Monitoring Minimum Cost Paths on Road Networks

Yuan Tian Ken C. K. Lee Wang-Chien Lee

Department of Computer Science Engineering
The Pennsylvania State University
University Park, PA 16802, USA
{yxt144,cklee,wlee}@cse.psu.edu

ABSTRACT

On a road network, the minimum cost path (or min-cost path for short) from a source location to a destination is a path with the smallest travel cost among all possible paths. Despite that min-cost path queries on static networks have been well studied, the problem of monitoring min-cost paths on a road network in presence of updates is not fully explored. In this paper, we present *PathMon*, an efficient system for monitoring min-cost paths in dynamic road networks. PathMon addresses two important issues of the min-cost path monitoring problem, namely, (i) *path invalidation* that identifies min-cost paths returned to path queries affected by network changes, and (ii) *path update* that replaces invalid paths with new ones for those affected path queries. For (i), we introduce the notion of *query scope*, based on which a *query scope index* (QSI) is developed to identify affected path queries. For (ii), we devise a *partial path computation algorithm* (PPCA) to quickly recompute the updated paths. Through a comprehensive performance evaluation by simulation, QSI and PPCA are demonstrated to be effective on the path invalidation and path update issues.

Categories and Subject Descriptors

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

Keywords

Location-based services, path search and updates, road networks, monitoring system

1. INTRODUCTION

Path search is one of the most popular and essential location-based services in our daily life. By issuing a path query that specifies a source address and a destination address to a path search service provider (e.g., MapQuest, Google Maps, Yahoo! Maps, Bing Maps[1] etc.), one can receive a recommended route displayed as an annotated path on the map together with an estimated mileage and turn-by-turn driving instructions. Dependent on various application contexts, the path costs can be either physical distance, travel time, toll charge or safety risk. Among all possible paths, the one with the smallest cost is considered to be desirable to the querying user. We call such a path the *minimum cost path* (or *min-cost path* in short).

However, changes of the underlying network may cause the cost of any min-cost path between a given source and a destination no longer the smallest. In other words, min-cost paths are subject to the underlying network status. In practice, keeping track of min-cost paths are very important to many applications. For example, trucking and courier companies expect their vehicles to travel on min-cost paths to keep their operation costs low. Emergency service vehicles need to take the fastest routes to arrive at the destinations for life-critical missions. Hence, a min-cost path monitoring system is demanded. In this paper, we present a min-cost path monitoring system called *PathMon*, to monitor the changes of dynamic road networks and determine the most current min-cost paths between specified sources and destinations.

[1]http://www.mapquest.com/, http://maps.google.com/, http://maps.yahoo.com/, http://www.bing.com/maps, respectively.

1.1 Continuous Min-Cost Path Query

We model an ever-changing network as a set of time-stamped graphs $\{G_t | G_t = \langle N_t, E_t, \mathcal{C}_t \rangle\}$, where N_t is a set of nodes representing the end points of roads and road intersections present at time t; $E_t = \{\langle n_i, n_j \rangle | \ n_i, n_j \in N_t\}$ is a set of edges standing for the road segments and $\mathcal{C}_t = \{C_{\langle n_i, n_j \rangle, t}\}$ is a set of positive values suggesting the cost of the edges at time t. Here, the cost may be the physical edge length, expected travel time, toll charge or others defined by applications.

Upon G_t, a path from a source node s to a destination node d ($s, d \in N_t$), denoted as $p(s,d)$, is a sequence of interleaving edges and nodes: $p(s,d) = (n_{i_0}, \langle n_{i_0}, n_{i_1} \rangle, n_{i_1}, \langle n_{i_1}, n_{i_2} \rangle, \cdots, n_{i_{k-1}}, \langle n_{i_{k-1}}, n_{i_k} \rangle, n_{i_k})$ where $n_{i_0} = s$ and $n_{i_k} = d$. Because in general, an edge can be uniquely identified with its end nodes, we only use the nodes to represent a path, i.e. $p(s,d) = (n_{i_0}, n_{i_1}, \cdots, n_{i_k})$. The total cost of a path $p(s,d)$ at time t, denoted by $C_{p(s,d),t}$, is equal to $\sum_{j=1}^{k} C_{\langle n_{i_{j-1}}, n_{i_j} \rangle, t}$. Among all possible paths formed between s and d, the min-cost path in G_t denoted by $P_t(s,d)$ is one with the smallest total cost. Hereafter, we denote the smallest path cost from s to d in G_t by $||s,d||_t$, and this is also referred to as the *network distance* from s to d. We use network distance and minimum path cost interchangeably in this paper.

Now, given a series of time-stamped graphs (i.e., $G_{t_0}, G_{t_1}, G_{t_2}, \cdots$) that represent a network at different times, a continuous min-cost path query $q(s,d)$, as formalized in Definition 1, provides a series of min-cost paths, i.e. $P_{t_i}(s_t, d)$, corresponding to each G_{t_i}.

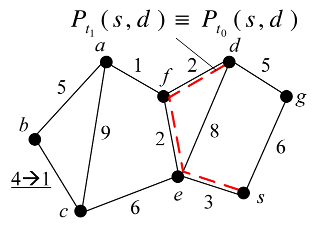
Definition 1. **Continuous min-cost path query.** *Given a dynamic road network G_t, a moving source node s (with the location at time t denoted as s_t) and a fixed destination node d, a continuous min-cost path query $q(s,d)$ returns a series of the latest min-cost* *paths $P_t(s_t, d)$ on G_t from the source node s_t to d, invoked by road network change at time t.*[2] □

In this paper, we address the issues of processing of continuous min-cost path queries, which is also referred to as *min-cost path monitoring*.

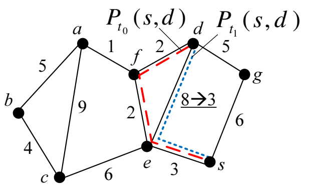
1.2 Research Issues and Our Solution

In a large-scale path monitoring system where a massive number of continuous min-cost path queries are running, an efficient mechanism to identify updated min-cost paths whenever network changes occur is needed. An intuitive approach to process continuous min-cost path queries is to reevaluate min-cost paths from scratch whenever a network update occurs. This is apparently inefficient since not all paths are necessarily *affected* by a network change. Moreover, path searches from scratch is computationally expensive. In fact, re-computation of some non-affected paths can be avoided and those affected queries can be partially recomputed.

Running example. In a simple road network as shown in Figure 1(a), a continuous min-cost path query is issued to monitor a min-cost path from node s to d. At time t_0, a result path $P_{t_0}(s, d) = (s, e, f, d)$ is indicated in a dashed line with path cost 7. At time t_1 ($t_1 > t_0$), the cost of edge $\langle b, c\rangle$ changes from 4 to 1. However, it does not lead to a new min-cost path, i.e. $P_{t_0}(s, d) = P_{t_1}(s, d)$. In this case, $\langle b, c\rangle$ is a *non-affecting edge* with respect to $P_{t_0}(s, d)$.



(a) $\langle b, c\rangle$ is a non-affecting edge



(b) $\langle b, c\rangle$ is an affecting edge

Figure 1: Example road network

Besides, at time t_2 ($t_2 > t_1$), the cost of $\langle e, d\rangle$ decreases from 8 down to 3 (as shown in Figure 1(b)). Now, $P_{t_2}(s, d)$ becomes (s, e, d). In this case, $\langle e, d\rangle$ is an affecting edge. □

From the example, we obtain two observations. First, with respect to a determined min-cost path, not all edge changes result in a new min-cost path. Therefore, those edges are non-affecting edges. Whenever an edge is identified to be a non-affecting edge to a min-cost path, it is safe to keep the original min-cost path. On the other hand, by detecting changes happened to the affecting edges, we can quickly identify which min-cost path queries are impacted by the changes. Second, although an existing min-cost path may be affected by an edge change, we expect a large portion of the existing min-cost path remaining in the new min-cost path. Recall in the above example that the edge (s, e) remains in the new path $P_{t_2}(s, d)$. As such, it is possible to reuse some common parts of an existing path and only to compute a new portion in the new min-cost path.

In light of these observations, we approach the min-cost path monitoring problem by addressing two important issues, namely, (i) *path invalidation* and (ii) *path update*, and develop a novel, efficient and scalable path monitoring system, called *PathMon*. The system model of *PathMon* is depicted in Figure 2. Here, a path monitoring server is responsible for coordinating path invalidation and path updates. At the back end is a network monitoring subsystem that reports the road network changes to the server. To the front end, a number of users submit path queries and register them on the server. Users are notified of the updated paths by the server when their queried paths are affected by changes in the underlying network and results are updated.

[2]We assume that a mobile user always follows the path returned by the query. Hence if road costs in the network does not change, the previously returned path needs no update.

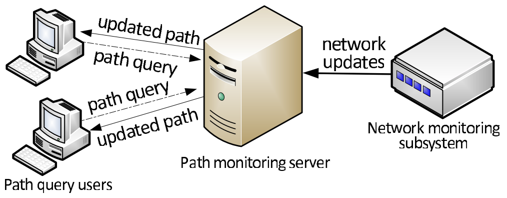


Figure 2: The system model of PathMon

The PathMon system contains two important components, namely, the *query scope index (QSI)* and the *partial path computation algorithm (PPCA)*. *QSI* keeps track of *query scopes*, i.e., subnetworks that are involved in derivation of answer paths for individual queries. In presence of edge cost updates within some query scopes, corresponding queries are detected to be affected (and the originally calculated min-cost path possibly becomes invalid). *PPCA* utilizes the intermediate processing states captured in previous path search processes to facilitate partial re-computation of affected paths. Since paths computed by a deterministic search algorithm on a slightly changed network would access almost the same set of road network data in the same order, some path re-computation can be alleviated by reusing some processing states of a previous search.

In summary, our contributions made in this paper are five-fold.

1. We analyze the min-cost path monitoring problem, identify two essential issues, namely, path invalidation and path update, and explore useful min-cost path properties to develop efficient index and algorithms for min-cost path monitoring.
2. We introduce the notion of query scopes in the road network, based on which we design the *query scope index (QSI)*, to facilitate identification of min-cost path queries affected by network changes. Furthermore, QSI is extended to maintain processing states for PPCA.
3. We develop the *partial path computation algorithm (PPCA)* to quickly determine a new min-cost path based on the idea of reusing the maintained *processing states*.
4. We develop *PathMon*, an efficient min-cost path monitoring system. The core of PathMon includes QSI and PPCA.
5. We conduct a comprehensive performance evaluation based on simulations. The experiment results show that our *PathMon* significantly outperforms existing representative works in identifying affected queries and the path re-computation.

The rest of this paper is organized as follows: Section 2 reviews the existing works related to this study. Section 3 details our path monitoring problem analysis and presents some useful properties of road network based on which our solution is developed. Section 4 details the design and implementation of the query scope index (QSI) and related operations. Section 5 presents the partial path re-computation algorithm (PPCA). In Section 6, we evaluate our approach in comparison with representative approaches. Section 7 concludes this paper and states our future work.

2. RELATED WORK

In the following, we review related work on dynamic road network models, path search algorithms, and continuous path search.

Dynamic Road Network Models. Some models such as stochastic models, speed patterns, time-delay functions consider the change of road networks to be predictable. In details, stochastic models [7] handles the network changes in accordance with a distributed Markov chain. Speed pattern [11, 12] specifies the road costs at different time for different vehicle types. Time-delay functions [8] are used to represent the travel costs on individual road segments. Based on the predicted network costs, shortest paths can be determined. On the other hand, some works consider the dynamic networks as unpredictable. One of the most popular model is the quasi-dynamic model, where the edge costs are assumed to change quickly and then remain invariant until next changes [13, 5]. In this paper, our study is based on this quasi-dynamic model.

Shortest path search. Shortest path search algorithms for static network are well studied. Single-source shortest path search algorithms include Bellman-Ford algorithm [1, 2], Dijkstra's algorithm [3] and A* algorithm [16], while all-pair shortest path algorithms include Floyd-Warshall algorithm [4] and Seidel's approach [6]. Since network traversal is a very expensive operation, especially in a large network, HEPV [14], HiTi [18] and a grid-based partition method in [17] were proposed to pre-compute and materialize some shortest paths to facilitate online path searches. Due to significantly large maintenance overhead, these approaches are not favorable to dynamic networks.

Path update algorithms. Single source shortest path update algorithms have been studied in [10, 15]. The basic idea is to propagate the changes from an updated edge to a set of vertices succeeding it in the shortest path tree. Then, all edges connecting to the affected vertices are examined to see whether they can form a path with smaller cost. However, these approaches are not scalable to the number of queries and updated edges, because 1) they ignore the path invalidation issue, checking each updated edge for every path, incurring excessive invalidation cost, and 2) the algorithms can only deal one edge update each time and take several rounds of path computations for multiple updated edges.

Continuous min-cost path monitoring. Most recently, [9] has studied the min-cost path monitoring problem. The main focus of that work is to quickly identify path queries affected by a network change. In [9], an ellipse bound method (EBM) is developed. By EBM, Each min-cost path is associated with an elliptic geographical area called *affecting area*, within which all updated edges are considered to affect the corresponding path. More specifically, for a min-cost path between a source node s and a destination d and a maximum speed V_{max} in the network, the elliptic affecting area is formed with s and d as the two foci and the length of the major axis is $||s, d|| \times V_{max}$, where V_{max} is the maximum travel speed in the network. This affecting area is guaranteed to be large enough to cover the corresponding min-cost path from s to d and all the edges that can affect the path. Further, a grid index is adopted to keep track of every affecting area as a number of cells. Then, each cell can be associated with multiple queries simultaneously. As long as the cost of an edge covered by a grid cell is updated, all the associated queries are determined to be affected by the change. The affected paths are then recomputed from scratch.

Nevertheless, using *geographical areas* to identify possible affected min-cost paths, this approach in fact covers a lot of unrelated edges and triggers many unnecessary path re-computations. Besides, the presumed maximum speed makes a conservative estimation which usually turns out a huge affecting area, thus further degrading the overall performance. Moreover, the geographical affecting area does not support non-spatially related cost models (e.g., toll charges, safety risks). Last but not least, since full re-computation is used, this EBM is not very efficient, as indicated by our performance evaluations.

3. PRELIMINARIES

In this section, we first provide a theoretical analysis on the properties of the road networks and min-cost path search algorithms. Based on the analysis, we present the theoretical foundation of our path invalidation and update algorithms. Then, we introduce the architecture of our PathMon system.

3.1 Theoretical Analysis

To address the path invalidation and path update issues, we need to look back to the fundamental operation, i.e. the path search algorithm, which provides critical information and useful properties for deriving solutions.

3.1.1 Min-cost path search properties

Without loss of generality, we consider Dijkstra's algorithm here due to its generality and popularity, and more importantly, its insightful properties. Please note that although the following discussions only focus on Dijkstra's algorithm, our theorems and algorithms can actually be easily generalized to other path search algorithms.

Dijkstra's algorithm employs the best-first strategy in network traversal. Starting from the destination d, in each step, the algorithm picks a node v with the minimum accumulative cost to d (denoted as c_v) among all unexplored nodes to visit.[3] Once v is being visited, the accumulative cost to d is determined as the network distance $||v, d||$. Then, each unvisited neighbor w of v is assigned with the accumulative distance $c_w = \min\{c_w, ||v, d|| + C_{\langle w,v \rangle}\}$. This process repeats until the source node s is visited, and the answer path can be constructed by tracing back from s to d. A min-cost path spanning tree T rooted at d is formed during the algorithm, in which the path from any tree node n to d is the min-cost path between n and d.

We illustrate the search process of Dijkstra's algorithm with the same sample road network as given in Figure 1(a). Two main data structures are utilized to support the search process. One is a visited node set V that prevents re-visiting of nodes and maintains the predecessor information for back tracking. The other is a priority queue H that sorts the unvisited nodes in non-descending order of accumulative costs to the destination.[4] The trace of the search is given in Figure 3.

Step	**Latest visited node (node id, predecessor, accumulative cost)**	**content of the heap (after exploring edges)**
0	$\emptyset$	$((d, \perp, 0))$
1	$(d, \perp, 0)$	$((f, d, 2), (g, d, 5), (e, d, 8))$
2	$(f, d, 2)$	$((a, f, 3), (e, f, 4), (g, d, 5), (e, d, 8))$
3	$(a, f, 3)$	$((e, f, 4), (g, d, 5), (b, a, 8), (e, d, 8), (c, a, 12))$
4	$(e, f, 4)$	$((g, d, 5), (s, e, 7), (b, a, 8), (e, d, 8), (c, e, 10), (c, a, 12))$
5	$(g, d, 5)$	$((s, e, 7), (e, d, 8), (b, a, 10), (c, e, 10), (s, g, 11)), (c, a, 12))$
6	$(s, e, 7)$	search terminates.

Figure 3: The trace of a search for $P(s, d)$

The min-cost path spanning tree formed by Dijkstra's algorithm is shown in Figure 4 in bold black lines (with the circled numbers indicating the order of node being visited). This spanning tree (denoted as T) has the following properties.

Property 1. *The min-cost spanning tree properties.*

1. Recursiveness. *Let $\hat{T}(u)$ denote the subtree of T rooted at node u, then $\hat{T}(u)$ is also a min-cost path spanning tree from all nodes in $\hat{T}(u)$ to u, i.e.* $\forall v \in \hat{T}(u)$, $P(v, u) \subseteq \hat{T}(u)$.
2. Distance priority. $\forall v \in \hat{T}(u)$, $||v, d|| \geq ||u, d||$.
3. Distance bound. $\forall v \in T$, $||v, d|| \leq ||s, d||$. *On the other hand,* $\forall v' \notin T$, $||v', d|| \geq ||s, d||$.

PROOF. The proof is straightforward according to Dijstra's algorithm and we omit it here to save space. ■

As exemplified in Figure 4, we have $e \in \hat{T}(f)$, thus $||e, d|| = 4 > ||f, d|| = 2$, and $P(e, f) = \{e, f\} \subset \hat{T}(f)$. Also, all the nodes covered by T have smaller or equal distances to d than s.

[3]Since the source node moves as the driver travels in the road network, we use the fixed destination as the starting node of the search.

[4]When two nodes have the same accumulative costs to the destination, we use the node IDs as the tie breaker.

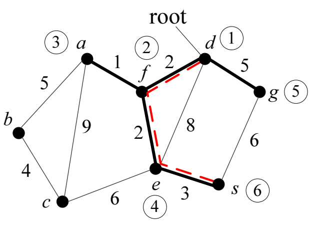


Figure 4: Illustration of the min-cost path spanning tree

3.1.2 Path invalidation with query scopes

Since s is a leaf node of T, if the min-cost path $P(s,d)$ changes, the structure of T must change. However, only monitoring the edges on the path is not enough. To show the relationship between the tree and the path in presence of changes, we consider the following cases (assuming $P_{t_0}(s,d) = (s,e,f,d), t_1 > t_0$):

1. The updated edge is already in the tree T: e.g., $\langle s,e\rangle \in T$. If $C_{\langle s,e\rangle,t_1}$ increases to 10, $P_{t_0}(s,d) = (s,e,f,d)$ will be invalid, and the new path will be $P_{t_1}(s,d) = (s,g,d)$.
2. The updated edge is not in T, but both of its end nodes are covered by T: e.g., $s,g \in T$ but $\langle s,g\rangle \notin T$. If $C_{\langle s,g\rangle,t_1}$ decrease to 1, then correspondingly we will have $P_{t_1}(s,d) = (s,g,d)$, and the original path will become invalid.
3. Some updated edges have only one end node in T: e.g., $e,a \in T$, $c \notin T$, so $\langle e,c\rangle \notin T$, $\langle c,a\rangle \notin T$. If $C_{\langle e,c\rangle,t_1}$ and $C_{\langle c,a\rangle,t_1}$ both decrease to 0.1, $P_{t_0}(s,d)$ will change to $P_{t_1}(s,d) = (s,e,c,a,f,d)$.
4. None of the updated edges has any end node covered by T: e.g., $c,b \notin T$. If $C_{\langle c,b\rangle,t_1}$ changes, no matter what the new value is, the path will not be affected.

From the above discussion, we can see that if an edge has at least one end node covered by T, it may result in a change of the min-cost path. In order to catch all possible affecting edge changes, we need to monitor all edges linking to the nodes in T. In other words, the key issue in path monitoring is to detect changes involving the min-cost path spanning tree. Based on the spanning tree, we define the *query scope* of a path query as in Definition 2.

Definition 2. **Query scope.** *Given a min-cost path query q with the answer path $P_t(s,d)$ and the min-cost path spanning tree $T_t(q)$, the query scope $QS_t(q)$ is a set of nodes covered by $T_t(q)$. Only the edges with at least one edge node in $QS_t(q)$ may be an affecting edge of q with respect to $P_t(s,d)$.* □

According to the definition of query scope and the properties of the min-cost path spanning tree, we obtain the following property and condition for eliminating irrelevant network updates.

Property 2. **Distance bound of query scope.** *For a query $q(s,d)$, at any time t, its query scope $QS_t(q)$ satisfies $\forall v \in QS_t(q)$, $||v,d||_t \leq ||s,d||_t$. Meanwhile, $\forall v' \notin QS_t(q)$, $||v',d||_t \geq ||s,d||_t$.*

Lemma 1. *If at time $t_1 > t_0$ no edge with at least one end node in $QS_{t_0}(q)$ is updated, i.e. $\forall\langle x,y\rangle$ whose cost is changed, $x \notin QS_{t_0}(q) \vee y \notin QS_{t_0}(q)$, the min-cost path $P_t(s,d)$ will not be affected, i.e. $P_{t_1}(s,d) \equiv P_{t_0}(s,d)$.* □

PROOF. The lemma can be proven by contradiction and induction. Due to the space limitation, we omit the proof here. ■

Further, when no edge inside the query scope is updated, it can be proved that not only the original min-cost path does not change, but the entire min-cost path spanning tree remains unchanged.

Lemma 2. *$T_{t_1}(q) \equiv T_{t_0}(q)$ if no edge with at least one end node in $QS_{t_0}(q)$ is updated at time t_1. Hence, the query scope $QS_{t_1}(q) \equiv QS_{t_0}(q)$.* □

PROOF. By treating each node in $T_{t_0}(q)$ as s and apply Lemma 1, we see that no path in $T_{t_0}(q)$ has been changed at time t_1. So, $T_{t_1}(q) \equiv T_{t_0}(q)$. ■

Due to the distance bound property, we can express Lemma 1 and 2 using the distance criterion, as stated in Corollary 1.

Corollary 1. *For a query $q(s,d)$, if at time $t_1 > t_0$, $\forall\langle x,y\rangle$ that is updated, $||x,d||_{t_0} > ||s,d||_{t_0}$ and $||y,d||_{t_0} > ||s,d||_{t_0}$, then $P_{t_1}(s,d) \equiv P_{t_0}(s,d)$, $QS_{t_1}(q) \equiv QS_{t_0}(q)$.* □

Given the query scope $QS_t(q) = \{v | v \in T_t(q)\}$ and the above properties, we can derive our path invalidation algorithm based on detecting the changes in the query scopes.

3.1.3 Path update with processing states

Next, let us consider the path update problem.

As we have discussed, to guarantee the correctness of the min-cost path, the whole min-cost path spanning tree needs to be always up-to-date. According to the definition of a min-cost path spanning tree T (rooted at the destination node d), $\forall v \in T$, $P(v,d) \subseteq T$. If after some network updates, some min-cost paths in T remain unchanged, they may be reused as a part of the new spanning tree, i.e. if $\exists v \in T_{t_0}$, $P_{t_0}(v,d) \equiv P_{t_1}(v,d)$ and $||v,d||_{t_1} < ||s,d||_{t_1}$, then $P_{t_1}(v,d) \subseteq T_{t_1}$. In the path search algorithm, this min-cost path information from each node in the spanning tree to the root is stored in the visited node set V and priority queue H. From the visited entries in V, we can depict the current structure of the spanning tree. Meanwhile, H provides a guidance for the future steps without needing to retrieve the neighbors of nodes in V repeatedly.

From the trace in Figure 3, we can also observe that, the node to be visited in a specific step is determined by the content of V and H at that step, and then V and H are updated according to the neighboring nodes and edges of the visited node. For instance, when $V = \{(d,\perp,0)\}$, $H = ((f,d,2),(g,d,5),(e,d,8))$, the node to be visited next must be f because it has the minimum accumulative distance to d in H and is not in V. This process is deterministic, i.e. if the network remains unchanged, the algorithm will always follow the same steps and find out the same min-cost path, and the final layout of the spanning tree will also be the same. Let us denote the content of V and H in the step that node n is visited as V^n and H^n, then (V^n, H^n) represents the state of the search process. We define (V^n, H^n) as the *processing state* at node n (see Definition 3), thus the entire search process can be viewed as a sequence of processing states, i.e. $S(n_1), S(n_2), \cdots, S(n_k)$ where $S(n_i) = (V^{n_i}, H^{n_i}), i = 1,2,\cdots,k$.

Definition 3. **Processing state.** *Given a min-cost path query $q(s,d)$, the processing state $S(n)$ at a node n contains a set of visited nodes V^n and the current content of priority queue H^n just after n is visited.* □

We exemplify the processing state in Figure 5. In the search for $P(s,d)$, the processing state $S(f)$ at node f contains the visited set $V^f = \{(d,\perp,0), (f,d,2)\}$ from which we can recover the spanning tree at the time f is visited, and the priority queue H^f with all nodes (a, e and g) linked to d and f but not yet explored. Since $(a,f,3)$ is the first entry in H^f, the processing state $S(f)$ determines that the node a is going to be visited next.

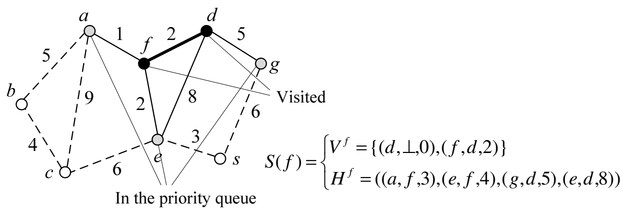

a
d
f
g
b
c
e
s
1
2
5
5
9
2
8
6
4
3
6
Visited
In the priority queue
$S(f) = \begin{cases} V^f = \{(d,\perp,0),(f,d,2)\} \\ H^f = ((a,f,3),(e,f,4),(g,d,5),(e,d,8)) \end{cases}$

Figure 5: Example of processing state

Treating a min-cost path search as a process of growing a spanning tree, a processing state is generated each time a new node is added to the tree. We have the following observations. Let G and G' represent a network before and that after an edge change, respectively. Given the same path query evaluated on G and G', we can obtain two sequences of states $\mathcal{S}$ and $\mathcal{S}'$, respectively. If both

$\mathcal{S}$ and $\mathcal{S}'$ are exactly the same, it indicates the spanning trees T (obtained in G) and T' (obtained in G') are identical (thus the min-cost paths must be identical). We refer two sequences of states to be the same if they consist of the same number of states and each corresponding pair of states have the same sets of visited nodes and contents of priority queues. On the other hand, if $\mathcal{S}$ and $\mathcal{S}'$ are different, the spanning tree on G' may possibly become different from one obtained in G (even though the min-cost paths may or may not be different). While determining $\mathcal{S}'$, which equals to a full path evaluation, is computationally expensive, it is economic to reuse some of the states in $\mathcal{S}$ if they are previously determined on G and maintained in the system. Logically, if not too many edges have been updated in G, some paths in T will still appear in T'. With the stored states of $\mathcal{S}$, we can directly rebuild the part of T that is unchanged in T' and avoid re-computation from scratch. Based on this idea, we can restart a path search at a state $S(n_k) \in \mathcal{S}$ which guarantees that $S(n_1), \cdots, S(n_k)$ are identical to their counterparts in $\mathcal{S}'$. We explain this in Lemma 3.

Lemma 3. *Let $S(n)$ be the processing state at node n ($||n, d||_{t_0} \leq ||s, d||_{t_0}$) in a search process for $P_{t_0}(s, d)$ before the network changes. If $\forall\langle x, y\rangle$ whose cost is updated, $||x, d||_{t_0} > ||n, d||_{t_0}$ and $||y, d||_{t_0} > ||n, d||_{t_0}$, and $S'(n)$ is the processing state at n in a search process after the changes, then $S'(n)$ is identical to $S(n)$, denoted as $S'(n) \equiv S(n)$.* □

PROOF. From Lemma 2 we know that $||x, d||_{t_0} > ||n, d||_{t_0} \Rightarrow ||x, d||_{t_1} > ||n, d||_{t_1}$, and similarly $||y, d||_{t_1} > ||n, d||_{t_1}$. $\langle x, y\rangle$ can be accessed only when x or y is visited. According to the node ordering in H, i.e. non-descending order of the accumulative cost to d, n has already been visited before the visit to x or y. Therefore, $S'(n)$ has already been determined by the search process before accessing the any changed edge $\langle x, y\rangle$. Since the search process has only accessed unchanged edges by the time of generating $S'(n)$, due to the determinism of the path search algorithm, $S'(n) \equiv S(n)$. ■

To illustrate Lemma 3, we consider the change of edge $\langle s, e\rangle$ as shown in Figure 5. Since $\min\{||s, d||, ||e, d||\} = 4$ and $||f, d|| = 2$, $S'(f)$ will remain the same as $S(f)$. In fact, because d, f and a are all nearer to d than $\langle s, e\rangle$, $S(d)$, $S(f)$ and $S(a)$ are not affected. Finally, as $||a, d|| = 3 > ||f, d|| > ||d, d||$, $S(a)$ should be the latest state not affected by the edge changed. Thus, the search can restart with the visited nodes and the priority queue of $S(a)$. This is analogy to "cut" the spanning tree to the shape stored in $S(a)$ and let it grow again.

Besides, if a state $S'(n) \in \mathcal{S}'$, which is obtained after incorporating the changed edges inside the (potentially enlarged) query scope, is identical to the state $S(n) \in \mathcal{S}$, it can be certain that the rest of states in $\mathcal{S}$ and $\mathcal{S}'$ are exactly the same. Thus, the computation of $\mathcal{S}'$ can be terminated earlier by sharing the remaining states after $S(n)$ from $\mathcal{S}$. Denote $\{\langle u, v\rangle^+\}$ as the edges whose cost are increased, and $\sum \Delta C_{\langle u,v\rangle^+}$ gives the upper bound of query scope enlargement (so that $||s, d||_{t_1} \leq ||s, d||_{t_0} + \sum \Delta C_{\langle u,v\rangle^+}$). The termination condition is stated in Lemma 4.

Lemma 4. *Let $S(n)$ be the processing state at node n ($||n, d||_{t_0} \leq ||s, d||_{t_0}$) in a search process for $P_{t_0}(s, d)$ before the network changes. Given that all updated edge $\langle x, y\rangle$ satisfies either 1) $||x, d||_{t_1} > ||s, d||_{t_0} + \sum \Delta C_{\langle u,v\rangle^+}$ and $||y, d||_{t_1} > ||y, d||_{t_0} + \sum \Delta C_{\langle u,v\rangle^+}$, or 2) $||x, d||_{t_1} < ||n, d||_{t_1}$ and $||y, d||_{t_1} < ||n, d||_{t_1}$, and $S'(n)$ is the processing state at n in a search process after the changes, if $S'(n) \equiv S(n)$, then for any node m such that $||n, d||_{t_0} < ||m, d||_{t_0} \leq ||s, d||_{t_0}$, $S'(m) \equiv S(m)$.* □

PROOF. We omit the proof here to save space. ■

Our partial path re-computation algorithm can be derived based on the above analysis. By ensuring two conditions: i) restarting from a processing state which was generated in a previous search and unaffected by updates; ii) terminating at another processing state that guarantees the equality between the following search steps and existing search results; we can effectively save computation cost on unnecessary network traversals while preserve the algorithm correctness.

3.2 PathMon System

The PathMon server, as shown in Figure 6, includes three major components: a) the network data index, b) the query index, and c) the path search/update processing engine. The data index maintains the network topological information, including nodes, edges and edge costs. In the implementation, we index those nodes and their edges as adjacent lists based on the CCAM storage scheme designed for road networks [19]. The data index is mainly used to support the evaluation of path queries and path updates. On the other hand, the query index keeps track of the registered continuous min-cost path queries. Upon network changes, the query index is looked up to determine affected path queries. Based on the idea of query scopes, we devise a *query scope index* as the query index. We leave the detailed discussion to Section 4. Last, the path search/update processing engine, triggered by events, is responsible for search or update paths. For example, when an edge is updated, the engine accesses the query index to identify affected queries and then accesses the data index to recompute the corresponding result paths. The engine also maintains the indices upon query arrival/removal and network updates. To reduce the cost of re-computation, we explore the partial path computation algorithm and detail it in Section 5.

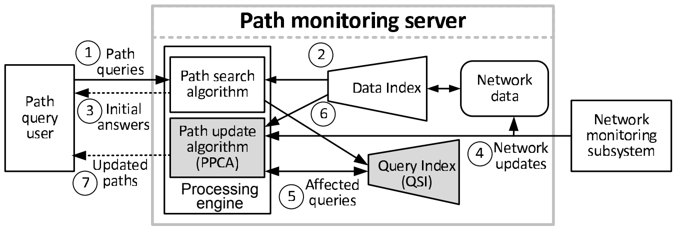


Figure 6: The PathMon system architecture

In the PathMon system, each continuous min-cost path query goes through a life cycle of three stages: (i) query initialization, (ii) path update and (iii) query removal. Query initialization starts when a new path query is registered to the path monitoring server (①). Then the server computes the answer path based on the current state of the network (②) and returns it to the user (③). Meanwhile, the query index is updated to accommodate this newly registered continuous query. Path update takes place when network updates from network monitoring subsystem are received (④). While network status and the data index are updated, the path update algorithm is triggered to detect affected queries from the query index (⑤) and accesses the data index for updated paths (⑥). Then, updated path is delivered to the user (⑦). In case that a registered query is not affected by the change, no path update is needed and the path previously delivered to the user is asserted to be valid. Query removal occurs when a query is terminated by the user. Terminated queries are removed from the query index.

4. PATH INVALIDATION

In this section we study different query scope representations, present the design of query scope index (QSI) and describe how to efficiently look up affected queries via QSI.

4.1 Query Scope Representation

According to Definition 2, a min-cost path query $q(s, d)$ has a query scope $QS_t(q)$ that contains all nodes in the spanning tree $T_t(q)$. In our study, we consider three alternative representations for query scopes in terms of representation precision and storage overhead, as described in the following:

1. **All-node materialization.** This approach records exactly every node inside a query scope. It is the most precise one (no irrelevant nodes are involved), but consumes the most storage space.
2. **Anchor node materialization.** Observing that nearby nodes (in terms of network proximity) are very likely to be accessed together by the path search algorithm, we may pick only one node among some nearby nodes as the "anchor" to refer a query (e.g. the query identifier) and the distance from the anchor to the destination. Other nodes can obtain the information by traversing neighboring subnetworks to visit nearby anchors, and then decide whether it is in a query scope. With only the anchor nodes recorded, the anchor node materialization reduces the storage cost, but upon checking the association between nodes and query scopes, this approach incurs additional computational and I/O cost due to extra network traversals.
3. **Grid-based materialization.** Rather than individual nodes, we use affecting edges to determine the bounding area for a query scope. Here, we first partitioning the areas covered by the road network into a grid. A grid cell is preserved as a part of the query scope if it covers any affecting edges. Though this representation is simple and takes less storage than the above two, this geo-spatial partition suffers from false hits since some edges within affected grid cells may not necessarily be affecting edges.

In PathMon, we adopt the anchor node materialization for the query scope as the system default, because it generalizes the all-node materialization and fully exploits the network connectivity. By carefully selecting anchor nodes, the storage overhead for keeping the query scopes is effectively reduced while not incurring significant extra traversal cost. We also implemented the grid-based materialization for comparison purpose.

4.2 Query Scope Index

To alleviate the affected path detection, we store the query scopes identified based on Definition 2 in the *query scope index* (in short, the *QSI*). Let $Q = \{q\}$ denote a set of continuous min-cost path queries registered in the PathMon system. QSI provides a reversed one-to-many mapping between queries and nodes covered by their query scopes, i.e., $m : N \rightarrow Q$. This mapping facilitates a quick lookup of affected queries from end nodes of updated edges.

In the all-node based query scope materialization, this mapping can be implemented by simply marking every node that is visited in a search process with the query ID. However, due to the high storage space overhead and update cost, we further consider the anchor node materialization. Since the path search algorithm generates a min-cost path spanning tree covering all nodes inside the query scope, the anchor node assignment can be naturally integrated with the search process with a minor computational overhead. The basic idea is to evenly spread a query (and its registered information) to nodes within its query scope based on their min-cost path to the destination node. A system parameter α is a preset number of maximum traversal steps between a new anchor node and its preceding anchor node on the same path of the spanning tree, so that it can control the number of anchor nodes. Thus, a node is assigned as an anchor node if the number of nodes on its min-cost path to the destination is a multiply of α. If α is small, more covered nodes will be assigned as anchor nodes, which improves search efficiency by consuming more storage. Otherwise, if α is large, storage overhead is reduced with a deterioration of search efficiency.

Fig 7 illustrates the anchor node selection and affected path search process. Here the min-cost path from s to d, along with the min-cost path spanning tree rooted at d, are shown in bold lines. Assume $\alpha = 2$, and the dark nodes are selected as anchor nodes. The nodes in grey are non-anchors but covered by the query scope,

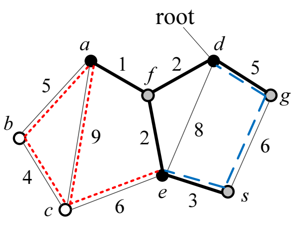


Figure 7: Illustration of partial materialization

while other white nodes lay outside the query scope (and their distance to d is greater than $||s, d||_{t_0}$). When an edge cost $C_{\langle c,b\rangle}$ changes, an expansion is performed from both c and b to visit the nodes that are reachable within $\alpha - 1$ hops (indicated by the dotted, red lines). During the expansion from c, anchor nodes e and a are found, and $||c, d||$ can be calculated by $||c, d|| = \min\{||c, e|| + ||e, d||, ||c, a|| + ||a, d||\}$. Similarly, for b, we have $||b, d|| = ||b, a|| + ||a, d||$. The calculation results show that $||b, d|| > ||s, d||$ and $||c, d|| > ||s, d||$, so we know that the edge $\langle c, b\rangle$ has no end node inside the query scope, and this edge update could not possibly change the min-cost path. On the other hand, consider another update of edge $\langle s, g\rangle$, since d is an anchor node visited in the expansion of $\alpha - 1$ hops from g (indicated by the dashed, blue lines), $||g, d|| = ||g, d|| + ||d, d|| < ||s, d||$, we know that the edge $\langle s, g\rangle$ has at least one end node covered by the query scope. The query $q(s, d)$ is then identified as an affected query.

In our implementation of PathMon server, QSI is realized by two sets of lists $\{QSI.\text{qlist}(n)|n \in N\}$ and $\{QSI.\text{anchors}(q)|\ q \in Q\}$, where $QSI.\text{qlist}(n)$ is a list of query IDs for an anchor node n and $QSI.\text{anchors}(q)$ is a list of anchor node IDs for a query q. Figure 8 illustrates the two basic data structures of QSI.

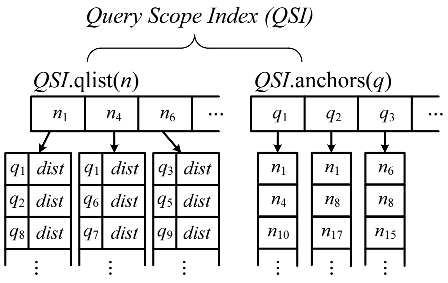


Figure 8: The query scope index structure

With $QSI.\text{qlist}(n)$ we can easily retrieve the queries that are mapped with a node n, and with $QSI.\text{anchors}(q)$ we can quickly obtain all anchor nodes that maintain information regarding to a query q. QSI is updated upon registration of a new query, as listed in Algorithm 1. This is integrated with the path search algorithm, which iteratively examines nodes in non-descending order of the network distance to the destination d of the new query q. The search starts from the destination d. A node n positioned at x hops away (along the min-cost path) from d is an anchor node if x modulo $\alpha = 0$ (line 6). The position of d is 0 (thus d is always an anchor node). The algorithm for finding queries affected by an edge $\langle n, n'\rangle$ is shown in Algorithm 2. It performs a traversal started from the end nodes of updated edges for at most $\alpha - 1$ nodes apart, while collecting queries that are associated with the traversed anchor nodes.

When a query q^* is terminated, we first obtain all anchor nodes of q^* from $QSI.\text{anchors}(q^*)$, and for each anchor node m, remove the entry of q^* from $QSI.\text{qlist}(m)$. After that, the entire list of $QSI.\text{anchors}(q^*)$ is deleted.

5. PARTIAL PATH RE-COMPUTATION

In Section 3.1, we have already presented the main idea of the

Algorithm 1 RegisterQuery(QSI,$q(s,d)$, t_0)

```
1: Visited node set $V \leftarrow \emptyset$
2: heap $H$.insert($d, \perp, 0, 0$)
3: while $H \neq \emptyset$ do
4:     $(n, pre_n, c_n, x_n) \leftarrow H$.getMin()
5:     if $n \in V$ then
6:         break
7:     end if
8:     if $x_n$ modulo $\alpha = 0$ then
9:         $QSI$.qlist($n$).add($q, c_n$) {assign $n$ as an anchor node}
10:        $QSI$.anchors($q$).add($n$) {map $n$ with $q$}
11:    end if
12:    $V$.insert($n$) {$||n,d|| = x_n$}
13:    if $n = s$ then break
14:    for each $n'$ such that $\langle n', n\rangle \in E$ do
15:        if $n' \notin V$ then
16:            $H$.insert($(n', n, c_n + C_{\langle n',n\rangle,t_0}, x_n + 1)$)
17:        end if
18:    end for
19: end while
```

Algorithm 2 SearchQuery(QSI,$\langle n, n'\rangle$)

```
1: $SQ \leftarrow \emptyset$ {$SQ$ is the set of affected queries}
2: Expand from $n$ and $n'$ for $\alpha - 1$ hops using Dijkstra's algorithm
3: for each visited node $m$ do
4:     if $QSI$.qlist($m$) $\neq \emptyset$ then
5:         for each $q \in QSI$.qlist($m$) do
6:             if $||n,m||_{t_1} + ||m,d_q||_{t_0} <= ||s_q,d_q||_{t_0}$ or $||n',m||_{t_1} + ||m,d_q||_{t_0} <= ||s_q,d_q||_{t_0}$ then
7:                 $SQ$.insert($q$)
8:             end if
9:         end for
10:    end if
11: end for
12: return $SQ$
```

path update algorithm, i.e. path update can start at a certain processing state of the previous search and safely terminate at another processing state existing in the previous search after incorporating all updates in the query scope range. As such, part of the path recomputation mandated for affected path queries can be eliminated. This section mainly focuses on the issues of implementing and optimizing the path re-computation algorithm.

The key idea of our partial path re-computation is to reuse the processing states explored in previous search processes. For this purpose, in each path search process, we select a set of *checkpoint nodes* at which the processing states are stored. Since it is expensive to maintain the processing states V^n, H^n for all visited nodes, we select a few nodes as the checkpoint nodes. Below we present five checkpoint node selection strategies. The performances of these strategies will be studied in Section 6.

1. Random selection (RAND). Randomly pick a given number of nodes as the checkpoint.
2. Associated with Anchor nodes (ANCH). Each anchor node also is responsible for carrying its corresponding processing state. Thus the restart point can be found along with the path invalidation process.
3. Fixed step interval (FSI). Store a processing state every fixed number of steps in the search. Thus the restarting processing state is guaranteed to be within a pre-set number of steps from visiting the updated edge.
4. Fixed cost interval (FCI). The checkpoint nodes are evenly distributed according to the distance to the destination.
5. Adaptive-to-updates (ATU). Initially, use one of the above strategies to select the checkpoint nodes. As the network updates happen, dynamically add checkpoints near the frequent-updated areas and remove the checkpoints where updates rarely occur. If the update distribution is skewed in the network, i.e. a few "hot-spots" are constantly updated while other places remain almost static, this approach is expected to achieve better performance.

Accordingly, we extend QSI to QSIU (namely, *QSI for Update*) in order to facilitate a quick lookup of preserved state information when affected queries are detected. Thus, while QSI is mainly developed for path invalidation, QSIU also supports efficient path updates by the Partial Path Computation Algorithm (PPCA). In addition to QSI, a *processing state repository* $QSIU$.procs(q, n) is introduced. Given a query q and a node n, $QSIU$.procs(q, n) returns the processing state in the path search process for q at node n. If n is not a checkpoint node of q, a NULL value is returned. Meanwhile, we maintain a list $QSIU$.check(q) to record all the checkpoint nodes for the query q to support query removal.

A processing state $S(n)$ at a checkpoint node n consists of 1) a visited node set V^n, and 2) a priority queue H^n. Since there are multiple queries registered in QSIU, we use a subscript q to indicate that $S_q(n)$ is the processing state of query q at node n, i.e. $S_q(n) = (V^n_q, H^n_q)$. $S_q(n)$ is available in the path search process of query q. When a node is selected as a checkpoint node, we maintain the visited node set and the priority queue of current state by adding $S_q(n)$ into $QSIU$.procs(q, n).

QSIU is not only used for identifying the affected paths, but also collecting important information in the search process for use in PPCA, i.e., the *affecting range*. For a query $q(s, d)$, the affecting range is the range of the distances from the updated edges located within the query scope to the destination. Let the affecting range be denoted as $[X_q, Y_q]$, $0 \leq X_q \leq Y_q$. All the updated edges $\langle n, n'\rangle$ in the network satisfy that $X_q \leq ||n,d|| \leq Y_q \vee X_q \leq ||n',d|| \leq Y_q$, or $||n,d|| > ||s,d|| \vee ||n',d|| > ||s,d||$. X_q provides the initialization information to PPCA, and Y_q is used to check whether all affecting updated edges have been visited and thus the re-computation algorithm may terminate. Particularly, in our algorithm we set X_q equal to the nearest distance from an updated edge to the destination, and the upper bound Y_q of the possible (updated) affecting range is determined as

$$Y_q = \max_{\Delta C_{\langle n,n'\rangle} \neq 0} \{\max\{||n,d||, ||n',d||\}\} + \sum_{\Delta C_{\langle n,n'\rangle} > 0} \Delta C_{\langle n,n'\rangle}$$

in which each $\langle n, n'\rangle$ is an affecting edge for q. Here, we conservatively incorporate all edge cost increments into the upper bound estimation. We will discuss the usage of the affecting range $[X_q, Y_q]$ and the starting node later. Algorithm 3 lists the query identification and re-computation information collecting process with QSIU. Given a set of network updates, the algorithm returns all affected query IDs and their corresponding $[X_q, Y_q]$.

It is noticeable that in Algorithm 3, the cost c_n computed from $||n,a|| + ||a,d_q||$ may deviate from the actual $||n,d_q||$ at time t_1 (after the network change) if a is not on the min-cost path from n to d_q or $||a,d_q||$ itself is updated. However, the correctness of the algorithm can still be guaranteed because the smallest $||n,d||$ value must be consistent with the current network state. This is quite clear with Lemma 2 and 3.

PPCA initializes the re-computation with a "stable" processing state that does not change upon edge updates, in order to eliminate the steps of path computation from destination node (the very beginning state) to this state. According to Lemma 3, this initial state is positioned in maximum number of steps from the destination node, without being affected by edge changes. Thereafter, the algorithm traverses the changed portion of road network, factoring in the updates to obtain new processing states for affected nodes, until either of the following termination conditions is satisfied: (1) *early termination* - after all the changed edges within the query scope are visited, there is a processing state at a certain node remaining the same as it was in the previous search algorithm; (2) *normal*

Algorithm 3 SearchQuery($QSIU$, t_1)

```
1: $SQ \leftarrow \emptyset$
2: for all $q$, $X_q \leftarrow \infty, Y_q \leftarrow 0, \Delta Y_q \leftarrow 0$
3: for all $\langle n, n' \rangle$ that is updated do
4:     $\Delta C \leftarrow C_{\langle n,n' \rangle, t_1} - C_{\langle n,n' \rangle, t_0}$
5:     Expand $\langle n, n' \rangle$ for $\alpha - 1$ hops using Dijkstra's algorithm
6:     for each anchor node $a$ visited in the expansion do
7:         for each $q \in QSIU$.qlist($a$) do
8:             $c_n \leftarrow ||n, a|| + ||a, d_q||$
9:             $c_{n'} \leftarrow ||n', a|| + ||a, d_q||$
10:            if $c_n <= ||s_q, d_q||_{t_0}$ or $c_{n'} <= ||s_q, d_q||_{t_0}$ then
11:                $SQ$.insert($q$)
12:                if $\Delta C > 0$ then $\Delta Y_q \leftarrow \Delta C + \Delta Y_q$
13:                if $c_n < X_q$ or $c_{n'} < X_q$ then $X_q \leftarrow \min\{c_n, c_{n'}\}$
14:                if $c_n > Y_q$ or $c_{n'} > Y_q$ then $Y_q \leftarrow \max\{c_n, c_{n'}\}$
15:            end if
16:        end for
17:    end for
18: end for
19: for all $q$ do $Y_q \leftarrow Y_q + \Delta Y_q$
20: return $SQ$,$\{[X_q, Y_q]\}$
```

termination - the source node s_t is visited. If the algorithm meets condition (1), based on Lemma 4, the rest of states in the search process remains unchanged and thus can be skipped. If condition (2) is meet, the algorithm terminates naturally.

With the information collected in QSIU search algorithm, the re-computation algorithm PPCA is performed in four steps, elaborated below. Algorithm 4 lists the pseudo code for this update process.

1. Given the lower bound of affecting range X_q, find the latest stable processing state $S_q(m)$ ($||m, d|| < X_q$), initialize the search algorithm with processing state $S_q(m)$ (lines 1-7).
2. Re-run the path search algorithm from the initialized state to update the min-cost path. In this step, update of the QSIU is performed seamlessly. If a node n is selected as a checkpoint node, the algorithm associates the new processing state $S'_q(n)$ to n by inserting new entries (if n was not a checkpoint node of q) or updating existing entries (if n was originally a checkpoint node of q) in $QSIU$.procs(n) (lines 17-19). If a previously selected checkpoint n is no longer a checkpoint node in the new search, its associated processing state expires and is removed from $QSIU$.procs(n) (lines 20-22). The anchor nodes of q are also updated with similar operations (lines 13-16).
3. If s is visited, the search terminates anyway. Otherwise, after all updated edges affecting q have been explored (we may check this by comparing the current expansion distance with Y_q), the path re-computation terminates if there exists an anchor node m' such that $S_q(m') \equiv S'_q(m')$ (m' is then called the termination node) (lines 24-25).
4. Finally, starting from s, trace back the precedences to construct the entire path (line 30).

While retaining the same methodology, an *incremental representation* of processing states is adopted in our PPCA implementation to further reduce the storage overhead, i.e. we differentiate the current processing state (V^n, H^n) from what has been stored at the last checkpoint node (V^m, H^m), and only keep the difference $(\Delta V^n, \Delta H^n)$. When (V^n, H^n) is to be retrieved from the processing state repository of QSIU, the incremental processing states $\{(\Delta V^m, \Delta H^m)\}$ corresponding to all the checkpoint nodes m visited before n are retrieved and then merged with $(\Delta V^n, \Delta H^n)$ to recover the complete (V^n, H^n).

6. EXPERIMENTAL EVALUATION

In this section, we evaluate the proposed QSI and PPCA, i.e., the core components of PathMon, via an extensive set of experiments by simulation. Besides, we compare our approaches with the

Algorithm 4 UpdatePath($QSIU$,$q(s_t, d)$, st_q, X_q, Y_q)

```
1: $Dist_{max} \leftarrow \infty, S_{start} \leftarrow$ NULL
2: for all $m \in QSIU$.check($q$) do
3:     if $||m, d|| < X_q$ AND $||m, d|| > Dist_{max}$ then
4:         $S_{start} \leftarrow QSIU$.procs($q, m$)
5:         $Dist_{max} \leftarrow ||m, d||$
6:     end if
7: end for
8: Visited node set $V \leftarrow S_{start}.V$, priority queue $H \leftarrow S_{start}.H$
9: while $H \neq \emptyset$ do
10:    $(n, pre_n, c_n, x_n) \leftarrow H$.getMin()
11:    if $n \notin V$ then
12:        $V$.insert($n, pre_n, c_n$)
13:        if $x_n$ modulo $\alpha = 0$ then
14:            $QSIU$.anchors($q$).add($n$)
15:            $QSIU$.qlist($n$).add($q$)
16:        end if
17:        if $n$ is selected as a checkpoint then
18:            $QSIU$.procs($q, n$).insert($V, H$) {add a new processing state}
19:            $QSIU$.check($q$).add($n$)
20:        else if $QSIU$.procs($q, n$) <> NULL then {$n$ is an expired checkpoint}
21:            $QSIU$.procs.remove($q, n$) {remove an obsolete processing state}
22:            $QSIU$.check($q$).remove($n$)
23:        end if
24:        if ($n = s_t$) OR ($c_n > Y_q$ AND $S_q(n) \equiv QSIU$.procs($q, n$)) then
25:            break
26:        end if
27:        for each $n'$ such that $\langle n', n \rangle \in E$ do
28:            $H$.insert(($n', n, c_n + C_{\langle n',n \rangle}, x_n + 1$))
29:        end for
30:    end if
31: end while
32: Trace back from $s_t$ to construct the path $P(s_t, d)$
```

most related approach, namely, EBM (reviewed in Section 2, and a naive approach that reevaluates all path queries upon any network change. In what follows, we first describe the experiment setups. Then, we examine the effectiveness of QSI and different checkpoint selection heuristics. Finally, we evaluate the overall performance of PathMon and compare it with EBM and the naive approach.

6.1 Experiment Setup

We implemented the aforementioned algorithms with GNU C++. For QSI (denoted by QSI), we implemented both anchor node materialization and grid based materialization and compare their effectiveness in terms of the ratio of identified affected queries. For PPCA and QSIU (denoted by QSIU+PPCA), we only used anchor node materialization. For anchor node materialization, we defaulted α to 10. For EBM (denoted by EBM), an elliptic affecting area is formulated for every min-cost path. Here, we assume V_{max} (see Section 2) to be 1.5. The naive approach (labeled as Naive) reevaluates all path queries when updates occur in the network. We use a grid file with 50×50 grid cells to support grid based materialization and as a grid index used by EBM. Here, except for PPCA that recomputes path partially, all others, including QSI, EBM and Naive, use Dijkstra's shortest path algorithm and/or A* algorithm to compute paths. Here, A* is applicable for situations where edge costs are related to geometrical distance.

We ran the simulation for each setting for 30 times on Linux 2.6.9 servers with Intel Xeon 3.2GHz CPU and 4GB RAM and report the average results. In each setting, we issued 1,000 continuous min-cost path queries. Source nodes of the queries are randomly picked, whereas destination nodes are selected about 0.15 through 0.35 of the network diameter away from respective source nodes. We summarize all the experiment parameters in Table 9.

6.2 Effectiveness of Query Scope Representations

We expect the representation of query scopes to have an impact

Parameter	Value	Default
Network	CA (21,048 nodes, 21,693 edges), NY (46,254 nodes, 63,552 edges), WM (61,333 nodes, 80,589 edges)	CA
No. of queries	1,000	1,000
No. of changed roads ($\lvert E_u \rvert$)	0.03%, 0.05%, 0.08%, 0.1%	0.05%
Length of paths (L_p)	0.15, 0.20, 0.25, 0.30, 0.35 of network diameter	0.25
Path search alg.	Dijkstra's, A*	Dijkstra's

Figure 9: The evaluation parameters

on performance of path monitoring system. In the naive approach, the query scope contains the entire network, while in EBM, the query scope is an ellipse approximated by grid cells. Our QSI and QSIU+PPCA are based on the "network connectivity" and thus consist of nodes covered by the query scope. In the first experiment, we compare the path invalidation ability of these approaches in terms of number of affected paths identified.

The result is plot in Figure 10. As shown, Naive incurs all path recomputed (i.e., 100%) due to blind path reevaluation. EBM saves some path re-computation for those edges not covered by the elliptic area. QSI, exploiting query scopes to precisely determine non-affecting edges, has a significantly lower re-computation ratio. QSIU+PPCA uses the same path invalidation method as QSI so they have the identical re-computation ratio. Meanwhile, the grid-based representation of QSI (denoted as Grid) has less false hits than EBM because it is more precise. Comparing with QSI, however, some redundant re-computations still exist in Grid.

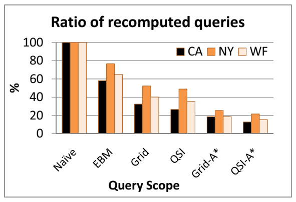
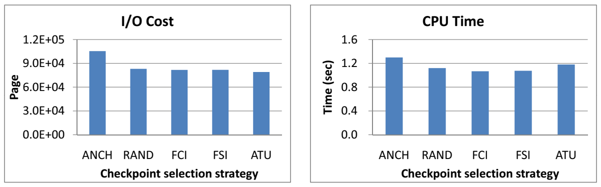


Figure 10: Re-computation ratio

In the scenario where certain geometric assumptions are applicable in the path cost estimation, path search algorithms utilizing the estimation heuristics can achieve a much smaller query scope size. For example, the QSI-A* and Grid-A* curves represent the ratio of recomputed queries identified by the query scope built based on the A* algorithm. While Grid-A$*$ incurs more re-computations than QSI-A*, they both have quite low ratio compared with the ones based on Dijkstra's algorithm. This is because, in the heuristic-based algorithms, both the connectivity and physical constraints of the road network are considered, resulting in effective elimination of the false hits.

6.3 Effectiveness of Checkpoint Selection

In QSIU+PPCA, five checkpoint placement strategies have been presented, namely, random (RAND), anchor-node-association (ANCH), fixed-step-interval (FSI), fixed-cost-interval (FCI) and adaptive-to-update (ATU). Here, we study the effectiveness of these strategies.



(a) I/O Cost (b) CPU Time

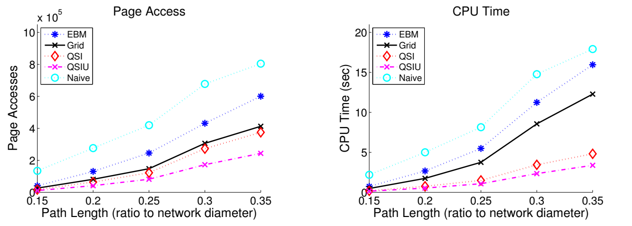
Figure 11: Checkpoint selection strategy comparison (CA)

In this set of experiments, the total number of checkpoints selected by the strategies is set to $|N|/100$ where $|N|$ is the number of nodes in the network. The results are plotted in Fig 11. In terms of I/O cost, RAND, FSI and FCI incur almost the same number of page accesses, because the checkpoint selected by these strategies tend to be evenly distributed in the network. The running times of these three strategies are also very close. Meanwhile, ATU achieves slightly better I/O efficiency due to adaptively selecting checkpoints that are close to the places where updates happen frequently. Nevertheless, due to the extra computational costs needed to re-allocate checkpoints, ATU takes longer CPU time to perform the re-computation. Last, ANCH incurs both the most I/O cost and CPU time among all five strategies, indicating that the anchor node assignment is not suitable for selecting checkpoints.

6.4 Path Monitoring Performance

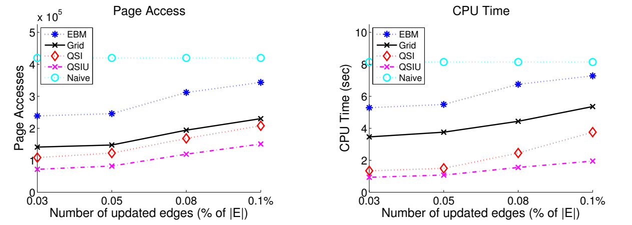
Next, we evaluate PathMon and compare its performance against other techniques in terms of CPU time and I/O costs, the two common metrics considered for system performance. CPU time measures the time the system spends to update path queries. I/O cost measures the number of disk pages accessed to detect/update affected path queries.

Impact of path length. The current min-cost path length directly affects the size of the query scope. Here, we evaluate its impact on the performance of examined techniques by varying the path length from 0.15 to 0.35 with a step of 0.05 of the network diameter. The results are reported in Figure 12. As the path length increases, the ratio of recomputed queries, processing time and I/O cost of all approaches increase accordingly. Due to effective elimination of redundant path update, QSIU+PPCA reduces both I/O cost and CPU time as shown in Figure 12(a) and Figure 12(b), respectively. Also, we can see that Naive is the worst, while QSIU+PPCA significantly outperforms EBM. With fewer paths recomputed, QSI performs better than EBM but still incurs about 50% more I/O cost and CPU time than QSIU+PPCA.



(a) I/O cost vs. path length (b) CPU time vs. path length

Figure 12: Performance under various path length



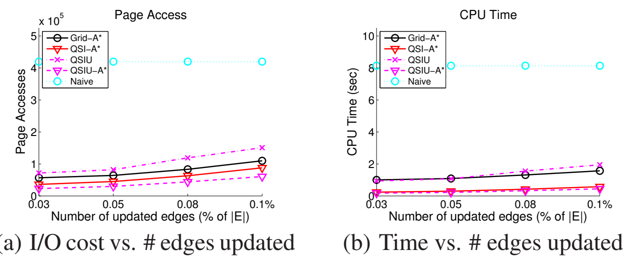
(a) I/O cost vs. # edges updated (b) Time vs. # edges updated

Figure 13: Performance under various # of updated edges

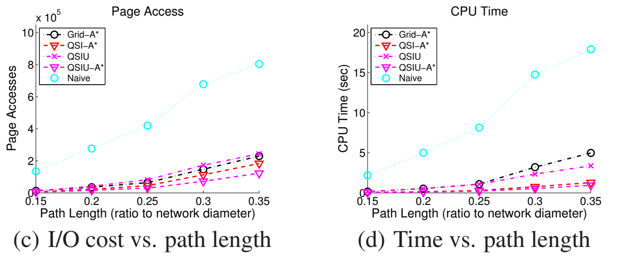
Impact of number of updated edges. If a network is highly dynamic, a large number of edges would be updated simultaneously. In this experiment, we model the degree of network dynamics by varying the number of updated edges up to 0.01% of the total number of edges. Please note that the query processing time is very short for our proposed algorithm (i.e. a few seconds for QSIU+PPCA), so the approach is capable to deal with high update frequency.

Again, I/O cost and CPU time are significantly saved by QSIU+PPCA as shown in Figure 13(a) and Figure 13(b), respectively. On the other hand, this experiment shows a trend that both performance of EBM and QSIU+PPCA degrade as the number of updated edges increase. Naive will eventually prevail, which indicates a room for future research in PathMon.

Impact of path search algorithm. The path search algorithm can significantly affect the query scope, and potentially lead to more efficient path monitoring. Here we show that, besides Dijkstra's algorithm, our proposed algorithms can be applied to other best-first path search algorithms (e.g., A* algorithm) and also achieve superior performance.



(a) I/O cost vs. # edges updated (b) Time vs. # edges updated

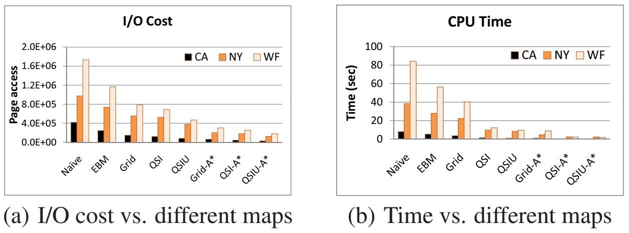


(c) I/O cost vs. path length (d) Time vs. path length

Figure 14: Performance under A* algorithm

We use Naive and QSIU as the baseline in our comparison. As shown in Fig 14, when A* algorithm is applied, both the I/O and time performance improve significantly for different query scope representations (i.e. the grid-based and the anchor node-based representation). Furthermore, we can see that the QSIU-A* approach outperforms QSI-A* and Grid-A*.

Evaluation on various maps. To validate the efficiency of our approach, a set of experiments on various cities has been conducted. We report the results in Fig 15. For all algorithms, as the network size increases, both I/O cost and CPU time grow. QSIU is superior among all Dijkstra's algorithm based approaches, and QSIU-A* performs the best in the A* algorithm based approaches.



(a) I/O cost vs. different maps (b) Time vs. different maps

Figure 15: Performance for various cities ($L_p = 0.25D_N$, $|E_u| = 0.05\%|E|$)

7. CONCLUSION

In this paper we addressed the continuous min-cost path monitoring problem, motivated by a great demand of LBS applications requiring min-cost paths in dynamic road networks. We studied two research issues of the problem, namely, path invalidation and path update, and developed novel techniques based on our problem analysis and our understanding of the min-cost path search properties. Based on the result of our analysis, we introduced an index structure called query scope index (QSI) to record query scopes, which consist of nodes in the min-cost path spanning trees and in turns cover all affecting edges. QSI is efficient to identify affected path queries. Also, we devised a partial path re-computation algorithm (PPCA) to partially recompute a new min-cost path when the old min-cost path becomes invalid, by resuming a search at certain processing states. To facilitate PPCA, another index called QSIU that extends QSI is devised to maintain processing states. Our PathMon system seamlessly integrates QSI, QSIU and PPCA. PathMon is a centralized system to support monitoring of massive continuous min-cost path queries. Upon network changes, the PathMon server identifies the queries affected by the changes from QSI/QSIU, and quickly recomputes the new min-cost paths using PPCA. Through comprehensive experiments, we examined the effectiveness of different designs for QSI and PPCA. Additionally, we studied the impact of expected query path length, network update rates and network size to the performance of the PathMon system. The experiment results demonstrate that the new techniques proposed in this paper outperform the representative approaches, in terms of both I/O cost and processing time. Currently we are building a PathMon system prototype. Besides, observing that some path queries may specify similar destinations, we are examining the feasibility of grouping multiple path queries in path monitoring as the next step of this work.

8. ACKNOWLEDGEMENT

This research was supported in part by the National Science Foundation under Grant no. IIS-0328881, IIS-0534343 and CNS-0626709.

9. REFERENCES

[1] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[2] D. P. Bertsekas. A Simple and Fast Label Correcting Algorithm for Shortest Paths. *Networks*, 23:703–709, 1993.

[3] E. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerical Mathematics*, 1959.

[4] R. W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6), 1962.

[5] A. Segali. Distributed network protocols. *IEEE Transaction on Information Theory*, 29, 1983.

[6] R. Seidel. On the All-Pairs-Shortest-Path Problem. In *STOC'92*, pages 745–749, 1992.

[7] A. O. *et al.* Minimum Delay Routing in Stochastic Networks. *IEEE/ACM Transaction on Networking*, 1(2), 1993.

[8] B. D. *et al.* Finding Time-Dependent Shortest Paths over Large Graphs. In *EDBT'08*, pages 205–216.

[9] C.-C. L. *et al.* Continuous Evaluation of Fastest Path Queries on Road Networks. In *SSTD'07*, pages 20–37, 2007.

[10] D. F. *et al.* Fully Dynamic Output Bounded Single Source Shortest Path Problem. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 212–221, 1996.

[11] E. K. *et al.* Finding Fastest Paths on A Road Network with Speed Patterns. In *ICDE'06*.

[12] H. G. *et al.* Adaptive Fastest Path Computation on a Road Network: A Traffic Mining Approach. In *VLDB'07*, pages 794–805.

[13] J. M. M. *et al.* The new routing algorithm for the ARPANet. *IEEE Transaction on Communications*, 28(5):711–719, 1980.

[14] N. J. *et al.* Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation. *TKDE*, 10(3):409–432, 1998.

[15] P. N. *et al.* New Dynamic Algorithms for Shortest Path Tree Computation. *IEEE/ACM Transaction on Networking*, 8(6):734–746, 2000.

[16] R. D. *et al.* Generalized Best-First Search Strategies and the Optimality of A*. *Journal of ACM*, 32(3):505–536, 1985.

[17] R. H. M. *et al.* Partitioning Graphs to Speedup Dijkstraąŕs Algorithm. In *ACM Journal of Experimental Algorithmics*, volume 11, pages 1–29, 2006.

[18] S. J. *et al.* An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *TKDE*, 14(5):1029–1046, 2002.

[19] S. S. *et al.* CCAM: A connectivity-clustered access method for networks and network computations. *Transaction on Knowledge and Data Engineering*, 9(1):102–119, 1997.