EFFECTIVE CACHING OF PATHS AND REGIONS FOR MOBILE SERVICES

JEPPE RISHEDE THOMSEN

Ph.D.

The Hong Kong Polytechnic University

2015

The Hong Kong Polytechnic University

Department of Computing

Effective Caching of Paths and Regions for Mobile Services

by

Jeppe Rishede Thomsen

A thesis submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

July 2014

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

....

Jeppe Rishede Thomsen

July 2014

Abstract

Cheap GPS-enabled mobile devices have made online location based services (LBSs) increasingly common. Examples of LBS are: *route planner, navigation assistance, restaurant recommendation, meeting point recommendation*, and *nearby friend notification.* In location based services, caching can be used to reduce computational load or communication latency for queries. In this thesis, we consider three problems on caching paths and regions for mobile services.

Our first problem is to investigate caching techniques for shortest paths. However, existing caching techniques do not exploit the optimal substructure property of shortest paths. We propose a novel caching problem for shortest paths, and develop algorithms and data structures for this problem.

Our second problem is to explore trade-offs between the lengths and the query coverage of concise shortest paths, in online driving direction services. Driving instructions are equivalent to so-called concise shortest paths, which occupy much less space than the corresponding shortest paths. The caching of concise shortest paths has two opposite effects on the cache hit ratio. The cache can accommodate a larger number of concise paths, but each individual concise path contains fewer nodes and so may answer fewer shortest path queries. We formulate the concept of a generic shortest path enabling trade-off between a paths size and the number of queries it can answer. We present algorithms for computing and caching generic concise shortest paths.

Our third problem is to compute safe regions for the sum-optimal meeting point notification problem. It has applications like social networking services or online games, in which multiple moving users in a group may wish to be continuously notified about the best meeting point from their locations. To reduce the communication frequency, an application server can compute safe regions, which capture the validity of query results with respect to users locations. Unfortunately, the safe regions in our problem exhibit characteristics such as irregular shapes and inter-dependencies, which render existing methods for single safe region inapplicable. We present algorithms for computing safe regions for sum-optimal meeting point.

In summary, we make the following contributions: (i) algorithms and data structures for caching of shortest paths; (ii) algorithms for computing generic concise shortest paths; (iii) algorithms for computing the safe regions for the sum-optimal point notification problem.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Man Lung Yiu, for providing me with the amazing opportunity to do my Ph.D. under his supervision. His valuable advice, guidance and support during the past few years has made me a better researcher and given me the practical skills necessary to succeed in my future career.

I am thankful to the talented researchers I have met and discussed with during my studies, especially the members of my research group, they have no doubt helped me grow as a researcher. A special thanks goes to Prof. Christian S. Jensen for his help and support throughout my studies; his comments and critique of my papers have been invaluable.

I want to thank the friends I have made at HKPU during my studies. A special thanks goes to Jianguo Wang for his friendship. He was hard to get to know, but one of the kindest people I've met. I would also like to thank Hill Yu, Jiwei Li, and Tony Tan for their sincere friendship and great company.

A special thanks goes out to my family, My Mom, Dad, and my Sister, without whom I would not be where I am today. Their support and encouragement during both the good and the bad times have ensured that I always kept trying

and succeeding.

Contents

List of Figures

Chapter 1

Introduction

Online location based services (LBSs) provide mobile users with services based on their own or friends' GPS locations [1, 2]. LBSs can be divided into three categories based on the type of service they provide:

- Services which do not require the user's current location. E.g., answering a shortest path query from location S to T, or providing the option to downloading a section of a map for offline usage.

- Service depending on the user's current location. E.g., driving instructions while navigating towards a target location, or nearby restaurant recommendation.

- Services depending on the current location of a group of users. E.g., Offering to notify a user about friends nearby, or assisting users in finding a suitable location to meet.

All major location based service providers use the client-server architecture [3–6]. An online location based service works by first receiving a query with GPS coordinates from a user, and second, returning an answer based on the service requested.

A popular application of the client-server architecture is web search, where a search engine is queried for a set of relevant web sites. A *cache* stores the results of frequent queries so similar queries can be answered quickly, bypassing the need to re-calculate the results. This improves both the query latency and lowers the computation time [7–10]. Figure 1.1 shows the cache can be located at 3 places, either a server, a proxy, or at the user. The cache is placed at the server primarily to save computation time, or at a proxy in a local network to reduce query latency and response time.

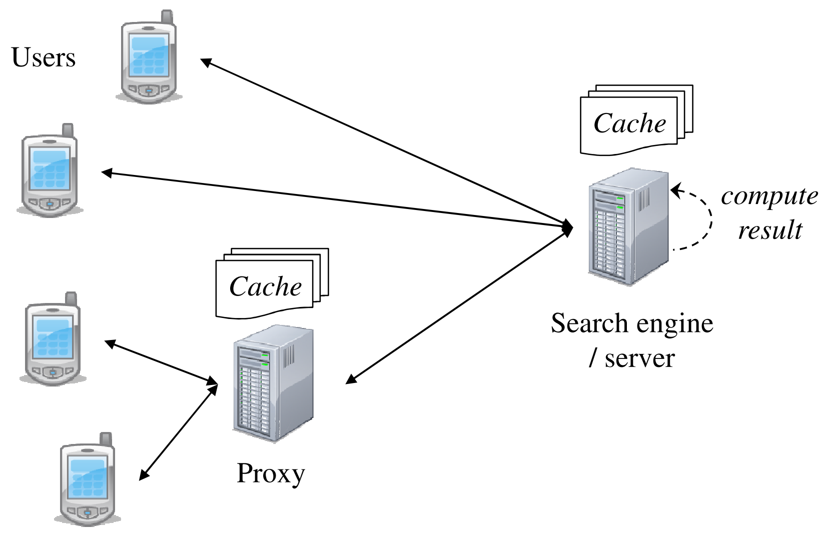


Figure 1.1. Scenario for web search

The cache can be placed at the mobile user to reduce the communication with the server. There are two categories of services that benefit from a cache at the user. (i) LBS that continuously monitor users' locations are query intensive, for these services users benefit from locally caching when they need to resend their current location. (ii) Services that return a result set limited by a geographical region. The users can save communication by sending a smaller query in case a partial result has already been retrieved from a previous query.

A cache placed at the users devices is almost always a semantic cache, meaning the cache is aware of the semantics of the cache items, enabling the cache to return partial answers (so a smaller query would be sent to the server), or allowing cache items to answer multiple types of queries.

In this thesis, we investigate novel caching problems and introduce a new type of caching not previously investigated. Specifically, we study the following interesting caching problems: (i) caching of shortest paths, (ii) calculation and caching of generalized concise paths, and (iii) computing of safe regions for notification of meeting points.

Shortest path caching is an important tool for online location based services, to support the growing number of mobile users who are always online. Shortest path caching is challenging as it differs from traditional caching, such as web caching, in a number of important aspects. (i) The sizes of shortest paths differ, the cache can store few long shortest paths, or many shorter ones. (ii) A long shortest path takes longer to compute than a shorter one, so the computational savings of a longer cache item becomes important. (iii) A shortest path exhibit the optimal subpath property (Lemma 3.1), meaning all paths on a shortest path is also a shortest path. This lends more value to longer shortest paths in the cache, as they can answer more queries.

Consider the example road network in Figure 1.2. The shortest path from v_1 to v_6 contains 5 vertices, while the one from v_2 to v_4 contains only 3 vertices. However, (v_1, v_6) can answer 10 queries, while (v_2, v_4) can only answer 3. Even though (v_2, v_4) takes up less space, (v_1, v_6) can answer more queries per vertex if added to the cache.

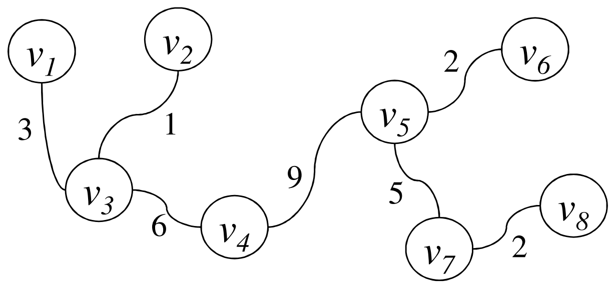


Figure 1.2. An example road network

A generalized concise path is a shortest path where some vertices in the path has been removed. Caching generalized concise paths are important for online location based service providers to optimize the utilization of the shortest path cache. The idea works by first identifying the smallest set of vertices which still allow for the original path to be traversed, then use query logs to identify historically important queries and add the vertices needed to answer these queries back into the path. The point is to omit the vertices which have historically not been useful, from the paths in the cache.

Consider the example road network in Figure 1.2. If we originally had a shortest path (v_1, v_8) and we had seen the query (v_3, v_5) at least once, but never the query (v_4, v_7), then it is a waste of space to store the full path in the cache, as we don't expect v_4 or v_7 to contribute. We suggest not storing the full path, but $\langle v_1, v_3, v_5, v_8 \rangle$ in the cache, which takes up less space.

The Sum-Optimal Meeting point notification problem helps a group of users to dynamically find the optimal meeting point for the whole group among a set of predefined options to minimize the total sum of distances traveled by all the group members. Safe regions for sum-optimal meeting points notification reduces the need for members of a group to constantly send a location based service their location. As an example of Sum-Optimal Meeting point notification, consider the group of users $\{U_1, U_2, U_3\}$ in Figure 1.3. At time T_1 the group wants to meet at a restaurant R_1 or R_2. The LBS recommends restaurant R_1 at time T_1 as it has the shortest total travel distance for the group members. At time T_2 user U_2 and U_3 have traveled much further than U_1, so when the LBS recalculates the sum-optimal meeting point, the recommended restaurant becomes R_2, as it is now the meeting point with the shortest total travel distance for the group.

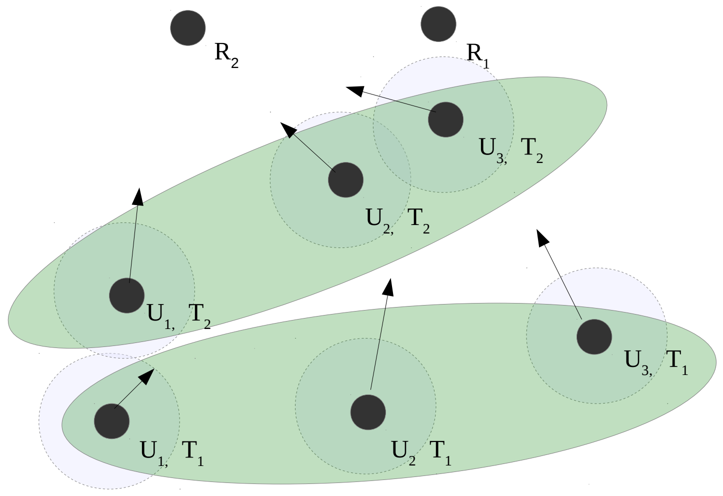


Figure 1.3. Finding sum-optimal meeting point

The main contribution of this thesis can be summarized as follows:

- We develop algorithms and data structures for caching of shortest paths.
- We develop algorithms for computing generic concise shortest paths.
- We present algorithms for computing the safe regions for the sum-optimal point notification problem.

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the existing work related to caching and shortest path computation.

Chapter 3 (based on [11]) studies caching of shortest paths. The problem is studied for both server and proxy caching. A model to capture the benefit of caching a shortest path is introduced. We propose a greedy algorithm to select beneficial paths for the cache, based on historical query information. We introduce effective structures to speed up cache look up and increase the cache space utilization. Finally, we do extensive performance evaluations to evaluate the cache performance.

Chapter 4 (based on [12]) studies how to increase the cache hit ratio by caching partial shortest paths, which still hold enough information to navigate the original shortest paths. We present the concept of a concise path. We introduce the idea of a generic concise path which addresses the trade-off between the path length and the number of queries a path can answer. We develop caching schemes to support generic shortest paths, and perform extensive experimental evaluations on both real and synthetic datasets

Chapter 5 (based on [13][1]) studies how to compute safe regions for the sum-optimal meeting point, where the sum of all users travel distance is minimized, for a group of users. The focus is on minimizing the communication frequency of the user. Efficient algorithms and optimizations are developed to efficiently calculate *independent safe regions* based on the concept of a sum-optimal meeting point.

Chapter 6 concludes the thesis.

[1]We contribute to Sections 5.4, 6, and 7.3 from [13]

Chapter 2

Literature Review

Caching continues to be one of the most important topics in computer science. Caching materializes the results of the most popular queries submitted to a server and transparently stores the data to answer future queries faster. A query resulting in a cache hit will negate the computing cost associated with calculating the result. Table 2.1 indicates the relevance between the main chapters (3, 4, 5) and the topics in this literature review.

Chapter	**Chapter 3**	**Chapter 4**	**Chapter 5**
Web Search Caching	✓	✓	
Shortest Path Computation	✓	✓	
Semantic Caching	✓	✓	✓

Table 2.1. Related work shared by chapter 3, 4, or 5

2.1 Web Search Caching

A web search cache stores titles and text snippets of each result associated with a search query in a search engine, such as Google, Bing, or Baidu.

A web search query asks for the top-K relevant documents (i.e., web pages) that best match a text query, e.g., "Paris Eiffel Tower." The typical value of K is the number of results (e.g., 10) to be displayed on a result page [10]; a request for the next result page is interpreted as an unrelated query.

The cache can be placed at the search engine to save computation time, or at a proxy in a sub-network to improve latency or response time (see Figure 1.1).

Web search caching is used to improve the performance of a search engine. When a query can be answered by the cache, the cost of computing the query result can be saved. Markatos et al. [14] present pioneering work, evaluating two caching approaches (dynamic caching and static caching) on real query logs. *Dynamic caching* [14–16] aims to cache the results of the most recently accessed queries. For example, in the Least-Recently-Used (LRU) method, when a query causes a cache miss, the least recently used result in the cache is replaced by the current query result. This approach keeps the cache up-to-date and adapts quickly to the distribution of the incoming queries; however, it incurs the overhead of updating the cache frequently.

On the other hand, *static caching* [7–10,14,17] aims to cache the results of the most popular queries. This approach exploits a query log that contains queries issued in the past in order to determine the most frequent queries. The above studies have shown that the frequency of queries follows Zipfian distribution, i.e., a small number of queries have very high frequency, and they remain popular for a period of time. Although the cache content is not the most up-to-date, it is able to answer the majority of frequent queries. A static cache can be updated periodically (e.g., daily) based on the latest query log. Static caching has the advantage that it incurs very low overhead at query time.

Early work on web search caching adopt the *cache hit ratio* as the performance metric. This metric reflects the number of queries that do not require computation cost. Recent work [9,10] on web search caching uses the *server processing time* as the performance metric. The motivation is that different queries have different query processing times, e.g., a query involving terms with large posting lists incurs a high processing cost. Thus, the actual processing time of each query in the query log is taken into account, and both frequency and cost information are exploited in the proposed static caching methods.

None of the above works consider the caching of shortest paths. In this work, we adopt both static and dynamic caching approaches as the best performance depends on which of our methods are used. Earlier techniques [9,10] are specific to web search queries and are inapplicable to our problem. In our caching problem, different shortest path queries also have different processing times. Thus, we also propose a cost-oriented model for quantifying the benefit of placing a path in the cache.

2.2 Semantic Caching

Semantic caching is a caching model which associates cached results with valid ranges. Semantic caching lets a cache answer more than one query per cache item. It also allows for a cache item to partially answering a query.

In a client-server system, a cache may be employed at the client-side in order to reduce the communication cost and improve query response time. A cache located at a client can only serve queries from the client itself, not from other clients. Such a cache is only beneficial for a query-intensive user. All techniques in this category adopt the dynamic caching approach.

Semantic caching [18] is a client-side caching model that associates cached results with valid ranges. Upon receiving a query Q, the relevant results in the cache are reported. A subquery Q' is constructed from Q such that Q' covers the query region that cannot be answered by the cache. The subquery Q' is then forwarded to the server in order to obtain the missing results of Q. Dar et al. [18] focus on semantic caching of relational datasets. As an example, assume that the dataset stores the age of each employee and that the cache contains the result of the query "find employees with age below 30." Now assume that the client issues a query Q "find employees with age between 20 and 40." First, the employees with age between 20 and 30 can be obtained from the cache. Then, a subquery Q' "find employees with age between 30 and 40" is submitted to the server for retrieving the remaining results.

Semantic caching has also been studied for spatial data [19–21]. Zheng et al. [19] define the semantic region of a spatial object as its Voronoi cell, which can be used to answer nearest neighbor queries for a moving client user. Hu et al. [20] study semantic caching of tree nodes in an R-tree and examine how to process spatial queries on the cached tree nodes. Lee et al. [21] build generic semantic regions for spatial objects so that they support generic spatial queries.

Semantic regions are semantically equal to safe regions, in that they are both polygons with the purpose of minimizing the number or range of spatial queries. The safe region concept has been widely used in moving query processing to reduce the communication cost between clients and servers. When a user registers a continuous query, the server will return POIs along with a safe region. The query result remains the same if the user stays inside the current safe region. Upon leaving the safe region, the user requests from the server a updated result together with a new safe region. The shape of the safe region depends on the query type, e.g., an order-k Voronoi cell for a kNN query [22], or an arc-based region for a range query [23]. Figure 2.1 illustrates the use of safe regions. The user wants to travel from v_1 to v_8 and be notified which road to take at each intersection. At v_1 the service returns safe region $R1$ so the user know that no location update needs to be sent until outside the boundary of $R1$. When arriving at v_3 the user is outside $R1$. After submitting the current location, the LBS returns the direction to take and $R2$. Finally, at v_5 the user sends the final location update and is returned the correct direction plus $R3$. Using safe regions the user only had to send 3 updates to get this, in stead of sending the current location continuously.

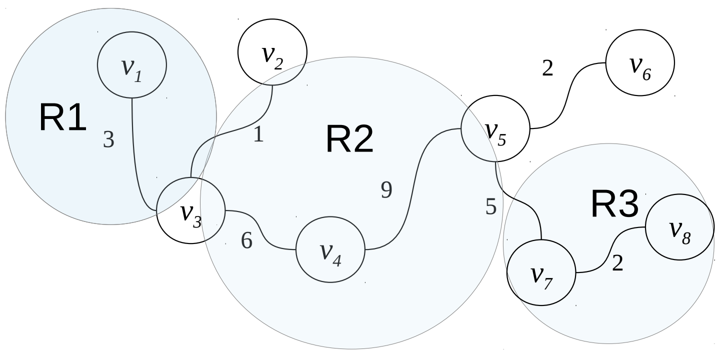


Figure 2.1. An example road network

2.3 Shortest Path Computation

Shortest path computation takes a shortest path query from a source node S to a target node T, on a weighted edge graph. It finds the path with the minimal sum on all the edges from S to T.

Existing shortest path indexes can be categorized into three types, which represent different trade-offs between their precomputation effort and query performance.

A basic structure is the adjacency list, in which each node v_i is assigned a list that stores the adjacent nodes of v_i. It does not store any pre-computed information. Uninformed search (e.g., Dijkstra's algorithm, bidirectional search) can be used to compute the shortest path; however, it incurs high query cost.

Partially-precomputed indexes, e.g., landmarks [24], HiTi [25], and TEDI [26], attempt to materialize some distances/paths in order to accelerate the processing of shortest path queries. They employ certain parameters to control the trade-offs among query performance, precomputation overhead, and storage space.

Fully-precomputed indexes, e.g., the distance index [27] or the shortest path quadtree [28], require precomputation of the shortest paths between any two nodes in the graph. Although they support efficient querying, they incur huge precomputation time ($O(|V|^3)$) and storage space ($O(|V|^2)$), where $|V|$ is the number of nodes in the graph.

Chapter 3

Caching of Shortest Paths

The world's population issues vast quantities of web search queries. A typical scenario for web search is illustrated in Figure 3.1. A user submits a query, e.g., "Paris Eiffel Tower," to the search engine, which then computes relevant results and returns them to the user. A *cache* stores the results of frequent queries so that queries can be answered frequently by using only the cache, thus reducing the amount of computation needed and improving query latency [7–10].

Specifically, the cache can be placed at the search engine to save its computation time, e.g., when the query (result) can be found in the cache. Or, to improve latency or response time, the cache can be placed at a proxy that resides in the same sub-network as the user. A query result that is available at the proxy can be reported immediately, without contacting the search engine.

The scenario in Figure 3.1 is applicable to *online shortest path search*, also called directions querying. Due to the increasingly mobile use of the web and advances in geo-positioning technologies, this has become a popular type of web query. This type of query enables users to, e.g., obtain directions to a museum, a gas station, or a specific shop or restaurant.

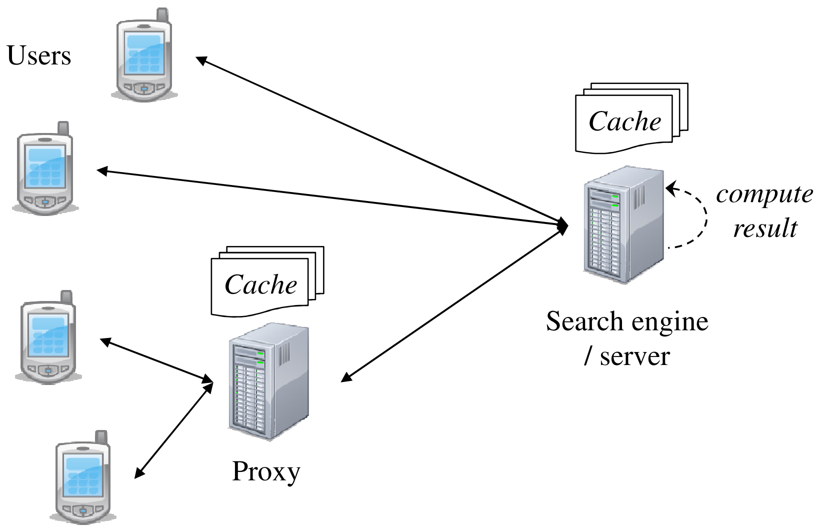


Figure 3.1. Scenario for web search

When compared to offline commercial navigation software, online shortest path search (e.g., Google Maps, MapQuest) provide several benefits to mobile users: (i) They are available free of charge. (ii) They do not require any installation and storage space on mobile devices. (iii) They do not require the purchase and installation of up-to-date map data on mobile devices.

Figure 3.2 shows a road network in which a node v_i is a road junction and an edge (v_i, v_j) models a road segment with its distance shown as a number. The shortest path from v_1 to v_7 is the path $\langle v_1, v_3, v_4, v_5, v_7\rangle$ and its path distance is $3 + 6 + 9 + 5 = 23$. Again, caching can be utilized at a proxy to reduce the response time, and it can also be used at the server to reduce the server-side computation.

We study the caching of path search results in the scenario shown in Fig-

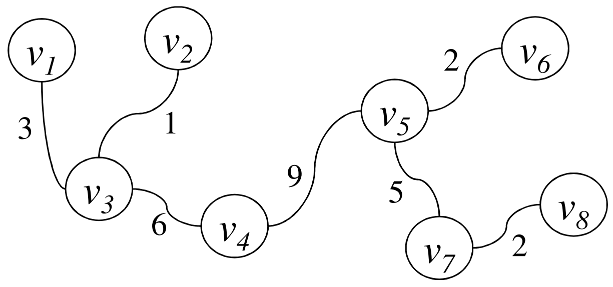


Figure 3.2. An example road network

ure 3.1. While shortest path search shares this scenario with web search, there are also crucial differences between general web search and shortest path search, rendering existing caching techniques for web results ineffective in our context.

- **Exact matching vs. subpath matching:** The result of a web query (e.g., “Paris Eiffel Tower”) seldom matches with that of another query (e.g., “Paris Louvre Palace”). In contrast, a shortest path result contains subpaths that can be used for answering other queries. For example, the shortest path from v_1 to v_7 ($\langle v_1, v_3, v_4, v_5, v_7 \rangle$) contains the shortest path from v_3 to v_5, the shortest path from v_4 to v_7, etc. We need to capture this feature when formulating the benefit of a path.
- **Cache structure:** Web search caching may employ a hash table to check efficiently whether a query can be found in the cache. However, such a hash table cannot support the subpath matching found in our setting. A new structure is required to organize the cache content in an effective way for supporting subpath matching. Furthermore, this problem is complicated by the overlapping of paths. For example, the shortest path $\langle v_1, v_3, v_4, v_5, v_7 \rangle$ and the shortest path $\langle v_2, v_3, v_4, v_5, v_6 \rangle$ have a significant overlap, although one does not contain the other. We will exploit the overlapping of paths

to design a compact cache structure.

- **Query processing cost:** At the server side, when a cache miss occurs, an algorithm is invoked to compute the query result. Some results are more expensive to obtain than others. To optimize the server performance, we need a cost model for estimating the cost of evaluating a query. However, to our knowledge, there is no work on estimating the cost of a shortest path query with respect to an unknown shortest path algorithm.

In order to tackle the above challenges, we make the following contributions:

- We formulate a systematic model for quantifying the benefit of caching a specific shortest path.
- We design techniques for extracting statistics from query logs and benchmarking the cost of a shortest path call.
- We propose an algorithm for selecting paths to be placed in the cache.
- We develop a compact and efficient cache structure for storing shortest paths.
- We study the above contributions empirically using real data.

The rest of the chapter is organized as follows. In Section 3.1 we outline the related work specific to this chapter. Section 3.2 defines our problem formally, and Section 3.3 formulates a model for capturing the benefit of a cached shortest path, examines the query frequency and the cost of a shortest path call, and presents an algorithm for selecting appropriate paths to be placed in the cache. Section 3.4 presents a compact and efficient cache structure for storing shortest paths. Our proposed methods are then evaluated on real data in Section 3.5. Section 3.6 concludes the chapter.

3.1 Related Work

The related work relevant to this chapter is described in Chapter 2, see Table 2.1. This Section will present the differences between this chapter and the related work presented in Chapter 2.

3.1.1 Web Search Caching

The related work on Web search caching is presented in Section 2.1. Web search caching is a common type of caching using the client server architecture, however, none of the work done in this area considers the caching of shortest path. In our caching problem, different shortest path queries also have different processing times, Thus we also propose a cost-oriented model for quantifying the benefit of placing a path in the cache. The cost model for web search caching is not applicable to our caching problem.

3.1.2 Semantic Caching

Prior work on semantic caching, using spatial data, have proposed using semantic regions or R-Trees to answer queries (see Section 2.2). However, no semantic caching techniques have been proposed for graphs or shortest paths. For client side semantic caching the client is locally storing previous results received from the server. Each result is associated with a valid range which can be used to answer full or partial queries. We have studied the semantic caching of shortest paths. By the optimal subpath property [29], a shortest path SP can answer any shortest path query $Q_{s,t}$ whose source s and target t both fall into SP. For example, in Figure 3.3, the shortest path $SP_{1,10} = \langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10} \rangle$ can answer the shortest path query $Q_{4,10}$ as the path $SP_{1,10}$ contains both v_4 and v_{10}.

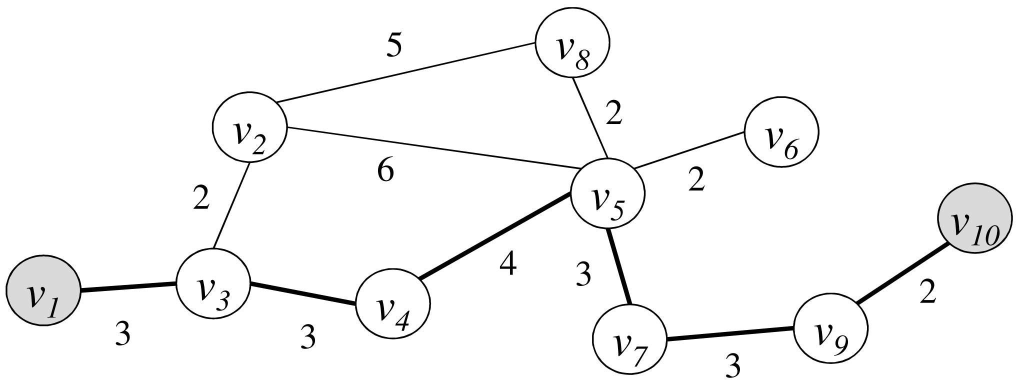


Figure 3.3. A road network, with a query from v_1 to v_{10}

3.1.3 Shortest Path Computation

Previous work on storing precomputed results for shortest path are all done on indexing structures (see Section 2.3).

As a possible approach to our caching problem, one could assume that a specific shortest path index is being used at the server. A portion of the index may be cached so that it can be used to answer certain queries rapidly. Unfortunately, this approach is tightly coupled to the assumed index, and it is inapplicable to servers that employ other indexes (or new index developed in the future).

In this chapter we view the shortest path method as a black-box and decouple it from the cache. The main advantage is that our approach is applicable to any shortest path method (including online APIs such as Google Directions), without knowing its implementation.

3.2 Problem Setting

Following a coverage of background definitions and properties, we present our problem and objectives. Table 3.1 summarizes the notation used in this chapter.

Notation	Meaning
$G(V,E)$	a graph with node set V and edge set E
v_i	a node in V
(v_i, v_j)	an edge in E
$W(v_i, v_j)$	the edge weight of $W(v_i, v_j)$
$Q_{s,t}$	shortest path query from node v_s to node v_t
$P_{s,t}$	the shortest path result of $Q_{s,t}$
$\|P_{s,t}\|$	the size of $P_{s,t}$ (in number of nodes)
$E_{s,t}$	the expense of executing query $Q_{s,t}$
$\chi_{s,t}$	The frequency of a Shortest Path (SP)
Ψ	the cache
$\mathfrak{U}(P_{s,t})$	the set of all subpaths in $P_{s,t}$
$\mathfrak{U}(\Psi)$	the set of all subpaths of paths in Ψ
$\gamma(\Psi)$	the total benefit of the content in the cache
$\mathcal{QL}$	query log

Table 3.1. Summary of notation

3.2.1 Definitions and Properties

We first define the notions of graph and shortest path.

Definition 3.1 **Graph model.**

Let $G(V,E)$ be a graph with a set V of nodes and a set E of edges. Each node

$v_i \in V$ models a road junction. Each edge $(v_i, v_j) \in E$ models a road segment, and its weight (length) is denoted as $W(v_i, v_j)$.

Definition 3.2 **Shortest path: query and result.**
A shortest path query, denoted by $Q_{s,t}$, consists of a source node v_s and a target node v_t.

The result of $Q_{s,t}$, denoted by $P_{s,t}$, is the path from v_s to v_t (on graph G) with the minimum sum of edge weights (lengths) along the path. We can represent $P_{s,t}$ as a list of nodes: $\langle v_{x_0}, v_{x_1}, v_{x_2} \cdots, v_{x_m} \rangle$, where $v_{x_0} = v_s$, $v_{x_m} = v_t$, and the path distance is: $\sum_{i=0}^{m-1} W(v_{x_i}, v_{x_{i+1}})$.

We consider only undirected graphs in our examples. Our techniques can be easily applied to directed graphs. In the example graph of Figure 3.2, the shortest path from v_1 to v_7 is the path $P_{1,7} = \langle v_1, v_3, v_4, v_5, v_7 \rangle$ with its length $3 + 6 + 9 + 5 = 23$. We may also associate a point location with each vertex.

Shortest paths exhibit the optimal subpath property (see Lemma 3.1): every subpath of a shortest path is also a shortest path. For example, in Figure 3.2, the shortest path $P_{1,7} = \langle v_1, v_3, v_4, v_5, v_7 \rangle$ contains these shortest paths: $P_{1,3}, P_{1,4}, P_{1,5}, P_{1,7}, P_{3,4}, P_{3,5}, P_{3,7}, P_{4,5}, P_{4,7}, P_{5,7}$.

Lemma 3.1 **Optimal subpath property (from [29]).**
The shortest path $P_{a,b}$ contains the shortest path $P_{s,t}$ if $v_s \in P_{a,b}$ and $v_t \in P_{a,b}$. Specifically, let $P_{a,b} = \langle v_{x_0}, v_{x_1}, v_{x_2} \cdots, v_{x_m} \rangle$. We have $P_{s,t} = \langle v_{x_i}, v_{x_{i+1}}, \cdots, v_{x_j} \rangle$ if $v_s = v_{x_i}$ and $v_t = v_{x_j}$ for some i, j that $0 \le i \le j \le m$.

As we will discuss shortly, this property can be exploited for the caching of shortest paths.

3.2.2 Problem and Objectives

We adopt the architecture of Figure 3.1 when addressing the caching problem. If a proxy server is used it will be run by the same owner as the server. Users with mobile devices issue shortest path queries to an online server. The cache, as defined below, can be placed at either a proxy or the server. It helps optimize the computation and communication costs at the server/proxy, as well as reduce the response time of shortest path queries.

Definition 3.3 **Cache and budget.**
Given a cache budget $\mathcal{B}$, a cache Ψ is allowed to store a collection of shortest path results such that $|\Psi| \leq \mathcal{B}$, where the cache size $|\Psi| = \sum_{P_{s,t} \in \Psi} |P_{s,t}|$ is the total number of nodes of all shortest paths in Ψ.

As discussed in Section 3.1, recent literature on web search caching [7–10] advocates the use of a static caching that has very low runtime overhead and only sacrifices the hit ratio slightly. Thus, we adopt the static caching paradigm and exploit a query log to build the cache.

Definition 3.4 **Query log.**
A query log $\mathcal{QL}$ is a collection of timestamped queries that have been issued by users in the past.

Figure 3.4 identifies essential components in a static caching system: (i) a shortest path API, (ii) a cache, (iii) an online module for cache lookup, and (iv) offline/periodically invoked modules for collecting a query log, benchmarking the API cost, and populating the cache.

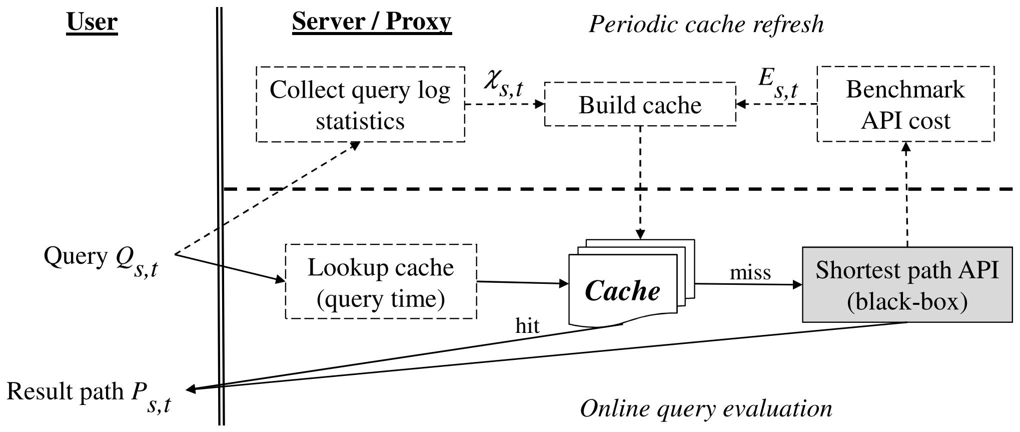


Figure 3.4. Components in a static caching system

The *shortest path component* (in gray) is external to the system, so we are not allowed to modify its implementation. For the server scenario, the shortest path API is linked to a typical shortest path algorithm (e.g., Dijkstra, A* search). For the proxy scenario, the shortest path API triggers a query message to the server. In either case, calling the shortest path API incurs expensive computation/communication, as defined shortly. Different queries may have different costs. In general, a long-range query incurs higher cost than a short-range query.

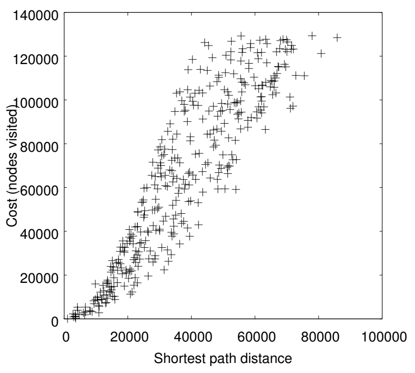
DEFINITION 3.5 **Expense of executing query.**
We denote by $E_{s,t}$ the expense (i.e., response time) of the shortest path API to process query $Q_{s,t}$.

We employ a *cache* to reduce the overall cost of invoking the shortest path API. Having received a query (at runtime), the server/proxy checks whether the cache contains the query result. If this is a hit, the result from the cache is reported to the user immediately. This saves the cost of calling the shortest path API. Otherwise, the result must be obtained by calling the shortest path API.

We observe that maximizing the cache hit ratio does not necessarily mean that the overall cost is reduced significantly. In the server scenario, the cost of calling the shortest path API (e.g., shortest path algorithm) is not fixed and depends heavily on the distance of the shortest path.

We conducted a case study and found a strong correlation between shortest path computation cost and distance. In the first test, Dijkstra's algorithm is used as the API. We generated 500 random shortest path queries on the Aalborg network (see Section 3.5). Figure 3.5a shows the shortest path distance (x-axis) and the number of nodes visited (y-axis) for each query. In the second test, the Google Directions API is used as the API. We tested several queries and plotted their shortest travel times and costs (i.e., response times) in Figure 3.5b. In summary, caching a short-range path may only provide a negligible improvement, even if the path is queried frequently. Therefore, we will study the *benchmarking* of the cost of calling the API.

Adopting the static caching paradigm, the server/proxy *collects a query log* and *re-builds the cache* periodically (e.g., daily). By extracting the distribution of a query log, we are able to estimate the probability of a specific shortest path being queried in the future. Combining such information with benchmarking, we can place promising paths in the cache in order to optimize the overall system performance. We will also investigate the structure of the cache; it should be compact in order to accommodate as many paths as possible, and it should



(a) Dijkstra on map of Aalborg

Source	Target	Travel time (s)	Response time (ms)
Capitol Building	The Smithsonian	372	101.24
The Smithsonian	Washington, DC	419	110.94
White house	War Memorials	41831	168.44
White house	Capitol Building	75805	278.44
White house	Statue of Liberty	88947	362.8
Capitol Building	Mount Rushmore	99456	364.68
White house	Golden Gate Bridge	108353	342.8

(b) Google Directions API

Figure 3.5. Cost vs. distance of a shortest path API

support efficient result retrieval.

Our main objective or problem is to *reduce the overall cost incurred by calling the shortest path API*. We define this problem below. In Section 3.3, we formulate a cache benefit notion $\gamma(\Psi)$, extract statistics to compute $\gamma(\Psi)$, and present an algorithm for the cache benefit maximization problem.

PROBLEM: **Static cache benefit maximization problem.**
Given a cache budget $\mathcal{B}$ and a query log $\mathcal{QL}$, build a cache Ψ with the maximum cache benefit $\gamma(\Psi)$ subject to the budget constraint $\mathcal{B}$, where Ψ contains result paths $P_{s,t}$ whose queries $Q_{s,t}$ belong to $\mathcal{QL}$.

Our secondary objectives are to: (i) develop a compact cache structure to maximize the accommodation of shortest paths, and (ii) provide efficient means of retrieving results from the cache. We focus on these issues in Section 3.4.

3.2.3 Existing Solutions for Caching Results

We revisit existing solutions for caching web search results [14] and explain why they are inadequate for shortest path caching.

Dynamic Caching—LRU: A typical dynamic caching method for web search is the Least-Recently-Used (LRU) method [14]. When a new query is submitted, its result is inserted into the cache. When the cache does not have space for a result, the least-recently-used result in the cache is evicted to make space.

We proceed to illustrate the running steps of LRU on the map in Figure 3.2. Let the cache budget $\mathcal{B}$ be 10 (i.e., it can hold 10 nodes). Table 3.2 shows the query and the cache content at each time T_i. Each cached path is associated with the last time it was used. At times T_1 and T_2, both queries produce cache misses and their results ($P_{3,6}$ and $P_{1,6}$) are inserted into the cache (as they fit). At time T_3, query $Q_{2,7}$ causes a cache miss as it cannot be answered by any cached path. Before inserting its result $P_{2,7}$ into the cache, the least recently used path $P_{3,6}$ is evicted from the cache. At time T_4, query $Q_{1,4}$ contributes a cache hit; it can be answered by the cached path $P_{1,6}$ because the source and target nodes v_1, v_4 fall on $P_{1,6}$ (see Lemma 3.1). The running steps at subsequent times are shown in Table 3.2. In total, the LRU cache has 2 hits.

LRU cannot determine the benefit of a path effectively. For example, paths $P_{1,6}$ and $P_{2,7}$ (obtained at times T_2 and T_3) can answer many queries at subsequent times, e.g., $Q_{1,4}, Q_{2,5}, Q_{3,6}, Q_{3,6}$. If they were kept in the cache, there would be 4 cache hits. However, LRU evicts them before they can be used to answer other queries.

Time	$Q_{s,t}$	$P_{s,t}$	Paths in LRU cache	event
T_1	$Q_{3,6}$	$\langle v_3, v_4, v_5, v_6\rangle$	$P_{3,6} : T_1$	miss
T_2	$Q_{1,6}$	$\langle v_1, v_3, v_4, v_5, v_6\rangle$	$P_{1,6} : T_2,\ P_{3,6} : T_1$	miss
T_3	$Q_{2,7}$	$\langle v_2, v_3, v_4, v_5, v_7\rangle$	$P_{2,7} : T_3,\ P_{1,6} : T_2$	miss
T_4	$Q_{1,4}$	$\langle v_1, v_3, v_4\rangle$	$P_{1,6} : T_4,\ P_{2,7} : T_3$	hit
T_5	$Q_{4,8}$	$\langle v_4, v_5, v_7, v_8\rangle$	$P_{4,8} : T_5,\ P_{1,6} : T_4$	miss
T_6	$Q_{2,5}$	$\langle v_2, v_3, v_4, v_5\rangle$	$P_{2,5} : T_6,\ P_{4,8} : T_5$	miss
T_7	$Q_{3,6}$	$\langle v_3, v_4, v_5, v_6\rangle$	$P_{3,6} : T_7,\ P_{2,5} : T_6$	miss
T_8	$Q_{3,6}$	$\langle v_3, v_4, v_5, v_6\rangle$	$P_{3,6} : T_8,\ P_{2,5} : T_6$	hit

Table 3.2. Example of LRU on a sequence of queries

As another limitation, LRU is not designed to support subpath matching efficiently. Upon receiving a query $Q_{s,t}$, every path in the cache needs to be scanned in order to check whether the path contains v_s and v_t. This incurs significant runtime overhead, possibly outweighing the advantages of the cache.

Static Caching—HQF: A typical static caching method for web search is the Highest-Query-Frequency (HQF) method [14]. In an offline phase, the most frequent queries are selected from the query log $\mathcal{QL}$, and then their results are inserted into the cache. The cache content remains unchanged during runtime.

Like in web caching, the frequency of a query $Q_{s,t}$ is the number of queries in $\mathcal{QL}$ that are identical to $Q_{s,t}$. Let us consider an example query log: $\mathcal{QL} = \{Q_{3,6}, Q_{1,6}, Q_{2,7}, Q_{1,4}, Q_{4,8}, Q_{2,5}, Q_{3,6}, Q_{3,6}\}$. Since $Q_{3,6}$ has the highest frequency (3), HQF picks the corresponding result path $P_{3,6}$. It fails to pick $P_{1,6}$ because its query $Q_{1,6}$ has a low frequency (1). However, path $P_{1,6}$ is more promising than $P_{3,6}$ because $P_{1,6}$ can be used to answer more queries than can $P_{3,6}$. This creates a problem in HQF because the query frequency definition does not capture characteristics specific to our problem—shortest paths may overlap, and the result of one query may be used to answer multiple other queries.

Shared limitations of LRU and HQF: Furthermore, neither LRU nor HQF consider the variations in the expense of obtaining shortest paths. Consider the cache in the server scenario as an example. Intuitively, it is more expensive to process a long-range query than a short-range query. Caching an expensive-to-obtain path could lead to greater savings in the future. An informed choice of which paths to cache should take such expenses into account.

Also, the existing approaches have not studied the utilization of the cache space for shortest paths. For example, in Table 3.2, the paths in the cache overlap and cause wasted space on storing duplicate nodes among the overlapping paths. It is important to design a compact cache structure that exploits path sharing to avoid storing duplicated nodes.

3.3 Benefit-Driven Caching

We propose a benefit-driven approach to determining which shortest paths should be placed in the cache. Section 3.3.1 formulates a model for capturing the *benefit* of a cache on potential queries. This model requires knowledge of (i) the frequency $\chi_{s,t}$ of a query, and (ii) the expense $E_{s,t}$ of processing a query. Thus, we investigate how to extract query frequencies from a query log in Section 3.3.2 and benchmark the expense of processing a query in Section 3.3.3. Finally, in Section 3.3.4, we present an algorithm for selecting promising paths to be placed in the cache.

3.3.1 Benefit Model

We first study the benefit of a cached shortest path and then examine the benefit of a cache.

First, we consider a cache Ψ that contains one shortest path $P_{a,b}$ only. Recall from Figure 3.4 that when a query $Q_{s,t}$ can be answered by a cached path $P_{a,b}$, this produces a cache hit and avoids the cost of invoking the shortest path API. In order to model the benefit of $P_{a,b}$, we must address two questions:

1. Which queries $Q_{s,t}$ can be answered by the path $P_{a,b}$?
2. For query $Q_{s,t}$, what are the cost savings?

The first question is answered by Lemma 3.1. The path $P_{a,b}$ contains the path $P_{s,t}$ if both nodes v_s and v_t appear in $P_{a,b}$. Thus, we define the *answerable query set* of the path $P_{a,b}$ as:

$$\mathfrak{U}(P_{a,b}) = \{P_{s,t} \mid s \in P_{a,b} \wedge t \in P_{a,b} \wedge s \neq t\} \tag{3.1}$$

This set contains the queries that can be answered by $P_{a,b}$. Taking Figure 3.2 as the example graph, the answerable query set of path $P_{1,6}$ is: $\mathfrak{U}(P_{1,6}) = \{P_{1,3}, P_{1,4}, P_{1,5}, P_{1,6},\ P_{3,4}, P_{3,5}, P_{3,6},\ P_{4,5}, P_{4,6}, P_{5,6}\}$. Table 3.3 shows the answerable query sets of other paths.

Regarding the second question, the expected cost savings for query $Q_{s,t}$ depends on (i) its query frequency $\chi_{s,t}$ and (ii) the expense $E_{s,t}$ for the shortest path API to process it. Since the path $P_{a,b}$ can answer query $Q_{s,t}$, we save cost

$P_{a,b}$	$\mathfrak{U}(P_{a,b})$
$P_{1,4}$	$P_{1,3}, P_{1,4}, P_{3,4}$
$P_{1,6}$	$P_{1,3}, P_{1,4}, P_{1,5}, P_{1,6}, P_{3,4}, P_{3,5}, P_{3,6}, P_{4,5}, P_{4,6}, P_{5,6}$
$P_{2,5}$	$P_{2,3}, P_{2,4}, P_{2,5}, P_{3,4}, P_{3,5}, P_{4,5}$
$P_{2,7}$	$P_{2,3}, P_{2,4}, P_{2,5}, P_{2,7}, P_{3,4}, P_{3,5}, P_{3,7}, P_{4,5}, P_{4,7}, P_{5,7}$
$P_{3,6}$	$P_{3,4}, P_{3,5}, P_{3,6}, P_{4,5}, P_{4,6}, P_{5,6}$
$P_{4,8}$	$P_{4,5}, P_{4,7}, P_{4,8}, P_{5,7}, P_{5,8}, P_{7,8}$

$\mathfrak{U}(\Psi)$, when $\Psi = \{P_{1,6}, P_{3,6}\}$
$P_{1,3}, P_{1,4}, P_{1,5}, P_{1,6}, P_{3,4}, P_{3,5}, P_{3,6}, P_{4,5}, P_{4,6}, P_{5,6}$

Table 3.3. Example of $\mathfrak{U}(P_{s,t})$ and $\mathfrak{U}(\Psi)$

$E_{s,t}$ a total of $\chi_{s,t}$ times, i.e., $\chi_{s,t} \cdot E_{s,t}$ in total.[1]

Combining the answers to both questions, we define the *benefit* of path $P_{a,b}$ as:

$$\gamma(P_{a,b}) = \sum_{P_{s,t} \in \mathfrak{U}(P_{a,b})} \chi_{s,t} \cdot E_{s,t} \tag{3.2}$$

The path benefit $\gamma(P_{a,b})$ answers the question: *"If path $P_{a,b}$ is in the cache, how much cost can we save in total?"*

Let us assume that we are given the values of $\chi_{s,t}$ and $E_{s,t}$ for all pairs (v_s, v_t), as shown in Figure 3.6. We study how to derive them in subsequent Sections. To compute $\gamma(P_{1,6})$ of path $P_{1,6}$, we first find its answerable query set $\mathfrak{U}(P_{1,6})$ (see Table 3.3). Since $\mathfrak{U}(P_{1,6})$ contains the path $P_{1,4}$, it contributes a benefit of $\chi_{1,4} \cdot E_{1,4} = 1 \cdot 2$ (by lookup in Figure 3.6). Summing up the benefits of all paths in $\mathfrak{U}(P_{1,6})$, we thus obtain: $\gamma(P_{1,6}) = 0+1\cdot2+0+1\cdot4+0+0+3\cdot3+0+0+0 = 15$. Similarly, we can compute the benefit of path $P_{3,6}$: $\gamma(P_{3,6}) = 0+0+3\cdot3+0+0+0 = 9$.

We proceed to extend our equations to the general case—a cache containing

[1]We ignore the overhead of cache lookup as it is negligible compared to the expense $E_{s,t}$ of processing a query $Q_{s,t}$. Efficient cache structures are studied in Section 3.4.

$\chi_{s,t}$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	/	0	0	1	0	1	0	0
v_2	0	/	0	0	1	0	1	0
v_3	0	0	/	0	0	3	0	0
v_4	1	0	0	/	0	0	0	1
v_5	0	1	0	0	/	0	0	0
v_6	1	0	3	0	0	/	0	0
v_7	0	1	0	0	0	0	/	0
v_8	0	0	0	1	0	0	0	/

(a) $\chi_{s,t}$ values

$E_{s,t}$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	/	2	1	2	3	4	4	5
v_2	2	/	1	2	3	4	4	5
v_3	1	1	/	1	2	3	3	4
v_4	2	2	1	/	1	2	2	3
v_5	3	3	2	1	/	1	1	2
v_6	4	4	3	2	1	/	2	3
v_7	4	4	3	2	1	2	/	1
v_8	5	5	4	3	2	3	1	/

(b) $E_{s,t}$ values

Figure 3.6. Example of $\chi_{s,t}$ and $E_{s,t}$ values for the graph

multiple shortest paths. Observe that a query can be answered by the cache Ψ if it can be answered by any path $P_{a,b}$ in Ψ. Thus, we define the answerable query set of Ψ as the union of all $\mathfrak{U}(P_{a,b})$, and we define the benefit of Ψ accordingly.

$$\mathfrak{U}(\Psi) = \bigcup_{P_{a,b}\in\Psi} \mathfrak{U}(P_{a,b}) \tag{3.3}$$

$$\gamma(\Psi) = \sum_{P_{s,t}\in\mathfrak{U}(\Psi)} \chi_{s,t} \cdot E_{s,t} \tag{3.4}$$

The cache benefit $\gamma(\Psi)$ answers the question: *"Using cache Ψ, how much cost can we save in total?"*

Suppose that the cache Ψ contains two paths $P_{1,6}$ and $P_{3,6}$. The answerable query set $\mathfrak{U}(\Psi)$ of Ψ is shown in Table 3.3. By Equation 3.4, we compute the cache benefit as: $\gamma(\Psi) = 1 \cdot 2 + 1 \cdot 4 + 3 \cdot 3 = 15$.

Note that $\gamma(\Psi)$ is not a distributive function. For example, $\gamma(P_{1,6}) + \gamma(P_{3,6}) = 15 + 9 = 24 \neq \gamma(\Psi)$. Since the path $P_{3,6}$ appears in both answerable query sets $\mathfrak{U}(P_{1,6})$ and $\mathfrak{U}(P_{3,6})$, the benefit contributed by $P_{3,6}$ is double-counted in the sum $\gamma(P_{1,6}) + \gamma(P_{3,6})$. On the other hand, the value of $\gamma(\Psi)$ is correct because the path $P_{3,6}$ appears exactly once in the answerable query set $\mathfrak{U}(\Psi)$ of the cache.

Benefit per size unit: The benefit model does not take the size $|P_{a,b}|$ of a path $P_{a,b}$ into account. Assume that we are given two paths $P_{a,b}$ and $P_{a',b'}$ that have the same benefit (i.e., $\gamma(P_{a,b}) = \gamma(P_{a',b'})$) and where $P_{a',b'}$ is smaller than $P_{a,b}$. Intuitively, we then prefer path $P_{a',b'}$ over path $P_{a,b}$ because $P_{a',b'}$ occupies less space, leaving space for the caching of other paths. Thus, we define the *benefit-per-size* of a path $P_{a,b}$ as:

$$\overline{\gamma}(P_{a,b}) \quad = \quad \frac{\gamma(P_{a,b})}{|P_{a,b}|} \tag{3.5}$$

We will utilize this notion in Section 3.3.4.

Recall from Section 3.2.2 that our main problem is to build a cache Ψ such that its benefit $\gamma(\Psi)$ is maximized. This requires values for $\chi_{s,t}$ and $E_{s,t}$. We discuss how to obtain these values in subsequent Sections.

3.3.2 Extracting $\chi_{s,t}$ from Query Log

The frequency $\chi_{s,t}$ of query $Q_{s,t}$ plays an important role in the benefit model. According to a scientific study [30], the mobility patterns of human users follow a skewed distribution. For instance, queries between hot regions (e.g., shopping malls, residential buildings) generally have high $\chi_{s,t}$, whereas queries between sparse regions (e.g., rural areas, country parks) are likely to have low $\chi_{s,t}$.

In this Section, we propose automatic techniques for deriving the values of $\chi_{s,t}$. In our caching system (see Figure 3.4), the server/proxy periodically collects the query log $\mathcal{QL}$ and extracts values of $\chi_{s,t}$. The literature on static web caching [7] suggests that the query frequency is stable within a month and that a month can be used as the periodic time interval. We first study a simple method to extract $\chi_{s,t}$ and then propose a more effective method for extracting $\chi_{s,t}$.

Node-pair frequency counting: With this method, we first create a *node-pair frequency table* χ with $|V| \times |V|$ entries, like the one in Figure 3.6a. The entry in the s-th row and the t-th column represents the value of $\chi_{s,t}$. The storage space of the table is $O(|V|^2)$, regardless of the query log size.

At the beginning, all entries in the table are initialized to zero. Next, we examine each query $Q_{s,t}$ in the query log $\mathcal{QL}$ and increment the entry $\chi_{s,t}$ (and $\chi_{t,s}$).

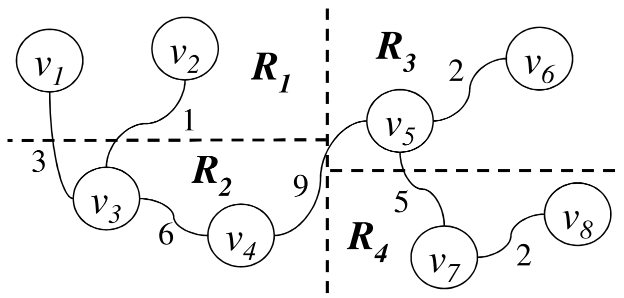
Consider the query log $\mathcal{QL}$ in Table 3.4 as an example. For the first query $Q_{3,6}$ in $\mathcal{QL}$, we increment the entries $\chi_{3,6}$ and $\chi_{6,3}$. Continuing this process with the other queries in $\mathcal{QL}$, we obtain the table χ shown in Figure 3.6a. The $\chi_{s,t}$ values in the table χ can then be used readily in the benefit model in Section 3.3.1.

Timestamp	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Query	$Q_{3,6}$	$Q_{1,6}$	$Q_{2,7}$	$Q_{1,4}$	$Q_{4,8}$	$Q_{2,5}$	$Q_{3,6}$	$Q_{3,6}$

Table 3.4. Query log $\mathcal{QL}$

Region-pair frequency counting: The node-pair frequency table χ requires $O(|V|^2)$ space, which cannot fit into main memory even for a road network of moderate size (e.g., $|V| = 100,000$). To tackle this issue, we propose to (i) partition the graph into L regions (where L is system parameter), and (ii) employ a compact table for storing only the query frequencies between pairs of regions.

For the first step, we can apply any existing graph partitioning technique (e.g., kD-tree partitioning, spectral partitioning). The kD-tree partitioning is applicable to the majority of road networks whose nodes are associated with coordinates. For other graphs, we may apply spectral partitioning, which does not require node coordinates. In Figure 3.7a, we apply a kD-tree on the coordinates of nodes in order to partition the graph into $L = 4$ regions: R_1, R_2, R_3, R_4. The nodes in these regions are shown in Figure 3.7b.



(a) a graph partitioned into 4 regions

R_1 :	$\{v_1, v_2\}$
R_2 :	$\{v_3, v_4\}$
R_3 :	$\{v_5, v_6\}$
R_4 :	$\{v_7, v_8\}$

(b) node sets of regions

χ_{R_i,R_j}	R_1	R_2	R_3	R_4
R_1	0	1	2	1
R_2	1	0	3	1
R_3	2	3	0	0
R_4	1	1	0	0

(c) region-pair frequency table $\widehat{\chi}$

Figure 3.7. Counting region-pair frequency in table $\widehat{\chi}$

For the second step, we create a *region-pair frequency table* $\widehat{\chi}$ with $L \times L$ entries, like the one in Figure 3.7c. The entry in the R_i-th row and the R_j-th column represents the value of $\widehat{\chi}_{R_i,R_j}$. The storage space of this table is only $O(L^2)$ and can be controlled by the parameter L. Initially, all entries in the table are set to zero. For each query $Q_{s,t}$ in the query log $\mathcal{QL}$, we first find the region (say, R_i) that contains node v_s and the region (say, R_j) that contains node v_t. Then we increment the entry $\widehat{\chi}_{R_i,R_j}$ and $\widehat{\chi}_{R_j,R_i}$. As an example, we read the query log $\mathcal{QL}$ in Table 3.4 and examine the first query $Q_{3,6}$. We find that nodes

v_3 and v_6 fall in the regions R_2 and R_3, respectively. Thus, we increment the entries $\widehat{\chi}_{R_2,R_3}$ and $\widehat{\chi}_{R_3,R_2}$. Continuing this process with the other queries in $\mathcal{QL}$, we obtain the table $\widehat{\chi}$ as shown in Figure 3.7c.

The final step is to describe how to derive the value of $\chi_{s,t}$ from the region-pair frequency table $\widehat{\chi}$ so that the table is useful for the benefit model. Note that the frequency of $\widehat{\chi}_{R_i,R_j}$ is contributed by any pair of nodes (v_s, v_t) such that region R_i contains v_s and region R_j contains v_t. Thus, we obtain: $\widehat{\chi}_{R_i,R_j} = \sum_{v_s \in R_i} \sum_{v_t \in R_j} \chi_{s,t}$. If we make the uniformity assumption within a region, we have: $\widehat{\chi}_{R_i,R_j} = |R_i| \cdot |R_j| \cdot \chi_{s,t}$, where $|R_i|$ and $|R_j|$ denotes the number of nodes in the region R_i and R_j, respectively. In other words, we compute the value of $\chi_{s,t}$ from the $\widehat{\chi}$ as follows:

$$\chi_{s,t} = \frac{\widehat{\chi}_{R_i,R_j}}{|R_i| \cdot |R_j|} \tag{3.6}$$

The value of $\chi_{s,t}$ is only computed when it is needed. No additional storage space is required to store $\chi_{s,t}$ in advance.

A benefit of the region-pair method is that it can capture generic patterns for regions rather than specific patterns for nodes. An example pattern could be that many users drive from a residential region to a shopping region. Since drivers live in different apartment buildings, their starting points could be different, resulting in many dispersed entries in the node-pair frequency table χ. In contrast, they contribute to the same entry in the region-pair frequency table $\widehat{\chi}$.

3.3.3 Benchmarking $E_{s,t}$ of Shortest Path APIs

In our caching system (see Figure 3.4), the shortest path API is invoked when there is a cache miss. The expense $E_{s,t}$ of a query is the round-trip communication time + the shortest path computation cost. Here, we study how to capture the expense $E_{s,t}$ of computing query $Q_{s,t}$.

Recall that the cache can be placed at a proxy or a server. For the proxy scenario, the shortest path API triggers the issue of a query message to the server. The cost is dominated by the communication round-trip time, which is the same for all queries. Thus, for simplicity, we define the expense $E_{s,t}$ of query $Q_{s,t}$ in this scenario as:

$$E_{s,t}(proxy) = 1 \tag{3.7}$$

Our subsequent discussion focuses on the server scenario. The communication round-trip time is not affected by cache misses, and is thus a cost that must always be paid. Let $\mathcal{ALG}$ be the shortest path algorithm invoked by the shortest path API. We denote the running time of $\mathcal{ALG}$ for query $Q_{s,t}$ as the expense $E_{s,t}(\mathcal{ALG})$.

We develop a generic technique for estimating $E_{s,t}(\mathcal{ALG})$; it is applicable to any algorithm $\mathcal{ALG}$ and to an arbitrary graph topology. To our best knowledge, this work is the first to explore this issue. There exists work on shortest path distance estimation [31], but no work exists on estimating the running time of an arbitrary algorithm $\mathcal{ALG}$. Even for existing shortest path indexes [24–28], only worst-case query times have been analyzed. They cannot be used to estimate the running time for a specific query $Q_{s,t}$.

A brute-force approach is to precompute $E_{s,t}(\mathcal{ALG})$ by running $\mathcal{ALG}$ for every pair of source node v_s and target node v_t. These values can be stored in a table, like the one in Figure 3.6b. However, this approach is prohibitively expensive as it requires running $\mathcal{ALG}$ $|V|^2$ times.

Our estimation technique incurs only a small precomputation overhead. Intuitively, the expense $E_{s,t}$ is strongly correlated with the distance of the shortest path $P_{s,t}$. Short-range queries are expected to incur small $E_{s,t}$, whereas long-range queries should produce high $E_{s,t}$. Our idea is to classify queries based on distances and then estimate the expense of a query according to its category.

Estimation structures: To enable estimation, we build two data structures: (i) a distance estimator and (ii) an expense histogram.

The *distance estimator* aims at estimating the shortest path distance of a query $Q_{s,t}$. We simply adopt the landmark-based estimator [31] as the distance estimator. It requires selecting a set U of nodes as landmarks and precomputing the distances from each landmark node to every node in the graph. This incurs $O(|U||V|)$ storage space and $O(|U||E|\log|E|)$ construction time. Potamias et al. [31] suggest that $|U| = 20$ is sufficient for accurate distance estimation. Figure 3.8a shows an example with two landmark nodes, i.e., $U = \{v_3, v_5\}$, together with their distances $d(u_j, v_i)$ to other nodes.

$d(u_j, v_i)$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
$u_1 : v_3$	3	1	0	6	15	17	20	22
$u_2 : v_5$	18	16	15	9	0	2	5	7

(a) distance estimator

d	0-5	5-10	10-15	15-20	20-25
$E(d)$	1.2	2.5	3.8	5.3	6.8

(b) expense histogram

Figure 3.8. Example of estimating expense $\chi_{s,t}$

We use an *expense histogram* for recording the average expense of queries with respect to their distances, as illustrated in Figure 3.8b. In general, the histogram consists of H categories of distances. Then, we execute the algorithm $\mathcal{ALG}$ on a sample of S random queries to obtain their expenses, and we update the corresponding buckets in the histogram. This histogram requires $O(H)$ storage space and $S \cdot O(\mathcal{ALG})$ construction time.

Estimation process: With the above structures, the value of $E_{s,t}$ can be estimated in two steps. First, we apply the distance estimator of Potamias et al. [31] and estimate the shortest path distance of $P_{s,t}$ as: $\min_{i=1..|U|} d(u_j, v_s) + d(u_j, v_t)$. This step takes $O(|U|)$ time. Second, we perform a lookup in the expense histogram and return the expense in the corresponding bucket as the estimated expense $E_{s,t}$.

Consider the estimation of $E_{1,4}$ as an example. Using the distance estimator in Figure 3.8a, we estimate the shortest path distance of $P_{1,4}$ as: $\min\{3+6, 18+9\} = 9$. We then do a lookup in the expense histogram in Figure 3.8b and thus estimate $E_{1,4}$ to be 2.5.

3.3.4 Cache Construction Algorithm

As in other static caching methods [7–10], we exploit the query log $\mathcal{QL}$ to identify promising results to be placed in the cache Ψ. Each query $Q_{a,b} \in \mathcal{QL}$ has a corresponding path result $P_{a,b}$. This Section presents a cache construction algorithm for placing such paths into cache Ψ so that the total cache benefit $\gamma(\Psi)$ is maximized, with the cache size $|\Psi|$ being bounded by a budget $\mathcal{B}$.

In web search caching, Ozcan et al. [10] propose a greedy algorithm to populate a cache. We also adopt the greedy approach to solve our problem. Nevertheless, the application of a greedy approach to our problem presents challenges.

Challenges of a greedy approach: It is tempting to populate the cache with paths by using a greedy approach that (i) computes the benefit-per-size $\overline{\gamma}(P_{a,b})$ for each path $P_{a,b}$ and then (ii) iteratively places items that have the highest $\overline{\gamma}(P_{a,b})$ in the cache. Unfortunately, this approach does not necessarily produce a cache with high benefit.

As an example, consider the graph in Figure 3.7a and the query log $\mathcal{QL}$ in Table 3.4. The result paths of the queries of $\mathcal{QL}$ are: $P_{1,6}$, $P_{2,7}$, $P_{1,4}$, $P_{4,8}$, $P_{2,5}$, $P_{3,6}$. To make the benefit calculation readable, we assume that $E_{s,t} = 1$ for each pair, and we use the values of $\chi_{s,t}$ in Figure 3.6a. In this greedy approach, we first compute the benefit-per-size of each path above. For example, $P_{1,6}$ can answer five queries $Q_{3,6}, Q_{1,6}, Q_{1,4}, Q_{3,6}, Q_{3,6}$ in $\mathcal{QL}$, and its size $|P_{1,6}|$ is 5, so its benefit-per-size is: $\overline{\gamma}(P_{1,6}) = 5/5$. Since $P_{3,6}$ has a size of 4 and it can answer three queries $Q_{3,6}, Q_{3,6}, Q_{3,6}$ in $\mathcal{QL}$, its benefit-per-size is: $\overline{\gamma}(P_{3,6}) = 3/4$. Repeating this process for the other paths, we obtain: $\overline{\gamma}(P_{1,4}) = 1/3$, $\overline{\gamma}(P_{1,6}) = 5/5$, $\overline{\gamma}(P_{2,5}) = 1/4$, $\overline{\gamma}(P_{2,7}) = 2/5$, $\overline{\gamma}(P_{3,6}) = 3/4$, $\overline{\gamma}(P_{4,8}) = 1/4$. Given the cache budget $\mathcal{B} = 10$, the greedy approach first picks $P_{1,6}$ and then picks $P_{3,6}$. Thus, we obtain the cache $\Psi = \{P_{1,6}, P_{3,6}\}$ with the size 9 (i.e., total number of nodes in the cache). No more paths can be inserted into the cache as it is full.

The problem with the greedy approach is that it ignores the existing cache content when it chooses a path $P_{a,b}$. If many queries that are answerable by path $P_{a,b}$ can already be answered by some existing path in the cache, it is not worthwhile to include $P_{a,b}$ into the cache.

In the above example, the greedy approach picks the path $P_{3,6}$ after the path $P_{1,6}$ has been inserted into the cache. Although path $P_{3,6}$ can answer the three queries $Q_{3,6}, Q_{3,6}, Q_{3,6}$ in $\mathcal{QL}$, all those queries can already be answered by the path $P_{1,6}$ in the cache. So while path $P_{3,6}$ has no benefit, the greedy approach still picks it.

A revised greedy approach: To tackle the above issues, we study a notion that expresses the benefit of a path $P_{a,b}$ in terms of the queries that can only be answered by $P_{a,b}$ and not any existing paths in the cache Ψ.

Definition 3.6 **Incremental benefit-per-size of path $P_{a,b}$.**
Given a shortest path $P_{a,b}$, its incremental benefit-per-size $\Delta\overline{\gamma}(P_{a,b}, \Psi)$ with respect to the cache Ψ, is defined as the additional benefit of placing $P_{a,b}$ into Ψ, per the size of $P_{a,b}$:

$$\begin{aligned} \Delta\overline{\gamma}(P_{a,b}, \Psi) &= \frac{\gamma(\Psi \cup \{P_{a,b}\}) - \gamma(\Psi)}{|P_{a,b}|} \\ &= \sum_{P_{s,t} \in \mathfrak{U}(P_{a,b}) - \mathfrak{U}(\Psi)} \frac{\chi_{s,t} \cdot E_{s,t}}{|P_{a,b}|} \end{aligned} \tag{3.8}$$

We propose a revised greedy algorithm that proceeds in rounds. The cache Ψ is initially empty. In each round, the algorithm computes the incremental benefit $\Delta\overline{\gamma}(P_{a,b}, \Psi)$ of each path $P_{a,b}$ with respect to the cache Ψ (with its current content). Then the algorithm picks the path with the highest $\Delta\overline{\gamma}$ value and inserts it into Ψ. These rounds are repeated until the cache Ψ becomes full (i.e., reaching its budget $\mathcal{B}$).

We continue with the above running example and show the steps of this revised greedy algorithm in Table 3.5. In the first round, the cache Ψ is empty, so the incremental benefit $\Delta\overline{\gamma}(P_{a,b}, \Psi)$ of each path $P_{a,b}$ equals its benefit $\overline{\gamma}(P_{a,b})$. From the previous example, we obtain: $\Delta\overline{\gamma}(P_{1,4}) = 1/3$, $\Delta\overline{\gamma}(P_{1,6}) = 5/5$, $\Delta\overline{\gamma}(P_{2,5}) = 1/4$, $\Delta\overline{\gamma}(P_{2,7}) = 2/5$, $\Delta\overline{\gamma}(P_{3,6}) = 3/4$, $\Delta\overline{\gamma}(P_{4,8}) = 1/4$. After choosing the path $P_{1,6}$ with the highest $\Delta\overline{\gamma}$ value, the cache becomes: $\Psi = \{P_{1,6}\}$. In the second round, we consider the cache when computing the $\Delta\overline{\gamma}$ value of a path. For the path $P_{3,6}$, all queries that can be answered by it can also be answered by the path $P_{1,6}$ in the cache. Thus, the $\Delta\overline{\gamma}$ value of $P_{3,6}$ is: $\Delta\overline{\gamma}(P_{3,6}) = 0$. Continuing this with other queries, we obtain: $\Delta\overline{\gamma}(P_{1,4}) = 0$, $\Delta\overline{\gamma}(P_{1,6}) = 0$, $\Delta\overline{\gamma}(P_{2,5}) = 1/4$, $\Delta\overline{\gamma}(P_{2,7}) = 2/5$, $\Delta\overline{\gamma}(P_{3,6}) = 0$, $\Delta\overline{\gamma}(P_{4,8}) = 1/4$. The path $P_{2,7}$ with the highest $\Delta\overline{\gamma}$ value is chosen and then the cache becomes: $\Psi = \{P_{1,6}, P_{2,7}\}$. The total benefit of the cache $\gamma(\Psi)$ is 7. Now the cache is full.

Round	Path						Cache Ψ	
	$P_{1,4}$	$P_{1,6}$	$P_{2,5}$	$P_{2,7}$	$P_{3,6}$	$P_{4,8}$	before round	after round
1	1/3	$\boxed{\mathbf{5/5}}$	1/4	2/5	3/4	1/4	empty	$P_{1,6}$
2	0	0	1/4	$\boxed{\mathbf{2/5}}$	0	1/4	$P_{1,6}$	$P_{1,6}, P_{2,7}$

Table 3.5. Incremental benefits of paths in our greedy algorithm (*boxed values indicate the selected paths***)**

Cache construction algorithm and its time complexity: Algorithm 3.1 shows the pseudo-code of our revised greedy algorithm. It takes as input the graph $G(V, E)$, the cache budget $\mathcal{B}$, and the query log $\mathcal{QL}$. The cache budget $\mathcal{B}$ denotes the capacity of the cache in terms of the number of nodes. The statistics of query frequency χ and query expense E are required for computing the incremental benefit of a path.

The initialization phase corresponds to Lines 1–5. The cache Ψ is initially empty. A max-heap H is employed to organize result paths in descending order of their $\Delta\overline{\gamma}$ values. For each query $Q_{a,b}$ in the query log $\mathcal{QL}$, we retrieve its result path $P_{a,b}$, compute its $\Delta\overline{\gamma}$ value as $\Delta\overline{\gamma}(P_{a,b}, \Psi)$, and then insert $P_{a,b}$ into H.

Algorithm 3.1 Revised-Greedy(Graph $G(V, E)$, Cache budget $\mathcal{B}$, Query log $\mathcal{QL}$, Frequency χ, Expense E)

```
1: $\Psi \leftarrow$ new cache;
2: $H \leftarrow$ new max-heap;                                  ▷ storing result paths
3: for each $Q_{a,b} \in \mathcal{QL}$ do
4:     $P_{a,b}.\Delta\overline{\gamma} \leftarrow \Delta\overline{\gamma}(P_{a,b}, \Psi)$;          ▷ compute using $\chi$ and $E$
5:     insert $P_{a,b}$ into $H$;
6: while $|\Psi| \leq \mathcal{B}$ and $|H| > 0$ do
7:     $P_{a',b'} \leftarrow H.pop()$;                          ▷ potential best path
8:     $P_{a',b'}.\Delta\overline{\gamma} \leftarrow \Delta\overline{\gamma}(P_{a',b'}, \Psi)$;        ▷ update $\Delta\overline{\gamma}$ value
9:     if $P_{a',b'}.\Delta\overline{\gamma} \geq$ top $\Delta\overline{\gamma}$ of $H$ then           ▷ actual best path
10:        if $\mathcal{B} - |\Psi| \geq |P_{a',b'}|$ then                  ▷ enough space
11:            insert $P_{a',b'}$ into $\Psi$;
12:    else                                                 ▷ not the best path
13:        insert $P_{a',b'}$ into $H$;
14: return $\Psi$;
```

The algorithm incorporates an optimization to reduce the number of incremental benefit computations in each round (i.e., the loop of Lines 6–13). First, the path $P_{a',b'}$ with the highest $\Delta\overline{\gamma}$ value is selected from H (Line 7) and its current $\Delta\overline{\gamma}$ value is computed (Line 8). According to Lemma 3.2, the $\Delta\overline{\gamma}$ value of a path $P_{a,b}$ in H, which was computed in some previous round, serves as an upper bound of its $\Delta\overline{\gamma}$ value in the current round. If $\Delta\overline{\gamma}(P_{a',b'}, \Psi)$ is above the top key of H (Line 9) then we can safely conclude that $P_{a',b'}$ is superior to all paths in H, without having to compute their exact $\Delta\overline{\gamma}$ values. We then insert the path $P_{a',b'}$ into the cache Ψ if it has sufficient remaining space $\mathcal{B} - |\Psi|$. In case $\Delta\overline{\gamma}(P_{a',b'}, \Psi)$ is smaller than the top key of H, we insert $P_{a',b'}$ back into H. Eventually, H becomes empty, the loop terminates, and the cache Ψ is returned.

LEMMA 3.2 **$\Delta\overline{\gamma}$ is a decreasing function of round i.**

Let Ψ_i be the cache just before the i-th round of the algorithm. It holds that: $\Delta\overline{\gamma}(P_{a,b}, \Psi_i) \geq \Delta\overline{\gamma}(P_{a,b}, \Psi_{i+1})$.

Proof. All paths in Ψ_i must also be in Ψ_{i+1}, so we have: $\Psi_i \subseteq \Psi_{i+1}$. By Equation 3.3, we derive: $\mathfrak{U}(\Psi_i) \subseteq \mathfrak{U}(\Psi_{i+1})$ and then obtain: $\mathfrak{U}(P_{a,b}) - \mathfrak{U}(\Psi_i) \supseteq \mathfrak{U}(P_{a,b}) - \mathfrak{U}(\Psi_{i+1})$. By Definition 3.6, we have $\Delta\overline{\gamma}(P_{a,b}, \Psi) = \sum_{P_{s,t} \in \mathfrak{U}(P_{a,b}) - \mathfrak{U}(\Psi)} \chi_{s,t} \cdot E_{s,t}/|P_{a,b}|$. Combining the above facts, we get: $\Delta\overline{\gamma}(P_{a,b}, \Psi_i) \geq \Delta\overline{\gamma}(P_{a,b}, \Psi_{i+1})$.

□

We proceed to illustrate the power of the above optimization. Consider the second round shown in Table 3.5. Without the optimization, we must recompute the $\Delta\overline{\gamma}$ values of the 5 paths $P_{1,4}, P_{2,5}, P_{2,7}, P_{3,6}, P_{4,8}$ before determining the path with the highest $\Delta\overline{\gamma}$ value. Using the optimization, we just need to pop the paths $P_{3,6}, P_{2,7}$ from the heap H and recompute their $\Delta\overline{\gamma}$ values. The other paths (e.g,, $P_{1,4}, P_{2,5}, P_{4,8}$) have upper bound $\Delta\overline{\gamma}$ values (1/3, 1/4, 1/4, from the first round) that are smaller than the current $\Delta\overline{\gamma}$ value of $P_{2,7}$ (2/5). Thus, we need not recompute their current $\Delta\overline{\gamma}$ values.

We then analyze the time complexity of Algorithm 3.1, without using the optimization. Let $|\mathcal{QL}|$ be the number of result paths for queries in $\mathcal{QL}$. Let $|P|$ be the average size of the above result paths. The number of paths in the cache is $\mathcal{B}/|P|$, so the algorithm completes in $\mathcal{B}/|P|$ rounds. In each round, we need to process $|\mathcal{QL}|$ result paths and recompute their $\Delta\overline{\gamma}$ values. Computing the $\Delta\overline{\gamma}$ value of a path $P_{a,b}$ requires the examination of each subpath of $P_{a,b}$ (see Definition 3.6). This takes $O(|P|^2)$ time as there are $O(|P|^2)$ subpaths in a path. Multiplying the above terms, the time complexity of our algorithm is: $O(|\mathcal{QL}|\cdot\mathcal{B}\cdot|P|)$. This running time is affordable for a static caching scheme. Also, our experimental results show that the running time of the optimized algorithm is notably better in typical cases.

3.4 Cache Structure

Section 3.4.1 presents a structure that supports efficient cache lookup at query time. Sections 3.4.2 and 3.4.3 present compact cache structures that enable a cache to accommodate as many shortest paths as possible, thus improving the benefit of the cache.

3.4.1 Efficient Lookup via Inverted Lists

Upon receiving a query $Q_{s,t}$, the proxy/server performs a cache lookup for any path $P_{a,b}$ that can answer the query (see Figure 3.4). We propose a structure that enables efficient cache lookup.

The proposed structure involves an array of paths (see Figure 3.9a) and inverted lists of nodes (see Figure 3.9b). The array stores the content of each path. In this example, the array contains three paths: Ψ_1, Ψ_2, Ψ_3. The inverted lists for nodes are used to support efficient lookup. The inverted list of a node v_i stores a list of path IDs Ψ_j whose paths contain v_i. For example, since paths Ψ_1 and Ψ_2 contain the node v_1, the inverted list of v_1 stores Ψ_1 and Ψ_2.

Given a query $Q_{s,t}$, we just need to examine the inverted lists of v_s and v_t. If these two lists have a non-empty intersection (say, Ψ_j) then we are guaranteed

Ψ_1	v_1, v_3, v_4
Ψ_2	v_1, v_3, v_2
Ψ_3	v_2, v_3, v_4, v_5

(a) path array

v_1	Ψ_1, Ψ_2
v_2	Ψ_2, Ψ_3
v_3	Ψ_1, Ψ_2, Ψ_3
v_4	Ψ_1, Ψ_3
v_5	Ψ_3

(b) inverted lists

Figure 3.9. Path array, with inverted lists

that the path Ψ_j can answer query $Q_{s,t}$. For example, for the query $Q_{2,4}$, we first retrieve the inverted lists of v_2 and v_4. The intersection of these two lists is Ψ_3 that can then be used to answer the query. Consider the query $Q_{1,5}$ as another example. Since the inverted lists of v_1 and v_5 have an empty intersection, we get a cache miss.

Cache size analysis: So far, we have only measured the cache size in terms of the paths or nodes in the cache and have not considered the sizes of the auxiliary structures (e.g., inverted lists). Here, we measure the cache size accurately and in an absolute terms, considering both (i) the sizes of paths/nodes in the path array and (ii) the sizes of inverted lists.

Let $|\Psi|$ be the number of nodes in the path array, and let m be the number of paths in the path array. Observe that an attribute with the domain size x can be stored as a binary string of $\mathcal{I}_x = \lceil \log_2 x \rceil$ bits.

In the path array, each node can be represented by $\mathcal{I}_{|V|}$ bits. Thus, the path array occupies $|\Psi| \cdot \mathcal{I}_{|V|}$ bits. In each inverted list, each path ID can be represented by $\mathcal{I}_m$ bits. Note that the total number of path IDs in inverted lists equals $|\Psi|$. Thus, the inverted lists occupy $|\Psi| \cdot \mathcal{I}_m$ bits. In summary, the total size of the structure is: $|\Psi| \cdot (\mathcal{I}_{|V|} + \mathcal{I}_m)$ bits.

3.4.2 Compact Cache via a Subgraph Model

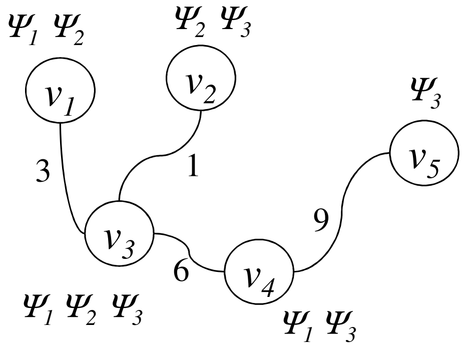
We propose a cache structure that consists of a subgraph G_Ψ (see Figure 3.10a) and inverted lists (see Figure 3.10b). The same inverted lists as in Figure 3.9b are used in Figure 3.10b. The main difference is that the path array in Figure 3.9a is now replaced by a subgraph structure G_Ψ that stores the adjacency lists of nodes that appear in the cache. The advantage of the subgraph structure is that each node (and its adjacency list) is stored at most once in G_Ψ.

v_1	v_3
v_2	v_3
v_3	v_1, v_2, v_4
v_4	v_3, v_5
v_5	v_4

(a) subgraph G_Ψ

v_1	Ψ_1, Ψ_2
v_2	Ψ_2, Ψ_3
v_3	Ψ_1, Ψ_2, Ψ_3
v_4	Ψ_1, Ψ_3
v_5	Ψ_3

(b) inverted lists



(c) visualization

Figure 3.10. Subgraph representation, with inverted lists

To check whether a query $Q_{s,t}$ can be answered by the cache, we just examine the inverted lists of v_s and v_t and follow the same procedure as in Section 3.4.1. There is a hit when the intersection of these two lists contains a path ID (say, Ψ_j). To find the result path $P_{s,t}$, we start from source v_s and visit a neighbor node v' whenever the inverted list of v' contains Ψ_j.

Figure 3.10c visualizes the structures in Figures 3.10a,b. Note that the subgraph G_Ψ only contains the adjacency lists of nodes that appear in the cache. Take query $Q_{2,4}$ as an example. First, we check the inverted lists of v_2 and v_4 in Figure 3.10b. Their intersection contains a path ID (Ψ_3). We then start from the source v_2 and visit its neighbor v_3 whose inverted list contains Ψ_3. Next, we examine the unvisited neighbors of v_3, i.e., v_1 and v_4. We ignore v_1 as its inverted list does not contain Ψ_3. Finally, we visit v_4 and reach the target. During the traversal, we obtain the shortest path: $\langle v_2, v_3, v_4 \rangle$.

Cache size analysis: We proceed to analyze the size of the above cache structure. As in Section 3.4.1, let $|\Psi|$ be the number of nodes in the cache and let m be the number of paths in the cache. Let $V_\Psi \subset V$ be the number of distinct nodes in the cache and let e be the average number of neighbors per node.

The inverted lists take $|\Psi| \cdot \mathcal{I}_m$ bits, as covered in the last Section. The subgraph occupies $|V_\Psi| \cdot e \cdot \mathcal{I}_{|V|}$ bits. Thus, the total size of the structure is: $|V_\Psi| \cdot e \cdot \mathcal{I}_{|V|} + |\Psi| \cdot \mathcal{I}_m$ bits.

Note that $|V_\Psi|$ is upper bounded by $|V|$ and that it is independent of the number of paths m in the cache. Thus, the subgraph representation is more compact than the structure in Section 3.4.1. The saved space can be used for accommodating additional paths into the cache, in turn improving the benefit of the cache.

3.4.3 Compact Inverted Lists

We present two compression techniques for reducing the space consumption of inverted lists. These are orthogonal and can be combined to achieve better compression. Again, the saved space can be used to accommodate more paths in the cache.

Interval path ID compression: This technique represents a sequence of consecutive path IDs $\Psi_i, \Psi_{i+1}, \Psi_{i+2}, \cdot, \Psi_j$ as an interval of path IDs $\Psi_{i,j}$. In other words, $j - i + 1$ path IDs can be compressed into 2 path IDs. The technique can achieve significant compression when there are long consecutive sequences of path IDs in inverted lists.

Figure 3.11a shows the original inverted lists, and Figure 3.11b shows the compressed inverted lists obtained by this compression. For example, the inverted list of v_3 (Ψ_1, Ψ_2, Ψ_3) is compressed into the interval $\Psi_{1,3}$.

v_1	Ψ_1, Ψ_2
v_2	Ψ_2, Ψ_3
v_3	Ψ_1, Ψ_2, Ψ_3
v_4	Ψ_1, Ψ_3
v_5	Ψ_3

(a) original

v_1	Ψ_{1-2}
v_2	Ψ_{2-3}
v_3	Ψ_{1-3}
v_4	Ψ_1, Ψ_3
v_5	Ψ_3

(b) interval compressed

	content	parent
v_1	Ψ_1, Ψ_2	NIL
v_2	$\cdots$	$\cdots$
v_3	Ψ_3	v_1
v_4	$\cdots$	$\cdots$
v_5	$\cdots$	$\cdots$

(c) prefix compressed

Figure 3.11. Compressed inverted lists

Prefix path ID compression: This technique first identifies inverted lists that share the same prefix and then expresses an inverted list by using the other inverted list as a prefix.

Consider the original inverted lists in Figure 3.11a. The inverted list of v_1 is a prefix of the inverted list of v_3. Figure 3.11c shows the compressed inverted lists produced by this compression. In the compressed inverted list of v_3, it suffices to store path IDs (e.g., Ψ_3) that do not appear in its prefix. The remaining path IDs of v_3 can be retrieved from the parent (v_1) of its inverted list.

3.5 Experimental Study

We proceed to evaluate the performance of our caching methods and the competitors on real datasets. The competitors, Least-Recently-Used (LRU) and Highest-Query-Frequency (HQF), are the dynamic and static caching methods introduced in Section 3.2.3. Our methods Shortest-Path-Cache (SPC) and its optimized variant (SPC*) share the same techniques in Section 3.3. Regarding the cache structures, LRU, SPC, and HQF use a path array cache (Section 3.4.1), whereas SPC* uses a compressed graph cache (Sections 3.4.2 and 3.4.3). All methods answer queries by using the optimal subpath property (Lemma 3.1). We implemented all methods in C++ and conducted experiments on an Intel i7 3.4GHz PC running Debian.

Section 3.5.1 covers the experimental setting. Then Section 3.5.2 and Section 3.5.3 presents findings for caches in the proxy scenario and the server scenario, respectively.

3.5.1 Experimental Setting

Datasets: Due to the privacy policies of online shortest path services (e.g., the Google Directions API), their query logs are unavailable in the public. Thus, we attempt to simulate query logs from trajectory data. For each trajectory, we extract its start and end locations as the source v_s and target v_t of a shortest path query, respectively. In the experiments, we used two real datasets (Aalborg and Beijing) as shown in Table 3.6. Each dataset consists of (i) a query log derived from a collection of trajectories, and (ii) a road network for the corresponding city.

Following the experimental methodology of static web caching [10], we divide a query log into two equal sets: (i) a historical query log $\mathcal{QL}$, for extracting query statistics, and (ii) a query workload $\mathcal{WL}$, for measuring the performance of caching methods. Prior to running a workload $\mathcal{WL}$, the cache of LRU is empty whereas the caches of HQF, SPC, and SPC* are built by using $\mathcal{QL}$.

Dataset	Trajectories	Road network
Aalborg	Infati GPS data [32] 4,401 trajectories	From *downloads.cloudmade.com* 129k nodes, 137k edges
Beijing	Geo-Life GPS data [33] 12,928 trajectories	From *downloads.cloudmade.com* 76k nodes, 85k edges

Table 3.6. Description of real data sets

Default parameters: Since the Aalborg and Beijing query logs are not large, we use scaled down cache sizes in the experiments. Thus, the default cache size is 625 kBytes and the maximum cache size is 5 MBytes. The default number of levels in the kD-tree (for query statistics extraction) is 14.

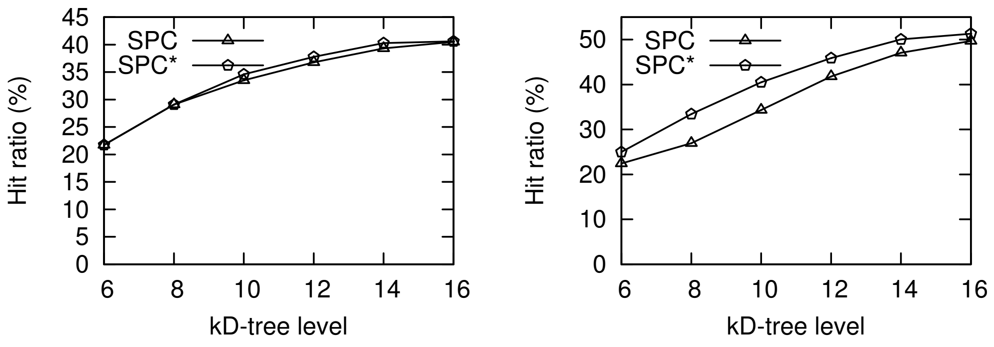
If we had access to huge query logs from online shortest path services, we would have used a large cache size, e.g., 1 GBytes.

3.5.2 Caching in the Proxy Scenario

In the proxy scenario, the shortest path API issues a query to the server, rather than computing the result by itself (see Section 3.3.3). Since its response time is dominated by the round-trip time with server, the *cache hit ratio* is used as the performance measure in this scenario.

Effect of the kD-tree level: Figure 3.12 plots the hit ratio of our methods

(SPC and SPC*) with respect to the number of levels in the kD-tree. The more levels the kD-tree contains, the more accurate its query statistics become, and this improves the hit ratio of our methods. Note that SPC* performs better than SPC on both datasets. Since SPC* has a more compact cache structure than SPC, it accommodates more shortest paths and thus achieving a higher hit ratio.

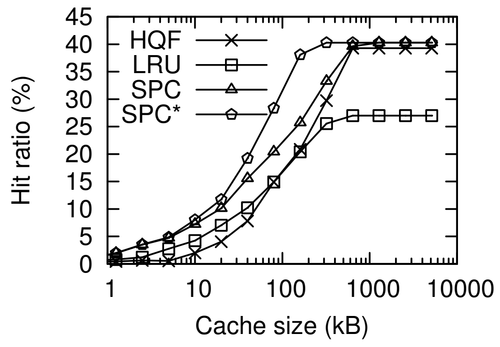


(a) Aalborg (b) Beijing

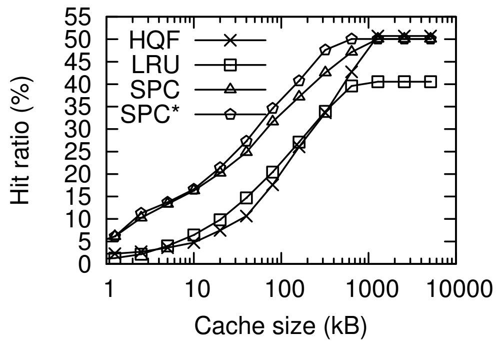
Figure 3.12. Hit ratio vs. levels

Effect of the cache size: Figure 3.13 shows the hit ratio of the methods as a function of the cache size. SPC has a lower hit ratio than SPC* for the reasons explained above. Observe that SPC* achieves double the hit ratio of HQF and LRU for small cache sizes (below 100 kBytes). This is because SPC* exploits historical query statistics to choose paths with high benefit values for inclusion into the cache. Our benefit model (Section 3.3.1) considers the number of historical queries answerable by (a subset of) a path $P_{a,b}$, not just the number of historical queries identical to $P_{a,b}$ (in HQF). At a large cache size (beyond 1000 kBytes), all static caching methods (SPC, SPC*, HQF) have similar hit ratios as the caches can accommodate all shortest paths from the historical query log.

Figure 3.14 shows the hit ratio of the methods versus the number of processed queries in the workload, at the largest cache size (5 MBytes). The hit ratio in this figure is measured with respect to the number of processed queries so far. Static caching is able to obtain a high hit ratio in the beginning. The hit ratio of dynamic caching (LRU) increases gradually and then converges to its final hit ratio.

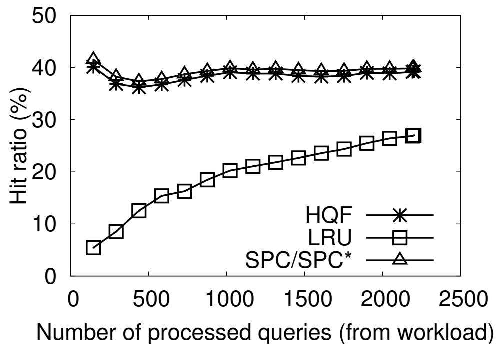


(a) Aalborg

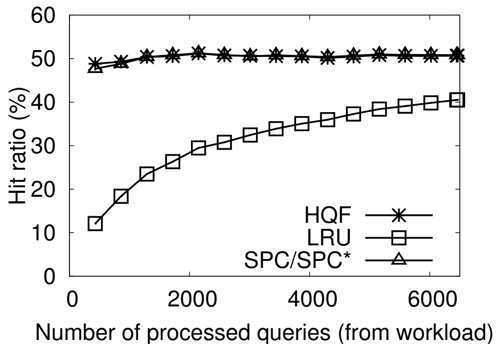


(b) Beijing

Figure 3.13. Hit ratio vs. cache size



(a) Aalborg



(b) Beijing

Figure 3.14. Hit ratio vs. processed queries (5 MBytes cache)

Cache construction time: Our methods incur low overhead on selecting cache items in the offline phase. SPC and SPC* require at most 3 minutes on both of the Aalborg and Beijing datasets in this phase. Since the cache structure of SPC is simpler than that of SPC*, its cache construction time is 50% of SPC*.

3.5.3 Caching in the Server Scenario

In the server scenario, the shortest path API invokes a shortest path algorithm (e.g., Dijkstra, A*) to compute a query result. The performance of a caching method C on a query workload is measured as (i) the total number of visited nodes, $nodes(C)$, and (ii) the total query time (including cache lookup overhead), $time(C)$.

As a reference for comparison, we consider a no-caching method NC that simply executes every query in the workload. Table 3.7 shows the total query time and the total number of visited nodes of NC (on the entire query workload), for each combination of dataset (Aalborg, Beijing) and shortest path algorithm (Dijkstra, A*).

Dataset	Shortest path algorithm	Query time (s) $time(NC)$	Visited nodes $node(NC)$
Aalborg	Dijkstra	18.5	18,580,659
Beijing	Dijkstra	43.5	41,615,917
Aalborg	A*	4.5	2,929,128
Beijing	A*	9.8	6,544,620

Table 3.7. Total query time and visited nodes on the entire workload, for the no-caching method

For a caching method C, we define its *visited nodes savings ratio* as $100\% \cdot (1 - (nodes(C)/nodes(NC)))$, and we define its *query time savings ratio* as $100\% \cdot (1 - (time(C)/time(NC)))$. We measure these ratios in the subsequent experiments.

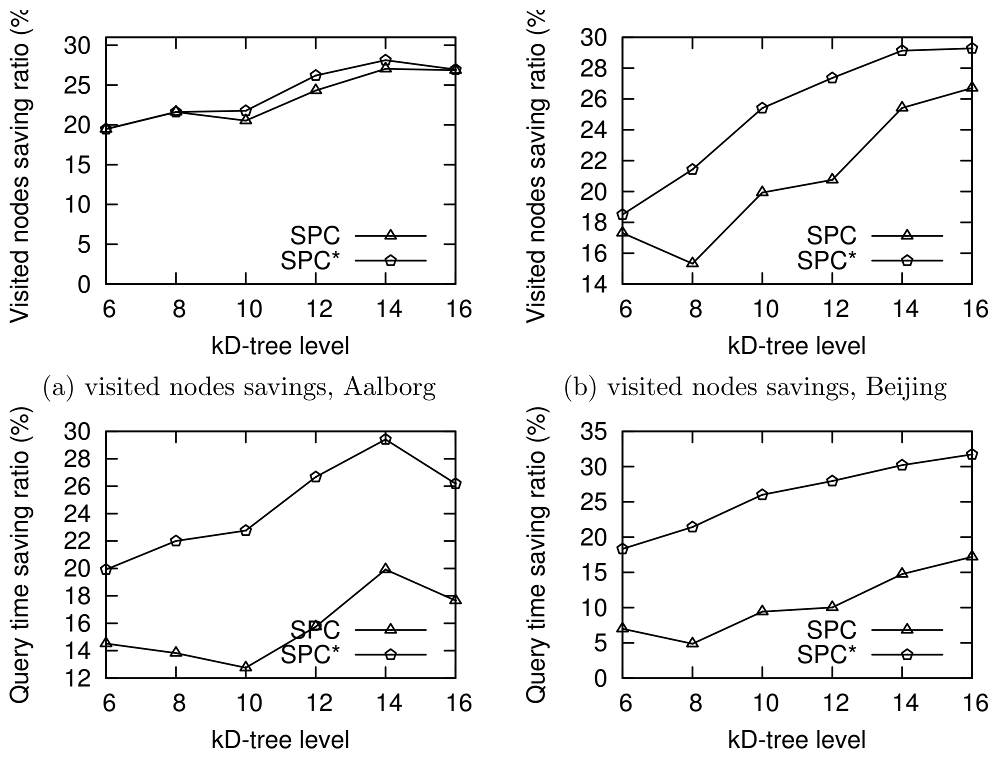
Effect of the kD-tree level: We investigate the effect of the number of levels in the kD-tree on the savings ratios of our methods (SPC and SPC*).

Figure 3.15 plots the savings ratios when Dijkstra is used as the shortest path algorithm. Like the trends in Figure 3.12, both methods obtain higher node savings ratios when the kD-tree used contains many levels and captures more accurate statistics (see Figure 3.15a,b).

However, there is a significant difference in the query time savings ratios (see Figure 3.15c,d). This is because the path array cache (used in SPC) incurs a high cache lookup overhead—all paths in the cache need to be examined when a query cannot be answered by the cache. On the other hand, the inverted lists (used in SPC*) support efficient cache lookup. SPC* achieves up to 30% savings of query time whereas SPC saves only up to 15-20% of query time.

Figure 3.16 shows the savings ratios of our methods when A* is used instead of Dijkstra. The node savings ratios of SPC and SPC* in Figure 3.16a,b exhibit similar trends as seen in Figure 3.15a,b. Note that the query time savings ratio of SPC is negative (see Figure 3.16c,d), meaning that it is slower than the no-caching method. Recall that the no-caching method requires only little running time when using A* for shortest path search (see Table 3.7). Thus, the high cache lookup overhead of SPC outweighs the benefit of caching. On the other hand, SPC* supports efficient lookup, achieving savings of up to 26% and 32% of the query time on the Aalborg and Beijing datasets.

In summary, SPC* consistently performs better than SPC, especially for the query time savings ratio.

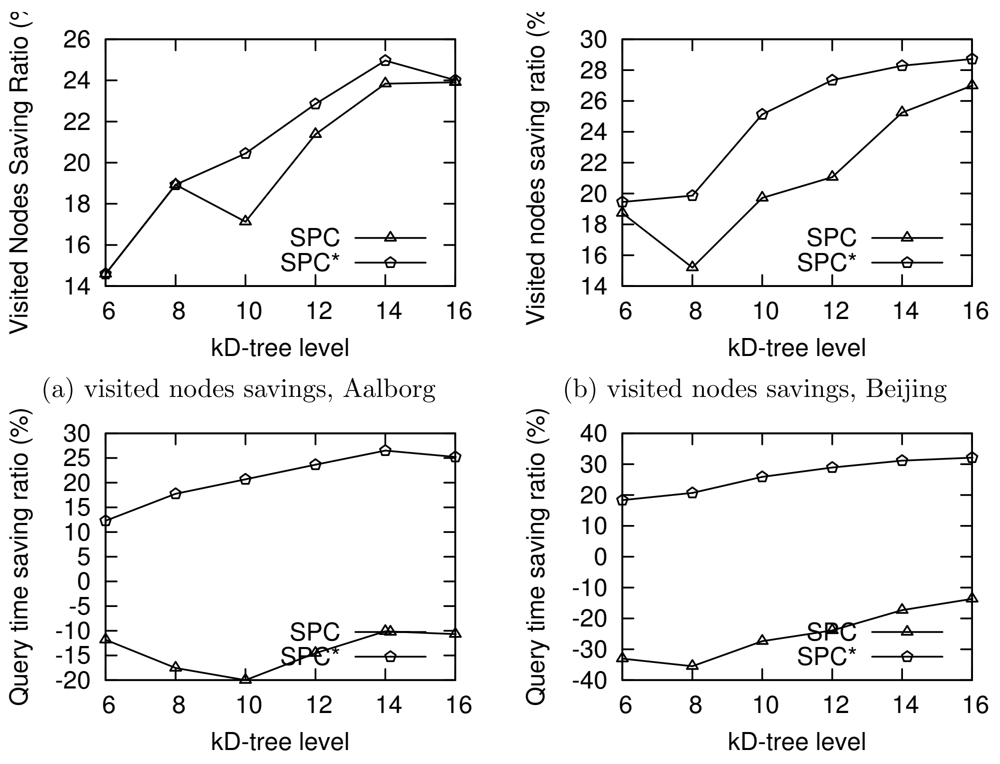


(a) visited nodes savings, Aalborg
(b) visited nodes savings, Beijing
(c) query time savings, Aalborg
(d) query time savings, Beijing

Figure 3.15. Performance savings vs. levels, using Dijkstra

Effect of the cache size: We proceed to study the impact of the cache size on the performance savings ratios of all caching methods (SPC, SPC*, HQF, LRU).

Figure 3.17 plots savings ratios when Dijkstra is used. At low cache sizes, SPC* outperforms the other methods in terms of the visited nodes savings ratio (see Figure 3.17a,b). At large cache sizes (beyond 1000 kBytes), all static caching methods (HQF, SPC and SPC*) have the same visited nodes savings ratio (28%). In Figure 3.17c,d, the query time savings ratios of all methods increase when the cache size increases. However, at large cache sizes, the query time savings ratios



(a) visited nodes savings, Aalborg

(b) visited nodes savings, Beijing

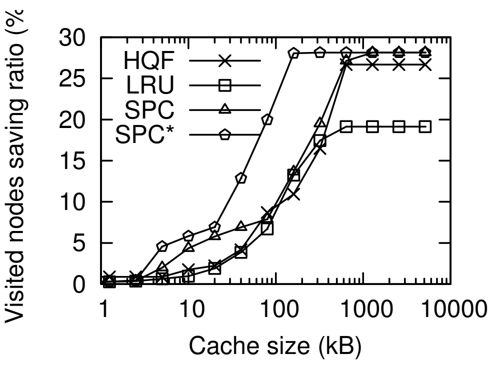
(c) query time savings, Aalborg

(d) query time savings, Beijing

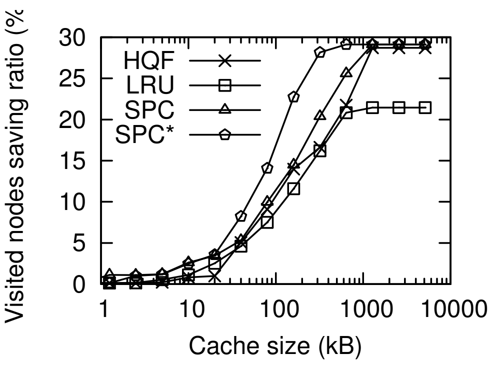
Figure 3.16. Performance savings vs. levels, using A^*

of HQF, LRU, and SPC drop slightly. Since they use the path array cache structure, their cache lookup overhead increases with the number of paths in the cache, reducing the overall utility of the cache. SPC* performs significantly better than the others, and its query time savings ratio remains comparable to its visited nodes savings ratio in Figure 3.17a,b.

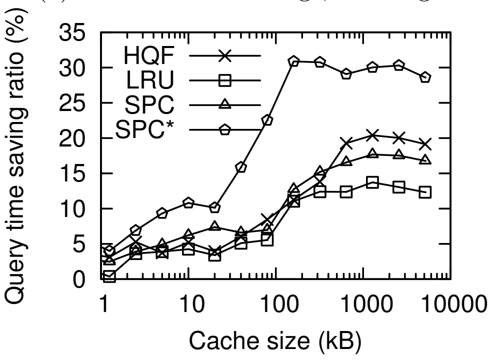
Figure 3.18 plots the savings ratio of the caching methods when using A*. The trends of the methods are similar to those in Figure 3.17. The only difference is that the query time savings ratio of HQF, LRU, and SPC are even more negative in Figure 3.18c,d. Since A* is much more efficient than Dijkstra, the high cache lookup overheads of HQF, LRU, and SPC become more apparent. They



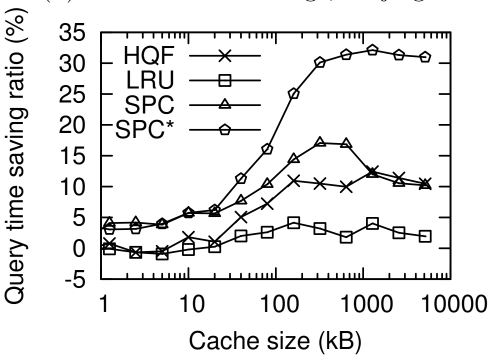
(a) visited nodes savings, Aalborg



(b) visited nodes savings, Beijing



(c) query time savings, Aalborg



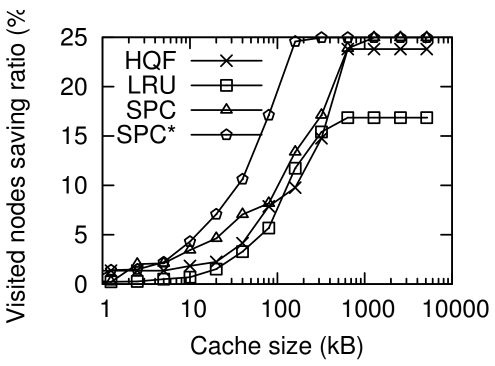
(d) query time savings, Beijing

Figure 3.17. Performance savings vs. cache size, using Dijkstra

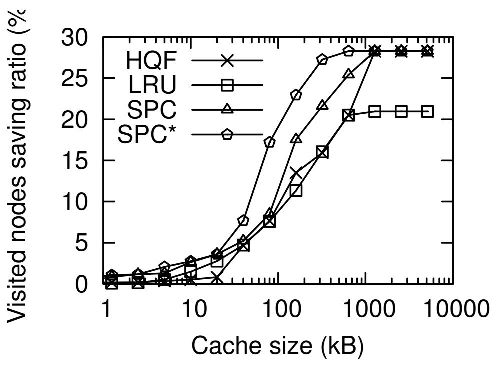
exhibit up to an additional 50-60% longer query time on the Beijing data set. SPC* remains the best method, and its query time savings ratio is comparable to its visited nodes savings ratio.

3.6 Conclusion

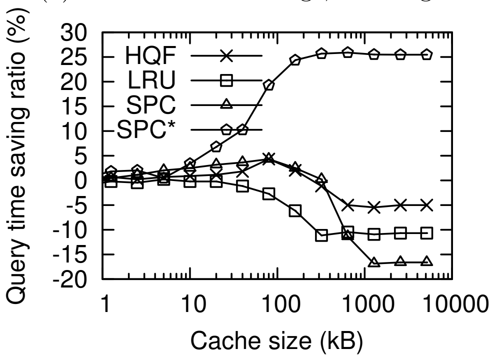
We study the caching of shortest paths in proxy and server scenarios. We formulate a model for capturing the benefit of caching a path in terms of its frequency and the cost of processing it. We develop techniques to extract query frequency statistics and to estimate the cost of an arbitrary shortest path algorithm. A greedy algorithm is proposed to select the most beneficial paths from



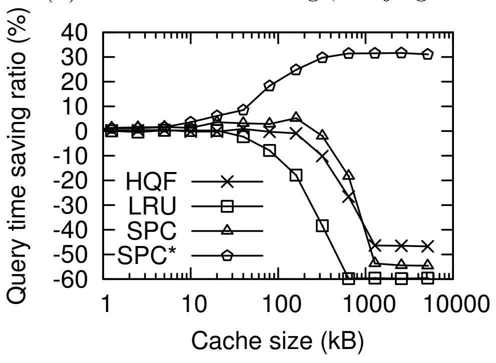
(a) visited nodes savings, Aalborg



(b) visited nodes savings, Beijing



(c) query time savings, Aalborg



(d) query time savings, Beijing

Figure 3.18. Performance savings vs. cache size, using A^*

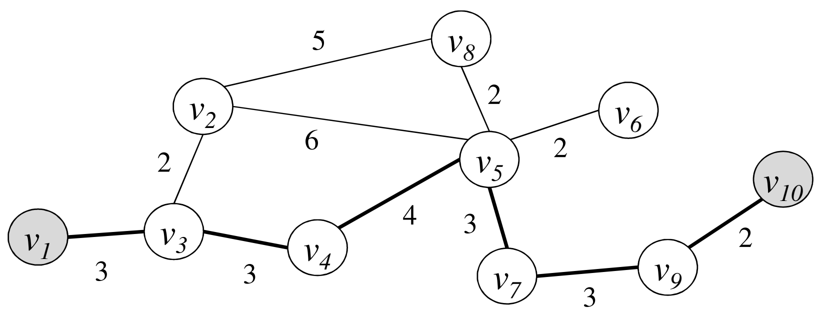
a historical query log for inclusion into the cache. Also, we provide cache structures that improve cache lookup performance and cache space utilization. Our experimental results on real data show that our best method, SPC*, achieves high hit ratio in the proxy scenario, as well as small lookup overhead and low query time in the server scenario.

Chapter 4

Concise Caching of Driving Instructions

Navigation services offer step-by-step navigation, which provide a driver with *driving instructions*[1] based on the driver's current GPS location [2]. We illustrate such driving instructions in Figure 4.1. Suppose that the driver starts at node v_1 and the target is node v_{10}. First, a detailed model uses point-by-point instructions, which essentially form the shortest path: $SP_{1,10} = \langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10} \rangle$. In contrast, a concise model uses turn-by-turn instructions that are necessary for navigation such as "continue" ($\Uparrow$) or "turn # degrees" ($\underline{\Rsh^{\#}}$), e.g., $\langle v_1, \underline{\Uparrow} v_5, \underline{\Rsh^{90}} \underline{\Uparrow} v_{10} \rangle$. These instructions combine a sequence of checkpoints (i.e. v_1, v_5, v_{10}) and turn instructions (i.e. $\underline{\Uparrow}$, $\underline{\Rsh^{90}}$). After leaving v_1, the user should continue ($\underline{\Uparrow}$) until reaching a checkpoint (v_5). Finally, the user should make a right turn ($\underline{\Rsh^{90}}$) and then continue ($\underline{\Uparrow}$) until reaching the target (v_{10}).

[1]These instructions can be spoken or given visually.



Shortest path (point-by-point):	$\langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10} \rangle$
Driving instructions (turn-by-turn):	$\langle v_1, \underline{\Uparrow} v_5, \underline{\overset{90}{\Rsh}\Uparrow} v_{10} \rangle$
Concise shortest path:	$\langle v_1, v_5, v_7, v_{10} \rangle$
Generic concise shortest path:	$\langle v_1, \mathbf{v_4}, v_5, v_7, v_{10} \rangle$

Figure 4.1. A road network, with a query from v_1 to v_{10}

Mobile users are increasingly using online driving direction services (rather than offline navigation software) as they are free of charge and do not require purchase and installation of up-to-date map data on mobile devices. Examples of driving direction APIs include Google Directions [4] and MapQuest Directions [3]. They can provide users with both shortest paths and turn-by-turn driving instructions.

The above services receive intensive workloads from mobile users on a daily basis. They can deploy web cache servers in order to reduce the network traffic [34, 35]. In our application context, we propose to cache driving instructions at web cache servers. For the sake of cache management, we represent concise driving instructions (with checkpoints and turn instructions) according to a unified format called *concise shortest path*. By replacing turn instructions by nodes, we obtain the following concise shortest path for our example: $CP_{1,10} = \langle v_1, v_5, v_7, v_{10} \rangle$. The default driving instruction is to go straight until reaching the next node in the concise shortest path (e.g., from v_1 to v_5). Upon reaching such a node (v_5), the driver checks whether the next such node (v_7) is adjacent; if yes, the user makes a turn there.

We elaborate on the construction of and navigation using such paths in Section 4.3. Figure 4.2 shows examples of a shortest path SP (crosses) and the corresponding concise shortest path CP (dots).

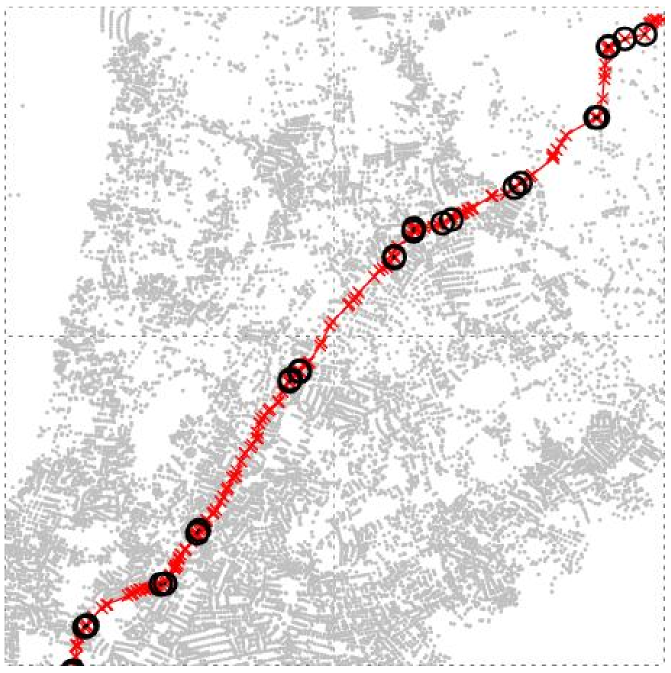


Figure 4.2. Shortest path (crosses) and concise shortest path (circles), in New York

The use of concise shortest paths in caching has two opposite effects on the cache hit ratio. Clearly, the cache can accommodate a larger number of concise paths than corresponding complete paths, suggesting a higher cache hit ratio. However, a concise path contains fewer nodes than its corresponding complete path and can answer fewer shortest path queries than the complete path. Consider the concise shortest path $\langle v_1, v_5, v_7, v_{10} \rangle$ in Figure 4.1. By the optimal subpath property [29], this path can answer any query whose end node lies on the path. Thus, this path can answer $Q_{5,10}$ (the query from v_5 to v_{10}) but not $Q_{4,10}$ (the query from v_4 to v_{10}).

Intuitively, if a query (say, $Q_{4,10}$) is frequent, then it is desirable to include its query nodes into a concise path. We use the term *generic concise shortest path* for a shortest path that extends a concise shortest path to include such frequent nodes. An example is the path $\langle v_1, \mathbf{v_4}, v_5, v_7, v_{10} \rangle$ as shown in Figure 4.1. Although such a path is slightly less compact than CP, it can answer one more frequent query (e.g., $Q_{4,10}$) than CP. In general, there are $2^{|SP|-|CP|}$ possible instances of generic shortest paths having the same start and end nodes. Which of them should be cached? Our key challenge is to select generic shortest paths such that the overall cache hit ratio is maximized.

Our contributions are:

- We propose the notion of a generic concise shortest path that enables a trade-off between the path size and the number of queries answerable by the path.
- We present a statistics-driven model for selecting generic concise paths in a static caching setting.
- We develop an adaptive technique for determining generic concise paths in a dynamic caching setting.

The rest of the chapter is organized as follows. Section 4.1 outlines the related work specific to this chapter. Section 4.2 describes the problem setting. Next, we introduce concise shortest paths in Section 4.3, and we present caching techniques for them in Sections 4.4 and 4.5. Then, we evaluate the performance of our proposed techniques in Section 4.6. Finally, Section 4.7 offers conclusions and rounds off the chapter.

4.1 Related Work

The related work relevant to this chapter is described in Chapter 2, (see Table 2.1), and in Section 4.1.1.

The related work of this chapter and Chapter 3 overlap. Refer to Section 3.1 for the differences between this chapter and the related work presented in Chapter 2.

4.1.1 Compact representations of shortest paths

Existing work on Compact representations of shortest paths cover methods to represent a shortest path using fewer vertices than the original shortest path.

The K-skip shortest path [36] is a path that contains at least one out of every K consecutive nodes in the corresponding shortest path. In Figure 3.2, the 3-skip shortest path of $SP_{1,10}$ is: $SKIP_{1,10} = \langle v_1, v_5, v_{10} \rangle$. Unfortunately, unlike concise shortest paths, K-skip shortest paths are lossy and may provide ambiguous driving instructions to the user. In practice, during driving, only a small portion of the road network is within the driver's eyesight. For example, when the user reaches v_3, the above 3-skip path $SKIP_{1,10}$ does not contain sufficient information to decide whether to go to v_2 or to v_4.

Batz et al. [37] develop a more effective compression method for shortest paths by using a shortest path index. However, this method requires installing a shortest path index at the client side in order to decode the compressed path correctly. Otherwise, the client cannot obtain unambiguous driving instructions.

Recently, Gotsman et al. [38] propose to map-match a trajectory to a path on a road network (e.g., $\langle v_{x_1}, v_{x_2}, \cdots, v_{x_n}\rangle$), then decompose it into a sequence of shortest paths: $SP_{a_1,a_2}, SP_{a_2,a_3}, \cdots, SP_{a_{m-1},a_m}$, and finally keep only the intermediate source and target nodes in the sequence ($\langle v_{a_1}, v_{a_2}, \cdots, v_{a_m}\rangle$).

However, if the user does not have the entire road network, then the sequence cannot be decoded into the original path, and the approach is therefor not suitable for mobile users.

4.2 Problem Setting

We first provide definitions for our problem, and then we introduce our objectives. Table 4.1 provides an overview of the notations used throughout the chapter.

Symbol	**Meaning**
$G(V, E, W)$	a graph
v_i	a node in the node set V
(v_i, v_j)	an edge in the edge set E
$Q_{s,t}$	a shortest path query from node v_s to node v_t
$SP_{s,t}$	a shortest path result of $Q_{s,t}$
$\lvert SP_{s,t}\rvert$	the size of $SP_{s,t}$ (in number of nodes)
CP	a concise shortest path
GCP	a generic concise shortest path
Ψ	the cache

Table 4.1. Table of symbols

Definition 4.1 (Graph) *Let $G(V, E, W)$ be a directed spatial graph with a set V of nodes and a set $E \subseteq V \times V$ of edges. Each node $v_i \in V$ models a road junction and has lat-long coordinates. Each edge $(v_i, v_j) \in E$ models a road segment, and $W{:}E \rightarrow R$ assigns a positive weight to each edge.*

Definition 4.2 (Query and Result) *A shortest path query, denoted by $Q_{s,t}$, consists of a source and a target node, v_s and v_t. The result of $Q_{s,t}$, denoted by $SP_{s,t}$, is a sequence of nodes from v_s to v_t such that it has the smallest sum of edge weights (among all paths from v_s to v_t in G).*

By the optimal subpath property [29], any subpath of a shortest path SP is also a shortest path. Thus, a query $Q_{s,t}$ can be answered by SP if $v_s, v_t \in SP$ and v_s appears before v_t in SP. We denote this event by: $Q_{s,t} \subset SP$. We proceed to define the concept of cache hit.

Definition 4.3 (Cache hit) *Let a cache Ψ be a set of shortest paths. A query $Q_{s,t}$ obtains a hit from Ψ if there exists some $SP \in \Psi$ such that $v_s, v_t \in SP$ and v_s appears before v_t in SP. We denote this event by: $Q_{s,t} \subset \Psi$.*

Definition 4.4 (Cache size) *Let $|P|$ denote the size (number of nodes) of a path. The total size of the cache is defined as: $||\Psi|| = \sum_{P \in \Psi} |P|$.*

We illustrate the above definitions with the example in Figure 4.1. Suppose that a cache contains two shortest paths: $\Psi = \{SP_{1,10}, SP_{2,6}\}$, where $SP_{1,10} = \langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10} \rangle$ and $SP_{2,6} = \langle v_2, v_5, v_6 \rangle$. The cache size is: $||\Psi|| = |SP_{1,10}| + |SP_{2,6}| = 7 + 3 = 10$. Consider the queries $Q_{3,7}$ and $Q_{1,2}$.

Query $Q_{3,7}$ obtains a cache hit because the cached path $SP_{1,10}$ contains both query nodes v_3 and v_7. On the other hand, the query $Q_{1,2}$ does not obtain a cache hit.

We adopt the standard client-server architecture shown in Figure 4.3 as the setting. A cache server is placed in-between the mobile clients and an online shortest path service (e.g., Google Directions [4]). Upon receiving a shortest path query $Q_{s,t}$, the cache server checks whether there is a cache hit. If yes then it returns the result. Otherwise, it forwards $Q_{s,t}$ to the online shortest path service and eventually returns the result path $P_{s,t}$ computed by the service.

In this chapter, the dominant cost is the network traffic between the cache server and the shortest path service. Thus, our objective is to **maximize the hit ratio of the cache server** (subject to a given cache capacity) [14, 34, 39]. While the shortest path service may deploy a shortest path index [40–43] for its own performance reasons, the introduction of such an index does not improve the cache hit ratio and is orthogonal to our work.

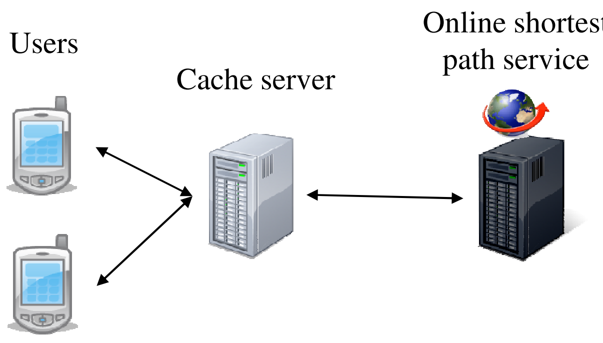


Figure 4.3. Client server architecture

Unlike our previous work [11], this chapter utilizes *concise shortest paths* to improve the hit ratio of caching. We elaborate on concise shortest paths and their operations in Section 4.3, and we present caching methods for them in Sections 4.4 and 4.5.

As a remark, two issues are orthogonal to this chapter. First, we reuse existing data structures for the caching of shortest paths [11] to reduce the CPU overhead for maintaining cache structures. Second, we assume that the edge weights $W(v_i, v_j)$ maintained at the online shortest path service are independent of time. This is the case for edge weights that model travel distances (e.g., in MapQuest). If this is not the case, we may associate each cache item (shortest path) with a expiry time [44].

4.3 Concise Shortest Paths

For navigation purposes, it suffices for mobile users to know driving instructions (e.g., going straight, making turns) instead of complete shortest paths. As discussed in the introduction, the notion of a concise shortest path is equivalent to driving instructions.

We first provide definitions for concise shortest paths. Then, we propose a server-side algorithm for converting a shortest path into a concise shortest path. Also, we present a client-side algorithm that enables a user to navigate using a concise shortest path. Finally, we introduce generic concise shortest paths.

4.3.1 Definition and Examples

We consider three possible driving instructions for mobile users, as illustrated in Figure 4.4. Suppose that the user is reaching a current node v_c from a previous node v_p. In each case in the figure, the shortest path is indicated by bold arrows.

- Case I: The driving instruction is `'continue'` to the next node v_n as this is the only option.
- Case II: There is more than one option (e.g., travel to v_{n1} or to v_{n2}). These options can be distinguished by their deviation angles from the previous travel direction (see Definition 4.5). In this figure, the deviation angles of (v_c, v_{n1}) and (v_c, v_{n2}) are $\delta_{p,c,n1}$ and $\delta_{p,c,n2}$, respectively. If the correct choice is the one with the smallest deviation angle, like in this figure, the driving instruction is `'go straight, by the smallest angle'`.
- Case III: The correct choice is not the option with the smallest deviation angle. In this case, we must provide an explicit driving instruction: `'turn clockwise/anticlockwise by X`^{o}`'`, where X^o is the deviation angle $\delta_{p,c,n}$ (see Definition 4.5).

The above driving instructions are not explicitly stored in a concise shortest path. In Section 4.3.2, we provide a client-side algorithm to reconstruct driving instructions from a concise shortest path, during navigation.

Definition 4.5 (Deviation angle) *Given two adjacent edges (v_p, v_c) and (v_c, v_n), the deviation angle between them is:*

$$\delta_{p,c,n} = |180^o - \measuredangle v_p v_c v_n| \tag{4.1}$$

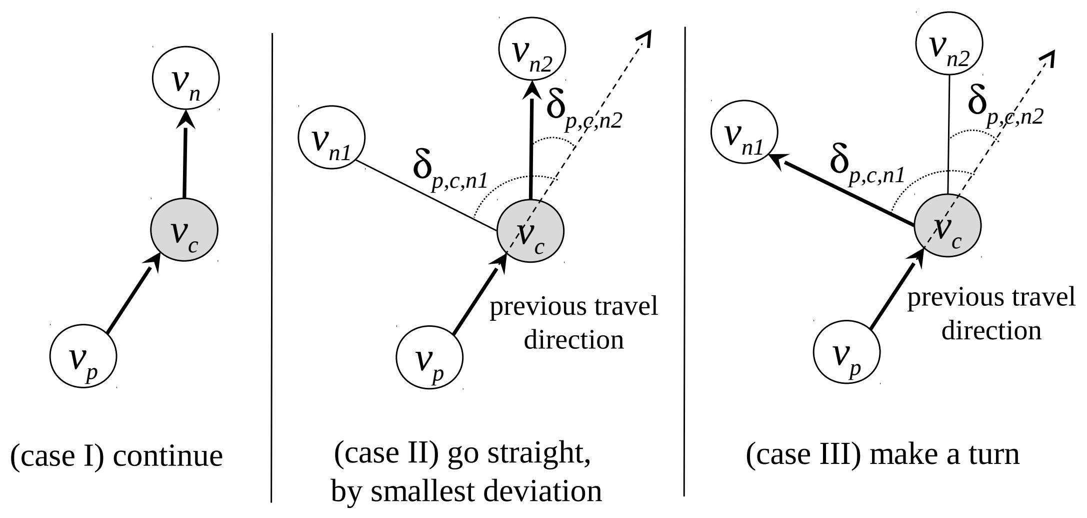


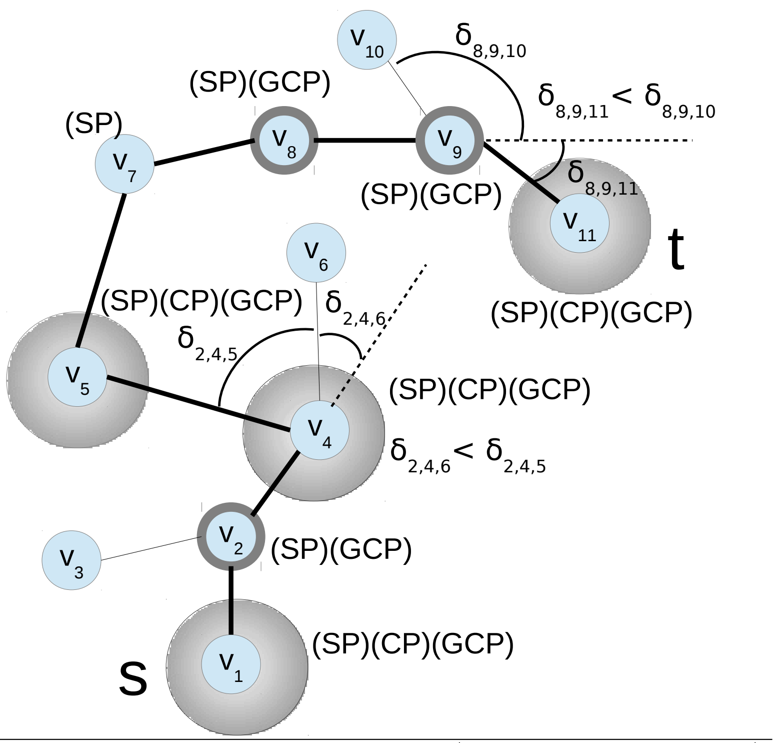
Figure 4.4. Possible driving instructions; the shortest path is indicated by bold arrows

Based on the above driving instructions, we define a concise shortest path as follows.

Definition 4.6 (Concise shortest path) .
Let a shortest path $SP_{s,t} = \langle v_{\rho(1)}, v_{\rho(2)}, \cdots, v_{\rho(l)} \rangle$ be given, where $v_{\rho(i)}$ is the i-th node in $SP_{s,t}$, $s = \rho(1)$, and $t = \rho(l)$. A concise shortest path $CP_{s,t}$ is a subsequence of $SP_{s,t}$ such that: (i) $CP_{s,t}$ contains v_s and v_t, and (ii) $CP_{s,t}$ contains $v_{\rho(i)}, v_{\rho(i+1)}$ if nodes $v_{\rho(i-1)}, v_{\rho(i)}$ and, $v_{\rho(i+1)}$ satisfy case III.

We illustrate a concise shortest path in Figure 4.5. Let the source and target nodes be v_1 and v_{11}, respectively. The shortest path $SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11} \rangle$ is shown with bold edges. According to Definition 4.6, the concise shortest path is: $CP_{1,11} = \langle v_1, v_4, v_5, v_{11} \rangle$. First, it must contain both the source and target nodes (v_1 and v_{11}). Since $\langle v_2, \mathbf{v_4}, v_5 \rangle$ matches case III, v_4 and the next node (v_5)

are in $CP_{1,11}$. The remaining nodes either match case I (e.g., v_7, v_8) or case II (e.g., v_2, v_9), so they are not in $CP_{1,11}$. Observe that a concise path describes the driving instructions for $SP_{s,t}$ unambiguously. Directions are given exactly when it is necessary to change the travel direction.



Shortest path:	$SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11} \rangle$
Concise shortest path:	$CP_{1,11} = \langle v_1, v_4, v_5, v_{11} \rangle$
Generic concise shortest path:	$GCP_{1,11} = \langle v_1, \mathbf{v_2}, v_4, v_5, v_{11} \rangle$

Figure 4.5. Concise shortest path

Concise shortest paths have the advantage that they consume less space while still offering sufficient information for driving.

4.3.2 Operations

Server-Side: Concise Path Extraction We study caching methods for concise shortest paths in Sections 4.4 and 4.5. Given a shortest path SP, the cache server can execute Algorithm 4.1 to extract a concise shortest path from SP.

First, we specify the initial travel direction (in Lines 3–6). We add the two first nodes of SP to CP if the first node has multiple branches, i.e., $deg(v_{\rho(1)}) > 1$. Otherwise, there is only one option so it suffices to add the first node of SP to CP.

For each subsequent node $v_{\rho(i)}$ on SP, we compute the deviation angle to each adjacent node and then take the smallest deviation angle as δ^*. If the next node $v_{\rho(i+1)}$ on SP does not have the smallest deviation angle, we specify the travel direction by adding the current and the next nodes $(v_{\rho(i)}, v_{\rho(i+1)})$ of SP to SP. Finally, we add the last node of SP to CP.

Client-Side: Concise Path Navigation Upon receiving a concise shortest path CP from the cache server, the client can apply Algorithm 4.2 to navigate along CP. Note that this algorithm also works correctly for extended versions of CP, which we discuss below.

4.3.3 Generic Concise Shortest Paths

Although a concise shortest path occupies less space than a corresponding (complete) shortest path, it covers fewer nodes and thus it may answer a smaller number of queries. For example, suppose that the query $Q_{2,11}$ is a frequently-

Algorithm 4.1 Server:ExtractConcise(Shortest path SP)

```
1: let $v_{\rho(i)}$ be the $i$-th node in $SP$
2: $CP \leftarrow \langle\rangle$                                        ▷ the concise path
3: if $deg(v_{\rho(1)}) > 1$ then
4:     append $v_{\rho(1)}, v_{\rho(2)}$ to $CP$
5: else
6:     append $v_{\rho(1)}$ to $CP$
7: for $i$ from 2 to $|SP| - 1$ do
8:     if $deg(v_{\rho(i)}) > 2$ then
9:         $\delta^* \leftarrow \delta_{\rho(i-1),\rho(i),\rho(i+1)}$                 ▷ deviation angle
10:        for each node $v_j$ adjacent to $v_{\rho(i)}$ do
11:            if $v_j \neq v_{\rho(i-1)}$ and $v_j \neq v_{\rho(i+1)}$ then
12:                $\delta^* \leftarrow \min\{\delta^*, \delta_{\rho(i-1),\rho(i),j}\}$
13:        if $\delta^* < \delta_{\rho(i-1),\rho(i),\rho(i+1)}$ then              ▷ include nodes
14:            append $v_{\rho(i)}$ to $CP$ if it is not in $CP$
15:            append $v_{\rho(i+1)}$ to $CP$
16: append $v_{\rho(|SP|)}$ to $CP$ if it is not in $CP$                 ▷ last node
17: Return $CP$
```

used query in Figure 4.5. Observe that the shortest path $SP_{1,11}$ can answer $Q_{2,11}$, while the concise path $CP_{1,11}$ cannot do so as it does not contain v_2.

To address this issue, we consider generic versions of concise shortest paths that include additional nodes:

Definition 4.7 (Generic concise shortest path) *A sequence $GCP_{s,t}$ is said to be a generic concise shortest path (from node v_s to node v_t) if it is a subsequence of $SP_{s,t}$ and a supersequence of $CP_{s,t}$.*

As an example, consider the map in Figure 4.5. Table 4.2 illustrates all generic concise shortest paths from source v_1 to target v_{11}. Note that the number of generic concise shortest paths is $2^{|SP_{s,t}|-|CP_{s,t}|} = 2^{8-4} = 16$. In the next section, we discuss how to select a generic concise shortest path in order to

Algorithm 4.2 Client:NavigateConcise(Concise path CP)

```
1: let $(v_{prev}, v_{cur})$ be the edge being traversed by the client
2: while $v_{cur} \neq v_t$ do                                  ▷ target not reached
3:     if $v_{cur}$ is not in $CP$ then                                ▷ go straight
4:         $\delta^* \leftarrow 180^o$                       ▷ the smallest deviation angle
5:         for each node $v_j$ adjacent to $v_{cur}$ do
6:             if $v_j \neq v_{prev}$ then
7:                 if $\delta^* > \delta_{prev,cur,j}$ then
8:                     $\delta^* \leftarrow \delta_{prev,cur,j}$
9:                     $v_{next} \leftarrow v_j$                         ▷ keep the best option
10:        display "continue to $v_{next}$"
11:    else                                                            ▷ follow $CP$
12:        let $v_{next}$ be the next node of $v_{cur}$ in $CP$
13:        display "turn to $v_{next}$"
14:    wait until the client reaches $v_{next}$
15:    $(v_{prev}, v_{cur}) \leftarrow (v_{cur}, v_{next})$
```

maximize the cache hit ratio.

$CP_{1,11}$	$\langle v_1, v_4, v_5, v_{11} \rangle$	
other generic concise paths	$\langle v_1, v_2, v_4, v_5, v_{11} \rangle$	$\langle v_1, v_4, v_5, v_7, v_{11} \rangle$
	$\langle v_1, v_4, v_5, v_8, v_{11} \rangle$	$\langle v_1, v_4, v_5, v_9, v_{11} \rangle$
	$\langle v_1, v_2, v_4, v_5, v_7, v_{11} \rangle$	$\langle v_1, v_2, v_4, v_5, v_8, v_{11} \rangle$
	$\langle v_1, v_2, v_4, v_5, v_9, v_{11} \rangle$	$\langle v_1, v_4, v_5, v_7, v_8, v_{11} \rangle$
	$\langle v_1, v_4, v_5, v_7, v_9, v_{11} \rangle$	$\langle v_1, v_4, v_5, v_8, v_9, v_{11} \rangle$
	$\langle v_1, v_2, v_4, v_5, v_7, v_8, v_{11} \rangle$	$\langle v_1, v_2, v_4, v_5, v_7, v_9, v_{11} \rangle$
	$\langle v_1, v_2, v_4, v_5, v_8, v_9, v_{11} \rangle$	$\langle v_1, v_4, v_5, v_7, v_8, v_9, v_{11} \rangle$
$SP_{1,11}$	$\langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11} \rangle$	

Table 4.2. Generic concise shortest paths (from v_1 to v_{11})

4.4 Static Caching Setting

Static caching focuses on caching the most popular data items [7, 8, 14]. It utilizes a query log that records past queries in order to determine the access frequencies of data items. Static caching incurs low overhead at runtime; however, it may not adapt well to a varying query distribution.

We present a caching method for generic concise shortest paths that applies an existing static caching policy for shortest paths [11]. The frequencies of queries in the query log are used to define a 'benefit' score that captures the 'importance' of a path in answering queries. Our main challenge is to find generic concise paths that have high 'benefit' scores.

In Section 4.4.1, we give a benefit score model for generic concise paths. Then, in Section 4.4.2, we present a method for computing a generic concise path with a high benefit score. Finally, we propose an efficient implementation in Section 4.4.3.

4.4.1 Benefit Model

We adopt the benefit model presented in previous work [11] to quantify the importance of a path in answering queries. First, we provide definitions of query log and query frequency.

Definition 4.8 (Query log and frequency)
A query log $\mathcal{QL}$ is a collection of shortest path queries that have been issued by users in the past.
The query frequency $\chi_{s,t}$ of a query $Q_{s,t}$ is defined as the number of occurrences of $Q_{s,t}$ in the query log $\mathcal{QL}$.

$$\chi_{s,t} = |\{Q_{s,t} \in \mathcal{QL}\}| \tag{4.2}$$

Table 4.3 illustrates the values of $\chi_{s,t}$ derived from an example query log for the map in Figure 4.5. For simplicity, since we assume $\chi_{s,t} = \chi_{t,s}$.

$\chi_{s,t}$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}
v_1	/	0	**10**	**50**	0	0	0	**16**	0	0	0
v_2		/	0	0	0	0	**35**	0	0	0	**12**
v_3			/	0	**20**	0	0	0	0	0	0
v_4				/	**2**	0	0	0	0	0	**20**
v_5					/	0	**5**	0	0	**18**	0
v_6						/	0	0	0	0	0
v_7							/	0	0	0	0
v_8								/	0	0	**30**
v_9									/	0	0
v_{10}										/	0
v_{11}											/

Table 4.3. Example of $\chi_{s,t}$ values for the graph, with $\chi_{s,t} = \chi_{t,s}$

Given a generic concise shortest path P, we define its *benefit* $\Upsilon(P)$ as:

$$\Upsilon(P) = \sum_{v_i, v_j \in P,\ Q_{i,j} \subset P} \chi_{i,j} \tag{4.3}$$

The benefit is the number of queries (in $\mathcal{QL}$) that can be answered by P. As a remark, a simple (nested-loop) implementation for computing $\Upsilon(P)$ would take $O(|P|^2)$ time.

Then, we define the benefit of a cache Ψ as follows:

$$\Upsilon(\Psi) = \sum_{v_i, v_j \in V,\ Q_{i,j} \subset \Psi} \chi_{i,j} \tag{4.4}$$

As an example, consider the cache content in Table 4.4 and the map in Figure 4.5. The path $CP_{1,11}$ has benefit: $\Upsilon(CP_{1,11}) = \chi_{1,4} + \chi_{1,5} + \chi_{1,11} + \chi_{4,5} + \chi_{4,11} + \chi_{5,11} = 50 + 0 + 0 + 2 + 20 + 0 = 72$. The benefit of the cache Ψ is: $\Upsilon(\Psi) = \chi_{1,4} + \chi_{4,5} + \chi_{4,11} + \chi_{3,5} = 50 + 2 + 20 + 20 = 92$. Observe that

Equation 4.4 avoids duplicate counting. For example, the term $\chi_{4,5}$ appears only once in the equation of $\Upsilon(\Psi)$, despite that the query $Q_{4,5}$ can be answered by paths $CP_{1,11}$, $CP_{3,8}$, and $CP_{4,9}$.

	concise path CP
$CP_{1,11}$	$\langle v_1, v_4, v_5, v_{11} \rangle$
$CP_{3,8}$	$\langle v_3, v_4, v_5, v_8 \rangle$
$CP_{4,9}$	$\langle v_4, v_5, v_9 \rangle$

Table 4.4. A cache Ψ of concise shortest paths

With the above benefit model, we formulate our static caching problem as follows:

PROBLEM 4.1 (STATIC CACHING PROBLEM) *Select a set of generic concise shortest paths $\Psi = \{P_{s,t}\}$ such that: (i) the benefit $\Upsilon(\Psi)$ is maximized, and (ii) the cache size $||\Psi||$ is bounded by a given cache capacity value.*

In previous work [11], we require each $P_{s,t}$ to be a shortest path. Here, we consider a much larger search space and allow each $P_{s,t}$ to be a generic concise shortest path.

4.4.2 Benefit-Based Generic Concise Path

This section presents a solution for computing a generic concise path. Specifically, given a shortest path $SP_{s,t}$, we want to construct a generic concise path $GCP_{s,t}$ such that it has a high benefit and a small size.

Given the paths present in the cache Ψ, we define the *marginal benefit* of a path P with respect to Ψ as:

$$\begin{aligned} \Upsilon_m(P \setminus \Psi) &= \Upsilon(\Psi \cup \{P\}) - \Upsilon(\Psi) \\ &= \sum_{v_i, v_j \in P,\ Q_{i,j} \in P,\ Q_{i,j} \not\subset \Psi} \chi_{i,j} \end{aligned} \tag{4.5}$$

This notion captures the total frequencies of queries (in $\mathcal{QL}$) that can be answered by P but not by paths in Ψ. Intuitively, it is desirable to cache a path P if the benefit $\Upsilon_m(P \setminus \Psi)$ is high and the size $|P|$ is small. Thus, we define the *normalized benefit* of a path P as: $\overline{\Upsilon_m}(P \setminus \Psi) = \frac{\Upsilon_m(P \setminus \Psi)}{|P|}$.

A brute-force approach is to enumerate all possible generic concise shortest paths (like in Table 4.2) and then find the one with the highest $\overline{\Upsilon_m}$ value. However, as discussed in Section 4.3.3, the number of possible generic concise shortest paths is exponential in the path size.

In the following, we present a greedy heuristic solution to solve this problem in polynomial time. Algorithm 4.3 is the pseudo-code of this solution. It takes a shortest path $SP_{s,t}$ and its corresponding concise path $CP_{s,t}$ as input. We denote BP as the path with the highest $\overline{\Upsilon_m}$ value found so far. It is initialized to $CP_{s,t}$. The set S contains the possible nodes that can be added to BP. In each iteration, we compute the benefit of the path $BP \cup \{v_c\}$ for each v_c (Lines 5–9), and find the node (say v_{best}) with the highest benefit. If the normalized benefit of $BP \cup \{v_{best}\}$ is higher than that of BP, we add v_{best} to BP and repeat the loop. Otherwise, the algorithm terminates.

We illustrate the working of Algorithm 4.3 in Table 4.5. Here, we are given a shortest path $SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11} \rangle$ and its corresponding concise

Algorithm 4.3 Static-Greedy (Shortest path SP, Concise shortest path CP, Cache Ψ)

```
1: $BP \leftarrow CP$                                        ▷ the best path found so far
2: $S \leftarrow$ the set of nodes in $SP - CP$
3: $\gamma_{cur} \leftarrow \Upsilon_m(BP \setminus \Psi)$
4: while $S \neq \emptyset$ do
5:     $\gamma_{best} \leftarrow 0$                          ▷ the best score in this iteration
6:     for each $v_c \in S$ do
7:         $\gamma_c \leftarrow \Upsilon_m((BP \cup \{v_c\}) \setminus \Psi)$
8:         if $\gamma_c > \gamma_{best}$ then
9:             $\gamma_{best} \leftarrow \gamma_c$; $v_{best} \leftarrow v_c$
10:    if $\frac{\gamma_{cur}}{|BP|} < \frac{\gamma_{best}}{|BP|+1}$ then
11:        remove $v_{best}$ from $S$
12:        insert $v_{best}$ into $BP$ (by the order in $SP$)
13:        $\gamma_{cur} \leftarrow \gamma_{best}$
14:    else
15:        Return $BP$
```

shortest path $CP_{1,11} = \langle v_1, v_4, v_5, v_{11} \rangle$. Note that Table 4.5a shows only the rows and columns whose nodes fall into path $SP_{1,11}$. Table 4.5b, shows the running steps of the algorithm. First, we initialize BP to $CP_{1,11}$ and the set S to $\{v_2, v_7, v_8, v_9\}$. Those $\chi_{s,t}$ entries that contribute to the benefit of BP are shaded light-gray (see Table 4.5a). In iteration 1, we add the best node v_8 to BP because its normalized benefit is higher than that of BP. Those $\chi_{s,t}$ entries that contribute to the additional benefit of BP are shaded medium-gray. In iteration 2, even for the best node v_2, the normalized benefit of $BP \cup \{v_2\}$ is smaller than that of BP. Thus, the algorithm terminates and returns the path $\langle v_1, v_4, v_5, v_8, v_{11} \rangle$.

As a remark, the above algorithm may not always return the optimal result (i.e., the generic concise path with the highest normalized benefit). Suppose that we add the nodes v_2, v_7, v_8 to $CP_{1,11}$ to form a generic concise path: $P^* =$

$\chi_{s,t}$	v_1	v_2	v_4	v_5	v_7	v_8	v_9	v_{11}
v_1	/	0	50	0	0	16	0	0
v_2		/	0	0	35	0	0	12
v_4			/	2	0	0	0	20
v_5				/	5	0	0	0
v_7					/	0	0	0
v_8						/	0	30
v_9							/	0
v_{11}								/

(a) relevant entries of $\chi_{s,t}$

Iteration	Steps	Path
Initialization	$CP = \langle v_1, v_4, v_5, v_{11}\rangle$ $S = \{v_2, v_7, v_8, v_9\}$	$\langle v_1, v_4, v_5, v_{11}\rangle$ $\overline{\Upsilon_m}$=72/4=**18**
(1)	v_2: (72+12)/5 v_7: (72+5)/5 v_8: (72+16+30)/5 v_9: (72+0)/5 Highest: 23.6 ¿ 18	**Add** v_8 $\langle v_1, v_4, v_5, v_8, v_{11}\rangle$ $\overline{\Upsilon_m}$=118/5=**23.6**
(2)	v_2: (118+12)/6 v_7: (118+5)/6 v_9: (118+0)/6 Highest: 21.67 ¡ 23.6	**STOP** $\langle v_1, v_4, v_5, v_8, v_{11}\rangle$

(b) running steps

Table 4.5. Finding the best generic concise shortest path, for $SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11}\rangle$ **and** $CP_{1,11} = \langle v_1, v_4, v_5, v_{11}\rangle$

$\langle v_1, v_2, v_4, v_5, v_7, v_8, v_{11}\rangle$. According to Table 4.5a, the normalized benefit of P^* is: $(118 + 35 + 12 + 5)/7 = 24.28$, which is higher than the algorithm's result (23.6).

Time complexity analysis Let n be the number of nodes in SP. Note that both the sizes of S and BP are upper-bounded by n.

In the while-loop (Lines 4–15), we remove a node v_{best} from S in each iteration, so it has at most n iterations. The for loop (Lines 6–9) has at most n iterations because S contains at most n nodes. In Line 7, each call to $\Upsilon_m((BP \cup \{v_c\}) \setminus \Psi)$ takes $O(n^2)$ time as the path size is at most n. By combining the above, the total running time of the while-loop is $O(n^4)$.

Before the while-loop, Lines 1–2 take $O(n)$ time and Line 3 takes $O(n^2)$ time. Thus, the time complexity of the algorithm is $O(n^4)$.

The computational cost is high, even though the typical path size n is in the order of hundreds on real road networks.

4.4.3 Efficient Implementation

We proceed to present a more efficient implementation of Algorithm 4.3. The idea is to identify shared expressions in the calculation and compute such expressions only once, regardless of the path size n ($n = |SP|$).

Recall that it takes $O(n^2)$ time to compute $\Upsilon_m(P \setminus \Psi)$, where P is a subsequence of SP. Consider the scenario that we need the updated γ value after adding a node v_c into P. Fortunately, we can apply Equation 4.6 to derive $\Upsilon_m((P \cup \{v_c\}) \setminus \Psi)$ from $\Upsilon_m(P \setminus \Psi)$ incrementally. This derivation requires only $O(n)$ time to compute $\sum_{v_j \in P,\, Q_{c,j} \not\subset \Psi} \chi_{c,j}$ because we have one node for v_c and at most n nodes in P.

We propose an efficient implementation in Algorithm 4.4. For each node v_c in S, we maintain its benefit $\Upsilon_m((P \cup \{v_c\}) \setminus \Psi)$ in an attribute $v_c.\gamma$. We first compute $\Upsilon_m(BP \setminus \Psi)$ at Line 3, then derive $v_c.\gamma$ for each node $v_c \in S$ incrementally.

The while-loop (Lines 6–21) repeats until S becomes empty or the benefit of BP cannot be improved further. In each iteration, we simply find the node (v_{best}) with the highest γ. If the normalized benefit $\overline{\Upsilon_m}$ of $BP \cup \{v_{best}\}$ is better than that of BP, we insert v_{best} into BP. Also, we need to update the value $v_c.\gamma$ for each remaining node $v_c \in S$ (Lines 13–17). By using the idea in Equation 4.6, we can derive a shared expression for updates (Line 13). We can compute it once before using it to update each $v_c.\gamma$.

$$\begin{aligned}
&\Upsilon_m((P \cup \{v_c\}) \setminus \Psi) \qquad (4.6)\\
&= \sum_{v_i, v_j \in P \cup \{v_c\},\ Q_{i,j} \not\subset \Psi} \chi_{i,j}\\
&= \sum_{v_i \in P \cup \{v_c\}} \ \sum_{v_j \in P \cup \{v_c\},\ Q_{i,j} \not\subset \Psi} \chi_{i,j}\\
&= \left(\sum_{v_i \in P} \ \sum_{v_j \in P,\ Q_{i,j} \not\subset \Psi} + \sum_{v_i \in P,\ Q_{i,c} \not\subset \Psi} + \sum_{v_j \in P,\ Q_{c,j} \not\subset \Psi} \right) \chi_{i,j}\\
&= \Upsilon_m(P \setminus \Psi) + 2 \cdot \sum_{v_j \in P,\ Q_{c,j} \not\subset \Psi} \chi_{c,j}
\end{aligned}$$

Time complexity analysis We proceed as in the time complexity analysis in Section 4.4.2. Let n be the number of nodes in SP. Again, both the sizes of S and BP are upper-bounded by n.

The while-loop (Lines 6–21) has at most n iterations. Within the while-loop, the first for loop (Lines 8–10) takes $O(n)$ time, the summation (Line 13) takes $O(n)$ time, and the second for loop (Lines 14–17) takes $O(n)$ time. Thus, the

Algorithm 4.4 Static-FastGreedy (Shortest path SP, Concise shortest path CP, Cache Ψ)

```
 1: $BP \leftarrow CP$                                              ▷ the result
 2: $S \leftarrow$ the set of nodes in $SP - CP$
 3: $\gamma_{cur} \leftarrow \Upsilon_m(BP \setminus \Psi)$
 4: for each $v_c \in S$ do
 5:     $v_c.\gamma \leftarrow \gamma_{cur} + 2 \cdot \sum_{j \in BP,\, Q_{c,j} \not\subset \Psi} \chi_{c,j}$
 6: while $S \neq \emptyset$ do
 7:     $\gamma_{best} \leftarrow 0$
 8:     for each $v_c \in S$ do                                 ▷ find the best $v_c$
 9:         if $v_c.\gamma > \gamma_{best}$ then
10:             $\gamma_{best} \leftarrow v_c.\gamma$; $v_{best} \leftarrow v_c$
11:     if $\frac{\gamma_{cur}}{|BP|} < \frac{\gamma_{best}}{|BP|+1}$ then          ▷ compare $\overline{\Upsilon_m}$
12:         remove $v_{best}$ from $S$
13:         $\Delta\gamma \leftarrow 2 \cdot \sum_{j \in BP,\, Q_{best,j} \not\subset \Psi} \chi_{best,j}$        ▷ shared part
14:         for each $v_c \in S$ do                             ▷ update $v_c.\gamma$
15:             $v_c.\gamma \leftarrow v_c.\gamma + \Delta\gamma$
16:             if $Q_{best,c} \not\subset \Psi$ then
17:                 $v_c.\gamma \leftarrow v_c.\gamma + 2 \cdot \chi_{best,c}$
18:         insert $v_{best}$ into $BP$ (by the order in $SP$)
19:         $\gamma_{cur} \leftarrow \gamma_{best}$
20:     else
21:         Return $BP$
```

time complexity of the while-loop is $O(n^2)$.

Before the while-loop, Lines 1–2 take $O(n)$ time, and Lines 3–4 take $O(n^2)$ time. Thus, the time complexity of the algorithm is $O(n^2)$.

4.5 Dynamic Caching Setting

We have examined the static caching approach in Section 4.4. In this section, we adopt the *dynamic caching* [14–16] approach, which intends to cache the most recently accessed data items. When a cache miss occurs and the cache is full, these policies decide which data item to evict from the cache. For instance, the Least-Recently-Used (LRU) policy evicts the least recently used item in the cache. These policies allow the cache to adapt dynamically to the query workload. However, they incur runtime overhead in maintaining the cache.

In this section, we present a dynamic caching method for generic concise shortest paths. It adopts the LRU policy. When we obtain a shortest path SP (after a cache miss), our main problem is how to compute a short generic shortest path that will be able to answer many queries (in the future). To tackle this problem, we employ two lightweight data structures for maintaining query statistics: (i) a sliding window $\mathbb{W}$ that keeps the most recent $\mathbb{W}_{size}$ queries, and (ii) an array μ that records the frequencies of the query nodes in $\mathbb{W}$.

Algorithm 4.5 shows the pseudo code for our dynamic caching method. It is invoked when there is a cache miss for a query $Q_{s,t}$. First, we calculate the shortest path SP for $Q_{s,t}$, and then compute a concise shortest path GCP for it. Then, for each node v_i in SP, we add it into the DGCP if v_i is in CP or its frequency in μ is non-zero. Next, we remove the oldest query from the sliding window $\mathbb{W}$ and insert the newest query into $\mathbb{W}$. Then, we update the frequency array μ accordingly. Finally, we apply the LRU policy to update the cache Ψ with the generic concise shortest path.

We proceed to illustrate the running steps of the algorithm. Table 4.6a shows a list of queries and their shortest paths. Table 4.6b depicts the running steps. ΔL shows which query will be added and removed from $\mathbb{W}$ at each timestamp. Array μ shows the frequency of query nodes in the sliding window. In this example, the extra node(s) are added to the CP path (shown in bold) in order to

Algorithm 4.5 Dynamic-GCP (Query $Q_{s,t}$, Cache Ψ)

```
    Global structures:
    $\mathbb{W}$: a sliding window for the most recent $\mathbb{W}_{size}$ queries
    $\mu$: the frequencies of query nodes in $\mathbb{W}$
 1: $SP \leftarrow$ calculate the shortest path from $Q_{s,t}$
 2: $CP \leftarrow$ calculate the concise shortest path of $SP$
 3: for each $v_i \in SP$ do                          ▷ calculate dynamic GCP path
 4:     if $v_i \in CP$ or $\mu_i > 0$ then
 5:         add $v_i$ to $DGCP$
 6: if $\mathbb{W}$ is full then                             ▷ remove old query
 7:     dequeue $Q_{s',t'}$ from $\mathbb{W}$
 8:     $\mu_{s'} \leftarrow \mu_{s'} - 1$; $\mu_{t'} \leftarrow \mu_{t'} - 1$
 9: enqueue $Q_{s,t}$ into $\mathbb{W}$                              ▷ insert new query
10: $\mu_s \leftarrow \mu_s + 1$; $\mu_t \leftarrow \mu_t + 1$
11: apply LRU policy to update $\Psi$ by $DGCP$                  ▷ update cache
```

obtain the DGCP path. At time $t = 1$, we receive the query (8,9) and calculate its DGCP path. Since calculation of DGCP paths happen before updating μ, the content of μ does not change from the initial state. Then, we remove (1,6) and add (8,9) to $\mathbb{W}$. At time $t = 2$, the query from $t = 1$ affects the content of μ. Window $\mathbb{W}$ is updated as before. Similarly, the running steps at $t = 3$ and $t = 4$ are shown in the table.

4.6 Experimental Study

Section 4.6.1 covers the methods considered in the experiments. Section 4.6.2 covers the experimental settings. Section 4.6.3 covers experiments on real trajectory induced workload, while Section 4.6.4 covers the experiments on synthetic workloads.

Query	Shortest Path
$Q_{8,9}$	$\langle v_8, v_5, v_7, v_9 \rangle$
$Q_{1,4}$	$\langle v_1, v_3, v_4 \rangle$
$Q_{1,10}$	$\langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10} \rangle$
$Q_{2,10}$	$\langle v_2, v_5, v_7, v_9, v_{10} \rangle$

(a) shortest paths of queries

Time t	Query	ΔL	μ	DGCP Path
0	N.A.	N.A.	1,3,5,6,7,10: $\underline{1}$	N.A.
1	(8,9)	$-(1,6)$ $+(8,9)$	1,3,5,6,7,10: $\underline{1}$	$\langle v_8, v_5, \mathbf{v_7}, v_9 \rangle$
2	(1,4)	$-(3,5)$ $+(1,4)$	3,5,7,8,9,10: $\underline{1}$	$\langle v_1, \mathbf{v_3}, v_4 \rangle$
3	(1,10)	$-(7,10)$ $+(1,10)$	1,4,7,8,9,10: $\underline{1}$	$\langle v_1, v_5, v_7, \mathbf{v_9}, v_{10} \rangle$
4	(2,10)	$-(8,9)$ $+(2,10)$	1: $\underline{2}$; 4,8,9,10: $\underline{1}$	$\langle v_2, v_5, v_7, \mathbf{v_9}, v_{10} \rangle$

(b) running steps

Table 4.6. Running steps of dynamic caching, $L_{size} = 3$, for the road network in Figure 4.1

4.6.1 Methods Considered

We evaluate the hit ratios of our caching methods and competitors on a machine running Debian. Table 4.7 lists all the methods used in our experiments. In the static caching setting, all methods use an existing caching policy for shortest paths [11]. In the dynamic caching setting, all methods use LRU for cache replacement.

Our proposed methods are: (i) CP, caching concise paths as defined in Section 4.3, (ii) GCP, the static caching method for generic concise paths in Section 4.4, and (iii) DGCP, the dynamic caching method for generic concise paths in Section 4.5.

As discussed in related work, K-Skip [36] is a lossy shortest path compression method, so we do not compare with it. We compare our methods with two competitors: (i) full shortest path (SP), and (ii) concise path skip (CP-Skip). CP-Skip is a variant of K-Skip that includes all nodes of CP and every k-th node of SP. Since CP-Skip contains CP, it is a lossless shortest path method.

As discussed in related work, K-Skip [22] is a lossy shortest path compression method, so we do not compare with it. We compare our methods with two competitors: (i) full shortest path (SP), and (ii) concise path skip (CP-Skip). CP-Skip is a variant of K-Skip that: includes all nodes from a CP path, and includes every k-th node of SP into the path. Since CP-Skip contains a CP path, it is guaranteed to always generate a lossless path.

Method	**Static caching policy**	**Dynamic caching policy**
CP	[11]	LRU
GCP	[11]	N.A.
DGCP	N.A.	LRU
SP	[11]	LRU
CP-Skip	[11]	LRU

Table 4.7. Methods used in experiments

4.6.2 Experimental Setting

Datasets and Workloads

Table 4.8 shows information on the road networks used in the experiments. Although a real query log for online direction services [45] exists, service providers do not make their query logs publicly available. Thus, we generate query logs from real trajectory data on the corresponding maps.

We use real trajectories from the Aalborg [32] and Beijing [33] road networks. From each trajectory, we extract the start and end locations as the source and target nodes of a shortest path query in a query log. Since the trajectory datasets for Aalborg and Beijing are small (4.3k and 13k trajectories, respectively), we enlarge the trajectory dataset (to 300,000 trajectories) by sampling trajectories in the log and deviating their coordinates within a given radius (default value: 0.5 km).

For each of the two larger road networks (Colorado and New York), we generate a query log as follows. In the real world, drivers start from a dense region (e.g., a residential region) and travel towards another dense region (e.g., an industrial region). To simulate such behavior, we randomly choose a set of cluster centers with a given radius to form a set of clusters. Next, we randomly pick a pair of nodes from any 2 clusters to form a shortest path. By default, we use 10 clusters, and the radius is 8 km.

Following an existing experimental methodology [11], we divide the query log into two equal parts: (i) a query log $\mathcal{QL}$, for extracting query statistics and training the cache, and (ii) a query workload $\mathcal{WL}$ used for measuring the hit ratios of the methods.

Road network	**Map size (km^2)**	**Road network**
Aalborg	60x60	*download.cloudmade.com* 129k nodes, 137k edges
Beijing	60x60	*download.cloudmade.com* 76k nodes, 85k edges
Colorado	1000x1000	*www.dis.uniroma1.it/challenge9* 435k nodes, 1,057k edges
New York	6000x4000	*www.dis.uniroma1.it/challenge9* 264k nodes, 733k edges

Table 4.8. Characteristics of datasets

Parameters and Default Values

Table 4.9 lists the parameters and their default values. The first two entries show the default radius for the different road networks. Since different road networks have different sizes, we use 0.5 km as the default radius for the Aalborg and Beijing road networks, and we use 8 km as the default radius for the Colorado and New York road networks. The *number of clusters* denotes the number of clusters used for generating the synthetic workload (for Colorado and New York). The *cache capacity (ratio)* expresses the cache space as a percentage of the space needed for storing the entire road network. Thus, the absolute cache space for the different road networks are different. The last two are parameters for specific to CP-Skip and DGCP. *Path size ratio* is used by CP-Skip to define the ratio of the CP-Skip path length to the full shortest path length. *Window size* is used by DGCP to define the size of the sliding window.

Parameter	**Default value**
Radius (for Aalborg, Beijing)	0.5 km
Radius (for Colorado, New York)	8 km
Number of clusters (for Colorado, New York)	10
Cache capacity (ratio)	50%
Path size ratio (for CP-Skip)	50%
Window size (for DGCP)	1000 queries

Table 4.9. Default parameters

4.6.3 Real Trajectory Induced Workload

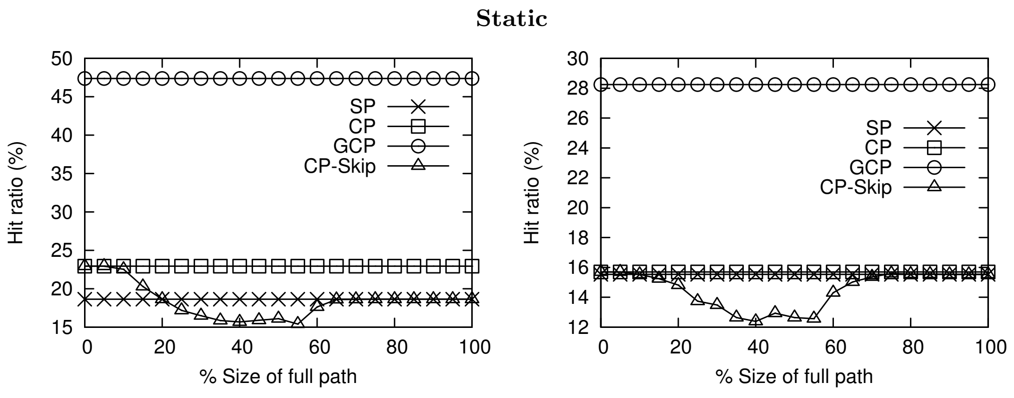
We proceed to report on experiments using real trajectory induced workloads (on the Aalborg and Beijing road networks).

First, we examine the average lengths (i.e., number of nodes) of paths in the cache for the different methods, see Table 4.10. Observe that full shortest paths (SP) are much longer than concise shortest paths (CP). Generic concise shortest paths (GCP/DGCP) are only slightly longer than concise shortest paths (CP). This means that SP paths contain many intermediate nodes that do not intersect query nodes in the workload. Note that the average path lengths of the methods (e.g., SP and CP) are similar in the static and dynamic settings.

Road Network	**Static**			**Dynamic**		
	SP	CP	GCP	SP	CP	DGCP
Aalborg	338.5	38.7	48.1	315.6	37.3	41.3
Beijing	166.1	26.4	35.6	129.8	23.1	26.1
Colorado	874.8	98.3	115.7	887.3	96.9	106.1
New York	501.9	71.3	73.3	514.7	70.4	72.0

Table 4.10. Average path length (in nodes)

Next, we study the effect of the path size ratio on the performance of CP-Skip, see Figure 4.6. The other methods (SP, CP, GCP) are included for reference only; they do not take the path size ratio as a parameter. Clearly, GCP outperforms CP-Skip in all cases. The results for dynamic caching are similar, so we omit them.



(a) Aalborg data (b) Beijing data

Figure 4.6. Hit ratio vs. path size ratio

Next, we investigate the effect of the window size parameter on the hit ratio of the DGCP method (in the dynamic caching setting). Figure 4.7 plots the hit ratio of DGCP with respect to the window size. The hit ratio increases until the window size reaches 10,000. Observe that we can achieve a high hit ratio when the window size is sufficiently large.

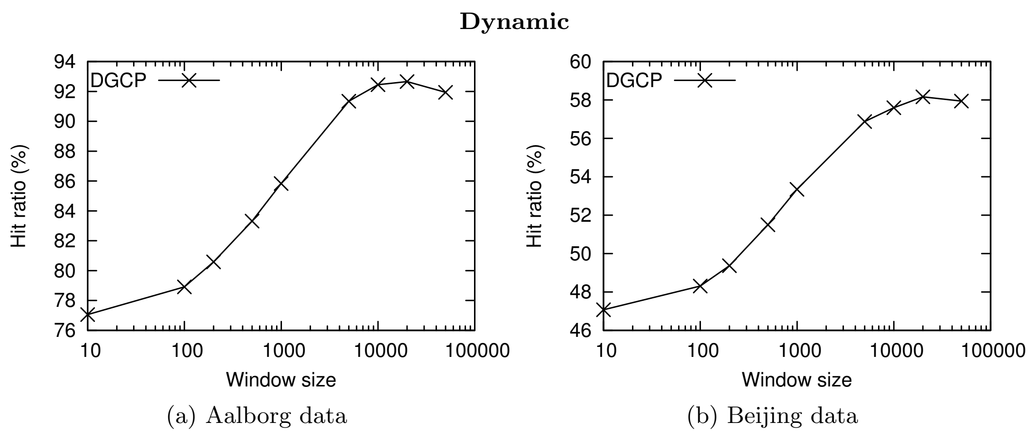
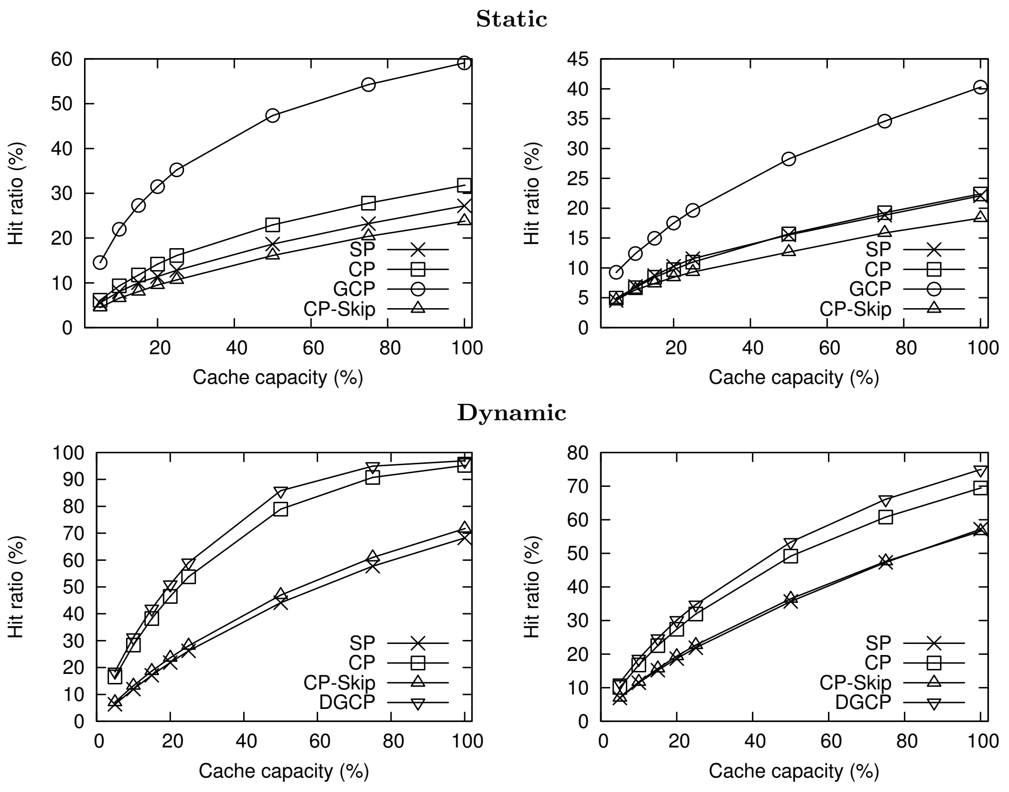


Figure 4.7. Hit ratio vs. window size

Figure 4.8 shows the hit ratios of the methods as a function of the cache capacity, Again, GCP outperforms the other methods significantly, in both the static and dynamic settings. Since CP-Skip and SP have similar hit ratios, we exclude CP-Skip from subsequent experiments.

Figure 4.9 shows the hit ratios of the methods for various cluster radius values. Observe that a smaller radius leads to a higher hit ratio. This happens because smaller clusters result in fewer possible unique queries.



(a) Aalborg (b) Beijing data

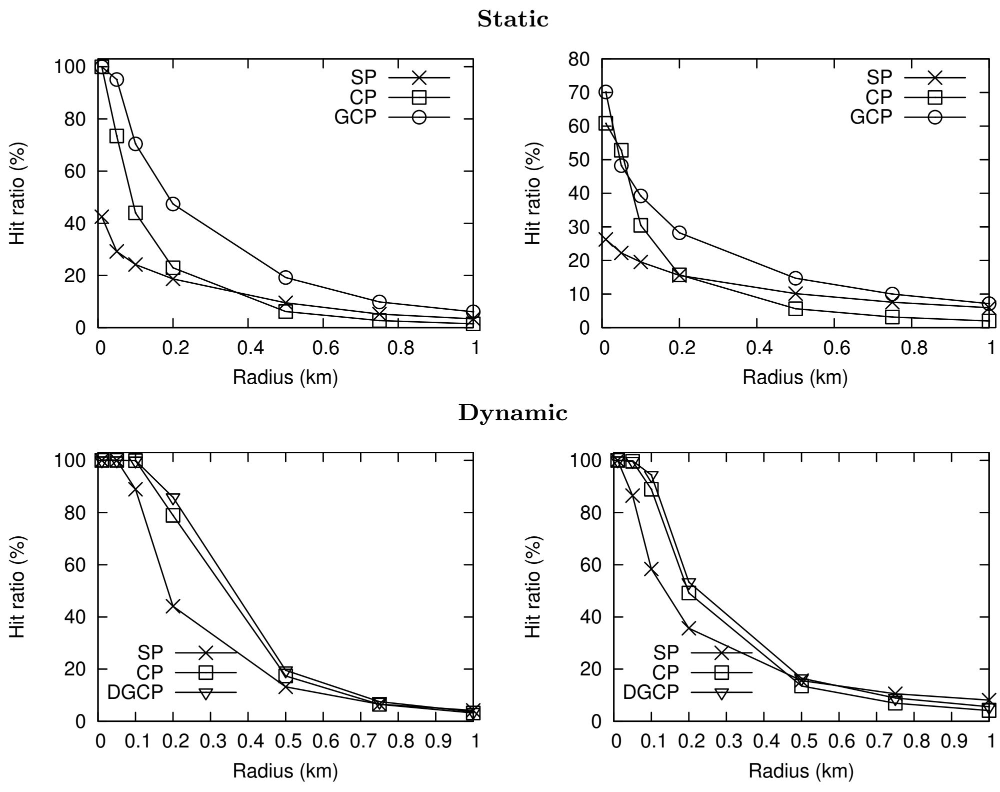
Figure 4.8. Hit ratio vs. cache capacity

4.6.4 Synthetic Workload

We proceed to report on experiments using synthetic workload (on the Colorado and New York road networks).

We plot the average path length of the methods, as shown in Table 4.10. In general, the results are similar to those for the real trajectory induced workloads.

Figure 4.10 shows the hit ratios of the methods with respect to the cache capacity. In the static (dynamic) caching setting, GCP/DGCP can achieve a hit ratio of 84.28% (99.75%) and 39.70% (94.13%) for Colorado and New York,



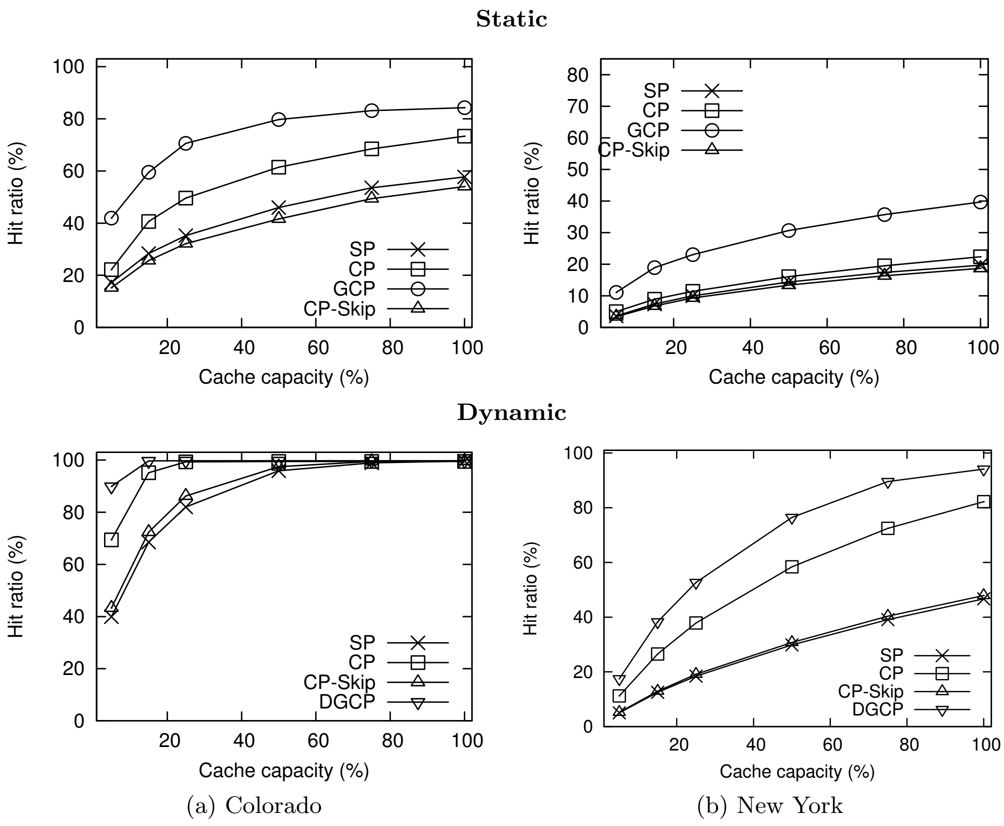
(a) Aalborg (b) Beijing

Figure 4.9. Hit ratio vs. cluster radius

respectively. Observe that, in the dynamic caching setting, we can achieve a very high cache hit ratio with a small cache capacity. In the remaining experiments, we do not include the results for CP-Skip as CP-Skip and SP have similar hit ratios.

Figure 4.11 shows the impact of the cluster radius on the hit ratios of the methods. Observe that a smaller radius leads to a higher hit ratio. This is expected as a smaller radius implies that there are fewer unique query pairs in the workload.

In the static caching setting, GCP always outperforms its competitors, and

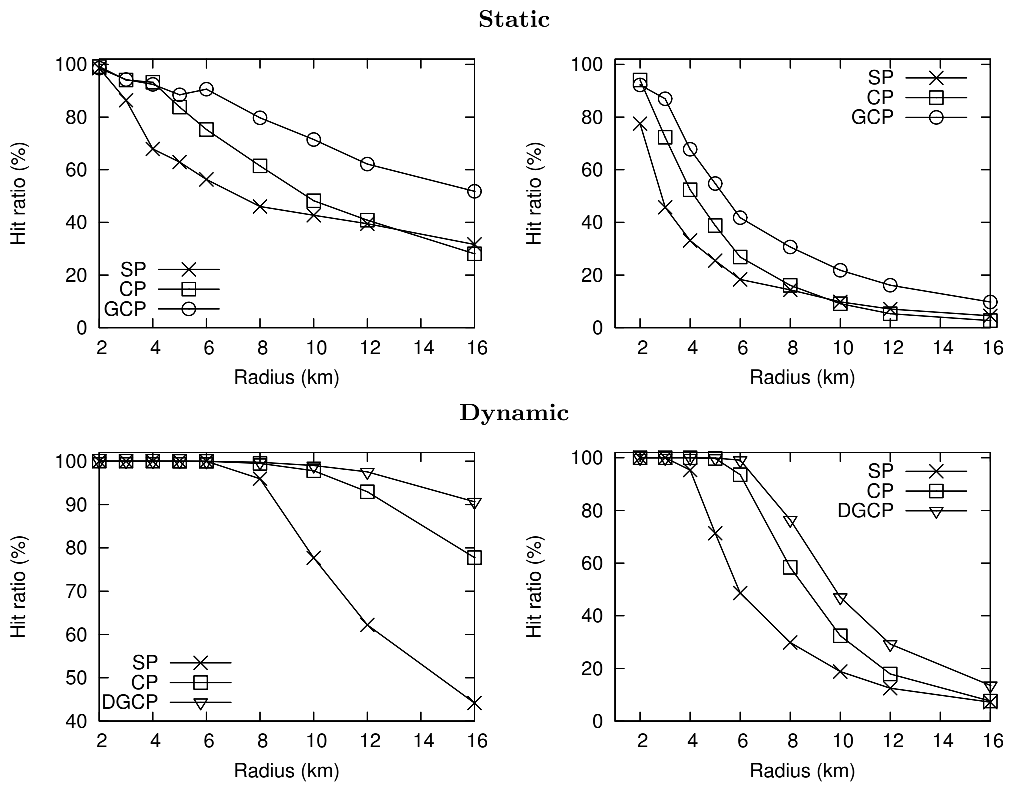


(a) Colorado (b) New York

Figure 4.10. Hit ratio vs. cache capacity

CP is the second best, followed by CP-Skip and SP in the last tier. In the dynamic caching setting, DGCP clearly outperforms its competitors.

Figure 4.12 shows the impact of the number of clusters on the cache hit ratio. GCP/DGCP consistently outperforms the competitors, with a lead to the next-best competitor by at least 20%.



(a) Colorado (b) New York

Figure 4.11. Hit ratio vs. cluster radius

4.7 Conclusions

In this chapter, we exploit concise shortest paths to boost the cache hit ratio of shortest path queries in cache servers. A concise shortest path occupies much less space than a complete shortest path, while providing sufficient navigation information to mobile users. First, we propose the notion of a generic concise shortest path that enables a trade-off between the path size and the number of queries that can be answered by the path. Then we develop static and dynamic caching techniques for generic concise shortest paths.

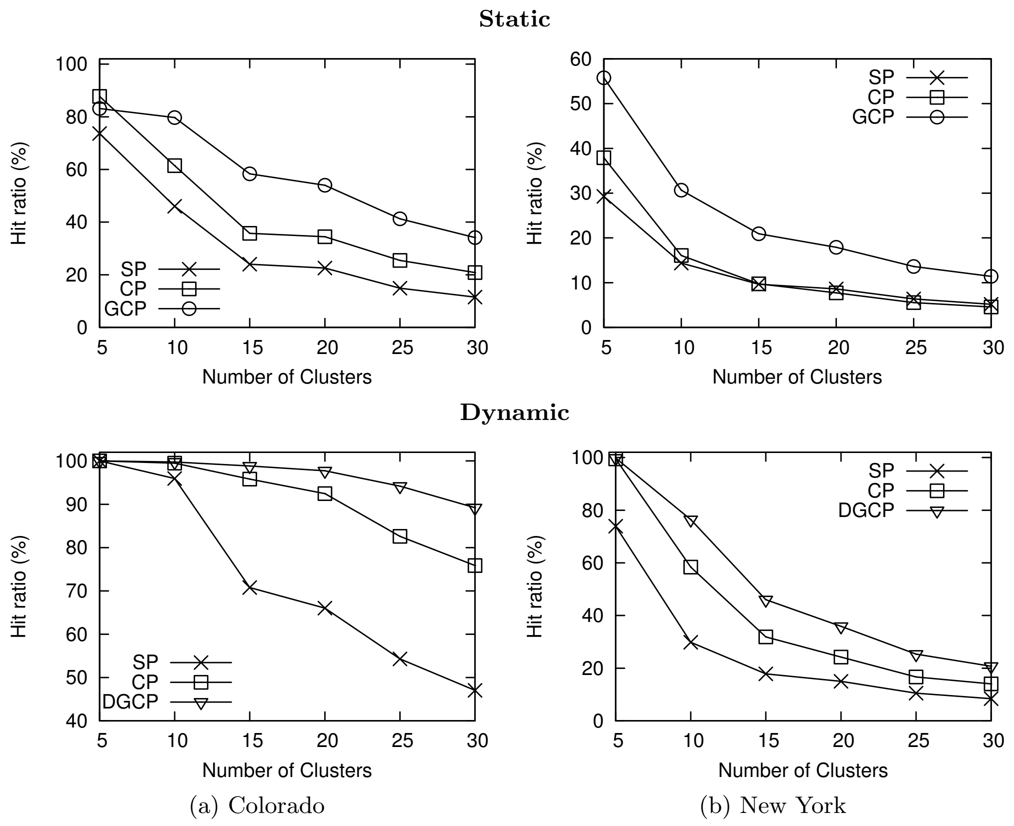


Figure 4.12. Hit ratio vs. number of clusters

Experimental results show that the hit ratios of our best methods (GCP and DGCP) are 10%–40% higher than that of competitors. Our methods are more robust with respect to the cache capacity.

As for the future work, we will study the maintenance of cache content with respect to dynamic traffic updates.

Chapter 5

Safe Regions for Meeting Point

In this chapter, we study a problem related to the sum-optimal meeting point, which aims to minimize the sum of distances traveled by users.

This is useful if e.g., a group of friends wants to share the fuel cost, and meet up at the restaurant which would cost the group the least for all group members to arrive at. Figure 5.1 gives a simple example with 2 moving users, u_1, u_2, and 2 restaurants, p_1, p_2. The answer to the sum-optimal meeting point problem is p_1 as it has the minimal sum of travel distances from u_1 and u_2 (11 units vs. 16.3 units to reach p_2).

Real applications relevant to this problem are EchoEcho[1] which lets users easily share their location, and Tourality[2] which lets users compete in various challenges using locations in their vicinity. However, current services such as EchoEcho and Tourality do not support solving the sum-optimal meeting point

[1]www.echoecho.me
[2]www.tourality.com

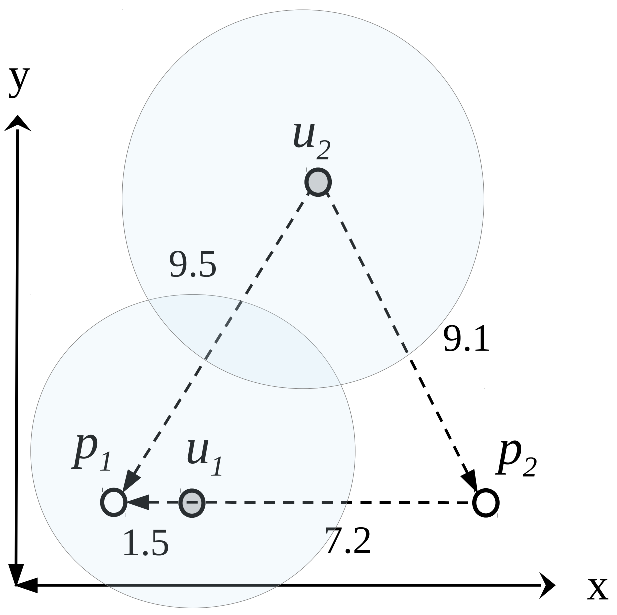


Figure 5.1. Example for the sum-optimal meeting point

problem. They can at most provide the user with the locations of friends. This could allow the user solve the snapshot version of the sum-optimal meeting point problem manually.

We will examine the continuous version of the problem, continuously monitoring which location minimizes the sum of distances of all users in a group. currently no commercial service enable users to use this type of service. We call the continuous version of this problem Sum-optimal Meeting Point Notification (Sum-MPN). The challenge in solving the Sum-MPN problem is that simply executing existing snapshot solutions to the problem will incur a prohibitively high computational load on the service provider, as well as incurring very high communication costs for the users.

We are using safe regions to reduce the communication frequency for the user, as well as the computational load for the service provider. The two are interdependent as fewer updates naturally decreases the load on the service provider. We propose two types of safe regions, circular (Section 5.3) and tiled (Section 5.4). It is challenging to calculate safe regions for the Sum-MPN problem as the safe regions are interdependent for a given group of users. .

To summarize; in order to address the above mentioned challenges we make the following contributions:

- Proposed circular and tiled safe regions for the Sum-Optimal Meeting Point Notification problem
- Design efficient algorithms and various optimizations to compute these safe regions.

The rest of the chapter is organized as follows. First, we introduce the problem and definitions in Section 5.1. Second, we review related work specific to this chapter in Section 5.2. Next, we present our solutions in Section 5.3 and Section 5.4, together with their optimizations. Our methods are evaluated using real and synthetic data in Section 5.5. Finally, we conclude the chapter in Section 5.6.

5.1 Problem Definition

We first provide the definitions for the sum distance and the sum-optimal meeting point.

Definition 5.1 (Sum distance) *The* sum distance *from a point p to a group* *of users U is:*

$$\|p, U\|_{sum} = \sum_{u_i \in U} \|p, u_i\|$$

Definition 5.2 (Sum-optimal meeting point) *Given a group of users U and a dataset of points P, the sum-optimal meeting point p^o is the point in P with the smallest $\|p^o, U\|_{sum}$. It is also called SUM-GNN [46].*

The sum-optimal meeting point is suitable when a group of users wishes to minimize the sum of their travel distances (and thus their total fuel cost). As for the incentive, the users in a group may agree on sharing the total fuel cost evenly when they reach the meeting point. Specifically, for those having fuel cost less than the average, they would contribute the cost difference (from the average) to other users in the group.

We illustrate an example of the sum-optimal meeting point in Figure 5.1. Assume that the user group is $U = \{u_1, u_2\}$ and the dataset is $P = \{p_1, p_2\}$. The sum-optimal meeting point is p_1 with the value $\|p_1, U\|_{sum} = 1.5 + 9.5 = 11$.

An example of safe regions for the sum-optimal meeting point is illustrated by circles in Figure 5.1.

Observe that Papadias et al. [46] have studied the snapshot version of our problem, i.e., computing the sum-optimal meeting point (called SUM-GNN in their work). In contrast, we focus on computing safe regions for such a meeting point.

The Sum-Optimal Meeting Point Notification (Sum-MPN) problem is then: Given a set of moving users U and a set of points of interest P, Sum-MPN continuously reports the sum-optimal meeting point $p^0 \in P$ to all users in U, such that the sum of all distances to p^0 from users in U is minimized.

5.2 Related Work

We present the related work specific to this chapter here. We have covered safe regions in Section 2.2.

Continuous monitoring aims to execute repeated spatial queries on the positions of moving objects. Prior work on continuous monitoring can be arranged into two categories: (i) continuous queries, e.g., kNN queries [47–51], moving window (rectangle range) queries [22, 52]; (ii) detecting proximity between moving objects [53]. Safe region is commonly used in works on continuous queries.

In this work, the shape of the safe regions are determined by two elements: (i) the location of points of interest, and (ii) the locations of other users in the group. Previous work is restricted to only one of these two, and is thus inapplicable to this problem.

Previous work on safe regions focus on regions like polygons (Voronoi cell) or arc-based regions. Such regions are relatively easy to compute. This work is described in Section 2.2.

Defining safe regions for our Sum-MPN problem is challenging because: (i) the safe regions for Sum-MPN have irregular shapes (see section 5.4) and are thus hard to compute; (ii) the safe regions of users are interdependent and the users change their locations dynamically and unpredictably, rendering precomputation techniques (e.g., Voronoi cells [19]) inapplicable.

The snapshot version of our problem is similar to two existing types of queries: group nearest neighbor (GNN) query [46] and group enclosing query [54]. GNN queries takes several query points and try to find their closest neighbor. Group enclosing queries takes a set of points P and a set of query points Q. It finds the point in P with the minimum distance to all points in Q.

The work closest to ours, [55], focuses on GNN monitoring on road networks. It differs from our work in two aspects. First, we do not consider the underlying road network. Secondly, [55] aims to minimize computations at the server side; conversely we focus on minimizing the communication cost. Thus, their methods cannot be applied to solve Sum-MPN.

5.3 Circular Safe Region Approach

In this section, we approximate the exact safe regions of users by circles due to simplicity. We first study the condition for verifying a set of safe regions. Then, we design an algorithm for computing circular safe regions.

Although exact safe regions have irregular shapes, they can be conservatively approximated as circles. We can approximate the irregular shape by finding the largest circle which can fit inside. We now assign each user u_i a circular safe region $R_i = \odot(u_i, r)$, where u_i is the current user location and r is the radius. Note that the same radius r is used across different R_i.

To reduce the communication cost between the server and the users, the value r should be as large as possible. The following theorem decides the maximum radius r such that the safe regions remain valid.

Definition 5.3 (Distances) *Let $\|p, l\|$ be the Euclidean distance between points p and l. The* minimum distance *and the* maximum distance *from a point p to a set/region S are:*

$$\|p, S\|_{min} = \min_{l \in S} \|p, l\| \tag{5.1}$$

$$\|p, S\|_{max} = \max_{l \in S} \|p, l\| \tag{5.2}$$

Theorem 5.1 (Sum-optimal maximal circles) *The maximum radius of circles for safe regions is:*

$$r_{max} = \frac{\min_{p \in P - \{p^o\}}(\|p, U\|_{sum}) - \|p^o, U\|_{sum}}{2m} \tag{5.3}$$

Proof. Let $R_i = \odot(u_i, r)$, a circle with radius r and center as the current user location u_i. We have: $\|p, R_i\|_{max} = \|p, u_i\| + r$ and $\|p, R_i\|_{min} = \|p, u_i\| - r$.

By applying the definition of safe regions for Sum-optimal meeting point, we derive the following inequality for any point $p \in P - \{p^o\}$:

$$\sum_{u_i \in U} (\|p^o, R_i\|_{max}) \leq \sum_{u_j \in U} (\|p, R_j\|_{min})$$

$$\sum_{u_i \in U} (\|p^o, u_i\| + r) \leq \sum_{u_j \in U} (\|p, u_j\| - r)$$

By rearranging the terms, we obtain:

$$2m \cdot r \leq \sum_{u_j \in U} (\|p, u_j\|) - \sum_{u_i \in U} (\|p^o, u_i\|)$$

which is equivalent to

$$r \leq \frac{\|p, U\|_{sum} - \|p^o, U\|_{sum}}{2m} \tag{5.4}$$

Note that Equation (5.4) holds for any point $p \in P - \{p^o\}$. By taking the minimum value of all $\|p, U\|_{sum}$, we obtain: $r_{max} = \frac{\min_{p \in P - \{p^o\}}(\|p,U\|_{sum}) - \|p^o,U\|_{sum}}{2m}$.

□

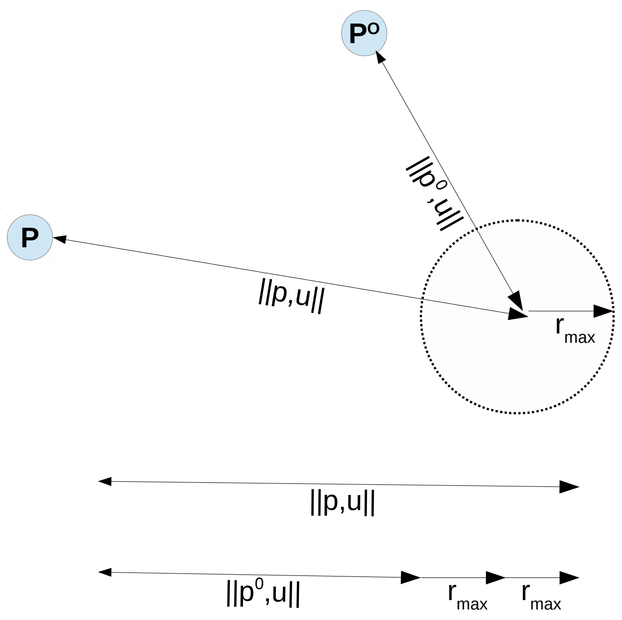


Figure 5.2. Illustration of r_{max} calculation

Figure 5.2 illustrates how the maximum radius r_{max} for a safe region is found for a single. By finding the sum-optimal meeting point and the second best meeting point we take the difference of their distances to the user (center of circle) divided by two.

Algorithm 5.1 is the pseudo-code for computing circular safe regions for users. Assume that the dataset set P is indexed by an R-tree. First, the algorithm finds the best two meeting points by calling an existing algorithm [56] on the R-tree of P. Note that the second best meeting point is the point p that contributes to $\min_{p \in P - \{p^o\}}(\|p, U\|_{sum})$. Then, it computes the maximum radius r_{max} by Equation (5.3) and returns the corresponding circular safe regions to the users.

Algorithm 5.1 Circle-MSR (Set of users U, Dataset P)

```
1: $p^o, p \leftarrow$ FindSumGNN($U$, $P$, 2)          ▷ apply algo. in [56]
2: compute the radius $r_{max}$                       ▷ apply Equation 5.3
3: for each user $u_i \in U$ do
4:     return the safe region $\odot(u_i, r_{max})$ to $u_i$
```

5.4 Tile-based Safe Region Approach

In this section, we model each safe region R_i as an irregular shape and approximate it by using a set of tiles. We model the safe regions as irregular shapes because the safe region of each user is determined by the relative location of the user to both a set of locations as well as the moving locations of a group of friends. We introduce Algorithm 5.2 to compute a safe region group for the sum-optimal meeting point. Also, we adopt the divide-and-conquer method in Algorithm 5.3 and 5.4, to check whether a tile s should be inserted into the safe region R_i of user u_i.

5.4.1 Main Algorithm

We are now ready to present an algorithm for computing tile-based safe regions (Algorithm 5.2). Each safe region R_i is modeled as a set of tiles, so it can be used to approximate an irregular shape. The main idea of the algorithm is to browse the tiles around each user u_i in a systematic way, apply verification on them, and then add valid tiles into a safe region R_i. The algorithm terminates when no more tiles can be added in those safe regions.

Recall that Algorithm 5.1 computes the safe region of each user u_i as a circle $\odot(u_i, r_{max})$. The maximal tile (square) in each circle must also be a valid safe region. Thus, we set the tile size $\delta = \sqrt{2} \cdot r_{max}$ and add a tile $\boxdot(u_i, \delta)$ into its corresponding safe region R_i (Lines 1–4).

The parameter α specifies the (maximum) number of tiles to be assigned to each safe region R_i. It can also be used to bound the number of iterations in Lines 5–11. In each iteration, the algorithm examines the safe regions of users in a round-robin manner.

We call a function *Next-Tile* to get the next tile s, in the clockwise direction, for user u_i. Then, it tests the new tile s with other users' safe regions by calling *Sum-Divide-Verify* (Line 9). The loop terminates either when (i) the test returns true, or (ii) s is empty, i.e., *Next-Tile* has exhausted all tiles for u_i. At the end, the algorithm returns a safe region R_i to each user u_i.

Figure 5.3 gives an example of how Algorithm 5.2 and 5.3 function. The circle around U_i represent r_{max}, and the tile inside is the initial safe region of U_i (Algorithm 5.2, line 4). Line 6-10 in Algorithm 5.2 finds all the tiles for user U_i by calling *Next-Tile()* and *Sum-Divide-Verify()* for each tile 1-8 around U_i. The tiles are added in a counter-clockwise manner from tile 1, up to tile 8 around U_i. The number of the tiles around U_i in Figure 5.3 illustrates the order in which they were added to U_i's safe region. *Sum-Divide-Verify()*, Algorithm 5.3, checks each tile added, to ensure it can be added around U_i, if it can't then the tile will

Algorithm 5.2 Tile-MSR (Set of users U, Dataset P, Tile limit α, Split level L)

```
1: compute $p^o$ and $r_{max}$                                    ▷ apply Algorithm 5.1
2: $\delta \leftarrow \sqrt{2} \cdot r_{max}$                        ▷ initial tile size
3: for each user $u_i$ in $U$ do
4:     $R_i \leftarrow \{ \boxdot(u_i, \delta) \}$                   ▷ initial safe region
5: for $\tau \leftarrow 1$ to $\alpha$ do                            ▷ control running time
6:     for each user $u_i \in U$ do                                  ▷ round robin
7:         repeat
8:             $s \leftarrow$ Next-Tile ( $u_i, \delta$ )              ▷ by tile ordering
9:             $flag \leftarrow$ Sum-Divide-Verify ( $\mathcal{R}, u_i, s, p^o, P, L$)   ▷ call Algorithm 5.3
10:        until $flag =$ true or $s = \emptyset$
11: for each user $u_i \in U$ do
12:     return the safe region $R_i$ to $u_i$
```

Algorithm 5.3 Sum-Divide-Verify (Safe region group $\mathcal{R}$, User u_i, Tile s, Optimal point p^o, Dataset P, Level L)

```
1: if $\forall\, p \in P - \{p^o\}$, Sum-GT-Verify ( $\mathcal{R}, u_i, s, p, p^o$ ) is true then
2:     $R_i \leftarrow R_i \cup \{s\}$
3:     return true
4: $flag \leftarrow$ false
5: if $L > 0$ then                                         ▷ control the recursion level
6:     divide $s$ into four sub-tiles
7:     for each sub-tile $s'$ of $s$ do
8:         if Sum-Divide-Verify ( $\mathcal{R}, u_i, s', p^o, P, L-1$ ) then
9:             $flag \leftarrow$ true
10: return $flag$
```

be divided into 4 sub tiles and *Sum-Divide-Verify()* will be called recursively on each sub tile (lines 6-9). Tiles 6-8 show how divided tiles will be added around U_i. They are subdivided because P_2 is closer to the left side of a full tile than P_1 would be, so to still grow the area of the safe region tiles 6-9 all need to be subdivided.

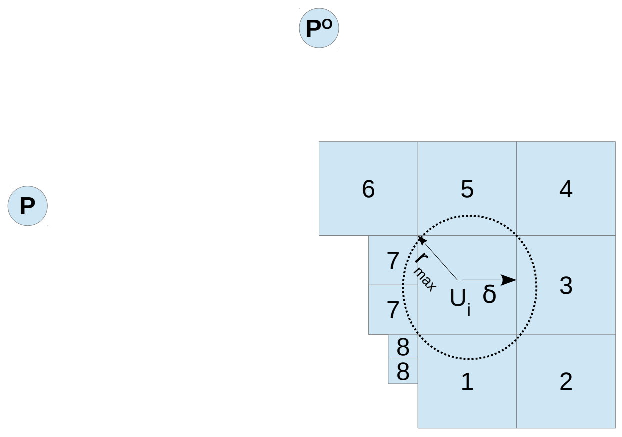


Figure 5.3. Example for Tile-MSR (Algorithm 5.2 & 5.3)

5.4.2 Group tile verification

Let $\langle R_i|_{i=1}^{m}\rangle$ be a valid safe region group obtained so far. Given a new tile s for user u_x, we want to verify efficiently whether the above safe region group is valid after inserting s into R_x (Algorithm 5.3, line 1). Let $L = \langle l_1, \cdots, l_m\rangle$ be a group location instance, where $l_x \in s$ and $l_i \in R_i$ for all $i \neq x$.

Specifically, we want to verify that, for every instance of users' locations L (as stated above), whether $\|p^o, L\|_{sum} \leq \|p', L\|_{sum}$ holds for every non-result point $p' \in P - \{p^o\}$. We define the comparison function $F(p', p^o, L)$ as:

$$\begin{aligned} F(p', p^o, L) &= \|p', L\|_{sum} - \|p^o, L\|_{sum} \\ &= \sum_{l_i \in L} (\|p', l_i\| - \|p^o, l_i\|) \end{aligned} \qquad (5.5)$$

The verification returns false if $F(p', p^o, L) < 0$ for some non-result point $p' \in P - \{p^o\}$ and some group location instance L.

For a given point $p' \in P - \{p^o\}$, we minimize the value of $F(p', p^o, L)$ in order to check whether it can become negative. Observe that, in Equation 5.5, we can minimize the term $\|p', l_i\| - \|p^o, l_i\|$ for each user u_i independently.

It turns out that the loci of $\|p', l\| - \|p^o, l\| = r$ can be described by hyperbola curves, as shown in Figure 5.4. In this example, $p^o = (1, 0)$ and $p' = (-1, 0)$. Given a square tile s, our task is to find the minimum value of $\|p', l\| - \|p^o, l\|$ among all location l of s. First, we divide the space by the axis $\overline{p'p^o}$ into the upper half-plane and the lower half-plane. Observe that, within the same half-plane, the same hyperbola curve can be either a decreasing curve or an increasing curve, but not both. As such, the minimum value along a straight line must occur at either of its end vertices. To find the minimum value of a tile s, it suffices to compute the value $\|p', v\| - \|p^o, v\|$ at: (i) each corner v of s (e.g., A,B,C,D), and (ii) any intersection v between s and the axis $\overline{p'p^o}$ (e.g., E, F).

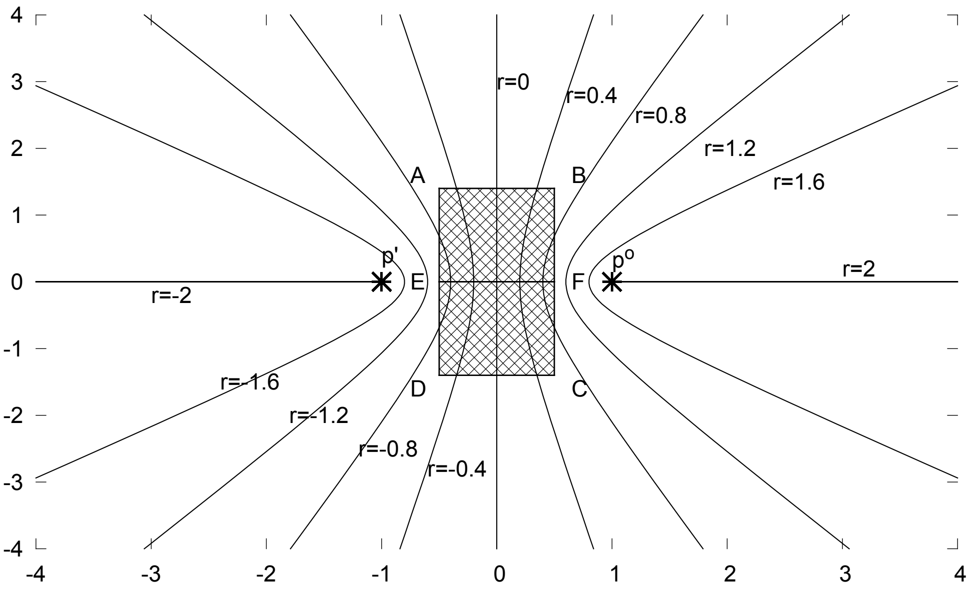


Figure 5.4. Hyperbola curves for $\|p', l\| - \|p^o, l\| = r$

This verification function is summarized as Algorithm 5.4. Observe that there are redundant computations during different calls of the algorithm (Lines 6–8). We can apply memorization techniques to avoid such redundant computations. The idea is to employ m hash tables: $H_1, H_2, \cdots, H_m$. For each user u_i, the minimum F_i value for point p' can be maintained at the hash entry $H_i(p')$. Then, we make two changes to the algorithm:

- replace Lines 6–8 by the statement: $F_i \leftarrow H_i(p')$
- at Line 12, we also execute $H_x(p') \leftarrow \min\{F_x, H_x(p')\}$ because the tile s will be inserted into the safe region of user u_x

Algorithm 5.4 Sum-GT-Verify(Safe region group $\mathcal{R}$, User u_x, Tile s, Point p, Optimal point p^o)

```
 1: $F_x \leftarrow \infty$
 2: for each vertex or intersection $v$ of tile $s$ do
 3:     $F_x \leftarrow \min\{F_x, \|p', v\| - \|p^o, v\|\}$
 4: for each user $u_i$ except $u_x$ do
 5:     $F_i \leftarrow \infty$
 6:     for each tile $s_i$ of safe region $R_i$ do
 7:         for each vertex or intersection $v$ of tile $s_i$ do
 8:             $F_i \leftarrow \min\{F_i, \|p', v\| - \|p^o, v\|\}$
 9: if $\sum_{i=1..m} F_i < 0$ then
10:     return false
11: else
12:     return true
```

5.4.3 Index pruning

Since it is expensive to invoke the above verification function for every point $p \in P - \{p^o\}$, we derive the following theorem to detect unpromising points that cannot become candidates.

THEOREM 5.2 *Given a safe region group $\mathcal{R}$, a point p cannot yield better result* *than p^o if,*

$$\|p, U\|_{sum} > \|p^o, U\|_{sum} + 2 \cdot \sum_{u_i \in U} r_i^\dagger \tag{5.6}$$

where $r_i^\dagger$ is the maximum distance between user u_i's current location and its safe region boundary.

The above pruning technique can also be extended to the MBRs in the R-tree. For instance, a MBR can be pruned if the value $\sum_{u_i \in U} d_{min}(MBR, u_i)$ is larger than the right-side of Equation (5.6).

5.4.4 Buffering optimization for index access

Observe that the computation of tile-based safe regions (Algorithm 5.2) invokes the *Sum-Divide-Verify* function multiple times, causing frequent accesses to the R-tree (of dataset P). In this section, we present an optimization method so that Algorithm 5.2 accesses the R-tree exactly once, regardless of the number of calls to *Sum-Divide-Verify*.

5.4.4.1 Buffering points for verification

Our idea is to retrieve a subset of points from the R-tree and only use them in subsequent calls to the *Sum-Divide-Verify* function. Given a parameter β, we define a distance threshold λ_β as follows. We will elaborate how to reduce the sensitivity of β later.

Definition 5.4 (Distance threshold) *The distance threshold λ_β is defined* *as follows:*

$$\lambda_\beta = \frac{\|p^{\beta+1}, U\|_{sum} - \|p^o, U\|_{sum}}{2m} \tag{5.7}$$

where point p^j denotes the j-th SUM-GNN of U.

Theorem 5.3 states that the best β SUM-GNNs (of U) are sufficient for verifying a group location instance L, provided that each $l_i \in L$ is within distance λ_β from u_i.

Theorem 5.3 (Sum-optimal buffering condition)
Let $P^*_{1..j} = \{p^1(= p^o), p^2, \cdots, p^j\}$ be the set of the best j SUM-GNNs.

Given an instance of users' locations $L = \langle l_1, \cdots, l_m\rangle$, if $\|l_i, u_i\| \le \lambda_\beta$ holds for every $1 \le i \le m$, then the SUM-GNN of L cannot be any point in $P - P^*_{1..\beta}$.

Proof. Let p' be an arbitrary point in $P - P^*_{1..\beta}$. Note that $\|p^{\beta+1}, U\|_{sum} \le \|p', U\|_{sum}$. We derive the following:

$$\sum_{u_i \in U}(\|p^o, u_i\|) + 2m\lambda_\beta \le \sum_{u_i \in U}(\|p', u_i\|) \tag{5.8}$$

From the given condition $\|l_i, u_i\| \le \lambda_\beta$, we can obtain: $\sum_{u_i \in U}(\|p^o, u_i\|) \ge \sum_{u_i \in U}(\|p^o, l_i\|) - 2m\lambda_\beta$ and $\max_{u_i \in U}(\|p', u_i\|) \le \max_{u_i \in U}(\|p', l_i\|) + 2m\lambda_\beta$. Combining these two inequalities with Equation 5.8, we get:

$$\sum_{u_i \in U}(\|p^o, l_i\|) \le \sum_{u_i \in U}(\|p', l_i\|)$$

Thus, the SUM-GNN of L cannot be p'. □

Figure 5.5 illustrates how the above Distance threshold techniques, presented above in Definition 5.4, works. Here we have $\beta = 2$, a set of users $\{u_1, u_2, u_3\}$, and a set of locations $\{p^0, p^1, p^2, p^3, p^4\}$. The locations are numbered by their SUM-GNNs rank, i.e. by their total distance to the set of users, p^0 having the smallest distance. The gray circles around each user have a radius of λ_β. Since $\beta = 2$, p^2 is the $p^{\beta+1}$'th SUM-GNN. Since $\beta = 2$ and both p^0 and p^1 are within λ_β distance of all users (gray circles), Theorem 5.3 states that the SUM-GNNs of $\langle p^0, p^1 \rangle$ are sufficient to verify them as a group location.

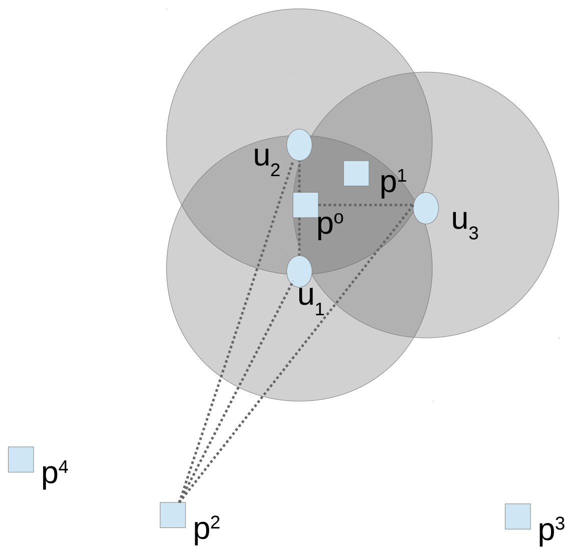


Figure 5.5. Example for using the distance threshold

We are now ready to present our buffering method. Specifically, before computing safe regions, we first retrieve the best $\beta+1$ SUM-GNN of U. When we verify a tile s for user i (*Sum-Divide-Verify*, Algorithm 5.3), we only process s if $\|s, u_i\|_{max} \leq \lambda_\beta$. This guarantees that the condition $\|l_i, u_i\| \leq \lambda_\beta$ in Theorem 5.3 is always satisfied. Then, we use the point set $P^*_{1..\beta}$ (instead of the entire P) in the verification function. We need not access the R-tree again since we have retrieved $P^*_{1..\beta+1}$ (which contains $P^*_{1..\beta}$).

5.4.4.2 Reducing the sensitivity of parameter β

Observe that the parameter β exhibits a tradeoff between the verification cost and the extent of safe regions. A small β limits the extent of safe regions significantly (due to the distance threshold λ_β).

To avoid overly small safe regions, we recommend to use a sufficiently large β.[3] However, the verification cost is directly proportional to β.

In the following, we provide an efficient implementation (Algorithm 5.5) whose verification cost is less sensitive to β. Now, we consider all β possible distance thresholds: $\lambda_1, \lambda_2, \cdots, \lambda_\beta$. To reduce the verification cost, we pick the smallest distance threshold λ_z such that it satisfies the condition of Theorem 5.3 for the current safe region group $\mathcal{R}$ and the new tile s.

This can be implemented efficiently in $O(\log \beta)$ time by binary search (Line 2). If such a distance threshold λ_z cannot be found, then the verification returns false as the new tile s violates the condition of Theorem 5.3.

Algorithm 5.5 Buffer-Sum-Divide-Verify (Safe region group $\mathcal{R}$, User u_i, Tile s, Optimal point p^o, Set $P^*_{1..\beta+1}$, Level L)

```
1: $dist \leftarrow \max\{\|u_i, s\|_{max}, \sum_{R_j \in \mathcal{R}} \|u_j, R_j\|_{max}\}$
2: find the minimum slot $z$ such that $dist \leq \lambda_z$        ▷ binary search
3: **if** no such $z$ exists **then**
4:     return false
5: **if** $\forall\, p \in P^*_{1..z} - \{p^o\}$, *Sum-GT-Verify* ( $\mathcal{R}, u_i, s, p, p^o$ ) is true **then**
6:     $R_i \leftarrow R_i \cup \{s\}$
7:     return true
8: apply Lines 4–10 of Algorithm 5.3
```

[3]We set $\beta = 100$ based on our experimental results

5.5 Experiments

5.5.1 Settings

In this section, we experimentally evaluate the performance of our proposed techniques. All methods were implemented in C++ and the experiments were performed on an Intel Core 2 Duo 2.66GHz PC with 8 GBytes memory, running on Ubuntu 10.04.

Dataset and Query Workload. We obtain a real dataset from Pocket-GPSWorld[4], which consists of $N = 21,287$ POIs. We simulate the movement of query users by using both synthetic and real trajectories: (i) *GeoLife*, a real trajectory set of taxi drivers released by Microsoft[5]; (ii) *Oldenburg*, a synthetic trajectory set generated from Brinkhoff's generator [57]. Each trajectory set consists of 60 trajectories that have above 10,000 timestamps. We partition each trajectory set into 10 user groups and then report the average performance on these user groups.

Measures. We evaluate our performance in three aspects: (i) *update frequency*, which reflects the frequency for users to issue update messages to the server, and (ii) *average running time*, which is the computation time for safe regions per update. (iii) *communication cost (packet count)*, measures the number TCP packets for messages sent between the server and the clients. A packet contains at most $(576 - 40)/8 = 67$ (double-precision) values since the typical Maximum Transmission Unit (MTU) over a network is 576 bytes and a packet

[4]www.pocketgpsworld.com
[5]www.microsoft.com

has a 40-byte header[6]. To represent a shape, we use 3 values per a circle, 3 values per a square, and 4 values per a rectangle.

Configurations. We study our proposed solutions with different variations. *Circle* denotes the *Circle-MSR* method in Section 5.3. *Tile* denotes the *Tile-MSR* method in Section 5.4 using undirected ordering on tiles and *lossless compression* in [58]. *Tile-D* is a variant of *Tile* using directed ordering on tiles [58]. Both *Tile* and *Tile-D* apply the *GT-Verify* function and the index pruning technique. Table 5.1 presents the default values and ranges of parameters in our experiments.

Table 5.1. Parameter values in experiments

Parameter	Default	Range
Data size n	N	$0.25N, 0.5N, 0.75N, 1.0N$
User group size m	3	$2, 3, 4, 5, 6$
User speed	V (speed limit)	$0.25V, 0.5V, 0.75V, 1.0V$
Tile limit α	30	/
Split level L	2	/

Our proposed methods require two extra parameters: (i) the tile limit α, and (ii) the split limit L. As the default setting in [58], we set $\alpha = 30$ and $L = 2$ as they achieve a good trade-off between the running time and the update frequency.

5.5.2 Scalability experiments

This section studies the scalability of our methods for the Sum-MPN problem.

Effect of user group size m. We vary the group size m in experiments on both datasets in Figure 5.6. The trend is similar to that in corresponding

[6]http://tools.ietf.org/html/rfc879

experiments in the previous subsection.

Tile-based safe region methods are effective in optimizing the update frequency and the communication cost.

Effect of data size n. Next, we vary the data size (i.e., the number of *POI*s) in Figure 5.7. When n is large, the data density in the space is high so all the methods have high update frequency. Nevertheless, the update frequency and the communication cost of tile-based methods increase at a slower rate than the circle-based method.

Effect of buffering parameter β. Figure 5.8 shows the performance of *Tile-D* and *Tile-D-β*. Again, the trend is similar to that in corresponding experiments in the previous subsection. *Tile-D-β* achieves a much smaller CPU time, while its update frequency stays close to *Tile-D* for a wide range of β values. Thus, it is safe to tune the parameter β to any value between 10 and 100.

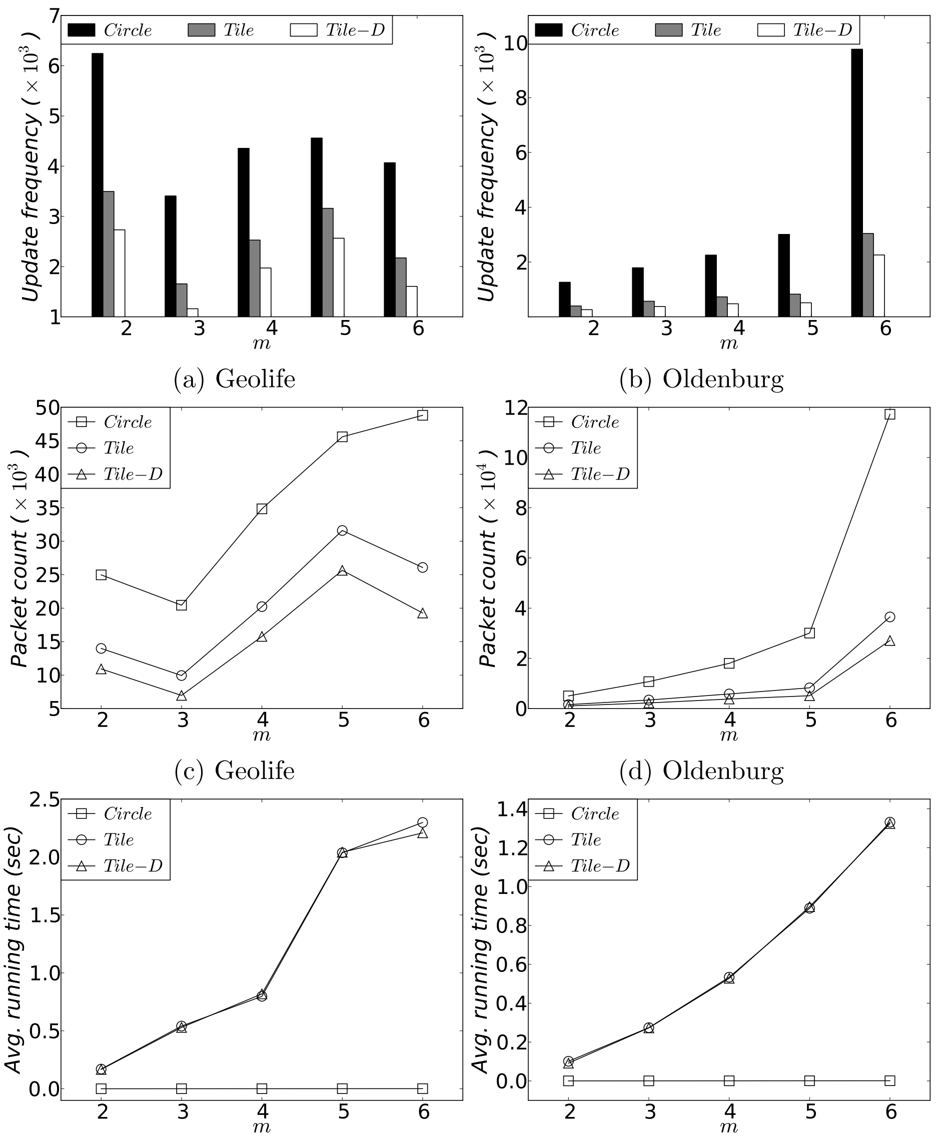
5.5.3 Summary of experimental results

Circle has the lowest running time, but it incurs higher update frequency and communication cost (packet count) than our tile-based methods.

Tile-D achieves the best update frequency and communication cost. Furthermore, our buffering optimization offers a substantial saving in the running time while only slightly increases the update frequency.

5.6 Conclusion

In this chapter, we focus on minimizing the communication cost for monitoring the sum-optimal meeting point for a group of users. We propose safe region group for this problem, in order to reduce the communication frequency of users. We design efficient algorithms and various optimizations to compute these safe regions.

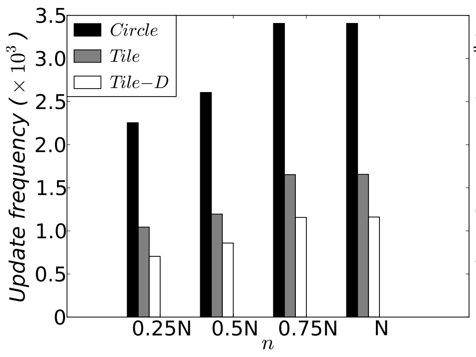


(a) Geolife (b) Oldenburg

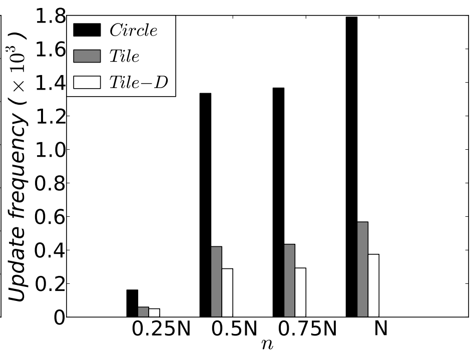
(c) Geolife (d) Oldenburg

(e) Geolife (f) Oldenburg

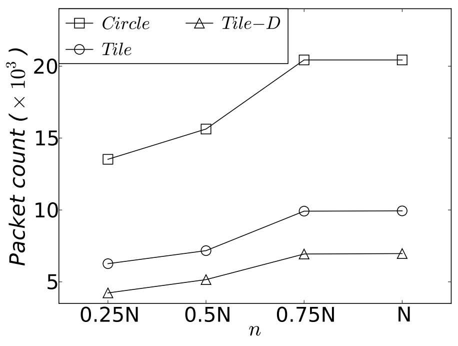
Figure 5.6. Vary group size m (for Sum-MPN)



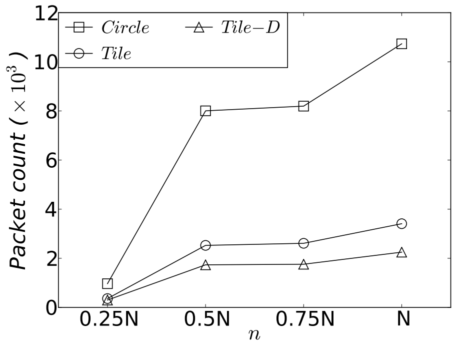
(a) Geolife



(b) Oldenburg

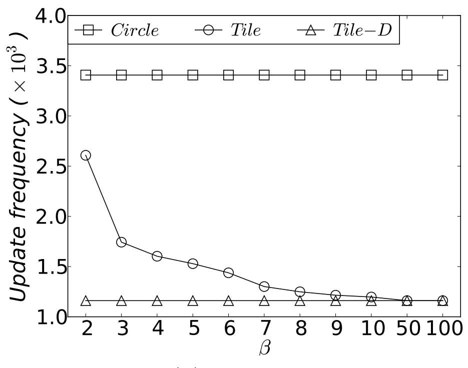


(c) Geolife

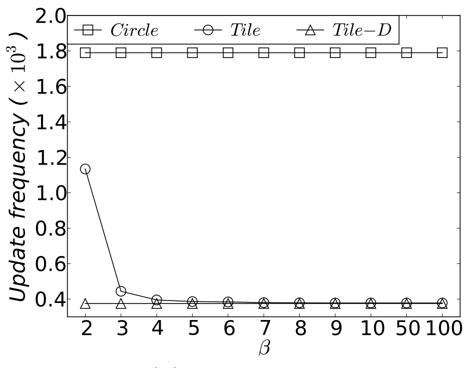


(d) Oldenburg

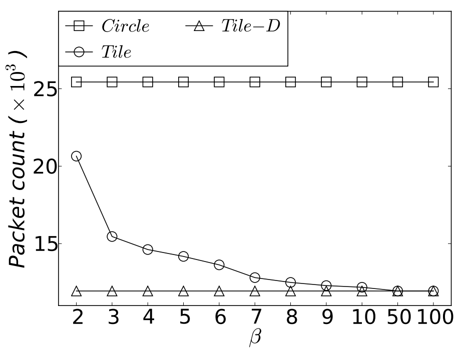
Figure 5.7. Vary POI number n (for Sum-MPN), as a fraction of data size N



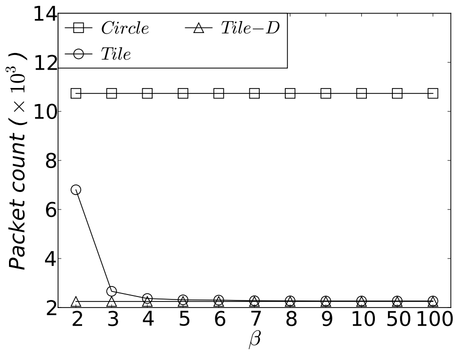
(a) Geolife



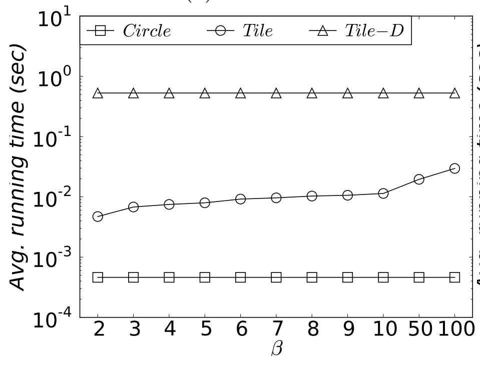
(b) Oldenburg



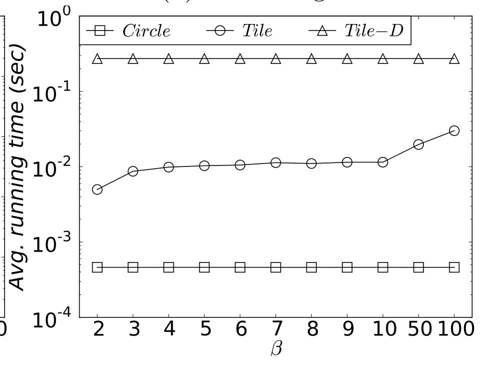
(c) Geolife



(d) Oldenburg



(e) Geolife



(f) Oldenburg

Figure 5.8. Vary buffering parameter β (for Sum-MPN)

Chapter 6

Conclusion

In this thesis, we have realized the following results.

- We develop algorithms and data structures for caching of shortest paths.
- We develop algorithms for computing generic concise shortest paths.
- We present algorithms for computing the safe regions for the sum-optimal point notification problem.

First we investigate caching of shortest paths. We study the problem for both proxy and server caching. We introduce a model to capture the benefit of caching a shortest path. We propose a greedy algorithm to select beneficial paths for the cache, based on historical query information. We introduce effective structures to speed up cache look up and increase the cache space utilization. Finally we do extensive performance evaluations to evaluate the cache performance.

Second, we study how to increase the cache hit ratio by caching partial shortest paths, which still hold enough information to navigate the original shortest paths. We present the concept of a concise path, the shortest possible lossless representation of a shortest path. We introduce the idea of a generic concise path which addresses the trade-off between the path length and the number of queries a path can answer. We develop caching schemes to support generic shortest paths, and perform extensive experimental evaluations on both real and synthetic datasets

Third, we investigate how to finding the optimal meeting point for a group of users. The focus is on minimizing the communication frequency of the user. Efficient algorithms and optimizations are developed to efficiently calculate *independent safe regions* based on the concept of a sum-optimal meeting point, where the sum of all users travel distance is minimized.

We proceed to outline several future research directions as follows. Our static caching problem in Chapter 3 is analogous to materialized view selection in data warehousing [59]. In future, we aim to utilize their ideas to build a shortest path cache with quality guarantees. We plan to extend our cache utilization techniques from Chapter 4 to handle dynamic traffic updates. We will implement procedures to keep the freshness of the cache and check the validity of shortest paths.

We plan to extend our techniques from Chapter 5 to the road network space. For *Circle*, we may replace a circular region by a range search region over road segments. For *Tile*, we may replace recursive tiles by recursive partitions of the road network. Also, we will develop a cost model for estimating the update frequency, the communication cost, and the running time of our methods.

Bibliography

[1] E. Sarigöl, O. Riva, P. Stuedi, and G. Alonso, "Enabling social networking in ad hoc networks of mobile phones," *Proceedings of the VLDB Endowment PVLDB*, pp. 1634–1637, 2009.

[2] K. Button and D. Hensher, *Handbook of Transport Systems and Traffic Control*. Handbooks in Transportation Research Series, Elsevier Science, 2001.

[3] "MapQuest Open Directions API." `http://developer.mapquest.com/web/products/open/directions-service`.

[4] "Google Directions API." `https://developers.google.com/maps/documentation/directions/`.

[5] "Bing Maps API." `http://www.microsoft.com/maps/developers/web.aspx`.

[6] "OpenStreetMap." `http://www.openstreetmap.org/`.

[7] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *International*

Conference on Research and Development in Information Retrieval SIGIR, pp. 183–190, 2007.

[8] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy, "Static query result caching revisited," in *Proceedings of the 17th international conference on World Wide Web WWW*, pp. 1169–1170, 2008.

[9] I. S. Altingövde, R. Ozcan, and Ö. Ulusoy, "A cost-aware strategy for query result caching in web search engines," in *Advances in Information Retrieval, 31th European Conference on IR Research, ECIR 2009, Toulouse, France, April 6-9, 2009. Proceedings*, pp. 628–636, 2009.

[10] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy, "A five-level static cache architecture for web search engines," *Information Processing & Management*, 2011.

[11] J. R. Thomsen, M. L. Yiu, and C. S. Jensen, "Effective caching of shortest paths for location-based services," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 313–324, 2012.

[12] J. R. Thomsen, M. L. Yiu, and C. S. Jensen, "Concise caching of driving instructions," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, 2014.

[13] J. Li, J. R. Thomsen, M. L. Yiu, and N. Mamoulis, "Efficient notification of meeting points for moving groups via independent safe regions," *IEEE Transactions on Knowledge and Data Engineering TKDE, to appear.*, 2014.

[14] E. P. Markatos, "On caching search engine query results," *Computer Communications*, vol. 24, no. 2, pp. 137–143, 2001.

[15] X. Long and T. Suel, "Three-level caching for efficient query processing in large web search engines," in *Proceedings of the 14th international conference on World Wide Web WWW*, pp. 257–266, 2005.

[16] Q. Gan and T. Suel, "Improved techniques for result caching in web search engines," in *Proceedings of the 18th international conference on World Wide Web WWW*, pp. 431–440, 2009.

[17] R. A. Baeza-Yates and F. Saint-Jean, "A three level search engine index based in query log distribution," in *String Processing and Information Retrieval SPIRE*, pp. 56–65, 2003.

[18] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pp. 330–341, 1996.

[19] B. Zheng and D. L. Lee, "Semantic Caching in Location-Dependent Query Processing," in *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD*, pp. 97–116, 2001.

[20] H. Hu, J. Xu, W. S. Wong, B. Zheng, D. L. Lee, and W. Lee, "Proactive caching for spatial queries in mobile environments," in *Proceedings of the 21st International Conference on Data Engineering, ICDE*, pp. 403–414, 2005.

[21] K. C. K. Lee, W.-C. Lee, B. Zheng, and J. Xu, "Caching complementary space for location-based services," in *Proceedings of the 10th International*

Conference on Advances in Database Technology, EDBT'06, pp. 1020–1038, Springer-Verlag, 2006.

[22] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee, "Location-based spatial queries," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 443–454, 2003.

[23] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang, "Multi-guarded safe zone: An effective technique to monitor moving circular range queries," in *Proceedings of the 26th International Conference on Data Engineering, ICDE*, pp. 189–200, 2010.

[24] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt, "Hierarchical graph embedding for efficient query processing in very large traffic networks," in *Scientific and Statistical Database Management, 21st International Conference, SSDBM*, pp. 150–167, 2008.

[25] S. Jung and S. Pramanik, "An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps," *IEEE Transactions on Knowledge and Data Engineering TKDE*, vol. 14, no. 5, pp. 1029–1046, 2002.

[26] F. Wei, "TEDI: Efficient Shortest Path Query Answering on Graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 99–110, 2010.

[27] H. Hu, D. L. Lee, and V. C. S. Lee, "Distance indexing on road networks," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, pp. 894–905, 2006.

[28] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 43–54, 2008.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 3'rd ed., 2009.

[30] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi, "Understanding Individual Human Mobility Patterns," *Nature*, vol. 453, no. 7196, pp. 779–782, 2008.

[31] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *International Conference on Information and Knowledge Management CIKM*, pp. 867–876, 2009.

[32] C. S. Jensen, H. Lahrmann, S. Pakalnis, and J. Runge, "The INFATI data," *Computing Research Repository (CoRR)*, vol. cs.DB/0410001, 2004.

[33] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining Interesting Locations and Travel Sequences from GPS Trajectories," in *Proceedings of the 18th international conference on World Wide Web WWW*, pp. 791–800, 2009.

[34] S. Podlipnig and L. Böszörményi, "A survey of web cache replacement strategies," *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[35] "Web cache softwares." `http://en.wikipedia.org/wiki/Web_cache`.

[36] Y. Tao, C. Sheng, and J. Pei, "On k-skip shortest paths," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 421–432, 2011.

[37] G. V. Batz, R. Geisberger, D. Luxen, P. Sanders, and R. Zubkov, “Efficient route compression for hybrid route planning,” in *Design and Analysis of Algorithms - First Mediterranean Conference on Algorithms, MedAlg*, pp. 93–107, Springer, 2012.

[38] R. Gotsman and Y. Kanza, “Compact representation of gps trajectories over vectorial road networks,” in *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD*, pp. 241–258, 2013.

[39] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The lru-k page replacement algorithm for database disk buffering,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 297–306, ACM, 1993.

[40] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, “Contraction hierarchies: Faster and simpler hierarchical routing in road networks,” in *Experimental Algorithms, 7th International Workshop, WEA*, pp. 319–333, 2008.

[41] H. Samet, J. Sankaranarayanan, and H. Alborzi, “Scalable network distance browsing in spatial databases,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 43–54, 2008.

[42] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck, “Highway dimension, shortest paths, and provably efficient algorithms,” in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pp. 782–793, 2010.

[43] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: towards bridging theory and practice," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 857–868, 2013.

[44] A. Dingle and T. Pártl, "Web cache coherence," *Computer Networks*, vol. 28, no. 7-11, pp. 907–920, 1996.

[45] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Y. Halevy, "Hyper-local, directions-based ranking of places," *Proceedings of the VLDB Endowment PVLDB*, vol. 4, no. 5, pp. 290–301, 2011.

[46] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui, "Aggregate nearest neighbor queries in spatial databases," *ACM Transactions on Database Systems TODS*, vol. 30, no. 2, pp. 529–576, 2005.

[47] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases*, pp. 287–298, 2002.

[48] G. S. Iwerks, H. Samet, and K. P. Smith, "Continuous k-nearest neighbor queries for continuously moving points with updates," in *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases*, pp. 512–523, 2003.

[49] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias, "Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 634–645, 2005.

[50] X. Yu, K. Q. Pu, and N. Koudas, “Monitoring k-nearest neighbor queries over moving objects,” in *Proceedings of the 21st International Conference on Data Engineering, ICDE*, pp. 631–642, 2005.

[51] X. Xiong, M. F. Mokbel, and W. G. Aref, “SEA-CNN: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases,” in *Proceedings of the 21st International Conference on Data Engineering, ICDE*, pp. 643–654, 2005.

[52] H. Hu, J. Xu, and D. L. Lee, “A generic framework for monitoring continuous spatial queries over moving objects,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 479–490, 2005.

[53] M. L. Yiu, L. H. U, S. Šaltenis, and K. Tzoumas, “Efficient proximity detection among mobile users via self-tuning policies,” *Proceedings of the VLDB Endowment PVLDB*, vol. 3, no. 1, pp. 985–996, 2010.

[54] F. Li, B. Yao, and P. Kumar, “Group enclosing queries,” *IEEE Transactions on Knowledge and Data Engineering TKDE*, vol. 23, no. 10, pp. 1526–1540, 2011.

[55] L. Qin, J. X. Yu, B. Ding, and Y. Ishikawa, “Monitoring aggregate k-nn objects in road networks,” in *Scientific and Statistical Database Management, 21st International Conference, SSDBM*, pp. 168–186, 2008.

[56] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis, “Group nearest neighbor queries,” in *Proceedings of the 20th International Conference on Data Engineering, ICDE*, pp. 301–312, 2004.

[57] T. Brinkhoff, “A Framework for Generating Network-Based Moving Objects,” *GeoInformatica*, vol. 6, no. 2, pp. 153–180, 2002.

[58] J. Li, M. L. Yiu, and N. Mamoulis, “Efficient notification of meeting points for moving groups via independent safe regions,” in *29th IEEE International Conference on Data Engineering, ICDE*, pp. 422–433, 2013.

[59] V. Harinarayan, A. Rajaraman, and J. D. Ullman, “Implementing Data Cubes Efficiently,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pp. 205–216, 1996.