Effective Caching of Shortest Paths for Location-Based Services

Jeppe Rishede Thomsen Man Lung Yiu
Hong Kong Polytechnic University
{csjrthomsen,csmlyiu}@comp.polyu.edu.hk

Christian S. Jensen
Aarhus University
csj@cs.au.dk

ABSTRACT

Web search is ubiquitous in our daily lives. Caching has 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.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms, Performance

Keywords

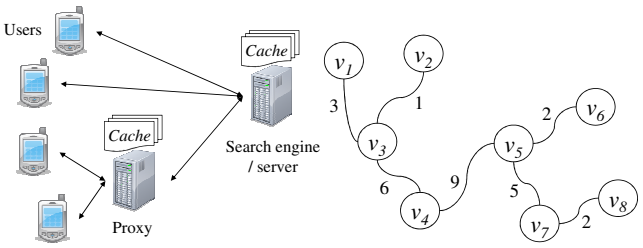
shortest path, caching, location-based service

1. INTRODUCTION

The world's population issues vast quantities of web search queries. A typical scenario for web search is illustrated in Figure 1a. 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 [1, 2, 17, 18].

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 1a 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.



(a) web search scenario (b) road network

Figure 1: Online shortest path search example

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 1b 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 Figure 1a. 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

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

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 paper is organized as follows. Section 2 studies work related to shortest path caching. 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. Section 7 concludes.

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 [17]; a request for the next result page is interpreted as an unrelated 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. [16] present pioneering work, evaluating two caching approaches (dynamic caching and static caching) on real query logs. *Dynamic caching* [6, 15, 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* [1–3, 16–18] 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 [1, 17] 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 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. Earlier techniques [1, 17] 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.

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 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 [5] 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. [5] 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 [10, 14, 22]. Zheng et al. [22] 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. [10] 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. [14] build generic semantic regions for spatial objects so that they support generic

spatial queries. However, no semantic caching techniques have been proposed for graphs or shortest paths.

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 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.

Fully-precomputed indexes, e.g., the distance index [9] or the shortest path quadtree [20], 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 [13], HiTi [12], and TEDI [21], 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.

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 paper, 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. PROBLEM SETTING

Following a coverage of background definitions and properties, we present our problem and objectives. Table 1 summarizes the notation used in the 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 notation

3.1 Definitions and Properties

We first define the notions 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 weight (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 $v_{x_0} = v_s$, $v_{x_m} = v_t$, 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 1b, 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 1): every subpath of a shortest path is also a shortest path. For example, in Figure 1b, 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 1. **Optimal subpath property (from [4]).**
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 $0 \leq i \leq j \leq m$.

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

3.2 Problem and Objectives

We adopt the architecture of Figure 1a when addressing the caching problem. 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. **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 given as the total number of nodes of shortest paths in Ψ.

As discussed in Section 2, recent literature on web search caching [1, 2, 17, 18] 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 4. **Query log.**
A query log $\mathcal{QL}$ is a collection of timestamped queries that have been issued by users in the past.

Figure 2 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.

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.

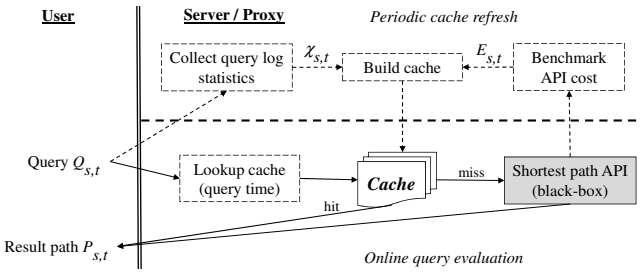


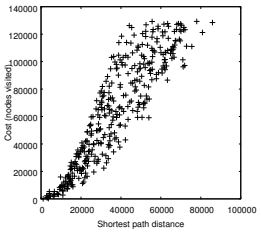
Figure 2: Components in a static caching system

DEFINITION 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 6). Figure 3a 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 3b. 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.



(a) Dijkstra

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

(b) Google Directions API

Figure 3: Cost vs. distance of a shortest path 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 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 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 $\mathcal{QL}$, 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 5.

3.3 Existing Solutions for Caching Results

We revisit existing solutions for caching web search results [16] 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 [16]. 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 1b. Let the cache budget $\mathcal{B}$ be 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 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

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.

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 [16]. 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 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.

4. BENEFIT-DRIVEN CACHING

We propose a benefit-driven approach to determining which shortest paths should be placed in the cache. Section 4.1 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 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 2 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 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:

$$\mathfrak{U}(P_{a,b}) = \{P_{s,t} \mid s \in P_{a,b} \wedge t \in P_{a,b} \wedge s \neq t\} \quad (1)$$

This set contains the queries that can be answered by $P_{a,b}$. Taking Figure 1b 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 shows the answerable query sets of other paths.

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

$\mathfrak{U}(\Psi)$, 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 $\mathfrak{U}(\Psi)$

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:

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

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 4. 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 4). 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 $P_{3,6}$: $\gamma(P_{3,6}) = 0 + 0 + 3 \cdot 3 + 0 + 0 + 0 = 9$.

$\chi_{s,t}$	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 4: Example of $\chi_{s,t}$ and $E_{s,t}$ values for the graph

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

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

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

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

[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.

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 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:

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

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.

4.2 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 [7], 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 2), the server/proxy periodically collects the query log $\mathcal{QL}$ and extracts values of $\chi_{s,t}$. The literature on static web caching [2] 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 4a. 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 $\chi_{t,s}$).

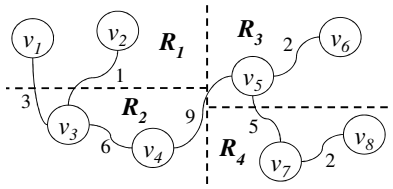
Consider the query log $\mathcal{QL}$ in Table 4 as an example. For the first query $Q_{3,6}$ in $\mathcal{QL}$, we increment the entries $\chi_{3,6}$ and $\chi_{6,3}$. Continuing this process with the other queries in $\mathcal{QL}$, we obtain the table χ shown in Figure 4a. The $\chi_{s,t}$ values in the table χ can then be used readily in the benefit model in Section 4.1.

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 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 5a, 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 5b.



(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 5: 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 5c. 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 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 5c.

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:

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

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 $\widehat{\chi}$.

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

In our caching system (see Figure 2), the shortest path API is invoked when there is a cache miss. 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, we define the expense $E_{s,t}$ of query $Q_{s,t}$ in this scenario as:

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

Our subsequent discussion focuses on the server scenario. 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 paper is the first to explore this issue. There exists work on shortest path distance estimation [19], but no work exists on estimating the running time of an arbitrary algorithm $\mathcal{ALG}$. Even for existing shortest path indexes [9, 12, 13, 20, 21], 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 4b. 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 [19] 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. [19] suggest that $|U| = 20$ is sufficient for accurate distance estimation. Figure 6a 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 6: 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 6b. 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. We recommend to use $H = 10$ and $S = 100$ on typical road networks.

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. [19] 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 6a, 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 6b and thus estimate $E_{1,4}$ to be 2.5.

4.4 Cache Construction Algorithm

As in other static caching methods [1, 2, 17, 18], 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. [17] 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 5a 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 4a. 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 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. 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 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 5: Incremental benefits of paths in our greedy algorithm (*boxed values indicate the selected paths*)

Cache construction algorithm and its 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.

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. 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.

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

```
1: Ψ ← new cache;
2: H ← new max-heap;                                  ▷ storing result paths
3: for each Q_{a,b} ∈ QL do
4:     P_{a,b}.Δγ̄ ← Δγ̄(P_{a,b}, Ψ);                  ▷ compute using χ and E
5:     insert P_{a,b} into H;
6: while |Ψ| ≤ B and |H| > 0 do
7:     P_{a',b'} ← H.pop();                           ▷ potential best path
8:     P_{a',b'}.Δγ̄ ← Δγ̄(P_{a',b'}, Ψ);              ▷ update Δγ̄ value
9:     if P_{a',b'}.Δγ̄ ≥ top Δγ̄ of H then            ▷ actual best path
10:        if B − |Ψ| ≥ |P_{a',b'}| then               ▷ enough space
11:            insert P_{a',b'} into Ψ;
12:    else                                           ▷ not the best path
13:        insert P_{a',b'} into H;
14: return Ψ;
```

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

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

We proceed to illustrate the power of the above optimization. Consider the second round shown in Table 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 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 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.

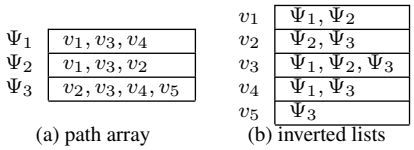
5. CACHE STRUCTURE

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 enable a cache to accommodate as many shortest paths as possible, thus improving the benefit of the cache.

5.1 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 2). We propose a structure that enables efficient cache lookup.

The proposed structure involves an array of paths (see Figure 7a) and inverted lists of nodes (see Figure 7b). 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.



(a) path array (b) inverted lists

Figure 7: Path array, with inverted lists

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 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.

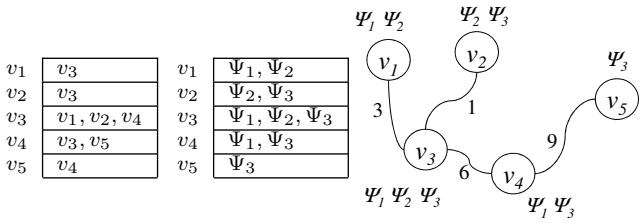
In the path array, each node can be represented by $\mathcal{I}_{|V|}$ bits. Thus, the path array occupies $|\Psi| \cdot \mathcal{I}_{|V|}$ bits. In each inverted list, each path ID can be represented by $\mathcal{I}_m$ bits. Note that the total number of path IDs in inverted lists equals $|\Psi|$. Thus,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

5.2 Compact Cache via a Subgraph Model

We propose a cache structure that consists of a subgraph G_Ψ (see Figure 8a) and inverted lists (see Figure 8b). The same inverted lists as in Figure 7b are used in Figure 8b. The main difference is that the path array in Figure 7a 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.

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 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 v_s and visit a neighbor node v' whenever the inverted list of v' contains Ψ_j.

Figure 8c visualizes the structures in Figures 8a,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 8b. Their intersection



(a) subgraph G_Ψ (b) inverted lists (c) visualization

Figure 8: Subgraph representation, with inverted lists

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 5.1, 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.

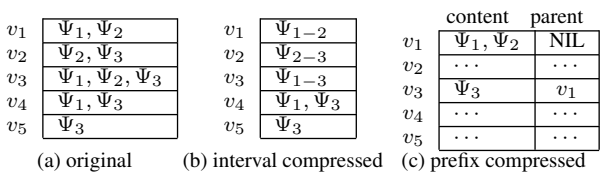
Note that $|V_\Psi|$ is upper bounded by $|V|$ and that it is independent of the number of paths m in the cache. Thus, the subgraph representation is more compact than the structure in Section 5.1. The saved space can be used for accommodating additional paths into the cache, in turn improving the benefit of the cache.

5.3 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 9a shows the original inverted lists, and Figure 9b 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 $\Psi_{1,3}$.



(a) original (b) interval compressed (c) prefix compressed

Figure 9: 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 9a. The inverted list of v_1 is a prefix of the inverted list of v_3. Figure 9c 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.

6. 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 3.3. Our methods Shortest-Path-Cache (SPC) and its optimized variant (SPC*) share the same techniques in Section 4. Regarding the cache structures, LRU, SPC, and HQF use a path array cache (Section 5.1), whereas SPC* uses a compressed graph cache (Sections 5.2 and 5.3). All methods answer queries by using the optimal subpath property (Lemma 1). We implemented all methods in C++ and conducted experiments on an Intel i7 3.4GHz PC running Debian.

Section 6.1 covers the experimental setting. Then Section 6.2 and Section 6.3 presents findings for caches in the proxy scenario and the server scenario, respectively.

6.1 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 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 [17], 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 [11] 4,401 trajectories	From *downloads.cloudmade.com* 129k nodes, 137k edges
Beijing	Geo-Life GPS data [23] 12,928 trajectories	From *downloads.cloudmade.com* 76k nodes, 85k edges

Table 6: Description of real data sets

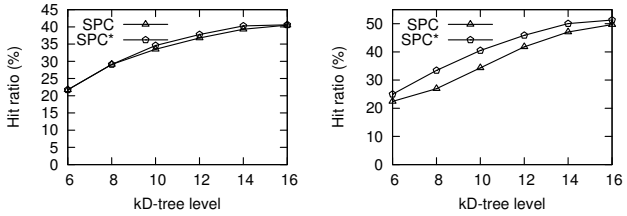
Default parameters: Since the Aalborg and Beijing query logs are not large, we use scaled down cache sizes in the experiments. Thus, the default cache size is 625 kBytes and the maximum cache size is 5 MBytes. The default number of levels in the kD-tree (for query statistics extraction) is 14.

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.

6.2 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 4.3). 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 10 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.

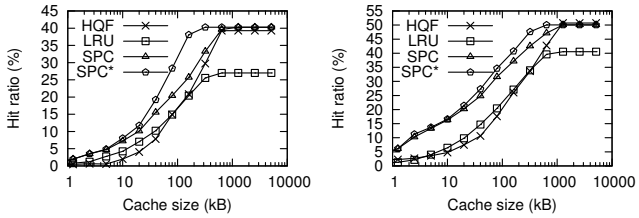


(a) Aalborg (b) Beijing

Figure 10: Hit ratio vs. levels

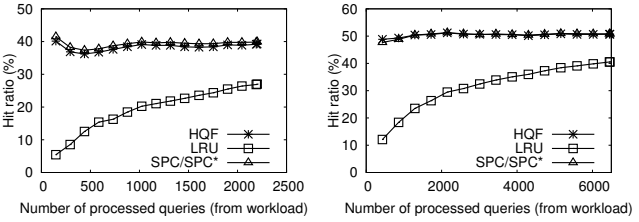
Effect of the cache size: Figure 11 shows the hit ratio of the methods as a function of the cache size. SPC has a lower hit ratio than SPC* for the reasons explained above. Observe that SPC* achieves double the hit ratio of HQF and LRU for small cache sizes (below 100 kBytes). This is because SPC* exploits historical query statistics to choose paths with high benefit values for inclusion into the cache. Our benefit model (Section 4.1) 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 12 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 11: Hit ratio vs. cache size



(a) Aalborg (b) Beijing

Figure 12: 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*.

6.3 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 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 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.

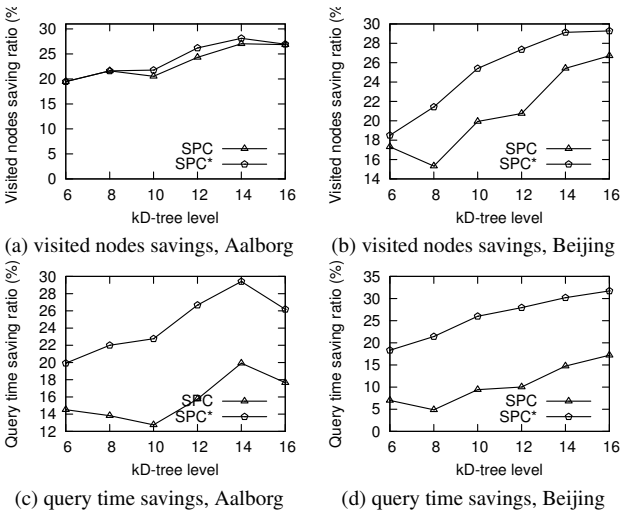
Effect of the kD-tree level: We investigate the effect of the number of levels in the kD-tree on the savings ratios of our methods (SPC and SPC*).

Figure 13 plots the savings ratios when Dijkstra is used as the shortest path algorithm. Like the trends in Figure 10, both methods obtain higher node savings ratios when the kD-tree used contains many levels and captures more accurate statistics (see Figure 13a,b).

However, there is a significant difference in the query time savings ratios (see Figure 13c,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 14 shows the savings ratios of our methods when A* is used instead of Dijkstra. The node savings ratios of SPC and SPC* in Figure 14a,b exhibit similar trends as seen in Figure 13a,b. Note that the query time savings ratio of SPC is negative (see Figure 14c,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 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.

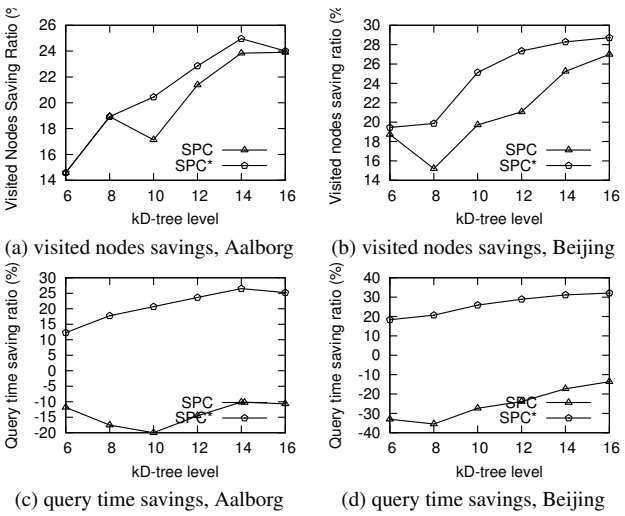
In summary, SPC* consistently performs better than SPC, especially for the query time savings ratio.



(a) visited nodes savings, Aalborg (b) visited nodes savings, Beijing

(c) query time savings, Aalborg (d) query time savings, Beijing

Figure 13: Performance savings vs. levels, using Dijkstra



(a) visited nodes savings, Aalborg (b) visited nodes savings, Beijing

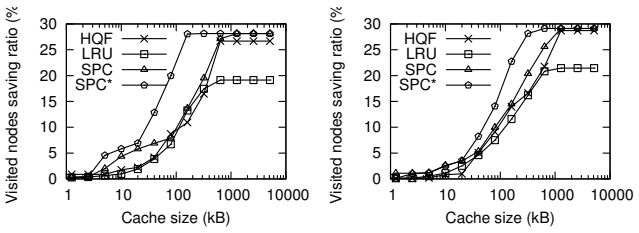
(c) query time savings, Aalborg (d) query time savings, Beijing

Figure 14: Performance savings vs. levels, using A^*

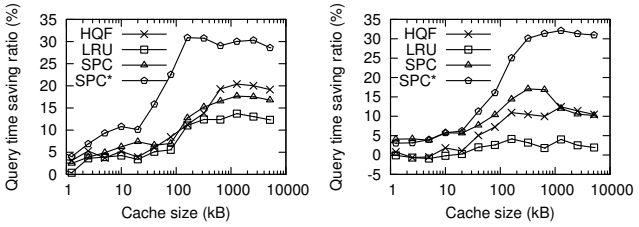
Effect of the cache size: We proceed to study the impact of the cache size on the performance savings ratios of all caching methods (SPC, SPC*, HQF, LRU).

Figure 15 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 15a,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 15c,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 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 15a,b.

Figure 16 plots the savings ratio of the caching methods when using A*. The trends of the methods are similar to those in Figure 15. The only difference is that the query time savings ratio of HQF,



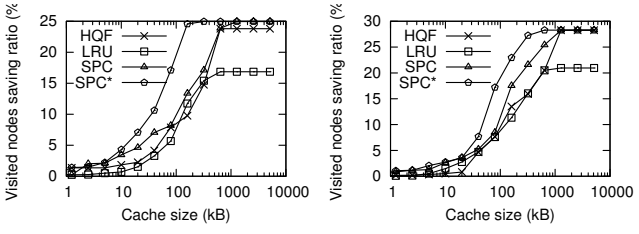
(a) visited nodes savings, Aalborg (b) visited nodes savings, Beijing



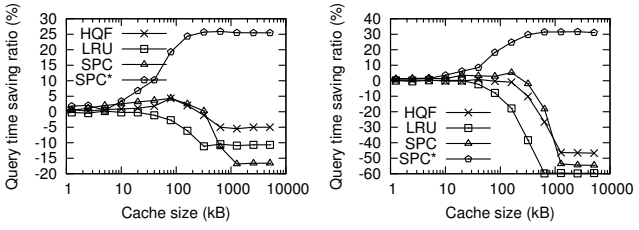
(c) query time savings, Aalborg (d) query time savings, Beijing

Figure 15: Performance savings vs. cache size, using Dijkstra

LRU, and SPC are even more negative in Figure 16c,d. Since A* is much more efficient than Dijkstra, the high cache lookup overheads of HQF, LRU, and SPC become more apparent. They 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.



(a) visited nodes savings, Aalborg (b) visited nodes savings, Beijing



(c) query time savings, Aalborg (d) query time savings, Beijing

Figure 16: Performance savings vs. cache size, using A^*

7. 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 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.

Our static caching problem is analogous to materialized view selection in data warehousing [8]. In future, we aim to utilize their ideas to build a shortest path cache with quality guarantees.

Acknowledgments

C. S. Jensen was supported in part by the EU STREP project, Reduction. We thank Eric Lo, Jianguo Wang, Yu Li, and the anonymous reviewers for their insightful comments.

8. REFERENCES

[1] I. S. Altingövde, R. Ozcan, and Ö. Ulusoy. A Cost-Aware Strategy for Query Result Caching in Web Search Engines. In *ECIR*, pp. 628–636, 2009.

[2] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The Impact of Caching on Search Engines. In *SIGIR*, pp. 183–190, 2007.

[3] R. A. Baeza-Yates and F. Saint-Jean. A Three Level Search Engine Index Based in Query Log Distribution. In *SPIRE*, pp. 56–65, 2003.

[4] T. H. Cormen, C. E. Leiserson, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[5] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *VLDB*, pp. 330–341, 1996.

[6] Q. Gan and T. Suel. Improved Techniques for Result Caching in Web Search Engines. In *WWW*, pp. 431–440, 2009.

[7] M. C. González, C. A. Hidalgo, and A. L. Barabási. Understanding Individual Human Mobility Patterns. *Nature*, 453(7196):779–782, 2008.

[8] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD*, pages 205–216, 1996.

[9] H. Hu, D. L. Lee, and V. C. S. Lee. Distance Indexing on Road Networks. In *VLDB*, pp. 894–905, 2006.

[10] H. Hu, J. Xu, W. S. Wong, B. Zheng, D. L. Lee, and W.-C. Lee. Proactive Caching for Spatial Queries in Mobile Environments. In *ICDE*, pp. 403–414, 2005.

[11] C. S. Jensen, H. Lahrmann, S. Pakalnis, and J. Runge. The Infati Data. *CoRR*, cs.DB/0410001, 2004.

[12] S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE TKDE*, 14(5):1029–1046, 2002.

[13] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt. Hierarchical Graph Embedding for Efficient Query Processing in Very Large Traffic Networks. In *SSDBM*, pp. 150–167, 2008.

[14] K. Lee, W.-C. Lee, B. Zheng, and J. Xu. Caching Complementary Space for Location-Based Services. In *EDBT*, pp. 1020–1038. 2006.

[15] X. Long and T. Suel. Three-Level Caching for Efficient Query Processing in Large Web Search Engines. In *WWW*, pp. 257–266, 2005.

[16] E. P. Markatos. On Caching Search Engine Query Results. *Computer Communications*, 24(2):137–143, 2001.

[17] 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.

[18] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Static Query Result Caching Revisited. In *WWW*, pp. 1169–1170, 2008.

[19] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast Shortest Path Distance Estimation in Large Networks. In *CIKM*, pp. 867–876, 2009.

[20] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable Network Distance Browsing in Spatial Databases. In *SIGMOD*, pp. 43–54, 2008.

[21] F. Wei. TEDI: Efficient Shortest Path Query Answering on Graphs. In *SIGMOD*, pp. 99–110, 2010.

[22] B. Zheng and D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In *SSTD*, pp. 97–116, 2001.

[23] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In *WWW*, pp. 791–800, 2009.