Path Problems in Geographic Information Systems

by

Masoud Taghinezhad Omran

A thesis submitted to the

Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Carleton University

Ottawa, Ontario

The undersigned hereby recommends to the

Faculty of Graduate and Postdoctoral Affairs

acceptance of the dissertation

Path Problems in Geographic Information Systems

submitted by **Masoud Taghinezhad Omran**

in partial fulfillment of the requirements for the degree of

Computer Science

Professor "Jörg-Rüdiger Sack", Thesis Supervisor

Professor "Anil Maheshwari", Department Examiner

Professor "Thomas Kunz", Internal Examiner

Professor "Nejib Zaguia", OCICS Examiner

Professor "Carola Wenk", External Examiner

Professor Michel Barbeau, Director,
School of Computer Science
Carleton University

Abstract

Path problems are among the most widely studied problems in graph theory. Various versions of the problem with distinct real world applications have been proposed and studied (Shortest Path, Longest Path, Hamiltonian Path, Traveling Salesperson Problem, among others). The shortest path problem for a network (or directed graph) is a fundamental problem in graph theory with applications related to Navigation/Transportation Systems, Computer Networks, VLSI Design, Social Networks, and many other networks. Unlike in well-studied conventional *static* shortest path problems, the characteristics of underlying networks may change over time in many real-world applications. An efficient solution to the Shortest Path problem, should therefore take the dynamics of the underlying graph into consideration. In this thesis, we study a dynamic version of the Shortest Path Problem in which travel times on links of the underlying graph change over time, and present improved and easily implemented exact and approximation algorithms.

With theoretical improvements in time-dependent shortest path computation and technical advancements in the provision of real-time traffic data, many travelers these days use and rely on navigation systems and time-dependent shortest path computation in their everyday lives. For example, GoogleMaps provides advice on best routes by using current traffic conditions and historical route data to keep users out of traffic jams. This may, however, compromise users' privacy since untrusted Location Based

Services (LBS) are able to use inference attacks to gain access to sensitive information about users, such as their driving habits, health conditions they may have, their sexual orientation, or details about their lifestyle. In this dissertation, we propose a heuristic algorithm that protects the privacy of users who submit frequent location based queries to a Location Based Service.

Although, shortest paths are preferred in most applications related to navigation and transportation, they can easily be inferred and regenerated, and should therefore be avoided in applications where user security is paramount. For example, a traveling VIP is more concerned about the security of her path rather than its length. We study the traveling VIP problem in which the goal is to randomly select a path from k candidate source–destination paths. When links are shared between many paths security is compromised and security guards may have to be placed on such links. The number of shared edges, then, should be minimized to reduce the costs associated with security guards who must be assigned to each shared link. We show that this problem is hard to approximate, and propose an approximation algorithm which has been experimentally evaluated on various networks.

Table of Contents

List of Figures

Chapter 1

Introduction

Path Problems are fundamental problems in graph theory with applications in numerous disciplines, such as, Geographic Informations Systems (GIS), Communication Networks, Robotics, Social Networks, and VLSI design [31]. In this thesis, we study three paths problems in the context of Geographic Information Systems. A Geographic Information System is an information system that is designed to work with spatial or geographical data.

1.1 Time-Dependent Shortest Path Problem

1.1.1 Motivation

Let $G = (V, E)$ be a graph with set of vertices V, set of Edges E, and non-negative real valued edge weight $w(e)$, for $e \in E$. Given source $s \in V$ and destination $d \in V$, a shortest path from s to d is a path between s and d with minimum sum of the weights of its constituent edges. Dijkstra's algorithm [49] computes a shortest path from s to d in $O(m + n \log n)$ time using Fredman and Tarjan's [63] implementation of the heap data structure, where m and n are the number of edges and vertices, respectively, in G. In many applications, however, the underlying network changes over time, and static

graphs do not address all shortest path computation needs. For example, navigation devices are currently capable of receiving and processing on-demand traffic data that can be used to compute shortest paths. Because of their limited processing power, such devices need to exploit pre-computation algorithms which can be inefficient, if time-dependent data changes frequently, and has to be stored for many links. To overcome this, navigation companies have started using historical road data compiled over years and computing a travel-time function for each link of the road network. These functions are proprietary, but allow the pre-computation of time-dependent data.

1.1.2 Problems

We study two variants of the problem [47, 48]. First, we consider networks in which the availability of links, changes over time. For each link, the availability is given by a set of disjoint time intervals. Each interval is assigned a non-negative real value which represents the travel time on the link, during the corresponding interval. The resulting Shortest Path problem is Time-Dependent Shortest Path problem for availability intervals ($\mathcal{TDSP}_{\text{int}}$), which asks to compute all shortest paths to any (or all) destination node(s) d for all possible start times at a given source node s. Second, we study time-dependent networks for which the cost of using a link is given by a non-decreasing (FIFO) piecewise linear function of a real-valued argument. Here, each piecewise linear function represents the travel time on the link based on the time when the link is used. The resulting Shortest Path problem is the Time-Dependent Shortest path problem for piecewise linear functions ($\mathcal{TDSP}_{\text{lin}}$) which asks to compute, for a given source node s and destination d, the shortest paths from s to d, for all possible start times. In transportation applications, a solution to this problem could be used for trip planning to provide a user who can be flexible in their departure time from

a source location with route options from which they could select the one that best suits their schedule.

Recently, it has been shown that the complexity of the $\mathcal{TDSP}_{\text{lin}}$ problem could be super-polynomial (i.e., $O(n^{\Theta(\log n)})$)[60]. An algorithm takes super-polynomial time if its time-complexity is $\omega(n^c)$ for all constants c, where n is the input parameter. Additionally, in most cases, link travel time functions are approximation of reality. Therefore, efficient approximation algorithms with customizable quality of results might be more favorable for the $\mathcal{TDSP}_{\text{lin}}$ problem. This problem has also been studied in this thesis.

1.1.3 Previous Results

For the $\mathcal{TDSP}_{\text{lin}}$ problem, Orda and Rom presented a label-correcting method [124] that has an $O(F_{max}|V||E|)$ time bound where F_{max} is the maximum output size for all possible destination nodes $d \in V$. Dean [46] proposed an improved label-setting algorithm that has a running time of $O(|E|F^* \log |V|)$, where F^* is the total number of pieces among all output functions. Dean [46] also conjectured that it is possible to have super-polynomial output size for the earliest arrival time function on some nodes of the network. Foschini *et al.* [60] proved the conjecture to be true. They also proposed an algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem that has time-complexity $O((\gamma + F_{max}|E|) \log^2 |V|)$, where γ is the input size (total number of linear pieces in arrival time functions on links).

When applied to the $\mathcal{TDSP}_{\text{int}}$ problem, Orda and Rom's algorithm requires time $O(\lambda|V||E|)$, Dean's algorithm requires time $O(\lambda|E| \log |V|)$, and Foschini *et al.*'s algorithm requires time $O(\lambda|E| \log^2 |V|)$ where λ is the input size (total number of linear pieces in arrival time functions on links).

Let t_{min} and t_{max} be the minimum and maximum possible departure time values

of the travel time function, respectively. Foschini *et al.*[60] propose an approximation algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem that runs in $O(\frac{\gamma}{\epsilon}\log(\frac{T_{max}}{T_{min}})\log(\frac{L}{\gamma\epsilon T_{min}}))$ time, where γ is the total number of linear pieces in travel time functions on links, $L = t_{max} - t_{min}$, and T_{min} and T_{max} are the minimum and maximum travel time values, respectively.

1.1.4 New Results

We present an algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem that runs in time $O((F_d+\gamma)(|E|+|V|\log|V|))$, where F_d is the output size (i.e., the number of linear pieces needed to represent the earliest arrival time function to d) and γ is the input size (i.e., the number of linear pieces needed to represent the local earliest arrival time functions for all links in the network). We then solve the $\mathcal{TDSP}_{\text{int}}$ problem in $O(\lambda(|E| + |V|\log|V|))$ time by reducing it to an instance of the $\mathcal{TDSP}_{\text{lin}}$ problem. Here, λ denotes the total number of availability intervals in the entire network. Both methods offer significant running time improvement on the previously developed algorithms [122].

We present two ϵ–approximation algorithms in Chapter 4 that outperform existing approximation solutions for the $\mathcal{TDSP}_{\text{lin}}$ problem. Our first algorithm runs $O(\frac{\gamma}{\epsilon}(\log(\frac{T_{max}}{T_{min}}) + \log(\frac{L}{\gamma T_{min}})))$ shortest path computations at fixed departure times. In our second algorithm, we reduce the dependency on L, by using only $O(\gamma(\frac{1}{\epsilon}\log(\frac{T_{max}}{T_{min}}) + \log(\frac{L}{\gamma\epsilon T_{min}})))$ total shortest path computations.

1.2 Protecting Privacy from Inference Attacks in Location Based Services

1.2.1 Motivation

Using traffic data, Location Based Services (LBS) can provide more accurate navigation information. This, along with other advances in on-line LBS technology (the ability to search a nearby area for a facility, for example, or a street view capability to provide a more realistic user experience), and Internet availability on various user devices (cell phones and tablet-PCs), is not only convenient for users throughout their daily lives, but also saves them time and money. But location information, when misused, can be a valuable source for malicious agents. LBS users who are not protected against location-based privacy attacks might be in danger of crucial personal information (about lifestyle, health, and sexual orientation) being accessible to malicious parties (such as spies, unsolicited marketers and spammers). Protecting LBS users from privacy breaches, has thus recently become an important subject for research activity (for recent results on location privacy protection see, for example, [44, 135, 140]).

As the frequency of LBS requests increases, adversaries (could be the LBS provider) can infer more private information about users. Users who submit frequent location-based service queries to an untrusted LBS provider at sensitive locations (hospitals, for example, or military zones, or night-clubs) may expose not only their identities, but also details about their lifestyles. For example, consider a user who uses an LBS, on a weekday, to get directions from her home to a hospital, and then from the hospital to a pharmacy, and finally from the pharmacy to her home. In this case, not only the LBS can use publicly available residential information to identify

the issuer of the queries, but it can also infer that she has an illness and cannot go to work.

1.2.2 Problems

We study the location privacy problem for cases in which adversaries have access to external information about road-networks, like speed limits and historical travel times on road-network links, as well as to additional information, such as the residential/commercial address directories and to interval times between user queries.

1.2.3 Previous Results

Well-known approaches to concealing a user's current location include hiding the identity among other fake/real user locations, deliberately degrading user's location information, and encrypting the query. These lead to three types of privacy protection strategies: anonymization [17, 66, 94], obfuscation/cloaking [10, 20, 52, 114], and cryptography [70, 155]. For an overview of the state-of-the-art in location privacy protection for sporadic query LBS, see [68]. However, the above privacy protection solutions may result either in privacy breaches or in degraded performance. Duckham *et al.* [53] were the first to introduce such a privacy threat scenario by pointing out a possible attack strategy employing maximum, minimum, or average speed of movement information. [34, 69, 148] study this issue and propose privacy protection solutions using the cloaking-based protection approach. Recently, Pan *et al.* [126] added a k-anonymity privacy protection approach to further improve previous methods by protecting user privacy against identity theft. In [33], a more general approach for protecting user privacy is presented by studying the impact of query dependency on user privacy. Cryptography-based location protection methods might provide a

better privacy protection, but they suffer from poor performance, and require specific infrastructure, which may not be supported by all LBS providers [67].

1.2.4 New Results

We introduce the notion of (i, j)-privacy [120] which generalizes previous privacy models (see Chapter 5). We design several heuristics implementing (i, j)-privacy. These heuristics still provide accurate responses to user queries, such as shortest routes or nearest neighbors computations. We evaluate our algorithms experimentally on different road-networks by varying a number of input parameters and present the results. Our experiments show that, for realistic user settings, our algorithms preform well in practice meaning that they are fast, enable accurate LBS query results, and guarantee optimal or close to optimal (i, j)-privacy in most cases.

1.3 Finding Paths with Minimum Shared Edges

1.3.1 Motivation

Despite the fact that in many applications shortest paths are widely used and usually preferred, in some applications (related to GIS and Communication Networks, for example), paths other than the shortest path might be favorable. For example, consider a VIP who wants to travel between a pair of points in a road network while avoiding, wherever possible, attacks along the route. A common strategy to avoid (terrorist) attacks is to determine k candidate (s, d)-paths (possibly not the shortest paths, to increase security) and then select one path randomly from among them. However, a security issue which has not been studied in the literature before may arise in this scenario. An attacker who is cognisant of the above security strategy

could analyze the network topology and determine a vulnerable section along the travel route, such as links/edges that are shared among several or most (s, d)-paths. Consider, for example, s and d to be at different sides of a river; crossing bridges is therefore inevitable. Consider also certain transportation media, such as a railway, in which certain routes are mandatory. To overcome these issues and to guarantee a desired level of security, network edges used in more than one (s, d)-paths are considered vulnerable and have to be patrolled/guarded. Since the cost of hiring security guards is high, finding k (s, d)-paths with minimum number of shared (vulnerable) edges is a desirable strategy. The same problem may appear in applications related to Communication Networks, where shared links require additional bandwidth and need to be minimized.

1.3.2 Problems

We refer to this problem as the Minimum Shared Edges problem which is formulated as follows: given a graph $G = (V, E)$, a source node s, a destination node d, and an input integer value k, the objective is to find k paths from s to d, such that the total number of shared edges is minimized. An edge is considered a shared edge if it is used in two or more (s, d)-paths.

1.3.3 Previous Results

No results has been reported for the Minimum Shared Edges problem. However, a closely related problem studied in the context of communication networks is the so-called "k-best paths" problem [25, 117]. In this problem, the objective is to find a set P of k paths with minimum edge sharability, with the only difference that here, for each edge e, $\lambda(e) = 0$ if e is used in at most one path of P, otherwise $\lambda(e)$ is equal to

the number of paths containing e minus 1. As shown in [104, 154], the k-best paths problem is polynomially solvable using a minimum-cost flow algorithm.

1.3.4 New Results

We show that finding paths with minimum shared edges is NP-hard (see Chapter 6). Moreover, we show that it is even hard to approximate the minimum number of shared edges within a factor of $2^{\log^{1-\epsilon}|V|}$, for any constant $\epsilon > 0$. On the positive side, we show that there exists a $(k-1)$-approximation algorithm for the problem, using an adaption of a network flow algorithm. We design heuristics to improve the quality of the output, and provide empirical results [123, 121].

1.4 Organization

The rest of the thesis is organized as follows: In Chapter 3, we first provide an overview of different TDSP settings, and survey existing solutions. Then, we present our solutions to two variants of the problem. In Chapter 4, we present our improved ϵ–approximation algorithm for the TDSP problem. In Chapter 5, we study location privacy for users who submit queries (shortest path queries, for example) to LBSs. Multiple heuristic algorithms are proposed which are evaluated through experiments. In Chapter 6, we define the Minimum Shared Edges problem and demonstrate the difficulty of approximating the problem. We also propose an approximation algorithm that can be implemented and evaluated on multiple networks.

Chapter 2

Preliminaries and Literature Discussion

In this chapter, we provide definitions of key terms, concepts, and variables that we use in this dissertation, and survey the theoretical foundations and literature upon which our research is based.

2.1 Networks and Graphs

In graph theory, a *graph* (denoted by $G = (V, E)$) is a representation of a set of objects, referred to as *vertices* (denoted by V). A pair of objects $e = (u, v)$, for $u, v \in V$, can be connected by links, referred to as *edges* (denoted by E). A graph is *directed* if the edges have a direction associated with them. Otherwise, it is *undirected*. A graph $G = (V, E)$ is *weighted* if to each edge $e \in E$ is assigned a weight $w(e)$. A *path* in a graph is a sequence of vertices connected by a sequence of edges. For a path p, denoted by $\langle v_0, v_1, \ldots, v_k \rangle$, where $v_i \in V$ for $i \in \{0, .., k\}$ and $(v_j, v_{j+1}) \in E$ for $j \in \{0, .., k-1\}$, the length of p number of edges in p, i.e., k. The notion of edge weights can be extended to paths. The weight of path p, denoted by $w(p)$, is the sum of weights on all edges of the path, namely $w(p) = \sum_{e \in p} w(e)$. For a finite path the first vertex is called a start (or source) vertex, and the last vertex is called an end (or

destination) vertex.

In computer science, graphs are used to represent networks for transportation, communications, data organization, computational devices, computational flows, among others. A road network, for instance, can be represented by a directed graph, on which the vertices (also referred to as nodes or points) represent intersections of roads, and directed edges (also referred to as links) represent roads between intersections.

2.2 Shortest Path Problem

The Shortest Path problem is a fundamental problem in computer science. In this section, we survey variants of the Shortest Path problem and summarize existing solutions for each problem. We categorize the results on the shortest path problem based on their underlying graph. First, we group them into results on *static* and *dynamic* graphs. In static graphs, all features are fixed in time. In dynamic graphs, however, one or more features of the graph change over the time. We further classify each group as *geometric* and *non-geometric* graphs. A graph is geometric if the vertices or edges are associated with geometric objects or configurations. A graph is non-geometric if no geometric information is assigned to the vertices and edges of the graph.

2.2.1 Static Shortest Path Problem

Several versions of the Shortest Path problem have been formulated based on the number of source and destination nodes involved, as well as on the number of requested paths. The most commonly used formulations are surveyed below.

Non-Geometric

Non-Geometric Shortest Path problems deal with cases in which no geometric position information is assigned to the nodes and edges of the graph.

- **Single-Pair Shortest Paths** In the Single-Pair Shortest Path problem, the goal is to find a path, given a source node s and a destination node d, with the shortest possible weight among all possible paths from s to d. The Single-Pair Shortest Path problem can be solved using solutions to the Single-Source Shortest Paths problem (defined next). Despite being a special case of the Single-Source Shortest Path problem, however, no asymptotically (or heuristically) faster algorithm for the Single-Pair Shortest Path problem has been reported. Later, we will see that using geometric information will let us develop effective heuristics for this problem.

- **Single-Source (Single-Destination) Shortest Paths** In the Single-Source Shortest Paths problem, the goal, given a source node s, is to find a shortest path originating from s to each node of the graph. Conversely, in the single-destination shortest paths problem, a destination node d is given and the goal is to find a shortest path terminating at d from each node of the graph. Note that the two problems are equivalent because one can use a solution to the first problem to obtain a solution to the second, simply by reversing the direction of all edges and interchanging s and d.

  For cases in which negative weights are allowed and no negative cycle exists in the graph, Bellman-Ford's algorithm gives the best known bounds [15, 58]. The algorithm works in $|V| - 1$ rounds, and relaxes all edges of the graph, in each round. Letting w_u and w_v be the weight of a shortest path to node u and v, respectively. Then, the relaxing operation computes $\min(w_v, w_u + w(e))$ for

every edge $e = (u, v) \in E$. The algorithm runs in $O(|V||E|)$ time. In 1970, Yen [151] improved Bellman-Ford's algorithm to runs in $\frac{|V|}{2}$ rounds. Assuming that weight values are integers bounded below by $-N$, the time complexity of Bellman-Ford's algorithm improves to $O(\sqrt{|V|}|E| \log N)$ [73].

In the case that weight values are non-negative, Dijkstra [49] proposed a greedy algorithm which slightly differs form Bellman-Ford's algorithm, in that it performs a relax operation only for edges adjacent to the node with the smallest current $w(v)$ value at each step of the algorithm. The best implementation of Dijkstra's algorithm, uses Fredman and Tarjan's [63] Fibonacci Heaps data structure and achieves $O(|E| + |V| \log |V|)$ running time. This is the best time-complexity to solve the single-source shortest path problem for general graphs in the worst case. Assuming, however, that weight values are integers, bounded above by N, the time complexity could be improved to $O(|E| + |V|\sqrt{\log N})$ [5]. For graphs with non-negative integer weights, Thorup [139] presented a linear time algorithm.

- **All-Pairs Shortest Paths** In the All-Pairs Shortest Paths problem, the objective is to compute shortest paths between all possible pairs of source and destination nodes. A naïve algorithm is to run a Single-Source Shortest Paths algorithm $|V|$ times with every node set as the source once. Faster and more efficient algorithms to solve this problem have been proposed. Floyd-Warshall's algorithm uses a dynamic programming approach and achieves $O(|V|^3)$ running time [57, 142]. Numerous studies have been devoted to get a sub-cubic time complexity. In [27], a survey of all previous works as well as an $O(|V|^3 \log^3 \log(|V|)/ \log^2 |V|)$ algorithm has been presented. To the best of our knowledge, this is the best result achieved for this problem on arbitrary dense

graphs. On sparse graphs, Johnson [89] presented an algorithm that improves running time. The algorithm first uses Bellman-Ford's algorithm to transform the input graph into a graph with non-negative edge weights, and then runs Dijkstra's algorithm starting from each vertex of the graph. The total run time is $O(|V||E|+|V|^2 \log |V|)$. In directed graphs, the second term has been improved to $O(|V|^2 \log\log |V|)$ [128] and for undirected graphs, it could be improved to $O(|V|^2\alpha(|E|,|V|))$ [129], where α is the very slowly growing inverse-Ackermann function [1].

- ***k*–Shortest Paths** Given a source node s, a destination node d, and an integer value k, the goal of the k-Shortest Paths problem is to find k different paths from s to d such that the total weight of all paths is minimized. Depending on whether simple paths (no vertex of the graph is repeated) are required or not, several algorithms with various time-complexity values exist. For finding k simple paths, an $O(k(|E| + |V| \log |V|))$ time algorithm in undirected graphs [93] and an $O(k|V|(|E| + |V| \log |V|))$ time algorithm in directed graphs [101] is known. The latter bound has been improved to $O(k|V|(|E| + |V| \log\log |V|))$ [75]. For the case that paths are not required to be simple, Eppstein presented a novel approach [54] that finds k shortest paths in $O(|E| + |V| \log |V| + k)$ between a pair of nodes and in $O(|E| + |V| \log |V| + k|V|)$ time from a given source node to all nodes in the graph. The main approach of the algorithm is to maintain a binary heap for each node to hold the edges that are not part of the shortest path tree which can be reached from that node by edges of the shortest path tree. They first use a persistent data structure to maintain common structures of heaps and to reduce the time and space requirements. Moreover, they use Frederickson's [62] tree decomposition technique to further improve time and space bounds.

- **Shortest Paths on Special Graphs** Besides the above general graphs, some special graphs arise from a variety of applications, and are of interest in their own right. Directed acyclic graphs and planar graphs are two commonly studied graphs. A directed acyclic graph is a directed graph with no directed cycles and is widely used in applications such as job scheduling, circuit design, and by compilers. Finding Single-Source Shortest Paths can be accomplished in linear time in directed acyclic graphs. This results in an $O(|V||E|)$ time algorithm for the All-Pairs Shortest Paths problem in directed acyclic graphs which can be obtained by running the single-source shortest path algorithm starting from each node of the graph [43]. Eppstein's algorithm [54] for finding k shortest paths, if applied to directed acyclic graphs, will result in an $O(|E| + k)$ time algorithm for finding k shortest paths between a given pair of nodes and an $O(|E| + k|V|)$ time algorithm for finding k shortest paths between a given vertex and each other vertex in the graph.

  A graph is Planar if it can be drawn on the plane so that no two edges intersect geometrically except at a vertex at which they are both incident [118]. They are, for example, used to model applications related to telecommunications, transportation (on roads without under-passes), and also in VLSI design. In planar graphs, we have $|E| = O(|V|)$ so that all shortest paths algorithms for graphs, if applied to planar graphs, result in reduced time bounds. Asymptotically faster algorithms, however, have been developed by exploiting the structural properties of planar graphs. For the Single-Source Shortest Paths problem with negative weights, Klein et al. [95] present an algorithm that runs in $O(|V| \log^2 |V|)$ time. This bound has been very recently improved to $O(|V| \log^2 |V| / \log\log |V|)$ by Mozes and Wulff-Nilsen [115]. For graphs with non-negative edge weights, Henzinger et al. [83] present a linear time algorithm . Let N be the absolute

value of the most negative weight in the input graph. They also present an $O(|V|^{\frac{4}{3}} \log(|V|N))$ time algorithm for graphs with negative weights.

For the All-Pairs Shortest Paths problem, Frederickson [61] proposes an algorithm for graphs with negative weights and non-negative cycles that runs in an optimal time, i.e., $O(|V|^2)$. A similar version of this problem asks for the sum of weights of All-Pairs Shortest Paths, referred to as *Wiener Index* of the graph. Wullf-Nilsen [145] show that the problem can be solved in sub-quadratic time $O(|V|^2(\log\log|V|)^4/\log|V|)$.

The k-Shortest Paths problem has also been considered specifically for planar graphs. In cases for which simple paths are required, recent results of Wulff-Nilsen [146] to find *"replacement paths"*, if applied to this problem, give an $O(k|V|\log|V|)$ time algorithm. If paths are allowed to be non-simple, the previous result of Eppstein [54] applies, and incurs $O(|V|\log|V| + k)$ and $O(|V|\log|V| + k|V|)$ time algorithms to find k paths between a pair of nodes and between a source and each node of the input graph, respectively.

Geometric

In geometric settings, every vertex in the input graph is assigned a value indicating its positional coordinates (in 2D or 3D) and edge weight is defined as the distance between two nodes forming an edge. Note that we only present results that are based on the *Euclidean* distance metric of measurement. Two different models are possible for the geometric shortest path problems. In the first model, a graph is given explicitly with node positions provided as extra information. All algorithms presented for the non-geometric scenario would also apply to the geometric case. Such extra information, however, would not result in any asymptotically better algorithms. Nonetheless, position information has a great practical impact by reducing the number

(a) Dijkstra (b) A* with Euclidean

Figure 1: Search space for shortest paths computation between a sample pair of nodes.

of visited nodes, resulting in an improved performance. Heuristics such as A* [82] can exploit distance information as an estimate of the distance from each node to the goal. This causes Dijkstra's algorithm to converge to a final result by exploring fewer nodes of the graph. Figure 1[1] illustrates a comparison of the search space for shortest paths computation between a sample pair of nodes using Dijkstra and A*.

In the second geometric model, which is more common, the graph is implicitly encoded by a given geometric object. Such graphs can be very large, so that in order to obtain efficient algorithms, construction of the whole underlying graph has to be avoided. The geometric shortest path problem, defined using this model, has a broad application in Geographic Information Systems (GIS), robotics, graphics, among others. In the following, we review recent solutions on the geometric shortest paths problem in 2 and 3 dimensions.

- **Shortest Paths in 2D** The geometric shortest path problem in 2D deals mostly

[1]The figure is obtained from `http://www.dbs.informatik.uni-muenchen.de/ kroegerp/papers/SSDBM08-Traf`

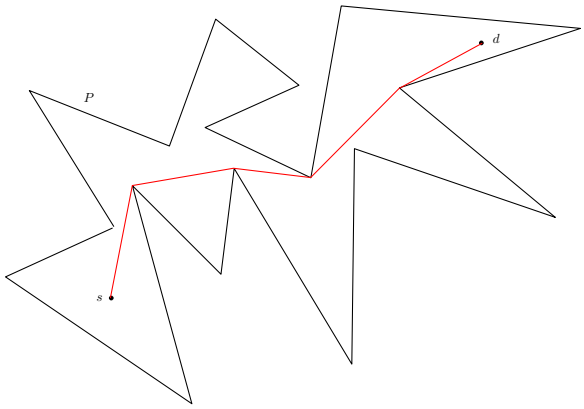


Figure 2: A sample shortest path in a simple polygon.

with finding shortest paths in *simple polygons*, *polygonal domains*, and *weighted regions*. A simple polygon P is a closed region with non-intersecting n boundary edges whose endpoints are nodes of P. In the geometric shortest paths problem for simple polygons, the goal is to find a shortest path from a given source point s to a destination d inside a simple polygon P, avoiding the outside of P (Figure 2). Because of the local optimality incurred from the triangle inequality, the shortest path from s to d is unique. Results of Chazelle [29] and Lee et al. [103] lead to $O(n)$ time algorithms. These results are based on a complicated $O(n)$ time algorithm for simple polygon triangulation [30]. No solution to the problem, without triangulation, that remains within the same time bounds has been proposed. Furthermore, Guibas and Hershberger [78] showed that with $O(n)$ preprocessing time, a data structure of size $O(n)$ that can answer shortest paths length queries between two points in $O(\log n)$ time can be developed. Their algorithm builds a hierarchy of nested sub-polygons over an underlying triangulation in such a way that any shortest path crosses a small number of sub-polygons. The information for the shortest path within each sub-polygon is associated with that sub-polygon. Upon receiving a query, the shortest path is derived using the information associated with the sub-polygons between the

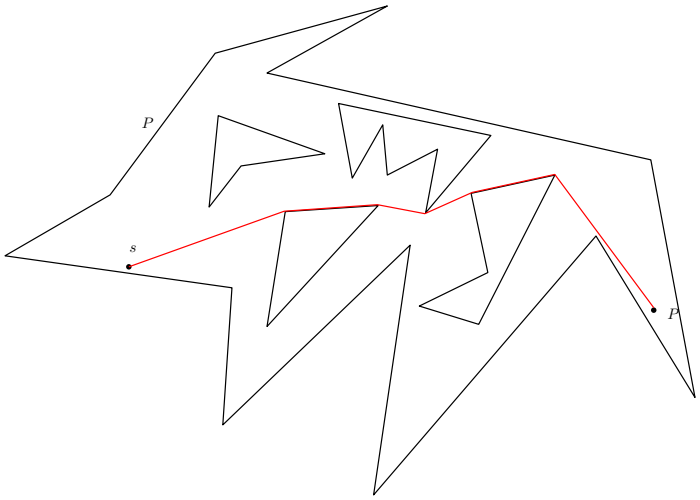


Figure 3: A sample shortest path in a polygonal domain.

query points. To get the path itself $O(k)$ time is needed, where k is the number of nodes in the output path. This result has been simplified using a new data structure introduced by Hershberger [84].

A polygonal domain P with n vertices and h holes is a closed region formed by $h + 1$ closed polygonal chains comprising n boundary edges. In polygonal domain, shortest paths between two points must avoid the outside of P which includes holes (Figure 3). Unlike for simple polygons, in polygonal domains, a unique shortest path may not exist. In fact, there could be an exponential number of them. There are two methods for computing shortest paths in polygonal domains: 1– running Dijkstra's algorithm in the visibility graph; 2–running an extension to Dijkstra's algorithm referred to as continuous Dijkstra. A visibility graph is obtained by connecting visible vertices of the polygonal domain P. Figure 4 shows a sample visibility graph.

Source and destination nodes can be considered as two holes in P. Based on this, it can be shown that a shortest path in P uses visibility graph edges. Once the visibility graph has been computed for P, Dijkstra's algorithm can be used

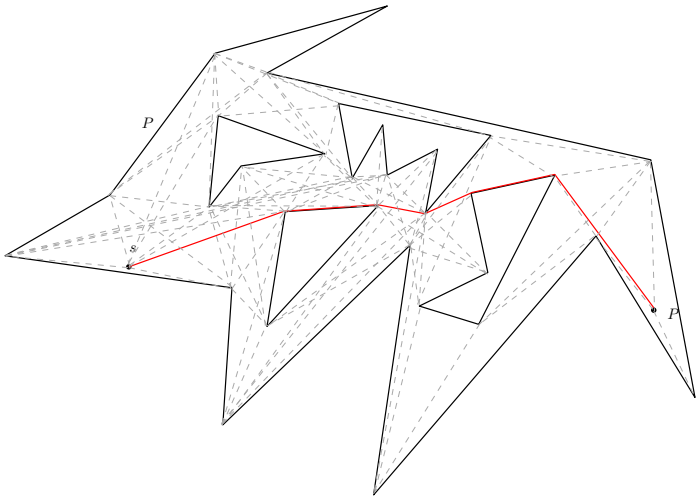


Figure 4: Visibility graph of the sample polygonal domain of Figure 3.

to find a shortest path from a source node to a destination node. The first method for computing visibility graphs uses a radial sweep approach for every vertex of P which results in an $O(n^2 \log n)$ algorithm [102]. Welzl [143] and Asano et al. [11] improved the time bound to $O(n^2)$ using a point-line duality approach. Despite the fact that $O(n^2)$ visibility edges are possible even in the worst cases, the number of edges could be much smaller so that an output-sensitive algorithm is more desirable. Let E_{VG} be the number of edges of the visibility graph. Ghosh and Mount [72] presented an output-sensitive algorithm for computing visibility graph that runs in $O(E_{VG} + n \log n)$ time and requires $O(|E_{VG}| + n)$ space. Optimal time and space ($O(E_{VG} + n \log n)$ time and $O(n)$ space) algorithms have been presented by Pocchiola and Vegter [130] and Riviére [131] for computing the visibility graph. Once the visibility graph is available, Dijkstra's algorithm gives shortest paths to all vertices of P from a given source in $O(E_{VG} + n \log n)$ time which is the same as time to compute the visibility graph. Using the A^* heuristic for computing shortest paths, a better performance can be obtained if a shortest path between given source and

destination vertices is required. Another visibility-graph-based approach has been developed by Kapoor et al. [91] which runs in $O(n + h^2 \log n)$ time where h is the number of holes in P.

The other method for finding shortest paths in polygonal domains is based on constructing shortest path maps directly by using a continuous Dijkstra paradigm [111]. The main idea is to simulate the effect of propagating a "wavefront" starting at a source node s and managing critical events that change the structure of the wavefront. Using this method, Hershberger and Suri [85] presented a nearly optimal algorithm with an $O(n \log n)$ time bound. It is not known whether or not it is possible to obtain an optimal algorithm with an $O(n + h \log h)$ time bound.

The query version of the geometric shortest path problem in a 2D polygonal domain has also been studied. Chen et al. [32] proposed an approach that computes shortest path maps and achieves $O(\min(Q_a, Q_b) \log n)$ query time by spending $O(n^2 \log n)$ time in preprocessing, where Q_a and Q_b are the number of vertices visible from s and d, respectively. The shortest path map with respect to a given source point s is the planar map whose boundaries are given by the boundaries of the k-visibility regions, where the k-visibility region is the set of all points in free space whose link distance from s is, at most, k. Quite recently, Guo et al. [79] present an algorithm that uses a *reduced skeleton graph* to tessellate P into a set of triangles and simple polygons. Their algorithm needs $O(n^2 \log n)$ preprocessing time and achieves $O(h \log n)$ query time. Unfortunately, devising algorithms with faster query times is challenging, as it results in high preprocessing time. Chiang and Mitchell [35] developed several algorithms with trade-offs between preprocessing and query time. They develop an algorithm with $O(\log n + h)$ query time and $O(n^5)$ preprocessing time.

They also achieve optimal $O(\log n)$ query time using an algorithm that needs $O(n^{11})$ preprocessing time. Alternatively, their algorithm requires $O(n^{10} \log n)$ preprocessing time to obtain $O(\log^2 n)$ query time.

Shortest paths are also studied on *weighted regions* as well as on simple polygons and polygonal domains. A weighted region is a planar space composed of triangular regions, each of which is associated with a positive weight value. In such settings, the objective is to find a path from s to d with minimum weighted sum of the lengths of the sub-paths within each region. The shortest path problem on weighted regions is a generalization of the problem on polygonal domain, because one can assign a small weight value to P and a relatively large weight values to the exterior region of P. In some literature, weighted regions have been referred to as *weighted polyhedral surfaces*. Let P be a polyhedral surface containing n triangular faces with vertices that have integer coordinates bounded by N. Moreover, suppose w and W are the smallest and the greatest weight in P, respectively. Mitchell and Papadimitriou [112] used the continuous Dijkstra paradigm, and proposed an algorithm that finds a shortest path whose weighted length is within a factor $1 + \varepsilon$ of the optimal solution, where ε is the degree of precision defined by the user. Their algorithm runs in $O(n^8 \log(nNW/\varepsilon))$ time.

More recently, Lanthier et al. [100] have shown that instead of using a complex continuous Dijkstra method, it is both practically and theoretically favorable to apply discretization methods, such as placing *Steiner nodes* on edges. In [100], the authors use an evenly-spaced method of placing Steiner nodes, which are then locally interconnected to obtain a new graph. The new graph is then searched for a shortest path, which leads to an approximate shortest path. The error induced by their method is additive and can be at most $w_{max}|l_e|$ where w_{max} is the maximum weight over all faces and $|l_e|$ is the length of the longest

edge in P. Their algorithm achieves $O(n^5)$ running time. In [110], another method has been established that constructs a "pathnet graph". A pathnet graph is built by tracing k evenly-spaced rays used to discretize the continuum of orientations for each vertex, and then connecting that vertex to one vertex within each k refraction points produced by the rays. Different models of placing Steiner nodes have been studied, which has led to various solutions for this problem. Given $0 < \varepsilon < 1$, the best known time bound for $(1+\varepsilon)$-approximation algorithms is presented by Aleksandrov et al. [8]. They place Steiner nodes on the bisectors of faces in a sophisticated manner, using which they obtain a path within $1 + \varepsilon$ factor of the optimal in $O(N^2 \log(\frac{NW}{w}) \frac{n}{\sqrt{\varepsilon}} \log \frac{n}{\varepsilon} \log \frac{1}{\varepsilon})$ time.

For the query version of the problem, recently Aleksandrov et al. [6] proposed an algorithm that given a pair of vertices, approximation parameter $\varepsilon \in (0, 1)$, and a query time parameter q, computes an ε-approximate distance query in $O(q)$ time. In polyhedral surfaces with *genus* g, their algorithm would need an $O(\frac{(g+1)n^2}{\varepsilon^{3/2}q} \log \frac{n}{\varepsilon} \log^4 \frac{1}{\varepsilon})$ of preprocessing time. As a basis for their data structure, they developed a new graph separator algorithm and a data structure that can answer single-source ε-approximate distance queries. It answers queries in $O(\log \frac{1}{\varepsilon})$ time and spends $O(\frac{n}{\sqrt{\varepsilon}} \log \frac{n}{\varepsilon} \log \frac{1}{\varepsilon})$ time for preprocessing.

- **Shortest Paths in 3D** Finding geometric shortest paths in 3D is more complex, because shortest paths in 3D can bend along the interior of edges and the number of combinatorially distinct shortest paths between a pair of points can be exponential. In fact, Canny and Reif [24] showed that the problem is NP-hard in its general form. Exact solutions, however, are proposed for special cases, such as convex polyhedral surfaces [2, 13, 132] In 3D, shortest paths has mostly been studied on three different settings namely on *polyhedral surfaces*,

weighted polyhedral surfaces, and *polyhedral domains*. A polyhedral surface is the union of a finite set of polygonal faces connected by a common edge, a common vertex, or not at all. Each edge is shared by exactly two polygonal faces. In the weighted version of the problem, each face is assigned a non-negative real value, and the cost of each sub-path within the face is the length of the sub-path times the weight of the face. A polyhedral domain is defined as a connected sub-set of $\mathcal{R}^3$ formed by the union of a finite number of triangles indicated as its boundary.

Shortest paths on polyhedral surfaces (weighted and unweighted) can be approximated using the same approach used for weighted regions. Most approximation algorithms for weighted regions have originally been presented for weighted polyhedral surfaces, and can also be used for unweighted polyhedral surfaces and weighted regions. There exists no known approximation algorithm for them that is asymptotically faster than using algorithms on weighted polyhedral surfaces. As mentioned above, the best known solution which applies to all these settings is presented by Aleksandrov et al [8, 6].

For the polyhedral domain setting, Papadimitriou [127] was the first to propose an approximation algorithm. His algorithm is an $(1+\varepsilon)$–approximation of the optimal solution and runs in $O(n^4(L+\log(n/\varepsilon))^2/\varepsilon^2)$ time, where L is the number of bits needed to represent the value of a vertex coordinate of P. Clarkson [41] improves the time complexity to $O(n^2\lambda(n)\log(n/\varepsilon)/\varepsilon^4 + n^2\log n\rho\log(n\log\rho))$, where ρ is the ratio of the length of the longest obstacle edge to the distance between source and destination nodes and $\lambda(n) = \alpha(n)^{O(\alpha(n)^{O(1)})}$, where $\alpha(n)$ is the inverse of Ackermann's function. Aleksandrov et al. [7] consider a more general case where the polyhedral domain is tetrahedralized and each tetrahedron is assigned a positive weight value. In this

setting, the cost of a sub-path inside each tetrahedra is the length of the path inside the tetrahedron multiplied by its weight. Let n be the number of tetrahedra and $c \geq 1$ be a constant. Their algorithm uses the same discretization approach used for polyhedral surfaces and achieves a $(1 + c\varepsilon)$–approximation algorithm that requires $O(n/\varepsilon^3 \log(1/\varepsilon)(1/\sqrt{\varepsilon} + \log n))$ time. The problem can be reduced to the Euclidean distance version if all the weights are the same in which case the time complexity would be $O(n/\varepsilon^3 \log(1/\varepsilon) \log n)$ (parameters that depend on the geometry of the input are ignored). Moreover, the authors note that their improved approach for polyhedral surfaces presented in [8] can also be applied to weighted polyhedral domains yielding a $(1+\varepsilon)$–approximation algorithm. This is obtained by placing $O(n/\varepsilon^2 \log(1/\varepsilon))$ Steiner points versus $O(n/\sqrt{\varepsilon} \log(1/\varepsilon))$ used for polyhedral surfaces.

Other than the algebraic complexity of the problem, which is based on the number of algebraic operations performed on real numbers to measure a problem's complexity, some work has focused on the bit framework of the complexity [36]. In the bit framework, the input is presented with binary strings and the time complexity is measured by considering the number of boolean operations performed on bits. For such settings, Har-Peled [81] present a technique for constructing a data structure that computes an approximate shortest path map in a polyhedral domain. Their algorithm takes roughly $O(n^4/\varepsilon^6)$ time (for precise result see [81]) and computes a subdivision of $\mathbb{R}^3$ which can be used to answer shortest path queries between two points in $O(\log(n/\varepsilon))$ time. For the same framework in polyhedral domains, Asano et al. [12] presented an algorithm that first computes a pseudo $(1 + \varepsilon)$–approximation algorithm (which is defined in the paper) and then convert it to a true $(1 + \varepsilon)$–approximation algorithm. They show that a $(1 + \varepsilon)$–approximation algorithm for finding a

shortest path between two points in a polyhedral domain can be computed in $O(n^4/\varepsilon^2 \log\log(2^L/d(x^*)))$ time, where L is the input precision parameter, x^* is an optimal solution, and d is the optimization function to be minimized. It has been shown that the lower bound of $d(x^*)$ is c^{-L} for constant $c > 1$.

2.2.2 Dynamic Shortest Path Problem

In many real-world applications that involve requests for a shortest paths computation, the time of the requests plays an important role in the solution. For example, shortest paths calculation is a fundamental part in vehicle routing systems. A shortest path computed during rush-hours, when the traffic is heavy, may be completely different from one computed during regular traffic periods (for example, highways should be avoided during rush-hours). Also, in communication networks, various paths can become optimal for packet routing based on the network load at the query time. Given a graph $G = (V, E)$ on which the weight of edge $e \in E$ is represented by arrival time functions, $A_e(t)$, the general dynamic shortest path problem is to find shortest paths between two points or a given source to all destinations given that input properties of the graph, namely nodes, edges, and arrival times change over time. The travel time on each link (u, v) of the graph and, accordingly, the arrival time at v are defined as a positive function of the departure time from u. In the relevant literature [46, 60, 90, 116], it is common to consider that the input graph is a FIFO network, and no waiting on nodes is allowed throughout the network. A FIFO network is a network for which the FIFO property holds for every link in the network. Let (u, v) be a link in the network, t_u and t'_u be two start times at u such that $t'_u > t_u$, and $A_{(u,v)}(t)$ and $A_{(u,v)}(t')$ be the corresponding arrival times at v, respectively. The FIFO property on links implies that for every link (u, v), $A_{(u,v)}(t') \geq A_{(u,v)}(t)$. In other words, for every link (u, v), the arrival time function to v for different departure

times from u is non-decreasing. A network in which link-arrival-time functions are non-FIFO, and in which waiting on nodes is allowed, can be converted to a FIFO network in which waiting is not beneficiary. However, finding shortest paths in the general case on a non-FIFO network with no-waiting policy has been proven to be NP-hard [124].

Non-Geometric

In the non-geometric setting, no extra information for node locations is provided alongside the input graph. Given a time-dependent graph $G = (V, E)$, and a particular start time t_0, shortest paths from s to all nodes in the network can be computed in the same way we discussed earlier, for static graphs. Dreyfus [51] shows that both standard label setting and label correcting methods can be modified to solve the time-dependent shortest path problem for a particular start time within the same time bounds. Label-setting algorithms remove a node with the smallest distance label from the list of nodes during each iteration of the algorithm. These algorithms greedily find a shortest path from the source to every other node in the graph. Because edge costs are non-negative, once a node is removed from the list, it will never re-enter. Therefore, the algorithm ends after $|V|$ iterations. Label-correcting algorithms do not permanently set distance labels. In fact, all labels are considered not correct until the algorithm ends. During each iteration of the algorithm, the node removed from the list does not necessarily have the minimum label. Therefore, a removed node may re-enter the list at a later time. In the literature, most work has been devoted to finding the time-dependent shortest paths for all possible start times, or for start times within a given time interval. Based on the representation of the time, two variants of the problem are defined: discrete and continuous time-dependent shortest paths problem. In what follows, we study previous results on both models.

- **Discrete Time Model** In the discrete time model, arrival time functions are integer-valued functions of an integer argument. A modified version of the standard label-correcting algorithm (Bellman-Ford's algorithm, for example) can solve the shortest path problem on networks using a discrete time model. Here, operations on scalar values will be replaced by operations on arrival time functions for all time values. Ziliaslopoulos and Mahmassani [156] propose this. Let T be the number of time instances at which they run an arrival time computation. In their algorithm, operations on scalar values are replaced by operations on vectors of size T, so that their algorithm runs in $O(|V||E|T)$ time.

  Additionally, label setting algorithms (Dijkstra's algorithm, for example) can also be used to solve the discrete version of the problem for fixed values of time. An algorithm that uses Dijkstra's algorithm to compute shortest paths for all T possible start times runs in $O(T(|E| + |V| \log |V|))$ time. An improvement to this algorithm is proposed by Cai et al. [23] and Chabini [26] who use the time-expansion paradigm. In this paradigm, a new time expanded network G_T is developed in which for every node $v \in V$ of G, a node (v, t), $t \in \{1, \ldots, T\}$, is added to G_T. Moreover, for every edge $(u, v) \in E$ of G, edge $((u, t), (v, A_{(u,v)}(t)))$, $t \in \{1, \ldots, T\}$, is added to G_T, where $A_{u,v}(t)$ is the arrival time function on edge (u, v). G_T is an acyclic network with $O(|V|T)$ nodes and $O(|E|T)$ edges, so that the shortest paths to all nodes from a given source node can be computed in linear time, namely $O(|E|T)$. The proposed algorithm is optimal, as it matches the lower bound on the time complexity of the problem.

- **Continuous Time Model** In the continuous time model, arrival time functions are real-valued functions of a real argument, and the objective is to find the earliest arrival times to destination node d for all possible start times at start

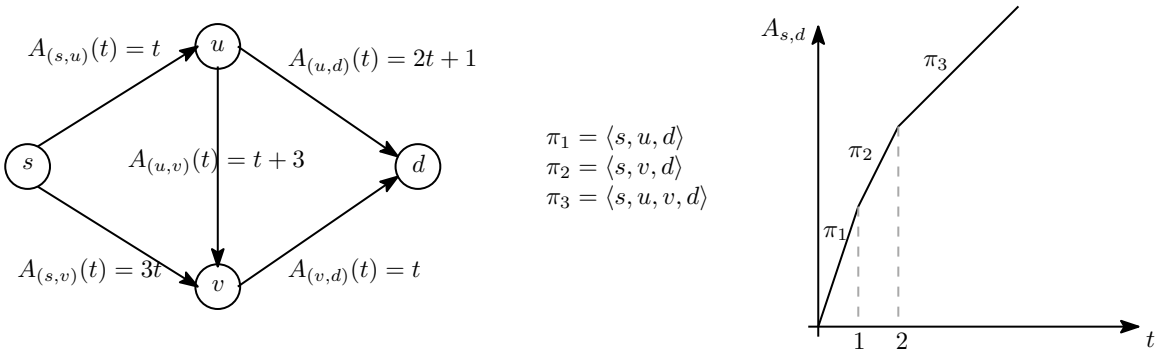


Figure 5: A time-dependent network and the earliest arrival time at d starting from s

node s. A modified version of the standard label-correcting algorithm can solve the shortest path problem on networks presented using the continuous time model. Here, rather than performing operations on scalar values, operations are performed on functions at each step. Figure 5 depicts a sample time-dependent network and the resulting earliest arrival time function $A_{s,d}(t)$ at destination node d. Here, $A_{(i,j)}$ is the arrival time function on edge (i, j). Paths π_1, π_2, and π_3 are the shortest path in a particular period of time, as shown in Figure 5.

For such cases, Orda and Rom [124, 125] propose an algorithm that runs in $O(|V||E|F)$ time, where F is the maximum complexity of the resulting earliest arrival time function to any node in the network. Their algorithm is based on a modified version of Bellman-Ford's algorithm. Despite being theoretically acceptable, general functions are not appropriate for practical applications because they cannot be efficiently represented by computers. As a result, piecewise linear functions are used to approximate general functions. This, coupled with the fact that in real-world applications, data is usually obtained by sampling at specific time points for the TDSP problem makes piecewise linear functions suitable for representing arrival time functions. Throughout the remainder of this section, we review previous results on the TDSP problem on networks with

piecewise linear arrival time functions. Recall that the literature for the TDSP problem commonly assumes the FIFO property. Therefore, we only survey results in which the FIFO property holds for arrival time functions.

Unlike label-correcting algorithms, it is not possible to simply replace scalar label values by functions to solve the TDSP problem, because a minimum element (i.e. one function which is minimum over the entire domain) may not exist. One way to overcome this issue is to determine for each node v, a time ϕ_v, such that for any time less than ϕ_v, the current arrival time function at the node is minimum. For FIFO networks with piecewise linear arrival time functions, Dean [46] suggests a label-setting algorithm that performs a single chronological scan throughout time to establish output functions. The algorithm employs the same approach used for solving the parametric shortest path problem. In the worst-case, this algorithm has a running time of $O(F^*|E|\log|V|)$, where F^* is the total number of pieces over all output functions in the network. Dean also conjectures that the complexity of the output function could be super-polynomial. Foschini *et al.* [60] prove this conjecture.

Ding et al. [50] presented a simpler label-setting algorithm for the TDSP problem in FIFO networks with piecewise linear functions. The algorithm scans a sequence of time steps the size of which depends on the values of the arrival time functions. A detailed analysis of this algorithm (in Section 3.2.1) shows that this approach yields a solution with $O(\alpha(|E|+|V|\log|V|))$ time complexity which contains a factor α that is possibly unbounded.

Foschini *et al.* prove, more recently, that the earliest arrival time function could have super-polynomial complexity (i.e., $O(n^{\Theta(\log n)})$). They also present an algorithm that uses kinetic data structures, and holds a set of certificates that

are linear functions of t, and that guarantees the correctness of the shortest path tree. They also keep a priority queue of events (certificate failure times) and update the corresponding certificates if an event is removed. Their algorithm has time-complexity $O((\lambda + F_{max}|E|)\log^2|V|)$, where F_{max} is the maximum of the complexity of arrival time functions from s to each nodes in the network and λ is the input size (total number of linear pieces in arrival time functions on links).

Geometric

Similar to the static shortest path problem, A* algorithm concepts have also been applicable to the TDSP problem. Kanoulas et al [90] presented an A* heuristic for the TDSP problem, whose worst case time bound is exponential (as it may visit all paths between two points in the network). Their algorithm uses a constant value function as an estimator of the travel time between each node in the network and the destination. A generalization of this method is proposed by Zhao et al. [152]. Their algorithm employs time dependent estimator functions to determine the shortest path between a node and the destination. They also generalize the notion of the optimality condition of the A* algorithm for the static shortest path problem to the time dependent scenario. Although, geometric information helps to decrease the number of unnecessary node visits, the time-dependent A* heuristics are not as effective here as they are for the static case. A constant value estimator function does not produce a good approximation of the exact travel time from each node to the destination for the reason that the travel time changes over time. To solve the problem, a good time dependent estimator function is required, but finding such a function requires additional computations, which results in a degraded performance.

To our knowledge, no result for the time-dependent shortest path problem on

polygonal or polyhedral domains has been discussed.

2.3 Privacy in Location Based Services

To give a deeper insight into various types of location-based attacks and to corresponding protection strategies, we categorize user queries into four types, based on the frequency of the LBS requests: *off-line, sporadic, frequent,* and *continuous.*

2.3.1 Off-Line Query LBS

In an *off-line query* LBS (e.g., in-car navigation system), a user's location and query are not visible to anyone but the user. User privacy, therefore, cannot be compromised in an off-line LBS.

2.3.2 Sporadic Query LBS

Sporadic query LBS refers to cases where user requests are issued on an occasional basis, so that two or more queries cannot be linked to infer additional information about the user (explained in the next paragraph). Users of sporadic LBS may occasionally submit queries such as *nearest neighbor* or *shortest path* calculations to the LBS provider. Consecutive queries in sporadic query LBS are not related to each other. Therefore, a user's current location does not reveal any information about their past locations. Well-known approaches to concealing a user's current location include hiding the identity among other fake/real user locations, deliberately degrading user's location information, and encrypting the query. These lead to three types of privacy protection strategies: anonymization [17, 66, 94], obfuscation/cloaking [10, 20, 52, 114], and cryptography [70, 155]. For an overview of the state-of-the-art in location privacy protection for sporadic query LBS, see [68].

2.3.3 Frequent Query LBS

Frequent query LBS specifies the type of LBS in which users send frequent service requests, which include their current location information. Consider a user who requests *shortest path* or *nearest facility* computations multiple times during their trip. With location information of residential and public buildings being publicly available, if all locations visited (e.g. home, hospital, and night-club) could be linked to the querying user, then an adversary can not only infer a user's identity, but may also be able to obtain information about the user's lifestyle or sexual orientation, or details about their health.

Existing privacy protection solutions for sporadic query LBS, if applied to frequent query LBS, may result either in privacy breaches or in degraded performance. Duckham *et al.* [53] were the first to introduce such a privacy threat scenario by pointing out a possible attack strategy employing maximum, minimum, or average speed of movement information. [34, 69, 148] study this issue and propose privacy protection solutions using the cloaking-based protection approach. Recently, Pan *et al.* [126] added a k-anonymity privacy protection approach to further improve previous methods by protecting user privacy against identity theft. In [33], a more general approach for protecting user privacy is presented by studying the impact of query dependency on user privacy. Cryptography-based location protection methods might provide a better privacy protection, but they suffer from poor performance, and require specific infrastructure, which may not be supported by all LBS providers [67].

2.3.4 Continuous Query LBS

As positioning technology evolves, more location-based applications are emerging, and more user devices are equipped with a positioning instrument. LBS providers

are now capable of continuously providing information regarding the nearest facilities to users' hand-held devices while they are traveling. Constantly probing vehicle location in Vehicular Ad-Hoc Networks (VANET) is a cost-effective way to obtain traffic information in road-networks. Providing privacy for these types of LBS, which we refer to as *continuous* query LBS, is more challenging, since in these cases it is much harder to provide privacy against spatial and temporal correlation attacks. Privacy issues in continuous query LBS are not addressed in this chapter. However, we are currently working on extending our work to cover continuous query LBS as well. For existing work on this issue, refer to [37, 38]. Note, however, that location privacy through collaboration methods [87] could also be applied to hide user locations in continuous query LBS. However, these methods rely on users' contributions and require a long period of time to collect enough contextual information to share with other users.

2.4 Disjoint Paths Problem

In some applications of graph theory, reporting multiple paths between a pair of nodes of a graph that are fully or partially disjoint is desired. In communication networks, for example, network design should allow packets to travel along non-intersecting (node/edge disjoint) paths to provide load balancing and network optimization. Note that it is possible to use the same algorithm that finds edge-disjoint paths to find vertex-disjoint paths, by replacing each vertex by a pair of adjacent vertices, one with all of the incoming adjacencies of the original vertex, and one with all of the outgoing adjacencies. Then, k edge-disjoint paths in this modified graph correspond to k vertex-disjoint paths in the original graph, and vice versa. In case that the network topology does not allow totally disjoint paths, network designers prefer to

route packets through partially disjoint paths that minimize the cost enforced by shared parts. In this section, we first survey different settings and results on totally disjoint paths problems. Then, we describe two cost models, related problems, and solutions for partially disjoint paths problems.

The simple case of finding two or more (shortest) disjoint paths from source s to destination node d can be solved using (Minimum Cost) Network Flow algorithms. In the Minimum Cost Network Flow (MCNF) problem, the goal is to find a flow of size F with minimum cost. Ford and Fulkerson [58] consider the problem of finding a pair of disjoint paths from s to d and propose an $O(|E| \log_{(1+|E|/|V|)} |V|)$ time algorithm using the solution to Minimum Cost Flow problem. This results in a $O(|E||V| \log_{(1+|E|/|V|)} |V|)$ time bound for finding all pairs of shortest disjoint paths to all $|V|$ destinations. This has been improved to $(|V|^2 \log |V|)$ by Suurballe [136] by combining the Dijkstra runs used by the algorithm. Suurballe and Tarjan [137] further improve the bound to $O(|E| \log_{(1+|E|/|V|)} |V|)$.

Based on their applications in various fields (like network design and optimization), many variants of the disjoint paths problem have been studied, most of which are proven to be NP-hard. The problem of finding two disjoint paths, such that the length of the shorter path is minimized, is addressed in [147] and proven to be NP-hard. The authors show that finding a k-approximation of the optimal solution is also NP-hard for any $k > 1$. Li et al. [106] study a similar problem in which the objective is to find a pair of disjoint paths between source and destination such that the length of the longer path is minimized, and show that it is NP-hard. Itai et al. [88] show that the problem of finding a pair of link-disjoint paths between two points, such that the shorter and the longer paths are less than or equal to a given bound Δ_1 and Δ_2, respectively, is also NP-hard. Although the shortest-pair disjoint-paths problem is polynomially solvable, considering any of the NP-hard problems discussed here as a

secondary objective to the shortest-pair disjoint-paths problem will result in an NP-hard problem. Yang et al. [150] explore finding the shortest pair of disjoint paths by applying an additional constraint: if there is a tie, the length of the shorter path is minimized and proven to be NP-hard. Beshir and Kuipers [18] study the shortest pair disjoint paths problem in combination with two other problems. They show both problems to be NP-hard. Li et al. [105] consider a different version of the shortest disjoint paths problem in which k different costs are assigned to every edge where the j^{th} link cost is associated with the j^{th} path. Here, the objective is to find k disjoint paths between s and d. They prove the problem to be NP-hard for node/edge-disjoint paths, as well as on directed and undirected networks. Moreover, they presented two heuristics, based on the solution to the minimum cost flow problem.

Another problem with finding edge-disjoint paths arises when a collection $C = \{(s_1, d_1), \ldots, (s_k, d_k)\}$ of pairs of source and destination nodes is given, and the objective is to find the maximum number of disjoint paths connecting source-destination pairs. The problem has been proven to be NP-hard in arbitrary networks [92] as well as in many other specific types of graphs, such as planar graphs [141], series-parallel graphs [119], and grid graphs [109]. An $\sqrt{|E|}$-approximation algorithm is presented for this problem on general graphs by Kolliopoulos and Stein [97]. This algorithm simply selects a shortest path (i.e. the path with the fewest number of edges), at each step, until no pairs can be connected by a disjoint path. Guruswami et al. [80] showed that, for any $\varepsilon > 0$, the problem cannot be approximated within a factor of $O(|E|^{0.5-\varepsilon})$ for arbitrary directed graphs, unless $P = NP$.

A more realistic formulation of the disjoint paths problem takes the travel time into account. Here, link costs are assumed to be proportional to the transit time, which depends on time of use. This has applications in transportation networks for traffic dispatching. Mahmassani et al. [108] and Ziliaskopoulos and Mahmassani

[156] employ time-dependent k-Shortest Paths for traffic routing. Their approach has the drawback of having many overlapping paths, which creates bottlenecks in the network. This leads to a time-dependent model of the disjoint-paths problem, which has been studied by Sherali et al. [134]. They considered different versions of the problem (such as min-max, min-sum, and FIFO link cost min-sum) and prove the NP-hardness for each.

Although disjoint paths are desirable for reliable routing and load balancing in network applications, it is not always possible to find such paths, due to network topology constraints. In case that some nodes/links have to be shared between paths from s to d, minimizing the cost enforced by shared edges is thus the most desirable approach. Based on the application, shared edges degrade the result in two ways: either the total cost depends on the number of times an edge is shared, or the edge cost only applies if the edge is used more than once. This leads to two models of cost, namely variable and fixed cost models.

In the variable cost model, the cost of a shared edge depends on the number of times it is shared by (s, d)-paths. The objective is to find k paths from s to d with minimum total cost of shared edges, where the cost of each shared edge is defined as the total number of times that it is shared by (s, d)-paths multiplied by the cost of sharing the edge. Note that, w.l.o.g., edge sharing cost is defined as equal to 1, so that the objective is to find the minimum number of shared edges. This problem is studied by Castanon [25] on Trellis graphs in order to find the minimum sum of shared nodes. Later, Nikolopoulos et al. [117] propose a heuristic for general graphs by converting arbitrary graphs into Trellis graphs. A new method for finding a minimum sum of shared edges is developed by Lee et al. [104]. Their algorithm converts the problem into an instance of the minimum cost flow problem. Using the same approach, in [153, 149], the authors present a polynomial time solution for finding minimum sum

of shared nodes/edges on both directed and undirected networks.

In the fixed cost model, the cost of a shared edge is fixed, and is applied when the edge is shared by two or more (s, d)-paths. The problem of finding the minimum cost of shared edges in this model has not been studied before. We study this problem in Chapter 6.

Chapter 3

Shortest Paths in Time-Dependent FIFO Networks

In this chapter, we study the *time-dependent shortest paths problem* for two types of time-dependent FIFO networks. Since the FIFO property appears to be a natural assumption holding in many applications, it is a common assumption in the literature (see, e.g., [46, 60, 90, 116]) and one we make in this chapter. First, we consider networks where the availability of links, given by a set of disjoint time intervals for each link, changes over time. Here, each interval is assigned a non-negative real value which represents the travel time on the link during the corresponding interval. The resulting shortest path problem is **Time-Dependent Shortest Path Problem** for availability intervals ($\mathcal{TDSP}_{\text{int}}$), which asks to compute all shortest paths to any (or all) destination node(s) d for all possible start times at a given source node s. Second, we study time-dependent networks where the cost of using a link is given by a non-decreasing piecewise linear function of a real-valued argument. Here, each piecewise linear function represents the arrival time on the link based on the time when the link is used. The resulting shortest path problem is **Time-Dependent Shortest Path Problem** for piecewise linear functions ($\mathcal{TDSP}_{\text{lin}}$) which asks to

compute, for a given source node s and destination d, the shortest paths from s to d, for all possible start times. In transportation applications, a solution to this problem could be used for trip planning to provide a user, who is flexible in time to leave the source location, with route options from which he/she could select the one that suits his/her schedule.

We present an algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem that runs in time $O((F_d + \gamma)(|E| + |V| \log |V|))$ where F_d is the output size (i.e., number of linear pieces needed to represent the earliest arrival time function to d) and γ is the input size (i.e., number of linear pieces needed to represent the local earliest arrival time functions for all links in the network). We then solve the $\mathcal{TDSP}_{\text{int}}$ problem in $O(\lambda(|E| + |V| \log |V|))$ time by reducing it to an instance of the $\mathcal{TDSP}_{\text{lin}}$ problem. Here, λ denotes the total number of availability intervals in the entire network. Both methods improve significantly on the previously known algorithms.

3.1 Overview

In this chapter, we consider shortest path problems for two types of time-dependent networks which will be introduced in the next two subsections.

The $\mathcal{TDSP}_{\text{int}}$ Problem

The first time-dependent shortest path problem, $\mathcal{TDSP}_{\text{int}}$, is defined for a network $G_i(V, E)$, where the *availability* of links changes over time. Each link $e = (u, v) \in E$ is assigned a set I_e of disjoint time intervals, referred to as *availability intervals*, during which the link is available. Availability intervals can model, for example, situations where some roads or lanes of a road network are only available during certain times. Typical examples include no left-turns during rush hours, border crossings that are

closed during the night, and lanes restricted for buses during certain peak periods. During times outside the set I_e of time intervals, the link e is not available and one has to wait until the start of the next availability interval in I_e. For every availability interval $i = [l_e^i, r_e^i] \in I_e$ ($l_e^i, r_e^i \in \mathbb{R}^+$ and $l_e^i \leq r_e^i$) on e is defined a non-negative cost $w_e^i \in \mathbb{R}^+$ which represents the time needed to traverse e during availability interval i. More precisely, starting from node u of $e = (u, v)$ at time ST_u, $l_e^i \leq ST_i \leq r_e^i$, one arrives at node v at time $AT_v = ST_u + w_e^i$. Let $j = [l_e^j, r_e^j] \in I_e$ be the interval immediately following i in ascending time order. Our model captures waiting at nodes in that for ST_u, $r_e^i < ST_u < l_e^j$, one can wait at u until the next interval is available. The start time would then be $ST_u = l_e^j$, resulting in arrival time $AT_v = l_e^j + w_e^j$. Note that for start times greater than the last interval on the link, e is not available and $AT_v = \infty$. Consider a path $p = \langle v_0, v_1, \ldots, v_k, v_{k+1} \rangle$ in a network $G_i(V, E)$. We say p is valid if there exist start times $ST_{v_0}, \ldots, ST_{v_k}$ and arrival times $AT_{v_1}, \ldots, AT_{v_{k+1}}$ such that $ST_{v_i} \in I_{(v_i, v_{i+1})}$, $AT_{v_{i+1}} = ST_{v_i} + w^i_{(v_i, v_{i+1})}$, and $AT_{v_{i+1}} \leq ST_{v_{i+1}}$ for all $0 \leq i \leq k$. Here, $ST_{v_i} \in I_{v_i, v_{i+1}}$ denotes that there exists an interval $i \in I_{(v_i, v_{i+1})}$ such that $ST_{v_i} \in i$. The arrival time of a valid path p at v_{k+1} for start time ST_{v_0} at v_0 is $AT_{v_{k+1}}$. If path p is invalid, then $AT_{v_{k+1}} = \infty$.

For the $\mathcal{TDSP}_{\text{int}}$ problem our goal is to find, for all start times at a given source node $s \in V$, the earliest arrival time at all destination node $d \in V$, considering all valid paths from s to d. A path p that returns the earliest arrival time at destination node d for a given start time t at source node s is called the *shortest path to d for start time t at s*. We use the terms earliest arrival time path and shortest path interchangeably. In this chapter, we solve the $\mathcal{TDSP}_{\text{int}}$ problem by reducing it to a special instance of $\mathcal{TDSP}_{\text{lin}}$ defined in the next subsection. Figure 6, shows a sample network with 3 possible paths from s to d and the resulting minimum arrival time function, $A_{s,d}(t)$.

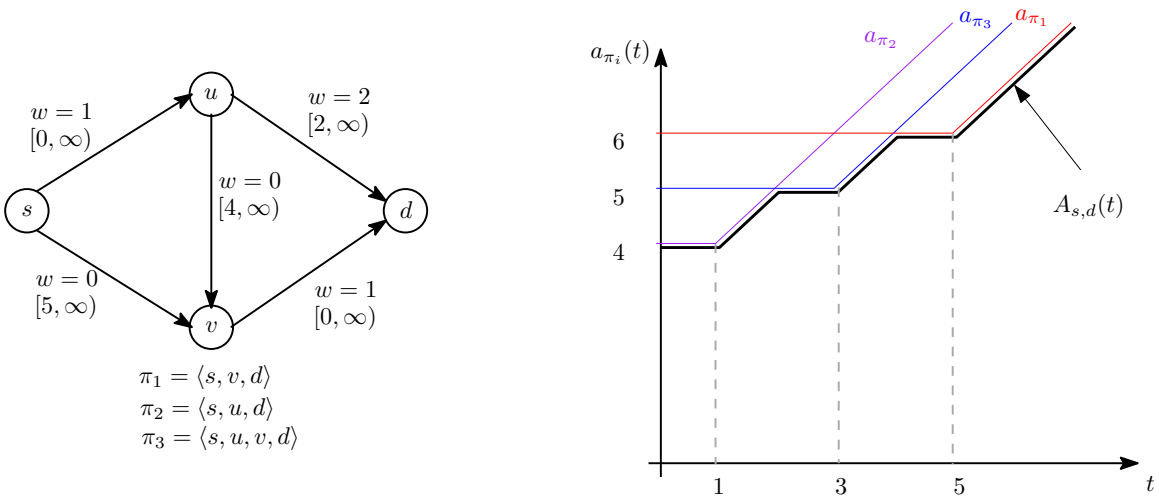


Figure 6: An instance of $\mathcal{TDSP}_{\text{int}}$ and the result $A_{s,d}(t)$ function.

The $\mathcal{TDSP}_{\text{lin}}$ Problem

The second problem that we solve in this chapter is the $\mathcal{TDSP}_{\text{lin}}$ problem which is a more general version of $\mathcal{TDSP}_{\text{int}}$. Here, we consider a time-dependent network $G_l(V, E)$ in which each link $e = (u, v) \in E$ is assigned a piecewise linear function $a_e(t)$ denoting the arrival time at v for start time t at u. The notion of arrival time function is also extendable to a path in the network. Suppose $p = \langle v_1, v_2, \ldots, v_k, v_{k+1} \rangle$ is a path of length k. Then, starting from v_1 at time t one arrives at v_k at time $a_{(v_k, v_{k+1})}(\ldots(a_{(v_2,v_3)}(a_{(v_1,v_2)}(t))))$ which we denote by $a_p(t)$. We study time-dependent networks where costs are changing linearly during specific periods of time. Often piecewise linear function are used to approximate also non-linear functions because of their relative simplicity. An example is a road network where travel times on some links may be modeled to increase linearly when snowfall piles up on the road. This can also model, for example, snow melting periods and snow removal schedules. In $\mathcal{TDSP}_{\text{lin}}$, our goal is to find for all possible start times t at source node s the earliest arrival time at a destination node d. In other words, considering P to be the set of all possible paths from s to d, we are interested in finding the minimum arrival

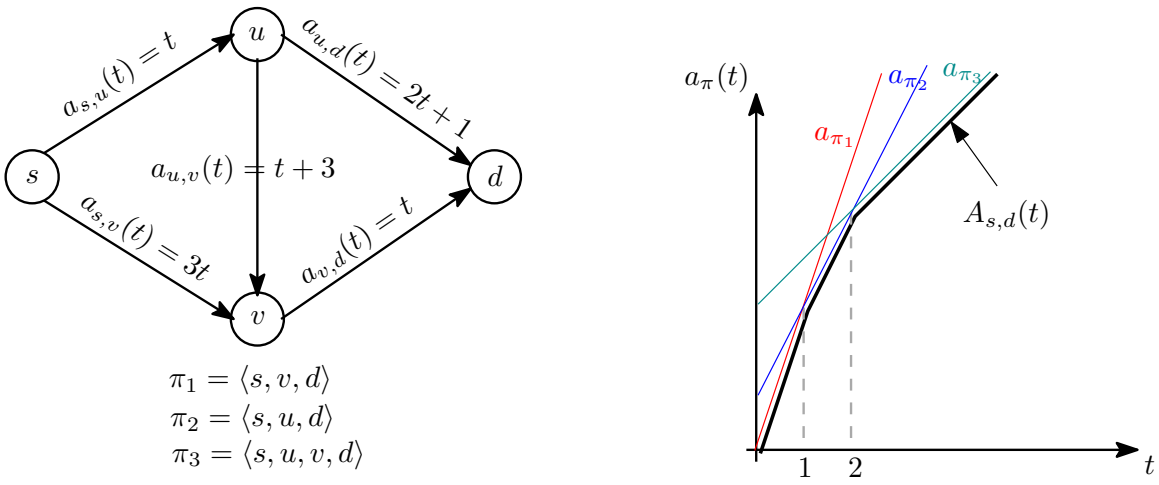


Figure 7: An instance of $\mathcal{TDSP}_{\text{lin}}$ and the result $A_{s,d}(t)$ function.

time function $A_{sd}(t) = \min_{p \in P}(a_p(t))$. Note that we only consider cases where all arrival time functions are piecewise linear real-valued functions. Additionally, the $a_e(t)$ functions are non-decreasing for all $e \in E$ so that the FIFO property holds. Figure 7, shows a sample network with 3 possible paths from s to d and the result minimum arrival time function, $A_{s,d}(t)$.

The problem of finding the earliest arrival time (or shortest path) between two points for a particular given start time can be solved efficiently by applying straightforward modifications to well-known static shortest path methods. For example, a simple modification to Dijkstra's shortest path algorithm [49] using Fredman and Tarjan's implementation [63] finds the earliest arrival time for a given start time in $O(|E| \log \gamma_{max} + |V| \log |V|)$ with the same approach as used in [22]. Here, γ_{max} is the maximum number of linear pieces on link arrival time functions, $a_e(t)$. However, solving the problem for all possible start times appears to be harder even when the $a_e(t)$ functions are piecewise linear and the FIFO property holds, because the shortest path to a destination node d from source node s depends on the start time from s. A naïve algorithm for solving the $\mathcal{TDSP}_{\text{lin}}$ problem computes the earliest arrival time function for every possible path from s to d and then calculates the lower envelope.

This algorithm would need exponential time in many cases because many networks would have an exponential number of possible paths between s and d. In fact, as stated in [125], the problem is NP-hard in its general form (arbitrary arrival time functions).

Results For The $\mathcal{TDSP}_{\text{lin}}$ Problem

We present a novel approach for solving $\mathcal{TDSP}_{\text{lin}}$ which improves on the previous results. Our method is based on the observation that the label-setting and label-correcting algorithms (discussed in Section 3.2.1) compute earliest arrival time functions for all intermediate nodes of the network. We present new, non-trivial, combinatorial properties of arrival time functions. These new insights allow us to focus on finding the earliest arrival time function for the destination node only, and only at crucial time-points. This way, we can discard unnecessary computations. In contrast to previous methods, our algorithms directly compute the earliest arrival time function for every destination node d by tracing time and finding the earliest arrival time only at time instances that may change the earliest arrival time function for d.

Our new algorithm for solving the $\mathcal{TDSP}_{\text{lin}}$ problem runs in time $O((F_d+\gamma)(|E|+|V|\log|V|))$, where F_d is the output size (number of linear pieces needed to represent the earliest arrival time function) and γ is the input size (total number of linear pieces in arrival time functions on links). Our method improves significantly upon the previously known methods. Orda and Rom's label-correcting method [124] had an $O(F_{max}|V||E|)$ time bound where F_{max} is the maximum output size F_d for all possible destination nodes $d \in V$, and Dean [46] had improved that with a label-setting algorithm that has a running time of $O(|E|F^* \log|V|)$, where F^* is the total number of pieces among all output functions. Dean [46] also conjectured that it is possible to have super-polynomial output size for the earliest arrival time function on

some nodes of the network. Foschini *et al.* [60] proved the conjecture to be true. Our method will have super-polynomial time only if the output size is super-polynomial.

Results For The $\mathcal{TDSP}_{\mathbf{int}}$ Problem

In order to solve the $\mathcal{TDSP}_{\text{int}}$ problem, we convert every instance of $\mathcal{TDSP}_{\text{int}}$ into an instance of the $\mathcal{TDSP}_{\text{lin}}$, where the slope of each linear piece is either 0 or 1. The algorithm determines for every start time s the earliest possible arrival time at a given destination d or all nodes of the network. Our approach solves the $\mathcal{TDSP}_{\text{int}}$ problem with $O(\lambda(|E|+|V|\log|V|))$ time complexity. When applied to the $\mathcal{TDSP}_{\text{int}}$ problem, Orda and Rom's algorithm [124] requires time $O(\lambda|V||E|)$ and Dean's algorithm [46] requires time $O(\lambda|E|\log|V|)$. A crucial observation that supports our approach is the fact that there are no more than $O(\lambda)$ pieces in the earliest arrival time function for d. This reduces the number of paths and time instances that need to be considered and improves the performance of our proposed method. The improved method uses structural properties of earliest arrival functions which are of interest in their own right.

Possible Improvements For Special Cases

Our method makes extensive use of static shortest paths algorithms as a base to build the earliest arrival functions to each node. We note that, our algorithm is independent of the static shortest paths algorithm used. One may, e.g., use more efficient static shortest path algorithms for further improvement in special cases. In particular, Thorup's linear time algorithm [139] could be used if link costs are positive integers. In case that the underlying network is a planar network, Henzinger et al. [83] proposed a linear time algorithm for the static shortest path problem that could

be applied to our algorithm. Heuristics like A* [82] could be applied as well.

Organization of the Chapter

The remainder of this chapter is organized as follows. Section 3.2 presents our new algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem and analyzes its running time and establishes its correctness. Section 3.3 presents a new algorithm for the $\mathcal{TDSP}_{\text{int}}$ problem that is based on our $\mathcal{TDSP}_{\text{lin}}$ solution. Section 3.4 concludes the chapter.

3.2 The Shortest Path Problem in a Time-Dependent Network with Piecewise Linear Functions ($\mathcal{TDSP}_{\text{lin}}$)

In this section, we consider the shortest path problem $\mathcal{TDSP}_{\text{lin}}$ on a time-dependent network in which link arrival time functions, $a_e(t), e \in E$, are piecewise linear real-valued functions of a real argument. Additionally, we assume that $a_e(t)$ functions are non-decreasing for all $e \in E$, so that the FIFO property holds. We present and analyze a new algorithm for $\mathcal{TDSP}_{\text{lin}}$ after giving a summary of previous work.

3.2.1 Previous Algorithms and Results

The problem of finding shortest paths in a time-dependent network (also referred to as "earliest arrival time" and "minimum travel time" problem in some texts) was first proposed in 1966 by Cooke and Halsey [42]. They considered time to have discrete values. Most current results are obtained for continuous time because it models real scenarios more accurately.

For the $\mathcal{TDSP}_{\text{lin}}$ problem, the goal is to find the earliest arrival time from s to d for all start times t from s. The arrival time on each path is a function of the start time from the source node and can be obtained by computing the composition of the link arrival time functions along the path. Consequently, the solution of the $\mathcal{TDSP}_{\text{lin}}$ problem is the minimum of all arrival time functions on all paths from s to d. This leads to a naive algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem: compute the arrival time functions of all paths from s to d and compute their lower envelope. For more information on lower envelope algorithms see, e.g., [133]. Although this gives the correct solution, such an algorithm is not efficient in that there could be an exponential number of paths from s to d leading to exponential time complexity.

In Section 2.2.2, we summarized solutions to the $\mathcal{TDSP}_{\text{lin}}$ problem. Note that most previous algorithms for solving the time-dependent shortest path problem [46, 60, 124, 125] consider the single-source shortest path version of the problem, and find arrival time functions to all destinations. A solution to the single-source shortest path version of the time-dependent shortest path problem, also solves the single-pair shortest path version of the problem within the same time bound. We provide asymptotically faster algorithms that solve the single-pair shortest path version of the problem and only find arrival time function at a given destination.

In the following, we point out a flaw in a recent algorithms for the $\mathcal{TDSP}_{\text{lin}}$ problem by Ding *et al.* [50] . They presented a label-setting algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem in FIFO networks with piecewise linear functions. The algorithm scans a sequence of time steps the size of which depends on the values of the arrival time functions. Careful analysis of this algorithm shows that this approach yields a solution with time complexity $O(\alpha(|E| + |V| \log |V|))$ which contains a factor α that depends on the *values* of arrival time functions and can be arbitrarily large, independent of the input size or output size. An example of an instance where α is independent

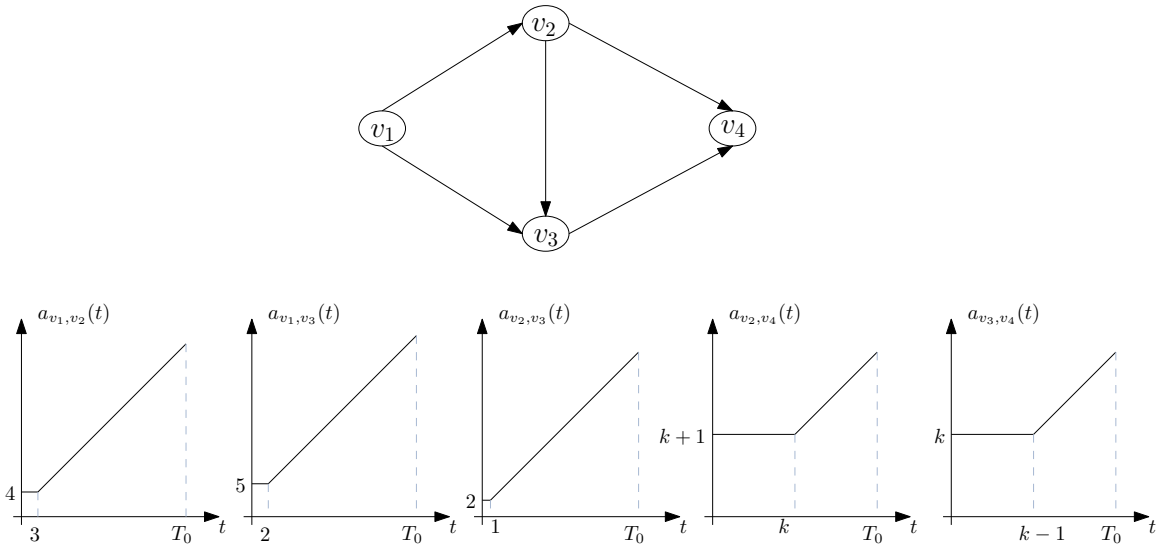


Figure 8: An example network for which the algorithm by Ding et al. [50] requires $\alpha = k - 3$ iterations where k is the *value* of the arrival time function for (v_3, v_4) at time t_s.

on $|E|, |V|$ and γ, and can be arbitrarily large is depicted in Figure 8. In the following, we briefly explain how the algorithm by Ding et al. [50] would process the network shown in Figure 8. For more details on the algorithm, refer to [50]. Consider a network $G(V, E, A)$ as shown in Figure 8 with starting node $v_s = v_1 \in V$, and destination node $v_e = v_4 \in V$ where V and E are the node and edge sets, and A is the set of piecewise linear function on links. Observe the values $k + 1$ and k for the arrival time functions of (v_2,v_4) and (v_3,v_4) at time t_s, respectively. Note that, k can be chosen arbitrarily large, without changing the input size or output size. Given a total time interval $T = [t_s, t_e] = [0, T_0]$, $T_0 > k$, the problem is to find the earliest arrival time function at v_e for all start times in T from v_s. Let $g_i(t)$ be the earliest arrival time function from v_s to v_i for all $v_i \in V$. The algorithm in [50] finds sub-intervals $I_i = [t_s, \tau_i]$ such that for all $t \in I_i$, g_i reflects the correct earliest arrival time function. As the algorithm proceeds, the intervals are extended until they cover T. The algorithm stops when $g_e(t)$ is defined for all $t \in T$ or if v_e is not reachable

from v_s. The main contribution of [50] is the way they extend the interval I_i on each node. The algorithm maintains a priority queue which holds $(\tau_i, g_i(t)),\ v_i \in V$, values sorted by ascending order of $g_i(\tau_i)$. At every iteration, it dequeues the top element of the queue in $(\tau_i, g_i(t))$ and assigns the next top element to $(\tau_l, g_l(t))$ without dequeuing. These values are then used to compute the next earliest possible arrival time from v_s to v_i via any edge (v_f, v_i), namely $\mathcal{EAT} = \min\{a_{f,l}(\tau_l),\ (v_f, v_l) \in E\}$, which is then used to extend τ_i. They show that τ_i can be extended to τ_i' which is the latest start time t that satisfies $g_i(t) \leq \mathcal{EAT}$. For any link (v_i, v_j), the new value $g_j'(t) = a_{i,j}(g_i(t)),\ t \in [\tau_i, \tau_i']$, is then compared with $g_j(t)$ and the minimum value is stored. Figure 9 shows a few iterations of the Ding et al. algorithm for our simple network shown in Figure 8. The first row, shows the starting state of the algorithm where $g_1(t) = t$ and $\tau_i = 0$ for $i \in \{1, 2, 3, 4\}$. The current value of τ_i is indicated by a black dot on the x-axis. The priority queue is shown on the right. All $g_i(\tau_i)$ values in the priority queue are ∞ except for $g_1(0)$ which is 0. In the first iteration (second row in Figure 8), $(\tau_1, g_1(t))$ is removed from the priority queue and the next element of the priority queue is used to obtain the new value of τ_1. The next top element of the priority queue has value $g_l(\tau_l) = \infty$ and there is no incoming edge to v_s, indicating that τ_1 can be extended to $t_e = T_0$. Then, the $g_2(t)$ and $g_3(t)$ functions will be updated, as well as their corresponding values in the priority queue. In the next iteration, $(\tau_2, g_2(t))$ is removed form the priority queue which has value $g_2(\tau_2) = 4$ and the next element is $g_l(\tau_l) = 5$. For all incoming links to v_2, the smallest arrival time for start time 5 is $\mathcal{EAT} = 6$. Hence, we can extend τ_2 to 5 and update g_3 and g_4 for interval $[0, 5]$. We also update the priority queue with new values of $g_i(\tau_i)$ for $i \in \{3, 4\}$. In the third iteration, $(\tau_3, g_3(t))$ is removed from the priority queue and the next element would be $g_l(\tau_l) = 6$. For all incoming links to v_3, the smallest arrival time for start time 6 is $\mathcal{EAT} = 7$. Hence, we can extend τ_3 to 5 and update

g_4 for interval $[0, 5]$. We also update the priority queue to reflect the updated value of $g_4(\tau_4)$. We observe that in all subsequent iterations, the elements removed from the priority queue toggle between $g_2(\tau)$ and $g_3(\tau)$ and increase only by value 1 in each iteration until they reach value of k. Hence, the algorithm of Ding et al. [50], applied to the network shown in Figure 8, requires $\alpha = k - 3$ iteration. We recall that $k+1$ and k are the *values* for the arrival time functions of (v_2,v_4) and (v_3,v_4) at time $t_s = 0$, respectively. Hence, $\alpha = k - 3$ can be chosen arbitrarily large without changing the network size, input size or output size.

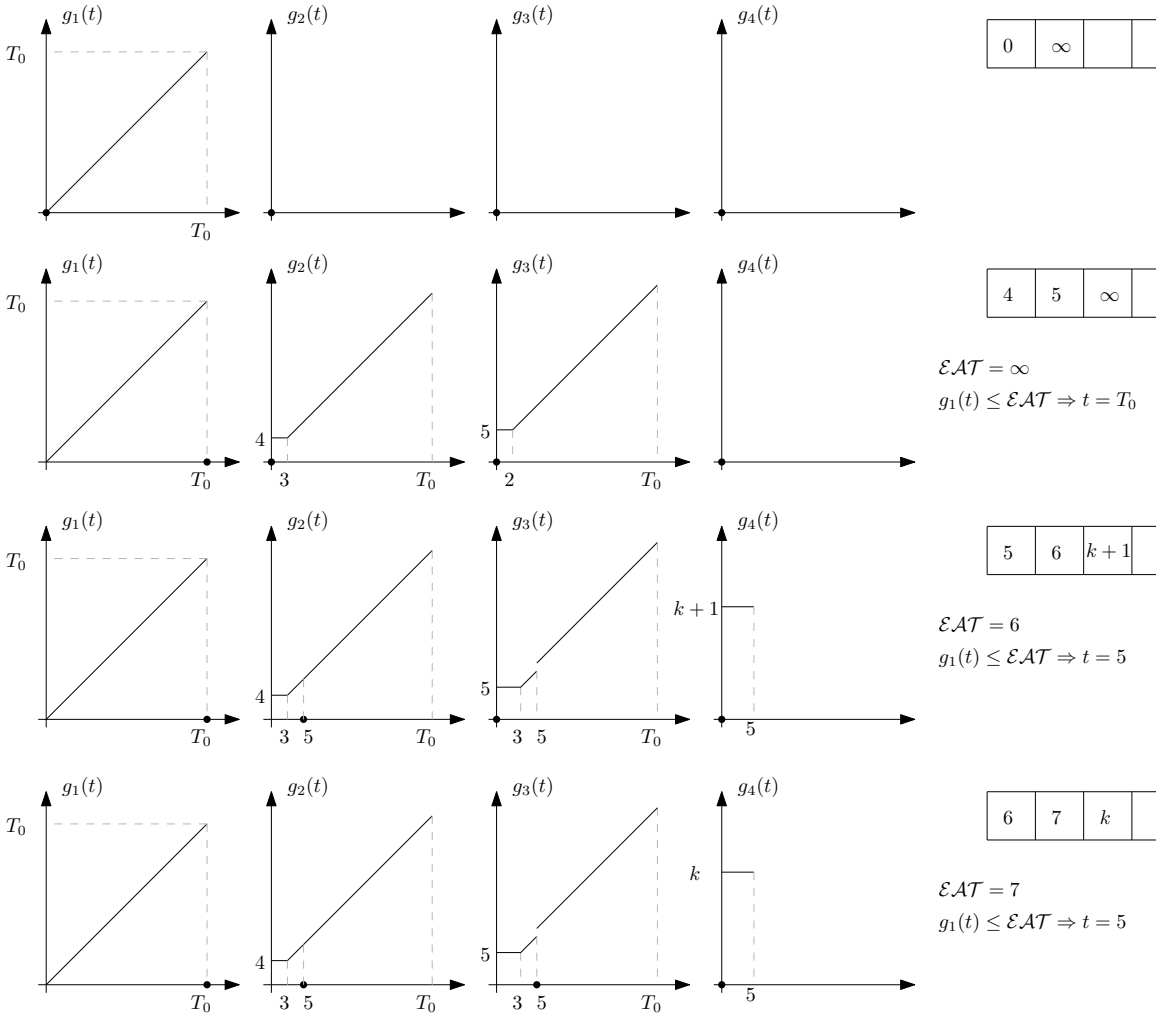


Figure 9: Illustration of the Ding et al. algorithm for the network shown in Figure 8.

3.2.2 Structural Properties

Our new algorithm makes extensive use of certain structural properties of the problem discussed in this section.

Structural Properties of the Earliest Arrival Time Function

The earliest arrival time function from s to d, $A_{sd}(t)$, is a piecewise linear function because all input arrival time functions are assumed to be piecewise linear functions and the function operators used to compute $A_{sd}(t)$ (*function inverse, linear combination, function compound, min, max*) do not change linearity of the result.

We are interested in the endpoints of the linear pieces on the curve $A_{sd}(t)$, and will refer to them as *breakpoints*. We differentiate between two types of breakpoints. First, a breakpoint may represent the intersection between two pieces of arrival time functions on different paths. Second, a breakpoint may represent a breakpoint on one of the arrival time functions for a path from s to d. We refer to the first type as X-point and to the second type as V-point. Figure 10 depicts an arrangement of arrival time functions for four paths and identifies X and V-points.

Every V-point corresponds to a breakpoint on the arrival time function, $a_p(t)$, for some path p from s to d. Each breakpoint on the $a_p(t)$ function is the result of a breakpoint between two linear pieces of arrival time functions on a link of p introduced because of a compound operation for computing $a_p(t)$. In the following lemma, we will show that every breakpoint of a link arrival time function can create at most one V-point on $A_{sd}(t)$.

Lemma 1. *Suppose P is the set of all paths that go through link $e = (v, w) \in E$ and $a_e(t)$ is the arrival time function for e and has γ_e breakpoints. Then, all arrival time functions $a_p(t), p \in P$, create, in total, at most γ_e V-points on $A_{sd}(t)$.*

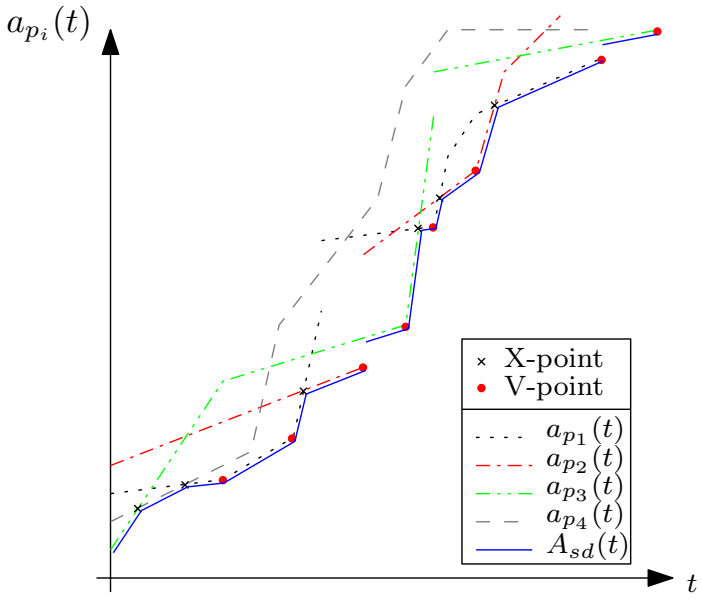


Figure 10: An illustration of X and V-points. Curves $a_{p_1}(t)$, ..., $a_{p_4}(t)$ are arrival time function for four paths p_1, ..., p_4, and $A_{sd}(t)$ is the final arrival time function from s to d.

Proof: Consider the following representation of the piecewise linear function $a_e(t)$:

$$a_e(t) = \begin{cases} \alpha^1 t + \beta^1 & 0 \le t \le T^1 \\ \alpha^2 t + \beta^2 & T^1 < t \le T^2 \\ \vdots & \vdots \\ \alpha^{\gamma_e} t + \beta^{\gamma_e} & T^{\gamma_e - 1} < t \le T^{\gamma_e} \\ \infty & T^{\gamma_e} < t \quad . \end{cases}$$

For every breakpoint $T^i, i = 1 \ldots \gamma_e$, consider path p^i to be the concatenation of a path with the latest start time (LST) from s which arrives at v at time T^i, link (u, v), and a path with earliest arrival time (EAT) to d which starts from w at time $\alpha^i T^i + \beta^i$. Because of the definition of p^i, for any path $p \in P$ other than p^i, T^i will

create a breakpoint either at (LST, EAT) or to the left and above this point. Since (LST, EAT) is on $a_{p^i}(t)$ and the FIFO property holds, any points that are to the left and above (LST, EAT) are not on $A_{sd}(t)$. Hence, other paths cannot create new breakpoints on $A_{sd}(t)$. This proves that all arrival time functions $a_p(t), p \in P$, create in total at most γ_e V-points on $A_{sd}(t)$. □

Let $\gamma = \sum_{e \in E} \gamma_e$ be the total number of breakpoints on link arrival time functions in the entire network. Since every V-point arises from a breakpoint on some link arrival time function, Lemma 1 implies that there cannot be more than $O(\gamma)$ V-points on $A_{sd}(t)$.

Super-Polynomial Output Size

In [46], the author conjectured that in a FIFO network with piecewise linear link arrival time functions, the earliest arrival time function $A_{sd}(t)$ from a source node s to a destination node d may have more than a polynomial number of linear pieces. In [60], the authors proved the conjecture. This means that there exist network structures that result in a super-polynomial complexity for earliest arrival time functions between some pairs of points in the network.

The *super-polynomial structure* could appear as a subnetwork of the actual input network, resulting in earliest arrival time functions with possibly super-polynomial number of pieces (breakpoints) for destination nodes whose shortest path from s passes through the super-polynomial structures. However, the earliest arrival time function from s to d could still be of linear size since the earliest arrival time path may not intersect the super-polynomial structure at all. In this case, F_{max} would be of super-polynomial size and F_d would be of linear size.

3.2.3 A New Algorithm For Instances With Polynomial Size Output

Our new algorithm [47, 48] is based on the idea that instead of building all earliest arrival time functions to all nodes in the network, we calculate only the earliest arrival time function to destination node d. An outline of our method is given in algorithm $\mathcal{A}_{\text{lin}}$. In the remainder, we discuss our method in more detail.

The main idea is to find all start times for which the earliest arrival time function from s to d, $A_{sd}(t)$, changes from one linear piece to another as well as linear functions on the left and right of these breakpoints. In Section 3.2.2, we introduced two types of breakpoints in $A_{sd}(t)$: V-points and X-points. We also showed that at most $O(\gamma)$ V-points exist on $A_{sd}(t)$, where γ is the total number of pieces in all input arrival time functions. All V-points that can potentially be on $A_{sd}(t)$ could be captured by computing, for every breakpoint at time T on the arrival time function $a_e(t)$ of each link $e = (v, w)$, the latest departure time (LDT) at s to arrive at v at time T and the earliest arrival time (EAT) at d for departure time $a_e(T)$ at w. Point (LDT, EAT) is a potential V-point on $A_{sd}(t)$. Note that the earliest arrival time at a destination node could be obtained by running a modified version of Dijkstra's static shortest path algorithm for a given start time at the source node. For more details on time-dependent version of Dijkstra's static shortest path algorithm see [46]. Similarly, a modified version of reverse Dijkstra's static shortest path algorithm could be used to obtain the latest departure time at the source node corresponding to a given arrival time at the destination node. Regarding the reversibility of the $\mathcal{TDSP}_{\text{lin}}$ problem see [45].

Let L be a list and S be a stack that hold information about V-points and X-points in Algorithm $\mathcal{A}_{\text{lin}}$, recursively. In the following, we define the functions used

in Algorithm $\mathcal{A}_{\text{lin}}$:

- $Dijkstra(w, d, a_e(T_e^i))$ returns the earliest arrival time at d for departure time $a_e(T_e^i)$ at w.
- $ReverseDijkstra(v, s, T_e^i)$ returns the latest departure time at s corresponding to arrival time T_e^i at v.
- $LeftFunction(s, d, LST)$ returns the linear function on $A_{sd}(t)$ which is on the right of V-point at time LST (see the following discussion for more details).
- $RightFunction(s, d, LST)$ returns the linear function on $A_{sd}(t)$ which is on the left of V-point at time LST (see the following discussion for more details).
- $InsertToList(L, \{LST, EAT, f_l, f_r\})$ adds the information of V-point at time LST to list L.
- $Sort(L)$ sorts list L by the ascending order of LST values.
- $RemoveItem(L)$ returns and removes the first element of L.
- $IntersectionPoint(RF_1, LF_2)$ finds the intersection point of the function on the right of left V-point and the one on the left of right V-point.
- $Overlap(RF_1, LF_2)$ returns true if the function on the right of left V-point has the same slope as the one on the left of right V-point, and false otherwise.
- $Push(S, (PX_2, PY_2, RF_1, LF_2))$ inserts the information of the intersection point to stack S.
- $Pop(S)$ returns and removes an element from the top of stack S.
- $AddLinearPiece(A_{sd}(t), f_l, PX_1, PX_2)$ adds linear function f_l to $A_{sd}(t)$ from time PX_1 to PX_2.

Algorithm $\mathcal{A}_{\text{lin}}$ (G, V, E, s, d)

```
1: A_sd(t) ← NULL
2: for every link e = (v, w) ∈ E do
3:     for i = 0 to λ_e do
4:         LST ← ReverseDijkstra(v, s, T_e^i)
5:         EAT ← Dijkstra(w, d, a_e(T_e^i))
6:         TMP ← Dijkstra(s, d, LST)
7:         if EAT = TMP then
8:             f_l ← LeftFunction(s, d, LST)
9:             f_r ← RightFunction(s, d, LST)
10:            InsertToList(L, {LST, EAT, f_l, f_r})
11: Sort(L)
12: {LST_1, EAT_1, LF_1, RF_1} ← RemoveItem(L)
13: while NotEmpty(L) do
14:     {LST_2, EAT_2, LF_2, RF_2} ← RemoveItem(L)
15:     if Overlap(RF_1, LF_2) then
16:         AddLinearPiece(A_sd(t), RF_1, LST_1, LST_2)
17:     else
18:         (PX_1, PY_1) ← (LST_1, EAT_1)
19:         (PX_2, PY_2) ← IntersectionPoint(RF_1, LF_2)
20:         Push(S, (PX_2, PY_2, RF_1, LF_2))
21:         while NotEmpty(S) do
22:             (PX_2, PY_2, f_l, f_r) ← Pop(S)
23:             TMP ← Dijkstra(s, d, PX_2)
24:             if TMP = PY_2 then
25:                 AddLinearPiece(A_sd(t), f_l, PX_1, PX_2)
26:                 PX_1 ← PX_2
27:             else
28:                 f_m ← LeftFunction(s, d, PX_2)
29:                 (IX_1, IY_1) ← IntersectionPoint(f_l, f_m)
30:                 (IX_2, IY_2) ← IntersectionPoint(f_m, f_r)
31:                 Push(S, (IX_2, IY_2, f_m, f_r))
32:                 Push(S, (IX_1, IY_1, f_l, f_m))
33:         AddLinearPiece(A_sd(t), f_r, PX_2, LST_2)
34:     {LST_1, EAT_1, LF_1, RF_1} ← {LST_2, EAT_2, LF_2, RF_2}
35: if EAT_1 ≠ ∞ then
36:     AddLinearPiece(A_sd(t), RF_1, LST_1, ∞)
37: return (A_sd(t))
```

In order to make sure that (LST, EAT) is on the final solution, we calculate the earliest arrival time to d starting from s at time LST. If the result arrival time is the same as EAT, then (LST, EAT) is indeed a V-point on $A_{sd}(t)$. In this case, we also compute the linear pieces to the left and right of each V-point. These are pieces with the earliest arrival time and the smallest (greatest) slope on the right (left) vicinity of each V-point. Figure 11 shows a sample $A_{sd}(t)$ function after all V-points have been detected. Lines 1 through 10 of algorithm $\mathcal{A}_{\text{lin}}$ state the pseudo-code to find all V-points along with their left and right linear functions. Note that, given a time t_0, the adjacent linear pieces with smallest (greatest) slope can be computed

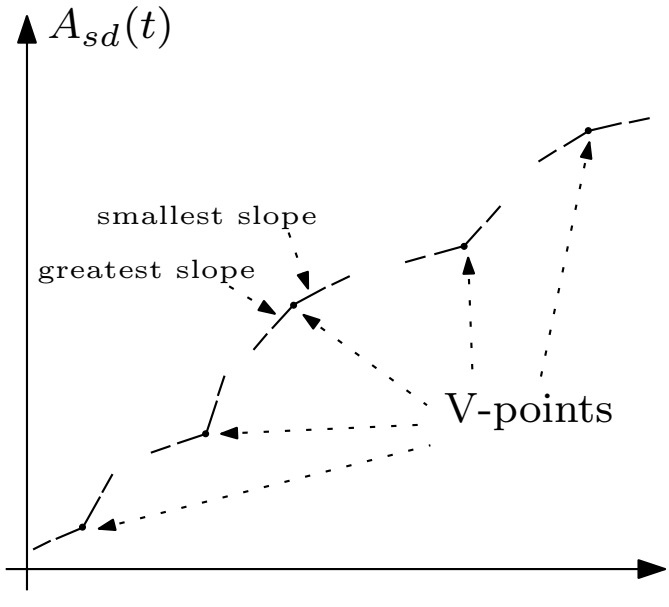


Figure 11: A sample $A_{sd}(t)$ function with all V-points and their adjacent linear functions.

while computing the earliest arrival time to d for start time t_0 (Lines 8-9). This is accomplished by keeping the product of slopes for each node in the shortest path tree as a secondary key when Dijkstra's algorithm finds two or more entries of the heap that have the same arrival time value. In this case, selecting the entry with smallest (greatest) slope leads to the adjacent linear pieces with smallest (greatest) slope. To show the correctness of this approach consider any two paths from s to d starting at time t_0 with equal arrival times but different slopes on their arrival time functions. The first time where they have equal arrival time values is either at d, or at some earlier node d'. In the latter case, they will share the same path from d' to d. In either case, selecting the smallest (greatest) product of slope values from the heap, among equal arrival time values, maintains the smallest (greatest) slope.

Thus far we have determined all V-points and the slope of $A_{sd}(t)$ in their vicinity. We build the remaining part of $A_{sd}(t)$ by adding all X-points and missing pieces between every pair of consecutive V-points on $A_{sd}(t)$. Due to the linearity of the

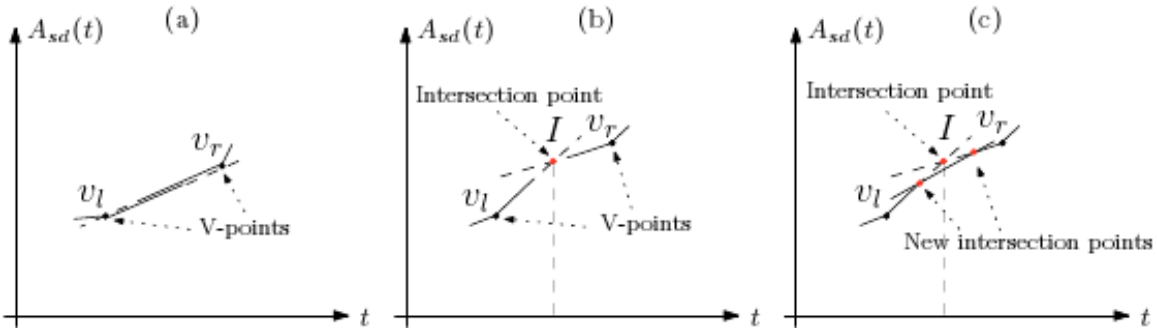


Figure 12: (a) Overlapping pieces (b) Intersection point on $A_{sd}(t)$ (c) Intersection point hidden by another linear piece

input arrival functions, the X-points between two consecutive V-points are in concave position (seen from below). Consider two consecutive V-points, V_l and V_r, found in the previous step together with the linear pieces in their vicinity. Two cases arise for the linear pieces to the right of V_l and to the left of V_r. They either overlap, or they intersect in some point $I = (x_I, y_I)$. If they overlap, the piece connecting the two V-points is part of the solution (Figure 12-a). In case of an intersection, two cases are possible. First, if calculating the earliest arrival time for start time x_I at s returns the same arrival time y_I at d as for the intersection point, the pieces v_l to I and I to v_r are part of the solution (Figure 12-b). Second, if calculating the earliest arrival time for start time x_I at s returns a value less than y_I at d, there exists a new linear piece that hides the intersection point $I = (x_I, y_I)$ (Figure 12-c). The linear pieces to the right of V_l and to the left of V_r intersect the new piece, and we recurse for new intersections. See Theorem 1 for further details. Lines 11 through 36 of algorithm $\mathcal{A}_{\text{lin}}$ are the pseudo-code for finding X-points and adding pieces to solution function $A_{sd}(t)$.

Theorem 1. *Given a source node s and destination node d, algorithm $\mathcal{A}_{lin}$, outlined above, correctly determines $A_{sd}(t)$ for all $t \in [0, \infty)$.*

Proof: The algorithm first finds all V-points on $A_{sd}(t)$ along with linear pieces to

the left and right of each V-point (Lines 1-10). From the proof of Lemma 1, it follows that no V-points other than those considered by algorithm $\mathcal{A}_{\text{lin}}$ can be on $A_{sd}(t)$. The algorithm picks every two consecutive V-points to compute all X-points between them (Lines 11-36). Let $v_l = (x_l, y_l)$ and $v_r = (x_l, x_r)$ be two consecutive V-points on $A_{sd}(t)$. Suppose that RF and LF are the linear pieces to the right of v_l and to the left of v_r, respectively. Either RF and LF overlap or they intersect. If they overlap, the linear piece on RF (or LF) from x_l to x_r is part of the solution function since no other V-points are possible between v_l and v_r. On the other hand, if the two functions intersect in some point $I = (x_I, y_I)$ and the intersection is on $A_{sd}(t)$, then the linear piece on RF from x_l to x_I is on $A_{sd}(t)$ since no other V-points are possible between v_l and v_r. If I is not on $A_{sd}(t)$, then there must be another linear piece preventing it from being on the solution. The algorithm determines such a piece with maximum slope. The extension of the linear piece must intersect both RF and LF, since otherwise, there must be another V-point between v_l and v_r. Let I_l and I_r be the two intersection points. The algorithm now recursively performs what has been done for I, first for I_l and for then I_r by pushing them into S (Lines28-32). Starting from v_l, it adds linear pieces to the solution function once I_l is found to be on $A_{sd}(t)$. Then, it moves to the next intersection. As a last step, the algorithm adds to the solution function the last piece on LF between the last intersection and x_r. Since every X-point is verified to be on $A_{sd}(t)$ and no more V-points are possible between two consecutive V-points, the algorithm finds all X-points. Since V-points and X-points are the only breakpoints on $A_{sd}(t)$, algorithm $\mathcal{A}_{\text{lin}}$ correctly finds all linear pieces of $A_{sd}(t)$. □

Theorem 2. *The time complexity of algorithm $\mathcal{A}_{lin}$ is $O((F_d+\gamma)(|E|+|V|\log|V|))$.*

Proof: The algorithm first executes a slightly modified version of both standard and reverse Dijkstra's shortest path algorithm for each breakpoint in every link arrival

time function in order to find all possible V-points. It then executes another modified version of Dijkstra's algorithm to find the greatest and smallest slope pieces close to each V-point. For every link (v, w) and start time at v we compute the arrival time at w in $O(1)$ time using an *amortized analysis*. This follows from the fact that breakpoints are sorted in time and we can therefore execute Dijkstra's shortest path algorithm incrementally. For a given start time at s, the earliest arrival time at d is computed in the same time as Dijkstra's algorithm, that is $O(|E| + |V| \log |V|)$. Consequently, the time-complexity for computing all V-points is $O(\gamma(|E| + |V| \log |V|))$ where γ is the total number of linear pieces in arrival time functions on links. Then, for computing all X-points, the algorithm executes a modified Dijkstra's algorithm as many time as we find intersection points. At each intersection point found, we determine the linear piece with greatest slope that hides the intersection point. This guarantees that every time we run a modified Dijkstra's algorithm at an intersection point we obtain a new linear piece that is part of the solution $A_{sd}(t)$. As a result, we will execute the modified Dijkstra's algorithm at most as many times as there are X-points on $A_{sd}(t)$. With F_d defined as the number of linear pieces on $A_{sd}(t)$, computing all X-points requires time $O(F_d(|E| + |V| \log |V|))$. Hence, the total time complexity of algorithm $\mathcal{A}_{\text{lin}}$ is $O(F_d + \gamma)(|E| + |V| \log |V|))$. □

3.3 The Shortest Paths Problem on Time-Dependent Networks with Availability Intervals ($\mathcal{TDSP}_{\mathbf{int}}$)

3.3.1 A Reduction from $\mathcal{TDSP}_{\mathbf{int}}$ to $\mathcal{TDSP}_{\mathbf{lin}}$

In the following, we discuss how every instance of a time-dependent network with availability intervals for the $\mathcal{TDSP}_{\text{int}}$ problem can be converted into an equivalent time-dependent network with piecewise linear functions by replacing intervals and weights with arrival time functions for every link in the network.

Let w_e^m and $[l_e^m, r_e^m]$ be the m^{th} weight and interval on link $e = (i, j) \in E$ in a time-dependent network with intervals (see Figure 13(a)). By definition of the $\mathcal{TDSP}_{\text{int}}$ problem, for all departure times t from i between l_e^m and r_e^m, the arrival time is $a_e(t) = t + w_e^m$. Additionally, for all times t not in any intervals for that link, the earliest departure time is the smallest available time greater than t, say l_e^s. In this case, the earliest arrival time at j is $a_e(t) = l_e^s + w_e^s$. Finally, for all times t greater than the closing time of the last interval of the link, ($[l_e^{M_e}, r_e^{M_e}]$), the link is not available and the earliest arrival time is ∞. Figure 13(b),(c) depicts the result of this conversion and piecewise linear function $a_e(t)$ for a link $e = (i, j)$.

Note that the resulting time-dependent network is indeed a FIFO network since arrival time functions are non-decreasing. Additionally, for each link e, $a_e(t)$ is a continuous piecewise linear function. Consequently, in the following we focus on algorithms that solve the $\mathcal{TDSP}_{\text{lin}}$ problem applied to the time-dependent network obtained by our conversion process.

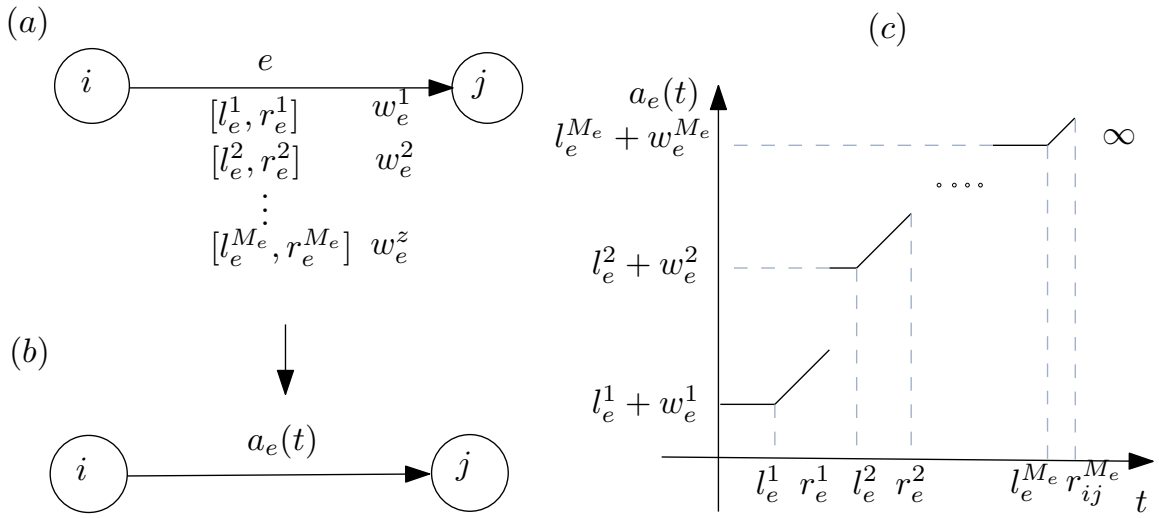


Figure 13: a) A link in the time-dependent network with intervals. b) A converted link in the new time-dependent network with piecewise linear functions. c) The arrival time function on the converted link.

3.3.2 Structural Properties

O or 1 Slopes on the Output Function

It follows from our conversion process discussed in Section 3.3.1 that arrival time functions on links are piecewise linear functions with FIFO property. Moreover, all slopes are either 0 or 1 since for any point in time, either a link is available for use or it is unavailable. As described earlier, on a path p from a source node s to destination d, the earliest arrival time for all times t, denoted by $a_p(t)$, is obtained by a sequential composition of the link arrival time functions for all links of the path. Similarly, the earliest arrival time function from s to d for all times t, denoted by $A_{sd}(t)$, is the minimum of the arrival time functions on all paths from s to d. Both operations, composition and minimum, do not change the slopes to any values other than 0 or 1. Consequently, all linear pieces on both $a_p(t)$ and $A_{sd}(t)$ have slope either 0 or 1.

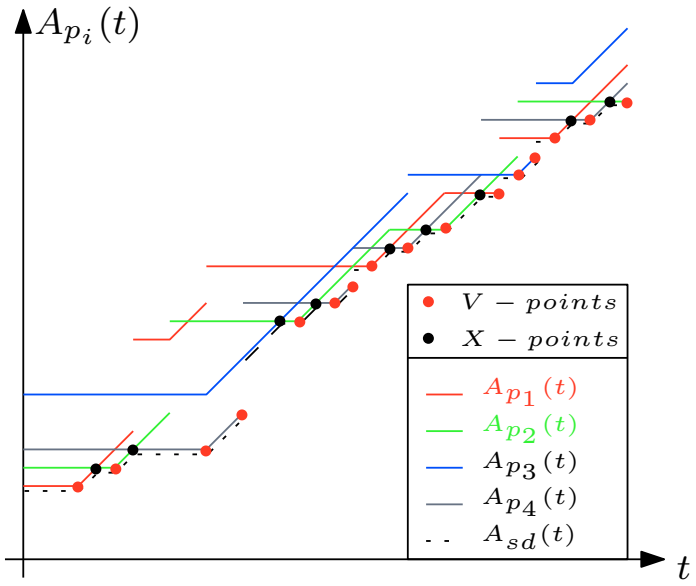


Figure 14: An example showing V-points and X-points.

$O(\kappa)$ Output Size

As discussed in Section 3.2.3, it follows from Lemma 1 that there are at most κ V-points on the solution function $A_{sd}(t)$ of the $\mathcal{TDSP}_{\text{int}}$ problem. Here, κ denotes the input size which is the total number of linear pieces in arrival time functions on links.

Lemma 2. *There are at most $O(\kappa)$ X-points on the solution function $A_{sd}(t)$.*

Proof: An X-point is the intersection of slope-0 and slope-1 linear pieces. For two consecutive V-points on $A_{sd}(t)$ it follows from the fact that all slopes are 0 and 1 there cannot be more than one X-point between these two consecutive V-points. By Lemma 1, there are $O(\kappa)$ V-points on $A_{sd}(t)$. Hence, there are at most $O(\kappa)$ X-points on $A_{sd}(t)$. □

Figure 14 shows V-points and X-points on an arrangement of four path arrival time functions.

Theorem 3. *There are at most $O(\kappa)$ breakpoints on the solution function of the $\mathcal{TDSP}_{int}$ problem, $A_{sd}(t)$.*

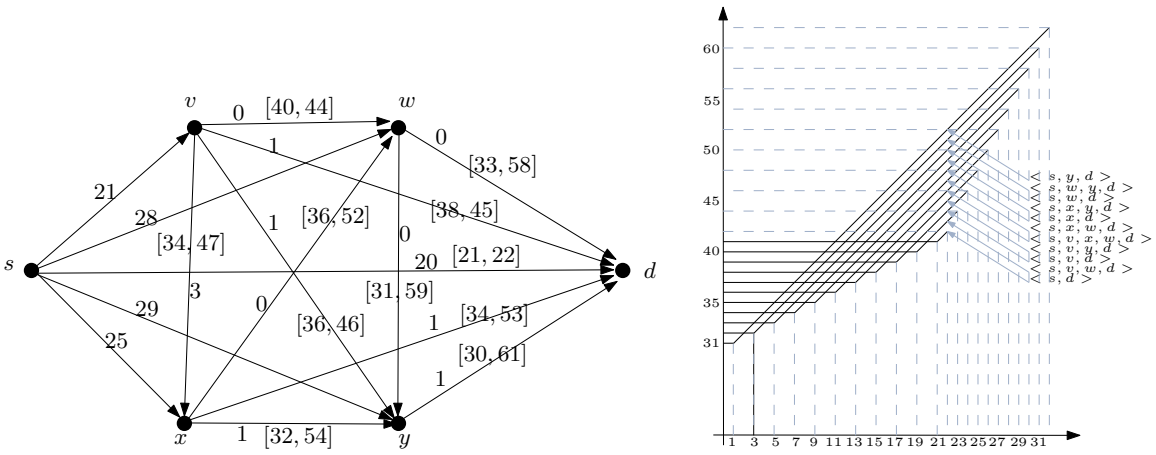


Figure 15: An example of a network with $O(E)$ pieces on the EAT function to d

Proof: The result follows from Lemma 1 and Lemma 2. □

Figure 15 shows a time-dependent network with availability intervals. There are $\Theta(|E|)$ breakpoints on $A_{sd}(t)$ for the case of $O(1)$ intervals on each link. Note that the example could be generalized to cases of multiple intervals (translates to κ_e linear pieces on $a_e(t)$ in the time-dependent network) if we build the i^{th} interval on a link by adding some constant value to the $(i-1)^{st}$ interval so that they do not overlap. This will result in a network with $\Theta(\kappa) = \Theta(\lambda)$ breakpoints on $A_{sd}(t)$, where λ is the total number of intervals in the network.

3.3.3 Solving the $\mathcal{TDSP}_{\text{lin}}$ Instances Created for Solving $\mathcal{TDSP}_{\text{int}}$ Problems

Applying Lower Envelope Algorithms

Suppose P is the set of all possible paths from s to d, and $a_p(t)$ is the arrival time function of path $p \in P$. Then, the earliest arrival time function from s to d for all times t, $A_{sd}(t) = \min_{p \in P}(a_p(t))$ is the lower envelope of all possible earliest arrival time functions. A naive algorithm for solving the $\mathcal{TDSP}_{\text{int}}$ problem could be to

use well-studied lower envelope algorithms (see, e.g., [133, 3]). Unfortunately, the number of such paths and, therefore, the number of arrival time functions could be exponential which makes this approach inefficient.

Applying Label-Correcting Algorithms

For general time-dependent networks for which the FIFO property holds and have piecewise linear functions, Orda and Rom [124] proposed an algorithm that has a time bound of $O(F_{max}|V||E|)$, where F_{max} is the maximum number of pieces in a given output function $A_{si}(t)$ for source node s to any node i. As shown, in the $\mathcal{TDSP}_{\text{int}}$ problem there are at most $O(\lambda)$ pieces in the output function for each node which results in a total running time of $O(\lambda|E||V|)$.

Applying Label-Setting algorithms

For time-dependent networks with FIFO property and piecewise linear arrival time function on links, Dean [46] suggested a label-setting algorithm. This algorithm has a running time of $O(|E|F^* \log |V|)$, where F^* is the total number of pieces among all output functions. When applied to the $\mathcal{TDSP}_{\text{int}}$ problem, this approach results in a running time of $O(\lambda|E| \log |V|)$.

Applying our Improved $\mathcal{A}_{\text{lin}}$ Algorithm to Instances Created for solving $\mathcal{TDSP}_{\text{int}}$ Problems

In Section 3.2, we presented a new algorithm that solves the $\mathcal{TDSP}_{\text{lin}}$ problem in time $O((F_d + \lambda)(|E| + |V| \log |V|))$ where F_d is the output size (number of linear pieces on the earliest arrival time function to d). For instances created for solving $\mathcal{TDSP}_{\text{int}}$ problems, the output size is $O(\lambda)$. Hence, the time complexity would be $O(\lambda(|E| + |V| \log |V|))$. Note that $\mathcal{TDSP}_{\text{lin}}$ considers only one destination node

Algorithm $\mathcal{A}_{\text{int}}$ (G, V, E, s)

```
1: for i = 1 to |V| do
2:     A[i] ← NULL
3: for every link e = (v, w) ∈ E do
4:     for i = 0 to κ_e do
5:         LST ← ReverseDijkstra(v, s, T_e^i)
6:         EAT ← DijkstraAllDestinations(v, T_e^i)
7:         TMP ← DijkstraAllDestinations(s, LST)
8:         f_l ← LeftFunction(s, LST)
9:         f_r ← RightFunction(s, LST)
10:        for i = 1 to |V| do
11:            if EAT[i] = TMP[i] then
12:                InsertToList(L[i], {LST, EAT[i], f_l[i], f_r[i]})
13: for i = 1 to |V| do
14:     Sort(L[i])
15: for i = 1 to |V| do
16:     {LST_1, EAT_1, LF_1, RF_1} ← RemoveItem(L[i])
17:     while NotEmpty(L[i]) do
18:         {LST_2, EAT_2, LF_2, RF_2} ← RemoveItem(L[i])
19:         if Overlap(RF_1, LF_2) then
20:             AddLinearPiece(A[i](t), RF_1, LST_1, LST_2)
21:         else
22:             (IX, IY) ← IntersectionPoint(RF_1, LF_2)
23:             AddLinearPiece(A[i](t), RF_1, LST_1, IX)
24:             AddLinearPiece(A[i](t), LF_2, IX, LST_2)
25:         {LST_1, EAT_1, LF_1, RF_1} ← {LST_2, EAT_2, LF_2, RF_2}
26:     if EAT_1 ≠ ∞ then
27:         AddLinearPiece(A[i](t), RF_1, LST_1, ∞)
28: return (A)
```

whereas $\mathcal{TDSP}_{\text{int}}$ requires shortest paths from s to *all* nodes in the network. Algorithm $\mathcal{A}_{\text{int}}$ computes, for a given source node s, the earliest arrival time functions to all nodes in the network within the same time-complexity $O(\lambda(|E| + |V| \log |V|))$.

Function $A[i](t)$ represents the earliest arrival time function from s to every node i of the network. In Line 2 of algorithm $\mathcal{A}_{\text{int}}$, this function has been initialized to ∞ for every node of the network. We will gradually build these functions by adding linear pieces to them. For all nodes i, Lines 3 through 12 find all V-points on $A[i](t)$

and add them to list $L[i]$ for later use. Here, $ReverseDijkstra(v, s, T_e^i)$ is the same as in algorithm $\mathcal{A}_{\text{lin}}$. It returns the latest possible start time from s to arrive at v at time T_e^i. Similarly, $DijkstraAllDestinations(v, T_e^i)$ is a slightly modified version of the static shortest path algorithm from v to every node i starting at breakpoint time T_e^i. It returns the earliest possible arrival time at *all* nodes if one starts from v at time T_e^i. Note that the result of this function is an array holding the earliest arrival time to node i in its i^{th} position. For every node i, functions $LeftFunction(s, LST)$ and $RightFunction(s, LST)$ return the linear function on the left and on the right of the V-point that occurs at time LST. Line 11 checks whether a V-point is on the solution function or not, and adds it to the list accordingly. The algorithm builds $A[i](t)$ for each node i by adding linear pieces between every two consecutive V-points. We first sort the list of V-points for each node (Line 13). Then, in Lines 15 through 27, the algorithm adds one or two linear pieces depending on whether linear pieces between two consecutive V-points overlap or intersect, respectively.

As a further extension, consider the all pair version of $\mathcal{TDSP}_{\text{int}}$ where shortest paths are calculated between all pairs of nodes. We observe that the reverse shortest path algorithm reports at each breakpoint not only the latest start time from source node s, but also from all nodes in the network. As a result, we can compute the earliest arrival time function from all source nodes to all destinations using a slight modification of algorithm $\mathcal{A}_{\text{int}}$. Thus, all $|V|^2$ earliest arrival time functions for the all pairs version of $\mathcal{TDSP}_{\text{int}}$ can be computed in time $O(\lambda|V|^2)$.

3.4 Conclusions and Future Work

In this chapter, we presented new algorithms for shortest path problems on two types of time-dependent networks with FIFO property: networks where edges have

time-dependent availability intervals ($\mathcal{TDSP}_{\text{int}}$), and networks where edges have piecewise linear arrival time functions ($\mathcal{TDSP}_{\text{lin}}$). We solved the $\mathcal{TDSP}_{\text{int}}$ problem by reducing to a special case of the second problem, $\mathcal{TDSP}_{\text{lin}}$. For this, we presented a novel solution based on new, non-trivial, combinatorial properties of arrival time functions. These new insights allow us to focus on finding the earliest arrival time function for the destination node only, and only at crucial time-points. This way, we can discard unnecessary computations. In contrast to previous methods, our algorithms directly compute the earliest arrival time function for every destination node d by tracing time and finding the earliest arrival time only at time instances that may change the earliest arrival time function for d. The algorithms presented in this chapter improve significantly upon the previously known methods for the $\mathcal{TDSP}_{\text{int}}$ and $\mathcal{TDSP}_{\text{lin}}$ problems.

Both of our methods make extensive use of static shortest paths algorithms. One may use other efficient static shortest path algorithms for further improvement in special cases. For example, in planar networks, applying linear time shortest path algorithms will further improve our results. In many practical networks, heuristics such as A^* could also be applied to improve the practical performance of our methods.

A natural extension and generalization of the $\mathcal{TDSP}_{\text{int}}$ and $\mathcal{TDSP}_{\text{lin}}$ problems is to apply the notion of time-dependence to polyhedral surfaces to capture changes in weight values. Unlike in time-dependent networks, on polyhedral surfaces paths not only utilize nodes and edges of the input, but also can cross faces. Thus the discrete problem encountered in networks becomes a continuous problem when studying polyhedral surfaces. Moreover, the travel-time function on a sub-path inside a face of a polyhedral surface depends on the size of the sub-path, whereas the travel-time function on an edge of a network is fixed and defined a priori. One could exploit Aleksandrov et al.'s [8] discretization method to obtain a time-dependent graph. Then,

applying our time-dependent shortest path algorithms to the resulting graph yields an approximation algorithm for this problem. However, one might be able to improve the time bound using structural properties inherent in the problem.

Chapter 4

Improved Approximation for Time-Dependent Shortest Paths

We study the approximation of minimum travel time paths in time-dependent networks. The travel time in each link of the network is a piecewise linear function of the start time from the start node of the link. Our objective is to find the minimum travel time to a destination node d, for all possible start times at source node s. Dehne *et al.* proposed an exact output-sensitive algorithm for this problem [47, 48] that, in most cases, improves upon existing algorithms. They also provide an approximation algorithm. In [59, 60], Foschini *et al.* show that this problem has super-polynomial complexity, and present an ϵ–approximation[1] algorithm that runs $O(\frac{\gamma}{\epsilon}\log(\frac{T_{max}}{T_{min}})\log(\frac{L}{\gamma\epsilon T_{min}}))$ shortest path computations, where γ is the total number of linear pieces in travel time functions on links, L is the horizontal span of the travel time function and T_{min} and T_{max} are the minimum and maximum travel time values, respectively.

In this chapter, we present two ϵ–approximation algorithms that improve upon Foschini's result. Our first algorithm runs $O(\frac{\gamma}{\epsilon}(\log(\frac{T_{max}}{T_{min}}) + \log(\frac{L}{\gamma T_{min}})))$ shortest path

[1] We denote by ϵ–approximation any algorithm whose quality is less than or equal to $(1+\epsilon)\cdot\mathcal{OPT}$, where $\mathcal{OPT}$ is the optimal achievable quality.

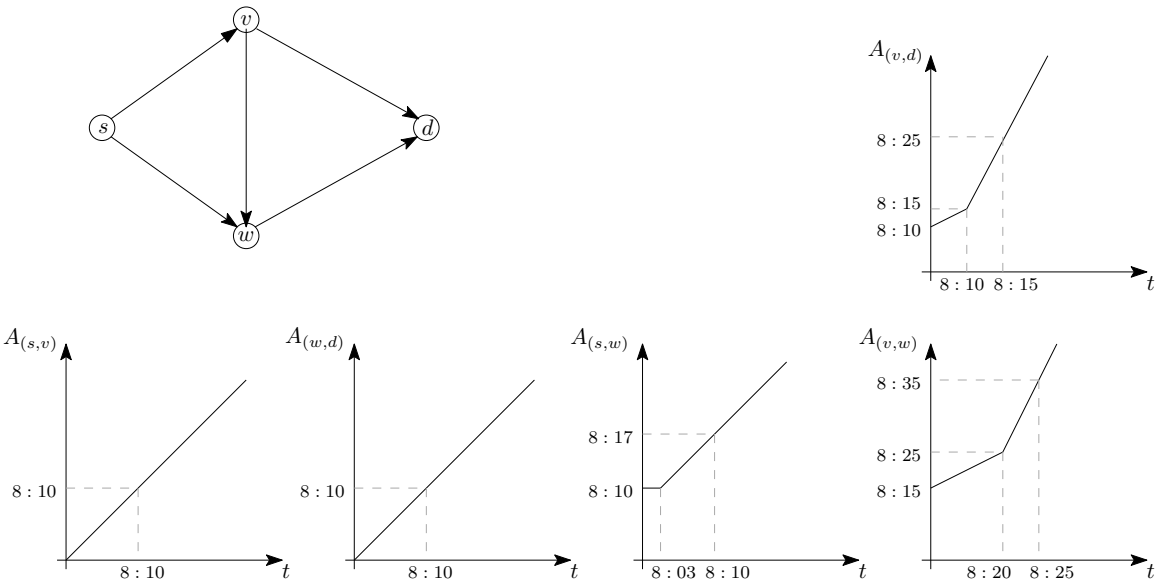


Figure 16: A time-dependent network and link arrival time functions

computations at fixed start times. In our second algorithm, we reduce the dependency on L, by using only $O(\gamma(\frac{1}{\epsilon}\log(\frac{T_{max}}{T_{min}}) + \log(\frac{L}{\gamma\epsilon T_{min}})))$ total shortest path computations.

4.1 Introduction

Problem Definition

In this chapter, we study the approximation of the $\mathcal{TDSP}_{\text{lin}}$ problem defined in Chapter 3. In the $\mathcal{TDSP}_{\text{lin}}$ problem, given a time-dependent network $G(E, V)$, each edge $e = (u, v) \in E$ is assigned a non-decreasing piecewise linear function $A_e(t)$ denoting the arrival time at v for a given start time t at u. The travel time function on each edge e is denoted by $T_e(t)$ is $A_e(t) - t$. Given a source node s and destination node d, the goal is to approximate the minimum travel time function, i.e., $T_{s,d}(t) = A_{s,d}(t) - t$. We say a function $T'_{s,d}(t)$ is an ϵ–approximation of $T_{s,d}(t)$ if $|T_{s,d}(t) - T'_{s,d}(t)| \leq \epsilon \cdot T_{s,d}(t)$ for all t.

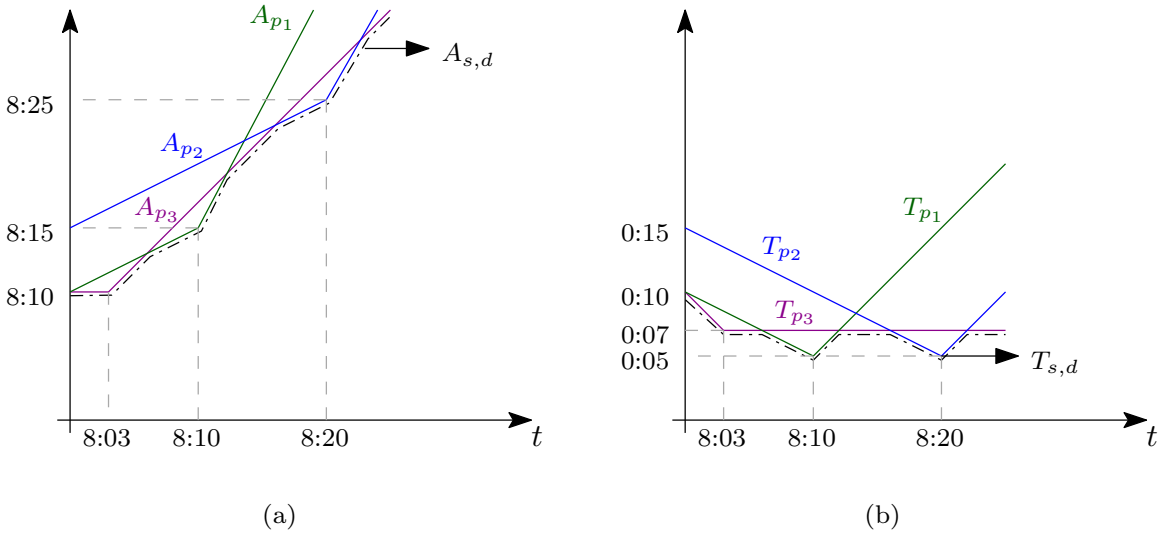


Figure 17: a) The earliest arrival time function for the network in Figure 16. b) The minimum travel time function for the network in Figure 16.

Figure 16 shows an instance of a time-dependent network and link arrival time functions. Let $p_1 = \langle s, v, d \rangle$, $p_2 = \langle s, v, w, d \rangle$, and $p_3 = \langle s, w, d \rangle$ be the three possible paths from s to d. Then, the arrival time functions on p_1, p_2, and p_3 are $A_{p_1} = A_{(v,d)} \circ A_{(s,v)}$, $A_{p_2} = A_{(w,d)} \circ A_{(v,w)} \circ A_{(s,v)}$, and $A_{p_3} = A_{(w,d)} \circ A_{(s,w)}$, respectively. Figure 17(a) shows the arrival time function on each path as well as the earliest arrival time function from s to d, $A_{s,d}$. For the same network, travel time functions on paths p_1, p_2, and p_3 are $T_{p_1} = A_{p_1} - t$, $T_{p_2} = A_{p_2} - t$, and $T_{p_3} = A_{p_3} - t$, respectively. Figure 17(b) shows the travel time function on each path as well as the minimum travel time function from s to d, $T_{s,d}$.

Related Work

Our goal is to find the minimum travel time to d for all possible start times t at s on a network, in which the travel time on each link is a piecewise linear FIFO function of the start time from s; in short, solve TDSP.

All algorithms overviewed in Section 2.2.2 provide exact solutions to TDSP. Dehne

et al.'s algorithm [47, 48] outperforms others in most cases, and provides a real output sensitive algorithms whose time complexity depends on the complexity of output function, i.e., F_d. The complexity of a minimum travel time function from s to d could be super-polynomial, and in most cases, link travel time functions are an approximation of reality. Efficient approximation algorithms with customizable quality of results are, therefore, both acceptable and often favorable for TDSP. In what follows, we provide an overview of existing approximation algorithms.

Dehne *et al.* [47] present an additive approximation algorithm for TDSP that approximates the earliest arrival time function from s to d. Their algorithm is based on running a reverse shortest path algorithm at arrival time values at d that are ϵ apart from each other. A reverse shortest path is obtained by running Dijkstra's algorithm on graph G^r obtained by reversing all links in G and inverting arrival time functions on links [45]. This returns, for each arrival time at d, the latest start time from s. The function obtained by connecting all such sample points is an approximation of the earliest arrival time function from s to d. Let A_{min} and A_{max} be the minimum and maximum value of the arrival time function from s to d, respectively. Their algorithm runs in $O(\frac{\Delta}{\epsilon}(|E| + |V| \log |V|))$ time, where $\Delta = A_{max} - A_{min}$.

Foschini *et al.* [60] subsequently propose an ϵ–approximation algorithm for TDSP that approximates the minimum travel time function from s to d. Note that, unlike earliest arrival time functions, it is not possible to run a reverse shortest path algorithm on minimum travel time functions directly to obtain, the start time at s, for a given travel time from s to d. This is because it is not possible to evaluate the travel time on each link without knowing the arrival time at d. They run a combination, therefore, of forward (standard) shortest path computations on travel time functions and reverse shortest path computations on arrival time functions to obtain the sample points. Let t_{min} and t_{max} be the minimum and maximum possible start time values of

the travel time function from s to d, respectively. Let T_{min} and T_{max} be the minimum and maximum travel time values of the function, respectively. Their algorithm requires $O(\frac{\gamma}{\epsilon}\log(\frac{T_{max}}{T_{min}})\log(\frac{L}{\gamma\epsilon T_{min}}))$ shortest path computations. where $L = t_{max} - t_{min}$ (see Section 4.2.1 for more details).

Contributions

We propose new algorithms that improve upon Foschini's algorithms by reducing the number of shortest path computations required [122]. Our first algorithm runs shortest path computations at fixed start times, chosen so that the relative ratio is fixed to $(1+\epsilon)$ for any consecutive pair of time instances. The total number of shortest path computations required by our algorithm is $O(\frac{\gamma}{\epsilon}(\log(\frac{T_{max}}{T_{min}})+\log(\frac{L}{\gamma T_{min}})))$. Next, we further reduce shortest path computations by increasing the horizontal distance between sample points as the slope function converges to zero. The total number of shortest path computations required for this approach is $O(\gamma(\frac{1}{\epsilon}\log(\frac{T_{max}}{T_{min}})+\log(\frac{L}{\gamma\epsilon T_{min}})))$.

Organization

In Section 4.2, we provide required details of previous algorithms and then present our solutions. In Section 4.3 we conclude the chapter.

4.2 Solutions

In this section, we sketch the previous approximation approaches by Dehne *et al.* [48] and Foschini *et al.* [60, 59] (Section 4.2.1). We then propose new ϵ–approximation algorithms that improve upon the running time of Foschini *et al.*'s algorithm (Section 4.2.2 and 4.2.3).

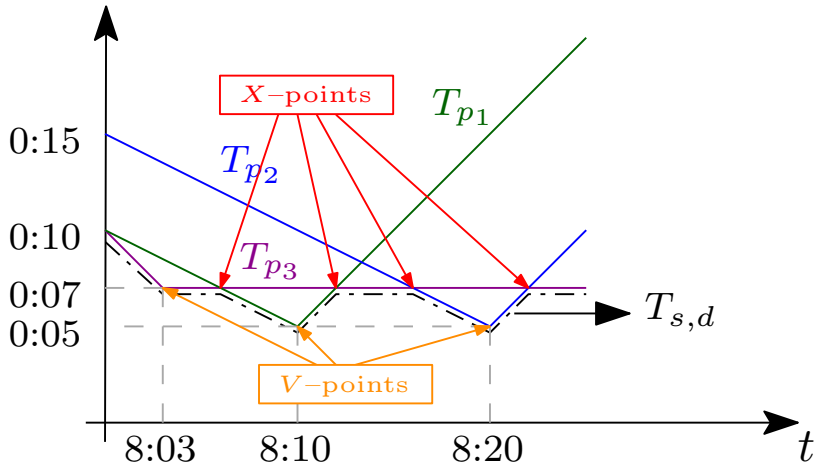


Figure 18: The V–points and the X–points of $T_{s,d}$ for the network shown in Figure 16.

4.2.1 Previous Algorithms

We showed that $T_{s,d}(t)$ is composed of two types of breakpoints: *V–points* and *X–points*. V–points originate from breakpoints in link functions and X–points are the points at which a shortest path from s to d switches to a new path (Figure 18). In Section 3.2.2 we also showed that between any consecutive pair of V–points, the sub-function is concave and has slope not less than -1. Let γ be the total number of breakpoints on all arrival time functions on links. The number of V–points in $T_{s,d}(t)$ is, at most, $O(\gamma)$. Thus, in order to approximate $T_{s,d}(t)$, it suffices to approximate $O(\gamma)$ concave sub-functions. Let t_{min} and t_{max} be the minimum and maximum possible start time values of $T_{s,d}(t)$, respectively. Let $v' = (t', T_{s,d}(t'))$ and $v'' = (t'', T_{s,d}(t''))$ be an arbitrary pair of consecutive V–points. We denote the sub-function of $T_{s,d}(t)$ between v' and v'' by $T_{s,d}[v', v'']$. Because of the concavity of $T_{s,d}[v', v'']$, the minimum value, τ_{min}, is $\min(T_{s,d}(t'), T_{s,d}(t''))$ and its maximum value, τ_{max}, is $2T_{s,d}((t'+t'')/2)$. Let T_{min} and T_{max} be the minimum τ_{min} and maximum τ_{max}, among all consecutive V–points, respectively. The general approach used by Foschini to approximate $T_{s,d}[v', v'']$ is to intersect $T_{s,d}(t)$ with the horizontal lines $y = (1+\epsilon)^k T_{min}$ for each

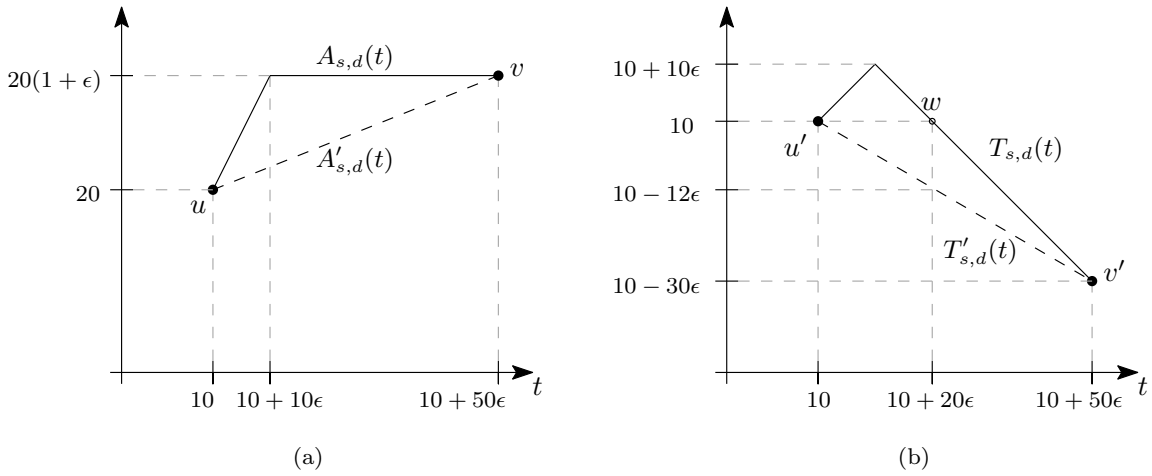


Figure 19: An example showing that an ϵ-approximation of $A_{s,d}(t)$ does not necessarily yield an ϵ-approximation for $A_{s,d}(t)$. a) Optimal and approximated arrival time functions. b) The corresponding travel time functions.

$k \geq 0$ such that $(1+\epsilon)^k T_{min} \leq T_{max}$. This generates $\approx \frac{2}{\epsilon}\log(T_{max}/T_{min})$ sample points. Connecting consecutive sample points with line-segments results in an ϵ-approximation of $T_{s,d}[v', v'']$.

The above approach requires $O(\frac{\gamma}{\epsilon}\log(\frac{T_{max}}{T_{min}}))$ reverse shortest path computations on $T_{s,d}(t)$. Reverse shortest path computation is only feasible, however, for $A_{s,d}(t)$, and not for $T_{s,d}(t)$. Also, note that an ϵ–approximation of $A_{s,d}(t)$ does not necessarily result in an ϵ–approximation for $T_{s,d}(t)$. Figure 19(a) shows $A_{s,d}(t)$ between points u and v (solid line) as well as its ϵ-approximation, $A'_{s,d}(t)$ (dashed line). Figure 19(b) shows the corresponding travel time functions, $T_{s,d}(t)$, and the approximated function,

$T'_{s,d}(t)$. For $T'_{s,d}(t)$ to be an ϵ-approximation of $T_{s,d}(t)$, we should have

$$\frac{|T'_{s,d} - T_{s,d}|}{T_{s,d}} \leq \epsilon,$$

Now at point w we must have

$$\frac{10 - (10 - 12\epsilon)}{10} \leq \epsilon \Leftrightarrow \frac{12\epsilon}{10} \leq \epsilon \Leftrightarrow 2\epsilon \leq 0 \Leftrightarrow \epsilon \leq 0.$$

but this contradicts that $0 < \epsilon < 1$. Foschini resolves this by running a combination of forward and reverse shortest path computations. They first partition the range of $T_{s,d}[v', v'']$ using horizontal lines $y = (1+\epsilon)^{i/2} T_{min}$ for $i \leq 2\log(T_{max}/T_{min})/\log(1+\epsilon)$. Starting from $t = t'$, on parts of $T_{s,d}[v', v'']$ with a slope of, at least, 1, they compute a reverse shortest path for $A_{s,d}(t) = t + T_{s,d}(t) \cdot \sqrt{1+\epsilon}$, and then set t to the resulting start time value. They show that this approach performs a constant number of reverse shortest path computations in each horizontal partition of the range of $T_{s,d}[v', v'']$. Therefore, the value of $T_{s,d}(t)$ at consecutive sample points differ by a factor of $1 + \epsilon$. Linearly interpolating the sample points obtained using the above approach provides an $\epsilon-$ approximation of $T_{s,d}[v', v'']$ for the parts with a slope of, at least, 1. In total, their approach performs $O(\frac{\gamma}{\epsilon}\log(\frac{T_{max}}{T_{min}}))$ reverse shortest path computation on $A_{s,d}(t)$.

They use a different approach to find sample points on parts of $T_{s,d}[v', v'']$ with a slope, at most, 1. They perform bisection with forward shortest path computations on $T_{s,d}(t)$ until the approximation error between consecutive sample points is at most $1+\epsilon$. They show that their approach requires $O(\frac{\gamma}{\epsilon}\log(\frac{T_{max}}{T_{min}})\log(\frac{L}{\gamma\epsilon T_{min}}))$ forward shortest path computations, where $L = t_{max} - t_{min}$. Refer to [60] for more details.

4.2.2 First Improved Algorithm

In this section, we present an improved algorithm, referred to as Aprx–A, which significantly reduces the number of shortest path probes. For two consecutive V–points, $v' = (t', T_{s,d}(t'))$ and $v'' = (t'', T_{s,d}(t''))$, we approximate $T_{s,d}[v', \bar{v}]$ as in [60] where $\bar{v} = (\bar{t}, T_{s,d}(\bar{t}))$ is the first point on $T_{s,d}[v', v'']$, in increasing order of times, whose slope is less than or equal to 1. However, on $T_{s,d}[\bar{v}, v'']$, we use a different approach to obtain sample points. Our new strategy is illustrated below.

Let $\bar{l}$ be the line, with slope 1, that goes through $\bar{v}$, and let l'' be the line, with slope -1, that goes through v'' (Figure 20). Let S be the set of start times $(\bar{t}_i)$ at s that correspond to the intersection of $\bar{l}$ with the horizontal line $y = (1+\epsilon)^i T_{s,d}(\bar{t})$ for $i \in [0, k]$ where k is the smallest value of i for which $y \geq \tau_{max}$. Starting from $\bar{v}$, we iterate along the start times in S and compute $T_{s,d}(\bar{t}_i)$ in increasing order of i. The operation terminates at the smallest x for which $T_{s,d}(\bar{t}_x) \geq T_{s,d}(\bar{t}_{x+1})$. Let S' be the set of start times (t''_j) at s that correspond to the intersection of l'' with the horizontal line $y = (1+\epsilon)^j T_{s,d}(t'')$ for $j \in [0, z]$ where z is the smallest value of j for which $y \geq \tau_{max}$. Starting from v'', we iterate along the start times in S' and compute $T_{s,d}(t''_j)$ in increasing order of j. The operation terminates at the smallest y for which $T_{s,d}(t''_y) \geq T_{s,d}(t''_{y+1})$. We then connect the set of sample points obtained using the above approach, organizing them by order of the start times (see Figure 20). The following theorem proves that this approach results in an ϵ–approximation of $T_{s,d}(t)$.

Theorem 4. *Algorithm Aprx–A is an ϵ-approximation of $T_{s,d}(t)$.*

Proof. Let $\bar{t}_{i-1}$ and $\bar{t}_i$ be the start times corresponding to two consecutive sample points of Algorithm Aprx–A. Let $\bar{v}_{i-1} = (\bar{t}_{i-1}, T_{s,d}(\bar{t}_{i-1}))$ be the $(i-1)^{\text{st}}$ sample point and $\bar{l}_{i-1}$ be the line that goes through $\bar{v}$ and $\bar{v}_{i-1}$. Also, let $v_c = (\bar{t}_{i-1}, T_{s,d}(\bar{t}_{i-1}))$, and

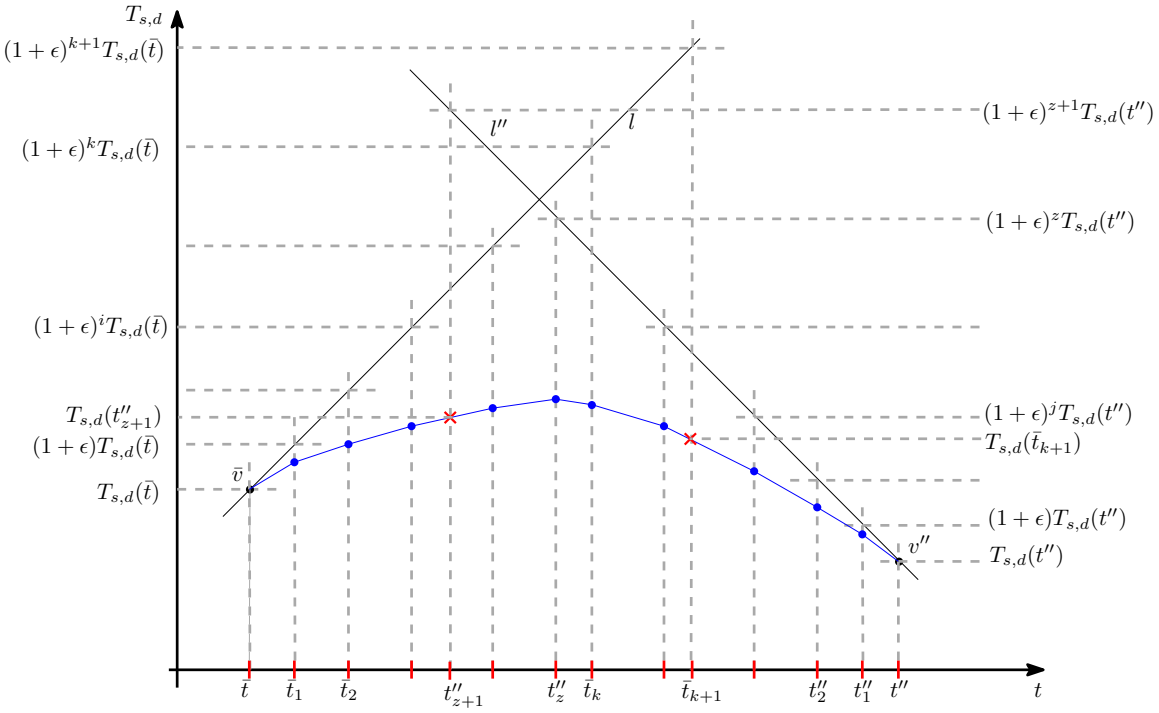


Figure 20: Sample point placement of Algorithm Aprx–A

v_d be the intersection point of the lines $\bar{l}_{i-1}$ and $x = \bar{t}_i$. We define vertical distances A, B, C, D, and Y, as shown in Figure 21. We show that between $\bar{t}_{i-1}$ and $\bar{t}_i$, the approximation ratio is, at most, $1+\epsilon$. We have:

$$m := \frac{B}{D} = \frac{A}{C} = \frac{Slope(\bar{l})}{Slope(\bar{l}_{i-1})} \tag{1}$$

$$B \geq D,\ A \geq C \Rightarrow m \geq 1 \tag{2}$$

$$D \geq C \tag{3}$$

$$\frac{B+Y}{A+Y} = 1+\epsilon \tag{4}$$

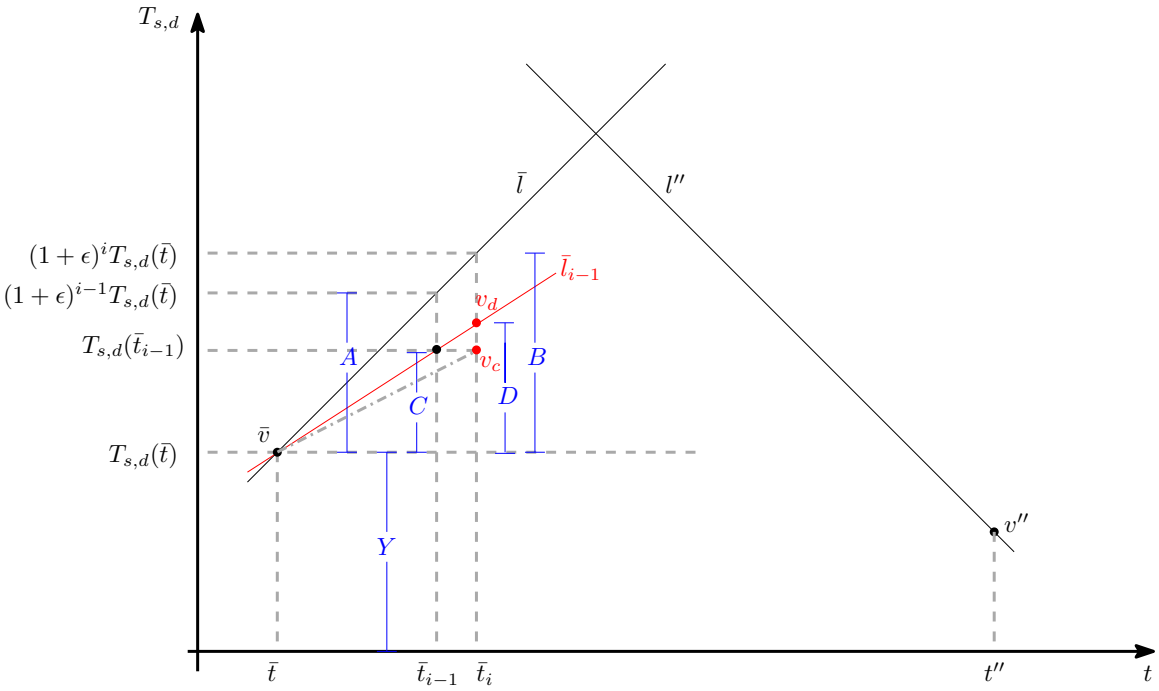


Figure 21: A sample step of Algorithm Aprx–A

Then:

$$\begin{aligned}
\frac{B+Y}{A+Y} - \frac{D+Y}{C+Y} &= \frac{BC+BY+CY+Y^2-DA-DY-AY-Y^2}{(A+Y)(C+Y)} \\
&\overset{(1)}{=} \frac{(B+C-D-A)Y}{(A+Y)(C+Y)} \overset{(1)}{=} \frac{(mD+C-D-mC)Y}{(A+Y)(C+Y)} \\
&= \frac{((m-1)D-(m-1)C)Y}{(A+Y)(C+Y)} \overset{(2)(3)}{\geq} 0
\end{aligned}$$

$$\Rightarrow \quad \frac{B+Y}{A+Y} - \frac{D+Y}{C+Y} \geq 0 \overset{(4)}{\Rightarrow} \frac{D+Y}{C+Y} \leq 1+\epsilon \tag{5}$$

Since $T_{s,d}(t)$ is a concave function, it lies inside the triangle $\triangle(\bar{v}v_c v_d)$. Per (5), the error ratio between $\bar{t}_{i-1}$ and $\bar{t}_i$ is, at most, $1+\epsilon$. Due to the symmetry of $\bar{l}$ and l'', a similar argument can be made for sample points along l''. Therefore, Algorithm Aprx–A provides an ϵ–approximation of $T_{s,d}(t)$ which proves the theorem. □

Theorem 5. *Approximation Algorithm Aprx–A requires* $O(\frac{\gamma}{\epsilon}(\log(\frac{T_{max}}{T_{min}}) + \log(\frac{L}{\gamma T_{min}}))$

shortest path computations.

Proof. Each V–point on $T_{s,d}(t)$ corresponds to a breakpoint on a link travel time function. The number of V–points is, therefore, $O(\gamma)$. On parts of $T_{s,d}(t)$ that have slope of at least 1, we run a reverse Dijkstra. In [60], it has been shown that there are $O(\frac{1}{\epsilon}\log(\frac{T_{max}}{T_{min}})))$ such points for any two consecutive V–points. Since $\bar{l}$ and l'' have slopes of 1 and -1, respectively, the vertical span of $T_{s,d}(t)$ between $\bar{v}$ and v'' is at most $2\mathcal{L}$, where $\mathcal{L}$ is the horizontal distance between $\bar{v}$ and v''. Our improved method would, therefore, place $O(\frac{1}{\epsilon}\log(\frac{\mathcal{L}}{\gamma T_{min}}))$ sample points along each line for consecutive V–points, v' and v''. Let $\mathcal{L}_i$ be the horizontal distance between the i^{th} pair of V–points. Then, the total horizontal span of $T_{s,d}(t)$, L, is $\sum_{i=1}^{\gamma}\mathcal{L}_i$. The above results can be extended to L using the log-sum inequality, $\sum_{i=1}^{\gamma}\log\frac{\mathcal{L}_i}{T_{min}} \leq \gamma\log\frac{L}{\gamma T_{min}}$. As a result, the total number of shortest path computations by Algorithm Aprx–A is $O(\frac{\gamma}{\epsilon}(\log(\frac{T_{max}}{T_{min}}) + \log(\frac{L}{\gamma T_{min}}))$. □

4.2.3 Second Improved Algorithm

In this section, we present a second algorithm, Aprx–B, for approximating $T_{s,d}(t)$. Similar to Aprx–A, Aprx–B approximates the travel time function between every consecutive pair of V–points. Now, we replace in algorithm Aprx–A the sample point placement on $T_{s,d}(t)$ when the slope is less than or equal to 1. Let $\bar{v} = (\bar{t}, T_{s,d}(\bar{t}))$ be the first point on $T_{s,d}(t)$, in increasing order of time, whose slope is less than or equal to 1. We present a new approach for finding sample points of an ϵ-approximation for $T_{s,d}[\bar{v}, v'']$. Here, our general approach is to partition the range of $T_{s,d}[\bar{v}, v'']$ and find at least one sample point in each partition. We set the i^{th} horizontal partitioning line to $y = (1+\epsilon)^{i/2}T_{s,d}(T_{min})$, so that the maximum error between any two consecutive sample points is at most $(1+\epsilon)T_{s,d}(T_{min})$.

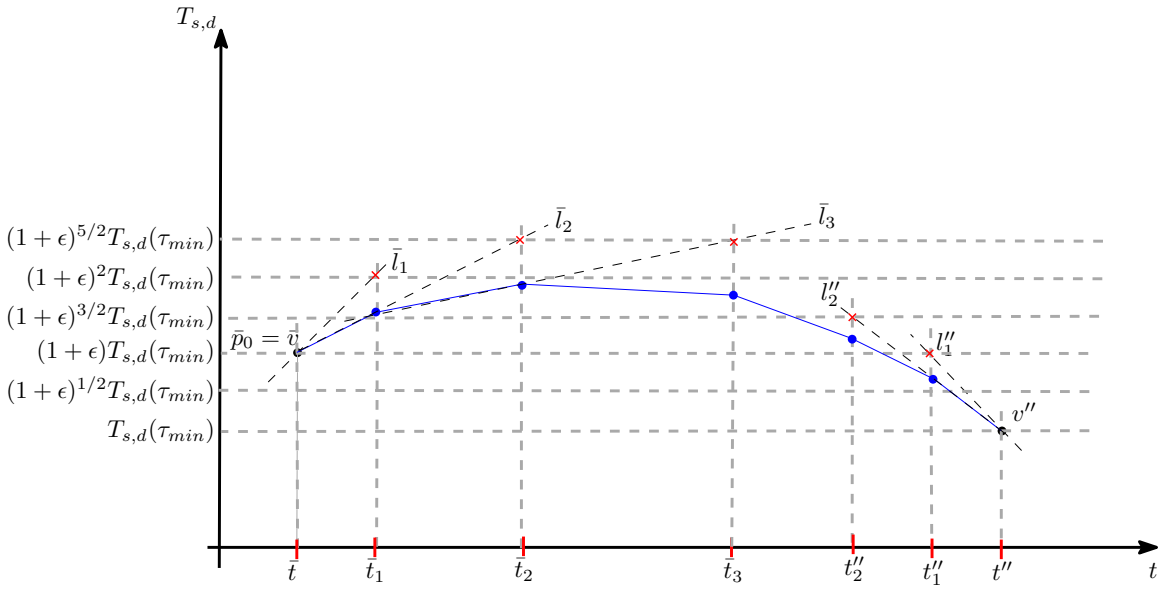


Figure 22: Sample point placement of Algorithm Aprx–B

We split $T_{s,d}[\bar{v}, v'']$ into two sections: the parts of the function with slopes between 0 and 1, and the remaining parts with slopes between -1 and 0. What follows is our approach to finding sample points on the first section of $T_{s,d}[\bar{v}, v'']$. Let $\bar{p}_0 = \bar{v}$ be the initial sample point and $\bar{l}_1$ be the line that goes through $\bar{p}_0$ with slope 1. Let $\bar{p}_0$ be in the r^{th} partition of the range. Let $\bar{t}_1$ be the time instance that corresponds to the intersection of $\bar{l}_1$ with the $(r+1)^{\text{st}}$ partitioning line, i.e., $y = (1+\epsilon)^{(r+1)/2}T_{s,d}(T_{min})$. We then obtain sample point $\bar{p}_1 = (\bar{t}_1, T_{s,d}(\bar{t}_1))$ by running a forward Dijkstra at time $\bar{t}_1$. Now, let $\bar{p}_{i-1}$ be in the k^{th} partition of the range and $\bar{l}_i$ be the line segment that goes through $\bar{p}_{i-2}$ and $\bar{p}_{i-1}$. Let $\bar{t}_i$ be the time instance that corresponds to the intersection of $\bar{l}_i$ with the $k+1^{\text{st}}$ partitioning line, i.e., $y = (1+\epsilon)^{(k+1)/2}T_{s,d}(T_{min})$. We then obtain sample point $\bar{p}_i = (\bar{t}_i, T_{s,d}(\bar{t}_i))$ by running a forward Dijkstra at time $\bar{t}_i$. The process stops at step j when $T_{s,d}(\bar{t}_{j-1}) > T_{s,d}(\bar{t}_j)$. To find sample points on parts of $T_{s,d}[\bar{v}, v'']$ with slope between -1 and 0, we run a process similar to that above, but with $p''_0 = v''$. Let l''_1 be the line that goes through p''_0 with slope -1. Figure 22 depicts steps of Aprx–B.

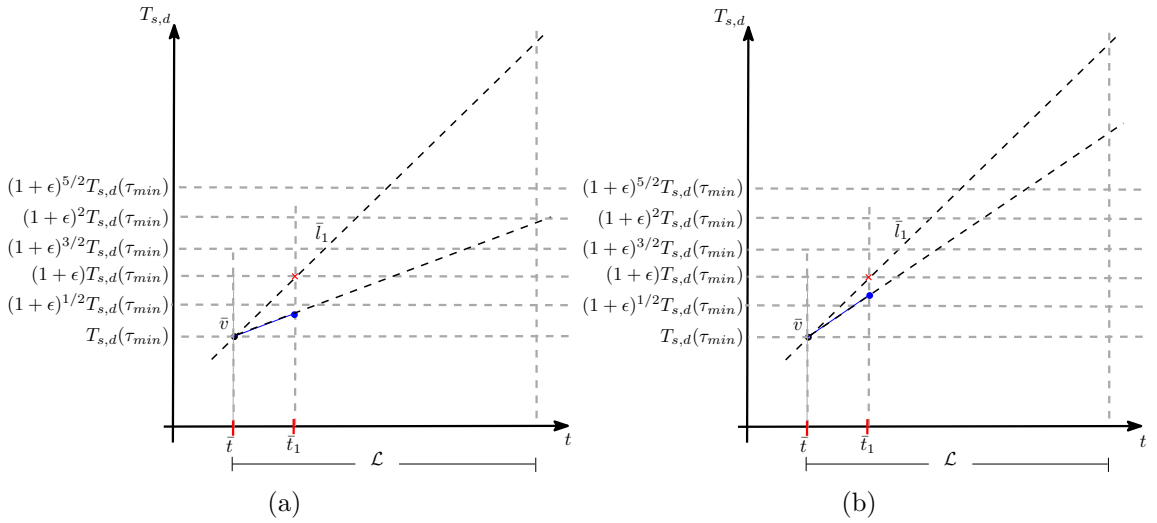


Figure 23: a) Consecutive sample points in the same partition of the range. b) Consecutive sample points in different partitions of the range.

Theorem 6. *Algorithm Aprx–B, is an ϵ-approximation of $T_{s,d}(t)$.*

Proof. Let p_i be the sample point obtained in the i^{th} iteration of Aprx–B located in the k^{th} partition of the range. Let l_i be the line that goes through p_{i-1} and p_i. In the $(i+1)^{\text{st}}$ iteration, the algorithm would place a sample point either on the k^{th} or on the $(k+1)^{\text{st}}$ partition of the range (Figures 23(a) and 23(b)). Note that we consider points that lie on line $y = (1+\epsilon)^{k/2}T_{s,d}(T_{min})$ to be inside the $k+1^{\text{st}}$ partition. Also, in the degenerate case that p_{i+1} is on the intersection of l_i and $y = (1+\epsilon)^{(k+1)/2}T_{s,d}(T_{min})$, the maximum error between p_i and p_{i+1} is equal to zero because the line segment connecting p_i and p_{i+1} would be on $T_{s,d}(t)$, as a result of the function's concavity. Therefore, Algorithm Aprx–B would place at least one sample point in each partition of the range, which proves the theorem. □

Theorem 7. *Approximation Algorithm Aprx–B requires* $O(\gamma(\frac{1}{\epsilon}\log(\frac{T_{max}}{T_{min}}) + \log(\frac{L}{\gamma\epsilon T_{min}}))$ *shortest path computations.*

Proof. As above, Algorithm Aprx–B would place at least a sample point in each horizontal partition of the function range between any two consecutive $V-$points

of $T_{s,d}(t)$, namely v' and v''. The total number of partitions of the function range is $O(\frac{\gamma}{\epsilon}\log(\frac{T_{max}}{T_{min}}))$. The algorithm may, however, place extra sample points in each partition. Below, we show that, in total, the algorithm may place $O(\gamma\log(\frac{L}{\gamma\epsilon T_{min}}))$ extra sample points in existing partitions. Between v' and v'', the vertical length of the i^{th} partition is

$$\begin{aligned} & (1+\epsilon)^{i/2}T_{s,d}(\tau_{min}) - (1+\epsilon)^{(i-1)/2}T_{s,d}(\tau_{min}) \\ = \;& ((1+\epsilon)^{1/2}-1)(1+\epsilon)^{(i-1)/2}T_{s,d}(\tau_{min}), \end{aligned}$$

where τ_{min} is the minimum travel time value between v' and v''. This increases for higher values of i. Therefore, at the i^{th} step of the algorithm, if p_i is in the same partition as p_{i-1}, then $slope(l_i) < \frac{1}{2}slope(l_{i-1})$. The algorithm terminates if $slope(l_i) < \frac{\epsilon T_{min}}{\mathcal{L}}$ since in that case, the maximum error value would be less than ϵT_{min}. So, the total number of extra sample points can be determined as follows:

$$\frac{1}{2^k} < \frac{\epsilon T_{min}}{\mathcal{L}} \quad \Rightarrow \quad \log 2^k > \log\frac{\mathcal{L}}{\epsilon T_{min}} \;\Rightarrow\; k > \log\frac{\mathcal{L}}{\epsilon T_{min}}$$

Let $\mathcal{L}_i$ be the horizontal distance between the i^{th} pair of consecutive V–points and $L = \sum_{i=1}^{\gamma}\mathcal{L}_i$. The above result can be extended to L using the log-sum inequality, $\sum_{i=1}^{\gamma}\log\frac{\mathcal{L}}{\epsilon T_{min}} \leq \gamma\log\frac{L}{\gamma\epsilon T_{min}}$. □

4.3 Conclusions

We studied the approximation of the time-dependent shortest path problem (TDSP), where travel times on links are piecewise linear FIFO functions. We proposed two ϵ-approximation algorithms (referred to as Aprx–A and Aprx–B, respectively) that improve upon the existing approximation algorithm for TDSP by Foschini *et al.* [60].

The algorithms we present use simple data structures, are easy to implement, and employ, as a key building block, the modification of Dijkstra's algorithm studied in [45].

In [48], it has been shown that within the same time complexity, for a given start time at s, not only the minimum travel time to d can be computed, but it is also possible to obtain the slope of the function at that time. This can be used in Algorithm Aprx–B as a heuristic improvement to reduce the number of computations by using the function tangent rather than lines connecting consecutive sample points. It does not, however, change the worst-case time bound.

Chapter 5

Protecting Privacy From Inference Attacks in Location Based Services

In this chapter, we study potential inference attacks targeting Location Based Service (LBS) users, introduce a new model for privacy protection, provide heuristic defense techniques to protect users' privacy from such attacks, and present the results of experiments performed to evaluate the heuristics.

By gaining access to supplemental information such as query times, speed limits/travel times of the underlying road-network, or the residential/commercial address directory, potential attackers could infer sensitive information about the user querying an LBS, such as location, identity, or lifestyle. Our objective is to prevent attackers from connecting external information to user queries. To address this objective under user-controlled privacy settings, we introduce the notion of (i, j)-privacy which generalizes previous privacy models. We have designed several heuristics implementing (i, j)-privacy. These heuristics still provide exact responses for user queries. We evaluate our algorithms experimentally on different road-networks by varying a number of input parameters and present the results here. The outcomes of our experiments show that, for realistic user settings, our algorithms rapidly provide high

quality results.

5.1 Introduction

With advances in on-line LBS technology (e.g., the use of real-time traffic data to provide more accurate navigation information, the ability to search a nearby area for the location of a specific facility, and "street view's" capacity to provide a more realistic user experience) and the availability of the Internet on various handheld devices (e.g., smart phones and tablet-computers), LBS applications are becoming convenient tools for mobile users to employ in their daily lives. In fact, users consider such tools to be not only convenient, but also essential because they save both time and money. Despite the fact that location information enables the use of valuable tools for mobile device users, such information can be misused, and become a crucial source for malicious activity.

Adversaries (e.g., spies, unsolicited marketers and spammers) can use LBS as a medium through which to access private user information and use it without the user's consent. LBS users who are not protected against location-based privacy attacks may run the risk of exposing sensitive personal information (about e.g., lifestyle, health condition, and sexual orientation) to malicious parties. Preventing LBS users from privacy breaches has, therefore, recently become an important subject for research activity (see [44, 135, 140]).

A malicious LBS provider who receives a user's location information could use it, together with a publicly available residential/commercial address database and/or a phone directory to infer user identities [77, 99]. The higher the frequency of a user's LBS queries, the more private information an adversary could infer about the user. Service users who submit multiple location-based service queries to an untrusted

LBS provider at sensitive locations (e.g., hospitals, military zones, and night-clubs) are in danger of unknowingly exposing their identities, as well as details about their lifestyles. As an example, consider a user who submits queries to an LBS to get directions from his/her home to a hospital, from the hospital to a pharmacy, and finally from the pharmacy back home, on a weekday. In this case, not only can the LBS use publicly available residential information to identify the user, but also it can infer that the user may have an illness. A recently published article by Wicker [144] provides examples where anonymized location data, when correlated with publicly available databases, could lead to potential privacy threats. Wicker also lists types of information that could be inferred from such correlations.

In this chapter, we focus on the scenario in which users repeatedly submit their location information to LBS providers in order to obtain typical services, such as ascertaining the shortest routes to a destination or finding their nearest neighbors. Our main objective is to protect users' locations from malicious LBS providers who attempt to use a user's location without their consent or knowledge. Such LBSs usually have access to external information about road-networks such as speed limits/travel time on road-network links, as well as access to additional information such as the residential/commercial address directories, and intervals times between user queries. The latter, in particular may enable critical inference.

5.1.1 Features/issues of existing location privacy protection methods

Underlying space model.

Currently, available privacy protection methods against speed-limit-based attacks assume that users are moving in open space and build up protection models based on

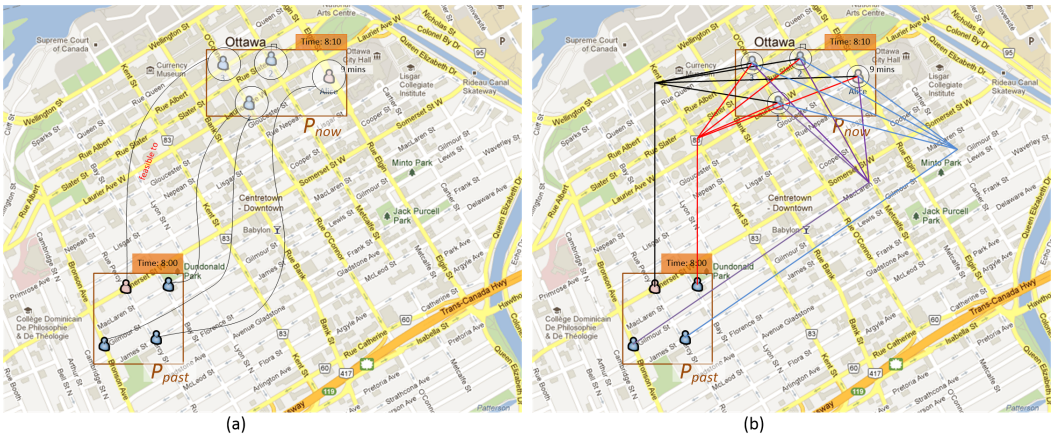


Figure 24: a) Weak privacy: one region is covered per location in the past. b) Strong privacy: all locations are covered by each past location in the past.

travel time in Euclidean space [34, 69, 126, 148]. However, users of LBS typically move along road-networks where such methods are not applicable.

Privacy levels.

Currently, in location privacy protection methods, the level of privacy is either fixed [34, 126, 148] or only customizable to *weak* and *strong* privacy settings [69]. In the weak privacy model, it suffices that there exists a user location in the destination region that is feasible for each location in the source region. In *strong* privacy model, all locations in the destination region must be feasible from each point in the source region. Figure 24 provides an example of the feasibility coverage for *weak* and *strong* privacy levels. These privacy levels do not provide users with a flexible customization of trade-offs between privacy and performance. In these models, feasibility is determined in a way that only future locations are protected. However, this may reveal past user locations (see the discussion, which illustrates this, at the end of Section 5.3).

Region vs. exact location queries.

Existing privacy algorithms for protecting frequent LBS users against adversary attacks that use speed-limit information, are based on cloaking strategies [34, 69, 126, 148]. In cloaking-based privacy protection strategies, a region (e.g., rectangle or circle) containing the user's exact location is reported to the LBS provider, instead of a user's exact location. However, most existing LBS providers (e.g., Google Maps, MapQuest, and Bing Maps) do not support regions-based queries. Shortest path queries between a current location and an intended destination, for example, would have to be phrased as a shortest path query between two regions. LBS typically cannot handle such region queries and, even if they could, they would not be able to provide an accurate answer to the intended query. Similarly, for other LBS queries such as nearest neighbor queries, the query has to be phrased as neighbors nearest to the region containing user's location.

Trusted third parties.

A trusted third-party is a necessary component of most location-based privacy protection methods. In early works on LBS privacy [34, 65, 76, 113, 138], the trusted third party was an integral part of the privacy protection system. It serves as an anonymizer of users' identities among other users when exclusively real user locations are reported to the LBS provider. Sophisticated region-based query models were developed in order to address privacy issues that emerged as a result of advances in location-based technology and an increased frequency of user queries. To provide users with the exact result for the service request, a variety of additional information might need to be transmitted and stored, and subsequent complex operations on these data may be required. For example, cloaking-based models may need a third-party to store enough map information to be able to refine the query result, and to

provide users with an accurate result. The user device may not be powerful enough to carry out these computations and/or store these data. Therefore, a third party is sometimes called for.

5.1.2 Contributions

Our objective is to protect user privacy against untrusted LBS providers who may use user/road-network speed limitations to infer additional information about users' whereabouts. We would like to accomplish this while users have the flexibility to customize the accuracy and privacy level in a number of ways, according to their needs. Our contributions can be summarized as follows [120]:

- We introduce the (i, j)-privacy model, which generalizes the existing models of weak and strong privacy in the following ways:

    1. The underlying space in which users travel is a network of roads rather than Euclidean space. This is more realistic. Our new model can accommodate a preference for Euclidean space.
    2. Our privacy protection approach provides more flexibility in setting levels of privacy and delivers more protection by verifying feasibility not only for future user locations, but also for past user locations.
    3. In contrast with existing privacy models, our model is compatible with all common LBS providers such as Google Maps, MapQuest, and Bing Maps.

- We conjecture that solving the privacy problem optimally is NP-complete.
- We propose heuristic solutions for computing the desired privacy.
- We test our solutions experimentally on numerous road network data and provide a detailed analysis of the results.

5.1.3 Organization of the Chapter

The remainder of the chapter is organized as follows. In Section 5.2, we describe the system's architecture. Section 5.3 formally defines the problem. In Section 5.4, we discuss the model and solutions. We propose several privacy protection algorithms in Section 5.6, and provide implementation outcomes in Section 6.5. We conclude the chapter and discuss future work in Section 5.8.

5.2 System Architecture

Our system is composed of three main components: *user*, *privacy protection service*, and *LBS provider* (Figure 25). *Users* are equipped with a hand-held device that has GPS functionality and an end-user map application. Each user sets some profile data to be used by the privacy protection service to apply a user's requested privacy level, depending on the situation. No map information is stored on user devices and all required road-network data would be provided by the LBS provider at query time.

The *privacy protection service* is responsible for receiving a user query, submitting the potentially inaccurate query to the LBS provider, receiving and processing the result returned by the LBS provider, and delivering the exact result to the user. This must be carried out while satisfying the privacy stipulation of the user profile. Depending on whether the privacy component is integrated into the user's device or offered as a separate service, we introduce two privacy models namely *2-tier* and *3-tier* (see Figure 25). The privacy protection service that we present in this chapter is operational in both 2-tier and 3-tier models. If real user locations are used for location anonymity, a 3-tier model is inevitable since a separate trusted third-party is needed to store user locations and remove their identity at query time, so as to hide the querying user among others. However, if all locations reported along with the user's

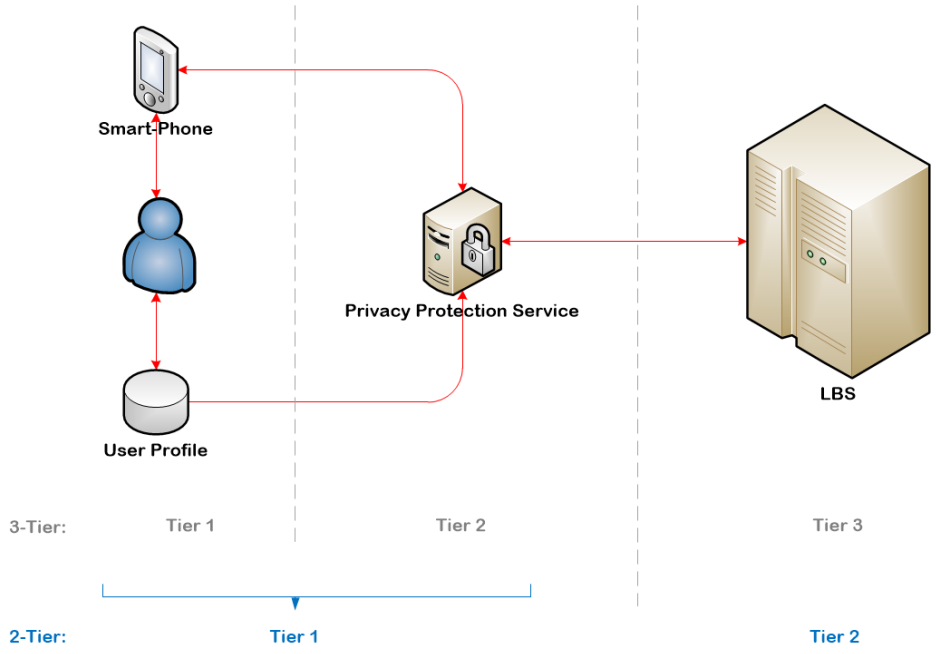


Figure 25: 2-tier and 3-tier design models

real location are fake, then using a third-party is not necessary, and a 2-tier model is sufficient. In this case, the privacy protection service component is integrated with the user's hand-held device.

The *LBS provider* is an on-line map application that can answer location-based queries. Given a source (usually, a user's current location) and a destination point on the map, it can compute a shortest path between the two points. It also provides users with the road-network data within a predefined region, thus avoiding the need to store that information on user devices. The LBS provider requires users to provide exact locations in their queries, e.g., coordinates, postal addresses, or intersections. The LBS provider is assumed to be insecure because it can correlate the location information in the query with publicly available external data and gain access to a user's private information (e.g., identity, lifestyle, or medical conditions). We also suppose that the LBS provider (also referred to as *adversary*) is aware of the privacy defense techniques and algorithms adapted by the protection service. Therefore,

fake/real locations are selected from a predefined set of subdivisions (regions) of the whole area of interest, in order to protect the user's location by hiding it among all fake/real locations [19]. To illustrate this, consider a location privacy system that selects fake/real locations from within a disk of a specific radius of the querying user. An adversary who is aware of this defense technique could easily infer that the querying user is approximately around the centroid of the reported locations. On the other hand, working on a subset of points within a region rather than all points within the area of interest would improve the query computation performance especially when the frequency of requests increases. For example, it is easier to manage the consistency of a fake user's movement if the locations reported are close to the querying user's location, than if the locations generated correspond to a valid movement unrelated to the movement of the querying user.

These basic region partitioning techniques are partitioning based on *quad-tree* structures [56], *k-d tree* structures [16], and *Hilbert space filling curves* [86]. In quad-tree partitioning, the whole region of interest is divided into four equal size regions recursively until some predefined conditions (e.g., minimum allowable area) hold. A sample region partitioning and its corresponding quad tree is shown in Figure 26. A quad-tree structure is then used to represent the hierarchy of the multi-layer grid of the whole region and to efficiently access each cell [114]. A 2-d region partitioning based on a k-d tree is a data-driven partitioning that splits points into two roughly equal partitions along the x and y axes. A sample region partitioning based on a k-d tree structure has been depicted in Figure 27. In partitioning based on Hilbert curves, the map of the region of interest is divided into $n \times n$ grids that can be referenced by their sequence number in the Hilbert space filling curve [71]. Note that, there is a high probability that, two geometrically close grid cells within the region also have close sequence numbers. Figure 28(a) depicts the Hilbert curve for 8×8 region

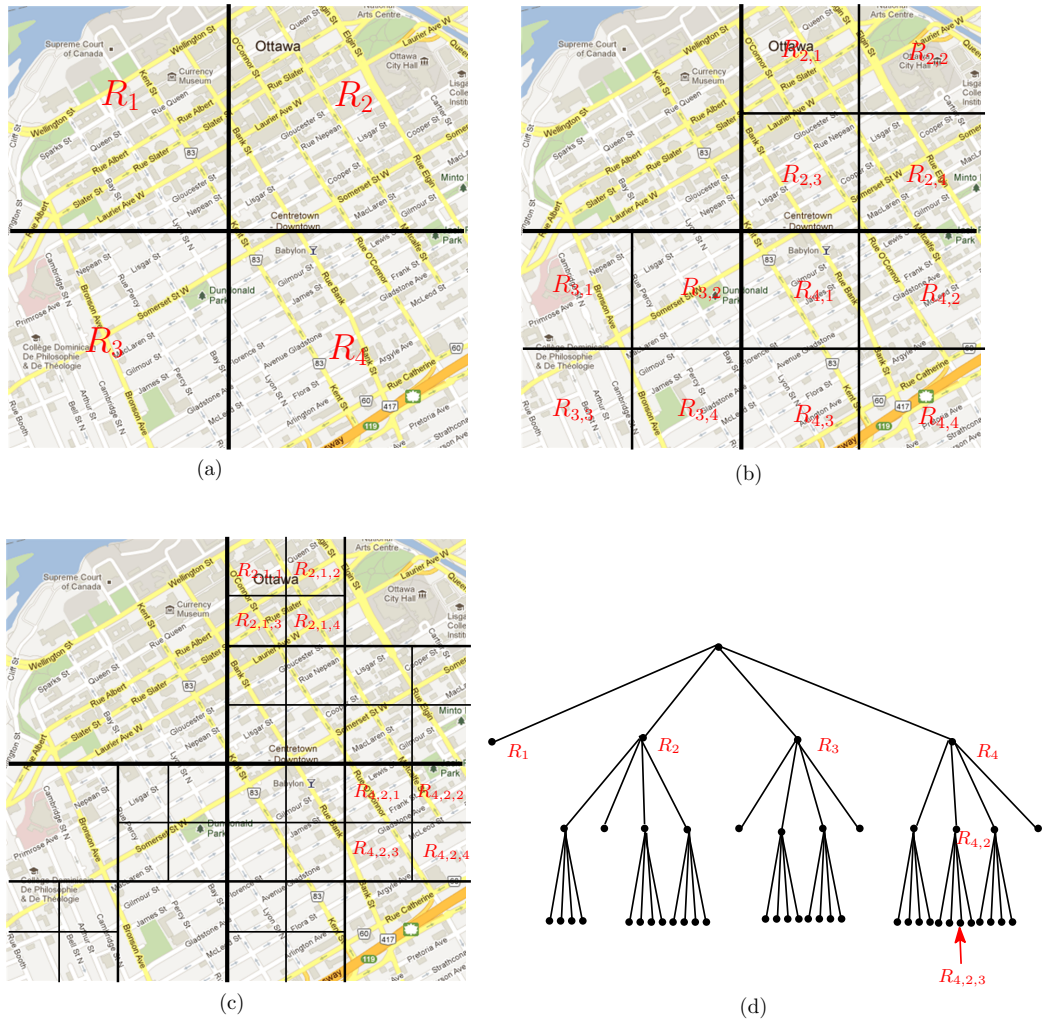


Figure 26: (a,b,c) Level 1, 2, and 3 of the region partitioning (d) The quad-tree of the region partitioning

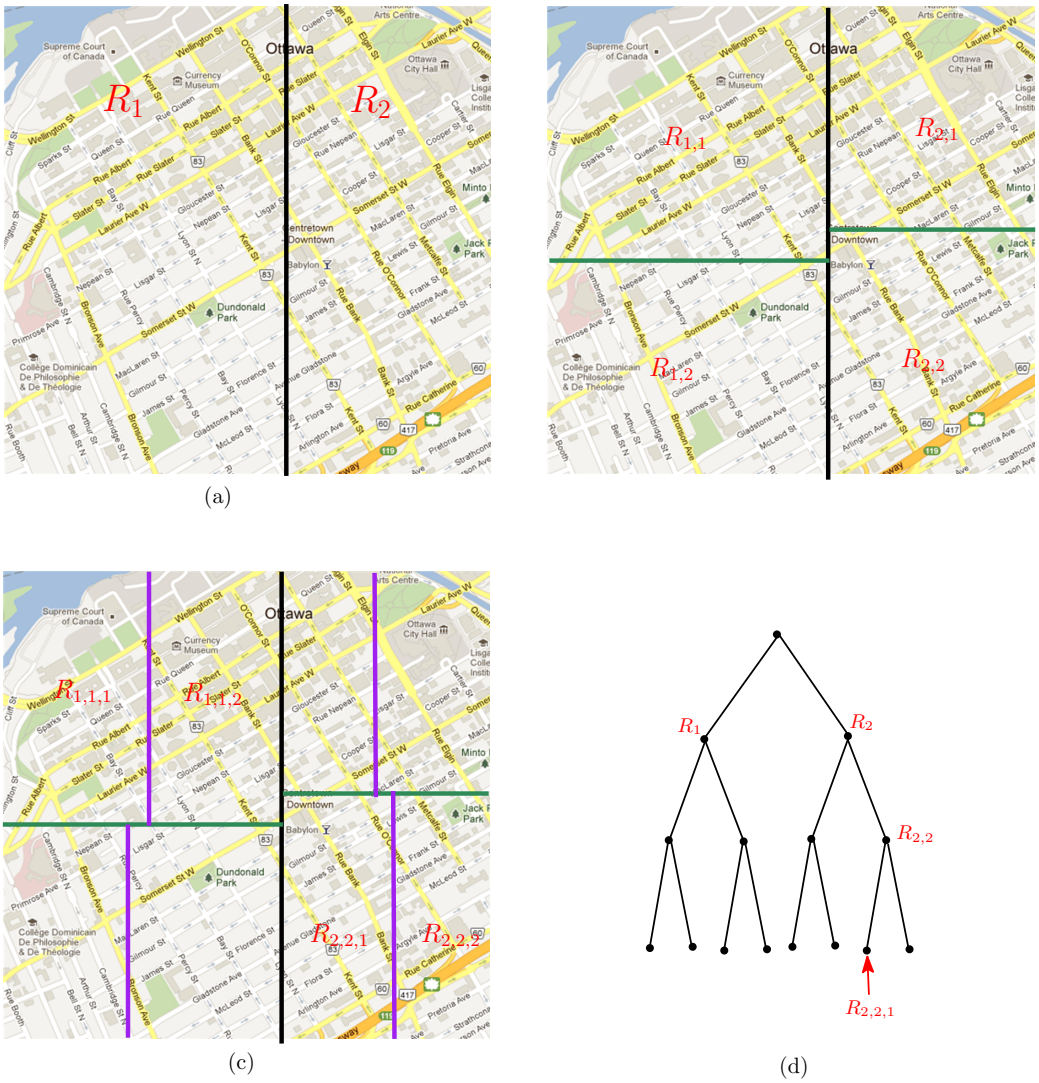
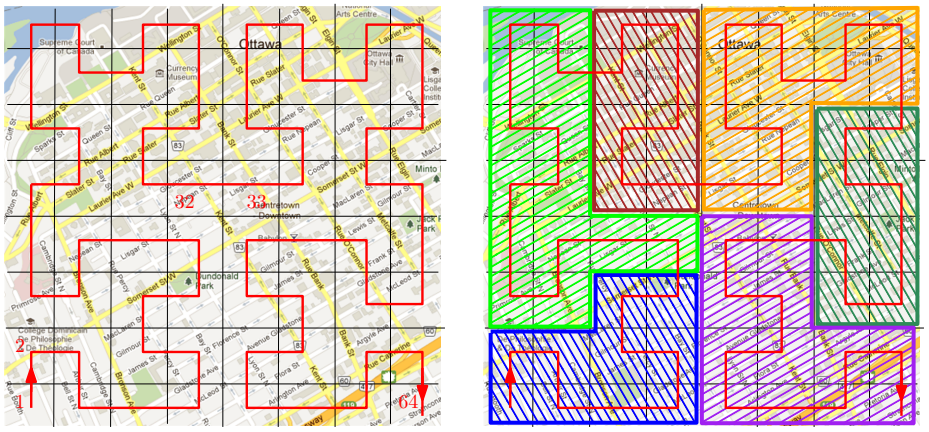


Figure 27: a,b,c) Level 1, 2, and 3 of the region partitioning d) The k-d tree of the region partitioning



(a) (b)

Figure 28: a) 8×8 Hilbert partitioning. b) A sample road-network partitioning using Hilbert approach.

partitioning and the corresponding sequence numbers. Figure 28(b) shows a sample partitioning of the road-network using Hilbert curves.

In existing privacy protection methods, the size of sub-divisions is considered to correspond directly to the user's location privacy level. Most privacy protection techniques therefore require users to provide a minimum allowable size defined by means of area, width/length, radius, etc. However, in road-networks, user movements are mostly restricted to road-links. Thus, increasing the size of sub-divisions may not necessarily result in an improved privacy. As a solution, we propose a new scheme that allows users to define the minimum *road-network length* enclosed inside the region as an additional partitioning criterion. For example, in Figure 26, a recursive partitioning process on each quadrant would stop if the total length of road-links inside one or more partitions of the next step falls bellow a specific threshold.

The region partitioning operations could be done by the user, a trusted third-party, or an LBS provider, depending on the system model and the availability of the service required (see further discussion below). To minimize user-side computations, our main approach is to have the LBS provider compute and store the partitioning information.

At query time, the partitioned area is traversed until a sub-region that contains the user's location and satisfies conditions specified in the user's profile is discovered. To achieve this, the LBS provider would, at each step of the traversal, provide the user with the boundary of the current sub-region. The user would terminate the traversal when the smallest qualifying sub-region is obtained. Clearly, partitioning methods that produce a hierarchical representation of the region (such as quad-tree partitioning) are more favored in this model for their faster traversal performance. However, this approach is only feasible if the necessary underlying structure is made available by the LBS provider. Otherwise, one of the partitioning techniques that has been illustrated earlier would be implemented either on the user device, or by a trusted third-party, depending on the system model. Note that only region boundary information is stored on the user's device (or the third-party component) and map information would be acquired from the LBS provider.

5.3 Problem Definition

Suppose that Alice wants to accomplish a few short tasks for which she needs to travel to different locations within a city. Along her way, she frequently uses a map application (e.g. Google Maps, MapQuest) on her hand-held device to find the shortest route between consecutive pairs of locations. Assume that Alice has submitted a request to the LBS provider at time q_{past}, while she was at location p_{past}. She submits a new request, at time q_{now}, with her current location p_{now}. Suppose that an anonymity-based algorithm is in place which reports $k-1$ other locations along with Alice's exact location each time she submits a query in order to mislead the malicious LBS provider regarding her exact location. Let P_{past} and P_{now} be the set of points (including Alice's location) reported at time q_{past} and time q_{now}, respectively.

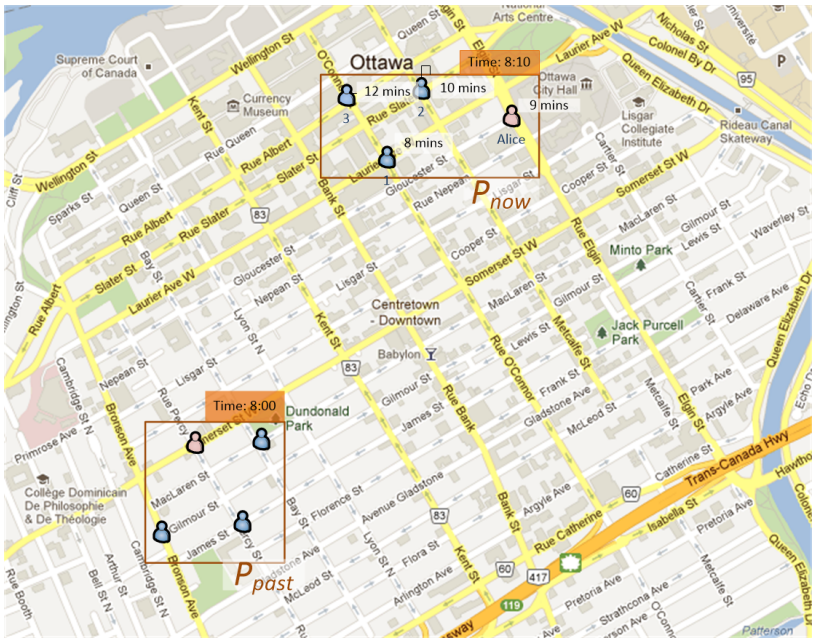


Figure 29: A sample case of inference attack using travel time information.

Due to speed limits on road links, some locations in P_{now} may not be feasible from any locations in P_{past} within the time period between two query submissions, namely $q_{now} - q_{past}$. We illustrate this next.

Consider the system snapshot of Figure 29, applied to two consecutive queries submitted by Alice at time $q_{past} = \texttt{8:00am}$ and time $q_{now} = \texttt{8:10am}$. A k-anonymity-based approach submits $k = 4$ locations (including Alice's location) to the LBS provider to hide the location of the querying user. For each location in P_{now}, the minimum travel time from any point in P_{past} is shown in the figure. Clearly, the location of User 3 is not feasible within $\Delta = q_{now} - q_{past} = 10$ minutes. Given this, a malicious LBS provider could discard the infeasible location in P_{now}, and obtain a better approximation of Alice's location. In the worst case, only her exact location at time q_{now} is feasible from points in P_{past} so that her exact location is exposed to the LBS provider.

A more general scenario is depicted in Figure 30. Figure 30(a) shows user locations

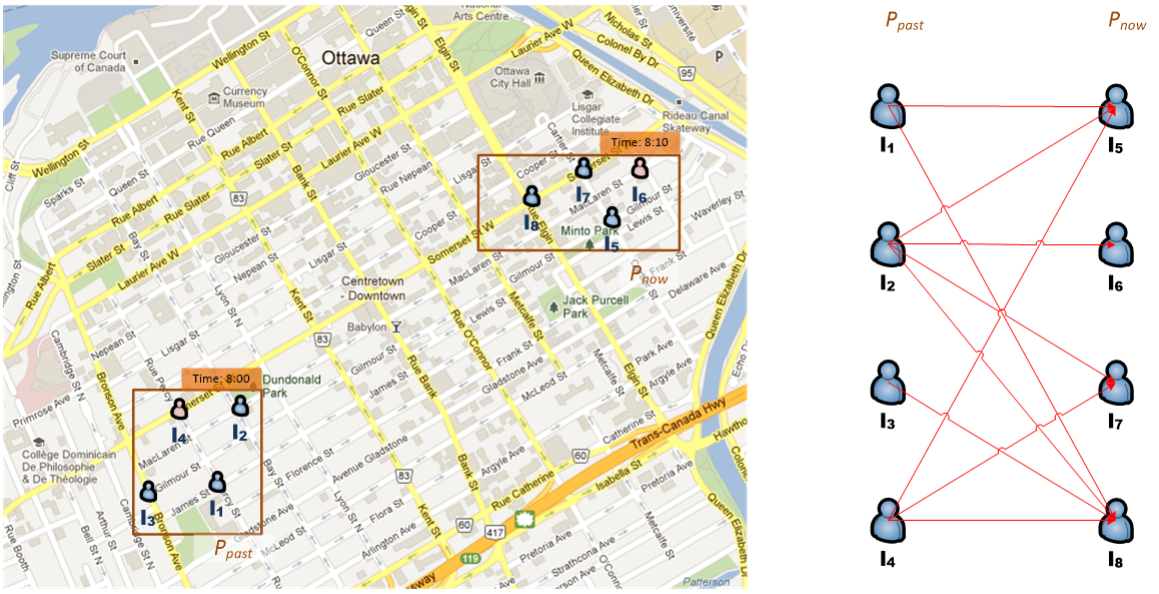


Figure 30: (a) Reported user locations for two consecutive queries (b) The feasibility graph

reported, per the k-anonymity method, at time q_{past} = `8:00am` and time q_{now} = `8:10am`. For each location in P_{past}, feasible locations in P_{now} are shown in Figure 30(b) where arrows represent feasibility between a pair of points. We say location p in P_{past} is feasible to location p' in P_{now} if the travel time between p and p' is less than or equal to $\Delta = q_{now} - q_{past} = 10$ minutes. Note that ℓ_6 is only feasible from location ℓ_2 in P_{past}, and ℓ_8 is feasible from all four locations $(\ell_1, .., \ell_4)$ in P_{past}. An adversary who had access to such location information could infer that the probability of the query originating from location ℓ_8 at time q_{now} is higher than that of it originating from ℓ_6, thus compromising privacy. On the other hand, ℓ_3 is only feasible to ℓ_5, and ℓ_2 is feasible to all four locations $(\ell_5, .., \ell_8)$ in P_{now}. Knowing this, an adversary could infer that the probability of the query originating from location ℓ_2 at time q_{past} is higher than that of it originating from ℓ_3; again, this constitutes a privacy breach. The problem is then to develop a protection system that protects privacy of users

against such inference attacks by untrusted LBS.

Note that similar scenarios could occur in region-based protection approaches such as obfuscation and cloaking. In these scenarios, a malicious LBS provider could use road-network speed-limit information to discard parts of the region or assign probabilities to different parts of the region in which the user is located, and thereby infer additional information about the user's identity and/or lifestyle.

5.4 Models and Solutions

As we discussed earlier, most commonly available LBS providers require that a user provides an exact location when submitting a query and location privacy protection methods that generate a generalized user location for queries (cloaking, for example, as well as some other obfuscation methods) are therefore not currently supported by most common LBS providers. Cryptography-based methods suffer from performance loss. K-anonymity-based location protection approaches are currently, therefore, the most efficient applicable privacy protection methods for map applications. A k-anonymity-based privacy protection service reports k locations (including that of the querying user) every time the user submits a query.

On its general setting, a location privacy technique based on k-anonymity, submits accurate user locations to the LBS provider. It is required to do so since commonly available LBS providers only accept precise input locations. These privacy protection solutions report the user's real location along with $k-1$ other locations, which originate either from real or from fake users. If $k-1$ locations are fake, then a malicious LBS might be able to use inference techniques to discard them and identify a specific user's identity. Alternatively, if all k locations are real, such an LBS would be able to obtain all k real locations. We provide a higher level of location protection and

hide the user's real location by applying inaccuracy to user queries, in that we only provide road-network intersections in the query, rather than the real location. Applying *inaccuracy* is a form of obfuscation in which a location other than the user's real location is reported to the LBS. Note that while the locations, as reported to the LBS, are inaccurate, they are precise and thus can be handled by current LBS.

Depending on the user's selected level of accuracy, our privacy protection system applies inaccuracy to the user's real location in one of two ways. When exact results are desired we report both endpoints of the link on which the user is currently located; this is depicted in Figure 31a. If the user is located at an intersection between two links, one of the adjacent links is selected at random. Conversely, if low accuracy is selected, an obfuscated location is reported to the LBS provider which is selected, at random, from among intersection points within distance r_i of the intersection point closest to the users real location (Figure 31b). (Low accuracy can be selected, for example, when a user knows the best driving route to nearby intersections.)

Note that, while the first approach provides exact results, its performance is inferior to that of a pure k-anonymity-based system because twice as many queries need to be submitted (one for each link end) to get an exact result. Conversely, in the second approach, the query performance would remain roughly the same as in a k-anonymity-based system. In this case though, the query result may possibly be inaccurate. Simply reporting the closest intersection point to the real location would, however, result in reasonable location obfuscation without sacrificing performance (Figure 31c). This approach picks the endpoints closest to the user and reports them. In approximately half of the cases, the result is accurate. For the other half, a good approximation is obtained, if we make the reasonable assumption that, within a city, links are not typically extremely long.

We have already shown that inference attacks may exploit speed limit information

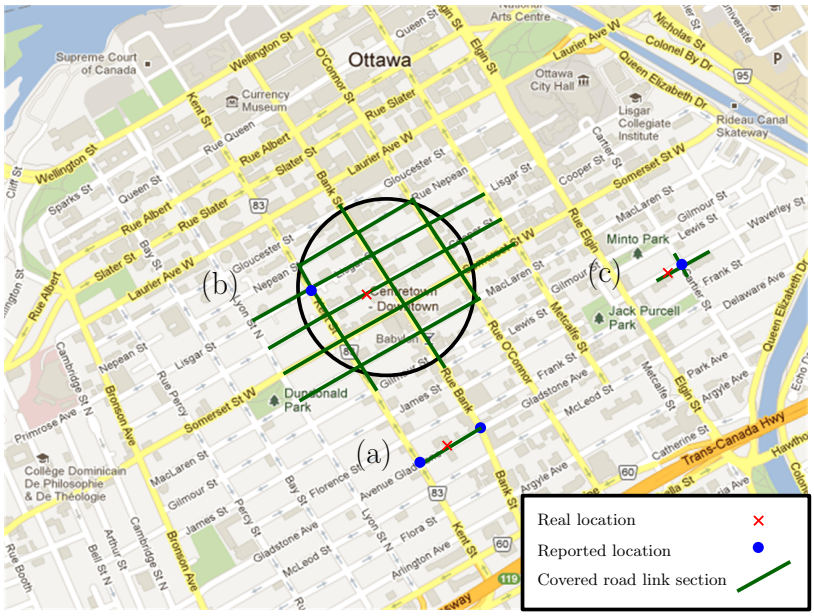


Figure 31: a) Location inaccuracy within a link b) Location inaccuracy within all road-links inside a disk c) Special case when the closest intersection is reported to the LBS

on road-networks to break a k-anonymity-based location protection system. Applying inaccuracy to user locations would only hide these among a few road-links. This may still reveal user identity and/or lifestyle information if an adversary were to obtain relevant contextual information. Applying inaccuracy therefore serves only as supplemental protection, and does not guarantee k-anonymity.

We introduce the notion of (i, j)-privacy, to protect users against attacks using travel time information that assign probabilities and discard less likely locations. Each user can set two values, $i, j \in \{1, \ldots, k\}$, either in their profile or at query time. Our protection service then determines whether each user location in P_{now} is feasible from at least i different locations in P_{past}, and whether each user location in P_{past} is feasible to at least j different locations in P_{now}. Larger values of i and j, provide a higher degree of anonymity in that the actual location of the user is hidden among other reported locations. Larger values of i provides more anonymity for user's past

location and larger values of j provides more anonymity for user's current location. However, as the value i and/or j increases, the likelihood that the (i, j)-privacy is met decreases. In cases in which (i, j)-privacy is not achieved for the user-selected values i, j, the user is notified and is provided with two solutions: (1) select a lower setting for privacy protection (i.e., chose lower values of i, j) or, (2) delay the query for some period of time. The time period is either calculated by our system and relayed to the user to make an informed decision, or pre-defined in the user's profile.

Let $G(V, E)$ be a road network, and $T_{x,y}(t)$ be a function that returns the shortest travel time for going from vertex x to vertex y at time t. We say G is transitive if for any three vertices of the road network, a, b, and c, and a start time t at a, we have $T_{a,c}(t) \leq T_{a,b}(t) + T_{b,c}(t')$, where $t' \geq t + T_{a,b}(t)$.

For transitive road networks, the notion of transitivity could be extended to (i, j)-privacy. Let A, B, and C be three consecutive sets of locations of size k that contain a user's location. Then, transitivity of (i, j)-privacy implies that if (i, j)-privacy holds from A to B, and from B to C, it must also hold from A to C. In the following, we show that (i, j)-privacy is transitive for transitive road networks. Let location a be feasible to location b for query times q_a and q_b, respectively, meaning that $T_{a,b}(q_a) \leq q_b - q_a$. Let location b be feasible to location c for query times q'_b and q_c, respectively, meaning that $T_{b,c}(q'_b) \leq q_c - q'_b$. Because A, B, and C correspond to consecutive queries, we have $q'_b \geq q_b$. By the network transitivity, and knowing that $q'_b \geq q_b \geq T_{a,b}(q_a) + q_a$, we have $T_{a,c}(q_a) \leq T_{a,b}(q_a) + T_{b,c}(q'_b) \leq q_b - q_a + q_c - q'_b \leq q_c - q_a$. Therefore, a is also feasible to c with respect to query times q_a and q_c, respectively. Thus, if each location in A is feasible to at least j locations in B, and each location in B is feasible to at least j locations in C, then, by the above argument, each location in A is also feasible to at least j locations in C. Therefore $(-, j)$-privacy holds for A and C. The same argument is true for $(i, -)$-privacy. Therefore, (i, j)-privacy is

transitive for transitive road networks.

However, for general road networks, (i, j)-privacy may not be transitive. A sample scenario where transitivity does not hold is when to go from a to c via b requires a prohibited left/right turn (or delays while trying to turn specially for large vehicles) and alternative (a, c)-paths result in longer travel times. To model this scenario, we could say that the cost of a path is the sum of the edge and vertex costs . Vertex costs capture the turning, and it is defined for each ordered pair of incident vertices (no left/right turn is cost infinity for the corresponding pairs of vertices). In such a case, we have $T_{a,c}(t) > T_{a,b}(t) + T_{b,c}(t')$, where $t' \geq T_{a,b}(t) + t$. Therefore, feasibility from a to b, and from b to c does not necessarily result in feasibility from a to c. Thus, for general road networks, if (i, j)-privacy holds from A to B, and from B to C, it does not necessarily hold from A to C.

In the following Section 5.5, we provide a more rigorous definition of the notion of (i, j)-privacy.

5.5 Formal Definition of $(i, j) - Privacy$

Suppose that a user had submitted a query to an LBS at source location s at time q_{past}. The user, meanwhile has traveled from s to destination d and now wants to submit a new query, say at time q_{now}. Let S and D be the source and destination regions obtained via a region partitioning technique of the road-network, where S and D hold the user locations s and d, respectively. Let P_S be the set of points in S reported to the LBS as per the anonymity-based approach. Let P_D be the set of points located inside D. Furthermore, let $T(p_s, p_d)$ be the travel time from $p_s \in P_S$ to $p_d \in P_D$. We say p_s is *feasible* to p_d (or p_d is *feasible* from p_s) with respect to query times q_{past} and q_{now}, if $T(p_s, p_d) \leq q_{now} - q_{past}$.

Given two integer values i and j, and a set of points $P_D^{ij} \subseteq P_D$, we say that P_D^{ij} is $(i, -)$-private, if for each $p_d \in P_D^{ij}$, there are at least i points in P_S that are feasible to p_d (see Figure 32a for an illustration). Similarly, we say that P_D^{ij} is $(-, j)$-private, if for each $p_s \in P_S$, there are at least j points in P_D^{ij} that are feasible from p_s (Figure 32b). Now, we say that P_D^{ij} is (i, j)-private, if P_D^{ij} is both $(i, -)$-private and $(-, j)$-private.

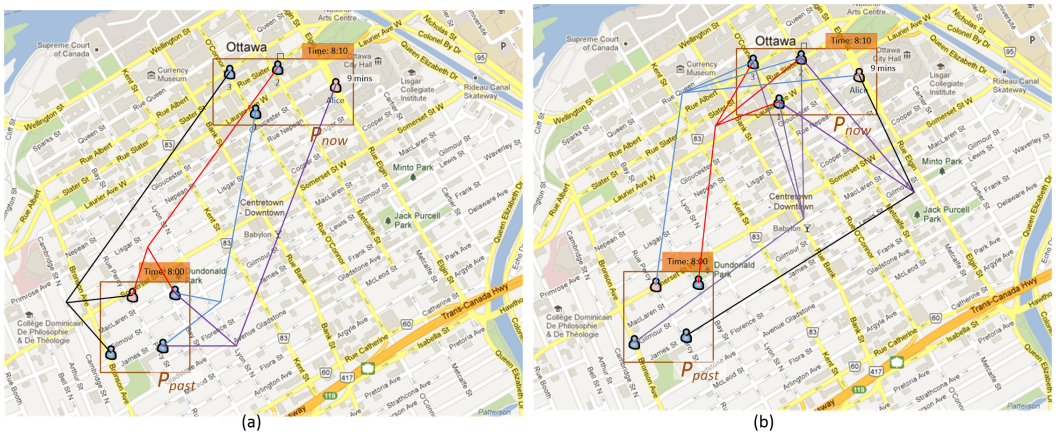


Figure 32: $(i, -)$-privacy and $(-, j)$-privacy coverage for $i = 2$ and $j = 3$.

5.6 Algorithms

In this section, we describe a heuristic solution to determine if (i, j)-privacy is guaranteed for a user making consecutive queries. We discuss several heuristics that are of different qualities; all of these are based on one core heuristic approach. Consider a user who is located at point s inside a given region S. Let P_S be a set of k locations inside S, including s, as reported by the k-anonymity approach at query time q_{past}. At time q_{now}, the user has moved to new location d, inside a region D, and submits a new query. Let P_D be a subset of size $n \geq k$ of all possible locations inside D. The task is to report k (i, j)-private locations in P_D, including the user's location, if such a solution exists.

5.6.1 (i, j)-privacy Algorithms

By definition, a subset of points in P_D is (i, j)-private, if it is both $(i, -)$-private and $(-, j)$-private.

$(i, -)$-privacy

Verifying the $(i, -)$-privacy is relatively simple. Let P_D^i be the subset of P_D in which all points that are not feasible from at least i points in P_S are discarded. It is then easy to see that a subset of P_D is $(i, -)$-private if it is a subset of P_D^i.

$(-, j)$-privacy

Conversely, finding a $(-, j)$-private subset of P_D of size k is more complicated. In fact, the minimization problem, in which the objective is to find the smallest size $(-, j)$-private subset, can be formulated as a $(0, 1)$-Integer linear programming problem which is NP-hard in general. The formulation is as follows:

$$\begin{aligned} &\text{minimize } \sum_{d \in P_D} x_d \\ &\text{subject to } \sum_{d \in P_D} f(s, d).x_d \geq j, \text{ for all } s \in P_S \\ &\text{where } x_d, f(s, d) \in \{0, 1\}, \text{ for } s \in P_S \text{ and } d \in P_D \end{aligned}$$

Here, x_d is equal to 1 if d is selected, and is 0, otherwise. The function, $f(s, d)$, returns 1 if s is feasible to d, and 0, otherwise.

We conjecture that solving $(-, j)$-privacy is NP-complete. In fact, it is straightforward to show that solving $(-, 1)$ privacy on general graphs is NP-complete. This can be achieved by relabeling an instance of the Set Cover problem. The Set Cover problem is defined as follows: we are given a triple $\langle X, C, \ell \rangle$, where X is a finite set

of elements, C is a collection of subsets of X, and ℓ is an integer. Is there a subset $C' \subseteq C$ of cardinality $|C'| \leq \ell$ such that the elements of C' cover X? We now relabel an instance of the Set Cover problem as an instance of the $(-, 1)$-privacy problem. We construct a feasibility graph G with set of source vertices P_S and destination vertices P_D as follows: for $x \in X$ add a vertex p_x to P_S, and for $y \in C$ add a vertex p_y to P_D. We add an edge (p_x, p_y) to G if $x \in y$. Let $k = l$. Then, a solution for the Set Cover problem is also a solution for the $(-, 1)$-privacy problem, and vice versa.

Given our NP-completeness conjecture, it may be unlikely that an efficient algorithm can be found. Therefore, it is desirable to devise approximation algorithms that do perform well in practice. Note that one could use greedy approximation algorithms for the Set Cover problem [39] to provide approximation algorithms for a special case of (i, j)-privacy problem where $j = 1$. Inapproximability results show that the greedy algorithm provides the best polynomial time approximation algorithm for the Set Cover problem under plausible complexity assumptions [9, 107]. We provide heuristic approximations that are also greedy and are based on the following common skeleton:

1. Determine the largest $(i, -)$-private subset of P_D, denoted by P_D^i.

2. Select a "well chosen" $(-, j)$-private subset of P_D^i. Let k' be the size of the selected subset.

3. Let k be the anonymity parameter provided by the user. If $k < k'$, inform the user that (i, j)-privacy is not possible. Otherwise, add additional points to the subset should that be required

Note that, when users are notified that (i, j)-privacy is not possible, they may either accept the risk and submit the query, or delay the query for a certain amount of time (calculated by the system or specified in the profile). In the later case, privacy is then re-examined later. In what follows, we provide three variations of the

above skeleton approach. In Section 6.5, these are then evaluated and experimentally compared.

Algorithm 1

In the first heuristic, we select points from P_D^i greedily starting from a point that is feasible from the maximum number of points in P_S. By selecting points from P_D^i, that is, the largest $(i,-)$-private subset of P_D, we ensure that the $(i,-)$-privacy is met (line 1). Next we determine the (i,j)-privacy using set P_D^i. Let $fn(a,A)$ be a function that returns the number of feasible points in set A to/from point a. To obtain the required greedy choice, we initialize priority queue Q implemented as a min-heap using -$fn(p,P_S)$ for $p \in P_D^i$ as the key (lines 2-3). Let L be a set that stores points selected by the algorithm for reporting to the LBS. We initialize L to $\{pt\}$, where pt is the querying user's location (line 4). We assume that $p_t \in P_D^i$. Throughout the main iteration of the algorithm, we remove an item with the smallest priority value and add it to set L while the priority queue is non-empty (lines 5-7). The *while loop* stops when L is (i,j)-private and has size less than k, in which case, $k-|L|$ points will be added to L as per k-anonymity property, and L is returned (lines 8-13). Otherwise, if such a set is not found, the algorithm detects that (i,j)-privacy cannot be met and reports "failure" (line 14), in which case, the user will be notified and asked to either accept that his/her privacy might be compromised, or delay the query for a certain amount of time.

Note that, while the selection of i do not affect the running time of the algorithm, setting larger values for j may increase the running time as more iterations may be required to satisfy $(-,j)$-privacy.

Algorithm 1 (P_S, P_D, i, j, k, pt)

```
1: set P_D^i = X ⊆ P_D : X is (i, −)-private, |X| is maximized
2: for p ∈ P_D^i do
3:     set Q(p) = −fn(p, P_S)
4: set L = {pt}
5: while Q ≠ ∅ do
6:     p ← popitem(Q)
7:     L ← L ∪ {p}
8:     if |L| ≥ k then
9:         break loop
10:    if L is (i, j)-private then
11:        for i = 0 to k − |L| do
12:            L ← L ∪ popitem(Q)
13:        return L
14: return ERROR: "privacy not possible for the chosen settings of (i, j)"
```

Algorithm 2

For the second algorithm, we modify Algorithm 1 to further improve the greedy choice. Let $f(a, A)$ be the set of all feasible points in $A \subseteq P_D$ from point $a \in P_S$ (or points in $A \subseteq P_S$ to point $a \in P_D$). If in some iteration of the algorithm, a point p in P_S is feasible to at least j points in L, then there is no need to check the $(-, j)$-privacy for p in the following iterations. In other words, for each point p' in $f(p, P_D^i - L)$, the $fn(p', P_S)$ value would decrease by one (the priority value increases by one). This affects both the priority values and the greedy choice. Thus, in Algorithm 2, we update the priority queue when a point in P_S is feasible to at least j points in L (lines 8-11). Therefore, a better greedy choice has been obtained for subsequent steps.

Algorithm 3

In our third heuristic solution, the algorithm selects, at each step, a point of P_D^i that is feasible from a point p in P_S for which $fn(p, L)$ is minimized. This way, the

Algorithm 2 (P_S, P_D, i, j, k, pt)

```
1: set P_D^i = X ⊆ P_D : X is (i, −)-private, |X| is maximized
2: for p ∈ P_D^i do
3:     set Q(p) = −fn(p, P_S)
4: set L = {pt}
5: while Q ≠ ∅ do
6:     p ← popitem(Q)
7:     L ← L ∪ {p}
8:     for p ∈ P_S do
9:         if p is feasible to at least j points in L then
10:            for p' ∈ f(p, P_D^i − L) do
11:                Increase-Key(Q, p', 1) //improves the greedy choice in future steps
12:    if |L| ≥ k then
13:        break loop
14:    if L is (i, j)-private then
15:        for i = 0 to k − |L| do
16:            L ← L ∪ popitem(Q)
17:        return L
18: return ERROR: "privacy not possible for the chosen settings of (i, j)"
```

algorithm first focuses on points in P_S that are furthest away from being feasible to at least j points of L and satisfying the $(-, j)$-privacy. As a result, the algorithm does not select too many points of P_D^i which are $(-, j)$-private for few points in P_S. We initialize priority queue Q with $Q(p) = 0$ for $p \in P_D^i$ (ine 3). Ties between priority queue elements are resolved by using $-fn(., P_S)$ value as the secondary priority value for priority queue items (line 4). In this way, the greedy choice is further improved. At each step of the algorithm, the priority queue is updated with $Q(p) = \min_{p' \in f(p, P_S)} fn(p', L)$ for $p \in P_D^i - L$, and ties are resolved to ensure that points are dequeued in the right order (line 9-11). Note that, in Algorithm 3, the secondary priority value is applied using a separate function for the sake of clarity. However, in the implementation, this is integrated in the priority queue operation functions.

Theorem 8. *Any solution returned by the heuristics 1, 2, and 3 guaranties* (i, j)*-privacy.*

Algorithm 3 (P_S, P_D, i, j, k, pt)

set $P_D^i = X \subseteq P_D : X$ is $(i, -)$-private, $|X|$ is maximized
for $p \in P_D^i$ **do**
    set $Q(p) = 0$
    resolve_tie$(Q, P_S, P_D, fn(.,.))$
set $L = \{pt\}$
while $Q \neq \emptyset$ **do**
    $p \leftarrow$ popitem(Q)
    $L \leftarrow L \cup \{p\}$
    for $p \in \mathcal{P}_D^i - L$ **do**
        $Q(p) = \min_{p' \in f(p, P_S)} fn(p', L)$
        resolve_tie$(Q, P_S, P_D, fn(.,.))$
    if $|L| \geq k$ **then**
        break loop
    if L is (i, j)-private **then**
        for $i = 0$ to $k - |L|$ **do**
            $L \leftarrow L \cup$ popitem(Q)
        return L
return ERROR: "privacy not possible for the chosen settings of (i, j)"

Proof. In all algorithms, we discard points that are not $(i, -)$-private. Additionally, all heuristics will only report a solution when the number of $(-, j)$-private locations (k') is not greater than k. Let P_D^{ij} be the k' locations found by an algorithm. P_D^{ij} holds feasibility, $(i, -)$-privacy, and $(-, j)$-privacy and adding more feasible and $(i, -)$-private locations, namely $k - k'$ points, would not violate these conditions. Therefore, k locations, reported by any of the algorithm, guarantee (i, j)-privacy. □

Note that these heuristics may find $k' > k$ locations to meet (i, j)-privacy. In such cases, they report an error, meaning that privacy cannot be met. However, it could be the case that $k_{opt} < k$, where k_{opt} is the optimal solution. Thus, our algorithm may incorrectly report an error when a solution exists. To see if this situation is frequent or rare, we turned to experiments. Our experimental results show that all algorithms perform well in practice and report optimal or close to optimal solutions in most cases (Section 6.5).

5.6.2 Distance Constrained (i, j)-privacy Algorithms

In a k-anonymity-based privacy protection approach, a malicious LBS provider might be able to classify k anonymous locations into a more general category (e.g., hospital, store, or bar), if these locations are in close proximity to each other. This may reveal a querying user's private information. To protect users from such attacks, we introduce a feature that we call an *empty-vicinity constraint*. For this, we create disks centered at each location reported to the LBS provider. The radii are selectable by users via a feature called an *empty-vicinity value* and denoted by r_e. No other location is allowed to fall inside these disks. If users choose a value for r_e that is too large, it may not be possible to guarantee privacy. We need to solve the following problem: given an empty-vicinity value r_e and a set of locations S, the decision problem is to **determine whether or not it is possible to find a subset of locations in S of size k such that their pairwise distance is at least r_e**. This problem is equivalent to finding a maximum independent set in a *Unit Disk Graph* which is known to be NP-hard [40]. The optimization version of the problem is stated in the following theorem.

Theorem 9. *The problem of finding the largest subset of a set of locations S, so that their pairwise distance is at least r_e, is NP-hard.*

Proof. Proof follows from the NP-hardness of [21]. □

Algorithm I4

We devised a greedy heuristic. Denote by c_p the number of points within distance r_e of $p \in P_D^i$. Our heuristic now selects points from a priority queue containing points in P_D^i with priority values c_p. In each iteration, the heuristic removes from its priority queue, a point with the smallest priority value and all the points within distance r_e from it. The priority value of the remaining points is updated accordingly. However,

this method alone does not protect against inference attacks that exploit travel time limitations on road-networks. To achieve a higher level of location privacy, we combine the concept of (i, j)-privacy and empty vicinity. This is accomplished by removing points in P_D^i that are within distance r_e of the selected point in each iteration of the $(i, j)-$privacy algorithms presented in Section 5.6.1. For future reference, we denote the algorithm obtained from such combination with Algorithm 2, by Algorithm *I4*.

Algorithm I5

We can further improve Algorithm *I4* by resolving ties between priority queue elements with respect to their c_p value (defined earlier). This approach forces the algorithm to discard fewer points of P_D^i in each iteration of the algorithm which, as a result, may cause the algorithm to select fewer points from P_D^i in order to satisfy both the (i, j)-privacy and empty vicinity conditions. The new improved algorithm is referred to as Algorithm *I5*.

Theorem 10. *A solution returned by heuristics I4 and I5 guarantees* (i, j)*-privacy.*

Proof. Since both algorithms are based on the greedy approach used for algorithms in Section 5.6.1, the feasibility condition and the (i, j)-privacy hold for them. Additionally, when a location is selected in one of the algorithms, all locations within distance r_e of that location will be discarded. Therefore, the k locations returned by both algorithms are within distance r_e of other locations, and meet the distance constraint. □

Note that, similar to Algorithms 1, 2, and 3, Algorithms *I4* and *I5* may not provide optimal solutions, meaning that they may return "ERROR" even if a solution exists. Also note that adding a distance constraint may increase the chance that a query

delay is required, because more feasible points might be acquired to compensate for the points that have been discarded per the requirement of the distance constraint.

5.7 Experimental Results

We implemented the algorithms, presented in Section 5.6, using *Google Map JavaScript API* [1] to test their performance and the quality of their results. We ran our tests on a netbook with 1.0 GHz Intel Atom CPU and 1 GB of memory on a 10 Mbps Internet connection. We have also run tests on a wireless device, more specifically, a Samsung Galaxy S3 smart phone running an Android OS with 4G network connection. To test the algorithms independence of the region partitioning scheme, we computed two random rectangular regions, S and D, representing source and destination regions, respectively. These regions are allowed to have different sizes, say between 20 to 200 intersections in order to avoid regions that contain too few or too many points. As mentioned earlier, in our model description, only road intersections and endpoints are reported to the LBS provider for the sake of location obfuscation. In our implementation, road intersections/endpoints inside a rectangular sub-region were obtained from OpenStreetMap[2] because we were not aware of any simple way to obtain this information using Google Map API. The set of points obtained from OpenStreetMap is used for running computations by Google Map API. Note that the two operations are independent and does not lead to any conflict. We also selected k random points from S to represent k locations reported to the LBS provider per the k-anonymity method at query time q_{past}.

Let P_S be a set of k locations inside S (P_S includes the user's location) and P_D be the set of intersections/endpoints inside D. We want (i, j)-privacy to hold for two

[1] Available at `http://code.google.com/apis/ maps/documentation/javascript/`

[2] Available at `http://www.openstreetmap.org`.

consecutive queries as described in Section 5.5. In our implementation, we assume that P_S for time q_{past} is given (black points in Figure 33), and the user has arrived at a new, obfuscated location inside P_D. Now, say at time q_{now}, the user would like to submit the next query. The query's goal is to select k locations from P_D, which include the querying user's location so that (i, j)-privacy is guaranteed. (A snapshot of the privacy protection system is shown in Figure 33.)

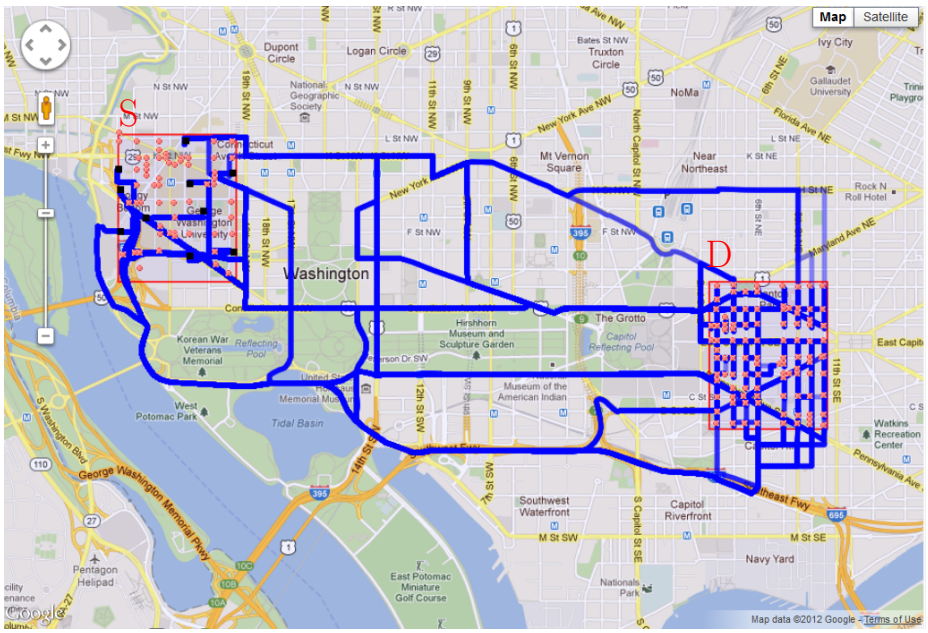


Figure 33: A snapshot of the privacy protection system in pre-computation step.

Verifying (i, j)-privacy requires testing the feasibility from locations in P_S to all locations in P_D. For that, the smallest travel time between each pair of points is needed. Smallest travel time computation is currently one of the basic features of most LBS providers. We use Direction API and Distance Matrix API from Google Map JavaScript API to obtain the travel times efficiently. Hundreds of queries can be answered in less than a second[3]. Figure 33 depicts all smallest travel time paths between points in regions corresponding to two consecutive queries.

[3]Distance matrix from Google Map API could answer 100 travel time queries in about half a second.

5.7.1 (i, j)-Privacy Algorithms Implementation and Results

In this section, we present the quality and performance analysis of the (i, j)-privacy preserving algorithms through experimental tests.

In most of the runs, all algorithms, presented in Section 5.6, could find j (i, j)-privacy preserving locations in P_D when at least j locations are found to be feasible to each location in P_S. The geometric properties intrinsic to the problem may play a major role explaining this behavior. Figure 35 displays the number of locations selected by each algorithm for different values of q_{now} (x-axis), for the pair of source and destination regions shown in Figure 34. In this example, for $i = 3$, $j = 8$, and $k = 10$, as soon as j feasible locations are available from all locations in P_S, (i, j)-privacy holds for those locations in all algorithms.

We also observe that the minimum number of locations required to satisfy (i, j)-privacy typically increases as the distance between source and destination regions decreases. Figure 36 depicts two regions that are $\sim 3 - 10$ minutes away from each other and Figure 37 shows the number of required locations to satisfy (i, j)-privacy. As the distance between source and destination regions is decreased, so does the size of the intersection of the feasible point-sets containing points in P_S. This is more noticeable in the marginal time that is the smallest query time at the destination region, namely q_{now}, at which, for all $s \in P_S$, s is feasible to at least j locations in P_D. An (i, j)-private subset of P_D, therefore, forms at a later query time. Conversely, the probability of having the same travel time value between pairs of points decreases when the distance between source and destination regions is decreased. Note that, in all algorithms, start times and travel times are stated in seconds.

It is easy to conclude that the worst-case scenario occurs when the source and destination regions are identical. Figures 38 and 39 depict such a case. This is in line with our observation of the geometric property.

Figure 40 shows the impact of the region size on the number of selected locations. In this example, the distance between regions is the same as Figure 36, but the region size is halved.

For the (i, j)-privacy and $k-$anonymity to hold, we need to delay the query for 20 seconds in the larger region (Figure 37) and 5 seconds in the smaller region (Figure 41).

Figures 42 and 43 show the result of Algorithms *I4* and *I5* on the four regions mentioned above (same regions, close regions, far regions, and halved regions). Here, the size of the empty disk is set to 100 meters. In general, the same properties, explained earlier for Algorithms 1, 2, and 3, hold for Algorithms *I4* and *I5* as well. However, the same number of (i, j)-private locations may only become available at a later query time. The reason for this is the additional distance condition, because of which, some locations that could positively contribute to $(-, j)$-privacy are discarded by the algorithms.

Let T_{SD} be the set of all travel times from points in P_S to points in P_D. Let q_{past} be equal to 0, and $t_m \in T_{SD}$ be the marginal query time. We refer to $T^v_{SD} = \{t \in T_{SD} : t \geq t_m\}$ as the set of valid query times. Note that for $q_{now} \in T_{SD} - T^v_{SD}$, (i, j)-privacy cannot be satisfied due to an insufficient number of feasible points in the destination region, for each point in the source region.

In the following, we present specific characteristics of our (i, j)-privacy preserving algorithms.

1. When the travel time between source and destination regions decreases, the number of locations that each algorithms selects, increases on average over all valid query times values, namely $q_{now} \in T^v_{SD}$. To further illustrate this property, we examined the problem in a purely geometric setting, where Euclidean distances are substituted by travel times. Figure 44 depicts this. Here, all points

in region S (namely s_1, s_2, and s_3) are vertically aligned and the regions induced by intersecting D with disks of a specific radius centered at s_1, s_2, and s_3 are shown (we refer to these regions as R_1, R_2, and R_3, respectively). The radii of the disks in Figures 44 (a) and (b) are selected in a way that the corresponding intersection regions have the same area size. Analogously, for the radii of the disks in Figures 44 (c) and (d). Notice that, while the intersection of R_1, R_2, and R_3 is non-empty for regions that are far apart (Figure 44 (a)), it is empty for regions that are relatively close (Figure 44 (b)). A similar scenario is likely to occur in the road-network setting. For close regions, the size of the set of points feasible to at least j points in P_S, would tend to be smaller than for the far regions. Therefore, for close regions, more locations should be selected to satisfy (i, j)-privacy.

2. For regions far enough apart, all algorithms show similar results and find j (i, j)-private locations in P_D, for all $q_{now} \in T^v_{SD}$. In all our experiments, this is the case for source and destination regions beyond city boundaries. The same is true, even within most city boundaries, unless the source and destination regions are unrealistically large, or are very close (in most cases it happens if they touch or overlap).

3. For regions close together, Algorithm 1 yields the worst results and Algorithm 3 shows the best quality results, on average. In almost all cases, Algorithm 2 selects fewer locations than Algorithm 1. In few instances, it may outperform Algorithm 3. Eventually, all algorithms converge to a solution of size j. This happens at about the same time for all the algorithms. Note that in a few instances, the number of points selected may increase as the query times increases. This is because ties are broken arbitrarily which may increase the

number of selected points even for later query times.

4. In general, Algorithms *I4* and *I5* perform about the same as do Algorithms 1, 2, and 3. On average, Algorithm *I5* performs better than Algorithms *I4* and selects fewer locations to report to the LBS provider. This is because fewer points are discarded at each step of Algorithm *I5*. Therefore, more points are available to contribute to satisfying (i, j)-privacy. Because of the additional distance condition, (i, j)-privacy generally holds for later query times in Algorithms *I4* and *I5*, when compared to Algorithms 1, 2, and 3.

In all tests for $k \leq 20$ and $P_D \leq 200$, the running time was less than 2 seconds. This is considered an acceptable delay for typical everyday LBS applications. This time becomes irrelevant if a greater delay is necessary to satisfy (i, j)-privacy. Note that we implemented the algorithms without attempting to optimize speed or perform additional tweaking. This is left to a potential commercial implementation.

5.8 Conclusions and Future Work

In this chapter, we presented a location privacy protection system for users that frequently submit location information to LBS providers. We studied possible inference attacks by adversaries and provided heuristic defense techniques to protect user privacy against such attacks. Our heuristic privacy-preserving algorithms provide users with flexible control parameters that allow them to customize privacy level, based on their needs. Our experimental evaluations show that, unless the parameters are set unrealistically, our algorithms provide quality results in less than few seconds.

A future work is to extend this work to provide location privacy protection for users who continuously report their location to a malicious LBS provider. In a similar setting, one could also study the applicability of these results to the trajectory

anonymization problem. We conjectured that finding $(-, j)$-privacy is NP-hard on road networks. It is still open to show whether or no the conjecture is true.

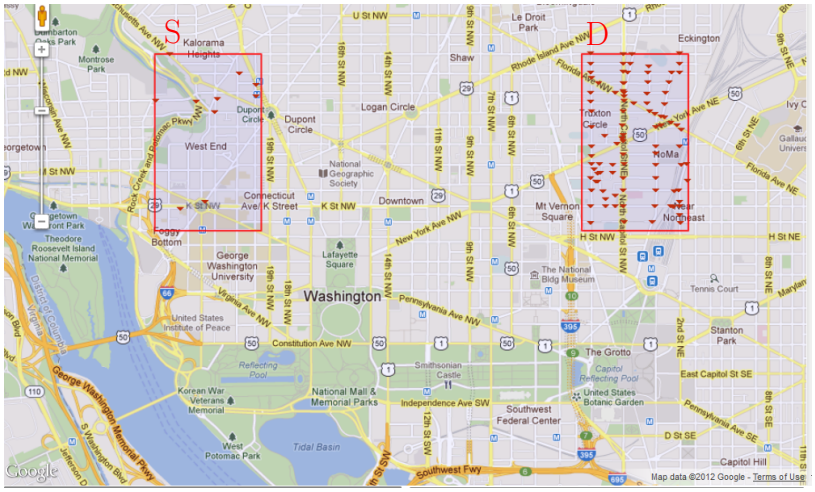


Figure 34: A sample source and destination regions that are $\sim 8-15$ minutes away.

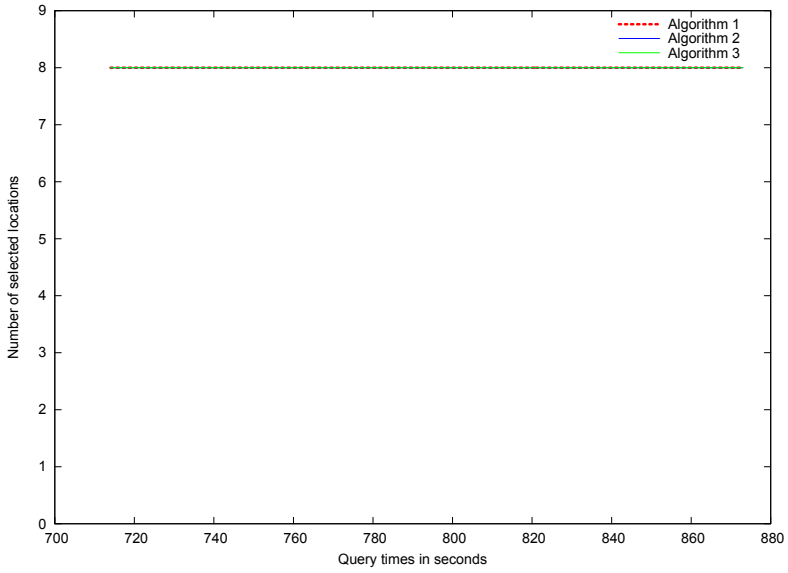


Figure 35: The number of selected locations for different query times.

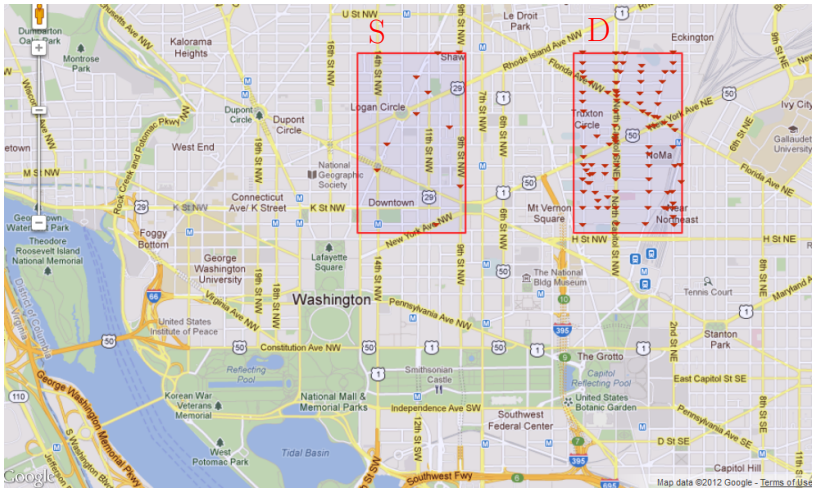


Figure 36: A sample source and destination regions that are $\sim 3-10$ minutes away.

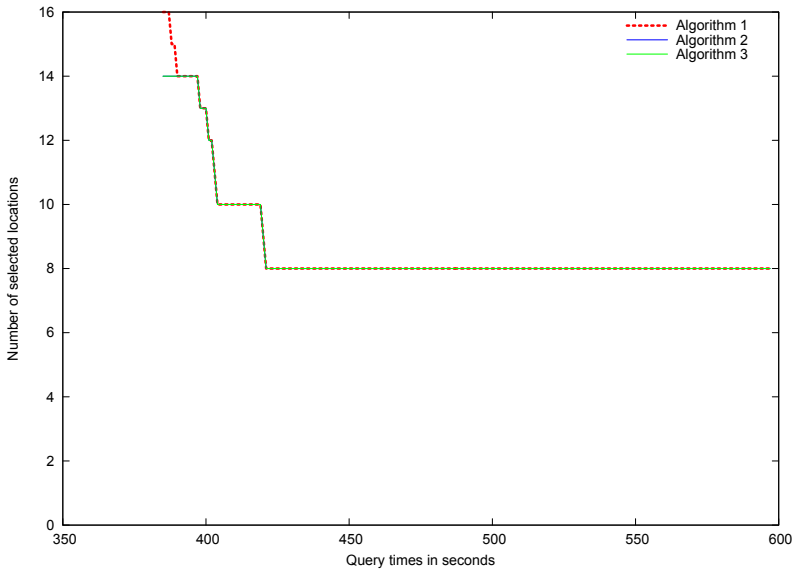


Figure 37: The number of selected locations for different query times.

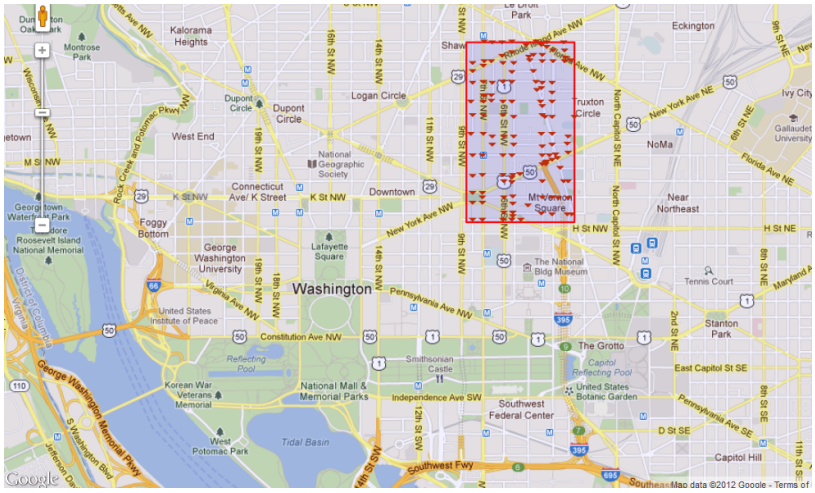


Figure 38: A sample case where source and destination regions are the same.

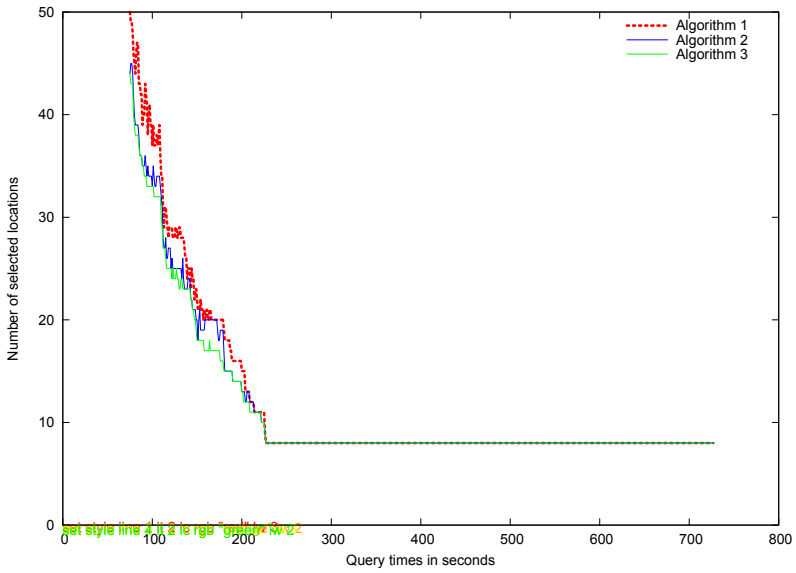


Figure 39: The number of selected locations for different query times.

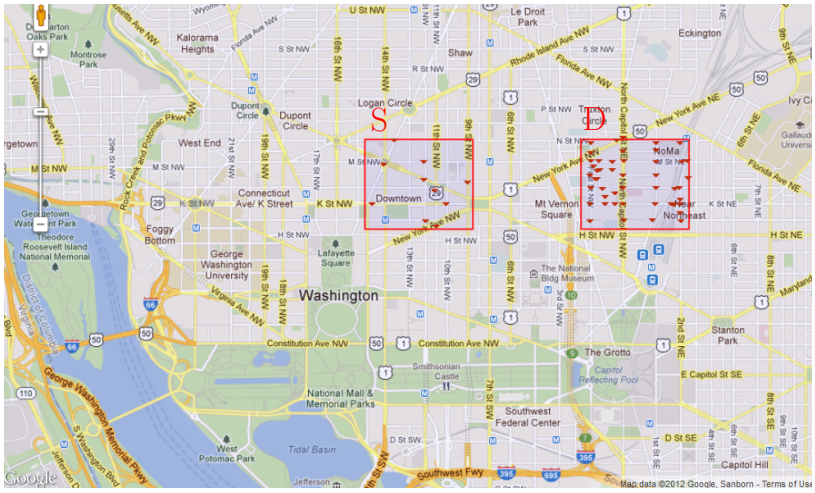


Figure 40: A sample case with half size regions.

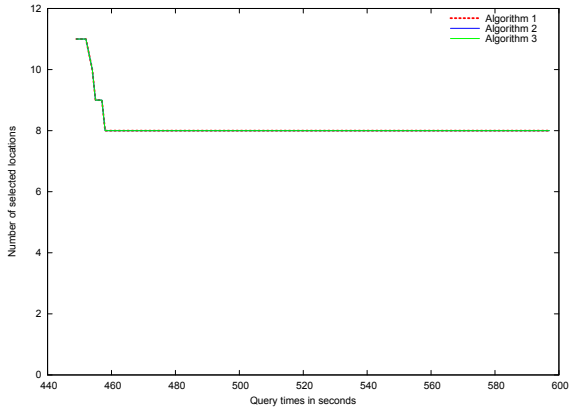


Figure 41: The number of selected locations for different query times.

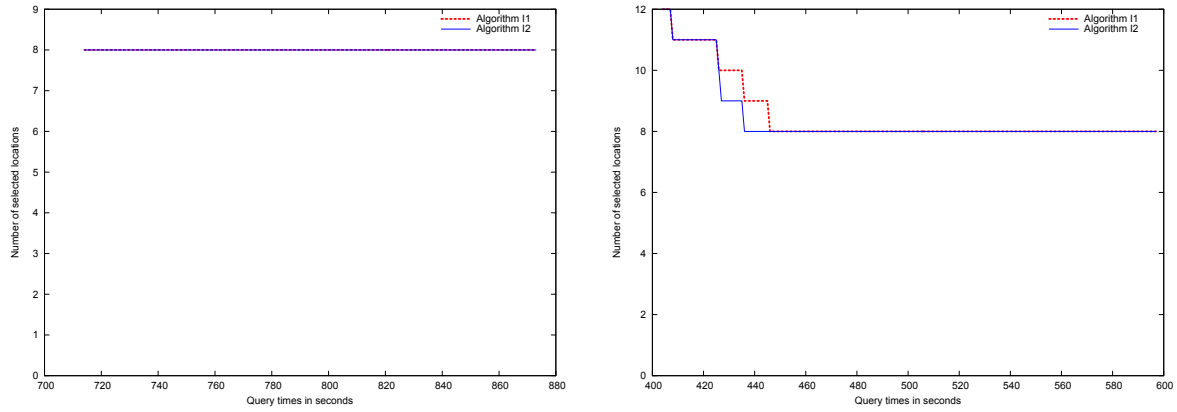


Figure 42: The number of selected locations with distance condition for the far regions example (left) and the close regions example (right).

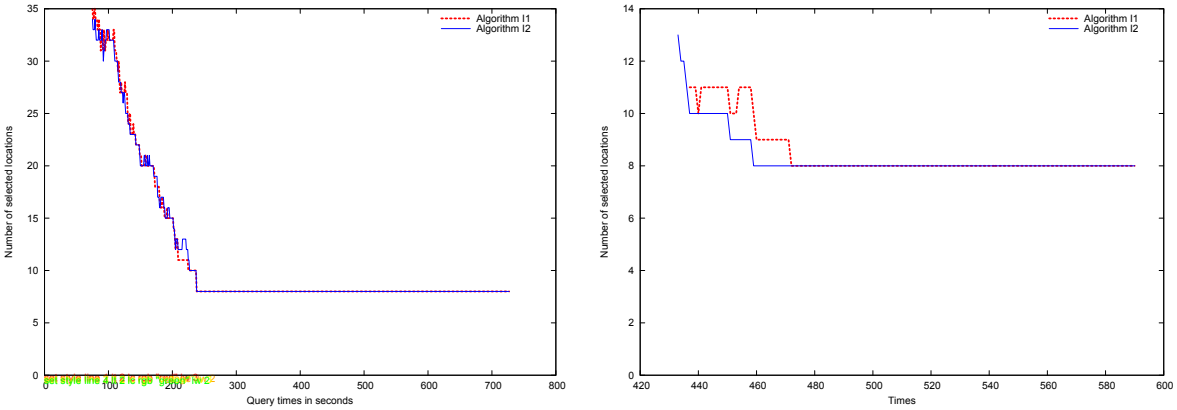


Figure 43: The number of selected locations with distance condition for the overlapping regions example (left) and the halved regions example (right).

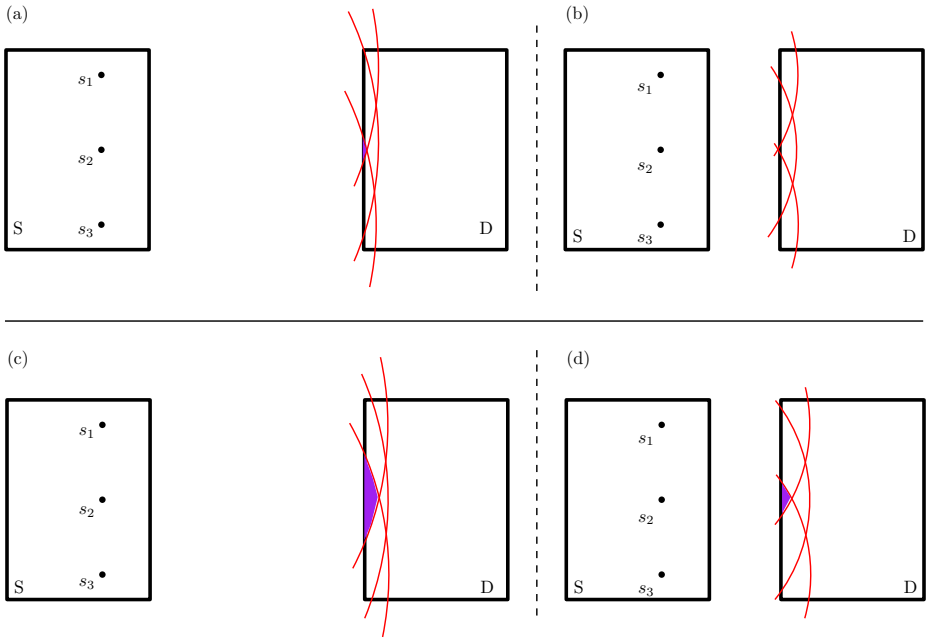


Figure 44: The effect of decreasing distances between regions in a pure geometric setting. a,b) small feasible regions. c,d) large feasible regions.

Chapter 6

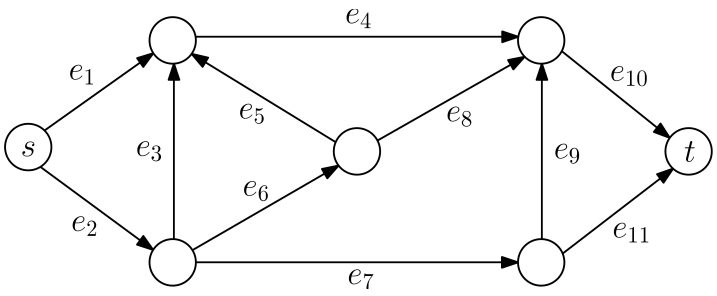
Finding Paths with Minimum Shared Edges

Motivated by a security problem in Geographic Information Systems, we study the following graph theoretical problem: given a graph G, two special nodes s and t in G, and a number k, find k paths from s to t in G so as to minimize the number of edges shared among the paths. This is a generalization of the well-known disjoint paths problem. While disjoint paths can be computed efficiently, we show that finding paths with minimum shared edges is NP-hard. Moreover, we show that it is even hard to approximate the minimum number of shared edges within a factor of $2^{\log^{1-\varepsilon} n}$, for any constant $\varepsilon > 0$. On the positive side, we show that there exists a $(k-1)$-approximation algorithm for the problem, using an adaption of a network flow algorithm. We design some heuristics to improve the quality of the output, and provide empirical results.

6.1 Overview

The *Minimum Shared Edges* problem is formally defined as follows:

Problem 1 (Minimum Shared Edges (MSE))**.** *Given a graph $G = (V, E)$, two special*



$\pi_1 = \langle e_1, e_4, e_{10} \rangle$
$\pi_2 = \langle e_2, e_7, e_{11} \rangle$
$\pi_3 = \langle e_2, e_6, e_8, e_{10} \rangle$
$\pi_4 = \langle e_2, e_3, e_4, e_{10} \rangle$
$\pi_5 = \langle e_2, e_7, e_9, e_{10} \rangle$
$\pi_6 = \langle e_2, e_6, e_5, e_4, e_{10} \rangle$

Figure 45: A graph G with six possible (s,t)-paths, denoted by π_1 to π_6.

nodes $s, t \in V$*, and an integer* $k > 0$*, find a set* P *of* k *paths from* s *to* t *in* G *so as to minimize* $c(P) = \sum_{e \in E} \lambda(e)$*, where* $\lambda(e) = 0$ *if* e *is used in at most one path of* P*, and* $\lambda(e) = 1$ *otherwise. An edge* e *with* $\lambda(e) = 1$ *is called a* shared edge.

We assume, without loss of generality, that the input graph is directed. Figure 45 illustrates an instance of the MSE problem on a sample graph. For $k = 2$, the minimum possible number of shared edges is zero, attained by two paths π_1 and π_2. For $k = 3$, the minimum number of shared edges is two, realized by the set $\{\pi_1, \pi_2, \pi_3\}$. Any other set of three paths leads to a higher number of shared edges.

For the special case, where the number of shared edges is required to be zero, the MSE problem is reduced to the "disjoint paths" problem which can be solved in polynomial time by using standard maximum flow algorithms. In particular, one can use Goldberg and Rao's binary blocking flow algorithm [74] to find k disjoint paths in a graph $G = (V, E)$ in $O(m \min(n^{2/3}, m^{1/2}) \log(n^2/m) \log k)$ time, where $n = |V|$ and $m = |E|$. An improved algorithm is provided for the special case of $k = 2$ [137]. See also [96] for the related problem of finding "shortest" disjoint paths in a graph.

More complex variants of the disjoint paths problem have also been studied in the literature. The problem of finding a pair of disjoint (s,t)-paths such that the length of the shorter path is minimized is addressed in [147] and proved to be NP-hard. The authors show that finding a k-approximation of the optimal solution is NP-hard for any $k > 1$. Li et al. [106] studied a similar problem in which the objective is to

find a pair of disjoint (s,t)-paths such that the length of the longer path is minimized and showed that it is NP-hard. Itai et al. [88] showed that the problem of finding a pair of disjoint (s,t)-paths such that the length of the shorter and longer path is bounded by Δ_1 and Δ_2, respectively, is also NP-hard. The general case of finding k min-sum (s,t)-disjoint paths where every edge is assigned k different costs and the j^{th} edge cost is associated with the j^{th} path, has been studied by Li et al. [105] and proved to be NP-hard.

A closely related problem studied in the context of communication networks is the so-called "k-best paths" problem [25, 117]. In this problem, the objective is to find a set P of k paths with minimum edge sharability, which is defined analogously to Problem 1, with the only difference that here, for each edge e, $\lambda(e) = 0$ if e is used in at most one path of P, otherwise $\lambda(e)$ is equal to the number of paths containing e minus 1. As shown in [104, 154], the k-best paths problem is polynomially solvable using a minimum-cost flow algorithm.

Despite its close similarity to the k-best paths problem, the minimum shared edges problem studied in this chapter turns out to be more challenging [121, 123]. In particular, we prove that the minimum shared edges problem is NP-hard. Moreover, we show that the problem admits no $2^{\log^{1-\varepsilon} n}$-factor approximation, for any constant $\varepsilon > 0$, unless $NP \subseteq \text{DTIME}(n^{\text{polylog}\, n})$ where $DTIME(f(n))$ is the class of all decision problems that are solvable by a deterministic Turing machine within time $f(n)$ on inputs of length n. On the other hand, we show that there exists a $(k-1)$-approximation algorithm for the problem, using a simple adaption of a network flow algorithm. We propose some heuristics for improving the quality of the algorithm. Our empirical results show that the resulting algorithm works reasonably well in practice.

6.2 NP-Hardness Proof

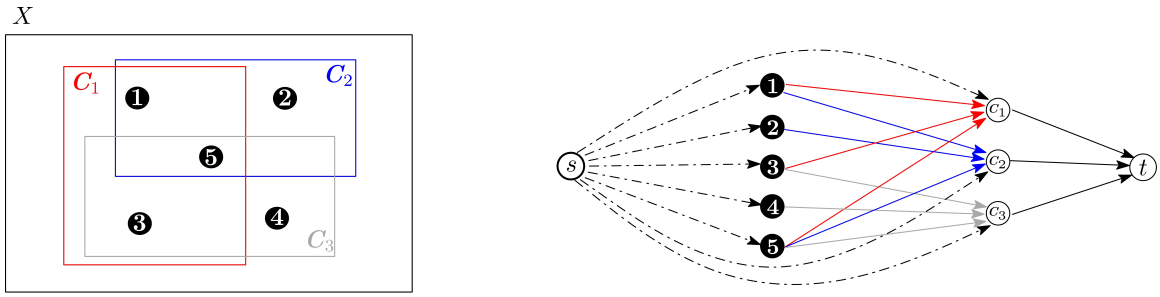
In this section, we prove that the MSE problem is NP-hard. The proof is by a reduction from the *Set Cover* problem. Given $\langle X, C, \ell \rangle$, where X is a finite set of elements, C is a collection of subsets of X, and ℓ is an integer, the decision version of Set Cover is defined as follows: is there a subset $C' \subseteq C$ with $|C'| \leqslant \ell$ such that the member elements of C' cover X?

Theorem 11. *The MSE problem is NP-complete.*

Proof. Given $\langle G, k, h \rangle$, where G is a graph with two distinguished nodes s and t, and $k, h \in \mathbb{N}$ are two numbers, we prove that the following decision version of MSE is NP-complete: is there a set P of k paths from s to t such that the number of edges shared among paths in P is at most h? It is easy to see that MSE is in NP. A certificate for this problem composed of k paths from s to t, and a certifier can then, in polynomial time, verify whether the number of shared edges is at most h.

We reduce Set Cover to MSE, by transforming each instance $\langle X, C, \ell \rangle$ of Set Cover to an instance $\langle G, k, h \rangle$ of MSE. The transformation is as follows. We first add to G the set of nodes $V = V_X \cup V_C \cup \{t\}$, where $V_X = \{v_x \,|\, x \in X\}$ and $V_C = \{v_{C_i} \,|\, C_i \in C\}$. We connect every node $v_x \in V_X$ to a node $v_{C_i} \in V_C$ by a directed edge if $x \in C_i$. Moreover, we connect every node $v_{C_i} \in V_C$ by a directed edge to t. Additionally, we add a node s to G and connect it to every node $v \in V_X \cup V_C$ using a path of length $\ell+1$. We call each of these paths a *chain*. Figure 46 illustrates our construction on a sample instance of Set Cover. We complete the transformation by setting $k = |X| + |C|$ and $h = \ell$.

Suppose that there is a set P of k (s,t)-paths in G with at most h shared edges. We show that there exists a collection $C' \subseteq C$ with $|C'| \leq \ell$ that covers X. It is easy to observe that each chain appears in at most one (s,t)-path, because otherwise more



(a) (b)

Figure 46: (a) An instance of the Set Cover problem, with a covering set $\{C_2, C_3\}$. (b) Reduction from Set Cover to MSE. Dashed lines represent chains of length $\ell + 1$.

than h $(= \ell)$ edges would be shared. Since the outdegree of s is equal to the number of paths, k, it follows that each chain is used exactly once, and thus, each vertex $v_x \in V_X$ appears in exactly one (s,t)-path. Therefore, only one outgoing edge from each $v_x \in V_X$ is used in P, and hence, shared edges are only among those incident to t. Now, let $V' = \{v \in V_C \,|\, (v,t) \text{ is a shared edge}\}$. Consider an (s,t)-path that goes through a node $v_x \in V_X$ and a node $v \in V_C$. We claim that $v \in V'$. Otherwise, node v is incident to two paths, one coming from v_x and the other coming from s via a chain, causing the edge (v,t) to be used in at least two paths; a contradiction. Therefore, in the induced subgraph $G[P]$, each node $v_x \in V_X$ is connected to a node $v \in V'$. The set $C' = \{C_i \,|\, v_{C_i} \in V'\}$ is thus a covering of X with $|C'| = \ell$.

Conversely, let $C' \subseteq C$ be a covering of X with $|C'| \leq \ell$. We show that in the corresponding graph G, there is a set P of k paths with at most h shared edges. Let $V' = \{v_{C_i} \in V_C \,|\, C_i \in C'\}$. For each $x \in X$, we define an (s,t)-path P_x as follows. We start from s and follow the chain to v_x. Since x is covered by a collection $C_i \in C'$, there is an edge (v_x, v_{C_i}) such that $v_{C_i} \in V'$. So, we use the edge (v_x, v_{C_i}) to reach from v_x to v_{C_i}, and then proceed to t. The set $P_X = \{P_x \,|\, x \in X\}$ consists of $|X|$ (s,t)-paths. Now, we define a set P_C of $|C|$ (s,t)-paths by concatenating, for each

$C_i \in C$, the chain from s to v_{C_i} and the edge (v_{C_i}, t). Let $P = P_X \cup P_C$. It is easy to observe that only edges between V_C and t can be used in more than one path of P. Since nodes in $V_C \setminus V'$ are not touched by the paths in P_X, each edge (v, t) for $v \in V_C \setminus V'$ is used exactly once in P, and hence, the number of shared edges in P is at most $|V'| = h$. □

6.3 Approximation Algorithm

In this section, we provide an approximation algorithm for the minimum shared edges problem by transforming it to a network flow problem, called "Minimum Edge-Cost Flow". The problem definition is as follows.

Problem 2 (Minimum Edge-Cost Flow (MECF))**.** *Given a graph $G = (V, E)$ with a capacity $u(e) \in \mathbb{Z}^+$ and a cost $c(e) \in \mathbb{Z}_0^+$ associated to each edge $e \in E$, find an integral flow f of value F from a source node s to a destination node t such that the total cost of edges sending non-zero flow, i.e., $\sum_{e \in E, f(e)>0} c(e)$, is minimized.*

It is known that the MECF problem is NP-hard [64]. Krumke *et al.* [98] have provided an F-approximation algorithm for the MECF problem. We use their technique to obtain a $(k-1)$-approximation algorithm for MSE. The following lemma provides the ingredient.

Lemma 3. *MSE can be reduced to MECF.*

Proof. We transform each instance of MSE on a graph $G = (V, E)$ to an instance of MECF on a graph $G' = (V', E')$. The transformation is as follows. We set $V' = V$, and for every edge $e \in E$, we add two edges e_1 and e_2 to E' with $u(e_1) = 1$, $c(e_1) = 0$, $u(e_2) = k - 1$ and $c(e_1) = 1$ (see Figure 47). Any solution of cost ℓ for MECF on G' corresponds to $k = F$ paths in G with ℓ shared edges. To see this, consider the

set of edges that have positive flow and cost 1 in a solution for MECF on G'. The corresponding edges in G are exactly those who are shared in a solution for MSE. Conversely, any solution of size ℓ for MSE on G corresponds to a solution of cost ℓ for MECF on G'. □

By Lemma 3, any α-approximation algorithm for MECF immediately gives an α-approximation for MSE. In the following, we provide an approximation algorithm for instances of MECF with maximum edge capacities $k-1$. Our algorithm is based on the solution for a well-known related problem, called Minimum-Cost Flow, defined as follows:

Problem 3 (Minimum-Cost Flow (MCF))**.** *Given a graph $G = (V, E)$ with a capacity $u(e) \in \mathbb{Z}^+$ and a cost $c(e) \in \mathbb{Z}_0^+$ associated to each edge $e \in E$, find an integral flow f of value F from a source node s to a destination node t such that $\sum_{e \in E} c(e)f(e)$ is minimized.*

Let $G' = (V', E')$ be a graph obtained from G using Lemma 3. We construct a graph G'' from G' by replacing the cost of each edge e in G'' by $c(e)/u(e)$ (the capacities of the edges remain the same). Let $\texttt{OPT}'$ be the cost of an optimal solution to the MECF problem on G', and $\texttt{OPT}''$ be the cost of an optimal solution to the MCF problem on G''. Let f be an integral flow that realizes an optimal cost $\texttt{OPT}''$ on G''.

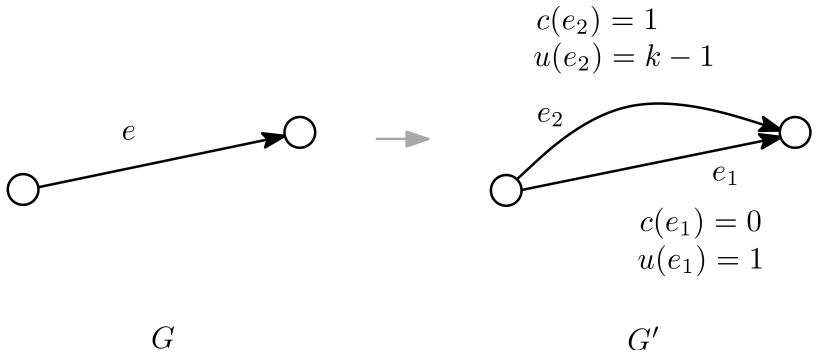


Figure 47: Transforming an edge in MSE to two edges in MECF.

Since capacities of the edges are the same in G' and G'', f is a valid flow in G' as well. Let $E^+ = \{e \in E' : f(e) > 0\}$. Then, using an argument similar to what is used in [98] we get

$$\begin{aligned} \texttt{OPT}' = \sum_{e \in E^+} c(e) &= \sum_{e \in E^+} u(e) \frac{c(e)}{u(e)} \\ &\leqslant (k-1) \sum_{e \in E^+} \frac{c(e)}{u(e)} \\ &\leqslant (k-1) \sum_{e \in E'} \frac{c(e)}{u(e)} f(e) \;=\; (k-1)\texttt{OPT}'', \end{aligned}$$

where the first inequality holds because the capacity of the edges are at most $k-1$ by our construction of G' in Lemma 3, and the second inequality holds because $f(e) \geqslant 1$ for all edges in E^+. Now, if $\texttt{OPT}$ is the cost of an optimal solution to MSE on G, combined with Lemma 3 we get

$$\texttt{OPT} = \texttt{OPT}' \leqslant (k-1)\texttt{OPT}''.$$

Therefore, any optimal algorithm for MCF yields a $(k-1)$-approximation algorithm for MSE. There are a number of efficient algorithms for the MCF problem. The best one for our setting is an algorithm due to Ahuja *et al.* [4] that runs in $O(nm \log(nC) \log\log U)$ time, where n, m, C, and U are the number of nodes, number of edges, maximum edge cost, and maximum edge capacity, respectively. Since in our transformation $C = 1$ and $U = k-1$, we get the following result.

Theorem 12. *There is a $(k-1)$-approximation algorithm for the MSE problem that runs in $O(nm \log n \log\log k)$ time.*

On series-parallel graphs, a fully polynomial time approximation scheme is given for the MECF problem in [98]. It leads to a $(1+\varepsilon)$-approximation algorithm for the

MSE problem on series-parallel graphs, with a running time of $O(m^3(1+1/\varepsilon)\log k)$.

Remark The MECF problem is listed in Garey and Johnson's book ([64], Problem [ND32]) as an NP-complete problem, leaving the proof to an unpublished work by Even and Johnson. As a by-product, Theorem 11 and Lemma 3 together provide a simple proof for the NP-completeness of MECF.

6.4 Inapproximability Result

In the previous section, we provided a $(k-1)$-approximation algorithm for the MSE problem. It is natural to ask if this is the best approximation factor one can achieve. In this section, we prove a lower bound on the approximability of the problem. The proof is based on the following theorem from [55] (here, n refers to the number of nodes in the input graph).

Theorem 13. (Even *et al.* [55]) *The MECF problem with uniform edge-costs does not admit a* $2^{\log^{1-\varepsilon} n}$*-ratio approximation, for any constant* $\varepsilon > 0$*, unless* $NP \subseteq \mathrm{DTIME}(n^{\mathrm{polylog}\, n})$*. This hardness holds even if only two edge capacity values are allowed, namely,* $u(e) \in \{1, \mathrm{poly}(n)\}$*, for every* e*.*

We establish an analogous hardness result for our problem, using an approximation preserving reduction from MECF with uniform edge-costs to MSE. The reduction is provided below.

Theorem 14. *The MSE problem admits no* $2^{\log^{1-\varepsilon} n}$*-ratio approximation, for any* $\varepsilon > 0$*, unless* $NP \subseteq \mathrm{DTIME}(n^{\mathrm{polylog}\, n})$*.*

Proof. Let $\mathcal{P}$ be the problem of finding an (s,t)-flow f of value F and cost C on a graph $G = (V, E)$ (see Problem 2). Each edge $e \in E$ is associated with an integer capacity

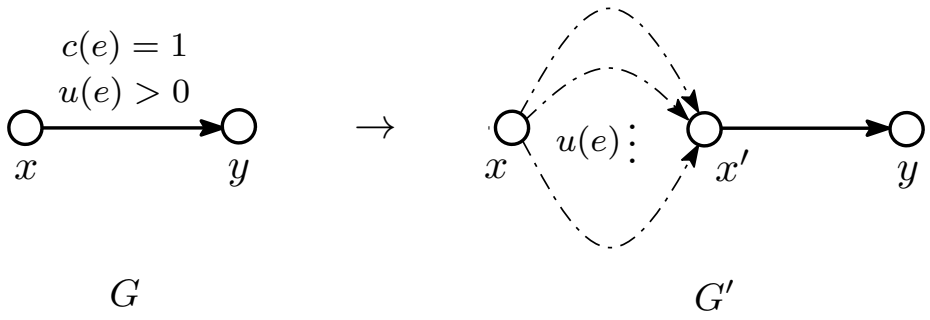


Figure 48: Conversion of an edge in MECF with uniform edge-costs to an edge component in MSE. Dashed lines represent chains of length $|E| + 1$.

$u(e) \in \{1, \text{poly}(n)\}$ and a uniform cost $c(e) = 1$. We construct a graph $G' = (V', E')$ from $G = (V, E)$ as follows. For each node $x \in V$ we insert a corresponding node x in V'. For each edge $e = (x, y) \in E$, we add an "edge component" between x and y in G', as depicted in Figure 48. Each edge component is composed of $u(e)$ parallel chains from x to a newly-added node x', and a directed edge from x' to y. Each chain is composed of $|E| + 1$ directed edges. We denote the edge component corresponding to an edge (x, y) by (x, x', y), and refer to chains connecting x to x' as Type-1 chains. Additionally, we add two nodes s' and t', and connect s' to s and t to t' with F chains. We call these chains Type-2 chains. Finally, we add for each edge component (x, x', y), a chain from s' to x' and a chain from y to t'. We refer to these chains as Type-3 chains. The resulting graph is illustrated in Figure 49.

Let $\mathcal{P}'$ be the problem of finding $k = |E| + F$ (s', t')-paths in $G' = (V', E')$ with $S \leqslant |E|$ shared edges. We show that solutions to $\mathcal{P}$ and $\mathcal{P}'$ are in one-to-one correspondence. First, we show that every solution to $\mathcal{P}'$ is a solution to $\mathcal{P}$. A solution to $\mathcal{P}'$ is a set P of k (s', t')-paths in G' with $S \leqslant |E|$ shared edges. Observe that none of the chains in G' can be on more than one path of P, otherwise, the number of shared edges would exceed $|E|$. Since the out-degree of s' in G' is $k = |E| + F$, each chain incident to s' must be on exactly one path of P. Similarly, each chain incident to t' is on exactly one path of P. Therefore, each edge (x', y) of an edge component

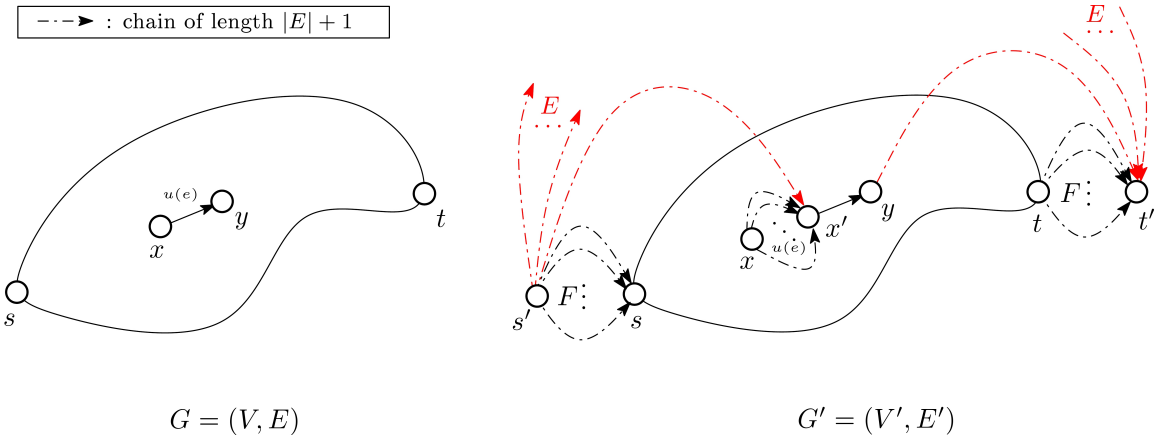


Figure 49: Reduction from MECF with uniform edge-costs to MSE.

(x, x', y) of G' is used in at least one path of P. Moreover, the only edges that can be shared among paths of P are these (x', y) edges.

Now, view P as a flow f' of value $|E| + F$ from s' to t'. We convert f' to a flow of value F in G as follows. First, for each path p of the form $s' \dashrightarrow x' \to y \dashrightarrow t'$, where $\dashrightarrow$ represents a chain, we remove a flow of value 1 along p from f'. After this step, f' has value F, and each Type-3 chain has flow zero. Thus, we can remove Type-3 chains from the graph. For each edge component (x, x', y) corresponding to an edge e of E, the remaining flow on (x', y) is at most $u(e)$. Since the whole flow of x' now comes from x and continues to y, we can contract Type-1 chains in between, and replace the edge component (x, x', y) by a single edge (x, y) of capacity $u(e)$, carrying the same amount of flow previously carried by (x', y). Similarly, we can contract Type-2 chains and merge s' to s and t' to t. The resulting graph is isomorphic to G, and the new flow f' corresponds to a feasible (s, t)-flow of value F in G. Observe that edges having positive flow in the new f' are exactly those edges having flow greater than 1 in the original f', and thus correspond to edges shared in P. As the cost of each corresponding edge in G is one, the total cost of flow f' is equal to the number of shared edges, namely $C = S$.

By reversing the above process, we can show that every solution to $\mathcal{P}$ is also a solution to $\mathcal{P}'$. Therefore, the one-to-one correspondence follows. The constructed graph G' has size $O(|V| + |E|(F + \sum_{e \in E} u(e)))$. Recall that $u(e) \in \{1, \text{poly}(n)\}$. Moreover, $F \leqslant |V|^2$ in the construction used in [55]. The reduction is thus polynomial, and the theorem statement follows. □

The lower bound proved in Theorem 14 is stated in terms of n. Since k is unbounded in the original definition of MSE, we cannot directly use the above theorem to get a lower bound in terms of k. The following lemma, however, enables us to bound the value of k, and get an analogous lower bound.

Lemma 4. *If $k > |E|$, then the minimum number of shared edges is equal to the size of the shortest (s, t)-path.*

Proof. It is easy to see that in any set of k paths, for $k > |E|$, there is a path in which all edges are shared. Because, otherwise, each path should have at least one edge distinct from other paths. For this to hold, each path requires more than $|E|$ edges which is impossible. □

Lemma 4 implies that the MSE problem is polynomially solvable on instances with $k > |E|$. Therefore, we can simply assume that $k \leqslant |E| = O(n^2)$. Theorem 14 thus implies a lower bound of $2^{\log^{1-\varepsilon} k}$ on the approximability of MSE.

6.5 Heuristic Improvements

In this section, we discuss some heuristics for improving the quality of the $(k-1)$-approximation algorithm described in Section 6.3. Experimental results from implementing the heuristics are also presented and compared.

6.5.1 Successive Cost Update

The approximation algorithm described in Section 6.3 is based on running a minimum-cost flow (MCF) algorithm, and returning the obtained flow as a $(k-1)$-approximation to MECF, which in turn, gives a $(k-1)$-approximation to MSE.

The MCF algorithm receives, as input, a transformed graph in which each edge has a cost whose value is in the set $\{0, 1/(k-1)\}$. When an edge of cost $1/(k-1)$ is used, the additive cost of selecting an edge, which is not sending any flow, is the same as that of an edge that is currently sending a positive flow. A positive flow on an edge of cost $1/(k-1)$ corresponds to a shared edge in the original graph G. Thus, it follows that in the $(k-1)$-approximation algorithm, there is no preference in reusing a previously shared edge over sharing a fresh edge. Our first heuristic attempts to force the approximation algorithm to reuse edges that previously have been used in the solution.

We implement this heuristic by exploiting an iterative cost update method. We want to encourage the MCF algorithm to reuse a previously shared edge. To achieve this, we select in each iteration an edge with maximum flow among edges that have positive cost, update the cost of that edge to zero, and re-run the MCF algorithm. It is easy to observe that this cost update operation does not affect the approximation factor of the algorithm. Details of the heuristic are provided in Algorithm 6.

6.5.2 Shortest Path Bound

The second heuristic is based on the fact that the minimum number of shared edges in the MSE problem is bounded from above by the number of edges in a shortest path from s to t. Let p be the size of a shortest (s,t)-path. If a feasible solution to MSE consisting of k (s,t)-paths uses more than p edges, we can reroute all k paths

Algorithm 6 MSE-APPROX(G, k, s, t)

1: construct G' from G using Lemma 3
2: obtain G_0 from G' by updating the cost of each edge e to $c(e)/u(e)$
3: compute a minimum-cost (s,t)-flow f of value k in G_0
4: set $i = 0$
5: **while** cost of $f \neq 0$ **do**
6: find an edge e with a maximum flow among positive-cost edges in G_i
7: obtain G_{i+1} from G_i by updating the cost of e to zero
8: compute a minimum-cost (s,t)-flow f of value k in G_{i+1}
9: $i \leftarrow i + 1$
10: **return** i

through a shortest path, and reduce the number of shared edges to p. We use this to modify Algorithm 6 and obtain the second heuristic as shown in Algorithm 7.

Algorithm 7 MSE-APPROX2(G, k, s, t)

1: let $r =$ MSE-APPROX(G, k, s, t)
2: let $p =$ size of a shortest (s,t)-path in G
3: **return** $\min(r, p)$

6.5.3 Random Heuristics

Uniform Random

In Algorithm 6, we select an edge with a maximum flow at each iteration, and update its cost to encourage the MCF algorithm to reuse that edge during the next round. One can easily see that this greedy choice might not be necessarily the best because a poor choice would negatively affect subsequent selections and increase the total number of shared edges. In our first random heuristic, we turn Algorithm 6 to a randomized one by changing line 6 as follows:

6: pick an edge e uniformly at random from the positive-cost edges in G_i

Weighted Random

In this heuristic, we use a weighted randomized approach for the edge selection criteria. Here, we assign to each edge e, a weight $w(e)$ equal to the number of times e is used in (s,t)-paths (i.e., the current value of $f(e)$ in Algorithm 6). Let W be the sum of the weights of all edges. As opposed to the first random heuristic, in each iteration of the algorithm, we pick an edge e at random with probability $w(e)/W$. Therefore, edges used in more (s,t)-paths are more likely to be selected.

Repeated Weighted Random

Finally, in the Repeated Weighted Random heuristic, we run the Weighted Random heuristic multiple times and report the minimum as the number of shared edges.

6.5.4 Experimental Results

We implemented the $(k-1)$-approximation algorithm described in Section 6.3 as well as the five heuristics described in this section. We evaluated our code on two families of graphs: road networks for large cities, and networks produced by benchmark graph generators. Figures 50 and 51 summarize the results of running our code on the two sample graphs: a road network for the city of Rome[1], and a random directed graph from DARPA HPCS SSCA#2 graph theory benchmark [14][2]. A SSCA#2 graph is a representative of computations in the field of national security, scientific computing, and computational biology. Both test graphs have 3350 nodes and 8870 edges.

The algorithms are run for $k = 1$ to 50, and the average number of shared edges are reported for 100 randomly-picked pairs of source and destination nodes. To force random pairs to be far enough, we discarded pairs of source and destination nodes

[1]The graph is available at: `http://www.dis.uniroma1.it/~challenge9/data/rome/rome99.gr`

[2]The generator is available at: `https://sdm.lbl.gov/~kamesh/software/GTgraph/`

that were less than $\sqrt{n}/4$ edges apart, for n being the number of nodes. We used the High Performance Computing Virtual Laboratory (HPCVL)'s Beowulf Cluster[3] that has 64 nodes of 4×2.2 GHz Opteron Cores with 8 GB RAM for running the experiments in parallel.

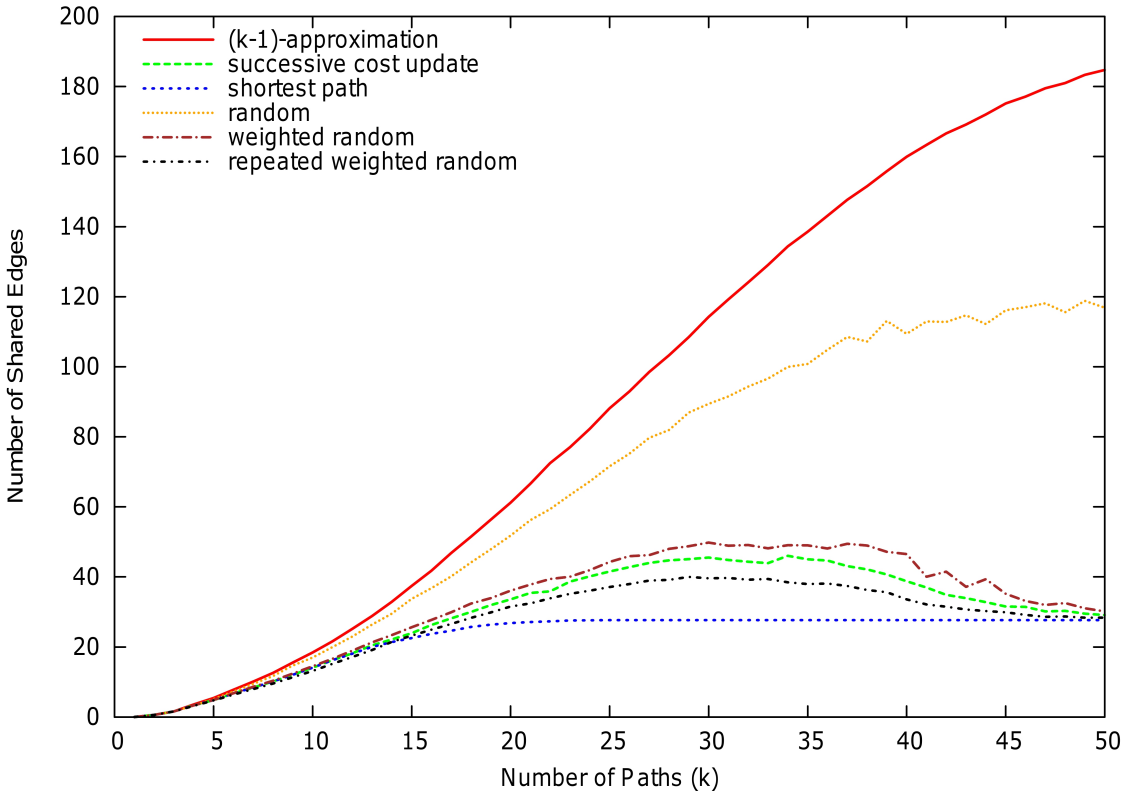
As can be seen in Figures 50 and 51, our heuristics perform significantly better when compared to the original $(k-1)$-approximation algorithm. (For the sake of a clear comparison, numerical data is provided for smaller values of k.) For large enough values of k, we get an improvement of 50% to 85% in the number of shared edges in these two graphs. The second heuristic performs better than the first one for some ranges of k. However, the two heuristics eventually converge for k sufficiently large. The reason for this convergence is that when the number of paths, k, is large, it is more likely for the edges on shortest paths to be shared in more paths, and thus, be selected by Algorithm 6 for cost update. For smaller values of k, the Repeated Weighted Random heuristic gives better results compared to other heuristics. The results shown here are for 10 repetitions. Better results can be obtained by increasing the number of repetitions.

6.6 Conclusions and Future Work

In this chapter, we studied the complexity of the minimum shared edges problem, and showed that the problem admits no $2^{\log^{1-\varepsilon} k}$-factor approximation, for any constant $\varepsilon > 0$. Moreover, we presented a $(k-1)$-approximation algorithm for the problem, and proposed some heuristics to improve it in practice. Heuristics presented in Section 6.5 (except the second one), can be indeed used as practical algorithms for the MECF problem.

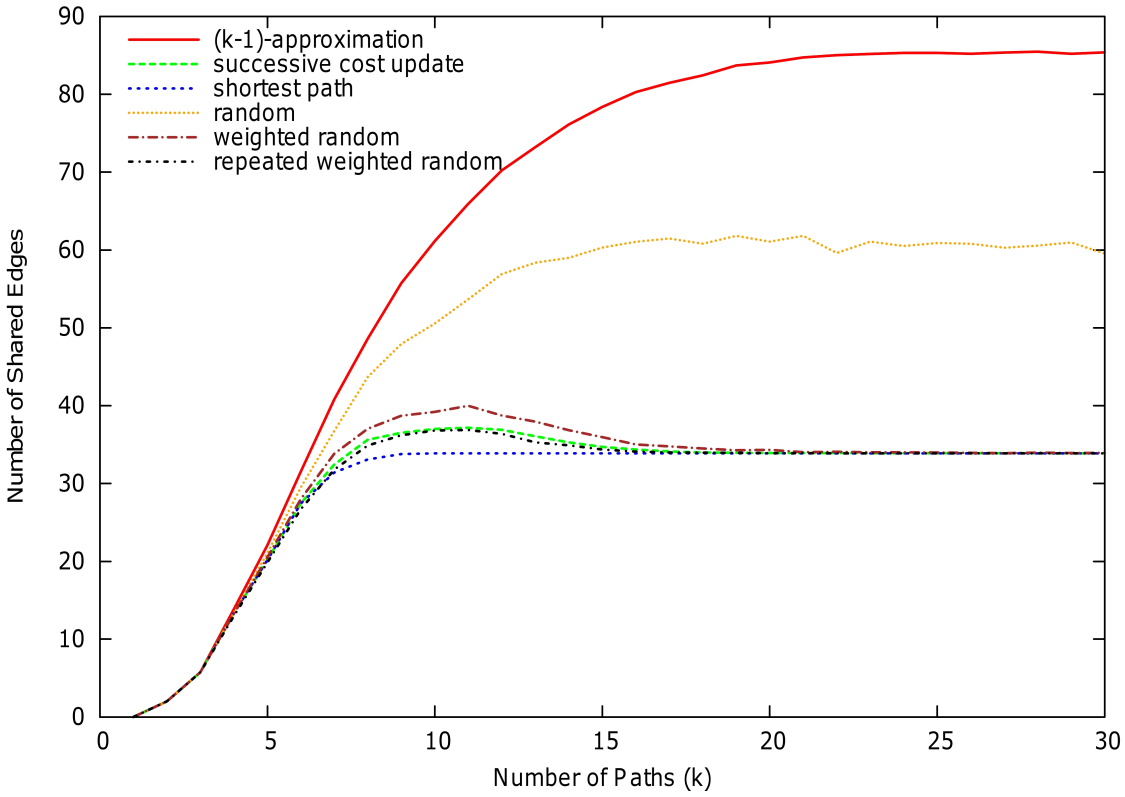
[3]The information is available at: `http://www.hpcvl.org/hpc-env-beowulf-cluster.html`

Whether or not the minimum shared edges problem is NP-complete, for small values of k (e.g., 3,4,...), remains a topic for future investigation. Another interesting open problem is to see if an algorithm with an approximation ratio better than $k-1$ exists for the minimum shared edges problem. Although our lower bound in Section 6.4 eliminates the possibility of having a poly-logarithmic approximation factor, we have not ruled out the possibility of having an approximation factor of $O(n^c)$, for a constant $c < 1$. (For example, see [28] for two variants of the Label Cover problem for which the same hardness of $2^{\log^{1-\varepsilon} n}$ holds, yet they admit a $O(n^{1/3})$-factor approximation.) Improving the lower bound on the approximability is another open problem. One could also study the harness of the minimum shared edges problem for other specific classes of graphs, e.g., series-parallel and planar graphs.



k	$(k-1)$-approx	successive cost update	shortest path	random	weighted random	repeated weighted random
1	0.0	0.0	0.0	0.0	0.0	0.0
2	0.6	0.6	0.6	0.6	0.6	0.6
3	1.6	1.6	1.6	1.6	1.6	1.6
4	3.57	3.31	3.31	3.28	3.31	3.19
5	5.44	4.81	4.81	5.1	4.88	4.73
6	7.77	6.72	6.72	7.24	6.77	6.35
7	10.11	8.43	8.43	9.44	8.59	7.95
8	12.61	10.16	10.16	11.7	10.34	9.57
9	15.5	12.05	12.05	14.66	12.37	11.32
10	18.43	14.09	14.09	16.97	14.5	13.14
11	21.58	16.21	16.21	19.94	16.62	15.19
12	25.1	18.17	18.04	22.95	18.92	17.11
13	28.79	20.47	20.01	26.37	21.3	19.1
14	32.88	22.12	21.39	29.48	23.4	21.45
15	37.39	23.93	22.55	33.75	25.61	23.22

Figure 50: Empirical results for the road network of Rome.



k	$(k-1)$-approx	successive cost update	shortest path	random	weighted random	repeated weighted random
1	0.0	0.0	0.0	0.0	0.0	0.0
2	2.01	2.01	2.01	2.01	2.01	2.01
3	5.75	5.75	5.75	5.75	5.75	5.75
4	13.83	13.3	13.3	13.34	13.25	12.97
5	22.08	20.18	20.18	21.07	20.54	19.85
6	31.55	27.45	27.44	29.5	28.0	26.71
7	40.82	32.43	31.42	36.77	33.83	31.79
8	48.61	35.63	33.08	43.69	37.03	34.88
9	55.75	36.51	33.78	47.94	38.7	36.2
10	61.14	37.01	33.88	50.56	39.2	36.81

Figure 51: Empirical results for a SSCA benchmark graph.

Chapter 7

Conclusions and Future Work

In this thesis, we first presented new algorithms for shortest path problems on two types of time-dependent networks with the FIFO property: networks where edges have time-dependent availability intervals ($\mathcal{TDSP}_{\text{int}}$), and networks where edges have piecewise linear arrival time functions ($\mathcal{TDSP}_{\text{lin}}$). We solved the $\mathcal{TDSP}_{\text{int}}$ problem by reducing it to a special case of the second problem, $\mathcal{TDSP}_{\text{lin}}$, for which we presented a novel solution based on new, non-trivial, combinatorial properties of arrival time functions. These new insights allow us to focus on finding the earliest arrival time function for the destination node only, and only at crucial time-points. We can in this way discard unnecessary computations. Our algorithms, in contrast to previous methods, directly compute the earliest arrival time function for every destination node d by tracing time and finding the earliest arrival time only at time instances that may change the earliest arrival time function for d. We presented algorithms that improve significantly upon the previously known methods for the $\mathcal{TDSP}_{\text{int}}$ and $\mathcal{TDSP}_{\text{lin}}$ problems.

Both of our methods make extensive use of static shortest paths algorithms. Other efficient static shortest path algorithms may be used, in special cases, for further improvement. For example, in planar networks, applying linear time shortest path

algorithms will further improve our results. In many practical networks, heuristics such as A^* could be applied to improve the practical performance of our methods.

We studied the approximation of the time-dependent shortest path problem ($\mathcal{TDSP}_{\text{lin}}$). We proposed two ϵ-approximation algorithms (referred to as Aprx–A and Aprx–B, respectively) both of which improve upon the existing approximation algorithm for $\mathcal{TDSP}_{\text{lin}}$ proposed by Foschini *et al.* [60]. The algorithms presented here use simple data structures, are easy to implement, and employ the modification of Dijkstra's algorithm studied in [45], as key building block.

In Chapter 3, we showed that within the same time complexity, for a given departure time at s, not only can we compute the minimum travel time to d, but it is also possible to obtain the slope of the function at that time instance. This can be used in Algorithm Aprx–B as a heuristic improvement, in order to reduce the number of computations required by using the function tangent instead of lines connecting consecutive sample points. It would not, however, change the worst-case time bound.

We presented a location privacy protection system for users that frequently submit their location information to LBS providers. We studied possible inference attacks by malicious parties and provided heuristic defense techniques to protect user privacy against such attacks. Our heuristic privacy-preserving algorithms provide users with flexible control parameters that allow them to customize the privacy level based on their needs. Our experimental evaluations show that, unless the parameters are set unrealistically, our algorithms provide quality results in less than few seconds.

In the future, we plan to extend our work to provide location privacy protection for users who continuously report their location to a malicious LBS provider. In a similar setting, we would also like to study how our results can be extended to the trajectory anonymization problem.

Finally, we studied the complexity of the minimum shared edges problem, and

showed that the problem admits no $2^{\log^{1-\varepsilon} k}$-factor approximation, for any constant $\varepsilon > 0$. Moreover, we presented a $(k-1)$-approximation algorithm for the problem, and proposed some heuristics to improve it in practice. Heuristics presented in Section 6.5 (except the second one), can indeed be used as a practical algorithm for the MECF problem.

List of References

[1] W. Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

[2] P. Agarwal, B. Aronov, J. O'Rourke, and C. Schevon. Star unfolding of a polytope with applications. In *SWAT 90*, volume 447 of *Lecture Notes in Computer Science*, pages 251–263. 1990.

[3] P. K. Agarwal, O. Schwarzkopf, and M. Sharir. The overlay of lower envelopes and its applications. *Discrete and Computational Geometry*, 15(1):1–13, 1996.

[4] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding minimum-cost flows by double scaling. *Mathematical Programming*, 53(1):243–266, 1992.

[5] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213–223, 1990.

[6] L. Aleksandrov, H. N. Djidjev, H. Guo, A. Maheshwari, D. Nussbaum, and J.-R. Sack. Algorithms for approximate shortest path queries on weighted polyhedral surfaces. *Discrete and Computational Geometry*, 44:762–801, 2010.

[7] L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Approximation algorithms for geometric shortest path problems. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 286–295, 2000.

[8] L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM*, 52:25–53, 2005.

[9] N. Alon, D. Moshkovitz, and S. Safra. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms (TALG)*, 2(2):153–177, 2006.

[10] C. Ardagna, M. Cremonini, S. De Capitani di Vimercati, and P. Samarati. An obfuscation-based approach for protecting location privacy. *IEEE Transactions on Dependable and Secure Computing*, 8(1):13–27, 2011.

[11] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1:49–63, 1986.

[12] T. Asano, D. Kirkpatrick, and C. Yap. Pseudo approximation algorithms with applications to optimal motion planning. *Discrete and Computational Geometry*, 31:139–171, 2004.

[13] F. Atlas, I. Cook, and C. Wenk. Shortest path problems on a polyhedral surface. *Algorithmica*, 69(1):58–77, 2014.

[14] D. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *High Performance Computing*, volume 3769 of *HiPC '05*, pages 465–476. 2005.

[15] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[16] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9):509–517, 1975.

[17] A. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2(1):46–55, 2003.

[18] A. Beshir and F. Kuipers. Variants of the min-sum link-disjoint paths problem. In *the 16th Annual Symposium on Communications and Vehicular Technology*, 2009.

[19] C. Bettini, S. Mascetti, X. S. Wang, D. Freni, and S. Jajodia. Privacy in location-based applications. chapter Anonymity and Historical-Anonymity in Location-Based Services, pages 1–30. 2009.

[20] C. Bettini, X. S. Wang, and S. Jajodia. Protecting privacy against location-based personal identification. In *Proceedings of 2nd VLDB Workshop on Secure Data Management*, pages 185–199, 2005.

[21] H. Breu and D. G. Kirkpatrick. Unit disk graph recognition is np-hard. *Computational Geometry*, 9(1):3–24, 1998.

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

[23] X. Cai, T. Kloks, and C. K. Wong. Time-varying shortest path problems with constraints. *Networks*, 29(3):141–150, 1997.

[24] J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 49–60, 1987.

[25] D. A. Castanon. Efficient algorithms for finding the k best paths through a trellis. *IEEE Transaction on Aerospace and Electronic Systems*, 26(2):405–410, 1990.

[26] I. Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Record: Journal of the Transportation Research Board*, 1645(1):170–175, 1998.

[27] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, pages 590–598, 2007.

[28] M. Charikar, M. Hajiaghayi, and H. Karloff. Improved approximation algorithms for label cover problems. In *Algorithms-ESA '09*, volume 5757, pages 23–34. 2009.

[29] B. Chazelle. A theorem on polygon cutting with applications. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 339–349, 1982.

[30] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991.

[31] D. Z. Chen. Developing algorithms and software for geometric path planning problems. *ACM Computing Surveys*, 28(4es), 1996.

[32] D. Z. Chen, O. Daescu, and K. S. Klenk. On geometric path query problems. In *Proceedings of the 5th International Workshop on Algorithms and Data Structures*, WADS '97, pages 248–257, 1997.

[33] X. Chen and J. Pang. Protecting query privacy in location-based services. *GeoInformatica*, 18:1–39, 2013.

[34] R. Cheng, Y. Zhang, E. Bertino, and S. Prabhakar. Preserving user location privacy in mobile data management infrastructures. In *Proceedings of the 6th Workshop on Privacy Enhancing Technologies*, volume 4258, pages 393–412, 2006.

[35] Y.-J. Chiang and J. S. B. Mitchell. Two-point Euclidean shortest path queries in the plane. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 215–224, 1999.

[36] J. Choi, J. Sellen, and C.-K. Yap. Approximate Euclidean shortest path in 3-space. In *Proceedings of the tenth annual symposium on Computational geometry*, SCG '94, pages 41–48, 1994.

[37] C.-Y. Chow and M. F. Mokbel. Enabling private continuous queries for revealed user locations. In *Proceedings of the 10^{th} international conference on Advances in spatial and temporal databases*, pages 258–273, 2007.

[38] C.-Y. Chow and M. F. Mokbel. Trajectory privacy in location-based services and data publication. *SIGKDD Exploration Newsletter*, 13:19–29, 2011.

[39] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[40] B. N. Clark, C. J. Colbourn, and D. S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86(1–3):165 – 177, 1990.

[41] K. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 56–65, 1987.

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

[43] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[44] C. D. Cottrill and P. Thakuriah. Consumer location privacy preferences: Survey analysis. *Transportation Research Board 91^{st} Annual Meeting*, 2012.

[45] C. F. Daganzo. Reversibility of the time-dependent shortest path problem. *Transportation Research Part B: Methodological*, 36(7):665–668, August 2002.

[46] B. C. Dean. Shortest paths in FIFO time-dependent networks: Theory and algorithms. Technical report, MIT Department of Computer Science, 2004.

[47] F. Dehne, M. T. Omran, and J.-R. Sack. Shortest paths in time-dependent FIFO networks using edge load forecasts. In *Proceedings of the Second International Workshop on Computational Transportation Science*, IWCTS '09, pages 1–6, 2009.

[48] F. Dehne, M. T. Omran, and J.-R. Sack. Shortest paths in time-dependent FIFO networks. *Algorithmica*, 62(1-2):416–435, 2012.

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

[50] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 205–216, 2008.

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

[52] M. Duckham and L. Kulik. A formal model of obfuscation and negotiation for location privacy. In *Proceedings of the Third international conference on Pervasive Computing*, volume 3468, pages 243–251, 2005.

[53] M. Duckham, L. Kulik, and A. Birtley. A spatiotemporal model of strategies and counter strategies for location privacy protection. In *Proceedings of the 4^{th} international conference on Geographic Information Science*, volume 4197, pages 47–64, 2006.

[54] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28:652–673, 1999.

[55] G. Even, G. Kortsarz, and W. Slany. On network design problems: fixed cost flows and the covering steiner problem. *ACM Transactions on Algorithms (TALG)*, 1(1):74–101, 2005.

[56] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[57] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345–, 1962.

[58] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[59] L. Foschini, J. Hershberger, and S. Suri. On the complexity of Time-Dependent shortest paths. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 327–341, 2011.

[60] L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, pages 1–23, 2012.

[61] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.

[62] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *Proceedings of the 32^{nd} annual symposium on Foundations of computer science*, SFCS '91, pages 632–641, 1991.

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

[64] M. Garey and D. S. Johnson. *Computers and intractability : A guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[65] B. Gedik and L. Liu. A customizable k-anonymity model for protecting location privacy. In *Proceedings of The 24^{th} International Conference on Distributed Computing Systems*, ICDCS '04, pages 620–629, 2004.

[66] B. Gedik and L. Liu. Protecting location privacy with personalized k-anonymity: Architecture and algorithms. *IEEE Transactions on Mobile Computing*, 7(1):1–18, 2008.

[67] G. Ghinita. Private queries and trajectory anonymization: a dual perspective on location privacy. *Transactions on Data Privacy*, 2:3–19, 2009.

[68] G. Ghinita. Privacy for location-based services. *Synthesis Lectures on Information Security, Privacy, and Trust*, 4(1):1–85, 2013.

[69] G. Ghinita, M. L. Damiani, C. Silvestri, and E. Bertino. Preventing velocity-based linkage attacks in location-aware applications. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 246–255, 2009.

[70] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K.-L. Tan. Private queries in location based services: anonymizers are not necessary. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 121–132, 2008.

[71] G. Ghinita, K. Zhao, D. Papadias, and P. Kalnis. A reciprocal framework for spatial K-anonymity. *Information Systems*, 35(3):299–314, 2010.

[72] S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing*, 20(5):888–910, 1991.

[73] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24:494–504, 1995.

[74] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[75] Z. Gotthilf and M. Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Information Processing Letters*, 109(7):352 – 355, 2009.

[76] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 31–42, 2003.

[77] M. Gruteser and B. Hoh. On the anonymity of periodic location samples. In *Proceedings of the Second International Conference on Security in Pervasive Computing*, pages 179–192, 2005.

[78] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39:126–152, 1989.

[79] H. Guo, A. Maheshwari, and J.-R. Sack. Shortest path queries in polygonal domains. In *Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, AAIM '08, pages 200–211, 2008.

[80] V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd, and M. Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In *Symposium on Theory of Computing*, STOC '99, pages 19–28, 1999.

[81] S. Har-Peled. Constructing approximate shortest path maps in three dimensions. In *Proceedings of the fourteenth annual symposium on Computational geometry*, SCG '98, pages 383–391, 1998.

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

[83] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[84] J. Hershberger. A new data structure for shortest path queries in a simple polygon. *Information Processing Letters*, 38:231–235, 1991.

[85] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28:2215–2256, 1999.

[86] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[87] J.-P. Hubaux, E. Kazemi, R. Shokri, G. Theodorakopoulos, P. Papadimitratos, et al. Hiding in the mobile crowd: Location privacy through collaboration. Technical report, 2013.

[88] A. Itai, Y. Perl, and Y. Shiloach. The complexity of finding maximum disjoint paths with length constraints. *Networks*, 12(3):277–286, 1982.

[89] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24:1–13, 1977.

[90] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 10, 2006.

[91] S. Kapoor, S. N. Maheshwari, and J. S. B. Mitchell. An efficient algorithm for Euclidean shortest paths among polygonal obstacles in the plane. *Discrete and Computational Geometry*, 18:377–383, 1997.

[92] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. 1972.

[93] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982.

[94] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *International Conference on Pervasive Services*, pages 88–97, 2005.

[95] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. *ACM Transactions on Algorithms*, 6:30:1–30:18, 2010.

[96] Y. Kobayashi and C. Sommer. On shortest disjoint paths in planar graphs. *Discrete Optimization*, 7(4):234–245, 2010.

[97] S. Kolliopoulos and C. Stein. Approximating disjoint-path problems using greedy algorithms and packing integer programs. In *Integer Programming and Combinatorial Optimization*, volume 1412 of *LNCS*, pages 153–168. Springer Berlin / Heidelberg, 1998.

[98] S. O. Krumke, H. Noltemeier, S. Schwarz, H.-C. Wirth, and R. Ravi. Flow improvement and network flows with fixed costs. In *Operations Research*, OR '98, pages 158–167, 1998.

[99] J. Krumm. Inference attacks on location tracks. In *Proceedings of the Fifth International Conference on Pervasive Computing (Pervasive), volume 4480 of LNCS*, pages 127–143, 2007.

[100] M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proceedings of the thirteenth annual symposium on Computational geometry*, SCG '97, pages 274–283, 1997.

[101] E. L. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[102] D.-T. Lee. *Proximity and reachability in the plane*. PhD thesis, 1978.

[103] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[104] S.-W. Lee and C.-S. Wu. A k-best paths algorithm for highly reliable communication networks. *IEICE Transaction on Communication*, E82-B(4):586–590, 1999.

[105] C. Li, S. T. McCormick, and D. Simchi-Levi. Finding disjoint paths with different path-costs: Complexity and algorithms. *Networks*, 22(7):653–667, 1992.

[106] C. Li, T. S. McCormick, and D. Simich-Levi. The complexity of finding two disjoint paths with min-max objective function. *Discrete Applied Mathematics*, 26(1):105–115, 1989.

[107] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM (JACM)*, 41(5):960–981, 1994.

[108] H. Mahmassani, T. Hu, and S. Peeta. Development and testing of dynamic traffic assignment and simulation procedures for ATIS/ATMS applications. Technical Report DTFH61-90-R-00074-FG, U.S. Department of Transportation, Federal Highway Administration, 1994.

[109] D. Marx. Eulerian disjoint paths problem in grid graphs is NP-complete. *Discrete Applied Mathematics*, 143(1-3):336–341, 2004.

[110] C. S. Mata and J. S. B. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions (extended abstract). In *Proceedings of the thirteenth annual symposium on Computational geometry*, SCG '97, pages 264–273, 1997.

[111] J. S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence*, 3:83–105, 1991.

[112] J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM*, 38:18–73, 1991.

[113] M. F. Mokbel. Towards privacy-aware location-based database servers. In *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ICDEW '06, pages 31–42, 2006.

[114] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new Casper: query processing for location services without compromising privacy. In *Proceedings of the 32nd international conference on Very Large Data Bases*, pages 763–774, 2006.

[115] S. Mozes and C. Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time. In *ESA (2)*, pages 206–217, 2010.

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

[117] S. D. Nikolopoulos, A. Pitsillides, and D. Tipper. Addressing network survivability issues by finding the k-best paths through a trellis graph. In *Proceeding of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE.*

[118] T. Nishizeki and N. Chiba. *Planar graphs: Theory and algorithms.* Elsevier, 1988.

[119] T. Nishizeki, J. Vygen, and X. Zhou. The edge-disjoint path problem is NP-complete for series-parallel graphs. *Discrete Applied Mathematics*, 115(1-3):177–186, 2001.

[120] D. Nussbaum, M. T. Omran, and J.-R. Sack. Techniques to protect privacy against inference attacks in location based services. In *Proceedings of the Third ACM SIGSPATIAL International Workshop on GeoStreaming*, IWGS '12, pages 58–67, 2012.

[121] M. Omran, J.-R. Sack, and H. Zarrabi-Zadeh. Finding paths with minimum shared edges. *Journal of Combinatorial Optimization*, 26(4):709–722, 2013.

[122] M. T. Omran and J.-R. Sack. Improved approximation for time-dependent shortest paths. *Manuscript accepted for presentation at COCOON'14*, 2014.

[123] M. T. Omran, J.-R. Sack, and H. Zarrabi-Zadeh. Finding paths with minimum shared edges. In *Proceedings of the 17th Annual International Conference on Computing and Combinatorics*, COCOON '11, pages 567–578, 2011.

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

[125] A. Orda and R. Rom. Minimum weight paths in time-dependent networks. *Networks*, 21:295–319, 1991.

[126] X. Pan, J. Xu, and X. Meng. Protecting location privacy against location-dependent attacks in mobile services. *IEEE Transactions on Knowledge and Data Engineering*, 24(8):1506–1519, 2012.

[127] C. H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. *Information Processing Letters*, 20(5):259–263, 1985.

[128] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312:47–74, 2004.

[129] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM Journal on Computing*, 34:1398–1431, 2005.

[130] M. Pocchiola and G. Vegter. Computing the visibility graph via pseudo-triangulations. In *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, pages 248–257, 1995.

[131] S. Rivière. Topologically sweeping the visibility complex of polygonal scenes. In *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, pages 436–437, 1995.

[132] Y. Schreiber and M. Sharir. An optimal-time algorithm for shortest paths on a convex polytope in three dimensions. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, pages 30–39, 2006.

[133] M. Sharir and P. K. Agarwal. *Davenport-Schinzel sequences and their geometric applications*. Cambridge University Press, New York, NY, USA, 1996.

[134] H. D. Sherali, K. Ozbay, and S. Subramanian. The time-dependent shortest pair of disjoint paths problem: Complexity, models, and algorithms. *Networks*, 31(4):259–272, 1998.

[135] K. Shin, X. Ju, Z. Chen, and X. Hu. Privacy protection for users of location-based services. *IEEE Wireless Communications*, 19(1):30–39, 2012.

[136] J. W. Suurballe. Disjoint paths in a network. *Networks*, 4(2):125–145, 1974.

[137] J. W. Suurballe and R. E. Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984.

[138] L. Sweeney. k-anonymity: a model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[139] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[140] J. Y. Tsai, P. G. Kelley, L. F. Cranor, and N. Sadeh. Location-sharing technologies: Privacy risks and controls. In *Research Conference on Communication, Information and Internet Policy*, 2009.

[141] J. Vygen. NP-completeness of some edge-disjoint paths problems. *Discrete Applied Mathematics*, 61(1):83–90, 1995.

[142] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9:11–12, 1962.

[143] E. Welzl. Constructing the visibility graph for n-line segments in $O(n^2)$ time. *Information Processing Letters*, 20(4):167–171, 1985.

[144] S. B. Wicker. The loss of location privacy in the cellular age. *Communication of the ACM*, 55(8):60–68, 2012.

[145] C. Wulff-Nilsen. Wiener index and diameter of a planar graph in subquadratic time. In *European Workshop on Computational Geometry*, pages 25–28, 2009.

[146] C. Wulff-Nilsen. Solving the replacement paths problem for planar directed graphs in $O(n \log n)$ time. In *proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 756–765, 2010.

[147] D. Xu, Y. Chen, Y. Xiong, C. Qiao, and X. He. On the complexity of and algorithms for finding the shortest path with a disjoint counterpart. *IEEE/ACM Transaction on Networking*, 14(1):147–158, 2006.

[148] J. Xu, X. Tang, H. Hu, and J. Du. Privacy-conscious location-based queries in mobile environments. *IEEE Transaction on Parallel and Distributed Systems*, 21:313–326, 2010.

[149] B. Yang, M. Yang, J. Wang, and S. Q. Zheng. Minimum cost paths subject to minimum vulnerability for reliable communications. In *International Symposium on Parallel Architectures, Algorithms and Networks*, ISPAN '05, pages 334–339, 2005.

[150] B. Yang, S. Q. Zheng, and E. Lu. Finding two disjoint paths in a network with minsum-minmin objective function. In *Proceedings of the 2007 International Conference on Foundations of Computer Science*, FCS '07, pages 356–361, 2007.

[151] J. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general network. *Quarterly of Applied Mathematics*, 27:526–530, 1970.

[152] L. Zhao, T. Ohshima, and H. Nagamochi. A* algorithm for the time-dependent shortest path problem. pages 36–43, 2008.

[153] S. Zheng, J. Wang, B. Yang, and M. Yang. Minimum-cost multiple paths subject to minimum link and node sharing in a network. *IEEE/ACM Transaction on Networking*, 18(5):1436–1449, 2010.

[154] S. Q. Zheng, B. Yang, M. Yang, and J. Wang. Finding minimum-cost paths with minimum sharability. In *Proceeding of the 26th IEEE International Conference on Computer Communications. IEEE*, INFOCOM '07, pages 1532–1540, 2007.

[155] G. Zhong, I. Goldberg, and U. Hengartner. Louis, Lester and Pierre: three protocols for location privacy. In *Proceedings of the 7th international conference on Privacy enhancing technologies*, pages 62–76, 2007.

[156] A. Ziliaskopoulos and H. Mahmassani. Time-dependent, shortest-path algorithm for real-time intelligent vehicle highway system applications. In *Transportation Research Record 1408*, pages 94–100, 1993.