Efficient Best Path Monitoring in Road Networks For Instant Local Traffic Information

Shuo Shang **Ke Deng** **Kai Zheng**

School of Information Technology and Electrical Engineering
the University of Queensland
St.Lucia, Brisbane, QLD 4072, Australia
Email: {shangs,dengke,kevinzheng}@itee.uq.edu.au

Abstract

The shortest path problem has been well studied previously. To improve the utility, the traffic conditions can be modeled to associate a weight to each road segment. The recent trend is to apply data mining techniques over use history which usually covers a long period of time such as months. However, this method fails to reflect the instant (i.e. temporary) traffic conditions change such as traffic accident or road work. Due to the temporary nature, the local instant traffic conditions makes more sense when an object is moving in road networks. In this work, we investigate the shortest path monitoring problem while the instant traffic conditions in local region update repeatedly around a moving object to a given destination. A simple way is to apply A* algorithm repeatedly. However, the weakness is obvious. Because only a small fraction, i.e. the local area, of the whole networks have changed and the other parts keep intact. That means, for many vertices, that their paths (or the lower bounds of their paths) to the destination are still valid. This motivates us to maintain these information and reuse in the following computations. Our method is based on two tree structures where one records the previous computing results and the other aims to reduce the search space of subsequent processing. The experiments over real data set demonstrate an improvement of processing efficiency by one degree of magnitude at a small memory cost. In addition, the tree can be shared when monitoring the shortest paths for several moving objects to the same destination.

Keywords: Shortest Path Monitoring, Instant Local Traffic Information

1 Introduction

The shortest path problem has been well studied previously. A good survey of shortest path methods has been made by Pallottino et al.(1997) and Fu et al.(2006). While the shortest path may need to be recomputed if the object deviates from the planned path for some reasons, it also happens due to the traffic conditions. More recently, the traffic conditions of road networks have been considered in order to accurately capture the real shortest path, such as speed patterns, driving patterns and a multitude of other factors (proposed by Gonzalez et al. 2007) that provide important information in the computation of desirable routes. To do that, each road segment is assigned a weight which is derived from the use history of massive local users. However, the historic traffic conditions sometimes may be unreliable, because the traffic conditions of road networks in the real life often change temporarily because of reasons such as traffic accidents and road works. Such kinds of information usually cannot be captured by the method weighting road segments based on historic traffic conditions.

One possible solution is to reflect the instant (i.e. temporary) changes in the global weighted road networks. However, it is not sensible to do so due to the following reason. The temporary changes of traffic conditions in road networks quite often, if not always, impact the proximity only. The users far away are not interested in these temporary impacts since these impacts change even when these users plan to travel these areas in the future. What these users need is the instant traffic conditions when they are close to these areas. Thus, the principle of processing instant traffic conditions is that more attention is paid on quantitative details of close region and on qualitative information of remote region. In this work, we studied the problem of the real-time shortest path monitoring due to the update of instant local traffic conditions.

The real-time shortest path planning monitoring is critical in various applications. While it is important in real life such as for route planning of rescue vehicle in emergency, we emphasize the scenarios in the virtual world such as traffic simulation system involving tens of thousands of agents (i.e. cars). Nowadays, more and more online racing games simulate the road networks and traffic conditions of real city such as *Midtown Madness* [1] and the car racing in *Second Life* game platform [2], and they usually attract millions to play online simultaneously. In a typical scenario, several online players race from the same source to the same destination. The instant local traffic situation around each car is the *non-player character* (NPC) which is controlled by the gamemaster in the games. The real-time shortest path needs to be updated to guide the tour and the quick response is essential to the success of such online race games.

In this work, we aims to efficiently monitor the shortest path when local traffic conditions change. Since the changes happen arbitrarily during the traveling, the shortest path is the best possible path at current local traffic conditions and thus they are potentially best traveling choice. But users can chose to follow or not. In any case, the instant suggestion as the response to the traffic changes is valuable to users. To do that, the basic idea in this work is to compute the shortest path once and effectively reuse the results in the following monitoring. Initially, the classic A* algorithms (proposed by Hart et al. 1968) are applied. This results are recorded using two tree-like data structures. While one tree is for recording the previous computing results, the purpose of the other tree is to reduce the search space of subsequent processing. At any instance when the latest local traffic conditions is known, the methods are developed to update the traverse trees and identify the new shortest path efficiently. The experiments over real data set demonstrate an improvement of processing efficiency by one degree of magnitude at a small cost

[1]www.microsoft.com/games/midtown/

[2]heidiballinger.wordpress.com/2008/07/19/car-racing-in-second-life/

of storage. The tree can be shared when monitoring the shortest paths for several moving objects (e.g. cars in the online games) to the same destination. Note that the proposed method is also applicable to provide instant shortest path suggestion to users who deviate the planned paths at his own discretion.

The shortest path monitoring is also known as the dynamic shortest path planning (DSP) (Frigioni et al.1996, 1998, 2000 and King 1999 and Demetrescu et al. 2004). In these works, the underlying networks with constant edge weights is allowed to be updated from time to time. The updates include insertion/deletion of edges and edge-weight updates, and these updates are known globally. Frigioni et al. (1996) studied the single-source DSP problem where the shortest path is maintained from a given source to all other vertices. Note that the single-source DSP is same to our problem only if no local instant traffic conditions is considered. King (1999) investigated the all-pair DSP problem where the shortest paths for all pairs of vertices are concerned. The all-pair DSP problem may deal with our problem but the local nature of this problem implies that it is not sensible to use the techniques for the all-pair DSP. Two steps are taken in all-pair DSP. The first step is to compute the shortest paths among all pairs using well known Floyd-Warshall algorithm in $O(|V|^3)$ or Johnson's algorithm for sparse network in $O(|V|^2 \lg |V| + |V||E|)$ where $|V|, |E|$ are the numbers of vertices and edges in the network separately (Cormen et al. 2001). Then, the maintaining algorithm is applied when the weights of the edges change. The most recent algorithm (Demetrescu et al. 2004) is in $O(|V|^2 \lg^3 |V|)$ amortized time per update. To solve our problem, all-pair DSP are much higher even comparing to direct applying Dijkstra's algorithm per update $O(|V| \lg |V| + |E|)$. Since heuristic algorithm A* is no worse than Dijkstra's algorithm, we compares our method with simply applying A* per update. Besides, our problem also differs from the time-dependent shortest path problem (Ding et al. 2008) where the job is to find the best departure time.

The contributions of this work are in three folds. First, this is the first effort to investigate the impact of instant local traffic conditions to the shortest path planning. Second, we proposed two tree structures to support the path monitoring to the single destination. The trees can be shared among several user if they have the same destination. Third, the proposed method is applicable to shortest path updating due to deviation of the planned path.

The remainder of the paper is organized as follows. In section II, problem description is presented and we introduce the related work in section III. Our approach including traverse trees, bounds and searching algorithm are introduced separately in section IV. Then, we discuss the memory cost and the traverse tree sharing among multiple moving objects in section V. The experiment results on real data set are demonstrated and explained in section VI. This paper is concluded in section VII.

2 Problem Description

In this paper, we use road networks to illustrate spatial networks. A road network can be modeled as a graph $G = (E, V)$, where V is a set of nodes corresponding to road junctions and E is a set of (non-directional) edges between two nodes in V corresponding to road segments. According to the traffic conditions, each road segment is associated with a non-negative weight. An edge can be a straight line or a poly line. If there is an edge in E linking two nodes in V, these two nodes are adjacent to each other. Let $Adj(v)$ denote all the adjacent nodes of $v \in V$. Let $d(v, v')$ be the distance along a path between two points v and v' (i.e. the total length of the edges along a network path between the two nodes). If there is no path connecting v and v', $d(v, v') = \infty$. The network distance

Table 1: **A list of notations**

Notation	Description
V	The set of all nodes
E	The set of all edges
o	The moving object,the central of range A
A	The circular region of instant local traffic conditions
r	The radii of range A
weight (n_1, n_2)	The weight of edge $(n_1; n_2)$
$d(v, v')$	the distance along a path between two points v and v'
$Adj(v)$	all the adjacent nodes of $v \in V$
(n_1, n_2)	The edge between n_1 and n_2, or the path from n_1 to n_2 if in a clear context
$SP(n_1, n_2), \|SP(n_1, n_2)\|$	The shortest path between n_1 and n_2 and its length
$d_N(n_1, n_2)$	The network distance between n_1 and n_2
$d_E(n_1, n_2)$	The Euclidean distance between n_1 and n_2
$lb(v) and v.lb$	the lower bound of node v
$ub(sp) and SP.ub$	the upper bound of shortest path from s to t
$Temp\_path$	the shortest one of all known access paths from s to t

between v and v', denoted as $d_N(v, v')$, is defined using the network shortest path between the two nodes, denoted as $SP(v, v')$. In addition, we use $d_E(v, v')$ to denote their Euclidean distance. The problem of this work can be described as follows:

Given a road network $G = (E, V)$ and two points $s, t \in V$, an object o moves from s to t. An initial shortest path $SP(s, t)$ is computed by A* algorithm. At any instance, o is informed about the instant local traffic conditions in the surrounding region A, i.e. the weights of the road segments in A are updated. Without loss of generality, the local region A is represented as a circular region with radius r around the current location of o. The problem is to update $SP(s, t)$ to $SP'(s, t)$ on G with the updated local traffic conditions. SP' can be same as SP or not. Table 1 shows a list of notations used in this paper.

3 Related work

Two representative network shortest path algorithms are the Dijkstra's algorithm (proposed by Dijkstra.1959) and the A* algorithm(proposed by Hart et al.1968). They both propagate a search "wavefront" from a source node v_s until a destination node v_d is reached. A heap H is used to keep all the nodes in the wavefront. The initial step of Dijkstra's algorithm is to put every node $v \in Adj(v_s)$, together with the distance $d(v_s, v)$ (set to the length of the edge linking v_s and v), into H. Then, the algorithm iterates an expansion process by replacing a node $v \in H$, where $d(v_s, v)$ is the minimum for all nodes in H, by all the nodes in $Adj(v)$. For each node $v' \in Adj(v)$, $d(v_s, v')$ is set to $d(v_s, v) + d(v, v')$ if v' is not yet in H or (in case v' in H) the existing estimate of $d(v_s, v')$ is larger than $d(v_s, v) + d(v, v')$. This process terminates when v_d is selected from H to expand (and $d_N(v_s, v_d) = d(v_s, v_d)$). Dijkstra's algorithm can compute the shortest paths from a source node to multiple destination nodes. If there is only one destination, the wavefront expansion process can be optimized towards the direction of the destination node. This is the key idea of A*. Instead of selecting $v \in H$ with the minimum $d(v_s, v)$, a node $v' \in H$ with the minimum $d(v_s, v') + d_E(v', v_d)$ is selected to expand[3]. That is, when selecting a node v', not only the computed network distance from v_s to v' is considered, the Euclidean distance from v' to v_d is also used as a directional guide. For any node $v \in H$, $d(v_s, v) + d_E(v, v_d)$ is called the *distance lower bound* of v from v_s to v_d (denoted as $lb(v, v_s, v_d)$, or $v.lb$ when not causing ambiguity). Clearly, any valid

[3] $d(v_s, v') = d_N(v_s, v')$ when v' is selected to expand.

network path from v_s to v_d via v cannot be shorter than $v.lb$.

Gonzalez et al.(2007) stresses the importance of road hierarchy, speed pattern and driving pattern in the path planning. The speed pattern means that the speed may change at different time and road segments due to the traffic congestion or road condition. Therefore, to compute the fastest path exactly, the speed of each road segment should be dynamic. The speed pattern can be extracted from the use history of local person through the data mining techniques. The basic idea is that the choices of local people should always be the best, because they must have enough reasons to choose this way, such as security, road conditions and so on. The traffic pattern mining has also been studied by Agrawal et al.(1995), Han et al.(2000), Pei et al.(2001), Kanoulas et al.(2006). However, there is still a notably weakness of using historic traffic data. That is, the road conditions are changing continuously, the speed patterns based on mining the historic traffic conditions sometimes would be unreliable. They may not reflect the real conditions of road network and some temporarily changing of road conditions, such as road construction or traffic accident, will lead to serious mistake in route planning.

The dynamic shortest-path (DSP) problem (Frigioni et al.1996, 1998, 2000 and King 1999 and Demetrescu et al. 2004) is to recompute the shortest-path repeatedly while the underneath graph with constant edge weights is allowed to be updated from time to time. The updates include insertion/deletion of edges and edge-weight updates. Frigioni et al.(1996) study the single-source DSP problem and the fully dynamic algorithms are proposed with optimal space requirements and processing time. Experimental evaluations for single-source DSP algorithms can be found in Frigioni et al.(1996). Our problem can be viewed as a single-destination DSP problem which is different from the single-source problem due to the consideration of instant local traffic conditions. King (1999) investigates the all-pair DSP problem and Demetrescu et al.(2004) improves all-pair DSP algorithm by giving the optimal worst-case time. Two steps are taken in all-pair DSP. The first step is to compute the shortest paths among all pairs using well known Floyd-Warshall algorithm in $O(|V|^3)$ or Johnson's algorithm for sparse network in $O(|V|^2 \lg |V| + |V||E|)$ (where $|V|, |E|$ are the numbers of vertices and edges in the network separately). The second step is to update the all impacted shortest paths when the weights of the edges change. The most recent algorithm (Demetrescu et al. 2004) is in $O(|V|^2 \lg^3 |V|)$ amortized time per update. The all-pair DSP problem may deal with our problem but the local nature of this problem implies that it is not sensible to use the techniques for the all-pair DSP.

The time-dependent shortest path problem, proposed by Ding et al.(2008), investigates to find the best departure time such that the total travel time can be minimized over a road network, where the traffic conditions change from time to time.

4 Our Approach

Once user receives the instant local traffic conditions, the system will recompute the shortest path from the current position to the destination. To do that, a simple way is to apply A* algorithm repeatedly. However, the weakness is obvious. Because only a small fraction, i.e. the local area, of the whole graph have changed and the other parts keep intact. That means, for many vertices, that their shortest paths (or the lower bounds of their shortest paths) to the destination are still valid. This motivates us to maintain these information and reuse them in the following computations.

We proposed an algorithm called Instant Shortest Path Monitoring (IBPM). Initially, the shortest path from the source to the destination is computed using A* algorithm. This results are recorded using two tree structures, called Lower Bound Tree F and Upper Bound Tree T. When the local instant traffic conditions is updated, two steps are taken to update the shortest path. First, the nodes in the traverse trees are updated for the corresponding road segments. Second, the traverse trees are searched and the search space is constrained by the lower bound and upper bound provided by the traverse trees. In the path update, the principle of A* algorithm is applied to guarantee the correctness. The details of tree structures and the techniques for update and search are introduced in details next.

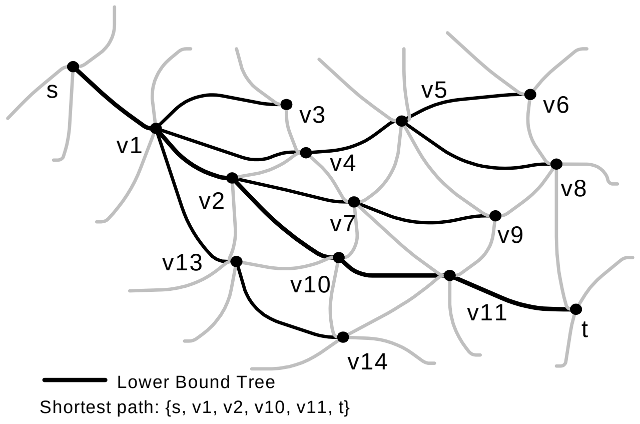


Figure 1: Once Searching Result

4.1 Two Tree Structures

Two traverse trees are established to help the shortest path monitoring. They are upper bound tree and lower bound tree. These trees are constructed when the initial shortest path is computed using A* algorithm and kept updating until the moving object arrives the destination. Each node in traverse trees corresponds a real vertex in road networks. At any instance, one vertex only appears once in upper/lower bound tree. The upper bound and lower bound are key points of optimizations in our method.

For the lower bound tree, the initial tree roots at the source s, see an example in figure.1. Each non-leaf node v is attached with a value which is the lower bound from current location (i.e. s initially) to t, denoted as $lb(v)$ or $v.lb$. In order to keep the size of the lower bound tree small, each vertex only corresponds to one node even though one vertex may be accessed through different paths (i.e. through different neighbor vertices). As a result, the path from each node to the root corresponds to the shortest path in the network. In the following process, since the weight of the edge changes, the attached lower bounds of nodes in the tree may change and consequently the tree structure may need to be updated. Some nodes will be removed from F and the lower bound tree may turn into a forest. One may encounter a problem during the tree updating. For a vertex v, its shortest path to source (i.e. the root in the tree) may go through another neighbor vertex due to the change of the weights of the networks. An example is shown in figure 2. The shortest path from source to v goes through v_1 initially and then changes to go through v_2. In this situation, two operations are taken. First, the predecessor of v is updated to v_2 and the link between v_1 and v is removed. Second, v_1 is set to be a leaf node in the tree by removing all other child nodes (explained later).

For the upper bound tree, it roots at the destination t. Initially, the computed shortest path using A* algorithm is recorded, see the shortest path in figure.1 . Each time when the shortest path is updated, the new path is

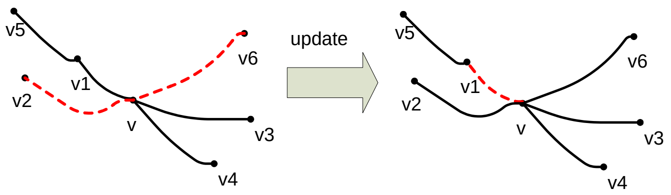


Figure 2: The Updating of Traverse Trees

recorded in the tree as well. Similar to lower bound tree, we keep one node for one vertex only in the upper bound tree. Thus, a node may appear in two shortest paths to t and thus it may have two predecessor nodes to the destination (i.e. root in the upper bound tree). The same processing applied as to lower bound tree. In the upper bound tree, the recorded shortest paths are called *access paths* since they are possible channels from the source to the destination. Among all access paths, the one with the minimum length is called $Temp\_path$. $Temp\_path$ is current known shortest path. It can be used as the upper bound of the shortest path from s to t (in other words, $|Temp\_path| \geq |SP(s,t)|$), denoted $|Temp\_path|$ as $ub(sp(s,t))$.

We have discussed the method to update lower and upper trees. Now, we focus on lower and upper bound calculation. The lower bound is discussed in this section and the upper bound will be introduced in next section. The lower bound of a node v in the lower bound tree $lb(v)$ is an estimated (or *heuristic*) distance from v to the destination t. A* algorithm usually uses the Euclidean distance between v and t, $d_E(v,t)$ as estimated distance of $|SP(v,t)|$. According to A* algorithm, the greater estimated distance leads to a smaller search space and better performance efficiency, and it must not be greater than $|SP(v,t)|$ (otherwise it may lead to a wrong result). The lower bound attached with v calculated from the lower bound tree is no less than $d_E(v,t)$ and no greater than $|SP(v,t)|$. That is,

$$|SP(v,t)| \geq lb(v) \geq d_E(v,t) \qquad (1)$$

In our method, the lower bound calculation is from leaf to root. Firstly, we set the lower bound of each leaf node v to be the Euclidean distance between v and t, $d_E(v,t)$. Then, we calculate the lower bound of other nodes based on their child nodes. In case one node v_1 has more than one child node and the lower bound calculated from different child nodes are different, we choose the smallest one as the lower bound of v_1. The details of how to calculate the lower bound of nodes in lower bound tree is introduced in Algorithm.1.

Algorithm 1: Lower Bound Calculation

```
  Data: Lower Bound Tree $F$
  Result: $lb(v)$ of each node
1 $\forall v \in F, lb(v) \leftarrow \infty$; for all leaf node $v$ in $F$ do
2     if $NeighbourNumber(v) = 1$ then
3         $lb(v) \leftarrow \infty$;
4     else
5         $lb(v) \leftarrow d_E(v,t)$;
6     if
      $lb(v.predecessor) > lb(v) + weight(v, v.predecessor)$
      then
7         $lb(v.predecessor) \leftarrow$
          $lb(v) + weight(v, v.predecessor)$;
8     $v \leftarrow v.predecessor$;
```

In Algorithm 1, all lower bounds are calculated from bottom to top. The leaf nodes have two different styles. If v has only one neighbor, its predecessor, which means v is at the end of a broken path and we set $lb(v) \leftarrow \infty$ (line 2-3). Else, all other leaves' lower bounds are assigned as $d_E(v,t)$ (line 4-5). For example, if v is equal to t, $lb(v)$ is assigned 0. Having fixed leaves' lower bound, the lower bound of all other nodes in F can be calculated from bottom to top. If one node has more than one child and the lower bounds from different child are different, we select the least one(line 6-7). For instance, as shown in figure.1, $\{s, v_1, v_2, v_{10}, v_{11}, t\}$ is the shortest path from s to t, signed as $SP(s,t)$. Based on Algorithm 1, $lb(v_{11}) = lb(t) + weight(v_{11}, t)$, $lb(v_{10}) = lb(v_{11}) + weight(v_{10}, v_{11})$, by this way, $lb(v_7) = lb(v_9) + weight(v_7, v_9)$. If one node have more than one child, such as v_2 has two children, v_7,v_{10}. For one child v_7, $lb(v_2) = lb(v_7) + weight(v_2, v_7)$ and for the other child v_10, $lb(v_2) = lb(v_{10}) + weight(v_2, v_{10})$. We select the least one of them and assign it as $lb(v_2)$. The calculated lower bound with each node v, the relation $|SP(v,t)| \geq lb(v) \geq d_E(v,t)$ is held and we prove it using lemma 1,2,3 to guarantee the correctness.

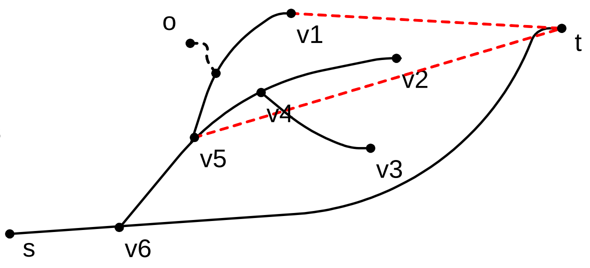


Figure 3: Lower Bound

Lemma 1 *$\forall v, t$, $d_E(v,t)$ is the shortest distance between v and t.*

Lemma 2 *$\forall v, t$, there is only one shortest path between v and t*[4].

Lemma 3 *If $lb(v)$ is valid, $|SP(v,t)| \geq lb(v) \geq d_E(v,t)$.*

Proof:
Step 1, prove $lb(v) \geq d_E(v,t)$.

In classic A* algorithm, $f(v_m) = d_N(s, v_m) + d_E(v_m, t)$. For any non-leaf node v, according to algorithm 1, $lb(v) = d_N(v, v_m) + d_E(v_m, t)$. v_m is the leaf node with the least $f(v_m)$ related to v. Because $d_N(v, v_m) + d_E(v_m, t) \geq d_E(v, v_m) + d_E(v_m, t)$, based on triangular inequality, we can get $d_E(v, v_m) + d_E(v_m, t) \geq d_E(v,t)$. So $d_N(v, v_m) + d_E(v_m, t) \geq d_E(v,t)$. That is $lb(v) \geq d_E(v,t)$. For example, in figure.3, assume that $f(v_1) < f(v_2) < f(v_3)$, then $lb(v_5) = d_N(v_5, v_1) + d_E(v_1, t)$. Because $d_N(v_5, v_1) + d_E(v_1, t) \geq d_E(v_5, v_1) + d_E(v_1, t)$ and $d_E(v_5, v_1) + d_E(v_1, t) \geq d_E(v_5, t)$. We can get $d_N(v_5, v_1) + d_E(v_1, t) \geq d_E(v_5, t)$. That is $lb(v_5) \geq d_E(v_5, t)$. For any leaf node v, if v has more than one neighbor, $lb(v) = d_E(v,t)$. Else, $lb(v) = \infty > d_E(v,t)$. For all stated above, $lb(v) \geq d_E(v,t)$.

Step 2, prove $|SP(v,t)| \geq lb(v)$.

For any leaf node v in F, if v has more than one neighbor, $lb(v) = d_E(v,t) \leq |SP(v,t)|$. Else, $lb(v) = \infty$, which means v at the end of a broken path, so $|SP(v,t)|$ is also equal to ∞. We can get $lb(v) = |SP(v,t)| = \infty$. For any non-leaf node v in F, $lb(v) = d_N(v, v_m) + d_E(v_m, t)$. If $SP(v,t)$ via one leaf node v_a, $|SP(s,v)| + |SP(v,t)| \geq f(v_a)$. For v_m is the leaf node with the least $f(v_m)$ related

[4]The tie is simple to process and not considered here.

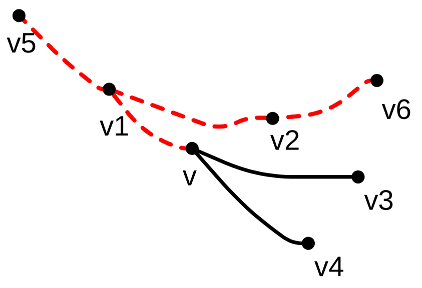


Figure 4: Lower Bound Tree Cleaning

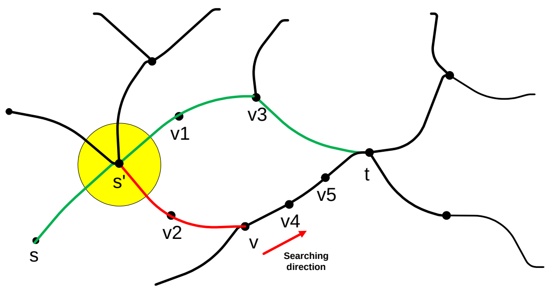


Figure 5: One possible access path

to v, so $f(v_a) > f(v_m)$, $|SP(v,t)|+|SP(s,v)| > f(v_m)$, $|SP(v,t)| > f(v_m) - |SP(s,v)|$. Based on the theory of A* algorithm, (s,v) is just the shortest path from s to v, so $|SP(s,v)| = d_N(v,v_m)$. Then, $|SP(v,t)| > lb(v)$. If $SP(v,t)$ is not via leaf node, $SP(v,t)$ must be via a candidate node o, as shown in figure.3, $|SP(s,v_5)| + |SP(v_5,t)| \geq f(o)$. Because v_m is a leaf node and o is still in the candidate heap (v did not have the chance to be popped in pre-searching phase), we can get $f(o) > f(v_m)$ and $|SP(s,v)| + |SP(v,t)| > f(v_m)$. Therefore, $|SP(v,t)| > lb(v)$. For all of above, $|SP(v,t)| \geq lb(v)$.
Integrating step 1 and step 2, when $lb(v)$ is valid, $|SP(v,t)| \geq lb(v) \geq d_E(v,t)$. □

According to Lemma.3, the range of $lb(v)$ has been controlled between $|SP(v,t)|$ and $d_E(v,t)$.

In our application scenario, the object is moving continuously. When the user received the instant local traffic information, some weights of road segments will be updated and some corresponding nodes' lower bounds will also be invalid. Then, we have to "clean" the lower bound tree and remove all the invalid nodes. Once the weight of one edge e has changed, the lower bounds of end vertices of e, $e.end1$ and $e.end2$, will be invalid. In Algorithm.1, the lower bound is calculated from leaf to root, so the predecessors of $e.end1$ and $e.end2$ will also have invalid lower bounds. To ensure the correctness of lower bound and avoid additional computation, we propose our method to clean the lower bound tree. For each updated edge e, we find both $e.end1$ and $e.end2$, then remove all the related nodes until root. For example, figure.4 shows a branch of lower bound tree originated from v_5. Assuming that v_6 is a vertex of an updated edge, we remove all the related nodes above v_6 (shown in dashed line). Because v_1, the predecessor of v, has been removed from F, v becomes the root of a new branch.

Algorithm 2: Lower bound tree cleaning

```
Data: Lower Bound Tree $F$
Result: updated Lower Bound Tree $F$
1 for each updated edge $e$ do
2     RemoveNodeUntilRoot($e.end1$);
3     RemoveNodeUntilRoot($e.end2$);
```

Algorithm 2 describes the details how to clean invalid nodes of F. Firstly, find all the updated edges.(line 1) For each updated edge e, remove all the related node above $e.end1$ and $e.end2$ until root. (line 2-3)

4.2 Instant Best Path Monitoring

This section introduces the Instant Best Path Monitoring (IBPM) algorithm. It covers the details of upper bound maintenance and how to utilize lower and upper bound to implement our algorithm.

Utilizing upper bound tree T, we may find more than one access path from current position s to destination t, and set the shortest access path as $Temp\_path$. We assign the length of $Temp\_path$ as the upper bound of shortest path from s to t, $|Temp\_path| = ub(sp(s,t))$. Different from lower bound, $ub(sp(s,t))$ will be updated in the searching phase continually. When the current accessing node v is in the upper bound tree T, we get a new access path from s to t. Utilizing the upper bound tree T, we can calculate the length of the new access path, (denote as $currentpath$) efficiently. Then, we compare $|currentpath|$ with $|Temp\_path|$, if $|currentpath| < |Temp\_path|$, update $Temp\_path$ to be $currentpath$. For instance, as shown in figure.5, $Temp\_path$ is s', v_1, v_3, t. Node v is the current accessing node and it is at the upper bound tree T. Then, we get a new access path $P = s', v_2, v, v_4, v_5, t$. If $|P| < |Temp\_path|$, we set P to be $Temp\_path$ and update the $ub(sp(s,t))$. The details will be introduced in Algorithm.3.

Algorithm 3: the Upper Bound Calculation

```
Data: Lower Bound Tree $F$, Upper Bound Tree $T$, Current
      accessing node $v$
Result: the upper bound of $SP(s,t)$
1  $|Temp\_path| \leftarrow \infty$;
2  $|currentpath| \leftarrow d_N(s,v)$;
3  if $v$ in $T$ then
4      while $v.predecessor \neq NULL$ do
5          if $v$ in $F$ then
6              $|currentpath|+ \leftarrow lb(v)$;
7              break;
8          else
9              $|currentpath|+ \leftarrow$
               $weight(v, v.predecessor)$;
10             $v \leftarrow v.predecessor$;
11     if $|currentpath| < |Temp\_path|$ then
12         $|Temp\_path| \leftarrow |currentpath|$;
```

In Algorithm 3, the function monitors the upper bound of shortest path $|Temp\_path|$ in searching phase. Firstly, we set the length of $Temp\_path$ to be ∞ and $|currentpath|$ be the network distance from s to v, $d_N(s,t)$. (line 2) When the current accessing node v is in the upper bound tree T, we can get the $|currentpath|$ by calculate the sum of $d_N(s,v) + d_N(v,t)$. If the lower bound of v is valid and $v \in T$, which means $lb(v) = d_N(v,t)$ and $lb(v)$ is just the shortest path from v to t. The $|currentpath|$ can be calculated directly.(line 5-6) Else, we have to calculate $d_N(v,t)$ step by step according to T, until $v = t$.(line 9-10). Having got the $|currentpath|$, we compare it with $|Temp\_path|$ and if $|currentpath|$ is less than $|Temp\_path|$, both $Temp\_path$ and $UP(sp(s,t))$ will be updated.(line 11-12)

Then, we utilize the bounds (lower bound and upper bound) to implement our IBPM algorithm based on classic A* algorithm. IBPM algorithm includes two steps, the first step (IBPM-I)uses bounds to pretreatment data, decreasing unnecessary data accessing to improve the efficiency. And the second step is based on some characters

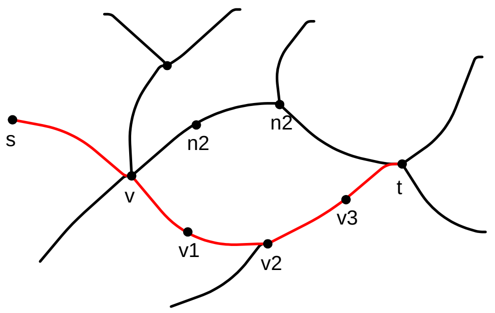


Figure 6: One Shortest path

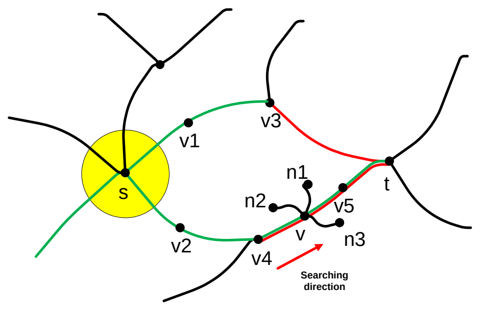


Figure 7: The Second Optimization Step

of A* algorithm and traverse trees, to achieve further optimization.

As mentioned in subsection 4.1, whether node v's lower bound is valid depends on two different conditions, $v \in F$ and $v.pre = F.getparen(v)$. $v \in F$ means that the road conditions after v have not been updated and $v.pre = F.getpre(v)$ means the current search is from the $v.predecessor$ in F to v. Because A* algorithm does not concern searching backward, if $v.pre \neq F.getpre(v)$, means the current search is from some other nodes and $lb(v)$ may be unreliable. For each node v, if $lb(v)$ is valid, we can use $lb(v)$ to check whether it can be the next candidate node in $SP(s,t)$ under current context.

From Lemma.3, if lower bound of node v is valid, $|SP(v,t)| \geq lb(v) \geq d_E(v,t)$. Then, we can get the following conclusions. For any node v with valid lower bound, if $d_N(s,v) + lb(v) \geq ub(sp(s,t))$, because $ub(sp(s,t)) \geq |SP(s,t)|$, we can get $d_N(s,v) + lb(v) \geq |SP(s,t)|$. Therefore, under current context, v can not be the next candidate node and it will not be push into candidate heap. Else, if lower bound of v is invalid, we can only use $f(v) = d_N(s,v) + d_E(v,t)$ as the estimated distance of $SP(s,t)$. Under current context, if $f(v) \geq ub(sp(s,t))$, means $f(v) \geq |SP(s,t)|$, so the shortest path must not be via v and v will not be push into candidate heap.

Until now, we introduce IBPM with optimization which is based on utilizing the lower and upper bounds to reduce the search space. We call it IBPM with the first optimization step, denoted as IBPM-I. Next, we proposed a second optimization step which is based on reusing the previous computed shortest path.

Lemma 4 *As shown by red line in figure.6, if $SP(s,t)$ is the shortest path from s to t, (v,t) must be the shortest path from v to t.*

Proof:
If(v,t) is not the shortest path from v to t, there must be another path $P = SP(v,t)$ and $|(v,t)| > |P|$.
$|SP(s,t)| = d_N(s,v) + |(v,t)| > d_N(s,v) + |P|$
There exist another access path from s to t and its length is less than $|SP(s,t)|$, which conflicts with our assumption. Therefore, $(v,t) = SP(v,t)$. □

The further optimization is based on the following conditions: the lower bound of v is valid and v is at a history shortest path ($v \in T$), as shown in figure.7, the green line describes the lower bound tree F and red line describes the upper bound tree T. Node v in both F and T means that the road conditions after v have not been updated and (v,t) is still the shortest path from v to t. We can use these conditions to improve our search algorithm ulteriorly. For the current searching direction is from v_4 to v and v_4 is the predecessor of v in F. So, v_4 will not be concerned as v's neighbors and not be put into candidate heap $Open\_list$, because A* algorithm does not concern searching backward. As the classic A* algorithm, only four neighbors of v (n_1, n_2, n_3 and v_5) will be concerned and put into $Open\_list$. Until now, the problem is obvious, is it necessary to put all the neighbors of v into $Open_list$?

In figure.7, (v,t) is a part of a history shortest path and the road conditions around (v,t) have not been updated. For LEMMA 4, (v,t) is still the shortest path from v to t, signed as $SP(v,t)$.

Algorithm 4: Shortest Path Determinant

```
   Data: Lower Bound Tree $F$, Upper Bound Tree $T$,
   Source $s$,destination $t$ and ub(sp(s,t))
   Result: $SP(s,t)$
1  Node $v, pre$;
2  Heap $Open\_list, Close\_list$;
3  $Open\_list.push(s)$;
4  while $Open\_list \neq NULL$ do
5      get node $v$ from $Open\_list$ with the least $f(v)$;
6      if $v = t$ then
7          Return $GetShortestPath(t)$;
8      if $v \in T\&F$ & $v.pre = F.getpre(v)$ then
9          $Open\_list.push(T.getpre(v))$;
10     else
11         for each unscanned neighbor $n$ of $v$ do
12             if $lb(n)$ is valid & $n.pre = F.getpre(n)$ then
13                 if $d_N(s,v) + weight(v,n) + lb(n) >$
                   $ub(sp(s,t))$ then
14                     continue;
15             else
16                 if $f(n) > ub(sp(s,t))$ then
17                     continue;
18             $Open\_list.push(n)$;
19     $v \leftarrow Open\_list.pop$;
20     $Close\_list.push(v)$;
21 $ShortestPath \leftarrow Temp\_path$;
22 Return $ShortestPath$;
23 begin
24     Procedure $Open\_list.push(n)$
25     if $n \in Open\_list$ & new $f(n)$ is not better then
26         Return;
27     if $n \in Close\_list$ & new $f(n)$ is not better then
28         Return;
29     $n.pre \leftarrow v$;
30     Remove any $n$ in $Open\_list$ and $Close\_list$;
31     $Open\_list.add(n)$;
32 end
```

For Lemma.2, there is only one shortest path between any two nodes. So if the shortest path between s and t is via v under current context, the following nodes must be in $SP(v,t)$. Or there would be another shortest path from v to t, but it is conflicted with Lemma 2. Therefore, we can only concern the nodes as $SP(v,t)$, nor any other neighbors of v. For next searching step, only push v_5, the next node of v in $SP(v,t)$, into candidate heap $Open\_list$. Other neighbors, n_1, n_2 and n_3, will not be concerned. Obviously, this optimization can improve the effect of our IBPM algorithm notably. Based on our experiment results, comparing with A* algorithm and naive algorithm IBPM-I, no matter time cost nor space cost, our IBPM algorithm improve the efficiency more than 10 times.

Algorithm 4 describes the whole process of the shortest path search, including all the details of our two-step optimization. Node v is the current accessing node, while node $v.pre$ is the predecessor of v under current context (line 1). As classic A* algorithm, we set a candidate heap $Open\_list$, and every time select the node with the least $f(v)$ from $Open\_list$ and put it into $Close_list$ (line 2,5). Search is from current source s and then we push s into candidate heap (line 3). If current accessing node $v = t$, means we have already find the shortest path, then current search is finished and returns the shortest path (line 6-7). If the lower bound of v is valid and v is in the upper bound tree T, we only push the next node of v in $SP(v, t)$, just the predecessor of v in T, into $Open\_list$ and no need to concern other neighbors of v (line 8-9). That is just the second step optimization we introduce above. Else, the other neighbors of v do also need to be concerned. For each neighbor n, if the lower bound of n is valid, we compare the estimated path length, $d_N(s, v) + weight(v, n) + lb(n)$, with the upper bound of the shortest path, only it is less than $ub(sp(s, t))$, n can be push into $Open\_list$ (line 12-13,18). If the lower bound is invalid, we have to use the classic estimated length $f(v) = d_N(s, v) + d_E(v, t)$ and compare it with $ub(sp(s, t))$ (line 16,18). Line 12-17 is just the first step optimization. The other parts of algorithm 4 are the same as classic A* algorithm, our optimization is based on classic A* algorithm and according to specifical application, add some restricting conditions to improve the efficiency.

5 Discussion

5.1 Memory Cost

The performance improvement to a large extents is on the cost of memory in order to reuse the previous computing results. Since the trees proposed in this work record each visited vertex only once (see section 4), the memory requirement is very small. In the worst case, the nodes need to be recorded in the memory is the size of entire network. That is the memory requirement is $\mathbf{O}(n)$. For Lower Bound Tree, each node has three properties $\{id, lowerbound, predecessor\}$ and for Upper Bound Tree, each node also has three properties $\{id, predecessor, Euclidean\ Distance\}$. Assume that each property is size 4 Bytes. In the worst case, the whole map with n nodes need to be saved in the traverse trees. So, the $Memory\_cost = 4Byte * 3 * n + 4Byte * 3 * n = 24nByte$. For example, we chosen the road network of California, including 20148 nodes and under the worst conditions, every node will be stored in the traverse trees. $Memory\_cost = 4byte * 3 * 21048 + 4byte * 3 * 21048 = 505KB$. This is reasonable small in most application scenarios. At the same time, the efficiency can be improved dramatically.

5.2 Multiple Moving Objects

As we discussed in section 1, our problem is different from single-source DSP problem due to the consideration of instant local traffic conditions. We view our problem as a single-destination DSP. When several users are moving to the same destination, they can share the tree structures proposed in the work. One scenario is in the online car racing games. Usually several (8-16) players start from the same source to the same destination. Since the shortest path instantly updated is a current best possible traveling suggestion considering the local temporary traffic conditions, the players can chose to follow or not. As a result after a period of time of the game, cars may scatter in the networks and thus their shortest paths to the destination vary. Since they share the same destination, the search spaces for them are overlapped to large extents, that is, the upper and lower bound discussed above in the overlapped region can be shared. That provides opportunity to share the tree structures among them such that the memory cost is on maintaining two tree structures only. Otherwise, if each car has its own tree structures, the memory requirement will be $n * m$ where n is the number of cars and m is the size of trees. The efficient memory cost is critical in the application of such online games since these tree structures are kept in server side, which usually needs to support a large number of games at the same time.

Table 2: **Experiment setting**

Name	Value
Length of query path	300 to 1500 in California Road Network, and 40 to 120 in Oldenburg Road Network.
Radius of Instant Range A	0.01 to 0.05 in California Road Network and 100 to 500 in Oldenburg Road Network. Default: 0.01 and 100
Moving Step	1 to 5 in both California Road Network and Oldenburg Road Network, Default: 1

6 Experiments

In this section, we conduct experiments on data sets of California Road Network and City of Oldenburg Road Network(stored as adjacency lists) [5], which contain 21,048 nodes and 6,105 nodes respectively (see figure 9 & figure 10). All algorithms are implemented in Java and tested on a windows platform with Intel Core2 CPU (2.13GHz) and 2GB memory. The main metric we adopt is the CPU time of the monitoring process of our algorithm during the user's movement from the start to the end. Apart from this, the number of visited nodes is also a very important factor we consider as space cost. The length of query path is calculated by the number of nodes in the corresponding path. In California Road Network, the length of query path is from 300 to 1500, while in City of Oldenburg Road Network, the length of query path is from 40 to 120. The radius of Instant Range A in California Road Network is from 0.01 Longitude to 0.05 Longitude (0.01 Longitude $\approx$ 0.8 Kilometer in California road network). In City of Oldenburg Road Network, the radius is from 100 unit to 500 unit (100 unit $\approx$ 0.8 Kilometer). The moving step describes the reception of frequency of instant traffic information. In our experiments, the moving step is from 1 to 5. ($Movingstep = 1$ means when the user moves forward one node, the navigation system will receive the instant traffic information and recompute the shortest path).

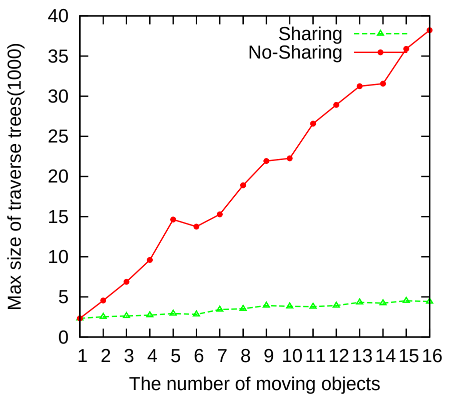
The setting of our experiments is summarized in Table 2. The default setting of the radius of Instant Range A is 0.01 in the California Road Network, 100 in using the Oldenburg Road Network (both 0.01 and 100 mean approximately 800 meters, just one-minute-drive distance). The default moving step is 1, when the user arrives at each new intersection, the system will recompute the best path from current position to the fixed destination.

For the purpose of comparison, we select A* algorithm as the basic line. In addition, a naive algorithm based on the first-step optimization of IBPM algorithm is also implemented, and we refer it as IBPM-I algorithm. In this algorithm, the way to establish, maintain and update the traverse trees is the same as that in our IBPM algorithm.

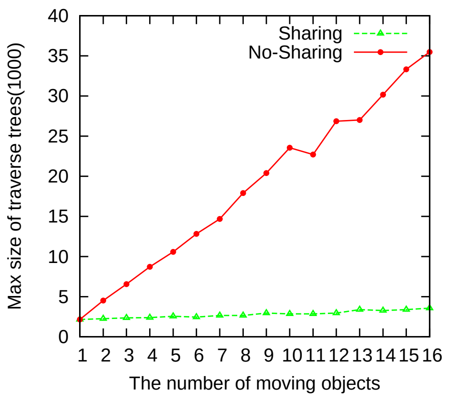
[5]www.cs.fsu.edu/ lifeifei/SpatialDataset.htm

Figure 8: California Road Network

Figure 9: City of Oldenburg Road Network



(a) The Max Size of traverse trees in California Road Network



(b) The Max Size of traverse trees in Oldenburg Road Network

Figure 10: The Max Size of traverse trees

6.1 Effect of Path Length

First of all, we study the effect of the length of query path on CPU time and the number of visited nodes. The query path is the route on which the user travels to the destination. It is expected that the monitoring cost is in linear to the length of the path because a longer distance requires more updates and more complicated path recomputing. In this experiment, we test the performance of the query path length varying from 300 to 1200, and 40 to 120 in the California and the City of Oldenburg Road Networks respectively. The radius of Instant Range A is set to be 0.01 in California Road Network and 100 in City of Oldenburg Road Network. And the moving step is set to be 1. In the California Road Network, as shown in figure 11(a), when the path length is between 300 and 1200, the CPU time of the IBPM algorithm increases gradually from less than 1 second to nearly 3 seconds, while the time of the A* algorithm rises dramatically to nearly 30 seconds. The time of IBPM-I algorithm is less than pure A* algorithm, however still over 20 seconds. figure 12(a) describes the impact of the query path length on the number of visited nodes under the default settings. Similar to the effect of the querty path length on the CPU time, the number of visited nodes are much more demanding in the A* algorithm and IBPM-I methods than in the IBPM method.

In the Oldenburg Road Network, the size of which is much smaller than California road network, the effect of IBPM algorithm is also substantially better as displayed in figure 13(a) and 14(a)). For the length of query path between 40 and 120, the CPU time of A* algorithm starts from less than 100 ms and goes over 700ms, the IBPM-I algorithm is slightly better than A* algorithm, though also reaching a maximum of almost 600ms. The IBPM algorithm costs only 300ms, less than half of the time of A* algorithm. The time saving in the smaller size Oldenburg Road Network is not as apparent as in the California Road Network due to the fact that the time of maintaining and updating the traverse trees is a large proportion of total time.

6.2 Effect of Radius of Range A

The radius of Instant Range A demonstrates the range of instant traffic information being covered. Generally speaking, the larger area is covered by Instant Range A, the better the calculating result is. However, the time cost and space cost will increase similarly as the possibility of deviating from the best path from current position to the destination is higher. In this experiment, we test the performance of IBPM alrotithm with the radius of Instant Range A varying from 0.01 to 0.05 in the California Road Networks and 100 to 500 in the Oldenburg Road Networks. The moving step is defaulted to be 1 again. In figure.11(b), the CPU time has been constant regardless that the radius has changed from 0.01 to 0.05. In contrast, the time cost in both A* algorithm and IBPM-I algorithm has increased proportionally as the radius increases. Again as displayed in figure.12(b), the number of visited nodes has similar correspondence to CPU time as in figure.11(b).

For the Oldenburg Road Network in figure.13(b), the time cost in IBPM rises slightly as radius goes up from just over 200ms to 280ms, while the time cost of both A* and IBPM-I algorithms has been more than 400ms. In figure.14(b), the number of visited nodes as required by IBPM is again much less than that by A* and IBPM-I algorithms. It is also worth being noted that the movement of time and space cost does not change as considerably as in the example of California Road Network. The reason behind is that there are limited routes to be chosen from and the change of radius will not consequently lead to different routes in smaller size road networks.

6.3 Effect of Step Length K

The moving step describes the frequency of receiving the instant traffic information and recomputing the best path from current position to the fixed destination. The bigger the moving step is, the lower frequency the information is being updated, and the less the total time cost as well as the space cost are. In California road network, the radius of instant Range A is 0.01. As can be seen from figure.11(c) and figure.12(c), the time and space cost is much less at the beginning of the travel in IBPM than in A* and IBPM-I algorithm and this advantage has been always maintained even with the substantial decrease of time and space costs in A* and IBPM-I algorithm with the increase of moving step from 1 to 5.

In Oldenburg Road Network (figure 13(c) and 14 (c)), we set the radius of instant Range A to be 100. The savings of time and space costs in IBPM than in A* and IBPM-I algorithms are again clearly demonstrated.

6.4 Multiple Moving Objects

Multiple moving objects such as cars in the online racing games move from the same source to the same destination. The upper and lower bound trees can be shared among them. This experiment tests the impact of the number of moving objects to memory cost as shown in figure.10. The maximum number of nodes recorded in the shortest path monitoring is shown. The memory size can be inferred directly by multiplying the size of a single node (12Bytes). When the number of moving objects changes from 1 to 16, it demonstrates a constant memory requirement if they share trees. The reason is that only two trees are maintained. In contrast, if each moving object maintains its own traverse trees, the memory cost increases proportional with the number of moving objects. In this experiment, the radius is 0.01 and the moving step is 1 for California data set; the radius is 100 and the moving step is 1 for Oldenburg data set.

7 Conclusions

In this paper we propose two tree structures and an algorithm call Instant Shortest Path Monitoring (IBPM) in order to efficiently monitor the shortest path for an moving object when it is moving to a given destination and the local traffic condition is updated repeatedly. Our problem is different from exiting dynamic shortest path problems, i.e. single-source and all-pairs DSP. We view our problem as a single-destination with instant local traffic condition updates. The efficient processing of this problem has a wide range of applications such as online racing games and large scale traffic simulations. Comparing to simple method by running A* algorithm over and over again, our method has dominant processing efficiency. The memory cost is linear with the nodes visited by running the A* algorithm once.

References

Agrawal, R. and Srikant, R. (1995), Mining sequential patterns, *in* ' Proceedings of ICDE', pp. 3-14 .

Chen, Z., Shen, H. T., Xu, Q. and Zhou, X. (2009), Instant Advertising in Mobile Peer-to-Peer Networks, *in* 'Proceedings of ICDE', pp. 736-747.

Chen, Z., Shen, H. T., Zhou, X. and Yu, J. X. (2009), Monitoring Path Nearest Neighbor in Road Networks, *in* 'Proceedings of ACM SIGMOD', pp. 591-602.

Cormen, T. H, Leiserson, C. E., Riverst, R. L and Stein, C. (2001), Introduction to Algorithms (second edition), The MIT Press.

Dijkstra, E. W. (1959), A note on two problems in connection with graphs, *in* 'Numerische Math', 1:269-271.

Demetrescu, C., Emiliozzi, S. and Italiano, G. F. (2004), Experimental analysis of dynamic all pairs shortest path algorithms, *in* 'Proceedings of SODA', pp. 369-378.

Demetrescu, C. and Italiano, G. F. (2004), A new approach to dynamic all pairs shortest paths, *in* 'Journal of the ACM', vol 51(6):968-992.

Ding, B., Yu, J. X. and Qin, L. (2008), Finding time-dependent shortest paths over large graphs, *in* 'Proceedings of EDBT', pp. 205-216.

Frigioni, D., Marchetti-Spaccamela, A. and Nanni, U. (1996), Fully dynamic output bounded single source shortest path problem (extended abstract), *in* 'Proceedings of SODA', pp. 212-221.

Frigioni, D., Ioffreda, M., Nanni, U. and Pasquale, G (1998), Experimental analysis of dynamic algorithms for the single-source shortest-path problem, *in* 'ACM Journal of Experimental Algorithms', vol 3:5.

Frigioni, D., Marchetti-Spaccamela, A. and Nanni, U. (2000), Fully dynamic algorithms for maintaining shortest path tree, *in* 'Journal of Algorithms', vol 34, pp. 351-381.

Fu, L., Sun, D. and Rilett, L. R. (2006), Heuristic shortest path algorithms for transportation applications: state of the art, *in* 'Computers in Operations Research', vol 33(11), pp. 3324-3343.

Gonzalez, H., Han, J., Li, X., Myslinska, M., and Sondag, J. P. (2007), Adaptive Fastest Path Computation on a Road Network: A Traffic Mining Approach, *in* 'Proceedings of VLDB', pp. 794-805.

Hart, P. E., Nilsson, N. J. and Raphael, B. (1968), A formal basis for the heuristic determination of minimum cost paths, *in* 'IEEE Transactions on Systems Science and Cybernetics', pp.4(2):100-107.

Han, J. and Pei, J. (2000), Mining frequent patterns by pattern-growth: Methodology and implications, *in* 'SIGKDD Explorations, Volume 2', vol.2, pp. 14-20 .

King, V. (1999), Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs, *in* 'Proceedings of FOCS', pp. 81-91.

Kanoulas, E., Du, Y., Xia, T., and Zhang, D. (2006), Finding fastest paths on a road network with speed patterns, *in* 'Proceedings of ICDE', pp. 10-19.

Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G. and Teng, S. (2005), On Trip Planning Queries in Spatial Databases, *in* 'Proceedings of SSTD', pp. 273-290.

Mouratidis, K., Papadias, D. and Hadjieleftheriou, M. (2005), Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring, *in* 'Proceedings of ACM SIGMOD'. pp. 634-645.

Nannicini, G., Baptiste, P., Krob, D. and Liberti, L. (2008), Fast Computation of Point-to-Point Paths on Time-Dependent Road Networks, *in* 'Proceeding of COCOA', pp. 225-234. Giacomo Nannicini, Philippe Baptiste, Daniel Krob, Leo Liberti

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and PrefixSpan, M.-C. Hsu. (2001), Mining sequential patterns efficiently by prefix-projected pattern growth, *in* 'SIGKDD Explorations, Volume 2', vol.2, pp. 14-20.

Pallottino, S. and Scutella, M. G. (1997), Shortest path algorithms in transportation models: classical and innovative aspects, *in* 'Technical Report TR-97-06, 14'.

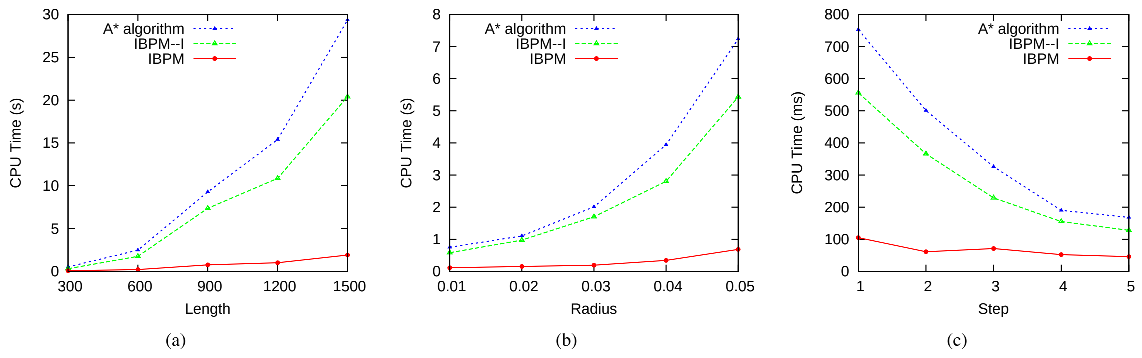


Figure 11: Performance of CPU Time in California Road Network

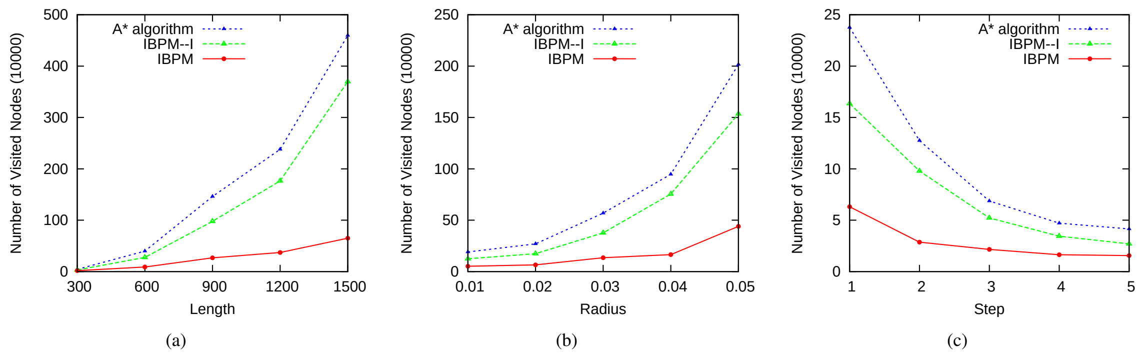


Figure 12: Performance of CPU Time in California Road Network

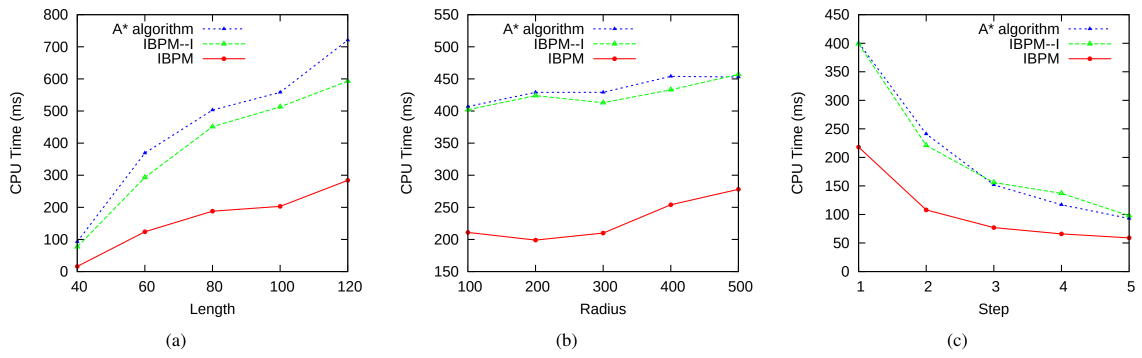


Figure 13: Performance of CPU Time in City of Oldenburg Road Network

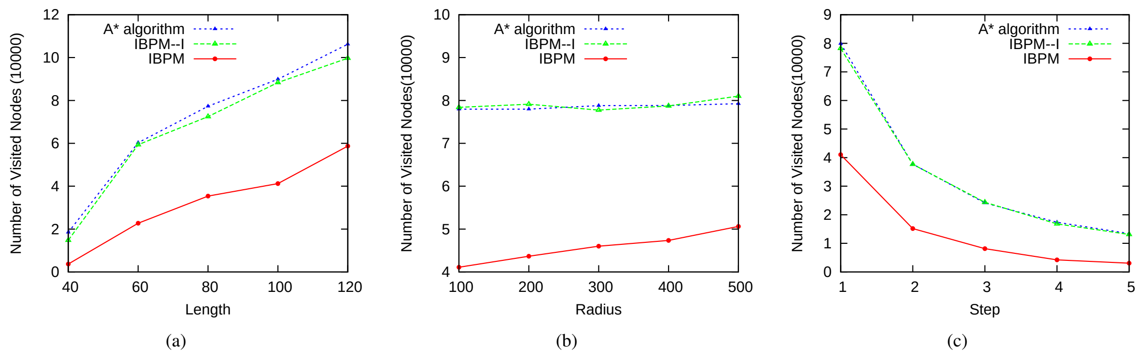


Figure 14: Performance of Visited Nodes in City of Oldenburg Road Network