Realtime Response of Shortest Path Computation

Lin Qi & Markus Schneider
University of Florida
Department of Computer & Information Science & Engineering
Gainesville, Florida, USA
{lqi, mschneid}@cise.ufl.edu

ABSTRACT

Computing the shortest path between two locations in a network is an important and fundamental problem that finds applications in a wide range of fields. This problem has attracted considerable research interest and led to a plethora of algorithms. However, existing approaches have two main drawbacks: complete path computation before movement and re-processing when node failure occurs. In this paper, two novel algorithms, *RSP* (*Realtime Shortest Path*) and *RSP+* (*Realtime Shortest Path Plus*), are proposed to handle both shortcomings. We perform a network pre-processing to ensure a constant time response of retrieving the shortest route for an arbitrary node to an important set of destinations. *RSP+* further divides the complete path into smaller partial paths, which can then be computed in parallel. Besides, considering the continuous changes of the network, like traffic jams and road constructions, where certain paths are blocked, a fast recovery method to efficiently find the best alternative route is integrated into *RSP+*. Empirical studies have shown that *RSP+* can achieve an average query processing time of 0.8 microseconds. Besides, the effectiveness of the recovery mechanism demonstrates that alternative routes can be obtained to avoid unavailable areas.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications – *data mining, spatial databases and GIS*

General Terms

Design, Algorithm, Performance

Keywords

Shortest path, road network, pre-processing, node failure

1. INTRODUCTION

The shortest path problem is one of the fundamental problems of graph theory with many practical applications on a wide spectrum of graph types such as road networks and social networks [6]. Finding a solution to this problem is a basic and critical step in GIS network analysis, such as urban traffic planning, optimal pipelining, robot navigation, and especially transportation.

Classical approaches like Dijkstra [4] and Bellman-Ford [3] algorithms traverse the nodes in the graph with an ascending order of their distances from a source to a destination and terminate once the destination has been reached. They are often inefficient for sizeable road networks. To improve the efficiency of shortest path processing, a number of algorithms that exploit the characteristics of road networks have been proposed [9, 11, 12]. They are based on the observation that shortest paths with source nodes close to each other will have an overlapping path if the destinations are geometrically adjacent. Methods described in [1, 2, 5, 8, 10, 13] rest on the property that some nodes or edges are more important for long distance travel (e.g. highways) and pre-compute all-pair shortest paths to these nodes.

A key observation from the related literature is that the shortest path between a source and a destination is computed completely, in which all nodes on the route have to be determined prior to any movement. In many cases, it would be preferable that a partial path that belongs to the optimal shortest path can be returned timely. For example, when using navigation products, it would be more satisfying to know which turn should be made immediately in front of an intersection. Moreover, considering that a road network is not be static all the time, in which traffic jam, accidents, or road constructions (called *node failures*) could occur. As a result, some nodes or paths may be blocked which will invalidate existing pre-processing results. Thus, a time-consuming reprocessing has to be performed.

In this paper, we propose a novel algorithm, *RSP*, which performs like a target-oriented depth first search. First, we apply a pre-processing step, in which we record all pairs of shortest distances for an arbitrary node to a set of important nodes in respect of network structure or user interests. Given a source node, by checking information of accumulative distance to the destination on all adjacent edges, the algorithm returns the edge with smallest value to guide the movement. After that, the algorithm moves to the node along the chosen edge and marks this edge as visited, then considers all adjacent edges of the new node and terminates once the destination is reached. The time complexity of the algorithm depends on the average degree in the network. Given that the degree of a road network is usually a small integer, this query can be executed in constant time.

ACM SIGSPATIAL IWCTS'14, November 4, 2012. Dallas, TX, USA

Considering the huge size of road networks, e.g. the total road length of United States is 6,586,610 km with about 24 million nodes and 58 million edges, the scalability of the algorithm should be taken into account. In order to further speed up the search and to take advantage of parallel computing, an improved algorithm, *RSP+*, is introduced. The key idea is to *cut* a whole shortest path into small segments, and return them in order. Multi-thread technologies are applied to the whole computation process as those partial paths can be computed in parallel. Empirical studies exhibit an average response at a scale of fraction of microseconds to shortest path queries. This outperforms Sanders [10] by two orders of magnitude and appears to be the fastest approach referring to the evaluation results [15].

In general, the main contributions of this paper include:

- An innovative and lossless spatial mapping from the original network to our spatial network model in a preprocessing step.
- a distance table assigned to both directions of each edge, which enables constant lookup time.
- two novel and parallel friendly algorithms to provide realtime response.
- a fast-recovery mechanism to quickly adjust to temporary node failures and layout changes without performing pre-processing each time.

The rest of the paper is organized as follows: Section 2 discusses about related work regarding shortest path computation. In Section 3 we will introduce the pre-processing method for the network. Our spatial network model as well as the structure of the distance table will also be described. The details of our shortest path algorithms are explained in Section 4. Various experiments are then performed and presented in Section 5. Finally, Section 6 concludes this paper and proposes potential working directions.

2. RELATED WORK

Over the past two decades, a plethora of techniques have been proposed to address the deficiency of Dijkstra's algorithm by exploiting the characteristics of road networks [15]. These research efforts have produced a number of shortest path algorithms as well as extensive empirical findings regarding the computational performance. Algorithms described in [2, 8, 13] are built upon the observation that certain vertices in a road network are more important in path queries. Methods in [5, 8, 10, 13] perform a pre-processing of the network by ordering the vertices according to their importance and pre-compute the shortest paths among these important vertices to accelerate query processing. However, a small change in the network layout, e.g. a road maintenance or a traffic light construction, may require the preprocessing to be performed again on the whole network. As *RSP+* offers a quick-recovery mechanism that could provide a feasible path even if maintenances happen in the network without additional efforts of pre-processing. Algorithms described in [1, 2] share some similarities to our approach: they introduce a concept of *transit nodes* where all pairwise shortest distances are precomputed among those nodes to help speed up non-local shortest path queries. Such transit nodes are a special case in our approach, where we could set important destinations as those transit nodes. In addition, we provide more flexibility that users could customize such important destinations based on their interests.

There exists some other research focuses including network hierarchy analysis, network characteristic analysis, etc. Some representative methods include *RE* [6], *Arc Flags* [7], etc. Among these methods, ALT preprocesses the road network by first selecting a small set of vertices, called the *landmarks*, and then it pre-computes the distance from each vertex to each landmark. This concept shares similarity to what we call *Important Points* (IPs) in our algorithm. But we further cluster those interested destinations into multiple levels to reduce the query processing time for the reason that if one directed arc does not have access to a higher scope, there is no need to examine the lower levels of IPs. In addition, researches in [16] provide an evaluation of particular predefined route from recorded GPS traces and authors in [14] estimate travel time by leverageing historical data archive.

3. NETWORK PRE-PROCESSING

In this section, we will introduce the network pre-processing, which consists of two steps: spatial network generation and distance information assignment.

3.1 Spatial Network Model

In an actual road network $G = (V, E, w)$ with a vertex set V, an edge set E and a weight function. Each edge $e \in E$ is associated with a weight $w(e) : e \to (R)$.

For a dual-directed edge, there are two travel directions. Generally, the accumulative travel cost to the same destination is not identical for two directions, thus we need to attach the distance information to the appropriate directed edge. We call such directed edges as *directed arcs*. An example of network generation is shown in Figure 1. One edge from the original graph can be mapped to either one (for a one-way edge (w, v)) or two directed arcs (for a two-way edge (u, v)). Let $SNet$ be the newly generated spatial network, with A as the set of directed arc, and N as the set of nodes. Therefore, the spatial network generation step can be defined as the following spatial mapping ξ:

$$\xi : G(V, E, w) \to SNet(N, A, w), where$$
$$(i)\ \forall \vec{a} \in A, \exists^* \vec{e} \in E, s.t.\ \vec{e} = \vec{a}\ ||\ \vec{e} = \vec{a}^T$$
$$(ii)\ \forall \vec{e} \in E, |\xi(\vec{e})| \leq 2$$
$$(iii)\ \forall n \in N, \exists^* v \in V, s.t.\ n = v$$
$$(iv)\ \forall \vec{e}, \vec{a}\ in\ (i), w(\vec{a}) \leftarrow w(\vec{e})$$
$$(v)\ Given\ v \in V, \forall \vec{e} \in C(v),$$
$$\exists \vec{a} \in C(\xi(n))\ where\ \vec{e} = \xi^{-1}(\vec{a})$$

where symbol * means uniqueness and ^{T} represents the transpose form. $C(n)$ represents the set of edges or arcs connected to n. Condition (i) and (ii) ensure that for each edge, there can be at most two directed arcs correspondingly. Similarly, condition (iii) restricts the mapping of each vertex to one node. The weight function remains the same after projection for the same edge as shown in (iv), however, we could also allow asymmetric travel cost easily by modifying the new weight function as per need. Condition (v) retains the connectivity and topology.

3.2 Distance Information Assignment

Storing all-pair distance information is a heavy burden and is not scalable when network becomes huge. However, there exist certain points in the network which are more important. For example, transportation hubs usually appear

in shortest paths, whereas places like travel spots and hotels are more of users' interest. In essence our approach will identify a small set of such Important Points (IPs), for example, people trend to travel to a certain set of destinations, and these destinations are of more importance and are also queried more frequently. Further, a cumulative distance to every IP is propagated and stored in each directed arc. For example, as in Figure 1, arc (w', v') has a total distance of 200 to node u' while arc (x', w') has a cumulative distance of 300 to u' by adding the length of itself. As a result, at query stage a simple lookup yields the exact distance from an arbitrary source node to any of the IPs, thus providing a realtime response.

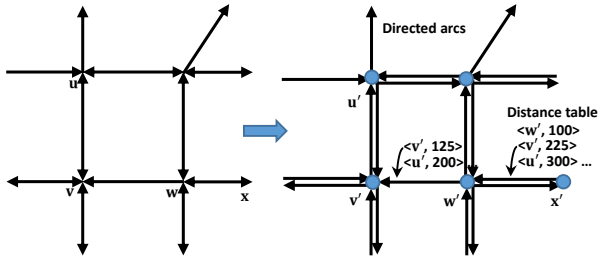

Figure 1: Mapping to spatial network from original network

A set of vertices are pre-defined for IPs, where users have the flexibility to add or remove destinations that they are interested in or not. The increment of the set of IPs will trigger a round of distance assignment, but only restricted to the newly added destinations. The form of distance information w.r.t. each IP is a key-value pair where the key represents the id of a certain IP and the corresponding value yields the shortest distance to this IP. Such distance information is attached to each directed arc, and each arc can hold multiple key-value pairs for all feasible IPs, formulating a *distance table*. To provide constant lookup time, a hash table is employed to represent distance table structure.

Time and Space Complexity. The assignment of distance information starts from all IPs. For each IP, for arcs directly connected to that point will have the distance value as their edge length. And then the distance value propagates in a back-tracking manner. The distance values are gradually increased by the length of corresponding directed arc as we travel along the network. Finally as soon as all directed arcs are spanned, the assignment is finished. Overall, this pre-processing takes $O(k * (|V| + |E|))$ where k is the number of IPs. After the assignment, based on the distance information stored at each directed arc, the pairwise shortest paths from an arbitrary node to this set of nodes are then established.

Space consumption of our approach is highly correlated to the number of IPs and total number of directed arcs. And for a huge network, the number of IPs could also be large. Considering the fact that shortest paths in road networks are often spatially coherent, i.e. the shortest paths to the destinations which are geometrically close to each other will share an overlapping path in long distance travels, we will cluster our IPs in a hierarchical structure based on their geographical location. For example, there could be 1000 hotels in Orlando city and one of them is our destination. But when we are driving from Atlanta, we only need the distance to Orlando city which is an estimate to our destination and by the time we enter the city area, detailed distance information to the hotels is then provided. In this case, for all edges representing the highways outside Orlando, we only need to store the distance to Orlando city itself, which is a higher-level IP. Furthermore, a multiple-layer clustering structure could be provided and adjustable to the network scale. In the U.S. road network, with 24 million of nodes and 58 million of edges, if 240,000 IPs (1 % of total nodes) are picked, in worst case where all IPs are accessible from all edges, it will consume around 1 MB per edge. With three layers of clustering at a ratio of 100, the space consumption will be reduced to 4 bytes per edge.

4. RSP AND RSP+ ALGORITHMS

After the network pre-processing, in this section, we will introduce our shortest path algorithms. We will start with *RSP* (Realtime Shortest Path), to show the steps how the shortest path is computed with distance information. After that, an improved version *RSP+* is then developed to fulfil the needs of providing realtime response. Both of the algorithms are performed in a manner of depth first search. But with the target as the destination, this search is not expanded arbitrarily but with a target orientation.

4.1 RSP

As shown in Algorithm 1, initially we start with the directed arc that roots at S with smallest distance value w.r.t. D in line 1. Next, we create a list of visitedArcs that stores the directed arcs that we have visited. This is to avoid traveling in loops. Each time we perform a lookup to pick the directed arc that is closest to the destination and add to visitedArcs. This is shown in line 4 to 11. We repeat this searching process until the destination is reached.

Algorithm 1: *RSP* Algorithm

Input: $SNet(N, A, w)$, Source S, Destination node D
Output: Shortest path from S to D

```
1  Pick the arc roots at S with smallest distance to D and
   assign to currArc;
2  Path FinalPath ← null;
3  while currArc ≠ null do
4      if visitedArcs.Contains(currArc) then
5          continue;
6      visitedArcs.Add(currArc);
7      if currArc.Head == Destination D then
8          break;
9      FinalPath.Add(currArc);
10     currArc ← the outgoing arc of currArc with
       smallest distance to D;
11 return FinalPath;
```

THEOREM 4.1. ***(RSP's Correctness)*** *Given T as a spatial network, if there exists a path from the source node S to the destination node D in T, RSP will compute and return the shortest one.*

PROOF. Given destination D, and the outgoing arcs of source $\{outArcs^S\}$, if $\forall arc \in \{outArcs^S\}$, D does not appear in the distance table of arc, there will be no possible path to D from S, and *RSP* will terminate.

Next, we need to prove that the path *RSP* returns, Φ, is the shortest one. Assume that there exists a different path Φ' which is the shortest one. Φ is consist of a list of directed arcs $< a_1, a_2, ..., a_n >$, and similarly for Φ', the list is $< a'_1, a'_2, ..., a'_n >$. Since Φ and Φ' is different, there must be a starting point $q \in [1, min(m, n)]$ where $a_q \neq a'_q$ and for $i \in [1, q]$: $a_i \neq a'_i$. Since the distance to D along the directed arcs in a path is strictly monotone decreasing, the path that has steeper slope would be the shortest one. From line 15 in Algorithm 1, we always pick the outgoing arc which has the smallest distance to D, i.e. closest to D. Therefore the distance to D from a_q will be less or equal to the distance from a'_q, in other words, we always pick the global minimum directed arc that can lead us to the destination. In this case, if path Φ' chooses a'_q, it picks a detour compared to a_q which will not give us the shortest path. This is contradictory to our assumption that Φ' is the shortest path. Therefore, *RSP* is guaranteed to return the shortest one. □

Algorithm 2: *RSP+* Algorithm

Input: $SNet(N, A, w)$, Source node S, Destination node D, Search diameter δ
Output: Partial path along the shortest path

```
1  Start one thread: Pick the arc roots at S with
   smallest distance to D and assign to currArc;
2  Path PartialPath ← null;
3  int partialPathLength ← 0;
4  while currArc ≠ null do
5      if visitedArcs.Contains(currArc) then
6          continue;
7      visitedArcs.Add(currArc);
8      if currArc.Head == Destination D then
9          return PartialPath;
10     else
11         if partialPathLength ≥ δ then
12             PartialPath ← GeneratePartialPath(S,
               currArc);
13             Start another thread: Recursively call
               RSP+ algorithm to compute the partial
               paths until we hit D with updated source
               node and search diameter;
14             return PartialPath;
15         PartialPath.Add(currArc);
16         currArc ← the outgoing arc of currArc with
           smallest distance to D;
17         partialPathLength ←
           partialPathLength + currArc.length;
```

4.2 RSP+

In a transportation application, for example a navigation system, it would be more satisfying that we could get the very first direction guidance with realtime response especially when the network is huge and when source and destination are located far apart. Therefore, our idea is to *cut* the complete shortest path into *partial paths* and return them in sequence.

As shown in Algorithm 2, we have introduced a parameter named *search diameter*. It is a threshold that indicates the extension boundary of the partial path, while we extend the path to a certain length which exceeds this threshold, we will cut the complete shortest path to this point and return the partial path. This serves as another exit criteria of the algorithm in line 12 to 15. In addition, while returning the partial path, another thread is initiated with updated source node to this current point and perform *RSP+* algorithm recursively to return a sequence of partial paths till the final destination is reached. Note that this can be done in background execution, but in front end, user will get the very first partial path as the result. Sub-procedure *GeneratePartialPath*(S, *currArc*) is a method that generates a path from precedent stored for each directed arc and trace back until we hit the source node. This process is similar to Algorithm 1 but with different terminating criteria. It is omitted here due to space limit.

As an improved algorithm based on *RSP*, *RSP+* enjoys the aforementioned features that *RSP* holds. Additionally, as the computation of partial paths are parallel friendly, we could feed the algorithm recursively with updated source node. In general, the time complexity of both algorithms depend on the average number of edges connected to each node in the network. However, such node degree is usually a small constant especially in road networks. In worst case, time complexity is still bounded $O(|E| + |V|)$ for both algorithms, when all edges and nodes are visited. But again, as the benefit of partial path concept provided in *RSP+*, from a user's perspective, he will be getting the response in constant time and rest of the computation can be done in parallel. It is noted that *RSP+* has a flexibility based on the size of search diameter δ. If we set δ to ∞, *RSP+* will turn into *RSP*. And by adjusting the values of δ, we have a strong control of how long we are expecting our partial path. Theoretically it is expected that with a smaller δ, we will achieve lower latencies. This is proved in the experiments in Section 5.

4.3 Fast Recovery Mechanism and Safe Route Generation

The algorithms described above assure us the shortest path from source to destination. However, in the underlying networks, node failures can happen from time to time. A node failure temporarily puts this node down from the network so that no movement unit could travel through this node. A typical example could be a construction site in road networks. In existing works, with such layout change, a redo of pre-processing of the whole network is required. However, to avoid such a time-consuming process from happening too often, we propose a fast-recovery heuristic of route regeneration, named *SafeRoute*, which is integrated in our *RSP+*.

It is possible that the failure node M has no impact in this path query from S to D. Therefore in line 1, we first check whether M appears in the shortest path from S to D. If not, we directly return the shortest path from S to D as the safe route. Otherwise we first construct a connected sub-network around M. This sub-routine will start from node M, and insert M's neighbors into the sub-network, including incoming and outgoing arcs. We continue extending from the neighbor nodes to their neighbors also until the generated sub-network is fully connected (i.e. there exist possible routes for all-pair nodes). We call M as a central node. It is possible that we cannot find such a network, because that M is located in a crucial connecting location of the network so that without it, the network will be isolated.

Algorithm 3: SafeRoute Algorithm

Input: $SNet(V, A, w)$, Source node S, Destination node D, Failure node M
Output: An alternative path excluding midpoint M

```
1  if AppearInPath(M, RSP+(S,D)) then
2      SNet T = ConstructSubNetwork(M, i);
3      if T == null then
4          return null;
5      Redo distance assignment within network T;
6      Path Φ1 ← Shortest path from S to the closest
       node B^in of the boundaries of T;
7      Choose node B^out in the boundaries with the
       smallest distance to D;
8      Path Φ2 ← RSP+(B^in,B^out);
9      Path Φ3 ← RSP+(B^out,D);
10     return Path Φ ← Combine(Φ1, Φ2, Φ3);
11 else
12     return RSP+(S,D);
```

In this case we will return null as the safe route. After the generation of the new sub-network, then we assign distance information within this new sub-network T, note that M is not included in this network thus it will not be considered in the distance assignment. This assignment is to ensure that every boundary node of T will store its shortest distance to all other boundaries.

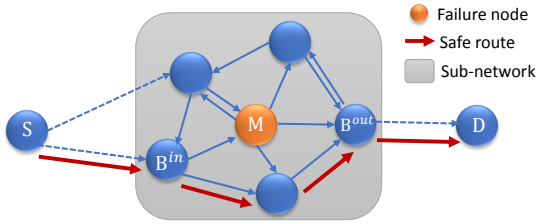

Figure 2: Illustration of SafeRoute algorithm of finding alternative route

Next, we pick the node from the boundaries which is closest to source S, we denote this node as B^{in}. Similarly, we also pick the node from the boundaries that is closest to destination D, named as B^{out}. Since M is in between of these nodes, it is certain that B^{in} is different from B^{out}. Finally our safe route from S to D is consist of three sub-paths: shortest path from S to B^{in}, from B^{in} to B^{out}, and from B^{out} to S in line 6 to 10. This combined path will be proved in Theorem 4.2 that it would not travel through node M, and therefore is a feasible path. Figure 2 gives an illustration example.

THEOREM 4.2. **(Safe Path Feasibility.)** *Given a spatial network and a path query from S to D, if there exists an alternative path w.r.t. the failure node M which lies on the shortest path from S to D, then SafeRoute algorithm will return one feasible path excluding M.*

PROOF. After construction of a new sub-network around node M, from one node S outside of this network, if there exists a shortest path from S to B^{in}, we must hit node B^{in} before we can come to node M as B^{in} is the precedent of M. Similarly, there will exist a shortest path from B^{in} to another boundary node B^{out} without traversing through M. By combining the paths $S \rightarrow B^{in}$, $B^{in} \rightarrow B^{out}$, and $B^{out} \rightarrow M$, a feasible route from S to D excluding M has been found. □

5. EXPERIMENTS

In this section, we will design and perform three sets of experiments regarding shortest paths and feasible routes. The results show that our algorithm is much more efficient than classical approaches. In case of random node failures in the network, our algorithm will return an alternative and feasible route avoiding unavailable areas.

5.1 Experimental Setup

The algorithms are implemented in Microsoft's Visual Studio framework, and they use common subroutines for similar tasks. We have conducted experiments on a computer running Windows 7 with an Intel i7-4770 3.4 GHz CPU and 16 GB RAM with stable working state. An 800 km of real-world rail network from Australia near Hedland is taken in our experiments. The network is an undirected graph consists of 26,140 nodes and 45,252 edges.

5.2 Performance Evaluation

The first set of queries is multiple-source shortest paths from a variety of locations to one fixed destination. We randomly pick 10, 50, 250, 1250 nodes respectively as the source nodes. In this experiment, the destination is fixed to a pre-defined IP. The total response time of different path search algorithms are shown in Figure 3.

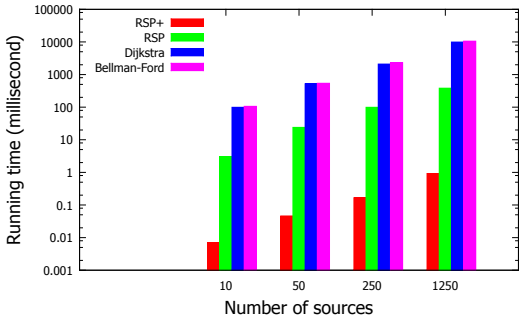

Figure 3: running time of different algorithms with increasing number of sources

We compare the efficiency for shortest path queries of four different algorithms: *RSP+*, *RSP*, Dijkstra's algorithm, and Bellman-Ford algorithm. Figure 3 shows that *RSP+* and *RSP* significantly outperform Dijkstra's algorithm and Bellman-Ford algorithm. Dijkstra performs slightly better than Bellman-Ford, and *RSP* leads the performance by a factor of 30. In addition, *RSP+*, dramatically reduces the searching time by five orders of magnitude. On average, each shortest path query is answered in 0.8 microsecond with *RSP+*. It is a realtime response resulting from a partial path concept and parallel computing technique. The search diameter δ is set to 20,000 meters as default value in *RSP+*.

Next, we evaluate the influence of changing δ values. Recall that δ indicates the search boundary of the algorithm, i.e. the maximum length of the partial path. As expected,

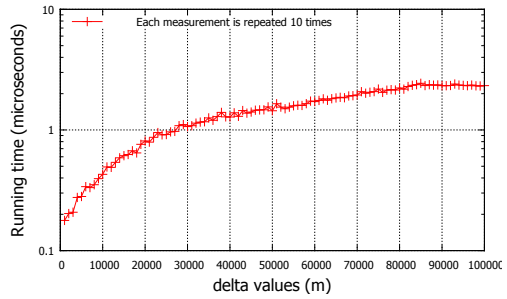

Figure 4: running time of *RSP+* over different δ values

shown in Figure 4, with the increasing of δ value, the running time also follows an increasing manner. We measure the running time with a step of 1 km increase starting from 1 km till 100 km. Interestingly, we observe a steeper slope of increasing before δ reaches around 20,000. After that, the running time increases with a steady but lower trend. This is because that when δ is small, we need more partial paths to formulate the shortest path, thus each partial path is relatively small. It will take less time to generate the first partial path. When δ becomes greater than around 90,000, the running time stops increasing but rather oscillating around 2.333 ms. This indicates that our δ has reached a peak that further increasing of this value will have negligible impact on the running time. The reason is that this δ value has covered the whole length of the complete shortest path. In this case, *RSP+* turns into *RSP* algorithm.

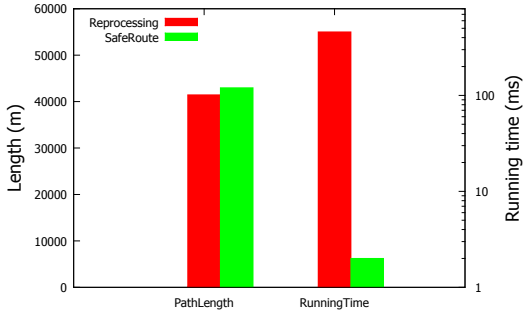

Figure 5: SafeRoute evaluation based on path length and running time

In the last set of experiments, we evaluate the effectiveness of *SafeRoute* algorithm described in Algorithm 3. From Figure 5, we could see that the algorithm could return us an alternative route with 43550 meters. After redo the pre-processing, we successfully find the shortest path with length 41440 meters, which is 4.8% less. However, the pre-processing takes around 480 ms, while directly running *SafeRoute* algorithm will only take 2 ms, which is 99.58% less. This result shows that the route that *SafeRoute* returns may not be the overall shortest path in the network, but it takes much less time to compute. The algorithm will be extremely practical in a realtime responsive system.

6. CONCLUSIONS AND FUTURE WORK

In this paper, two novel shortest path computation algorithms are introduced. They could provide constant time response of retrieving shortest path for an arbitrary node to an important set of destinations. The algorithms are designed to be applicable to all kinds of networks by allowing users to add or remove interested destinations to the pool based on their interests. In future, we will be conducting more experiments on various networks, especially large-scale networks.

7. REFERENCES

[1] H. Bast, S. Funke, and D. Matijevic. Transit: ultrafast shortest-path queries with linear-time preprocessing. In *Proc. of the 9th DIMACS Implementation Challenge*, pages 175–192, 2006.

[2] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[3] R. E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[4] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[5] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[6] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *ALENEX*, pages 129–143, 2006.

[7] M. Hilger, R. H. Möhring E. Köhler, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *Proc. of the 9th DIMACS Implementation Challenge*, pages 73–92, 2006.

[8] M. Rice and V. J. Tsotras. Graph indexing of road networks for shortest pth queries with label restrictions. In *PVLDB*, volume 4, pages 69–80, 2010.

[9] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.

[10] P. Sanders and D. Schultes. Engineering highway hierarchies. In *14th European Symposium on Algorithms (ESA'06)*, pages 804–816, 2006.

[11] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.

[12] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. Knowl. Data Eng*, 22(8):1158–1175, 2010.

[13] D. Schultes. *Route Planning in Road Networks*. PhD thesis, University Karlsruhe (TH), 2008.

[14] Y. Wang, Y. Zheng, and Y. Xue. Travel time estimation of a path using sparse trajectories. In *Proceeding of the 20th SIGKDD Conf. on Knowledge Discovery and Data Mining*, volume 2, 2014.

[15] L. Wu, X. Xiao., D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. In *PVLDB*, volume 5, pages 406–417, 2012.

[16] B. Yang, C. Guo, and C. S. Jensen. Travel cost inference from sparse, spatio temporally correlated time series using markov models. In *The 39th International Conference on Very Large Data Bases (VLDB)*, volume 2, 2013.