and slower to implement. Additionally, CH is very sensitive to the metric used, making the preprocessing significantly slower for (i) travel distances (ii) turn restrictions [24]. That is why Bing Maps uses CRP which might be orders of magnitude slower than HL or TNR, but has faster preprocessing times (few secs) and uses always the same shortcuts regardless of the metric used [30, 24].

This chapter aims at significantly improving the performance of the ALT (A^* + Landmarks + Triangle equality) algorithm [50] and making it suitable for a dynamic navigation scenario. We focus our attention on the ALT algorithm, since it has none of the aforementioned disadvantages of hierarchical methods, i.e., (i) it is very robust with respect to the metric used [50] (ii) it requires no path unpacking (producing the actual road network path of the shortest route) (iii) its storage requirements and auxiliary data structure size depend solely on the number of landmarks (and not on the utilized metric) and most importantly (iv) the typical landmarks-based preprocessing may also be used for estimating the graph distance between any two vertices. For that reason, it has been used in other contexts outside road networks (such as social networks) in cases when the actual distance between two nodes can be sufficiently replaced by the close estimation provided by the typical landmarks preprocessing ([97],[115]).

Our specific contributions aim to improve (i) ALT's preprocessing time and (ii) its shortest-path query performance. By proposing a novel, simple, yet efficient landmark selection strategy and exploiting several optimization strategies, we managed *to lower the preprocessing time from several minutes [28] to just a few seconds*. Moreover, we also improved ALT's *query phase* and *tripled unidirectional ALT performance* while also *improving bidirectional performance by 44%*. Although we did not alter the actual algorithm, our efforts significantly broadened ALT's entire scope, since: (i) its preprocessing is now fast enough for supporting dynamic road networks with frequent traffic updates (ii) ALT algorithm is now fast enough to support real-time SP queries for global scale mapping services. The efficiency and performance of our approach is already demonstrated in our live-system protorype [38] of the SimpleFleet [109] project, that will be presented in Chapter 7 and uses live-traffic information updated every 5 minutes.

In addition, we also filled a gap in ALT's bibliography, since to the best of our knowledge there was no previous work examining its performance for varying number of landmarks and the travel distances metric. By documenting those experiments we get an additional insight of the performance characteristics of the algorithm.

The outline of this chapter is as follows. Section 3.2 describes our scientific contribution beyond the current state-of-the-art in terms of ALT's preprocessing and performance. Experiments establishing the superiority of our approach are provided in in Section 3.3. Finally, Section 3.4 gives conclusions and directions for future work.

3.2 Supercharging ALT

In this section we are going to describe in detail the various optimizations and techniques we used during the preprocessing and SP query phases of the ALT algorithm, to dramatically reduce preprocessing time and achieve superior performance, compared to previous approaches.

3.2.1 Preprocessing

The preprocessing stage for the ALT algorithm is divided in two phases, the landmarks selection process and the computation of distances of all other graph vertices from and to the landmarks. Most of the preprocessing time is dominated by the landmark selection process, which usually is done by sophisticated algorithms such as Avoid and MaxCover [50]. Unfortunately for continental road network graphs and $|S| > 16$, the MaxCover heuristic is not longer applicable due to its high memory requirements [28]. Therefore it is obvious we need a simpler and faster landmark selection strategy that will also provide comparable performance.

3.2.1.1 Landmark Selection

We propose a novel and straightforward strategy for selecting landmarks. We partition the graph (using a partitioning tool) into cells and from each cell we select the four cornermost vertices (top, bottom, left, right according to their coordinates) as landmarks. So, if we are going to use 32 landmarks we partition the graph into 8 cells and get the 4 cornermost vertices per cell. If for example, the top node coincides with the leftmost node for a particular cell, we take the second best in one of those directions. Our new landmark selection strategy will be denoted hereafter as the *partition - corners* method.

On a side note, the partitioning of the graph should not considered part of the actual preprocessing, since it is metric independent and happens only once even for dynamic scenarios (as previous works [24] suggest). After partitioning the graph and efficiently storing the cell of each node, the selection of landmarks actually takes less than 1-2 sec, since it only requires a linear sweep in the vertex information vector of size $|V|$.

At first glance, our *partition - corners* selection strategy seems naive but it has many important advantages: (i) It is extremely fast (ii) It ensures that landmarks are uniformly distributed within the graph (iii) The acquired landmarks may accelerate even local SP queries (between nodes belonging to the same cell). Still, since there is no quality guarantee for the selected landmarks, during the shortest-path query phase we do not use the *Active landmarks* optimization (see Section 2.2.1) used in earlier works. This way, all available landmarks participate in every query, which compensates for their supposed "lower" quality (see 3.2.2 for details).

In terms of partitioning tools, we used the state-of-the-art partitioning tool Buffoon / KaFFPa [106], which was kindly provided to us by its authors. Buffoon / KaFFPa creates far better quality partitions (fewer border nodes) than its predecessor METIS [68] which was used many times before, in the context of SP computation ([46] and [47]). Still, the actual quality of the partitioning plays very little role in our landmark selection process, since we are not interested in minimizing the number of border nodes. Therefore, our approach will work with any partitioning tool.

3.2.1.2 Landmark Distances Calculation

During the second phase of the ALT algorithm preprocessing, we need to calculate distances of all graph vertices to and from the landmarks. Note that is very important to accelerate this particular preprocessing phase, since to adapt ALT to a dynamic scenario (where edge weights change frequently due to traffic updates), this second phase has to run at every batched traffic update. On the contrary, the landmark selection phase has to run only once even for dynamic graphs, since all previous approaches [28] assume using static landmarks, i.e., they do not reposition landmarks if the graph weights are altered.

In order to calculate distances of all graph vertices to and from the landmarks, we need to run two Dijktra algorithms from each landmark, one that runs in the forward graph and one that runs in the reverse graph, for a total of $2|S|$ Dijkstra searches. Since each Dijkstra search is independent from the others, this process may be easily parallelized. Still, this is not good enough if we want to provide preprocessing times of less than a minute. We also need to accelerate each of those individual Dijkstra searches. For that purpose, we applied the following four optimizations:

Dijkstra Heaps. A Dijkstra implementation with a heap structure that only supports Insert and Delete-Min operations (without a Decrease-Key operation), hereafter referred to as *Dijkstra - NoDec*), performs more heap operations and is theoretically inferior to the asymptotic running time of Dijkstra implementations with decrease-key (denoted as *Dijkstra - Dec*). However, previous works have shown that such streamlined heaps are likely to be more efficient. In fact [19] has shown all *Dijkstra - NoDec* implementations for various graphs (including road networks) are at least 1.4 times faster than their Dijkstra-Dec counterparts. This improved performance was also evident in our experiments.

Priority Queue Optimization. Instead of using binary heaps for our priority queue implementation, we used the *aligned 4-ary heap* which is a highly-optimized heap for cache memory implemented by Sanders [103]. This array-based heap aligns its data to cache blocks, which in turn reduces the number of cache-misses when accessing any data item. The Sanders implementation we used, supports Insert and Delete-Min operations in $O(log4N)$ time and block transfers each. We were able to use such an implementation, only after we employed the previous *Dijkstra - NoDec* optimization.

Although buckets-based priority queues are shown to perform even better for the Dijkstra algorithm [19],[23], since we are going to run multiple Dijkstra searches in parallel, we did not want to use such memory intensive data structures whose efficiency and size needed depends on the smallest and largest edge weight of the graph. Our experiments have shown that indeed our *aligned 4-ary heap* Dijkstra - NoDec implementation is very fast and memory efficient at the same time and scales pretty well for multicore processors.

Node Reordering. Delling et al. report [23] that Dijkstra's performance improves significantly, if we reorder the vertices so that neighboring vertices have similar IDs, in order to reduce cache misses during computation. Based on [104], they also show that a simple depth first search layout, i.e., reordering the vertices according to a simple depth first search (DFS) improves Dijkstra's speed by 2.8 times. Following those observations, we initially reordered vertices ID according to a DFS layout and then we reordered vertices again, using the partition obtained during the landmark selection process, so that nodes within the same cell are assigned consecutive nodeIDs, as suggested by [47].

Keep in mind that the finalized nodes ordering (DFS layout + partition) not only lowers preprocessing times but it additionally accelerates the SP query phase of the ALT algorithm, as evidenced by our experiments (see Sec. 3.3).

Graph Data Management. Typically when we want to run bidirectional SP queries

on a graph, we use a compact modified adjacency array [85] representation of both forward and reverse graphs, which stores two additional bits per edge in order to separate incoming from outgoing edges per vertex. Although storing the forward and reverse graphs together as a single adjacency array representation is very memory efficient, our experiments have shown that storing forward and reverse graphs separately is significantly faster during preprocessing. Consequently, since forward and reverse graphs are stored separately, during preprocessing we first run all the Dijkstra algorithms from each landmark in the forward graph and once we are done we run the same Dijkstra searches in the reverse graph. That way, the parallel threads only operate on one of the two adjacency arrays graph structures, which makes the entire process significantly faster.

Storing the two graphs separately, also accelerates the SP query phase of the ALT algorithm, especially for the unidirectional ALT which runs only in the forward graph. Although storing separately the forward and reverse graphs requires almost double the main memory, the corresponding graph data structures will always be much smaller than the main memory required for storing the landmarks distances. Therefore, it has no impact on the scalability of the ALT algorithm for larger networks.

Conclusively, our experiments (see Sec. 3.3) showed that by using those four optimizations described earlier, for 32 landmarks and the benchmark continental road network of Western Europe, even our *sequential* calculation of vertex distances from and to landmarks is 3 times faster than previously best published landmarks paper [28] in terms of preprocessing. If we parallelize the process, it takes merely 30s on a commodity workstation, which makes landmarks competitive in terms of preprocessing with the fastest (in terms of preprocessing) CRP acceleration technique. As a result, the ALT algorithm may now be used in dynamic road networks with frequent traffic updates as well.

3.2.2 Shortest-Path Querying

All of the previous preprocessing optimizations (namely: Avoiding decrease-key operations, the aligned 4-ary heap, nodes reordering and storing forward and reverse graphs separately) have also a positive impact on the SP query phase of the ALT algorithm. Still we can do even better. So, we applied 3 additional optimizations to the SP query phase of the ALT algorithm:

Active Landmarks Purging. Previous landmark approaches [51] used the active landmarks optimization (see Sec. 2.2.1), i.e., they use a subset of the available landmarks during the query phase, which is updated dynamically during the search. This optimization has the disadvantage of requiring to dynamically update the priority queues during the search. Moreover, the ALT algorithm cannot longer terminate as soon as the two opposing searches meet. We dropped this optimization entirely and during the search we get lower bounds based on all available landmarks. This lowers the number of settled nodes (especially for the unidirectional version of ALT), without imposing an unbearable burden on the computation cost.

Keep in mind that by using all available landmarks we more than compensate for their supposed "lower" quality, due to our simple *partition - corners* strategy (see Sec. 3.2.1.1). Still, since we use all available landmarks for calculating lower bounds, we need to significantly accelerate the process, which can be done with our next two optimizations:

Landmark Distance Records. Similar to [28], we store landmarks distances in a 32-bit vector of size $2 \cdot |S| \cdot |V|$. Distance of node with nodeID $i \in [0, |V| - 1]$ from landmark number $j \in [0, |S| - 1]$ is stored at position $2 \cdot |S| \cdot i + 2 \cdot j$ and the distance of node i

to the landmark j is stored in the next position $(2 \cdot |S| \cdot i + 2 \cdot j + 1)$. But what we do entirely differently, is that we store landmarks distances *from landmarks to nodes* negated (as negatives), because this is how they are going to be used during estimation of lower bounds (for the forward search).

In the case of a bidirectional search, at its beginning we cache *the opposite of landmark distances of start and goal node* in two separate vectors of size $2|S|$ (denoted hereafter as the *opposite-StartNodeVector* and *oppositeGoalNodeVector*). As a result, at each node expansion $\pi_f = max(nodeVector + oppositeGoalNodeVector)$ and $\pi_r = -min(nodeVector + oppositeStartNodeVector)$. By storing the opposite of landmarks distances from landmarks in the aforementioned 32-bit vector, we avoid unnecessary additive inversions during the calculation of lower bounds, which makes calculation faster and prepares the ground for our next optimization.

SSE Instructions Current x86-CPUs have special 128-bit SSE registers that hold four 32-bit integers and allow basic operations, such as addition, minimum and maximum to be executed in parallel. By using these 128-bit registers we can significantly accelerate the computation of the lower bounds $\pi_f = max(nodeVector + oppositeGoalNodeVector)$ and $\pi_r = -min(nodeVector + oppositeStartNodeVector)$ computation. This optimization alone gives a solid 10-20% improvement. Although [23] has used SSE instructions for accelerating SP computation from multiple sources, to the best of our knowledge we are the first that utilize this optimization within a single source SP computation. Moreover, latest Intel Haswell processors already possess 256-bit registers (512-bits registers are in the works) and as a result, this optimization will be even more efficient in the near future.

By all those optimizations, with bidirectional ALT we can achieve SP query times with 48 landmarks, better than those previously reported for 64 landmarks. Moreover, we managed to triple unidirectional ALT performance, as will be shown in the next section.

3.3 Experiments

The experimentation that follows, assesses the performance of our optimizations for the preprocessing and query phases of unidirectional and bidirectional versions of the ALT algorithm for varying number of landmarks.

Experiments were performed on a workstation with a four-core Intel Core i7 processor clocked at 3.4GHz and 32Gb of main memory, running Ubuntu 12.10 64bit. Our code was written in C++ and was compiled with GCC 4.7 and using optimization level 3. We used OpenMP for parallelization. Although the preprocessing stage used all four cores (with hyperthreading), SP queries used only one core for accurate benchmarking. We used the strongly connected component of the European road network with 18 million nodes and 42 million arcs made available by PTV AG for the 9th DIMACS Implementation Challenge [33]. Both nodeIDs and edge weights are 32-bit integers. We experimented with both travel times and travel distances.

3.3.1 Travel times

We compare our approach with the previously best (in terms of efficiency and performance) published ALT paper [28]. During their experiments, they used a slower workstation than ours (dual AMD Opteron 252 at 2.6 GHz with 16 Gb of RAM). Their codebase was also in C++ and was compiled with GCC 4.1, using optimization level 3. In contrast to our approach, no parallelization was used for preprocessing. Their experiments were

Table 3.1: *ALT's preprocessing time for varying number of landmarks, in comparison to [28] (travel times)*

	[28]	OURS	[28]	OURS
Algorithm	**time (s)**	**time (s)**	**dist (s)**	**dist (s)**
ALT-8	1566 (1205)	8	168 (128)	7
ALT-16	5112 (3932)	16	330 (254)	15
ALT-24		23		22
ALT-32	1626 (1251)	31	666 (512)	30
ALT-40		38		37
ALT-48		46		45
ALT-56		55		53
ALT-64	4092 (3148)	60	1326 (1020)	58

Table 3.2: *ALT's query performance for varying number of landmarks, in comparison to [28] (travel times)*

	QUERY UNIDIR.				QUERY BIDIR			
	[28]	OURS	[28]	OURS	[28]	OURS	[28]	OURS
Algorithm	**# settled nodes**	**# settled nodes**	**time (ms)**	**time (ms)**	**# settled nodes**	**# settled nodes**	**time (ms)**	**time (ms)**
ALT-8	1,019.843	1,140,887	391.6 (301)	175.3	163,776	465,503	127.8 (98.3)	115.7
ALT-16	815,639	804,663	327,6 (252)	124.0	74,669	248,247	53.6 (41.2)	60.9
ALT-24		677,446		110.2		134,315		35.3
ALT-32	683,566	506,805	301.4 (232)	85.9	40,945	74,423	29.4 (22.4)	17.5
ALT-40		449,259		81.0		53,410		15.2
ALT-48		430,389		78.4		48,499		12.4
ALT-56		400,483		71.7		38,140		10.8
ALT-64	604,698	385,322	288.5 (221)	70.6	25,324	36,607	19.4 (14.8)	10.3

based on the same benchmark European road network. The results (as well as ours) are based on 10,000 random $s - t$ queries. For an accurate comparison we present their originally recorded times and the same times divided by a of factor of 1.31 (difference between our processors' clock speeds). Their experiments were done for 8, 16, 32, 64 landmarks. We experimented with 8-64 landmarks, at steps of 8.

Results are presented in Tables 3.1 and 3.2. Regarding preprocessing time, the column "time" refers to total preprocessing time (landmark selection + calculating landmark distances) and the column "dist" refers to the time required strictly for calculating landmark distances. For [28], the numbers in parentheses represent the simulated times, which are the quotients of the original times divided by 1.31.

Results for preprocessing clearly show the inferior performance of previous methods. We use a simpler landmark-selection strategy that requires merely 1-2s instead of previous time-consuming and complicated strategies. Moreover, our approach is superior, even for the preprocessing time required for updating the landmarks distances. Even if we divide the simulated times of [28] by 5.6 (the typical parallel speedup encountered on our 4-core processor with hyperthreading), our approach is still consistently 3 times faster. It is therefore obvious that our various optimizations for preprocessing have really paid off and as a result, the improved preprocessing time always remains consistently below 1min.

In terms of unidirectional queries, we see that unidirectional ALT is now 3 times faster but also settles fewer nodes. This fact is a clear indication that the active landmarks

optimization does not work for unidirectional queries and thus, dropping it, was the right choice. By using all the available landmarks, we can easily achieve query times of less than 72*ms* and the unidirectional ALT scales better for increasing number of landmarks.

In the case of bidirectional queries, for $|S| \leq 16$, the lower quality of our selected landmarks comes into play and our method settles more nodes and is slightly slower than [28]. On the contrary, for $|S| \geq 24$, results are entirely different. Our method, due to its aggressive optimizations, is consistently faster by 4-5*ms* from the simulated query times of [28], which constitutes an average improvement of 22-44%. By using 48 landmarks, we get even more improved query times than those previously achieved with 64 landmarks. For $|S| \geq 48$ we are able to achieve query times $\leq 12ms$, which means that bidirectional ALT is now capable of handling real-time SP queries, since contrary to hierarchical methods, it does not require extra time for returning full paths (path unpacking).

3.3.2 Travel distances

We also repeated the same experiments, using travel distances for the same road network. This effort was necessary to cover a significant gap in the ALT's large bibliography, since (to the best of our knowledge) there is not some previous work demonstrating the performance of ALT algorithm for travel distances, $|S| > 16$ and varying number of landmarks. Table 3.3 presents our results. We use the exact same landmarks as before.

Table 3.3: *Performance of ALT for varying number of landmarks and travel distances metric*

	PREPROCESS.		**QUERY UNIDIR.**		**QUERY BIDIR**	
Algorithm	**time (s)**	**dist (s)**	**# settled nodes**	**time (ms)**	**# settled nodes**	**time (ms)**
ALT-8	6	5	1,176,419	170.8	1,165,631	228.1
ALT-16	14	13	682,947	101.1	604,168	127.7
ALT-24	20	19	511,786	91.3	317,227	88.7
ALT-32	26	25	348,060	59.7	160,836	45.1
ALT-40	33	32	319,109	53.0	142,400	39.1
ALT-48	38	37	294,548	48.7	123,952	32.8
ALT-56	44	43	278,579	44.8	112,515	30.0
ALT-64	51	48	264,516	44.5	101,957	29.1

Results for travel distances are exactly what we expected. Preprocessing is *15-22% faster*, since the individual Dijkstra searches typically perform better for travel distances. After a node is encountered for the first time, it is less frequent for further expansions to improve its cost. That is after all one of the advantages of the ALT algorithm in comparison to hierarchical methods, i.e., its preprocessing is faster for travel distances, whereas methods such as CH require 7 times more preprocessing time for the travel distances[24] when compared to the travel times.

We also see that unidirectional ALT is now almost competitive with bidirectional ALT, both in SP query times and number of settled nodes. This was something to be expected, since previous works [50] has recorded the quite similar efficiency of both methods when travel distances were used. The interesting fact though is, that *unidirectional ALT is now faster for travel distances than travel times* similarly to plain Dijkstra. To the best of our knowledge, we are the first to pinpoint this very interesting fact.

Moreover, since ALT is very robust with respect to different metrics [12], switching to travel distances only makes bidirectional ALT 2-3 times slower, when, hierarchical meth-

ods become at least one order of magnitude slower. Because of its robustness, the ALT algorithm has been successfully used for other kinds of graphs, such as social networks [97, 115], where most hierarchical, road-network oriented methods would fail.

3.4 Conclusion and Future Work

In this chapter, we have significantly improved the classic ALT algorithm, both in terms of preprocessing time and shortest-path query performance. Our improvements were considerable: We lowered preprocessing times to $< 1min$ (a total of 40-52 times improvement) in comparison to previous published works and also tripled unidirectional ALT SP query performance and improved bidirectional ALT performance up to 44%. Our efforts significantly altered the ALT's scope since (i) its preprocessing is now fast enough for *supporting dynamic road-networks with frequent traffic updates* and (ii) the ALT algorithm may now support *real-time SP queries* for global-scale mapping services.

As shown by previous works, for real-world services we do not always use the fastest algorithm but the most practical one. The ALT algorithm already has several excellent qualities. Robustness to the metric used, the ability to return full paths, robustness to the graph density and stable auxiliary data memory size for all metrics. Through our efforts, the ALT algorithm has now, and what was missing, *practical preprocessing times and fast enough performance for real-world mapping services*. The efficiency and performance of our approach is already demonstrated in a live-system prototype [38] addressing fleet management needs. Given this effort, the ALT algorithm is now ready for practical use.

We can give the following directions for future work. Now that ALT has been significantly improved, it would be easy to combine it with other fast preprocessing methods for road networks, like CRP, to further boost SP query performance, without a significant increase in preprocessing times. Such a combination, will be presented in Chapter 6. Moreover, since ALT has been used in other contexts outside road networks, it would be interesting to show how our method performs for other kind of graphs as well. But most of all, we hope to encourage more researchers to add the ALT algorithm to their arsenal, in the context of practical real-world applications.

Chapter 4

Crowdsourcing Computing Resources for Shortest-Path Computation

Crowdsourcing road network data, i.e., involving users to collect data including the detection and assessment of changes to the road network graph, poses a challenge to shortest-path algorithms that rely on preprocessing. Hence, current research challenges lie with improving performance by adequately balancing preprocessing with respect to fast changing road networks. In this chapter, we take the crowdsourcing approach further in that we solicit the help of users not only for data collection but also to provide us their computing resources. A promising approach is parallelization, which splits the graph into chunks of data that may be processed separately. This chapter of the dissertation extends this approach in that small-enough chunks allow us to use browser-based computing to solve the pre-computation problem. Essentially, we aim for a Web-based navigation service that whenever users request a route, the service uses their browsers for partially preprocessing a large but changing road network. This work presents performance studies that highlight the potential of the browser as a computing platform and showcases a scalable approach, which almost eliminates the computing load on the server. This chapter's content is mostly based on the diploma thesis [113] of Dimitris Theodorakis co-supervised by the dissertation author and our [47] publication, which was awarded best poster award in ACM SIGSPATIAL 2012.

4.1 Introduction

Crowdsourcing is a process that outsources tasks to a distributed group of people. It affects the present context in two ways. For once, the focus of this chapter is how to deal with constantly changing road networks in the context of shortest-path (SP) computation, and, secondly, on how to utilize the computing resources of the crowd during the process.

The single-pair shortest path (SPSP) problem of finding an exact shortest path of length $d(s,t)$ between a source s and target t in a graph $G = (V, E, w)$ is addressed by the classic Dijkstra algorithm [34], which unfortunately requires few seconds on continental sized road networks. More efficient algorithms involve a *preprocessing stage*, which produces a (linear) amount of auxiliary data that is then used to accelerate SP queries. While many effective techniques exist, an important category of SP algorithms is based on *graph separators* (GS). The most prominent example of this category is Customizable Route Planning (CRP) [24, 30]. During preprocessing, a multilevel partition of the graph is computed, in order to create a series of interconnected overlay graphs. A

query starts at the lowest (local) level and moves to higher (global) levels as it progresses. When considering frequent updates to the road network (e.g., OpenStreetMap), GS methods have the advantage that changes remain local and therefore have limited impact on the overall shortest-path computation. Another advantage of GS approaches is that their *preprocessing may be easily parallelized*, since during preprocessing each thread needs to have access only to a limited portion of the road network. Hence, CRP significantly reduced its preprocessing time, when tested on a typical multicore workstation.

The objective of this chapter is to embrace change and trivialize the preprocessing stage. By defining an architecture that would allow us to use Web-browsers as a computing platform, we effectively delegate the task of incorporating the change back to the user and crowdsource not only the updates to our dataset but also the respective computing resources needed to deal with the consequences of such a change. Our system architecture is able to distribute typical GS preprocessing to an unlimited number of Web-clients. All preprocessing takes place on the clients, with the server only distributing data and gathering results from clients. What is more impressive, instead of setting up dozens of clients running custom OSes, we use Web-browsers and Javascript as our computing platform. To the best of our knowledge, this is the first work that advocates such an approach. Our extensive experimentation will show that even with a limited number of clients, we require significantly less preprocessing time than most SP algorithms and still are able to answer SP queries on continental road networks in almost 2ms.

The outline of this chapter is as follows. Section 4.2 describes related work in the context of distributed shortest-path computation and browser-based computing. Section 4.3 describes our system architecture and design. Experiments establishing the performance characteristics of our system are given in Section 4.4. Finally, Section 4.5 gives conclusions and directions for future work.

4.2 Related Work

In this section we will mainly focus on additional information missing from Chapter 2 and we will focus on the subjects of distributed shortest-path computation and using Javascript as a computing platform. In detail:

4.2.1 Distributed shortest-path computation

Separating SP preprocessing and actual SP queries, so that those two processes run on different devices is not by any means a novel concept. Up until now, the typical approach was to execute preprocessing for the road network on a quite powerful workstation and create there the necessary auxiliary data, so that a less powerful mobile device may execute SP queries locally without the need of a separate server. Examples of this approach are in chronological order: [51], [104] and [80]. All those approaches work in the aforementioned way: A preprocessing algorithm runs on a typical workstation (ALT [50] for [51] and Contraction Hierarchies for [104] and [80]) and then the created auxiliary data is stored on the secondary storage of the mobile device (PDAs or mobile phones), so that the mobile device may run SP queries locally. Our approach completely inverses this typical approach, since preprocessing now takes place on the client's browser (which may be a mobile device's browser as well) and SP queries will then run on the server.

In the case of distributed SP preprocessing, the only work we are aware of is [71]. In that work, Kieriz et al. distribute the SP preprocessing (and SP queries as well) of

Time Dependent Contraction Hierarchies [11] on their dedicated cluster of 128 nodes, each equipped with two 2.667GHz Quad-Core Intel Xeon X5355 processors and 16Gb of RAM, connected by an InfiniBand 4xDDR switch. Their implementation is based on Message Passing Interface (MPI) which is the de-facto standard for running parallel applications on distributed memory systems. For the same benchmark Europe road network we used, they managed to achieve a speedup of 45 for up to 64 processors. Unfortunately, memory requirements for more than 32 nodes, are above 2Gb on each node and progressively increase to more than 4Gb for 128 nodes. Although this work impressively succeeds in parallelizing an algorithm which is not inherently parallelizable such as Contraction Hierarchies, it requires the use of a dedicated cluster of powerful workstations connected to a solid network infrastructure, which is not always feasible or cost-effective. On the contrary, our experiments will show that our distributed browser-based approach *runs on pure commodity hardware* (even our server may run on a commodity workstation) requiring absolutely no supporting computing infrastructure and is *easy to set-up within minutes*. Therefore, it may be used by anyone without any budget constraints.

Another fundamental difference between previous approaches such as [71] and ours is that our method does not only distribute computation load to nodes (actually web-clients) prepared by us, but it also *exploits the computing resources of potential clients that will use our service* in a pure crowdsourcing context. Hence, we move from a plain distribution of computation load to multiple nodes to actually crowdsourcing computation load on large numbers of potential users. In that sense, our system is infinitely scalable.

4.2.2 Javascript as a computing platform

Throughout this work, all SP preprocessing takes place on web-clients and therefore we use Javascript as our computing platform. During the last years, Javascript's popularity is continuously increasing. Major browser vendors are constantly improving their Javascript engines to provide fast execution with advanced techniques such as just-in-time (JIT) compilation and efficient garbage collection [70]. JavaScript is now among the first 10 most popular programming languages [95] and various projects, such as Lively Kernel [62] or Node.js [114] leverage JavaScript for advanced uses beyond adding simple interactivity to Web pages. To the best of our knowledge, our work is *the first that showcases the usage of Javascript for parallel SP specific preprocessing*. We will also demonstrate that Javascript is fast enough for our purposes and therefore we hope to encourage other researchers to use it for significantly parallelizing their computations by sharing computation resources with Web-clients.

4.3 Crowdsourcing shortest-path preprocessing

The main contribution of this work is to showcase a system, which efficiently distributes typical graph-separator SP preprocessing to multiple web-clients. This section provides the architecture details of the implemented system and we will describe all the necessary optimizations that had to be added to the typical GS preprocessing for adapting it to the distributed nature of such a system.

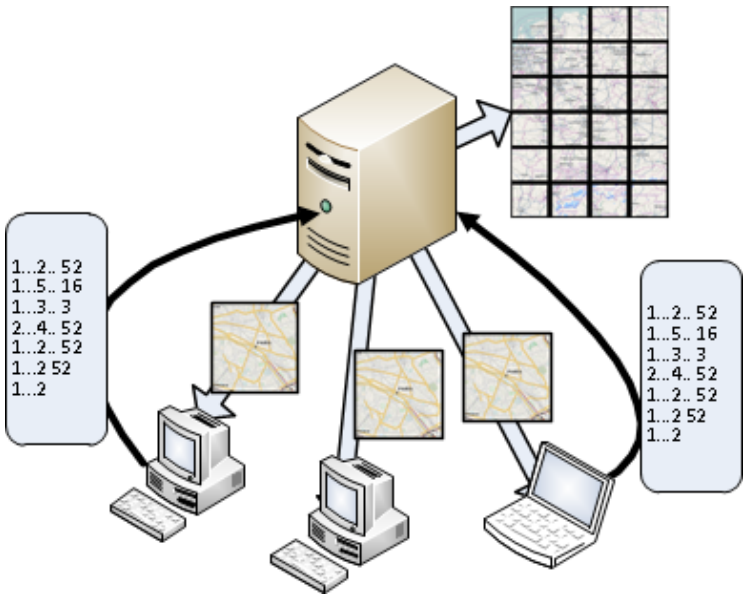


Figure 4.1: *System architecture. The server sends the cells' graph to the web-clients and web-clients return results to the server, once they finish clique calculation*

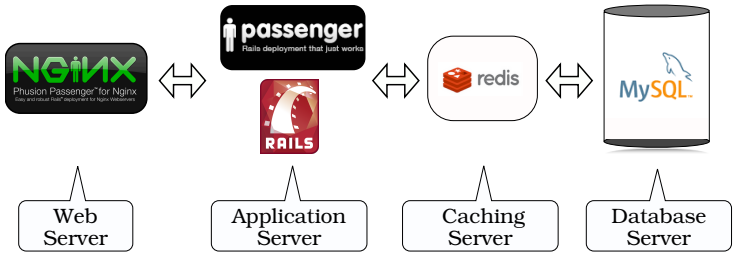


Figure 4.2: *Server architecture*

4.3.1 System architecture

We deliberately tried to keep our system architecture as basic and straightforward as possible. We have a server, which keeps the road network graph and whenever a new client (browser) connects, the server sends one or more cells (actually the cell's graph restricted to inner arcs of the cell) to the browser for processing. When the client finishes calculating the clique for the specific cell(s) assigned to it, it returns the results to the server through AJAX. The server always keeps track of the either calculated or already assigned cells, so that when the next client connects to the server it is assigned some unprocessed and unassigned cells for clique calculation (Fig. 4.1).

Regarding the server, the individual cells' graphs are stored in a database. We also use a key-value store as a caching layer, so cells are already preloaded in the cache (as JSON objects) to minimize accesses to the database server. We additionally use the combination of a web/application server, which receives requests from clients, assigns cells to them and gathers their results. Results are also stored on the caching layer for added speed and efficiency and are moved offline to the database for permanent storage, if needed. For a system with live-traffic updates, results will never have to be stored permanently on the database layer, since they are valid for a very limited time (usually 15-30 min). The server architecture is shown on Figure 4.2. For simplicity reasons, all the separate components of our server architecture existed on the same virtual machine.

As far as technical details are concerned, we intentionally and exclusively used free /

open-source tools for our server implementation. Accordingly, we used MySQL [88] as the back-end database server, REDIS key-value store as the caching layer and the combination of Nginx [89] and Phusion Passenger [94] for our web server - application server combination. The server application was written using the Ruby-on-Rails framework. For additional details, refer to Section 4.4.1

4.3.2 Adapting road network preprocessing

This section describes the typical graph-separator SP preprocessing and all the necessary optimizations we added for its efficient adaptation to a crowdsourcing context.

Initially, to partition the graph G, we used METIS [69], a graph partitioning tool used frequently in the context of shortest-path computation [86, 46]. Since we will use multiple levels of overlay graphs, we create nested partitions by using METIS in a top-down fashion. Similar to previous approaches, partitioning the graph is not considered part of the actual preprocessing process, since (i) it is done only once, and (ii) it is metric independent and will not have to be repeated when arc weights change.

In typical GS preprocessing, as described in CRP, to calculate cliques, we must run a Dijkstra algorithm (restricted to inner arcs of each cell c) from each boundary node of this cell. Clique calculation here is done entirely in the client's browser with Javascript and the server only sends cells to the browser and gathers the results (the clique arcs calculated) from clients once clique calculation is finished. Therefore, to minimize network communication time, we must minimize the amount of data moved between the server and the browser clients. This can be done either by (i) nodes reordering and algorithmic optimizations and (ii) network optimizations and batch grouping of data.

4.3.2.1 Algorithmic optimizations

The first necessary optimization was *nodes reordering*, which is a common optimization technique for SP approaches. Since the cells' graph is sent to the clients in JSON format (a text format compacter than XML), we reordered the nodes, so that nodeIDs within a cell are consecutive. That way we transmit nodeIDs starting from *zero* to *max number of nodes per cell* for all cells. When the server gathers results from the clients, it just adds the minimum nodeID of each cell to all results before storing them. This way, we effectively minimize the cell's graph size sent to the client and the clique arcs size returned from the client to the server, in terms of their respective text (JSON) representation.

An important advantage of GS approaches is that *overlay graphs of higher level partitions may be computed by using the overlay graphs of lower levels* to dramatically reduce preprocessing time [24]. Although this technique indeed reduces significantly computation time on the client, it has another advantage as well: It also reduces network traffic, since overlay graphs within a cell are smaller than the original graph for the same cell. Experiments will show (see Section 4.4.2.3) that this technique is very beneficial for minimizing, both, computation and network communication time.

Another important change is, that contrary to CRP which uses an adjacency matrix representation of cliques for query efficiency, we cannot send results of clique calculation over the network in such a wasteful format. Therefore, we resort to a standard adjacency list representation of clique arcs. Additionally, in order to minimize the number of clique arcs sent from the client to the server, we report only distances of boundary nodes that are direct descendants of the root of each Dijkstra algorithm we run. This specific *arc-reduction optimization* preserves correct distances between boundary nodes on the overlay

graph, has no negative impact on computation time and leads to a 56 – 71% reduction of the number of clique arcs created (depending on the cell size). Moreover, this optimization also significantly reduces the result sizes sent to the server. Such optimizations are crucial on the client, since most Web servers support GZIP compression, but browsers cannot send back compressed JSON, at least not in a trivial way.

By using those algorithmic optimizations, we are able to effectively minimize the size of the data exchanged between the server and the clients to below 300KB in the worst case, a reasonable size for AJAX requests.

4.3.2.2 Network optimizations

Our experimentation showed that even with Javascript, graph-separator preprocessing is quite fast. Therefore the most important bottleneck is the network communication cost between the server and the Web-clients. Although nodes reordering and the algorithmic optimizations described in Section 4.3.2.1 reduce the overall size of cells and the number of clique arcs, we can further reduce network traffic by *batch grouping cells and results*. Instead of the server sending a single cell to each client and then collecting a single result, the server now sends more cells (in a batch) to each client. The number of batched cells was chosen carefully for each network partitioning, so that HTTP responses and requests do no exceed 300KB, our self-imposed limit. Moreover, we used JSON as our exchange format and enabled GZIP compression in our Nginx web server. All the aforementioned network optimizations significantly reduced network communication time, which was necessary for a successful adaptation of typical GS preprocessing in a crowdsourcing context. Note, that our solution is essentially piggybacking on another Web service (navigation) to crowdsource computing resources and any noticeable delay would discourage people from using such a service and deprive us from their computing resources.

Consequently, by using all those algorithmic and network optimizations and techniques, both on the server and client side, our experimentation described on Section 4.4 will show that even with a limited number of clients, we can achieve smaller preprocessing times than most other algorithms and still achieve reasonable SP query times (typical of GS approaches) in the range of 1.5-2ms.

4.4 Experiments

The experimentation that follows assesses the performance of our distributed browser-based approach for various partition sizes (number of cells). The experiments will report total time, network communication time and pure computation time for the majority of our experiments and will also document how batch grouping of, both, cells and results improves total performance. Moreover, we have experimented with *1, 2, 4* and *8* Web-clients to demonstrate how our architecture scales with the number of utilized clients. We additionally report memory usage on the client machines and data sizes transmitted over the network. Finally, we show average SP query times to compare our method with state-of-the-art graph-separator techniques.

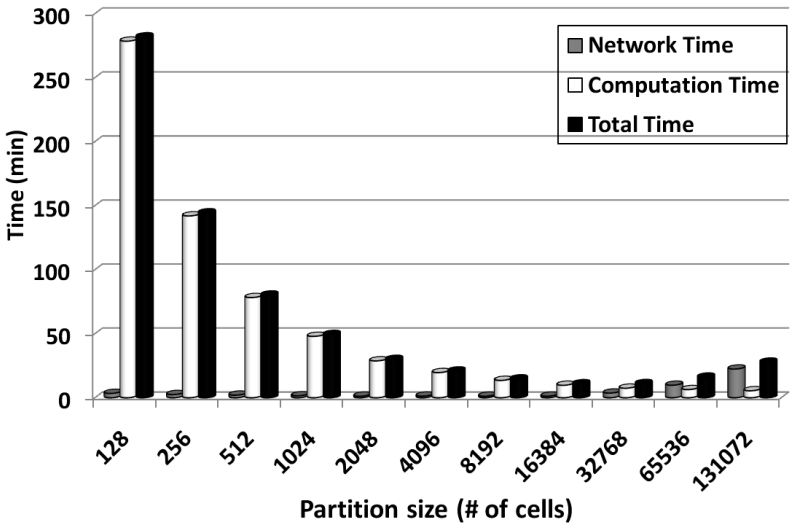


Figure 4.3: *Total, computation and network communication time for various partition sizes, for the whole road network for one web client (Travel-times)*

4.4.1 Experimental setup

Our benchmark road network instance is the strongly connected component of the European road network with 18M nodes and 42M arcs made available by PTV AG for the 9th DIMACS Implementation Challenge [33]. During our experiments, the total number of cells per partition ranged from 128 to 131072 cells per partition.

All our experiments were conducted in a cloud environment using Amazon Web Services (AWS). Our system consists of a total of 5 virtual machines. Four of them are simulating web clients and the latter is the actual server. All of them are Medium High-CPU 64-bit instances according to Amazon's classification and employ 1.7GB of memory, 350GB of local storage and 2 virtual cores with 2.5 EC2 Compute Units each. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0 - 1.2GHz 2007 Opteron or 2007 Xeon processor [4]. Consequently, it is evident that even our server may run on commodity hardware by current standards.

The server VM is running Ubuntu Server 11.10 64-bit, MySQL 5.5.22 as the database server, NGINX 1.0.15 for web serving, Redis 2.2.12 key-value store for caching, Ruby 1.9.3 and Rails 3.0.7 with Passenger 3.0.12 application server. On the other hand, the Web-clients are running Windows Server 2008 R2 Base 64-bit with two Google Chrome processes each, summing up to 8 simultaneous Web-clients.

4.4.2 Overall performance

On our first experiment we *use the entire road network for every overlay graph calculation* for varying partition sizes (number of cells). We used a single Web-client and no batch-grouping of cells. This implementation will serve as the benchmark for all other implementations and optimizations.

Results are presented in Fig. 4.3 and show that without batch grouping of cells or using intermediate overlay graphs, clique calculation of high-level partitions (i.e., partitions with small number of cells), *requires a total preprocessing time of more than 4h*.

Regarding the distribution of total time, for low-level partitions, network communication is the main bottleneck of our distributed browser-based approach. On the contrary,

Table 4.1: *Number of batched cells for varying partition sizes for the whole road network*

# cells per partition	# grouped cells
16384	8
32768	16
65536	32
131072	64

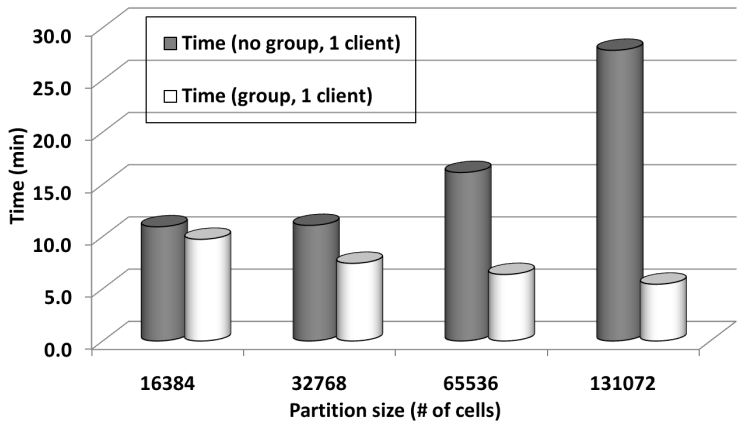


Figure 4.4: *Effect of batch grouping of cells for varying partition sizes and the whole road network (Travel times)*

for high-level and medium-level partitions (i.e., *128 - 16384* cells), only a limited fraction of total time is devoted to network traffic and computation time on the client remains the main bottleneck. Although typically for GS approaches, high-level partitions result in denser overlay graphs (and therefore large number of clique arcs), through our *clique arc-reduction optimization* (cf. Section 4.3.2.1), we significantly reduce the number of clique arcs returned to the server and, therefore, network traffic in high-level partitions is effectively minimized.

4.4.2.1 Batch grouping of cells

To show how batch grouping of cells minimizes network traffic time, we repeated the above experiments for the lower-level partitions (since on those partitions network traffic is the main bottleneck), once again for the whole road network (i.e., not using any intermediate overlay graphs), but this time batching cells so that each group of cells fits within the 300KB implicit limit (see Sec. 4.3.2.2). Results are shown in Table 4.1 and Fig. 4.4.

Table 4.1 shows that for the lowest-level partition used (the *131072* partition) we can batch-group as many as 64 cells within the 300KB limit (per HTTP response from server). Therefore for this particular partition, *through batch grouping of cells we were able to reduce total time from 27.8 min to 7.6 min*, for a total speedup of 3.6. We also see that through batch grouping of cells, lower-level partitions are now faster to calculate than higher level partitions.

4.4.2.2 Multiple Web-clients

Using multiple Web-clients, we repeated the above experiments using the batch-group settings of Table 4.1 for *1, 2, 4* and *8* clients. Results are presented in Figure 4.5. It is evident that our approach scales very well for at least up to 8 clients. For 4 web clients we get a speedup of 3.6 and adding another 4 web clients (for a total of 8) results in an

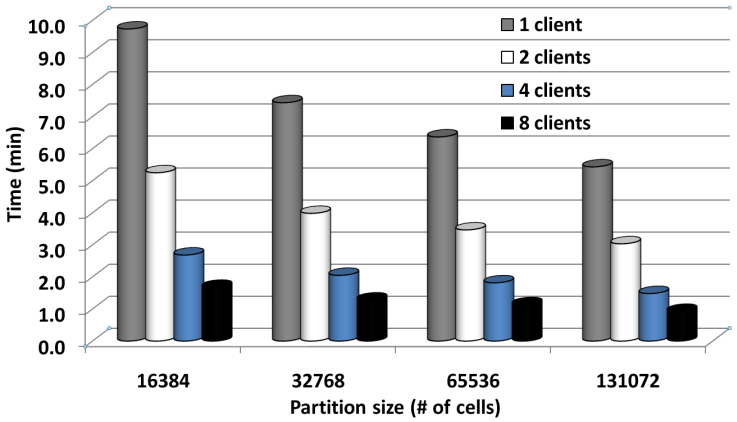


Figure 4.5: *Effect of multiple web clients for varying partition sizes for the whole road network and batch grouping of cells (Travel times)*

Table 4.2: *Number of batched cells for varying partition sizes and helper partitions*

# cells helper partition	# cells per partition	# grouped cells
128	32	1
1024	128	1
1024	256	1
1024	512	1
16384	1024	1
16384	2048	2
16384	4096	4
16384	8192	8
131072	16384	32

total speedup of *5.8* speedup. Combining, both, batch grouping and multiple web clients, we can calculate any low-level partition (up to 16384 cells) clique, in *less than 2 minutes* by using 8 web clients, which is vast improvement over the *27.8 min* worst initial case.

4.4.2.3 Using overlay graphs of lower-level partitions

Now that we have established how batch grouping of cells and multiple Web-clients reduce total time, we use overlay graphs of lower-level partitions (cf. [24]) to calculate overlay graphs of higher-level partitions. For those experiments, we directly used four Web-clients and batched as many cells as possible within the implicit limit of 300KB. We experimented with four intermediate "helper" levels: The 131072 partition overlay graph was used for calculating the overlay graph of the 16384 partition, the 16384 partition was used for the overlay graphs of the 8192, 4096 and 2048 partitions, the 1024 partition was used for the overlay graphs for the 128, 256 and 512 partitions and the 128 partition was used for calculating the highest-level overlay graph of the 32 partition. Results are shown in Table 4.2 and Figure 4.6.

Table 4.2 shows, that even by using "helper" partitions, we can no longer group cells for high-level partitions with less than 2048 cells. Fortunately for all those high-level partitions, network time is less than *13s* for all partition sizes. For the 8192 partition, network time is 45% of total time but this is progressively reduced to 5% for high-level partitions (see Fig. 4.6). Figure 4.6 also shows that *by using "helper partitions", the total*

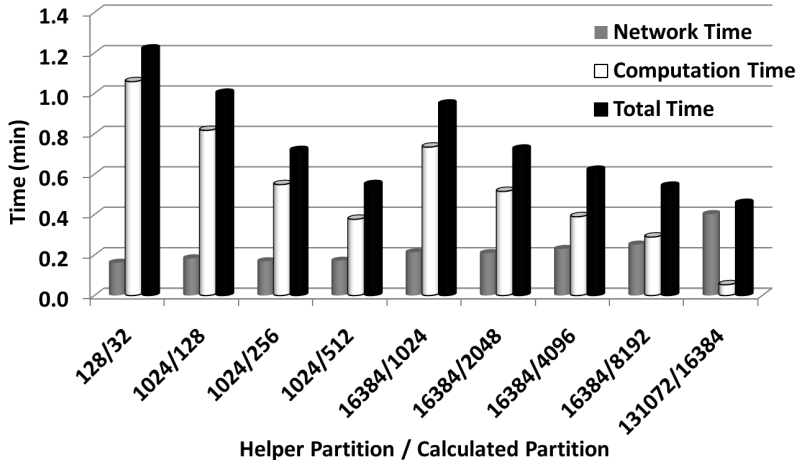


Figure 4.6: *Total, computation and network communication time for various partition sizes, using intermediate helper partitions, four web clients and batch grouping of cells (Travel-times)*

time is close to one minute for all available partition sizes. This fact clearly shows, that even with our distributed browser-based GS preprocessing, similar to CRP, computing the overlay graphs for the lowest level is still the most time consuming process, because it operates directly on the entire road network.

Conclusively, by using three intermediate "helper" partitions and four web clients, we may calculate the *128* partition within a total time of *4min*. To the best of our knowledge, no other SP algorithm distributed implementation requires this little preprocessing time to provide SP queries time in the range of *1.5-2ms* (see Section 4.4.7).

4.4.3 Memory usage

During our experiments, we monitored the memory usage of our web-clients virtual machines, each of them running 2 Google Chrome processes. On those machines *memory usage of Chrome never exceeded 150MB* in any of our experiments. Therefore, in terms of memory requirements, our browser-based implementation is extremely modest and hence it could run on any commodity workstation. This is a great advantage over all other SP preprocessing implementations, which were all tested on quite powerful workstations (usually with at least 6Gb of main memory). Even the only distributed SP preprocessing effort [71] we are aware of, operates on a cluster of nodes, each having 16Gb of main memory. Memory usage on those cluster nodes could get as high as 4Gb. Likewise, our server VM had only 1.7Gb of main memory and could still easily handle our continental road network, since it distributed its computation load entirely to the Web-clients.

4.4.4 Network packets' sizes

An important aspect of our distributed browser-based SP preprocessing is the amount of data transmitted over the network. The server sends the cells' graph to the clients and the clients return an adjacency list of clique arcs calculated for their assigned cells (see Section 4.3.2.1). Both network communications use a JSON representation of cells and results. As expected, each cell's size in KB increases for higher-level partitions. Since for lower-level partitions up until 2048 cells we may send more than one cell to each client, Figure 4.7 shows the average size in KB of a network packet sent from the server

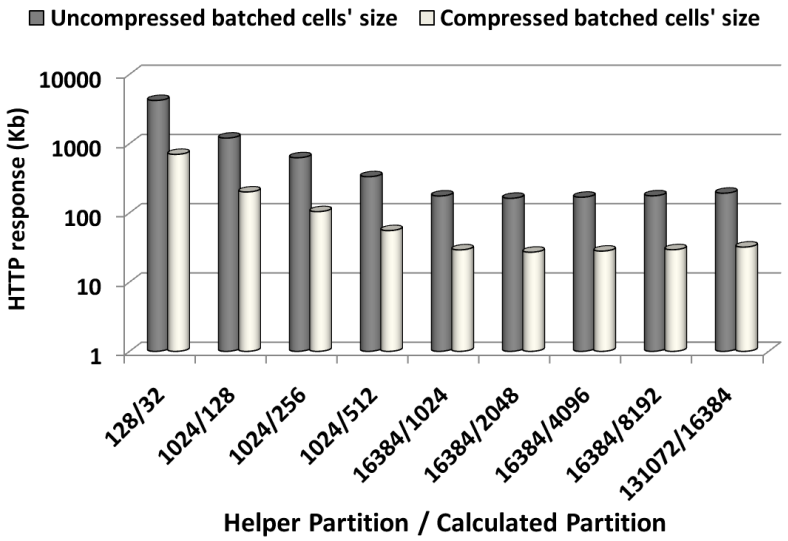


Figure 4.7: *A network packet's size (batched cells) sent from the server to the client for varying partition sizes using intermediate overlay graphs (before and after GZIP compression)*

to the client for varying partition sizes before and after enabling GZIP compression on our web-server.

Figure 4.7 shows that for high-level partitions, an uncompressed JSON representation of a cell's graph occupies almost 4Mb for the 32 partition and almost 1MB for the 128 partition, which is above the 300KB implicit limit imposed. Fortunately, after enabling GZIP compression on our web server, we can further compress network packets sizes by at least a *factor of 6*.

With GZIP compression enabled, network packets (i.e., size of each batched cell group) for all partitions up-until the 128 partition are below 200Kb. Unfortunately, a single cell of the 32 partition occupies more than 700Kb, even after GZIP compression. This is a disadvantage of our method over CRP, since the use of a high-level partition with a very small number of cells was shown there to reduce SP query time to less than 1ms. In our approach we try to avoid such sizes since sending 500Kb - 1Mb of data to a browser-client through AJAX may decrease browser's responsiveness for a potential visitor using our web-service.

In terms of results (i.e., JSON representation of an adjacency list of the clique arcs calculated on the web clients), the average size remains below 100Kb for most partition sizes. Although there are certain cells with a larger number of boundary nodes, the result's size remains stably below 300Kb. This is attributed mainly to the *arc-reduction optimization*, *nodes reordering* (see Section 4.3.2.1), and the compactness of the adjacency list representation we used. Therefore, the size of network packets moved from the client back to the server always remains rather small and consequently has no major impact on overall performance.

4.4.5 Travel distances

To evaluate the impact of a different metric on our approach, we also experimented with travel distances for the same European road network. Results for batched cells, four web clients and the same intermediate "helper" partitions are presented in Fig. 4.8.

Results show that the total time for computing travel distances is *10-15% higher* than compared to computing travel times. With travel distances overlay graphs in lower and

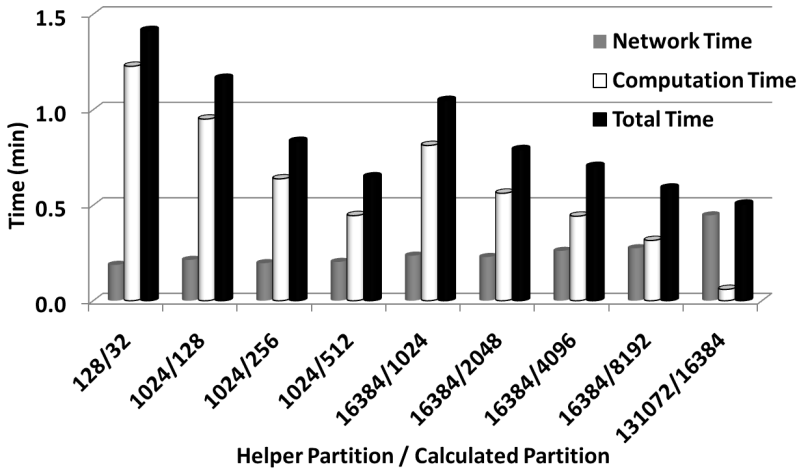


Figure 4.8: *Total computation and network communication time for various partition sizes using intermediate helper partitions, four web clients, and batch grouping of cells (Travel-distances)*

medium level partitions are denser (have 5% more arcs) when compared to travel times graphs. This, in turn, increases both computation and network communication time. This was expected since our arc-reduction optimization works worse for travel distances, which is a common pattern for most arc-reduction SP techniques. On the other hand, a total 10-15% increase translates to merely 30s more computation time, making our approach also a contender for travel-distance computation.

4.4.6 Road network partitioning

For all partition-based shortest-path methods (graph separators included), the partitioning quality plays an important role on, both, preprocessing time and SP query times. To partition the road network graph, we used METIS, which is a general purpose partitioner and therefore is not particularly tuned to road networks. Recently, high quality road network partitioners such as KaFFPa [105] or PUNCH [25] have emerged. These methods focus on road networks and create therefore much better partitions (in terms of number of boundary nodes and arcs) than METIS. Unfortunately at the time of writing this work, we did not have access to those two tools. Therefore, although experiments showed that METIS provides good-quality partitioning, we strongly believe that using a more efficient partitioning tool could further improve the overall performance of our distributed browser-based SP preprocessing method.

4.4.7 Shortest-path queries

Although the main focus of our work was to distribute typical GS preprocessing to multiple browsers for clique calculation, our work would not be complete without stating how those results may be used for typical SP computation and what SP query times may be achieved. Our SP query experiments were performed on a workstation with a six-core AMD Phenom clocked at 3.3GHz and 16Gb of main memory, running Ubuntu 12.04 64bit to provide testing results comparable to those used in state-of-the-art pure graph separator method CRP. Our source code was written in Java using a standard binary heap priority queue implementation and running on a Oracle 7u4 JVM instance. For accurate benchmarking, we used only one core for SP computation.

Table 4.3: *SP query performance for travel-times and travel-distances and 3 levels of overlay graphs. Preprocessing time is calculated with a total of 4 web clients. Customizable Route Planning performance statistics are taken directly from [24]*

			Travel times		**Travel distances**	
			SP Query	**Preproc.**	**SP Query**	**Preproc.**
# cells level 3	**# cells level 2**	**# cells level 1**	**time (ms)**	**time (min)**	**time (ms)**	**time (min)**
32	512	16384	1.812	5.7	2.811	6.2
32	1024	16384	1.909	5.1	2.970	5.6
32	2048	16384	2.122	5.8	3.250	6.4
128	512	16384	2.191	4.4	3.598	4.9
128	1024	16384	2.145	3.9	3.557	4.3
128	2048	16384	2.176	4.6	3.602	5.1
Customizable Route Planning statistics						
MLD-1 [2^{14}]			5.81	0.09	6.12	0.13
MLD-2 [$2^{12} : 2^{18}$]			1.82	0.09	1.83	0.13
MLD-3 [$2^{10} : 2^{15} : 2^{20}$]			0.91	0.09	0.98	0.13
MLD-4 [$2^{8} : 2^{12} : 2^{16} : 2^{20}$]			0.72	0.09	0.79	0.13

To evaluate query performance, we report the resulting query times for a set of 10,000 random SP queries. Results for travel times and distances are presented in Table 4.3. We used 3 levels of overlay graphs, which proved to provide sufficient acceleration with reasonable preprocessing times. Preprocessing times are calculated using 4 Web-clients. For 8-Web clients preprocessing time reduced by a factor of *1.6*. We also provide CRP results as documented in [24]. CRP code was written in C++ (with OpenMP for parallelization) and was compiled with Microsoft Visual C++ 2010. CRP uses 4-heaps as priority queues and experiments run on a Intel Core-i7 920 workstation with four cores clocked at 2.67 GHz and 6 GB of DDR3-1066 RAM, running Windows Server 2008 R2. Contrary to ours, sizes of CRP cells are counted in terms of nodes per cells.

The results show that with *3min* of distributed browser-based preprocessing we can easily achieve SP query times of about *1.8 - 2.2ms* (for travel times and 8 Web clients). As expected, the SP query times recorded by us are little slower than those achieved with the state-of-the-art pure GS approach, since: (i) CRP used 4 levels of overlay graphs - adding more levels increases GS performance but also increases preprocessing time, which in our case should not exceed certain limits (300KB). Additionally, as stated in Section 4.4.4 cells' sizes for high-level partitions are becoming too big for distribution to web clients. Without such a high-level partition, even CRP times are in the range of 2ms. (ii) CRP used the PUNCH [25] partitioning tool which is a better than METIS (iii) CRP used C++ which according to a recent Google benchmark paper [60] is still considerably faster ($\simeq$ 5 times) than Java. (iv) CRP used 2 cores and openMP for SP queries, instead of a sequential implementation we used. (v) CRP uses a streamlined matrix representations of cliques. On the contrary, we wanted to use results directly as we received from the preprocessing process (see Section 4.3.2.1), without any further improvements. Still, our query times are fast enough for real-time services. Also, similar to CRP, our system supports live updates quite efficiently. If a single cell arc-weights change on the lowest level (due to a traffic jam), we must recompute at most one cell on each level, which requires less than 2s.

4.4.8 Summary

Experimentation results of our distributed browser-based GS preprocessing for a continental road network have shown that the preprocessing time can be as low as *3 min* when using 8 Web clients. Moreover, the preprocessing results can be used directly for SP computation and achieve SP query times of *1.8 - 2.2ms* for travel times and *2.8 - 3.5ms* for travel distances, which are comparable to existing methods that rely on sophisticated algorithmic optimizations. Those times could be further improved, by using a more efficient partitioning tool than METIS.

Additionally, we have documented that our approach is extremely modest in terms of memory requirements and costs, since even our server may run on pure commodity hardware and may be built within minutes, using exclusively open-source software. Moreover, memory usage of the Web-clients is extremely low and thus, it may work on any browser of a typical workstation or even on a mobile device. Each cell calculation on the client takes less than 2s and therefore it hardly affects the overall user-experience of a visitor using our web-service. In conclusion, our system is entirely capable of handling continental road networks and graph updates, since its concept is highly scalable and relying on commodity hardware and free software.

4.5 Conclusion and Future Work

This chapter introduced a novel approach for distributing SP preprocessing to multiple Web-clients. Instead of using a dedicated cluster of nodes connected to a network infrastructure, we use Web-browsers and Javascript as our computing platform. All the necessary computation work is distributed to the Web-browsers and the server just transmits cells and collects their results. Hence, not only the computation load on our server remains minimal but even the Web-clients hardly experience any load, since each cell's clique calculation takes less than 2s and memory usage remains below 150MB. Therefore, our client-side approach may work on any conventional workstation or contemporary mobile device. Our extensive experimentation with a continental road-network showed that our approach is feasible, fast and efficient. *With 8 Web-clients, preprocessing for a continental road network requires 3min and we can answer SP queries in almost 2ms*.

Although our work is the first distributed SP preprocessing effort in which clients do not share any common data structures, the *true novelty of our work is the use of Javascript and Web-browsers in the context of SP preprocessing*. Javascript is still considered a toy language, which is not truly capable of handling computing-intensive applications. Our work clearly demonstrated that this is no longer the case. Furthermore, with the popularity of the Web, researchers now have access to an unlimited pool of computing resources and this work showcases plausible and cost-effective ways to do that. By setting up a minimal Web server and relying entirely on open-source tools on a public Web service, we were able to create our dedicated cluster of unlimited nodes within minutes. In that spirit, we seriously hope we will encourage other researchers to use Javascript and Web-browsers as a means to parallelize their computing-intensive problems.

Chapter 5

GRASP. Extending Graph Separators for the single-source shortest-path problem

Many algorithms exist for accelerating point-to-point shortest-path queries on road networks. Contrarily, only few works exist for solving the related single-source shortest-path problem of finding shortest path distances from a single source s to all other vertices in the graph. In this chapter, we extend graph separator methods to handle this specific problem and its one-to-many variant, i.e., calculating the shortest path distances from a single source s to a set of targets $T \subseteq V$. Not only our novel family of algorithms denoted GRASP provide superior preprocessing times than previous methods, thus making them suitable for dynamic scenarios, i.e., road networks with changing edge weights due to traffic updates but they are also able to solve range / isochrone queries not handled by previous approaches. This chapter's content was initially included on our [44] publication presented in the 22nd Annual European Symposium on Algorithms (ESA 2014).

5.1 Introduction

For the single-pair shortest-path problem (SPSP) in road networks, several techniques can be much faster than the Dijkstra algorithm by using a two-phase approach: *Preprocessing* requires a few minutes (or hours) and produces auxiliary data that is subsequently used to accelerate shortest-path (SP) queries. The related research has been so rapidly evolving that even recent surveys [27] had to be updated in later publications [10].

While many efficient techniques exist, an important category of SP algorithms is based on graph separators (GS). The most prominent example of this category is Customizable Route Planning (CRP) [24, 30]. Although CRP is one order of magnitude slower than hierarchical approaches such as Contraction Hierarchies (CH) [49] it is still fast enough for real-time services and offers multiple advantages: (i) It offers very fast preprocessing times of few seconds for continental road networks, making this method suitable for dynamic road networks, (ii) it is very resilient to the metric used, including travel distances and turning costs. This is the reason why global Mapping services such as Bing Maps prefer to use CRP, over faster but less practical solutions such as CH.

In the case of the single-source shortest-path problem (SSSP), given a source vertex s, the goal is to find SP distances from s to *all other graph vertices*. This problem is also addressed by the classic Dijkstra algorithm. Quite recently, [23] introduced the

PHAST algorithm that, compared to Dijkstra, needs fewer operations, has better locality, and exploits parallelism at multi-core and instruction levels. Thus, it is orders of magnitude faster. Later works [26] presented RPHAST for solving the one-to-many variant: given a set of targets T, compute the distances between s and all vertices in T. Since both PHAST and RPHAST are extensions of the CH algorithm, their preprocessing is slow, making those methods practically unsuitable for dynamic scenarios. Moreover, since CH is essentially a bidirectional method, when applied to the SSSP problem, it may only be used if we know the target nodes in advance, as in the entire road network in PHAST, or a subset T of nodes in RPHAST. Therefore, these methods cannot be extended to *range queries*, i.e., find all nodes reachable from s within a given timespan) or *isochrone queries*, i.e., find all nodes AND edges reachable from s within a given timespan, since for those queries we do not know the target nodes in advance.

In this chapter, our goal is to create SP methods that (i) are very fast, (ii) have very short pre-processing times and (iii) may be used for most (if not all) SSSP cases. We achieve this by extending Graph Separators methods and create a novel set of algorithms named GRASP (Graph separators, RAnge, Shortest Path) that have none of the inherent shortcomings of previous approaches. All GRASP algorithms (i) have very short preprocessing time of few seconds, making them suitable for dynamic road networks, i.e., road networks with changing edge weights due to traffic updates, (ii) are very robust with respect to the metric used (either travel times or travel distances), and most of all, (iii) they may efficiently solve range/isochrone queries. As recent works have suggested (cf. [38, 40]), this type of queries is very important for a spectrum of application contexts, including fleet management, urban planning and geomarketing.

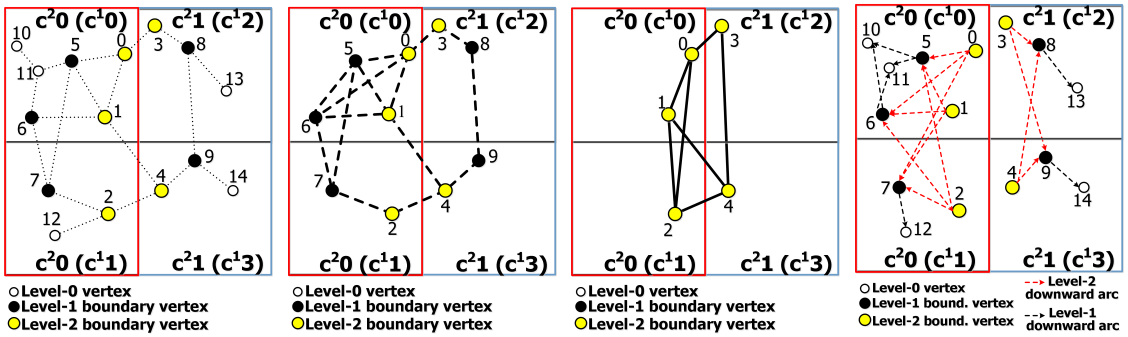
The outline of this chapter is as follows. Section 5.2 describes our family of GRASP algorithms and their implementation. Experiments establishing the benefits of our approach are provided in Section 5.3. Finally, Section 5.4 gives conclusions and directions for future work.

5.2 The GRASP algorithm

In this chapter we extend graph separator (GS) methods to efficiently solve all variations of the SSSP problem. Our novel family of algorithms denoted GRASP have all the advantages of graph separator methods: Extremely fast preprocessing times, fast SP query performance and robustness to the metric used. But they also efficiently answer range / isochrone queries not addressed by previous CH based methods.

In GS techniques, during preprocessing, we compute SP distances (restricted to the inner arcs) between the border vertices of each cell, at each partition level. As shown in [24], the SP distances between (level-ℓ) border vertices of cell c^ℓ may be calculated by running one Dijkstra search per border vertex, in the union of all subcells of c^ℓ at level $\ell-1$. Since those searches use the level $\ell-1$ overlay graph, they are extremely fast.

Extending this observation, GRASP preprocessing (by using the exact same Dijkstra searches as before) additionally computes the SP distances between all border vertices of level ℓ and all vertices of level $\ell-1$ within each cell c^ℓ. To differentiate between the two kind of arcs calculated by the same search, we will denote as (i) *clique arcs* those added overlay arcs that connect border vertices of the same level ℓ and (ii) *downward arcs* of level ℓ those connecting vertices at different levels, i.e., ℓ and $\ell-1$. Adopting the notation of [24], the *overlay graph* containing border vertices, border arcs and clique arcs will be denoted hereafter as H. For added efficiency, *downward arcs* are stored as a separate



(a) A graph G. $|V|$= 15, L=2, $|C^L|$=2 (b) Level-1 overlay graph (c) Level-2 overlay graph (d) The $G_{GS}\downarrow$ graph

Figure 5.1: *Overlay graphs and the $G_{GS}\downarrow$ graph for an example graph G*

graph (using a single adjacency array representation [85]), hereafter referred to as $G_{GS}\downarrow$. Since we use nested partitions, H and $G_{GS}\downarrow$ do not necessarily need to have the same number of levels. The intermediate steps for building the overlay graph H and the $G_{GS}\downarrow$ for a sample graph G, are shown in Fig. 5.1.

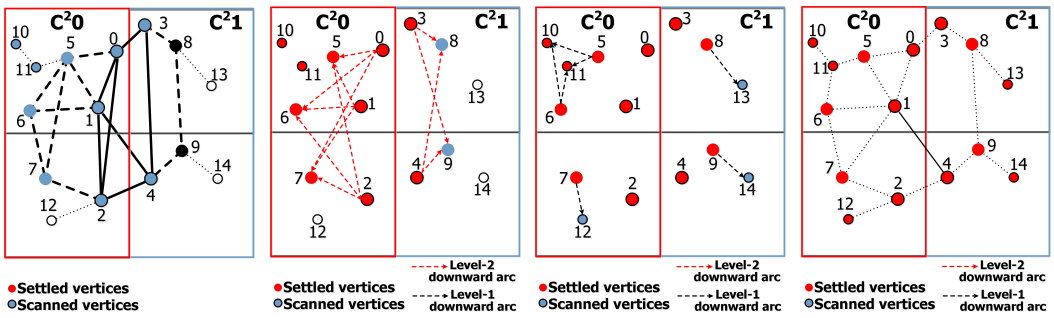
When GRASP executes a SSSP query from source s, it runs a simple Dijkstra search in the union of the cell $c^0(s)$ and H until the priority queue is empty. This will be referred to as the *upward phase* of GRASP. The upward phase settles all border vertices of level L, all vertices of $c^0(s)$ and some additional vertices inside $c^L(s)$. Note that, *all vertices settled by the upward phase are assigned correct SP distances from s*, due to the definition of the overlay graphs. For a random cell c^L by using the level-L downward arcs (inside c^L) we may calculate correct SP distances to all level-L-1 border vertices from s. Recursively, in descending order of GS levels, we may calculate correct SP distances for all vertices inside c^L. This process is repeated for all (level-L) cells. This second stage is denoted as the *scanning phase* of GRASP (see Procedure GRASP). Figure 5.2 shows the various stages of the algorithm.

```
GRASP(s, d(V), nsVec(V), G, H, G_GS↓)
1  Init nsVec(V) to FALSE for level − L
2  Dijkstra(s, c0(s)∪H, d(V), nsVec(V))
3  parallel for each cell in C^L partition
4      for level = L − 1 to 0
5          for vertex v of level in cell
6              if nsVec[v] == FALSE
7                  Scan(v, d(V), G_GS↓)
8              else nsVec[v] = FALSE
```

```
SCAN(v, d(V), G_GS↓)
1  for edge in G_GS↓ incoming to v
2      tl = edge.tail
3      if d[v] > d[tl]+edge.weight
4          d[v] = d[tl]+edge.weight
```

Theorem 5.1. *The GRASP algorithm is correct.*

Proof. As shown in [24] the graph H is an overlay, i.e., the distance between any two vertices in H is the same as in original graph G. That means that the union of $c^0(s)$ and H suffices to calculate correct shortest-path distances from s to any vertex of H. The upward phase of GRASP settles all border vertices of level L, all vertices of $c^0(s)$ and some additional vertices inside $c^L(s)$. In addition, *all vertices settled by the upward phase are assigned correct SP distances from s*, due to the definition of H. This is a fundamental difference of GRASP over PHAST, since PHAST's upward phase does not guarantee calculating correct SP distances for all vertices settled by it.



(a) Upward phase, settling all level-2 border vertices

(b) Level-2 scanning phase, settling all level-1 border vertices

(c) Level-1 scanning phase, settling all level-0 vertices

(d) Scanning phase has assigned correct distances to all vertices

Figure 5.2: *The stages of GRASP algorithm from source vertex with ID=10*

Since all border vertices of level L are assigned correct SP distances from s by the upward phase, by definition of the downward arcs, the scanning phase of GRASP calculates correct SP distances from s, for all level-L cells. The only less obvious case seems to be $c^L(s)$, due to the $c^0(s)$ (the lowest level cell that contains s), since there may be a direct shortest-path from s to any of the vertices located inside $c^0(s)$, without using any border vertices. Luckily by definition, the upward phase of GRASP has already settled those vertices correctly. In addition, the upward phase has already settled all level $L-1$ border vertices of $c^L(s)$ and therefore GRASP calculates correct SP distances for all other vertices inside $c^L(s)$ as well. The only adjustment that needs to be made during the scanning phase of GRASP is, that if a vertex is already settled by the upward phase it may be safely be ignored, since it is already assigned a correct SP distance from source s. □

5.2.1 GRASP Tuning

Although GRASP's simplicity and correctness is evident by its definition,we need to take several steps to further improve its query performance. Those steps include:

Initialization. All Dijkstra based variants assume that distance labels for all vertices are set to ∞ during initialization. This requires a linear sweep over all distance labels, which can be slow. To improve speed and to avoid scanning a considerably percentage of downward arcs, GRASP uses a bit vector of size $|V|$, termed *nodeScannedVector* (cf. [26]). In the GRASP upward phase, visited vertices have their associated bit set to true and correct distance labels assigned. During the scanning phase, GRASP avoids scanning vertices with set bits (line 6 of procedure GRASP) and resets their associated bit (line 8) for subsequent searches. This avoids scanning a considerably percentage of downward arcs and saves computation time.

Number of levels. With respect to the required levels of overlay graphs, four levels of overlay graphs suffice to achieve fast SPSP query times [24]. In our case, mi-nimizing the number of downward arcs for $G_{GS}\downarrow$ requires as many intermediate levels as possible. Experimentation showed that $L = 16$ yields best results. For the upward phase of GRASP and overlay graph H we need to only "use" four of those levels (namely: C^4, C^9, C^{12}, C^{16}). For point-to-point SP queries, typically performance improves with a decreasing number of $|C^L|$ cells at the highest level partition. However in our case, decreasing $|C^L|$ creates a larger number of downward arcs in $G_{GS}\downarrow$ and limits GRASP's parallel performance. Thus, we get optimal results for medium number of cells at the highest level partition, experimentally established to be $|C^L| = 128$.

Nodes reordering. To improve performance, we *reorder the vertices of graph G* (cf. [14]), such that overlay vertices are assigned smaller IDs (ordered by descending level), breaking ties with cell. Non-border vertices are ordered by their level-1 cells. In addition, within the same cell (and level) we order vertices by a DFS layout similar to [43]. This nodes' reordering improves (i) spatial locality for preprocessing and (ii) both the upward and scanning phases' performance of GRASP.

Arc reduction. Previous GS approaches [24, 14] compute all available overlay arcs between border vertices. This would be very wasteful for GRASP, especially for the downward arcs. For GRASP, we extend the arc-reduction optimization of [47], which during preprocessing, reports only distances of boundary nodes that are direct descendants of the root of each executed Dijkstra algorithm. This optimization preserves correct distances between boundary vertices on the overlay graph and leads to a 56-71% reduction in the number of arcs created (depending on the cell size). Although originally used for clique arcs, it works even better for downward arcs. This optimization has (i) no negative impact on preprocessing time, since it is integrated in each Dijkstra search, (ii) creates fewer arcs (both clique and downward) and therefore makes GRASP less memory intensive, (iii) arc-reduction is achieved individually per cell (each cell computation is still independent from other cells), which is especially important in cases where traffic updates are restricted to a limited number of cells.

Parallelization A final performance boost comes from exploiting the parallel nature of GRASP. For a single-tree computation, PHAST requires to pause and synchronize threads at each CH level. Since the number of CH levels is quite large, (140 in the case of continent-size networks), parallel performance is not optimal. Contrarily, GRASP only has 16 GS levels. This allows for better parallel scaling of GRASP. Still, we can do even better. By definition, each level-L cell may be processed independently - see line 3 of procedure GRASP. Thus, the parallel implementation of GRASP requires no intermediate barriers for a single-tree computation and consequently scales much better for a large number of cores. As will be shown in Section 5.3, parallel GRASP is as fast as parallel PHAST, while requiring only a fraction of PHAST's preprocessing time. As a result, GRASP is the most competitive solution for answering SSSP queries for dynamic (live-traffic) road networks.

5.2.2 Range and Isochrone Queries

Another variation of the SSSP problem, is Range / Isochrone Queries. *Range queries* identify and assign correct SP distances to all vertices reachable from a source s within a given range (either travel times or distance) limit e. *Isochrone queries* find all nodes *and* edges reachable from s within a given limit e. They are an extension of range queries, which simply requires an additional linear sweep over the original graph adjacency array of G to discover which edges are reachable from s within the given range. This process (described by the procedure RangeToIsoc) is very fast and requires less than few ms for continental road networks. Unfortunately, this type of queries cannot be handled by previous approaches, such as PHAST or RPHAST due to the inherent bidirectional nature of CH, which requires to know the target vertices in advance. In the following, we will describe how our novel isoGRASP algorithm (i.e., isochrone GRASP, an adaptation of GRASP) efficiently solves range queries.

```
RangeToIsoc(e, d(V), nsVec(V), esVec(E), G)
1  for vertex v = 0 to |V| − 1
2      if nsVec(V) == TRUE
3          for edge in G outgoing from v
4              if d[v] + edge.weight ≤ e
5                  esVec[edge]=TRUE
```

Range queries mark and assign correct SP distances to all vertices reachable within the limit e. To this end, isoGRASP uses the nodeScannedVector (similar to GRASP). The isoGRASP algorithm uses the exact same preprocessing as before, i.e., building the H and $G_{GS}\downarrow$ graphs and also has an upward and a scanning phase. In the upward phase, isoGRASP runs a *RangeDijkstra* search in the union of $c_0(s)$ and H, which is a modified Dijkstra algorithm that only allows vertices u with SP distance $d(u) \leq e$ to enter the priority queue. As a result, the upward phase of isoGRASP terminates early and settles only those level-L border vertices and some additional vertices inside $c^L(s)$ that are reachable from source within the specified limit e. Moreover, during the upward phase, isoGRASP marks those level-L cells which are reachable from source. To achieve this, we use an additional bit vector of size $|C^L|$ (128 in our experiments), denoted hereafter as the *cellScannedVector* (and $csVec(C^L)$ in the pseudocode).

During isoGRASP's scanning phase, by utilizing the cellScannedVector, we restrict calculations to those level-L cells reachable from source within the limit e. This saves a lot of computation time for smaller values of e. Contrary to GRASP, when we scan a node u, we only look at downward arcs where the tail vertex v of this arc has its associated nodeScannedVector bit set to true, i.e, the tail v of this arc is reachable within e. If $d(v)+w(v,u) \leq e$ and $d(v)+w(v,u) < d(u)$ then $d(u) = d(v)+w(v,u)$ and the nodeScannedVector bit of u is set to true. Thus, after isoGRASP's scanning phase, all vertices reachable from a source within e are assigned correct SP distances and have their nodeScannedVector bit set to true (see procedure IsoGRASP).

Lemma 5.1. *The isoGRASP algorithm is correct.*

Proof. Since the GRASP algorithm is correct, the upward phase of isoGRASP assigns correct SP distances (and sets their associated nodeScannedVector bits to true) only to level-L border vertices and some additional vertices inside $c^L(s)$ which are reachable from source within the limit e. In addition, the upward phase of isoGRASP marks which level-L cells are reachable from source within e, based on the observation that if no level-L border vertices of a level-L cell is settled within range e, then it is impossible for any vertex inside this cell to be reachable within the range as well. This applies to all cells except $c^L(s)$ that always has its associated cellScannedVector bit set to true.

By using the cellScannedVector information acquired during the upward phase, the scanning phase of isoGRASP ignores level-L cells not reachable from source within e and scans only downward arcs where the tail vertex is reachable within e. Since the scanning phase sets $d(s,u) = d(s,v)+l(v,u)$ only when $d(s,v)+l(v,u) \leq e$, then it assigns correct SP distances (and sets the corresponding nodeScannedVector bits to true) to all vertices reachable within e (and only them). Thus, isoGRASP is correct. □

5.2.3 One-to-Many Queries

The *one-to-many* SSSP variant finds SP distances from a source s to a non-empty set of targets $T \subseteq V$. We call the respective algorithm *reGRASP* (restricted GRASP). Similar

```
isoGRASP(s, e, d(V), nsVec(V), csVec(C^L),
G, H, G_GS↓)
1  Initialize nsVec(V) to FALSE
2  Initialze csVec(C^L) to FALSE
3  RangeDijkstra(s, c_0(s)∪H, d(V),
           nsVec(V), csVec(C^L))
4  parallel for each cell in C^L partition
5      if csVec[cell] == TRUE
6          for level = L − 1 to 0
7              for vertex v of level in cell
8                  if nsVec[v]==FALSE
9                      RScan(v, e, d(V),
                       nsVec(V), G_GS↓)
```

```
RScan(v, e, d(V), nsVec(V), G_GS↓)
1  for edge in G_GS↓ incoming to v
2      tl = edge.tail
3      if nsVec(tl) == true
4          if d[v] > d[tl]+edge.weight
            & d[tl]+edge.weight ≤ e
5              d[v] = d[tl]+edge.weight
6              nsVec[v]=TRUE
```

to [26], reGRASP has a *target selection phase*, which only depends on the targets' set T. During the *target selection phase*, reGRASP marks the vertices T' ($T \subseteq T' \subseteq V$) that are necessary for computing SP distances to all vertices $\in T$.

Here, we use a new bit vector of size $|V|$, the *restrictedVector* (denoted as $rVec(V)$ in the pseudocode). All its bits are set to false except those referring to T vertices IDs. Then we sweep this vector from the higher bits to the lower. When we meet a marked vertex (true bit), by using the adjacency array representation of the $G_{GS}\downarrow$ graph we mark the vertices that need to be added to T'. Since, downward arcs connect higher level vertices (which correspond to smaller IDs, according to our nodes reordering) with lower level vertices (with larger IDs), each vertex needs to be scanned only once.

We may further accelerate the target selection phase by using again the *cellScanned-Vector*. By the definition of $G_{GS}\downarrow$, we know that all vertices belonging to T' belong to the same level-L cells as T vertices. Therefore at step 1 of the target selection process, we mark the corresponding bits of level-L cells of vertices belonging to T to true. Therefore during this phase, we may safely ignore level-L cells with their corresponding cell bits set to false. Moreover, each such cell may be processed in parallel since the $G_{GS}\downarrow$ graph only connects vertices belonging to the same level-L cell. This is a major advantage of GRASP over RPHAST, where the respective time-consuming target selection phase cannot be parallelized. Procedure TS shows the finalized pseudocode.

After the target selection phase, reGRASP has an upward and a scanning phase. The upward phase is exactly the same as GRASP and settles all level-L border vertices and some additional nodes inside $c^L(s)$. Then during the scanning phase (i) we ignore cells with their *cellScannedVector* bits set to false and (ii) we do not scan vertices not belonging to T' (i.e, have their restrictedVector bits set to false). Again, during the scanning phase each level-L cell may be processed in parallel (see Procedure reGRASP).

Lemma 5.2. *The reGRASP algorithm is correct.*

Proof. Since the GRASP algorithm is correct, the upward phase of reGRASP assigns corrects SP distances (and sets their associated nodeScannedVector bits to true) to all level-L border vertices and some additional vertices inside $c^L(s)$ from source s. Moreover, the target selection phase of reGRASP has already marked those cells and vertices necessary for calculating correct SP distances for all vertices belonging to T from any source (including s). Since the scanning phase scans all vertices marked by the target selection

```
reGRASP(s, T, d(V), nsVec(V),
csVec(C^L), rVec(V), G, H, G_GS↓)
 1  TS(T, csVec(C^L), rVec(V), G_GS↓)
 2  Init nsVec(V) to FALSE for level − L
 3  Dijkstra(s, c_0(s)∪H, d(V), nsVec(V))
 4  parallel for each cell in C^L partition
 5      if csVec[cell] == TRUE
 6          for level = L − 1 to 0
 7              for vertex v of level in cell
 8                  if nsVec[v]==FALSE
                     & rVec[v]==TRUE
 9                      Scan(v, d(V), G_GS↓)
10                  else
11                      nsVec[v]=FALSE
```

```
TS(T, csVec(C^L), rVec(V), G_GS↓)
 1  Initialze rVec(V) to FALSE
 2  Initialze csVec(C^L) to FALSE
 3  for each vertex t in T
 4      rVec[t] = TRUE
 5      csVec[c^L(t)] = TRUE
 6  parallel for each cell in C^L partition
 7      if csVec[cell] == TRUE
            // Skip level-L
 8          for level = 0 to L − 1
 9              for vertex v of level in cell
10                  if rVec[v] == TRUE
11                      for edge in G_GS↓
                        incoming to v
12                          tl = edge.tail
13                          rVec[tl]=TRUE
```

phase, it calculates correct SP distances from source s for all vertices belonging to T, due to the correctness of the $G_{GS}\downarrow$ graph and GRASP. □

The main advantage of reGRASP over RPHAST, is that (i) we may ignore level-L cells that do not contain T vertices and (ii) that *each level-L cell may be processed in parallel, even during the target selection phase*. This way, parallel reGRASP always exhibits excellent performance.

Conclusively, the most important aspect and outcome of our approach is not three distinct algorithms but a unified framework that solves all variants of the SSSP problem. The exact same preprocessing and the same data structures (the overlay graph H and the $G_{GS}\downarrow$ graph) suffice to solve all these different variations. In this scenario, building a server that concurrently answers all types of queries (one-to-all, range/isochrone and one-to-many) as they arrive from clients, is possible for the first time.

5.3 Experiments

The scope of our experiments is to evaluate the performance of all GRASP algorithms for different SSSP variations. The experiments were performed on a workstation with a 4-core Intel i7-3770 processor clocked at 3.4GHz and 32 GB of RAM, running Ubuntu 12.10. Our code was written in C++ with GCC 4.7 and optimization level 3. We used OpenMP for parallelization. We used the the European and the full USA road networks (18M nodes / 42M arcs and 24M nodes / 58M arcs respectively) made available from the 9th DIMACS Implementation Challenge [33]. We experimented with both travel times and travel distances. For partitioning the graph into nested-multilevel partitions, similarly to [43], we used Buffoon / KaFFPa [106] in a top-down approach. For the lowest four level partitions ($C^1, \ldots C^4$), we switched to METIS [68] for faster computation. For both benchmark road networks used, we used a total of 16 partitioning levels (i.e., $L = 16$) and the C^{16} partition contains 128 cells. Each partition below that, contains double the cells of the previous highest level partition (i.e., C^{15} has 256 cells, C^{14} has 512 cells and so-on).

To compare the performance to existing approaches, authors of PHAST / RPHAST

Table 5.1: *Comparison of GRASP and PHAST for both travel times and travel distances*

	Preprocessing Time (s)				**Query Time (ms)**			
	Travel times		**Travel distances**		**Travel times**		**Travel distances**	
	Europe	**USA**	**Europe**	**USA**	**Europe**	**USA**	**Europe**	**USA**
PHAST	99	160	824	475	**39 (103)**	60 (**169**)	50 (**139**)	**64 (179)**
GRASP	**8**	**12**	**10**	**13**	43 (150)	**58** (207)	**46** (156)	66 (218)

have kindly conducted all experiments for the same road networks on a similar workstation setup to ours (they could not provide direct access to source code due to IPR claims). Their setup is a workstation with an Intel i7-3770K (almost identical to ours, except it is clocked higher at 3.5GHz) with the same 32 GB of main memory. Their workstation runs Windows 8.1, their code is also written in C++ (and OpenMP) and was compiled using Visual Studio 2012. Since their processor is clocked 0.1Ghz higher (is 2.9% faster than ours), we multiply their timing results with 1.029, similar to [43]. Still, in most cases this tweaking has minimum impact on the observed results.

5.3.1 GRASP vs. PHAST.

We compare GRASP and PHAST's preprocessing and query times for calculating SP distances of all graph vertices from a single source. Preprocessing times refer to parallel execution and for query times we report both sequential and parallel times for a single-tree computation. Query times are averaged over 10,000 randomly selected sources. Results are presented in Table 5.1. The best results in each case are highlighted in bold and the numbers in parentheses refer to sequential times.

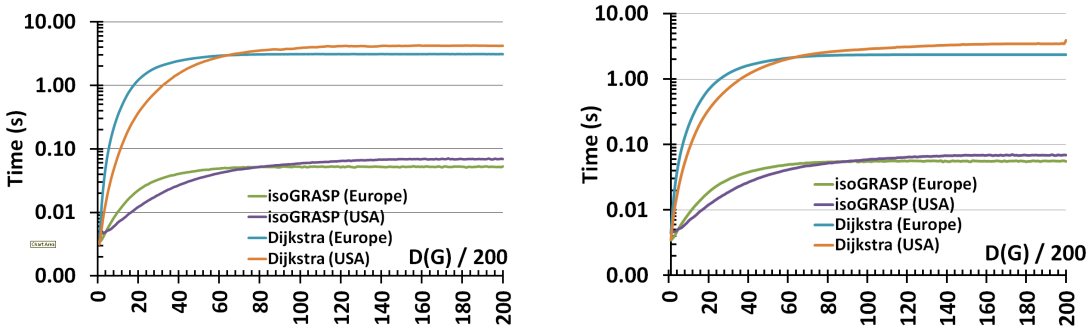
Regarding preprocessing, GRASP is notably more efficient than PHAST. GRASP's preprocessing is *13 times* faster than PHAST for travel times and *37-82* times faster for travel distances. *GRASP's preprocessing takes less than 15s for both metrics* and shows little change when using travel distances, which is in stark contrast to PHAST. In terms of memory consumption, PHAST is slightly better, but GRASP has also very modest requirements, since it requires no more than $1Gb$ for storing the H and $G_{GS}\downarrow$ graphs.

In terms of *query performance*, results are evenly mixed. Although sequential PHAST is slightly faster than GRASP, the parallel implementation of GRASP scales better. As a result, parallel GRASP is faster for the USA network for travel times and Europe network for travel distances.

As a result of this first experiment, we observe that GRASP is a more complete solution for solving the one-to-all variant of the SSSP problem. It requires a fraction of PHAST's preprocessing time, is more robust to the metric used and scales better for multicore processors. Also, GRASP's robustness to the metric used, indicates that it will probably outperform PHAST when we switch to additional metrics (e.g., applying turn-costs), where CH based solutions typically perform poorly.

5.3.2 isoGRASP.

The following experimentation evaluates isoGRASP's performance for the *range variant of the SSSP problem*, i.e., assign correct SP distances to all nodes reachable from a single source within a range limit e. To this end, we chose 1,000 random vertices as sources s and performed range queries for multiple values of e. Since our benchmark Europe and USA networks have different units for travel times, we use range limits that are fractions of the



(a) Travel times (b) Travel distances

Figure 5.3: *Parallel isoGRASP performance compared to a sequential Dijkstra for varying values of e compared to total road-network diameter D(G). Y-axis is on logarithmic scale*

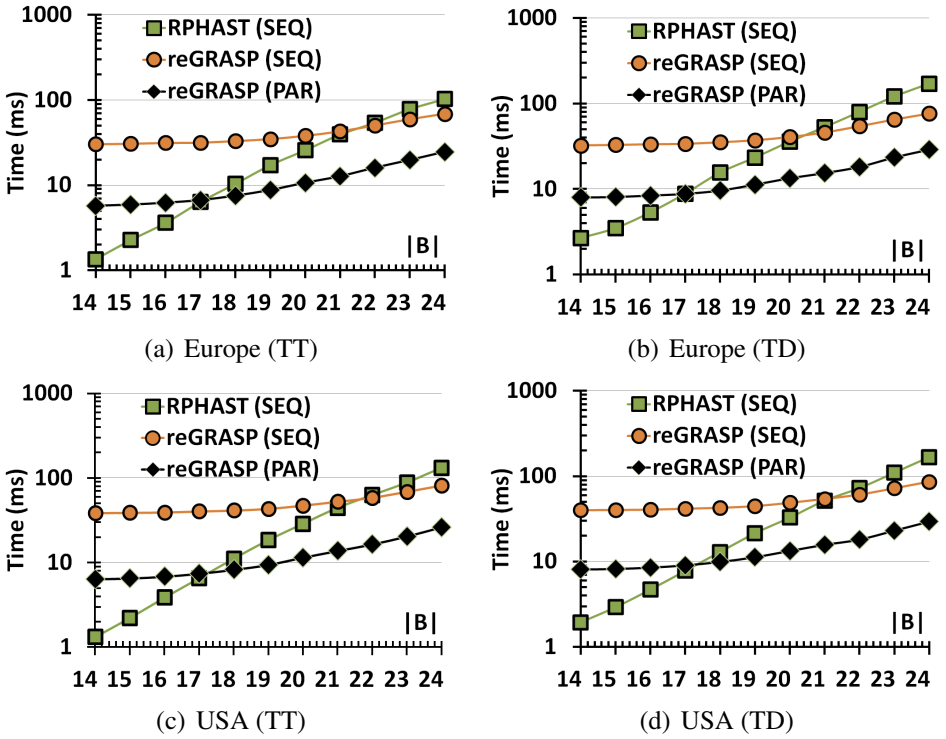
road network diameters (denoted hereafter as $D(G)$), as estimated by a total of 1000 SSSP queries from random vertices. Since no other pure algorithmic methods exist for this particular problem, we compare our parallel isoGRASP implementation to a sequential Dijkstra implementation. Results are presented in Fig. 5.3(a) and 5.3(b), respectively.

Results are similar for both networks and metrics. IsoGRASP is orders of magnitude faster than Dijkstra and exhibits stable performance for both metrics. In addition for $e < 0.2 \times D(G)$, parallel isoGRASP is at least twice as fast as parallel GRASP / PHAST. Since in range queries we are usually interested in small ranges close to the source vertex, isoGRASP should always be the algorithm of choice, instead of using PHAST / GRASP for calculating all graph vertices distances and then sweep the distances vector to determine which of those are reachable within the limit e.

5.3.3 reGRASP vs RPHAST.

The final set of experiments compares reGRASP and RPHAST for the *one-to-many* variant of finding SP distances from a source s to a non-empty set of targets $T \subseteq V$. We adopt the methodology of [26], which picks a random vertex and performs a Dijkstra search until reaching a fixed number of vertices. If B is the set of vertices settled during this search, our targets T are chosen as a random subset of B. Hence, our input parameters are the number of targets $|T|$ and the size of $|B|$. This simulates different scenarios, with either targets close together or spread throughout the graph. For each set of parameters, we test 100 different sets of targets, each with 100 different sources. We keep the number of targets $|T|$ fixed at 16,384 (2^{14}) and experimented with different values of $|B|$ ranging from $2^{14} \ldots 2^{24}$. We report total times (target selection + query phase) for sequential RPHAST, sequential reGRASP and parallel reGRASP. Results for the Europe and USA road networks are shown in Fig. 5.4.

In terms of sequential performance, RPHAST is faster for $|B| \leq 2^{20}$ and reGRASP in all other cases. However, RPHAST performance degrades quickly and for random objects (i.e., $|B| = 2^{24}$) it takes more than $100ms$, i.e., it is even slower than parallel PHAST / GRASP. In contrast, parallel reGRASP scales well using all available cores and offers stable performance. Even for $|B| = 2^{24}$ a search takes less than $30ms$ for both networks and metrics, i.e., *parallel reGRASP is always faster than parallel PHAST or GRASP*. Thus, parallel reGRASP is the method of choice, even for large $|B|$ values, or with changing targets $|T|$. RPHAST on the other hand should only be used for the case of a static set of points of interest and small $|B|$ values. When compared to RPHAST,



(a) Europe (TT) (b) Europe (TD)

(c) USA (TT) (d) USA (TD)

Figure 5.4: *Parallel and sequential reGRASP performance compared to RPHAST for travel times (TT) and travel distances (TD). X and Y-axes are on logarithmic scale*

Table 5.2: *Summary of results for all variants of the SSSP problem*

Type of queries	Parameter	Best solution
One-to-all	**Preproc. time / Dynamic networks**	GRASP
	Query time (SEQ)	PHAST
	Query time (PAR)	GRASP / PHAST
	Scales better on multicores	GRASP
Range / Isochrone	**Preproc. time / Dynamic networks**	isoGRASP / GRASP
	Query time (PAR)	isoGRASP
One-to-many	**Preproc. time / Dynamic networks**	GRASP / reGRASP
	No extra data structures	GRASP / reGRASP / PHAST
	Query time (static T, $\|B\| \leq 20$)	RPHAST
	Query time (rest)	reGRASP (PAR)

parallel reGRASP does not require any additional graph structures and is 2.6 – 7 times faster, while requiring only a fraction of RPHAST's preprocessing time.

Although at first, it seems unfair to compare sequential RPHAST with parallel reGRASP, RPHAST stands to benefit little from parallelization due to its time-consuming target selection, which is hard to parallelize. Also, RPHAST's query phase uses the $G_T^{\downarrow}$ graph and therefore it is already fast. As a result, very minimal improvements could be achieved through parallelization, due to the small number of edges present in $G_T^{\downarrow}$.

5.3.4 Summary.

Table 5.2 summarizes our results. It is evident that all GRASP variants are overall the most complete algorithmic solutions for solving all variations of the SSSP problem. They have very short preprocessing times, are therefore suitable for dynamic road networks,

provide excellent parallel performance, scale better on multicore processors and offer better or comparable performance to state-of-the-art approaches. Moreover, all GRASP variants utilize the same graph structures and therefore can be used within the same server infrastructure to concurrently answer all related queries. Thus, it is the only approach to offer this kind of simplicity and efficiency for all SSSP cases.

5.4 Conclusion and Future Work

This chapter introduced novel graph separator methods to efficiently handle all variations of the single-source shortest-path (SSSP) problem. The three proposed algorithms, GRASP, isoGRASP and reGRASP are each tailored to a specific SSSP problem (one-to-all, range, and one-to-many). All GRASP algorithms rely on the same preprocessing and graph structures and hence, they may be used as a unified framework for answering SSSP queries of any kind. They also require minimal preprocessing time and, thus, are the only viable solution for handling dynamic road networks, i.e., road networks with changing edge weights due to traffic updates. Moreover, they provide excellent parallel performance for both travel times and travel distances metrics. As a result, the GRASP family of algorithms is the best overall solution for processing most SSSP problems.

In terms of future work, we will focus on expanding our results to other types of graphs, for which existing approaches typically do not perform well. Moreover, we aim at creating a unified framework that may concurrently answer point-to-point and SSSP queries on road networks. This effort will be described in the following chapter.

Chapter 6

SALT. A unified framework for all shortest-path query variants on road networks

Although recent scientific output focuses on multiple shortest-path problem definitions for road networks, none of the existing solutions does efficiently answer all different types of SP queries. This chapter proposes SALT, a novel framework that not only efficiently answers SP related queries but also k-nearest neighbor queries not handled by previous approaches. Our solution offers all the benefits needed for practical use-cases, including excellent query performance and very short preprocessing times, thus making it also a viable option for dynamic road networks, i.e., edge weights changing frequently due to traffic updates. The proposed SALT framework is a deployable software solution capturing a range of network-related query problems under one "algorithmic hood". This chapter's content is mostly based on our [45] preprint, publicly available on the Computing Research Repository (CoRR).

6.1 Introduction

During the last decades, recent scientific literature has focused on researching efficient methods for shortest-path (SP) related problems. The related research has evolved so rapidly that even the recent overviews of [27, 12] had to be updated in subsequent publications [10]. Unfortunately, despite this plethora of efficient algorithms only *few of them may actually be used in a practical application context*. The requirements to such a potent approach should be (i) preprocessing time of few *seconds* for continental road networks and (ii) SP query times of a few *ms*. Currently, only two candidates fit these strict requirements. The *graph-separator* approach of Customizable Route Planning (CRP) [24, 30] and the recent adaptation [43] of the ALT [50] algorithm. Due to these specific properties, said algorithms are used in commercial solutions, such as Bing Maps and SimpleFleet [38] the commercial prototype of the authors), for their live traffic-based routing.

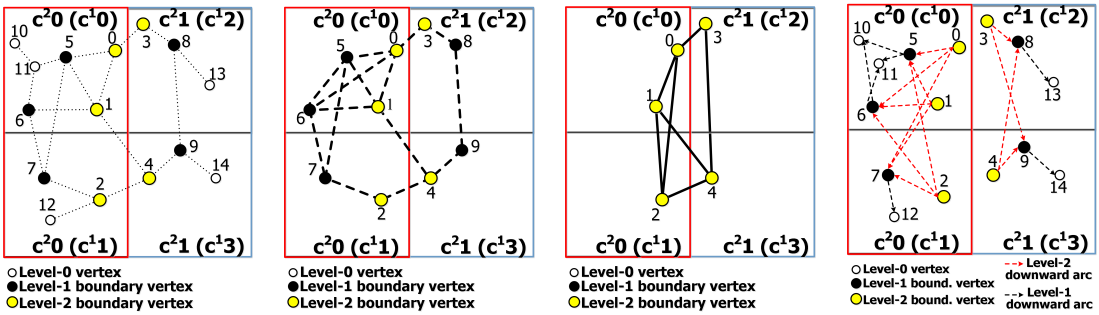
Unfortunately, most of the developed algorithms are tuned to solving a specific problem efficiently, but are rather inefficient when used in a different context. Contrarily, engineering a framework that efficiently solves multiple shortest-path problems, would not only increase the commercial potential of such a solution but would also be the first step towards the direction of a *grand unified SP toolkit*. To this purpose, Efentakis et al. [44] extended graph-separators and proposed the novel set of GRASP (Graph sep-

arators, RAnge, Shortest Path) algorithms that solve most variants of the single-source shortest-path problems on road networks, including *one-to-all* (finding SP distances from a source vertex s to all other graph vertices), *one-to-many* (computing the SP distances between the source vertex s and all vertices of a set of targets T) and *range queries* (find all nodes reachable from s within a given timespan / distance). GRASP requires minimal preprocessing time and provides excellent parallel query performance needed in the context of practical applications and respective commercial solutions.

Another fundamental problem frequently encountered in location-based services is the k-NN query, i.e., given a query location and a set of objects on the road network, the k-NN search finds the k-nearest objects to the query location. Unfortunately, most proposed approaches, such as SILC [102] or ROAD [73, 74] cannot scale for continental road networks [126]. Even recent attempts (improving previous methods), such as G-tree [126], are not scalable with respect to the network size, since they require preprocessing of *several hours* for continental road networks. In addition, for a large number of randomly distributed objects, an efficient Dijkstra implementation could answer k-NN queries (for small values of k) by settling a few hundreds nodes and requiring $< 1ms$. This performance benchmark is often overlooked when assessing novel k-NN methods. Moreover, most existing approaches (contrarily to Dijkstra) require a *target-selection phase*, i.e., they need to mark the objects location within the underlying index. This phase takes a few seconds, hence having limited appeal for applications involving moving objects (e.g., vehicles). Therefore, it only makes sense to use a complex (as in non-Dijkstra) k-NN processing framework in cases of either rather "small" numbers of objects or objects following skewed distributions (e.g., POIs located near the city center), i.e., for cases in which Dijkstra does not perform well.

Moreover, there is another significant limitation to previous approaches. Specialized indexes such as G-tree, may only answer kNN queries. As a result, we require additional data structures for answering the most common type of queries, which is the typical p2p shortest-path queries. Having several data structures for different types of queries within the same application, increases the complexity (and the required resources) and as a result it is not suitable for practical real-world applications. This particular problem has been tried to be addressed by an adaptation of CRP in [31] but unfortunately it shares the major limitation of previous methods: It performs well only for larger number of objects (where Dijkstra already provides excellent performance) and also needs a target-selection phase, which still requires a few seconds. As a result, CRP may only be applied for static objects.

Putting everything together, the ambition of this chapter is to provide a unified algorithmic solution that may be used in a *dynamic road network* context by having very short preprocessing times and competitive query times, while covering a *wide range of shortest-path and network search problems*, such as (i) single-pair, (ii) one-to-all, (iii) one-to-many, (iv) range and (v) k-NN queries. Specifically, we aim at combining the fragmented approaches related to the various shortest-path problem definitions and instead propose a unified framework that tackles all of them. Our proposed **SALT** (graph Separators + ALT) framework requires only seconds for preprocessing continental road networks and provides excellent query performance for a wide range of problems. We will show that SALT is (i) $3-4\times$ faster for point-to-point queries when compared to existing methods of similar preprocessing times, (ii) it answers one-to-all, one-to-many and range queries with comparable performance to state-of-the-art approaches, and most importantly, (iii) it may also answer k-NN queries in $< 1ms$, for both, static or moving objects. As such, our SALT framework could be a *swiss-army-knife for tackling all shortest-path problem*



(a) A sample graph G. $|V|$= 15, L=2, $|C^L|$=2

(b) Building the level-1 overlay graph

(c) Building the level-2 overlay graph

(d) Downward arcs and the $G_{GS}\downarrow$ graph

Figure 6.1: *SALT's GS customization phase. Building the overlay graph H and the $G_{GS}\downarrow$ graph*

variants, making it a serious contender for use in commercial applications.

The outline of this chapter is as follows. Section 6.2 describes our novel SALT framework and algorithms. Experiments establishing the benefits of our approach are provided in Section 6.3. Finally, Section 6.4 gives conclusions and directions for future work.

6.2 The SALT framework

The main contribution of this chapter is to propose SALT (graph Separators + ALT algorithm), a unified framework for answering point-to-point, single-source (one-to-all, one-to-many and range) and especially k-NN queries which are not handled efficiently by existing approaches. The main advantage of SALT is, that the exact same data structures may service all the different type of SP queries and hence, SALT may be easily integrated into commercial, real-world applications. What follows is a detailed discussion of the SALT framework.

6.2.1 Preprocessing

SALT's preprocessing consists of two distinct phases, (i) the *graph-separator (GS) phase* and (ii) the *landmarks preprocessing phase*.

The *GS phase* of SALT mimics the preprocessing of GRASP [44] (see Fig. 5.1). During this phase, we use the Kafpaa/Buffoon [105] partitioning tool to create nested multi-level partitions of the road network graph in a top-down fashion. This initial partitioning phase is *metric independent* and needs to be executed only once, i.e., even in the case of arc-weights changes or for different metrics. Following partitioning, the *customization stage* builds the overlay graph H containing all boundary vertices and arcs of G. The graph H also contains a clique for each cell c: for every pair (u, v) of boundary vertices in c, we create a shortcut arc (u, v) whose cost is the same as the shortest-path (restricted to inner edges of c) between u and v (see Fig. 6.1(b), 6.1(c)). Similar to [44], we also calculate the SP distances between all border vertices of level ℓ and all vertices of level $\ell-1$ within each cell c^ℓ (see Fig. 6.1(d)). To differentiate between the two kinds of arcs computed, we will denote as (i) *clique arcs* the added overlay arcs that connect border vertices of the same level ℓ and (ii) *downward arcs* of level ℓ the vertices connecting different levels, i.e., ℓ and $\ell-1$. For added efficiency, *downward arcs* are stored as a separate graph, referred to, as $G_{GS}\downarrow$. Both types of arcs are computed bottom-up and starting at level one. To process a cell, the GS customization stage for SALT executes a Dijkstra algorithm from

each boundary vertex of the cell. We also apply the arc-reduction optimization of [47], which reports only distances of boundary vertices that are direct descendants of the root of each executed Dijkstra algorithm.

Although SALT's GS preprocessing phase is conceptually similar to GRASP, there are two major differences. (i) For accelerating SALT's preprocessing, H and $G_{GS}\downarrow$ have the same number of levels ($L = 6$ in our experiments) with $|C^L| = 16$ (cf. the original GRASP paper with $|C^L| = 128$ and $L = 16$). Using a smaller number of cells at the upper level of the cell hierarchy slightly lowers one-to-all query parallel performance, but *accelerates point-to-point queries and reduces preprocessing time*. Hence, it is a very logical compromise, since our focus is on increased versatility. (ii) Moreover, we have to repeat SALT's GS customization stage twice, one for the forward and one for the reverse graph. This is necessary for the landmarks phase of SALT, but it also allows to answer, both, forward and reverse single-source queries. Thus, at the end of SALT's GS preprocessing we have built two versions of the overlay graphs, H and $G_{GS}\downarrow$, one for forward and one for reverse graph queries, respectively.

The *landmarks preprocessing phase* for SALT extends the preprocessing proposed by [43], which optimized and tailored the ALT algorithm for use with dynamic road networks. Landmarks are selected by the *partition - corners* landmarks selection strategy, in which the we use the cells created by Kafpaa and from each cell we select the four corner-most vertices as landmarks. For SALT, we accelerate the computation of distances of all graph vertices from and to landmarks by executing two sequential GRASP algorithms (forward and reverse) instead of using plain Dijkstra (as in all previous approaches). Moreover, we may perform those $2|S|$ GRASP algorithms in parallel. By using these optimizations, the landmarks preprocessing phase of SALT never takes more than $4s$ *for 24 landmarks* and is therefore *at least* $6\times$ *faster than any existing work.*

Conclusively, at the end of the preprocessing stage of SALT, we have built the overlay graphs H and $G_{GS}\downarrow$ for both forward and reverse searches and calculated distances for all vertices from and to the selected landmarks. This is all the information required for answering point-to-point, single-source and k-NN queries. For *dynamic road networks*, we only need to repeat the GS customization stage and the computation of distances of all vertices from and to the landmarks. Both these phases *require less than* $19s$ for the benchmark road networks we used. This makes SALT suitable for dynamic scenarios.

6.2.2 Single-pair shortest-path queries

Using the SALT preprocessing data, we can accelerate point-to-point shortest-path queries by combining our custom CRP (with arc-reduction) with the goal-direction technique of the ALT algorithm. In CRP, to perform a SP query between s and t, Dijkstra's algorithm must be run on the graph consisting of the union of H, $c^0(s)$ and $c^0(t)$. The difference in SALT is that, instead of Dijkstra, we use the ALT algorithm on the graph consisting of the union of H, $c^0(s)$ and $c^0(t)$. Note that both ALT and CRP may also be used in, either, a unidirectional or a bidirectional setting. A similar combination of CALT [12] and CRP was unofficially introduced in [24], which uses the landmark derived lower-bounds only on the upper-level of the graph-separator overlay graph. Therefore, *local searches* could not be accelerated. Local search is crucial for k-NN queries, since the k-NN results for small values of k are usually located close to the query location. In contrast, our *SALT-p2p algorithm*, combining pure ALT and SIMD instructions for lower bound calculations and our custom CRP, is going to be significantly more efficient than stand-alone ALT or CRP.

In addition, since both methods are extremely robust to the metric used [12, 24], their combination will provide excellent performance for both travel times and travel distances.

Theorem 6.1. *The SALT-p2p algorithm for SPSP queries is correct.*

Proof. Since the SALT-p2p algorithm for SPSP queries combines CRP and ALT, which both provide optimal results then the SALT-p2p algorithm also provides correct results. □

6.2.3 k-NN queries

SALT's preprocessing data can be used to answer k-NN queries. Instead of initiating a k-NN search from a query location s to objects O, we start a search from all the objects at the same time *to* the query location in the reverse graph. Hence, we take advantage of, both, GS and ALT acceleration to guide the search towards the query location. The SALT-kNN algorithm's query phase may be divided in two independent stages. The *Pruning phase* excludes objects that cannot possibly belong to the k-NN set by using the upper and lower-bounds provided by the landmarks preprocessing data. The *Main phase* executes a unidirectional SALT-p2p algorithm in the reverse graph from all remaining objects at the same time to the query location until the query location is settled. Now we have found the first nearest neighbor. This process has to be repeated another $k-1$ times until all k-NN are discovered. The algorithm is detailed in the following.

Pruning phase. To prune objects that are too far away from the query location and thus cannot belong to the k-nearest neighbors set, we (i) calculate the k-th lowest upper-bound of graph distances between the query location and the objects (cf. Equation 2.2) and (ii) pruning/exclude objects whose distance lower-bounds between them and the query location (cf. Equation 2.1) exceed the k-th lowest upper-bound. To the best of our knowledge, this is the *first work to utilize upper and lower landmark bounds in the context of k-NN queries*.

Theorem 6.2. *The pruning phase of the SALT-kNN algorithm is correct.*

Proof. When we calculate the k-th lowest upper-bound of distances between the query location and the objects, we can guarantee that there are at least k-neighbours within this distance from the query location. So, any object located farther than that (as provided by the landmarks provided lower-bounds) may be safely pruned. □

```
getKthLowestUpperBound(s, O)
 1  Q = emptyMaxHeap
 2  m = 0
 3  for each obj in O
 4      if m < k
 5          Q.push(upperBoundDist(s, obj))
 6          m = m + 1
 7      elseif (upperBoundDist(s, obj) < Q.top())
 8          Extract − max(Q)
 9          Q.push(upperBoundDist(s, obj))
10  return Extract − max(Q)
```

To accelerate the process of computing the k-th lowest upper-bound between the query location and the objects we can use a bounded max-heap Q that only stores k-upper-bounds and procedure GETKTHLOWESTUPPERBOUND.

Since the bounded max-heap Q only stores k-upper-bound distances, we only need to compare the next objects's upper-bound with the top of the heap. If we have found a lower upper-bound, we remove the top of the heap and add the new upper-bound to Q. At the end of the procedure, the top of the max-heap is the k-th lowest upper bound of distances between the query location and the objects. The pruning phase is now implemented by procedure PRUNINGPHASE.

PRUNINGPHASE(s, O)

```
1  O_small = {}
2  kthUpperBound = getKthLowestUpperBound(s, O)
3  for each obj in O
4      if lowerBoundDist(s, obj) ≤ kthUpperBound
5          O_small.add(obj)
6  return O_small
```

At the end of the pruning phase, instead of using the objects in O, we only need to check for the k-nearest neighbors within the objects in O_{small}. Our experimentation has shown that the pruning phase is very effective, since it efficiently prunes more than 60% of the total number of objects in O.

Main phase. Following the pruning phase, to find the first nearest neighbor we start by performing a search simultaneously from all objects in O_{small} to the query location in the reverse graph. To do so, we use the idea of [83]. We add a new vertex T' connected to all objects in O_{small} using zero-weight edges and then perform a unidirectional SALT-p2p algorithm from T' to the query location s in the reverse graph (see Figure 6.2). At the end of this process, we have found the first nearest neighbor of query location s. Then we eliminate this vertex from O_{small} and repeat the process for another $k-1$ iterations to retrieve the full k-NN set (see procedure MAINPHASE).

Theorem 6.3. *The main phase of the SALT-kNN algorithm is correct.*

Proof. As [83] has shown, by adding a new vertex T' connected to a set of vertices and then running any correct shortest-path algorithm from T' to a destination vertex, we can find the minimum shortest-path distance between the vertices set and the destination. As a result, since the SALT-p2p algorithm is correct and this algorithm is run in the reverse graph, at the end of the first iteration we have found the shortest-path distance between the query location s and one of the objects in O_{small}, which is the first nearest-neighbour. Since, we eliminate this vertex from O_{small} and repeat this correct process for another $k-1$ times, the main phase of the SALT-kNN algorithm is correct. □

MAINPHASE($s, O_{small}, k, \overline{G}$)

```
1  for i = 0 to k−1
2      T' =new vertex
3      for each obj ∈ O_small
4          Connect T' to obj with zero-weight edges
5      (iNN, iNNdist) = SALT−p2p(T', s, Ḡ)
6      O_small = O_small − iNN
```

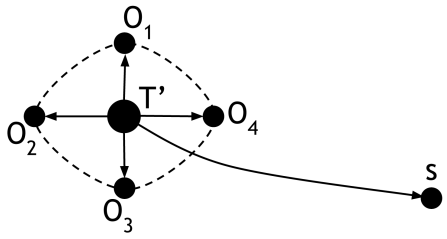


Figure 6.2: *The i-th iteration of the SALT-kNN algorithm*

To retrieve not only the shortest-path distance between the query location s and the objects in O_{small}, but also the actual k-NN vertices, we need to maintain for each labeled vertex a reference that points to the originating vertex in the objects' set O_{small}. Thus, when we extract the query location s from the priority queue and terminate the SALT-p2p algorithm at the i-th iteration, we know not only the i-th shortest-path distance but the i-th nearest neighbor as well. Moreover for each object o in O_{small}, we need to store the cell ID $c^1(o)$ of the cell this object belongs at the lowest level of the graph separator hierarchy, so to be able to traverse the overlay graph H during each iteration of the SALT-p2p algorithm. Note it is sufficient to store only the $c^1(o)$, since cell IDs for higher levels may be calculated from that.

Although the SALT-k-NN algorithm will be very fast for retrieving the first NN result object, it will become progressively slower when retrieving the additional $k-1$ NN vertices, since at each iteration, the SALT-p2p algorithm will start from scratch. To remedy this, at the beginning of the i-th iteration, we reload the corresponding priority queue with all vertices labeled during the $i-1$ iteration except those originating from the previous NN vertex found, since most of those labeled vertices were already assigned correct SP distances. For previously labeled vertices of which the SP distance can be improved, by using a min-heap priority queue (as all Dijkstra variants), the i-th iteration of the algorithm will further assign correct SP distances. This optimization significantly improves query times and still ensures correctness of the SALT-k-NN algorithm.

6.2.4 Single-source shortest-path queries

Although our main contributions are the SALT-p2p and the SALT-kNN algorithms for handling SPSP and k-NN queries, the SALT framework may still be used for other types of single-source shortest-path (SSSP) queries, including one-to-all, one-to-many and range queries by using the GRASP, reGRASP and isoGRASP algorithms presented in [44]. The major improvement in SALT is that by tweaking the number of cell levels ($L = 6$) and the number of cells at the upper cell level ($|C^L| = 16$), *we may efficiently answer both forward and reverse SSSP queries* without increasing the preprocessing time significantly. As shown in previous works [40], this type of flexibility is extremely important for a wide range of geomarketing applications.

6.2.5 SALT Tuning

Although SALT correctness for both p2p and k-NN queries was rather straightforward to prove, we describe some of the necessary optimizations for an efficient SALT implementation. These optimizations include the most optimal schema for storing shortest-path distances of graph vertices *from* and *to* the landmarks, the use of SIMD instructions dur-

ing the query phase of the SALT-kNN and SALT-p2p algorithms and how we assign IDs to vertices for fewer cache misses and acceleration of SALT's preprocessing and query phases. In detail:

Landmark Distance Records. In [43] landmark distances were stored in a 32-bit vector of size $2{\cdot}|S|{\cdot}|V|$. The distance of node with nodeID $i \in [0, |V|-1]$ from landmark number $j \in [0, |S|-1]$ was stored at position $2{\cdot}|S|{\cdot}i + 2j$ and the distance of node i to the landmark j was stored in the next position $(2{\cdot}|S|{\cdot}i + 2j + 1)$. Moreover, landmark distances *from landmarks to nodes* were stored negated (as negatives), since this is how they are used for estimating lower-bounds. Although this storage schema facilitates fast calculation of lower-bounds, it is not optimal for calculating upper-bounds, as needed during the pruning phase of the SALT-kNN algorithm. To calculate the upper-bound of the distance between the query location s and any object in O (cf. Equation 2.2) we only need the distances from the object *to* the landmarks and the distances *from* the landmarks to the query location s. Thus, it is better to store landmark distances *from* all landmarks on consecutive memory locations per vertex (and negated as before) and then the distances *to* all landmarks. Hence, we use again a 32-bit vector of size $2{\cdot}|S|{\cdot}|V|$ for storing the landmark distances, but now the distance of vertex i *from* landmark j is stored at position $2{\cdot}|S|{\cdot}i + j$ and the distance of node i *to* the landmark j is stored in the position $(2{\cdot}|S|{\cdot}i + |S| + j)$. With this optimization, to calculate the upper-bounds, we access $|S|$ consecutive memory locations per object instead of $2|S|$. Also, since landmark distances to vertices are stored negated, instead of addition, we use subtraction during the upper-bound calculation.

Node Reordering. Similar to previous works [14, 44], assigning smaller IDs to border vertices of higher levels and breaking ties within the same level by cell, has shown to improve performance for both SALT-p2p and SALT-k-NN algorithms. As a result, this is also the node-ordering of choice for our SALT framework.

SIMD Instructions. Current x86-CPUs have special 128-bit SSE registers that hold four 32-bit integers and allow basic operations, such as addition, minimum, and maximum to be executed in parallel. Efentakis et al. [43] have utilized those 128-bit SSE registers to significantly accelerate the computation of the landmarks based lower-bounds. We further expand this optimization by applying the above method to the efficient calculation of upper-bounds as well. Consequently, the pruning phase of the SALT-kNN algorithm requires significantly less than $1ms$. To the best of our knowledge, this work is the *first to utilize SIMD instructions within the context of k-NN queries*.

6.2.6 Summary and Expectations

Although our experimentation will show that SALT is very efficient for all types of shortest-path queries, the main phase of SALT-k-NN could be performed with any valid unidirectional SP algorithm. The use of SALT-p2p has multiple advantages (i) Its constituent algorithms, ALT and CRP, are the only algorithms with fast enough preprocessing times to be used for the case of *dynamic road networks*. SALT-p2p "inherits" this important property necessary for providing the optimal algorithmic foundation for live traffic-based services. (ii) The pruning phase of SALT-k-NN is very crucial for a fast implementation. *Only the landmarks preprocessing data could provide this type of functionality* that could potentially replace R-tree based approaches in other location-based services as well. (iii) SALT-p2p is very *robust with respect to the metric used*. In fact, its query performance is slightly better for travel distances, the metric for which most hi-

Table 6.1: *SALT, GRASP and G-tree preprocessing times*

	Preprocessing time (s)			
	Travel Times (TT)		**Travel Distances (TD)**	
	EUR	**USA**	**EUR**	**USA**
SALT (GS custom. phase)	11.1 (5.5)	14.82 (7.4)	11.3 (5.7)	15.4 (7.7)
SALT (Landmarks phase)	2.6 (1.3)	3.6 (1.8)	2.7 (1.4)	3.6 (1.8)
SALT (Total)	13.7 (6.9)	18.4 (9.2)	14.0 (7.0)	18.9 (9.5)
GRASP (Orig)	8 (8)	12 (12)	10 (10)	13 (13)
G-tree	(198,479)	(5,736)	(25,918)	(5,001)

erarchical SP approaches perform badly. This is an important property for k-NN queries identifying Points-Of-Interest based on walking distance. (iv) Our results show that unidirectional SALT-p2p actually provides better performance than bidirectional SALT-p2p. This is an advantage over existing hierarchical methods, since most can only be used in a bidirectional setting. (v) Finally, the main phase of the SALT-k-NN algorithm initially expands vertices closer to the query location. As such, "unattractive" objects furthest from the query location (as estimated by the lower-bounds) that cannot be excluded during the pruning phase do not slow down SALT-k-NN queries. In fact, experiments will show that finding the first nearest neighbor is almost as fast as a plain SALT-p2p query. Hence, it is hard to provide a significantly better theoretical solution, using standard SP techniques, with fast enough preprocessing times suitable for dynamic road networks.

6.3 Experiments

The experimentation that follows, assesses the performance of the SALT framework and the respective SALT-p2p and SALT-kNN algorithms. For completeness, we also report the performance of sequential and parallel GRASP [44] algorithm within the SALT framework for single-source (one-to-all) queries.

Experiments were performed on a workstation with a four-core i7-4771 processor clocked at 3.5GHz with 32 GB of RAM, running Ubuntu 14.04 64bit. Our code was written in C++ and compiled with GCC 4.8 (and OpenMP). Query times are executed on one core and augmented with SSE instructions. We used the European road network with 18M nodes / 42M arcs and the full USA road network with 24M nodes / 58M arcs [33] and experimented with both travel times and travel distances.

For partitioning the graph into nested-multilevel partitions, similarly to [44], we used Buffoon / KaFFPa [105] in a top-down approach. We use a partitioning setup similar to the best recorded CRP results of [22] with total number of overlay levels set to L=6 and $|C^1|$=1048576, $|C^2|$=65536, $|C^3|$=8192, $|C^4|$=1024, $|C^5|$=128 and $|C^6|$=16. We also used 24 landmarks, since adding more landmarks did not offer significant performance benefits for either SALT-p2p or SALT-kNN algorithms.

6.3.1 Preprocessing

In this section we will report the preprocessing times for SALT, in comparison to the original GRASP version (as reported on [44]) and G-tree [126] (G-tree source code was kindly provided by its authors). Note, that contrary to the SALT framework that may simultaneously answer single-pair, single-source (one-to-all, one-to-many, range) and k-

NN queries, GRASP only focuses on single-source queries and G-tree may only be used for undirected networks and k-NN queries. SALT and GRASP preprocessing times refer to parallel execution, using all available cores and G-tree preprocessing times are sequential. For GRASP and SALT and its graph-separator sub-phase we only report preprocessing times for the customization stage, similar to [24] and [44], since this is the preprocessing that must be repeated when edge-weights change, as in the case of live-traffic road networks. For a fair comparison, for G-tree we do not report the partitioning time required for the building of the G-tree index (which uses METIS [68]) and we only report the preprocessing time for calculating the SP distances between the vertices inside the respective index structure. Results are presented on Table 6.1. Numbers inside parentheses represent preprocessing times for undirected versions of the road networks.

Results clearly show that: (i) G-tree preprocessing times are very disappointing, especially for Europe and travel times, when more than 24h are required for preprocessing, which is in huge contrast with SALT's preprocessing time *which never exceeds 19s* for all networks and metrics. (ii) In comparison to GRASP, SALT may calculate both forward and reverse graph SSSP queries. If GRASP was to be extended for reverse graph SSSP queries, its preprocessing time would double and hence *it would be* 16−43% *slower than SALT*. (iii) SALT's preprocessing time *is very robust to the metric used* and preprocessing time is similar for both metrics. (iv) For undirected versions of the road networks (for comparing results to G-tree), SALT's preprocessing time drops in half, both for the GS customization and landmarks phase. Note that although SALT's total preprocessing time is better than any other previous ALT based approach including [43], the GS customization phase could be potentially further accelerated by using the optimizations of [30], such as SIMD instructions or contraction. But even without those potential optimizations, SALT still provides excellent preprocessing time, considering the fact that SALT may answer all variants of shortest-path queries on road networks.

6.3.2 Single-pair / single-source shortest-path queries

In this section we will describe unidirectional and bidirectional SALT-p2p query performance for single-pair shortest-path (SPSP) queries, compared to its individual algorithmic components, namely ALT [50] as augmented in [43] and our customized CRP [24] with the arc-reduction of [47], within the SALT framework. To that purpose, we executed 10,000 point-to-point queries with the pair of vertices selected uniformly at random. Regarding single-source shortest-path (SSSP) queries, we report sequential and parallel performance of GRASP for one-to-all queries within the SALT framework and compare it with the original version of GRASP (as reported on [44] on an almost identical setting to ours). For both GRASP versions, the number in parentheses represent sequential times. Results are presented in Table 6.2.

Considering SPSP query performance, results show that: (i) Unidirectional SALT-p2p is always faster than bidirectional SALT-p2p. This is in stark contrast with its individual components (ALT and CRP), in which bidirectional performance is significantly better. Thus, to the best of our knowledge, *uniSALT-p2p is the faster unidirectional algorithm for road networks, with preprocessing times of few seconds*. (ii) SALT-p2p is 100−266 times faster than ALT and 3−4 times faster than CRP. Note that our CRP's query performance is almost identical to the best CRP implementation of [22]. Moreover, SALT-p2p path unpacking (i.e., providing full paths) would also be faster than CRP, since it uses bidirectional ALT instead of bidirectional Dijkstra used by CRP [22]. (iii) SALT-p2p is

Table 6.2: *SALT-p2p and GRASP query performance*

	SPSP Query times (ms)			
	Travel Times (TT)		**Travel Distances (TD)**	
	EUR	**USA**	**EUR**	**USA**
biALT	103	60	133	89
CRP (+AR)	1.6	1.8	2	2
uniSALT-p2p	0.6	0.6	0.5	0.5
biSALT-p2p	0.9	0.9	0.9	0.9
	SSSP Query times (ms)			
GRASP (Orig)	43 (150)	58 (207)	46 (156)	66 (218)
GRASP (SALT)	50 (169)	65 (224)	53 (175)	68 (228)

impressively robust to the metric used. In fact, *uniSALT-p2p is slightly faster when we switch from travel times to travel distances*.

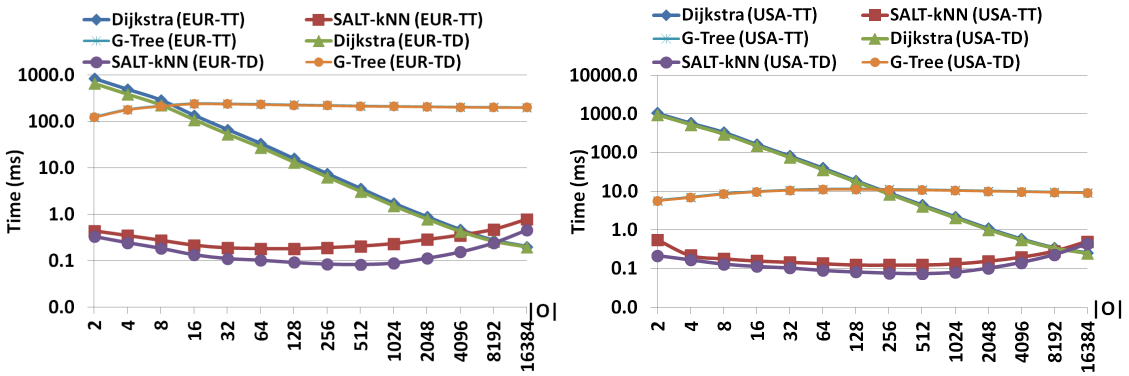
Regarding SSSP queries, the GRASP implementation within the SALT framework is $5-12\%$ slower for sequential and $3-16\%$ slower for parallel execution than the original GRASP implementation. Still, it is fast enough for most practical cases and the SALT framework may also execute forward and reverse SSSP queries, which is also a considerable advantage. Note, that the slightly less efficient implementation of GRASP within SALT is mainly attributed to the fact that now $|C_L| = 16$ (in comparison to $|C_L| = 128$ in the original paper). Still, setting $|C_L| = 16$ is the optimal setting for SPSP and k-NN queries, which constitute the most typical queries encountered in any shortest-path framework on road networks. In this sense, we decided to use this setting that benefits the most frequent type of queries.

6.3.3 k-NN queries

In this section, we compare performance between SALT-kNN, Dijkstra and G-tree [126], in the context of k-NN queries. For each experiment we generate 100 sets of random objects of varying size $|O|$ and for each such set we generate 100 random query locations, for a total of $10,000$ k-NN queries per $|O|$. The same $10,000$ queries per $|O|$ are executed for varying values of $k = \{1, 2, 4, 8, 16\}$ and we report average query times. Note, that G-tree also requires a *target selection phase*, for each set of objects $|O|$ (which takes almost $1.9-2.4s$). Thus, contrarily to both Dijkstra and SALT-kNN, G-tree cannot be used for moving objects. Results for $k = 1$ and $k = 4$ are presented in Fig. 6.3.

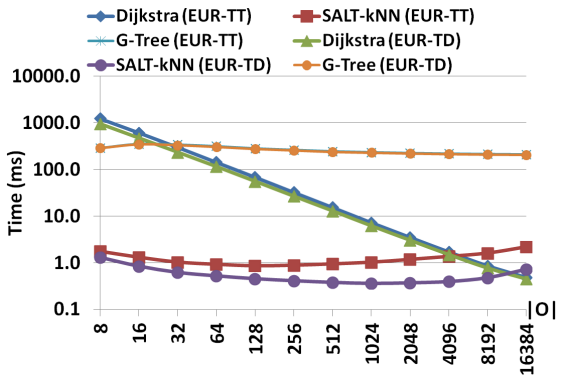
Results clearly show that SALT-kNN provides stable performance and query times significantly below $1ms$ for k=1. Contrarily, G-tree is almost *two - three orders of magnitude slower* and therefore cannot compete with either SALT-kNN or Dijkstra. Dijkstra starts very slow for small values of $|O|$ but manages to surpass SALT-kNN performance for $|O| > 8192$. This was inevitable to happen for any k-NN method, since *if the number of objects is significantly large and their distribution is random, an efficient Dijkstra implementation would only have to scan a few hundred nodes*. Still, since for static points of interest we are usually interested in a specific type of objects (e.g., gas stations) and in the case of moving objects we rarely have such large vehicle fleets (i.e., taxis, trucks) to monitor and we usually aim for k-NN queries among the *available* vehicles (a much smaller subset of total vehicles), then the SALT-kNN algorithm is surely to perform better for most practical applications.

After establishing the superiority of SALT-kNN over G-tree, in our second set of ex-

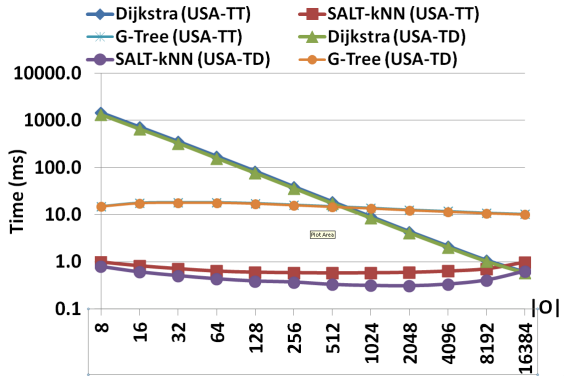


(a) Europe (k=1)

(b) USA (k=1)



(c) Europe (k=4)

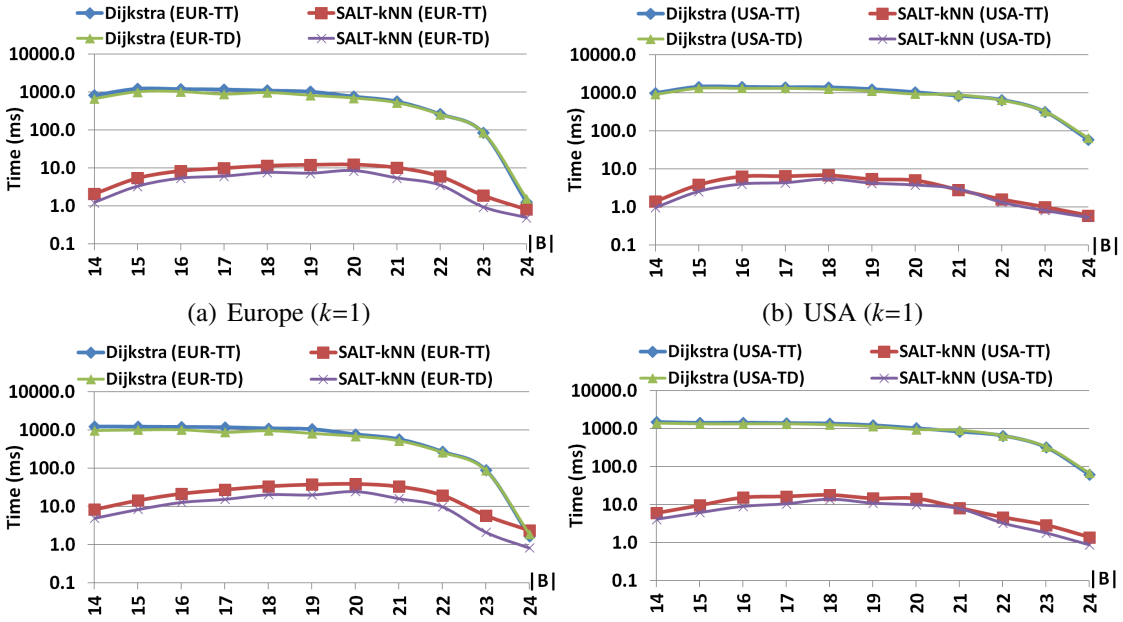


(d) USA (k=4)

Figure 6.3: *SALT-kNN, Dijkstra and G-tree comparison for* k=1 *and* k=4 *and varying values of* $|O|$.

periments, we evaluate the impact of objects distribution to SALT-kNN and Dijkstra's performance. To that purpose, we adapt the methodology of [26]. We pick a vertex at random and run Dijkstra's algorithm from it until reaching a predetermined number of vertices $|B|$. If B is the set of vertices visited during this search, we pick our objects O as a random subset of B. We keep the number of objects $|O|$ steady at 2^{14} and we experiment with different values of $|B|$ ranging from $2^{14} \ldots 2^{24}$, to simulate cases of either: (i) points of interest mainly located near the city-center or (ii) vehicle fleets which may service an entire continent but operate mainly on a particular country. Results for $k = 1$ and $k = 4$ are presented in Fig. 6.4.

Result show, that once again SALT-kNN provides excellent performance regardless of the object distribution, contrarily to Dijkstra which is 1-2 orders of magnitude slower when objects are not uniformly located in the road network (which is the typical case, either for static or moving objects). Thus, SALT-kNN is the only algorithm that guarantees excellent and stable performance, regardless of: (i) the number of objects and (ii) the objects distribution. Moreover, it does not need a target selection phase, such as G-tree or CRP and therefore, it may be used for either static or moving objects. Note, than even without building an index, CRP would still require $10ms$ for the target selection phase for 16384 objects for the Europe road network (as recorded in [29]) and therefore, CRP would *be at least 10 times slower than SALT-kNN for moving objects*.



(a) Europe (k=1)

(b) USA (k=1)

(c) Europe (k=4)

(d) USA (k=4)

Figure 6.4: *SALT-kNN and Dijkstra comparison for* $|O| = 2^{14}$, $k = 1$ *and* $k = 4$ *and varying values of* $|B|$.

6.4 Summary and Conclusions

This chapter presented SALT, a novel framework for answering shortest-path queries on road networks, including point-to-point, single-source (one-to-all, one-to-many, range) and k-NN queries. By combining ideas from the ALT, CRP and GRASP algorithms, the SALT framework efficiently answers point-to-point queries 3−4 times faster than previous algorithms of similar preprocessing times and answers k-NN queries orders of magnitude faster than previous index-based approaches. Moreover, the proposed SALT-kNN algorithm was shown to be especially robust, regardless of the metric used, the number of objects or the distribution of objects in the road network. Hence, it presents itself as an excellent solution for most practical use-cases.

Despite its excellent query performance, the most important advantage of the SALT framework is its flexibility and versatility with respect to the different variants of the shortest-path queries it services. The exact same data structures efficiently tackle a wide range of different shortest-path problems, with preprocessing time of only a few seconds, making SALT suitable for dynamic (live-traffic) road networks as well. To the best of our knowledge, there is no other framework that matches the benefits and versatility of SALT. We truly consider it the algorithmic version of a swiss army knife for shortest-path queries and the best overall solution for real-world applications.

Chapter 7

Towards a Flexible and Scalable Fleet Management Service

GPS positioning devices are becoming a commodity sensor platform with the emergence and popularity of smartphones. This abundance of GPS trajectories has fueled significant research around map-matching and related applications such as traffic assessment and prediction. Unfortunately, this research has only been used in costly and complex fleet management solutions. Our latest research endeavor addresses this issue by presenting cost-effective solutions for adapting state-of-the-art research around map-matching and live-traffic assessment in the context of fleet management applications. This chapter showcases various of our previous research results, wrapped in a single extensible fleet management platform. This chapter's content was initially included on our [38] publication presented in 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science, held in conjunction with the ACM SIGSPATIAL GIS 2013.

7.1 Introduction

GPS-tracking devices are becoming a commodity sensor platform with the emergence and popularity of smartphones. This abundance of usually low-sampling-rate (e.g., one point every 1-5 minutes) GPS trajectories have lead to a significant increase in research activities around map-matching, the process of aligning a sequence of observed user traces to the underlying road network graph. Nevertheless, so far, practical uses of this research have only been considered in costly and complex fleet management applications. On a quite similar note, novel shortest-path (SP) algorithms (by relying in extensive preprocessing of the road network graph) may answer SP queries in continental networks in few μs. Unfortunately, those algorithms are not efficiently tuned for handling live-traffic updates, such as those produced by the aforementioned map-matching algorithms.

Our latest effort in [109] tries to address these shortcomings by creating an efficient infrastructure for low-cost fleet management solutions. The core components of our system (referred hereafter as the *SimpleFleet service*) include (i) a collection mechanism for vehicle tracking data, i.e., Floating Car Data, (ii) a map-matching algorithm that relates the vehicle trajectories to an underlying road network and allows us to derive travel times in relation to the road network, (iii) an efficient data aggregation mechanism to derive speed profiles for the road network, (iv) a shortest-path algorithm that takes live-traffic conditions and actual travel times into account and (v) a visualization platform to interact with the system and visualize traffic conditions based on traffic maps and isochrones.

The outline of this chapter is as follows. Specific scientific innovations and a description of available system services is presented in Section 7.2. Section 7.3 describes the SimpleFleet service architecture and implementation. Section 7.4 presents the Web interface used to access (a subset) of the implemented fleet management functionalities. Section 7.5 presents some performance numbers and discussing possible system loads. Finally, Section 7.6 gives conclusions and directions for future work.

7.2 Services

Implementing a fleet management infrastructure requires a certain number of interconnected services, i.e., data collection and management methods, as well as implementing efficient map-matching and shortest-path algorithms. What follows is a description of these services and the respective innovations that were needed for their efficient implementation, in order to facilitate our SimpleFleet system as an efficient infrastructure for providing cost-effective fleet management solutions.

7.2.1 Data Collection

Essentially, we are dealing with two data sources. One is the actual road network (graph), which, in our case, is based on OpenStreetMap data. Since in our system we are dealing with specific geographic areas, we converted OSM data to a routable road-network graph, of which we finally used its largest strongly connected component. Strong connectivity is a necessary requirement for the map-matching and shortest-path algorithms used in the following. The second dataset we have to collect is the actual vehicle tracking data. Hence, we created an efficient mechanism for collecting and storing considerable amounts of Floating Car Data (FCD) from fleet vehicles. For each urban area covered by our system, we are typically dealing with 2,000 - 5,000 vehicles producing a data point (GPS position sample) every 60 - 180s.

7.2.2 Travel Time Derivation

Aligning the collected GPS traces to the road network graph requires state-of-the-art map-matching (MM) algorithms. In our framework, we use the Fréchet-distance-based curve matching algorithm of [17, 118] and the [67] implementation of the ST-matching algorithm [79]. Still, we had to significantly enhance both implementations to handle FCD streams. In our approach, we divide the incoming FCD stream to five minutes intervals, in order to create small trajectories and then performed map-matching on those small subsequences to obtain partial paths and travel time information.

Then we had to aggregate map-matching results per edge for the same interval (5min) to provide live-traffic assessment information. Map-matching results were also aggregated on a monthly basis per edge, weekday, hour and quarter-of-an-hour to build historic speed profiles for providing traffic information for areas with no available live-traffic data.

7.2.3 Shortest-Path Computation

The combination of live-traffic and speed profiles is used to provide dynamic shortest-path (SP) computation. To that purpose, we used our customized version of unidirectional ALT (A^* + Landmarks + Triangle equality) [50] algorithm, presented in Chapter 3. We

chose the ALT algorithm, since (i) it is very robust with respect to the metric used [50], (ii) it requires no path unpacking (producing the actual road network path of the shortest route), and (iii) its storage requirements and auxiliary data structure size depend solely on the number of landmarks (and not on the utilized metric). We avoided using hierarchical approaches such as Contraction Hierarchies [49] because there, the required shortcut edges need to be re-computed at every batched edge-weight update. Hence creating and dropping shortcuts every five minutes was an additional overhead we needed to avoid. Moreover, the use of shortcuts makes path-unpacking slower, since in our case the full path needs to be returned to the user.

Although there were some previous shortest-path research works [80] also using OpenStreetMap data for creating the road network graph, they did not have access to live-traffic information, as our work does. This is a huge advantage of our approach, since similar to major services like Google and Bing Maps, we are able to suggest the best route according to current live-traffic conditions.

7.2.4 Isochrone Computation

Another important focus of our work was to provide novel and innovative ways to visualize and represent traffic situation in urban areas. A crucial tool for this effort is the concept of isochrones. Isochrones are defined in [13] as the "*set of all points from which a specific point of interest is reachable within a given time span*". Although our service is mainly aimed towards fleet management, isochrones are equally important within other contexts, such as geomarketing (e.g. where a new franchise store should be opened) or urban planning (e.g. where a hospital should be built to accommodate uncovered city areas). Another paper relative to our work was [82], since it was the first to claim that the whole spatial area covered by an isochrone is important and introduced the "Edges' Hull" algorithm which creates a single area composed of the outermost edges of the isochrone network. This approach offers increased accuracy in comparison to convex hull approaches.

Although isochrones have been used before in public transport and walking combinations [13], to the best of our knowledge we are the first to combine the state-of-the-art isochrone computation of [82] with live-traffic data, in addition to providing this real-time system showcasing our results. Moreover, in our recent work of [40] (presented in detail in the Chapter 8) we have already combined the acquired live-traffic isochrone computation with demographic data to demonstrate the impact of traffic fluctuations in a geomarketing context. There, one can see in a quantitative way that the influence of live-traffic information is considerably important, especially in heavy traffic conditions. Hence, the live-traffic isochrones introduced for the first time here, may be extremely useful for many, seemingly unrelated, scientific areas.

7.3 System Implementation

The product of the aforementioned processes / innovations is the *SimpleFleet service* which consists of several components - virtual machines (VMs) that interconnect and cooperate (Fig. 7.1). The various components are described in the following.

All SimpleFleet system VMs are hosted in õkeanos [91] IaaS (Infrastructure as a Service) platform of the Greek Research and Technology Network. õkeanos is a cloud service comparable to Amazon Elastic Compute Cloud (EC2) making a potential migration of our entire architecture to such a commercial service simple and seamless.

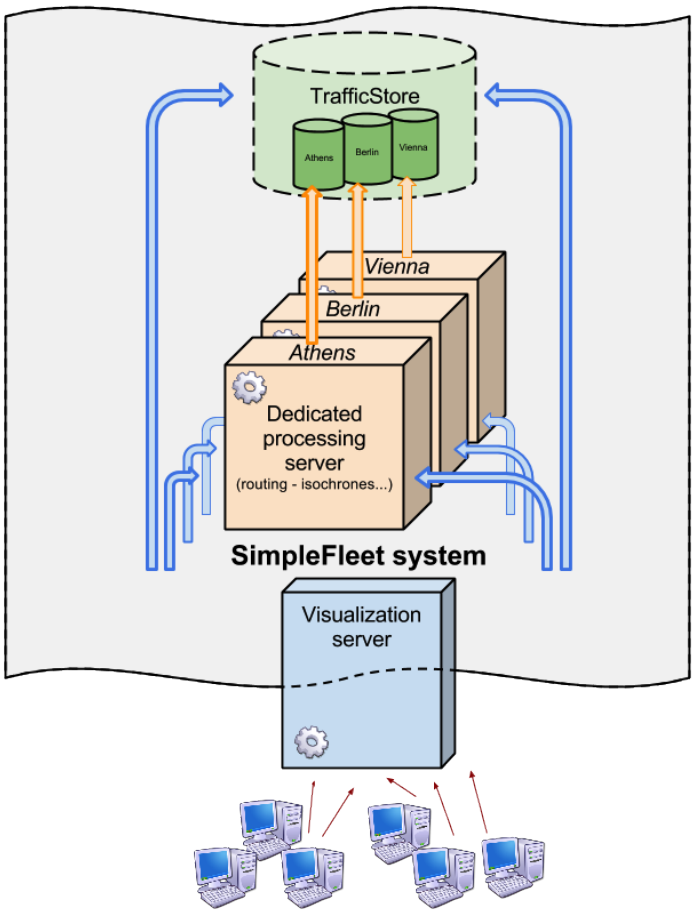


Figure 7.1: *The SimpleFleet service*

7.3.1 TrafficStore

The *TrafficStore* is the major key component of the SimpleFleet system and stores all available input and output data. Thus, all additional services are built on top of it. TrafficStore is a complete, integrated data management system for the traffic data-pool. It is implemented using a PostgreSQL / PostGIS DBMS. The data management functionality includes FCD collection, map-matching, computation of live-traffic assessment and speed profiles. A separate TrafficStore instance is set up for each area covered by our system. Currently our service covers the cities of Athens, Berlin and Vienna. Figure 7.2 gives an overview of the processes running in the TrafficStore and their interactions.

7.3.2 Dedicated processing server

Dedicated processing servers are used to handle computations in relation to user requests. Typically, one server (VM) is used per respective area. Such requests currently refer to (i) live-traffic shortest-path computation and (ii) calculation of isochrones (areas on the map that can be accessed within a given timespan). As shown in Figure 7.1, each processing server communicates only with the TrafficStore repository of its respective city. This was a deliberate choice, for ensuring maximum efficiency, isolation and scalability. The services are accessed using a RESTful HTTP API utilizing an Apache Tomcat server that can efficiently forward the requests to optimized Java algorithms that typically respond in less than 50ms, even for up to 500 concurrent users (see Sec. 7.5).

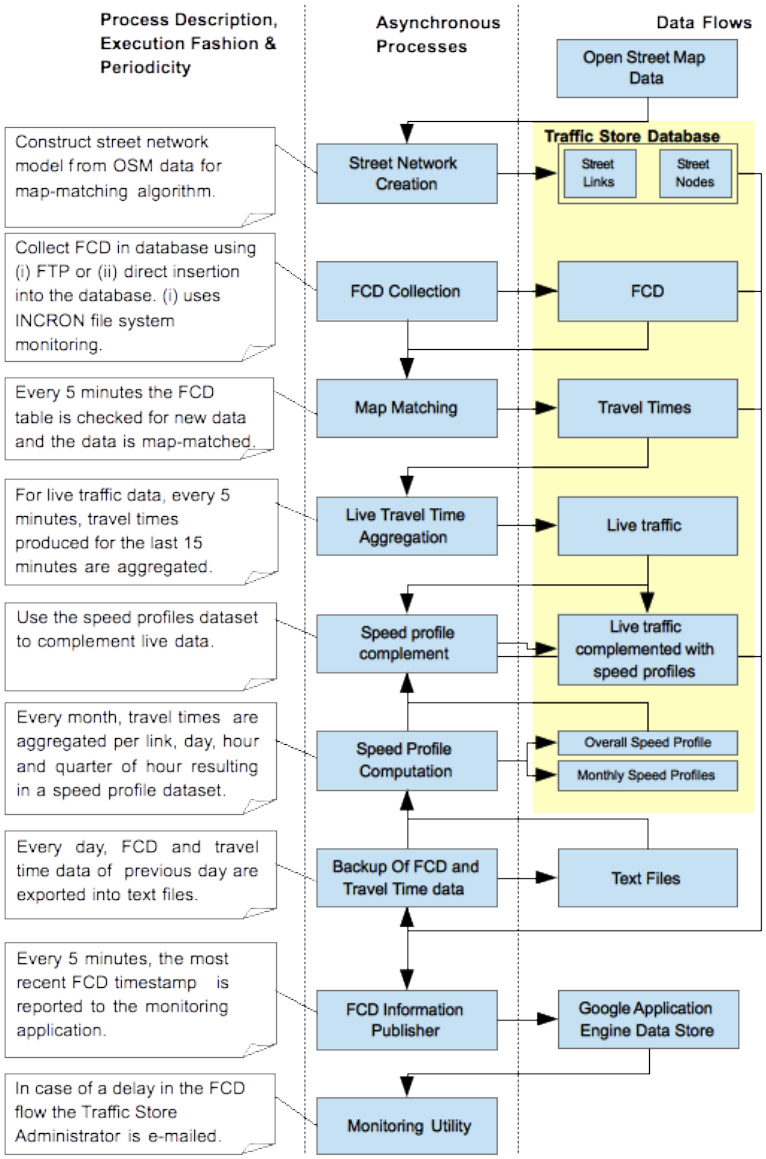


Figure 7.2: *TrafficStore functionality*

7.3.3 Visualization server

The *visualization server* supports the interactive Web front-end of the system (Sec. 7.4). The underlying architecture exposes most of its visualization functionality through APIs, allowing the service to be also used by third-party Web applications.

The online interface is powered by a Ruby-on-Rails (RoR) application and served via the Apache web-server through the Phusion Passenger library. This ROR application generates the main page that contains the map, as well as the administration panel that is used to easily monitor and alter configuration settings. All configuration settings (data sources, styles) for supported cities and layers are stored in a local Postgres database along with a collection of OSM resources for generating the base map tiles.

The interactive map interface relies on the OpenLayers JavaScript framework as front-end and the TileStache map tile caching server as the back-end. TileStache is a lightweight web server that utilizes the mapnik framework for converting vector data (stored in PostGIS) to image tiles, adhering to certain style rules (colors, line widths) specified in a CSS-

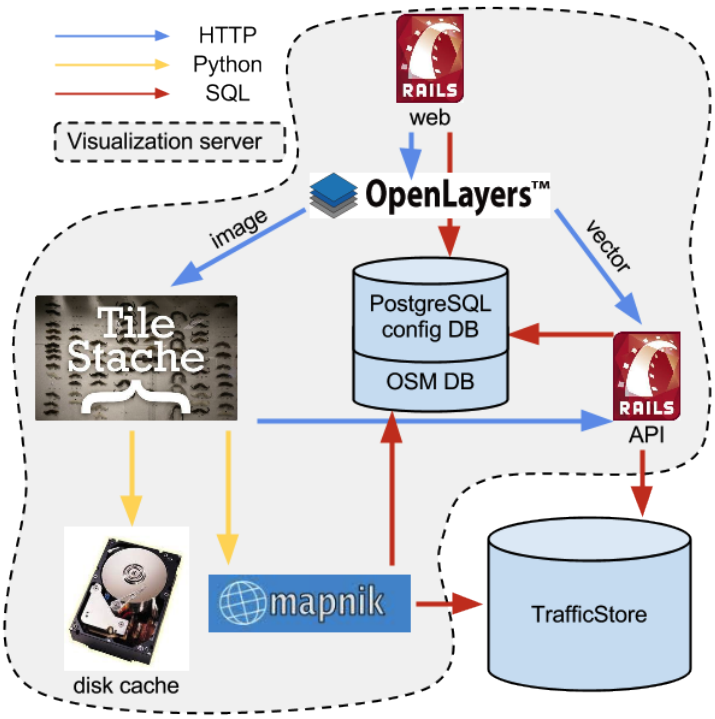


Figure 7.3: *The visualization server's components*

like format. Figure 7.3 depicts how these visualization server's components interconnect and cooperate. Note, that there is one common visualization server for all cities/urban areas covered by our system. In this way there is only one point of entry to the entire SimpleFleet service, which was a conscientious decision that ensures increased security and easier logging.

7.3.4 Expanding System Scope

The modularity of the first two components, i.e. the *TrafficStore* and the *dedicated processing server* and the easy configuration of the *visualization server* makes our system very easily extensible for covering additional geographic areas. To do so, the first step would be to prepare the respective TrafficStore for the new city. This essentially means obtaining the OSM data for the new region and preprocess it so that it can be used by the implemented algorithms for map-matching, shortest-path and isochrone computation. The second step, involves cloning a dedicated processing server and configure it to access data from the newly set TrafficStore instance. The third and final step is to add the new area to the *visualization server*'s configuration. This involves two major tasks: Initially we have to pre-render the map tiles for the new region, a task similar to the setup of the TrafficStore, since OSM data is being converted to map-tile images and then stored in a cache for faster access. Then the new region may be added to the configuration via the available administration panel; the basic set-up involves setting the URL of the respective dedicated processing server and then choosing which layers would be available for the newly added area.

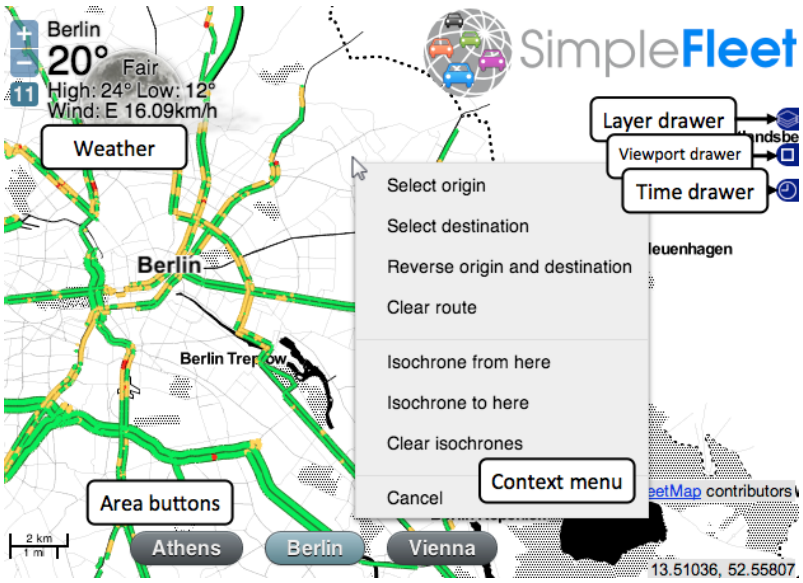


Figure 7.4: *The basic interface of the online demo*

7.4 Web Application

The Web front-end of our SimpleFleet service features an interactive "slippy map" interface that allows switching between the available geographic areas covered by the system. For each area the following data/services are available.

- Live-traffic map - visualization of traffic conditions, updated every 5*min*
- Speed profiles - visualization of traffic trends to complement live-traffic assessment
- Traffic message channel alerts (TMCs available only for Berlin)
- Isochrones - based on live-traffic
- Shortest-path routing - based on live-traffic

In terms of *data*, the first two layers, i.e., live-traffic and speed profiles are available as map tiles (png images), while information about the last three vector layers is available as JSON. From a *service* point of view, the first three layers are directly accessible from the TrafficStore, whereas isochrones and routing features are available from the respective dedicated processing servers (Sec. 7.3.2).

The three TrafficStore layers (live-traffic, speed profiles, TMCs) and the background road-map layer (the choices here are (i) the default black & white theme, (ii) Google Maps layer and (iii) OSM layer) may be independently activated by using the *Layer drawer* control located at the right of the map interface. The remaining two vector layers (routing directions and isochrones) may be activated by right-clicking anywhere on the map and selecting the appropriate action from the displayed context menu.

To minimize network time, all vector data from either isochrone or shortest-path routing responses is returned in Google's encoded polyline format [56] that achieves 90% compression. GZIP compression is also enabled, both on the the visualization and the processing servers, to further reduce network latency.

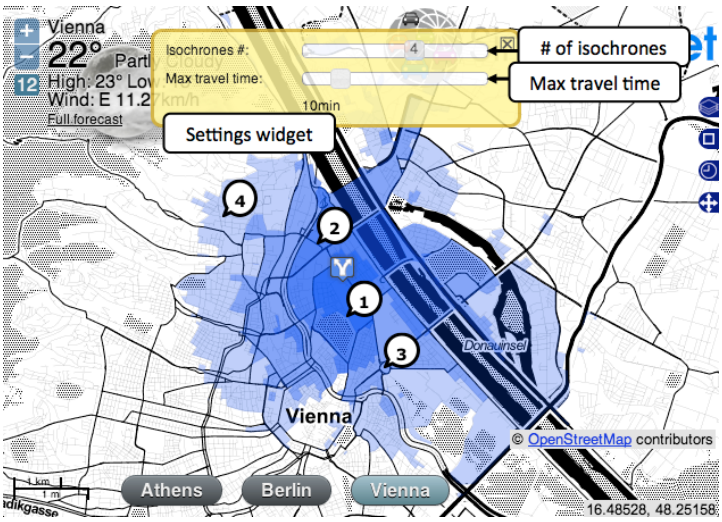


Figure 7.5: *Computing Isochrones*

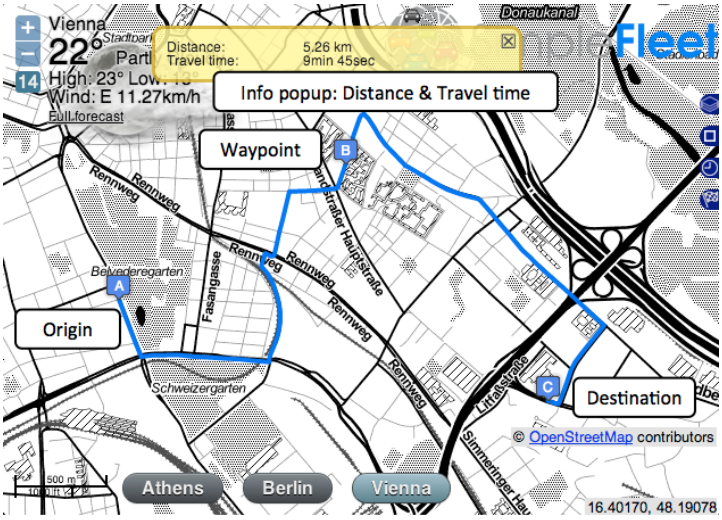


Figure 7.6: *A sample route between an Origin and a Destination passing via a specified Waypoint*

7.4.1 Isochrones

The Web interface facilitates isochrone computation "from" (and also "to") any location on the map (context menu). The user may either change the total traveled time (up to 30*min*) or the number of isochrone areas returned (up to six). For visualization purposes, each isochrone area has a different opacity with the smaller one being more opaque and the larger one being more transparent. Once the isochrones have been drawn on the map, the starting (or ending) marker may be dragged and dropped to a different location for requesting new isochrones to be drawn.

In our default setting, six isochrones are returned and the overall maximum travel time is set to 30 minutes. This means that each isochrone's maximum travel time is uniformly distributed in this duration, i.e., the first one covers the area reachable in 5*min*, the second one in 10*min*, etc. Figure 7.5 shows a tweaked example where the user has requested 4 isochrones and the maximum travel time for the largest one is set to 10*min*.

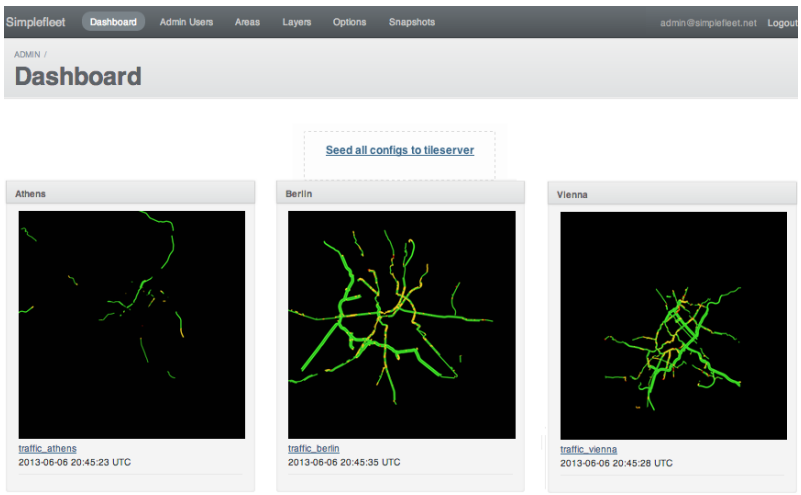


Figure 7.7: *The administration panel*

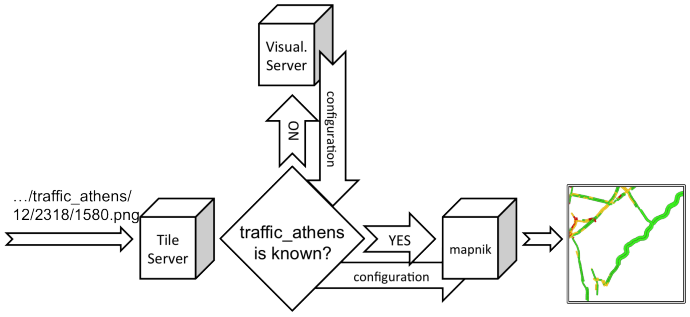


Figure 7.8: *Flow diagram of a request for a map tile.*

7.4.2 Routing

Similar to isochrone computation, routing requests may be computed between any two locations selected on the map (context menu). The server responds with a Google's encoded polyline representation of the calculated path, along with the travel time computed for this route and the total distance traversed. The travel info appears as a balloon tip on top of the map, while the actual route is drawn. Similar to massive online mapping services, the user may drag the Origin and Destination markers or add/delete intermediate points to a route.

7.4.3 Traffic Messages

The TMC layer shows Traffic Message Channel (TMC) alerts and is only available for Berlin. TMC alerts are short informative messages which appear in the electronic road signs above major roads and are broadcasted by conventional FM broadcasts. The user may click on the corresponding TMC icons and retrieve the respective message.

7.4.4 Administration

To help administer the service, i.e., add new cities and map layers, an online administration panel (available only to super-users via password authentication) was built (see Fig. 7.7). The main two data components available in this panel are *areas* and *layers*. Each area corresponds to a separate region and layers are used for storing configuration

parameters for the different type of data we want to display in the map area of the interface. The configurations settings are stored in the local PostgreSQL database of the visualization server.

The *admin panel* allows the super-user to add new spatial areas or hide/remove existing areas from the list of available ones via the "Areas" section. An administrator may easily specify the URL of the *dedicated processing server* that corresponds to each area, so that all requests for that respective area will be accordingly routed to the proper server. Similarly, one may also specify which layers will be available for each area via the "Layers" configuration page. This page allows a user to add or remove new layers, to enable or disable existing layers or change a layer's behavior, such as enabling or disabling caching for it. For image-based layers, the user has the ability to specify the data-source (a specific *TrafficStore* instance) along with the exact SQL query that will be used to fetch the results, as well as the specific styling rules that will be used for displaying the layer on the map. For example, for a traffic layer an example query would be to fetch all the road segments that are currently congested, i.e., have a travel time that indicates that vehicles are moving on it at 25% or less of the normal road speed.

Dynamic Configurations. The main advantage of having all the layer settings configurable via the *admin panel* is the ability to update the layer settings on demand with minimum effort. In a classic setup all the layer configuration such as the data-source to connect to, the exact query to fetch the data and the list of styling rules to apply when drawing the map tiles of that layer are saved in a static XML file on the disk. If a setting changes, one would typically have to log in to the server, edit and save the file, then restart the tile server (*TileStache*). In our setup we use the "Dynamic Layer" module provided by our tile server in order to read each layer's configuration from the *visualization server* the first time that a specific layer is requested. Once a "never-seen-before" layer is requested, an extra request is made to the *visualization server* to look up its configuration and export it to the tile server using a HTTP request. Figure 7.8 demonstrates the aforementioned scenario. The layer's configuration is then cached to boost performance. Since a typical single page request contains (on average) around 30 image tiles, it would be very inefficient to ask for the same configuration over and over again. A small patch has been applied to this "Dynamic Layer" module in order to allow this configuration to be updated on demand, i.e. by expiring the cache.

Overall, the user may change any setting of any layer and then use the link provided in the Dashboard section of the admin panel to seed this new configuration to the tile server, forcing it to expire any previously cached configuration and read the new settings from the *visualization server*. This approach is especially helpful when fine-tuning and testing new settings, e.g., trying out new map styles or queries.

Snapshots. For image-based layers (live-traffic and historic speed profiles), the administration panel offers a "snapshot" functionality which may be configured to run automatically at predefined intervals or requested to run on demand. The "auto-snapshot" process, which runs every 15 minutes, creates a zoom 11 snapshot of the entire bounding box of the layer. The Snapshot screen lists all the available stored snapshots, tagged with the time they were taken. The user has the ability to filter the displayed snapshots by layer and within a specified time range. The resulting snapshots may additionally be viewed as a slide-show (movie) by clicking the "Slideshow" button on the upper right corner (Fig. 7.9) creating visualizations of traffic patterns.

Figure 7.9: *Slideshow functionality of the administration panel*

7.5 Performance

Despite the SimpleFleet service's impressive features, we still have to test its performance. To that purpose, we used the popular regression-test tool Apache JMeter [6]. Apache JMeter is an open-source Java desktop application designed to stress-test functional behavior and simulate server load to analyze overall performance under different load types.

In our test scenario we experimented with 500 concurrent users, executing 60 mixed (for all cities) routing requests with a delay of 10s between requests to emulate realistic usage of the service, i.e., each user sends a request and waits for the response before proceeding to another request. Results showed that each of those 30,000 requests is answered in average time of 45ms. This clearly shows that *the minimal setting of our prototype architecture can efficiently handle a significant number of concurrent users*. Keep in mind that those recorded times are dominated by network latency, i.e., the time required for the actual response to be sent to the user. The actual time required for calculating a shortest-path is typically less than 5ms. After all, we aim at providing an infrastructure to service a limited number of fleet management companies and not on competing with well-established global mapping services, such as Google or Bing Maps.

7.6 Conclusion

This chapter described a unified fleet management system in terms of its available services, their implementation and its existing Web interface. The latter is used to access all provided functionality and showcases example services such as traffic maps, shortest-path and isochrone computation based on collected Floating Car Data. Our SimpleFleet system binds several web and server technologies together in order to provide a powerful and integrated platform that provides extensibility and scalability. We have also described its basic usage scenario, its administration panel, as well as its basic performance for a significant number of concurrent users.

Despite its, we believe, strong characteristics, this prototype application is still a work in progress. New services are being added to our infrastructure and the modularity of its architecture allows for the simple (comparatively) addition of new and probably more impressive features. Therefore we believe that it will play a crucial role in demonstrating our scientific results' huge potential. In the following chapter, we will demonstrate how the results of this service could be used in the context of geomarketing applications.

Chapter 8

Isochrones, Traffic and DEMOgraphics

Catchment area and reachability analysis, i.e., the area from which a location attracts visitors and the minimum distance to a target location, respectively, are interesting problems when studied in the context of time-parameterized networks, such as road networks affected by traffic. This chapter of the dissertation focuses on utilizing live-traffic assessment results produced by the SimpleFleet service (cf. Chapter 7) to such crucial geomarketing test cases. We combine state-of-the-art isochrone computation utilizing live-traffic and demographics data to provide efficient catchment area and reachability calculations. The online demo presented here, showcases the critical impact of live-traffic assessment on business intelligence decisions related to space. This chapter's content initially appeared on our [40] publication, presented in ACM SIGSPATIAL GIS 2013.

8.1 Introduction

Our latest research efforts described in Chapter 7, aimed towards applying state-of-the-art research about Floating Car Data, map-matching, historic speed profile computation, live-traffic assessment and time dependent shortest path computation to provide cost-effective fleet management solutions. This process required several intermediate steps such as: Creating road network graphs from OpenStreetMaps data, collecting a large amount of Floating Car Data (FCD) from fleet vehicles, applying state-of-the-art map matching algorithms on this data and consequently producing high-quality historic speed profiles along with frequently updated live-traffic assessment. This combination of live-traffic information and speed profiles will be subsequently used to provide up-to-date live-traffic shortest path computation (updated every 5 minutes).

As a result of these efforts, our integrated prototype SimpleFleet service is fully operational for three European cities namely: Athens (Greece), Berlin (Germany) and Vienna (Austria). Floating Car Data (FCD) for Vienna and Berlin originate from taxi fleets of 2000-5000 vehicles respectively, where for Athens FCD is provided from two commercial fleets of 1000 vehicles each. The online service presented in Chapter 7 and [38] clearly demonstrated the huge potential of our effort.

Although our main focus in [109] is mainly towards fleet management, the live-traffic assessment results produced, may prove extremely beneficial to other contexts as well. Such a context is geomarketing, i.e., the integration of geographical information into various aspects of business intelligence, such as marketing, sales and distribution. A crucial component in the effort of integrating traffic information in a geomarketing context is the concept of *isochrones*, which are informally defined as the area from which a specific

point of interest is reachable within a given time interval. In this sense, isochrones may provide accurate information about where to build a new franchise store to reach a larger pool of customers or identify areas less covered by existing stores. Since we are able to provide state-of-the-art isochrone computation based on live-traffic, this increased accuracy offers a unique advantage in comparison to typical static road network approaches.

To this end, we developed an additional geomarketing demonstration application (referred hereafter as SimpleFleet geomarketing demo) that combines live-traffic, isochrones and demographic / business data for Berlin and Vienna in order to demonstrate the actual impact of traffic fluctuations to business intelligence decisions. The geomarketing demo presented here, supports most modern Web-browsers and is available at `http://webgistu.wigeogis.com/protozone/gm/simplefleet/start.php`.

The outline of this work is as follows: Section 8.2 describes our scientific contribution beyond the current state-of-the-art. The implementation details of the demo are provided in Section 8.3. The actual demo and its interface are presented in Section 8.4 along with a summary of of the results obtained by the application and their significance. Finally, Section 8.5 gives conclusions and provides directions for future work.

8.2 Our Contribution

Isochrones are defined in [13] as the "set of all points from which a specific point of interest is reachable within a given time span". An important paper for the concept of isochrones and related to our work is [82], since it was the first to claim that the whole spatial area covered by an isochrone is important. In addition, that work introduced the "Edges' Hull" algorithm which creates a single area which is defined by a polygon composed of the outermost edges of the isochrone network. This approach offers increased accuracy in comparison to previous, typical convex hull approaches.

Although isochrones have been utilized in public transport and walking combinations in [13] and [48], to the best of our knowledge we were the first to combine the state-of-the-art isochrone computation of [82] with real live-traffic data (refreshed every five minutes). This functionality was already documented in [38] and [42]. Our aforementioned SimpleFleet service also features a *Visualization API* exposing its core functionality, so that isochrone computation may also be used by third-party apps.

In this work, we utilize the Visualization API and further expanded previous results by combining the live-traffic isochrones with demographics / business data for Berlin and Vienna provided by business partner WIGeoGIS. Hence, we are the first to provide accurate, substantial but most of all quantitative evidence (based on *our live-traffic data*, instead on relying on third party sources) about the impact of traffic in business intelligence decisions. Our results clearly show that this impact is much bigger than what it was expected (see Sec. 8.4.2). Therefore, approaches that either: (i) ignore traffic by assigning typical static speeds per road class or (ii) rely on conventional convex hull approaches, are inherently imprecise by a big margin, since they greatly overestimate the area from which a specific point of interest is reachable.

8.2.1 Methodology

The SimpleFleet service provides access to its data via a simple REST endpoint (i.e., the Visualization API), which is publicly accessible. For the purposes of the geomarketing demo, live-traffic isochrone data for Vienna and Berlin was requested in regular intervals

of 15 minutes, from April 26th 2013 until May 31st 2013, using the provided API. Each such isochrone request returns six isochrones with travel times starting from 5 minutes (for the first isochrone) and a 5 minutes step, so that the last isochrone covers a travel time-span of 30 minutes.

All six isochrone multi-polygons are subsequently stored in a database along with a timestamp and a location identifier (Berlin, Vienna). Then a separate offline process takes place and calculates the values for population, households, banks and retail stores for each such isochrone record. The calculated value is merely a count, derived by overlaying each isochrone multi-polygon with a $250x250m$ raster containing population data. Banking and retail store information is extracted from business directories for each city and were provided by commercial partner WIGeoGIS.

Since *a picture is worth a thousand words* the better way to showcase these results is through an intuitive Web application publicly available. To that purpose, in the next section we will describe the several geospatial and Web technologies required for actually building and servicing our geomarkerting demo.

8.3 System Implementation

Our demo web application is built with standard modern technologies for managing and displaying spatiotemporal data online. For our persistent storing mechanism, we utilized a Postgis enabled PostgreSQL database. PostgreSQL is a well known, powerful open source relational database management system. In addition, PostGIS "spatially enables" the PostgreSQL server, allowing it to be used as a backend spatial database for geographic information systems. Hence, their combination provides excellent performance and efficient spatial calculations. The Javascript front-end of the demo is designed with DHTMLX, a rich Javascript library that delivers a complete set of UI components and has built-in AJAX support for fetching data asynchronously from the remote server. The bar chart (Fig.8.1) displaying the data, uses D3JS, which is a powerful Javascript library for manipulating documents based on data. The interactive map is powered by OpenLayers, a free Javascript mapping framework that allows creation of dynamic maps on any web page. The map theme is called "Toner" and was designed by "Stamen" (a design and technology studio based in San Francisco). Still, the OpenLayers library would easily work with any other available WMS source.

The demo's front-end consumes data from the database via simple PHP scripts, using AJAX requests. To fetch the corresponding data, each request requires the following four parameters:

- The travel time span of the isochrones in minutes (5, 10, 15,. . . , 30 min)
- A specific date (between April 26th, 2013 until May 31st, 2013 - the time period covered by the demo)
- A time range within the specified date (e.g., between 08:00 and 17:30)
- The demographic / business data type requested (population, households, banks or retail stores)

As described in Section 8.2, all available data is stored as a single table in the underlying PostgreSQL database, so the query may efficiently select all records matching the

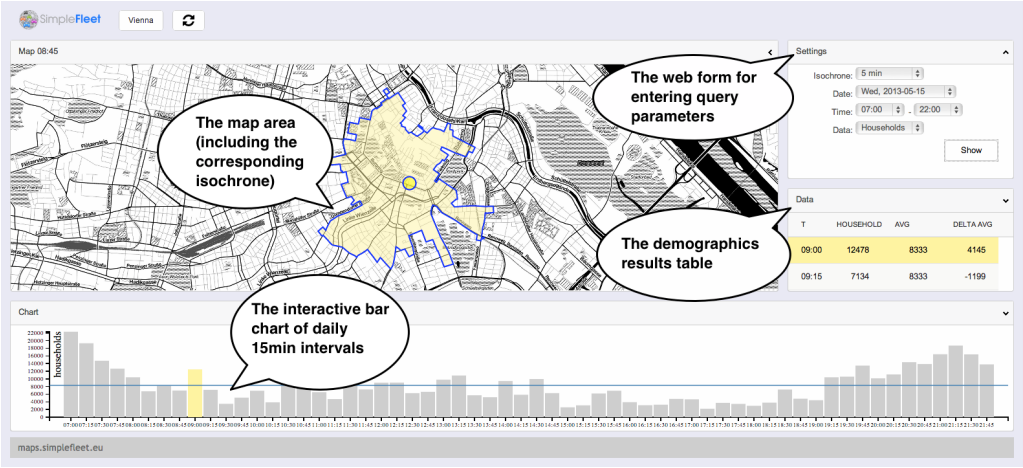


Figure 8.1: *Vienna city center, on Wednesday 15th May 2013, at 09:00 am. Approximately 12k households are reachable within five minutes from the city center at that time.*

user's specified criteria. The corresponding server's response is an array of objects (JSON format), each containing the following information:

- The timestamp value,
- The demographic / business data absolute count,
- The demographic / business data average count,
- The delta between the average and the absolute count,
- The travel time of the isochrone (5, 10, 15,. . . , 30 min)
- The ID of the stored isochrone multi-polygon in the database.

The isochrone itself is not included in the response for bandwidth and performance reasons. To actually draw the isochrone on the map, when required, another AJAX call is made, using the isochrone ID as its only parameter. The multi-polygon returned is encoded using the WTK (Well Known Text) representation.

8.4 Online application

The geomarketing demo is an interactive web application that showcases one of the many potential uses of the produced results of the SimpleFleet service, in scenarios not directly related to fleet management. *Population*, *households*, *banks* and *retail stores* data is correlated with isochrones to provide valuable information, such as catchment areas (for population and household data) and reachability analysis (for bank branches and retail stores). The demo focuses on this kind of data and intuitively demonstrates how it changes over time through the fluctuation of traffic; within a month period or even during a certain day. This visual representation easily showcases the benefits of computing isochrones using live-traffic data as opposed to a typical traffic-less static road network approaches.

8.4.1 Interface

The demo is available online as a single page web-application. When it loads up, the demo's interface is divided into four main areas (see Figure 8.1):

- A *web-form* for setting the query / isochrones parameters
- An interactive *bar chart* of daily 15min intervals
- The *demographics results table*, displaying the demographic/ business data contained inside the isochrone area
- The *map area* (including the isochrone multi-polygon)

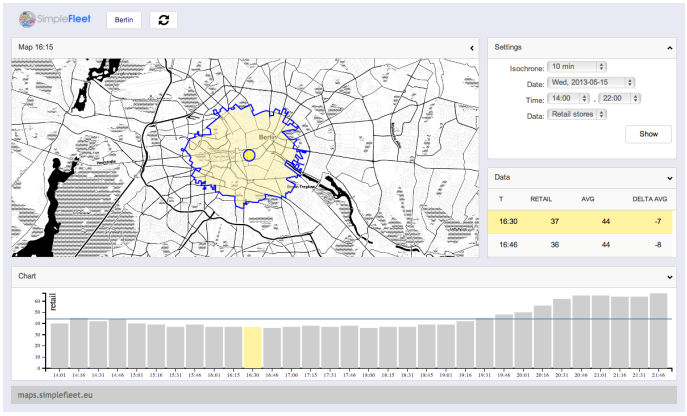
On the top of the page there is a drop-down menu that allows the user to choose between the two available cities: Vienna (set as the default area) and Berlin. The user may set the desired options / request parameters (see Section 8.3), in the web form located in the upper right corner of the screen and click the "Show" button. This action will submit the form via AJAX and return the respective demographic / business counts requested. When the AJAX call completes, the newly retrieved data is displayed in the table panel and simultaneously gets graphed in the interactive bar chart area located at the bottom. In the bar chart area, the horizontal axis represents the timestamp during the selected day and the vertical axis shows the count of the demographic / business data requested. There is also a horizontal line marking the respective average count that spans the entire graph. Finally the requested isochrone is drawn in the map area.

Both the bar chart and the results table are interactive in the sense that hovering the mouse over a specific bar or row will immediately cause the matching isochrone to be drawn on the map. One can hover the mouse in a horizontal smooth manner over the graph and observe a beautiful isochrone sequence on display, that showcases the impact of traffic (during the day's duration) to the respective isochrone coverage. Results of this impact, will be further quantified in the next section.

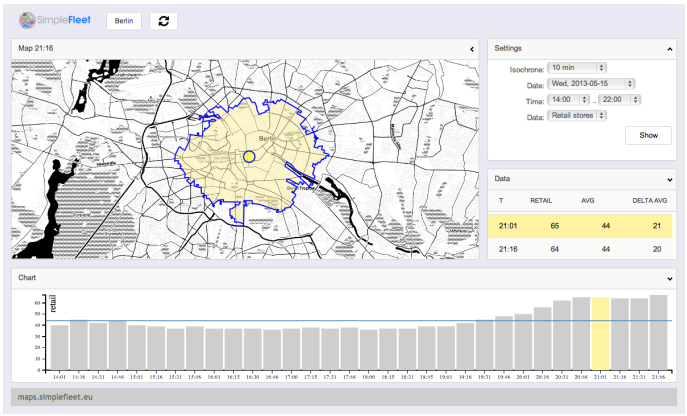
8.4.2 Results

Although our geomarketing demo clearly demonstrates the impact of traffic to typical business intelligence decisions for Berlin and Vienna in a concise, clear and intuitive way, it is beneficial to further quantify the statistics presented in the demo. This way we may easily answer questions, such as: *"To what extent does traffic affect reachability analysis"* or *"To what degree typical traffic-less geomarketing approaches are erroneous"*. The results that easily surpassed our expectations, are presented in Figures 8.3 and 8.4.

Figure 8.3 shows that for both Berlin and Vienna, *if we take traffic into account and for the 5 minute timespan - isochrone, we reach less than 15% of the potential customers we would have calculated on a static traffic-less road network graph*. Although this gap decreases for larger timespans, still for a timespan of 15 minutes the impact of traffic is more then 40% for Berlin and 20% for Vienna, i.e., the number of actual customers we reach within 15min is actually 20-40% smaller than the number calculated by the typical traffic-less scenario. If we also consider the fact that typical geomarketing applications employ convex-hull approaches (which further overestimate the respective areas, as shown in [82]), we see that our proposed approach to business intelligence issues is significantly more accurate and realistic.



(a) Evening at 16:30



(b) Night at 21:00

Figure 8.2: *Berlin city center, on Wed 15th May 2013. Depending on the time of the day there is a significant change in the covered area. This clearly shows the benefits of computing isochrones based on live traffic.*

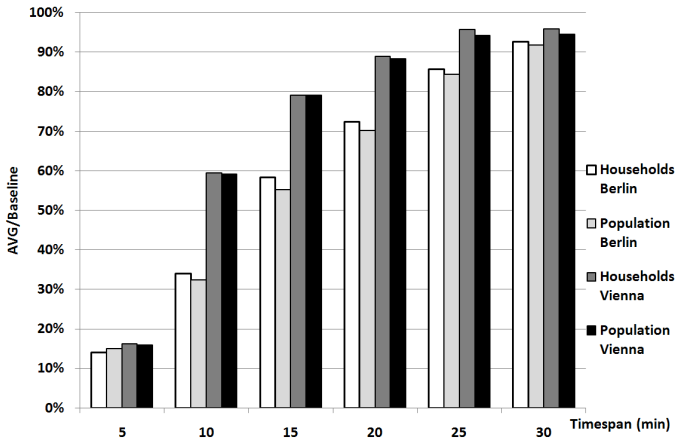


Figure 8.3: *Average values (typical traffic situation) in comparison to traffic-less values for different timespans - isochrones*

This is also evident in Figure 8.4 which compares the values acquired from the demo in case of traffic congestion with again the same baseline of a static traffic-less road network graph. Here the impact of traffic is even more dramatic, i.e., even for a timespan of 15 minutes the traffic's impact is more than 60% for both cities, i.e., the potential customers we may reach are lowered by an additional 20-45% (compared to Fig. 8.3) in case

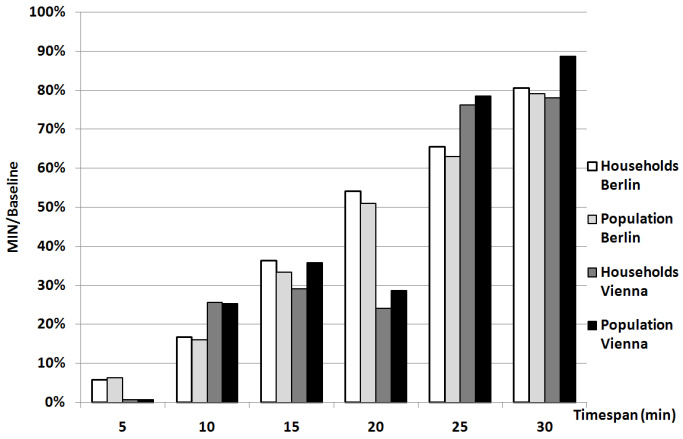


Figure 8.4: *Minimum values (traffic-jams) in comparison to traffic-less values for different timespans - isochrones*

of traffic congestion. Since stores' opening hours usually coincide with traffic rush-hours, the worse case scenario of Fig. 8.4 might provide an even more realistic picture of how traffic decreases the potential pool of customers we may reach in a geomarketing decision.

8.5 Conclusion

In this chapter, we have described the SimpleFleet geomarketing demo and presented its basic design, implementation and core functionality. Still, despite its responsiveness, ease of use and the rest of its impressive characteristics, the demo's most amazing feature is the actual results: We have shown in a concise, accurate but most of all quantitative way, the huge impact of traffic to informed business intelligence decisions. Moreover, we have clearly described the methodology of acquiring those results, by combining live-traffic information with state-of-the-art isochrone computation and demographic / business data. In the following chapter, we will further take advantage of the results of the service described in Chapter 7, to infer information about existing turning restrictions in the road-network dataset.

Chapter 9

Crowdsourcing turning-restrictions from map-matched trajectories

The abundance of relatively cheap GPS-enabled devices has generated massive amounts of GPS-tracking data produced by vehicles traversing the road-network. Unfortunately, this data has not been effectively used for identifying turning restrictions in the underlying road-network graph. In this chapter, we propose a novel, efficient and straightforward method to deduce turning restrictions for OpenStreetMap data, by using historic map-matched trajectories from an existing fleet management service. Our extensive experimental evaluation and verification process, using online map-services, satellite imagery / street view and public APIs from two commercial map-vendors proves the efficiency and reliability of our proposed method. This chapter's content is based on our [39] publication presented in Mining Urban Data workshop co-located with EDBT/ICDT 2014 and the extended version of this work submitted to the Special Issue (Mining Urban Data) of the journal of Information Systems (Elsevier).

9.1 Introduction

Our research efforts of [109] and Chapter 7 proposed an optimal workflow for combining state-of-the-art research about road networks, Floating Car Data (FCD), map-matching algorithms, historic speed profile computation, live-traffic assessment and time-dependent shortest-path computation in the context of fleet management solutions. Its end-result, our fully functional SimpleFleet fleet management system was implemented, focusing on the urban regions of three major European cities namely: Athens (Greece), Berlin (Germany) and Vienna (Austria). According to the proposed workflow, several intermediate steps were required such as: Creating road-network graphs from OpenStreetMaps data, collecting huge amounts of Floating Car Data from fleet vehicles, applying state-of-the-art map-matching algorithms to align the observed GPS traces to the road-network graph and consequently producing high-quality historic speed profiles along with frequently updated live-traffic information. This combination of live-traffic information and speed profiles was subsequently used to provide up-to-date, live-traffic, shortest-path and isochrone computation (refreshed every 5 minutes), using the shortest-path implementation presented in Chapter 3. Moreover, our follow-up work of [40] (cf. Chapter 8) clearly showcased the impact of traffic fluctuations in a geomarketing context, by combining the live-traffic isochrone functionality of this system with demographic data.

Since our efforts were focused on improved efficiency and low costs, the SimpleFleet

system used OpenStreetMaps (OSM) data for constructing the road-network graphs. Still, running this service for more than a year and for the three urban regions has revealed an inherent limitation of the OSM dataset: Its limited information for turning restrictions, i.e., a transition from one network edge to another (via an intersection node) that is prohibited due to local traffic rules. Although OSM supports turning restrictions by using an additional relation tag (Relation:restriction [93]), only a small number of users contribute to this information. This is particularly evident, considering that OSM includes more than 2.1 billion Nodes, Ways and Relations [92] and less than 230,000 relations actually represent turning restrictions [93]. Our individual test cases confirm this observation: For the Athens area and its 277K vertices road network, only 214 turning restrictions have been recorded by OSM users. This observed lack of data is mainly attributed to the fact that there are no public datasets for traffic signs easily found (if any), satellite imagery cannot testify to the existence of such restrictions and contributing turning restrictions even for a single road to the OSM dataset may be extremely time-consuming.

Despite the fact, that turning restrictions are especially important for any public mapping service, there is only a limited number of scientific literature focusing on them, mostly due to the fact that *"no publicly-available realistic turn data exist"*[24]. This is truly surprising, since turning restrictions have a more dramatic impact on shortest paths provided by mapping services than traffic: While ignoring traffic returns a suboptimal, yet valid route to the user, ignoring turning restrictions provides erroneous paths that may lead to accidents. Thus, providing a semi-automatic method for identifying turning restrictions is extremely important for any public mapping service.

During our research on related work, we found a significant body of work focusing on Floating Car Data (FCD) (see [125] for a partial overview on GPS related research). The only previous works relevant to solving (or even acknowledging) our actual problem also use FCD for calculating turn delays [8, 78, 111, 122, 124]. However, no scientific literature exists that utilizes map-matched (MM) trajectories to derive turning restrictions. As such, this chapter presents the first approach that automatically identifies and infers turning restrictions based on historic map-matched trajectory datasets, i.e., we crowdsource the identification of turning restrictions to local vehicle drivers, by mining the map-matched trajectories produced by them when their vehicles traverse the road network. This approach has several benefits in that map-matched trajectories are (i) more condensed, i.e., instead of random locations in the plane we use edge sequences in a graph and (ii) less ambiguous and susceptible to errors, i.e., movement is interpolated using the actual road network. Although this work focuses on OpenStreetMaps data, it may also be used for any road network dataset, i.e., for cases in which the road network evolves faster than commercial map updates.

The outline of this chapter is as follows. Section 9.2 describes our scientific contribution towards identifying turning restrictions in the OSM dataset by utilizing historic map-matched trajectories. Section 9.3 summarizes the results of our approach. Finally, Section 9.4 gives conclusions and directions for future work.

9.2 Crowdsourcing Turning Restrictions

The core contribution of this work is a methodology for extracting turning restrictions from historic map-matched trajectories. In process, we also describe the OpenStreetMap road-network dataset and its properties.

(a) Prohibited U-turn (b) Prohibited left turn (c) Prohibited right turn (d) No entry sign

Figure 9.1: *Prohibitory traffic signs for turning restrictions*

9.2.1 Definitions and Preliminaries

In the discussion that follows, a road-network is represented as a directed weighted graph $G(V, E, w)$, where V is a finite set of vertices / nodes, $E \subseteq V \text{x} V$ are the edges of the graph and w is a positive weight function $E \rightarrow R^+$. Typically the weight w represents the travel time required to traverse the edge. In other cases, w may refer to the length of the edge in meters (for travel distances metric).

The degree of a vertex u, denoted as $deg(u)$, is the number of edges incident to the vertex. *Intersection nodes* are the road-network vertices with node-degree larger than two, i.e., $I = \{v_i \in V, deg(v_i) > 2\}$. A *turning restriction* is an ordered sequence of two or more network edges connected via intersection nodes that is prohibited due to local traffic rules. Drivers are alerted for existing turning restrictions through standardized traffic signs (see Fig. 9.1). In this paper, we only cover those edge sequences that consist of *a single ordered pair of two edges connected via a single intersection node*. This constellation represents the majority of turning restrictions in typical road-networks. Note that turning restrictions do not refer to one-way streets, because (i) even a single edge may be marked as unidirectional and (ii) turning restrictions may refer to roads that are bidirectional, but it is only their sequential traversal that is prohibited. Moreover, unidirectional streets are easily modeled in every directed graph representation, whereas *turning restrictions are a distinguishing characteristic of road-networks*, which differentiates them from other types of networks.

9.2.2 OpenStreetMap and Turning Restrictions Coverage

OpenStreetMap (OSM) provides unlimited and free access to the entire map dataset under an Open Database License [1]. This massive amount of data may be downloaded in full, but is also available through APIs and Web services. Users may participate in the OSM community by providing their local knowledge-based feedback and edit the map. Although OSM contains a relation tag (Relation:restriction [93]) for describing turning restrictions, only a small number of OSM users are aware of this property. This fact was easily confirmed for the cases of three European cities (Athens, Berlin, Vienna) covered by our service. The results for September 2013 presented in Table 9.1 show that the available data for turning restrictions is low when compared to road-network sizes of Table 9.2. We obtained similar or worse results for other European cities, especially for countries with less extensive coverage (e.g., Albania, Montenegro).

Using map-matched trajectories, our method for discovering turning restrictions identifies turns that, although allowed in the original map dataset, are rarely executed by

[1] http://opendatacommons.org/licenses/odbl/1.0/

Table 9.1: *Turning restrictions added in OSM per year for the cities covered by our service*

city	2009	2010	2011	2012	2013	Total
Athens	-	11	1	75	127	**214**
Berlin	8	26	101	386	147	**668**
Vienna	33	36	99	307	324	**799**

Table 9.2: *OSM road-networks of the three cities covered by our service*

			# intersection vertices		# intersection vertices for roads ≤ 10	
city	**# vertices**	**# edges**	**total**	**%**	**total**	**%**
Athens	277,719	329,444	100,422	26%	34,921	13%
Berlin	89,598	103,486	51,935	58%	21,119	24%
Vienna	100,579	112,478	44,874	45%	16,104	16%

drivers. Exhibiting such an unusually low frequency, such turns have a very high probability to be actually prohibited. Essentially, our approach is based on crowdsourcing driver behavior as evidenced by their tracking data.

9.2.3 Methodology

Our basic methodology for inferring / identifying OSM turning restrictions can be described by the simplified diagram of Figure 9.2. The following sections will elaborate on the independent stages of this process.

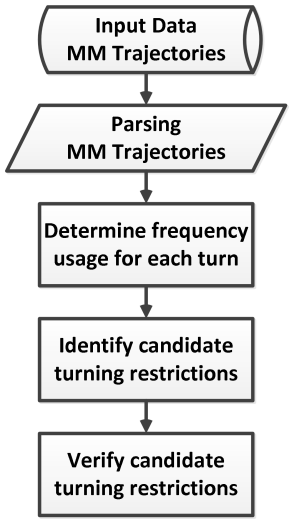


Figure 9.2: *Methodology for identifying OSM turning restrictions by using historic map-matched trajectories*

9.2.3.1 Input data (Map-matched trajectories)

In our system GPS-traces of fleet vehicles for the three European cities arrive in a streaming fashion. For each geographic area, we monitor $2,000-5,000$ vehicles producing a GPS position sample every $60-180$s. GPS trajectories for each vehicle are subsequently map-matched. The end-result of this process is an ordered sequence of road edges that

Table 9.3: *Road categories for the OSM road-networks*

CategoryID	Road category	Athens	Berlin	Vienna
1	motorway	4287	1420	2410
2	motorway link	3747	2012	4386
3	trunk	1343	111	171
4	trunk link	567	0	227
5	primary	16210	5203	8913
6	primary link	1257	347	422
7	secondary	42881	21250	12894
8	secondary link	0	45	0
9	tertiary	58722	9678	11576
10	tertiary link	0	6	0
11	unclassified	13484	2792	3060
12	road	395	28	0
13	residential	186459	58338	67482
14	living street	92	2256	937

Table 9.4: *Typical size of compressed MM trajectory archives*

Size	Athens	Berlin	Vienna
per day	22.3 MB	224 MB	76.3 MB
per month	0.67 GB	6.74 GB	2.29 GB

each vehicle has traversed. The traffic datastore of the service, including also the OSM road-network graphs, Floating Car Data and map-matched trajectories, is implemented using a PostgreSQL/PostGIS database (one database instance per city).

As a vehicle traverses the road-network, it traverses roads of varying importance (cf. Table 9.3 for the distribution of the OSM road-networks per their respective category). However, the typical usage of each road is directly linked to its respective category. To reduce the bulk of data stored in our datastores, a separate process eliminates map-matched edges that belong to edges of less important roads, i.e., those that correspond to road-categories greater than 10 (OSM categories for unclassified, road, residential and living street - see Table 9.3). Depending on the time-period and the traffic patterns in each city, about $12-15\%$ of the map-matched records are eliminated using this "sanitation" process.

Since map-matched records are primarily used to offer real-time traffic information, older data is periodically removed from the respective PostgreSQL datastores (every 5 minutes) and archived into csv files for offline use. At the end of each day, a batch process compresses those csv files created during the day. A copy of this compressed file is then sent to a backup server for permanent archiving. Table 9.4 indicates the typical size of compressed archives produced per day and month for each city.

After a whole year of data collection (Oct. 2012 - Sept. 2013), several GBs of compressed historic map-matched trajectories are archived for each of the cities covered by our service. The real challenge is how to utilize this significant wealth of data to infer turning restrictions for the respective OSM road-networks.

9.2.3.2 Parsing map-matched trajectories and optimizations

The scope of our work is to identify specific turns (i.e., ordered pairs of edges connected via an intersection node) that, in an unusual way, are infrequently executed by vehicles.

This frequency will be determined by parsing the compressed archives of the historic map-matched trajectories produced for the three cities during the one-year operational period of our service. Since, the respective OSM road-networks comprise hundreds of thousands of nodes and edges (see Table 9.2), we need to somehow limit the number of turns that need to be examined.

The first optimization is to identify those pairs of consecutive edges that connect at intersection vertices. There is no need to study vertices of degree 2 (with just one incoming and one outgoing edge) or lower, since in those cases the driver has no choice but to travel in one direction (no actual intersection). In this way, we can effectively limit the number of candidate turns since the number of intersection vertices is much smaller (less than 60% or even less) than the number of total vertices (see Table 9.2).

The second optimization relates to the archived data. Since we only store map-matched trajectories that include major roads, i.e., OSM categories ≤ 10, we are also only interested in those intersection vertices connected to such roads. In making this assumption, we might miss some intersection vertices (strictly connected to unimportant roads). However, turn restrictions on minor roads have not only little to no impact on the overall traffic, but are also not always clear to express and enforce. On the other hand, intersections involving major roads are more likely to be used by vehicle drivers and, thus, have an overall dominant impact on traffic. The process of minimizing the number of examined turns is described in the pseudocode of Algorithm calculateExaminedTurns. Table 9.2 shows that major roads' intersection vertices are less than 25% of total vertices of all cities covered by our service.

calculateExaminedTurns($G(V, E, w)$)

```
1  examTurns = {}
2  totalTurns(V) = calculateAllTurnsOf(G)
3  for each turn(v) ∈ totalTurns(V)
4      if v.degree > 2  &  v.connectedToMajorRoad == TRUE
5          examTurns = examTurns ∪ turn(v)
6  return examTurns
```

These two optimizations considerably reduce the number of unique turns/pairs of consecutive edges we need to examine, which is a considerable improvement (see Table 9.5). Since the OSM road-networks of each city are stored in the respective PostgreSQL datastores, Algorithm calculateExaminedTurns for determining intersection vertices of interest and their corresponding turns may be easily implemented using plain SQL commands.

Table 9.5: *Total counted instances for all examined turns between Oct 2012 and September 2013*

city	# intersection vertices for roads ≤ 10	#examined turns	# total instances	# instances per inters. vertex for roads ≤ 10
Athens	34,921	75,552	144,451,729	4,137
Berlin	22,119	44,636	2,054,969,090	97,304
Vienna	16,104	36,484	610,902,632	37,935

calculateExaminedTurnsUsage(*examTurns*, *MmResults*, *examTimePeriod*)

```
1  for each examTurn ∈ examTurns
2      freqCounter[examTurn] = 0
3  for each day ∈ examTimePeriod
4      for each vehicleId ∈ MMResults(day)
5          for each turn ∈ MMResults(day, vehicleId)
6              if turn ∈ examTurns
7                  freqCounter[turn] = freqCounter[turn] + 1
8  return freqCounter[examTurns]
```

As a companion to the examined-turns detection algorithm, we implemented a custom Java application that (i) parses the compressed archives of historic map-matched trajectories (see Section 9.2.3.1), (ii) counts the instances encountered for each examined turn and (iii) stores the results in the respective PostgreSQL datastores. Algorithm calculateExaminedTurnsUsage describes this process and the results, i.e., the total counted instances for all examined turns during our one-year testing period, are given in Table 9.5. It is shown that on average for every intersection vertex connected to major roads (i.e., their respective road category ≤ 10), we have counted turn instances, ranging from 4,137 (Athens) up to 97,304 (Berlin). These results represent a sufficiently large number of measurements per intersection vertex.

9.2.3.3 Identifying candidate turning restrictions

At this point in our approach, we have identified the examined turns and counted the number of times each examined turn has been traversed by a vehicle. We now need to examine, which of those turns are *rarely used*. Since, both, turns and results of the enumeration process are stored in the respective datastores, it is easy to group results/turns by *entrance* edge and direction (for bidirectional edges). Each such group contains all possible turns a vehicle may execute after following a specific entrance edge (and direction). Likewise, each turn belongs to a single, specific group of turns. Since we know the number of instances encountered for each one of the turns belonging to the same group, it is easy to calculate the usage percentage or each one (cf. Algorithm calculateTurnPercentPerGroup). An example group for a specific entrance edge is shown in Figure 9.3.

calculateTurnPercentPerGroup(*examTurns*, *freqCounter*(*examTurns*))

```
1  groupsOfTurns = group(examTurns, entranceEdge, direction)
2  for each group ∈ groupsOfTurns
3      freqCounter[group] = 0
4      for each examTurn ∈ group
5          freqCounter[group] = freqCounter[group] + freqCounter[examTurn]
6  for each group ∈ groupsOfTurns
7      for each examTurn ∈ group
8          percent[examTurn] = freqCounter[examTurn] / freqCounter[group]
9  return percent[examTurns]
```

As we notice in the example group of Figure 9.3, most drivers (66%) continue straight when they traverse the entrance edge (A, B) leading to the intersection vertex B. Some others (32%) prefer to turn right. But a very small percentage of them (2%) turn left. This

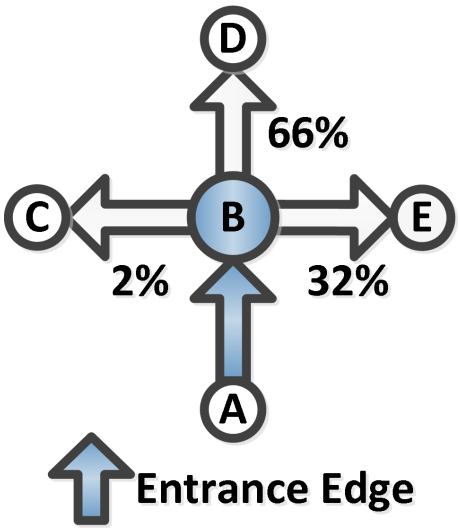


Figure 9.3: *A simple example of grouping turns per entrance edge (A, B) at an intersection vertex (B) for calculating usage percentage per turn*

is a very strong indication that this low frequency-usage of the left-turn actually represents erroneous map-matched trajectories (even the most efficient MM algorithms have a small error rate). Next, we made the assumption that turns with frequency usage percentage lower than a 5% threshold are most likely prohibited. The choice of this threshold was established after an initial set of experiments resulting in encouraging turning restriction results. Table 9.6 shows the respective number of the candidate turning restrictions discovered for each city for, both, 5% and 2.5% thresholds.

Table 9.6: *Number of candidate turning restrictions discovered for 5% and 2.5% thresholds*

		# turning restrictions		**turning restrictions (%)**	
city	**# turns**	**5%**	**2.5%**	**5%**	**2.5%**
Athens	75,552	5,287	3,596	7.00%	4.76%
Berlin	44,636	2,653	1,582	5.94%	3.54%
Vienna	36,484	1,739	1,261	4.77%	3.46%

However, estimating candidate turning restrictions is not enough. For each such turn, we need to additionally calculate its direction, to conclude if it is a straight, right, left, or U-turn. The direction calculation is comparatively easy, since we have already stored the angular direction of each edge in the respective datastore as needed for the isochrone functionality of our service [38]. Table 9.7 shows the categorization of the discovered candidate turning restrictions with respect to their direction. As expected, most of them (particularly in Berlin and Vienna) represent left-turns. As our verification results will show, the direction of the candidate turning restrictions has a very strong impact on the turning restrictions' validity.

To additionally improve the validity assessment of identified turning restrictions, we devised the methods described in the following section.

Table 9.7: *Categorization of candidate turning restrictions per direction for 5% threshold*

city	# turning restrictions	straight	left	right	U-turn
Athens	5,287	6.5%	45.4%	41.6%	6.5%
Berlin	2,653	1.6%	64.6%	18.6%	15.2%
Vienna	1,739	10.5%	44.8%	30.0%	14.7%

9.2.4 Verification process

Although identifying candidate turning restriction based on the frequency usage of their respective turns is an important criterion, we still need to verify our results in comparison to the actual road network conditions. To this effect, we will use (i) a mapping application for visualizing turning restrictions in comparison to a web mapping service, (ii) publicly available satellite imagery and Google Street View (wherever available) for the respective intersections, and (iii) propose an efficient method to cross-check our results with two separate map vendors' public APIs. This verification process offers multiple advantages over data-mining techniques, since *we are able to cross-reference our results in comparison to the actual road-network conditions*. Luckily, the new verification process proposed in this work (i.e., satellite imagery and Google Street View), an extension of the original work in [39], turned out to be the best method for providing the most realistic feedback for the performance of our approach and, in addition, how to calibrate its accuracy.

9.2.4.1 Visualizing results with a mapping application

Our first verification method entails the visualization of the candidate restricted turns. An intuitive way to do this is to (i) visualize each such turn with the appropriate traffic sign (depending on the direction of the turn according to Table 9.7) located at the corresponding intersection vertex coordinates and (ii) rotate each such traffic sign according to the entrance edge direction and to "simulate" what the driver witnesses before entering the corresponding intersection vertex. Since the second of these criteria cannot be achieved through either Google Maps or Google Earth [52], we used QGIS [99], a popular, free and open source GIS application that runs in all major operation systems, instead. In addition, QGIS may directly visualize geometry features directly retrieved from PostGIS enabled databases (such as our datastores) and, thus, we can avoid an unnecessary export process of our data. Moreover, we used a Google Maps Layer in QGIS as the background map layer to compare results with the data retrieved from an external mapping service. Figure 9.4 shows some typical examples of the results of this visualization process for some of the candidate turning restrictions.

- Figure 9.4(a) depicts an intersection familiar to most local drivers in the center of Athens. This type of restrictions were easily verified by our personal experience and they effectively demonstrate how easily, critical turning restrictions are discovered through our method.

- Figure 9.4(b) shows a case of a prohibited U-turn in the Berlin area. There, many disallowed U-turns are missing from the OpenStreetMap dataset.

- Figure 9.4(c) shows that the Google Maps layer visually confirms the discovered turning restriction.

(a) Turning restrictions visualization in Athens

(b) Turning restrictions visualization in Berlin

(c) Turning restrictions visualization in Vienna

Figure 9.4: *Visualizing turning restriction with QGIS*

Although our first verification method is an elegant, straightforward, and efficient way for visualizing and cross-referencing our results, it has one major drawback. It still requires local knowledge of the examined area to confirm the turning restrictions. As such, it may be very useful for experienced end-users with extensive local knowledge, such as fleet managers or taxi drivers, but it does not benefit the typical (Web) user dealing with areas unknown to him. To this effect, we devised the following methods.

9.2.4.2 Using satellite imagery and Street View

Our second verification method is to export the candidate turning restrictions in a format that may be used by Google Earth to cross-reference results with data collected by Google, such as the actual satellite imagery and Street View. We use the export functionality of PostGIS to create KML [53] representations, the data format used in Google Earth, of the line geometries of each candidate turning restriction and create one large KML file per city. In addition, during our KML export, we add the name of the edges/roads participating in each restriction in alphabetical order for easy lookup, as shown in Figure 9.5. We can get an even clearer impression of the respective intersections should the selected areas be available in Google Street View [54]. Some of the results of this visualization process are shown in Figure 9.6.

Cross-referencing results with satellite imagery has revealed several interesting details about our method. First of all, it showed that most U-turn restrictions may be easily verified (see Figures 9.6(c), 9.6(d)). However, the most important aspect is that those verified U-turn restrictions do not even have a traffic-sign assigned in the actual road network, since *no actual driver would effectively use them*, i.e., they are too dangerous. Nevertheless, these turning restrictions should still be added on the digital representation of the road-network graph for accurate shortest-path computation by routing engines. Therefore, our method not only discovers true turning restrictions (as represented by traffic-signs), but also *discovers those restrictions not explicitly prohibited by a traffic-sign due to their extremely low probability of actually being used*. This is an interesting discovery, since even if we were given full access to a worldwide traffic-sign database (if such a database existed), our method could still pinpoint additional existing turning restrictions.

Our method also produces some false-positives as revealed by satellite imagery. Examples are shown in Figure 9.7. Sometimes, even satellite imagery is not enough for revealing useful details and we need to resort to Google Street View for a more accurate

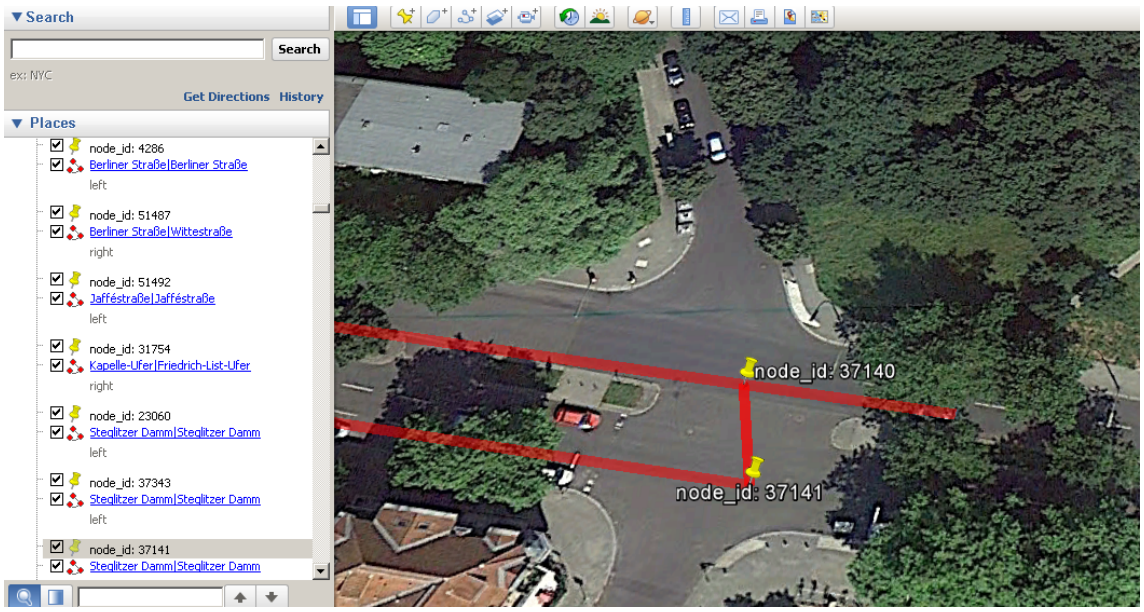


Figure 9.5: *The KML file created for Berlin*

depiction of the local intersection. Using Street View revealed several details for the examined intersections ranging from existing traffic-signs (as shown in Figure 9.8) to errors on the OSM mapping process (as shown in Figure 9.9)

When we closely inspected false-positives to identify a common pattern, we realized that many of them represent "straight" turns, i.e., turns with small angles between entrance and exit edge (see Figure 9.7). Such turns are mostly encountered on highways for exiting the main road or performing U-turns. The fact that the vehicles we monitored (during the one-year period) rarely used such turns is mainly attributed to the fact that we dealt with fleets using professional drivers (trucks in Athens, taxis in Vienna and Berlin), who (i) know precisely how to reach their destination and never have to backtrack when on a highway and (ii) usually use highways only for distant trips and not nearby destinations. In this sense, we should probably treat these "straight" turns as less promising to represent actual turning restrictions, a fact that will also be verified during the final phase of our verification process.

9.2.4.3 Sourcing external mapping services

Although visual inspection (by satellite imagery and Street View) is a convincing, qualitative way of validating results and provided significant insight into potential drawbacks of our method, it would be best if we could further verify and quantify our findings through an *automatic process*. We propose a method that compares our results with those provided by external Web-based routing APIs, the Google Directions API [55] and the Bing Maps Routes API [81]. Using two in instead of just one, not only provides a more credible verification of our results, but also allows us to assess the difference of the results provided by separate map vendors.

The Google Directions API and the Bing Maps Routes API are two public REST APIs that allow the calculation of directions between locations using HTTP requests. For both services, users may search for directions using different transportation modes, include driving, transit, walking, and cycling. Directions may specify origins, destinations, and via-waypoints, either as text strings or as latitude/longitude coordinates.

(a) A turning restriction in Athens verified by satellite imagery

(b) A turning restriction in Berlin verified by satellite imagery

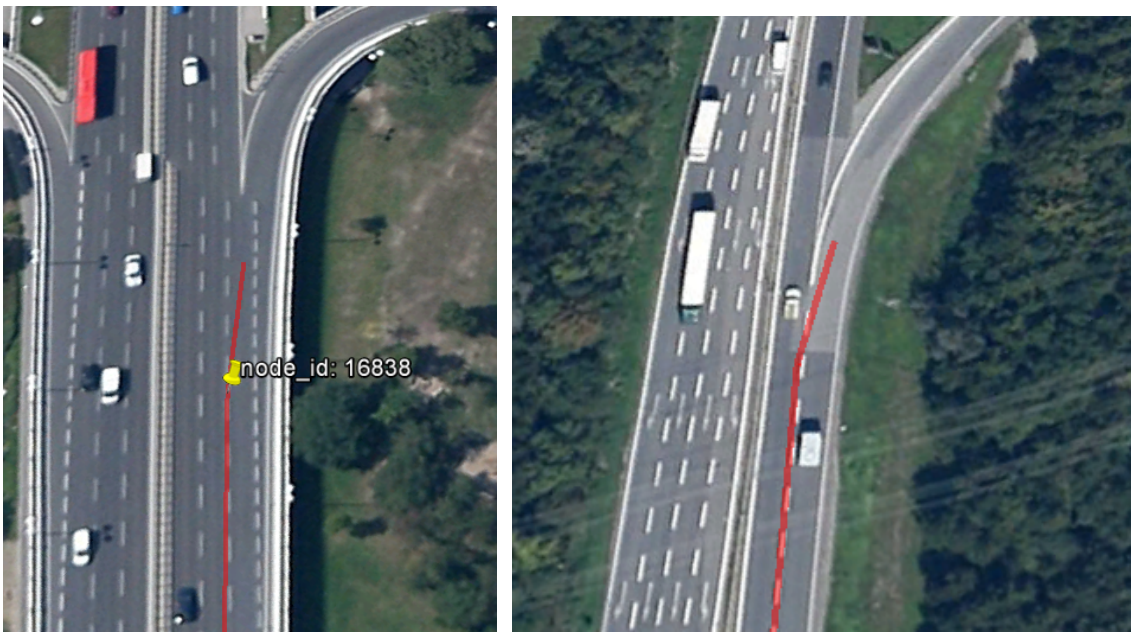
(c) Another turning restriction in Berlin verified by satellite imagery

(d) A turning restriction in Wienna verified by satellite imagery

Figure 9.6: *Verifying turning restrictions on Google Earth*

The Google Directions API allows only 2,500 directions requests per 24h period from a single IP address (free service). Contrarily, the Bing Maps Routes API offers a free 90-day Trial Key that allows one to evaluate Bing Maps for development and may be used for up to 10,000 transactions/routes calculations within a 30-day span during the evaluation period. In both cases, due to the aforementioned limits, by first identifying (a rather limited number) of candidate turning restrictions as our method does, we can verify our results within the limits allowed for free users (Bing Maps) and within a few days (Google Directions). In any such HTTP request to the APIs, certain parameters are required, while others are optional. The most important required parameters (relative to our problem) are:

- Origin (Google Directions) or waypoint.n (Bing Maps) - The origin location FROM which we want to calculate directions.
- Destination (Google Directions) or waypoint.n (Bing Maps)- The destination loca-



(a) A false-positive turning restriction in Vienna as revealed by satellite imagery

(b) Another false-positive turning restriction in Vienna as revealed by satellite imagery

Figure 9.7: *False positives produced by our method for "straight" turns*

Figure 9.8: *A false-positive turning restriction in Athens, as revealed by Google Street View*

tion TO which we want to calculate directions.

Two additional, optional parameters useful to our purpose are:

- Mode (Google Directions) or travelMode (Bing Maps). Both use driving as the default value - Specifies the mode of transport to use when calculating directions.
- Waypoints (Google Directions) or viaWaypoint.n (Bing Maps) - For defining additional intermediate locations that the route must travel through.

Given the above and with reference to Figure 9.3 in which the Turn($A \rightarrow C$ *via* B) has a low frequency usage, an HTTP request to verify this candidate turning restriction

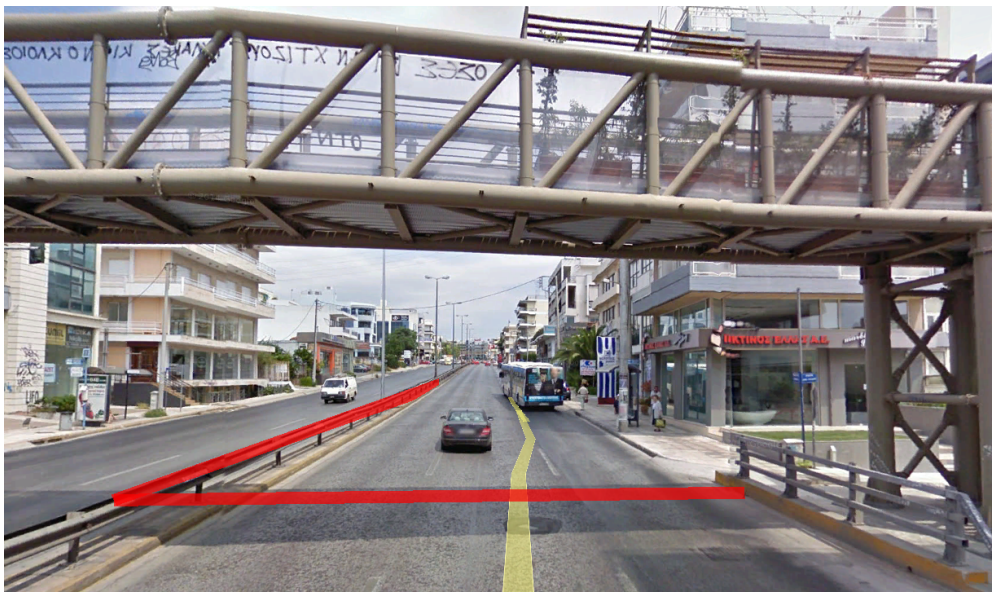


Figure 9.9: *An error in OSM maps that confirms the existence of the discovered turning restriction, as revealed by Google Street View*

```
http://maps.googleapis.com/maps/api/directions/json?
origin={A_coordinates}&destination={C_coordinates}&
waypoints=via:{B_coordinates}&sensor=false
```

Figure 9.10: *A sample Google Directions API request*

```
http://dev.virtualearth.net/REST/v1/Routes?wayPoint.0={A_
coordinates}&viaWaypoint.1={B_coordinates}&waypoint.2={C_
coordinates}&key=BingMapsKey
```

Figure 9.11: *A sample Bing Maps Routes API request*

would be similar to Figure 9.10 for Google Directions API and Figure 9.11 for Bing Maps Routes API. Both requests return a JSON object with the proposed route by the respective APIs. The process for verifying the turning restriction is described in Algorithm VERIFYRESTRICTION, which compares the distance / length (in meters) calculated by the APIs with the sum of lengths of edges (A, B) and (B, C). If the API provided distance is significantly greater than the sum of lengths of edges (A, B) and (B, C), then we may safely assume that indeed there is a turning restriction and the respective API has to follow a much longer route than simply $(A, B) \rightarrow (B, C)$.

VERIFYRESTRICTION($Turn(A \rightarrow C \text{ via } B)$)

1 $apiPath = DirectionsAPICall(A \rightarrow C \text{ via } B)$
2 **if** $length(apiPath) >> length(A \rightarrow B) + length(B \rightarrow C)$
3 $\quad TurningRestriction(A \rightarrow C \text{ via } B).verified = \text{TRUE}$

In order to access both APIs, we implemented a Java command-line application that retrieves turns below a threshold frequency usage (5% in our case) from the datastores, constructs an appropriate request string similar to Figure 9.10 and 9.11 for each turn, and retrieves the distance of the route returned by each API. To avoid overloading the API servers and rejected requests, we enforced a 500ms gap between requests. The distance

Table 9.8: *Differences on the results between Google Directions and Bing Maps Routes API*

city	Total	Google < Bing	Google > Bing	Google = Bing
Athens	5,287	1,457 (27.6%)	3,705 (70.1%)	125 (2.4%)
Berlin	2,653	762 (28.7%)	1,794 (67.6%)	97 (3.7%)
Vienna	1,739	526 (30.2%)	1,186 (68.2%)	27 (1.6%)

results returned from both APIs are also stored in the respective PostgreSQL datastore for easy access and querying.

An obvious problem to this approach for verifying results, is the usage limits of both APIs (especially the rather limited number of requests allowed by the Bing Maps Routes API for free users). Although we are dealing with road networks with hundreds of thousands of nodes, edges and possible turns, through our optimizations (see Section 9.2.3.2) and by restricting the usage of the APIs to strictly confirm the candidate prohibited turns found by our proposed method, we only need to evaluate a few thousands turns. The obtained results are presented in the following section.

9.3 Evaluation Results

Having developed automatic verification methods for our crowdsourced turning restrictions, the following section summarizes the assessment results produced by the respective methods.

9.3.1 Verified turning restrictions

Our verification methods, based on using the Google Directions API and Bing Maps Routes APIs, were described in Section 9.2.4.3. Before presenting the actual assessment results, we want to highlight the differences with respect to the data returned by the two APIs, shown in Table 9.8. Several interesting results emerge. Surprisingly, on only very rare occasions (less than 4%) the two APIs return the same route and route length. Besides algorithmic differences, this also shows the considerable differences between the map datasets used in each case. For most instances (> 67%), the Google Directions API returns a longer path than Bing Maps. This needs to be taken into account when trying to verify our candidate turning restrictions against the two APIs.

Tables 9.9 and 9.10 show the number of restrictions verified for both 5% and 2.5% implicit usage thresholds for each API case, for both APIs and for either of the two APIs, as well as their respective percentages in comparison to the total candidate restrictions. Note that usually the paths returned by both APIs are significantly larger (85-90% of the verified restrictions give at least two-times larger paths) than the sum of lengths (A, B) and (B, C), which means that there is an actual turn restriction in place that has not been considered in the OSM dataset. This is an indication to the validity of our verification method.

We notice that the majority of the candidate restrictions are successfully verified by one of the mapping services' APIs. In fact, for the case of Athens and Vienna, more than 74% of the extracted turning restrictions are verified by at least one of the APIs. For Berlin, the verified restrictions are 66% using the 5% threshold and 72% in case of the 2.5% threshold. Another observation is that by moving from the 5% to the 2.5% threshold, the verified restrictions' percentage increases slightly, but, we are also missing

Table 9.9: *Number of verified restrictions for 5% implicit threshold*

	candidate turning restrictions	# verified			
city	**Total**	**Google**	**Bing**	**Both**	**Either**
Athens	5,287	3,521 (67%)	2,624 (50%)	2,222 (42%)	3,923 (74%)
Berlin	2,653	1,546 (58%)	1,081 (41%)	886 (33%)	1,741 (66%)
Vienna	1,739	1,167 (67%)	683 (39%)	811 (47%)	1,295 (74%)

Table 9.10: *Number of verified restrictions for 2.5% implicit threshold*

	candidate turning restrictions	# verified			
city	**Total**	**Google**	**Bing**	**Both**	**Either**
Athens	3,596	2,470 (69%)	1,921 (53%)	1,643 (46%)	2,748 (76%)
Berlin	1,582	1,040 (66%)	736 (47%)	632 (40%)	1,144 (72%)
Vienna	1,261	879 (70%)	629 (50%)	550 (44%)	958 (76%)

Table 9.11: *Number of verified restrictions per angle for the 5% threshold*

	Left		Right		Straight		U-turns	
City	**Total**	**Verified**	**Total**	**Verified**	**Total**	**Verified**	**Total**	**Verified**
Athens	2,402	1,829 (76%)	2,199	1,589 (72%)	342	163 (48%)	344	342 (99%)
Berlin	1,714	1,058 (62%)	494	264 (53%)	43	20 (47%)	402	399 (99%)
Vienna	779	584 (75%)	521	348 (67%)	183	108 (59%)	256	255 (100%)

a significant number of restrictions (compare columns "Either" for 5% and 2.5%). This means, that there is a sizable number of existing (and verified) restrictions "contained" in the turn usage interval between 2.5% and 5%, which testifies to our choice of the threshold value of 5%. Hence, the following results are all based on a 5% threshold.

In Section 9.2.4.2 we observed a correlation between the entrance - exit edge direction angle of each candidate turning restriction with the validity of the restriction. Table 9.11 shows now the results of an explicit comparison of the number of verified restrictions (either API) using 5% threshold to this direction difference.

The results clearly confirm our empirical observations of Section 9.2.4.2. *Almost all (99%) of U-turn turning restrictions discovered by our method are verified by the APIs*. In contrast, "straight" turning restrictions (with small direction changes between entrance and exit edges) show a rather small verification rate of only about 50%. This is a strong indication that this particular type of turning restriction is more susceptible to errors and therefore additional means of verification are required. Moreover, the fact that many of these restrictions are highway exits is clearly confirmed by the rather limited number of those restrictions encountered in Berlin. In this case, the road network considered did not include the inner-city highways, which we did consider for Athens and Vienna. On the other hand, left turns show slightly better results than right turns. However both cases exhibit a similar mean verification percentage as shown in Table 9.9.

Finally, Table 9.12 compares the total turns, examined turns, candidate and verified turning restrictions to the turning restrictions of the OSM datasets for the three respective cities. The results are quite encouraging. Instead of examining hundreds of thousands of turns and focusing only on intersection nodes connecting major roads and utilizing historic map-matched trajectories, we discovered only a few thousand candidate turning restrictions that needed verification. By using the Google Directions and the Bing Maps Routes API, we could verify most of the identified candidate turning restrictions. Also, the

Table 9.12: *Total turning restrictions results for 5% implicit threshold in comparison to existing OSM's restrictions*

city	total turns	examined turns	candidate turning restrictions	verified turning restrictions	OSM turning restrictions
Athens	900,397	75,552	5,287	**3,923**	214
Berlin	252,271	44,636	2,653	**1,741**	668
Vienna	256,185	36,484	1,739	**1,295**	799

number of verified turning restrictions is significantly larger than the restrictions existing in the original datasets. Especially for Athens, the number of verified turning restrictions is 18× larger than those existing in the current OSM dataset. Even for Vienna and Berlin the number of the verified prohibited turns is still $1.5-2.6\times$ larger than those in OSM. Our results lead us to assumption that for countries and respective cities with less good map coverage, e.g., Albania, Montenegro, the proposed approach could significantly improve map datasets.

9.3.2 False positives?

Another important question is what we really can infer for those turning restrictions that were not verified by either API. Here we refer to the unverified left and right turns for which the distances returned by the Google Directions and Bing Maps Routes API are quite similar to the sum of lengths of their constituent edges (A, B) and (B, C). We examined several of these cases and found that for a small number of routes (almost 1% for all three cities of those unverified left and right restrictions) that the distances returned by the APIs is *less* than 90% of the sum of lengths of the constituent edges (A, B) and (B, C), i.e., there is a shorter route than making a simple turn. When examining these anomalies, we found that most of the times there was an inconsistency between OSM and the commercial map. For these cases, as to whether the turn is actually allowed or not is very debatable.

Still, even if we assume that all unverified left and right turning restrictions are indeed permitted, i.e., our method produces false-positives, we cannot ignore the fact that only *a very small percentage of the professional drivers we monitored actually use them*. Hence, a good-quality shortest-path solution *would still have to penalize (by increasing the respective turn cost) such "unappealing" turns*. As such, even unverified turning restrictions are still useful in revealing typical drivers' patterns and behaviors.

9.4 Conclusion and Future Work

This chapter proposed a new and efficient, semi-automatic way to infer and identify turning restrictions for OpenStreetMap data by utilizing historic map-matched trajectories from an existing fleet management service. Our experimentation covered three major European cities and a period of twelve months. Overall, $66-74\%$ of the turning restrictions we identified were successfully verified through a rigorous verification method, including visual inspection through a mapping application, satellite imagery and the use of public mapping APIs. However, the most important outcome is that we have identified and verified $2-18\times$ more turning restrictions than those existing in the current OSM dataset. This impressive feat testifies to the credibility of our method.

To the best of our knowledge, this is the first work to utilize historic map-matched trajectories for such a task. This is after all, the main contribution of this chapter, since the few existing works addressing the related subject of intersection delays base their research on raw GPS trajectories. In addition, most previous works use either simulated data or data covering smaller time periods (up to a month) and were focused on a particular geographic area. Our results are based on three European cities, originate from three medium to large vehicle fleets of 2,000-5,000 vehicles each, and cover an entire year of operation. The results in terms of discovered data for the three areas were almost identical, which further testifies to the robustness and validity of our method. In addition, by comparing our results with two external mapping APIs (the Google Directions API and the Bing Maps Routes API) we show the validity of our approach.

We can propose several interesting directions for future work. Since the proposed method is able to identify and confirm turning restrictions in the OSM data, we can expand it to automatically contribute those verified restrictions back to the OSM project. In this way, the outcome of our work could be shared by the mapping community and, thus, increase its impact. Our results could also further improve the quality of existing map-matching algorithms. Many of these algorithms use partial shortest-path calculations to align the raw GPS traces to the road network graph. Up until now, those SP computations do not take turning restrictions into account. Since our approach identifies such restrictions, those newly found constraints could be integrated back into the map-matching algorithms to further improve their results. In this way, for the first time, a self-improving, evolutionary map-matching algorithm might become a reality.

Chapter 10

Sentiment mapping travelblogs

One common theme in multiple aspects of our work is crowdsourcing, i.e., exploiting the results, processing power and data of individual users and online communities to further extract additional meaningful information, that in our case has also a prominent geographic footprint. In [47], we showcased a system, which efficiently distributes graph-separator shortest-path preprocessing to multiple web-clients and took advantage of the processing power of potential users using the aforementioned infrastructure. In [38] we used the GPS-traces created by vehicle fleets to extract information about live-traffic and build historic speed-profiles for three European cities. In [39], we used the map-matched trajectories created by the service of [38] to infer information about existing turning-restrictions in the road network dataset. In this chapter, we focus on using travelblog entries to extract, aggregate and visualize the user sentiment in relation to their geographic location. By crawling various travel related web sites (e.g. travelpod.com, travelblog.org), we created a corpus of 150,000 texts. Analyzing this text at the paragraph level for sentiment information, we essentially created a geospatial sentiment-map based on the user-contributed information contained in the articles of the respective travelblogs. This chapter's content is included on our [36] publication presented in Terra Cognita 2011 Workshop, held in Conjunction with the 10th International Semantic Web Conference and our [21] work to appear in the International Journal of Geographical Information Science.

10.1 Introduction

Crowdsourcing moods and opinions, and in our case sentiments from user contributed data, has recently become an interesting field after micro-blogging services (e.g., Twitter) have risen to prominence. There, blog entries reflect a myriad of different user sentiments that when integrated can give us valuable information about, e.g., the stock market [16]. In this chapter, our main focus is on (i) extracting the user sentiment about geographic location using travelblog entries as out input data, (ii) aggregating such sentiment data and, finally, (iii) finding an intuitive way to visualize / map the aggregated user sentiments.

In terms of individual contributions close to our specific focus and the concept of information visualization using maps, we can cite the following related work [112, 3, 101, 32, 123] and [64]. For the purpose of recognizing toponyms, existing approaches share basic concepts from the field of Natural Language Processing (NLP), Part-Of-Speech (POS) tagging and a part of *Information Extraction* (IE) related tasks, particularly Named Entity Recognition (NER) [66]. NER approaches can be roughly classified as, either machine learning - statistical [75, 77, 112, 100] or rule-based [18, 20, 35, 98, 127]. Machine

learning / statistical approaches view Named Entity Recognition as a classification problem. Those methods have recently gained increased popularity due to their relatively rapid development / domain customization and the reduced amount of human effort required. On the other hand, for *Rule-based approaches*, the Named Entity Recognition is based on linguistic/semantic rules defining the possible linguistic patterns denoting Named Entity concepts, such as the approaches adopted by ANNIE[1], and CAFETIERE [15]. These approaches can achieve better results than most statistical or machine learning methods but require extensive human effort for the development of the necessary knowledge resources (rules and lexico-semantic resources). For this reason, the adaptation of rule-based systems to new domains is a slow and laborious process.

With respect to geocoding, we can exemplary mention [84], that describes a navigational tool for browsing web resources by geographic proximity as an alternative means for Web navigation. Web-a-Where [5] is another system for geocoding Web pages that assigns a geographic focus to each page and a locality that the page discusses as a whole. Moreover, the tagging process targets large collections of Web pages to facilitate a variety of location-based applications and data analysis. The work presented in [76] also identifies and disambiguates references to geographic locations. Another method that uses information extraction techniques to geocode news, is described in [112]. Other toponym resolution strategies involve the use of geospatial measures such as minimizing total geographic coverage [75], or minimizing pairwise toponym distance [77]. While statistical NER methods can be useful for analysis of static corpora, they are not well-suited for case of continuously user contributed travel narratives, due to their dynamic and ever-changing nature [110]. To this purpose, the work of [37] focusing on the extraction of routes from narratives, relies on a powerful IE rule-based solution based on a modular pipeline of distinct, independent and well-defined components based on NLP and IE methods. The proposed IE approach there has been further adapted and augmented to fit the requirements of our work here.

The specific contributions in this chapter are as follows. Initially, several travelblog Web-sites have been crawled to create a corpus of over 150k texts. The collected texts are then geoparsed and geocoded to link placename identifies (toponyms) to location information. With paragraphs as the finite granularity for sentiment information, texts are then assessed with a subjectivity / sentiments detector tool for assigning a sentiment score to each paragraph ranging from very negative to very positive. Scores are then linked to the bounding box of the paragraph and are aggregated using a global grid, i.e., the score of each specific paragraph is associated with all intersecting grid cells. Aggregation of sentiments is then performed simply by computing the average of all scores for each grid cell. Finally, the score can be visualized / mapped by assigning colors to each cell.

The remainder of this chapter is organized as follows. Section 10.2 describes the information extraction techniques employed in our approach dealing specifically with the aspects of geoparsing and geocoding travelblog entries. Section 10.3 outlines a method for computing sentiment biases from travelblog entries. Section 10.3 presents our method for mapping / visualizing geospatial sentiment data along with specific examples highlighting our results. Finally, Section 10.5 presents conclusions and directions for future work.

[1] http://gate.ac.uk/

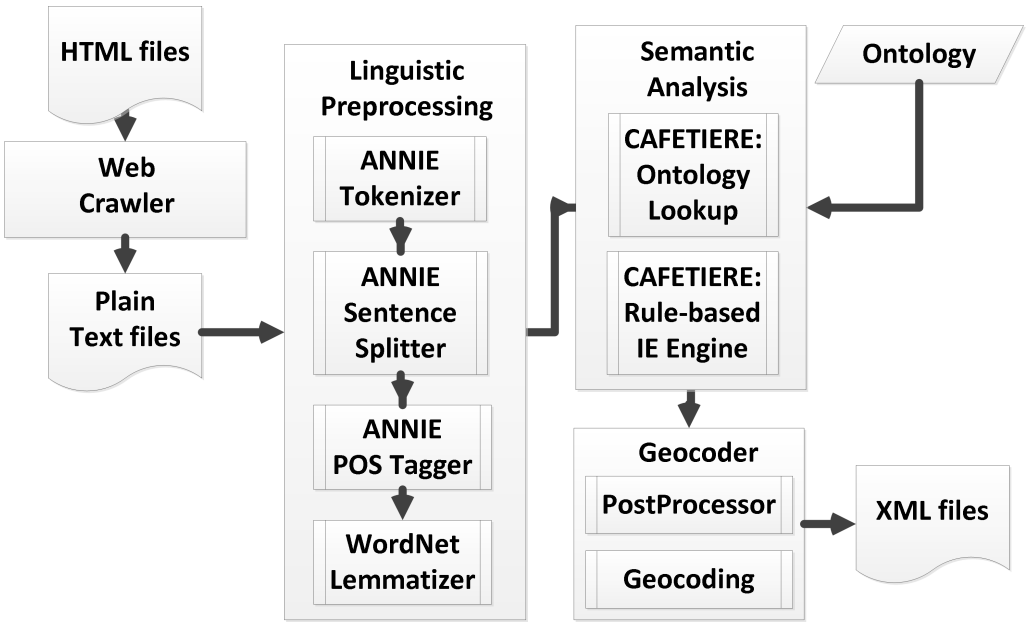


Figure 10.1: *IE architecture pipeline*

10.2 Information Extraction

Our first step towards visualizing / mapping the sentiments of the users of travelblogs in relation to their geographic location, is to identify toponyms within these travelblogs. The specific processing pipeline to achieve this, initially uses an HTML document (travelblog article) as input and produces a structured XML file containing the various geographic entities and their respective attributes (toponyms, coordinates and information for the enclosing paragraph / text document) (cf. Figure 10.1). The pipeline consists of four steps (i) the HTML parsing module, (ii) the linguistic preprocessing, (iii) the Information Extraction (IE) engine system (semantic analysis) and (iv) the geocoding-postprocessing part. Each of the steps will be described in detail, in the following sections.

10.2.1 Web Crawling

For collecting travelblog articles containing rich geospatial information, we crawled various Web sites that allow any user to create and share travel-related blogs and stories. To this purpose, we crawled travelpod.com, travelblog.org, traveljournals.net and worldhum.com, resulting in a total of almost 150,000 documents. For the task of crawling those Web sites, we used the Regain crawler[2], which creates a Lucene[3] index for indexing the documents' information, while for HTML parsing and the extraction of useful plain text narratives, we used the Jericho HTML parser[4].

10.2.2 Linguistic preprocessing

In order to prepare the input for the core IE engine for extracting points-of-interest, the created plain text documents must be prepared accordingly. Such preparation includes linguistic preprocessing tools that analyze natural language documents in terms of distinct

[2]http://regain.sourceforge.net/
[3]http://lucene.apache.org/
[4]http://jericho.htmlparser.net/

base units (i.e., words), sentences, part-of-speech and morphology. To this purpose, we are using the ANNIE IE tools, contained in the GATE language processing release[5]. The linguistic preprocessing module comprises of a set of four sub-modules: (i) the ANNIE tokenizer, (ii) the (ANNIE) Sentence Splitter, (iii) the ANNIE POS Tagger and (iv) the WordNet Lemmatiser. The intermediate processing results are passed on to each subsequent analysis tool as GATE document annotation objects. The individual steps of this process are:

- *Tokenization*, i.e., recognizing in the input text basic text units (tokens), such as words and punctuation and the association of orthographic features, such as capitalization, use of special characters and symbols to the recognized tokens. The tools used are ANNIE Tokenizer and Orthographic Analyzer.

- *Sentence splitting*, aiming at the identification of sentence boundaries in a text.

- *Part-of-speech* (POS) tagging is the process of assigning a part-of-speech class, such as Noun, Verb etc. to each word in the input text. The ANNIE POS Tagger implementation is a variant of Brill Transformation-based learning tagger, which applies a combination of lexicon information and transformation rules for the correct POS classification.

- *Lemmatisation* is used for text normalisation purposes. With this process we retrieve the token base form e.g., for words [travelling, traveler, travel] the corresponding lemma is: travel. For this purpose, we used the JWNL WordNet Java Library API[6] for accessing the WordNet relational dictionary. The output of this step is included it in GATE document annotation information. We will further exploit this information in the semantic rules section, presented later on.

The final output of the linguistic preprocessing module is the analyzed document, transformed in CAS/XML format, which will be passed to the subsequent semantic analysis component Cafetiere IE engine, as input. CAS (Common Annotation Scheme) is an XML schema, for defining a wide range of structural, lexical, semantic and conceptual annotations. Cafetiere IE engine will then combine this linguistic information with knowledge resources information (i.e., the lookup ontology) and the analysis rules to semantically analyze the documents and recognize spatial information, as we will describe in the following section.

10.2.3 Semantic Analysis

The Semantic analysis module relates the linguistic preprocessing results to ontology information and applies semantic analysis grammar rules, i.e., documents are analyzed semantically to discover spatial concepts and relations.

For this purpose, we used the Cafetiere IE engine [15] , whose objective is to compile a set of semantic analysis grammar rules in a cascade of finite state transducers, to recognize the concepts of interest inside the text. Cafetiere IE Engine combines all previously acquired linguistic and semantic information with contextual information. We modified Cafetiere and implemented it as a GATE pipeline module for the purpose of performing

[5]http://gate.ac.uk/

[6]http://sourceforge.net/projects/jwordnet/

ontology lookup and rule-based semantic analysis on information acquired from previous pipeline modules, in the form of GATE annotation sets. The input to this process are the GATE annotation objects resulted from the linguistic preprocessing stage, transformed in CAS/XML format (as needed by Cafetiere) for each individual document.

10.2.3.1 Cafetiere Ontology Lookup

The use of knowledge lexico-semantic resources assists in the identification of named entities. These semantic knowledge resources may be in the form of lists (gazetteers). Thus, the NE recognizer can use the provided gazetteer information for classifying a text string, as an entity belonging a particular class. In our case, the named entities we want to extract with IE methods are geographic locations. For example, a gazetteer for location designators might have entries such as "Sq.", "blvd.", "st." etc. that denote squares, boulevards and streets accordingly. Moreover, more sophisticated knowledge resources in the form of ontologies (providing mappings of text strings to semantic categories) and knowledge bases may also be used to provide this type of richer semantic information and allow for the specification and representation of additional information, if necessary, than plain identity and class inclusion.

Since CAFETIERE Ontology lookup module may access previously built ontologies to retrieve potential semantic class information for individual tokens or phrases and we are interested in identifying geographic locations, it made sense to manually create a custom-made ontology for geographic locations, by analyzing a large number of texts. We further, iteratively refined this specific ontology with words (e.g. verbs) that when matched to a text phrase are likely to indicate a spatial relationship between the corresponding referenced concepts. Summarizing, the lookup stage of analysis:

- Supplies semantic classes (concepts) corresponding to words and phrases.
- Supplies object identifiers for known instances, including where aliases and abbreviations name the same instance (For example "National Technical University of Athens", "NTUA").
- Supplies properties of known instances, for example the country of which a city is the capital.
- Uses verbs of interest to identify potential unknown instances of toponyms contained in the text.

10.2.3.2 Cafietiere IE engine

Since, Cafetiere is a rule-based system for IE, a set of linguistic patterns (i.e., extraction rules) was written taking into account the lookup ontology and all previously acquired information from linguistic preprocessing. The semantic analysis rules, were developed as a set of context-sensitive/context-free grammar (CSG/CFG) rules and were compiled in a cascade of finite state transducers to recognize the concepts of interest in plain texts.

10.2.4 Geocoder

In this module, all information regarding each toponym for each document in the collection is imprinted as a GATE annotation set object. For each document, we have collected

```
On Christmas Eve the Spanish eat a big dinner which usually
includes seafood. And "Papa Noel" is gaining popularity, but
it's more traditional to give gives on "El Día de los Reyes
Magos" (The Day of the Wise men), which falls on January 5th
this year. I have heard that I have a better chance of
seeing snow in Salamanca, so....
Plaza de Fonseca, this small plaza is located right behind
the Cathedral.
Plaza de Obradoiro
Very impressive.
Nativity
```

Figure 10.2: *Sample plain text*

```
- <Document filename="data/texts1/729.txt" placeReferred="Spain"
  meanCoords="42.018,-6.07" standardD="564.67" totalNumOfTokens="524"
  totalGeocodedSuccessfuly="11" totalExtracted="12">
    <poi name="Salamanca" startOffset="2509" endOffset="2518" sentenceID="30"
    paragraphID="15" coords="40.9642,-5.66385" accurCode="0"/>
    <poi name="Plaza de Fonseca" startOffset="2558" endOffset="2574" sentenceID="31"
    paragraphID="16" coords="40.579929,-6.584242" accurCode="0"/>
    <poi name="is located right" startOffset="2592" endOffset="2608" sentenceID="31"
    paragraphID="17" coords="ISRELATION"/>
    <poi name="Cathedral" startOffset="2620" endOffset="2629" sentenceID="31"
    paragraphID="17" coords="NULL" accurCode="NULL"/>
    <poi name="Plaza de Obradoiro" startOffset="2631" endOffset="2649" sentenceID="32"
    paragraphID="18" coords="39.256985,-5.806305" accurCode="0"/>
```

Figure 10.3: *Resulting XML file*

information about all extracted entities, along with their respective paragraph, sentence and character offset in this document. During the HTML parsing process we kept the geographic scope that each document is referred, in order to use this information during the geocoding process of each extracted entity. Regarding geocoding, we initially used YAHOO! Placemaker[7] in combination with Cafetiere's output. We observed that although PlaceMaker worked well for disambiguating some entities, it identified significant fewer geographic entities than our IE engine. Thus, in the remaining entities extracted by Cafetiere, we applied YAHOO! Placefinder[8] to geocode this place information passing the scope information threshold described below, for delivering more accurate results.

Finally, for each HTML travelblog entry (narrative), we create a collection of extracted geo-entities. For each of these entities there is specific information (acquired from each of the previous pipeline steps) about where they were encountered in the respective document, namely, sentence, paragraph and offset character. Additionally, for each document, we calculate the mean coords and standard distance from all extracted geocoded locations. This information, along with the local parsed text file path and the respective URL of the document, are stored into one XML file per each plain text document. Samples of input plain text narrative and the corresponding output structured XML file are shown in Figures 10.2 and 10.3 respectively.

10.3 Sentiment Assignment

During the IE preprocessing phase, we have successfully geocoded the geographic locations inside the travelblog entries and associated each such location with its enclosing text document / paragraph. In the following step, we want to assign sentiment ("mood") to

[7] http://developer.yahoo.com/geo/placemaker/
[8] http://developer.yahoo.com/geo/placefinder/

each paragraph of the text. To this end, we use OpinionFinder[9], a system that performs *subjectivity analysis*, automatically identifying when opinions, sentiments or speculations are present in text. It aims to identify subjective sentences and marking various aspects of subjectivity in these sentences, including the source (holder) of the subjectivity and words that are included in phrases expressing positive or negative sentiments [120].

OpinionFinder operates as one large pipeline. Conceptually, this pipeline can be divided into two parts. The first part performs mostly general purpose document processing (e.g., tokenization and part-of-speech tagging). The second part performs the subjectivity analysis. The results of the the subjectivity analysis are returned to the user in the form of SGML/XML markup of the original documents.

For the first part, OpinionFinder takes the incoming text source and removes HTML or XML meta info. Sentences are split and POS tagged using OpenNLP[10], the open source solution providing a variety of java-based NLP tools, using the OpenNLP Maxent machine learning package. Next, stemming is accomplished using Steven Abneys' SCOL v1K stemmer program[11]. SUNDANCE (Sentence UNDerstanding And ConceptExtraction) [120], is used to provide semantic class tags, identify extraction patterns needed by the sentence classifiers, identifying the source of subjective content and distinguishing author statements from related or quoted statements. A final batch mode parser establishes constituency parse trees, which are converted to dependency parse trees for Named Entity and subject detection.

At this point, for the second part, a Naive Bayes classifier identifies subjective sentences. The classifier is trained against subjective and objective sentences generated by two additional rule-based classifiers drawing from large corpora [119]. Next, a direct subjective expression and speech event classifier, tags the direct subjective expressions and speech events found within the document, using WordNet[12]. The final step applies actual sentiment analysis to sentences that have been identified as subjective. This is accomplished with two classifiers that were developed using the BoosTexter [107] machine learning program and trained on the MPQA Corpus[13].

OpinionFinder was applied to all texts of our collection. In the next section, we will describe how this information is used in the context of visualizing / mapping the sentiments of the travelblogs users in relation to geographic location.

10.4 Mapping Sentiment Scores

Since individual paragraphs are our chosen level of granularity, each paragraph must be assigned both a geographic and a sentiment dimension. As each paragraph contains zero, one or multiple geographic entities (that were suitably geocoded) the spatial extent of a paragraph is represented by the Minimum Bounding Rectangle (MBR) of the geographic entities included in this paragraph. We chose to spatially visualize only paragraphs including more than one geographic entities and the corresponding MBR of the contained toponyms does not exceed 0.5 degrees in either dimension (e.g., $Max(lat) - Min(lat) \leq 0.5$ AND $Max(lon) - Min(lon) \leq 0.5$). Consequently, only paragraphs of limited and focused

[9] http://mpqa.cs.pitt.edu/opinionfinder/
[10] http://opennlp.sourceforge.net/
[11] http://www.vinartus.net/spa/
[12] http://wordnet.princeton.edu/
[13] http://nrrc.mitre.org/NRRC/publications.htm.

Result (positive - negative)	Colour
≤ -3	Red
=-1 OR =-2	Orange
0	Yellow
=1 Or =2	Olive
≥ 3	Green

Table 10.1: *Sentiment mapping to color representation*

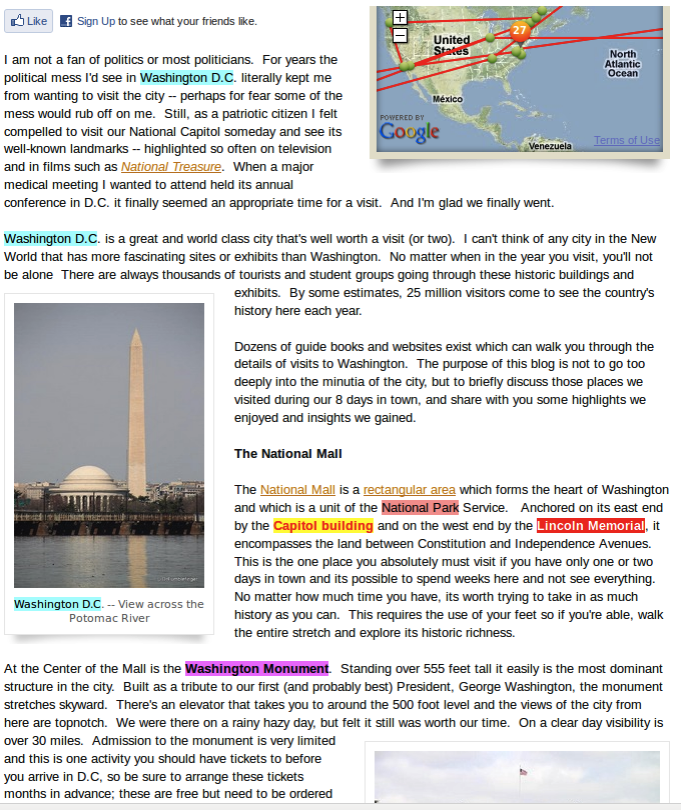


Figure 10.4: *Washington D.C. - sample document and toponyms*

spatial extent are visualized, thus preventing paragraphs that refer to larger geographic entities (e.g., Europe) to overshadow our results.

For each of those aforementioned paragraphs of limited and focused spatial extent, we kept the total referred positive and negative sentiment subjectivity expressions respectively, as captured by our OpinionFinder module. We used five different categories for visualizing sentiment scores for each paragraph. The categories used and the color assigned per category are displayed on Table 10.1.

The proposed approach is clarified by the following example. A sample document[14] (Figure 10.4) contains several paragraphs mentioning Washington D.C. and its monuments. Each of this document's paragraphs was assigned a MBR containing the geographic entities included per paragraph and a review category color according to Table 10.1. Therefore, this document may be easily visualized on a map as shown on Figure 10.5.

Although this approach is viable when there is a limited number of documents and paragraphs, multiple paragraphs from different documents and different reviews may par-

[14]http://www.travelpod.com/travel-blog-entries/drfumblefinger/1/1269627251/tpod.html

Figure 10.5: *Washington D.C. - geospatial opinion visualization*

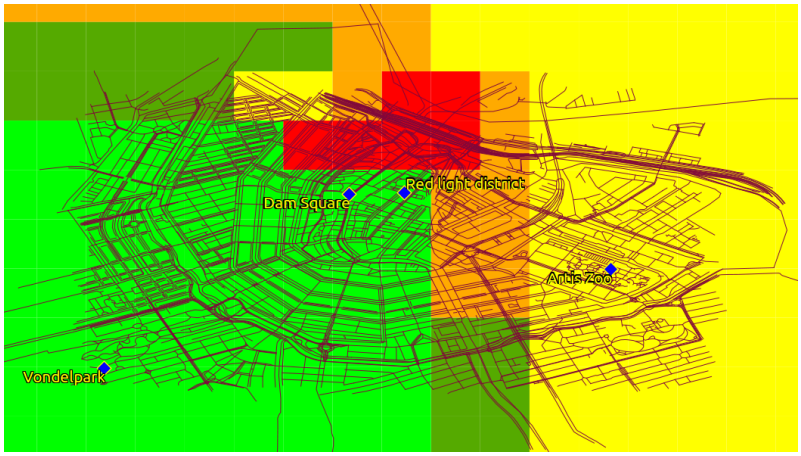


Figure 10.6: *Amsterdam, The Netherlands - geospatial opinion visualization*

tially target the same area, e.g., we need to visualize partially overlapping MBRs of different reviews (colors). To this end, we split each paragraph MBR into very small cells of a regular grid of 0.0045 degrees (corresponding to 500m) in each dimension. For each of those cells we sum up the sentiment score from all the containing paragraph MBRs. With this approach, instead of trying to visualize *overlapping* paragraph MBRs with different scores (colors), we visualize *distinct* small cells with each being assigned a unique score and color according to Table 10.1. Consequently, it is easy to visualize the overall sentiment scores independent of how many paragraphs or documents target the same area.

Further examples of our proposed method are displayed in Figures 10.6 and 10.7. Figure 10.6 shows the overall geospatial sentiment map of Amsterdam. Note, that while most of the city is shaded green (corresponding to positive sentiments), the area around the train station and the Red Light district are shown in red, i.e., the travelblog users were expressing rather negative opinions about these particular city areas.

Figure 10.7 gives a geospatial opinion map of Central Europe indicating the areas mentioned in the travelblogs. Note, that positive sentiments are associated with areas in Switzerland and also Italy, while urban areas such as Brussels receive overall more

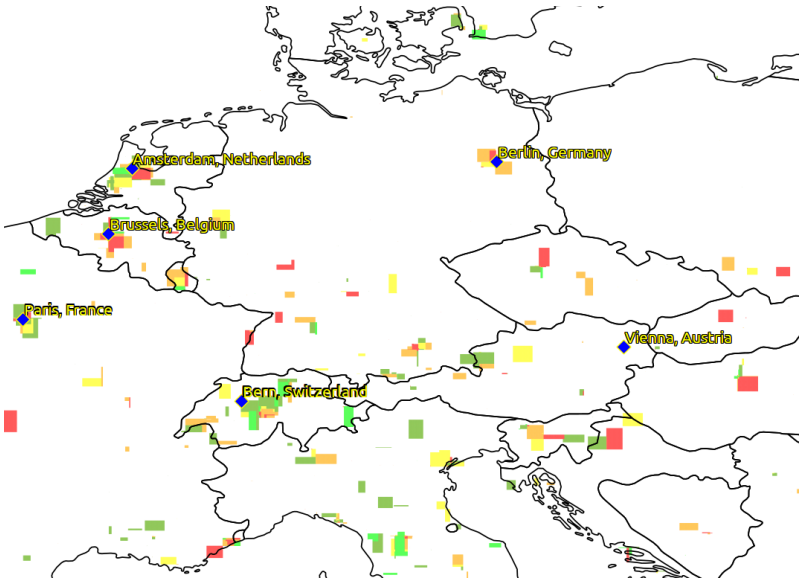


Figure 10.7: *Europe - geospatial opinion visualization*

negative sentiments from travelblogs users.

Our initial experiments with the creation of geospatial sentiment maps derived from subjective travelblog entries show that there is a clear bias for certain geographic areas shared by people. However, since in this work we only performed a simple aggregation of the scores generated by the OpinionFinder tool, it will require more in-depth analysis of the results to generate accurate statements and trends.

10.5 Conclusions

Aggregating sentiments is important for utilizing user-generated content. This chapter proposed a method for visualizing sentiments for specific geographic areas, as derived from travelblog entries. To demonstrate our approach, several travelblog sites were crawled and a total of more than 150K pages/articles were processed. Using (i) geoparsing and geocoding tools this content was geo-referenced and (ii) sentiment information was derived using the OpinionFinder tool. In the proposed approach, sentiment information from various articles relating to the same geographic area is aggregated and visualized accordingly by means of a customized, geospatial heat map.

Directions for future work are as follows. The current approach for aggregating user sentiments for geographic areas is rather simple and a more in-depth analysis of the results will be required to generate accurate statements and trends. An obvious improvement will also be to examine / include microblogging content streams. Here, sentiment information will be updated live and thus represent an accurate picture of the situation of a specific geographic area over time.

Chapter 11

Conclusions and future work

In this dissertation we have covered a lot of ground. We presented our work on improving existing algorithms for the single-pair shortest-path problem and we developed novel algorithms and frameworks for multiple variations of shortest-path queries on road networks. We significantly increased the scope of our results by creating real-world systems and services and we have taken advantage of those systems' results to extract additional meaningful information about the impact of traffic and existing turning restrictions on the road network datasets. Finally, we have showcased how to map the spatial distribution of users' sentiments about specific areas, by analyzing a huge corpus of travelblog entries.

In the remainder of this chapter, we will further briefly summarize our contributions and we will identify potential extensions of our results for future work.

11.1 Summary

In this section we will summarize the results produced by this dissertation. Our individual contributions may be classified in two major, distinct yet interconnected scientific areas: First, we focused on implementing efficient algorithms and frameworks for handling most variations of shortest-path related queries on road networks. Our second focus was on creating real-world systems and services and taking advantage of those systems' results in combination with crowdsourced information, to further extract and infer additional meaningful information missing from our input datasets. In more detail:

The first contribution of our work was to significantly improve the performance of the ALT single-pair shortest-path (SPSP) algorithm and making it suitable for a dynamic navigation scenario. Our specific contributions were to improve ALT's preprocessing time and query performance to significantly broaden its scope, thus making this specific algorithm suitable for handling dynamic road networks with frequent traffic updates.

Our second contribution in terms of SPSP queries was to showcase and implement a system, which efficiently distributes typical graph-separator shortest-path preprocessing to multiple Web-clients. To this purpose, instead of using a dedicated cluster of nodes connected to a network infrastructure, we used Web-browsers and Javascript as our computing platform. Our extensive experimentation with a continental road network have showed that this proposed approach was very fast, efficient and scalable.

Regarding single-source shortest-path (SSSP) queries, our contribution was to create novel graph separator methods to efficiently handle all variations of the single-source shortest-path problem. The three proposed algorithms, GRASP, isoGRASP and reGRASP are each tailored to a specific SSSP problem (one-to-all, range, and one-to-many) and

require minimal preprocessing time (their preprocessing time is 1-2 orders of magnitude faster than previous approaches) and, thus, are the only viable solutions for handling dynamic road networks, i.e., road networks with changing edge weights due to traffic updates. Moreover, they provide excellent parallel performance for both travel times and travel distances, scale better on multicore processors and offer better or comparable performance to state-of-the-art approaches. But most of all, they may efficiently solve range / isochrone queries, not addressed by previous solutions.

Building on our previous algorithmic contributions, this dissertation also proposed the unified SALT (graph Separators + ALT) framework that requires only seconds for preprocessing continental road networks and provides excellent query performance for a wide range of shortest-path and network search problems, such as (i) single-pair, (ii) one-to-all, (iii) one-to-many, (iv) range and (v) k-NN queries. We have showed that SALT is (i) $3-4\times$ faster for point-to-point queries, compared to existing methods of similar preprocessing times, (ii) it answers one-to-all, one-to-many and range queries with comparable performance to state-of-the-art approaches, and most importantly, (iii) it may also answer k-NN queries in $< 1ms$, for both, static or moving objects. As such, our SALT framework could be a serious contender for use in commercial applications.

Our aforementioned algorithmic contributions may also be used in a range of practical real-world applications. By combining state-of-the-art research about road networks, Floating Car Data, map-matching, historic speed profile computation, live-traffic assessment and time-dependent shortest-path computation we have implemented the fully functional SimpleFleet fleet-management service that now covers the urban regions of the European cities of Athens, Berlin and Vienna. To the best of our knowledge, this was the first work to combine the state-of-the-art isochrone computation of [82] with live-traffic data, in addition to providing this real-time system showcasing our results.

We have also expanded the results produced by the aforementioned SimpleFleet service to two different and interesting directions: (i) We have developed an additional geomarketing application that combines live-traffic, isochrones and demographic / business data for Berlin and Vienna to demonstrate the actual impact of traffic fluctuations to business intelligence decisions and (ii) we used the map-matched trajectories created by the service to infer information about existing turning-restrictions in the road network dataset. Both approaches have provided solid results: (i) The impact of traffic is more than 20% for normal conditions and exceeds 60% in case of traffic congestion and (ii) our method for discovering turning restriction from map-matching results has identified and verified 2-18 times more turning restrictions than those existing in the original datasets.

Finally, since multiple aspects of our work revolve around the concept of crowdsourcing, this dissertation also focused on using travelblog entries to extract, aggregate and visualize user-opinions in relation to geographic location. By crawling various travel-related Web-sites and analyzing this created corpus of text at the paragraph level for sentiment information, we essentially created a geospatial sentiment-map based on the user-contributed information contained in the articles of the respective travelblogs.

11.2 Future Work

During the course of this dissertation, we have identified several interesting research directions to follow for future work.

The first obvious direction, is to expand our SALT framework to support additional types of shortest-path queries. There are multiple types of queries on road networks

that have only been addressed by secondary-storage, indexing methods, tested on road-networks of city-scale. Such approaches cannot scale for continental road networks or adapt for dynamic road networks with frequent traffic updates. In this sense, answering those queries within the SALT framework would further expand its versatility and flexibility and would significantly increase its commercial applicability.

Second, it would also be interesting to apply the novel shortest-path algorithms developed in this dissertation as subroutines to map-matching algorithms to significantly boost their performance in the context of real-time, live-traffic systems. This way, the aforementioned map-matching algorithms could be extended to handle continental road networks and be fast enough for handling even more frequent traffic updates.

Moreover, since our algorithmic methods were only tested on road-networks, it would be interesting to see how the proposed methods can handle social / citation networks. In that case, the proposed methods would probably require serious changes and optimizations to adapt to those huge, scale-free networks.

Finally, we seriously believe that the architecture of the system we have presented for crowdsourcing computing resources, using Web-browsers and Javascript as our computing platform, may be easily expanded to support various distributed algorithms besides shortest-paths preprocessing. Therefore, it would make sense to use the suggested platform for several unrelated problems on a wide range of practical applications.

In conclusion, there are multiple interesting and novel topics relevant to this dissertation and we really hope that this thesis will be a starting point for igniting further research in the suggested directions.

Bibliography

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 230–241, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European conference on Algorithms*, ESA'12, pages 24–35, 2012.

[3] M. D. Adelfio, M. D. Lieberman, H. Samet, and K. A. Firozvi. Ontuition: intuitive data exploration via ontology navigation. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 540–541, New York, NY, USA, 2010. ACM.

[4] Amazon. EC2 FAQ. `http://aws.amazon.com/ec2/faqs/`, 2012.

[5] E. Amitay, N. Har'El, R. Sivan, and A. Soffer. Web-a-where: geotagging web content. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pages 273–280, New York, NY, USA, 2004. ACM.

[6] Apache. Apache JMeter [Online]. `http://jmeter.apache.org/`.

[7] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 55–66. Springer Berlin Heidelberg, 2013.

[8] J. Ban, R. Herring, P. Hao, and A. M. Bayen. Delay pattern estimation for signalized intersections using sampled travel times. *Transportation Research Record*, pages 109–119, 2009.

[9] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, Apr. 2007.

[10] Bast, Hannah and Delling, Daniel and Goldberg, Andrew and Müller-Hannemann, Matthias and Pajor, Thomas and Sanders, Peter and Wagner, Dorothea and Werneck, Renato. Route Planning in Transportation Networks. Technical report, Microsoft Research, 2014.

[11] G. V. Batz, D. Delling, P. S, and C. Vetter. Time-dependent contraction hierarchies. In *In Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX*, pages 97–105. SIAM, 2009.

[12] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *J. Exp. Algorithmics*, 15:2.3:2.1–2.3:2.31, March 2010.

[13] V. Bauer, J. Gamper, R. Loperfido, S. Profanter, S. Putzer, and I. Timko. Computing isochrones in multi-modal, schedule-based transport networks. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, GIS '08, pages 78:1–78:2, New York, NY, USA, 2008. ACM.

[14] M. Baum, J. Dibbelt, T. Pajor, and D. Wagner. Energy-optimal routes for electric vehicles. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 54–63, New York, NY, USA, 2013.

[15] W. J. Black, J. McNaught, A. Vasilakopoulos, K. Zervanou, B. Theodoulidis, and F. Rinaldi. Cafetiere: Conceptual annotations for facts, events, terms, individual entities and relations. Technical report, Jan 2005. Parmenides Technical Report, TR-U4.3.1.

[16] J. Bollen, H. Mao, and X.-J. Zeng. Twitter mood predicts the stock market. *Journal of Computational Science*, 2(1):1–8, 2011.

[17] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *Proc. 31st VLDB Conference*, pages 853–864, 2005.

[18] D. Buscaldi and P. Rosso. A conceptual density-based approach for the disambiguation of toponyms. *Int. J. Geogr. Inf. Sci.*, 22:301–313, January 2008.

[19] M. Chen, R. A. Chowdhury, V. Ramachandran, D. L. Roche, and L. Tong. Priority queues and Dijkstra's algorithm, 2007.

[20] P. Clough. Extracting metadata for spatially-aware information retrieval on the internet. In *Proceedings of the 2005 workshop on Geographic information retrieval*, GIR '05, pages 25–30, New York, NY, USA, 2005. ACM.

[21] A. Crooks, D. Pfoser, A. Jenkins, A. Croitoru, S. Karagiorgou, A. Efentakis, G. Lamprianidis, D. Smith, and A. Stefanidis. Crowdsourcing Urban form and Function. *International Journal of Geographical Information Science*, To appear.

[22] D. Delling, A. Goldberg, T. Pajor, and R. Werneck. Customizable route planning in road networks. working paper, submitted for publication., 2013.

[23] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 921–931, Washington, DC, USA, 2011.

[24] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 376–387, Berlin, Heidelberg, 2011.

[25] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph partitioning with natural cuts. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1135–1146, 2011.

[26] D. Delling, A. V. Goldberg, and R. F. F. Werneck. Faster batched shortest paths in road networks. In *ATMOS*, pages 52–63, 2011.

[27] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks. Lecture Notes In Computer Science*. Springer, 2009.

[28] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 52–65, Berlin, Heidelberg, 2007. Springer-Verlag.

[29] D. Delling and R. Werneck. Customizable point-of-interest queries in road networks. Technical Report MSR-TR-2013-90, September 2013.

[30] D. Delling and R. Werneck. Faster customization of road networks. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer Berlin Heidelberg, 2013.

[31] D. Delling and R. F. Werneck. Customizable point-of-interest queries in road networks. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 500–503, New York, NY, USA, 2013. ACM.

[32] J.-Y. Delort. Hierarchical cluster visualization in web mapping systems. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 1241–1244, New York, NY, USA, 2010. ACM.

[33] C. Demetrescu, A. V. Goldberg, and D. Johnson. *The shortest path problem. Ninth DIMACS implementation challenge, Piscataway, NJ, USA, November 13–14, 2006. Proceedings.* DIMACS Book 74. AMS , 2009.

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

[35] J. Ding, L. Gravano, and N. Shivakumar. Computing geographical scopes of web resources. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 545–556, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[36] E. Drymonas, A. Efentakis, and D. Pfoser. Opinion mapping travelblogs. In *Proc. Terra Cognita Workshop (in conjunction with International Semantic Web Conf.)*, pages 23–36, 2011.

[37] E. Drymonas and D. Pfoser. Geospatial route extraction from texts. In *DMG '10: Proceedings of the 1st ACM SIGSPATIAL International Workshop on Data Mining for Geoinformatics*, pages 29–37, New York, NY, USA, 2010. ACM.

[38] A. Efentakis, S. Brakatsoulas, N. Grivas, G. Lamprianidis, K. Patroumpas, and D. Pfoser. Towards a flexible and scalable fleet management service. In *Proceedings of the Sixth ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS '13, pages 79:79–79:84, New York, NY, USA, 2013. ACM.

[39] A. Efentakis, S. Brakatsoulas, N. Grivas, and D. Pfoser. Crowdsourcing turning restrictions for openstreetmap. In *EDBT/ICDT Workshops*, pages 355–362, 2014.

[40] A. Efentakis, N. Grivas, G. Lamprianidis, G. Magenschab, and D. Pfoser. Isochrones, traffic and demographics. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 538–541, New York, NY, USA, 2013. ACM.

[41] A. Efentakis, N. Grivas, D. Pfoser, and Y. Vassiliou. Crowdsourcing turning-restrictions from map-matched trajectories. *Mining Urban Data - Special Issue - Information Systems (Elsevier)*, Submitted.

[42] A. Efentakis and G. Lamprianidis. SimpleFleet Deliverable D6.5. SimpleFleet Online Demo [Online]. `http://www.simplefleet.eu/?page_id=84`, 2013.

[43] A. Efentakis and D. Pfoser. Optimizing landmark-based routing and preprocessing. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS '13, pages 25–30, New York, NY, USA, 2013.

[44] A. Efentakis and D. Pfoser. GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem. In A. Schulz and D. Wagner, editors, *Algorithms - ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 358–370. Springer Berlin Heidelberg, 2014.

[45] A. Efentakis, D. Pfoser, and Y. Vassiliou. SALT. A unified framework for all shortest-path query variants on road networks. *CoRR*, abs/1411.0257, 2014.

[46] A. Efentakis, D. Pfoser, and A. Voisard. Efficient data management in support of shortest-path computation. In *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, CTS '11, pages 28–33, New York, NY, USA, 2011. ACM.

[47] A. Efentakis, D. Theodorakis, and D. Pfoser. Crowdsourcing computing resources for shortest-path computation. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 434–437, New York, NY, USA, 2012. ACM.

[48] J. Gamper, M. Böhlen, and M. Innerebner. Scalable computation of isochrones with network expiration. In *Proceedings of the 24th international conference on Scientific and Statistical Database Management*, SSDBM'12, pages 526–543, Berlin, Heidelberg, 2012. Springer-Verlag.

[49] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008.

[50] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2004.

[51] A. V. Goldberg and R. F. F. Werneck. Computing point-to-point shortest paths from external memory. In *Algorithm Engineering and Experimentation*, 2005.

[52] Google. Google Earth [Online]. `https://www.google.com/earth/`.

[53] Google. Keyhole Markup Language [Online]. `https://developers.google.com/kml/documentation/kmlreference`.

[54] Google. Street View [Online]. `https://www.google.com/maps/views/streetview?gl=us`.

[55] Google. The Directions API [Online]. `https://developers.google.com/maps/documentation/directions/`.

[56] Google. Encoded Polyline Algorithm Format [Online]. `https://developers.google.com/maps/documentation/utilities/polylinealgorithm`, 2013.

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

[58] T. Hastie and W. Stuetzle. Principal curves. *Journal of the American Statistical Association*, 84(406):502–516, 1989.

[59] R. Herring, P. Abbeel, A. Hofleitner, and A. Bayen. Estimating arterial traffic conditions using sparse probe data. *Proceedings of the 13th International IEEE Conference on Intelligent Transportation Systems, September 19-22, Madeira Island, Portugal*, pages 929–936, 2010.

[60] R. Hundt. Loop recognition in c++/java/go/scala. In *Proceedings of Scala Days 2011*.

[61] T. Ikeda, M. Y. Hsu, H. Imai, S. Nishimura, H. S. Moura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. 1994.

[62] D. Ingalls. The lively kernel: just for fun, let's take javascript seriously. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 9:1–9:1, New York, NY, USA, 2008. ACM.

[63] C. S. Jensen, J. Kolářvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, GIS '03, pages 1–8, New York, NY, USA, 2003. ACM.

[64] R. Jianu and D. Laidlaw. Visualizing gene co-expression as google maps. In G. Bebis, R. Boyle, B. Parvin, D. Koracin, R. Chung, R. Hammound, M. Hussain, T. Kar-Han, R. Crawfis, D. Thalmann, D. Kao, and L. Avila, editors, *Advances in Visual Computing*, volume 6455 of *Lecture Notes in Computer Science*, pages 494–503. Springer Berlin / Heidelberg, 2010.

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

[66] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 1 edition, 2000.

[67] L. Kabrt. Travel Time Analysis. `http://code.google.com/p/traveltimeanalysis/source/browse`, 2010.

[68] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998.

[69] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[70] H. M. Kienle. It's about time to take javascript (more) seriously. *IEEE Software*, 27:60–62, 2010.

[71] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed time-dependent contraction hierarchies. In *Proceedings of the 9th international conference on Experimental Algorithms*, SEA'10, pages 83–93, Berlin, Heidelberg, 2010. Springer-Verlag.

[72] E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *IN: 9th DIMACS Implementation Challenge*, 2006.

[73] K. C. K. Lee, W.-C. Lee, and B. Zheng. Fast object search on road networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1018–1029, New York, NY, USA, 2009. ACM.

[74] K. C. K. Lee, W.-C. Lee, B. Zheng, and Y. Tian. Road: A new spatial object search framework for road networks. *IEEE Trans. on Knowl. and Data Eng.*, 24(3):547–560, Mar. 2012.

[75] J. L. Leidner. Toponym resolution in text: annotation, evaluation and applications of spatial grounding. *SIGIR Forum*, 41:124–126, December 2007.

[76] M. D. Lieberman, H. Samet, and J. Sankaranarayanan. Geotagging with local lexicons to build indexes for textually-specified spatial data. In *International Conference on Data Engineering*, pages 201–212, 2010.

[77] M. D. Lieberman, H. Samet, J. Sankaranarayanan, and J. Sperling. Steward: architecture of a spatio-textual search engine. In *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, GIS '07, pages 25:1–25:8, New York, NY, USA, 2007. ACM.

[78] X. Liu, F. Lu, H. Zhang, and P. Qiu. Estimating beijing's travel delays at intersections with floating car data. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS '12, pages 14–19, New York, NY, USA, 2012. ACM.

[79] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate gps trajectories. In *Proc. 17th ACM SIGSPATIAL GIS conf.*, GIS '09, pages 352–361, New York, NY, USA, 2009. ACM.

[80] D. Luxen and C. Vetter. Real-time routing with openstreetmap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 513–516, New York, NY, USA, 2011. ACM.

[81] B. Maps. Routes API [Online]. `http://msdn.microsoft.com/en-us/library/ff701705.aspx`.

[82] S. Marciuska and J. Gamper. Determining objects within isochrones in spatial network databases. In *Proceedings of the 14th east European conference on Advances in databases and information systems*, ADBIS'10, pages 392–405, Berlin, Heidelberg, 2010. Springer-Verlag.

[83] J. Maue, P. Sanders, and D. Matijevic. Goal directed shortest path queries using precomputed cluster distances. In *5th Workshop on Experimental Algorithms (WEA), Number 4007 IN LNCS*, pages 316–328. Springer, 2006.

[84] K. S. McCurley. Geospatial mapping and navigation of the web. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 221–229, New York, NY, USA, 2001. ACM.

[85] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin, 2008.

[86] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *J. Exp. Algorithmics*, 11, February 2007.

[87] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 43–54. VLDB Endowment, 2006.

[88] MySQL. The world's most popular open source database. `http://www.mysql.com`, 2012.

[89] Nginx. `http://wiki.nginx.org/Main`, 2012.

[90] O. A. Nielsen, R. D. Frederiksen, and N. Simonsen. Using expert system rules to establish data for intersections and turns in road networks. *International Transactions in Operational Research*, 5(6):569 – 581, 1998.

[91] Okeanos. õkeanos IaaS [Online]. `https://okeanos.grnet.gr/home/`, 2013.

[92] OpenStreetMap. Stats - openstreetmap wiki [online]. `http://wiki.openstreetmap.org/wiki/Stats#OpenStreetMap_Statistics_Available`, 2011.

[93] OpenStreetMap. Relation:restriction [Online]. `http://wiki.openstreetmap.org/wiki/Relation:restriction`, 2013.

[94] P. Passenger. http://www.modrails.com, 2012.

[95] L. D. Paulson. Developers Shift to Dynamic Programming Languages. *Computer*, 40(2):12–15, Feb. 2007.

[96] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-object representations. In *Proceedings of the 6th International Symposium on Advances in Spatial Databases*, SSD '99, pages 111–132, London, UK, UK, 1999.

[97] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 867–876, New York, NY, USA, 2009. ACM.

[98] R. S. Purves, P. Clough, C. B. Jones, A. Arampatzis, B. Bucher, D. Finch, G. Fu, H. Joho, A. K. Syed, S. Vaid, and B. Yang. The design and implementation of spirit: a spatially aware search engine for information retrieval on the internet. *Int. J. Geogr. Inf. Sci.*, 21:717–745, January 2007.

[99] QGIS. A Free and Open Source Geographic Information System [Online]. http://www.qgis.org/.

[100] G. Quercini, H. Samet, J. Sankaranarayanan, and M. D. Lieberman. Determining the spatial reader scopes of news sources using local lexicons. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 43–52, New York, NY, USA, 2010. ACM.

[101] R. E. Roth, K. S. Ross, B. G. Finch, W. Luo, and A. M. MacEachren. A user-centered approach for designing and developing spatiotemporal crime analysis tools. Zurich, Switzerland, 14-17th September, 2010 2010. GIScience.

[102] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 43–54, New York, NY, USA, 2008. ACM.

[103] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5:312–327, 1999.

[104] P. Sanders, D. Schultes, and C. Vetter. Mobile route planning. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 732–743, Berlin, Heidelberg, 2008. Springer-Verlag.

[105] P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In *Algorithms - ESA 2011*, Lecture Notes in Computer Science, pages 469–480. 2011.

[106] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 16–29, 2012.

[107] R. E. Schapire and Y. Singer. BoosTexter: A Boosting-based System for Text Categorization. *Machine Learning*, 39(2/3):135–168, 2000.

[108] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: an empirical case study from public railroad transport. *J. Exp. Algorithmics*, 5, Dec. 2000.

[109] SimpleFleet. Democratizing fleet management [online]. `http://www.simplefleet.eu`, 2013.

[110] N. Stokes, Y. Li, A. Moffat, and J. Rong. An empirical study of the effects of nlp components on geographic ir performance. *Int. J. Geogr. Inf. Sci.*, 22:247–264, January 2008.

[111] L. Sun. *An approach for intersection delay estimate based on floating vehicles*. Dissertation for Master Degree. Beijing: Beijing University of Technology(in Chinese), 2007.

[112] B. E. Teitler, M. D. Lieberman, D. Panozzo, J. Sankaranarayanan, H. Samet, and J. Sperling. Newsstand: a new view on news. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, GIS '08, pages 18:1–18:10, New York, NY, USA, 2008. ACM.

[113] D. Theodorakis. *Map Reduce Implementation on Spatial Data with Dijkstra's algorithm*. Dissertation for diploma degree. School of Electrical and Computer Engineering of the National Technical University of Athens (in Greek), 2011.

[114] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[115] 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 *Proc. 20th CIKM conf.*, pages 1785–1794, 2011.

[116] F. Viti and H. J. van Zuylen. Modeling queues at signalized intersections. *Transportation Research Record: Journal of the Transportation Research Board*, (1883):68–77, 2004.

[117] H. Wang and R. Zimmermann. Processing of continuous location-based range queries on moving objects in road networks. *Knowledge and Data Engineering, IEEE Transactions on*, 23(7):1065–1078, July 2011.

[118] C. Wenk, R. Salas, and D. Pfoser. Addressing the need for map-matching speed: Localizing global curve-matching algorithms. In *Proc. 18th SSDBM conf.*, pages 379–388, 2006.

[119] J. Wiebe and E. Riloff. Creating subjective and objective sentence classifiers from unannotated texts. In *Proceedings of the 6th International Conference on Intelligent Text Processing and Computational Linguistics (CICLing-2005)*, pages 486–497, Mexico City, Mexico, 2005.

[120] T. Wilson, P. Hoffmann, S. Somasundaran, J. Kessler, J. Wiebe, Y. Choi, C. Cardie, E. Riloff, and S. Patwardhan. Opinionfinder: a system for subjectivity analysis. In *Proceedings of HLT/EMNLP on Interactive Demonstrations*, HLT-Demo '05, pages 34–35, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[121] S. Winter. Modeling costs of turns in route planning. *GeoInformatica*, 6(4):345–361, 2002.

[122] H. Zhang, F. Lu, L. Zhou, and Y. Duan. Computing turn delay in city road network with gps collected trajectories. In *Proceedings of the 2011 International Workshop on Trajectory Data Mining and Analysis*, TDMA '11, pages 45–52, New York, NY, USA, 2011. ACM.

[123] J. Zhang, H. Shi, and Y. Zhang. Self-organizing map methodology and google maps services for geographical epidemiology mapping. *Digital Image Computing: Techniques and Applications*, 0:229–235, 2009.

[124] M. Zhao and X. Li. Deriving average delay of traffic flow around intersections from vehicle trajectory data. *Frontiers of Earth Science*, 7(1):28–33, 2013.

[125] Y. Zheng and X. Zhou, editors. *Computing with Spatial Trajectories*. Springer, 2011.

[126] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *Proceedings of the 22nd ACM International Conference on Conference on Information Knowledge Management*, CIKM '13, pages 39–48, New York, NY, USA, 2013. ACM.

[127] W. Zong, D. Wu, A. Sun, E.-P. Lim, and D. H.-L. Goh. On assigning place names to geography related web pages. In *Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '05, pages 354–362, New York, NY, USA, 2005. ACM.