Refreshment of the Shortest Path Cache with Change of Single Edge

Xiaohua Li[1], Tao Qiu[1], Ning Wang[1,], Xiaochun Yang[1], Bin Wang[1], Ge Yu[1]

[a]*School of Computer Science and Engineering, Northeastern University, Liaoning, China*
[b]*Department of Information Management, School of Management, Shanghai University, Shanghai, China*

Abstract

The problem of caching shortest paths has been widely studied. All of existing methods that address this problem assume that the condition of road networks does not change with time. In this paper, we study how to refresh a cache when one edge of the underlying road network (graph) changes. A bitmap-based cache structure is proposed to store and give access to shortest paths. In the following, algorithms are developed to detect shortest paths that are affected by the change of edge. After detecting affected paths, several heuristic-based refreshment strategies are proposed to update the cache. We have conducted a series of experiments to compare the performance of proposed strategies. It shows that replacing affected shortest paths with new paths whose benefit values are the largest should be applied in the shortest path caching applications such as navigation and map services.

Keywords: `shortest path caching, cache refreshment strategy, change of road network, affected shortest paths`

1. Introduction

Shortest path query systems are now widely seen in our daily life [**? ? ? ? ?**]. People use map service for shortest paths to restaurants, parking lots, cinemas, playgrounds, shopping malls and so on. With the uberisation, drivers of private cars also frequently use map service for various destinations. When a user has a shortest path

*Corresponding author.

query, a cache in a local server is accessed; the result, if existing, will be directly returned to the user. If the cache does not have an off-the-shelf result to the query, the system accesses a global server which runs a shortest path algorithm for the query. The latter situation undoubtedly costs communication and computational time [**? ? ? ? ?**]. The cached contents are, therefore, very critical for the efficiency of the whole caching system.

In the shortest path caching problem, all of existing works discuss the cache initialization problem which addresses how to load an empty cache when a road network and a query log are known, so that the cache works in a high level of efficiency in the future. Researchers usually load paths with the highest query frequencies into the cache, or those paths that contain the most numbers of nodes [**? ?**]. However, existing works do not discuss how to refresh the cache when the road condition or characteristic of queries change. Actually, the road condition cannot remain the same all the time; on contrast, it changes frequently. For example, traffic congestion or a traffic accident affects the traffic capability of a road. If the cached paths are not adjusted after the condition of a road network changes, some paths in the cache may become invalid or the utilization of the cache decreases. How to maintain a cache after changes of a road network is seldom discussed in the literature. Apparently, a straightforward way to refresh a cache is to empty the cache and conduct an initialization process once more. Such a method is naive, because computing and evaluating all the shortest paths once again are time-consuming.

In this paper, we discuss how to refresh a cache when the weight of one single edge changes. Assuming that only one edge changes its weight is reasonable. On one hand, studying one single edge is a good point to set about. Some generic conclusions can be drawn from the single-edge scenario and be applied to multi-edge scenarios. On the other hand, single-edge does have real applications. For example, the traffic congestion may happen in main streets and these main streets are far away from each other and unrelated; the congestion in every main street can be considered separately as a single-edge scenario.

We first develop a cache structure composed of a path array and a bitmap. We then introduce the concept of "affected paths" and develop algorithms to detect affected

shortest paths due to the change of edge weight. Next, we design four strategies to refresh the cache and compare their performance. Our work contributes to the literature in two aspects. First, to the best of our knowledge, our work is the first one to discuss the shortest path caching problem in a changing graph. This problem is undoubtedly more close to real applications. Second, four cache refreshment strategies are proposed to update the cache. It is noteworthy that this work is an extension of [**?**] which is published in a conference. This paper extends the initial work, by 1) adopting a bitmap index to efficiently answer queries of shortest paths; 2) introducing several optimization techniques to further improve the efficiency of detecting affected shortest paths; 3) performing more comprehensive experiments on real data sets, including Aalborg, Beijing and Singapore road networks. Especially, the data set of Singapore is newly added and is not used in the conference version.

The rest of the paper is organized as follows. In Section **??**, we review works related to the shortest path caching problem. In Section **??**, we formally define the problem and design a bitmap-based cache structure to store shortest paths. Then we explore the properties of changed graphs and present our algorithm of detecting affected shortest paths in Section **??**. In the following, four cache refreshment strategies are proposed in Section **??**. In Section **??**, we conduct experimental comparisons of the proposed methods on real data sets. Finally, we give closing remarks in Section **??**.

2. Literature Review

Our problem is related to works of two streams. One stream is to detect which shortest paths are changed after the graph changes. We call this kind of problem as "affected shortest path detection" problem. The other stream addresses the problem of caching shortest paths. In the following, we introduce concrete works in the two streams.

2.1. Affected Shortest Path Detection

The shortest path of a certain node-pair may change after the weights of some edges change in a graph. The affected shortest path detection problem deals with finding node-pairs whose shortest paths are affected and update new shortest paths for

these node-pairs. Based on whether the network changes in a regular manner, it has predictable and unpredictable networks.

Predictable Network Models aim at finding the expected fastest (shortest) paths based on the known distribution of predicted networks. In details, [**?**] developed the concept of speed pattern to specify the predicted network costs (weights) at different time, and the fastest paths corresponding to different time can be computed based on the predicted network cost. [**?**] modeled the cost of each edge as a continuous stochastic process, and then they found the expected fastest paths. Similarly, [**?**] used time-delay functions to represent the predicted edge costs.

Unpredictable Network Models assume that the costs of graph change randomly and cannot be depicted by a statistical distribution. Unpredictable network problem has offline and online versions.

For the offline research, it has pre-processing and needs extra space to store auxiliary information. In [**?**], an index named QSI was used to identify the shortest paths affected by the change of network. QSI keeps the subnetworks that are involved in derivation of answering shortest paths. In [**?**], in order to process more than one shortest path query simultaneously, a grid-based index structure was designed to record all the affecting areas. Besides, arc-flags is a typical index-based approach to compute the shortest paths. It has been adopted by [**?**] for a static network. [**? ?**] studied the problem of updating arc-flags in dynamic networks.

For the online research, [**? ? ? ? ?**] studied how to update the shortest paths starting from a specified node. In general, these algorithms preprocess the shortest paths starting from a given source node, and store them in a shortest path tree (SPT). To update the SPT, the basic idea is to utilize the information of the outdated SPT and update only the part of the SPT that is affected by the changed edge. [**?**] studied the problem of continuous evaluating shortest path queries. For several queries, they developed an ellipse bound method (EBM) where each shortest path corresponds to an elliptic geographical area, called affected area, and all the updated edges are considered to affect their corresponding paths.

It can be seen from the above review, offline study of shortest path detection problem uses extra space. For the online problem, some research maintain a novel designed

data structure or they can only detect known paths. In our problem, the space is reserved for the cache; it cannot tell which paths are affected (or no longer the shortest) before the graph changes. Thus, extant algorithms for detecting affected shortest paths cannot be used in our problem directly.

2.2. Caching Problem

Another steam is the caching problem. Caching techniques have been studied extensively in many domains [**?** **?**]. The main focus is to decide the contents that are cached.

Web Search Caching. In [**?**], caching strategies are classified into static caching and dynamic caching. Static caching does not change the contents of cache once the cache is loaded based on the historical log [**?** **?** **?**]. The static caching usually updates periodically based on the latest query log. Dynamic caching focuses on suiting for new arrival queries and weeding out old contents [**?** **?** **?**].

Shortest Path Caching. [**?** **?** **?**] have studied the caching of shortest paths. In details, [**?**] is the pioneer work on caching of shortest paths. They proposed a cost-oriented model to quantify the benefit of loading paths to a cache. Based on this model, a static caching strategy is proposed to initialize the cache. [**?**] also focuses on a static strategy. In order to store as many paths as possible, an improved cost-oriented model was proposed to measure paths appeared in the query log. [**?**] proposed the notion of generically concise shortest paths that enables a trade-off between the size of paths and the number of queries answered by paths. They considered both the static and dynamic caching strategies for selecting generically concise paths. For the dynamic strategy, they update the cache in a least-recently-used policy when missing targets occur.

In applications of caching problem such as web search caching, queries change with time while the data (answers) will change in no way. The inefficiency of cache is fully resulted from the change of queries. Hence, “dynamic” caching strategies are motivated by the change of queries. However, in the shortest path caching problem, the change of graph (data) may also lead to the incorrectness of the answer and inefficiency of the cache. In this paper, we investigate the problem of shortest path caching when

the graph changes, and we extend the concept of "dynamic" caching to the change of data.

3. Problem Description and Cache Structure

3.1. Notation and Problem Description

The graph $G = (V, E)$ studied in this paper is undirected; $V = \{v_1, v_2, ..., v_n\}$ is the set of nodes and $E = \{e_1, e_2, ..., e_k\}$ is the set of edges. The weight of an edge e is denoted by w_e. $P_{a,b}$ represents the shortest path between v_a and v_b. If there exists more than one shortest paths between v_a and v_b, $P_{a,b}$ represents one of them. $Q_{a,b}$ represents a query of the shortest path between v_a and v_b. A cache is denoted by Ω and its capacity is $|\Omega|$. Ψ refers to cached contents. The size of a cache and cached contents are measured by the number of nodes. It is clear that the size of Ψ is always no larger than the cache capacity, i.e., $|\Psi| \leq |\Omega|$ all the time.

The problem is formally defined as follows: given an undirected graph G and a query log L, the query log records the queries of shortest paths of G. A cache Ω at a server caches a part of shortest paths. For missing queries, an API of shortest paths in the server is invoked. Now the cache Ω has been fully loaded and the weight of a single edge e in G changes from w_e to w'_e. Some paths in the cache may not be the shortest paths any more. The objective of our problem is to refresh the content of cache Ω so that the path information is updated and the hit ratio of future queries is as high as possible. The size of cache is assumed to be smaller than the size of shortest paths queried in L.

3.2. Cache Structure

In this paper, we design a cache structure to store and access paths in a cache. In essence, the cache structure needs to satisfy two requirements: fast response and space efficiency. We use a bitmap structure to index shortest paths. The bitmap structure was first developed for database use in the Model 204 product from Computer Corporation of America [**?**]. Now bitmap is commonly used in databases and data warehouses [**?**].

The cache structure proposed includes two parts of information: the path array that stores shortest paths and the bitmap that is used to index paths. A bitmap is a 2-dimensional array. The horizonal and vertical indices are paths and nodes, respectively. If a path contains a certain node, the unit crossed by the path and node is marked with 1; otherwise, it is marked with 0. The storage space of each unit only takes one bit. Here the name, bitmap. The horizontal array indicated by a node v_i is called a node vector $\vec{v_i}$. Take Figure **??** as an example. Three shortest paths $P_{0,5}$, $P_{2,4}$ and $P_{3,6}$ are stored in the form of arrays (Figure **??**), and the bitmap of nodes is shown in Figure **??**.

$P_{0,5}$: v_0-v_1-v_4-v_5
$P_{2,4}$: v_2-v_3-v_4
$P_{3,6}$: v_3-v_4-v_6

(a)

	$P_{0,5}$	$P_{2,4}$	$P_{3,6}$
v_0	1	0	0
v_1	1	0	0
v_2	0	1	0
v_3	0	1	1
v_4	1	1	1
v_5	1	0	0
v_6	0	0	1

(b)

Figure 1: An example of cache structure.

Given a query $Q_{s,e}$, we need to look up shortest paths that contain v_s and v_e. The caching system looks up the bitmap and finds the node vectors $\vec{v_s}$ and $\vec{v_e}$. An AND operate is conducted to acquire $\vec{v_r} = \vec{v_s}$ AND $\vec{v_e}$. In $\vec{v_r}$, the bits whose value are 1 correspond to paths that contain $P_{s,e}$.

Suppose the number of paths in the cache is α and the average number of nodes of each cached path is *avg*. Each node of a path is recorded by an integer in the path array. The total space used by the path array is $32*\alpha*avg$ bits. Suppose the number of unique nodes in the cache is β. Each vector of a node records the node ID (an integer) and whether this node is in paths (bits). The total space used by the bitmap is $(\alpha + 32) * \beta$ bits.

Now we compare the bitmap with the inverted list of [**?**]. The inverted list is composed of vectors. Each vector records a node ID and the path IDs which contain the node. Suppose on average, a node appears in γ paths. Each record of a node needs

space for node ID (an integer) and path IDs (γ integers). The total space used by the inverted list is $\beta(32 + 32\gamma)$. Moreover, $\gamma = \alpha * avg/\beta$. Therefore, when $\beta < 32avg$, the bitmap needs smaller space than the inverted list. According to our experiment, β and $32avg$ have similar scales, even, β is larger than $32avg$. Hence, the space advantage of our bitmap is not obvious. However, in terms of the lookup time, the bitmap only needs an AND operation while the inverted list needs a comparison of two sequences.

From the perspective of implementation, the space of the path array is flexible, since it can be enlarged as paths are added. Both bitmap and inverted list are arrays of vectors, and we have to know the dimension of arrays when initializing. As the cache is used continuously, parameters α, β, and *avg* in the cache can be learnt. For a given cache size, the space of bitmap is reserved, and the left space is for as many paths as possible.

Now we analyze the space complexity of bitmap. For a graph with N nodes, the maximum number of node-pair is $\frac{N*(N-1)}{2}$. For any node-pair, at most one shortest path is loaded into the cache. Hence, the number of shortest paths in the cache is no larger than $\frac{N*(N-1)}{2}$. Moreover, the minimum size of a shortest path is 2, as a shortest path contains at least 2 nodes. Therefore, a cache Ω can store at most $\min\{\frac{|\Omega|}{2}, \frac{N*(N-1)}{2}\}$ shortest paths. Hence the number of columns in a bitmap is $\min\{\frac{|\Omega|}{2}, \frac{N*(N-1)}{2}\}$. The most number of rows is the number of nodes N. The number of bits needed in a bitmap is therefore $\min\{\frac{|\Omega|}{2}, \frac{N*(N-1)}{2}\} * N$; and the number of bytes needed is $\lceil \frac{\min\{\frac{|\Omega|}{2}, \frac{N*(N-1)}{2}\}*N}{8} \rceil$. The space complexity of bitmap is $\mathcal{O}(\min\{|\Omega| * N, N^3\})$.

4. Affected Shortest Path Detection

Before we refresh the cache, it is better for us to know which shortest paths are no longer shortest paths due to the change of an edge.

A shortest path $P_{i,j} = \langle v_i, ..., v_j \rangle$ is **not affected** if the new shortest path $P'_{i,j} = \langle v_m, ..., v_n \rangle$ due to the change of an edge e is totally the same with $P_{i,j}$ in the node sequence. Otherwise, $P_{i,j}$ is affected. Hereafter, "=" is used if two paths have the same sequence. e refers to the edge whose weight w_e changes. P and P' denote the shortest

paths before and after w_e changes, respectively.

It is noteworthy that even though e is a part of $P_{i,j}$, if $P_{i,j} = P'_{i,j}$, $P_{i,j}$ is not affected. That is, the weight of a path has nothing to do with its being affected or not. The only thing that counts is the sequence.

Based on the definition of affected paths, we have the following corollaries.

Corollary 1. *When w_e increases, if $P_{i,j}$ is an affected path, then $P_{i,j}$ must contain e.*

Proof 1. *If $P_{i,j}$ does not contain e, then $P_{i,j}$ is still the shortest path when w_e increases, that is, $P_{i,j}$ is not an affected path. Hence, if $P_{i,j}$ is an affected path, $P_{i,j}$ must contain e*

Compared to the increase of w_e, it is more difficult to detect affected paths when w_e decreases, since paths that do not contain e before w_e decreases can also be affected paths. The following corollary deals with the case that w_e decreases.

Corollary 2. *When w_e decreases, if $P_{i,j}$ is an affected path, then $P'_{i,j}$ must contain e.*

Proof 2. *Assume a shortest path $P_{i,j}$ is an affected path, and $P'_{i,j}$ does not contain edge e, then it has $P_{i,j} = P'_{i,j}$, indicating that $P_{i,j}$ is not an affected path. There is a conflict with the assumption. Therefore, $P'_{i,j}$ must contain edge e.*

Corollary 3. *The endpoints of e are v_a and v_b. v_i and v_j are two nodes where $v_j \neq v_a$, $v_j \neq v_b$. Suppose $P_{i,j}$ does not contain e (if there exist more than one shortest paths between v_i and v_j, we suppose that at least one shortest path does not contain e). For any adjacent node v_{adj} of v_j, if $P_{i,adj}$ is in the form of $P_{i,adj} = \langle v_i, ..., v_j, v_{adj} \rangle$, $P_{i,adj}$ does not contain e either.*

Proof 3. *Since $v_j \neq v_a$, $v_j \neq v_b$, no matter what v_{adj} is, $\langle v_j, v_{adj} \rangle$ can not be e. In addition, sequence $\langle v_i, ..., v_j \rangle$ must be $P_{i,j}$ which does not contain e. Thus, $P_{i,adj}$ does not contain e.*

Since v_a and v_b are symmetric, the following statement still stands: v_i and v_j are two nodes where $v_i \neq v_a$, $v_i \neq v_b$. Suppose at least one path of $P_{i,j}$ does not contain e. For any adjacent node v_{adj} of v_i, if $P_{adj,j}$ is in the form of $P_{adj,j} = \langle v_{adj}, v_i, ..., v_j \rangle$, $P_{adj,j}$ does not contain e either.

4.1. Node Expansion Algorithm

Since it is easy to check affected shortest paths when w_e increases, in the following, we pay our attention to the algorithms of detecting affected paths when w_e decreases. We propose a node expansion algorithm (NEA) in this subsection.

Given the paths in the cache, we want to know which paths are affected and are deleted. For the affected shortest path detection problem, the key problem is to decide which paths are detected. The paths that contain e should be detected while the paths that do not contain e are not necessary to be detected. The graph is our problem is undirected, therefore a path from v_i to v_j is just the path from v_j to v_i. To avoid detecting a path two times, we artificially assign a direction for paths. C_s (C_t) is used to record start nodes (termination nodes) of to-be-detected paths. Every time, one node v_s from C_s and one node v_t from C_t comprise a node-pair, and we detect whether its shortest path is affected. C_s (C_t) is expanded by adding adjacent nodes of v_s (v_t).

For the general purpose of detecting affected paths, all the node-pair will be detected and the detection is time-consuming. Since our ultimate goal is to update the cache, it is not necessary to detect all node-pair. We define that the **node distance** $d(i, j)$ between v_i and v_j represents the minimum number of nodes among all paths from v_i to v_j. When weights of edges in a graph become all-1, $w(i, j)$ is equal to $d(i, j)$. Suppose the maximum node distances of paths in the cache is K, we will ignore the detection of node-pairs whose node distances are longer than K. When a node-pair is detected, we check whether it is in the cache. If all the paths in the cache has been checked, the algorithm terminates.

The pseudo code is illustrated in Algorithm **??**. The endpoints of e are v_a and v_b. From the implementation perspective, both C_s and C_t are queues in a FIFO manner. First, C_s is initialized with the only start node v_a. In the outer loop, start node v_s is the first node popped from C_s. If the new shortest path $P'_{s,b}$ contains e (line **??**), then adjacent nodes of v_s are appended into C_s unless they have been added into C_s ever before or the node distances from v_a exceed K. If $P'_{s,b}$ does not contain e, according to Corollary **??**, the adjacent node v_{adj} of v_s are not necessarily put into C_s since $P'_{adj,b}$ does not contain e.

In the inner loop, v_s is a fixed node. C_t is initialized with v_b. v_t is obtained by

Algorithm 1: NEA

Input: A graph $G(V, E)$, the edge e which decreases its weight and its endpoints v_a and v_b, and a cache Ψ;

Output: The cache after deletion of affected paths

```
1  add v_a to C_s;
2  while C_s is not empty && there are paths in Ψ which are not checked do
3      v_s ← pop the first element from C_s;
4      calculate P'_{s,b};
5      if P'_{s,b} contains e then
6          for each adjacent node v_adj of v_s do
7              if v_adj has not been added into C_s && v_adj ≠ v_b then
8                  append v_adj to C_s;
9          add v_b to C_t;
10         while C_t is not empty do
11             v_t ← pop the first element from C_t;
12             calculate P'_{s,t};
13             if P'_{s,t} contains e then
14                 if d'(s,a) + d'(b,t) < K then
15                     for each adjacent node v_adj of v_t do
16                         if v_adj has not been added into C_t && v_adj ≠ v_a then
17                             append v_adj to C_t;
18                 if P_{s,t} ∈ Ψ then
19                     calculate P_{s,t};
20                     if P_{s,t} ≠ P'_{s,t} then
21                         delete P_{s,t} from Ψ;
22                     else
23                         mark "checked" for P_{s,t};
24             else if P_{s,t} ∈ Ψ then
25                 mark "checked" for P_{s,t};
26 return Ψ;
```

popping a node from C_t. If $P'_{s,t}$ contains e, then adjacent nodes of v_t are appended into C_t unless they have been added into C_t ever before. For any path $P_{s,t} \in \Psi$, the algorithm checks whether $P_{s,t} = P'_{s,t}$, if not, $P_{s,t}$ is deleted from Ψ. If $P_{s,t} \in \Psi$ but $P_{s,t}$ is not affected, the "checked" mark is labeled.

NEA satisfies both soundness and completeness. We first prove that the NEA is sound, i.e. deleted paths are affected paths. According to line **??** of the algorithm, if a path $P_{s,t}$ is deleted from Ψ, it has $P_{s,t} \neq P'_{s,t}$, so $P_{s,t}$ is an affected path.

Now we prove that NEA is complete. Assume that there exists an affected path $P_{s,t}$ that is not deleted from Ψ. Based on Corollary **??**, we know that $P'_{s,t}$ must contain the changed edge e. Hence, v_s (v_t) is direct or indirectly connected with v_a (v_b). According to NEA, v_s (v_t) with node distance smaller than K (K node distance cover all paths in the cache) must have been added into C_s (C_t). Thus, $P_{s,t}$ must have been compared with $P'_{s,t}$. Therefore, NEA can recognize $P_{s,t}$ as affected, and the assumption does not hold. NEA is complete.

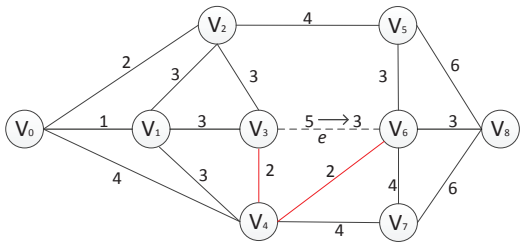


Figure 2: An example of w_e decreases.

Figure **??** shows an example of computing affected paths on the graph when the weight of e ($\langle v_3, v_6 \rangle$) decreases from 5 to 3. Table **??** shows the iteration process.

4.2. Outward Node Expansion Algorithm

In the above subsection, we introduce a basic method NEA to compute affected shortest paths when the weight of a certain edge decreases. The above algorithm mainly has two components: which paths are detected and how to check whether they are affected. In this part, we will present how to improve determining which paths are detected.

Table 1: Iteration example of NEA.

	C_s	C_t	v_s	v_t	Affected paths
1	$\{v3\}$	$\emptyset$	null	null	
2	$\emptyset$	$\emptyset$	$v3$	null	
3	$\emptyset$	$\{v6\}$	$v3$	null	
4	$\{v1, v2, v4\}$	$\emptyset$	$v3$	$v6$	
5	$\{v1, v2, v4\}$	$\{v4, v5, v7, v8\}$	$v3$	$v6$	$P_{3,6}$
6	$\{v1, v2, v4\}$	$\{v5, v7, v8\}$	$v3$	$v4$	$P_{3,6}$
7	$\{v1, v2, v4\}$	$\{v7, v8, v2\}$	$v3$	$v5$	$P_{3,6}, P_{3,5}$
8	$\{v1, v2, v4\}$	$\{v8, v2\}$	$v3$	$v7$	$P_{3,6}, P_{3,5}$
9	$\{v1, v2, v4\}$	$\{v2\}$	$v3$	$v8$	$P_{3,6}, P_{3,5}, P_{3,8}$
10	$\{v1, v2, v4\}$	$\emptyset$	$v3$	$v2$	$P_{3,6}, P_{3,5}, P_{3,8}$
11	$\{v2, v4\}$	$\emptyset$	$v1$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}$
12	$\{v4\}$	$\emptyset$	$v2$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}$
13	$\{v4, v0, v5\}$	$\{v4, v5, v7, v8\}$	$v2$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}$
14	...	...	...	...	...
15	$\{v4, v0, v5\}$	$\emptyset$	$v2$	$v8$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
16	$\{v0, v5\}$	$\emptyset$	$v4$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
17	$\{v5\}$	$\emptyset$	$v0$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
18	$\emptyset$	$\emptyset$	$v5$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$

For ease of illustration, let $S_d(v_a)$ be the set of nodes whose node distance from a is d.

Corollary 4. *Suppose the endpoints of e are v_a and v_b, and v_i is an arbitrary node. If for any node $v_j \in S_d(v_b)$, $P_{i,j}$ does not contain e, then for any node $v_{j+1} \in S_{d+1}(v_b)$, one of the following stands: $S = \langle v_i, ..., v_j, v_{j+1} \rangle$ is a shortest path and it does not contain e; S is not a shortest path.*

Proof 4. *For any node $v_{j+1} \in S_{d+1}(v_b)$, if $P_{i,j+1} = \langle v_i, ..., v_j, v_{j+1} \rangle$, $v_j \in S_d(v_b)$, and $P_{i,j}$ does not contain e, then according to Corollary **??**, $P_{i,j+1}$ does not contain e.*

Recall the algorithm NEA, although start nodes are added to C_s in a FIFO order i.e., nodes are expanded in a breath-first manner, the nodes in C_s are not discriminated in terms of the node distance from v_a. In this subsection, we expand nodes based on their distances from v_a. Thus, the algorithm is called outward node expansion algorithm (ONE). An auxiliary set C'_s is employed to store the nodes that are expanded from the nodes in C_s, therefore, $C_s = S_d(v_a)$ while $C'_s = S_{d+1}(v_a)$.

For a fixed node v_a, if a node v_t satisfies that (1) v_t is connected with v_b; and (2) $P'_{a,t}$ contains e, we add v_t into C_t (line **??**-**??** of Algorithm **??**).

Starting from iteration $d = 1$, we try to reduce the size of C_t. For a node $v_t \in C_t$, if $P'_{s,t}$ does not contain e for any node $v_s \in S_d(v_a)$, then according to Corollary **??**, $P'_{s,t}$ does not contain e for any node $v_s \in S_{d+1}(v_a)$. Therefore, v_t is deleted from C_t, reflecting by not being added into C'_t. Hence, ONE saves the time of visiting v_t in the $d + 1$th (and latter iterations) and calculating corresponding shortest paths.

Based on ONE, the iteration process of example Figure **??** is displayed in Table **??**.

4.3. Inward Node Expansion Algorithm

In this section, we present another two techniques to improve the efficiency of affected shortest path detection.

Corollary 5. *Suppose the end nodes of e are v_a and v_b. When w_e decreases, if $P_{a,b}$ is an affected path, then any shortest path $P'_{s,t}$ which contains e is an affected path, i.e., $P'_{s,t} \neq P_{s,t}$.*

Algorithm 2: COMPRISING C_t

Input: A graph $G(V, E)$, the edge e which decreases its weight and its endpoints v_a and v_b, and a cache Ψ;

Output: The set of termination nodes C_t;

```
1  add v_b to C_t and C'_t;
2  while C_t is not empty && there are paths in Ψ which are not checked do
3      v_t ← pop the first element from C_t;
4      calculate P'_{a,t};
5      if P'_{a,t} contains e then
6          if d(b,t) < K then
7              for each adjacent node v_adj of v_t do
8                  if v_adj has not been added into C_t && v_adj ≠ v_a then
9                      append v_adj to C_t;
10                     append v_adj to C'_t;
11         if P_{a,t} ∈ Ψ then
12             calculate P_{a,t};
13             if P_{a,t} ≠ P'_{a,t} then
14                 delete P_{a,t} from Ψ;
15             else
16                 mark "checked" for P_{a,b};
17     else if P_{a,b} ∈ Ψ then
18         mark "checked" for P_{a,b};
19 return C_t;
```

Algorithm 3: ONE

Input: A graph $G(V, E)$, the edge e which decreases its weight and its endpoints v_a and v_b, and a cache Ψ;

Output: The cache Ψ after deletion of affected paths;

Comprising C_t; replace C_t with C'_t;
for *each adjacent node* v_{adj} *of* v_a **do**
    add v_{adj} to C_s;
while C_s *is not empty && there are paths in* Ψ *which are not checked* **do**
    $C'_s \leftarrow \emptyset$; $C'_t \leftarrow \emptyset$; `// initialize auxiliary sets`
    for *each node* v_s *in* C_s **do**
        calculate $P'_{s,b}$;
        if $P'_{s,b}$ *contains* e **then**
            if $d'(s, a) < K$ **then**
                for *each adjacent node* v_{adj} *of* v_s **do**
                    if v_{adj} *has not been added into* C_s && $v_{adj} \neq v_b$ **then**
                        add v_{adj} to C'_s; `// store` v_{adj} `for the next round`
        for *each* v_t *in* C_t **do**
            calculate $P'_{s,t}$;
            if $P'_{s,t}$ *contains* e **then**
                add v_t to C'_t; `// store` v_t `for the next round`
                if $P_{s,t} \in \Psi$ **then**
                    calculate $P_{s,t}$;
                    if $P_{s,t} \neq P'_{s,t}$ **then**
                        delete $P_{s,t}$ from Ψ;
                    else
                        mark "checked" for $P_{s,t}$;
            else if $P_{s,t} \in \Psi$ **then**
                mark "checked" for $P_{s,t}$;
    replace C_s and C_t with C'_s and C'_t, respectivley;
return Ψ;

Table 2: Iteration example of ONE.

	C_s	C_t	v_s	v_t	Affected paths
1	$\{v3\}$	$\emptyset$	null	null	
2	$\emptyset$	$\emptyset$	$v3$	null	
3	$\emptyset$	$\{v6\}$	$v3$	null	
4	$\{v1, v2, v4\}$	$\emptyset$	$v3$	$v6$	
5	$\{v1, v2, v4\}$	$\{v4, v5, v7, v8\}$	$v3$	$v6$	$P_{3,6}$
6	$\{v1, v2, v4\}$	$\{v5, v7, v8\}$	$v3$	$v4$	$P_{3,6}$
7	$\{v1, v2, v4\}$	$\{v2, v7, v8\}$	$v3$	$v5$	$P_{3,6}, P_{3,5}$
8	$\{v1, v2, v4\}$	$\{v7, v8\}$	$v3$	$v2$	$P_{3,6}, P_{3,5}$
9	$\{v1, v2, v4\}$	$\{v8\}$	$v3$	$v7$	$P_{3,6}, P_{3,5}$
10	$\{v1, v2, v4\}$	$\emptyset$	$v3$	$v8$	$P_{3,6}, P_{3,5}, P_{3,8}$
11	$\{v2, v4\}$	$\emptyset$	$v1$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}$
12	$\{v4\}$	$\{v6, v5, v8\}$	$v2$	$null$	$P_{3,6}, P_{3,5}, P_{3,8}$
13	$\{v4, v0, v5\}$	$\{v5, v8\}$	$v2$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}$
14	$\{v4, v0, v5\}$	$\{v8\}$	$v2$	$v5$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}$
15	$\{v4, v0, v5\}$	$\emptyset$	$v2$	$v8$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
16	$\{v0, v5\}$	$\emptyset$	$v4$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
17	$\{v5\}$	$\emptyset$	$v0$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
18	$\emptyset$	$\emptyset$	$v5$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$

Proof 5. *Assume that there exists a path $P'_{s,t}$ which contains e and $P_{s,t} = P'_{s,t}$. As $P_{a,b}$ is an affected shortest path, we know $P_{a,b} = S1 = \langle v_a, v_m, ..., v_n, v_b \rangle$ and $P'_{a,b} = S2 = \langle v_a, v_b \rangle = e$ and $w'_{S2} < w'_{S1} = w_{S1} < w_{S2}$. Since $P'_{s,t}$ contains e, the sequence of $P'_{s,t}$ must be $S3 = \langle v_s, ..., v_a, v_b, ..., v_t \rangle$. If we substitute $\langle v_a, v_b \rangle$ ($S2$) in $S3$ with $S1$, we have $S4 = \langle v_s, ..., v_a, v_m, ..., v_n, v_b, ..., v_t \rangle$. It is clearly that $w_{S4} < w_{S3}$, therefore $P_{s,t}$ can not be $S3$. The assumption does not hold; therefore our corollary stands.*

According to Corollary **??**, the process of detecting whether a path is affected is shortened. If $P_{a,b}$ is an affected shortest path, we can avoid the calculation of $P_{s,t}$ in line **??** and comparison with $P'_{a,b}$ in line **??** of Algorithm **??**. Since $P'_{s,t}$ contains e (line **??** of Algorithm **??**), so we can know $P_{s,t}$ is an affected path directly.

We now focus on the order of generating node-pairs with each from C_s and C_t, respectively.

Corollary 6. *Suppose v_a and v_b are endpoints of e, v_s (v_t) is a node connected with v_a (v_b). If $P_{s,t}$ contains e, then for a node v_k after v_b in the sequence of $P_{s,t}$, at least one path of $P_{s,k}$ contains e.*

Proof 6. *Since $P_{s,t}$ is a shortest path, its subpath $\langle v_s, ..., v_a, v_b, ..., v_k \rangle$ is also a shortest path, i.e., $P_{s,k} = \langle v_s, ..., v_a, v_b, ..., v_k \rangle$ and $P_{s,k}$ contains e.*

Corollary **??** tells us that for a node v_s connected with v_a, if a path $P_{s,t}$ ($v_t \in S_d(v_b)$) contains e, then $P'_{s,k}$ (v_k is on $P'_{s,t} = \langle v_s, ..., v_a, v_b, ..., v_k, ...v_t \rangle$) contains e as well. It is not necessary to compute $P'_{s,k}$ and compare it with $P_{s,k}$. Therefore, for a fixed v_s from C_s, we should prioritize v_t with large $d_{b,t}$ to comprise a node pair.

We have an optimized algorithm-inward node expansion algorithm (INE). The term "inward" stems from that fact that nodes of C_t are visited from far nodes to close nodes, at last v_b. Since the main logic of INE is the same with ONE, we only demonstrate different lines. INE replaces line **??**-**??** of ONE with the code in Algorithm **??**.

In Algorithm **??**, line **??**-**??** indicate that if v_k is on a shortest path which contains e, then $P'_{s,k}$ must contain e, therefore v_k is added into C'_t for the next iteration directly without computing $P'_{s,k}$. The IF sentence in line **??** makes use of Corollary **??**.

Algorithm 4: INE

```
1  while C_t is not empty do
2      v_t ← pop the node in C_t which has the largest node distance from v_b;
3      calculate P'_{s,t};
4      if P'_{s,t} contains e then
5          for each v_k lies on path P'_{s,t} && v_k is after v_b do
6              add v_k into C'_t;
7              pop v_k from C_t;
8              if P_{a,b} is an affected path && P_{a,b} ∈ Ψ then
9                  delete P_{s,k} from Ψ;
10             else if P_{s,t} ∈ Ψ then
11                 calculate P_{s,k};
12                 if P_{s,k} ≠ P'_{s,k} then
13                     delete P_{s,k} from Ψ;
14     else if P_{s,t} ∈ Ψ then
15         mark "checked" for P_{s,t};
```

Take Figure **??** as an example, the iteration process by INE is illustrated in Table **??**.

Table 3: Iteration example of INE.

	C_s	C_t	v_s	v_t	Affected paths
1	$\{v3\}$	$\emptyset$	null	null	
2	$\emptyset$	$\emptyset$	$v3$	null	
3	$\emptyset$	$\{v6\}$	$v3$	null	
4	$\{v1, v2, v4\}$	$\emptyset$	$v3$	$v6$	
5	$\{v1, v2, v4\}$	$\{v4, v5, v7, v8\}$	$v3$	$v6$	$P_{3,6}$
6	$\{v1, v2, v4\}$	$\{v5, v7, v8\}$	$v3$	$v4$	$P_{3,6}$
7	$\{v1, v2, v4\}$	$\{v2, v7, v8\}$	$v3$	$v5$	$P_{3,6}, P_{3,5}$
8	$\{v1, v2, v4\}$	$\{v7, v8\}$	$v3$	$v2$	$P_{3,6}, P_{3,5}$
9	$\{v1, v2, v4\}$	$\{v8\}$	$v3$	$v7$	$P_{3,6}, P_{3,5}$
10	$\{v1, v2, v4\}$	$\emptyset$	$v3$	$v8$	$P_{3,6}, P_{3,5}, P_{3,8}$
11	$\{v2, v4\}$	$\emptyset$	$v1$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}$
12	$\{v4\}$	$\{v6, v5, v8\}$	$v2$	$null$	$P_{3,6}, P_{3,5}, P_{3,8}$
13	$\{v4\}$	$\{v5\}$	$v2$	$v8$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
14	$\{v4, v0, v5\}$	$\emptyset$	$v2$	$v5$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
15	$\{v0, v5\}$	$\emptyset$	$v4$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
16	$\{v5\}$	$\emptyset$	$v0$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$
17	$\emptyset$	$\emptyset$	$v5$	$v6$	$P_{3,6}, P_{3,5}, P_{3,8}, P_{2,6}, P_{2,8}$

Now we compare the performance of three algorithms. The nodes visited by the three proposed algorithms are displayed in Table **??**. The numbers of nodes visited satisfy NEA>ONE>INE, and the numbers of iterations are still NEA>ONE>INE.

5. Refreshment Strategies

In this section, we introduce four heuristic-based cache refreshment strategies.

Table 4: Performance comparison of algorithms.

	v_s	C_t (NEA)	C_t (ONE)	C_t (INE)
1	$v3$	$\{v2, v4, v5, v6, v7, v8\}$	$\{v2, v4, v5, v6, v7, v8\}$	$\{v2, v4, v5, v6, v7, v8\}$
2	$v1$	$\{v6\}$	$\{v6\}$	$\{v6\}$
3	$v2$	$\{v4, v5, v6, v7, v8\}$	$\{v5, v6, v8\}$	$\{v5, v8\}$
4	$v4$	$\{v6\}$	$\{v6\}$	$\{v6\}$
5	$v0$	$\{v6\}$	$\{v6\}$	$\{v6\}$
6	$v5$	$\{v6\}$	$\{v6\}$	$\{v6\}$

5.1. Reload Whole Cache

Reloading the whole cache strategy (RWC) is to empty the whole cache and reload it based on benefit values. When computing new benefit values, the new shortest paths of queries in the query log are calculated. This is the most straightforward idea to refresh the cache. The strategy can be used in both cases where w_e increases and decreases.

Benefit values are proposed in [**?**]. To make this paper self-contained, we briefly introduce how the benefit oriented model works. All of us know that the cache utilization, or the cost reduction by using cache, is affected by the query frequency and the computational time of shortest paths. When calculating the cost saved by loading a path $P_{i,j}$ into a cache, we consider: 1) which queries can be answered by $P_{i,j}$; and 2) the saved computational cost if answering a query $Q_{i,j}$ by a cache directly.

For the first consideration, we know that a path can answer all the queries on its subpaths [**?**]. For the second consideration, it depends on how frequently a query arrives and the computational cost of that query. More specifically, it is the multiplier of the query frequency and the computational cost.

From the above analysis, the utilization of storing $P_{i,j}$ into a cache is:

$$B(P_{i,j}, \Psi) = \frac{\sum_{P_{s,t} \in S(P_{i,j}) \& P_{s,t} \notin S(\Psi)} F_{s,t} \cdot C_{s,t}}{|P_{i,j}|}. \quad (1)$$

Here, $S(P_{i,j})$ is the set of subpaths of $P_{i,j}$ and $S(\Psi)$ is the set of subpaths of paths in Ψ. When the cache is placed at the server, the time of computing shortest paths

mainly lies in the run time of API [**?**]. The numerator as a whole is the total cost reduced by caching path $P_{i,j}$. After normalized by the size of $P_{i,j}$, $B(P_{i,j})$ is the unit-size benefit brought by loading $P_{i,j}$ into a cache. Readers can understand this as the value of per unit weight in a knapsack problem. Paths with larger $B(P_{i,j})$ have higher priority to be loaded into a cache.

5.2. Replace Affected Shortest Paths with Largest Benefit

RWC wastes a lot of time. Since only an edge changes its weight, its impact on the whole network is limited. The majority of shortest paths do not change at all. As a result, reloading the whole cache is redundant. In the second strategy, we detect and update the new path information of affected paths in the cache and query log. The affected paths in the cache are removed from the cache. In the following, the benefit values of new paths are calculated and we try to add these paths into the cache one by one.

We call the second strategy RLB. In the process of detecting affected shortest paths, the new shortest path information is computed.

5.3. Replace Affected Shortest Paths with Highest Frequency Paths

RLB strategy need update benefit values of affected paths. In the third method, we detect the affected paths in the cache and query log and then update new information. The cache is filled with paths of highest query frequencies. The strategy is named RHF. Query frequencies, can be easily obtained from the query log and never change even though the network changes. It avoids computing the benefit of each path.

5.4. Replace Affected Shortest Paths Using Roulette Wheel Selection

At last, the fourth strategy is to delete all the affected paths in the cache and load new paths to the cache by the method of *roulette wheel* (RRWS). As the historical queries are not the true queries in the future, on one hand, we consider it as a reference for the future queries; on the other hand, we avoid over fitting the cache content to it.

We use a roulette wheel to determine which paths should be loaded into the cache. One feature of the roulette wheel selection is that alternatives with low attractiveness

can also be selected, although with a relatively low possibility. We adopt the idea of roulette wheel as follows: for a path p in the query log, its probability of being selected $\Pr(p)$ is proportional to its query frequency f_p, as shown in Equation **??**.

$$\Pr(p) = \frac{f_p}{\sum_i f_i}. \tag{2}$$

The summation of the probabilities of all the paths is 1. The probability of selecting each path corresponds to the probability of a variable X with a standard uniform distribution falling into an interval as shown in Equation **??**.

$$\Pr(p) = \Pr(X \in [\frac{\sum_{i=1}^{p-1} f_i}{\sum_i f_i}, \frac{\sum_{i=1}^{p} f_i}{\sum_i f_i}]). \tag{3}$$

In another word, to realize the probabilities for all the paths is equivalent to generate a random number which falls at interval [0, 1].

Take Table **??** for example, it records the path-interval correspondence. When we need to select a path into the cache, we generate a random number according to a standard uniform distribution. Suppose the randomly generated number is 0.6, then $P_{2,11}$ will be selected because 0.6 belongs to $[0.5, 0.75]$.

Table 5: A example of roulette wheel selection.

Path	Frequency	Probability	Interval
$P_{1,12}$	6	0.3	$[0, 0.3]$
$P_{1,10}$	3	0.15	$[0.3, 0.45]$
$P_{6,9}$	1	0.05	$[0.45, 0.5]$
$P_{2,11}$	5	0.25	$[0.5, 0.75]$
$P_{1,8}$	2	0.1	$[0.75, 0.85]$
$P_{7,12}$	3	0.15	$[0.85, 1.0]$

Discussion. In terms of the computational time, reloading the whole cache apparently costs the most time. The shortest paths of all queries in the query log will be computed and the benefit values are calculated and sorted. RLB, RHF and RRWS only detect and update affected shortest paths. The unaffected paths in the cache do not need to be reloaded. RHF costs time in sorting frequencies of all shortest paths. RRWS has

a process of roulette wheel selection. RLB will cost the least time if the benefit values are recorded in the initialization of cache, since only a small set of affected paths update their benefit values.

6. Experiments

We used three data sets Aalborg[1], Beijing[2], and Singapore [**?**] in our experiments. Each data set consists of a weighted road network and a query log. Since the numbers of queries in the Aalborg and Beijing query logs are relatively small, we enlarge the query logs. The query log of each data set is divided into two parts; one half is the training set and the other half is the test set. Training sets act as historical logs while test sets act as future queries. The frequency statistics are extracted from the training set, and the cache is filled based on benefit values. Table **??** summarizes the information on the data sets. Columns "Size of training set" and "Size of test set" indicate the number of queries in the training set and test set, respectively. We assume that the query trend on training sets and test sets are similar, and the statistic information from training sets can predict future queries to some extent.

Table 6: Information of data sets.

Data set	No. of nodes	No. of edges	Size of training set	Size of test set
Aalborg	129k	137k	23,720	23,720
Beijing	76k	85k	64,640	64,640
Singapore	20,801	55,892	150,000	150,000

We generated 50 instances based on each data set; one half of the instances increase weights of edges and the other half decrease weights. The increased and decreased edges must a relatively high impact on the response efficiency. To meet the criteria, we firstly generated instances with weights increased, then the decreased instances. Hereafter, we explain the process of generating instances for Aalborg. For the other data sets, the process is totally the same.

[1] http://www.dbxr.org/experimental-guidelines/

[2] http://arxiv.org/abs/cs/04100001

The cache was initialized based on the road network and training set of Aalborg. For each path in the cache, two consecutive nodes of the path forms an edge. We sorted all edges formed by the cached paths based on edges' frequencies of appearance. The 25 edges with the highest frequencies were marked. The road network in Aalborg act as the road network before the weight of one edge is increased (original network). Each time, we selected one marked edge e and increased w_e by 10 times, resulting in a changed road network (new network). The 25 edges were sequentially increased and 25 corresponding instances with weights increased were obtained. For the instances of weights decreased, new networks of increased instances act as the original networks. When w_e of marked edges recovered to their weights recorded in Aalborg, we yielded instances with decreased weights.

All the code was implemented in GNU C++. The experiments were run on a PC with an Intel i7 Quad Core CPU clocked at 3.10 GHz.

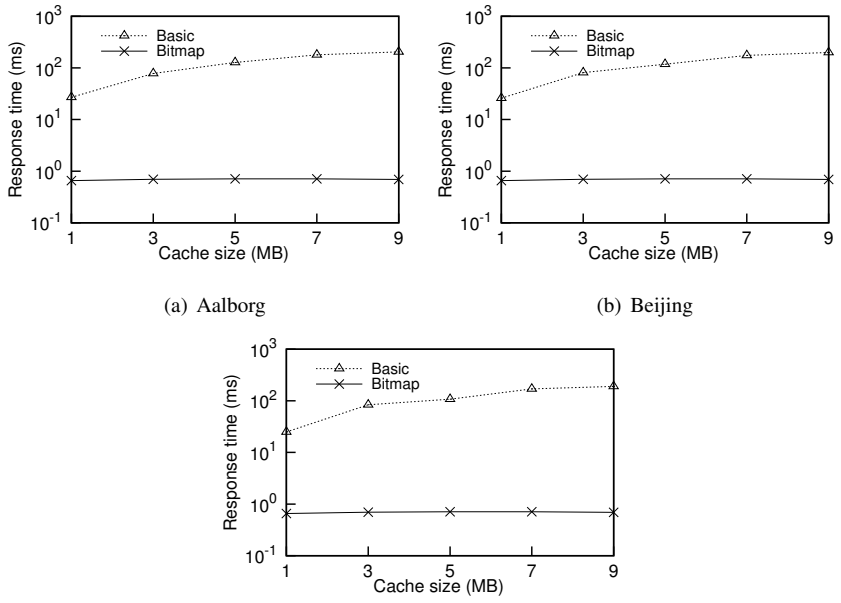
6.1. Effect of Cache Structure

We use a bitmap-based structure in Section **??** to facilitate the cache on answering users' queries. To have an impression of how the bitmap structure works, we compare it with a benchmark structure, which only records shortest paths in a path array. Queries are answered by scanning the array. The benchmark structure is called a basic cache.

For each data set, the cache is firstly initialized based on the largest benefit values of paths in the query log. Queries in the test sets arrive one by one and the total response time by two cache structure is displayed in Figure **??**. We can see that the time performance of bitmap structure is much better than that of basic structure. The response time of bitmap is only 0.7ms when the cache size is 9MB for the Aalborg data set, while it costs 205ms for the basic structure. We can get consistent results on Beijing and Singapore data sets. Moreover, the response time is almost constant with the increase of cache size for the bitmap structure.

6.2. Efficiency of Detecting Affected Paths in the Cache

Detecting affected paths in a cache is an important step before refreshing a cache. To demonstrate the performance of our algorithms, a naive method (NM) is used as a



(a) Aalborg

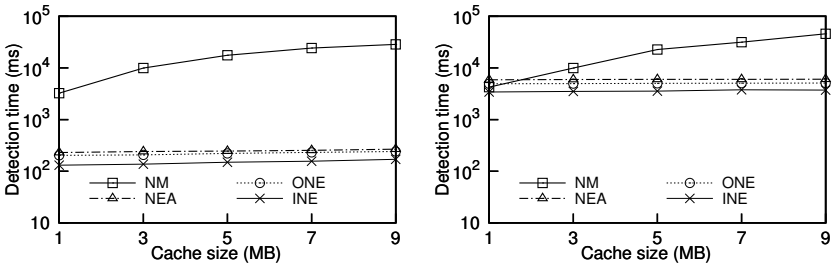
(b) Beijing

(c) Singapore

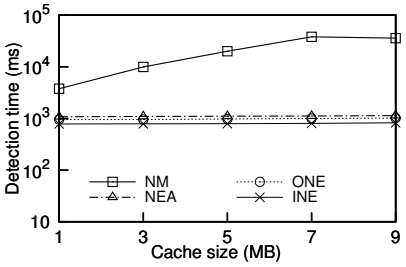
Figure 3: Response time of cache structures.

benchmark method. Due to the size of graph, NM only re-computes new shortest paths for all pairs of nodes in the cache and compares them with the original ones. If the two shortest paths for a pair of nodes are different, then an affected path is detected. We test our three affected paths detecting algorithms NEA, ONE, INE and NM on Aalborg, Beijing and Singapore data sets and the detection time of the four detection methods is regarded as the performance indicator.

The experimental result on each data set is the average of 25 instances where w_e decreases. As seen from Figure **??**, all of our algorithms have a better performance than NM. For example, in the Aalborg data set, when the cache size is 9MB, the algorithm INE achieves the best time performance and its detection time is only 131ms, while NM costs 28.67s. Meanwhile, we observe that INE is slightly superior to ONE and ONE is superior to NEA. This is consistent with our expectation.



(a) Aalborg (b) Beijing



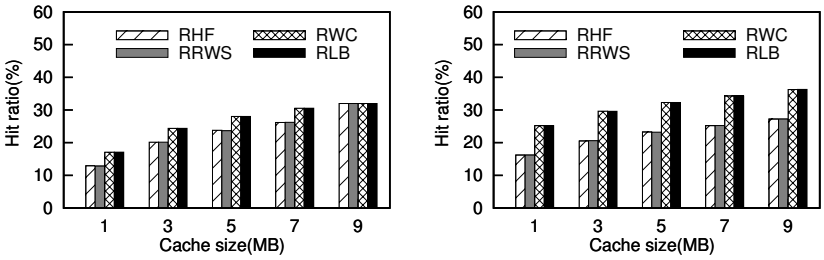
(c) Singapore

Figure 4: Performance of affected path detection methods.

6.3. Performance of Refreshment Strategies

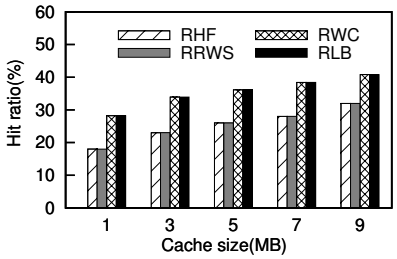
Figure **??** shows the hit-ratios of four refreshment strategies under different scales of caches. We sum up hit ratio of 50 instances for ease of read, since the differences among strategies are tiny. We can observe that RWC and RLB achieve the highest hit ratio. The reason is that even though RWC and RLB are based on different logics to update caches, the paths which are loaded into the cache are actually the same. The hit ratios of RHF and RRWS are similar and a little bit worse than the former two. This result is consistent with the logics of methods, since RHF and RRWS are not fully based on large benefit values. From the perspective of cache size, when the cache size increases, the cache will hold more paths. The cache will answer more queries, subsequently, the hit ratio increases.

Next we test the refreshment time of these four replacement strategies along with various cache sizes on three data sets. The refreshment time includes three parts: the time of detecting affected paths, the time of computing shortest paths by the server, and the time of selecting new paths into the cache. For RWC, the time of detecting affected



(a) Aalborg

(b) Beijing



(c) Singapore

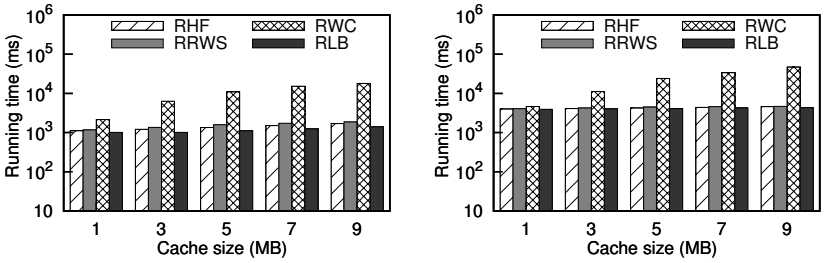
Figure 5: Hit ratios of refreshment strategies.

paths is zero. The experimental result (rf. Figure **??**) on each data set is the average of 50 instances. The performance on three data sets is consistent; each method displays similar trends on three data sets. RWC spends the most refreshment time, and the time increases as the cache size increases. It can be predicted that the disadvantage of RWC in refreshment time will amplify more if more shortest paths are queried, since RWC takes a lot of time to computing shortest paths every time the cache is emptied. RLB, RHF, and RRWS spend similar time. Precisely, RLB has the shortest refreshment time. This is consistent with our analysis in Section **??**.

From the above experiments, we can see that RLB has the best running time and hit ratio. Hence, in the real applications, RLB is suggested to be used.

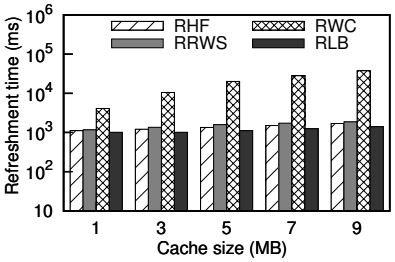
7. Conclusions

We address the problem of refreshing cache contents in a changed network. Changes of road conditions are commonly seen in the real world, i.e., traffic congestion or road



(a) Aalborg

(b) Beijing



(c) Singapore

Figure 6: Refreshment time of refreshment strategies.

constructions. Shortest path caches need refreshing periodically, so that the information is accurate and valid and the cache utilization is maximized. To the best of our knowledge, our work is the first one to discuss the cache refreshment problem when an edge changes its weight. In order to store shortest paths and answer queries efficiently, a bitmap-based cache structure is used. We introduce the concept of affected paths, and then exploit the properties associated with affected paths. In the following, we develop algorithms to detect affected shortest paths resulted from road network changes. Sequentially, four cache refreshment strategies are illustrated.

A series of experiments are conducted. The performance of cache structure and affected path detection methods are considerable. Computational results showcase that RLB is as good as RWC in terms of hit ratio while the former has a much shorter refreshment time. Our work has many applications, such as refreshment of shortest paths caches of map companies. We suggest that RLB is used to refresh caches at short intervals and RWC is used at long intervals. How to trade off the use frequency of two methods is based on the real applications.

The weakness of our work, even of most works of this kind, is that we rely heavily on the information of query logs, although we introduce roulette wheel selection to avoid over fitting. It is clearly that future queries are not a precise copy of query logs. Hence, the hit ratio is to some extent restricted by query logs. In the future, works may focus on the following directions. The cache structure, the method to detect affected paths and the strategies to refresh caches can be studied solely and then consolidated, obtaining a strengthened cache refreshment solution. The problem of "batch of updates" should be considered. In this paper, we only consider the change of single edge. The story of batch changes is not a simple repetition of single change, since edges could interfere. The impact of multiple edges is combinatorial. Currently we just consider the changes of the graph; it would be interesting to investigate the changes of query patterns. Of course, it may be possible to study directed networks in the future as in the real life one direction of a road may be closed for construction or other purposes while the other direction is still open.

Acknowledgments.

The work is partially supported by the National Basic Research Program of China (973 Program) (No. 2012CB316201), the National Natural Science Foundation of China (No. 61322208, 61272178, 61129002), the Doctoral Fund of Ministry of Education of China (No. 20110042110028), the Fundamental Research Funds for the Central Universities (No. N120504001, N110804002), and the Support Plan for Young Teachers of Shanghai (No. ZZSD15095).

References