A Border Line-Based Pruning Scheme for Shortest Path Computations

JinKyu Park[1], Daejin Moon[2] and Eenjun Hwang[2]
[1]Security Research Dept., LG Electronics
19-1 Cheongho-ri Jinwi-myeon Pyeongtaek-si Gyeonggi-do, Korea
[e-mail:jinkyu.park@lge.com]
[2]School of Electrical Engineering, Korea University
Anam-dong, Seongbuk-gu, Seoul, Korea
[e-mail: {wizardyk, ehwang04}@korea.ac.kr]
*Corresponding author: Eenjun Hwang

Received February 25, 2010; revised July 12, 2010; revised September 2, 2010; accepted October 6, 2010; published October 30, 2010

Abstract

With the progress of IT and mobile positioning technologies, various types of location-based services (LBS) have been proposed and implemented. Finding a shortest path between two nodes is one of the most fundamental tasks in many LBS related applications. So far, there have been many research efforts on the shortest path finding problem. For instance, A* algorithm estimates neighboring nodes using a heuristic function and selects minimum cost node as the closest one to the destination. Pruning method, which is known to outperform the A* algorithm, improves its routing performance by avoiding unnecessary exploration in the search space. For pruning, shortest paths for all node pairs in a map need to be pre-computed, from which a shortest path container is generated for each edge. The container for an edge consists of all the destination nodes whose shortest path passes through the edge and possibly some unnecessary nodes. These containers are used during routing to prune unnecessary node visits. However, this method shows poor performance as the number of unnecessary nodes included in the container increases. In this paper, we focus on this problem and propose a new border line-based pruning scheme for path routing which can reduce the number of unnecessary node visits significantly. Through extensive experiments on randomly-generated, various complexity of maps, we empirically find out optimal number of border lines for clipping containers and compare its performance with other methods.

Keywords: Dijkstra's algorithm, path-finding, shortest path container, pruning method, minimum bounding rectangle, convex hull, border-line

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-C1090-1001-0008)

DOI: 10.3837/tiis.2010.10.014

1. Introduction

There have been many research efforts to solve the shortest path finding problem for graphs and maps. As one of the well-known solutions to the problem, Dijkstra's algorithm finds shortest paths from one node to all the other nodes in the map based on a breadth first search [1]. However, this algorithm may not be appropriate if the goal is to find a shortest path between two specific nodes. Many approaches such as goal directed search, bidirectional search, multi-level map, and pruning have been proposed to handle this problem [2].

The popular A* algorithm performs a goal directed search based on a best-first search paradigm [3][4][5][6]. This algorithm estimates neighboring nodes using a heuristic function and selects one as the closest node to the destination. This improves the search speed by a factor of roughly 1.5 compared to the Dijkstra's algorithm [7].

The bidirectional approach searches the shortest path from the source to the destination and from the destination to the source simultaneously [9][10][11]. Using this method, the search time can be reduced by a factor of 2 [8].

The performance of shortest path finding algorithms can be also improved by reducing their search space. For example, in the multi-level map method, higher level maps contain fewer nodes than lower level ones, which mean that there are fewer nodes to consider at once. This approach improves the search speed by a factor of 11 [11][12][13][14].

In order to prune unnecessary search space, shortest path container has been proposed for each edge of the map to provide routing information as to whether the edge has to be exploited for the shortest path. To construct such shortest path containers, we first need to pre-compute the shortest paths for all node pairs in the map. After that, for each edge, we collect nodes whose shortest path has this edge. So far, various types of shortest path container have been investigated including circle, ellipse and rectangle. Among them, rectangle called Minimum Bounding Rectangle (MBR) was most popular. It was reported that the MBR can improve the search speed by factors in the range between 10 and 20 [15].

There have been several attempts to combine these methods for better performance. For example, the shortest path containers could achieve speedup factors in the range from about 0.5 to 30 when combined with other algorithms. Except with the multi-level map method, its speedup factor is at least 5. On the other hand, the other algorithms and their combinations show the speedup factor in the range from about 0.5 to 5 [2].

According to [2], the shortest path containers method showed the best result as stand-alone and combining speed-up techniques. However, its pruning method would take more time in computing a shortest path than the A* algorithm. In this paper, we investigate this problem and propose a new border-line based pruning method for shortest path containers to reduce the search time significantly.

This paper is organized as follows. In Section 2, we introduce the shortest path containers method briefly and explain the false hit problem in its pruning. In Section 3, we describe how to construct border lines for better pruning. In Section 4, we present some experimental results and finally, in the last section, we conclude this paper.

2. Shortest Path Containers

2.1 Pruning Methods

Intuitively, if we have a database of pre-computed shortest paths for all node pairs, then the shortest path problem can be reduced to just finding an entry for the path from the database using two nodes as indices. However, this approach might suffer from a storage space problem. For example, if there are N nodes in a map, its storage requirement is much greater than N^2. In the shortest path containers method, such information can be stored in a space-efficient way. Each pre-computed shortest path consists of several edges. For each edge, a container is constructed to enclose all the destination nodes whose shortest path includes the edge. Usually, geometric objects such as rectangles are used for containers. This way, the storage requirements can be reduced to O(E) where E is the number of edges in a map.

There are many different types of shortest path containers such as disk center at the tail, ellipse, angular sector, minimum bounding rectangle, convex hull, and so on. Among them, the convex hull is the best container in terms of the number of unnecessary node visits. In fact, experimental results in [15] show that the convex hull minimizes the number of visited nodes. On the other hand, the MBR is best with respect to the search time.

While computing a shortest path between two nodes, these containers are used for pruning edges that will not lead to the destination. In Dijkstra's algorithm, starting from the source node, the neighboring nodes with smallest costs are visited in turn till it arrives at the destination. We can identify edges that lead to the destination from the shortest path containers and thus avoid unnecessary node visits. The containers are geometric objects, and edges that lead to the destination are easily identified by checking whether or not the containers include the destination. Compared to the convex hull, the MBR is much faster in check this and thus gives the best search time even if its pruning power is worse than that of the convex hull. In fact, it is already proved that the shortest path found by this pruning method is always same as the one obtained by Dijkstra's algorithm [15]. In this paper, we call this method MBR-based pruning.

2.2 False Hits in Pruning

The routing time of the shortest path containers method is largely dependent on the shape of the shortest path container. First, the MBR is a good choice in this aspect. The MBR of an edge *e* is a minimum rectangle enclosing all the nodes whose shortest paths pass through the edge *e*. Such nodes can be found, for instance, using Dijkstra's algorithm. These nodes are called *Valid Nodes* (VNs) of the MBR. Since the MBR is a rectangle including all the valid nodes of the edge *e*, it might be possible for the rectangle to include nodes whose shortest path does not pass through the edge. Such nodes are called *Invalid Nodes* (INs) of the MBR. Hence, an MBR of an edge consists of VNs and possibly empty INs. Details on the construction of VNs and INs can be found in [15]. In fact, this classification is just for the convenience of description. That is, we do not know whether a node in an MBR is a VN or not. To find a shortest path from a source *s* to a destination *t*, the pruning method checks whether the neighboring node's (or edge's) MBR includes *t*. Invalid nodes will cause unnecessary explorations to some neighboring nodes which are not on the shortest path to the destination. We call these explorations *false hits*. In the worst case, all neighboring MBRs include *t* but only one MBR includes *t* as a VN and the others as an IN. In this case, the pruning method cannot be of any help. It works just the way Dijkstra's algorithm works.

Fig. 1 shows an example addressing this problem. In the example, VNs are marked in black and INs in white. Suppose that we are searching for a shortest path from s to t. Since the start node s has two edges e_1 and e_2, we can construct two MBRs for s. The large MBR has four VNs v_1, v_2, v_3 and v_4 and one IN t, whereas the small MBR has three VNs v_5, v_6 and t. However, as we mentioned before, we do not know if a node is a VN or not and we only can test if a node is enclosed by the MBR. Therefore, even if the shortest path between s and t (marked as bold arrows) passes through e_2, the routing algorithm explores M_1 and M_2 because they both contain t.

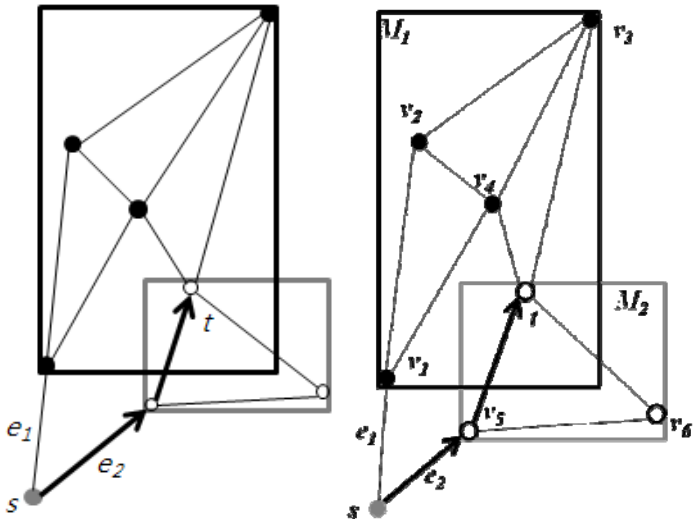


Fig. 1. False hit in the pruning

On a typical map, as the number of edges branching out from a node increases, the total number of INs would also increase. Eventually, this will cause unnecessary node explorations (false hits) and hence increase the routing time. If we can reduce those false hits, we could reduce the routing time, too.

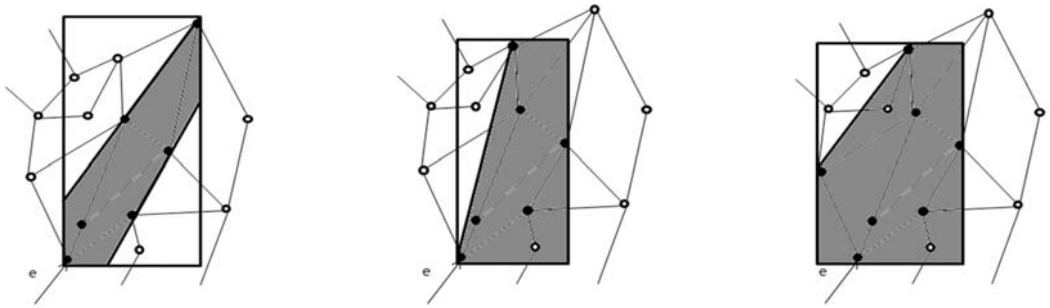
The convex hull can be used to minimize the number of INs in the container. A convex hull for a set of points X in a real vector space is the minimal convex set including all the points in X. However, due to its complexity, it could take more time to check whether the destination is inside the container. In this paper, we propose a new pruning scheme that can reduce the number of INs and the checking time compared to the MBR and the convex hull, respectively. Roughly, we first define a convex hull inside the MBR and then construct border lines based on it to exclude as many INs as possible, without sacrificing the routing accuracy.

3. Construction of Border Lines

3.1 Border Lines

In this section, we describe how to construct an MBR for a set of VNs and generate border lines from its convex hull to minimize the number of INs. For convenience, we divide each MBR into two regions: Valid Region (VR) and Invalid Region (IR). A VR includes all VNs of an MBR and possibly some INs. The rest of the MBR becomes an IR. They are separated by the border lines.

For a set of VNs of an edge *e*, an MBR can be defined using the 2~4 outermost VNs and the exact number is determined by their distribution. For example, **Fig. 2** shows three examples where an MBR is determined by 2, 3, and 4 border nodes. In this figure, nodes in black are VNs and nodes in gray are INs. Also, shaded regions inside the MBR are VRs of the MBRs. **Fig. 2-(a)** shows an example where two diagonal nodes are enough to define an MBR and two border lines are used to define its VR. In this figure, most VNs are distributed along the diagonal band and most INs are located outside the band. We can see that two border lines are enough to exclude most INs. **Fig. 2-(b)** shows an example where 3 nodes and one border line are used to define an MBR and its VR. **Fig. 2-(c)** shows an example where 4 nodes and one border line are used to define an MBR and its VR. Even though we could not remove all the INs completely, we could exclude a significant number of INs using one or two border lines.



(a) 2 border lines in an MBR with 2 border nodes

(b) 1 border line in an MBR with 3 border nodes

(c) 1 border line in an MBR with 4 border nodes

Fig. 2. Illustration to construct MBRs using different number of border nodes

As the shape of a VR gets more complex, it takes more time to check the inclusion of the destination for the VR. Hence, it is not cost-effective to allow arbitrary number of border lines to exclude just a small number of INs. Therefore, in this paper, we will limit the number of border lines allowed for an MBR to k and add border lines only when the ratio of INs to the nodes in the MBR is higher than some threshold ε. However, empirical data shows that $k = 2$ are enough to exclude most INs and adding more border lines will not pay off due to the increased checking complexity.

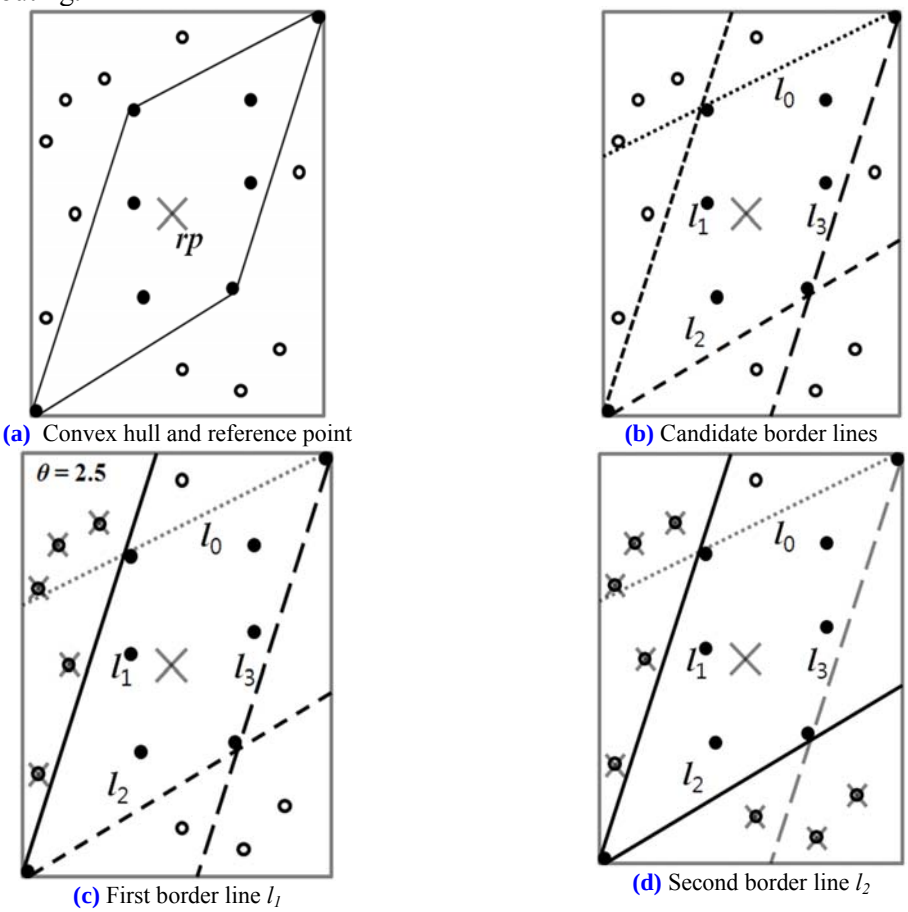
3.2 Border Line Representation

A line can be represented either by $ax + by + c = 0$ or by $y = ax + b$. The former can represent a line parallel to the y-axis, but it requires one more real value than the latter. Even though the latter is not appropriate to represent a line parallel to the y-axis, we use the latter to represent border lines since this problem can be easily solved by using a flag, *fVertical*, to indicate whether it is a vertical line. If this flag is on, the border line is considered as $x = b$. The border line needs another flag, *fUpsideVR*, to indicate which side of the line is the VR.

3.3 Border Line Construction

To construct border lines for an MBR, we use four input values; a list of VNs, a list of INs, the maximum number of border lines k, and a threshold ε. First, we check whether the MBR contains enough INs for creating a border line by calculating the ratio of VNs to all nodes in the MBR. If the ratio is smaller than the threshold ε, we go to the next steps for creating a convex hull for it. Otherwise, we do not create any border line and the MBR itself will be used

for the routing.



(a) Convex hull and reference point

(b) Candidate border lines

(c) First border line l_1

(d) Second border line l_2

Fig. 3. Example of border line construction

In order to construct a convex hull of VNs, we use the Graham Scan method as shown in **Fig. 3-(a)**. This method is known to compute the convex hull for a finite set of points in the plane with time complexity O(n log n). Details on this method can be found in [16].

The convex hull itself can be represented by a set of adjacent vertices. For the convex hull, we calculate a reference point rp by averaging those vertices. This point will be used for judging which side of a border line is the VR. By extending the sides of the convex hull within the MBR, we can define border line candidates. For instance, in **Fig. 3-(b)** we can define 4 candidate lines, $l_0 \sim l_3$. For each candidate line, we also decide its flag, *fUpsideVR*, from the line and the reference point according to their relative position.

To select border lines from the candidate lines, we count the total number of INs that would be excluded by each line. Then, we select the line with the biggest number as the first border line and adjust another threshold θ to half of the number of those excluded INs as in **Fig. 3-(c)**. Next, to avoid counting repeatedly INs which were excluded in the previous step, INs which were excluded by some border lines are ignored when we count INs that will be excluded by the remaining candidate lines. For example, in **Fig. 3-(d)**, l_2 is selected as the second border line because it will exclude 3 INs. Note that θ is 2.5 at this moment. We repeat these steps until we find k border lines or there is no valid candidate line. Detailed steps for generating border lines are described in **Algorithm 1**.

Algorithm 1: Constructing border lines

Input:
VN[n_1]: an array of valid nodes
IN[n_2]: an array of invalid nodes
k: maximum number of border lines
ε : border line creation threshold
Output: border lines BL

```
1:  if sizeof(VN)/sizeof(IN+VN) > ε
2:          return null
3:  vertex[] ← GrahamScan(VN)
4:  rp ← avg(VN)
5:  lines[]← GetLineFromVertex(vertex, rp)
6:  if sizeof(lines) < k
7:          k ← sizeof(lines)
8:  enodes[]← ExcludedNodes(lines, IN)
9:  sort(enodes)
10: θ ← enodes[0] * 0.5
11: border_lines[0] ← lines[enodes[0]]
12: bl_cnt ← 1
13: tot_enodes.add(enodes[0])
14: for i =1 to i =size(enodes) {
15:    old_cnt ← size(tot_enodes)
16:    tot_enodes.add(enodes[i])
17:    if size(tot_enodes) – old_cnt > θ {
18:       border_lines[bl_cnt++] ← lines[i]
19:       if k > bl_cnt then
20:          break
21:    }
22: }
23: return border_lines
```

3.4 Destination in the Valid Region

During the routing step, we need to check whether the destination is in the VR of the edge under consideration. **Algorithm 2** shows the detailed steps for this. For a given set of border lines and a destination, the algorithm first checks the line type. If the line is vertical, it compares the b value of the line and the x value of the destination. If $x > b$, the destination is on the right side of the line, and if *fUpsideVR* is true, which means the right side of vertical line is VR, then the destination is in VR for this line. In addition, if the destination is on left side and *fUpsideVR* is false, the destination also is in VR. If the line is not vertical, it checks the relative position of the destination against the border line using the y coordinate of the destination and $f(x)$. If $y > f(x)$, which means that the destination is above the border line, and *fUpsideVR* is true, then the destination is in VR. If $y < f(x)$ and *fUpsideVR* is false, it also indicates that the destination is in the VR.

If the destination is not in the VR for any border line, it is not in the VR for the set of border lines and the algorithm returns false.

Algorithm 2: Checking node validity

Input:
border_lines: border lines for an edge

```
dst: destination node
Output: true or false
1:    for each border line bl in border_lines {
2:      if bl.fVertical
3:        fValid = !(dst.x >= bl.b) ^ bl.fUpsideVR
4:      else {
5:        fx = bl.a * dst.x + bl.b
6:        fValid = !(dst.y >= fx) ^ bl.fUpsideVR
7:      }
8:      if fValid is false
9:        return false
10:   }
11:   return true
```

Algorithm 3: Routing using border lines

```
Input:
map: node & edge data
src: source node
dst: destination node
BL: list of border lines for each edge
Output: shortest path
1:    for each node v in map {
2:      dist[v] ← infinity
3:      prev[v] ← undefined
4:    }
5:    dist[src] ← 0
6:    Q ← map
7:    while Q is not empty {
8:      u ← GetHighestPriority(Q)
9:      for each neighbor v of u {
10:       if dst ∈ C(u,v) {    /* MBR Container C of edge (u,v) */
11:         if dst ∈ BL(u,v) {  /* BL represents VR of edge (u,v) */
12:           cur_dist ← dist [u] + cost(u,v)
13:           if cur_dist < dist [v] {
14:             dist[v] ← cur_dist
15:             prev[v] ← u
16:   }  }  }  }  }
```

3.5 Routing Algorithm

In this paper, to find out the shortest path between two nodes, we first construct an MBR for each edge in the map and a set of border lines for each MBR in the preprocessing step. In the simple MBR-based routing, pruning can be easily implemented by adding a few lines to Dijkstra's algorithm for checking if the destination node is in its MBRs (line 10 in Algorithm 3). If the destination is in an MBR, our proposed algorithm checks whether it is in its VR using **Algorithm 2** (line 11 in **Algorithm 3**). Since our algorithm is based on the Dijkstra's algorithm, its time complexity is $O(n^2 \log n)$ plus the time to construct the containers. **Fig. 4** depicts overall steps of our routing algorithm.

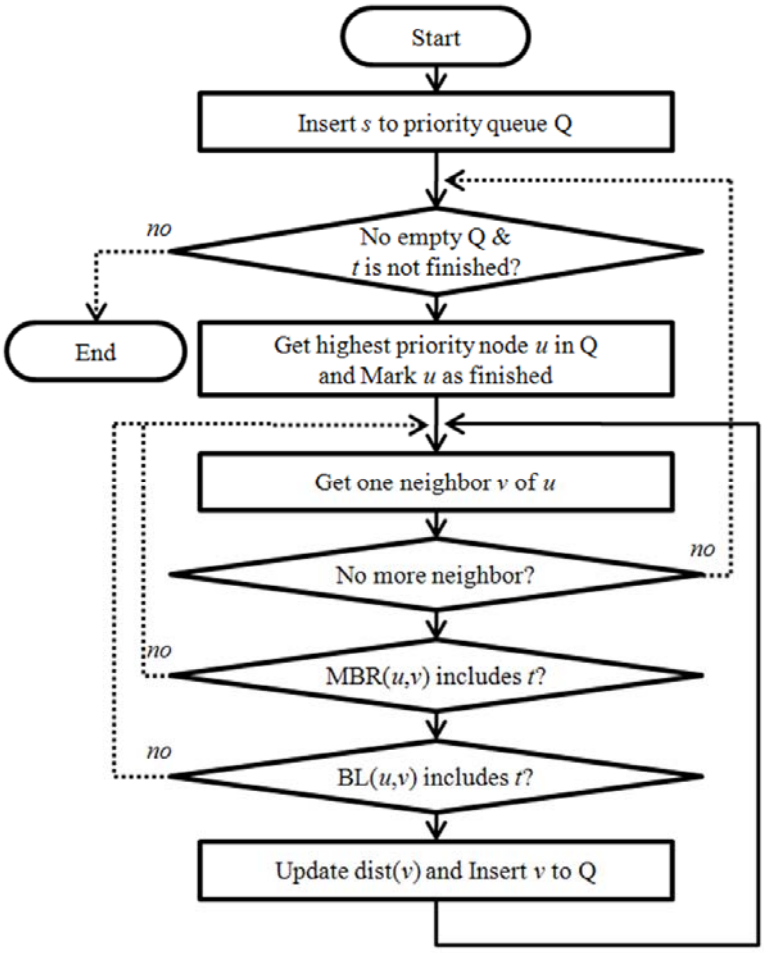


Fig. 4. Flowchart for routing using border lines

4. Experiments

In order to evaluate the performance of our method, we carried out extensive experiments and compared the results with A* and simple MBR-based routing method. These algorithms are implemented in C++ using Microsoft Visual Studio 2005. The platform used in the experiment was a PC with an Intel Pentium D 3.4 GHz and 4 GBytes RAM. We considered 6 input maps with different characteristics as summarized in TABLE I. They are all Waxman random graphs [17] in the XML-based GraphML file format [18]. For all of the shortest paths in each map, we measured the average search time and the number of visited nodes by the three algorithms. **Fig. 5** shows a snapshot of our prototype path routing system. In addition to the BL-based routing, other methods including A* and pure MBR-based pruning are implemented for the comparison.

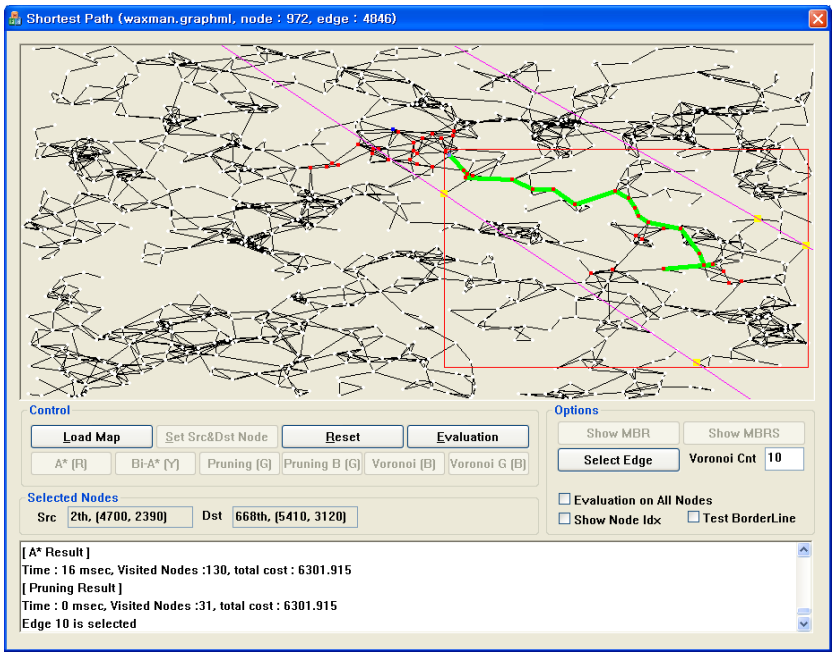


Fig. 5. Snapshot of our prototype routing system

Table 1. Map characteristics

Map #	Nodes	Edges	Paths
1	972	4846	9.4*10^5
2	1180	5792	1.4*10^6
3	1372	6778	1.9*10^6
4	1566	7586	2.5*10^6
5	1782	8678	3.2*10^6
6	1980	9678	3.9*10^6

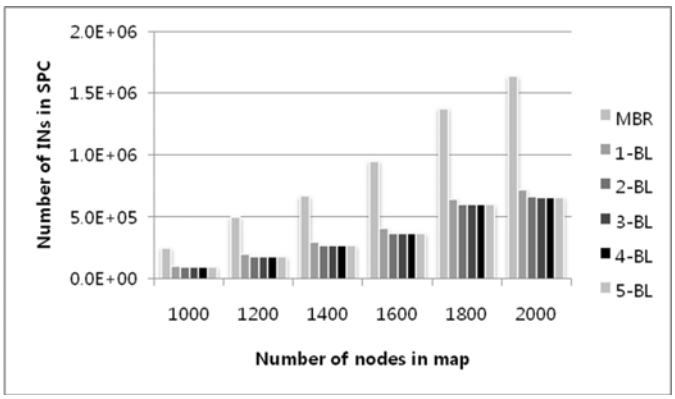
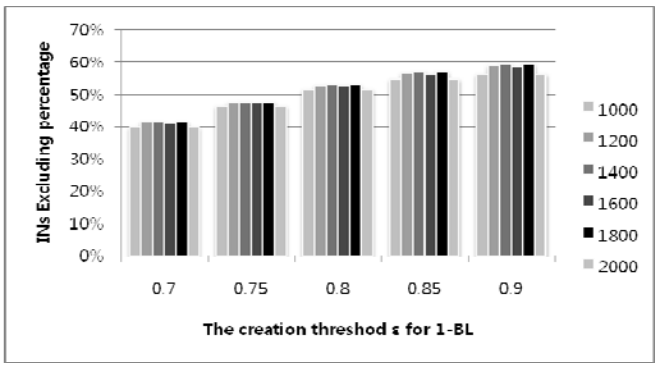
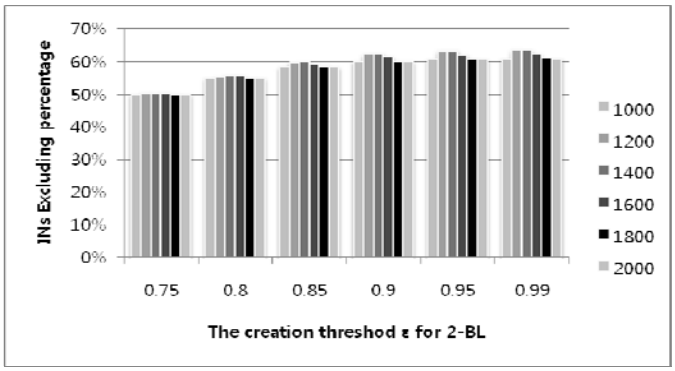


Fig. 6. Number of INs in the shortest path container



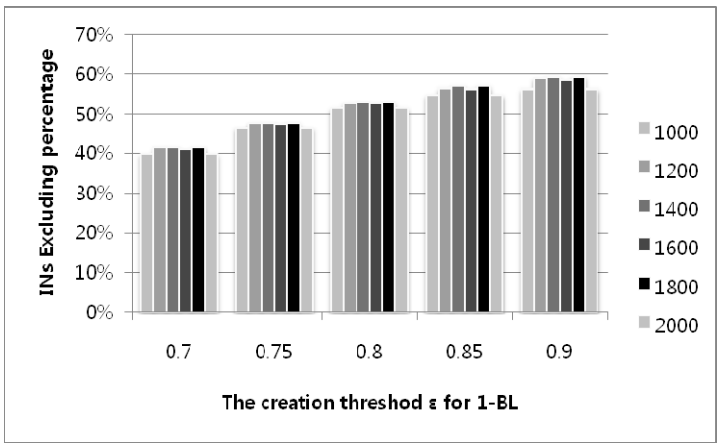
(a) Using 1 border line



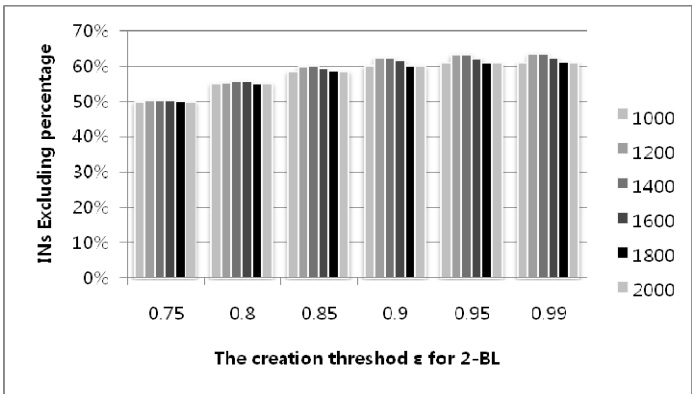
(b) Using 2 border lines

Fig. 7. Exclusion ratio of INs depending on the creation threshold ε

Fig. 6 shows how many INs are filtered out during the routing depending on the number of border lines. Intuitively, more border lines will exclude more INs. However, in most cases, two border lines are enough for this purpose. In the figure, we can observe that 3 or more border lines do not give any benefit. Hence, in the following experiments, we will add at most two border lines for the MBRs.



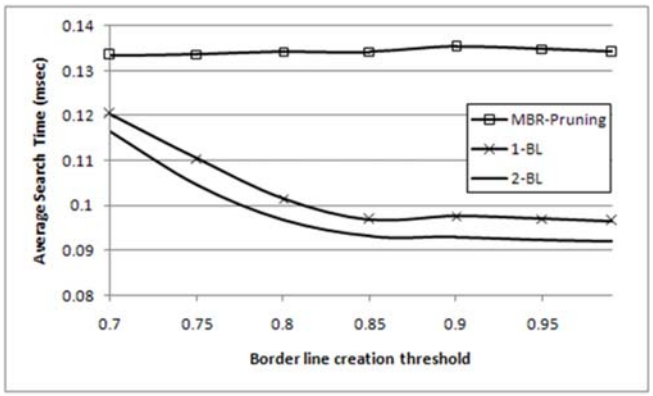
(a) Using 1 border line



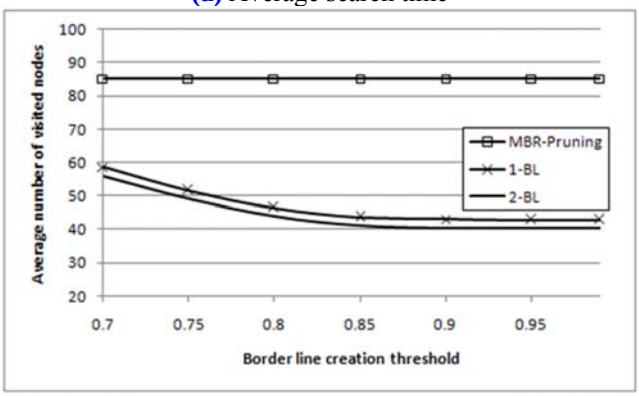
(b) Using 2 border lines

Fig. 7 (a) and **(b)** show how many INs are excluded depending on the value ε when using 1 border line and 2 border lines, respectively.

From the figure, we can see that pruning performance was improved noticeably until ε is to 0.9. Beyond that, it does not show any noticeable improvement. In the next experiment, we try to find out the border line creation threshold ε to exclude the INs cost-effectively.

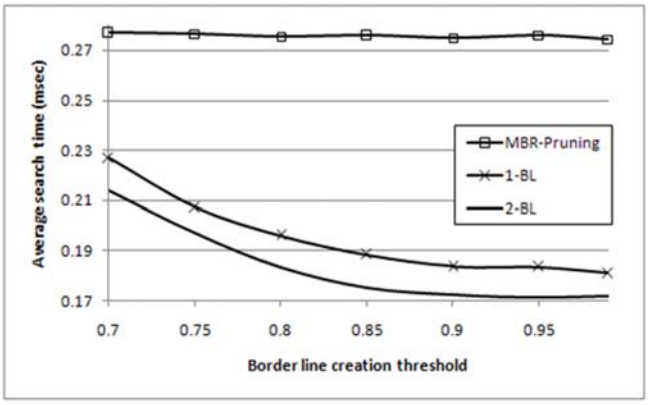


(a) Average search time

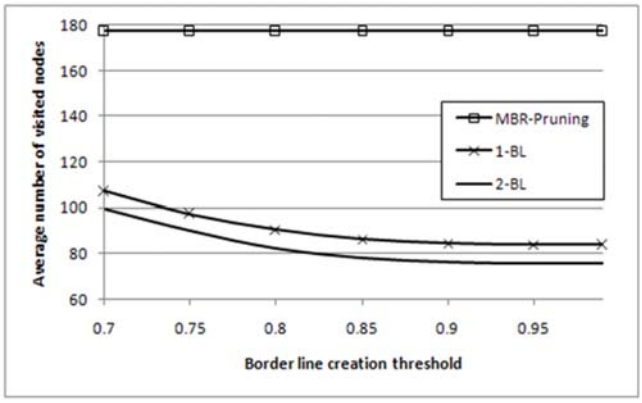


(b) Average number of visited nodes

Fig. 8. Average search time and visited nodes for 1000 node map



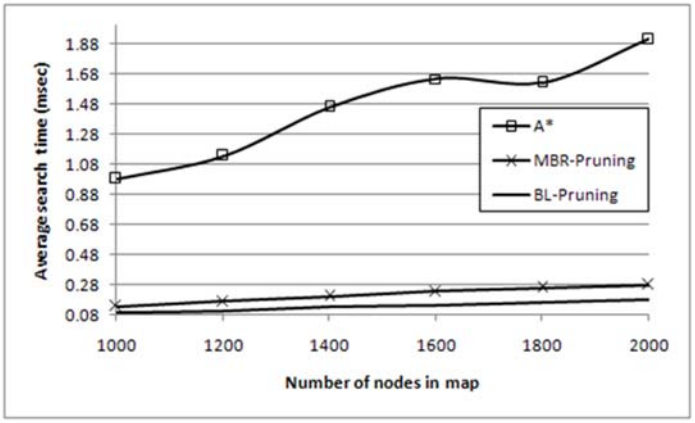
(a) Average search time



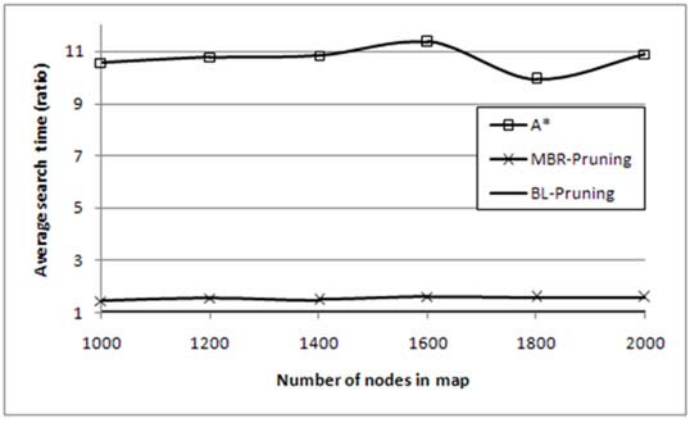
(b) Average number of visited nodes

Fig. 9. Average search time and visited nodes for 2000 node map

Next, we measured the search time and the number of visited nodes depending on the threshold ε. **Fig. 8** shows that the search time and the number of visited nodes for 1000-node maps decreased linearly as the threshold increased in range [0.7, 0.9]. The search time and the number of visited nodes are saturated when the threshold is greater than 0.9. This can be explained by the fact that the number of excluded INs doesn't change much for the threshold greater than 0.9 . This property was also observed in the other types of maps. For example, **Fig. 9** shows the performance for 2000 node maps

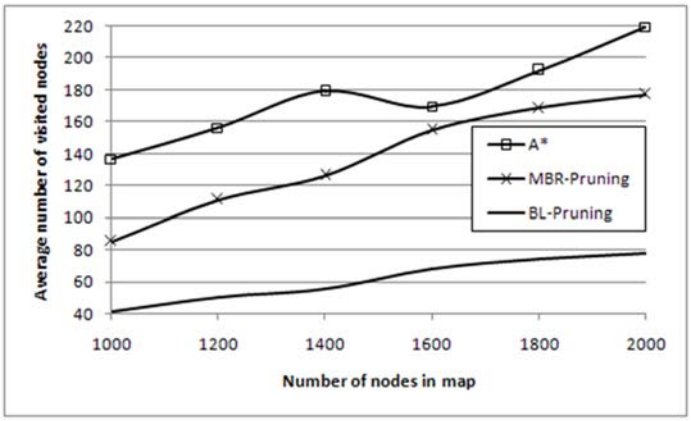


(a) Average search time in msec

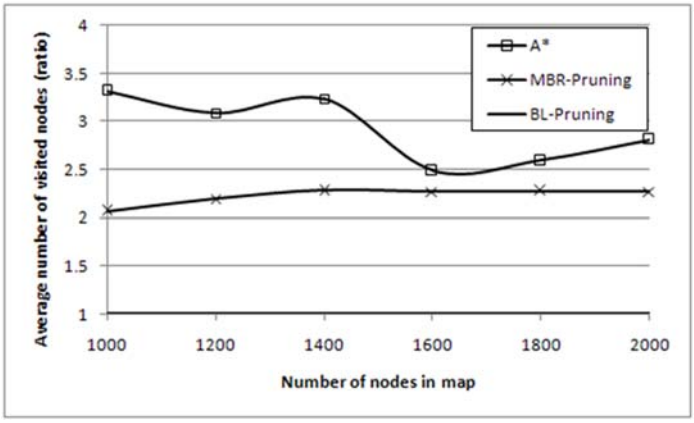


(b) Average search time relative to BL-based pruning

Fig. 10. Search times of 3 routing methods



(a) Average number of visited nodes



(b) Average number of visited nodes relative to BL-based pruning

Fig. 11. Visited nodes of 3 routing methods

Fig. 10 shows the average search time in *msec* when borderline (BL)-based pruning was tested with $k = 2$ and $\varepsilon = 0.9$. The MBR and BL-based algorithms were less influenced by the number of nodes in the map. A* and the MBR-based pruning took 11 times and 1.5 times more time than BL-based pruning on the average, respectively. Overall, the search time increased as the map got larger.

The average number of visited nodes is depicted in **Fig. 11**. Since the input maps were generated randomly, the number of visited nodes did not increase monotonically. Overall, compared with our border line-based pruning, A* and the MBR-based methods visited 3 times and 2.3 times more nodes, respectively.

5. Conclusion

In this paper, we proposed a new borderline-based pruning method to improve the pure MBR-based pruning for path routing. By constructing border lines inside MBRs based on a convex hull, its search space and time were reduced significantly. To show the effectiveness of our scheme, we first found the effective number of border lines and the line creation threshold value empirically, and then through extensive experiments, we measured the performance of our proposed scheme and compared it with other popular methods. Overall, our proposed scheme provides 1.6 times faster search time and 45% fewer nodes visited than MBR-based pruning on the average. Even though we used the Waxman random maps in the experiment, our proposed scheme can be applied easily for other types of maps just by adjusting the number of border lines k and the border line creation threshold ε.

References

[1] E.W Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269-271, Dec. 1959.

[2] M. Holzer, F. Schulz, D. Wagner and T. Willhalm, "Combining Speed-up Techniques for Shortest-Path Computations," *ACM Journal of Experimental Algorithmics*, vol. 10, no. 2.5, 2005.

[3] Hart, P. E., Nilsson, N. J. and Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] Sedgewick, R. and Vitter, J.S., "Shortest paths in Euclidean Space," *Algorithmica,* vol. 1, no. 1-4, pp. 31–48, 1986.
[5] Shekhar, S., Kohli, A. and Coyle, M., "Path computation algorithms for advanced traveler information system (atis)," in *Proc. of 9th IEEE International Conf. Data Eng*., pp. 31–39, 1993.
[6] Jacob, R., Marathe, M. and Nagel, K., "A computational study of routing algorithms for realistic transportation networks," *Journal of Experimental Algorithmics*, vol. 4, no. 6, 1999.
[7] Schulz, F.,Wagner, D. and Andweihe, K., "Dijkstra's algorithm on-line: An empirical case study from public railroad transport," *Journal of Experimental Algorithmics*, vol. 5, no. 12, 2000.
[8] Pohl, I., "Bi-directional and heuristic search in path problems," *Techical Report 104, Stanford Linear Accelerator Center, Stanford, CA*. 1969.
[9] Ahuja, R., Magnanti, T., and Orlin, J., "Network flows: Theory, Algorithms, and Applications," *Prentice–Hall*, 1993.
[10] Kaindl, H. and Kainz, G., "Bidirectional heuristic search reconsidered," *Journal of Artificial Intelligence Research*, vol. 7, pp. 283–317, Dec. 1997.
[11] Pohl, I., "Bi-directional search," *Machine Intelligence*, vol. 6, American Elsevier, New York, pp. 127–140, 1971.
[12] Schulz, F., empirical, D., and Zarolliagis, C., "Using multi-level graphs for timetable information in railway systems," *Lecture Notes in Computer Science*, vol. 2409, Springer-Verlag, New York, pp. 43–59, Jan. 2002.
[13] Jung, S. and Pramanik, S., "An efficient path computation model for hierarchically structured topographical road maps," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1029–1046, Sept. 2002.
[14] Jung, S. and Pramanik, S., "Hiti graph model of topographical road maps in navigation systems," in *Proc. 12th IEEE International Conf. Data Eng*, pp. 76–84, Mar. 1996.
[15] Wagner, D. and Willhalm, T., "Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs," *Lecture Notes in Computer Science*, vol. 2832, Springer-Verlag, New York, pp. 776–787, 2003.
[16] Graham, R.L., "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set," *Information Processing Letters*, vol. 1, no. 4, pp. 132-133, Jun. 1972.
[17] Waxman, B.M., "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, issue 9, pp. 1617 – 1622, Dec. 1988.
[18] Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Scott, M., "GraphML progress report," *Lecture Notes in Computer Science*, vol. 2265, Springer-Verlag, New York, pp. 501–512, 2001.

Jinkyu Park received his B.S. and M.S. degree in Electrical Engineering from Korea University, Seoul, Korea, in 2002 and 2008, respectively. Currently he is with Security Research Dept., LG Electronics, Gyeonggi-do, Korea. His research interests include content-based image retrieval, feature extraction, telematics and navigation algorithms.

Daejin Moon received his B.S. degree in Computer Information Engineering from Dongseo University, Korea, in 2009. Currently he is pursuing the M.S. degree in the School of Electrical Engineering in Korea University. His research interests include navigation algorithms, telematics and information retrieval.

Eenjun Hwang received his B.S. and M.S. degree in Computer Engineering from Seoul National University, Seoul, Korea, in 1988 and 1990, respectively; and his Ph.D. degree in Computer Science from the University of Maryland, College Park, in 1998. From September 1999 to August 2004, he was with the Graduate School of Information and Communication, Ajou University, Suwon, Korea. Currently he is a member of the faculty in the School of Electrical Engineering, Korea University, Seoul, Korea. His current research interests include database, multimedia systems, information retrieval, audio/visual feature extraction and representation, and Web applications.