NOVEL SPATIAL QUERY PROCESSING TECHNIQUES FOR SCALING LOCATION BASED SERVICES

A Thesis
Presented to
The Academic Faculty

by

Peter Pesti

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2012

NOVEL SPATIAL QUERY PROCESSING TECHNIQUES FOR SCALING LOCATION BASED SERVICES

Date Approved: September 24, 2012

To my parents.

ACKNOWLEDGEMENTS

My dissertation would not have been possible without the generous help and advice from my mentors, colleagues, friends and family. I would like to express my gratitude to those who have enabled me to travel the long and winding road of the PhD journey leading to this dissertation. Here I can only mention a few of these individuals.

First and foremost, I would like to acknowledge the strong support and guidance received from my advisor, Ling Liu. She was the one who encouraged me most to start my PhD years, convincing me that I had the secret sauce that would allow me to succeed. Throughout my time at Georgia Tech, she has unwaveringly supported me both in research and in spirit. I appreciate her contribution towards the development and sharpening of my research skills. She has taught me to focus on the bigger picture and draw out the overarching ideas, while giving me great freedom to dig into the tiny details of our research problems. I'm indebted to her as a friend – one who has cheered me up when hitting a road bump, and has enthusiastically opened up perspectives for me in life that I did not see before. I have learned from her a great deal, and I hope to be able to pass much of it to others in the future.

I would like to give my thanks to the members of my thesis committee: Edward Omiecinski, Wonik Choi, Leo Mark and Calton Pu. Their feedback and critiques have contributed significantly to my dissertation and PhD work, from my qualification exam on to my defense.

Special thanks go out to my internship mentors and collaborators at IBM T. J. Watson Research Center, Microsoft Research and Google: Arun Iyengar, Jeremy Elson, Jon Howell, Drew Steedly, Matt Uyttendaele, Parag Samdadiya, Eli Brandt and Ashesh Bakshi. They have allowed me to broaden my perspective and gain industry experience, while working on interesting problems closely tied to my research at Georgia Tech.

I am grateful to my colleagues in the DiSL research group for their friendship, and being always ready to discuss both research and affairs far beyond the walls of university. I would especially like to give my thanks to Bhuvan Bamba, Ting Wang, Danesh Irani, Shicong Meng, Balaji Palanisamy, Matt Weber, Yuzhe Tang, Gong Zhang, Sankaran Sivathanu, Myungcheol Doo, Qingyang Wang and Kisung Lee.

Most importantly, I would like to acknowledge the support and love of my parents, who enabled me to set my sights high and turn dreams to reality.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Location based services (LBS) are gaining widespread user acceptance and increased daily usage. GPS based mobile navigation systems (Garmin), location-related social network updates and check-ins (Facebook), location-based games (Nokia), friend queries (Foursquare) and ads (Google) are some of the popular LBSs available to mobile users today. Despite these successes, current user services fall short of a vision where mobile users could ask for continuous location-based services with always-up-to-date information around them, such as the list of friends or favorite restaurants within 15 minutes of driving. Providing such a location based service in real time faces a number of technical challenges.

In this dissertation research, we propose a suite of novel techniques and system architectures to address some known technical challenges of continuous location queries and updates. Our solution approaches enable the creation of new, practical and scalable location based services with better energy efficiency on mobile clients and higher throughput at the location servers. Our first contribution is the development of RoadTrack, a road network aware and query-aware location update framework and a suite of algorithms. A unique characteristic of RoadTrack is the innovative design of encounter points and system-defined precincts to manage the desired spatial resolution of location updates for different mobile clients while reducing the complexity and energy consumption of location update strategies. The second novelty of this dissertation research is the technical development of Dandelion data structures and algorithms that can deliver superior performance for the periodic re-evaluation of continuous road-network distance based location queries, when compared with the alternative of repeatedly performing a network expansion along a mobile users trajectory. The third contribution of this dissertation research is the FastExpand

algorithm that can speed up the computation of single-issue shortest-distance road network queries. Finally, we have developed the open source GT MobiSim mobility simulator, a discrete event simulation platform to generate realistic driving trajectories for real road maps. It has been downloaded and utilized by many to evaluate the efficiency and effectiveness of the location query and location update algorithms, including the research efforts in this dissertation.

CHAPTER I

INTRODUCTION

Location based services (LBS) are gaining widespread user acceptance and increased daily usage. GPS based mobile navigation systems (Garmin), location-related social network updates and check-ins (Facebook), location-based games (Nokia), friend queries (Foursquare) and ads (Google) are just some of the popular LBS available to mobile users today. According to the International Telecommunications Union (ITU), there were 5.3 billion mobile subscribers worldwide (or 77 percent of the world population) in 2010, and one out of six mobile subscribers could access the mobile Internet. Usage is expected to double within five years as mobile overtakes the PC as the most popular way of getting on the Web. Many consumers prefer mobile browsers for banking, travel, shopping, local info, news, video, sports and blogs, and prefer apps for games, social media, maps and music. Additionally, many enterprises use location based applications such as vehicle fleet management and urban traffic analytics (IBM).

Despite these successes, current user services fall short of a vision where mobile users could ask for continuous location-based services with always-up-to-date information about the world around them. Consider a simple continuous location query, where a moving user asks for a constant update of the list of restaurants and friends within 10 minutes of driving from her current location, while she is on the move. Providing such a service in real time faces a number of technical challenges due to limited battery, limited network and computational resources. First, the user is interested in locations and directions that she can actually follow and drive to (road network travel distance), rather than those which are physically close (Euclidean distance). Finding the coverage of a network distance based query in a huge road network (graph) is computationally expensive. This problem is seriously aggravated when such location-based services need to be delivered continuously in real-time with super-fast response time. Second, the ability to obtain the up-to-date location of mobile users is critical to both the quality of location queries and the range of location query services one can offer. However, it is widely recognized that frequent updates cause high update processing cost at the location server and high power consumption at the mobile clients. Unfortunately, existing location update strategies are inefficient because they are common to all mobile users and they assume that location updates of mobile clients are independent of each other. We argue that location update is an essential metric for performance optimization of real time LBS delivery. Intelligent customization and differentiation are critical to both the effectiveness of location update management and location query quality assurance.

In this dissertation research, we propose a suite of novel techniques and system architectures to address the above challenges. Our solution approaches enable the creation of new, practical and scalable location based services with better energy efficiency on the clients and higher throughput at the location servers. First, we propose the Dandelion algorithm and a set of specialized data structures that speed up the periodic re-evaluation of continuous road-network distance based location queries, when compared with the alternative of performing a network expansion along a mobile users trajectory repeatedly while users are on the move. The key idea of our Dandelion development is to reduce the amount of unnecessary re-computations of continuous location queries by careful identification, administration and incremental adjustment of key coverage locations in the graph. Although the Dandelion algorithm is fast and effective, it can only improve the subsequent computations of continuous road-network location queries. The second contribution of this dissertation is the development of the FastExpand algorithm that can speed up the initial computation of a road network query, e.g., the coverage of a range query, using a hybrid expansion approach. The main idea of the FastExpand development is to partition the large road-network (graph) into smaller units in order to perform the shortest path computations using a multi-step process. Concretely, we precompute and use shortest path shortcuts inside precincts, and only perform local graph search near the focal location and in the border regions of the query. Our third technical contribution is the development of RoadTrack, a road network aware and query-aware location update framework and a suite of algorithms. A unique characteristic of RoadTrack is the ability to conserve the battery power of mobile clients and reduce server bandwidth and load by making the location update schedule query-aware through three novel techniques. We introduce the concept of encounter points as a baseline query awareness mechanism to control and differentiate location update strategies for mobile clients in the vicinity of active location queries, while meeting the need of location query evaluation. We employ system-defined precincts to manage the desired spatial resolution of location updates for different mobile clients and to control the scope of query awareness to be capitalized on by a location update strategy, thus reducing the complexity of graph calculations and network usage. Finally our road-network based check-free interval optimization further enhances the effectiveness of the RoadTrack query-aware location update scheduling algorithm, offering significant cost reduction for location update management at both mobile clients and location servers. Finally, we have developed the GT MobiSim mobility simulator, which is used to generate realistic driving trajectories for real road maps, and serves as the discrete event simulation platform for evaluating the efficiency and effectiveness of the location query and location update algorithms in the dissertation. This mobility simulator has been downloaded more than a hundred times since its first public release.

1.1 Roadmap

This thesis is organized as a series of chapters, each one dedicated to a topic within the scope of spatial query-processing techniques and location based services in general. Each chapter gives a brief overview of the problem motivation and formulation, before delving into the technical details and our contributions. Experimental results highlight the performance of our proposed solutions under various realistic scenarios, showcasing the flexibility of our algorithms under the most salient parameterizations. Our chapters also survey the related work.

Chapter 2 is dedicated to RoadTrack, our road network aware and query-aware location update framework and a suite of algorithms, focusing on efficient and scalable query answering in an environment populated by a large number of users.

Chapter 3 presents our Dandelion algorithms and a set of specialized data structures that speed up the periodic re-evaluation of continuous road-network distance based location queries, when compared with the alternative of performing a network expansion along a mobile users trajectory repeatedly while users are on the move.

Chapter 4 is a presentation of our FastExpand algorithm for the fast evaluation of single-issue road-network distance based location queries.

Chapter 5 continues our focus on location based services from a different perspective, whereby the proposed MapStitcher algorithm and processing tool allows the semi-automatic creation of self-made aerial imagery layers for GIS web applications.

The thesis concludes in Chapter 6.

1.2 Bibliographic notes

Material in Chapter 2 appears in a paper co-authored with Bhuvan Bamba, Arun Iyengar, Matt Weber and Ling Liu [37]. Material in Chapter 5 appears in a paper co-authored with Jeremy Elson, Jon Howell, Drew Steedly and Matthew Uyttendaele [36].

CHAPTER II

ROADTRACK

Mobile commerce and location based services (LBS) are some of the fastest growing IT industries in the last five years. Location update of mobile clients is a fundamental capability in mobile commerce and all types of LBS. Higher update frequency leads to higher accuracy, but incurs unacceptably high cost of location management at the location servers. We propose ROADTRACK – a road-network based, query-aware location update framework with two unique features. First, we introduce the concept of precincts to control the granularity of location update resolution for mobile clients that are not of interest to any active location query services. Second, we define query encounter points for mobile objects that are targets of active location query services, and utilize these encounter points to define the adequate location update schedule for each mobile. The ROADTRACK framework offers three unique advantages. First, encounter points as a fundamental query awareness mechanism enable us to control and differentiate location update strategies for mobile clients in the vicinity of active location queries, while meeting the needs of location query evaluation. Second, we employ system-defined precincts to manage the desired spatial resolution of location updates for different mobile clients and to control the scope of query awareness to be capitalized by a location update strategy. Third, our road-network based check-free interval optimization further enhances the effectiveness of the ROADTRACK query-aware location update scheduling algorithm. This optimization provides significant cost reduction for location update management at both mobile clients and location servers. We evaluate the ROADTRACK location update approach using a real world road-network based mobility simulator. Our experimental results demonstrate that the ROADTRACK query aware location update approach outperforms existing representative location update strategies in terms of both client energy efficiency and server processing load.

A version of this chapter was published as a paper co-authored with Bhuvan Bamba, Arun Iyengar, Matt Weber and Ling Liu [37].

2.1 Introduction

We are entering a wireless and mobile Internet era where people and vehicles are connected at all times. In the past five years we have witnessed an astonishing growth of mobile commerce and location based applications and services, which not only extend many traditional businesses into new product offerings (e.g., location based advertisement, location based entertainment) but also create many opportunities for new businesses and innovations. Consider a metropolitan area with hundreds of thousands of vehicles. Drivers and passengers in these vehicles are interested in information relevant to their trips. For example, some driver would like her vehicle to continuously display on a map the list of Starbucks coffee shops within 10 miles of her current location. Another driver may want to monitor the traffic conditions five miles ahead of its current location (e.g., traffic flow speed). The challenge is how to effectively monitor the location updates of mobile users and continuously serve location queries (traffic conditions, parking spaces, Starbucks coffee shops) with an acceptable delay, overhead, and accuracy, as the mobile users move on the road.

There are two key performance challenges that may affect the system scalability and service quality in future mobile systems supporting location-dependent services and applications: (1) the high cost of network bandwidth and energy consumed on the mobile clients for frequent location tracking and updates at the location servers; and (2) the challenge of scaling large amount of location updates at the location server as the number of mobile clients demanding to be tracked increases in a location determination system. Furthermore, handling frequent load peaks at location update synchronization points is also a challenge, since the server has to simultaneously handle location updates from a large number of mobile clients, and re-evaluate all registered spatial location query services.

Location Update Problems and Existing Approaches

Monitoring location updates and evaluation of location queries over static and moving objects upon location updates have become the necessity for many mobile systems and location-based applications, such as fleet management, cargo tracking, child care, and location-based advertisement and entertainment. Frequent updates cause high update processing cost at the location server and high power consumption at the mobile clients [1]. Some European mobile service providers have started the cost-based location management for mobile object tracking. For instance, different pricing models are applied to high frequency location updates at different time intervals, such as every three minutes, every one minute, every 30 seconds, and so forth.

In contrast to location determination systems where localization techniques are employed to determine the position of a mobile subscriber within the area serviced by the wireless network, the location update management addresses the problem of when and where to update the locations of mobile subscribers currently hosted in the system. Representative location update strategies to date include periodic update (time based scheme), point-based update using dead-reckoning, velocity vector based update, and segment based updates [10]. However, existing location update strategies are inefficient because i) they are common to all mobile users, and ii) they assume that location updates of mobile clients are autonomous and all mobile users should manage their location updates using a uniform strategy. To the best of our knowledge, no customization or differentiation is incorporated to the design of location update management strategies.

We argue that, as mobile and hand-held devices become more pervasive, more capable, and both GPS and WiFi enabled [42, 23], as the operation cost of location update management continues to grow, these assumptions are no longer realistic. For instance, most of the mobile systems and applications today need to manage a large and evolving number of mobile objects. Often, only a subset of mobile objects is of interest to registered location query services. Thus, tracking location updates of all mobile clients uniformly is no longer a cost effective solution. It is obvious that the location update strategy for those clients that are of no interest to any nearby and active location query services should be different from and less costly compared to the location update strategy designed for mobile objects that are the targets of active location query services in the system.

Motivated by these observations, in this chapter we present ROADTRACK – a road-network based, query-aware location update framework by introducing precincts and encounter points as two basic techniques to confine location updates to the need of existing location query services. These two basic building blocks enable us to effectively differentiate and manage location updates for mobile objects traveling on road networks. We utilize precincts to manage the spatial resolution of location updates for mobile clients that are not immediate targets of any existing location query services. We introduce encounter points to implement the query-aware location update strategy for mobile clients nearby active location queries. By combining precincts and encounter points, we can balance the benefit and cost of query awareness and speed up the computation of encounter points. The ROADTRACK location update management offers three unique advantages. First, encounter points as a fundamental query awareness mechanism enable us to control and differentiate location update strategies for mobile clients in the vicinity of active location queries from the rest. Second, by employing system-defined precincts, we can effectively manage the desired spatial resolution of location updates for mobile clients with different needs for query awareness. Third but not the least, we improve the efficiency of ROADTRACK location update approach by employing a suite of road-network based check-free interval optimization techniques. We evaluate the ROADTRACK approach to location update management based on a real world road-network mobility simulator [34]. Our experimental results show that by making location update management *query aware*, ROADTRACK approach significantly outperforms existing representative location update strategies in terms of both client energy efficiency and server processing load.

The rest of the chapter is organized as follows: We outline the reference system model and discuss the design philosophy through an analysis of existing representative location update strategies in Section 2.2. In Section 2.3, we introduce the concept, the computation, and the usage of encounter points and the precinct and encounter based location update strategy, including the data structure used at both the server and the client side. We present the encounter points based check-free interval optimization in Section 2.4. Section 2.6 reports our experimental evaluation on the effectiveness of our ROADTRACK query aware location update approach. We conclude the chapter with related work and a summary of contributions.

2.2 System Overview

A location update and monitoring system typically consists of a location database server, some base-stations, application servers, and a large number of mobile objects (mobile clients) and static objects (such as gas stations, restaurants, and so on). The location database server (location server for short) manages the locations of the moving objects. The application servers register location queries of interest, and synchronize with the location server to continuously evaluate the queries against location updates.

Figure 1 gives an architectural overview of the reference location monitoring system used in the context of ROADTRACK development. We assume that mobile clients and the location server have a local copy of the same road network database that constrains the movement of the clients; clients may store this on an SD card. For the clients with limited storage, a tile based partitioning of the road network map can be used [31]. We assume that the mobile clients are able to communicate with the server through wireless data channel, and they have computing capabilities to run our light-weighted road network locator, which uses a static R-tree index on road segments to find their own road network locations based on their GPS positions through map matching. Mobile clients may also obtain their positions from the location determination system they subscribe to, such as Google's locator service available on iPhone and other hand-held devices.

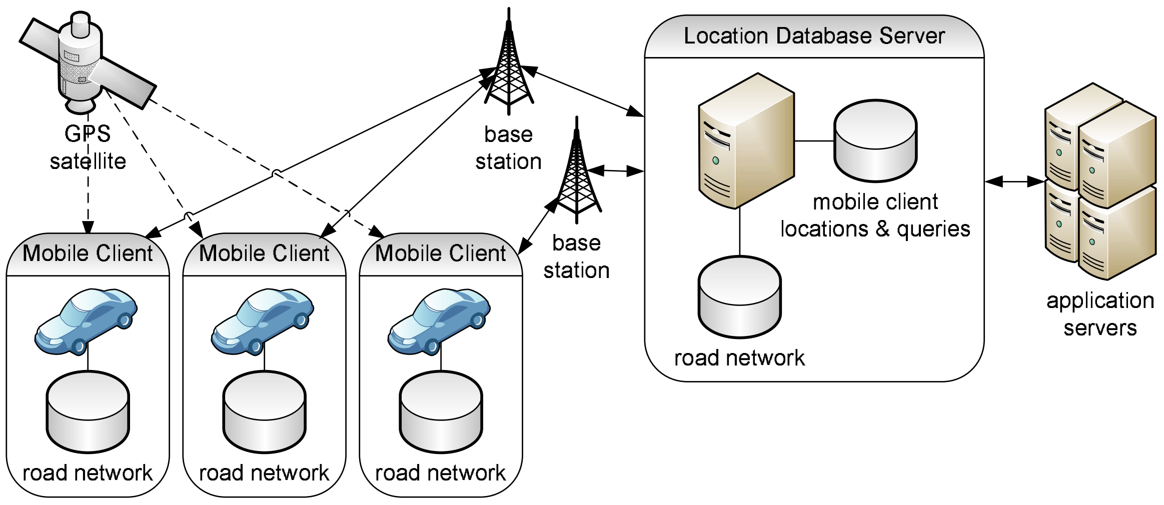


Figure 1: Overview of the system architecture

2.2.1 Road network model

The road network is represented by a single undirected graph $G = (\mathcal{V}, \mathcal{E})$, composed of the junction nodes $\mathcal{V} = \{n_0, n_1, \ldots, n_N\}$ and undirected edges $\mathcal{E} = \{n_i n_j | n_i, n_j \in \mathcal{V}\}$. In this chapter we frequently refer to an edge $n_i n_j$ as a road segment connecting the two end nodes n_i and n_j. The listing order of the two end nodes of a segment $n_i n_j$ serves as the basis to determine the direction of the *progress* coordinate axis from node n_i to node n_j along the segment $n_i n_j$. In other words, the segment $n_i n_j$ runs from $p = 0$ at the first listed node (n_i) to $p = length(n_i n_j)$ at the second listed node (n_j). Though in this chapter we model the road network using undirected graphs for simplicity, our methods can be extended to directed graphs. Junction nodes have either two or more connecting road segments, or are dead-end nodes with only one connecting road segment. A *road network location*, denoted by $L = (n_i n_j, p)$, is a tuple of two elements: a road network segment $n_i n_j$ and the *progress* p along the segment. The road network distance is used as the distance metric in our system. The distance between two locations $L_1 = (n_{i_0} n_{i_1}, p_1)$ and $L_2 = (n_{i_k} n_{i_{k+1}}, p_2)$ is the length of the shortest path between the two positions L_1 and L_2, formally defined as follows:

$$\begin{aligned} dist(L_1, L_2) = & \, length(n_{i_0} n_{i_1}) - p_1 + p_2 \\ & + \min_{\{i_1, i_2, \ldots, i_k\}} \sum_{\alpha=1}^{k-1} length(n_{i_\alpha} n_{i_{\alpha+1}}). \end{aligned}$$

2.2.2 Design Guidelines

A number of positioning systems are made publicly available for tracking the location update of mobile objects moving on the road network, such as Google's Latitude and Skyhook wireless WiFi positioning system [42]. Frequent location updates enable the location server to keep track of mobile clients' current locations and ensure the accuracy of the location query results. The algorithm that mobile clients employ to determine when and where to update their locations is often referred to as the location update strategy. We below describe the motivation, the advantages, and the challenges of our query-aware location update framework by analyzing and comparing a number of representative location update strategies.

Periodic update strategy. A *periodic update strategy* is the simplest time-based location update strategy, in which the location server maintains the location update for each mobile client at a fixed time interval. This update strategy implies that mobile clients are treated as stationary between updates.

Point-based update strategy. This approach uses the distance-based scheme and the server only record an update when the mobile client travels more than a delta threshold away in distance from the location of last update. The number of location updates per unit time will depend upon the speed of the mobile user.

Vector-based update strategy. A *vector based update strategy* uses the velocity vector of the mobile client to make a simple prediction about its location. An update is only sent when the current location of the mobile client deviates from its predicted location by an amount that is larger than a system-defined delta distance threshold. This strategy treats the velocity vector of the client as constant between updates.

Segment based update strategy. A *segment based update strategy* utilizes the underlying road network to limit the number of updates. Mobile clients are assumed to move at a constant speed on their current road segment. An update is sent when the distance between the current and the predicted location is larger than a system-defined delta threshold. We assume that mobile clients change their velocities at the end of each segment, i.e., the mobile client is assumed to have stopped at the segment end node and can change its movement speed and direction and move forward accordingly. Thus an update will be sent when the mobile client departs from a segment end node by delta distance. We refer the reader to [10] for more on these strategies.

Motivation of Our Approach.

We have discussed four representative location update strategies and each of them has some weakness in terms of both client energy-efficiency and network bandwidth or server load optimization. Furthermore they all suffer from the common inefficiency – the location update decision of mobile clients is independent of whether there are any location query requests nearby. It is obvious that when mobile clients travel in a region where there are no location queries, one can benefit by using a location update strategy that enable the location server to record their location updates at some critical location points, leading to significant saving in terms of client energy and bandwidth consumption as well as server load reduction. In ROADTRACK two criteria are used to determine what should be considered as critical location update points. First, we need to increase the location query awareness of mobile clients. By making mobile users aware of queries in their vicinity, one can avoid making those superfluous updates. Second, we need to maintain certain freshness of location updates for those mobile clients that are not in the vicinity of any location queries to maintain adequate location tracking capability of the system. The second criterion ensures that all mobile clients need to update their current location at the location server from time to time in order to keep their location record update to date at the location server, though different mobile clients may use different scale of location resolution.

Bearing these two design guidelines in mind, we develop a *query-aware, precinct based update strategy*. Concretely, we introduce the concept of encounter point and the concept of precinct as two building blocks. By keeping track of the encounter points for each mobile client moving on the road network, we are able to use the query awareness to differentiate the location update strategy used for mobile clients that are in the vicinity of active queries from the location update strategy used for the mobile clients that are not targets of any location queries. The use of precincts constrains the set of encounter points that a mobile client needs to keep track of to be small, and sets an upper bound on when the mobile clients have to update their locations regardless of whether there are location queries nearby. To further reduce the cost of checking whether a mobile is close to the border points of its current precinct or one of its encounter points, we develop a road network distance based check-free interval optimization, providing significant reduction in terms of the number of wakeups at the mobile client and the server update load.

The ROADTRACK query aware location update strategy is applicable to all moving objects in a road network setting, be it vehicles or pedestrians. This research is based on the assumption that all moving objects are either moving on the public road networks, or walk paths such as indoor buildings or university campus walk paths. As long as these walk paths can be modeled as graphs, our approach can be applied directly.

2.3 Precinct based update strategy

In this section we describe the basic design of our precinct and encounter point based location update method, and defer the check-free interval based optimization to the next section.

2.3.1 Precinct and Encounter Point

Precinct.

Precinct is introduced in ROADTRACK for dual purposes. First, every mobile object is associated with a precinct in which it currently resides. We use precinct as the spatial upper bound to enforce location updates of all mobiles when they cross their current precinct boundary and enter a new neighbor precinct. Second, we employ precinct to limit the scope of query awareness and balance the tradeoff between the level of location accuracy maintained at the server and the reduction of location update cost at the server. For example, queries about the restaurants in Miami are far away from the current location of a mobile client traveling in Atlanta downtown. Thus, the mobile clients in Atlanta downtown should not be made aware of queries about restaurants in Miami. By introducing system-defined *precincts*, we can conveniently limit the scope of query awareness for mobile clients residing within their precincts. This also ensures that the number of encounter points maintained at a mobile client is small.

A precinct $P = \{\mathcal{V}_P, \mathcal{E}_P\}$ is a subgraph of the road network $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}_P \subset \mathcal{V}$ and $\mathcal{E}_P \subset \mathcal{E}$. Nodes in $\mathcal{V}_P$ are either *internal* or *border* nodes. Each internal node is reachable from all other nodes of a precinct on a path composed of only internal nodes. All edges in $\mathcal{E}$ that are connected to an internal node in $\mathcal{V}_P$ are also in $\mathcal{E}_P$. The partitioning of the road network graph is created during the system initialization, and is stored together with the road network data maintained at both the server and the mobile clients. We present the precinct construction algorithms in the next section.

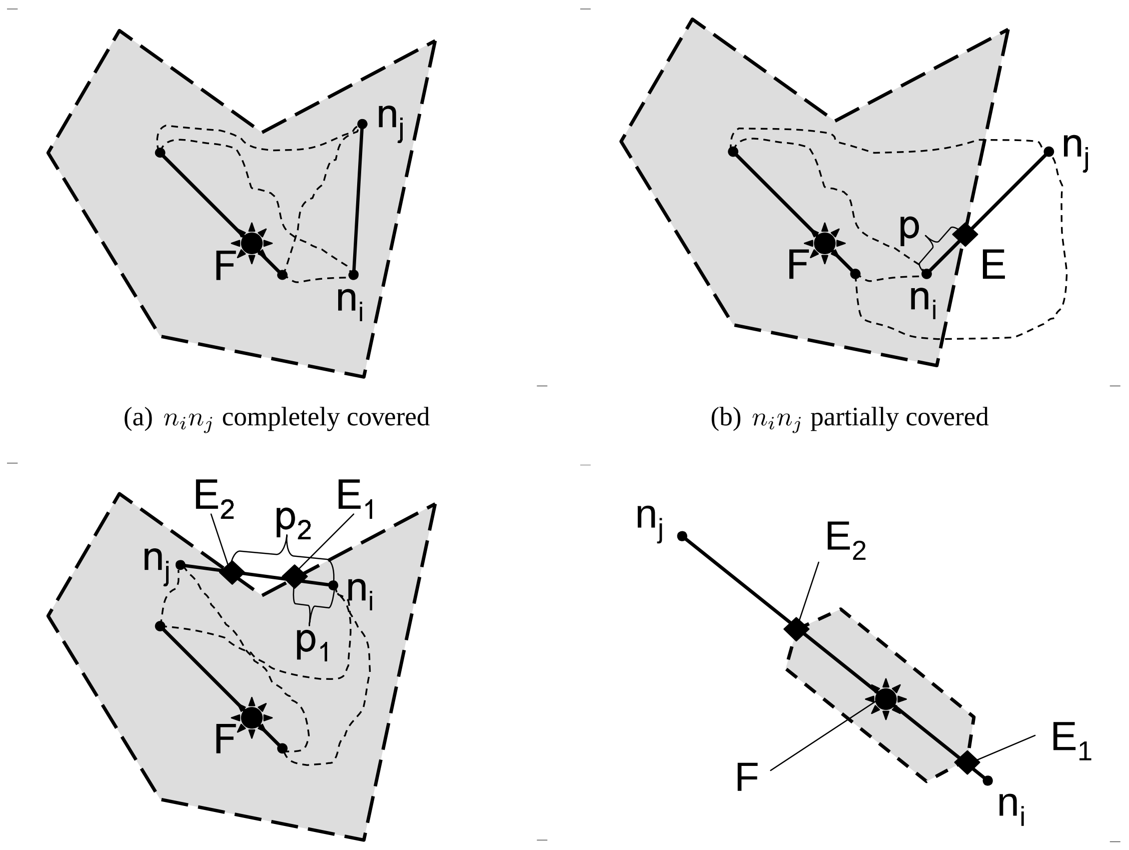
Encounter Points.

We first informally introduce the concept of encounter point. Let $P = \{\mathcal{V}_P, \mathcal{E}_P\}$ denote a precinct and $Q(R, F)$ denote an active location query, where R is the query radius in road network distance and F is the focal location of Q represented using the road network location defined in Section 2.2. The query Q is said to be relevant to the precinct P if a segment $n_i n_j \in \mathcal{E}_P$ is entirely included in the query region R as shown in Figure 2(a) or partially covered by the query region R. Assume that the shaded area in Figure 2 represents the query region computed in terms of road network distance from the focal location of the query, e.g., the query range of 2 miles from the focal location F. If a segment crosses the query boundary, i.e., one end-node is inside the query region and the other end-node is outside R, then we say that the segment is partially covered by the query. We call the road network location where a partially covered segment crosses the query boundary an *encounter point*. Figure 2(b) shows an example encounter point E. It is important to note that even if both end-nodes are inside the query region, the segment may only be partially covered, if there exists a network location L on the segment whose distance to F is greater than the query range specified, i.e., $\exists L | dist(F, L) > R$. In this case there are two encounter points for the query on a single segment (see Figure 2(c)). When the query range is small, it is possible that the query only covers a portion of the segment on which the query focal location F resides, thus there are two encounter points on a single segment but with both end-nodes outside the query region (Figure 2(d)).

Formally, given a set of location queries (Q) over the road network $G = (\mathcal{V}, \mathcal{E})$, one can determine the set of encounter points $\mathbb{E}_F = \{E_1, \ldots, E_n\}$, each of which (E_j) is associated with a range query $Q_i(R_i, F_i)$ with focal location F_i and range R_i, and is represented as a road network location that is exactly R_i distance from F_i. In other words, the set of encounter points $\mathbb{E}$ satisfies that $\forall E_i \in \mathbb{E}_F, \exists Q_i(R_i, F_i)$ such that $dist(F_i, E_i) = R_i$ and $\nexists L | dist(F_i, L) = R \wedge L \notin \mathbb{E}_F$, i.e., every encounter point is a road network location that is exactly range R_i distance from F_i. The encounter points are defined on the road network. When a mobile client meets or crosses an encounter point, it indicates that the client exits or enters the scope in which the query result is computed. Therefore, we use the encounter points as the critical location reference points for those mobile clients to update their locations at the server whenever they encounter these critical points on the move.

Comparison with existing update strategies.

In Figure 3(a) we show five mobile clients traveling on a portion of a road network, each following a distinct update strategy. The two precincts (west and east) have the common border points B_3, B_4, B_9, B_{10}, and connect to the rest of the road network at all the other



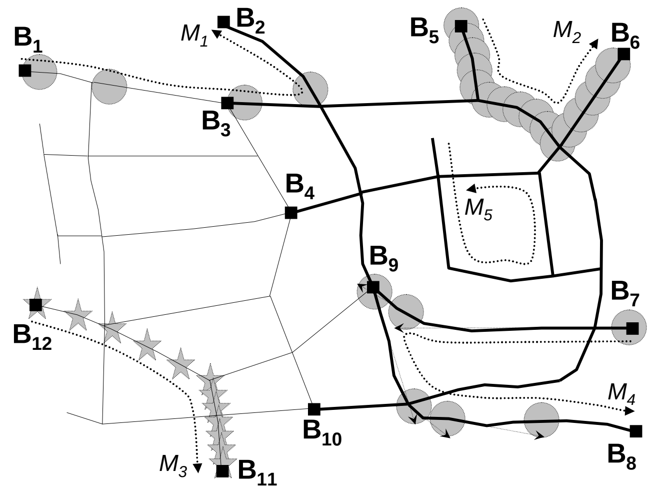
(a) n_in_j completely covered

(b) n_in_j partially covered

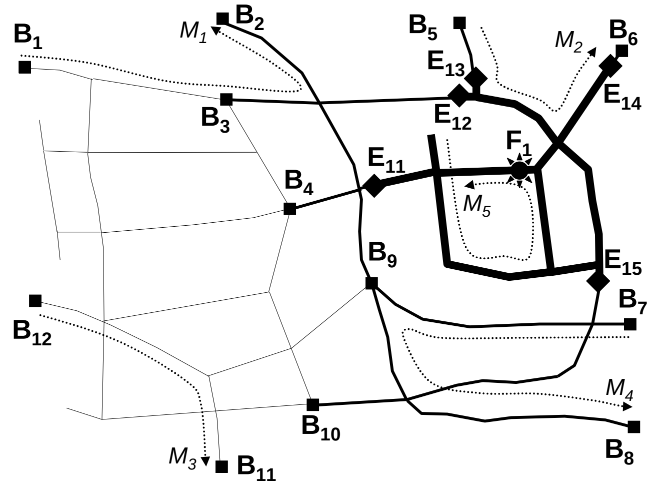
(c) n_jn_i partially covered near both ends (two encounter points)

(d) n_jn_i partially covered in the center (two encounter points)

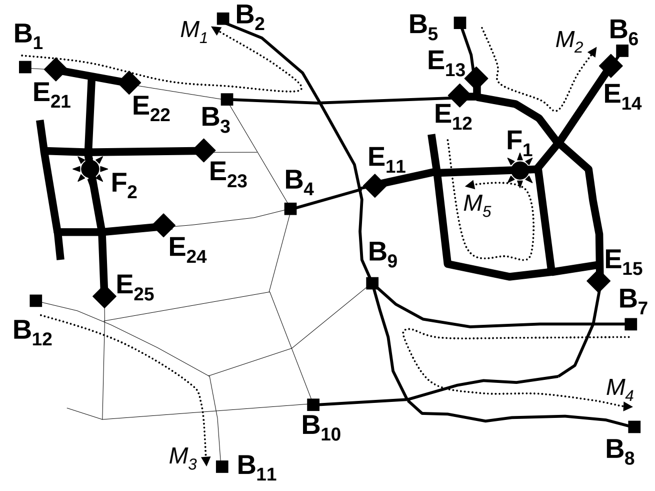
Figure 2: Four major cases for determining encounter points on the segment with end-nodes n_i and n_j. The shaded coverage area represents the query region of query $Q(R, F)$ computed R road network distance away from focal location F.



(a) Updates without query-awareness



(b) Single query (F_1)



(c) Two queries (F_1, F_2)

Figure 3: Example scenario with encounter points (E) and precinct border points (B) as update trigger points

border points (all border points shown as black squares). M_1 (upper left) is doing segment-based updates, triggering updates each time the client departs a segment end-node by $delta$ distance. The grey circles show the delta-radius circles around the mobile's location when the updates occur. M_2 (upper right) has a point-based update strategy, and thus sends an update whenever its current location is at least $delta$ distance from its last reported location. M_3 (lower left) is a periodic update mobile client, updating every t seconds. The mobile initially travels fast, continuing at a slow pace; as a result, updates may be spatially too sparse initially, and too dense when speeds are low. We show the locations at the time of updates as stars, since – unlike for M_1, M_2 and M_4 – there is no distance threshold for periodic updates. M_4 (lower right) has a vector-based update strategy, and consequently segment geometry along the trajectory is the primary determinant of update scheduling. However, all these mobiles' updates are wasted, as there are no outstanding queries on this portion of the road network. The fifth mobile client, M_5, following a RoadTrack update strategy, sends no updates, as there are no queries present, and its trajectory does not cross any precinct boundary points.

In Figure 3(b) a range query with focal location F_1 (sun symbol) is installed, with the associated encounter points $E_{11} \dots E_{15}$ (black rhombus symbol). Note that dead-ends are not E points inside a query coverage area (and not B points inside a precinct). We now ask all mobiles to follow a RoadTrack strategy: M_1 and M_3 cross and update on precinct boundary points only (B_1, B_2, B_3; and B_{12}, B_{11}). M_2 enters, then exits the query region, and thus also updates on encounter points (B_5, E_{13}, E_{14}, B_6). M_4 crosses boundary point B_9, but remains in the same precinct, and thus only updates on B_7, B_8. Note that B_9 is a real boundary point, as not all connected segments are in the same precinct, and thus a precinct crossing is possible; whether this occurs or not is not known in advance, so it is imperative for M_4 to consider B_9 as a potential update trigger. Finally, M_5 sends no updates, as it does not cross any B or E points. Note that being on the inside or outside of a query region makes little difference to mobile clients: after the initial query evaluation (during query insertion), neither client activity completely outside, nor completely inside the query coverage area changes the query result. Furthermore, as precincts are used to scope query awareness, mobiles in the west precinct (e.g. M_3) need not even consider the query's encounter points (which are all in the east precinct).

In Figure 3(c) an additional range query is added, in the west precinct. M_1 now also updates on this new query region's encounter points (E_{21}, E_{22}), but after entering the east precinct via B_3 it no longer needs to consider any points inside the east precinct.

2.3.2 Construction of Precincts

Clearly the entire road network is a legitimate precinct. Similarly, the other extreme is the single-segment precinct, where each segment of the road network is considered as one precinct. We can use road network distance or hop count to define the size of the preferred precincts. Assume that we use a system defined network distance threshold to partition the road network into precincts. The algorithm for constructing precincts is similar to a network expansion algorithm. A precinct is constructed by starting at the chosen segment and expanding along the neighboring segments and computing the network distance. This process repeats until the network distance threshold is reached. The construction process is repeated on the remaining segments until all segments in the road network are grouped into precinct-based partitions. A distance-metric based partitioning uses $dist(n_c, n_k) = dist(n_c, n_j) + length(n_j n_k)$ for distance expansion. The algorithm for constructing the precinct partition of a given road network proceeds in three steps. (1) The partition algorithm starts by marking all segments and all junctions as 'uncovered'. (2) A precinct center node n_c is selected at from an ordered queue of uncovered nodes (we elaborate on this ordering below). A queue is maintained during the precinct construction process, which contains a list of candidate nodes in ascending order of their distance from n_c. A node in the road network is a candidate node for the precinct centered at n_c if its distance to n_c is within the system supplied distance threshold. The queue initially contains only n_c. At each expansion step, the entry $(n_j, dist(n_c, n_j))$ at the head of the queue is removed, n_j is marked as 'internal', and all uncovered segments connected to n_j are added to the list of segments covered by the precinct. For segment $n_j n_k$, n_k is added to the list of nodes covered by the precinct, and this node's distance is calculated by $dist(n_c, n_k) = dist(n_c, n_j) + length(n_j n_k)$. If n_k is marked as 'border' (for some other precinct), then it is added to the list of nodes covered by the current precinct with a 'border' flag; otherwise, n_k is marked as 'internal' and $(n_k, dist(n_c, n_k))$ is added to the queue, unless a $(n_k, dist(n_c, n_k)')$ is already in the queue with $dist(n_c, n_k) \geq dist(n_c, n_k)'$. When the distance of the queue head node is larger than the specified precinct range, the precinct construction is concluded by marking all remaining nodes in the queue as 'border', and adding them to the list of nodes covered by the current precinct with the 'border' flag. (3) The algorithm continues with the creation of the next precinct until there are no uncovered nodes. When no uncovered nodes remain, there may still be uncovered segments, whose both end-nodes are border-points for other precincts. Single-segment precincts are constructed for each of these remaining uncovered segments.

An alternative approach to constructing precincts is to use the segment count (or hop count) metric, i.e. we use $dist(n_c, n_k) = dist(n_c, n_j) + 1$. Figure 4 shows a partitioning of an example graph with both methods. The randomly selected precinct center nodes are marked by n_1, n_2, n_3 in both cases and are selected in the order of node index. Border nodes are shown with a solid square. Single-segment precincts are highlighted with a grey background. Both hop-count based partitioning (left in Figure 4) and the distance based partitioning (right in Figure 4) shows five precincts: three precincts centered by n_1, n_2, n_3 respectively and two single-segment precincts.

As we mentioned, nodes are selected to serve as precinct centers according to a pre-specified ordering. The ordering method has no bearing on the correctness or utility of the precincts, but may have implications for both the number of client wakeups and the number of updates received by the server. As a result, we can use a random seeding of precincts as

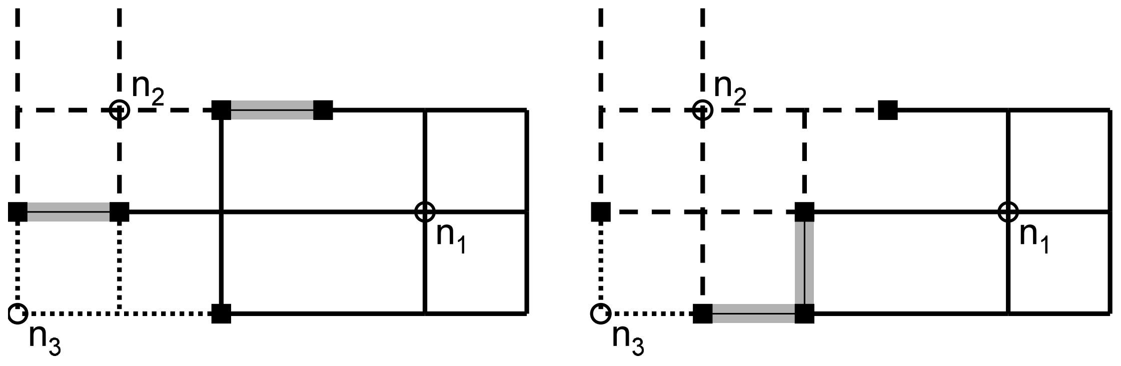


Figure 4: Graph partitioning with h=2 hop-based (left) and r=2 km distance-based (right) algorithm

our baseline scenario. Instead of such a naïve approach, a node ordering heuristic may be applied, whereby the algorithm prioritizes nodes that lie on many fast roads, as such nodes are likely to be important traffic junctions. This means that we score nodes by the sum of speed limits of their connecting segments, and always choose an uncovered node with the highest score as the next precinct center. In formulating this heuristic, our expectation is that if mobile clients take the shortest path to their destinations, high-speed roads and junctions will see more traffic than low-speed ones. Then, as we place junctions with high potential throughput in precinct centers, high-traffic portions of the road network are covered with relatively fewer precincts, and thus have the prospect of saving some border-point triggered updates and allowing longer check-free intervals between client wakeups.

Let deg denote the average degree of a node. With h-hop based partitioning, the average number of nodes in a precinct may be estimated as:

$$|\mathcal{V}_P|_{avg} \approx 1 + deg \cdot \sum_{i=0}^{h-1} (deg-1)^i,$$

and the average total length of the segments in a single precinct is calculated by

$$Len_P \approx \sum_{i=1}^{h} l \cdot deg^i = \frac{l(deg - deg^{h+1})}{1-deg}.$$

With d-distance based partitioning, we can substitute $h = \frac{d}{l}$ above, where l is the average length of a road network segment.

$|\mathcal{V}_P|_{avg}$ is independent of the size of the complete road network. For each precinct, distances between all nodes are pre-computed using the Floyd-Warshall algorithm and stored as a D distance matrix for this precinct. The complexity of this step for all precincts is $\frac{|\mathcal{V}|}{|\mathcal{V}_P|_{avg}} \cdot O(|\mathcal{V}_P|^3_{avg}) = O(|\mathcal{V}| \cdot |\mathcal{V}_P|^2_{avg}) = O(|\mathcal{V}|)$. Thus, given a road network $G = (\mathcal{V}, \mathcal{E})$ and its precinct partition $P = \{\mathcal{V}_P, \mathcal{E}_P\}$, the total storage space for the D distance matrices require $\frac{|\mathcal{V}|}{|\mathcal{V}_P|_{avg}} \cdot O(|\mathcal{V}_P|^2_{avg}) = O(|\mathcal{V}| \cdot |\mathcal{V}_P|_{avg}) = O(|\mathcal{V}|)$ storage space. The distance between an arbitrary location $L = (n_i n_j, p)$ and node n can be computed using the node-to-node distances from n to the two end-nodes (n_i and n_j) of the segment $n_i n_j$ that L lies on: $dist(L, n) = min(dist(n_i, n) + p, dist(n_j, n) + (length(n_i n_j) - p))$. The distance between any two locations $L_1 = (n_i n_j, p_1)$ and $L_2 = (n_k n_l, p_2)$ can be computed as the minimum of the lengths of four potential routes as follows (see Figure 5):

$$\begin{aligned}
&dist(L_1, L_2) = \\
&\quad = min(route_{ik}, route_{il}, route_{jk}, route_{jl}) \\
&\quad = min(\, dist(n_i, n_k) + p_1 + p_2, \\
&\qquad\quad dist(n_i, n_l) + p_1 + (length(n_k n_l) - p_2), \\
&\qquad\quad dist(n_j, n_k) + (length(n_i n_j) - p_1) + p_2, \\
&\qquad\quad dist(n_j, n_l) + (length(n_i n_j) - p_1) \\
&\qquad\qquad - (length(n_k n_l) - p_2)\,).
\end{aligned}$$

2.3.3 Data structures

In this section we give a brief overview of the data structures used at the server-side and the client-side to facilitate the understanding of our precinct based location update framework.

Server side data structures.

Node Table, NT = (<u>nid</u>, $\{sid\}$) stores road network nodes with the sid segment identifiers for the segments that connect to the node. A hash table index on the nid node identifiers allows constant speed lookup.

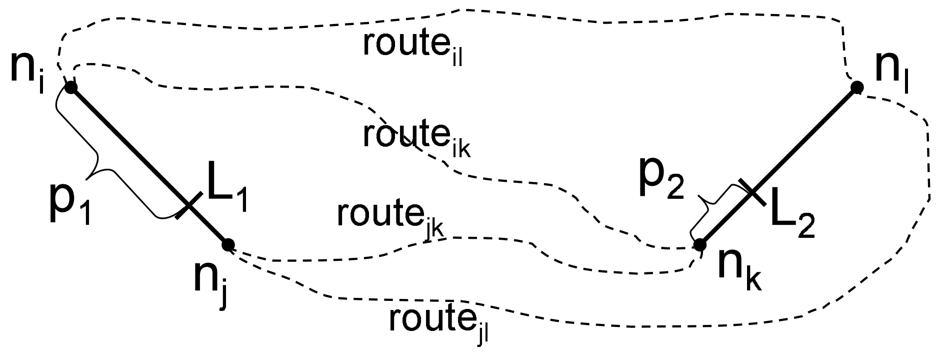


Figure 5: O(1) computation of distances between two arbitrary road network locations L_1 and L_2.

Segment Table, ST = (<u>sid</u>, nid_1, nid_2, pid, $\{oid\}$, $\{qid\}$) stores road network segments with the two end-nodes (nid_1 and nid_2). A hash table on the sid segment identifiers allows constant speed lookup. We store the identifier of the precinct covering the segment (pid), the client identifiers for clients on the segment ($\{oid\}$), and the list of query identifiers for queries (fully or partially) covering the segment ($\{qid\}$).

Precinct Table, PT = (<u>pid</u>, $\{sid\}$, $\{(nid, isBorder)\}$, D) stores information about a precinct with the identifier pid, along with the list of road network segments covered ($\{sid\}$), the list of nodes covered along with a flag showing whether the node is 'border' or 'internal' ($\{(nid, isBorder)\}$), and the pre-computed node-to-node distance table (D).

Query Table, QT = (<u>qid</u>, oid, range, F, $\{(sid, E, dir)\}$, $\{result\}$) stores queries in the system with the qid query identifier, the oid identifier of the client the query is attached to, the $range$ specifying the road network distance based range of the query, and F giving the focal location of the query. The $\{(sid, E, dir)\}$ list contains tuples of segment identifiers of segments at least partially covered by the query, encounter point locations for segments not fully covered (or $null$ for a completely covered segment), and a flag indicating which part of the segment is inside the query region (source-side or target-side). The $\{result\}$ list stores client identifiers for the clients that satisfy the query.

Client Table, CT = (<u>oid</u>, L, M) stores information about mobile clients in the system. The table is indexed on the client identifier attribute oid. L is the most recently updated road network location of the client, stored as a (sid, p) tuple, comprising of the *sid* segment identifier and the *p* progress. The M provides the client's mobility features required by the system, such as movement speed, trajectory, and so forth.

Client side data structures

NT, ST, and PT are also present on the client side as part of their map database.

Current Encounter points Table, CET = (sid, $\mathbb{E}$*, dir)* contains the encounter points found for all queries in the client's current precinct. Each mobile client only stores the encounter points for the precinct that includes the segment on which it is located. The CET is delivered to the client by the server when a client informs the server that she enters a new precinct. Also the CET at a client is incrementally updated by the server to reflect query insertions or deletions.

2.3.4 Computing with Encounter Points

Encounter points need to be computed whenever a new query is inserted into the system, or an existing query is terminated and removed from the system.

Computing encounter points for query insertion

A mobile user can issue a new location query Q by sending a message to the server in the form of $(oid, F, range)$. If the location of the mobile client with identifier *oid* in the CT table is older than F, its location information is updated with *F* and the new query is inserted into *QT* with a new unique query identifier *qid*.

The algorithm to calculate the encounter points and the set of segments covered by the query maintains a queue of $(nid, dist(F, nid))$ tuples, storing node distances from F in ascending order; and a hash-table (initially empty) for segments, where segment identifiers inserted into the hash-table indicate covered segments. The algorithm starts by investigating the distances of the two end-nodes of the segment n_in_j on which F is located, to detect any encounter points lying on this segment (Figure 2(d)). If $p > range$, then n_i is outside the query range, and an encounter point is at $E = (n_i n_j, p - range)$; otherwise n_i is inserted in the queue. If $length(n_i n_j) - p > range$, then n_j is outside the query range, and an encounter point is at $E = (n_i n_j, p + range)$; otherwise n_j is inserted in the queue.

Tuples are removed from the queue head, and all uncovered segments reachable from the current node n_i are investigated: the segment (of the form $n_i n_j$ or $n_j n_i$) is marked as covered by inserting its sid in the hash-table, and the distance of the segment's other end-node n_j is computed as $dist(F, n_j) = dist(F, n_i) + length(n_i n_j)$. If $dist(F, n_j) > range$, then the segment crosses the query boundary, and an encounter point is located at $E = (n_i n_j, length(n_i n_j) - (dist(F, n_j) - range))$ for $n_i n_j$ (Figure 2(b)), or at $E = (n_j n_i, dist(F, n_j) - range)$ for $n_j n_i$. Otherwise, the segment is entirely covered by the query region, and the tuple $(n_j, dist(F, n_j))$ is inserted into the queue, unless another $(n_j, dist(F, n_j)')$ is already in the queue with $dist(F, n_j) \geq dist(F, n_j)'$. The algorithm terminates when the queue is empty, with the list of encounter points, and the list of (completely or partially) covered segments $\mathcal{E}_q$ stored in the hash-table. Note that the case of two encounter points on a single segment (Figure 2(d)) is handled correctly by adding E_1 when the current node is n_i, and adding E_2 when the current node is n_j.

The segments in $\mathcal{E}_q$ are retrieved from the segment table ST, and the query identifier qid is appended to the list of queries covering the segment.

Using encounter points to answer a query

The set of completely or partially covered segments ($\mathcal{E}_q$) and encounter points of a query are computed using a network expansion algorithm when the server is notified of the query insertion. The initial result of the query is calculated by retrieving all segments of $\mathcal{E}_q$ from ST, then retrieving all oid clients that are listed on these segments. For segments with no encounter points, all mobile clients oid on the segment are added to the result set; otherwise mobile client locations are retrieved from CT to determine if they lie inside the query region. For a client that lies on a segment with a single encounter point, E's location must enclose the client's location, determined by the condition

$$\begin{aligned}&enclosing((E, dir), L) := \\ &\qquad (dir = source \wedge L_{oid} \leq E.p) \vee \\ &\qquad (dir = target \wedge L_{oid} \geq E.p),\end{aligned}$$

to be added to the result set. For a client that lies on a segment with two encounter points, we distinguish two cases: if the query coverage extends to an area around the end-nodes (Figure 2(c); $E_1.p < E_2.p \wedge dir_1 = target \wedge dir_2 = source$), then one of $enclosing((E_1, dir_1), L_{oid})$ or $enclosing((E_2, dir_2), L_{oid})$ must be true; if the query coverage area is the middle of the segment, with the end-nodes uncovered (Figure 2(d); $E_1.p > E_2.p \wedge dir_1 = source \wedge dir_2 = target$), then both $enclosing((E_1, dir_1), L_{oid})$ and $enclosing((E_2, dir_2), L_{oid})$ must be true.

Throughout the iteration over the segments of $\mathcal{E}_q$, a list of precincts that overlaps with the query range is built. The query will be installed on the clients residing within all these covered precincts. However, clients in different precincts will be aware of a different – precinct-specific – set of encounter points associated with this query. Also some covered precincts might be exempt from the need of being query-aware, such as those that do not contain any encounter points. Each precinct is retrieved from PT, and its segments are retrieved from ST. In the first iteration over segments in the precinct, a list of encounter points found in the current precinct are built ($\mathbb{E}_q^{pid}$). If $\mathbb{E}_q^{pid}$ is not empty, then in the second iteration clients on each segment in the current precinct are sent a query-installation message containing $\mathbb{E}_q^{pid}$. Clients residing in precincts that cover the boundaries of the query will be aware of the query. Clients in precincts further away will be unaware of the new query; and if the query range covers a sufficiently large area, some precincts entirely covered by the query (near the central area of the query area) will contain no encounter points, so clients in these precincts will also be unaware.

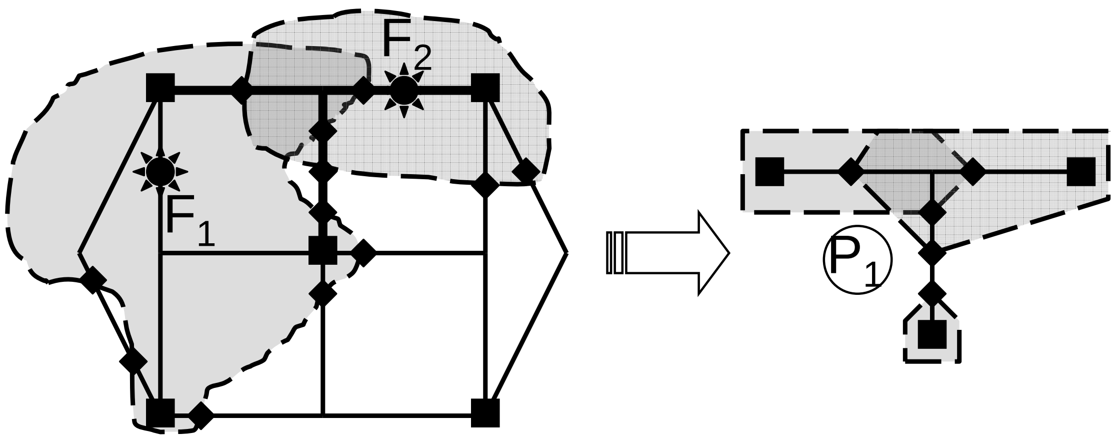


Figure 6: Two overlapping range queries with focal locations F_1 and F_2, and radiuses d_1=1.75 km and d_2=1 km (left), and precinct P_1 with queries displayed (right).

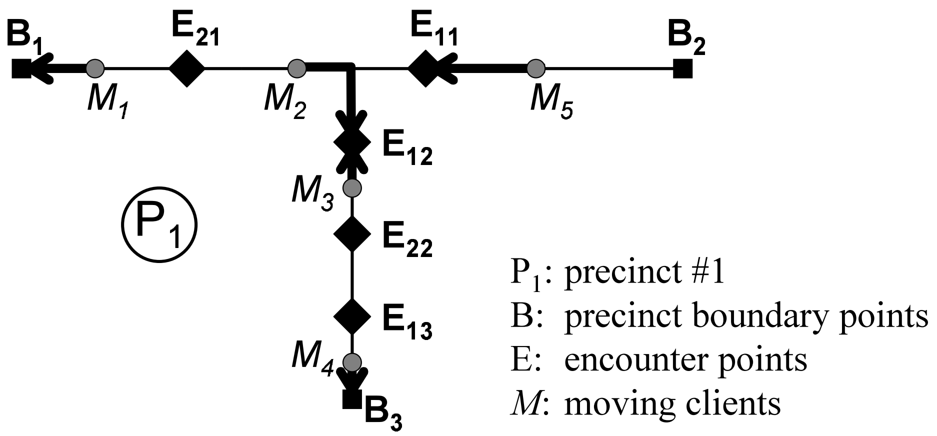


Figure 7: Check-free paths for mobiles $M_1, \ldots, M_5$, that are inside precinct P_1, when queries present are those shown on Figure 6.

2.4 *Optimization with encounter dependent check-free interval*

When a mobile client first becomes a registered user, it submits an orientation request to the server, including her current location. Mobile users registered with the system can be either active or disconnected. A mobile client is required to send a location update message to the server in three cases: (i) When a mobile user is becoming active from a disconnected state, she sends the location database server a location update message of type P. The server responds to a P message by sending the list of encounter points ($\mathbb{E}$) in the user's precinct. (ii) When a mobile user is crossing a precinct boundary ($\mathfrak{U}_B(oid, L)$), she sends the server a location update message of type B. The server responds to the B-message by sending the list of all current encounter points ($\mathbb{E}$) found inside the new precinct. (iii) When a mobile user is crossing an encounter point ($\mathfrak{U}_E(oid, L, E)$), the client sends to the server a location update message of type E. When the server receives an E-message, it updates the result set of the query attached to the E encounter point, either inserting (when entering a query region) or removing (when exiting a query region) oid, and notifying the issuer of the query corresponding to the encounter point of the change in the result set of the query.

A naïve approach to implementing the precinct-based location update scheduling is periodic checking of whether a mobile client has crossed a boundary point or encounter point and thus needs to send a location update to the server. Such decision is typically made based on the motion behavior of the client, the nearby queries and the corresponding encounter points, and the precinct boundary points. An obvious drawback of the periodic checking method is the unnecessary energy and resource consumption at each mobile client, especially when the mobile client is far away from any of the boundary points or encounter points for a given time period. We optimize the periodic checking method by introducing the road network based check-free interval mechanisms, which allows us to significantly enhance the performance of our precinct-based update scheduling algorithm.

Check-Free Road Network Locations

For each mobile user, we can compute a road network based check-free zone, based on its road network location L_c, its movement speed, its trajectory if available, and all the encounter points ($\mathbb{E}$) and boundary points ($\mathbb{B}$) of its current precinct. By check-free, we mean that as long as the mobile client travels within this portion of the network, no location update is necessary. One way to compute the check-free locations of a mobile client is to start from its current network location and perform the following three tasks. First, find the dominating encounter points and boundary points. Second, compute all the paths from the client's current location to every dominating encounter point or boundary point. We call these paths dominating check-free paths. Third, compute the region covered by the dominating check-free paths obtained in the previous step. Intuitively the dominating encounter or boundary points are those that are closer to the current network location of the mobile client. Given two encounter points E_1 and E_2, if the distance of E_1 to L_c is smaller than the distance of E_2 to L_c and the path from L_c to E_1 is covered by the path from L_c to E_2, then we say E_1 is dominating E_2 with respect to L_c.

Check-Free Interval

In order to detect when a mobile user on the move crosses an encounter point or a precinct boundary point, we need to determine when to perform the crossing check. To address the inefficiency of periodic checking for the mobile clients that are far away from any encounter point or boundary point, we introduce the check method based on a *check-free interval* computed for each mobile client. A check-free interval is the longest time that a client can sleep without comparing its location against any dominating boundary or encounter points, while being assured that any such update triggering points are not missed. The check-free interval can be computed as the shortest of the maximum-speed weighted distances (i.e., shortest travel time) to all B and E points within the current precinct. The maximum speed is a road segment specific constant (v_{max}^{seg}) stored with the road network data. The pre-calculated node-to-node distance table D is used for the fast calculation of the *check-free path* lengths (Figure 5). For a given road network location L_c, the check-free interval t_{cf} is computed as follows:

$$t_{cf} = \min_{L \in \mathbb{B} \cup \mathbb{E}} \frac{dist(L_c, L)}{v_{max}^{seg}}.$$

Consider the case of two overlapping queries on the road network with F_1 and F_2 as the focal location respectively as shown in Figure 6. For the purposes of the check-free interval computation, it is actually irrelevant to consider which parts of the segment are inside or outside one, two, or more query regions; only the locations of E and B points are important. The check-free paths for five mobiles (M_1, M_2, M_3, M_4, M_5) in this example are shown as darker line fragments in Figure 7.

Detection of crossing encounter or boundary points

We compute a check-free interval for every mobile client in the context of its current precinct using all the encounter points and boundary points. The mobile client does not need to perform any crossing check with respect to the encounter points and boundary points until its check-free interval is over. The mobile client may enter sleep mode if it does not have other active services. Upon the expiration of its check-free interval, the mobile client needs to determine whether it has crossed any E or B points. If the precinct (pid) of the segment at the last location is different than the precinct of the current location, then the client has crossed at least one B point, and thus a $\mathfrak{U}_B$ update is issued to the server, which in turn sends the encounter point set $\mathbb{E}$ of the new precinct to this client.

If no precinct change has occurred, then we perform the encounter point crossing detection. Given the last and current locations, there may be multiple paths between the two locations and each path may have a different set of E points. Given that the result of a query is independent of which concrete path the mobile has actually taken to move from the last location to the current location, any path between the last and the current location is suitable. We choose the shortest path to collect the E points located on this path. For any set $\mathbb{E}_q$ of encounter points associated with a query, crossing an even number of E points will leave the query result unchanged, since the mobile remains inside (or remains outside) the query range bounded by the two E points both before and after his movement. However, if there are an odd number of E points on this shortest path, this means that the overall movement of the mobile changes the query result. For all queries that have an associated E point on the shortest path, we determine the number of E points owned by that query. If any of these numbers is odd, then a query result has changed, and an $\mathfrak{U}_E$ update is issued.

2.5 Location query answering accuracy

The goal of any location update scheme is to enable the system to answer location-referencing queries. One implication of this goal is that an update scheme is feasible, which reduces the number of updates while still allowing the system to meet its goal of answering queries – by making clients query-aware. However, it is also possible to propose a very simple update scheme that greatly reduces the number of location updates: by having clients send their updates only every 20 minutes, for example. Such a system will still be able to answer location queries, but the quality of query results will be subpar: a driver moving at 60 kph might be 20 km from its last known location. While it is clear, for example, that a 1 minute periodic update scheme is more accurate than a 2 minute periodic update scheme, it is not readily apparent how either of these compare with eg. a precinct- and encounter-point based update scheme, which does not have an update period parameter. To assess the quality of various location query answering systems (including the location update scheme used in a system), we need a definition of *location query answering accuracy* that is universal, i.e. is a metric computable for and comparable across location query answering systems. In the following, we consider the major requirements for designing such a metric.

R1) The metric should be sensitive to the presence of queries. Consider for a moment a simple inaccuracy metric: Let the *mean location delta on update (MLDU)* measure the distance (in meters) between the reported location of a client at the time of update, and the location of the client as known by the server just before this update; and averaged over all updates of all clients within a specified time window:

$MLDU = \frac{\sum_t \sum_i |dist(L_i(t), \widehat{L_i(t)})|}{|U|}$.

As the goal of the location query answering system is the accurate answering of queries, a basic requirement for a metric is that when no queries are present, then query-answering accuracy should be indicated as ”perfect” (inaccuracy is 0). For example, the MLDU metric fails this test, because it doesn’t consider when the deviation matters (when a query is present), and when it is irrelevant (when there are no queries in an area, but clients there still send updates, as in the case of periodic updates). In other words, the query answering accuracy metric should not attempt to measure accuracy when there are no queries to be answered.

R2) The metric should be independent of the number of queries. If we did not demand this, then testing a system with 100 queries might have an inaccuracy 10 times as large as when testing with 10 queries. This requirement seemingly contradicts R1, but there is a clear domain for both requirements: While in R1 we suggest that we do not measure inaccuracy when and where there is nothing to measure (because there are no queries in an area), here we suggest that we do not measure inaccuracy excessively when and where there is something to measure (because there are many queries).

R3) The metric should be independent of the number of times a query (re-)evaluation is attempted. While an encounter-point based update scheme may come with incremental query evaluation on every update, a periodic update scheme may come with a single re-evaluation of all queries after all synchronized clients submitted their updates. Measuring inaccuracy only when (re-)evaluation takes place would mean that simply reducing periodic update frequency from once every 2 seconds to once every 4 seconds would reduce inaccuracy by half, when in fact inaccuracy has likely just doubled.

R4) The metric should be dependent on query result reporting. A location update changing a query result that is known by the server, but not reported to the query originator means that there is an inaccuracy in query answering. Together with 3), this means that we don’t care when or how (incrementally or not) a system processes location updates; we only care how fast location updates are propagated through to query originators.

R5) The metric should be independent of the length of the measurement time window. During a time window of 10 minutes, there are roughly twice as many opportunities to make some inaccuracies in answering queries, than in a time window of 5 minutes – this should not be reflected in our metric.

R6) The metric should be independent of the number of clients. A system that has twice as many clients also has roughly twice as many updates, and twice as many opportunities to mis-report client locations in query results – this should also not be reflected in our metric.

R7) The metric should aggregate inaccuracies in units of time (seconds). There are several ways in which ”inaccuracy” may be measured, aggregated for all queries and times, and related to the maximum possible inaccuracy, to arrive at a percentage. Consider the query Q issued by client q (the query originator). A client that is inside the query region Q, but is regarded as outside Q by q (i.e. a false negative), or a client that is outside the query region Q, but is regarded as inside Q by q (i.e. a false positive) are said to be *in fault with respect to Q*. While a client may be in fault with Q_1, it may simultaneously be not in fault with another Q_2. A *fault(Q, c, t_1, t_2)* is the movement of a client c from the moment it ought to change a query result of Q, to the moment it actually changes the query result Q. Every time a client enters or exits a query area, a fault happens (possibly for a very brief time): the query originator’s view of the moved client is momentarily out of sync with reality, and the query result is temporarily inaccurate. Let us briefly consider three possible fault measurement and aggregation modes:

- *Count*. Count the number of false negative and false positive query results, and relate to the total number of correct query results.

- *Distance (meters)*. Measure the distances traveled by clients a) after the client enters a query region Q, before they are recognized as query results (i.e. undetected incursions), and b) after exiting a query region, before they are recognized as no longer query results (i.e. undetected excursions).

- *Time (seconds)*. Measure the elapsed time between when a client enters/exits a query region and when the resulting query result change is recognized. This approach is similar to distance-based aggregation.

A fault typically starts with a client entering (exiting) a query region, and terminates when the query originator is informed of the entry (exit). A client may enter, then exit (or exit, then reenter) a query region without the result change ever being realized by the query originator – for example, because the incursion (excursion) happens entirely between two updates several minutes apart. Nonetheless, such a fly-through fault is a source of inaccuracy. In addition to faults in the vicinity of a query boundary, on the initiation of a new query, both false positives and false negatives may cause faults throughout the query region: For example, the locations of clients in a precinct with no queries (and thus infrequent updates) are poorly known until the clients send location updates in response to the encounter points of the newly installed query.

2.6 Experimental evaluation

In this section we present the experimental evaluation of our query-aware location update approach through four sets of experiments. We first compare our ROADTRACK location update approach with the four representative update strategies discussed in Section 2.2 in terms of number of updates per unit time at both server and client under two types of road networks: urban and rural. We show that the query-aware location update strategy significantly outperforms existing update strategies in terms of both client computation cost (#wakeups) and server updates for both urban and rural road networks. The second set of experiments measures the scalability of ROADTRACK by varying the number of mobile objects in the system. The third set of experiments examines the effect of different mobility models of mobile clients, different query characteristics, and the precinct size on the effectiveness of our query aware location update approach. Our experiments show that the query-aware update strategy offers consistent performance in terms of both server

Table 1: Road networks used in experiments

Style	County location	Total length	Segments	Junctions	Avg. segment length	Junction degree
urban	Miami-Dade, FL	15 650 km (315 h)	109 416	79 101	143.0 m (10.4 sec)	mean: 3.4, max: 8
rural	Coconino, AZ	36 212 km (733 h)	81 918	67 911	442.1 m (32.2 sec)	mean: 2.4, max: 6

update load and client wakeup load under different road network mobility models, different precinct sizes , different query loads, different query radius, and different query distribution models (uniform and hotspot). The last set of experiments examines the cost of precinct construction in terms of computation time, average number of nodes, number of precincts, size of precinct.

2.6.1 Experiment setup

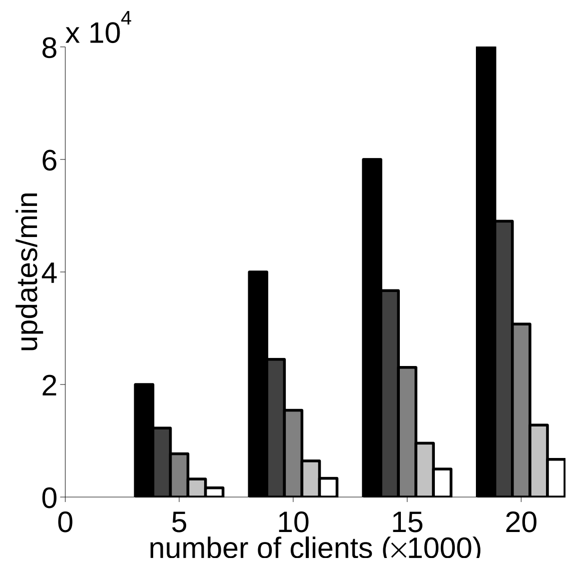
We use real road networks obtained from the US Census Bureau's TIGER/Line collection [43] in our experiments (Table 1). Maximum speeds are specified for each of four road classes at 30 mph for residential, 55 mph for highway, 70 mph for freeway, and 30 mph for freeway interchange (i.e. 48, 89, 113 and 48 km/h).

We created an event-based simulator for the evaluation of our framework. Instead of applying a timestepping approach, a central ordered event queue is used to schedule four types of events: change in the mobility pattern of an object (velocity vector change), query insertion, query deletion, and client wakeup. The single-threaded simulation consists of removing the events from the queue head, taking the assigned event action (eg. run client code on client wakeup, which might issue an update, which in turn causes the execution of server code), and inserting new events into the queue (such as the next requested wakeup with a future timestamp). The queue is initiated with the mobility pattern change and query insertion/deletion events, generated by a *mobility model* and *query model*, for the entire requested duration of the simulation. We consider two mobility models in this chapter: random waypoint movement on road network (RWR), and random trip model on road network (RTR). In both mobility models, each mobile object moves independently of others, with a speed that changes only when entering a new segment, and which is chosen according to the speed limit and speed distribution defined for the segment. In a RWR model, the client selects and travels a new segment at random at each junction; then repeats. In a RTR model, the client chooses a random trip destination on the map, travels the fastest route; then repeats. Client speeds are chosen from a bell-curve distribution (a Gaussian with a standard deviation of 0.2 times the mean) that is cropped above its mean (segment speed limit).

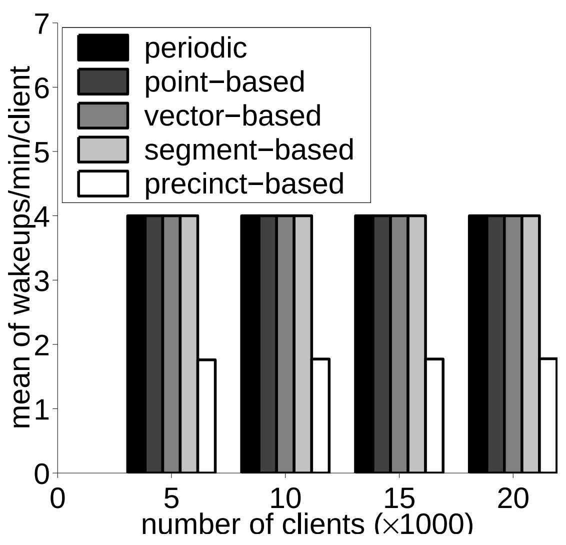
The query model we used maintains a 10% location query load in the system by default (i.e., the number of queries is one tenth of the number of mobile clients). Note that this is an aggressive query load, as it signifies that our system actively engages the attention of $\frac{1}{10}th$ of the population at any one time. Query ranges are chosen from a Gaussian distribution with a mean of 1 km, and standard deviation of 0.1 km. In order to simulate a more realistic scenario than that given by a uniform distribution of query centers, we create a query hotspot scenario, whereby queries are highly concentrated in some region of the map. The center of a hotspot is a road network location chosen from a random distribution over all road network locations in the network. Once the hotspot center is established, a weight is assigned to each road segment in the network. If the shortest road network distance between the hotspot center and the mid-point of a segment is d in kilometers, then we assign the weight $w_i = \alpha^d$, with $\alpha = 0.5$. Each segment then has a $\frac{w_i}{\sum_i w_i}$ chance of being selected as a query center location. Finally, the mobile object closest to the midpoint of the selected segment is chosen as the query's originator.

2.6.2 Messaging cost of update strategies

We compare the number of client wakeups and server update loads for various location update approaches by varying the number of users in the system. This set of experiments uses a client population with size ranging from 5000 to $20\,000$ clients. We compare the following strategies: (1) periodic updates every 15 seconds; (2) point-based tracking, (3)



(a) Server update load



(b) Client wakeup load

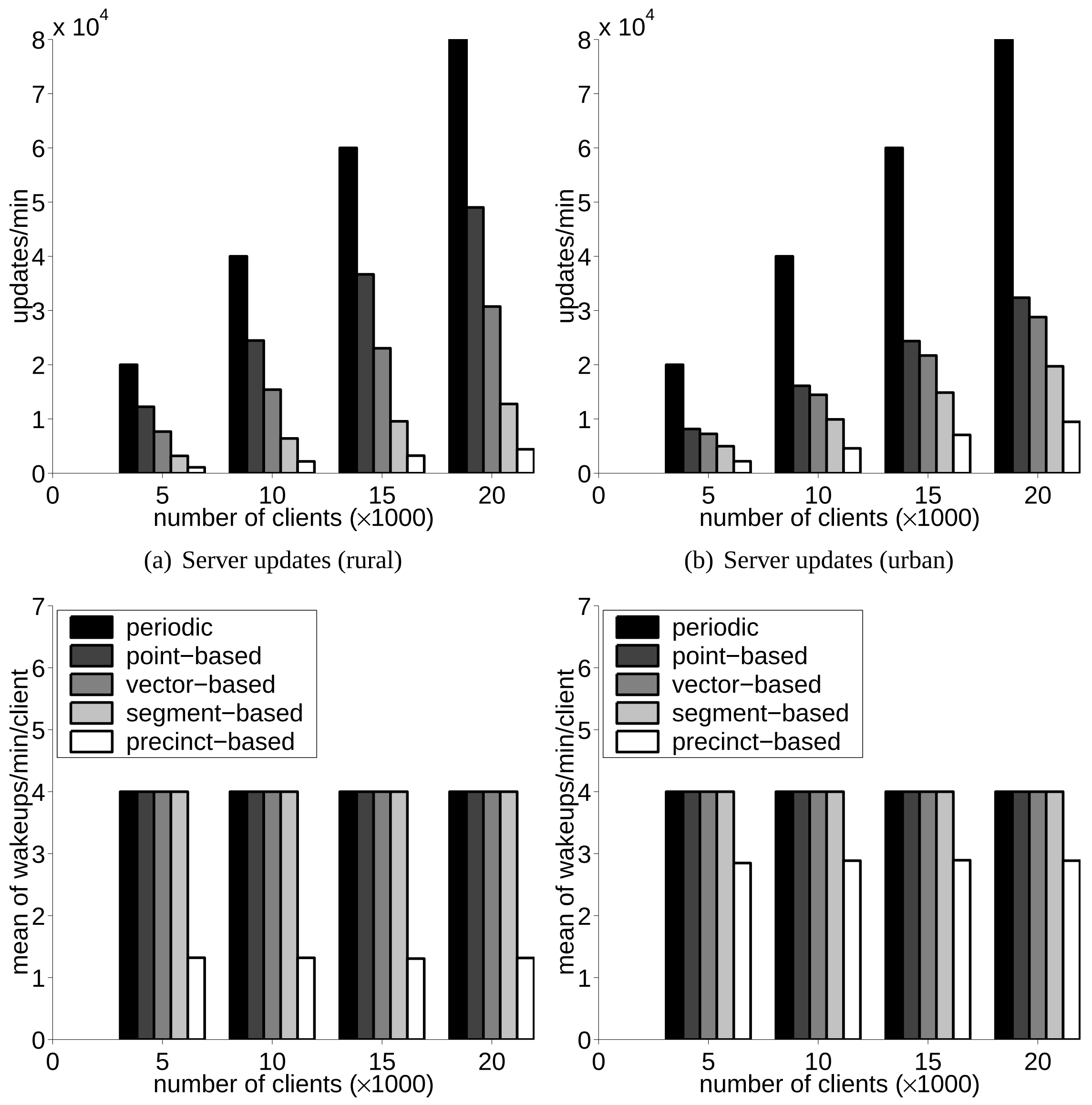
Figure 8: Scaling of update strategies with number of mobile clients (rural map, partition $r = 4\times$)

vector-based tracking, (4) segment-based tracking, and (5) encounter point-based wakeup and update strategy. For the first four query unaware approaches, the wakeup frequency and the reevaluation frequency at the server is set at 15 seconds, and the deviation threshold is set to 25 m. For the query-aware RoadTrack approach, we set a maximum wakeup frequency of once every 15 seconds (4/min), in order to allow performance of all methods at similar operating points with regards to accuracy. The comparison on the rural Coconino County map, with partition radii of 4 times the mean segment length (i.e., $r = 1768\ m$) shows that the encounter based method results in a significantly reduced rate of wakeups (Figure 8(b)). The advantage of RoadTrack is the that wakeups are unnecessary when a client is distant from encounter points (query boundaries) and precinct border nodes. Note that periodic, point-, vector-, and segment-based approaches all produce 4 wakeups per minute due to their 15 sec reevaluation setting – since a check-free interval type optimization is not available, they wastefully execute periodic self-checks.

The number of server side updates is shown in Figure 8(a). This experiment confirms the conceptual insight that the precinct-based RoadTrack approach outperforms all existing approaches even in the worst case. Note that since the query load is a constant 10%, the increase in the number of mobile clients also brings a proportional increase in the number of queries at the same time. As a result, not only does the number of mobiles per precinct increase, but the number of encounter points per precinct also increases. As each encounter point is an update trigger, the number of updates issued also necessarily increases. The RoadTrack strategy allows a reduction to 8%, 14%, 22%, and 52% of updates, relative to the other four comparison methods, respectively, even at the highest mobile load studied.

We further explore the scalability of our system by using precincts with radii that are 8 times the mean segment length (i.e., $r = 3536\ m$), and also running the simulations on the urban Miami-Dade County map (where $r = 1144\ m$). The larger precinct size provides a significant boost for RoadTrack: Wakeups are reduced on longer check-free intervals, as border points are – on average – further from mobile clients (Figure 9(c)). At the highest load setting, updates are reduced to 6%, 9%, 14%, and 34% of the query unaware approaches' updates (Figure 9(a)). The $\frac{1}{3}$ mean segment length of the urban, high density topology means that the distance-based partitioning creates segments that cover smaller areas, and thus the average distance from mobile clients to precinct boundaries increases. The high density also means that a query with the same radius (as measured on the roads) produces more encounter points. These factors bring an increase in the number of both wakeups and updates when compared with the low-density rural map, but the update count is still significantly lower in comparison with the four reference strategies (11%, 28%, 31%, and 46% of the updates of those methods, Figure 9(b) and 9(d)).

We also plot the effect of precinct seeding and the partitioning metric, when no queries are present, and thus all updates and wakeups are triggered by precinct boundaries only (Figure 10). We point out that the presence of precinct boundaries causes wakeups and updates even without the presence of queries. This property, by design, ensures that the server maintains the location tracking ability for all mobile clients, regardless of whether there are queries nearby or not. The benefit of an encounter-based strategy over strategies



(a) Server updates (rural) (b) Server updates (urban)

(c) Client computation cost (#wakeups, rural) (d) Client computation cost (#wakeups, urban)

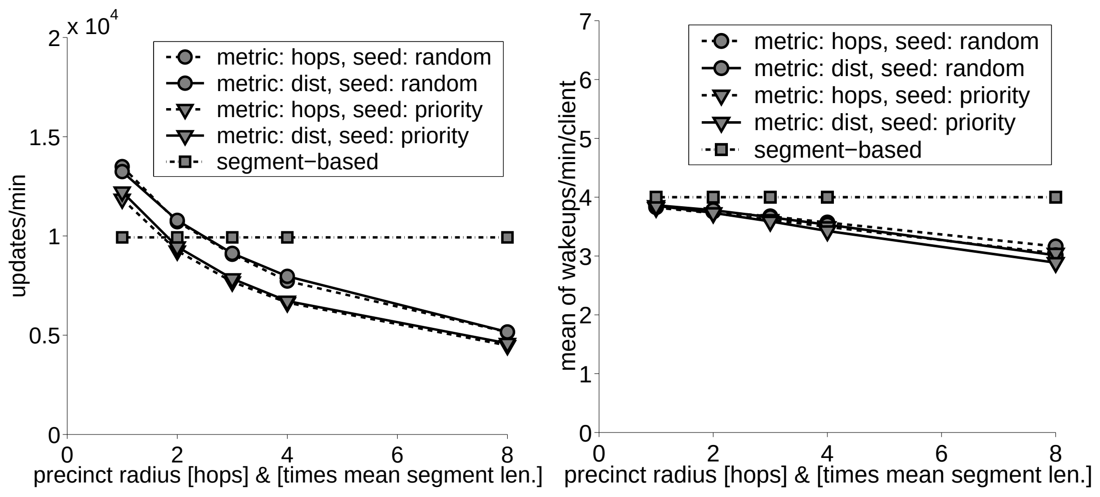
Figure 9: Comparison of rural and urban performance (with 8× distance-metric partitioning for the two maps in Table 1)

that are query-unaware (such as periodic, point-based, etc. methods) is the reduction of unnecessary wakeups and updates. On the other hand, if updates were only issued at query boundaries, in the case of very few queries in a region, a client could go for an extended period of time without an update, and the server would be unable to contact the client for a location update in order to answer a new query. The requirement to issue updates at precinct boundaries not only allows a client to be aware of query border points in its vicinity (after the server sends this information about the new precinct), but also allows the server to maintain an approximate location (bounded by the current precinct's boundary) of the client's whereabouts. These figures consistently show that larger partitions help reduce both updates and wakeups. We compare our precinct-based strategy with segment-based updates – at a radius of 1 hop, precinct-based updating is very similar to segment-based updates. As a result, the number of wakeups are only slightly improved from segment-based periodic wakeups when the precinct radius is small, but the improvement gap increases linearly with precinct size (Figure 10(b)). The number of updates is higher for low precinct sizes, as the $delta$ threshold used for segment-based updates (and the resulting inaccuracy) is not present in RoadTrack, but the update count drops to half of the segment-based updates at a radius of 8 times mean segment length (Figure 10(a)).

In the following we concentrate on the urban map, which is a more challenging terrain for RoadTrack due to the higher density network topology.

2.6.3 Effect of client behavior profile

We investigate our method with respect to its sensitivity to different characteristics of client behavior in two dimensions: mobility model and query radius distribution (Figure 11(c) and 12(c)). For mobility models, we consider the RWR and RTR type behaviors; for query size distributions, we vary the mean of the Gaussian distribution defining query radii chosen by clients, while keeping the standard deviation of the Gaussian at 10% of the mean. Our qualitative conclusion is that the RWR movement model is slightly more advantageous for



(a) Server updates (b) Client wakeups

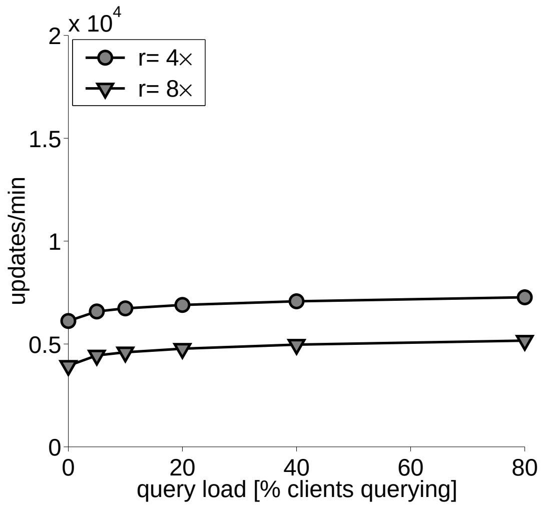
Figure 10: Comparison of precinct-triggered and segment-based strategies (urban map, precinct $r = 8\times$)

our approach, but this advantage decreases as query radii increase at client side. In all other experiments reported in this chapter, we employ the RTR movement model, to avoid any unfair comparisons.

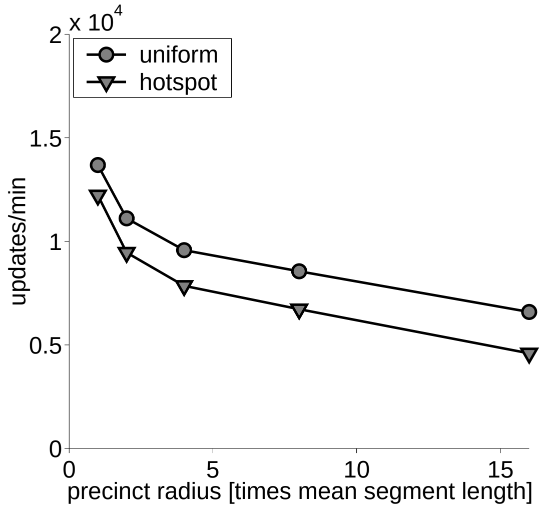
2.6.4 Effect of precinct size and query load

We investigate the effect of precinct size on our metrics with $10\,000$ clients, uniform and hotspot query distribution (Figure 11(b) and 12(b)). The simulations show that a hotspot distribution of query centers takes advantage of the features of our approach, producing fewer updates and wakeups.

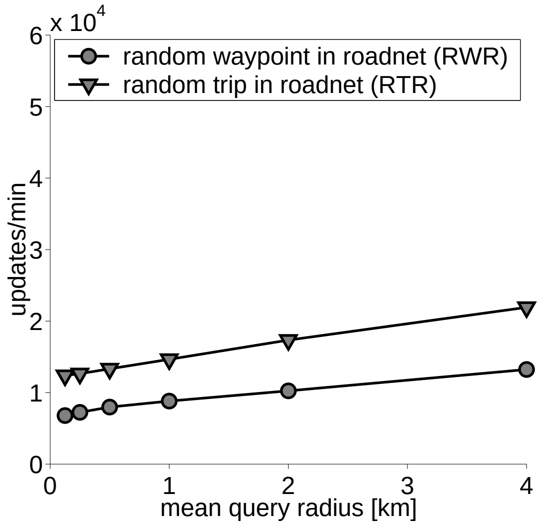
We inject a query load varying from 0% to 40% (i.e., 0 to 4000 queries), and run measurements using distance metric partitioning with the radius set to 4 and 8 times the mean segment length (i.e., $r = 572\ m$ for $4\times$, and $r = 1144\ m$ for $8\times$), with the results shown in Figure 11(a) and 12(a). The number of wakeups decreases with growing precinct size, as the influence of precinct boundaries on the check-free interval decreases. As many wakeups are false alarms (an update is not actually required), the number of wakeups is less impacted by an increase in query load, than the number of updates (Figure 11(a)).



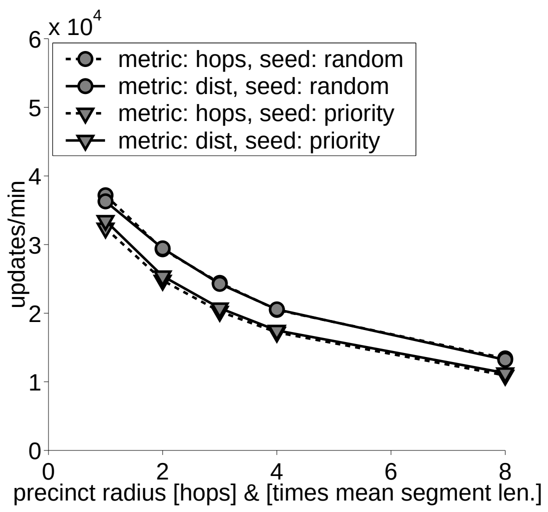
(a) Scaling with query load and precinct radius



(b) Effect of precinct size and query distribution

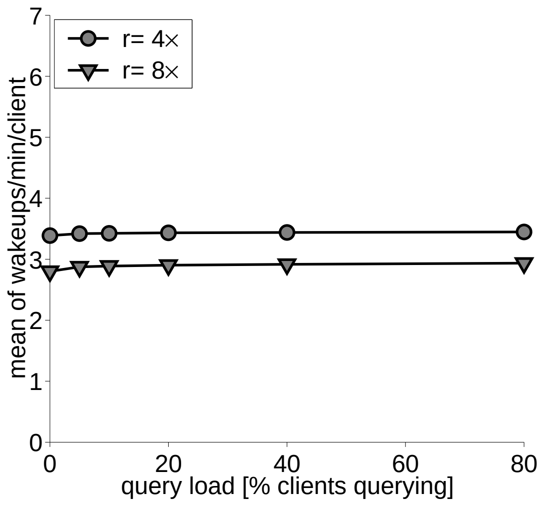


(c) Effect of client behavior characteristics

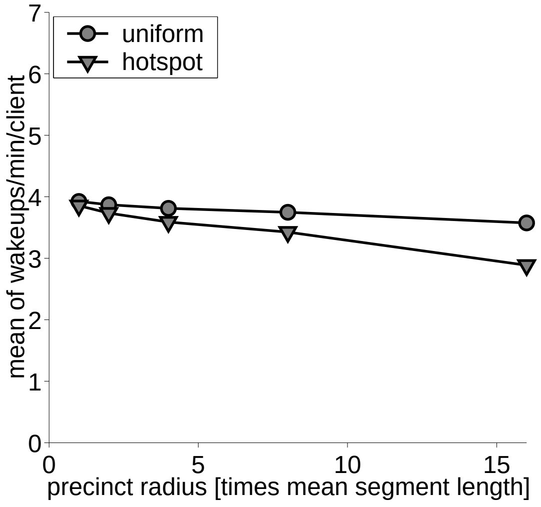


(d) Precinct-triggered updates

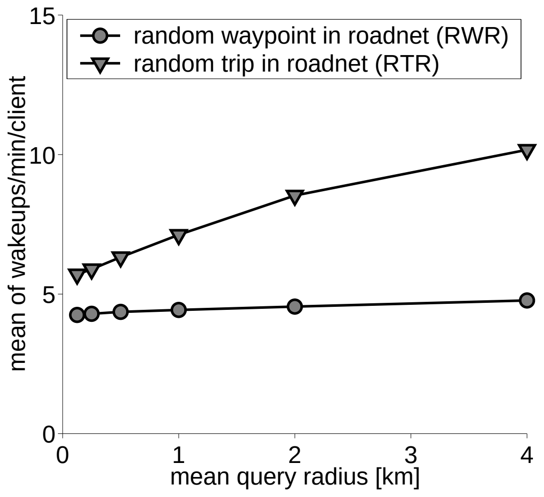
Figure 11: Server update load [fastest wakeup setting: $(a), (b)$: 15 sec, $(c), (d)$: 5 sec]



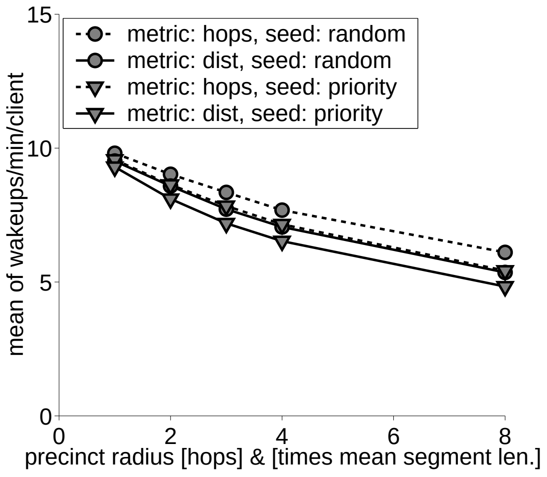
(a) Scaling with query load and precinct radius



(b) Effect of precinct size and query distribution



(c) Effect of client behavior characteristics



(d) Precinct-triggered wakeups

Figure 12: Client wakeup load (urban map) [fastest wakeup setting: $(a), (b)$: 15 sec, $(c), (d)$: 5 sec]

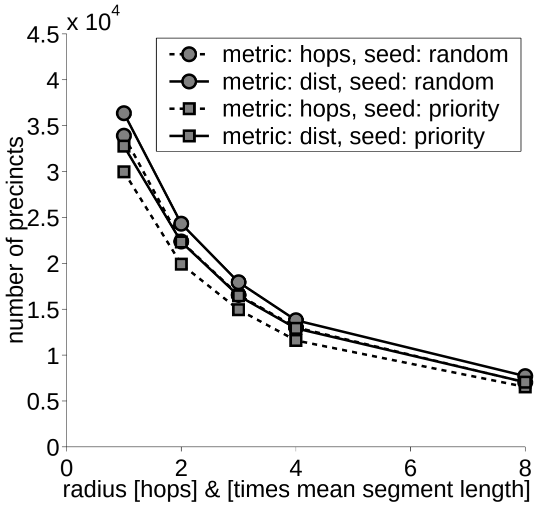
2.6.5 Precinct construction

The number of partitions created as a function of the requested precinct radius is shown in Figure 13(a). Distance-based partitioning is shown as a function of distance values that are multiples of the average segment length of 143 m, offering convenient comparison with hop-based partitioning, shown as a function of the hop count (e.g., partitioning with ”3 [hops]”, and ”3 [times mean segment length] ($= 3 \cdot 143\ m$)” settings are shown at the same X axis value). The average number of graph nodes per precinct grows only linearly with the precinct radius, largely due to the skew effect of more ”leftover” smaller precincts when the requested precinct size is large (Figure 13(c)). The storage space required for the pre-computed node-to-node distance matrices is defined by the number of node pairs per precinct. This storage requirement (Figure 13(d)), and the wall clock time required to compute it (Figure 13(b)) grow approximately as the square of precinct size. We remark that when precinct center nodes are selected using our heuristic-based seeding method, the number of precincts is reduced. In our experiments – unless noted otherwise – we thus used heuristic-based precinct seeding, and distance-metric partitioning with 8 times the mean segment length (i.e., $r = 1144\ m$).

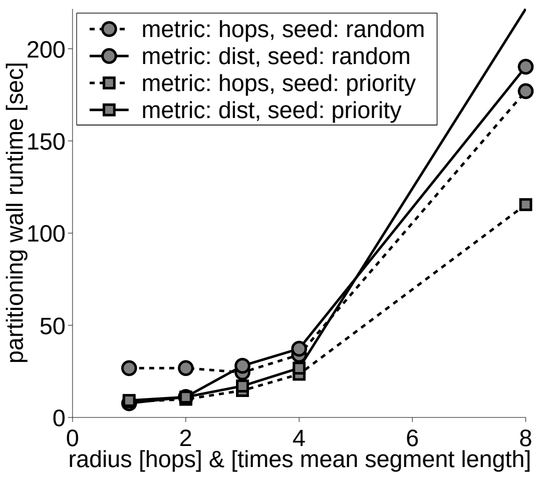
We also provide a comparison with partitioning the large rural map (Figure 14(a)–14(d)), and note that while the average number of nodes per precinct is almost independent of the partitioning method, the number of node pairs (and the resulting increased pre-computation time) increases markedly with distance-based partitioning for our rural map, due to the different topology.

2.7 Related work

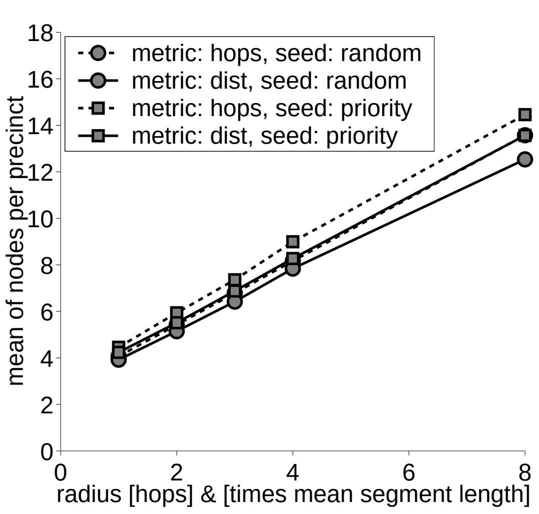
We review several threads of related work, which are most relevant to the location update efficiency, and which we present grouped in a number of themes. The aspects we consider are (i) whether the mobile clients are query-aware, (ii) whether the goal is to reduce the number of updates or something else, and (iii) whether the movements of mobile clients



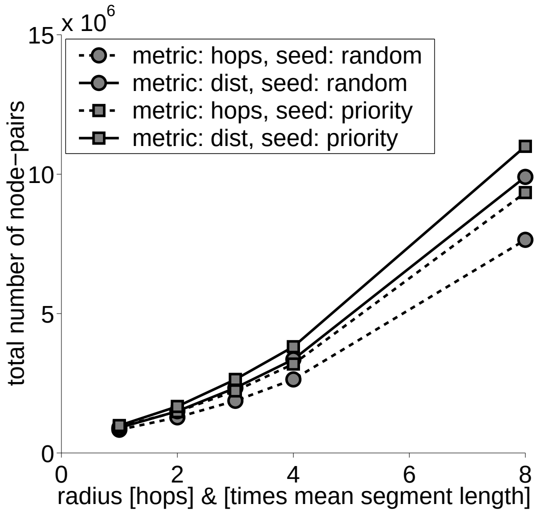
(a) Number of precincts



(b) Partitioning and pre-computation wall clock time

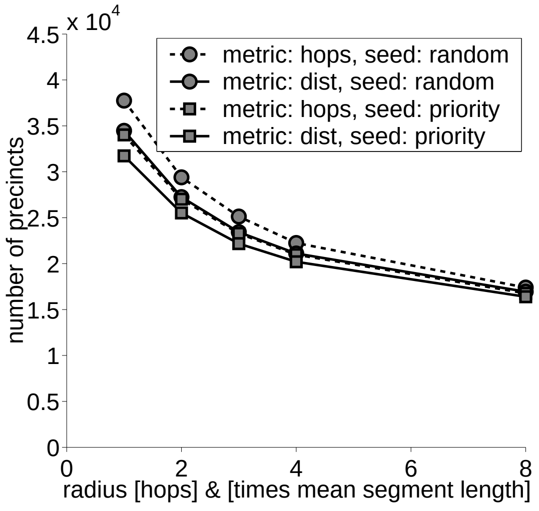


(c) Average number of nodes per precinct

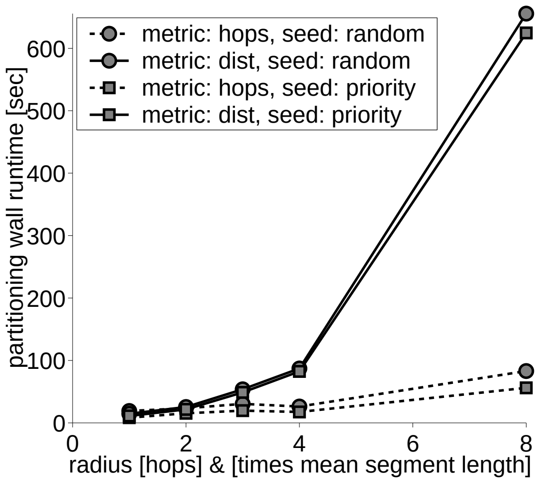


(d) Total number of node pairs

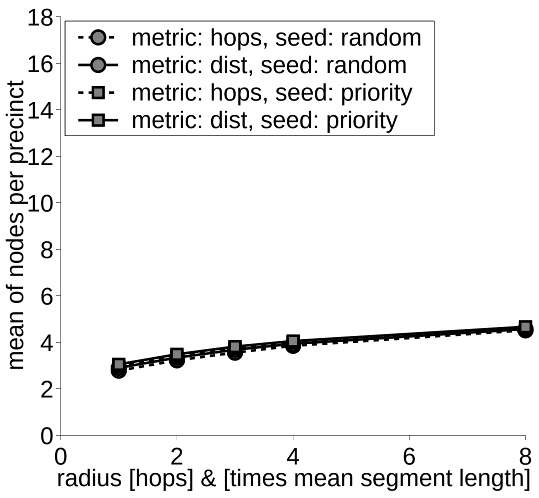
Figure 13: Effects of precinct radius on partitioning with hop and distance metrics (urban map)



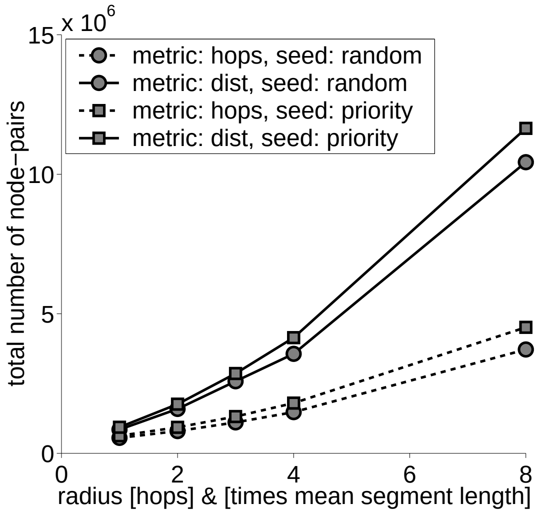
(a) Number of precincts



(b) Partitioning and pre-computation wall clock time



(c) Average number of nodes per precinct



(d) Total number of node pairs

Figure 14: Effects of precinct radius on partitioning with hop and distance metrics (rural map)

are on a road network. For our grouping, we do not consider (but mention) other important dimensions, such as (i) query type (eg. window, circular range or kNN type), (ii) query persistence (eg. snapshot or continuous), and (iii) the four combinations of query and client mobility assumptions (static or moving).

The first group of work explores the idea of reducing the number of location updates in presence of a road network, but without making mobile clients query aware. The clients run the prediction model and only issue an update when the prediction deviation from the actual location grows higher than a system defined threshold. For example, [10] compares a number of query unaware client location tracking approaches: point-based, vector-based, constant speed segment-based, and constant acceleration segment-based with acceleration profile which groups several basic segments together to improve basic segment-based tracking. As we show in this paper, the number of location updates can be significantly reduced when clients are query-aware, as there is no need for clients to issue updates in locales where outstanding queries are scarce. Even in the worst case, the precinct based approach outperforms the existing solutions.

The second group of work explores the idea of reducing query processing load at the server by making clients query-aware but in a world where constraints on client mobility do not exist (i.e., without a road network). For example, MobiEyes [15] uses the grid structure to define a monitoring region for each query, and only clients within the monitoring region need to be aware of the query. To make clients query-aware, the proposed techniques use some information derived from and aggregated across multiple queries, and not the full query information. This derived information is sent to the clients on a downlink connection, and then used to determine when or where the query result may have changed, prompting the posting of an update on the uplink connection to the server. [22, 8, 38] give a solution for static continuous queries over moving objects, by monitoring violations of safe region areas. The safe region of a mobile client is a rectangular area where answers to all queries remain the same. The work in [48] introduced the concept of *validity period*, which is a time period during which the query answer remains the same, based on a maximum query speed, though the solution is developed for 1NN moving queries over static objects. [47] introduced the concept of *validity region*, which is an area around a moving query's focal location where the query answer is the same. The techniques are for the server-side computation of validity regions for moving kNN and window queries over static objects. The validity region is represented as a convex polygon (and as a circle in certain cases). The solution applies to snapshot queries, as opposed to continuous queries, as it is not concerned with keeping results up-to-date. The authors in [29] describe a solution for moving continuous queries over moving objects, by monitoring violations of distance thresholds. Thresholds are created to signal a potential change of query results, and clients are aware of each query's focal location and one or two thresholds. kNN queries are discussed in particular, but the solution could be adapted to other types of queries. One common limitation for all the work in this group is the lack of consideration of road network constraints in their mobility model.

As we show in this chapter, when road constraints on queries exist (such as a distance or travel time range measured in the network), the solution must address the jump in complexity: we use encounter points to implement the query awareness and identify the critical points on road network segments where location update should be performed. In addition, we use precincts to impose locality on query-awareness and to set the upper bound for mobiles to update their locations.

The third group of work explores the idea of reducing server load for answering queries in the presence of periodic updates, without making clients query-aware or considering the mobility constraints of clients. An example is [46], which compares grid index based indexing of objects and kNN queries.

The fourth group of work explores the idea of reducing server load for query processing in the presence of a road network. The incorporation of road networks in server optimization of mobile queries started to gain attraction by [35, 24]. A most influential line of work in this group is the idea of speeding query answering by pre-computing distances after partitioning the road network graph [24]. However, no consideration is given to improve the server load for query processing by utilizing a road network based, query-aware location update scheme. We believe that the RoadTrack development can be beneficial for further reduction of server load for processing location queries on road networks.

[35] suggests Euclidean restriction and a network expansion approach to prune the search space for snapshot kNN and range queries over static objects. Static objects serve as generator points for a disjoint set of Voronoi-polygons covering the road network, such that a polygon's generator point is the closest generator point to all road network locations inside a polygon. Road network distances are pre-computed between all *border points* inside a Voronoi-polygon, which are then used to find the k polygons with the nearest neighbors. [9] uses pre-computed NN lists to speed up continuous moving kNN queries over static objects. [20] proposes storing a *distance signature* at nodes (containing approximate distance information for a limited number of relevant objects) to speed up snapshot kNN and range location queries over static objects.

[21] proposes a network reduction technique which simplifies the road graph, while preserving network distances, in order to answer snapshot kNN queries over static objects. [30] proposes a solution for moving continuous kNN queries over moving objects, using memory-resident server-side data structures. An edge table stores the list of queries affected by each edge (*influence list*), along with the *influencing interval* (the progress component of the road network location in our terminology) of the *mark* (query boundary locations) for each query. A query table stores the basic query information (focal location and k), the result set, and an expansion tree rooted at the focal location. The expansion tree is incrementally updated, which allows for an incremental query reevaluation.

Most of the existing approaches in this group fail to exploit the opportunities in making clients query aware in terms of server side performance optimization. Thus they fail to scale the location update scheduling in the presence of a growing number of clients and when hotspots query workloads exist.

Even though location queries have to be at least partially reevaluated on location update, efficient location query evaluation is traditionally considered a separate problem from efficient location updates; our work is in the latter category. Although not our focus in this chapter, when objects being queried on have their update count reduced, query evaluation costs are also reduced.

The fifth group of work explores the idea of reducing server load by making clients query-aware, but in a world where constraints on client mobility do not exist (i.e. without a road network). [38] gives a solution for static continuous window queries over moving objects, introducing the concept of *safe region*, which is a rectangular (or circular) area around a moving client's location, where all query answers remain unchanged. Safe regions are computed by the server, and sent to the clients on a downlink connection. The computational load of query answering is reduced by the combination of a frequently refreshed velocity-constrained index on mobile clients (for answering new queries), and an occasionally refreshed index on queries (for incrementally updating query results and computing safe regions). [7, 8] builds on the solution, proposing to reduce the safe region computation cost caused by large numbers of queries in the system. The universe is partitioned into disjoint rectangular *domains*, and each query's overlap with each domain is handled as a separate *monitoring region*. Mobile clients are only aware of monitoring regions inside their current *resident domain*, and send updates when crossing the boundary of a monitoring region or the resident domain. Domains are managed with a quad-tree-like hierarchy, allowing clients with a larger computational capacity to have a larger resident domain (with more monitoring regions), and thus need to issue fewer updates. [15] builds on these ideas to give a solution for moving continuous circular range queries over moving objects. The universe is covered with a uniform grid, and all grid cells at least partially covering a query region are part of the query's *monitoring region* (this is a different definition). Mobile clients are aware of queries' individual geometries, for those queries whose monitoring region overlaps the client's current grid cell, and only send updates when crossing the boundary of a query region or a grid cell. Clients compute a *safe period* using their maximum velocity assumption; during this time period the client's movement is ensured not to prompt an update, and thus the client is temporarily relieved from checking for the aforementioned update conditions. Velocity vector-based tracking is used for mobile queries: clients which carry a mobile query also send an update when their predicted and real locations are further than a given threshold; these updates are then used to readjust the set of grid cells that compose the query's monitoring region.

2.8 Conclusion

In recent years, some LBS providers have initiated a pay-as-you-go model for location tracking and location update services, with the primary objective of avoiding unexpected sudden load surges at location servers. For example, mobile users can pay a fixed price for being tracked or for keeping their location updated every five or 10 minutes. With the rapid escalation of location based applications and services and the growing demand of being informed at all times, the problem of scaling location updates and location tracking systems and services, if not addressed, will become a performance bottleneck for the success of the mobile commerce and mobile service industry. In this chapter we have presented ROADTRACK – a query-aware, precinct based location update framework for scaling location updates and location tracking services. ROADTRACK development makes three original contributions. First, we introduce encounter points as a fundamental query awareness mechanism enable us to control and differentiate location update strategies for mobile clients in the vicinity of active location queries. Second, we employ system-defined precincts to manage the desired spatial resolution of location updates for all mobile clients and to control the scope of query awareness capitalized by a location update strategy. Third but not the least, we develop a road network distance based check-free interval optimization, which further enhances the effectiveness of ROADTRACK and enables us to effectively manage location updates of mobile clients traveling on road networks by minimizing the unnecessary checks of whether they have crossed an encounter point or precinct boundary point. We evaluate the ROADTRACK location update approach using a real world road-network based mobility simulator. Our experimental results show that the ROADTRACK query aware, precinct-based location update strategy outperforms existing representative location update strategies in terms of both client computation efficiency and server update load.

CHAPTER III

DANDELION

When a driver asks for nearby gas stations within 10 minutes travel distance, to be answered continuously for the next 8 hours, while she is driving on her road trip from Atlanta to Florida, then she has issued a continuous spatial network range query. This driver is the query focal object and is a) interested in some locations that are within a specified travel time or road distance, measured along the segment edges of the road network graph, b) frequently in motion, and c) needs query results relative to her current location over some period of time, while the query focal location is constantly changing as the mobile user moves on the road.

The computational costs of answering such continuous network range queries are prohibitive, as a shortest path based network expansion needs to be run repeatedly at each and every location where the query is evaluated. We argue that continuous network range queries, whose focal locations are ”not far” from each other, have substantial overlap in their segment coverage. Such a large overlap may offer significant reuse opportunities for performance acceleration. We propose the Dandelion approach for fast re-evaluations of continuous network range queries with three original contributions. First, we propose the concept of Dandelion tree and associated techniques to accurately represent the coverage of a network range query with arbitrary range. Second, we design a suite of primitive operations to compute the coverage at a current focal location by reuse of the coverage at a previous query focal location. Third, we develop three Dandelion reuse algorithms, each powered with additional reuse abstraction techniques, to efficiently identify the portion of the Dandelion tree that can be used as the basis for reuse and further expansion. An extensive experimental evaluation on the Dandelion approach shows that the Dandelion reuse model and algorithms can significantly outperform the conventional shortest path network expansion model (NE) in terms of coverage computation cost for non-trivial radius size and high re-evaluation frequency.

3.1 Introduction

We are entering an era of ubiquitous connectivity and continuous services while moving on the road. One of the most frequently used continuous services by mobile users is continuous spatial range queries over nearby points of interests, such as gas stations, restaurants, and so forth, while moving on the road networks. ”Inform me of the gas stations within 10 miles of my current location in the next 3 hours of my travel” is one example of such road-network distance based continuous spatial range query, which runs continuously in the next 3 hours, returning the gas stations within 10 miles of my current location. The re-evaluation frequency of this query can be set by a mobile user explicitly or by a system-default interval, such as every 5 minutes.

Continuous spatial range queries over road networks have three unique features. First, a mobile user who initiates such a continuous query is interested in some road-network locations that they can travel to by following the spatial constraints of road networks. Thus, given a current location of a mobile user, the range query evaluation needs to use the road network distance measure, namely a shortest path distance over a road network graph, no matter if the road network distance is based on travel time or segment length. Second, the focal point of the query, which is the current location of the mobile user, is constantly changing. Thus, mobile users would like to continuously receive updated results of such spatial range queries as they move on the road over some period of time during their trips.

Determining the area covered by a Euclidean range query is straightforward: for a query with range r and focal location (x_0, y_0), all (x, y) locations that satisfy $(x - x_0)^2 + (y - y_0)^2 < r^2$ fall under the query area, i.e. locations that are within a geometric circle centered at the focal location. When a range query is based on the road network distance, such a neat geometric description of the covered locations is no longer possible. In order to find all the road network segments and thus locations covered by a given road-network distance range, the spatial network graph must be analyzed. The standard algorithm for determining graph vertices that are within the given range of a vertex is the network expansion (NE) algorithm, a variant of Dijkstra's algorithm [12] for finding shortest paths from a single source. A spatial network range query is a standing query, continuously running over the given period of time, during which the query focal location keeps moving, and continuous network range query evaluation needs to perform the network expansion (NE) from scratch as the mobile moves from one location to another. The cost of such computation is prohibitively expensive.

In this chapter we argue that when the continuous network range queries are evaluated frequently, and the current focal point is close to the previous focal query point, many shortest paths computed based on the previous focal query point can be reused to compute the shortest paths from the current focal point, as there is a huge overlap of the road segments in the search space of these consecutive evaluations of the same network range query.

Existing research efforts have focused on designing new indexing algorithms that are road-network aware in order to speed up such network range query computations. However, few efforts have been dedicated to exploring the reuse opportunities for speeding up the evaluation of continuous network range queries.

Applications of continuous spatial network range queries.

Many location based applications are based on continuous spatial network range queries, ranging from tourism assistant, mobile commerce, location-based advertisement, to location based social media. Concretely, a taxi driver in Manhattan might want to monitor for customers within 900 feet. It is known that a standard Manhattan block is a rectangle of approximately 264 by 900 feet. Thus, specifying ”1 block” for this query is not optimal, as it may evaluate the query using a road distance of $3.4\times$ larger on longer Street blocks as on shorter Avenue blocks. Additionally, there are many one-way travel restrictions in the road networks of New York City and a grid network topology, and thus a range limit given in Euclidean distance might often turn out to be much farther according to the odometer. Similarly, with an Euclidean range limit, customers in Brooklyn, Queens or New Jersey might also be returned as the matches of this range query, while reaching them from the current location of the taxi driver would require a substantial detour to cross a body of water on a crowded tunnel or bridge. Furthermore, during jam-packed rush hours, the taxi driver might want to find customers within 3 minutes (when taking traffic into account), instead of within a certain network distance. Taking all these considerations, the taxi driver will want to use a road network distance based range query with the option of switching between segment length based and travel time based network range measure. Alternatively, a UPS or FedEx express delivery driver might want to continuously monitor his 5-minute travel distance neighborhood as he moves on the road to find and service any new package pickup requests. This would enable UPS and FedEx to improve their service efficiency and reduce their cost of per-package pickup. Similarly, a tourist driver on vacation might want to keep an eye on major attractions within 50 miles of driving using her GPS device. An SUV driver might want to know at all times the gas stations that he can reach in 2 minutes over the next hour of driving. A businessman might want to receive notifications about the next contacted services that he can drive to within an hour. A college student might want to keep track of which of his buddies are nearby, given the campus walking paths and bike routes.

Scope and Contributions.

In this chapter we present the design and implementation of Dandelion, a smart reuse framework for efficient evaluation of continuous network range queries. First, we propose a smart reuse algorithm and related data structures. This development allows the reuse of both query results and shortest paths computed at the previous focal point of a spatial range query to compute the query coverage at the new focal point. As a result, continuous (moving) spatial network range queries can be evaluated with significantly reduced cost, compared to repeatedly running network expansion from scratch at each of the focal locations as a mobile user travels on the road. In summary, the design of our Dandelion approach is based on the fundamental observation that snapshot spatial network range queries, whose focal points are not far from each other, have substantial overlap in the set of segments that they cover, offering significant opportunities for reuse of a good number of shortest path computations. Our key contribution lies in devising techniques that can efficiently reuse previous shortest path computations, when the segment coverage from subsequent network range query evaluations have high overlap with the previous evaluation.

The dandelion framework consists of three original contributions. First, we propose the concept of Dandelion tree to accurately represent the coverage of a network range query with arbitrary range, by keeping track of three key network location points: border points (BOP), dead-end points (DEP), and zip points (ZIP). Second, we design three BOP-Push and three BOP-Pull primitive operations to compute the coverage at F by maximum reuse of the coverage at previous query focal location F. Third but not the least, we define the data structures and three Dandelion reuse algorithms to efficiently identify the portion of the Dandelion tree that can be used as the basis for reuse and further expansion. The basic Dandelion algorithm enables reuse by dividing the Dandelion tree (query coverage) of a query into the forward (FWD) and backward (BWD) halves, allowing separate maintenance of the key data structures for each half to reduce the search space. The Dandelion-T algorithm introduces and utilizes the Trident and Guide data structures to compose a more reuse-efficient Dandelion-T tree, leading to faster query re-evaluation than the Dandelion basic algorithm. Finally the Dandelion2 algorithm further enhances Dandelion-T in terms of query re-evaluation cost, by introducing two primitive transformation operations *move* and *jump*. This development can effectively transform one Dandelion tree to another with a minimum set of primitive transformation operations.

We conduct a series of extensive experiments and our results show that the Dandelion reuse model and algorithms can significantly outperform the conventional shortest path network expansion algorithm (NE) in terms of coverage computation cost for non-trivial radius size and high re-evaluation frequency.

Finally, we would like to note that the dandelion algorithm for the reuse of spatial network range query coverage is generic, and is not restricted to answering continuous queries.

The rest of the chapter is organized as follows: Section 3.2 presents an overview and notations for Dandelion reuse. Section 3.3 contains the formulation of our coverage reuse model. Section 3.4, Section 3.5 and Section 3.6 describe the construction of a Dandelion tree and re-evaluation of coverage at current focal F using our three successively more sophisticated data structures and algorithms. Section 3.7 reports the results of a series of experiments running on real road network maps. We outline the related work in Section 3.8 and concluding in Section 3.9.

3.2 Dandelion: Design Overview

3.2.1 Basic Concepts and Notations

Road Network. In Dandelion, a spatial network is defined as a directed graph $G = (V, E)$, in which V consists of the set of N vertices (also called road junctions), each of which is assigned a unique identifier v_i ($1 \leq i \leq N$); and E consisting of the set of N_E edges (also called road segments), each of which is assigned a unique identifier $e_{i,j}$ ($1 \leq i, j \leq N$), denoting a directed segment from v_i to v_j. Edges are undirected by default, but travel restrictions (such as one-way streets) may be imposed on queries. We do not distinguish multiple lanes of the same direction on a road segment. For each road segment, road-related information can be maintained, such as segment length (e.g. 0.7 miles), speed limit (e.g. 55 mph), restrictions (e.g. one-way road), etc. The length and speed limit of a road segment $e_{i,j}$ is denoted by $seglength(e_{i,j})$ in miles and $speedlimit(e_{i,j})$ in miles per hour respectively. Other road-related information such as current traffic data, if available, can be easily incorporated to provide more accurate travel time.

Segment Length based Shortest Path. Let v_1 and v_2 denote two road junction nodes and $v_1, v_2 \in V$. We define a path from a node junction v_1 to a node junction v_2 as a sequence of road segment edges, one connected to another, denoted as e_{1,i_1}, $e_{i_1,i_2}, \ldots, e_{i_{k-1},i_k}$, $e_{i_k,2}$ ($k > 0$). The length of a path h between v_1 and v_2 is computed as follows:

$$\begin{aligned} pathlength(h) = & seglength(e_{1,i_1}) + seglength(e_{i_k,2}) + \\ & \sum_{\alpha=1}^{k-1} seglength(e_{i_\alpha,i_{\alpha+1}}) \end{aligned}$$

For any two junction nodes v_1 and v_2, there may exist more than one way to travel from v_1 to v_2, we use $PathSet(v_1, v_2)$ to denote the set of all paths between v_1 and v_2. We define a segment length-based shortest path between v_1 and v_2, denoted by $sl\_shortestpath(v_1, v_2)$, as follows:

$$\begin{aligned} sl\_shortestpath(v_1, v_2) = \{h_{sl} | pathlength(h_{sl}) = \\ \min_{h \in PathSet(v_1,v_2)} pathlength(h)\} \end{aligned}$$

Travel Time based Shortest Path. The travel time of a road segment $e_{i,j}$ is $\frac{seglength(e_{i,j})}{speedlimit(e_{i,j})}$. Thus the travel time of a path h is calculated as follows:

$$\begin{aligned} pathtime(h) = & \frac{seglength(e_{1,i_1})}{speedlimit(e_{1,i_1})} + \\ & \frac{seglength(e_{i_k,2})}{speedlimit(e_{i_k,2})} + \sum_{\alpha=1}^{k-1} \frac{seglength(e_{i_\alpha,i_{\alpha+1}})}{speedlimit(e_{i_\alpha,i_{\alpha+1}})} \end{aligned}$$

The travel time-based shortest path between v_1 and v_2, denoted by $tt\_shortestpath(v_1, v_2)$, is defined as follows:

$$\begin{aligned} tt\_shortestpath(v_1, v_2) = \{h_{tt} | pathtime(h_{tt}) = \\ \min_{h \in PathSet(v_1,v_2)} pathtime(h)\} \end{aligned}$$

To differentiate the standard shortest path computed using Dijkstras network expansion algorithm, which computes shortest path between two vertices, we call the shortest path computed between any two network locations using the formula above *the extended shortest path*.

Network Location. A *network location* is defined in terms of segment ID and progress, denoted as $L = (e, p)$, on a segment e ($e \in E$). If e connects two vertices v_i and v_j and $i \neq j$, then $L = (e, p)$ with $p = 0$ denotes the network location at v_i, and $L = (e, p)$ with $p = length(e)$ denotes the network location at v_j. The *progress* p ($0 < p < length(e)$) denotes (determines) where the location $L = (e, p)$ lies on the $e_{i,j}$ segment between v_i to v_j.

Road Network Distance. The road network distance between two road network locations $L_1 = (e_{i_1,i_2}, p_1)$ and $L_2 = (e_{j_1,j_2}, p_2)$ is the length of the shortest path between L_1 and L_2 in terms of either segment length or travel time. The *segment length-based road network distance* and *travel time-based road network distance* are formally defined respectively as follows:

$$\begin{aligned} sldistance(L_1, L_2) = & seglength(e_{i_1,i_2}) - p_1 + p_2 + \\ & pathlength(sl\_shortestpath(v_{i_2}, v_{j_1})) \end{aligned}$$

$$\begin{aligned} ttdistance(L_1, L_2) = & \frac{seglength(e_{i_1,i_2}) - p_1}{speedlimit(e_{i_1,i_2})} + \\ & \frac{p_2}{speedlimit(e_{j_1,j_2})} + \\ & pathtime(tt\_shortestpath(v_{i_2}, v_{j_1})) \end{aligned}$$

Even though the segment length-based distance is the most commonly used distance measure on road networks, it may not provide sufficient and accurate distance information in terms of actual travel time from the current location (L_1) to the destination (L_2). For instance, highway road segments are much longer but also with much higher speed limits and thus may have relatively lower travel time compared to some local road segments. To ensure high accuracy and high performance of query processing, in DANDELION we use the travel time-based distance as the default shortest network distance between two network locations, denoted by $network\_dist(L_1, L_2)$.

Query Coverage and Border Points. A continuous network range query is defined by query identifier Q, query focal point F and query radius r, denoted as $Q(F, r)$. A *query focal point* F refers to the network location where the query Q is issued or evaluated. For presentation brevity, we refer to a query simply by its focal location when it causes no ambiguity in the rest of the chapter. We define the coverage of a query $Q_i(F, r)$ by the set of all network locations that satisfy the query range condition of $network\_dist(F, L) \leq r$, denoted as $coverage(Q_i(F, r))$.

$$\textit{coverage}(Q_i(F, r)) = \{L | network\_dist(F, L) \leq r\}$$

We also refer to network locations inside the coverage as *covered* locations.

The *segment coverage* of a query Q is the set of all segments that contain at least one network location that is covered by the query Q. Formally we have

$$\begin{aligned} \textit{seg\_coverage}(Q_i(F, r)) = \{e_{ij} | e_{ij} \in E, v_i, v_j \in V, \\ network\_dist(F, v_i) \leq r \vee \\ network\_dist(F, v_j) \leq r\} \end{aligned}$$

We say that a segment e_{ij} connecting v_i and v_j is fully covered by $Q_i(F, r)$ if both $network\_dist(F, v_i) \leq r$ and $network\_dist(F, v_j) \leq r$ hold. We say that a segment e_{ij} is partially covered by $Q_i(F, r)$ if either $network\_dist(F, v_i) > r$ or $network\_dist(F, v_j) > r$ holds. On a *fully covered segment*, all network locations on the segment satisfy the query range condition of $network\_dist(F, L) \leq r$, whereas on a *partially covered segment* some network locations on the segment satisfy the range query condition of $network\_dist(F, L) \leq r$, and others do not, namely $network\_dist(F, L) > r$.

To evaluate a query $Q(F, r)$, we need to start from the focal location F and expand segment by segment until the length of the path equals r. The ending network location of each such a path of length r is referred to as a border point (BOP) of $Q(F, r)$ such that $network\_dist(F, BOP) = r$. The set of border points of a network range query $Q_i(F, r)$, denoted as $BOP(Q_i(F, r))$, is the set of the network locations that have the network distance of r from F, and is formally defined as follows:

$$\textit{BOP}(Q_i(F, r)) = \{L | network\_dist(F, L) = r\}$$

Fully covered segments of a query Q contain no border points. Partially covered segments contain one or more *border-points* (BOP), each of which is a network location that demarcates a portion of the segment covered and not covered. A road segment can have 0, 1 or 2 BOPs associated with a query. Note that when a road segment has two BOPs associated with a query Q, the network distance based range of Q can be either smaller or longer than the segment length.

Segment to objects mapping. Queries are usually answered in terms of (static or moving) objects of interest, which satisfy a given query condition (such as being within a certain road network distance). When a mapping of segments to objects of interest is available, updating query results can be weaved into our proposed Dandelion algorithm. Objects on segments (or portions of segments) that are no longer covered/newly covered after the focal location of the query Q has moved, can be removed/added to the query result set incrementally.

3.2.2 Problem Statement and Design Objective

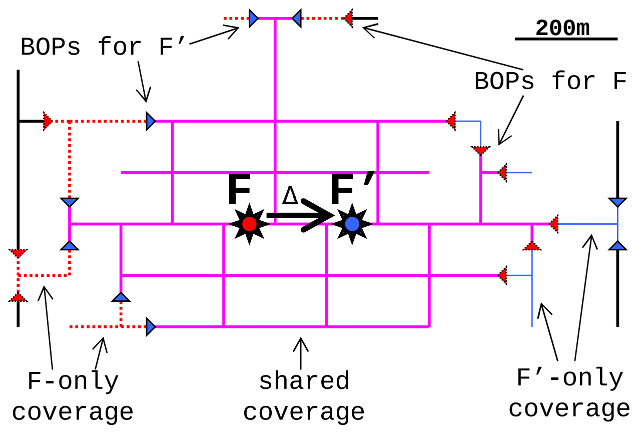
A continuous network range query, issued over a time period T by a mobile user A, denoted by $CQ(F, r, T)$, can be practically evaluated by processing a time series of snapshot (static) network range queries at successive query focal locations $F_0, F_1, F_2, \ldots, F_m$ ($m > 1$) as the mobile user moves on the road network. The set of focal locations ($F_0, F_1, F_2, \ldots, F_m$, $m > 1$) forms a time series of network location samples of the network paths traveled by this mobile user in the given query interval T, say 3 hours.

All the snapshot queries say $Q(F_i, r)$ ($i = 1, 2, \ldots, m$) are evaluated independently using a shortest path network expansion algorithm in two steps: segment coverage of the snapshot and the segment-to-object mapping.

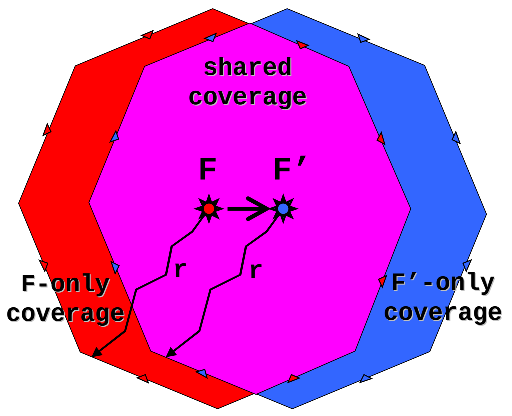
Concretely, let $Q(F_k, r)$ be a snapshot query to be evaluated. Assuming F_k is at a junction node v_i, the snapshot query evaluation will be performed in two phases: First, starting at v_i, for each one-hop path connecting v_i with v_j (j¿0), we run a test to compare the length of the path from v_i to v_j to the query radius r. If r is a time interval, say 5 minutes, we use travel distance of the path. If r is spatial distance range, say 2 miles, we use segment length of the path. If r is larger than the path considered, we extend the paths used for comparison; otherwise, we go to the next phase. At the end of the first phase, we have computed the segment coverage of the snapshot query $Q(F_k, r)$. In Phase 2, we obtain the objects-of-interest by performing a segment-to-object mapping using an inverted segment-object index. As the segment coverage of a query is the main bottleneck of a snapshot query evaluation, in the rest of the chapter we focus on fast computation of the coverage of a road-network range query (in terms of segments and partial segments.

In Dandelion, we argue that the problem of *efficiently evaluating a continuous network range query with moving focal point* should explore reuse opportunities between consecutive snapshot query evaluations for a number of reasons.

First, given a continuous network range query $CQ(F, r, T)$, the sequence of snapshot evaluations, $Q(F_0, r), Q(F_1, r), \ldots, Q(F_m, r)$ ($m \geq 1$), share the same query radius r and the same interest in the type of objects in the vicinity of travel paths of the same mobile user. Second, these snapshot queries differ only by their focal points of the queries. Thus, the segment coverage of a snapshot range query with focal location F_i often has significant overlap with the segment coverage of the subsequent query evaluation with focal location F_{i+1} ($0 \leq i \leq m$). Second, the snapshot coverage at F_i can be effectively reused to calculate the snapshot coverage at the successive location F_{i+1}. Third, if such reuse can be computed significantly faster than the time complexity of re-computing segment coverage of each snapshot evaluation by the Network Expansion (NE) algorithm from scratch, then



(a) Shared coverage and differences in BOPs between two queries.



(b) Highlight of shared and unique coverage areas of two queries. (All dist. on road – real coverage not octogonal.)

Figure 15: Observation I: Large overlap (pink) exists between the coverages of two nearby queries with focal locations F (red) and F' (blue), both with $r = 600\ m$ range.

this reuse capability can be the key to efficiently answering the continuous network range query Q_i.

3.3 Dandelion Reuse Model

We observe that the spatial network range queries with focal locations (say F and F) are ”not far” from each other, have substantial overlap in their segment coverage. Such a large overlap may offer significant reuse opportunities to accelerate the processing of continuous network range queries by minimizing the amount of duplicate shortest path computations. Bearing this observation in mind, we propose a Dandelion reuse framework and a suite of algorithms for fast re-evaluations of continuous network range queries.

Based on this problem formulation, we now introduce Theorem 1 and three lemmas which form the basic design of Dandelion.

Theorem 1: *Given two snapshot road network range queries, both with the same range, at nearby focal locations F and F', there exists an efficient way to transform the coverage at F into the coverage at F'.*

In the following, we describe three lemmas that lead to the proof of the above theorem.

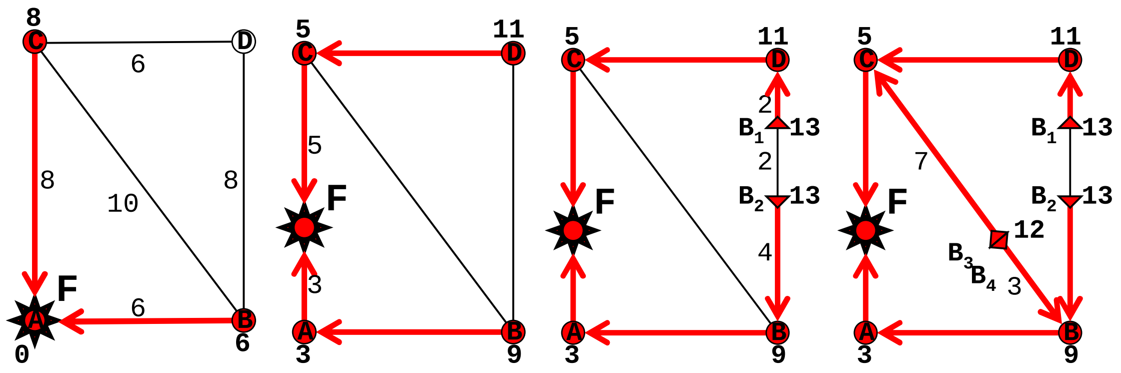
Lemma 1: *Let F and F denote two snapshot range queries with the same query radius r. If F and F are sufficiently close in terms of road network distance with respect to r, namely $network\_dist(F, F) \ll r$, we say that there is a substantial overlap between the coverage of the two nearby snapshot range queries and thus some portions of the computed coverage at focal location F may be reused efficiently to derive the coverage at focal location F'.*

Consider the two snapshot queries with query radius $r = 600\ m$ at F (red sun) and F' (blue sun) respectively in Figure 15(a), within $\Delta = network\_dist(F, F) = 200\ m$ of each other. The red dotted border-points (BOPs shown as triangles on segments) for the query at F are replaced by blue BOPs for F'; segments marked by red dotted lines are no longer covered at F', and segments marked by blue thin lines were not covered before at F. Despite these differences, many network locations (segments marked in pink thick lines) are shared by both queries F and F. We note that this large coverage overlap exists when the network distance $\Delta = network\_dist(F, F)$ between the two focal locations is relatively small compared to r (e.g., $\frac{1}{3}$ in Figure 2). Although we can expect that coverage reuse is most effective when $\Delta \ll r$, reuse should be beneficial as long as the condition of $network\_dist(F, F) < r$ holds.

The diagram in Figure 15(b) gives an intuitive illustration of the coverage arithmetic: By representing the coverage at F as *F-only* + *shared,* the coverage at F' as *shared* + *F'-only,* the dandelion approach is aimed at fast algorithms to compute the following transformation:

(coverage at F') = (coverage at F) - (F-only) + (F'-only).

From this example, we observe that a large number of segments are shared in both snapshots before and after an F displacement, especially when F and F are relatively close with respect to the query radius, and thus $F - only$ and $F' - only$ are relatively small compared to ($coverageatF$). The set of these common segments provide the potential of maximal reuse.



(a) Shortest path tree (SPT).
(b) SPT from non-vertex location.
(c) SPT to border-points.
(d) SPT to border- and zip-points (Dandelion).

Figure 16: Observation II: A shortest path tree to border- and zip-points (Dandelion tree) may be used to represent coverage of a query. ($r = 13$)

We now discuss how to efficiently compute the coverage of a continuous network range query, and especially how to compute the shared segment coverage between F and F, the F-only coverage and the F'-only coverage.

Recall the shortest path network expansion algorithm, when applying it to a network distance based range query, we will need to construct the shortest path tree anchored at the focal location of the query. By using the traditional shortest path NE, several problems may occur. First, focal locations of queries need to be anchored at a road junction. Second, partially covered segments will not be included in the SPT due to the fact that the network expansion condition is bounded to r. Third, no mechanisms to handle the situation where two border points on the same segment cross each other, which may consequently lead to errors in the evaluation.

Consider a simple snapshot range query with $r = 13$ on Figure 16, the traditional *shortest path tree* (SPT; Fig. 16(a)), calculated by NE, is only able to represent queries rooted at vertex locations. Parent pointers (shown as arrows) from each covered (red) vertex within the range r are used to navigate the tree. Thus, only two segments are included in the SPT: AB segment and AC segment.

In the case of a snapshot query with non-vertex focal location F, as shown in Fig. 16(b), we can add the expansion from non-vertex location F on segment AC to both end nodes of the segment. From A and B, we apply the standard shortest path network expansion. As shown in Fig. 16(b), three segments marked in thick red color are included in the SPT: AC, AB, CD; while the segments CB and BD are treated as not covered. Thus the objects of interest to this query that reside on these latter two missing segments are missed in the result of the query.

By extending the standard NE expansion to include the handling of non-vertex focal point and non-vertex border points, we can correctly compute the segments covered by a query either fully or partially.

Based on the extended shortest path and the border points, we below define the extended shortest path tree for a snapshot query.

Extended shortest path tree. Let $Q(F, r)$ denote a continuous range query with focal point F and query radius r. Let $BP(Q)$ denote the set of border points of Q such that $\forall B_i \in BP(Q)$: $network\_dist(F, BP_i) = r$. We refer to the tree with F as the root and all BP_i as leaf nodes, as the *extended shortest path tree* of Q. Note that all internal nodes of this tree are road junction nodes. The root and the leaf nodes may or may not be junction nodes. Each child node maintains a pointer to its parent node in this extended SPT. It can trace the shortest path of r length from all BOPs to F. Also the segments associated to the root F and border points of Q are recorded as a part of the root and leaf node, respectively. The tree traversal path from leaf node BP_i to root F represent the shortest path from query focal location F to network location BP_i.

Covered, Partially covered and Uncovered Segments. For a given query and its extended shortest path tree, all the internal nodes of this extended SPT are road junction nodes. For each pair of internal nodes v_i and v_j, if v_i is a parent node of v_j or vice versa, then we call the segment with v_i and v_j as the two end nodes a *covered* segment. Otherwise, if v_i and v_j are two end nodes of a segment in the road network, we call this segment *uncovered* by the query Q. We call all the segments that are attached to root or leaf nodes of the extended SPT the *partially covered* segments.

Given a continual range query and a sequence of its snapshot evaluations, say $F_0, F_1, F_2, \ldots, F_m$, we construct the initial extended SPT using network expansion at the initial installation time. The construction algorithm takes the query as the input and utilizes the network distance based shortest path formula to compute the internal nodes, starting from the focal location F of the query, until it reaches all the border points of the query using network expansion. For each subsequent snapshot query evaluation, we use Dandelion algorithms to construct the extended SPT for the next snapshot evaluation by maximizing the reuse potential of the previous extended SPT.

Lemma 2: *An extended shortest path tree is well suited to represent all the network locations covered by a query.*

This lemma states that every network location covered by a query Q is included in the extended shortest path tree of Q. More specifically, all road junction nodes covered by the query are also represented as the internal nodes in this extended SPT. Segments that are fully or partially covered by the query are represented in this extended SPT. By introducing border-points (BOPs) on partially covered segments, we are able to keep track of all network locations that are covered.

Concretely, given an extended shortest path tree rooted at the focal point F of a query, we refer to the internal node v_i who is the parent of some leaf nodes (i.e., border points of the query) as the last junction node of SPT and we have $network\_dist(F, v_i) \leq r$.

When the extended shortest path network expansion has reached the last junction node v_i and $network\_dist(F, v_i) + d_i = r$, where d_i is the final distance on the edge from v_i to v_j and $network\_dist(F, v_j) > r$, then a BOP is placed on the edge v_iv_j, at $r - d_i$ distance from v_i (i.e., at exactly r distance from F). Thus, the BOPs, just like covered junction vertices, keep a pointer to their parent vertex in this *extended shortest path tree to border*, ensuring that . all shortest paths of r length $-$ from all BOPs to F are included in this extended SPT.

Fig. 16(c) shows that the segment CD is partially covered by the query as it has two border points B_1 and B_2 residing on the segment CD. The shortest path from F via C to D ending at B_1 on the segment from D to B satisfies $network\_dist(F, D) \leq r$, $network\_dist(F, B_1) = r$ and $network\_dist(F, B) > r$. Similarly, the shortest path from F via B and ending at B_2 on the segment from B to D is represented by $network\_dist(F, B) \leq r$, $network\_dist(F, B_2) = r$ and $network\_dist(F, D) > r$. These two border points do not cross each other. We refer to them as non-crossing border points.

Fig. 16(d) shows the case in which two border points are crossing one another on the same segment CB. The shortest path from F expanded to C and ending at B_3 on the segment from C to B is represented by $network\_dist(F, C) \leq r$, $network_dis(F, B_3) < r$ and $network\_dist(F, B) > r$. Similarly, the shortest path from F expanded to B and ending at B_4 (crossing B_3) on the segment from B to C is represented by $network_dis(F, B_4) < r$. These two border points are crossed by each other. We refer to this case as crossing border points. To facilitate the query coverage reuse, in Dandelion, we introduce zip point to represent two crossing border points residing on a single segment.

Zip Points. Let v_i and v_j be the two end nodes of a segment in the road network graph. If v_i and v_j are the last junction nodes in the extended SPT (with final distances d_i via v_i and d_j via v_j), then the segment with v_i and v_j as the end nodes is either an uncovered or a partially covered segment by the extended SPT. Thus, there exists a network location Z on this segment that is equidistant from F via both vertices v_i and v_j. Let $a = network\_dist(Z, v_i)$ and $b = network\_dist(Z, v_j)$. We have $network\_dist(F, v_i)+a = network\_dist(F, v_j)+b$ and $a+b = seglen(v_i, v_j)$. If such a location did not exist, then a shortest path from F to v_j would exist via v_i, v_i would be a parent node of v_j (or vice versa), and the segment connecting v_i and v_j would be a covered segment by the extended SPT, which is a contradiction. In Dandelion, we mark such locations as *zip points* (ZIP).

In fact, zip point is resulting from the crossing of two BOPs on a single segment. As the focal location of a query moves, the two BOPs on a single segment (covering some portion from the two opposing ends of the segment) move closer, the distance on the common segment between the two BOPs grows smaller, until it disappears; when two BOPs are crossing each other, then two ZIPs are created. The Zip name comes from the effect of ”zipping up” two sub-trees of the Dandelion shortest path tree with the two BOPs as leaf nodes respectively.

Dandelion Tree. Let $CQ(F, r, T)$ be a continuous road network range query and let $Q(F_0, r), Q(F_1, r), Q(F_2, r), \ldots, Q(F_k, r)$ denote the sequence of evaluations of $CQ(F, r, T)$ in the duration defined by T. For each snapshot query $Q(F_i, r)$, we call its extended shortest path tree (with border-points and zip-points as leaf nodes and the focal location F as the root) a *Dandelion tree*. We refer to internal nodes of a Dandelion tree as internal covers.

When a mobile user moves from a focal location F to the next focal location F, if $network_disl(F, F) \ll r$ (one extreme case where F is 500 meters forward from F on the same road segment), then most of the internal covers in the Dandelion tree rooted at F will stay the same in the Dandelion tree rooted at F. Thus we can reuse the Dandelion tree rooted at F to compute the Dandelion tree rooted at F as the update to the Dandelion tree rooted at F is limited only to the root F (moved to F) and the leaf nodes (BOPs and ZIPs), which moved by $network_disl(F, F)$.

Lemma III: *The coverage represented by a Dandelion can be unambiguously separated into a forward half (FWD) and a backward half (BWD). The two halves are separated by ZIP points with the special property that the two equidistant paths to a ZIP start in the opposite direction at F.*

We first define the FWD half and BWD half of the coverage of a query. Given a continuous road network range query $CQ(F, r, T)$, the evaluation of this standing query is performed by executing the snapshot query $Q(F_i, r)$ over the time duration T periodically until T expires, and the focal location F_i changes as the query issuer moves forward ($i = 0, 1, 2, \ldots, k$, $F = F_0$). We refer to these snapshot queries as $Q(F_0, r)Q(F_1, r), Q(F_2, r), \ldots, Q(F_k, r)$. The sequence of road network locations $F_0, F_1, F_2, \ldots, F_k$ forms a travel trajectory of the query issuer. Given any focal location F_i, we call the moving direction of this trajectory from F_i to F_{i+1} the forward direction. Regardless of the actual travel direction of the query issuer, there are always two root covers emanating from the query focal F in the opposite direction on the same segment, say $e_{ij} = (v_i, v_j)$. If F is moved to F via v_i or towards v_i, then we call the root cover from F to v_i the *fwd* cover and refer to the root cover from F to v_j the *bwd* cover. All covers in the tree are descendants of either textitfwd or *bwd*, and thus can be easily separated into a FWD and a BWD set. Given a focal F and the coverage of the snapshot query $Q(F, r)$, those internal nodes with v_i as their ancestor node in the Dandelion tree rooted at F form the FWD half of the coverage of $Q(F, r)$. Similarly, those internal nodes with v_j as their ancestor node in the Dandelion tree rooted at F form the BWD half of the coverage.

In the case where the query focal F is located at one of the two ends of a segment, if the end node v_i is a junction connected by d segments, and the other end nodes of the $d-1$ segments are denoted by $v_{s_1}, v_{s_2}, \ldots, v_{s_{d-1}}$. If F moves to the junction v_i from v_{s_l} ($2 \leq l \leq d-1$), then those internal nodes with v_l as their ancestor node in the Dandelion tree rooted at F form the BWD half of the coverage of $Q(F, r)$. Those internal nodes with v_i as their ancestor node in the Dandelion tree rooted at F are part of the FWD half of the coverage.

ZIP Active Point – ZAP. While ZIPs are reachable via two equidistant paths originating from F, the FWD/BWD delimiter ZIP points' paths are different from the very first segment (either textitfwd or *bwd*). We may refer to ZIP points as ”ZIP Active Points” (ZAP), if they need to be distinguished from regular ZIP points, but behave similarly otherwise.

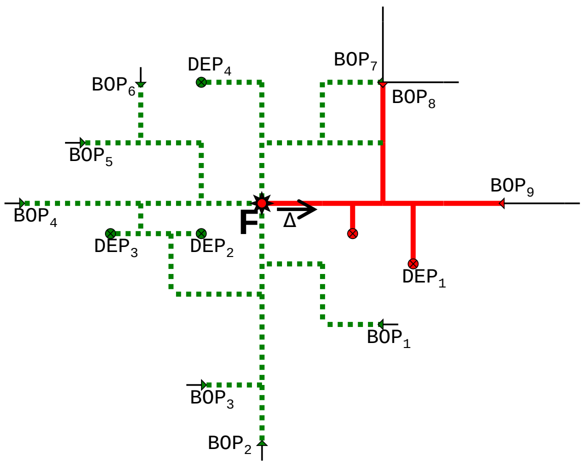
The main idea of the Dandelion basic algorithm is to utilize the FWD half coverage and the BWD half coverage of a snapshot query to maximize the reuse when using the coverage of query at F to compute the coverage of query at F. Concretely, the BOPs in the two halves of the coverage need to be updated differently to ensure the network distances from the new border points to the new focal F equals to r.

Figure 17 provides an illustrative example. The coverage of a query with initial focal point F is shown in Figure 17(a). The segments marked in thick red lines are those in the FWD half and the segments marked in thick dotted green lines are the BWD half. The coverage of the query after its focal point moved from F to F is shown in Figure 17(b). Compared to Figure 17(a), only the three border points BOP_7, BOP_8, BOP_9 are moved forward by distance of $network\_dist(F, F)$. Thus to compute the FWD half and the BWD half coverage of F, we can reuse most of the coverage of F. This is because only the update of these three BOPs are performed to obtain the FWD half of F. Similarly only a few updates are needed to compute the BWD half of F. The coverage reuse is shown in Figure 17(c).

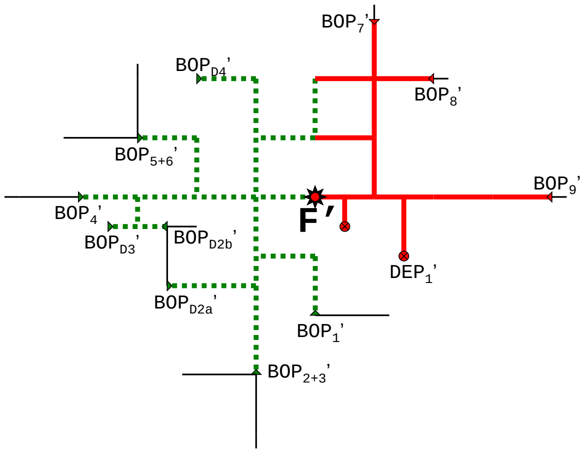
3.4 Dandelion Basic Algorithms

In this section we will present the basic algorithm for constructing a Dandelion tree anchored at F and the algorithm for reuse of the Dandelion tree anchored at F to compute the Dandelion tree at F. The algorithm for constructing the initial query coverage (Dandelion tree) takes the query focal F and the query radius r and computes the Dandelion tree anchored at F one hop at a time. We can model this process as growing the Dandelion. The FWD and BWD coverages are grown simulatenously.

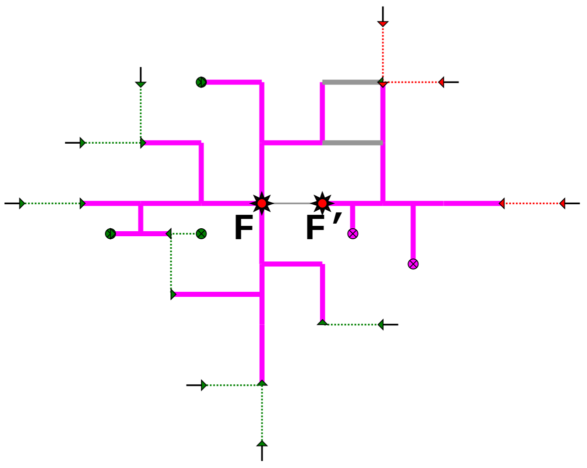
The algorithm for Dandelion reuse takes as input a Dandelion tree at F and the new query focal location F, and outputs the coverage of $Q(F, r)$, assuming F is the previous snapshot focal point and r is the query radius. One way to maximize the reuse opportunity is to compute $F - shared$ cover, $F - only$ cover and $F - only$ cover such that the coverage of F will be the union of $F - shared$ cover and $F - only$ cover. $F - only$ cover can be computed by examining the growing part of the Dandelion tree when the focal location is moved from F to F. Similarly, $F - shared$ can be computed by examining both the FWD coverage and the BWD coverage of the Dandelion tree. Thus we will divide our discussion



(a) Coverage at F.

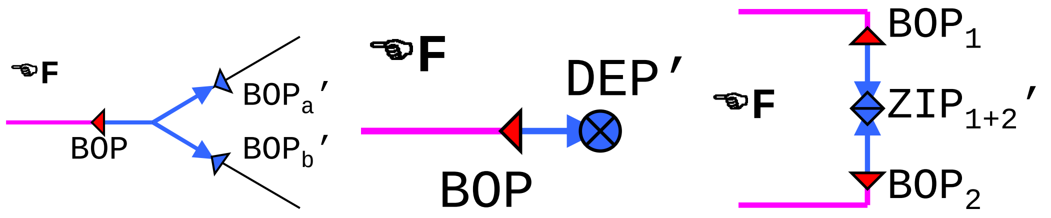


(b) Coverage after `mov` (at F').



(c) Coverage reuse.

Figure 17: Evolution of BWD (green) and FWD (red) portions of coverage of a query at F as it is transformed to F'.



(a) BOP-Push-Split. (b) BOP-Push-Dead. (c) BOP-Push-Zip.

Figure 18: Primitive BOP Push Operations

into two phases: growing Dandelion (FWD) and BWD Dandelion. In the growing Dandelion computation, the Dandelion tree may grow by updating the BOPs, ZIPs and some of its internal nodes. To maximize reuse of the query coverage of F in computing the coverage of F, we introduce three primitive BOP push operations, which transform BOPs into new BOPs, ZIPs or dead points in the road network, called DEPs. Similarly, we introduce three primitive BOP pull operations for computing the BWD Dandelion at F.

In the subsequent sections, we first introduce the BOP Push and Pull Operations and then we describe the data structures for promoting Dandelion reuse. We will also describe the Dandelion basic reuse algorithm, which divides the query coverage into FWD and BWD halves. To improve the efficiency of Dandelion basic algorithms, we introduce two advanced Dandelion algorithms – Dandelion-Trident and Dandelion2 in Section 3.5 and Section 3.6 respectively. All three Dandelion algorithms utilize BOPs, ZIPs, fully covered and partially covered segments, query coverage and Dandelion tree as the fundamental basics, but one improves the other by using a more-compact and reuse-conscious data structure.

3.4.1 BOP Push and Pull Operations

We conceptualize the growing phase of a Dandelion tree as a series of local BOP update operations, in which a BOP is ”pushed” outward and to a location that is outside the query coverage of F. There are only three types of BOP push operations (**BOP-push ops**) that are used in the growing of a Dandelion tree: BOP-Push-Split, BOP-Push-Dead, and BOP-Push-Zip.

BOP-Push-Split: BOP $\rightarrow$ (d-1) $\times$ BOP.

At a d-way junction, a single BOP, residing on the segment from v_1 to v_2, is split into $d - 1$ BOPs, denoted by $BOP_1, BOP_2, \ldots, BOP_{d-1}$ ($d > 1$), one on each of the $d - 1$ not-covered segments connected to the junction. Thus the segment from v_1 to v_2 becomes an internal cover. An internal cover can never serve as the basis for further expansion (only covers in the perimeter of the query), thus we no longer need to update its distance from F, even after the query moves to a new F'' location. This is a primary source of performance improvement in the Dandelion algorithm, as the internal segments do not need to be updated. Fig. 33(b) shows an illustrative example.

BOP-Push-Dead: BOP $\rightarrow$ (d-1) $\times$ DEP.

Reaching the end of a dead-end segment, a BOP is transformed into a DEP, and its location no longer needs to be updated, as it cannot serve as a basis for further expansion in the growing Dandelion phase. Fig. 33(d) gives an example illustration. However, we need to keep track of the distances at these ”unmovable” BOPs as they may serve as the basis for the Dandelion shrinking if F is moving backward. Thus we include them in the perimeter.

BOP-Push-Zip: 2 $\times$ BOP $\rightarrow$ 2 $\times$ ZIP. When two BOPs are pushed across one another on a segment, they are transformed into two ZIPs, as shown in Fig. 33(f). ZIPs – like DEPs – are unmovable, but we need to keep track of their distances from the focal point F, as they can serve as the basis for further expansion. Consequently, ZIPs are considered part of the perimeter.

Similarly, when the query focal F is moved to F, some part of the Dandelion tree at F will no longer be included in the Dandelion tree at F. We can conceptualize this shrinking of a Dandelion tree at F as a series of local BOP update operations, in which a BOP is ”pulled” and merged through three types of primitive BOP pull operations: BOP-Pull-Merge, BOP-Pull-Undead, BOP-Pull-Unzip.

BOP-Pull-Merge: $(d - 1)\times$ BOP $\rightarrow$ BOP.

At a d-way junction, two or more BOPs, residing on different segments sharing the same end node, say v_1, are merged into one BOP on a segment connected to v_1. We add this BOP into the perimeter set to replace the previous BOPs in the coverage of F. The segment from v_1 to v_2 becomes only partially covered. This is the reverse of the example shown on Fig. 33(b).

BOP-Pull-Undead: DEP $\rightarrow$ BOP.

Some border points of DEP-type points may be pulled to alive when query focal is moved from F to F. This is the reverse of the example shown on Fig. 33(d).

BOP-Pull-Unzip: 2 $\times$ ZIP $\rightarrow$ 2 $\times$ BOP. When two ZIPs are pulled from opposite directions on a segment, they are transformed into two BOPs, in the reverse of the example shown on Fig. 33(f). Update to the perimeter set is performed to reflect such transformation.

We have discussed four important road network points: border points (BOP), dead-end points (DEP), inactive zip points (ZIP), and active zip points (ZAP). We distinguish these four types of points in the FWD and BWD portion of the Dandelion tree with a prefix (e.g. fBOP is a border point in the forward tree and bZIP is a ZIP in the backward tree). Figure 19 gives a brief overview of the state transitions of these four network location points. First, consider the construction of the Dandelion tree as a gradual growing of the tree by incrementally increasing its range from 0 to r. Initially, there is only a single F focal point, and the query has a range of 0. When the range is first increased, the simplest tree is created: with one fBOP and one bBOP, assuming that F is not at a junction node. As we grow the query range continuosuly until reaching r, all other points are derived from these two points.

3.4.2 Data structures and Dandelion Tree Construction

In this section, we describe four key data structures for construction and reuse of a Dandelion tree: *cover, SegCovMap, ordered priority queue* and *perimeter*.

A **cover** is a fundamental data structure representing the basic information about a fully covered or a partially covered segment. A cover is specified by the following components:

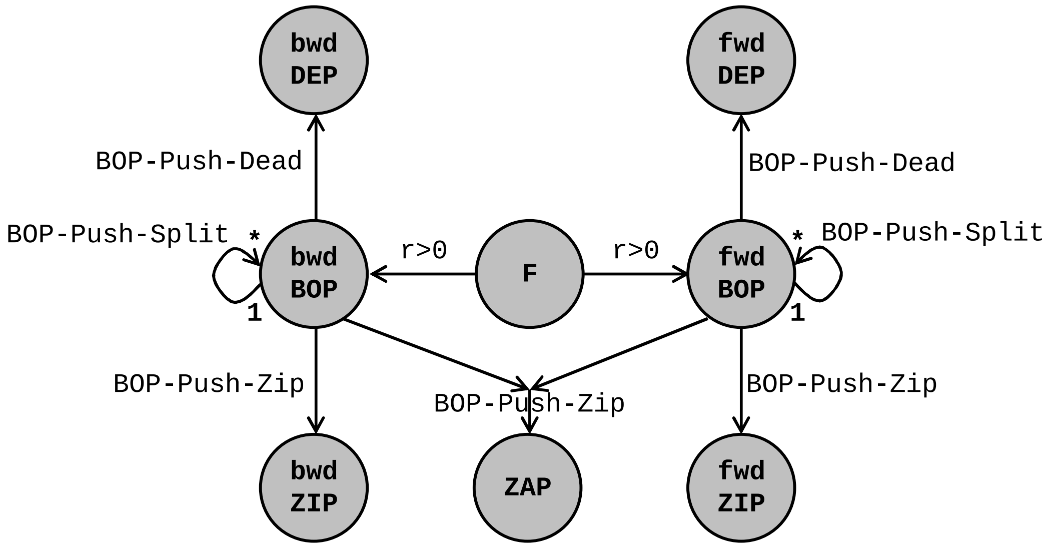


Figure 19: State transition and lifecycle of four points in a $grow$ operation.

(i) the cover type (BOP, DEP, ZIP or internal); (ii) a *seg* pointer to its underlying segment; (iii) the *coveredEndIdx*, indicating which of the two ends of the segment is closer to F (and is thus guaranteed to be within the query coverage area, while the other end may fall outside of it); (iv) the *progress* indicating the portion of the segment that is covered (which is by definition the full segment length for DEP and internal covers); (v) a *parent* pointer to the internal-type cover that is upstream from the cover on the path towards F, the root of the Dandelion tree; and (vi) for ZIP points an additional *zipper* pointer to the paired ZIP cover. Furthermore, Dandelion covers contain a single *half* bit, which can be used to determine if two covers are in the same half of the Dandelion tree. To facilitate reuse, the *half* bit by itself does not indicate whether that half is FWD or BWD. In order to determine whether that half is FWD or BWD, the comparison must be performed to find whether the half bit of the cover is the same as the half bit of the *fwd* or *bwd* cover of F. This allows the reuse of the Dandelion tree of F to compute the Dandelion tree of F regardless which direction the path from F to F is.

A **SegCovMap** is a hash mapping from segment ID to the list of covers on that segment. A segment may map to either 1 cover (1 BOP; 1 DEP; 1 internal), or 2 covers (2 BOPs; 2 ZIPs). The segment of F is special case, as it may map to 1 BOP + 1 internal cover; or in a special situation to 2 BOPs + 1 internal cover; or 2 ZIPs + 1 internal cover. The SegCovMap can be used to for coverage checking (i.e. to determine, whether any specified road network location is covered by the query or not), and is also used to find existing covers on a segment (e.g. for detecting whether a BOP-Push-Zip operation needs to occur). The SegCovMap can be viewed as an inverted index that maps segments to the covers on them and is updated mostly by adding $d-1$ new covers at a d-way junction during a BOP-Push-Split operation. Removals also occur to SegCovMap. For example, segments in the portion of FWD, which are not shared by the FWD at F, must be removed one by one when reusing the FWD at F to compute the FWD at F.

An **ordered priority queue** is used to process covers during coverage growing, and perform the BOP push operations in the correct sequence. The queue is ordered by the smallest distance from F. Three basic operations are performed on the ordered priority queue. An ”enqueue” operation happens when a cover is inserted into the ordered priority queue. A ”dequeue” operation takes place when a previously enqueued cover must be removed from the queue. A ”popqueue” operation occurs when the head of the queue is popped out for processing. The initialization of the queue may differ depending on whether the queue is used to create the initial Dandelion tree (coverage) or the queue is used to compute the coverage at F by maximizing the reuse of the coverage at F.

A **perimeter set** is an unordered set, containing all the non-internal covers, as these may serve as the basis for further expansion. In the basic Dandelion reuse algorithm, a hash table is used for quick containment checking and removal of a non-internal cover from the perimeter.

Dandelion Tree Construction.

Upon installation of a continuous road network range query $CQ(F, r, T)$, its initial evaluation is to construct a Dandelion tree anchored at F scoped by the network distance range r. The algorithm starts the creation of an initial Dandelion tree using the ordered priority queue. We assume that F is a location on a segment. First, the queue is initialized with the two root BOP covers (*fwd* and *bwd*), both located at exactly F, but each with a different *coveredEndIdx*. The Dandelion tree grows as we examine each cover in the queue and update the queue accordingly. BOP-Push operations are performed in a *split-then-push-out* fashion: when a cover is popped from the queue, if the cover is a partial covered segment, then it is pushed out to the end of the current segment, and if its distance to F is less than query radius r, then its distance is updated, and this cover is re-enqueued. When this cover is encountered at the head of the queue again, its *progress* is at a junction node, and if the junction is not a dead end, then this cover can be immediately pushed and split onto the connecting segments. This ensures that the border points of the query are computed hop by hop from F with r radius. Continuing this process, the children covers are in turn pushed out to the end of their respective segments, if their distances to F are less than query radius r, then their *progress* and distances from F are updated, and they are enqueued for subsequent examination.

If a push-out results in a distance to F farther than r, then the push is limited by placing a BOP on the segment such that the BOP is at exactly r distance from F, and this BOP is not enqueued. Immediately after the push-out of a child BOP, the *overpush-detection* is performed: SegCovMap is consulted for other covers on the segment, and for each such cover, we check (i) if the pushed-out BOP is the only BOP on the segment or else if it does not cross any other BOP on the segment, if yes, we add this BOP to the *perimeter set* and also add this BOP as a leaf in the Dandelion tree. (ii) Otherwie, if the pushed-out BOP has been *overpushed* (i.e., pushed beyond the location of another BOP on the same segment), then two BOPs have crossed over each other and a BOP-Push-Zip is performed, and the two BOPs equidistant ZIP location is computed (which is guaranteed to be on the current segment due to the processing order guaranteed by the priority queue). The ZIPs are added to both the Dandelion tree as lead nodes and to the *perimeter set*.

When a BOP-Push-Split is pushed and splits the current BOP into $d-1$ BOP segments, some of them may no longer be pushed before reaching the road-network distance r from

F, because they are dead-end segments and at a dead end (DEP) there are no outgoing segments to split onto, then a BOP-Push-Dead is invoked instead BOP-Push-Split.

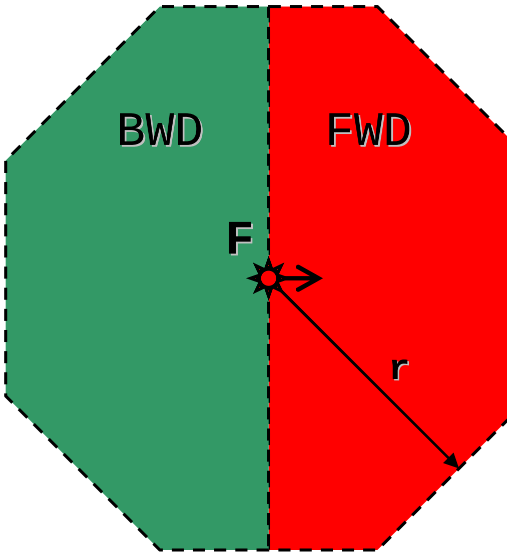
When the cover popped from the ordered priority queue is a fully covered segment, it is added to the Dandelion tree as an internal node and is not enqueued.

This process iterates until all covers in the queue are examined. As a result, the Dandelion tree is fully constructed.

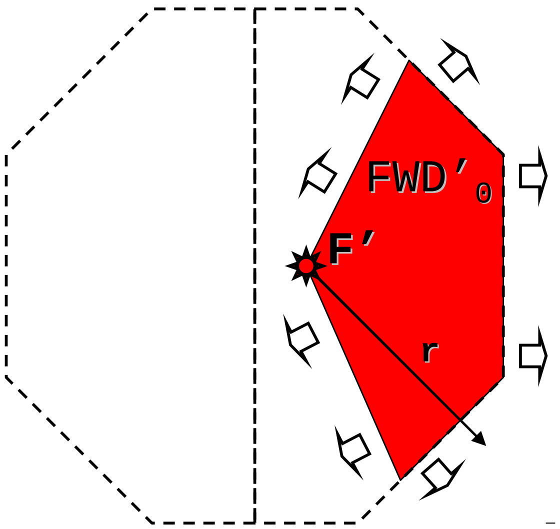
To maximize the reuse opportunity, we keep a separate SegCovMap and a separate perimeter set for FWD and BWD. There are two obvious advantages of this design. First, it allows the wholesale disposal of the segment-to-cover mapping in the BWD portion of a Dandelion tree efficiently. In addition, should F' be in BWD (if the user travels in a beeline after the query coverage re-evaluation at F), then the FWD and BWD SegCovMap and perimeter set are simply swapped first. This swapping also explains why a cover's *half* bit is not by itself an indicator of containment in FWD. If the bit were a direct indicator of one half, then after a swap, all the covers in the newly-FWD Dandelion half tree would need to be traversed for updating this bit. This is another example of the Dandelion design that maximizes both the reuse potential and the reuse efficiency by minimizing unnecessary computation and update operations.

3.4.3 Dandelion Basic FWD-BWD Reuse Algorithm

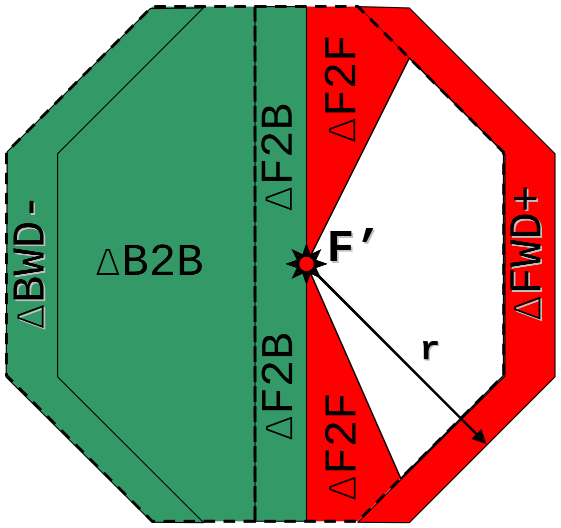
We have described the algorithm to construct an initial Dandelion tree (coverage) for the initial evaluation of a continuous road network range query using the ordered priority queue. Given the query focal F, the the queue is initialized with only two root covers: the *fwd* cover and the *bwd* cover. Upon the completion of the tree construction, we obtain the FWD portion of the coverage and the BWD portion of the coverage from the Dandelion tree respectively, as shown in Figure 31(a). In the basic Dandelion reuse, we keep a separate SevCovMap and a separate perimeter set for FWD coverage and BWD coverage in order to speed up the computation of the coverage at F by reuse of the coverage at F.



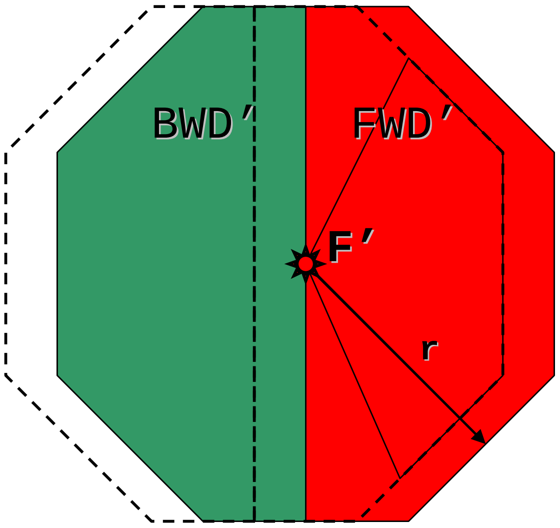
(a) Coverage at F.



(b) Part of F coverage reusable at F'.

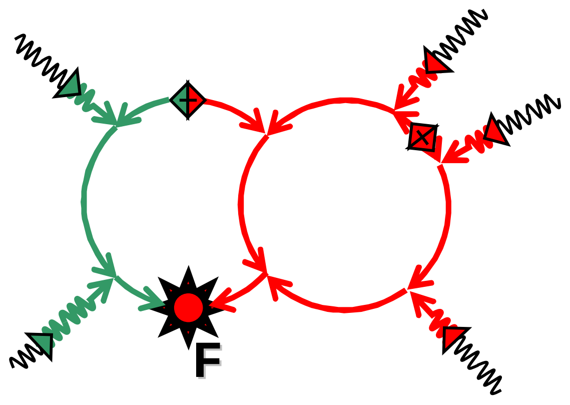


(c) Coverage changes in a F-to-F transformation.

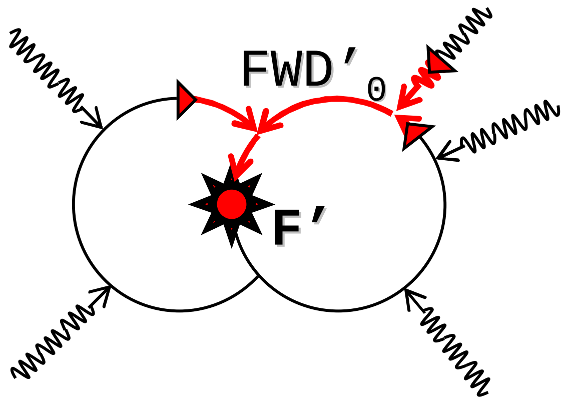


(d) Coverage at F'.

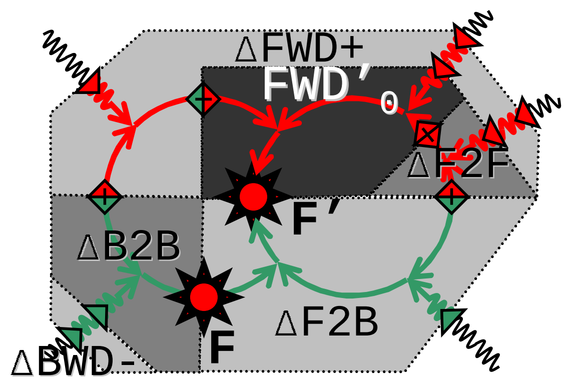
Figure 20: Schematic of evolution of BWD and FWD portions of coverage of a query at F as it is transformed to F'. (All dist. on road – real coverage not octogonal. Coverage at F shown dashed on all figures for reference.)



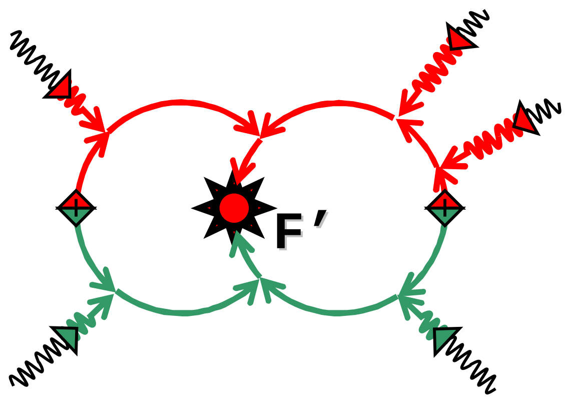
(a) Coverage at F.



(b) Portion of coverage at F reusable at F'.



(c) Changes in coverage during a transformation from F to F'.



(d) Coverage at F'.

Figure 21: Example of evolution of BWD and FWD portions of coverage of a query at F as it is transformed to F'.

When the query focal point is moved from F to F, the basic Dandelion reuse algorithm aims at computing the FWD coverage and the BWD coverage at F by maximizing the reuse of the FWD coverage and BWD coverage at F. The key insight for Dandelion tree reuse is that a forward portion of the Dandelion tree will have the same structure at both F and F', denoted by FWD$_0$ (see Figure 31(b)), and all distances of BOPs, DEPs and ZIPs in FWD simply need to be increased by the dx displacement between F and F' to be correct in the new FWD' tree (regardless of the actual route taken by the user, as a coverage only depends on F's location and r). The FWD$'$ subtree is simply the portion of FWD that is also forward from the new focal location F'.

However, those segments that are in FWD, but not in FWD$'$ need to be removed from both the SegCovMap and the perimeter set (see Figure 31(c)). These segments can be found by traversing in the Dandelion tree all of the side-trees along the shortest path from F to F'. As covers in these side-trees are removed, the ZIP points that straddle the demarcation line between FWD$'$ and the non-overlapping part of FWD are turned into BOPs (by a simple type change). Finally, the FWD perimeter set will only contain those BOPs whose distances and paths are correct with respect to the new focal location F'.

Construction of Dandelion Tree at F by Reuse. One way to reuse the coverage at F in constructing the Dandelion tree at F is to initialize the ordered priority queue with both the FWD$'_0$ perimeter set (instead of a new root *fwd* cover), and a new root *bwd* cover. Subsequently, the queue is processed iteratively as described in the initial query coverage computation. As a result, only the perimeter of FWD$'_0$ has been traversed, and its internal covers (which may be numerous) have been reused completely, without ever being touched during the re-use.

Figure 31 shows the conceptual formulation of FWD$'_0$. Figure 21 gives a detailed example. FWD$'_0$ is the reusable portion of the coverage. ΔFWD+ is the FWD portion of the query coverage at F' that has not been covered at F. ΔF2F is the portion of the query coverage that is in both FWD and FWD', but via different paths, and is thus not reusable.

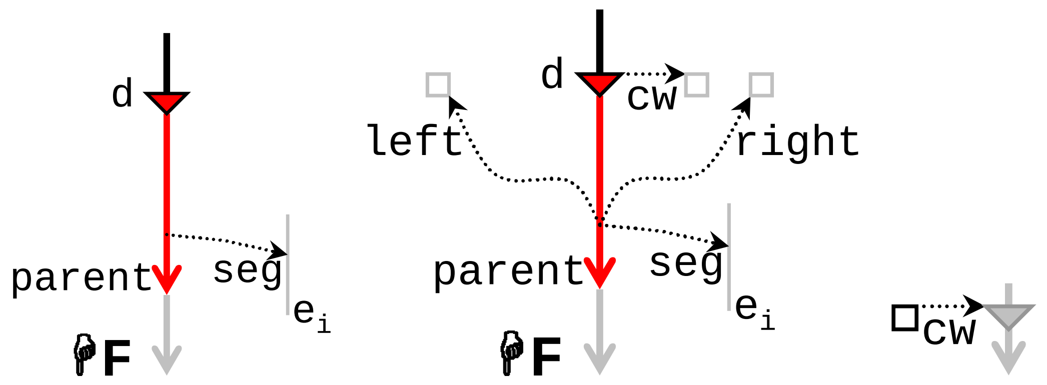
ΔF2B is the portion of the FWD tree that is in BWD' at F'. ΔB2B is the portion of the BWD tree that is also in BWD', but still must be recomputed, as the distances into it may have changed in unpredictable ways (notably, due to paths that lead into this portion via ΔF2B, as they are shorter than those from F' backwards via F). ΔBWD- is the portion of the BWD tree that is no longer covered at F'.

In summary, we can classify covers into three categories on each re-evaluation: (i) New covers signify segment covers that have been created during the re-evaluation step. (ii) Reused, but updated, covers are those that were created during a previous re-evaluation, but have been modified in the current step (for example, reused BOPs would fall in this category). (iii) Reused and not updated covers are those that are present in a current re-evaluation step in exactly the same form as in the previous step (e.g. internal covers in FWD$'_0$). Our experiences with Dandelion show that when the query radius reaches a non-trivial size, the ratio of the three reuse types stabilizes, indicating good scalability of the Dandelion reuse algorithms.

3.5 Dandelion-T

In the Dandelion basic reuse algorithm, to compute the Dandelion tree at the new focal location F, we need to first identify FWD$'_0$, namely the FWD portion that is shared at both F and F. This requires the algorithm to traverse the perimeter set of the FWD coverage in order to find those BOPs in FWD$'_0$, and then follow the parent pointers at BOPs to obtain the entire shared portion of FWD, namely FWD$'_0$. Given that the perimeter set is an unordered set of all non-internal covers (partially covered segments) with a hash table for quick containment check and removal, when the size of the perimeter set is large, a sequential scan of the perimeter set can be quite expensive, even though we keep a separate perimeter set for FWD and BWD.

We envision that a fast way to identify those covers shared by the Dandelion tree (coverage) at F and the Dandelion tree (coverage) at F is to have the capability to find all the



(a) Simple cover. (b) Trident. (c) Guide.

Figure 22: Trident and Guide data structures.

non-internal covers in a subtree anchored at a given internal cover. This capability will allow us to quickly find those non-internal covers that are part of the subtree anchored at an internal cover, which is included in both the coverage at F and the coverage at F, without scanning irrelevant covers in the perimeter set. This motivates us to introduce Trident, a new data structure that allows us to create an inverted cover-to-perimeter index such that we can find the set of non-internal covers in the subtree anchored at any given cover efficiently, speeding up the Dandelion reuse based query re-evaluations.

Clearly, in Dandelion-T we no longer need to keep a separate perimeter set for FWD and BWD due to the integral update of a correctly ordered perimeter-list during all BOP-push and BOP-pull operations.

In the rest of the chapter we call the Dandelion basic reuse algorithm simply as Dandelion and refer to the Dandelion that supports the Trident structure as **Dandelion-T**.

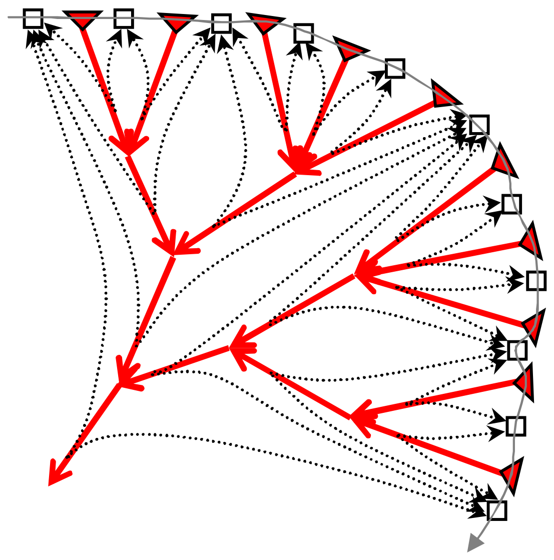
Trident is a value-added auxiliary data structure to extend a cover in the basic Dandelion in order to further improve the performance of Dandelion reuse. A Trident cover is an extended cover that contains three additional pointers in addition to the information about a cover, which allows the management of the perimeter set as a linked list (instead of a hash table) and also enables us to quickly find the set of BOPs in the perimeter set that are descendants of the Trident cover. Formally, a **trident** cover (Fig. 22) contains all the attributes of a simple cover (Fig 22(a)), except for the *half* bit, and also contains: a *left* and a *right* pointer, which ensure that the perimeter list traversed from *left* to *right* contains all the BOP, DEP and ZIP covers that are descendants of the trident; and a *cw* (clockwise) pointer, pointing clockwise towards tridents in the perimeter of the Dandelion. All three pointers of a Trident $-$ *left*, *right* and *cw* point to a **guide** data structure (Fig. 22(c)), which does not represent any location on the road network, and simply serves as an interstitial between covers. A guide contains a single *cw* (clockwise) pointer to a trident in the clockwise direction. With this guide structure, all covers in the perimeter are linked in a clockwise fashion, making the access to the BOPs relevant to any given cover convenient and fast. Therefore the perimeter is composed of an alternating list of tridents and guides; each trident is an extended cover of either type BOP or DEP or ZIP. The FWD half of a Dandelion tree is accessible by iterating from fwd.left to fwd.right (and similarly for BWD). Also it is worth to note that fwd.left = bwd.right, and fwd.right = bwd.left.

Figure 23 gives an intuitive view of some part of a sample Dandelion-T. Figure 23(a) shows the sample portion of the Dandelion-T with internal and BOP type tridents. Figure 23(b) shows the sample portion of the Dandelion-T with tridents of all four types (including ZIP and DEP tridents).

The correctness of a perimeter list with tridents and guides can be maintained using the three primitive BOP-Push operations for trident covers, which is a slight modification of the three basic BOP-Push operations in the Dandelion basic algorithm (recall Section 3.4.1). Figure 24 provides an intuitive illustration of trident structure updates.

By design, any complex operation in the FWD Dandelion can be broken down into the three simple BOP-Push operations (and similarly for the BWD Dandelion), while keeping all trident covers in a correct clockwise ordering in the perimeter. Figure 25 shows an example of how a proposed new BOP-Push-Merge subop can be replaced with a BOP-Push-Split followed by a BOP-Push-Zip.

All ZIP points also remain in the perimeter. To help understand what a clockwise ordering means in the case where BOPs are pushed into ZIPs, we provide an example in


(a) Dandelion-T with internal and BOP type tridents.

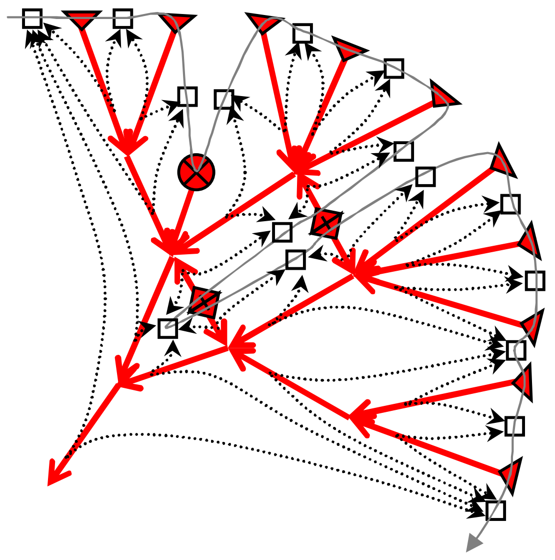
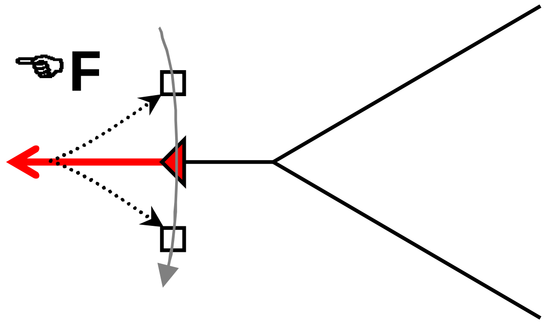

(b) Dandelion-T with internal, BOP, DEP and ZIP type tridents.

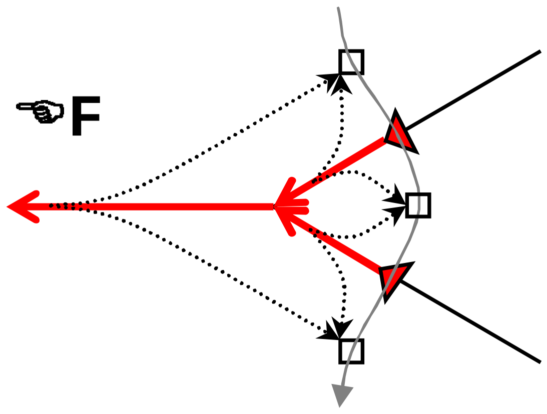
Figure 23: Detail of a sample Dandelion-T tree portion.

Figure 26 to show the detailed steps of how the ordering is maintained once the creation of a ZIP is taken into account. Concretely, Figure 26(a) shows two BOPs before entering the crossing street. Figure 26(b) shows the two BOPs are now split into four BOPs after expansion of each BOP onto the other two segments by crossing the nearby three way junction. Figure 26(c) shows the case in which two BOPs on the cross-street expand by crossing one another and BOP-Push-Zip generates two ZIPs to replace the two BOPs on the cross-street segment. Due to the space constraint, in this chapter we omit the theoretical analysis of the correctness of the ordering of tridents and guides in the perimeter set.

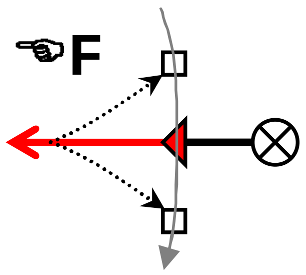
We have implemented Dandelion (basic reuse) and Dandelion-T (trident powered reuse) in our first prototype system. To provide an intuitive visualization of Dandelion reuse enabled network range query evaluation, we present some visualization screenshots taken from our prototype of Dandelion and Dandelion-T, ranging from the simplest query coverage with only two BOPs (Fig 27(a)), to simple queries with only BOPs (Fig 27(b) and Fig 27(c)), and to highly complicated query coverage trees (Fig 28(a) and Fig 28(b)). Fig 27(b) shows a query with very small query radius in terms of network distance of



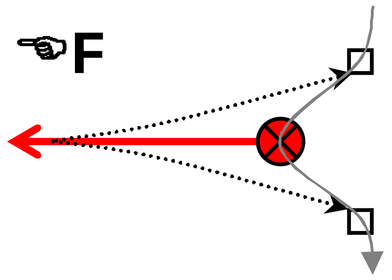
(a) Before BOP-Push-Split.



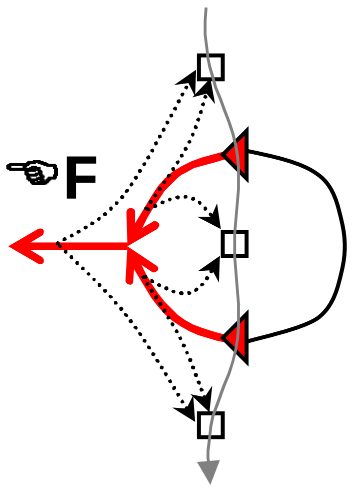
(b) After BOP-Push-Split.



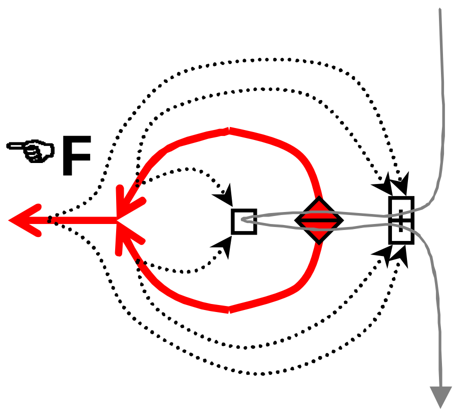
(c) Before BOP-Push-Dead.



(d) After BOP-Push-Dead.

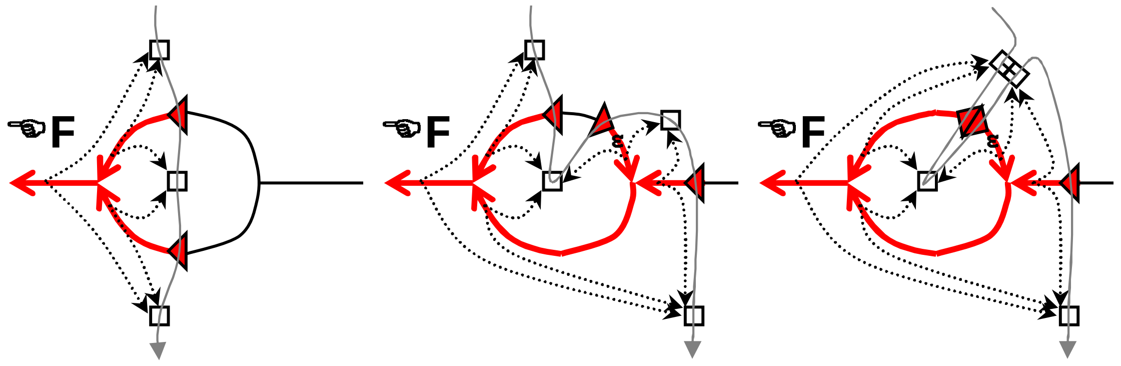


(e) Before BOP-Push-Zip.



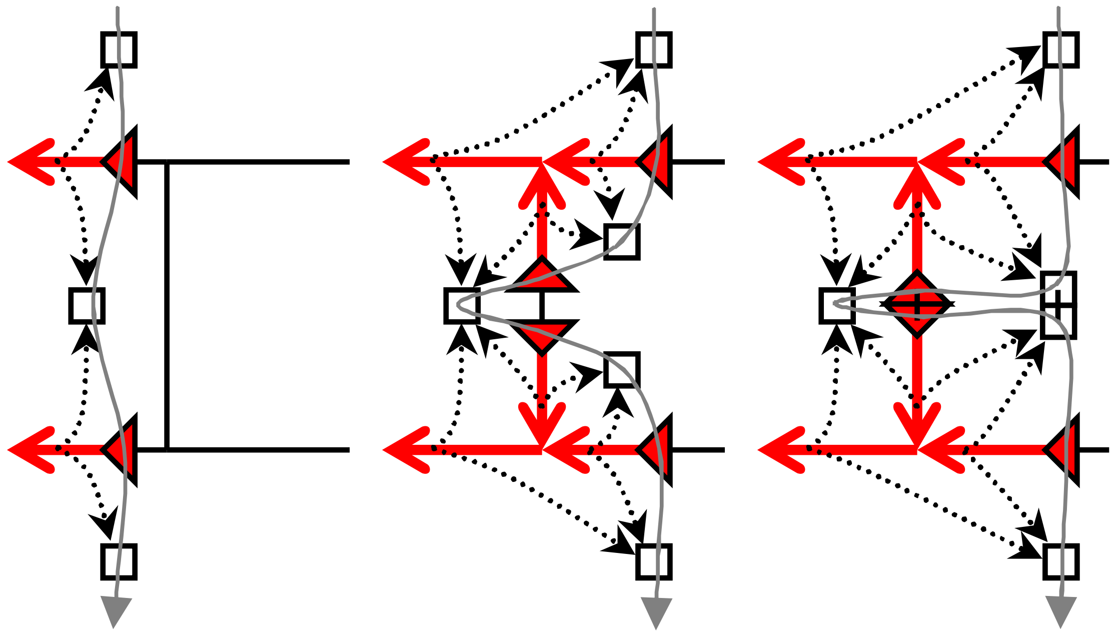
(f) After BOP-Push-Zip.

Figure 24: Dandelion-T data structure updates after FWD BOP-Push ops.



(a) Before BOP-Push-Merge. (b) After BOP-Push-Split. (c) After BOP-Push-Zip.

Figure 25: Decomposition of a complex BOP-Push-Merge situation into basic suboperations.



(a) 2 BOPs before cross-street. (b) 4 BOPs after expansion ont cross-street. (c) 2 BOPs + 2 ZIPs after zipping on cross-street.

Figure 26: Ladder example showing the correct ordering of ZIPs in the perimeter.

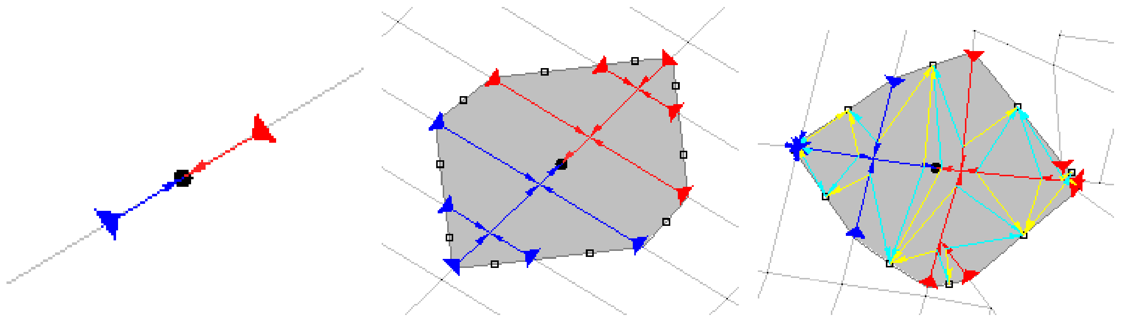

(a) Simplest possible query. (b) Small query. (c) Small query with left/right pointers.

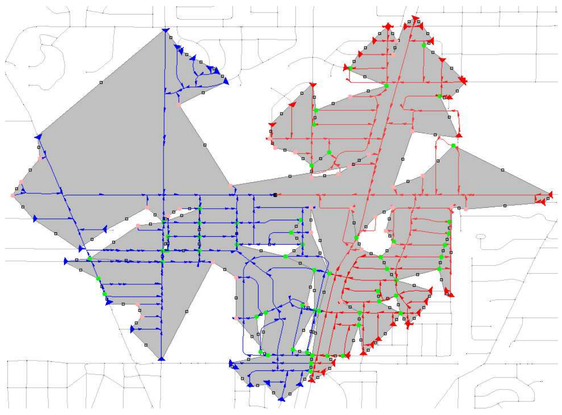
Figure 27: Sample small queries. ($r = 20 - 200\ m$; map scales not comparable)

less than 200 meters. Fig 27(c) shows a small radius query with tridents and guides in the perimeter, while highlighting *left* and *right* pointers. Fig 28(a) shows a sample query with relatively larger radius of 2000 meters with the query coverage computed using Dandelion. Fig 28(b) shows a sample query with the same radius of 2000 meters but at a different query focal point with the query coverage computed using Dandelion-T, and highlighting the *left* and *right* pointers.

3.6 Tree-transformation with Dandelion2

We have shown that Dandelion2 is faster and more effective than Dandelion basic algorithm for speeding up the re-evaluations by maximum reuse, thanks to the compact trident data structure to maintain a correctly ordered perimeter list during the transformation of the Dandelion tree at F to the Dandelion tree at F. However, by carefully examining the transformation process in Dandelion and Dandelion-T, we observe that when the query focal point is moved from F to F, the transformation of the Dandelion tree (coverage) at F to the Dandelion tree (coverage) at F can be done more intelligently and more efficiently.

First, we argue that the coverage at any location depends only on query parameters (such as focal location and query range) and the topology of the underlying spatial network, and it should be independent of the method used to calculate the coverage. Concretely, there may be multiple paths from F to F. Thus, transformation operations along any path from F


(a) Large query.

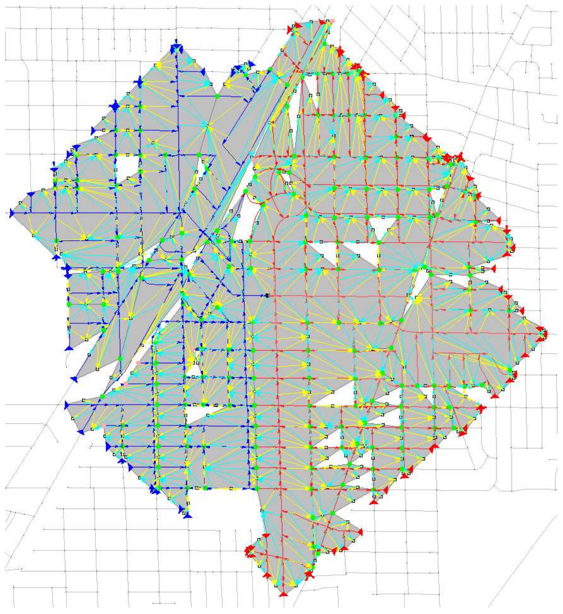

(b) Large query with left/right pointers.

Figure 28: Sample large queries. ($r = 2000\ m$; map scales not comparable)

to F' must yield the same coverage at F'. By utilizing this path-independence property, we can perform the transformation along the shortest path or the path with the fewest number of segments (and thus operations), instead of using the actual path taken by the mobile user who issued the continuous range query.

Second, the coverage-transformation from F to F' can be broken down into a series of primitive `mov` (move) and `jmp` (jump) operations.

A **`mov`** dx operation transforms the coverage at $F_1 = (e, p_1)$ into the coverage at $F_2 = (e, p_2)$, where $p_2 = p_1 + dx$; i.e., it moves the query's coverage from a focal location F_1 on a segment to another location F_2 on the same segment. The parameter of `mov` is a real value dx, which is the distance movement on the segment, considering the origin ($p = 0$) of the segment is at the end-vertex with the lower index. If $dx > 0$, then the movement is towards the end-vertex with the higher index (the ”end” of the segment); and if $dx < 0$, then the movement is towards the lower index vertex (the ”start” of the segment).

A **`jmp`** e_2 operation transforms the coverage at $F_1 = (e_1, len(e_1))$ into the coverage at $F_2 = (e_2, 0)$, where e_1 and e_2 are connected edges and $F_1 \equiv F_2$; i.e., the primitive `jmp`

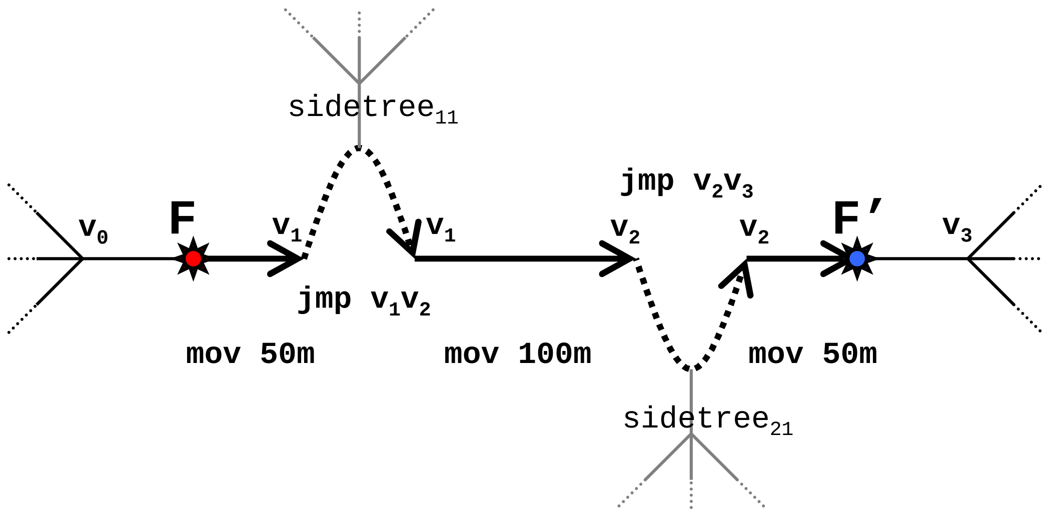


Figure 29: Observation III: Breakdown of tree-transformation from F to F', as shown on Figure 15(a), into a series of mov and jmp operations.

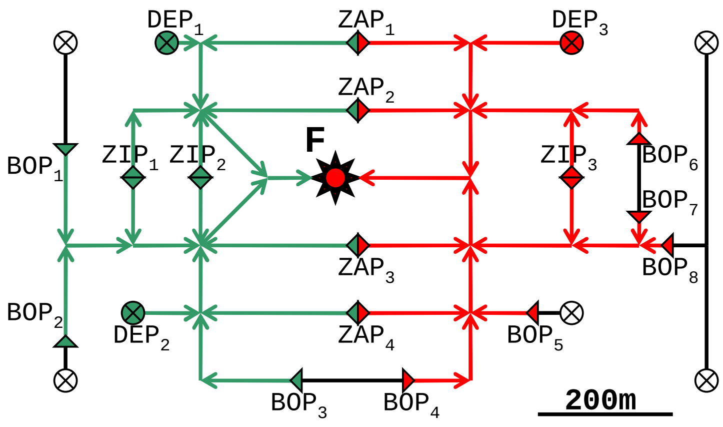
operation jumps the query's coverage from one segment's end to the next segment's beginning, without actually moving the coverage by any distance. The parameter of `jmp` is the edge to jump to, and a jump is only a valid operation at the end of a segment. By $F_1 \equiv F_2$, we mean that the network location exactly at a d-way vertex can be described by d different forms, one per edge, say (e_i, p) ($i = 1, 2, \ldots, d$), and all these forms represent the same network location. Furthermore, the progress p can be any value in the range of [0, $len(e)$] (the total length of the edge), in all possible combinations. Clearly, a `jmp` operation does not move either the focal location, the border-points, or the covered network locations. Although the network locations before and after the jump are equivalent, `jmp` transforms the coverage from a focal location immediately before a junction, to the coverage immediately past the junction. Using the example in Figure 29, it is a move from $F_1 = (e_1, len(e_1) - \epsilon)$ to $F_2 = (e_2, \epsilon)$ while $\epsilon \to 0$).

Consider Figure 29, which breaks the $\Delta = 200\ m$ distance between the two queries of Figure 15 into a series of 5 transformation steps: (1) `mov 50m` moves the query focal location from F to $(v_0v_1, 100m)$; (2) `jmp` v_1v_2 jumps the focal location from the current $(v_0v_1, 100m)$ to $(v_1v_2, 0m)$, both of which are exactly at v_1; (3) `mov 100m` moves us from the start of the v_0v_1 edge to its end; (4) `jmp` v_2, v_3 is a new jump without movement; and finally (5) `mov 50m` moves the focal location to the desired F'.

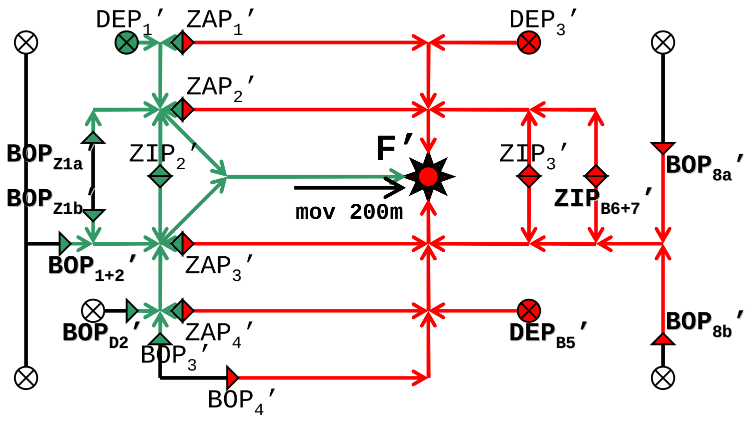
In short, we argue that the coverage at any location is independent of the method used to calculate the coverage, though it may depend on query parameters, such as focal location and range and the underlying spatial network topology. Thus, we can prove that the series of `mov` and `jmp` transformation operations along any path from F to F' must yield the same coverage at F'. Thus, by utilizing the pair of primitive transformation operations `mov` and `jmp`, one can obtain an optimal implementation of the coverage transformation from F to F by simply performing the transformation along the shortest path or the path with the fewest number of segments (and thus operations), instead of being concerned with the actual path taken by the mobile user who issued the continuous network range query.

Figure 30 shows the evolution of BWD (green) and FWD (red) portions of the coverage of a query at F, as it is successively transformed from F (Figure 30(a)) to F' (Figure 30(b)) and then to F'' (Figure 30(c)), with $r = 600\ m$ and all distances are to scale. From this example, we can see the advantage of the `mov` and `jmp` transformation for Dandelion reuse. Note that F and F are on the same segment. With a `mov` operation, F is pushed to the end-vertex with the higher index on the same segment. The vertex is a 3-way junction and thus the segment where F was located is now marked as a cover in BWD (color of the segment is changed from partially red to green). The other two segments are covers common in FWD at both F and F. The four ZAPs are pushed by the equal distance of $network\_dist(F, F)$ and the ZAP segments are updated in FWD and removed from BWD at F. ZIP_3 and DEP_3 are unchanged. BOP_6 and BOP_7 become a ZIP_{B6+7}. BOP_8 is push-split into BOP_{8a} and BOP_{8b}. BOP_5 is push-dead to DEP_{B5}. Similarly, the coverage after `jmp` at F chooses the right segment to push forward, making the left cover to be removed from FWD and inserted into BWD. The change of the left cover at F to be in the BWD coverage triggered the transformation of all segment covers connected to this cover into the same half (BWD in this case).

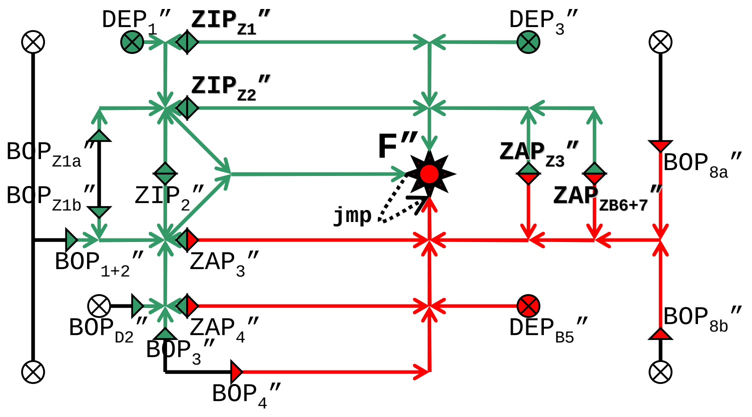
Now consider the Dandelion tree with the constant range r: by using `mov` and `jmp`



(a) Coverage at F.

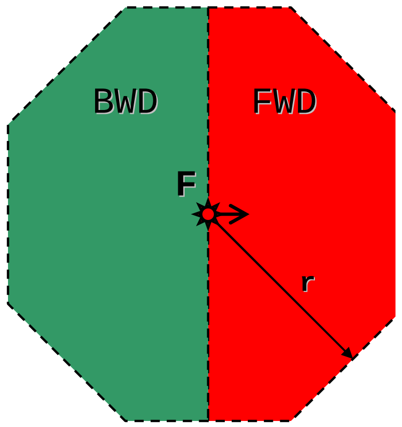


(b) Coverage after `mov` (at F').

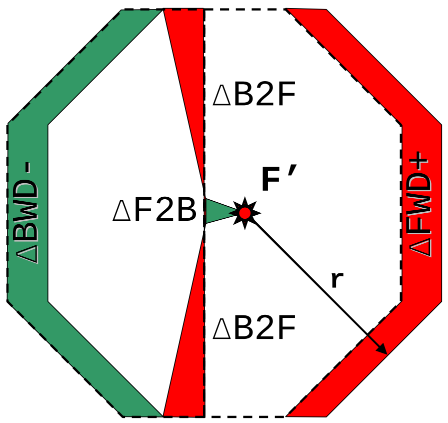


(c) Coverage after `jmp` (at F'').

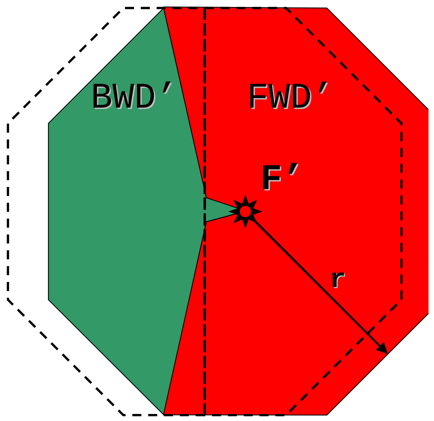
Figure 30: Evolution of BWD (green) and FWD (red) portions of coverage of a query at F as it is successively transformed to F' and then to F''. ($r = 600\ m$ and all distances are to scale.)



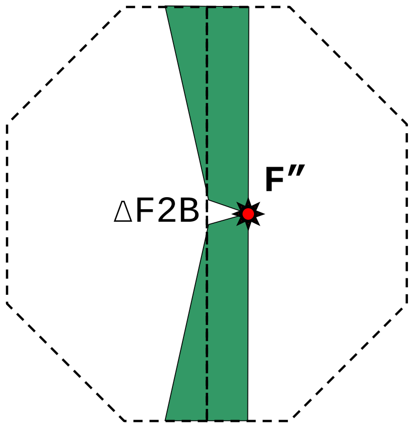
(a) Coverage at F (see Figure 30(a)



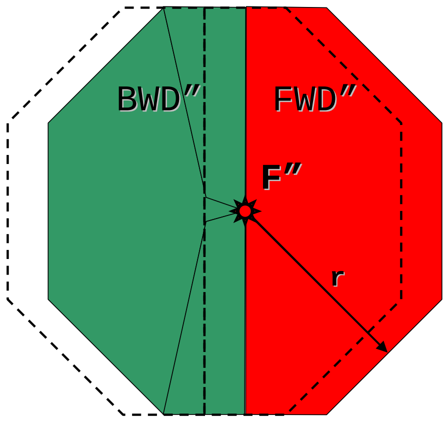
(b) Changes in coverage in a `mov` $-to-F'$ operation.



(c) Coverage after `mov` (at F'; see Fig ure 30(b)).



(d) Changes in coverage in a `jmp` $-to-F''$ operation.



(e) Coverage after `jmp` (at F''; see Figure 30(c)).

Figure 31: Schematic of evolution of BWD and FWD portions of coverage of a query at F as it is successively transformed to F' and then to F''. (All dist. on road – real coverage not octogonal. Coverage at F shown dashed on all figures for reference.)

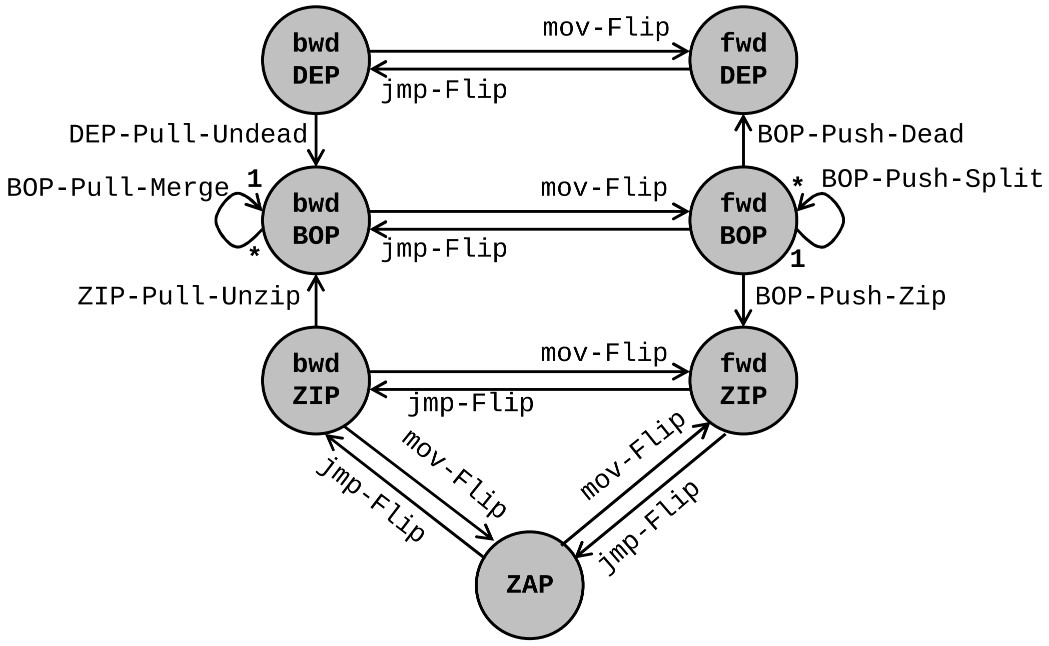
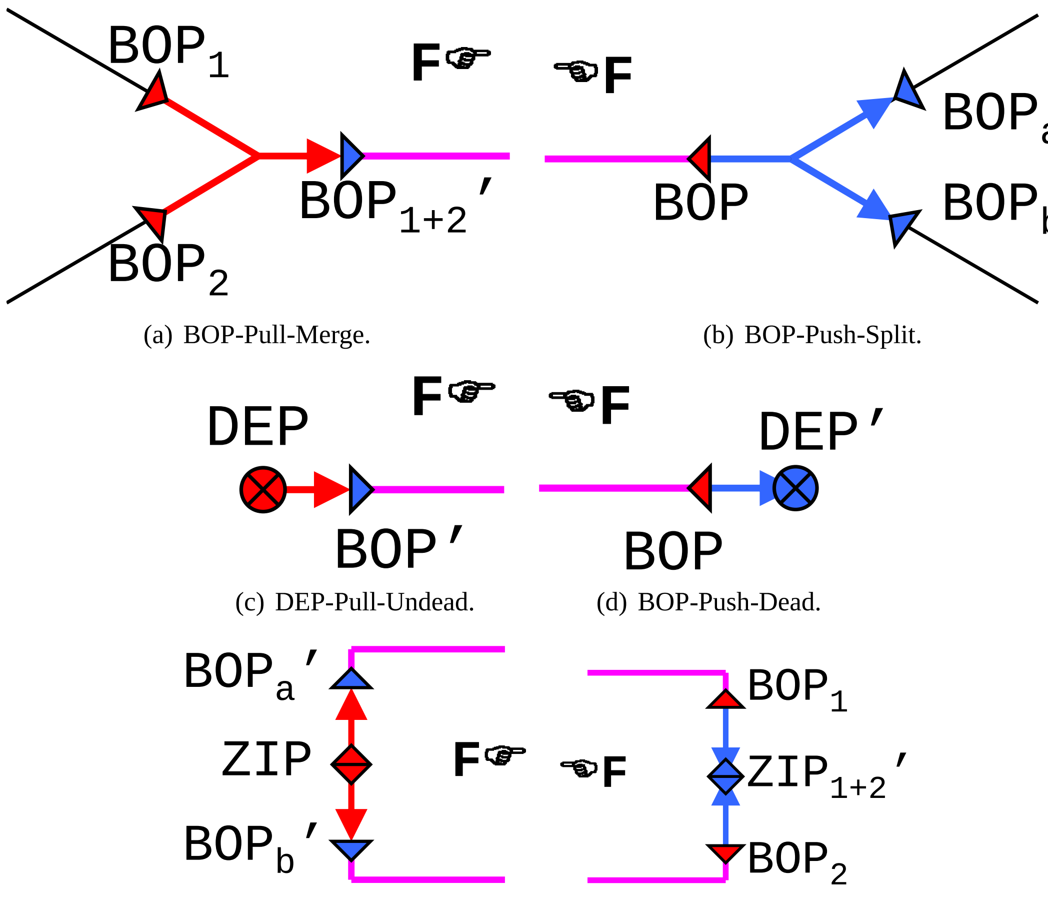


Figure 32: State diagram of the lifecycle of points in the course of mov and jmp operations. (Reversal of direction not shown.)

operations we transform it from F to new focal locations in the steps shown in Figure 32. On the forward side, a `mov` may transform a single fwd BOP into multiple fwd BOPs when the movement takes it beyond a junction. A single fwd BOP may also be transformed into a fwd DEP when reaching the end of a dead-end segment, or transformed together with another fwd BOP, into a single fwd ZIP, when pushed together on a single segment. A `jmp` transforms all fwd non-internal covers in the sidetree(s), regardless of the specific type, that are being jumped over, into bwd non-internal covers. On the backward side, the operations are symmetrical to the forward side operations but in reverse order, as summarized in Table 2. The three principal operation-pairs are illustrated on Figure 33.

Subop	Before	After	Inverse subop
BOP-Push-Split	1 BOP	* BOP	**BOP-Pull-Merge**
BOP-Push-Dead	1 BOP	1 DEP	**DEP-Pull-Undead**
BOP-Push-Zip	1 BOP	1 ZIP	**ZIP-Pull-Unzip**
BOP-Push-Merge	* BOP	1 BOP	BOP-Pull-Split
ZAP-Push-Merge	* BOP	1 BOP	ZAP-Push-Split

Table 2: Suboperations that change the number or type of points. (Frequent subops in bold. Push subops are applicable only on the FWD side, and Pull subops only on the BWD side.)



(a) BOP-Pull-Merge.

(b) BOP-Push-Split.

(c) DEP-Pull-Undead.

(d) BOP-Push-Dead.

(e) ZIP-Pull-Unzip.

(f) BOP-Push-Zip.

Figure 33: Subops.

3.7 Experimental evaluation

This section presents the experimental evaluation of the proposed Dandelion approach. We present three sets of experimental results: First, the performance comparison of the Dandelion algorithms with the conventional shortest path network expansion (NE) algorithm. Second, we performed the comparison and evaluation of the impact of different parameters on Dandelion reuse efficacy at various query ranges and re-evaluation periods. Third, we presented detailed, un-aggregated measurements on the life of an example query to gain further understanding of Dandelion algorithms and the effectiveness of Dandelion reuse.

3.7.1 Experimental Setup

All experimental results reported in this chapter were conducted using our prototype implementation of Dandelion, using Java 1.7.0 with a version 2.6 Linux kernel, on a 3.0 GHz Intel Xeon machine with 8 GB memory.

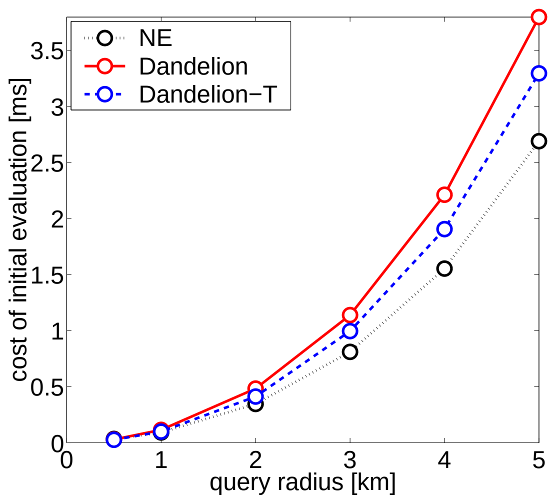
In all the experimental results reported in this section, each data point is the result of a single simulation run using multiple simultaneous, similarly parameterized continuous queries executed along different routes in the road network. We eliminate first-run effect artifacts by preceding each simulation run with a warm-up period. The mobility traces of query issuing mobile objects are generated by a random trip model of the GT MobiSim simulator [34], wherein each mobile client selects a random destination on the road network, then travels the fastest route to that destination, using speeds at or below the posted speed limits on the traveled road segments; and finally, after reaching its destination, repeats with the next randomly selected destination and starts the trip again. The movements of the individual queries are independent of each other, and the queries do not interfere with each other. The road networks used in simulations are full county maps from the US Census [43].

3.7.2 Comparing Dandelion with standard NE

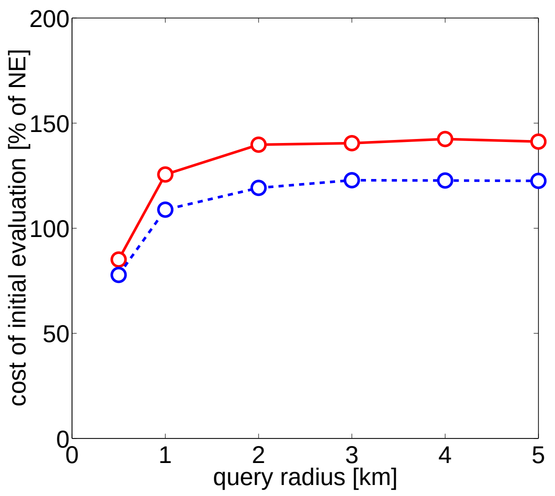
In order to make a fair comparison with standard NE, we divide the cost of Dandelion into initial query coverage computation at initial focal point F and subsequent coverage re-evaluations at subsequent focal locations. We show that although the initial coverage computation of Dandelion is relatively expensive compared to the standard NE, the subsequent coverage re-evaluations through Dandelion reuse provide significant payoffs over the one-time initial computation cost.

Figure 34(a) shows the initial evaluation computation costs for NE, Dandelion and Dandelion-T as a function of the query radius, measured in wall clock time. Additionally, Figure 34(b) plots the performance of the two Dandelion algorithms as a percentage of the NE performance over similarly parameterized queries. The initial evaluation cost is the time required to calculate the coverage data structure (a shortest path tree for Network Expansion; or a Dandelion data structure) for a newly issued query (i.e. when no reuse is possible). Since the initial evaluation of a query is a one-time event, it is only dependent on the (randomly chosen) focal location and surrounding map topology of the query issuers at the start of the simulation, and is independent of temporal proprieties, such as the re-evaluation period or the route taken and the speed of the mobile users on the traveled segments. Consequently, the initial evaluation cost is shown without reference to the irrelevant re-evaluation period. As the number of segments covered by a query, is proportional to the square of the query radius, we see a marked rise in the initial evaluation cost with an increase in query radius.

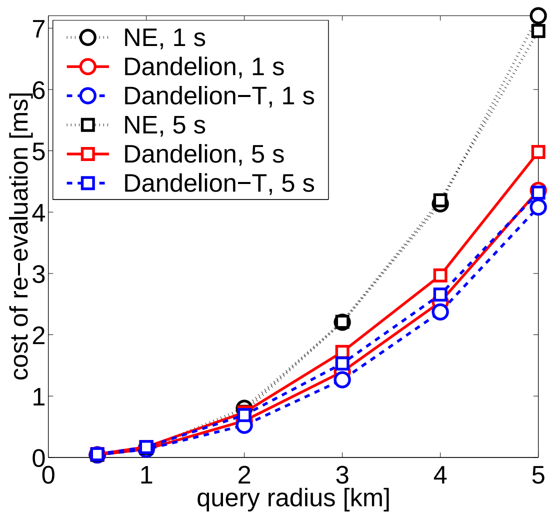
As the Dandelion data structure is more sophisticated than the simple shortest path tree, we also observe that the initial cost of building a Dandelion tree is around 40% higher than that of NE when the query radius is large. However, the extra cost pays off handsomely for subsequent re-evaluations through maximum reuse. It is also interesting to note that but the initial evaluation of Dandelion is lower than NE when the query radius is very small (500 m), as a result of the extremely simple structure of the coverage. The initial



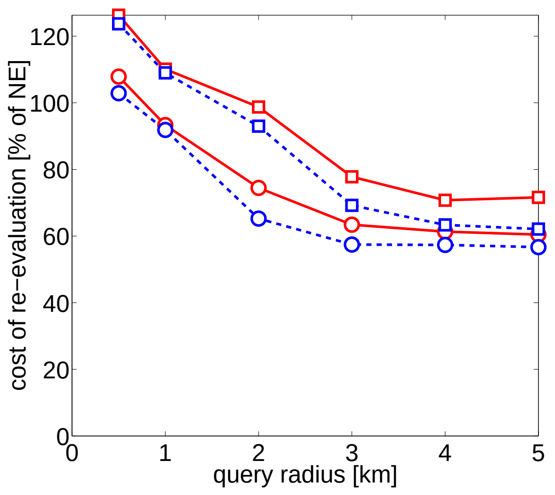
(a) Initial evaluation cost



(b) Initial evaluation cost (as percentage of NE)



(c) Reevaluation cost



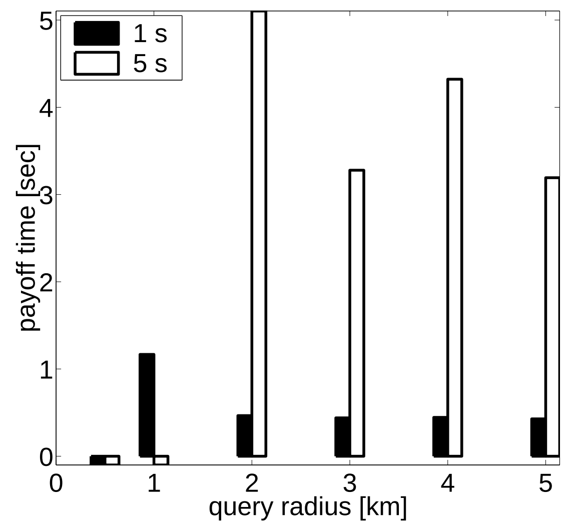
(d) Reevaluation cost (as percentage of NE)

Figure 34: Average of initial and reevaluation query calculation costs of Dandelion, with re-evaluation periods of $1\ s$ and $5\ s$.

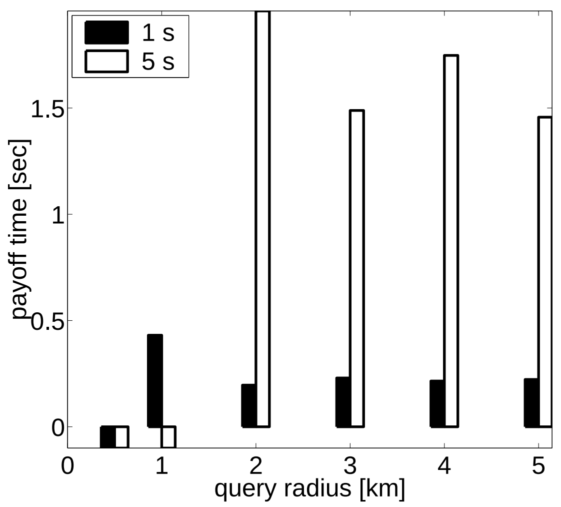
evaluation cost of Dandelion-T bears a similar but slightly better profile, as its costs are relatively lower, at around a 20–30% overhead to NE. This improvement is due to the superior perimeter set management via trident covers in Dandelion-T. The savings come from not having to maintain a separate perimeter-set hash-table and perform unnecessary search and computation on irrelevant non-internal covers.

Figure 34(c) and Figure 34(d) show the average re-evaluation costs as a function of the query radius. In both figures, Dandelion and Dandelion-T are compared to NE in terms of cost of re-evaluation in seconds and in percentage of NE performance respectively. The average re-evaluation cost at the current focal location F is the average time required to calculate the coverage data structure, given that the data structure is already available from the previous, nearby focal location. As the NE approach is unable to reuse a previously computed shortest path tree, it is at a severe disadvantage to Dandelion, which is designed to maximize reuse. Note that the re-evaluation cost for NE is independent of the re-evaluation period. The reason that re-evaluation cost in Figure 34(c) is not directly comparable to the initial evaluation cost shown in Figure 34(a) is the following: as the re-evaluation costs are averaged over not only all queries, but the entire lifetime of all queries, and is thus dependent on the network topology surrounding the trajectories of the mobile users who issued queries. In comparison, Dandelion maintains the core FWD'_0 portion of the expansion tree unchanged, and only calculates the BWD subtree and a portion of the FWD tree in each step. As a result, the re-evaluation cost of Dandelion is decreasing as a function of the query range, since larger queries provide an opportunity for reuse of not only more segments but also a larger proportion of the total number of segments in the coverage. When the query radius is not too small (more than 500–1000 m), re-evaluation using a Dandelion tree is faster than using NE, providing savings in the 20–40% range over NE for large queries in each re-evaluation.

Cost Analysis of NE and Dandelion. We would like to note that comparing to NE, although Dandelion has higher overhead for the initial evaluation, it is a one-time cost



(a) Dandelion breakeven query runtimes against NE



(b) Dandelion-T breakeven query runtimes against NE

Figure 35: Breakeven query runtime of Dandelion algorithms against NE, with re-evaluation periods of $1\ s$ and $5\ s$.

and the saving from subsequent re-evaluation is recurring each time when the query is re-evaluated. Thus the overall payoff is significant, especially for long running continuous network range queries with reasonably sized radius. This observation is confirmed by the relative performance of Dandelion at 1 sec and 5 sec re-evaluation periods in Figure 34: higher re-evaluation frequency decreases the computational cost by around 10–20% and larger query radius reduces the computational cost up to 40–50%. These factors indicate that Dandelion is especially well suited for large road network queries that must be continuously re-evaluated in near real-time, at a high frequency. Additionally, by incorporating Trident covers in the perimeter-set management, Dandelion-T offers additional performance improvement over the basic Dandelion algorithm.

Figure 35 provides the experimental results on the breakeven query runtime of Dandelion against NE. Before we illustrate the plots in Figure 35, we first provide a brief analysis of the cost comparison between Dandelion and NE.

Let dt denote a re-evaluation period. Dandelion's higher t^D_{init} initial cost and lower t^D_{re} re-evaluation cost lead to a total computation cost of $C^D = t^D_{init} + t^D_{re} \cdot \frac{t}{dt}$ after t seconds of query time. Similarly, NE's lower t_{init}^{NE} initial cost and higher t_{re}^{NE} re-evaluation cost lead to a total computation cost of $C^{NE} = t_{init}^{NE} + t_{re}^{NE} \cdot \frac{t}{dt}$ after t seconds of query time. Therefore, given a road network map and a constant query radius, there exists a break-even time t_b which is when $C^D = C^{NE}$. If a query runs shorter than t_b, then $C^D > C^{NE}$, and therefore it is cheaper to simply re-evaluate the coverage every time from scratch using NE. However, if a query runs longer than t_b, then $C^D < C^{NE}$, and therefore using Dandelion pays off. The t_b break-even time is derived as:

$$C^D = C^{NE}$$

$$t_{init}^{D} + t_{re}^{D} \cdot \frac{t_b}{dt} = t_{init}^{NE} + t_{re}^{NE} \cdot \frac{t_b}{dt}$$

$$t_b = dt \cdot \frac{t_{init}^{D} - t_{init}^{NE}}{t_{re}^{NE} - t_{re}^{D}}$$

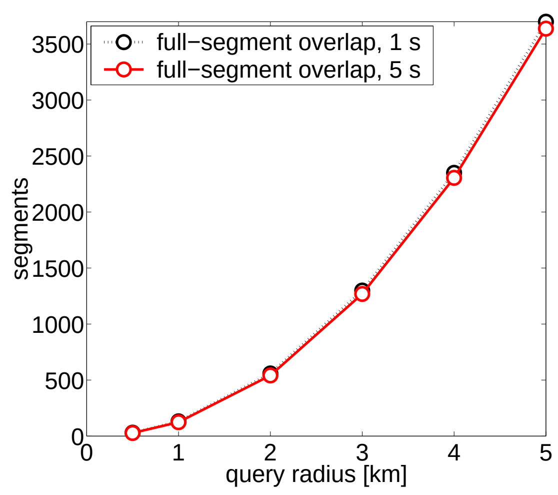
We conduct a set of experiments to measure such break-even time for the Dandelion algorithm against NE by varying query radius from 500 meters to 5000 kilometers. This set of experiments helps answer the question such as ”How long should a query run, so that the it becomes cheaper overall to run Dandelion than NE?” or in other words, ”After how many seconds does the investment of the Dandelion reuse data structures and algorithms in the higher initial computation pay off (due to the much lower subsequent incremental coverage computation cost)?”. Figure 35(a) shows the payoff time in seconds for the Dandelion algorithm comparing with NE.

First, we observe that for very small queries (500 m radius), the higher initial cost of Dandelion *does not* pay off, as the re-evaluation cost is also relatively higher for Dandelion compared with NE. The experimental result matches our analysis that when queries have small radius, the number of segments covered is so few that simply recomputing the shortest path tree from scratch is inexpensive in comparison to building the Dandelion reuse structures. Thus, using NE is recommended. However, as queries get larger (with 1 km or higher radius), running Dandelion does in fact become cheaper after some time, but this primarily depends on the re-evaluation period. With re-evaluation at every second, after only around 1 sec of query runtime (i.e., after a single re-evaluation following the initial evaluation), Dandelion already breaks even. When re-evaluations are performed less frequently (e.g., every 5 sec), the re-evaluation is more costly due to the higher displacement, thus for the queries with radius of 1000 m moving along the same spatial region of the same map as the ones we used in the experiments, NE wins the competition. However, with a larger query radius of 2 km or higher using re-evaluation frequency of 5 sec, the break-even time is around 5 sec, which again indicates that Dandelion pays off after a single re-evaluation following the initial evaluation. As queries get larger, the break-even time drops below the periodic re-evaluation frequency of dt. From Figure 35(a), we also observe that the payoff time increases slightly when the radius is increased from 3 km to 4 km and then drops back to around 3 seconds when the radius rises to 5 km. These ups and downs are due to the routes on the map the query issuers took given the random trip model used in the simulation to generate queries and mobility traces of mobile users. In summary, Dandelion generally outperforms NE for queries with larger radius and higher frequency of evaluations.

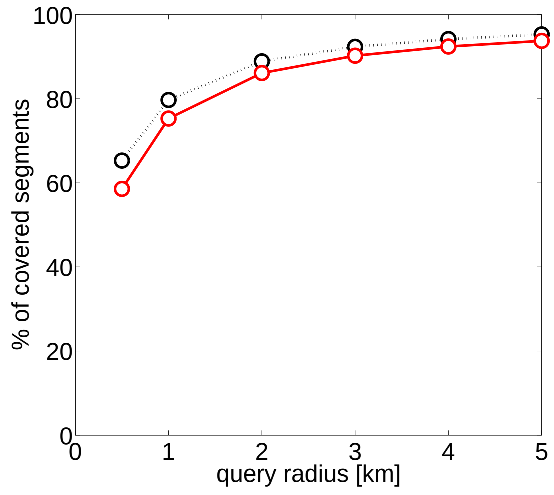
Figure 35(b) shows the break-even time for the Dandelion-T algorithm against NE. We note that for Dandelion-T, the payoff time t_b (see y-axis) is much lower than Dandelion basic algorithm, around $\frac{1}{3}dt$, indicating that there is substantial margin of safety in performance when using Dandelion-T compared to NE, and that Dandelion comfortably outperforms NE with non-trivial query radius and long running continuous range queries on road networks.

3.7.3 Impact of Different Parameters on Reuse Efficiency

In this set of experiments, we investigate the effect of two factors on query performance. First, we vary the query radius from 500 m to 5 km. Second, we vary the re-evaluation period from 1 sec to 5 sec, and preform query re-evaluations for a simulated 10 minutes.



(a) Overlapping fully covered segments



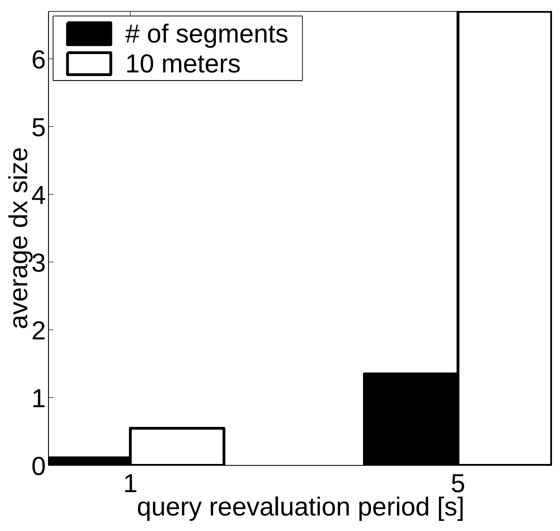
(b) Overlapping fully covered segments (as percentage of covered segments)

Figure 36: Average number of overlapping segments across coverages at consecutive re-evaluation F locations, with re-evaluation periods of $1\ s$ and $5\ s$.

Note that both the re-evaluation period and the query lifetime are measured with the internal clock of the simulated world

Figure 36(a) shows the potential for reuse, by plotting the number of fully-covered (i.e. internal) segments, averaged over all consecutive re-evaluation focal locations F and F'. With an increase in query size, the overlap, and thus the potential for cover reuse grows quadratically. Additionally, Figure 36(b) plots the overlapping segment count as a proportion of the total number of covered segments. While small queries offer less overlap between two re-evaluation locations, the ratio of the overlap to the entire coverage increases above 90–95% as query radii grow. Comparing the overlap given two re-evaluation periods, we observe that a higher frequency of re-evaluations also increases the ratio of the overlap, due to smaller query focal location displacement.

We show the average dx displacement on Figure 37(a), both in terms of number of segments, and in terms of meters (actual road network distance between F and F'). With a high re-evaluation period (1 sec), most re-evaluations take place on the same segment ($dx < 1$ segment), while with a lower period (5 sec), most re-evaluations take place on connected segments ($dx > 1$ segment). The average displacement is less than 10 meters



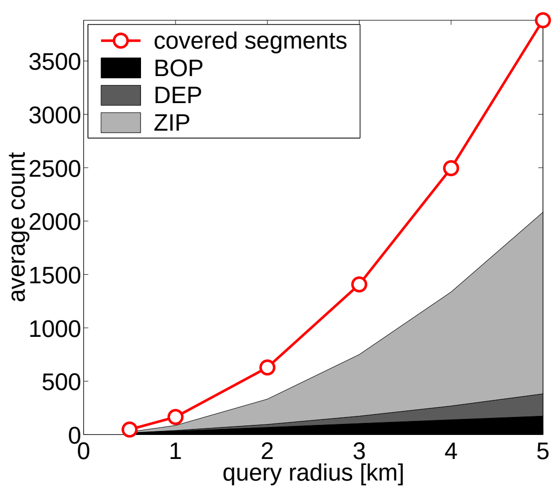
(a) Re-evaluation period in distance units.

Figure 37: Re-evaluation displacement.

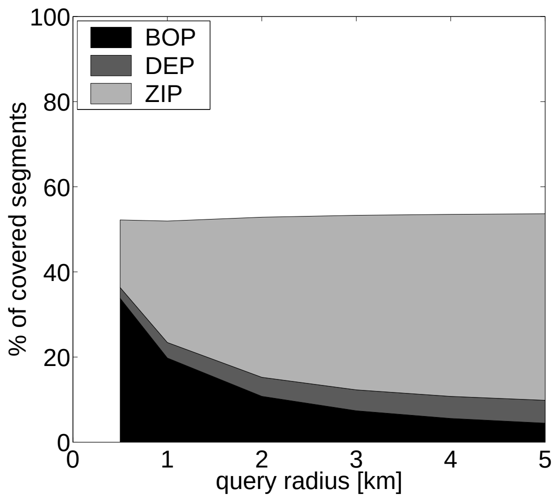
for the faster, and more than 60 meters for the slower re-evaluation.

Figure 38(a) and Figure 38(b) show the measurement of the number of Dandelion cover types and a percentage of the total coverage in a stack on style, respectively. While the need to manage these key road network locations may seem tedious, their total is less than 60% of the count of covered segments. With small query radii, the proportion of BOPs dominates, but as the query radius increases, BOPs make for around 5% of the total coverage (as they are only present in the outer perimeter of the query, whose length is proportional to the query radius, while the size of the coverage is proportional to the square of the query radius). Furthermore, the large number of ZIPs (and also DEPs) are immovable when reused, leading to further performance gains over NE.

Figure 39 evaluates the efficiency of reuse by measuring the average number of covered (reused) segments and the percentage of covered segments for queries with varying radius. Covers are classified into three categories on each re-evaluation: New covers signify segment covers that have been created during the re-evaluation step. Reused, but updated covers are those that were created during a previous re-evaluation, but have been modified in the current step (for example, reused BOPs would fall in this category). Reused and not

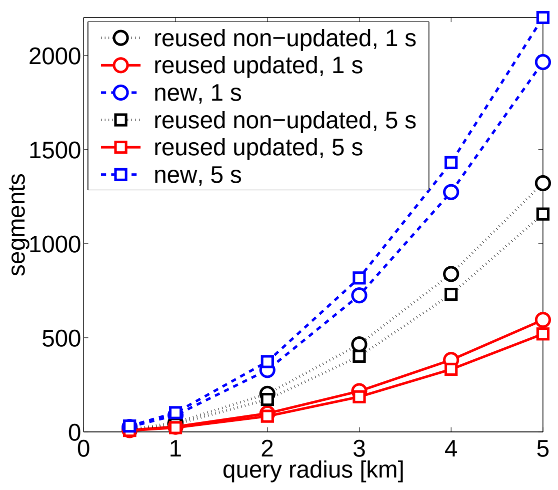


(a) Cover types

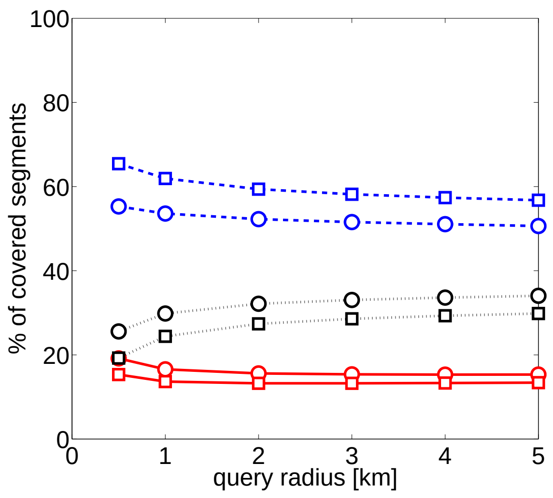


(b) Cover types (as percentage of covered segments)

Figure 38: Cover types in Dandelion coverage.

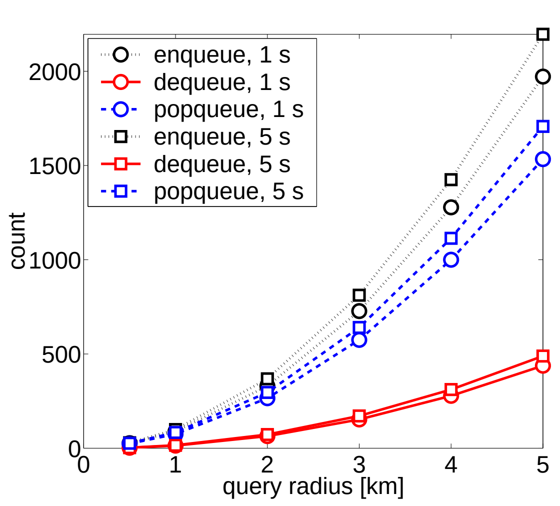


(a) Reuse of segments

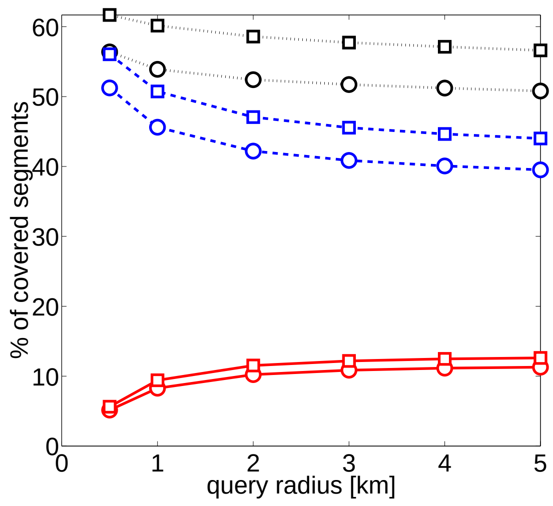


(b) Reuse of segments (as percentage of covered segments)

Figure 39: Average number of segments reused, with re-evaluation periods of $1\ s$ and $5\ s$.



(a) Queue operations



(b) Queue operations (as percentage of covered segments)

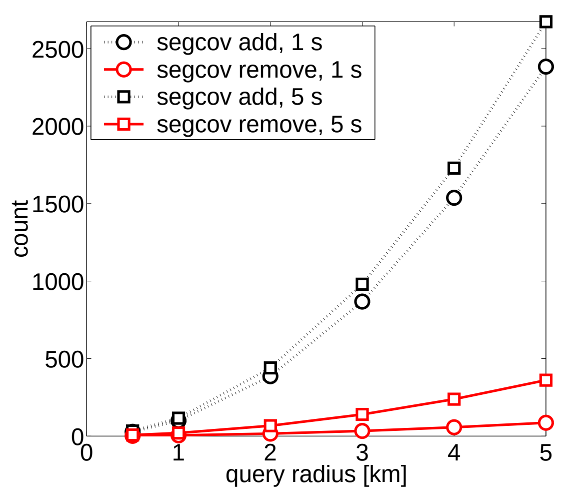
Figure 40: Average number of queue operations performed, with re-evaluation periods of $1\ s$ and $5\ s$.

updated covers are those that are present in a current re-evaluation step in exactly the same form as in the previous step (e.g. internal covers in FWD$'_0$). Figure 39(a) shows all three types of covers measured at re-evaluation frequency of 1 sec and 5 sec for varying radius. The number of reused and non-updated covers is the highest for both frequencies. The reused and updated covers are the lowest in comparison for both 1 sec and 5 sec frequency, and the number of new covered segments inserted to the coverage are in the middle. Figure 39(b) shows the percentage of the covered segments over the total coverage. We would like to note that after an initial rise in the percentage of covered segments, the ratio of the three reuse types over the total number of covered segments stabilizes as query radius increases, showing an excellent scalability of Dandelion reuse.

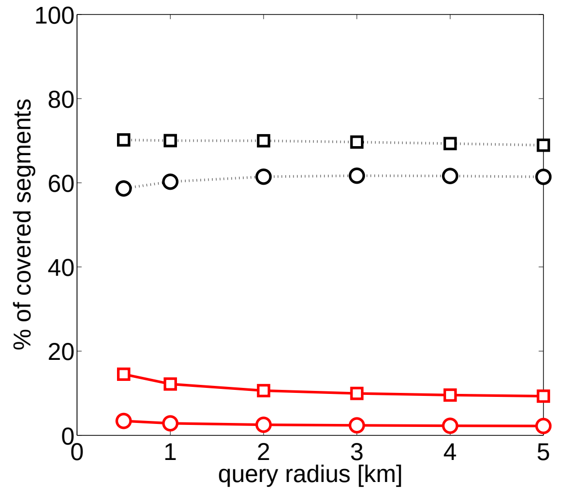
3.7.4 Effectiveness of Data Structures

In this section we evaluate the key data structures used in Dandelion and how effective they are with respect to reuse scalability.

Recall Section 3.4.2, where we have discussed the ordered priority queue and three types of queue operations (enqueue, dequeue and popqueue) for computing the coverage



(a) SegCovMap updates

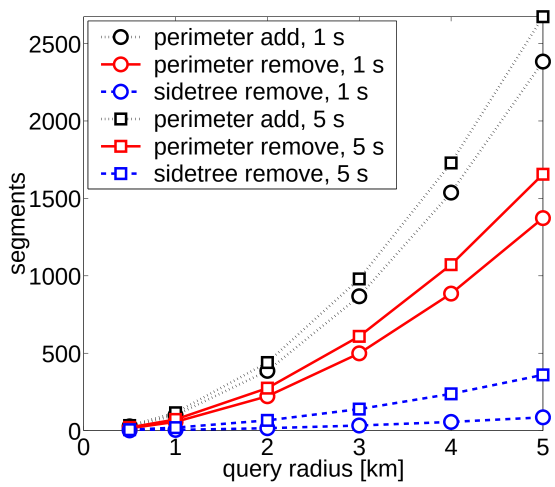


(b) SegCovMap updates (as percentage of covered segments)

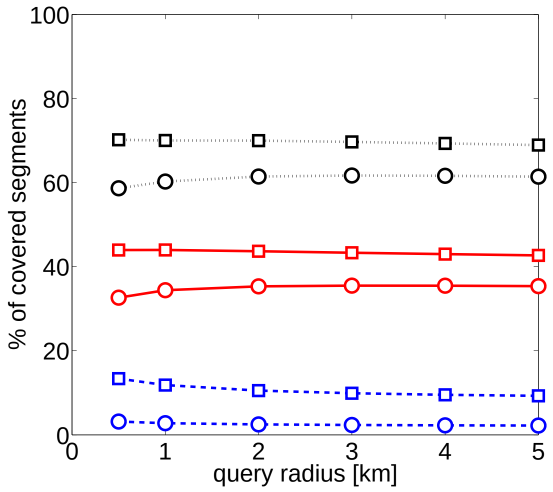
Figure 41: Average number of SegCovMap updates, with re-evaluation periods of $1\ s$ and $5\ s$.

initially and incrementally. Figure 40 measures the average number of queue operations performed at different evaluation intervals. The number of queue operations performed is shown in Figure 40(a) and the percentage of queue operations over the total coverage is given in Figure 40(b). From both figures we observe that enqueue operations are slightly lower than the number of popqueue operations, and dequeue operations are the lowest for varying radius and re-evaluation frequency, a good indicator of why Dandelion reuse is effective and profitable.

We now evaluate the effect of SegCovMap on Dandelion reuse efficiency. The Seg-CovMap is served as an inverted index of segments to the covers on them, and is updated mostly by adding new covers for all but the entry segment at a junction during a BOP-Push-Split operation. Figure 41 measures the average number of updates to SegCovMap at different re-evaluation frequencies. Fig 41(a) measures the total count of SegCovMap updates and Fig 41(b)) shows the percentage of SegCovMap updates over the total number of covered segments. However, SegCovMap removals also occur, as segments in the portion of FWD, which are not present in FWD$'_0$, must be removed one by one. For this experiment, we use the implementation of Dandelion basic algorithm that maximizes reuse



(a) Perimeter updates

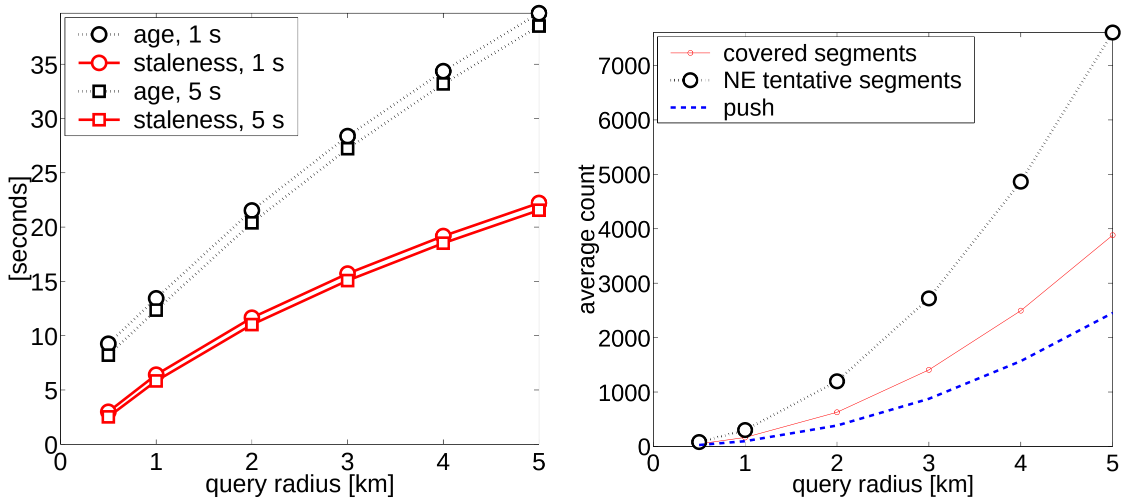


(b) Perimeter updates (as percentage of covered segments)

Figure 42: Perimeter updates and side-tree SegCovMap removal operations, with re-evaluation periods of $1\ s$ and $5\ s$.

of FWD. Thus the SegCovMap for BWD is wholesale discarded, eliminating the need for any SegCovMap removal updates. That is why both figures show that the removal of covers in the bypassed side-trees is manageably low at around 15% when the re-evaluation period is high (5 sec), and almost negligible at around 5% when the re-evaluation period is low (1 sec).

In the next set of experiments we evaluate the effectiveness of the perimeter set. Recall that for Dandelion, we keep a separate perimeter set for FWD and BWD, which is hashed for quick containment checking and beneficial for performance of the Dandelion, as FWD and BWD are always examined separately. However, such a separate perimeter set is obviated in Dandelion-T due to the integral update of a correctly ordered perimeter-list during all push and pull operations. Figure 42(a) shows the number of segments that have performed perimeter updates and side-tree SegCovMap removal operations, and Figure 42(b) shows the same measurement result as a percentage of the total coverage. In both figures, we break out the aforementioned SegCovMap removal costs into perimeter set removals and side-tree removals. From the measurements we observe that perimeter set removals are the smallest in comparison to side-tree removals and perimeter set add



(a) Age and staleness (b) Total operations.

Figure 43: Average age and staleness of covers, with re-evaluation periods of $1\ s$ and $5\ s$.

operations, which is the highest of the three types of operations measured.

3.7.5 Staleness of Covers and Maps on Reuse Efficacy

In this section we measure some additional parameters, to gain additional insight into the effectiveness of Dandelion reuse. First, we want to measure the age of a cover, which is defined by the time elapsed since its creation, regardless of any subsequent updates to the cover (such as pushing and distance changes for a BOP). Second, we want to measure the staleness of a cover, which is the time elapsed since its last update of any kind, and is thus an even stricter measure of how long covers live. Both measures can shed some light on how effective the Dandelion coverage reuse is in subsequent re-evaluations. Intuitively, higher age and higher staleness both indicate that the reuse is effective. For example, at 5 km radius, the average life of a cover is 35–40 seconds, and has gone for 20–25 seconds without any update. Note that for NE, both age and staleness are always 0, as the coverage is always entirely recomputed (even if e.g. the mobile object is not moving).

Figure 43(a) shows the age and the staleness of covers with varying re-evaluation frequencies. This experimental result shows that both age and staleness of covers are increasing as the query radius increases, though neither of them is sensitive to the frequency of

Table 3: Road networks used in experiments

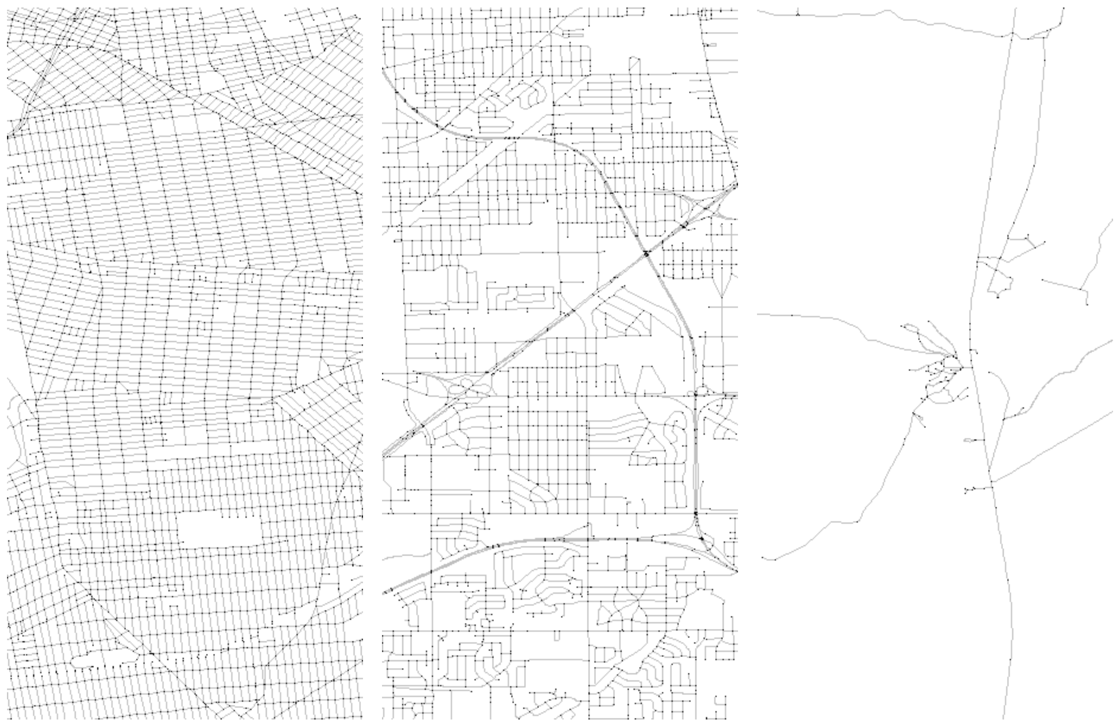
Style	County location	Total length	Segments	Junctions	Avg. segment length	Junction degree
urban	Kings, NY	3 011 km (62 h)	21 954	13 003	137.2 m (10.1 sec)	mean: 3.4, max: 8
suburban	Cook, IL	26 022 km (524 h)	213 306	165 061	122.0 m (8.8 sec)	mean: 2.6, max: 9
rural	Coconino, AZ	40 437 km (819 h)	91 346	81 396	442.7 m (32.3 sec)	mean: 2.2, max: 6

re-evaluation. However, the increasing gap between the age curves and the staleness curves shows the effectiveness of cover reuse since many covers are long lived. Figure 43(b) shows the result of comparing covered segments in Dandelion with NE traversed segments and the number of push operations performed. It shows that comparing to the total number of segments traversed in NE, the total number of covered segments is smaller and increases slower as the radius increases. Furthermore the total number of push operations is the smallest of all three and grows much slower as the query radius increases. This set of experiments demonstrates again that Dandelion coverage reuse is highly effective as it uses fewer covered segments and fewer push operations, compared to the number of traversed segments in NE.

In the next set of experiments, we compare the Dandelion algorithm against the baseline Network Expansion algorithm on three maps (Table 3): Cook county, IL (Chicago area, Fig. 44(b)) is a suburban city map, with residential areas and dead-end streets or cul-de-sacs. Kings county, NY (Brooklyn area, Fig. 44(a)) is a built-up city map, with a dense, regular grid structure, short streets, and most intersections with four connecting streets. Coconino county, AZ (Fig. 44(c)) is a rural map, with long highways passing across a desert region, with an occasional small town.

From Figure 44 we can observe that the three maps represent three different scales of the road network topologies (urban, suburban and rural) in terms of geometry and spatial density.

Figure 45 shows the initial evaluation cost and re-evaluation cost as function of the query radius. Although the computational cost is heavily dependent on the topology of the map (especially striking for the rural map), and the initial evaluation cost for all three maps are more expensive than NE, as shown in Figure 45(a). Figure 46(b) shows that



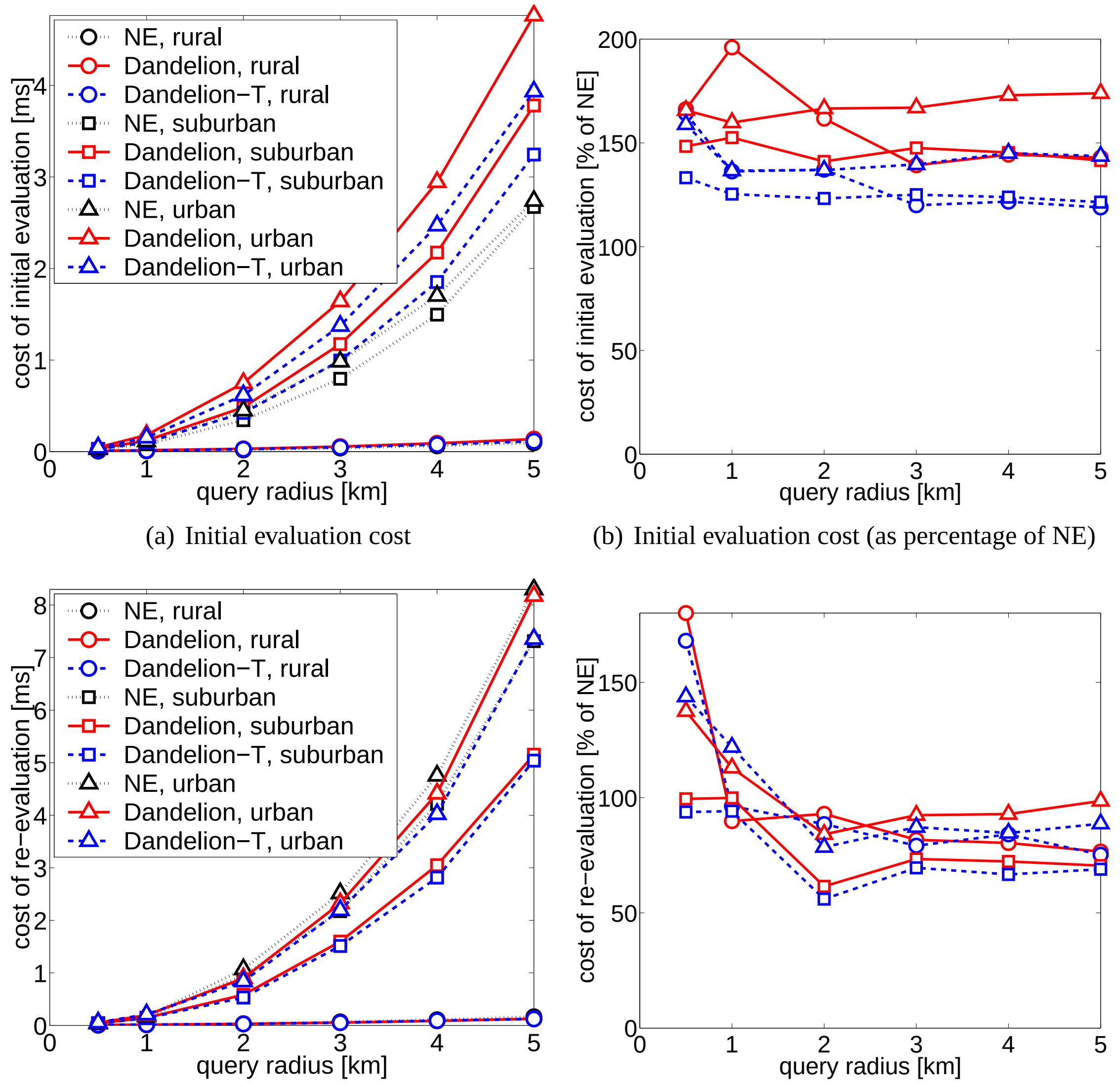
(a) Kings, NY (b) Cook, IL (c) Coconino, AZ

Figure 44: Typical map sections (same scale).

both Dandelion and Dandelion-T incur smaller cost of re-evaluation when the query radius increases above 1 km and the cost reduction is more significant when the radius gets larger. Also Dandelion-T consistently outperforms Dandelion basic algorithm in all three types of road networks. Figure 45(b) and Figure 46(d) show the cost of re-evaluation as a percentage of the performance of the NE algorithm for initial evaluation and reuse based re-evaluations respectively. Clearly, both Dandelion and Dandelion-T pay more in the initial evaluation cost and pay less in the re-evaluation cost. In summary, Dandelion algorithms in general outperform NE with high costs at low query radii, and lower and stable costs as queries increase in size.

3.7.6 Life of a query

Our experimental results presented so far have been averaged over simulations of many mobile users and many continuous network range queries of similar temporal and spatial features. In this section we present four sets of experimental results about the life of a



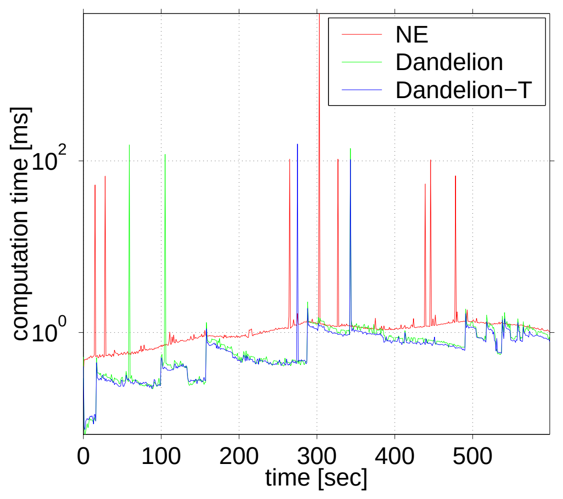
(a) Initial evaluation cost

(b) Initial evaluation cost (as percentage of NE)

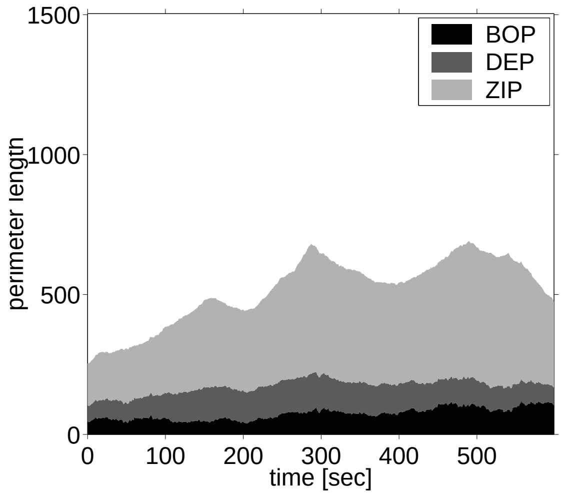
(c) Reevaluation cost

(d) Reevaluation cost (as percentage of NE)

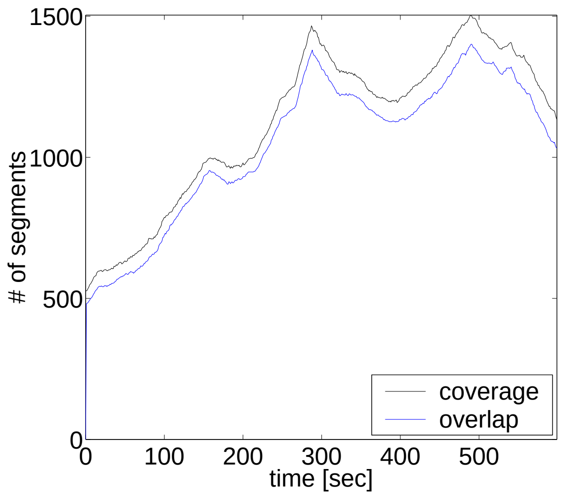
Figure 45: Average of initial and reevaluation query calculation costs of Dandelion, on three maps.



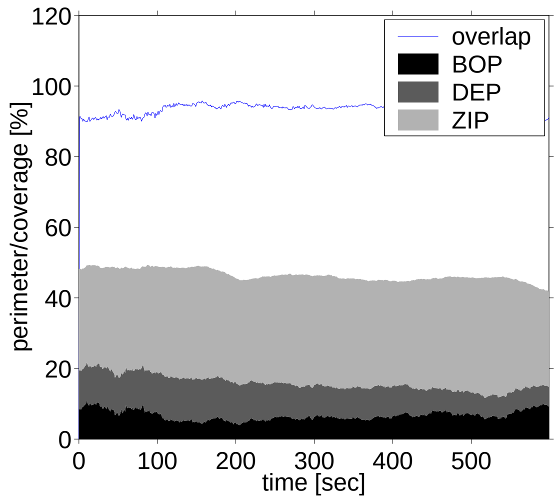
(a) Computational cost of algorithms.



(b) Dandelion cover types



(c) Total coverage and overlap.



(d) Dandelion cover types and overlap (as percentage of covered segments)

Figure 46: Computational cost, overlap and covers.

single example query, to gain insight into the un-averaged performance of the Dandelion algorithms against the standard NE algorithm. We follow the life of a continuous road-network range query with 5 km radius, which is re-evaluated every second, for a total simulated time of 10 minutes. The mobile user who issued the query is traveling along a pre-selected long route in the suburban Chicago area, at speeds within the posted speed limits on respective road segments.

Coverage Computation Cost.

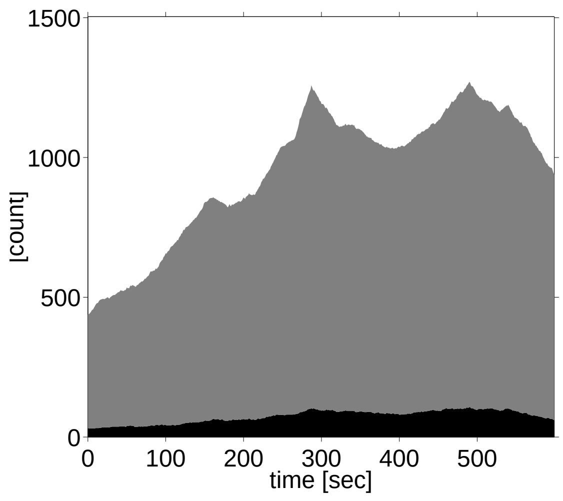
Figure 46(a) shows the computational cost of evaluating the query coverage of Dandelion, Dandelion-T and standard NE. First, we note that the cost of evaluating the coverage using NE tracks the size of the coverage, and is thus dependent on the location of the query focal object, but not the temporal properties of the query, resulting in a smooth cost-curve on the top in red (colored red) – which is higher than the cost-curves of the Dandelion (green curve) and Dandelion-T (blue curve). The cost-curves of both Dandelion algorithms show that the ability to reuse portions of the previously computed query coverage tree can indeed reduce the computation required to calculate a coverage at a next location. However, we note several singularities in the Dandelion cost-curves, where the cost of re-evaluation exhibits a sudden jump from one focal location to the next, and falling sharply from this local maximum in the subsequent re-evaluations. These cost spikes are due to the topology of the road network, and the trajectory that the user takes in it, and represent situations, where FWD'_0 is too small to be beneficial. We note that the spikes are not due to stop-and-reverse mobility characteristics, as in such a case the data for the BWD and FWD halves are swapped (i.e., BWD becomes FWD), and thus normal reuse is possible and maximized.

Figure 46(c) shows the total number of segments covered (completely or partially) by this query. Note that the higher curve represents the cover count update and the lower curve represents the overlapping segment count update, both in the life time of the query in 600 seconds, as the query focal object is moving on the road network. At the beginning of the trip, the coverage is around 500 segments initially, and it rises up to 3 times at the peak. This variation in the extent of the coverage is solely a function of the query focal location and surrounding map topology at each re-evaluation, and is entirely independent of both the query processing method used to calculate the shortest path coverage and the temporal properties of the query re-evaluation (such as re-evaluation frequency, user travel speed). The lower curve in this figure shows the number of segments overlapping between two consecutive re-evaluation locations, initially at 0 (when the query is issued). Interestingly, the overlapping segments closely follow the curve of the extent of the coverage in the entire duration of the simulation. This intuitively verifies the observation that as the coverage increases, the overlapping cover occurances are increased as well. We would like to note that unlike the coverage count, the overlapping segment count does depend on the temporal properties of the query, such as the re-evaluation frequency, which can in turn depend on the travel speed of the mobile user. However, we conjecture that with slightly lower frequency of re-evaluation, the curve shape will remain approximating the curve of the coverage update but the overlapping curve at lower frequency will be lower with a bigger gap to the coverage curve, compared to the overlapping curve at high frequency of every one second.

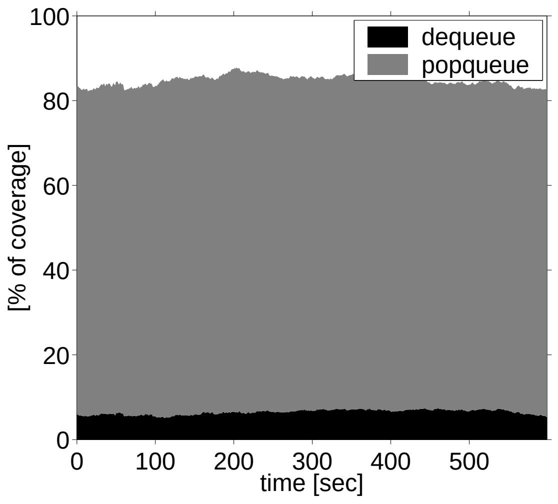
Figure 46(b) measure the perimeter size of the three cover types (BOP, DEP, ZIP) present in a Dandelion tree (stacked on) and Figure 46(d) shows the ratio of the perimeter size over the total coverage size at each re-evaluation location. The maximum potential for reuse is given by the overlap (the number of completely covered segments), which is around 90% throughout the lifetime of this query, mostly independent of the changes in the size of the coverage along the route (but dependent on the temporal properties of the query). The total number of all non-internal covers is around 50% for this query. Furthermore, the majority of the non-internal covers are DEP and ZIP types, giving potential for further optimizations, as these points are generally immobile; only BOP covers (which only make up 5–10% of the number of covered segments) will conceivably be moved in the subsequent re-evaluation for maximum reuse.

Queue Operations

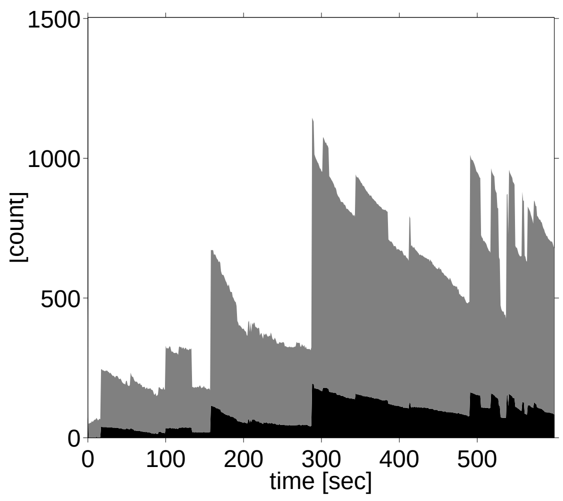
Fig. 47(a) and Fig. 47(b) show the number of queue operations for the NE algorithm, stacked on for dequeue and popqueue. The NE algorithm pops the front of the queue around 70%, and dequeues around 5% of the total coverage at any location, regardless of the absolute number of queue operations tracking the size of coverage as it changes with the movement of the focal location on the network. The dequeue operations occur when a shorter path via a new node is found to a node that was enqueued with a longer tentative



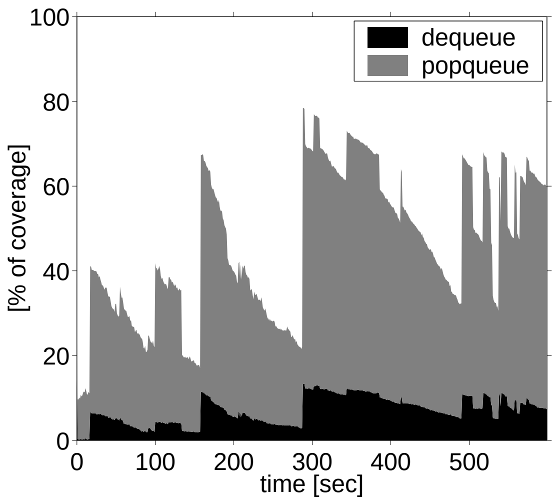
(a) Total NE queue operations.



(b) Total NE queue operations (as percentage of covered segments)

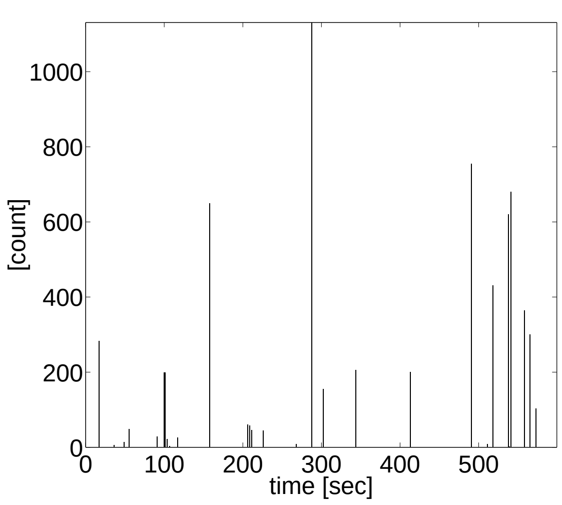


(c) Total Dandelion queue operations.

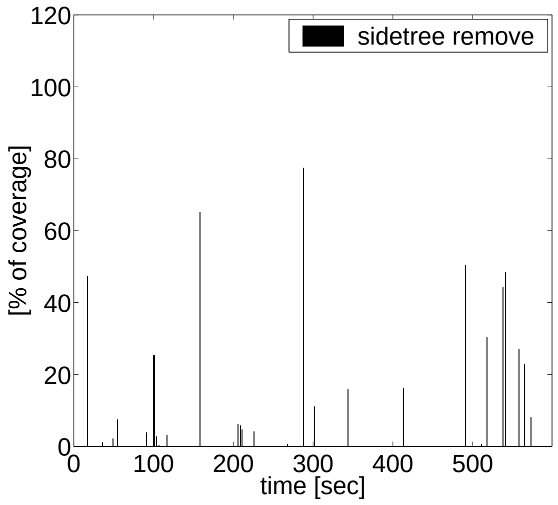


(d) Total Dandelion queue operations (as percentage of covered segments)

Figure 47: Queue operations (NE compared with Dandelion).



(a) Side-tree cover removals.



(b) Side-tree cover removals (as percentage of covered segments)

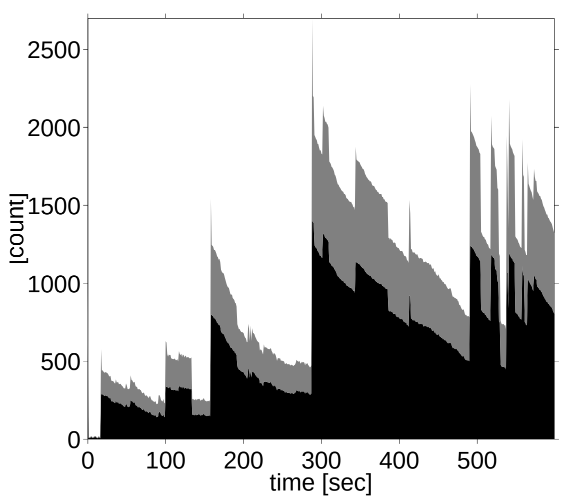
Figure 48: Side-tree SegCovMap removal operations for Dandelion.

distance. We note that 15–20% of the covered segments are never enqueued, as no further expansion is possible from them.

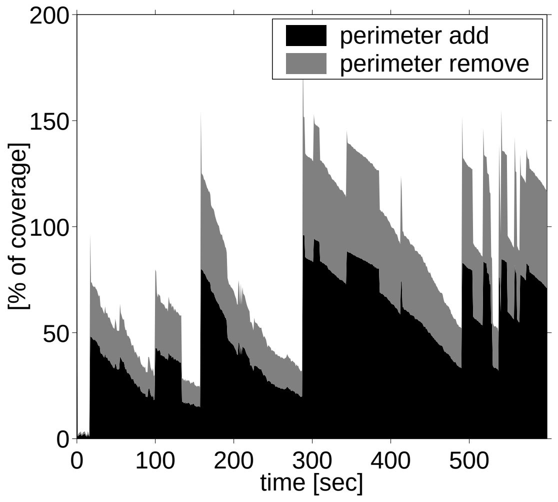
Fig 47(c) and Fig. 47(d) show that the queue operation profile for Dandelion is noticeably different compared to Fig. 47(a) and Fig. 47(b). This highlights the computational cost improvement shown previously in Fig. 46(a). First, we observe that the total number of dequeue and enqueue operations can be as low as only 10% of the total number of covered segments, indicating massive tree reusability. Second, the cost-spikes for Dandelion algorithms correspond to spikes in both the number of enqueued and dequeued segments.

Impact of Reuse Data Structures.

The entire BWD tree (SegCovMap and perimeter set) can be discarded in a single step, as the two halves of the query are maintained in separate data structures. However, the portion of the FWD tree (ΔF2F) that has been bypassed, and which was forward of F, but is no longer forward at the subsequent F' location, needs to be traversed and its covers removed one by one from the SegCovMap. We show the total number of such side-tree cover removals on Figure 48. While such side-tree removals are costly (e.g. having to remove around 80% of the covered segments in the worst case here), they are also rare.



(a) Perimeter updates



(b) Perimeter updates (as percentage of covered segments)

Figure 49: Perimeter updates (basic Dandelion only).

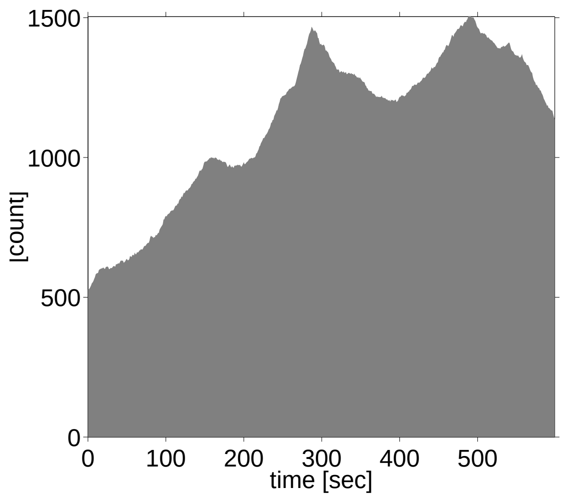
Figure 48(b) sheds light on the origin of the performance deterioration spikes in the previous figures: As the focal object crosses an intersection, side-roads at that intersection may be the roots of an extensive side-tree, covering e.g. 80% of the total number of segments in the coverage. As the user bypasses such a side-tree, a large portion of the total coverage ceases to be in FWD (necessitating a costly removal of the many segments in the side-tree from SegCovMap), and is subsequently to be found in BWD (necessitating the recomputation of the Dandelion tree covering these segments, as the BWD tree needs to be discarded between re-evaluations).

Furthermore, for the basic Dandelion algorithm, we maintain the set of non-internal covers (the perimeter set), so that we can initialize the queue with this set, enabling full reuse of all internal covers in FWD'_0. The number of add and remove operations performed on the perimeter set are shown on Figure 49. The ordinary perimeter update operations take place in the course of a DEP-Push-Split suboperation, with the removal of a single BOP cover from the perimeter set (as it becomes an internal cover), and the addition of one BOP cover for each of the newly partially covered segments. However, Fig. 49(b) also exhibits the spikes (corresponding to side-tree bypassing) seen in previous figures, with the aggregate of cover additions and removals both at 80% in one case (corresponding to the largest side-tree bypass on Fig. 48(b)). We conclude that the perimeter set update operations place additional computational costs on Dandelion, at exactly the same time when the SegCovMap removal costs jump, and when the size of FWD drops (and thus the reusable FWD_0' size drops), and the size (and thus cost) of re-calculating BWD jumps. All these effects are due to the same root cause of bypassing a large FWD side-tree, which then on is part of BWD. We note that as Dandelion-T is designed to automatically keep the perimeter in order without any additional computations, the entire perimeter set update cost is saved, resulting in a performance improvement over basic Dandelion in exactly the aforementioned critical high-cost situations.

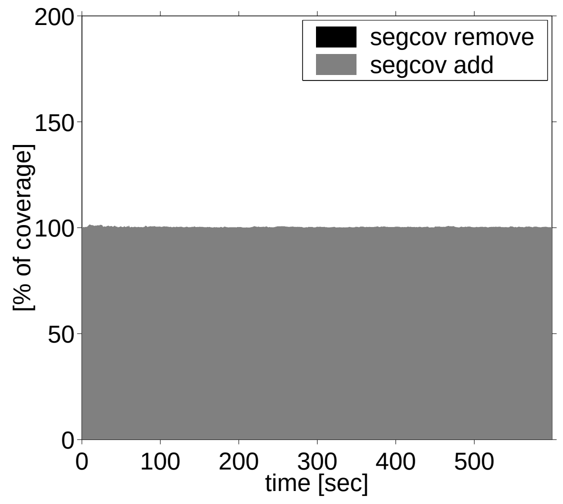
The update of the SegCovMap only involves additions (never removals) in the case of NE (Fig. 50(a)), as covers are never individually removed. Rather, the entire SegCovMap is wholesale discarded and entirely recomputed at each step, resulting in all covered segments being added to SegCovMap. As some segments contain 2 BOPs (from the direction of the two endpoints, but not joining as ZIPs), the total number of SegCovMap add operations is marginally higher than 100% of the number of covered segments (Fig. 50(b)). In contrast, the Dandelion algorithms save the cost of adding covers in FWD_0', displaying the now-well-known pattern that is lower (most often substantially lower) than 100% of the number of covered segments, spiking when side-tree segment removals occur sporadically (Fig. 50(c) and Fig. 50(d)).

Effectiveness of Dandelion Reuse

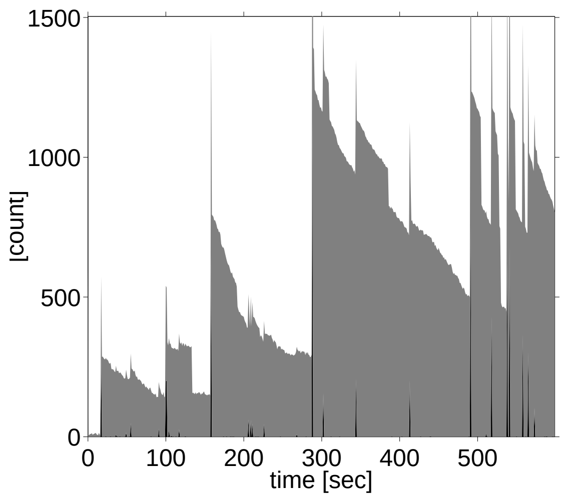
This set of experiments help us gain additional insight into the effectiveness of reuse. Figure 51(a) shows the average age and staleness of covers at each re-evaluation. Recall that the age of a cover is the time elapsed since its creation, regardless of any subsequent updates to the cover (such as pushing and distance changes for a BOP). The staleness of a cover is the time elapsed since its last update of any kind, and is thus an even stricter measure of how long covers live.



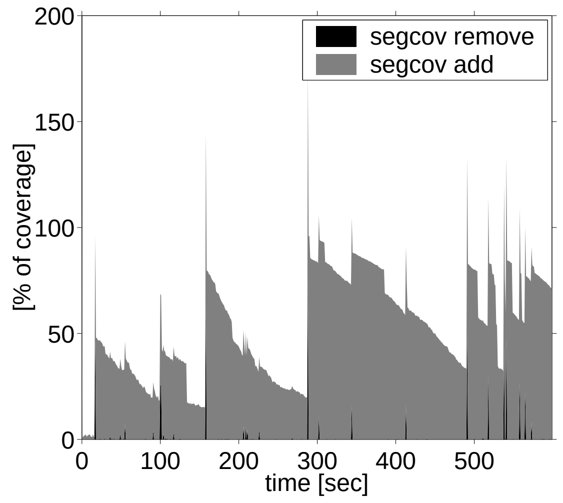
(a) Total NE SegCovMap cover operations.



(b) Total NE SegCovMap cover operations (as percentage of covered segments)

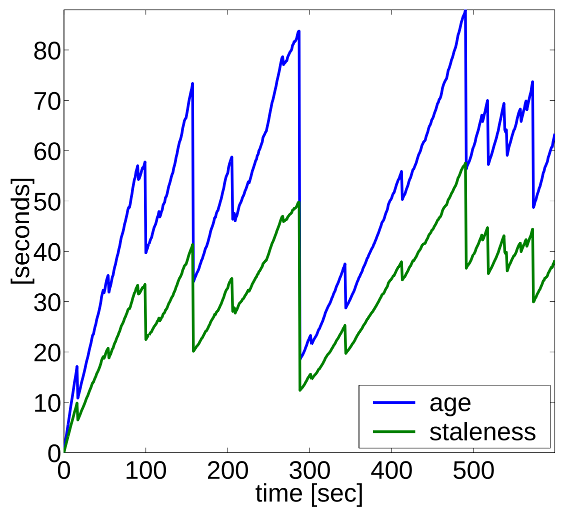


(c) Total Dandelion SegCovMap cover operations.

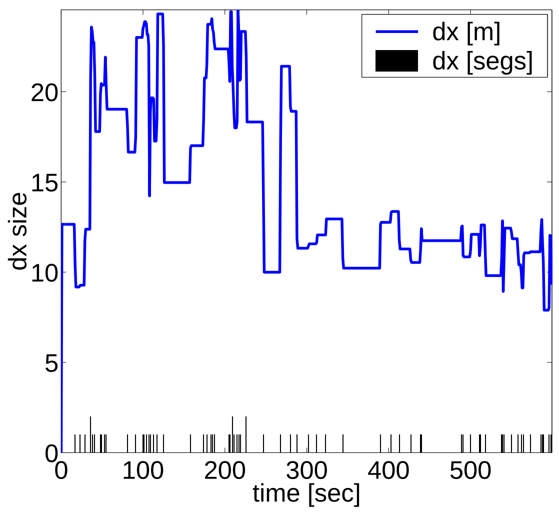


(d) Total Dandelion SegCovMap cover operations (as percentage of covered segments)

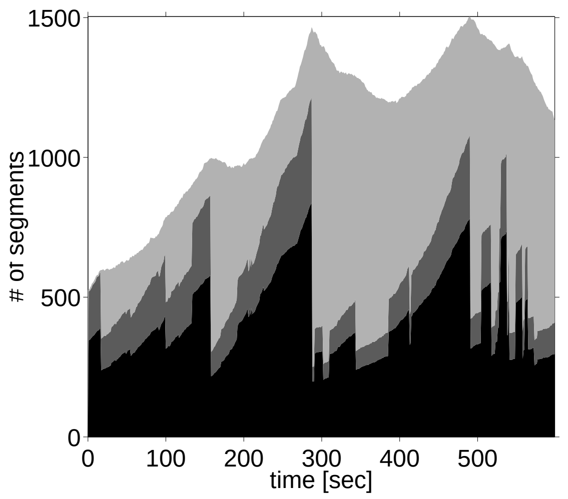
Figure 50: Total SegCovMap adjustment operations (NE compared with Dandelion).



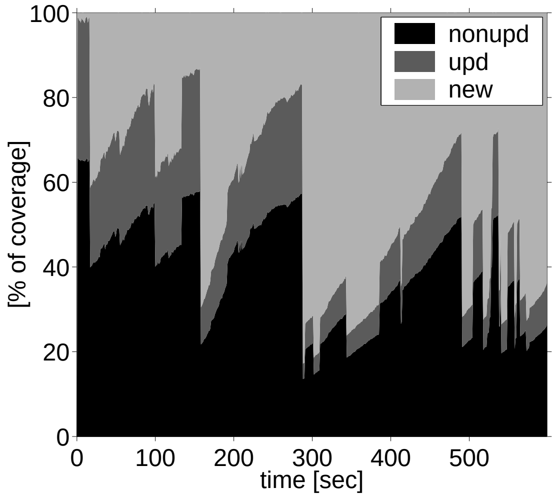
(a) Average age and staleness of covers.



(b) Displacement between re-evaluation locations.



(c) Cover reuse modes.



(d) Cover reuse modes (as percentage of covered segments)

Figure 51: Effectiveness of Dandelion cover reuse.

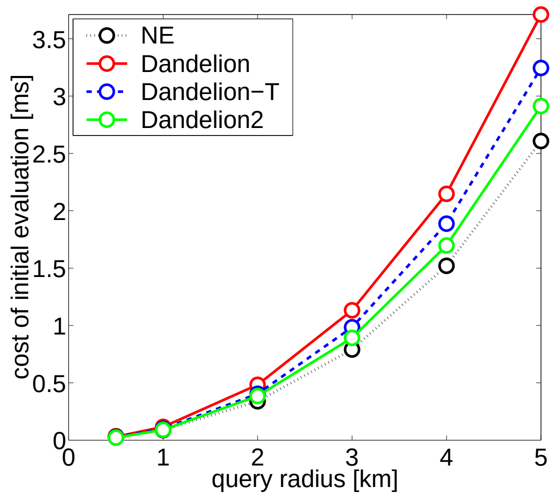
With a re-evaluation frequency of 1 second for our example query, we observe that, rising from an age of 0 initially, covers – on average – can live for more than 80 seconds and be reused without any update whatsoever for more than 50 seconds. Both cover lifetime metrics (age and staleness) exhibit sudden drops, which correspond to the bypassing of large side-trees (recall Fig. 48).

An alternate view of reuse is shown on the stacked Figure 51(c) and Figure 51(d), where all covers are either marked ”new” (created during the latest re-evaluation), ”upd” (updated during the latest re-evaluation, but previously existing, i.e. reused after an update), or ”nonupd” (neither created, nor updated in the latest re-evaluation, i.e. completely reused).

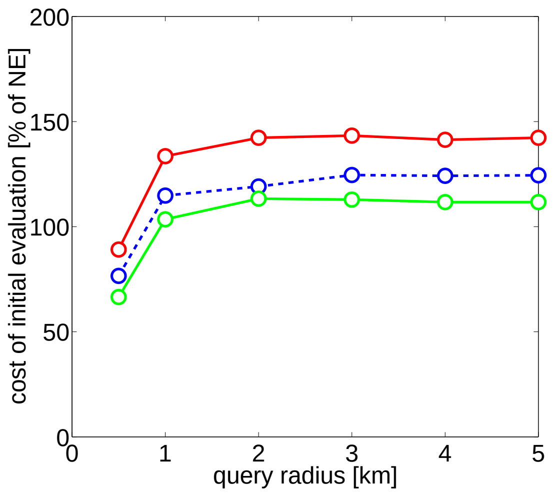
Finally, we show the amount of displacement (dx) between each F and consecutive F' re-evaluation location, in Figure 51(b). Most dx values are zero-segment displacements (i.e. both F and F' are on the same segment), with some relating to two connected segments (dx = 1 segment), and only a few to segments further apart. Viewed as actual road distances in meters, dx is dependent on the speed chosen by the user, and reflects the segment-wise constant speed mobility model followed by the user in our simulation.

3.7.7 Reuse Effectiveness of Dandelion2

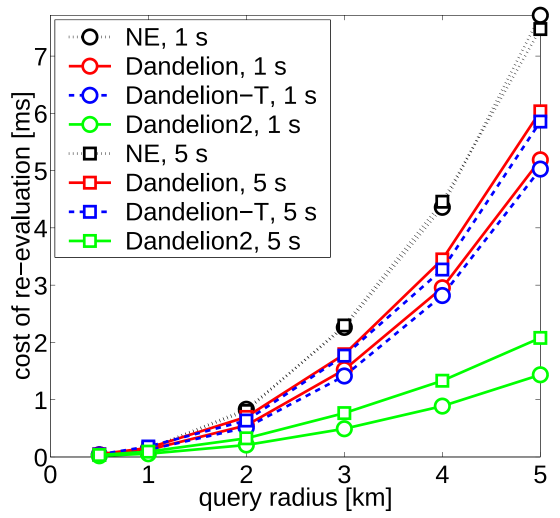
Figure 52 presents the performance comparison of the Dandelion2, powered by FWD/BWD, trident, mov and jump transformation primitives, with Dandelion, Dandelion-T and NE in terms of cost of initial evaluation and re-evaluation with varying radius. Fig. 52(a) and Fig. 52(b)) show the initial evaluation cost and the cost ratio of Dandelion over NE. Although the initial evaluation cost is higher for all Dandelion algorithms due to the build-up of more complex data structures, the initial evaluation cost for Dandelion2 is only approximately 10% higher than it is for NE. In contrast, Fig. 52(c)) shows that all three Dandelion algorithms result in faster re-evaluation costs than NE when the query radius is larger than 1 km. Fig. 52(d)) shows that Dandelion2 algorithm significantly outperforms Danderlion-T, Dandelion and NE for re-evaluation frequency of 1 sec and 5 sec. The cost of re-evaluation



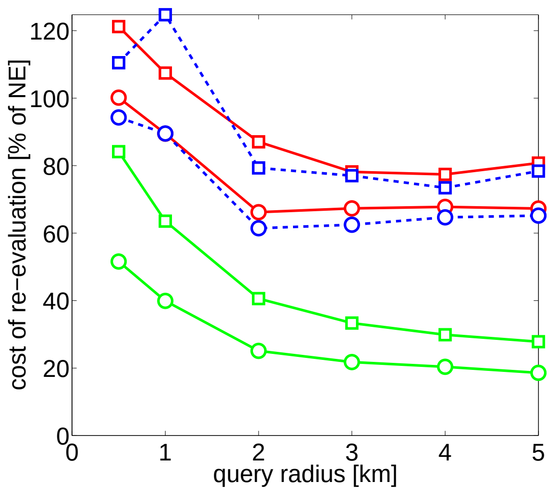
(a) Initial evaluation cost



(b) Initial evaluation cost (as percentage of NE)



(c) Reevaluation cost



(d) Reevaluation cost (as percentage of NE)

Figure 52: Average of initial and reevaluation query calculation costs of NE and three Dandelion versions, with re-evaluation periods of $1\ s$ and $5\ s$.

using Dandelion2 drops to around 20% of the cost of re-evaluation using NE. The re-markable $5\times$ speed-up by Dandelion2 is a substantial performance improvement, and such speed-up *increases* as the radius of queries and thus the cost of re-evaluation increase.

3.8 Related work

Graph algorithms are subject to general interest in computer science due to their widespread applicability to many problems that can be modeled as a graph. Dijkstra's keystone paper on the calculation of single-source shortest paths in a network [12] has been written more than 50 years ago, but retains its relevance today from robotics to Internet routing, and is the algorithm of choice for calculating routes and nearby point-of-interest queries in commercially available personal navigation devices. The fact that Disjktra's algorithm (and its broader interpretation as the Network Expansion algorithm) has retained such a central role is not only a testament to Dijkstra's insight, but also to the fundamental difficulty of improving upon it.

The challenge of any proposed improvement on NE is highlighted by the careful study in [35]. The paper – among other contributions – investigates whether road network range query evaluation could be improved by proposing a *Euclidean restriction*, i.e. applying a filter first in the Euclidean (non-network) space, and only performing graph search after this initial culling of results. The two approaches are termed Range Euclidean Restriction (RER; restrict then expand) and Range Network Expansion (RNE; expand then restrict). The experimental results show that, consistent with the theoretical framework, RER does not improve, but rather increase the computational cost over RNE.

Recently, the work in [30] considers improvements to nearest-neighbor (kNN) continuous road network query answering by attempting to reuse certain information from one query location to the next. The paper highlights the relevance of maintaining an *expansion tree* in proposing an Incremental Monitoring Algorithm (IMA). The concept of *marks* is introduced, which denote the boundaries of the query, and keeping track of partially covered segments is recognized in the concept of the *influencing interval*. The incremental monitoring approach attempts to identify a *valid expansion tree*, which corresponds to the portion of the expansion tree that can be reused. However, the proposed approach suffers from two fatal flaws. First, the algorithm does not consider a network where the valid and invalid portions of the expansion tree can meet, which means that the algorithm is only workable on unrealistic tree type network graphs (i.e. road networks where one cannot drive around a block in a loop, to return to the same location). Secondly, the algorithm does not consider how the valid portion of an expansion tree can be identified, and how expansion at a new location could be initialized with the border points of such a valid expansion tree. As a result, even the valid portion of the expansion tree must be traversed at a new location, entirely negating any performance improvements that might be realized after finding the valid expansion tree.

The related problem of computing nearest neighbors in a regular land surface vertex graph is considered in [45]. The proposed algorithm proposes an Angular Surface Index Tree (ASI-Tree), a thin and tall tree, that partitions the coverage of the query into angular sub-trees, and succeeds in maintaining these sub-trees to keep track of the kNN objects due to the high density of the vertex network, the relatively small displacement of the query center, and the relative sparseness of nearest neighbors in a large network.

The problem of computing shortest path trees (SPT) in the face of edge length changes is investigated in [32], primarily in the context of Internet packet routing. While different from our problem, in that it does not consider the displacement of the root of the SPT itself, the Dynamic SPT computation proposed in the paper is relevant, as it highlights the possibility for partial reuse in the face of at least a limited number of changes in the network graph.

The authors of the above paper further propose a ball-and-string model of dynamically rebalancing SPT trees in the face of network edge updates in [33]. The ball-and-string conceptualization is an important conceptual step forward, and the paper makes the insight that the problem of re-calculating SPTs after network edge updates can be formulated as a linear programming task.

3.9 Conclusion

The computational costs of answering continuous network range queries are known to be prohibitively high, as a shortest path based network expansion needs to be run repeatedly at each and every location where the query is evaluated. We argue that continuous network range queries, whose focal locations are ”not far” from each other, have substantial overlap in their segment coverage. Such a large overlap may offer significant reuse opportunities for performance enhancement. We have presented the design and implementation of Dandelion reuse framework and a suite of algorithms for fast re-evaluations of continuous network range queries. The chapter makes three original contributions. First, we propose the concept of Dandelion tree to accurately represent the coverage of a network range query with arbitrary range, by keeping track of three key network location points: border points (BOP), dead-end points (DEP), and zip points (ZIP). Second, we design three BOP-Push and three BOP-Pull primitive operations to compute the coverage at F by maximum reuse of the coverage at previous query focal location F. Third but not the least, we define the data structures and three Dandelion reuse algorithms to efficiently identify the portion of the Dandelion tree that can be used as the basis for reuse and further expansion. The basic Dandelion algorithm enables reuse by dividing the Dandelion tree (query coverage) of a query into the forward (FWD) and backward (BWD) halves, allowing separate maintenance of the key data structures for each half to reduce the search space. The Dandelion-T algorithm introduces and utilizes the Trident and Guide data structures to compose a more reuse-efficient Dandelion-T tree, leading to faster query re-evaluation than Dandelion basic algorithm. Finally the Dandelion2 algorithm further enhances Dandelion-T in terms of query re-evaluation cost by introducing the two primitive transformation operations *move* and *jump*. This development can effectively transform one Dandelion tree to another with a minimum set of primitive transformation operations. We conduct a series of extensive experiments and our results show that Dandelion reuse model and algorithms can significantly outperform the conventional shortest path network expansion algorithm (NE) in terms of coverage computation cost for non-trivial radius size and high re-evaluation frequency.

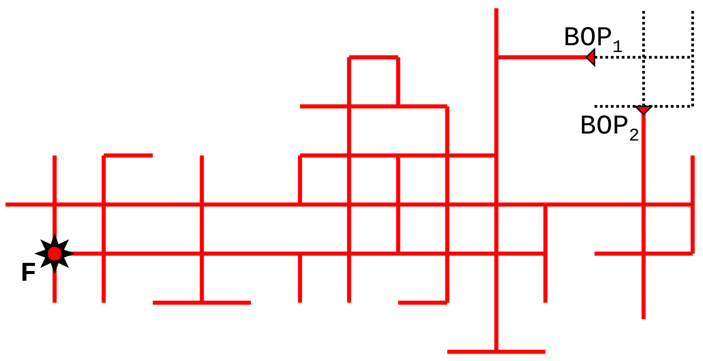
CHAPTER IV

FASTEXPAND

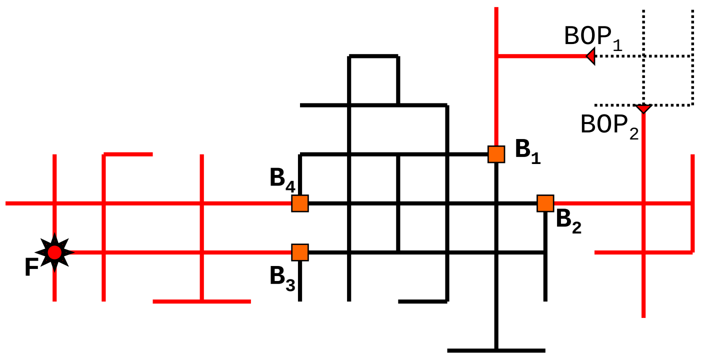
We addressed the problem of speeding up continuous road-network queries in Chapter 3. In this chapter, we briefly consider the problem of accelerating the computation of range query coverages in road networks, even when the query is only evaluated a single time, and thus a reuse-oriented approach is not applicable. Our approach is to divide-and-conquer by constructing precincts over the road network graph. The concept of precinct was first introduced in Chapter 2, and the alternatives of hop- or distance-based precinct radius definitions apply here as well, including the choice of road network distances or road network travel times to be used as the relevant metric.

4.1 FastExpand query coverage

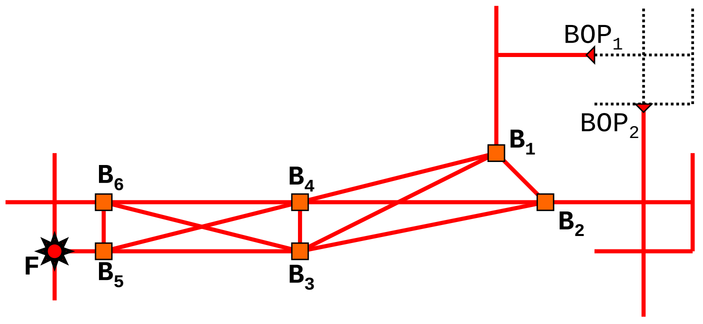
Figure 53(a) shows a range-query that is evaluated along the edges of the road network. The query has a range of 1500 m (with a standard block size of 100 m), and produces two border points (BOPs). Such a long-range query produces a high segment coverage, with a large number of connected segments on the inside of the query being completely covered. This observation is further highlighted on Figure 53(b), where the high-connectivity intermediate neighborhood bounded by four boundary points (B_1, B_2, B_3, B_4) is highlighted. This high-connectivity intermediate neighborhood is completely covered by the query, but contains many segment that add to its complexity, but do not contribute to the distance calculations of a long-range shortest path query. Figure 54(a) further highlights that the many small local-neighborhood segments do not play a role, when one is only concerned with the shortest paths between other points in the network. In this case, only the four boundary point serve as entry points into this neighborhood, and any further internal graph structure may be disregarded. In fact, we may replace the complex neighborhood with six fast-track



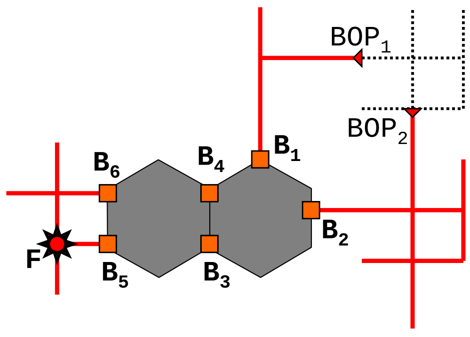
(a) Query in the original road network.



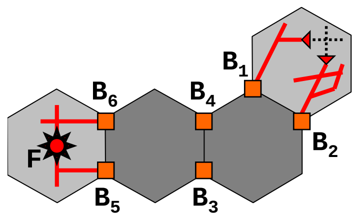
(b) A completely covered high-connectivity intermediate neighborhood.



(c) Road network simplified by shortcuts in completely covered precincts.



(d) Precincts used to simplify core coverage.



(e) A seed, two core and one border precinct.

Figure 53: Construction steps of a FastExpand expansion network for an $r = 1500\ m$ radius range query.

traversal shortcuts, as shown on Figure 54(b), and still ensure the correctness of all distance calculations that contain this selected neighborhood. We use the already familiar *precinct* term to denote such neighborhoods, and their pre-computed traversal shortcuts. All high connectivity neighborhoods may be ”flattened” in this way, by creating a precinct coverage of the entire road network, as described in Chapter 2. Figure 53(c) continues our example with the shortcuts of the intermediate precincts shown in this simplified network.

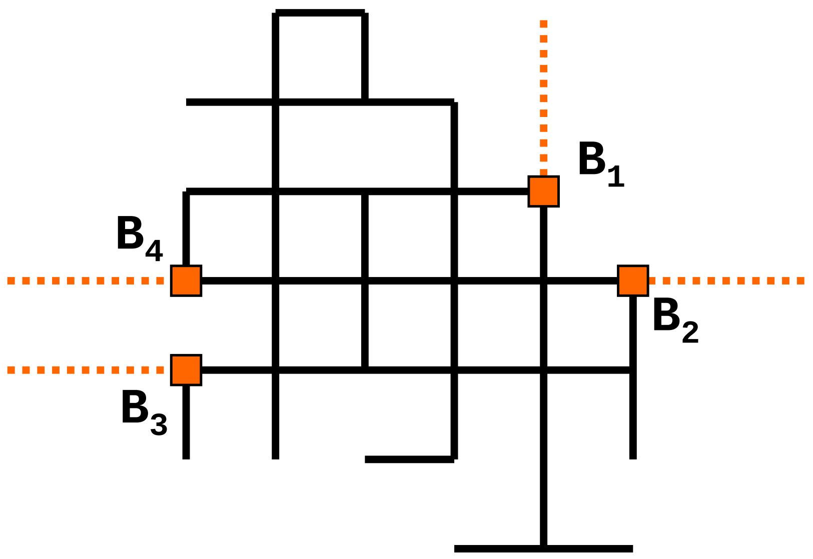
For purposes of shortest path distances, precincts are black boxes, as illustrated on Figure 53(d), as the pre-computed shortest paths between pairs of precinct boundary points are utilized as shortcuts, without the need to refer to the internal graph structure found inside a precinct. The distance computation can’t take advantage of precinct-based shortcuts in the immediate vicinity of the query focal location F, and near the query border points locations BOP_i, even though these locations are also found inside some neighborhood. This gives rise to three distinct precinct types.

A *seed precinct* is the single precinct containing the F focal location. Low-level local graph search must be performed in a seed precinct, until the search reached the boundary points of the seed precinct.

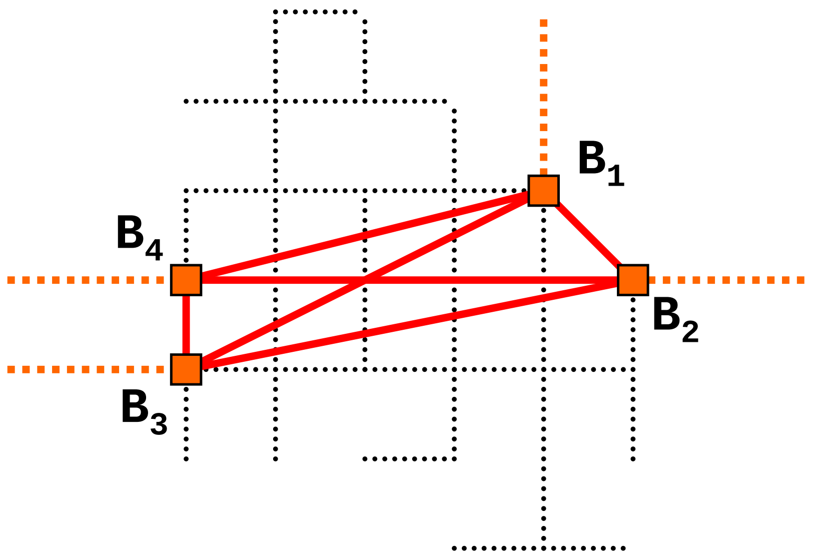
The *core precincts* are those fully covered intermediate neighborhoods that can be safely traversed using pre-computed shortcuts, as they are neither close to the focal location, nor close the any border point.

The *border precincts* are those partially covered neighborhoods that contain at least one border point of the query. Because some segments inside a border precinct are covered, while other segments are not covered, a local search must be performed, and the shortcuts can’t be taken advantage of.

Figure 53(e) shows our continuing example, with all segments assigned to one of the three precinct types. Figure 55 shows a general example, where a road network range query’s coverage is tiled by a composite of seed, core and border precincts. Figure 56 shows a screenshot of the FastExpand coverage of an example query, with shortcuts in core



(a) Detail of intermediate neighborhood.



(b) Fast-track traversal shortcuts replace local complexity.

Figure 54: Detail of a completely covered high-connectivity intermediate neighborhood from Figure 53(b).

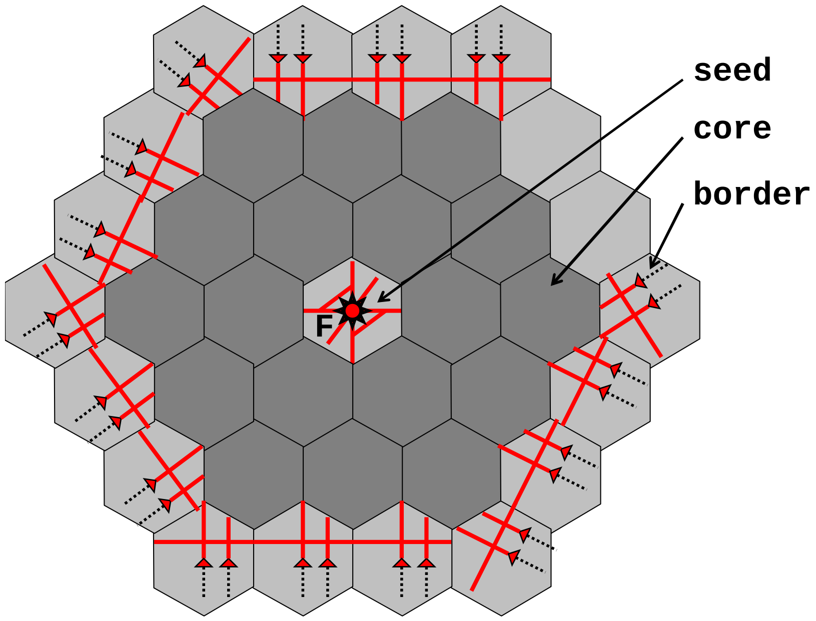


Figure 55: Tiling of a road network range query's coverage by a composite of seed, core and border precinct types.

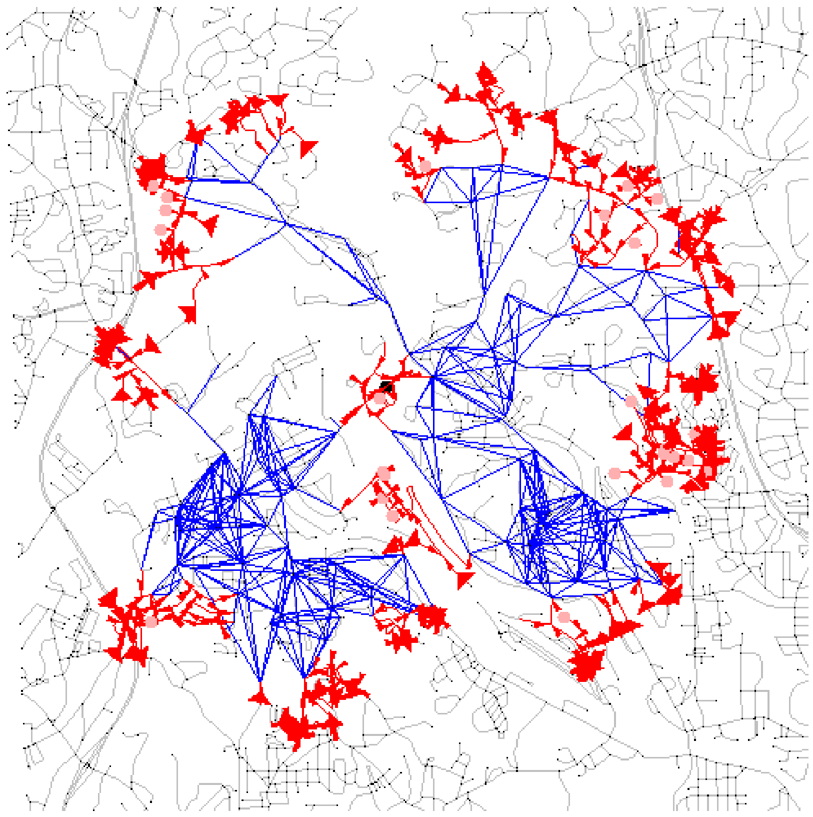


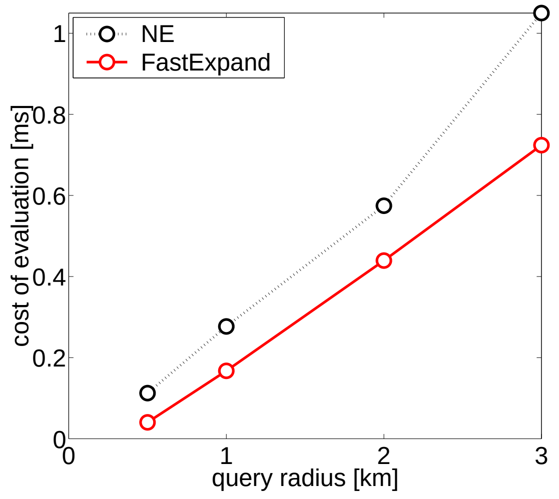
Figure 56: Screenshot of the FastExpand coverage of a query (shortcuts in core precincts in blue; local segments in seed and border precincts in red).

precincts drawn in blue, while local segments in seed and border precincts are drawn in red. The F focal location is the black dot inside the red local segments of the central seed precinct.

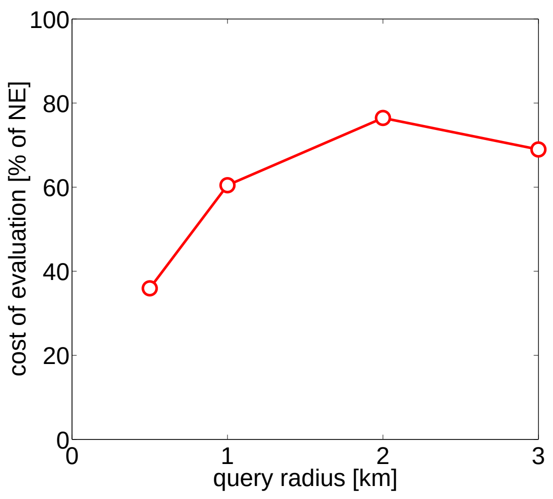
The evaluation of the coverage of a road network range follows a modified Network Expansion algorithm, whereby a priority queue of shortest tentative node distances is maintained, and the least distance node is processed at each step. The expansion start regularly from F, with a local search in the graph. However, unique to the FastExpand algorithm is the need make a choice between continuing with a local search or with a shortcut-based fast search, at each precinct boundary point. For each boundary point B, the ***reach*** distances (ρ) are pre-computed and stored. The reach of B_{ij} within precinct P_j is the $\rho_{ij} = max(network\_dist(B_{ij}, L))$ distance, such that $\exists L \in P_j$. One reach distance is stored for each precinct that a boundary point is connected to. When a yet-unexamined precinct P_j is first encountered by popping its B_{ij} boundary point from the queue, we test whether it is possible to reach an L location within P_j, such that the path from F to L via B_{ij} is longer than the r range of the query. If such a path exists, then $network_dist(F, B_{ij}) + \rho_{ij} > r$. Testing this *drop-down criterion* ensures that we stop the network expansion using the core precinct shortcuts as soon as the potential for encountering a query coverage border point (BOP) inside the next precinct arises, and we drop down to pursue the expansion in the original graph. Subsequently, the local search terminates regularly, when a BOP border point at exactly r distance from F is met.

4.2 Experimental evaluation

In our first set of experiments, we use a $d = 500\ m$ distance-based precinct radius partitioning of the Cook county, IL city map, and perform a comparison with Network Expansion. Figure 57(a) shows the query evaluation costs using the two algorithms, with various query radius settings. Figure 57(b) shows the query evaluation costs for FastExpand as a percentage of the Network Expansion baseline. As expected, FastExpand outperforms NE by



(a) Query evaluation cost



(b) Query evaluation cost (as percentage of NE)

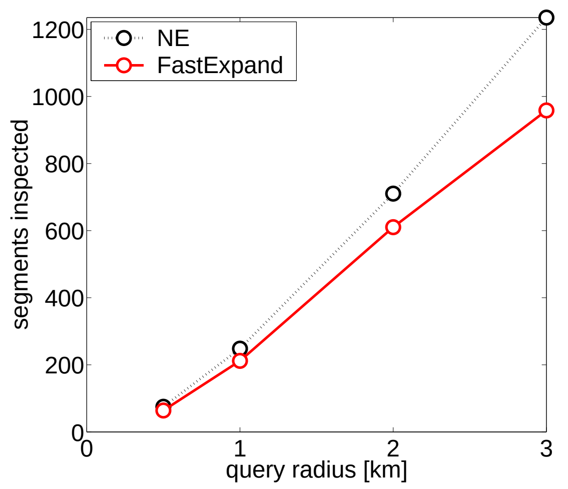
Figure 57: Road network range query evaluation costs of FastExpand, with increasing query radius settings.

taking advantage of the pre-computed shortcuts.

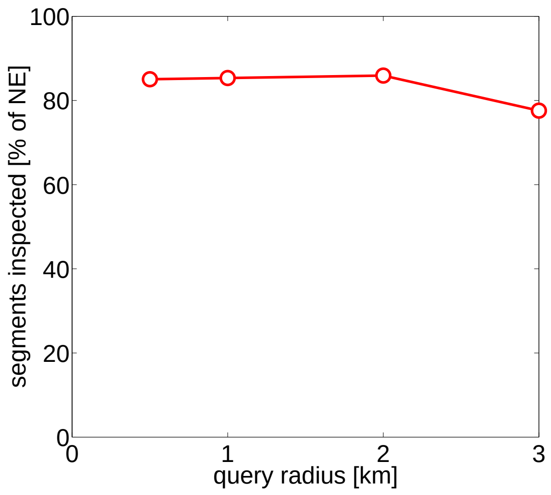
Figure 58(a) shows the number of segments inspected using the two algorithms, with various query radius settings. Figure 58(b) shows the number of inspected segments for FastExpand as a percentage of the Network Expansion baseline. We note the same pattern of FastExpand outperformance, as on the previous set of graphs.

We compare the FastExpand algorithm against the baseline Network Expansion algorithm on two maps (Table 3), using $d = 200\ m$ precinct radius partitioning: Cook county, IL (Chicago area, Fig. 44(b)) is a suburban city map, with residential areas and dead-end streets or cul-de-sacs. Coconino county, AZ (Fig. 44(c)) is a rural map, with long highways passing across a desert region, with an occasional small town.

Figure 59(a) shows the number of segments inspected using the two algorithms, during expansion for the rural and urban maps. Figure 59(b) shows the number of inspected segments for FastExpand as a percentage of the Network Expansion baseline. We observe that our shortcut- and precinct-based algorithm performs even better on rural maps, where the network topology allows the creation and use of an even more effective precinct partitioning.

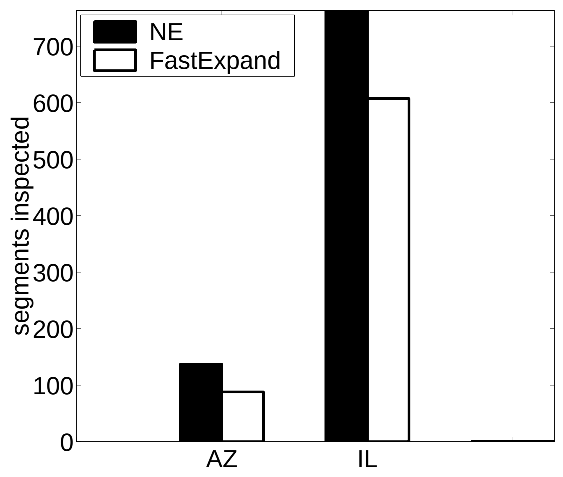


(a) Segments inspected

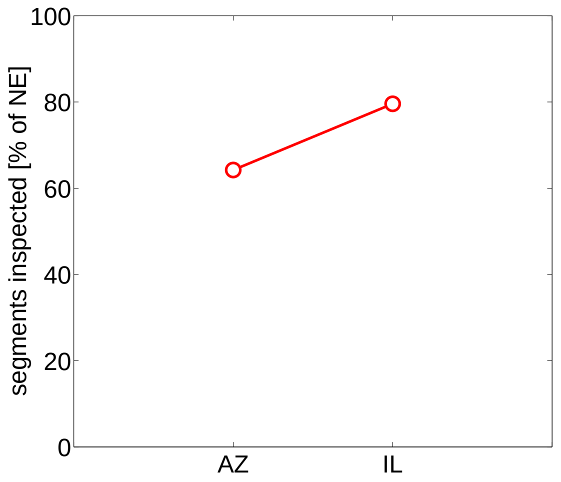


(b) Segments inspected (as percentage of NE)

Figure 58: Segments inspected during evaluation of a road network range query, with increasing query radius settings.



(a) Segments inspected



(b) Segments inspected (as percentage of NE)

Figure 59: Segments inspected during evaluation of a road network range query, on a rural (AZ) and an urban map (IL).

4.3 Conclusion

In recent years, algorithms for finding shortest paths in road networks have enjoyed ongoing interest, as seen in [40], [2], [3], [41], [16], [28] and many more. The problem of computing the full coverage of a road network range query, – where no target exists, but only a source, – is a related, albeit somewhat different problem.

In this brief chapter, we considered the problem of accelerating the computation of range query coverages in road networks, even when the query is only evaluated a single time, and thus a reuse-oriented approach is not applicable. We presented our approach of constructing precincts over the road network graph to eliminate the unnecessary complexity of local neighborhood streets and replaced them with fast shortcuts. We provided a classification of precincts into seed, core and border types, and a criterion to determine when the coverage computation should choose local search instead of shortcut based search. Our experimental results showed that this approach is able to speed up the computation of individual static road network queries.

CHAPTER V

MAPSTITCHER

Commercial aerial imagery websites, such as Google Maps, MapQuest, Microsoft Virtual Earth, and Yahoo! Maps, provide high-resolution seamless orthographic imagery for many populated areas, employing sophisticated equipment and proprietary image post-processing pipelines. There are many areas of the world with poor coverage where locals might benefit from recent, high-resolution orthographic imagery, but which do not fit into the schedules and scaling model of the big sites.

This chapter describes MapStitcher, a system that orthorectifies and geographically registers imagery using only low-cost capturing equipment. MapStitcher combines manually-entered relationships between images and known ground references with a MOPs-based image-stitching technique that automatically discovers image-to-image relationships. Our image registration pipeline first extracts and matches feature points, then clusters images, then uses RANSAC-initialized bundle adjustment to simultaneously optimize all constraints over the entire image set. Simultaneous optimization balances the requirements of precise stitching and absolute placement accuracy. We used this technique to image a portion of the Skagit River Valley in the vicinity of the town of Concrete, WA (pop. 790) at 0.15 m/pixel. Our technique is more accurate than stitching followed by "rubber-sheeting" (deforming the stitched image into global coordinates), while it also requires less effort and produces a better-stitched composite than rubber-sheeting images separately.

A version of this chapter was published as a paper co-authored with Jeremy Elson, Jon Howell, Drew Steedly and Matthew Uyttendaele [36].

5.1 Introduction

Commercial aerial imagery sites, such as Google Maps, MapQuest, Microsoft Virtual Earth, and Yahoo! Maps, provide high-resolution seamless orthographic imagery for densely populated areas. To be able to image large areas in a cost-efficient manner, their techniques depend on special-purpose cameras mounted in gyro-stabilized mounts and flown in autopilot-equipped airplanes. Together, these components tightly constrain the parameters of the captured images, easing the task of post-processing the collection of images into a single orthorectified image mosaic. While allowing to amortize the cost of the system by imaging large areas, the equipment is also quite expensive; for example, the Vexcel UltraCam-D camera costs over half a million dollars. Furthermore, there are only a few competitors, and they tend to prioritize imaging populous markets. Users in small markets would also stand to benefit from access to recent, high-resolution geographically registered aerial imagery. However, it is beyond the means of small communities and other "long tail" users to purchase the expensive tools used by the large imaging operations. In addition, the post-processing pipelines used in the industry are proprietary, posing an additional barrier to entry for localized operations.

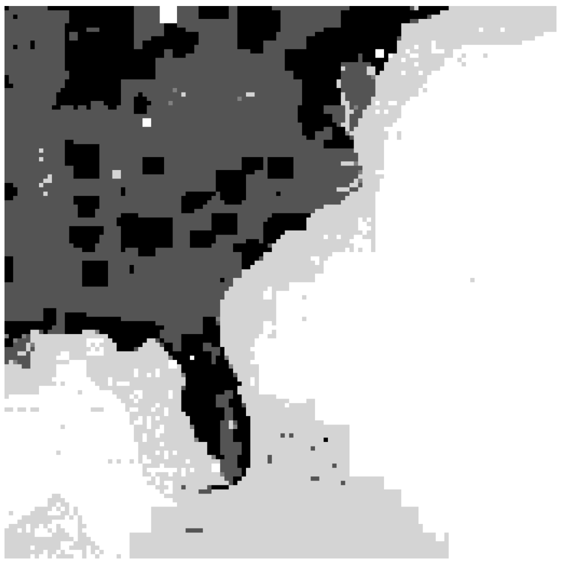
A quick survey of the image tiles available on public imagery sites reveals the lack of resolution for many regions of the Earth. For example, while most of the United States is covered at a 1 m/pixel resolution, with metropolitan areas imaged at 0.25 m/pixel (see Figure 60(a) and Figure 60(b), showing the eastern United States), other continents are mostly covered at 16 m/pixel (see Figure 60(c) and Figure 60(d), showing an area of the Earth bounded by the Equator (S), the Arctic Circle (N), 0°(W). and 90°E longitudes (E); some large cities and Western European countries have higher resolution coverage). Furthermore, the imagery update schedules of the big sites are independent of important changes in the environment, such as natural disasters, construction and demolition of roads, buildings and parking spaces. This chapter describes a system designed to provide such imagery at a low cost of entry in a timely fashion: imagery is captured with a consumer-grade camera mounted on hardware-store plumbing pipe in a minimally-equipped light airplane, and post-processed with a generic pipeline that depends on a small amount of human annotation. While this approach has a higher cost per image of human annotation, the dramatically lower capital costs lead to lower overall cost for a small imaging project.

We contrast our approach with two simpler techniques for orthorectifying poorly-constrained aerial imagery.

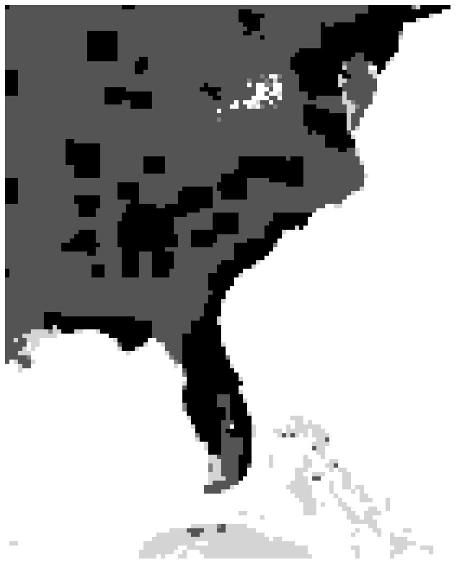
The first approach is to simply manually annotate every captured image and then deform each image into place ("rubber-sheeting") with a tool such as MapCruncher [13]. MapCruncher scales well, allowing users to readily reproject existing maps, publishing multi-gigapixel images on the web in a client-bandwidth-friendly tiled format that interoperates with Microsoft Virtual Earth. Our experiments with this approach identified two problems: First, because the post-processing system only had information about global placement, relative inter-image placement often suffered, leading to obvious discontinuities at image boundaries. Second, where the images covered undifferentiated or entirely changed terrain, such as a construction site, there was no easy way to manually label the images with ground reference pairs.

In the second approach, the captured images are stitched into a single image of large extent using a modern photo stitching tool [4] that makes inter-image camera-pose estimates to reproject the images to eliminate boundary discontinuities. The resulting "panoramic" image represents a single theoretical image taken from a single logical viewpoint; this image is then related to ground references, and translated to a browser accessible user interface [13]. In practice, the lack of global constraints causes the photo stitcher to accumulate error and emit images that correspond to no real viewpoint of the original terrain.

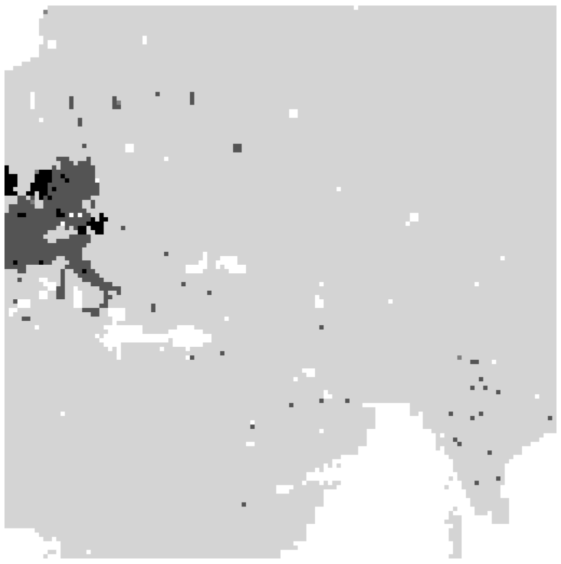
This chapter describes MapStitcher, an image orthorectification system that combines the two approaches above simultaneously. MapStitcher's stitching component discovers



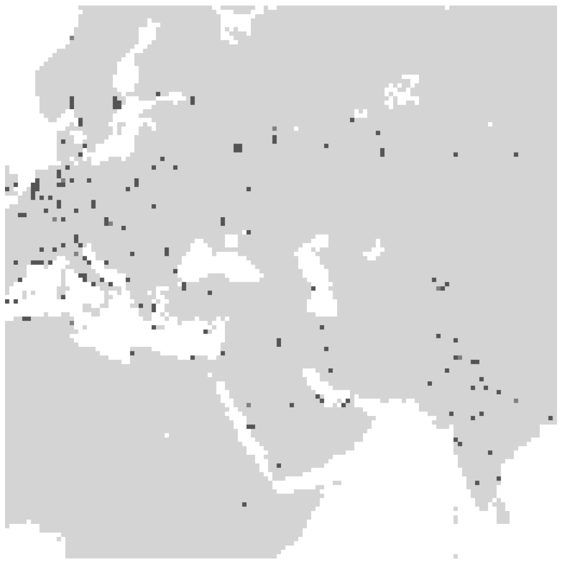
(a) Resolution of coverage in Virtual Earth over the eastern United States.



(b) Resolution of coverage in Yahoo! Maps over the eastern United States.



(c) Resolution of coverage in Virtual Earth over portions of Africa, Europe and Asia.



(d) Resolution of coverage in Yahoo! Maps over portions of Africa, Europe and Asia.

■	0.25 m/pixel
▩	1 m/pixel
▨	16 m/pixel

Figure 60: Resolution of orthophoto coverage in large mapping websites.

inter-image constraints. A human annotates a few images with ground reference constraints. Then MapStitcher estimates the pose of each image's camera by first initializing

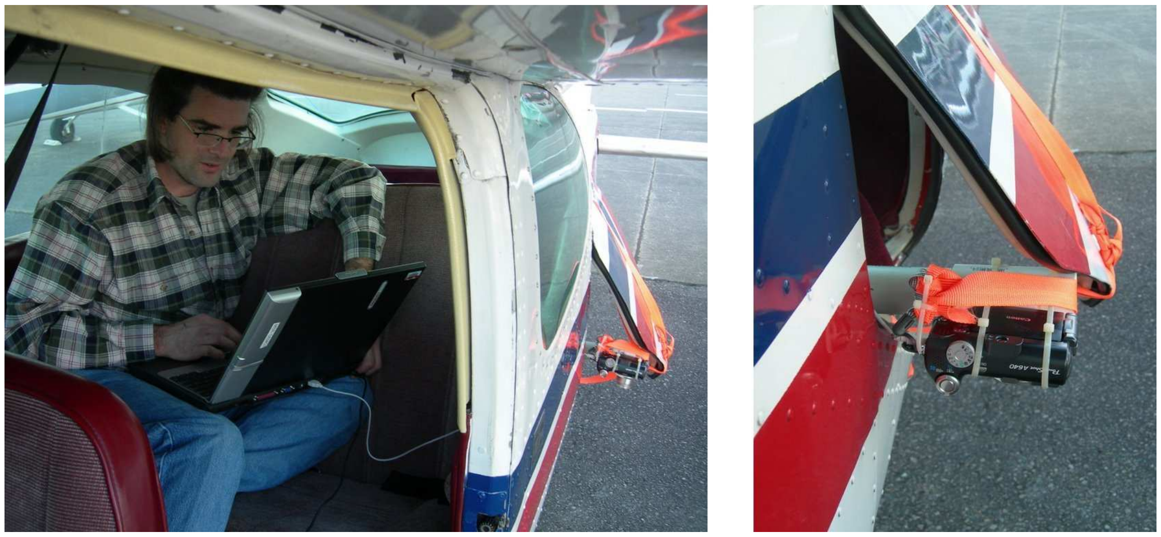


Figure 61: Our operation: a consumer camera zip-tied to a PVC pipe protruding from a hand-flown Cessna 177.

with RANSAC, a general technique for fitting a model in the presence of outliers. Then it uses bundle adjustment to minimize error across the entire constraint set, both relative and global. The resulting system is robust to poorly-constrained camera geometry, requires global constraints on only a small subset of images, and produces output with minimal image-boundary discontinuities.

We demonstrate MapStitcher by capturing imagery of the Skagit River Valley in the vicinity of the town of Concrete, Washington. Concrete's population of 790 has a long wait before major services will find it profitable to send a photo mission with expensive equipment. Our mission, in contrast, involved an ordinary four-seat Cessna ($160/hour rental, including pilot), three feet of PVC pipe, a consumer digital camera ($300), and two people: one pilot and one to operate the camera shutter and change the batteries (Figure 61). In post-processing, we identified 25 ground reference pairs, and used 60 photos to produce a 208 megapixel image at a resolution of 0.15 m/pixel (Figure 62).

5.2 Related Work

The creation of aerial mosaics to form composite photomaps is described in [11]. Our method is analogous with the creation of semicontrolled mosaics, where ground reference

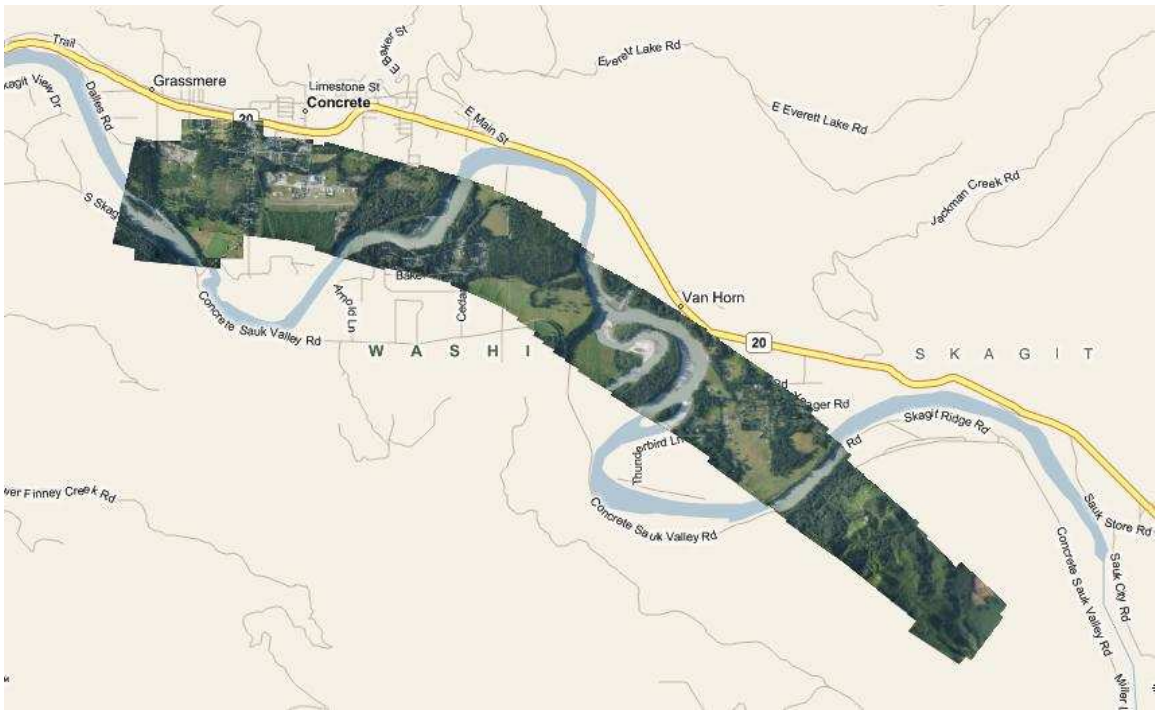


Figure 62: Our 0.15 m/pixel composite aerial imagery, showing a portion of the Skagit River Valley near Concrete, WA, overlaid on a map of the area.

pairs on a small number of images are combined with tie points between images to compute the transformation parameters. Our *stitch-first* control method is analogous with the creation of uncontrolled mosaics, and the *no-stitch* method is analogous with the creation of controlled mosaics. However, these digital mosaicking approaches only attempt to solve for rotation and translation parameters, assuming vertical camera positions during image acquisition.

In order to perform digital mosaicking with less constrained cameras, the problem of estimating camera parameters must be tackled. Analytical aerotriangulation with simultaneous bundle adjustment aims to recover the 3D coordinates of object points, and the 3D location and exterior orientation parameters of all exposure stations [11]. These goals are similar to our objectives in our camera parameter estimation step. Using GPS to obtain $a\ priori$ knowledge about the three-dimensional position of the exposure stations is a possible improvement [44]. Alternatives to bundle adjustment for solving the equations to estimate projection matrices and scene point locations are explored in [26]. For an introduction to 3D reconstruction of cameras and scene structure from photographs, we refer the reader to [17]. The problem of 3D scene reconstruction using bundle adjustment has also been explored recently in a computer vision context [5]. Bundle adjustment based methods [27] can benefit from initialization with RANSAC [14]. Specific techniques also exist for the estimation of interior [19] and exterior parameters [18] of cameras from line measurements, and for n-point camera pose determination [39].

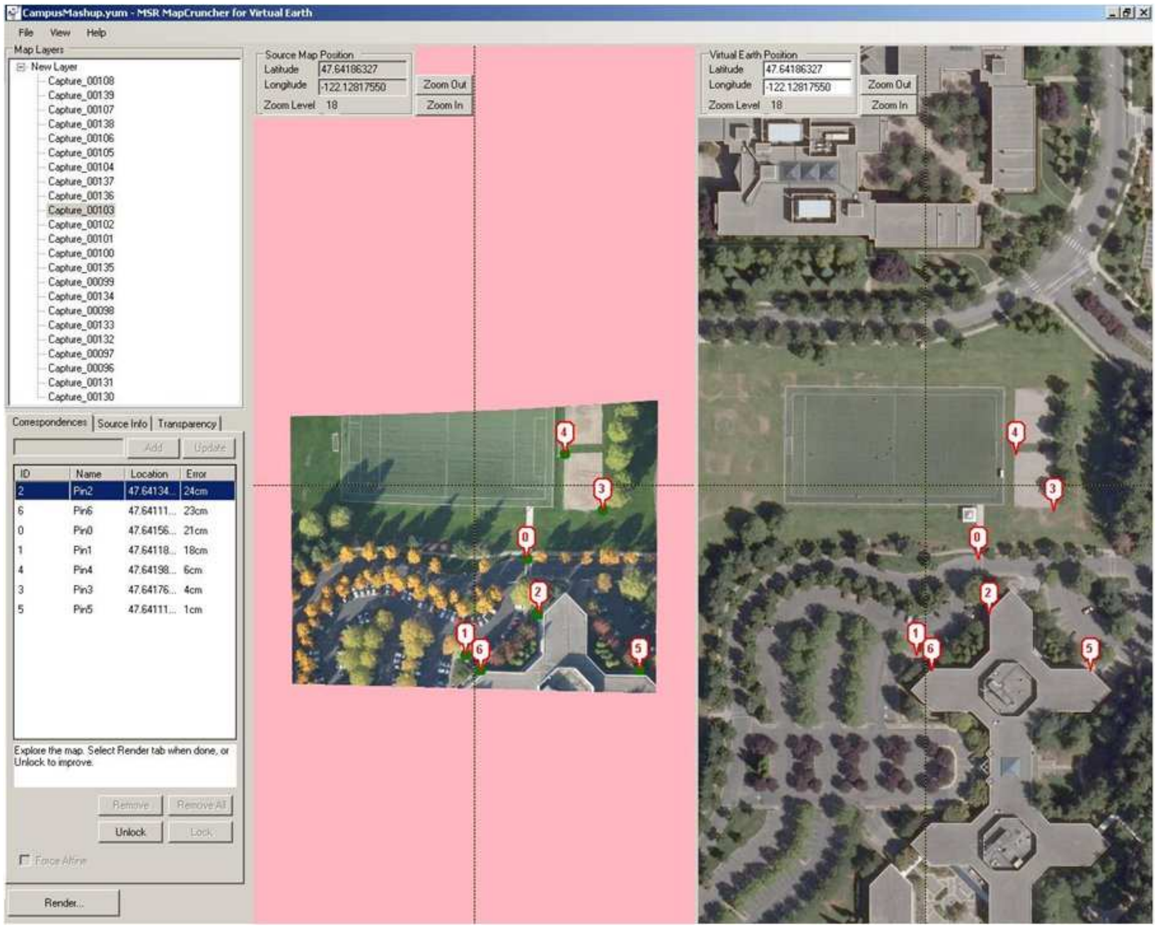
5.3 Goals of Aerial Image Composition

Before describing MapStitcher's image processing pipeline, we first describe its design goals.

The pipeline should convert an input set of overlapping images, acquired individually, into a single virtual image that covers the same area. In constructing this composite image, we would like to simultaneously optimize for two goals. The first is geographic fidelity: features should have the correct shape in the composite image. For example, a straight road should not appear to curve in the image. The second goal is seamlessness: the boundaries between the input images should be invisible in the composite image. That is, there should not be visible discontinuities in features such as roads.

To ensure our system is practical and economical, we also have two usability goals. The first is that our pipeline should accept reasonably unconstrained input images—for example, it should not require pictures taken exactly straight down, or with cameras whose exact geometry or position is known. Such stringent requirements would significantly increase the cost of image acquisition. Our second usability goal is that the pipeline should require a minimum of user effort. A few hours of image acquisition should not be followed by weeks of manual post-processing.

In light of these goals, it is instructive to consider the weaknesses of other methods for generating a geographically accurate composite image. In this section, we will consider



Source Map Position
Latitude 47.64186327
Longitude -122.12817550
Zoom Level 18

Virtual Earth Position
Latitude 47.64186327
Longitude -122.12817550
Zoom Level 18

ID	Name	Location	Error
2	Pin2	47.64134...	24cm
6	Pin6	47.64111...	23cm
0	Pin0	47.64156...	21cm
1	Pin1	47.64118...	18cm
4	Pin4	47.64198...	6cm
3	Pin3	47.64176...	4cm
5	Pin5	47.64111...	1cm

Explore the map. Select Render tab when done, or Unlock to improve.

Figure 63: The user interface of both MapCruncher and MapStitcher. Users can specify ground reference pairs by finding the same feature in their own image and the standard Virtual Earth imagery. If the area has been manually surveyed, latitude and longitude can also be entered numerically.

two that are commonly used in low-cost applications: individually "rubber-sheeting" each photo in the set, and rubber-sheeting a composite photo that was created with an image stitching tool. The main weakness of these methods is that they optimize for only one goal—geography or seamlessness—at a time.

The first method is exemplified by the previous work, MapCruncher [13], which can perform approximate Mercator reprojection of any image drawn to scale after being given a few correspondence points as exemplars. We call these points *ground reference pairs*—that is, correspondences between a pixel in an input image and a latitude and longitude in WGS84. MapCruncher has a simple interface, depicted in Figure 63, for specifying these pairs. Although surveying techniques (e.g., GPS) can be used, the fastest and easiest way is to establish ground reference pairs is to visually compare the newly acquired imagery with the existing imagery that is part of Microsoft Virtual Earth. We have found this technique useful because a typical use-case is overlaying recent high-resolution images on top of extant older or lower-resolution images. MapCruncher shows the user's images in one window and Virtual Earth in another.

MapCruncher was originally designed for use with maps. Our initial tests in using it for aerial image compositing were promising, but had two major drawbacks. First, MapCruncher considers the placement of each image individually, without global constraints. As a result, relative inter-image placement often suffers, causing obvious discontinuities at image boundaries, such as those shown in Figure 64. Second, where the images cover undifferentiated or entirely changed terrain, such as a new construction site, generation of ground reference pairs is difficult. The evaluation refers to this technique as *no-stitch*.

A second common approach is a two-step procedure. First, use a modern photo stitching tool [4] that makes inter-image camera-pose estimates and reprojects the images to eliminate boundary discontinuities. Next, rubber-sheet the mosaic to fit it to the depicted geography. In practice, we have found the lack of geographic constraints during the mosaic step causes the photo stitcher to accumulate error and emit images that correspond to no real viewpoint of the original terrain. For example, the mosaic shown in Figure 65 depicts about a mile of a straight north-south street, captured with a dozen individual photos shot from an airplane. Without geographic constraints, the stitcher incorrectly emits a (seamless) photo of a curving road. The evaluation refers to this technique as *stitch-first*.

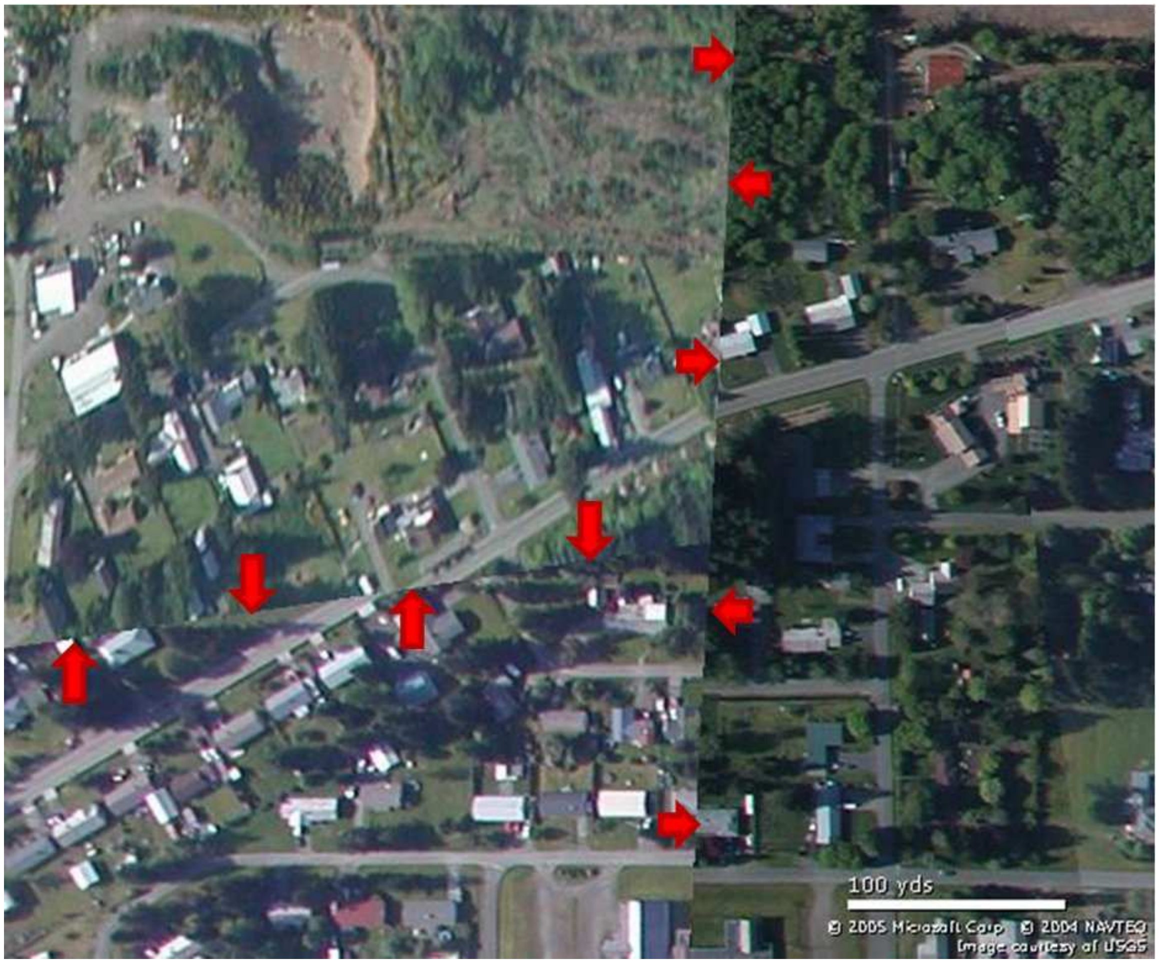


Figure 64: When overlapping aerial images are rubbersheeted individually, discontinuities at the image boundaries are obvious.

5.4 The MapStitcher Image Pipeline

The MapStitcher image pipeline works by simultaneously combining user-specified geographic image constraints, similar to MapCruncher, and automatically generated image-stitching constraints, similar to a photo stitcher. With relatively little user effort, MapStitcher can convert a series of overlapping aerial images into a seamless, orthorectified, and geographically accurate composite. Users typically only need to specify a small number (e.g., 10) of ground reference pairs. For example, references might be set for only the first and last images in a series; the positions of intermediate images are estimated automatically using feature comparisons in the overlapping regions.

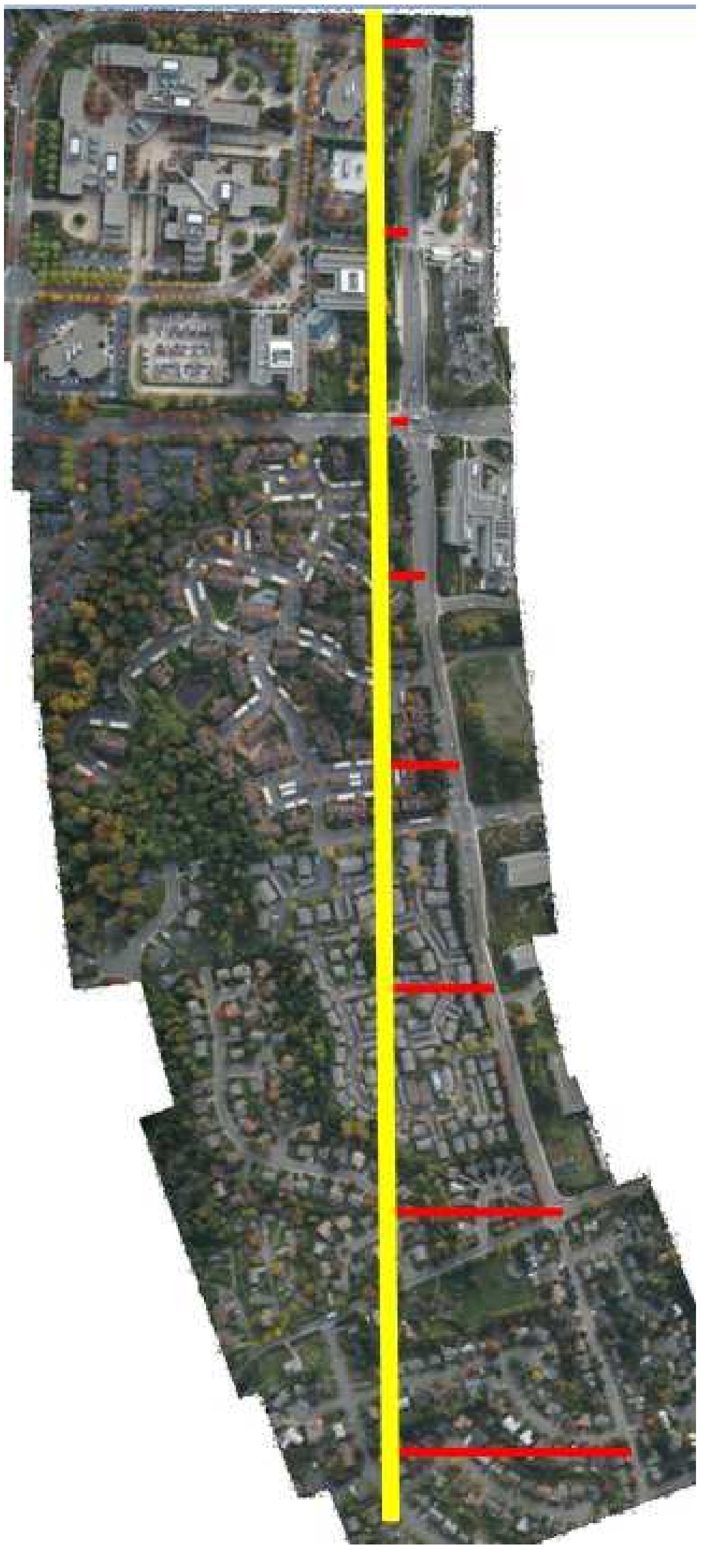


Figure 65: A straight road, captured with 12 aerial photographs and mosaicked using an image stitcher. Without geographic constraints, the road appears to curve.

Image compositing is accomplished by first solving for the position and orientation of the camera at the moment each image was acquired. Then, each image is reprojected into an orthographic approximation and superimposed.

A homographic projection is used to model the view of the camera at each instant it acquires each image. Our model includes both *intrinsic* and *extrinsic* camera parameters. Intrinsic parameters are properties of the camera itself: currently just its focal length, captured in the F matrix. The extrinsic camera parameters are the translation and rotation, captured in the T and R matrices, respectively. In our model, a ground point (p_{ground}) is projected to an image point (p_{image}) according to the chained transformations:

$$\begin{aligned} q =& F \cdot T \cdot R \cdot M_{pre} \cdot p_{ground}, \\ p_{image} =& M_{post} \cdot \begin{pmatrix} \frac{q_x}{q_z} & \frac{q_y}{q_z} \end{pmatrix}', \end{aligned}$$

where p_{ground} and q are 3D points represented as 4D homogeneous coordinates; F, T, R and M_{pre} are 4D matrices; M_{post} is a 2D matrix; and p_{image} is a 2D point. As a typical scene spans 10^{-6} equatorial circumferences in Mercator coordinates, the M_{pre} pre-transform matrix is used to scale the scene so that its size is comparable to the size of its projection on the camera's image plane, which has a largest dimension of 1.0. This scaling avoids rounding errors that lead to ill-conditioned optimizations. The M_{post} post-transform matrix ensures that the scene's projection is centered on the image plane. This centering is required to model the symmetry of the perspective projection around the center of the real camera's imaging surface.

The remainder of this section will describe how all of the camera parameters are estimated for each image acquired. Generally speaking, the procedure entails the following steps:

1. The user specifies ground reference pairs for a subset of the images to be stitched (Figure 63).

2. MapStitcher automatically finds common features in images that overlap (Section 5.4.1).

3. Each camera's model parameters are initialized to the "not estimated" state.

4. Iterate:

   (a) Initial estimates for camera model parameters are made for each camera in a "not estimated" state, that has sufficient ground reference pairs (Section 5.4.2).

   (b) Nonlinear optimization (bundle adjustment) is used to globally optimize the parameters of all cameras with estimates. Both the user-supplied ground reference pairs and constraints introduced by feature match pairs are used in this global optimzation step (Section 5.4.3).

   (c) Synthetic ground reference pairs are temporarily created where two images overlap, and at least one has a camera with a known model (Section 5.4.4). These are used to initialize camera parameter estimates in future iterations of Step 4a.

5. ... until there are no camera poses given new estimates in Step 4a.

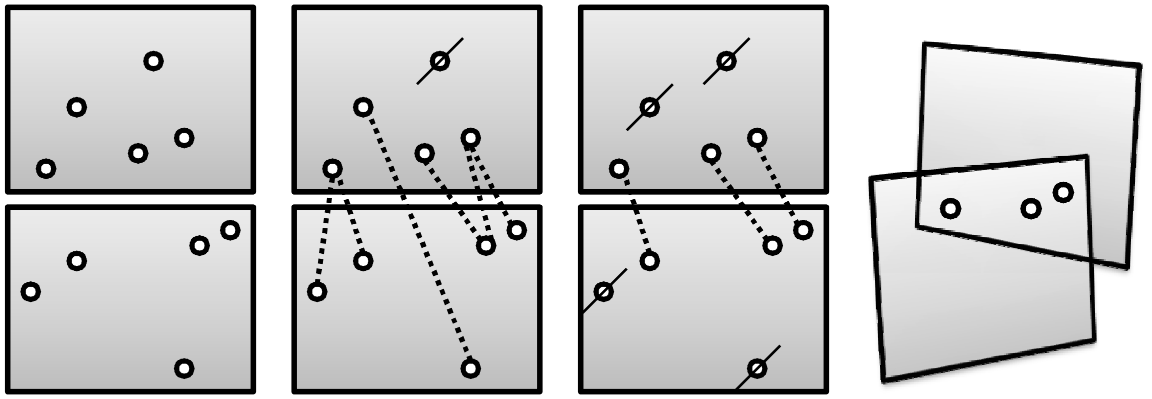
5.4.1 Automatic Extraction and Matching of Feature Points

MapStitcher uses Multi-Scale Oriented Patches (MOPs) [6] to identify corresponding features in the overlapping portions of adjacent images. MOPs can robustly identify features in common across images, even if they vary in scale, orientation and intensities.

The extraction of feature-matches is a five step process:

1. Interest points are identified (Figure 66(a)) on each image separately as local maxima of a "corner strength" function. The orientation of interest points is also computed.

2. The number of interest points is reduced for each image, while a uniform distribution of point locations on the image is maintained. The goal of this step is to reduce the



(a) Feature points. (b) Feature matches. (c) Geometrically consistent feature matches after outlier rejection. (d) Transformed images based on established tie points.

Figure 66: Automatic establishment of feature match point correspondences between two images.

total number of interest points, since the computational requirements for matching are superlinear.

3. A 64-dimensional feature descriptor vector is computed for each remaining interest point using the local image structure.

4. The lowest three non-zero wavelet frequencies of the feature vectors are used to create a three dimensional hash-table. This hash-table provides fast lookup for feature points. Fast approximate feature matching is performed by lookups in this hash-table: a set of approximately matching feature points are found – across all images – for each feature point. Some of the matches are eliminated as outliers using a simple heuristic (Figure 66(b)).

5. Finally, RANSAC is applied to remove additional outliers, by finding geometrically consistent feature matches (Figure 66(c)).

We refer the reader to [6] for specific details of the algorithm.

After the feature matching step is complete, MapStitcher has a list of *feature point* *matches* (Figure 66(d))—that is, pairs of points on overlapping photos that visually correspond to the same features on the ground.

5.4.2 Camera Parameter Initialization

Nonlinear estimation algorithms converge most reliably when given an initial estimate in the neighborhood of the final answer. Therefore, we estimate each camera's parameters before starting bundle adjustment.

The camera extrinsics (rotation and translation) for each image are initialized by performing RANSAC [14] on two sets of points: the ground-point and image-point half of each ground reference pair. First, the inverse of the post-transformation matrix is applied to the image points, to ensure correct centering ($M_{post}^{-1} \cdot p_{image}$). Second, the pre-transformation matrix is applied to the ground points, to ensure correct scaling ($M_{pre} \cdot p_{ground}$). Finally, RANSAC is preformed between these two sets of points, resulting in a transformation matrix for each image, that is then used as the first estimation in the bundle adjustment algorithm.

The camera intriniscs (i.e. the focal length) are directly initialized from the EXIF metadata fields recorded in the image file by the actual camera. If EXIF information is unavailable, we assume the image was taken with a 40° angle of view.

5.4.3 Optimization Using Bundle Adjustment

Once camera models have been given initial estimates, they are refined using an iterative nonlinear optimization process called *bundle adjustment* [11]. Given a number of parameters to adjust (known in bundle-adjustment terminology as *active states*), and an error metric based on those parameters, a bundle adjuster iteratively makes small updates to the parameters until the error metric falls below a threshold.

As discussed in previous sections, MapStitcher has two types of constraints: constraints that pull images towards their correct geography and constraints that place images to minimize seams at their overlap points. These two constraints are represented by two different types of error metrics to the bundle adjuster.

The representation of the geographic constraints are straightforward. The camera intrinsics and extrinsics are represented as active states. The user-supplied ground reference pairs are used to compute the error metric. MapStitcher computes the projection of the ground point into the image plane using the hypothesized camera parameters. The distance from the projected ground point to the user-selected image point is the error.

Image-stitching constraints are somewhat more complex to model. In this case, the stitcher does not have a known ground point—only a set of image points that, according to the feature matcher (Section 5.4.1), depict the same ground feature. We add a new active state for each group of feature match points; it represents the hypothesized point on the ground depicted by those features. The initial estimate of this ground point is the centroid of the projection of all the feature match points onto the ground, given the estimates of those images' camera models. In each iteration of the bundle adjuster, the hypothetical ground point is projected back into the image plane of each image using the updated camera models. The error metric is the sum (over each image) of the distances in image space from these projections to the corresponding feature match points.

For further technical details, we refer the reader to [5], which describes the application of the bundle adjustment algorithm in a similar context.

5.4.4 Grounding Images Iteratively

If the user originally supplies ground reference pairs for *every* image in the mosaic, the procedure described above will work in a single step. Each camera's parameters could be initially estimated based on its image's ground reference pairs, and all parameters could be optimized in a single bundle-adjustment operation. However, such a system would be difficult to use: it can be time-consuming to find ground reference pairs manually and many mosaics contain dozens or hundreds of images. To minimize user effort, MapStitcher creates synthetic ground reference pairs using adjacent overlapping images that already have camera model estimates.

For example, imagine that our mosaic has images A and B. A has user-supplied ground reference pairs, but B does not. The feature matching algorithm tells us that pixel (A_x, A_y) in image A depicts the same feature as pixel (B_x, B_y) in image B. MapStitcher first "bootstraps" the mosaic using A's ground reference pairs to estimate A's camera parameters. It then uses those parameters to project (A_x, A_y) onto a ground point (A_{xg}, A_{yg}), and creates a synthetic ground reference pair for image B: (B_x, B_y) corresponds to (A_{xg}, A_{yg}). This technique can be used iteratively to propagate camera model estimates to an entire contiguous set of overlapping images. We call this successive propagation the *ripple algorithm*. Note that after each ripple, a *global* bundle adjustment is performed, as described in the previous section.

An example for a succession of ripple steps is shown in Figure 5.4.4. (For illustrative purposes, we depict only a small number of feature match points.) In the initial ripple, ground reference pairs (marked (i) on Figure 67(a)) are used to calculate the homographic transformations for image #2 and #9.

In the second ripple, feature match point pairs (marked (ii) on Figure 67(b)) are found that have one of their points on the known-model images: #2 and #9. These feature matches add images #1, #3 and #8 to the ripple. Note that although #1 and #3 overlap, the feature extraction and matching algorithm didn't find any feature match points between them in this case. The ground location of the feature match points are calculated using the homographic transformation obtained for image #2 and #9 in the initial ripple. After bundle adjustment, the ripple's three new images will also have their parameters for homographic transformation.

In the third ripple (Figure 67(c)), feature match points on images #3 and #8 add images #4, #5 and #7 to the ripple. Note that the two feature match points between the floating images #4 and #5, marked (iv), are feature match points without at least one image with a known position, and thus are not used in the RANSAC initialization of the third ripple.

In the fourth ripple (Figure 67(d)), feature match points attach image #6 to both #5 and #7. Note that up until this ripple, there were two independent image groups: images #1–#5 were grounded based on ground reference pairs from image #2, and images #7–#9 were grounded based on ground reference pairs from image #9. The link provided by #6 joins these two groups, and the subsequent bundle adjustment jointly refines all 9 camera models *together* for the first time in search of a globally optimal solution. In addition, the feature match points marked (iv) between #4 and #5 can now be grounded (using the homographic projections from the previous ripple), which allows them to be used in the bundle adjustment. After the fourth ripple, all images in the cluster are grounded with homographic transformations, and the algorithm terminates.

5.5 Evaluation

MapStitcher is designed to produce a well-georeferenced aerial imagery layer stack with low human data-entry cost. To evaluate its design, we perform an experiment that compares a MapStitcher orthorectified image with two control methods, *no-stitch* and *stitch-first*. We measure each method on two criteria: cost of registration measured in number of manual ground reference pairs, and quality of registration measured in deviation of unreferenced points from ground truth. In these experiments, “ground truth” is defined by the lower-resolution Virtual Earth aerial photography of the subject region, and is affected by distortions in the Virtual Earth orthorectification pipeline.

5.5.1 Experiment Description

For this experiment, we use as input 60 source images we captured of the Skagit River Valley in the vicinity of the town of Concrete, WA. We used each of the three techniques to combine all source images to produce a single orthorectified, tiled composite image of 208 megapixels.

5.5.2 Measuring Cost

For the *no-stitch* method, we registered 257 points (mean 4.3 points per image; Figure 68(a)).

For the *stitch-first* method, we stitched the images with the fully automatic photo stitcher described in [4]. We georegistered the resulting composite image with 25 manually-entered ground reference pairs (Figure 68(b)).

For the MapStitcher method, we registered 25 points spread out on 5 images (mean 0.4 points per image over the whole set; Figure 68(c)). We show the five images with manually entered ground reference pairs after transformation, on Figure 69, while Figure 62 shows all 60 images georegistered based on these five images.

Figure 70 shows the number of manual ground reference pairs for the three methods.

5.5.3 Measuring Quality

We manually selected 12 recognizable points in the scene, each from separate source images, none of which were used as manually-entered reference points in any of the methods. We measured the "ground truth" position of each point in the low-resolution Virtual Earth image. For each method, we computed the mean distance between where the method geolocates each point versus the point's ground truth position.

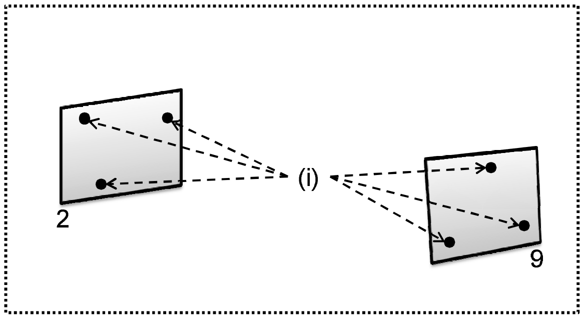
Figure 71 shows the mean and standard deviation of the registration errors for the three methods. The *no-stitch* method produces the best quality orthorectification, with $25.1\ m$ mean error and $15.8\ m$ standard deviation, but using 10.3 times as many manual points as the other methods. The reference *stitch-first* method results in a mean error of $354.1\ m$ (with a large $167.9\ m$ standard deviation), showing that it is difficult to recover geography as a discrete step if a mosaic is created using seamless-boundary constraints alone. Our method, which jointly optimizes image-to-ground and image-to-image alignment, results in a mean error that is 234% ($58.83\ m$) of the no-stitch method (with a standard deviation of $37.9\ m$), while needing only 9.7% of the manually entered ground reference pairs of the latter method. The increased error may be due to placing too much relative weight on image-to-image alignment—that is, in some cases, we may be sacrificing absolute positional accuracy for the sake of output that looks better.

5.6 Future Work

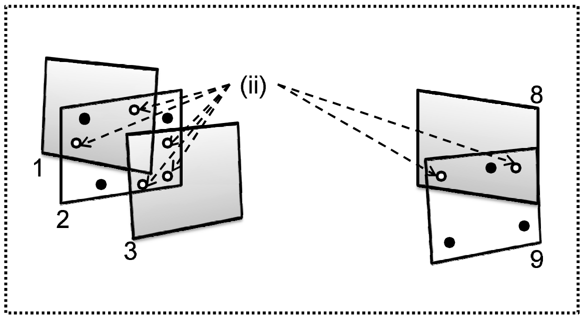
While our current system produces composite imagery whose georeferencing quality approaches that of the manual no-stitch method, it suffers from similar problems as that method: image boundaries remain clearly visible at some image boundaries. Panorama stitching techniques employ graphcut algorithms to reduce visible seams in the final composite [25], and gain compensation and multi-band blending is used to correct for unmodelled camera effects (e.g. vignetting) [4]. Our application would also benefit from these techniques. MapStitcher currently has no *a priori* information about the relative positions of any images, and thus must attempt to find feature matches between all image pairs. Adding a constraint that indicates potential image overlaps will simplify the problem of finding feature matches, as the number of candidate images to be considered will be reduced from $O(n^2)$ to a constant-sized neighborhood. This will significantly improve processing speed and reduce feature match outliers, and can be achieved using a low-cost (consumer-grade) GPS that is only loosely coupled to the image acquisition process.

5.7 Conclusion

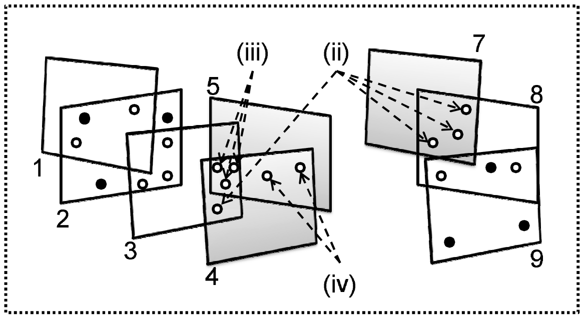
MapStitcher produces orthorectified aerial imagery mosaics from images with poorly constrained geometry and only minimal manual labeling. The result is a system with low capital cost that produces high-quality image mosaics. We anticipate that access to such low-cost imaging will lead to a much wider grass-roots effort to produce aerial photography. We hope to facilitate community-supported efforts aimed, for example, at better coverage of non-urban areas, timely coverage of special events or natural disasters, or more frequent coverage of fast-changing areas. Ultimately, if aerial imaging becomes as cheap and easy to produce as a blog, we may see aerial imagery with the same rich, decentralized diversity as the blogosphere.



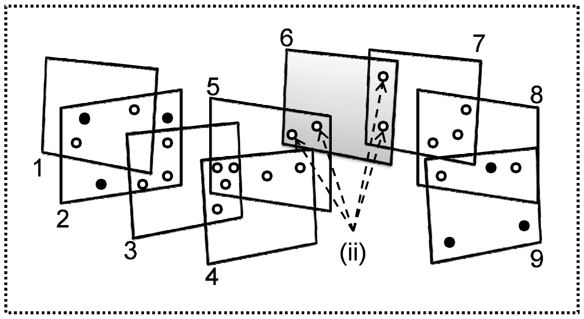
(a) Initial ripple; only ground reference pairs are used



(b) Second ripple; feature match points link some floating images to already grounded ones

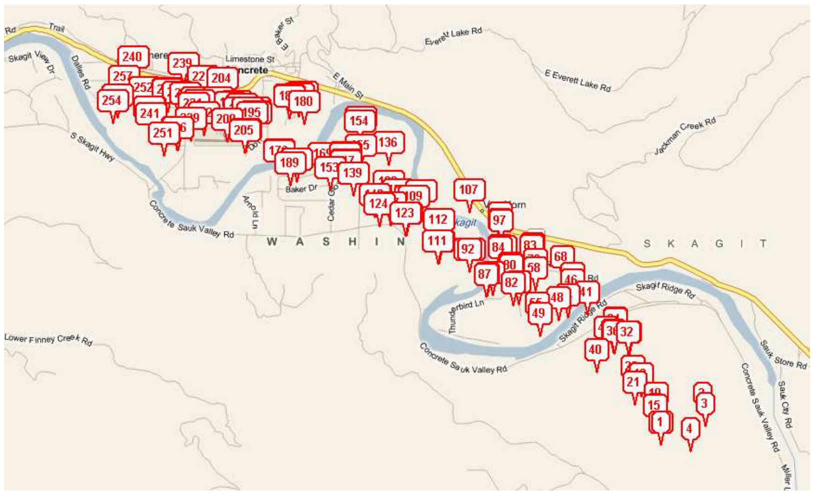


(c) Third ripple; some feature match points link more than two images

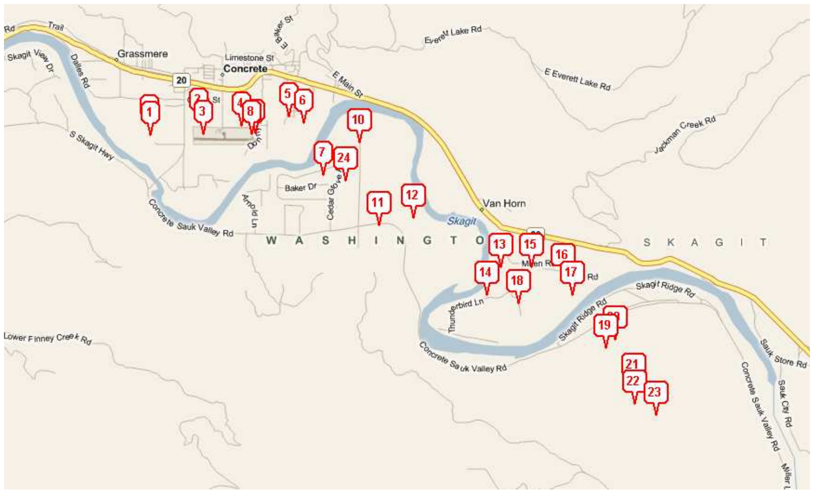


(d) Fourth and final ripple; a globally optimal solution is approached when independently estimated image groups join

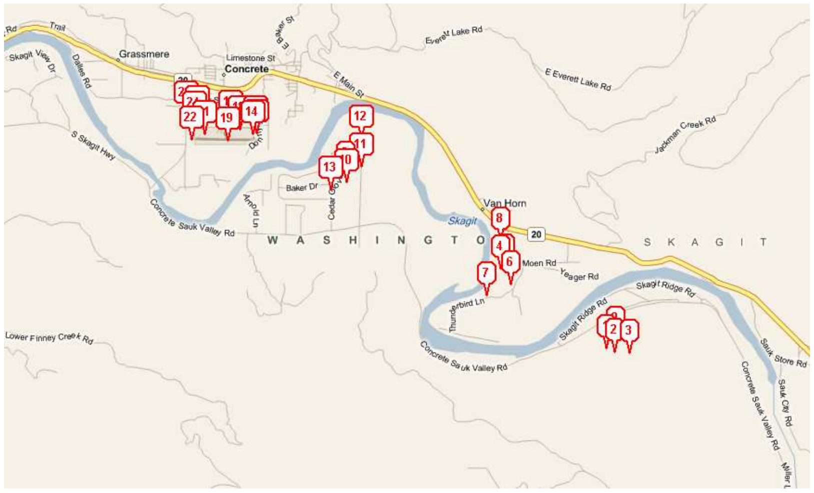
Figure 67: A succession of ripples is used to estimate the position of all images, even though only a subset have user-specified ground reference pairs.



(a) Ground reference points for *no-stitch* method.



(b) Ground reference points for *stitch-first* method.



(c) Ground reference points for MapStitcher method.

Figure 68: Locations of ground reference points.

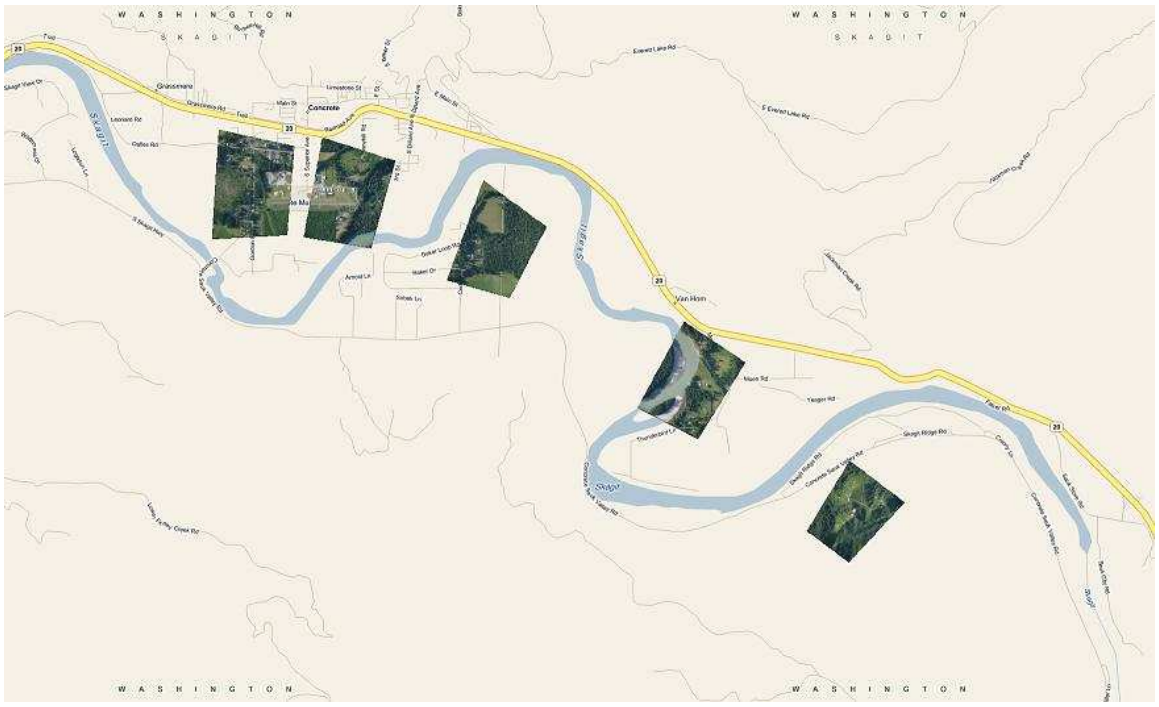


Figure 69: The only images with manually entered ground reference pairs in our MapStitcher example.

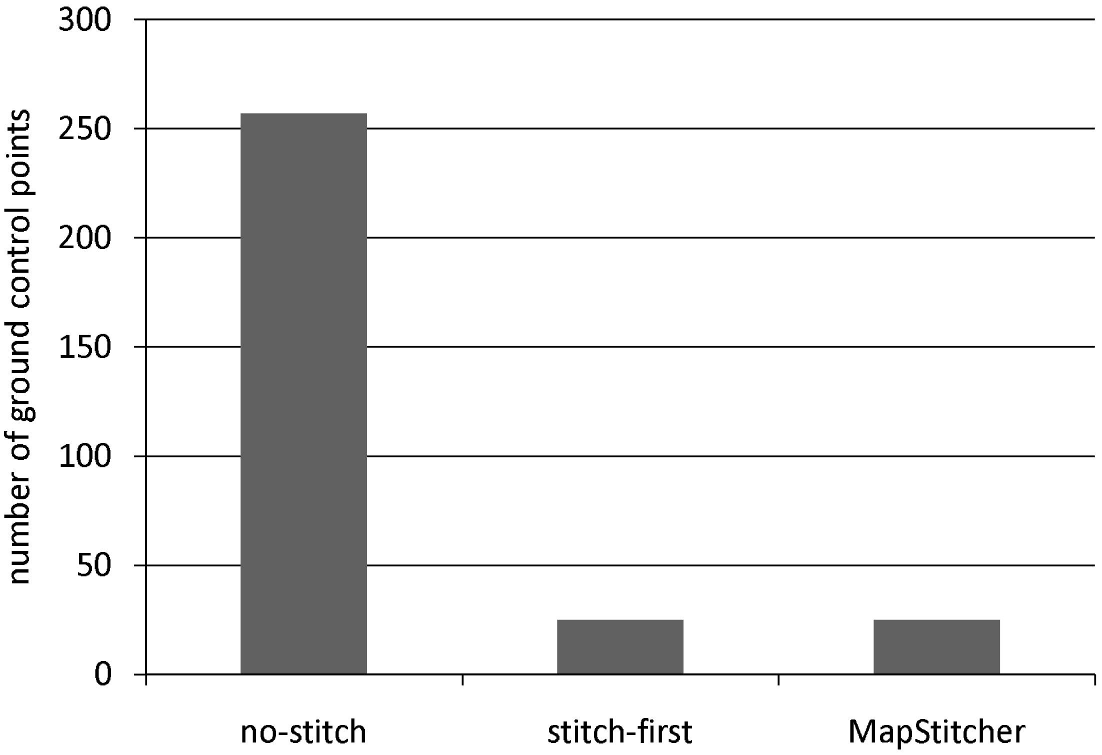


Figure 70: Number of manually entered ground reference pairs.

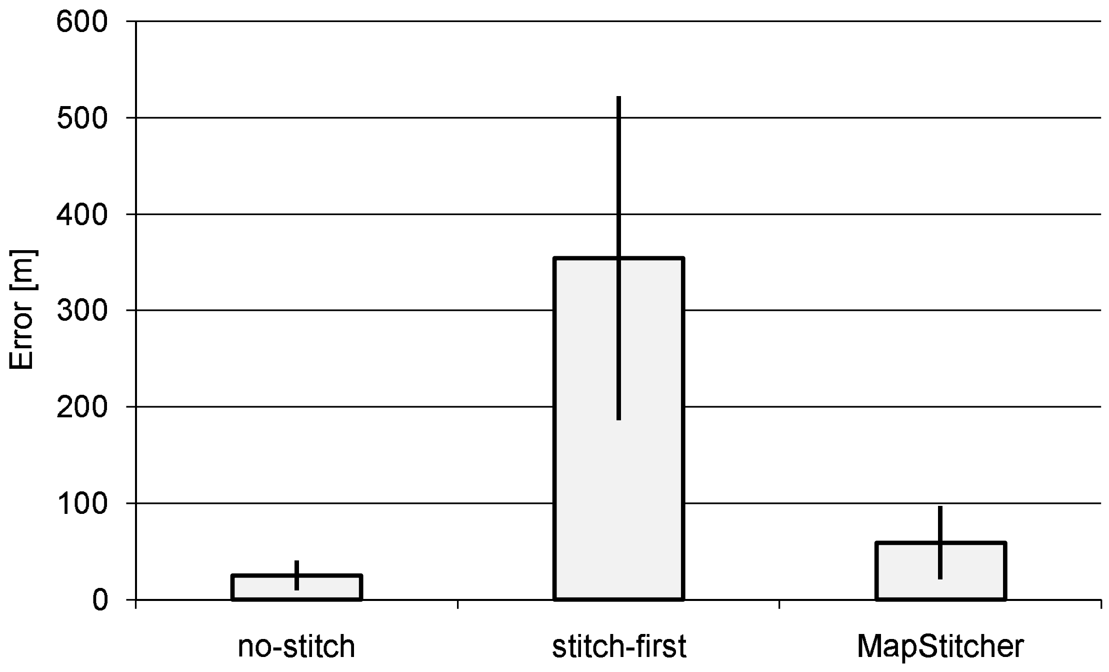


Figure 71: Mean and standard deviation of registration error.

CHAPTER VI

CONCLUSION

The theme of this thesis is the creation of location based services that are efficient, scalable and available to all end users. In this chapter, we provide a recap of the main topics and conclusions of our work.

We have presented ROADTRACK – a query-aware, precinct based location update framework for scaling location updates and location tracking services. ROADTRACK development makes three original contributions. First, we introduce encounter points as a fundamental query awareness mechanism enable us to control and differentiate location update strategies for mobile clients in the vicinity of active location queries. Second, we employ system-defined precincts to manage the desired spatial resolution of location updates for all mobile clients and to control the scope of query awareness capitalized by a location update strategy. Third but not the least, we develop a road network distance based check-free interval optimization, which further enhances the effectiveness of ROADTRACK and enables us to effectively manage location updates of mobile clients traveling on road networks by minimizing the unnecessary checks of whether they have crossed an encounter point or precinct boundary point. We evaluate the ROADTRACK location update approach using a real world road-network based mobility simulator. Our experimental results show that the ROADTRACK query aware, precinct-based location update strategy outperforms existing representative location update strategies in terms of both client computation efficiency and server update load.

The computational costs of answering continuous network range queries are known to be prohibitively high, as a shortest path based network expansion needs to be run repeatedly at each and every location where the query is evaluated. We argue that continuous network range queries, whose focal locations are ”not far” from each other, have substantial overlap in their segment coverage. Such a large overlap may offer significant reuse opportunities for performance enhancement. We have presented the design and implementation of Dandelion reuse framework and a suite of algorithms for fast re-evaluations of continuous network range queries. The chapter makes three original contributions. First, we propose the concept of Dandelion tree to accurately represent the coverage of a network range query with arbitrary range, by keeping track of three key network location points: border points (BOP), dead-end points (DEP), and zip points (ZIP). Second, we design three BOP-Push and three BOP-Pull primitive operations to compute the coverage at F by maximum reuse of the coverage at previous query focal location F. Third but not the least, we define the data structures and three Dandelion reuse algorithms to efficiently identify the portion of the Dandelion tree that can be used as the basis for reuse and further expansion. The basic Dandelion algorithm enables reuse by dividing the Dandelion tree (query coverage) of a query into the forward (FWD) and backward (BWD) halves, allowing separate maintenance of the key data structures for each half to reduce the search space. The Dandelion-T algorithm introduces and utilizes the Trident and Guide data structures to compose a more reuse-efficient Dandelion-T tree, leading to faster query re-evaluation than Dandelion basic algorithm. Finally the Dandelion2 algorithm further enhances Dandelion-T in terms of query re-evaluation cost by introducing the two primitive transformation operations *move* and *jump*. This development can effectively transform one Dandelion tree to another with a minimum set of primitive transformation operations. We conduct a series of extensive experiments and our results show that Dandelion reuse model and algorithms can significantly outperform the conventional shortest path network expansion algorithm (NE) in terms of coverage computation cost for non-trivial radius size and high re-evaluation frequency.

We also considered the problem of accelerating the computation of range query coverages in road networks, even when the query is only evaluated a single time, and thus a reuse-oriented approach is not applicable. We presented our approach of constructing precincts over the road network graph to eliminate the unnecessary complexity of local neighborhood streets and replaced them with fast shortcuts. We provided a classification of precincts into seed, core and border types, and a criterion to determine when the coverage computation should choose local search instead of shortcut based search.

Finally, in a different flavor of location based services, but continuing our focus on applicability to realistic scenarios, MapStitcher produces orthorectified aerial imagery mosaics from images with poorly constrained geometry and only minimal manual labeling. The result is a system with low capital cost that produces high-quality image mosaics. We anticipate that access to such low-cost imaging will lead to a much wider grass-roots effort to produce aerial photography. We hope to facilitate community-supported efforts aimed, for example, at better coverage of non-urban areas, timely coverage of special events or natural disasters, or more frequent coverage of fast-changing areas. Ultimately, if aerial imaging becomes as cheap and easy to produce as a blog, we may see aerial imagery with the same rich, decentralized diversity as the blogosphere.

We believe that location based services – while available to users in many forms today – are still an area in its infancy. We hope that this thesis can be a useful contribution to the furtherance of knowledge in this field.

REFERENCES

[1] BAR-NOY, A., KESSLER, I., and SIDI, M., “Mobile users: to update or not to update?,” *Wireless Networks*, vol. 1, no. 2, pp. 175–185, 1995.

[2] BAST, H., FUNKE, S., and MATIJEVIC, D., “TRANSIT – ultrafast shortest-path queries with linear-time preprocessing,” in *9th DIMACS Implementation Challenge*, 2006.

[3] BAST, H., FUNKE, S., MATIJEVIC, D., SANDERS, P., and SCHULTES, D., “In transit to constant time shortest-path queries in road networks,” in *ALENEX*, SIAM, 2007.

[4] BROWN, M. and LOWE, D. G., “Automatic panoramic image stitching using invariant features,” *International Journal of Computer Vision*, vol. 74, pp. 59–73, Aug. 2007.

[5] BROWN, M. and LOWE, D. G., “Unsupervised 3D object recognition and reconstruction in unordered datasets,” in *3DIM*, pp. 56–63, IEEE Computer Society, 2005.

[6] BROWN, M., SZELISKI, R., and WINDER, S., “Multi-image matching using multi-scale oriented patches,” in *CVPR*, pp. 510–517, IEEE Computer Society, 2005.

[7] CAI, Y., HUA, K. A., and CAO, G., “Processing range-monitoring queries on heterogeneous mobile objects,” in *IEEE MDM*, 2004.

[8] CAI, Y., HUA, K. A., CAO, G., and XU, T., “Real-time processing of range-monitoring queries in heterogeneous mobile databases,” *Proc. IEEE Trans. Mob. Comput*, vol. 5, no. 7, pp. 931–942, 2006.

[9] CHO, H.-J. and CHUNG, C.-W., “An efficient and scalable approach to CNN queries in a road network,” in *VLDB*, 2005.

[10] CIVILIS, A., JENSEN, C. S., and PAKALNIS, S., “Techniques for efficient road-network-based tracking of moving objects,” *Proc. IEEE TKDE*, vol. 17, no. 5, pp. 698–712, 2005.

[11] DEWITT, B. A. and WOLF, P. R., *Elements of Photogrammetry (with Applications in GIS).* McGraw-Hill Higher Education, 2000.

[12] DIJKSTRA, E. W., “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[13] ELSON, J., HOWELL, J., and DOUCEUR, J. R., “Mapcruncher: integrating the world’s geographic information,” *Operating Systems Review*, vol. 41, no. 2, pp. 50–59, 2007.

[14] FISCHLER, M. A. and BOLLES, R. C., “Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography,” *Commun. ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[15] GEDIK, B. and LIU, L., “Mobieyes: A distributed location monitoring service using moving location queries,” *Proc. IEEE Trans. Mobile Computing*, vol. 5, no. 10, pp. 1384–1402, 2006.

[16] GEISBERGER, R., SANDERS, P., SCHULTES, D., and DELLING, D., “Contraction hierarchies: Faster and simpler hierarchical routing in road networks,” in *WEA*, pp. 319–333, 2008.

[17] HARTLEY, R. I. and ZISSERMAN, A., *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second ed., 2004.

[18] HEUVEL, F. A. V. D., “Exterior orientation using coplanar parallel lines,” *Proceedings of the 10th Scandinavian Conference on Image Analysis*, pp. 71–78, 1997.

[19] HEUVEL, F. A. V. D., “Estimation of interior orientation parameters from constraints on line measurements in a single image,” *International Archives of Photogrammetry and Remote Sensing*, vol. 32, pp. 81–88, 1999.

[20] HU, H., LEE, D. L., and LEE, V. C. S., “Distance indexing on road networks,” in *VLDB*, 2006.

[21] HU, H., LEE, D. L., and XU, J., “Fast nearest neighbor search on road networks,” in *EDBT*, 2006.

[22] HU, H., XU, J., and LEE, D. L., “A generic framework for monitoring continuous spatial queries over moving objects,” in *SIGMOD*, 2005.

[23] JONES, K. and LIU, L., “What Where Wi: An analysis of millions of wi-fi access points,” in *IEEE PORTABLE*, 2007.

[24] KOLAHDOUZAN, M. R. and SHAHABI, C., “Voronoi-based K nearest neighbor search for spatial network databases,” in *VLDB*, 2004.

[25] KWATRA, V., SCHÖDL, A., ESSA, I. A., TURK, G., and BOBICK, A. F., “Graphcut textures: image and video synthesis using graph cuts,” *ACM Trans. Graph*, vol. 22, no. 3, pp. 277–286, 2003.

[26] MAHAMUD, S., HEBERT, M., OMORI, Y., and PONCE, J., “Provably-convergent iterative methods for projective structure from motion,” in *CVPR*, pp. 1018–1025, IEEE Computer Society, 2001.

[27] MCLAUCHLAN, P. F. and JAENICKE, A., “Image mosaicing using sequential bundle adjustment,” *Image Vision Comput*, vol. 20, no. 9-10, pp. 751–759, 2002.

[28] MIYAMOTO, Y., UNO, T., and KUBO, M., “Levelwise mesh sparsification for shortest path queries,” in *ISAAC (1)*, pp. 121–132, 2010.

[29] MOURATIDIS, K., PAPADIAS, D., BAKIRAS, S., and TAO, Y., "A threshold-based algorithm for continuous monitoring of k nearest neighbors," *Proc. IEEE TKDE*, vol. 17, no. 11, pp. 1451–1464, 2005.

[30] MOURATIDIS, K., YIU, M. L., PAPADIAS, D., and MAMOULIS, N., "Continuous nearest neighbor monitoring in road networks," in *VLDB*, 2006.

[31] MURUGAPPAN, A. and LIU, L., "An energy efficient approach to processing spatial alarms on mobile clients," in *Proc. of ISCA International Conference on Software Engineering and Data Engineering*, 2008.

[32] NARVÁEZ, P., SIU, K.-Y., and TZENG, H.-Y., "New dynamic algorithms for shortest path tree computation," *IEEE/ACM Trans. Netw.*, vol. 8, no. 6, pp. 734–746, 2000.

[33] NARVÁEZ, P., SIU, K.-Y., and TZENG, H.-Y., "New dynamic SPT algorithm based on a ball-and-string model," *IEEE/ACM Trans. Netw.*, vol. 9, no. 6, pp. 706–718, 2001.

[34] P. PESTI, B. BAMBA, M. DOO, L. LIU, B. PALANISAMY AND M. WEBER, "GT-MobiSIM: A mobile trace generator for road networks." College of Computing, Georgia Inst. of Tech., Sept. 2009. http://code.google.com/p/gt-mobisim/.

[35] PAPADIAS, D., ZHANG, J., MAMOULIS, N., and TAO, Y., "Query processing in spatial network databases," in *VLDB*, 2003.

[36] PESTI, P., ELSON, J., HOWELL, J., STEEDLY, D., and UYTTENDAELE, M., "Low-cost orthographic imagery," in *ACM GIS*, 2008.

[37] PESTI, P., LIU, L., BAMBA, B., IYENGAR, A., and WEBER, M., "RoadTrack: Scaling location updates for mobile clients on road networks with query awareness," in *VLDB*, 2010.

[38] PRABHAKAR, S., XIA, Y., KALASHNIKOV, D. V., AREF, W. G., and HAMBRUSCH, S. E., "Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects," *Proc. IEEE Trans. Computers*, vol. 51, no. 10, pp. 1124–1140, 2002.

[39] QUAN, L. and LAN, Z.-D., "Linear N-point camera pose determination," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 21, no. 8, pp. 774–780, 1999.

[40] SANDERS, P. and SCHULTES, D., "Highway hierarchies hasten exact shortest path queries," in *ESA* (BRODAL, G. S. and LEONARDI, S., eds.), vol. 3669 of *Lecture Notes in Computer Science*, pp. 568–579, Springer, 2005.

[41] SCHULTES, D., *Route Planning in Road Networks.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, Institut für Theoretische Informatik, 2008.

[42] SKYHOOK WIRELESS, "Hybrid Positoning System (XPS)." http://www.skyhookwireless.com/howitworks/.

[43] U.S. Census Bureau, "TIGER/Line Shapefiles." http://www.census.gov/geo/www/tiger/.

[44] Vandeportaele, B., Dehais, C., Cattoen, M., and Marthon, P., "ORIENT-CAM, A camera that knows its orientation and some applications," in *CIAPR* (Trinidad, J. F. M., Carrasco-Ochoa, J. A., and Kittler, J., eds.), vol. 4225 of *Lecture Notes in Computer Science*, pp. 267–276, Springer, 2006.

[45] Xing, S., Shahabi, C., and Pan, B., "Continuous monitoring of nearest neighbors on land surface," *PVLDB*, vol. 2, no. 1, pp. 1114–1125, 2009.

[46] Yu, X., Pu, K. Q., and Koudas, N., "Monitoring K-nearest neighbor queries over moving objects," in *ICDE*, 2005.

[47] Zhang, J., Zhu, M., Papadias, D., Tao, Y., and Lee, D. L., "Location-based spatial queries," in *SIGMOD*, 2003.

[48] Zheng, B. and Lee, D. L., "Semantic caching in location-dependent query processing," in *SSTD*, 2001.

VITA

Peter Pesti was brought up in Budaörs, near the capital city of Hungary. He started his undergraduate education in 2000, and his research career in speech synthesis in 2003 at the Budapest University of Technology and Economics, and received an M.Sc. in Technical Informatics there in 2006. He simultaneously enrolled at the Georgia Institute of Technology in 2004 with the sponsorship of the Naumann-Etienne Foundation's full scholarship, and received an M.S. in Computer Science from Tech in 2006. He continued at Georgia Tech in the pursuit of his dissertation research in location-based services and systems, working under the guidance of Prof. Ling Liu in the Distributed Data Intensive Systems Lab at the College of Computing. He has done internships multiple times at both Microsoft Research and Google, working on mapping applications and large-scale processing tools for location-related data, with some of this work receiving a patent. His list of honors includes receiving the Scholarship of the Hungarian Republic, being selected for the GE Foundation Scholar-Leaders Program and an invitation to a McKinsey & Company EuroAcademy event.