IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 9, 2017, accepted July 1, 2017, date of publication July 11, 2017, date of current version July 31, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2725308

Efficient Path Query Processing Through Cloud-Based Mapping Services

DETIAN ZHANG[1], YUAN LIU[1], AN LIU[2], XUDONG MAO[3], AND QING LI[3], (Senior Member, IEEE)
[1]School of Digital Media, Jiangnan University, Wuxi 214122, China
[2]School of Computer Science, Soochow University, Suzhou 215006, China
[3]Department of Computer Science, City University of Hong Kong, Hong Kong

Corresponding author: An Liu (anliu@suda.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Project 61572336 and Project 61672264, in part by the National Key Technology R&D Program of China under Project 2015BAH54F01, and in part by a research grant from the City University of Hong Kong under Project 93601539.

ABSTRACT Shortest path queries have been widely used in location-based services (LBSs). To calculate the shortest path from an origin to a destination, an LBS provider usually needs to know the map data of the underlying road network, which can be rather costly especially if such data need to be kept continuously and up to date. A cost-effective way is that LBS providers outsource the shortest path query processing to cloud-based mapping services such as Google Maps, by retrieving the detailed path information from them through external requests. Due to the high cost of accessing data through external requests and the usage limits of mapping services, we propose two optimization techniques in this paper, namely, path sharing and path caching, to reduce the number of external requests and the user query response time. Unlike previous work where the underlying road network is given, this paper optimizes path query processing only based on query origins and destinations. The basic idea of path sharing optimization is that the path information of a query can be shared with another query q if q values origin and destination both lie on the path. To achieve this, we propose an effective method to compute whether or not a query origin/destination lies on a path only based on the Euclidean distance between them. Path caching, an extension of path sharing, lets an LBS provider answer path queries directly based on cached paths. To accomplish this, we first formulate the problem of constructing path cache as a knapsack problem and design a greedy algorithm to solve it; then, we devise an effective cache structure to support efficient cache lookup. Extensive experiments on Bing Maps and real data sets are conducted, and the results show the efficiency, scalability, and applicability of our proposed approaches.

INDEX TERMS Shortest path queries, location-based services, mapping services, cloud computing, path sharing, path caching.

I. INTRODUCTION

With the fast development of wireless networks and GPS-enabled devices, location-based services (LBS) are getting more and more popular and important in our daily life [1], such as finding the shortest driving route from our home to the train station by Google Maps, locating the nearest taxi with the help of Uber, searching all the restaurants in a given region by Yelp, etc.. The basic principle underlying LBS are spatio-temporal queries and their processing. Typical spatio-temporal queries include shortest path queries, k-nearest-neighbor queries, range queries and so on, among which shortest path queries are the most widely used.

To calculate the shortest path from an origin to a destination, an LBS provider usually needs to know the topology of the underlying road network and the real-time traffic data (for computing the shortest path by driving). However, not every LBS provider, especially those small and start-up companies, has enough resources to collect and maintain so much data not to mention the need to keep the data accurate. A cost-effective way is that LBS providers outsource the shortest path processing to cloud-based mapping services like Google Maps, Bing Maps, MapQuest Maps, Baidu Maps, Amap and so on, and focus on their own POI (Point of Interests) maintaining and business processing. In other words, when an LBS provider receives a user query to access the shortest path from an origin to a destination, it can pass the path query to a mapping service through an external request, instead of processing the query by itself.

These cloud-based mapping services are owned by large and specialized companies, such as Google, Microsoft and Baidu, which have enough resources to collect and maintain map data, like collecting GPS data to obtain the road network topology and deploying road-side cameras and sensors to monitor live traffic conditions. Moreover, they are provided with such user-friendly and easy-to-use APIs like the Google Maps Directions API, the Bing Maps Routes API and the Baidu Maps Direction API, that anyone can easily retrieve the shortest path between two locations through them. Figure 1 shows an example of the request and response format of Google Directions API. The Bing Maps Routes API and Baidu Maps Direction API share the similar format.

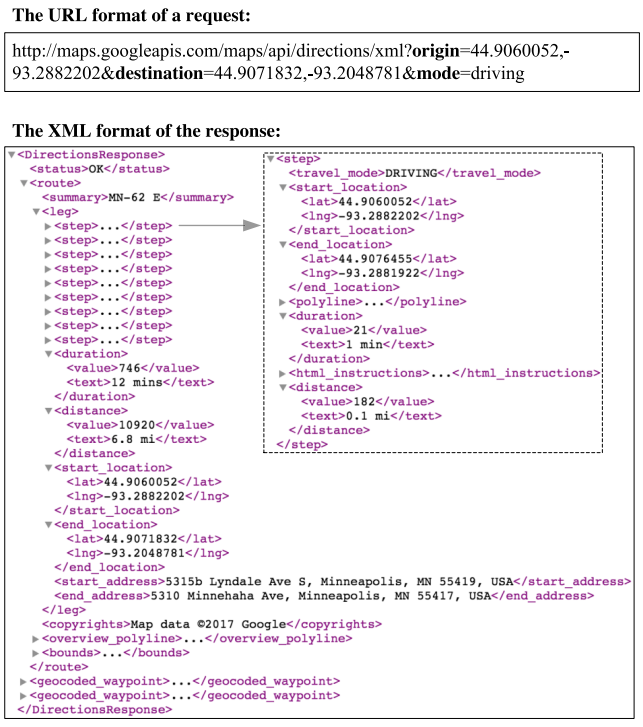


FIGURE 1. The request and response format of Google Directions API.

The request is a URL query string, whose parameters contain the origin and destination locations in latitude and longitude, as well as the travel mode. In this example, the origin is at (44.9060052,-93.2882202), the destination is at (44.9071832,-93.2048781), and the travel mode is ''driving''. The response is an XML (or JSON, which can be selected in the URL request) document that stores a sequence of path/route steps from the origin to the destination. Each path step, enclosed by ⟨step⟩ tags, contains its start and end locations, its travel distance, and its travel time by driving (see the ⟨duration⟩ tag). The start location, end location, travel distance and travel time of the first step in this path (see the dashed box in Figure 1) are ⟨44.9060052, −93.2882202⟩, ⟨44.9076455,−93.2881922⟩, 182 meters and 21 seconds, respectively. The rest of path steps are folded for brevity. Besides, the XML response contains the total travel distance (e.g., 10,920 meters) and the total travel time (e.g., 746 seconds) on the path, i.e., the sum of travel distance and time on all steps.

In this way, with the help of cloud-based mapping services, an LBS provider does not need to process map data by itself. Instead, it can focus on its core business activities and reduce its operational costs. For example, `Yelp.com`, a leading Internet consumer rating and review website, outsources its mapping services to Google Maps.

However, retrieving the shortest paths from cloud-based mapping services suffers from some critical limitations, including the following: [2]: (1) It is costly to access the shortest path information from a cloud-based mapping service through external requests. For example, retrieving travel distance and time from the Bing Maps Routes API takes 502 ms while the time needed to read a cold and hot 8 KB buffer page from the disk is 27 ms and 0.0047 ms, respectively [3]. (2) There is a charge on the number of external requests to a cloud-based mapping service. For example, the Google Maps Directions API allows only 2,500 requests per day for evaluation users and 100,000 requests per day for business license users [4]. An LBS provider needs to pay for higher usage limits. Therefore, when an LBS provider endures high workload like a large number of concurrent path queries, it needs to issue a large number of external requests to cloud-based mapping services, which not only yields the high business operation cost, but also induces the long response time to its users.

To reduce the number of external requests to cloud-based mapping services, and to speed up the query response time to LBS users, we propose two optimization techniques in this paper, namely *path sharing* and *path caching*.

Given a shortest path query $q_i = (o_i, d_i)$, the basic idea of the *path sharing* optimization is that the path information from the origin o_i to the destination d_i (i.e., $Path(o_i \rightarrow d_i)$) can be shared with another query $q_j = (o_j, d_j)$ if both of its origin o_j and destination d_j lie on $Path(o_i \rightarrow d_i)$. Since the underlying road network topology is assumed to be unknown to the LBS provider, we propose an effective method in this paper to compute whether or not o_j and d_j lie on $Path(o_i \rightarrow d_i)$ according to their locations.

Path caching is an extension of *path sharing*, which caches selected paths at an LBS provider. When the LBS provider receives a user's path query, it checks the path cache first. If there is a cache hit, i.e., the cache contains the shortest path of the query, it directly returns the cached path to the query user; otherwise, it issues an external request so as to retrieve the desired path and return it to the user. Thus, it can effectively reduce the number of external requests and the user query response time. To this end, we construct the path cache for an LBS provider based on historical query paths as [5] and [6]. We first formulate the problem of constructing path cache as a knapsack problem, which is NP-complete. We design a greedy algorithm to solve it according to the concept of sharing ability per node. To save cache space and support efficient cache lookup, we also design an effective cache structure.

The main contributions of our work in this paper can be summarized as follows:

- We have proposed an effective method to compute whether or not a query origin/destination lies on a path without knowing the underlying road network topology for path sharing.
- We have formulated the problem of constructing path cache as a knapsack problem, and devised a greedy algorithm to solve it.
- We have designed an effective cache structure to save cache space and support efficient cache lookup.
- We have conducted extensive experiments to evaluate the performance of the proposed algorithms by using a real cloud-based mapping service and real data sets.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III describes the system model. Section IV presents the path sharing optimization. Section V describes the path caching optimization. Experimental results are given and analyzed in Section VI. Finally, Section VII concludes this paper.

II. RELATED WORK

A. SHORTEST PATH ALGORITHMS

Shortest path query processing has always been a hot research topic in academia and industry [7]. The most classic shortest path algorithms include Dijkstra's algorithm [8], BellmanFord algorithm [9], and A* search algorithm [10]. In recent years, more efficient algorithms have been developed, such as parallel approaches for the shortest path computation over large-scale graphs or networks like Δ-stepping [11], PHAST [12] and Pregel [13]. To further improve the path processing efficiency, pre-computing techniques are fully exploited, including transit node routing [14], arterial hierarchy [15] and metric backbone [16]. Unsurprisingly, all the researches are based on the condition that the underlying graph structure or network topology is known to the server.

B. QUERY PROCESSING THROUGH CLOUD-BASED MAPPING SERVICES

Different from the above mentioned studies, our approach lets an LBS provider access the shortest paths through external requests to cloud-based mapping services, e.g., Google Maps.

There have been some studies related to spatial query processing through cloud-based mapping services, including k-nearest-neighbor query [2], [17], [18], concurrent k-nearest-neighbor query [19], range query [18] and the shortest path query [20], [21] processing algorithms in road networks, where the distance metric is travel time instead of network distance. Because of dynamic traffics, travel time is harder for LBS providers to obtain than network distance. Therefore, these studies assume that LBS providers have the static road network data in their local databases, while the distance metric (i.e., travel time) is retrieved from cloud-based mapping services through external requests. To reduce the number of such external requests, and to speed up the query response time for users, pruning [2], [17], grouping [2], [17], direction sharing [17], [20], [21], parallel requesting [19], [21], route log [18] and waypoint [20], [21] techniques are proposed based on the distribution of query origins and destinations over the underlying road network.

Even though these researches are based on cloud-based mapping services, and share with our objectives to minimize the number of external requests and the user query response time, their optimization techniques and algorithms are designed based on given road networks.

C. SHORTEST PATH CACHING

There have been some studies on path sharing and caching [5], [22]. Thomsen *et al.* [5] proposed a path cache SPC to reduce the path computation cost for a server. To construct a good path cache, SPC computes a benefit value to score a path when determining whether or not to preserve it in the cache. The benefit value of a path is the summation of the benefit value of each sub-path in the entire path. The benefit value of a path consists of two parts: the popularity of a path and its expense. The popularity of a path is evaluated based on the number of its sub-paths, and the expense of a path represents the computational time of the shortest path algorithm. To support efficient cache lookup, a subgraph model and inverted lists are utilized in the cache structure. However, SPC works only when a cached path matches perfectly with the query. In [22], another path caching system, i.e., Path Planning by Caching (PPC), is proposed. PPC leverages the partially matched cached paths to answer part(s) of a new query. As a result, the server only needs to compute the unmatched path segments, thus significantly relieving the workload of the overall system.

However, both SPC and PPC still need the underlying graph or network data to construct the cache (e.g., to estimate path expense by SPC) or to answer a new query (e.g., to compute the unmatched path segments by PPC). In contrast, our proposed optimization techniques, i.e., path sharing and path caching, are only based on the arriving queries, and does not need any extra information like the underlying network data.

III. SYSTEM MODEL

A. SYSTEM ARCHITECTURE

Figure 2 shows our proposed system architecture, which consists of three entities:

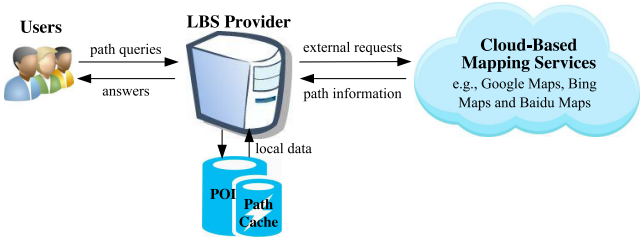


FIGURE 2. System architecture.

Users: Users browse POIs and the related information provided by an LBS provider (e.g., restaurant information in Yelp), and send path queries to the LBS provider to retrieve the shortest paths from their locations to the specified POIs (e.g. finding the shortest path from our home to the highest rating restaurant in the city).

LBS provider: The LBS provider is the owner who has at least one kind of POI datasets like restaurants, hotels, gas stations and so on, and provides services to its users. Since the LBS provider may not have enough resources to possess map data by itself, it outsources path processing to a cloud-based mapping service. In other words, when receiving a path query from a user, the LBS provider will retrieve the path information from the cloud-based mapping service by issuing an external request, and then return the result to the user.

Cloud-based mapping service: Typical cloud-based mapping services are Google Maps, Bing Maps and Baidu Maps. These mapping services usually are possessed by large and specialized companies or enterprises, which have enough resources to host and update map data continuously to keep them accurate. Besides, they are provided with user-friendly and easy-to-use APIs, such as the Google Maps Directions API, the Bing Maps Routes API and the Baidu Maps Direction API.

B. OUR PROBLEM AND OBJECTIVES

Given a set of shortest path queries $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$ arriving at an LBS provider concurrently or within a short time period, the LBS provider returns the shortest path with the detailed step information from o_i to d_i for each $q_i = (o_i, d_i)$ in $\mathcal{Q}$ like the XML format of Figure 1, which is retrieved from the cloud-based mapping service through external requests.

Definition 1 (Shortest Path Query): A shortest path query $q_i = (o_i, d_i)$ sent from a user, consists of an origin o_i and a destination d_i.

Definition 2 (External Request): An external request, which is issued by an LBS provider to a cloud-based mapping service, consists of at least an origin and a destination, e.g., the URL format of Figure 1.

Due to the critical limitations to accessing the path information from a cloud-based mapping service as is stated in Section I, i.e., high cost and usage limits, our objectives are to reduce the number of external requests issued by the LBS provider and speed up the query response time to users.

IV. PATH SHARING

In this section, we will introduce the path sharing optimization to reduce the number of external requests, as well as the user query response time. We will first introduce the optimal sub-path property. Then, we will propose an effective method to compute whether or not a path can be shared with another query without knowing the underlying road network topology. At last, we will present the algorithm of path sharing.

A. OPTIMAL SUB-PATH PROPERTY

As is presented in Section I, the path information returned by the cloud-based mapping service includes the detailed turn-by-turn step information. For example, the shortest path of $q_1 = (o_1, d_1)$ is $Path(o_1 \rightarrow d_1) = \{\langle o_1 = (0, 9), v_1 = (3, 5), l(o_1 \rightarrow v_1) = 6\rangle, \langle v_1 = (3, 5), v_2 = (6, 5), l(v_1 \rightarrow v_2) = 3\rangle, \langle v_2 = (6, 5), v_3 = (6, 2), l(v_2 \rightarrow v_3) = 4\rangle, \langle v_3 = (6, 2), d_1 = (9, 2), l(v_3 \rightarrow d_1) = 3\rangle\}$, as is shown in Figure 3, where a, b and $l(a \rightarrow b)$ are the start location, the end location and the travel distance of the path step $a \rightarrow b$, respectively. For the sake of simplicity, we denote $Path(o_1 \rightarrow d_1) = \{o_1 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow d_1\}$.

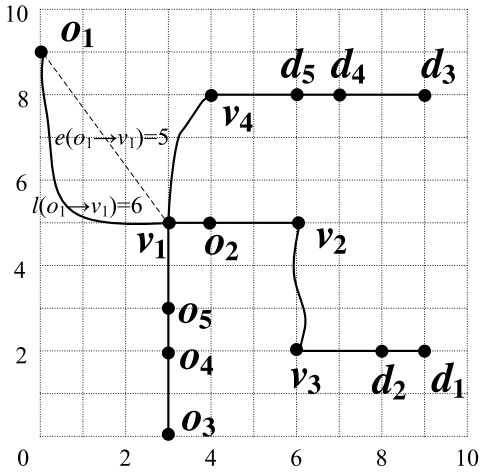


FIGURE 3. Path information example.

The shortest paths exhibit the optimal sub-path property [23], i.e., every sub-path of the shortest path is also the shortest path (see Lemma 1). In other words, the shortest path $Path(o_i \rightarrow d_i)$ returned from the cloud-based mapping service for a query $q_i = (o_i, d_i)$ can be shared with another query $q_j = (o_j, d_j)$, if both o_j and d_j lie on the path (If the path is directional, o_j and d_j should lie on it in order), i.e., $o_j \in Path(o_i \rightarrow d_i)$ and $d_j \in Path(o_i \rightarrow d_i)$; thus, there is no need for the LBS provider to issue another external request for $q_j = (o_j, d_j)$ any more to retrieve $Path(o_j \rightarrow d_j)$. As a result, the number of external requests and the response time for q_j can be reduced.

Lemma 1 (Optimal Sub-Path Property): Let $Path(o_i \rightarrow d_i) = \{o_i \rightarrow \cdots \rightarrow o_j \rightarrow \cdots \rightarrow d_j \rightarrow \cdots \rightarrow d_i\}$ be the shortest path for a query with the origin location o_i to the destination location d_i. $Path(o_j \rightarrow d_j) = \{o_j \rightarrow \cdots \rightarrow d_j\}$ is also the shortest path from o_j to d_j.

Proof: See the proof of [23, Lemma 24.1]. □

B. PATH SHARING CHECK

Since the LBS provider does not have the topology data of the underlying road network, it will be impossible to check whether or not a path $Path(o_i \rightarrow d_i)$ can be shared with another query $q_j = (o_j, d_j)$ simply based on their locations

in the road network. In this section, we design an effective method for path sharing check solely based on $Path(o_i \rightarrow d_i)$ and $q_j = (o_j, d_j)$, which does not need any other information.

For $Path(o_i \rightarrow d_i)$, we first decide for each path step $a \rightarrow b$ if it is a straight line or not. If the travel distance of a step is equal to its Euclidean distance, i.e., $l(a \rightarrow b) = e(a \rightarrow b)$, which means that this path step is the shortest way from a to b in the Euclidean space, the path step is a straight line; otherwise, it is not. For a straight-line path step, we check whether or not the origin o_j of the query q_j lies on it still by the Euclidean distance between them. If $e(a \rightarrow b) = e(a \rightarrow o_j) + e(o_j \rightarrow b)$, we conclude that o_j lies on the step $a \rightarrow b$ based on Lemma 2. Therefore, we have o_j lies on the path, i.e., $o_j \in Path(o_i \rightarrow d_i)$. Similarly, we can check whether or not d_j lies on the path. If both o_j and $d_j \in Path(o_i \rightarrow d_i)$, the path information of $Path(o_i \rightarrow d_i)$ can be shared with the query $q_j = (o_j, d_j)$.

Lemma 2 (Location-on-Path Check): Given a path from location a to location b and another location c, if $l(a \rightarrow b) = e(a \rightarrow b)$ *and* $e(a \rightarrow b) = e(a \rightarrow c) + e(c \rightarrow b)$, *we have c located on the path, where* $l(\blacktriangle \rightarrow \blacktriangledown)$ *and* $e(\blacktriangle \rightarrow \blacktriangledown)$ *stands for the travel distance and the Euclidean distance from* ▲ *to* ▼, *respectively.*

Proof: As $l(a \rightarrow b) = e(a \rightarrow b)$, we have the path from a to b is a straight line. Assume that c not lies on the path, we have $e(a \rightarrow c) + e(c \rightarrow b) > e(a \rightarrow b)$ based on "the sum of any two sides of a triangle is greater than the third one". This contradicts with the given condition: $e(a \rightarrow b) = e(a \rightarrow c) + e(c \rightarrow b)$. □

Take $Path(o_1 \rightarrow d_1) = \{o_1 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow d_1\}$ as an example, the start location o_1, the end location v_1, and the travel distance $l(o_1 \rightarrow v_1)$ of the path's first step $o_1 \rightarrow v_1$ are $(0, 9)$, $(3, 5)$, and 6, respectively, as is shown in Figure 3. It is easy to calculate that the Euclidean distance of the step is 5, i.e., $e(o_1 \rightarrow v_1) = 5$, which is not equal to its travel distance. Therefore, the path step $o_1 \rightarrow v_1$ is not a straight line. Similarly, we can conclude that $v_1 \rightarrow v_2$ and $v_3 \rightarrow d_1$ are straight lines, while $v_2 \rightarrow v_3$ is not.

Given another query $q_2 = (o_2, d_2)$, whose origin location o_2 and destination location d_2 are $(4, 5)$ and $(8, 2)$, respectively, as $e(v_1 \rightarrow o_2) + e(o_2 \rightarrow v_2) = 1 + 2 = 3$, which is equal to $e(v_1 \rightarrow v_2) = 3$, and $v_1 \rightarrow v_2$ is a straight line, we can induce that o_2 is located on $v_1 \rightarrow v_2$ by Lemma 2. Likewise, d_2 is located on $v_3 \rightarrow d_1$. As both the origin and destination locations of q_2 lie on $Path(o_1 \rightarrow d_1)$, the path information of q_1 can be utilized to answer q_2, i.e., $Path(o_2 \rightarrow d_2) = \{o_2 \rightarrow v_2 \rightarrow v_3 \rightarrow d_2\}$ (see Figure 3). Therefore, only one external request is needed instead of two for the LBS provider to answer q_1 and q_2.

C. PATH SHARING ALGORITHM

Given a set of path queries $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$ arriving at an LBS provider concurrently or within a short time period, to maximize the path sharing optimization, the LBS provider should first process the query, the path of which has the maximum *sharing ability* (see Definition 3).

Definition 3 (Sharing Ability): Let $\mathcal{Q}$ *be a set of path queries, and* $Path(o_i \rightarrow d_i)$ *be the shortest path from origin* o_i *to destination* d_i. *The* sharing ability *of* $Path(o_i \rightarrow d_i)$ *with respect to* $\mathcal{Q}$ *(denoted as* $SA(Path(o_i \rightarrow d_i), \mathcal{Q})$*) is the number of queries in* $\mathcal{Q}$, *where the path information of* $Path(o_i \rightarrow d_i)$ *can be shared with these queries, i.e.,*

$$SA(Path(o_i \rightarrow d_i), \mathcal{Q}) = |\{q_j = (o_j, d_j) \in \mathcal{Q} \mid Path(o_j \rightarrow d_j) \subset Path(o_i \rightarrow d_i)\}|.$$

However, the path information is unknown until the LBS provider issues external requests to the cloud-based mapping services. It will be impossible to evaluate the path sharing ability for each query in $\mathcal{Q}$ in advance. Considering that the path with the longer Euclidean distance has the bigger chance to be shared by other queries (i.e., the larger sharing ability), our plan is to process queries in $\mathcal{Q}$ based on the Euclidean distance from the origins to their respective destinations, as is shown in Algorithm 1 (denoted as PSA).

Algorithm 1 Path Sharing Algorithm (PSA)

```
input: the query set Q
compute the Euclidean distance from the origin to the
  destination for each query in Q;
while Q is not empty do
  q_s ← the query with the largest Euclidean distance in
    Q;
  retrieve Path(o_s → d_s) for q_s by issuing an external
    request to the cloud-based mapping service;
  for each query q_i = (o_i, d_i) in Q do
    if both o_i and d_i locate on Path(o_s → d_s) then
      use Path(o_s → d_s) to answer q_i;
      remove q_i from Q;
    end if
  end for
end while
```

For example, given a query set $\mathcal{Q} = \{q_1, q_2, q_3, q_4, q_5\}$, it is easy to compute the Euclidean distance for these queries based on their query origins and destinations, as is shown in Figure 4. Since the Euclidean distance of q_1 is the largest, PSA retrieves the path of q_1 first through an external request. Suppose that $Path(o_1 \rightarrow d_1) = \{o_1 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow d_1\}$ (as is shown in Figure 3), we can get that both o_2 and d_2 are located on the path through path sharing check (see Section IV-B), i.e., $Path(o_1 \rightarrow d_1)$ can also answer q_2, so both q_1 and q_2 are removed from $\mathcal{Q}$. Currently, $\mathcal{Q} = \{q_3, q_4, q_5\}$ and q_3 has the largest Euclidean distance, so $Path(o_3 \rightarrow d_3) = \{\langle o_3 = (3, 0), v_1 = (3, 5), l(o_3 \rightarrow v_1) = 5\rangle, \langle v_1 = (3, 5), v_4 = (4, 8), l(v_1 \rightarrow v_4) = 4\rangle, \langle v_4 = (4, 8), d_3 = (9, 8), l(v_4 \rightarrow d_3) = 5\rangle\} = \{o_3 \rightarrow v_1 \rightarrow v_4 \rightarrow d_3\}$ is accessed by another external request. Similarly, we can obtain that the query origins and destinations of both q_4 and q_5 are located on $Path(o_3 \rightarrow d_3)$ (see Figure 3). Therefore, besides q_3, q_4 and q_5 also can be answered by the path. As a result, all of them are removed from $\mathcal{Q}$. As $\mathcal{Q}$ is empty now, the algorithm terminates here. Therefore, we can see that only

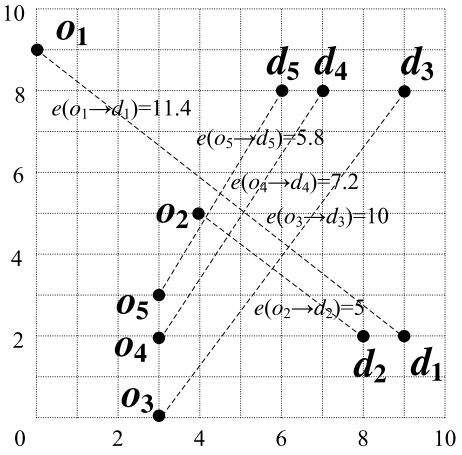


FIGURE 4. An example query set $\mathcal{Q} = \{q_1, q_2, q_3, q_4, q_5\}$.

two external requests are needed instead of five to process all queries in $\mathcal{Q} = \{q_1, q_2, q_3, q_4, q_5\}$ with the path sharing optimization.

V. PATH CACHING

As is presented in Section IV, it is expected that the more path queries in $\mathcal{Q}$ the greater power that the path sharing optimization has. However, in the real world, the number of path queries arriving concurrently or within a short time period at an LBS provider may be small, especially for small and start-up LBS providers, which makes the path sharing optimization ineffective. In this section, we will propose another optimization, namely path caching, which is an extension of path sharing. In the path caching optimization, the selected paths are cached at an LBS provider, based on which each path query is answered. Only when there is no cached path that can answer the query, the LBS provider issues an external request to retrieve the path information for the query. Based on path caching, the LBS provider can effectively process path queries when the number of arrived queries is small.

A. PRELIMINARIES

In this section, we give the preliminary knowledge of path caching. A path cache and its size are defined as follows:

Definition 4 (Path Cache and Cache Size): A path cache $\mathcal{C}$ contains a collection of shortest paths. The cache size of $\mathcal{C}$ (denoted as $|\mathcal{C}|$) is measured by the total number of nodes of the cached paths in $\mathcal{C}$, and $|\mathcal{C}|$ should not be larger than the system's maximum cache capacity Ψ, i.e.,

$$|\mathcal{C}| \leq \Psi.$$

As the shortest paths exhibit the optimal sub-path property (see Lemma 1), the shortest path(s) in the cache can be taken as an answer for any query as long as both its origin and destination lie on the path(s), i.e., a cache hit happens (see Definition 5).

Definition 5 (Cache Hit and Cache Miss): Given a path cache $\mathcal{C}$ and a path query $q_j = (o_j, d_j)$, a cache hit means there is at least one path in the cache that contains the query origin o_j and destination d_j at the same time, i.e.,

$$\exists\, Path(o_i \to d_i) \in \mathcal{C},\quad o_j \in Path(o_i \to d_i) \wedge d_j \in Path(o_i \to d_i);$$

otherwise, a cache miss happens.

Since the system's maximum cache capacity Ψ is limited, the path with greater sharing ability but less nodes is much more preferred to be put into the cache, than the ones with smaller sharing ability but more nodes. Therefore, we define the sharing ability per node of a path for cache construction in Definition 6.

Definition 6 (Sharing Ability Per Node): Let $\mathcal{Q}$ be a set of path queries, and $Path(o_i \to d_i)$ be the shortest path from origin o_i to destination d_i. The sharing ability per node *of $Path(o_i \to d_i)$ with respect to $\mathcal{Q}$ (denoted as $\overline{SA}(Path(o_i \to d_i), \mathcal{Q})$) is defined as:*

$$\overline{SA}(Path(o_i \to d_i), \mathcal{Q}) = \frac{SA(Path(o_i \to d_i), \mathcal{Q})}{|Path(o_i \to d_i)|},$$

where $SA(Path(o_i \to d_i), \mathcal{Q})$ and $|Path(o_i \to d_i)|$ stands for the sharing ability of $Path(o_i \to d_i)$ with respect to $\mathcal{Q}$, and the number of nodes in $Path(o_i \to d_i)$, respectively.

Take $\mathcal{Q} = \{q_1, q_2, q_3, q_4, q_5\}$ as an example, since two queries in $\mathcal{Q}$ (i.e., q_1 and q_2) can share the path information of $Path(o_1 \to d_1)$, as is shown in Figure 3, the sharing ability of $Path(o_1 \to d_1)$ with respect to $\mathcal{Q}$ is 2, i.e., $SA(Path(o_1 \to d_1), \mathcal{Q}) = 2$. As the number of nodes in $Path(o_1 \to d_1)$ is 5, i.e., $|Path(o_1 \to d_1)| = 5$, it is easy to compute that the sharing ability per node of $Path(o_1 \to d_1)$ with respect to $\mathcal{Q}$ is $\frac{2}{5}$, i.e., $\overline{SA}(Path(o_1 \to d_1), \mathcal{Q}) = \frac{2}{5} = 0.4$. In the same way, we can compute $\overline{SA}(Path(o_3 \to d_3), \mathcal{Q}) = \frac{3}{4} = 0.75$. Therefore, $Path(o_3 \to d_3)$ is more preferred by the cache than $Path(o_1 \to d_1)$, as it can be shared by more queries with less space.

B. CACHE CONSTRUCTION

Like most caching systems [5], [6], historical queries can reflect the query situation in the future. In this section, our plan is to construct a path cache based on historical queries and their paths, which can be collected by the LBS provider over the time.

Given the system's maximum cache capacity Ψ, a set of historical path queries $\mathcal{Q}_h = \{q_1, q_2, \ldots, q_m\}$ and their paths, our goal is to build an optimal path cache, in which the paths are not sub-paths of each other, yet can be maximally shared by queries in $\mathcal{Q}_h$, i.e., the overall sharing ability of the cached paths with respect to $\mathcal{Q}_h$ is the largest. Therefore, the goal can be formulated as follows:

$$\textbf{maximize} \sum_{i=1}^{m} SA(Path(o_i \to d_i), \mathcal{Q}_h) x_i$$

$$\textbf{subject to} \sum_{i=1}^{m} |Path(o_i \to d_i)| < \Psi,$$

and $x_i \in \{0, 1\}$,

and $Path(o_i \to d_i)$ is not a sub-path.

This is essentially the same as the well-known 0-1 knapsack problem [23] which is NP-complete. There is no efficient way to find the optimal solution in polynomial-time.

Here, we propose a greedy algorithm to construct a near optimal path cache, as is shown in Algorithm 2 (denoted as PCCA). Based on Definition 6, PCCA first computes the sharing ability per node of each historical path with respect to $\mathcal{Q}_h$ (Lines 4 to 16 of Algorithm 2). As the path with the longest travel distance is impossible to be a sub-path of others, the sharing ability per node is computed based on the travel distance (Line 5). If a path is a sub-path, its sharing ability (as well as the sharing ability per node) is set to zero (Line 11). After finishing the above computation, PCCA greedily constructs the path cache by continuously including the path, which has the largest sharing ability per node (the value should be larger than zero) and has not been put into the cache before, into the cache until it reaches the system's maximum cache capacity Ψ (Lines 17 to 20). Finally, the filled cache is returned.

Algorithm 2 Path Cache Construction Algorithm (PCCA)

1: **input:** the set of historical queries $\mathcal{Q}_h = \{q_1, q_2, \ldots, q_m\}$ and their paths, the maximum cache capacity Ψ
2: **output:** a path cache $\mathcal{C}$
3: $\mathcal{C} \leftarrow \emptyset$;
4: **while** $\mathcal{Q}_h$ is not empty **do**
5: $Path(o_s \rightarrow d_s) \leftarrow$ the path with the longest travel distance, and $q_s \in \mathcal{Q}_h$;
6: $SA(Path(o_s \rightarrow d_s), \mathcal{Q}_h) \leftarrow 1$;
7: remove q_s from $\mathcal{Q}_h$;
8: **for** each query $q_i = (o_i, d_i)$ in $\mathcal{Q}_h$ **do**
9: **if** both o_i and d_i locate on $Path(o_s \rightarrow d_s)$ **then**
10: remove q_i from $\mathcal{Q}_h$;
11: set $SA(Path(o_i \rightarrow d_i), \mathcal{Q}_h)$ to 0;
12: increase $SA(Path(o_s \rightarrow d_s), \mathcal{Q}_h)$ by 1;
13: **end if**
14: **end for**
15: calculate $\frac{SA(Path(o_s \rightarrow d_s), \mathcal{Q}_h)}{|Path(o_s \rightarrow d_s)|}$
16: **end while**
17: **while** $|\mathcal{C}| \leq \Psi$ **do**
18: $Path(o_s \rightarrow d_s) \leftarrow$ the path with the largest sharing ability per node and its value is larger than 0, and $Path(o_s \rightarrow d_s) \notin \mathcal{C}$;
19: insert $Path(o_s \rightarrow d_s)$ into $\mathcal{C}$;
20: **end while**
21: return $\mathcal{C}$;

For example, suppose that the system's maximum cache capacity $\Psi = 10$, the historical query set $\mathcal{Q}_h = \{q_1, q_2, q_3, q_4, q_5\}$, and the path information of these queries are shown in Figure 3, it is easy to calculate that $\overline{SA}(Path(o_1 \rightarrow d_1), \mathcal{Q}_h) = 0.4$ (see Section V-A). As $Path(o_2 \rightarrow d_2)$ is a sub-path of $Path(o_1 \rightarrow d_1)$, we have $\overline{SA}(Path(o_2 \rightarrow d_2), \mathcal{Q}_h) = 0$. Similarly, we can compute that $\overline{SA}(Path(o_3 \rightarrow d_3), \mathcal{Q}_h) = 0.75$, $\overline{SA}(Path(o_4 \rightarrow d_4), \mathcal{Q}_h) = 0$ and $\overline{SA}(Path(o_5 \rightarrow d_5), \mathcal{Q}_h) = 0$. As $Path(o_3 \rightarrow d_3)$ owns the largest sharing ability per node, it is the first one to be selected into the cache, i.e., $\mathcal{C} = \{Path(o_3 \rightarrow d_3)\}$. Since $|\mathcal{C}| = |Path(o_3 \rightarrow d_3)| = 4 < \Psi = 10$, PCCA continuously selects the next one with the largest sharing ability per node from the remaining historical paths, i.e., $Path(o_1 \rightarrow d_1)$, and puts it into the cache. Then, we have $\mathcal{C} = \{Path(o_3 \rightarrow d_3), Path(o_1 \rightarrow d_1)\}$. Currently, $|\mathcal{C}| = 9$. Because all paths (except for sub-paths) have been put into the cache, the algorithm stops here and returns $\mathcal{C} = \{Path(o_3 \rightarrow d_3), Path(o_1 \rightarrow d_1)\}$.

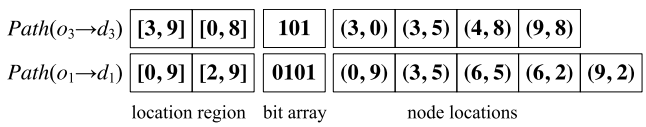


FIGURE 5. The cache structure of $\mathcal{C} = \{Path(o_3 \rightarrow d_3), Path(o_1 \rightarrow d_1)\}$.

C. CACHE STRUCTURE

Upon receiving a path query $q_j = (o_j, d_j)$, the cache system performs a lookup operation by checking each path in $\mathcal{C}$ until a path can answer the query (i.e., cache hit) or all paths have been checked (i.e., cache miss). To support efficient cache lookup, for each path $Path(o_i \rightarrow d_i)$ in $\mathcal{C}$, we also:

1) Store the location region of the path. By this way, the system can quickly filter the locations beyond the region, which are impossible to be located on the path. In other words, the path can not answer the query, if its origin or destination is not in the path location region.
2) Pre-compute and record the straight line situation of each path step to avoid repeated calculation during cache lookup. To save cache space, bit arrays are utilized for recording.

Take $Path(o_1 \rightarrow d_1) = \{o_1 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow d_1\}$ as an example, based on the location of each node in the path, we can calculate that the horizontal range of the path is [0, 9] and the vertical range of the path is [2, 9] (see the red dashed box in Figure 6). In Section IV-B, we have computed that the first, second, third and fourth steps of the path are non-straight, straight, non-straight and straight lines, respectively. To save cache space, a bit array with the value of 0101 is used to record this, where a bit with the value of 1 represents a straight line and 0 represents a non-straight line.

In the same way, we can compute that the horizontal range, the vertical range and the bit array of $Path(o_3 \rightarrow d_3)$ are [3, 9], [0, 8], and 101, respectively. Hence, we can have the cache structure of $\mathcal{C} = \{Path(o_3 \rightarrow d_3), Path(o_1 \rightarrow d_1)\}$, as is shown in Figure 5.

D. PATH CACHING ALGORITHM

Given a path cache $\mathcal{C}$ that is constructed by PCCA, and a set of path queries $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$ arriving at an LBS provider concurrently or within a short time period, the proposed path caching algorithm (i.e., Algorithm 3, denoted as PCA) processes each query in $\mathcal{Q}$ through a cache lookup

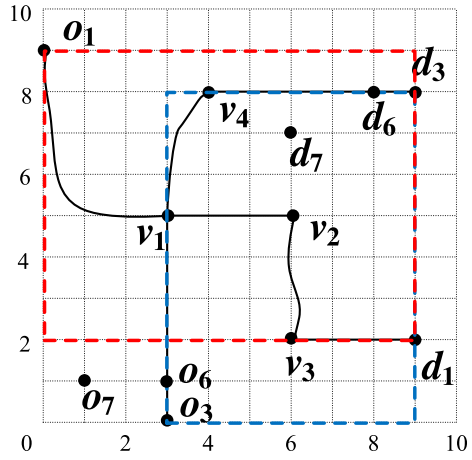


FIGURE 6. The location region of paths in $\mathcal{C} = \{Path(o_3 \rightarrow d_3), Path(o_1 \rightarrow d_1)\}$ and the query set $\mathcal{Q} = \{q_6, q_7\}$.

Algorithm 3 Path Caching Algorithm (PCA)

```
1:  input: the query set Q, the path cache C
2:  for each query q_j = (o_j, d_j) in Q do
3:    for each path Path(o_i → d_i) in C do
4:      if both o_j and d_j locate on Path(o_i → d_i) then
5:        use Path(o_i → d_i) to answer q_j;
6:        remove q_j from Q;
7:        break;
8:      end if
9:    end for
10: end for
11: process Q by PSA (Algorithm 1);
```

first (Lines 2 to 10). If there is a cache hit, i.e., there is a path in the cache that can be taken as the query answer, the LBS provider can reply the query directly and does not need to issue an external request any more. For the queries that can not be answered by the cache, the LBS provider processes them in the same way with PSA (Line 11).

Take Figure 6 as an example, the path cache and the query set are $\mathcal{C} = \{Path(o_3 \rightarrow d_3), Path(o_1 \rightarrow d_1)\}$ and $\mathcal{Q} = \{q_6, q_7\}$, respectively. The cache structure of $\mathcal{C}$ is shown in Figure 5.

For query $q_6 = (o_6, d_6)$, as its origin o_6 is not in the location region of $Path(o_1 \rightarrow d_1)$ (see the red dashed box in Figure 6), it is impossible that $Path(o_1 \rightarrow d_1)$ can answer q_6. While both o_6 and d_6 are in the location region of $Path(o_3 \rightarrow d_3)$ (see the blue dashed box in Figure 6), we need to continuously check whether or not both o_6 and d_6 lie on the path. Based on the bit array of $Path(o_3 \rightarrow d_3)$ (see Figure 5), the first and third steps of the path (i.e., $o_3 \rightarrow v_1$ and $v_4 \rightarrow d_3$) are straight lines, so we only need to decide whether or not o_6 and d_6 are located on these two steps. Since $e(o_3 \rightarrow v_1) = e(o_3 \rightarrow o_6) + e(o_6 \rightarrow v_1)$, we have o_6 lies on path step $o_3 \rightarrow v_1$. Similarly, we have d_6 lies on path step $v_4 \rightarrow d_3$. Hence, $Path(o_3 \rightarrow d_3)$ can be used to answer q_6 since both o_6 and d_6 are on the path, i.e., a cache hit happens.

For query $q_7 = (o_7, d_7)$, as o_7 is not in the location region of any cached path (see Figure 6), we can learn that there is no path in the cache that can answer the query. Then, PCA processes it in the same with PSA, i.e., issuing an external request to retrieve the path information for the query.

VI. PERFORMANCE EVALUATION

A. EXPERIMENTAL SETTING

To the best of our knowledge, there is no other existing algorithm that focuses on path query processing through cloud-based mapping services, when the underlying road network topology is unknown to the LBS provider. Therefore, we only evaluate the performance of the two proposed algorithms, i.e., PSA (Algorithm 1) and PCA (Algorithm 3). The performance metrics include:

1) The average number of external requests per path query, which are issued by the LBS provider to the cloud-based mapping service;
2) The average query response time that the LBS provider answers path queries;
3) The cache hit ratio of PCA.

The response time of a query is the difference between from the time when the path query is received by the LBS provider to the time when the answer is returned to the query user; thus, it includes the local CPU processing time, the communication time between the LBS provider and the cloud-based mapping service, and the remote processing time at the cloud-based mapping service provider.

In our experiments, we generated 18k shortest path queries from the GeoLife trajectory dataset [24], which were collected from 182 users in a period of over three years by Microsoft Research Asia, by extracting the start and end locations of each trajectory as the query origin and destination. Then, we randomly selected 13k queries as the historical queries and constructed the path cache based on the path information of these queries by PCCA (i.e., Algorithm 2). The rest 5k path queries were utilized to evaluate the performance of our proposed algorithms. Bing Maps was taken as the cloud-based mapping service in our experiments. Unless mentioned otherwise, the number of path queries arriving at the LBS provider concurrently or within a short time period was 300, and these queries were randomly selected from 5k path queries. The default cache size was 30k nodes, i.e., nearly 120kB if four bytes a node. A large-scale cache can be used if the LBS provider has a huge number of historical paths.

B. EXPERIMENTAL RESULTS

We evaluate the scalability, efficiency and applicability of PSA and PCA with respect to the number of path queries and cache size.

1) EFFECTS OF THE NUMBER OF PATH QUERIES

In this experiment, we evaluate the performance of PSA and PCA with respect to the number of path queries arriving

concurrently or within a short time period from 50 to 500, as is shown in Figure 7.

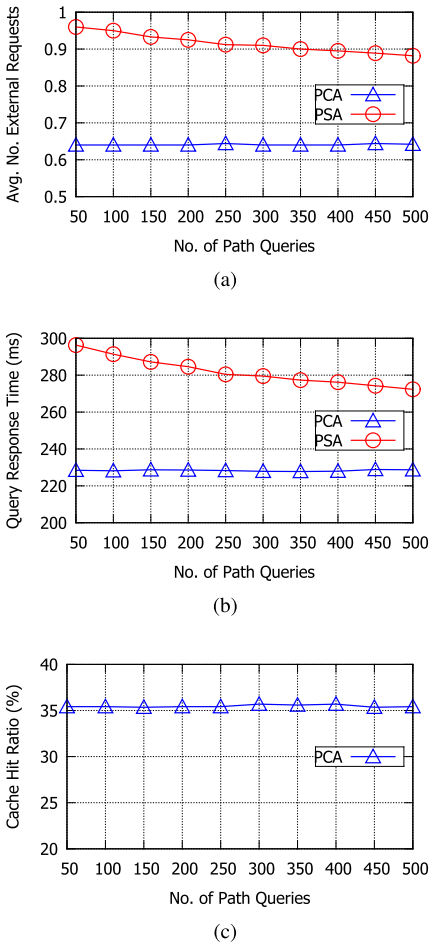


FIGURE 7. Effects of the number of path queries. (a) Average number of external requests. (b) Average query response time. (c) Cache hit ratio.

Without any optimization, one path query needs one external request, and the average response time of external requests from Bing Maps is 304 ms according to our experiments. By PSA, we can see that nearly 8.4% external requests are reduced on average because of the path sharing optimization (Figure 7a). It is expected that the more queries arriving concurrently or within a short time period the greater path sharing power PSA has, so the number of external request and the response time per path query gradually reduce with the increase of the number of path queries (see PSA in Figure 7a and Figure 7b).

With the help of the path caching optimization, 35.8% external requests are reduced on average (see PCA in Figure 7a). This is because PCA caches many selected historical paths and uses them to answer path queries, while PSA dose not. Since the cache size is fixed, i.e., 30k nodes, the cache hit ratio remains unchanged with the increase of the number of path queries, i.e., 35.4% on average (see Figure 7c). Consequently, the average number of external requests and the average response time of a path query by PCA almost remain the same, as are shown in Figure 7a and Figure 7b, respectively.

2) EFFECTS OF THE CACHE SIZE

Figure 8 shows the performance of PSA and PCA with respect to the cache size from 10,000 to 50,000 nodes. When the cache size gets larger, i.e., more paths can be cached, it is expected that the cache hit ratio of PCA increases, as is shown in Figure 8c. As a consequence, the average number of external requests per path query decreases, as is shown in Figure 8a.

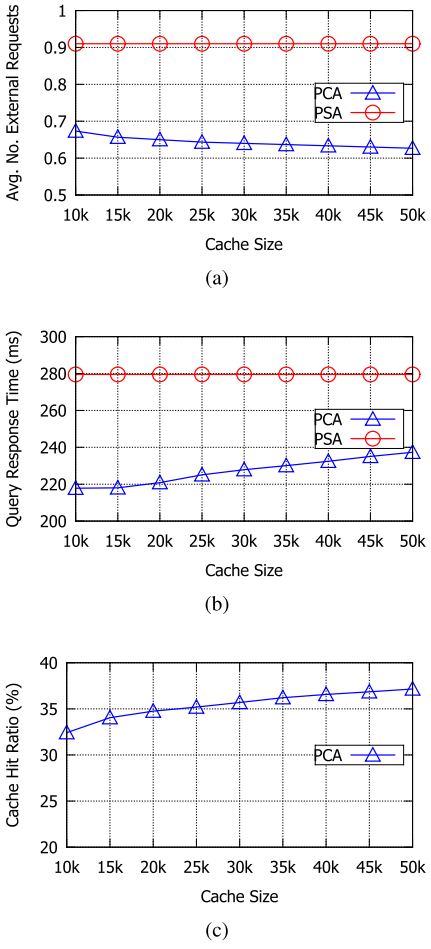


FIGURE 8. Effects of the cache size. (a) Average number of external requests. (b) Average query response time. (c) Cache hit ratio.

However, the average query response time gets longer along with the increase of cache size. The reason for this is that the query response time for a path query consists of the cache lookup time (i.e., the local CPU processing time), and the request response time from the cloud-based mapping service (i.e., the communication time between the LBS provider and the cloud-based mapping service and the remote processing time at the cloud-based mapping service provider). When the cache size gets larger, PCA needs more time to search the cache, i.e., the cache lookup time gets longer; hence, the average query response time becomes

longer. Despite this, the average query response time of PCA is still only 81% of PSA. Since the path cache optimization is not employed in PSA, the performance of PSA is not affected by different cache sizes, as shown in Figure 8a and Figure 8b.

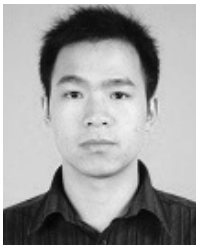
This set of experiments not only demonstrates that a large cache size can effectively reduce the number of external requests, but also shows that a large cache size may induce the long query response time due to the costly cache lookup. Therefore, an efficient cache structure plays an important role in the cache system, as it can not only save cache space but also improve the cache lookup efficiency.

VII. CONCLUSION

It is challenging to process shortest path queries when the map data of the underlying road network is not given/available to an LBS provider. To address this problem, we have presented a system architecture, in which an LBS provider retrieves path information from cloud-based mapping services like Google Maps, Bing Maps and Baidu Maps, by issuing external requests. Since that accessing data through external requests takes much more time than accessing local data, we have proposed two optimization techniques in this paper, namely path sharing and path caching, to reduce the number of external requests and the user query response time. For path sharing optimization, we have proposed an effective method to compute whether or not a query origin/destination lies on a path without knowing the underlying road network topology, and then given an algorithm of path sharing based on the Euclidean distance between each query's origin and destination. For path caching optimization, we have formulated the problem of constructing path cache into a knapsack problem and designed a greedy algorithm to solve it; we also have devised an effective cache structure to save space and support efficient cache lookup. In the end, we have conducted extensive experiments on real datasets by taking Bing Maps as the cloud-based mapping service. Experimental results show that more than 35% external requests can be reduced by our proposed techniques.

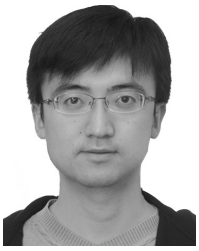
REFERENCES

[1] S. Ilarri, E. Mena, and A. Illarramendi, "Location-dependent query processing: Where we are and where we are heading," *ACM Comput. Surv.*, vol. 42, no. 3, p. 12:1–12:73, 2010.
[2] D. Zhang, C.-Y. Chow, Q. Li, X. Zhang, and Y. Xu, "Efficient evaluation of k-NN queries using spatial mashups," in *Proc. SSTD*, 2011, pp. 348–366.
[3] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa, "Preference query evaluation over expensive attributes," in *Proc. CIKM*, 2010, pp. 319–328.
[4] *Google Maps APIs Terms of Service*, accessed on Jan. 23, 2017. [Online]. Available: https://developers.google.com/maps/terms?hl=en
[5] J. R. Thomsen, M. L. Yiu, and C. S. Jensen, "Effective caching of shortest paths for location-based services," in *Proc. SIGMOD*, 2012, pp. 319–328.
[6] R. Ozcan, I. S. Altingovde, and Ö. Ulusoy, "Static query result caching revisited," in *Proc. 17th Int. Conf. World Wide Web*, 2008, pp. 1169–1170
[7] C. Sommer, "Shortest-path queries in static networks," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 45:1–45:31, 2014.
[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
[9] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, no. 1, pp. 87–90, 1958.
[10] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968.
[11] U. Meyer and P. Sanders, "Delta-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.
[12] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "PHAST: Hardware-accelerated shortest path trees," *J. Parallel Distrib. Comput.*, vol. 73, no. 7, pp. 940–952, 2013.
[13] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. SIGMOD*, 2010, pp. 135–146.
[14] H. Bast, S. Funke, P. Sanders, and D. Schultes, "Fast routing in road networks with transit nodes," *Science*, vol. 316, no. 5824, p. 566, 2007.
[15] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: Towards bridging theory and practice," in *Proc. SIGMOD*, 2013, pp. 857–868.
[16] V. Kalavri, T. Simas, and D. Logothetis, "The shortest path is not always a straight line: Leveraging semi-metricity in graph analysis," *VLDB Endowment*, vol. 9, no. 9, pp. 672–683, 2016.
[17] D. Zhang, C.-Y. Chow, Q. Li, X. Zhang, and Y. Xu, "SMashQ: Spatial mashup framework for k-NN queries in time-dependent road networks," *Distrib. Parallel Databases*, vol. 31, no. 2, pp. 259–287, 2013.
[18] Y. Li and M. L. Yiu, "Route-saver: Leveraging route APIs for accurate and efficient query processing at location-based services," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 1, pp. 235–249, Jan. 2015.
[19] D. Zhang, C.-Y. Chow, Q. Li, X. Zhang, and Y. Xu, "A spatial mashup service for efficient evaluation of concurrent k-NN queries," *IEEE Trans. Comput.*, vol. 65, no. 8, pp. 2428–2442, Oct. 2016.
[20] D. Zhang, C.-Y. Chow, Q. Li, and A. Liu, "Efficient evaluation of shortest travel-time path queries in road networks by optimizing waypoints in route requests through spatial mashups," in *Proc. APWeb*, Sep. 2016, pp. 1–12.
[21] D. Zhang, C.-Y. Chow, A. Liu, X. Zhang, Q. Ding, and Q. Li, "Efficient evaluation of shortest travel-time path queries through spatial mashups," in *GeoInformatica*. New York, NY, USA: Springer, 2017, pp. 1–26.
[22] Y. Zhang, Y.-L. Hsueh, W.-C. Lee, and Y.-H. Jhang, "Efficient cache-supported path planning on roads," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 4, pp. 951–964, Apr. 2016.
[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[24] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining interesting locations and travel sequences from GPS trajectories," in *Proc. 18th Int. Conf. World Wide Web*, 2009, pp. 791–800.

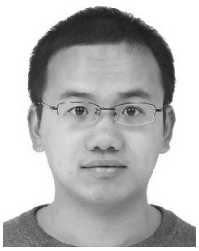


DETIAN ZHANG received the joint Ph.D. degrees in computer science from the University of Science and Technology of China and the City University of Hong Kong in 2014. He is currently a Lecturer with the School of Digital Media, Jiangnan University, Wuxi, China. His research interests include location-based services, spatio-temporal databases, and wireless networks.

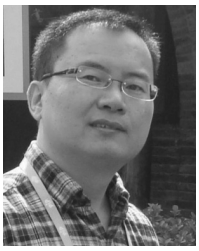
YUAN LIU received the B.S. degree in electrical engineering from Fudan University, Shanghai, China, in 1987, and the M.S. degree in control science and engineering from Jiangnan University, Wuxi, China, in 1998. He is currently a Full Professor with the School of Digital Media, Jiangnan University. His research interests include multimedia communications, network protocols, and information security.



AN LIU received the Ph.D. degrees in computer science from the City University of Hong Kong and the University of Science and Technology of China in 2009. He is currently an Associate Professor with the Department of Computer Science and Technology, Soochow University. His research interests include spatial databases, crowdsourcing, data security and privacy, and cloud/service computing. He has authored or co-authored more than 80 papers in referred journals and conferences, including IEEE TKDE, IEEE TSC, GeoInformatica, KAIS, ICDE, and WWW. He served as the Workshop Co-Chair of WISE 2017 and DASFAA 2015. He is on the Reviewer Board of several top journals, such as IEEE TKDE, IEEE TSC, IEEE TII, IEEE TCC, ACM TOIT, JSS, DKE, FGCS, WWWJ, and JCST.



XUDONG MAO received the B.Sc. degree in information security from Nankai University, Tianjin, China, and the M.Phil. degree from the City University of Hong Kong, Hong Kong. He is currently a Hong Kong Ph.D. Fellowship Researcher with the Department of Computer Sciences, City University of Hong Kong. Before joining CityU HK, he was a Research Scientist with Alibaba (China), one of largest e-commerce companies. His research interests include deep generative models, object detection, and deep learning.



QING LI (SM'07) received the B.Eng. degree from Hunan University, Changsha, China, and the M.Sc. and Ph.D. degrees from the University of Southern California, Los Angeles, all in computer science. He is currently a Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His current research interests include dynamic object modeling, multimedia and mobile information retrieval and management, distributed databases and data warehousing/mining, and workflow management and Web services.