Effective Caching of Shortest Paths for Location-Based Services

ABSTRACT

Web search is ubiquitous in our daily lives. Caching have been extensively used to reduce the computation time of the search engine and reduce the network traffic beyond a proxy server. Another form of web search, known as online shortest path search, is popular due to advances in geo-positioning. However, existing caching techniques are ineffective for shortest path queries. This is due to several crucial differences between web search results and shortest path results, in relation to query matching, cache item overlapping, and query cost variation.

Motivated by this, we identify several properties that are essential to the success of effective caching for shortest path search. Our cache exploits the optimal subpath property, which allows a cached shortest path to answer any query with source and target nodes on the path. We utilize statistics from query logs to estimate the benefit of caching a specific shortest path, and we employ a greedy algorithm for placing beneficial paths in the cache. Also, we design a compact cache structure that supports efficient query matching at runtime. Empirical results on real datasets confirm the effectiveness of our proposed techniques.

1. INTRODUCTION

Web search is ubiquitous and extensively used in our daily lives. The scenario for typical web search is illustrated in Figure 1. A user may submit a query "Paris Eiffel Tower" to the search engine, which then computes relevant results and reports them back to the user. Usually, a *cache* is used to keep the results of frequent queries in order to achieve a high hit ratio [**?**, **?**, **?**, **?**]. The cache can be employed at the search engine to save its computation time, e.g., when the query (result) can be found in the cache. To improve response time, a cache can also 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 of Figure 1 is applicable to *online shortest path search* on a map as well, which are popular due to advances in geo-positioning capability of mobile devices (e.g., PDA, smartphone). It has various applications for mobile users, e.g., a tourist asking for directions to a museum, a driver finding the route to a gas station, and a customer wanting to reach a restaurant. When compared to offline commercial navigation software, online search (e.g., Google Maps, MapQuest) provide two benefits to mobile users: (i) they are free to use, and (ii) they do not require any installation and storage space at mobile devices.

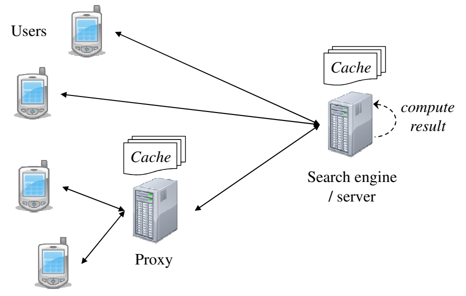


Figure 1: Scenario for Web Search

Figure 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, or it can be used at the server to reduce the server's computation time.

+++ CSJ: Yes, it may be wise to emphasize the proxy part so that we aim to reduce the Internet latency. Then we may want to understand/explain in a bit more detail how proxies are used today for Internet search.

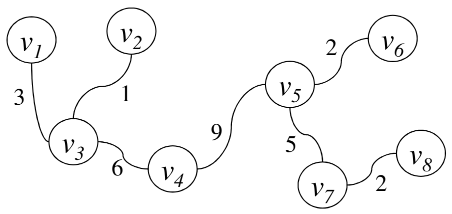


Figure 2: An Example Road Network

We study the caching of path search results in a scenario as shown in Figure 1. Nevertheless, there are crucial differences between general web search results and shortest path results, 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 re-visit the model for defining 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 significant overlap, although each of them does not contain the other. It is challenging to develop a compact structure for storing these paths.
- **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 other results. To optimize the server performance, a cost model has been developed for estimating the cost of a web query [**?**], in order to decide the importance of a result. However, to our knowledge, there is no work on estimating the cost of a shortest path query with respect to a given 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 paper is organized as follows. Section 2 studies the work related to the shortest path caching problem. Section 3 defines our problem formally, and Section 4 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 5 presents a compact and efficient cache structure for storing shortest paths. Our proposed methods are then evaluated on real data in Section 6, followed by a conclusion in Section 7.

2. RELATED WORK

Web Search Caching: 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 [**?**]; a request for the next result page is interpreted as another query.

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. [**?**] present pioneering work on evaluating two caching approaches (dynamic caching and static caching) on real query logs. *Dynamic caching* [**?**, **?**, **?**] 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 will be 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* [**?**, **?**, **?**, **?**, **?**, **?**] aims to cache the results of the most popular queries. It exploits a query log, which contains past users' queries in the past, 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., every day) based on the latest query log. Static caching has the advantage that it incurs very low overhead at query time.

Early works 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 [**?**, **?**, **?**] on web search caching use the *server processing time* as the performance metric. They show that different queries have different query processing times. For example, a query that involves terms with large posting lists is expected to incur high cost. Refs. [**?**, **?**] propose models for estimating the cost of processing queries based on the sizes of inverted lists, and they exploit both frequency and cost information in static caching methods.

None of the above works have studied the caching of shortest paths. In this paper, we adopt a static caching approach because it performs well on query logs that are typically skewed in nature and incurs very low overhead at query time. The cost models of [**?**, **?**] 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 will study a cost-oriented model for quantifying the benefit of placing a specific path in the cache.

Semantic Caching: In a client-server system, a cache may be employed at the client-side in order to reduce the communication cost and improve response latency of the client. The 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 [**?**] is a client-side caching model that associates cached results with valid ranges. Upon receiving a query Q, the relevant results from 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 remaining results of Q. Ref. [**?**] focuses 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 server for retrieving the remaining results.

Semantic caching has also been studied for spatial data [**?**, **?**, **?**]. Zheng et al. [**?**] define the semantic region of a spatial object as its Voronoi cell, which can then be utilized to answer nearest neighbor queries for a moving client user. Hu et al. [**?**] study semantic caching of tree nodes in an R-tree and examine how to process generic spatial queries on the cached tree nodes. Lee et al. [**?**] build generic semantic regions for spatial objects so that they are able to support generic spatial queries. However, no semantic caching techniques for graphs or shortest paths have been proposed.

Shortest Path Computation: 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 for storing 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. *Fully-precomputed* indexes, e.g., the distance index [**?**] or the shortest path quadtree [**?**], 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. *Partially-precomputed* indexes, e.g., landmarks [**?**], HiTi [**?**], and TEDI [**?**], attempt to materialize some distances/paths in order to accelerate the processing of shortest path queries. They employ some parameter to control the trade-offs among query performance, precomputation overhead, and storage space.

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. Also, it cannot be applied directly to any new index developed in the future.

In this paper, we view the shortest path index/algorithm as a black-box and decouple it from the cache. The main advantage is that our approach is applicable to any shortest path method, without knowing its implementation. Any new shortest path method can be integrated with our proposed cache seamlessly.

3. PROBLEM SETTING

We first introduce some background definitions and properties. Then we present our problem and objectives. Table 1 summarizes the notations to be used in this paper.

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 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 1: Summary of Notations

3.1 Definitions and Properties

We first provide the definitions of graph and shortest path.

DEFINITION 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 length is denoted as $W(v_i, v_j)$.

DEFINITION 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 $v_{x_0} = s$, $v_{x_m} = t$, 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 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 $3 + 6 + 9 + 5 = 23$.

Shortest paths exhibit the optimal subpath property (see Lemma 1). It states that every subpath of a shortest path is also a shortest path. For example, in Figure 2, the shortest path $P_{1,7} = \langle v_1, v_3, v_4, v_5, v_7\rangle$ contains these shortest paths: $P_{1,3}, P_{1,4}, P_{1,5}, P_{1,7}, P_{3,4}, P_{3,5}, P_{3,7}, P_{4,5}, P_{4,7}, P_{5,7}$. As we will discuss shortly, this property can be exploited for caching shortest paths.

LEMMA 1. **Optimal subpath property (from [?]).**
The shortest path $P_{a,b}$ contains the shortest path $P_{s,t}$ if $s \in P_{a,b}$ and $t \in P_{a,b}$. 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 $s = v_{x_i}$ and $t = v_{x_j}$.

3.2 Problem and Objectives

We adopt the architecture of Figure 1 to our caching problem. Using mobile devices with geo-positioning capabilities, users 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 improve the computation cost and the communication cost at the server/proxy, as well as reduce the response time of shortest path queries.

DEFINITION 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 $|\Psi|$ is taken as the total number of nodes of shortest paths in Ψ.

As discussed in Section 2, recent literature on web search caching [**?**, **?**, **?**, **?**] advocates the use of a static cache rather than a dynamic cache. Static caching has very low runtime overhead, while it only sacrifices the hit ratio a little. Thus, we also adopt the static caching paradigm and exploit a query log to build a cache.

DEFINITION 4. **Query log.**
A query log $\mathcal{QL}$ is a collection of timestamped queries that have been issued by users in the past.

We identify essential components in our static caching system in Figure 3. It involves: (i) a shortest path API, (ii) a cache, (iii) an online module for cache lookup, and (iv) offline/periodic modules for collecting query log, benchmarking the cost of API, and building the cache.

Observe that the *shortest path component* (in gray) is given by 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 the issue of 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 5. **Expense of executing query.**
We define $E_{s,t}$ as the expense 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. Upon receiving 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.

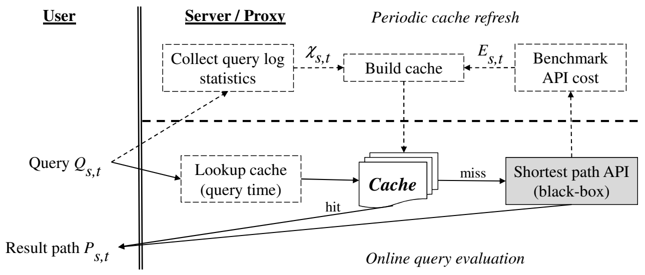


Figure 3: Components in a Static Caching System

Unlike web search caching, we observe that minimizing 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.

As a case study, we randomly generate 500 shortest path queries on two real road networks (their descriptions are available in Section 6), and execute a shortest path algorithm (Dijkstra) for them. In Figure 4, each subfigure (for each road network) shows the shortest path distance (x-axis) and the cost (y-axis) of each individual query. Observe that there is a strong correlation between the cost and the shortest path distance. Caching a short-range path may only provide a negligible improvement, even if the path is queried frequently. Therefore, we plan to design a *benchmark* for the cost of calling the API.

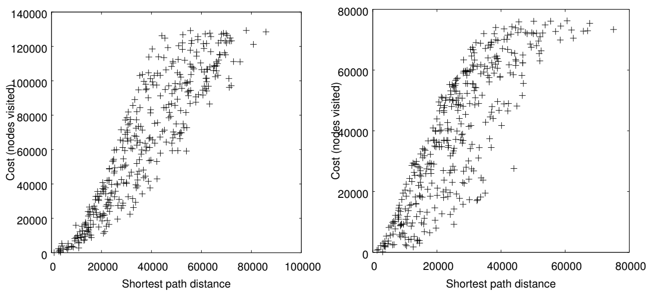


Figure 4: Cost vs. Distance of a Shortest Path Algorithm

Adopting the static caching paradigm, the server/proxy *collects the query log* and *builds the cache* periodically (e.g., daily). By extracting the distribution of the query log, we are able to estimate the probability of a specific shortest path being queried in the future. Combining such information with the benchmark model, we can place promising paths in the cache in order to minimize the overall cost of the system. 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 support efficient result retrieval.

Our main objective is to *reduce the overall cost incurred by calling the shortest path API*. We define this as the problem below. In Section 4, 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 $\mathcal{QL}$, this problem is to 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 for retrieving results from the cache. We will focus on these issues in Section 5.

3.3 Existing Solutions for Caching Results

We now revisit existing solutions for caching web search results [**?**, **?**] and then 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 [**?**]. When a new query is submitted, its result is inserted into the cache. The least-recently-used result is evicted from the cache when it becomes full.

We proceed to illustrate the running steps of LRU on the map in Figure 2. Suppose that the cache budget $\mathcal{B}$ is 10 (i.e., it can hold 10 nodes). Table 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 1). The running steps at subsequent times are shown in Table 2. In total, the LRU cache has only 2 hits.

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 2: Example of LRU on a Sequence of Queries

Observe that 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.

Another limitation of LRU is that it 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 s and t. This incurs significant runtime overhead, outweighing the advantages of the cache.

Static Caching—HQF: A typical static caching method for web search is the Highest-Query-Frequency (HQF) method [**?**]. 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.

According to Ref. [**?**], the frequency of a query is counted as the number of queries in $\mathcal{QL}$ that are identical to it. 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 $P_{1,6}$ simply because its query $Q_{1,6}$ has a low frequency (1). In fact, the path $P_{1,6}$ is more promising than $P_{3,6}$ because $P_{1,6}$ can be used to answer more queries than $P_{3,6}$. 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 a query may be used to answer multiple other queries.

Other common 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 appropriate choice of cached path should take such expenses into account.

Also, they have not studied the utilization of the cache space for shortest paths. For example, in Table 2, the paths in the cache overlap and cause wasted space on storing duplicate nodes among the paths. It is important to design a compact cache structure that exploits the sharing among paths to avoid storing duplicated nodes.

4. BENEFIT-DRIVEN CACHING

In this section, we propose a benefit-driven approach to decide which shortest paths should be placed in the cache. Section 4.1 formulates a systematic model for capturing the *benefit* of a cache on potential queries. This model requires the 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 4.2 and benchmark the expense of processing a query in Section 4.3. Finally, in Section 4.4, we present an algorithm for selecting promising paths to be placed in the cache.

4.1 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 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 the questions below:

1. Which queries $Q_{s,t}$ can be answered by the path $P_{a,b}$?
2. For query $Q_{s,t}$, how much cost can we save?

The first question is answered by Lemma 1. The path $P_{a,b}$ contains the path $P_{s,t}$ when 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:

$$\mathfrak{U}(P_{a,b}) = \{P_{s,t} : s \in P_{a,b}, t \in P_{a,b}, s \neq t\} \tag{1}$$

It contains the set of queries that can be answered by $P_{a,b}$. Taking Figure 2 as the example graph, the answerable query set of path $P_{1,6}$ is: $\mathfrak{U}(P_{1,6}) = \{P_{1,3}, P_{1,4}, P_{1,5}, P_{1,6}, P_{3,4}, P_{3,5}, P_{3,6},$

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

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

$P_{4,5}, P_{4,6}, P_{5,6}\}$. Table 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 $E_{s,t}$ 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:

$$\gamma(P_{a,b}) = \sum_{P_{s,t} \in \mathfrak{U}(P_{a,b})} \chi_{s,t} \cdot E_{s,t} \tag{2}$$

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 of v_s and v_t, as shown in Figure 5. 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). 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 5). Summing up the benefits of all paths in $\mathfrak{U}(P_{1,6})$, we thus obtain: $\gamma(P_{1,6}) = 0 + 1 \cdot 2 + 0 + 1 \cdot 4 + 0 + 0 + 3 \cdot 3 + 0 + 0 + 0 = 15$. Similarly, we can compute the benefit of path $P_{3,6}$: $\gamma(P_{3,6}) = 0 + 0 + 3 \cdot 3 + 0 + 0 + 0 = 9$.

We then extend our equations to the general case—a cache with 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 define the benefit of Ψ accordingly.

$$\mathfrak{U}(\Psi) = \bigcup_{P_{a,b} \in \Psi} \mathfrak{U}(P_{a,b}) \tag{3}$$

$$\gamma(\Psi) = \sum_{P_{s,t} \in \mathfrak{U}(\Psi)} \chi_{s,t} \cdot E_{s,t} \tag{4}$$

The cache benefit $\gamma(\Psi)$ answers the question: *"Using this 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. By Equation 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

[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 5.

correct because the path $P_{3,6}$ appears exactly once in the answerable query set $\mathfrak{U}(\Psi)$ of the cache.

Benefit per size: The benefit model does not consider the size $|P_{a,b}|$ of a path $P_{a,b}$ into account. Suppose that we are given two paths $P_{a,b}$ and $P_{a',b'}$ such that they have the same benefit (i.e., $\gamma(P_{a,b}) = \gamma(P_{a',b'})$) and $P_{a',b'}$ has a smaller size than $P_{a,b}$. Intuitively, we prefer path $P_{a',b'}$ rather than path $P_{a,b}$ because $P_{a',b'}$ occupies less space, leaving more cache space for placing other paths. Thus, we define the *benefit-per-size* of a path $P_{a,b}$ as:

$$\overline{\gamma}(P_{a,b}) \quad = \quad \frac{\gamma(P_{a,b})}{|P_{a,b}|} \tag{5}$$

We will utilize this notion in Section 4.4.

Recall from Section 3.2 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.

$\chi_{s,t}$	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 5: Example of $\chi_{s,t}$ and $E_{s,t}$ values for the graph

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

The frequency $\chi_{s,t}$ of query $Q_{s,t}$ plays an important role in our benefit model. According to a scientific study [?], the mobility patterns of human users follow a skewed distribution. For instance, hot regions (e.g., shopping malls, residential buildings) are expected to have high $\chi_{s,t}$, whereas rural regions 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), the server/proxy periodically collects the query log $\mathcal{QL}$ and extracts the values of $\chi_{s,t}$. The literature on static web caching [?] 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 $\chi_{s,t}$.

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 4: Query log $\mathcal{QL}$

Node-pair frequency counting: In this method, we first create a *node-pair frequency table* χ with $|V| \times |V|$ entries, like the one in Figure 5a. 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 how large the query log is.

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 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 5a. The $\chi_{s,t}$ values in the table χ can then be readily used by our benefit model in Section 4.1.

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 the query frequency among pairs of regions only.

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 6, 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.

For the second step, we create a *region-pair frequency table* $\widehat{\chi}$ with $L \times L$ entries, like the one in Figure 7b. 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 it 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 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 7b.

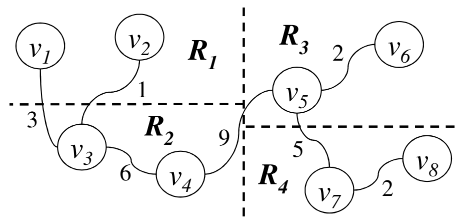


Figure 6: A graph partitioned into 4 regions

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

(a) 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

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

Figure 7: Counting region-pair frequency in table $\widehat{\chi}$

To enable the computation of our benefit model in Section 4.1, we now discuss how to derive the value of $\chi_{s,t}$ from the region-pair frequency table $\widehat{\chi}$. 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 then we have: $\widehat{\chi}_{R_i,R_j} = |R_i| \cdot |R_j| \cdot \chi_{s,t}$, where $|R_i|$ and $|R_j|$ denote the number of nodes in the regions R_i and R_j, respectively. In other words, we compute the value of $\chi_{s,t}$ from the $\widehat{\chi}$ as follows:

$$\chi_{s,t} = \frac{\widehat{\chi}_{R_i,R_j}}{|R_i| \cdot |R_j|} \tag{6}$$

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.

Another benefit of this 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 area to a shopping area. Since these 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 $\widehat{\chi}$.

4.3 Benchmarking $E_{s,t}$ of Shortest Path API

In our caching system (see Figure 3), the shortest path API is invoked when there is a cache miss. We study how to capture the expense $E_{s,t}$ of query $Q_{s,t}$ in this section.

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, we define the expense $E_{s,t}$ of query $Q_{s,t}$ in this scenario as:

$$E_{s,t}(proxy) = 1 \quad (7)$$

Our subsequent discussion focuses on the server scenario. Let $\mathcal{ALG}$ be a shortest path algorithm to be called by the shortest path API. We denote the running time of $\mathcal{ALG}$ for query $Q_{s,t}$ as the expense $E_{s,t}(\mathcal{ALG})$.

In the following, we present a generic technique to estimate $E_{s,t}(\mathcal{ALG})$; it is applicable to any algorithm $\mathcal{ALG}$ and to arbitrary graph topology. To our best knowledge, we are the first to explore this issue. There exists work on shortest path distance estimation [**?**], but no work exists on estimating the running time of arbitrary algorithm $\mathcal{ALG}$. Even for existing shortest path indexes [**?**, **?**, **?**, **?**, **?**], only their worst-case query times have been analyzed. However, 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 5b. However, this approach is prohibitively expensive as it requires running $\mathcal{ALG}$ for $|V|^2$ times.

We plan to design an estimation technique that incurs 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 need to 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 [**?**] 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. Ref. [**?**] suggests that $|U| = 20$ is sufficient for accurate distance estimation. Figure 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.

We propose to build an *expense histogram* for recording the average expense of queries with respect to their distances, as illustrated in Figure 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 update the corresponding buckets in the histogram. This histogram requires $O(H)$ storage space and $S \cdot O(\mathcal{ALG})$ construction time. We recommend to use $H = 10$ and $S = 100$ based on our experiments.

According to our experimental results, the construction of these structures takes less than 1 minute on typical road networks.

$d(u_j, v_i)$	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 8: Example of estimating expense $\chi_{s,t}$

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 Ref. [**?**] 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 only. Second, we lookup the expense histogram and return the expense in the corresponding bucket as the estimated expense $E_{s,t}$.

Let us take the estimation of $E_{1,4}$ as an example. Using the distance estimator in Figure 8a, we estimate the shortest path distance of $P_{1,4}$ as: $\min\{3 + 6, 18 + 9\} = 9$. We then lookup the expense histogram in Figure 8b and thus estimate $E_{1,4}$ to be 2.5.

4.4 Cache Selection Algorithm

As in other static caching methods [**?**, **?**, **?**, **?**], 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 selection algorithm for selecting these paths into cache Ψ such 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. [**?**] has proposed a greedy algorithm to fill the cache with results. Thus, we also adopt the greedy approach to solve our problem. Nevertheless, there are some challenges on applying the greedy approach to our problem.

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

As an example, we consider the graph in Figure 6 and the query log $\mathcal{QL}$ in Table 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 5a. 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 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 would first pick $P_{1,6}$ and then pick $P_{3,6}$. 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 content of the cache Ψ when it chooses a path $P_{a,b}$. Observe that, if many queries that can be answered by path $P_{a,b}$ can already be answered by some existing path in Ψ then it is not worthwhile to include $P_{a,b}$ into Ψ. In the above example, the greedy approach plans to pick the path $P_{3,6}$ next time 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. There is no need for path $P_{3,6}$, but the greedy approach will still pick it.

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

DEFINITION 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 $P_{a,b}$:*

$$\Delta\overline{\gamma}(P_{a,b}, \Psi) = \frac{\gamma(\Psi \cup \{P_{a,b}\}) - \gamma(\Psi)}{|P_{a,b}|} = \sum_{P_{s,t} \in \mathfrak{U}(P_{a,b}) - \mathfrak{U}(\Psi)} \frac{\chi_{s,t} \cdot E_{s,t}}{|P_{a,b}|} \quad (8)$$

We propose a revised greedy algorithm that proceeds in rounds. A 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 the selected paths so far). 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 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 to 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 and no more paths can be inserted into the cache.

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	$\boxed{5/5}$	1/4	2/5	3/4	1/4	empty	$P_{1,6}$
2	0	0	1/4	$\boxed{2/5}$	0	1/4	$P_{1,6}$	$P_{1,6}, P_{2,7}$

Table 5: Incremental benefits of paths in our greedy algorithm (*boxed values indicate the selected paths*)

Our algorithm and time complexity: Algorithm 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 1 Revised-Greedy(Graph $G(V, E)$, Cache budget $\mathcal{B}$, Query log $\mathcal{QL}$, Frequency χ, Expense E)

```
1:  Cache $\Psi \leftarrow$ create a new cache;
2:  $H \leftarrow$ create a 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 $|\Psi| \le \mathcal{B}$ 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 $P_{a',b'}.\Delta\overline{\gamma} \ge$ top $\Delta\overline{\gamma}$ of $H$ then          ▷ actual best path
10:         if $\mathcal{B} - |\Psi| \ge |P_{a',b'}|$ 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 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. Then, we insert the path $P_{a',b'}$ into the cache Ψ when 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 to H. Eventually, H becomes empty, the loop terminates, and the cache Ψ is returned.

LEMMA 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: $\Delta\overline{\gamma}(P_{a,b}, \Psi_i) \ge \Delta\overline{\gamma}(P_{a,b}, \Psi_{i+1})$.*

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, 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 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: $\Delta\overline{\gamma}(P_{a,b}, \Psi_i) \ge \Delta\overline{\gamma}(P_{a,b}, \Psi_{i+1})$. $\Box$

We proceed to illustrate the power of the above optimization. Let's consider the second round of Table 5. Without the optimization, we must recompute the $\Delta\overline{\gamma}$ values of 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. For the other paths (e.g,, $P_{1,4}, P_{2,5}, P_{4,8}$), their upper bound $\Delta\overline{\gamma}$ values (1/3, 1/4, 1/4, from the first round) are smaller than the current $\Delta\overline{\gamma}$ value of $P_{2,7}$ (2/5), thus we do not need to recompute their current $\Delta\overline{\gamma}$ values.

We then analyze the time complexity of Algorithm 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 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 examining each subpath of $P_{a,b}$ (see Definition 6); this takes $O(|P|^2)$ time as they are $O(|P|^2)$ subpaths in a path. Multiplying the above terms together, 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 improved significantly in typical cases.

5. CACHE STRUCTURE

We focus on the design of cache structure in this section. Section 5.1 presents a structure that supports efficient cache lookup at query time. Sections 5.2 and 5.3 present compact cache structures that can accommodate as many shortest paths as possible, and thus improve the benefit of the cache.

5.1 Efficient Lookup via Inverted Lists

Upon receiving a query $Q_{s,t}$, the proxy/server lookups the cache for any path $P_{a,b}$ that can answer the query (see Figure 3). In this section, we propose a cache structure to support efficient lookup.

We propose a cache structure that involves an array of paths (see Figure 9a) and inverted lists of nodes (see Figure 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 non-empty intersection (say, Ψ_j), then we are guaranteed 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. Thus, the path Ψ_3 can be used to answer the result. Let's take the query $Q_{1,5}$ as another example. Since the inverted lists of v_1 and v_5 have empty intersection, this is a cache miss.

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 9: Path array, with inverted lists

Cache size analysis: In previous sections, we only measure the cache size in terms of the paths/nodes in the cache, but not the size of auxiliary structures (e.g., inverted lists).

Now, we measure the cache size accurately in an absolute unit (e.g., bits, bytes) and consider 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 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 with $\mathcal{I}_x = \lceil \log_2 x \rceil$ 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 to $|\Psi|$. 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.

5.2 Compact Cache via Subgraph Model

We then propose a cache structure that consists of a subgraph G_Ψ (see Figure 10a) and inverted lists (see Figure 10b). The same inverted lists from Figure 9b are used in Figure 10b. The main difference is that the path array in Figure 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 this subgraph structure is that each node (and its adjacency list) is stored at most once in G_Ψ.

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 discussed in Section 5.1. 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 s and visit a neighbor node v whenever the inverted list of v contains Ψ_j.

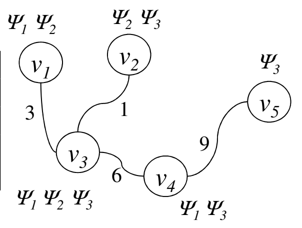
Figure 10c is a visualization of the structures in Figures 10a,b. Note that the subgraph G_Ψ only contains the adjacency lists of nodes that appear in the cache. Let's take query $Q_{2,4}$ as an example. First, we check the inverted lists of v_2 and v_4 of inverted lists in Figure 10b. Their intersection 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: $\langle v_2, v_3, v_4 \rangle$.

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 10: Subgraph representation, with inverted lists

Cache size analysis: We proceed to analyze the cache size of the above cache structure. As in Section 5.1, we let $|\Psi|$ be the number of nodes in the cache, and m be the number of paths in the cache. Let $V_\Psi \subset V$ be the number of distinct nodes in the cache, and e be the average number of neighbors per node.

The inverted lists take $|\Psi| \cdot \mathcal{I}_m$ bits, as computed 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 it is independent of the number of paths m in the cache. Thus, the subgraph representation is compact than the structure in Section 5.1. The saved space can then be used for accommodating additional paths into the cache, in turn improving the benefit of the cache.

5.3 Compact Inverted Lists

In this section, we present two compression techniques for reducing the space consumption of inverted lists. Again, the saved space can be used to accommodate more paths into the cache, improving the benefit of the cache.

Interval path ID compression: This technique represents a consecutive sequence of 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, these $j - i + 1$ path IDs can be compressed into 2 path IDs. The technique can achieve significant compression power when there are long consecutive sequences of path IDs in inverted lists.

Figure 11a shows the original inverted lists and Figure 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 $\Psi_{1,3}$.

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 prefix.

Let's consider the original inverted lists in Figure 11a. The inverted list of v_1 is the prefix of the inverted list of v_3. Figure 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 by following the parent (v_1) of its inverted list.

Both compression techniques can be combined together in order to achieve a higher compression power.

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 11: Compressed inverted lists

6. EXPERIMENTS

In this section, we evaluate the performance of our methods with our competitors on real datsets. We have implemented two variants of our methods (SPC and SPC*). They share the same techniques in Section 4, and only differ in their cache structures: (i) SPC uses a path array cache (Section 5.1), and (ii) SPC* uses the compressed graph cache (Sections 5.2,5.3). Our competitors are LRU (a dynamic caching method) and HQF (a static caching method). They have been introduced in Section 3.3. All the above methods are written in C++. We conduct our experiments on an Intel i7 3.4GHz PC running Debian.

We will evaluate the above methods for a cache located at the proxy, and for a cache located at the server. For the proxy scenario, the performance measure is the hit ratio. For the server scenario, the performance measures are: (i) the total running time of the server, and (ii) the total number of road network nodes visited.

6.1 Experimental Setting

We are unable to obtain real query log from online shortest path services (e.g., Google Map), due to their privacy policies. Thus, we can only simulate a query log from a trajectory dataset. For each trajectory, we extract its start location and its end location as the source v_s and destination v_t of a shortest path query respectively.

We have used two real datasets and their details can be found in Table 6. Each dataset consists of (i) a collection of trajectories (which can be used to simulate a query log), and (ii) a corresponding road network for the trajectories.

Following the experimental methodology of static caching [**?**], we divide the query log into two equal sets. The *historical query log* set is used for defining query frequencies and for filling the cache content. The *query workload set* set can only used for testing the performance of our method.

All caching methods, SPC, SPC*, HQF, and LRU, share a number of common settings which, unless stated otherwise, will be set to their default values: The number of levels in the kD-tree is 14

Dataset	# Trajectories / Queries	# Nodes in Road Network
Aalborg	3,286	136,347
Beijing	12,956	79,603

Table 6: Description of real datasets

(i.e. 16,384 regions). We will use the list cache representation as the default cache representation, where each vertex use one byte. The default cache size is set to 625 kB.

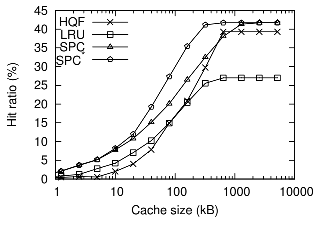
The default cache size, as well as the maximum cache size in later experiments (Fig. 12, 17, 18), is choosen baring in mind the size of the Aalborg and Beijing query logs. Had we had access to large datasets of real query logs, from online shortest path serveice providers, we would use a large cache size like 1 GB.

6.2 Caching in the Proxy Scenario

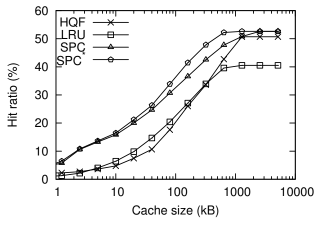
In the proxy scenario, the shortest path call API would issue a shortest path query to the server, rather than performing computation by itself (see Section 4.3). Since the total cost is dominated by the communication round-trip time with server, we use the cache hit ratio as the performance measure in this scenario.

For both datasets we vary the cache size and kD-tree levels to show the impact on the cache hit ratio. We have implemented a number of optimizations to the cache storage (See sec. 5) and show their impact on the cache hit ratio.

Effect of the cache size: In Figure 12, we measure the hit ratio of the methods while varying the cache size from 10 kB to 2.5 MB.



(a) Aalborg



(b) Beijing

Figure 12: Hit ratio vs. cache size

We can see that LRU perform well on small cache sizes, growing faster than both SPC and HQF. However, the cache hit ratio quickly levels out and stabilizes at a hit ratio of about 20 and 40 % for city X and Y respectively. Since LRU levels out very at quite small cache size and stop growing, it cannot scale to larger cache sizes. The performance of HQF is very consistent, with the hit ratio varying only slightly. The hit ratio stays at just over 1% for Aalborg experiments and around 0.2% for Beijing experiments. The low hit ratio makes it not scalable and unusable for SP caching. SPC does not grow as fast as LRU for smaller cache sizes, but while the growth of LRU quickly stabilizes then SPC continues to grow and outperforms both LRU and HQF. SPC* outperforms both LRU and SPC by a large margin, only shortly having a lower cache hit ratio at smaller cache sizes. The test on Aalborg vs. Beijing show that SPC performs much better on the Aalborg set, where as LRU performs about 50% worse. This can be explained by the fact that both methods rely on users behaving consistently over time, but where SPC relies on global consistency in user behavior, LRU need local consistency too. This makes SPC/SPC* more robust and we expect them to always outperform LRU.

Effect of the kD-tree level: In Figure 14, we vary the kD-tree level from 8 to 18 levels and show that using a kD-tree of about 14 levels

$\vert U\vert$	Aalborg	Beijing
5	35.5	33.3
10	25.4	28.2
20	22.2	23.9
40	19.4	23.0
80	19.9	21.9

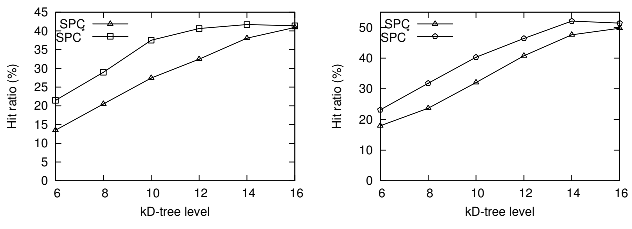
(a) varying number of landmark $|U|$

S	Aalborg	Beijing
25	23.4	28.8
50	20.7	28.4
100	22.2	23.9
200	20.4	22.7
400	21.1	21.3

(b) varying sample size S

Table 7: Average error percentage of cost estimation

can significantly increase the cache hit ratio of both the Aalborg and Beijing dataset. SPC* performs better than SPC.



(a) Aalborg (b) Beijing

Figure 13: Hit ratio vs. Levels

On both dataset SPC* outperforms SPC by a large margin. On the city X dataset SPC* achieves more than a 50% increase in hit ratio, and when compared to SPC, it achieves a relative performance gain of 500-900% at low levels up to the peak performance at level 14. On the Y dataset SPC* gets a lower hit ratio, but it still achieves more than 50% cache hit ratio. The results are still very impressive, with SPC* performing 200-400% better than SPC at all levels.

6.3 Caching in the Server Scenario

In the server scenario, the shortest path API has to invoke a shortest path algorithm. Thus, the running time is the most important performance measure. We also measure the number of nodes visited in a shortest path algorithm, which serves an indicator of the running time. As a case study, we use the Dijkstra's algorithm as the shortest path algorithm.

In the following experiments, the running time (and nodes visited) refer to the total running time (and nodes visited) for processing the entire query workload. The running time is measured in the unit of seconds.

Estimation of shortest path running cost: First, we test the estimation error of our cost estimation technique proposed in Section 4.3. We measure the error percentage in terms of the relative error between the actual cost and the estimated cost. Table 7 shows the estimation error of our technique as a function of: (a) the number of landmark $|U|$, and (b) the size of the samples S. The default values are: $|U| = 20$ and $S = 100$. The majority of errors are below 30% and thus our estimation technique is reasonably accurate.

Effect of the kD-tree level: In Figure 14, we again vary the kD-tree level from 8 to 18 levels. All three experiments shows a consistent advantage of using SPC* over SPC. SPC* still has an even higher advantage than in the proxy scenario, performing around 950% better than SPC at its best. SPC, however, has a very low hit ratio in the server scenario, on both the city X and Y datasets. In figure 15a and b we can see that SPC* is very fast at answering the query workload. SPC has a quite stable time in both figure15a and b, while the time for SPC* to answer the query workload actually decreases with more regions. In figure 16 we can see that SPC* visits far fewer nodes than SPC, during shortest path calculations. This is expected since SPC* can answer many more queries from its cache.

(a) Aalborg (b) Beijing

Figure 14: Hit ratio vs. Levels

(a) Aalborg (b) Beijing

Figure 15: Runtime vs. Levels

Effect of the cache size: In the following experiments, we vary the cache size and observe the effect on the running time and nodes visited during shortest path calculations.

In Figure 17 and 18, for both the city X and Y dataset, we can see that the graph looks similar to Figure 12, but in the upside-down manner. This is to be expected, since a higher cache hit ratio gives fewer SP calculations, and a lower runnig time when answering queries.

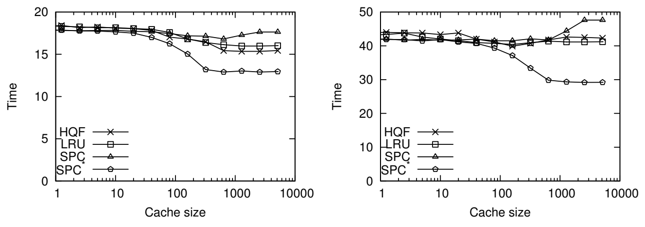
7. CONCLUSION

In this paper, we have studied the caching of shortest paths in the proxy scenario and the server scenario.

We formulate a benefit model for capturing the benefit of a path in terms of its frequency and processing it. We also propose an estimation method for estimating the cost of a specific shortest path query on any arbitrary shortest path algorithm. Our experimental

(a) Aalborg (b) Beijing

Figure 16: Nodes Visited vs. Levels



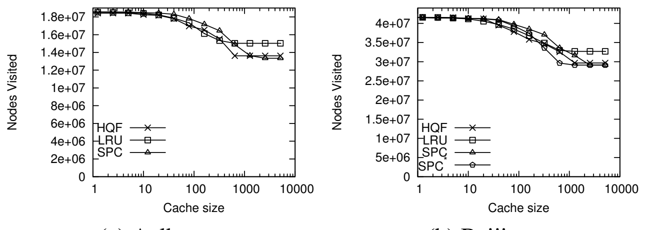
(a) Aalborg (b) Beijing

Figure 17: Runtime vs. Cache Size

results on real data show that our proposed cache selection algorithm is able to achieve high hit ratio and reduce the running time. Also, our proposed cache structures can further improve the lookup performance and maximize the utilization of cache space.

Acknowledgment

We thank Eric Lo, Jianguo Wang, and the anonymous reviewers for their insightful comments.



(a) Aalborg (b) Beijing

Figure 18: Nodes Visited vs. Cache Size