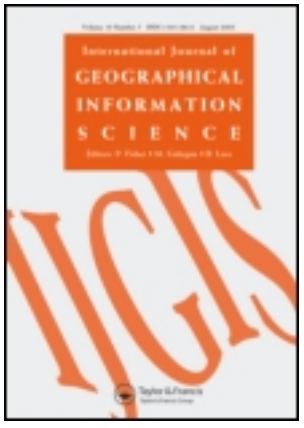

International Journal of Geographical Information Science

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tgis20

A shortest path algorithm with novel heuristics for dynamic transportation networks

B. Huang [a], Q. Wu [b] & F. B. Zhan [c]

[a] Department of Geography and Resource Management, The Chinese University of Hong Kong, Shatin, NT, Hong Kong

[b] MRF Geosystems Corporation, 625—14th Street, NW, Calgary, Canada T2N 2A1

[c] Texas Center for Geographic Information Science, Department of Geography, Texas State University, San Marcos, TX 78666

Available online: 07 Jun 2007

To cite this article: B. Huang, Q. Wu & F. B. Zhan (2007): A shortest path algorithm with novel heuristics for dynamic transportation networks, International Journal of Geographical Information Science, 21:6, 625-644

To link to this article: http://dx.doi.org/10.1080/13658810601079759

International Journal of Geographical Information Science
Vol. 21, No. 6, July 2007, 625–644

Research Article

A shortest path algorithm with novel heuristics for dynamic transportation networks

B. HUANG*†, Q. WU‡ and F. B. ZHAN§

†Department of Geography and Resource Management, The Chinese University of Hong Kong, Shatin, NT, Hong Kong

‡MRF Geosystems Corporation, 625—14th Street, NW, Calgary, Canada T2N 2A1

§Texas Center for Geographic Information Science, Department of Geography, Texas State University, San Marcos, TX 78666, USA

(*Received 23 June 2006; in final form 18 October 2006*)

Finding an optimal route in dynamic real-time transportation networks is a critical problem for vehicle navigation. Existing approaches are either too complex or incapable of managing complex circumstances where both the location of a mobile object and traffic conditions change over time. In this paper, we propose an incremental search approach with novel heuristics based on a variation of the A* algorithm–Lifelong Planning A*. In addition, we suggest using an ellipse to prune the unnecessary nodes to be scanned in order to speed up the dynamic search process. The proposed algorithm determines the shortest-cost path between a moving object and its destination by continually adapting to the dynamic traffic conditions, while making use of the previous search results. Experimental results evince that the proposed algorithm performs significantly better than the well-known A* algorithm.

Keywords: Dijkstra's algorithm; A* algorithm; Lifelong planning A*; Navigation; Dynamic shortest paths; Routing; Constrained ellipse

1. Introduction

The complexity of transportation systems presents considerable challenges to technology innovations related to Location-Based Service (LBS) and vehicle navigation in an urban environment. While LBS can now provide information about real-time traffic conditions which are collected through loop detectors, probe vehicles, and video surveillance systems, the utilization of such information for improving services such as real-time en route navigation guidance still lags behind (Huang and Li 2004, Huang and Wu 2007). A challenging issue is to develop efficient and less computationally intensive algorithms that can be used to compute dynamic shortest paths between moving objects and given destinations in dynamic changing networks.

The computation of shortest paths has been a subject of extensive research in Artificial Intelligence, Operations Research, and Transportation for many years (Dijkstra 1959, Pallottino 1984, Zhan and Noon 1998). Dijkstra's algorithm (Dijkstra 1959) and the A* algorithm (Hart *et al.* 1968) are two well-known shortest-path algorithms. Existing approaches using these algorithms were mostly devised for

*Corresponding author. Email: bohuang@cuhk.edu.hk

International Journal of Geographical Information Science
ISSN 1365-8816 print/ISSN 1362-3087 online © 2007 Taylor & Francis
http://www.tandf.co.uk/journals
DOI: 10.1080/13658810601079759

static networks, so they are not efficient when directly applied to dynamic shortest path planning in a real-time traffic environment. In particular, it may take a long time for some mobile terminals, such as in-car computers and personal digital assistants, to re-plan a new route from scratch using a static approach while en route because of the limited computational power of these mobile devices.

In a dynamic traffic environment, traffic conditions are time-dependent. For instance, when travelling from an airport to a conference centre, although we can plan a shortest path prior to departure according to the traffic conditions at that time, we may have to adjust the route while en route because traffic conditions change all the time. In some cases, we have to modify the traveling route from time to time and re-plan a new route from our current location to the destination based on the real-time traffic information. Under these circumstances, not only traffic conditions but also the location of the traveller (mobile user) change over time. To accommodate path planning for mobile users in these situations, we need an algorithm that can take into account both the changing locations of the traveller and changing traffic conditions in the area. To the best of our knowledge, little has been reported on the computation of this type of dynamic shortest paths. Currently, most existing dynamic algorithms only consider changes in traffic conditions and compute a dynamic shortest path between a pair of or all pairs of *fixed* nodes on a network, or consider changes in the position of the start node but re-compute the shortest path from scratch when traffic condition changes (see also Chabini 1998, Pallottino and Scutella 1998, 2003, Frigioni *et al.* 2000). These approaches do not serve the purpose of real-time navigation guidance well.

We propose a novel dynamic shortest path algorithm to solve these problems. The proposed algorithm efficiently computes the dynamic shortest path between a moving object and the destination on a dynamic network in which traffic conditions are updated in real time. Because the A* algorithm is generally more efficient than Dijkstra's algorithm (Preygel 1999), we developed the dynamic shortest path algorithm based on a variation of the A* algorithm–the Lifelong Planning A* (LPA*) algorithm (Koenig *et al.* 2004).

The rest of the paper is organized as follows. Section 2 provides some background information about the shortest-path problem and reviews the A* algorithm and the Lifelong Planning A* (LPA*) algorithm. Section 3 describes and illustrates the proposed algorithm. Section 4 discusses the experiments that we used to evaluate the proposed algorithm and results of the experiments. Section 5 concludes the paper.

2. Related work

A road network consists of nodes and links. Nodes are usually the intersections or dead ends and links are the road segments connecting two nodes. A road network can be represented as a weighted directed graph $G=(N, L, W)$ consisting of a node set N, a link set L, and a cost set W. The link set L is a subset of the cross-product $N*N$, and the cost set W is the weight vector representing the costs associated with the links. Each element (u, v) in L denotes a directed link joining nodes u and v from u to v. The costs associated with link (u, v) are denoted by $W(u, v)$. Cost $W(u, v)$ takes a value from the set of real numbers.

A path in a graph from a source node s to a goal node d is a sequence of nodes $(v_0, v_1, v_2, \ldots, v_k)$ where $s=v_0$, $d=v_k$, and the links (v_0, v_1), (v_1, v_2), …, (v_{k-1}, v_k) connecting these nodes. The number of links in a path is called the cardinality of the

path. The cost of the path is the sum of the weights of links, i.e. $\sum_{i=1}^{k} W(v_{i-1}, v_i)$, on the path. An optimal path from node s to node d is the path with the minimum cost, denoted by $c(s, d)$. If there exists a path from every node to all other nodes in a graph, then the graph is called connected.

Even though scholars from different disciplines tend to group the types of shortest-path problems slightly differently, one can discern, in general, between paths that are calculated as one-to-one, one-to-all, or all-to-all shortest paths (Dreyfus 1979). In this paper, we will focus our discussions on the one-to-one shortest-path problem in a connected graph only.

2.1 *Dijkstra's algorithm*

Dijkstra's algorithm, named after its inventor, has been influential in path-computation research. The algorithm works by examining all temporarily labelled nodes in the network starting with the source node. At each iteration, it selects the labelled, but not examined, node with the least labelled distance from the set of temporarily labelled nodes (denoted as set S'), updates its distance label, and puts the node into a set of examined and permanently labelled nodes (denoted as S). In the node-examination process, a tree connecting all examined nodes is created, and the permanently labelled distance associated with each examined node is the shortest path distance from the source node to that node.

Initially, S' contains all the nodes, and then nodes are moved from S' to S one by one during the node examination process. In a typical implementation of the Dijkstra's algorithm, all temporarily labelled nodes in S' are managed using a priority queue and ordered in the queue based on their labelled distances. At each iteration, the node (u) at the top of the priority queue is selected, then examined, marked as an examined node with a permanently labelled distance, added to S, and each edge emanating from the node is relaxed. This process is also called node expansion. If the labelled distance of node u plus the weight of link (u, v) is shorter than the labelled distance of node v, then the estimated shortest distance from the source node to node v is updated with a value equal to the sum of the labelled distance of node u plus the weight of link (u, v). The algorithm continues the node examination process and takes the next node at the top of the priority queue as the next node to be examined. The algorithm terminates when the goal node is reached or when the priority queue becomes empty.

Since this algorithm does not utilize any heuristic, it searches by expanding out equally in every direction on the network and explores a too large and often unnecessary search area before it reaches the goal node. Although this algorithm is guaranteed to find the optimal path, it has not been used a great deal because of its relatively poor computational time. This situation has led to the development of heuristic-based shortest path search algorithms. Among them, the A* algorithm is regarded as the most efficient method (Preygel 1999).

2.2 *A* algorithm*

The A* algorithm, a variant of Dijkstra's algorithm, is a heuristic algorithm. Fu *et al.* (2006) provides a survey review of several heuristic shortest path algorithms, including A*, one of the most well known among them. A* adds a heuristic to speed up the search process in the computation of the shortest path between a source node and a goal node. Like Dijkstra's algorithm, the search space is also divided into two

sets: S, the set of nodes whose shortest paths to the source node are already determined, and S', the set of nodes whose shortest paths to the source node is yet to be determined. The A* algorithm differs from Dijkstra's algorithm in that it not only considers the labelled distance of the node to be examined (called 'start distance' denoted as $g(n)$), but also takes into consideration the 'closeness' of the node to be examined to the goal node. The 'closeness' is an estimated 'goal distance', denoted as $h(n)$. It refers to the estimated cost of moving from the current node n to the goal node. Because the path is not yet complete, this value is not determined and has to be 'guessed'. This is where the heuristic method comes into play.

In general, a search algorithm that is guaranteed always to find the shortest path to a goal node is called admissible. If A* uses a heuristic that never overestimates the value of $h(n)$, then it has been proven that A* is admissible (Koenig *et al.* 2004). The heuristic that makes the A* algorithm admissible is called an admissible heuristic. For A*, if the value of $h(n)$ is set to zero, this algorithm works the same way as Dijkstra's algorithm does. The best possible heuristical value of $h(n)$, although usually impractical to compute, is the actual minimal distance from the start node to the goal node. One example of a practical admissible heuristic value of $h(n)$ that we can use is the straight-line distance between the start node and the goal node.

The A* algorithm balances these two distances $g(n)$ and $h(n)$ when searching for the next node to be examined. It ranks all temporarily labelled nodes based on the values computed by the expression: $f(n)=g(n)+h(n)$, and always selects the node *with* the lowest $f(n)$ value as the next node to be examined. This heuristic ensures that A* only tries to select nodes that mostly likely directly lead to the direction towards the goal node. This heuristic and its consequent search process help reduce the computation time of the search process. Hence, the A* algorithm is faster than Dijkstra's algorithm (Hart *et al.* 1968). It is an example of best-first search. However, although the search space is reduced by the heuristic, most examined nodes in the search process of A* still do not lead to the optimal path. Hence, there is still room for improvement in reducing the number of unnecessarily examined nodes to speed up the search process of the A* algorithm. The Lifelong Planning A* Algorithm described in the next section partly achieves this objective.

2.3 *Lifelong planning A**

Lifelong Planning A* (LPA*) is an incremental version of A* that can reuse the results in previous searches (Koenig *et al.* 2004). LPA* has two searches. The first search of LPA* is the same as that for A*, but all subsequent searches are much faster because it reuses those parts of the previous search tree that are identical to the new search tree. The main principle of LPA* is described in the following statements. Assume S denotes the finite set of nodes of the graph and $\text{succ}(s)\subseteq S$ denotes the set of successors of node $s \in S$. Similarly, $\text{pred}(s)\subseteq S$ denotes the set of predecessors of node $s \in S$. In this case, $0<c(s, s')\leqslant\infty$ denotes the cost of moving from node s to node $s' \in \text{succ}(s)$, and $g(s)$ denotes the start distance of node $s \in S$, i.e. the cost of a shortest path from s_{start} to s. As for A*, the heuristic approximates the goal distance of the node s. They need to be consistent, i.e. satisfy $h(s_{\text{goal}})=0$ and $h(s)<c(s, s')+h(s')$ for all nodes $s \in S$ and $s' \in \text{succ}(s)$ with $s\neq s_{\text{goal}}$. If both the topology of the graph and the current link costs are known, LPA* always determines a shortest path from a given start node $s_{\text{start}} \in S$ to a given goal node $s_{\text{goal}} \in S$.

There are three estimates held by LPA* in its lifetime. The first is the $g(s)$ of the start distance of each node s, which directly corresponds to the g-values of A* and can be reused in subsequent searches. The second is the $h(s)$ of the approximate distance to s_{goal}, which has the same meaning as the h-value in A* and is used to drive the search to the goal direction. The last is another estimate of the start distances, namely *rhs* values which are one-step look-ahead values based on the g-values and thus are potentially better informed than the g-values. They always satisfy the following relationship: $rhs(s)=0$ when s is the start node or $rhs(s)=\text{Min}_{s' \in \text{pred}(s)}(g(s')+c(s, s'))$ otherwise. As with the LPA* algorithm, each node is locally consistent if its g-value equals its *rhs*-value. This concept is important because the g-values of all nodes equal their start distances if all nodes are locally consistent. Actually, there is no need to make every node locally consistent in LPA*. Instead, it uses the $h(s)$ heuristic to converge the search and update only the g-values involved in the shortest path computation from s_{start} to s_{goal} (Koenig *et al.* 2004).

LPA* maintains a priority queue that always exactly contains the locally inconsistent nodes. These are the nodes whose g-value may need to be updated in order to make them locally consistent. The node keys in the priority queue correspond to the f-values used by A*. Similar to A*, LPA* always expands the node in the priority queue with the smallest key (f-value). The key, $k(s)$, of node s is a vector with two components: $k(s)=[k1(s); k2(s)]$, where $k1(s)=\text{Min}(g(s), rhs(s))+h(s)$ and $k2(s)=\text{Min}(g(s), rhs(s))$. Clearly, $k1(s)$ corresponds directly to the f-values ($f(s)=g(s)+h(s)$) used by A* because both the g-values and *rhs*-values of LPA* correspond to the g-values of A*, and $k2(s)$ corresponds to the g-values of A*. Similar to A*, LPA* always expands the node in the priority queue with the smallest $k1$-value (f-value). Any ties are broken favouring the node with the smallest $k2$-value (g-value). The resulting behaviour of LPA* and A* is also similar. LPA* expands nodes until s_{goal} is locally consistent and the key of the node set for expansion next is no less than the key of s_{goal}. While LPA* was designed for robot navigation on a grid map (Koenig *et al.* 2004), the next section will illustrate how to apply LPA* to shortest path search using a simple road network.

3. Applying LPA* to path finding in road networks

As shown in figure 1, the goal is to find the shortest path from A to K in the graph representing a simple road network. The upper-left graph gives the weight for each link. For illustration convenience, the start distance and heuristic are also given in parentheses near each node. When LPA* performs the first search, it initializes the g-value and *rhs*-value of all nodes as infinity. Actually, we cannot initialize all of the nodes in a large map and only initialize each node whenever we encounter it while searching. In the following iterations, there is also a bracket for each node: the two values denote the $k1$-value and $k2$-value, respectively. The number above the parentheses is the start distance (g-value). Any single values in parentheses denote the g-value of the nodes which are locally consistent. The black square indicates a node that is being visited in the current iteration. In this example, we use the Manhattan distance between any node and goal node as the heuristic for LPA*.

In iteration #1, the search expands from start node A, finds three successors (B, E, and D), assigns their keys, and inserts them into a priority queue. They are ordered in the queue based on the vaue of their keys. Next, the node with the smallest priority taken (popped) from the priority queue is node E ($k1=44$). The node E is now locally consistent and has been popped from the priority queue. In

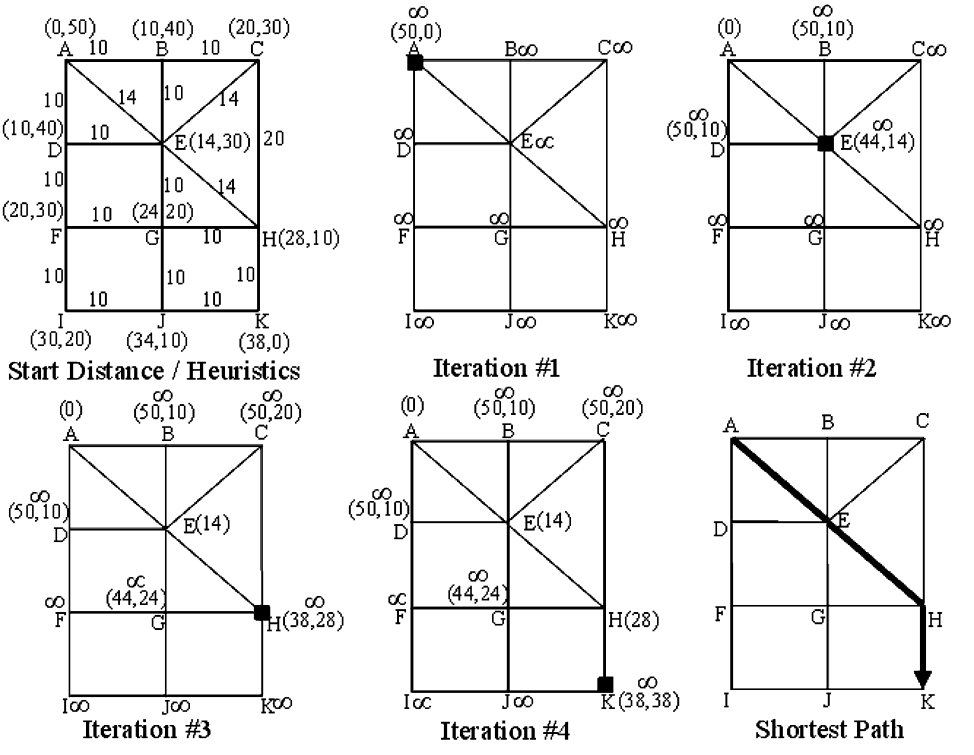

Figure 1. LPA* first search.

the same way, the search expands to the nodes C, H, and G. In this iteration, *rhs*(C) has been updated by 20 because the smallest *g*-value of its neighbours is *g* (B)=10, and its parent is assigned as B. Hence, we maintain the shortest path from the start node to each visited node. Finally, H ($k1=38$) is popped from the priority queue. The search terminates when node K is reached, and it is locally consistent, as any node expanded from K does not have a smaller key than that of K.

Figure 2 is an example showing what happens when the weight of any link arbitrarily changes. In this case, the weight of the link EH increases by 10. To adapt to this change, we first check the estimates (*g*, *rhs*) of the nodes around the Link EH, which have the most potential to be affected by this change. The nodes closest to this link are nodes E and H. Although node E is not affected by this change, both the start distance and *rhs*-value of node H change. After updating, the start distance changes to 44 and the *rhs*-value changes to 34. The next step is to update node K, since it has become locally inconsistent. So far, node G has been popped from the priority queue. By expanding nodes G to H and J, the search is led to the current shortest path without visiting many unnecessary nodes that are not affected by the changes. The LPA* can reuse the calculation result coming from the last search and facilitate faster route recalculation by incrementally updating the locally inconsistent nodes.

The main advantage of LPA* is the capability of carrying forward the start distances (*g*-value) and reusing them from search to search. Although LPA* can efficiently manage dynamic environments, it cannot deal with the cases where the start node changes its position over time. While a mobile user is moving along the

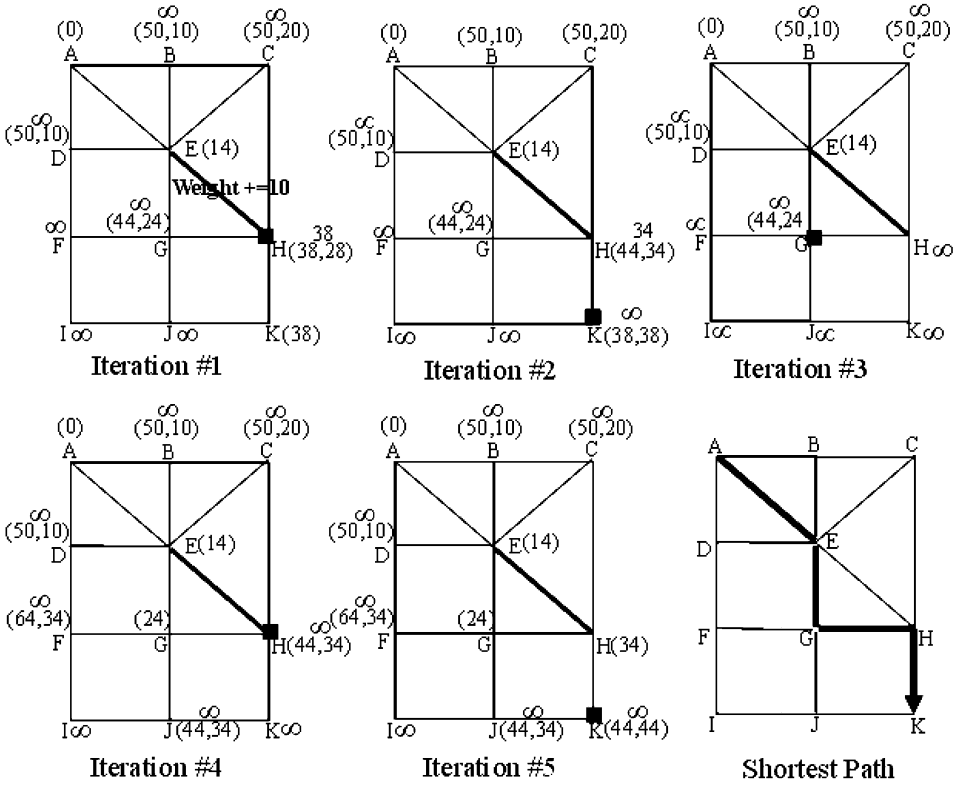

Figure 2. LPA* second search.

previous shortest path and querying new optimal routes to adapt to changes in the environment, the LPA* is not able to perform an incremental search as the start distances (g-value) are no longer valid for the current start node. With the current configuration, it is impossible to rebuild the g-values for these nodes unless an independent search is performed from scratch which loses the power of LPA*.

4. Proposed algorithm

The proposed algorithm consists of three components: (1) an improved LPA* for computing dynamic shortest paths; (2) a new search heuristic; and (3) the Minimum Bounded Rectangle (MBR) constrained shortest path search. Each of the three components is described in detail below.

4.1 *An improved LPA* for moving object*

Since the start distance (g-value) of a node is very important in LPA*, in order to utilize the strength of LPA*, the key problem is how to retain the start distance of a node calculated in previous searches. Observe that the goal node always remains the same whereas the start node changes for a moving object. Based on this observation, we can modify the original LPA* as follows: when a user plans to move from node v to w (v, $w \in S$) and computes the shortest path between these two nodes, we can switch the search direction when computing the shortest path. That is, instead of searching from node v to w, we treat w as the source node and search from w to v.

After this switch, the source node at the root of the shortest path tree no longer changes; only the goal node (the original start node) changes. Hence, the labelled start distances of all nodes can be carried forward to subsequent searches and can be reused from search to search.

Apparently, the value of $h(n)$ associated with each node should be modified accordingly after the switch. No matter what kind of metric $h(n)$ employs as a guide for the heuristic (either the Manhattan distance or Euclidean distance), it is easy to update for each node in the priority queue. Note that one should apply the opposite link weight to the search process in a directed graph when updating the labelled start distance of each node, to ensure the result shortest path start from the source node with changing locations to the goal node.

We illustrate the improved LPA* using the application described below (figures 3 and 4). The graphs are the same as those shown in figures 1 and 2. Imagine that a mobile user wants to move from node A to node K. In the first LPA* search shown in figure 3, we follow the reverse direction to perform a LPA* search from K to A and use this search to acquire the labelled distances for all nodes involved.

As shown in figure 4, a mobile user begins moving along a designed optimal route when they initiate the trip. When the user reaches node E, they are informed, through real-time traffic condition information services, that there is a traffic jam along link EH. In this graph, this traffic jam is represented as an increase of 10 in the weight of link EH. To determine the optimal path after this change, we must update the g-value and rhs-value for node E because its g-value depends on the weight associated with link EH. In addition, the labelled distances associated with the nodes in the shortest path tree between nodes E and H have to be updated. After these updates, node G is popped up from the priority queue, and expanded to E. A new

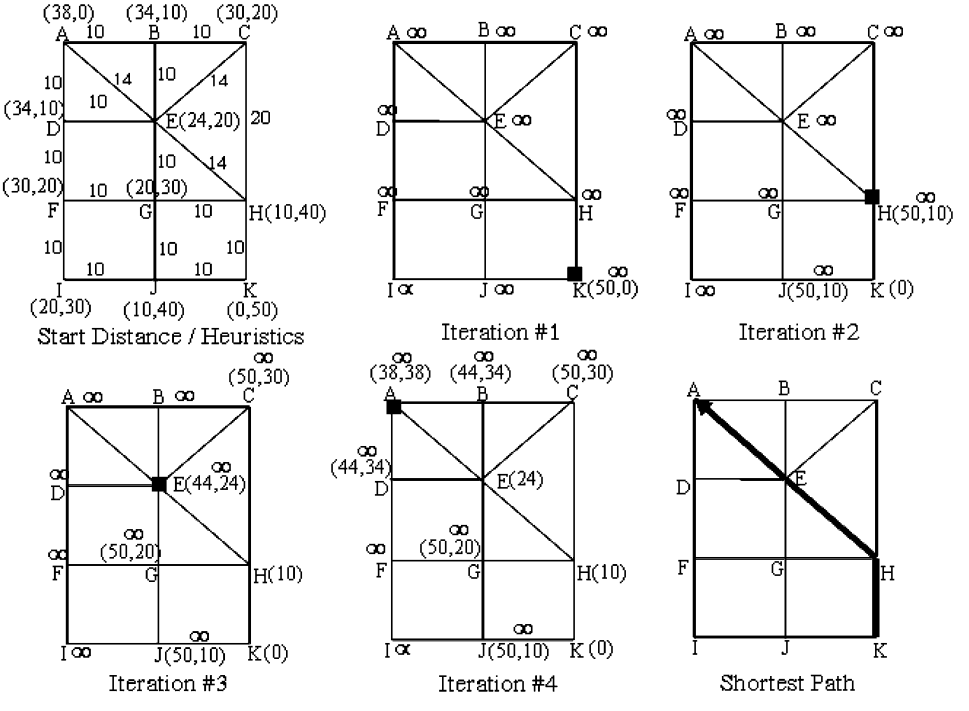

Figure 3. Improved LPA* first search.

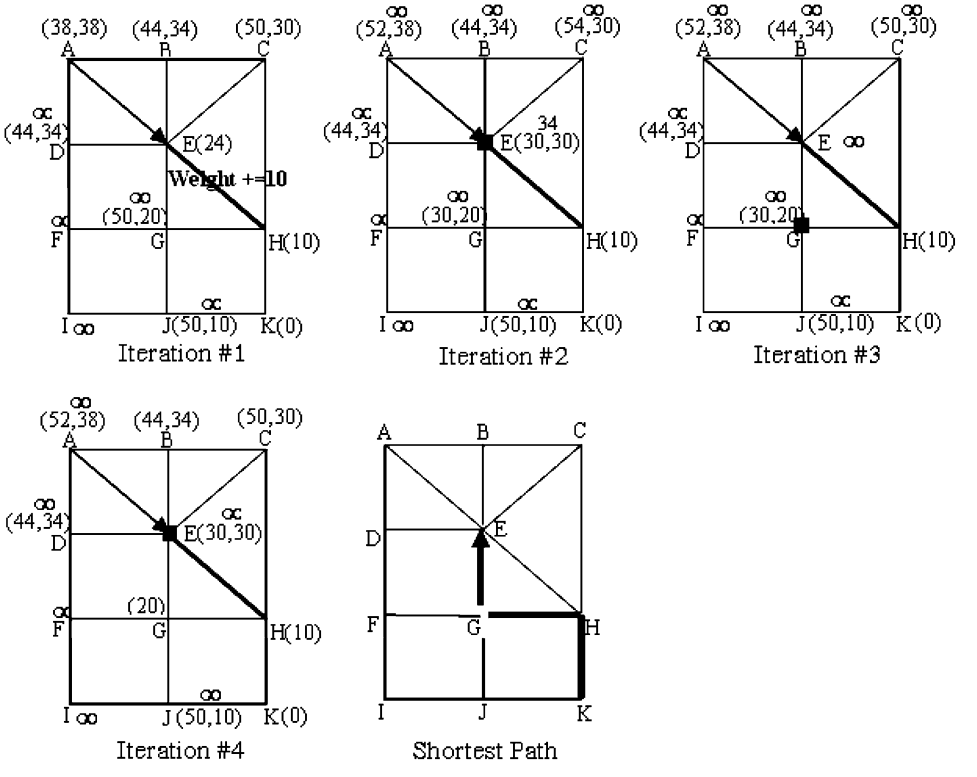

Figure 4. Improved LPA* second search.

route is determined successfully, and the route has been re-calculated. Table 1 provides the detailed pseudo-code of the improved LPA* algorithm.

4.2 *A new search heuristic*

A path cost estimator in a graph is a function $f(u, v)$ which is used to compute an estimated cost of an optimal path between two nodes u and v in the graph. As discussed earlier, a path cost estimator is a perfect estimator if it computes the exact cost of an optimal path between two given nodes. A path cost estimator is admissible if it always underestimates the cost of the path. The straight-line distance between two nodes u and v is always less than their network distance based on the theorem of triangle inequality. This distance thus always underestimates the cost of the path (u, v), and it is often used in A* to guide the heuristic-based search. We use h_1 to denote this distance in this discussion.

We propose another heuristic to enhance the intelligence in node selection in the algorithm. This new heuristic is based on the observation that a straight-line path is always the shortest path between the start node and the goal node if such a path exists. However, in reality, a straight-line shortest path hardly exists in a real transportation network. Nevertheless, human intuition suggests that although the straight-line path does not exist, the actual shortest path is usually close to the virtual straight-line between two nodes in most cases. This situation indicates, when selecting the next node to be examined in the search process, that it is better to select the nodes that are closer to the virtual straight line because these nodes have a

Table 1. Improved LPA* with consideration of the moving start point.

The pseudo-code uses the following functions to manage the priority queue:
U.TopKey() returns the smallest priority of all nodes in priority queue U. (If U is empty, then U.TopKey() returns $[\infty; \infty]$.)
U.Pop() deletes the node with the smallest priority in priority queue U and returns the node.
U.Insert(s, k) inserts node s into priority queue U with priority k.
U.Remove(s) removes node s from priority queue U.
Swap(s_{start}, s_{goal}) switch the start and goal node to perform reversed search

```
Procedure CalculateKey(s)
return [min(g(s), rhs(s))+h(s);min(g(s), rhs(s))];
Procedure Initialize()
U=φ;
for all s ∈ S rhs(s)=g(s)=∞;
rhs(s_start)=0;
U.Insert(s_start, [h(s_start); 0]);
Procedure UpdateNode(u)
if (u≠s_start) rhs(u)=Min_{s'∈pred(u)}(g(s')+c(s', u));
if (u ∈ U) U.Remove(u);
if (g(u)≠rhs(u)) U.Insert(u, CalculateKey(u));
Procedure ComputeShortestPath()
while (U.TopKey()'<CalculateKey(s_goal) OR rhs(s_goal)≠g(s_goal))
{
    u=U.Pop();
  if (g(u)>rhs(u))
        {
          g(u)=rhs(u);
            for all s ∈ succ(u) UpdateNode(s);
  }
  Else
  {
          g(u)=∞;
          for all s ∈ succ(u) ∪ {u} UpdateNode(s);
  }
    }
    Procedure Main()
Initialize();
Swap(s_start, s_goal);
while (s_start≠s_goal)
{
          ComputeShortestPath();
              s_start=Top(Pathlist).next
              Move to s_start
              Detect the weight change in graph
              If any change occurs
              {
                for all directed links (u, v) with changed link costs
                {
                  Update the link cost c(u, v);
                  UpdateNode(v);
            }
              for all s ∈ U
              {
                  U.Update(s, CalculateKey(s));
            }
          }
}
```

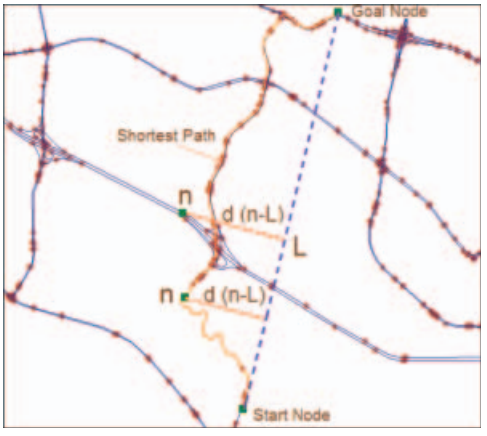

Figure 5. Distance used in the new search heuristic.

greater possibility of lying on the shortest path rather than those that are farther away from the virtual line.

Based on this assumption, the new heuristic we proposed is the distance between a node and the virtual straight line as shown in figure 5. This new heuristic is defined as follows: assume that a virtual straight line L connecting the start node and the goal node, $d\ (n-L)$ is the Euclidean distance from any node n to L. This Euclidean distance is used as the new heuristic denoted as h_2. As demonstrated below, when combined with h_1, this new value is used in the new heuristic to help select nodes to be examined that are closest to the trajectory of the shortest path.

To avoid overestimating the distance from each node to the goal node, we cannot directly apply the formula $f\ (n)=g\ (n)+h_1\ (n)+h_2\ (n)$ as an estimator in the algorithm and let it select the node with the lowest $f\ (n)$ as the next node to be examined. The reason is that $h_1\ (n)+h_2\ (n)$ may be far greater than the actual network distance from the node to the goal node. Thus, the algorithm may overestimate the goal distance—the distance from a node to the goal node—too much and incur low accuracy in the computation of a shortest path.

To solve this problem, we use two levels of ordering to prioritize the nodes to be examined in a priority queue. At the top level, nodes are prioritized in the same way as the original A* algorithm based on values of $f\ (n)$ computed from the formula $f\ (n)=g\ (n)+h_1\ (n)$. At the second level, nodes in the priority queue with similar f values are grouped first and then prioritized based on values of $h_2\ (n)$. In each group of nodes, the largest f value is only greater than the smallest one by 5%. The nodes in each group are then reordered based on their h_2 values. The values of h_2 are only used to fine-tune the priority queue between the nodes with similar f values in each group.

The equation of distance calculation between a point and straight line is used to determine the value of $d\ (n-L)$ for each node to be examined. Assume that the equation of the straight line that connects the start node $(x1,\ y1)$ and the goal node $(x2,\ y2)$ is $ax+by+c=0$; then one can calculate the value of $d\ (n-L)$ using equation (1).

$$d_{(n-L)}=\frac{|ax+by+c|}{\sqrt{a^2+b^2}} \tag{1}$$

where

$$a = y2 - y1 \quad b = x1 - x2 \quad c = -b * y1 - a * x1 \tag{2}$$

Although $\sqrt{a^2 + b^2}$ involves power and evolution functions in its computation, it needs to be computed only once for each start node. For each node, $|ax + by + c|$ can be computed in linear time; this computation is simple and does not increase the computational time markedly. The pseudo-code of the algorithm with the proposed new heuristic is given in table 2. The priority queue is implemented as an Open list. The Closed list contains all examined nodes.

4.3 *Minimum bounded rectangle (MBR) constrained shortest path search*

To further improve the efficiency of the proposed algorithm, we develop a constrained search using the Minimum Bounded Rectangle (MBR) to reduce the search space of the improved LPA*. In a typical traffic network, each link (or road segment) is connected only to the neighbouring nodes (e.g. intersections), and the travel time on a link is generally correlated with its length. This attribute allows the search area to be constrained within a specified area surrounding the origin and destination nodes. The nodes outside this area are assumed to have little probability of being on the shortest path and therefore can be dismissed without any further examination during the search process (Fu *et al.* 2006).

Since an ellipse is the simplest geometric shape that we can employ besides a circle to deal with distance (Li *et al.* 2005), we first discuss how to reduce the search space of the improved LPA* using some features of an ellipse. An ellipse is the trajectory of a point whose combined distance to two specific locations (i.e. the two foci of the ellipse) is fixed, and the distance is equal to the length of its principal axis. An important characteristic of an ellipse is that all points within the ellipse are closer to the two foci than those on its boundary, and all points outside an ellipse to the two foci are farther than those on its boundary.

In figure 6, if we know the network distance d between two nodes s and g in a graph, then we can use d as the length of the principal axis and use the locations of the two nodes s and g to position the foci to construct an ellipse. We can assert that if there is a shortest path between s and g, then this path lies within the ellipse. To prove this, we assume a node v belongs to the shortest path of s and g, and that v is located outside the ellipse; we then have $|sv| + |vg| > d$. Therefore, even if there exist straight-line paths between s and v as well as v and g, the length of this path must be greater than d. Therefore, node v does not belong to the shortest path between s and g. Based on this theorem, we can use an ellipse to prune the nodes that cannot possibly be on the shortest path.

The next problem that we need to solve is to find a way of determining the size of the ellipse. Recall that in the search process of the improved LPA*, we can derive the network distance between the start node and the goal node based on dynamically updated weights of the links in the network. It is possible to define an ellipse using this network distance as the principal axis, and employing the start and goal nodes as the foci. Thus, if there exists a shorter alternative path between the start node and the goal node, then all nodes on the path must lie within the ellipse. Any node outside this ellipse can be safely pruned from the search space, and thus the efficiency in the search process of the improved LPA* is improved.

Table 2. Improved LPA* with the new heuristic

```
Procedure ComputeShortestPath()
{
    Initialize(all nodes);
    source->cost=0;
    Open.insert(source);
    Closed=Null;
    a=y_goal-y_source;
    b=x_source-x_goal;
    c=-b* y_source-a* x_source;
    c1=sqrt(a*a+b*b);   //get square root value
    While (not Open.empty())
    {
      u=Open.pop();
      Closed.insert(u);
      If (u=goal) return true;
      For all v ∈ successor (u)
      {
        g=u->g+cost (u, v);
        v->h1=Euclidian Distance (v, goal);
        v->h2=abs(a*x_v+b*y_v+c)/c1;   //get absolute value
        v->f1=g+v->h1;
        v->f2=g+v->h2;
        If (v->g>g)
        {
          v->parent=u;
          v->g=g;
          v->f1=v->g+v->h1;
          If (v ∈ Open) Open.remove(v);
          If (v ∈ Closed) Closed.remove(v);
        }
        Else continue;
        Open.insert(v);
      }
    }
}
Procedure Open.insert (node)
{
    While (not Open.empty())
    {
      n=Open->current;
      If (node->f1-n->f1<n->f1*5%) compare (node->f2, n->f2);
      Else compare (node->f1, n->f1);
      If (priority is determined)
      {
        Insert (node);
        Return;
      }
    }
}
```

In actual computation, the direct use of an ellipse's boundary as the bounded area of search is less efficient because it involves computation procedures associated with too many power and evolution functions. To solve this problem, we utilize the Minimum Bounded Rectangle (MBR) of the constrained ellipse as shown in figure 7 to simplify the calculations. The MBR for a given ellipse can be computed as

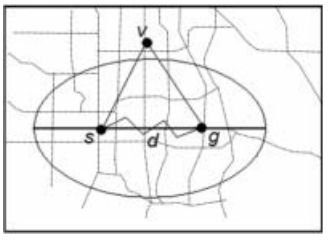

Figure 6. Shortest path search space constrained by an ellipse.

follows. Suppose an ellipse has two foci ($x1$, $y1$) and ($x2$, $y2$) and a principal axis, the ellipse can be represented by equation (3). If partial derivatives of x and y for the ellipse equation are used, then we can obtain the extreme values of x_m and y_m from equation (5). The ellipse is bounded by the MBR whose corner coordinates are the x_ms and y_ms. The pseudocode of the improved LPA* with the MBR constrained search is illustrated in table 3.

$$\frac{[\cos\theta(x-x_c)+\sin\theta(y-y_c)]^2}{a^2}+\frac{[-\sin\theta(x-x_c)+\cos\theta(y-y_c)]^2}{b^2}=1 \tag{3}$$

where

$$\theta=\arctan\left(\frac{y_2-y_1}{x_2-x_1}\right) \quad x_c=\frac{x_1+x_2}{2} \quad y_c=\frac{y_1+y_2}{2} \quad c=\frac{\sqrt{(y_2-y_1)^2+(x_2-x_1)^2}}{2} \tag{4}$$
$$b=\sqrt{a^2-c^2}$$

and

$$x_m=x_c\pm\sqrt{a^2\cos^2\theta+b^2\sin^2\theta} \quad y_m=y_c\pm\sqrt{a^2\sin^2\theta+b^2\cos^2\theta} \tag{5}$$

5. Experimental results

To examine the suitability and performance of the proposed algorithm, we used PARAMICS to simulate real-time traffic conditions using road networks from Calgary in Canada (figure 8) and Singapore (figure 9). PARAMICS is a suite of high-performance software tools used to model the movement and behaviour of vehicles in urban and highway networks. The accurate simulation of drivers and vehicles enables PARAMICS to build a complex picture of traffic conditions and help us obtain dynamic traffic conditions that are close to those in the real world.

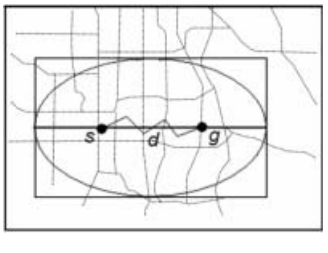

Figure 7. Minimum Bounded Rectangle (MBR) constrained shortest path search.

Table 3. Final version of the improved LPA*

```
Procedure UpdateNode(u)
if (not check(u,MBR)) return;
if (u≠s_start) rhs(u)=Min_{s'∈pred(u)}(g(s')+c(s', u));
if (u ∈ U) U.Remove(u);
if (g(u)≠rhs(u)) U.Insert(u, CalculateKey(u));

Procedure Main()
Initialize();
Swap(s_start, s_goal);
while (s_start≠s_goal)
{
    ComputeShortestPath();
  s_start=Top(Pathlist).next
  Move to s_start
  Detect the weight change in graph
    If any change occurs
    {
      calculate_MBR(s_start, s_goal);
    for all directed links (u, v) with changed link costs
      {
        Update the link cost c(u, v);
        UpdateNode(v);
      }
      for all s ∈ U
      {
        U.Update(s, CalculateKey(s));
    }
  }
}
```

PARAMICS also provides an Application Programming Interface (API) that enables users to customize the system to meet different needs. The Calgary network contains about 2000 nodes and 2500 links. The Singapore network consists of about 6000 nodes and 8000 links.

We used a desktop PC with 1 GB of RAM running Windows XP for the simulations. For each combination of road network and comparison discussed below, we took 1000 shortest-path queries for a randomly selected pair of start and goal nodes, and obtained the number of nodes examined in the computation of a shortest path related to each query. In addition, we recorded the cardinality of each computed shortest path as well as the computer running time for obtaining each shortest path. We compared the improved LPA* with or without the MBR constrained search against the A* using results from the experiments. The reason that we used the A* instead of the LPA* for the comparison is because the performance of the first search of LPA* is the same as that of A* if the new heuristic is not applied (Likhachev and Koenig 2002).

We first selected different routes and used their approximate lengths to examine to what extent the improved LPA* is superior to A*. Table 4 summarizes the experiment results. Recall that the cardinality of a shortest path refers to the number of nodes on the final shortest path. In general, the difference in cardinality is directly related to the difference in the length of a shortest path. Paths with longer lengths may contain more nodes. The number of nodes in the paths shown in table 4 varied from about 11 to 51. In this experiment, only a few links (<5% of the links) were set

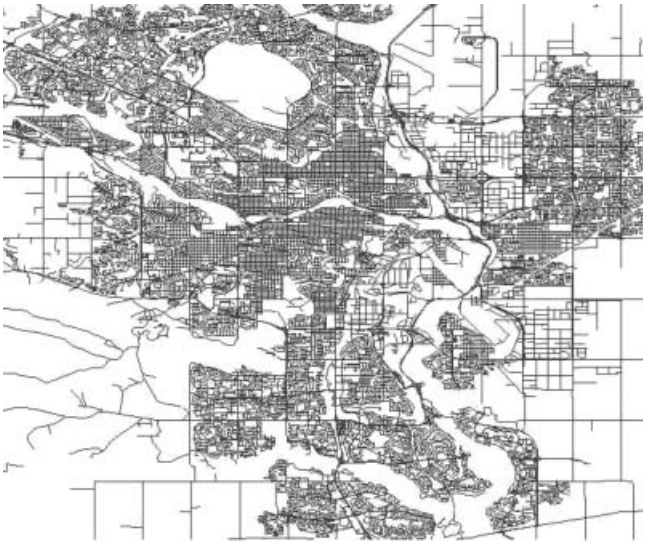

Figure 8. Road network of Calgary.

to have changing weights. As can be seen from the table, the number of examined nodes in the improved LPA* is significantly lower than that of the A* algorithm. The improved LPA* algorithm with the MBR constrained search has the least number of examined nodes in all cases. In some cases, the number of examined nodes in the improved LPA* algorithm with the MBR constrained search is only about 10% of that in the A* algorithm.

We had similar results when we set the experiments to have a modest proportion of links with changing weights (table 5). Table 5 lists the experiment results in a dynamic network with 10% of the links having changing weights.

We conducted additional experiments to examine the performance of the algorithms under different proportions of links with changing weights. The proportions varied from 5% to 40%. Figure 10 illustrates the number of nodes examined in a dynamic network with different proportions of links with changing weights for a route containing about 40~50 nodes. The results shown in figure 10 clearly indicate that the improved LPA* algorithm with the MBR constrained

Figure 9. Road network of Singapore.

Table 4. Comparison of the number of examined nodes in the A* and the improved LPA* algorithms in computing shortest paths in a dynamic network with a few links changing weights.

	Cardinality of the path				
Approach	11	19	33	43	52
Original A*	34	75	156	284	497
Improved LPA* without the MBR constrained search	7	20	32	73	128
Improved LPA* with the MBR constrained search	5	16	25	55	98

search had significantly fewer examined nodes in computing the shortest paths compared with the A* algorithm in all proportions. The results in figure 10 also suggest that the number of examined nodes in the improved LPA* algorithm with the MBR constrained search tends to increase and approach that of the A* algorithm when the proportion of links with changing weights increases.

Table 6 illustrates the experimental results on the performance of the improved LPA* and the A* in computing shortest paths with various cardinalities. In this experiment, the constrained MBR is not applied. From table 6, we can observe the average percentages of computational time gain of the improved LPA* over A* in computing different shortest paths. The results demonstrate that the improved LPA* can save 24~28% computational time, and the improvement of the proposed algorithm is more noticeable in longer paths. The reason for this situation is that, for a given network, when path length increases, the number of nodes involved in the computation of the shortest path also increases. Because the computational time of the improved LPA* increases linearly with respect to the number of examined nodes, and that of A* increases in a nonlinear manner, the improved LPA* algorithm saves more computational time when the number of examined nodes increases.

As a cost of this improvement, the proposed algorithm does not always guarantee that the optimal shortest path can be found, yet the difference is minor. In general, the accuracy is 99% (table 7). This means that the final path derived from the improved LPA* with a new heuristic may be longer than the actual optimal shortest path by 1%. Therefore, this error can generally be ignored in real applications.

6. Conclusions

This paper reports the development of a novel approach that extends the existing Lifelong Planning A* algorithm (LPA*) to solve the dynamic shortest-path problem

Table 5. Comparison of the number of examined nodes in the A* and the improved LPA* algorithms in computing shortest paths in a dynamic network with a modest proportion of links (10%) changing weights.

	Cardinality of the path				
Approach	11	19	33	38	55
Original A*	32	78	165	280	512
Improved LPA* without the MBR constrained search	9	28	37	72	134
Improved LPA* with the MBR constrained search	6	19	26	59	104

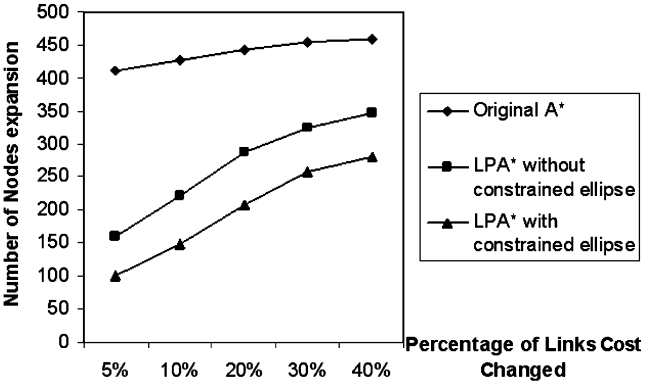

Figure 10. Number of examined nodes versus different proportions of links with changing weight values.

in navigation services where users may have to repeatedly calculate the optimal shortest path between two nodes while travelling in a dynamic transportation environment. The improved LPA* provides three main improvements over the LPA*. First, we extended the LPA* to retain all previously computed information on a shortest-path tree by switching the search direction relative to the source node and the goal node while computing the dynamic shortest path in a network with changing link weights. This switch enables the algorithm to retain all information associated with nodes on the shortest path tree that was obtained in previous steps. Second, we introduced a new search heuristic to speed up the search process. The heuristic computes the Euclidean distance from a candidate node to be examined to a straight line connecting the location of the moving object (the start node) and the goal node, and uses the distance as a heuristic to fine-tune the priority queue used in the LPA* algorithm to speed up the selection of the next node to be examined. This fine-tuned selection process avoids accessing many unnecessary nodes and thus makes the computation of the shortest path converge to the goal node more quickly. Third, we developed a Minimum Bounded Rectangle (MBR) constrained search based on features of an ellipse to further restrict the search space of nodes to be examined and hence improve the performance of the improved LPA*.

Our experiment results indicate that the number of examined nodes in the improved LPA* could be up to 70~80% (according to figure 10) less than that of the A* algorithm. This improvement leads to approximately 17–31% savings in computational time when calculating shortest paths using the test road networks in the experiments.

Table 6. Number of nodes examined in paths with different cardinalities.

	Cardinality of the path			
Method	<=20	20~40	40~60	>60
Original A*	264	416	784	968
Improved LPA* with new heuristic	219	324	581	667
Ratio (improved/original) (%)	83	78	74	69
Average percentage improvement	17	22	26	31

Table 7. Accuracy comparison between the original A* and the improved LPA* with the new heuristic

	Original A*		Improved LPA* with new heuristic			
Experiments	No. of nodes examined	Final length	No. of nodes examined	Final length	Performance gain (%)	Accuracy loss (%)
1	1418	18 583.05	1012	18 760.36	28.6	0.95
2	581	6471.96	488	6493.61	16.0	0.33
3	850	15 129.45	662	15 256.07	22.1	0.84
4	1027	20 752.14	787	20 980.47	23.3	1.10
5	2265	24 792.12	1807	25 073.60	20.2	1.13
6	1557	17 879.13	1310	17 934.21	15.8	0.30
7	544	10 501.73	432	10 620.13	20.6	1.13
8	750	15 474.96	624	15 612.41	16.8	0.89
9	4265	37 055.57	3439	37 485.23	19.4	1.15
10	1846	16 137.57	1495	16 261.42	19.0	0.77
Average					20.2	0.86

References

Chabini, I., 1998, Discrete shortest path problems in transportation. *Transportation Research Record*, **1645**, pp. 170–175.

Dijkstra, E.W., 1959, A note on two problems in connection with graphs. *Numeriche Mathematik*, **1**, pp. 269–271.

Dreyfus, S.E., 1979, An appraisal of some shortest path algorithms. *Operations Research*, **17**, pp. 395–412.

Frigioni, D.A., Marchetti-Spaccamela, D. and Nanni, U., 2000, Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, **34**, pp. 351–381.

Fu, L., Sun, D. and Rilett, L., 2006, Heuristic shortest path algorithms for transportation applications: state of the art. *Journal of Computers & Operations Research*, **3**, pp. 3324–34343.

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

Huang, B. and Li, H.G., 2004, Developing location-aware navigation guides that use mobile Geographic Information Systems. *Transportation Research Record*, **1879**, pp. 108–113.

Huang, B. and Wu, Q., 2007, A spatial indexing approach for high performance location based services. *Journal of Navigation*, **60**, pp. 1–11.

Koenig, S., Likhachev, M. and Furcy, D., 2004, Lifelong Planning A*. *Artificial Intelligence*, **155**, pp. 93–146.

Li, H.G., Huang, B., Lu, H. and Huang, Z.Y., 2005, Two ellipse based pruning methods for group nearest neighbor queries. In *13th ACM International Symposium on Advances in Geographic Information Systems*, 4–5 November, Bremen, Germany.

Likhachev, M. and Koenig, S., 2002, Incremental replanning for mapping. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, pp. 667–672.

Pallottino, S., 1984, Shortest-path methods: complexity, interrelations, and new propositions. *Networks*, **14**, pp. 257–267.

Pallottino, S. and Scutella, M.G., 1998, Shortest path algorithms in transportation models: Classical and innovative aspects. In *Equilibrium and Advanced Transportation Modelling*, P. Marcotte and S. Nguyen (Eds), pp. 245–281 (Amsterdam: Kluwer).

PALLOTTINO, S. and SCUTELLA, M.G., 2003, A new algorithm for reoptimizing shortest paths when the arc costs change. *Operations Research Letters*, **31**, pp. 149–160.

PREYGEL, A., 1999, Pathfinding: A comparison of algorithms. Available online at: http://www.cpcug.org/user/scifair/Preygel/Preygel.html (accessed on February 1, 2005).

ZHAN, F.B. and NOON, C.E., 1998, Shortest path algorithms: an evaluation using real road networks. *Transportation Science*, **32**, pp. 65–73.