AN EFFICIENT ALGORITHM FOR SHORTEST PATH FINDING IN REAL ROAD NETWORKS

V. Sreelesh* and R. V. Nataraj**

*Software Engineer, GeoEdge Technologies, Coimbatore.
**Associate Professor, Department of Information Technology, PSG College of Technology, Coimbatore

ABSTRACT

Finding of shortest path in graph/road networks has been an area of research for the past so many years and so many algorithms have been proposed in the literature including Dijkstra Algorithm, Prim Algorithm, A* Algorithm etc. With the recent advancements in mapping of road networks and with the invention of efficient data structures to store and retrieve map data, the demand for more efficient algorithms to find shortest path in road networks has increased. In this paper, we propose a GPS based vehicle routing algorithm for finding shortest path in real road networks. This paper proposes IA* algorithm to efficiently find route between two locations in a road network. The proposed algorithm exploits several optimization techniques including database projection, remapping of nodes and bucket sorting with indexing. For our experiments we have used real road networks, namely the road network map of Bangalore city.

Index Terms- Shortest Path finding - Road Networks - Heuristic algorithm.

I INTRODUCTION

With the advancement of technology in the last decade it has become possible to create maps of cities and roads all over the world. Even applications which have mapped the whole road networks in the world map are available, which can be used in car navigators. This has given rise to a revival of shortest path algorithms and these real road networks contain millions of nodes and edges, whereas previous experiments used small artificial networks. But with availability of these road networks, it has been revealed that the existing algorithms which are used for shortest path finding have several short comings. This has led to the demand for designing more efficient algorithms or optimizing the existing algorithms to match the new requirements.

In this paper, we consider the point-to-point instance of the shortest path problem. The best-known algorithms in Operations Research are Bellman-Ford and Dijkstra. In Artificial Intelligence, the A* algorithm, which assumes the availability of a heuristic estimate, is also widely known[2]. If we compare Dijkstra's Algorithm with A* Algorithm, Dijkstra's is like a puddle of water flooding outwards on a flat floor, whereas A* is like the same puddle expanding on a bumpy and graded floor towards a drain (the target node) at the lowest point in the floor. Instead of spreading out evenly on all sides, the water seeks the path of least resistance, trying new paths only when something gets in its way.

In this paper, we propose IA* (Improved A*) based on the A* algorithm for finding shortest path in real road networks. We have used certain techniques which have reduced the running time of the algorithm to a great extent. We have also done a comparative study of the performance of the existing algorithm to the algorithm proposed by us and the results have been explained at the end of the paper. The rest of the paper is organized as follows. In section II, we present the preliminaries about the algorithm being used. In section III, we explain the algorithm and the various optimization techniques used. In section IV, we give our experimental results and in section V, we conclude the paper with future work.

II PRELIMINARIES

The basic problem in shortest path finding is to find the best way to move from the start node p to destination node q. In the case of graph based problems there is a probability that the edge cost might be negative. But since in this paper we are dealing with real road problem all the edge costs are distances which in turn are positive. In the context of shortest path finding, a map database D is a tuple $D = (N, R)$, where N is a finite set of nodes along with their associated latitude and longitude. Each node corresponds to some landmark or locations or road junctions of specific importance in the map. $R \subseteq N \times N$ is a binary relation between nodes. Each pair $(p, q, d) \in R$ denotes the fact that start node $p \in N$ is connected to the end node $q \in N$ with d being the distance between p and q. The whole road network of a city can be represented using this database with each landmark location or road junction having its own unique node identifier in the database.

III SHORTEST PATH ALGORITHM

Our proposed method uses a projected database to optimize the processing space and assumes that entire database of road network and associated data required like latitude, longitude (and all data structures) completely fit into main memory. Our method creates an adjacency matrix version of projected database and projects only the required nodes between the source and destination. We assume safely that the maximum in-degree/out-degree of a given node is not more than fifteen, whereas in most of the practical scenarios, it is much lower than fifteen. This assumption is based on the following fact; if a junction in a road is considered to be a node, then the number of roads connecting to that junction will be four if it is an intersection of two roads. In general, most junctions won't have more than four of five roads connecting to it, so our assumption that the maximum in-degree/out-degree is less than fifteen hold.

i). A* Algorithm

In computer science, A* (pronounced "A star") is a best-first, graph search algorithm that finds the least-cost path from a given initial node to one goal node (out of one or more possible goals).

It uses a distance-plus-cost heuristic function (usually denoted as f(x)) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions: the path-cost function (usually denoted as g(x), which may or may not be a heuristic) and an admissible "heuristic estimate" of the distance to the goal (usually denoted h(x)). The path-cost function, g(x), is the cost from the starting node to the current node. Since the h(x) part of the f(x) function must be an admissible heuristic, it must not overestimate the distance to the goal. Thus, for an application like routing, h(x) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points (or nodes for that matter).

The algorithm was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael. In their paper, it was called algorithm A. Since using this algorithm yields optimal behavior for a given heuristic, it has been called A*. Like all informed search algorithms, it first searches the routes that appear to be the most likely ones to lead toward the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account (the g(x) part of the heuristic is the cost from the start, and not simply the local cost from the previously expanded node).

The algorithm traverses various paths from start to goal. For each node x traversed, it maintains 3 values:

- g(x): the actual shortest distance traveled from initial node to current node
- h(x): the estimated (or "heuristic") distance from current node to goal
- f(x): the sum of g(x) and h(x)

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the open set (not to be confused with open sets in topology). The lower the value of f(x) for a given node x, the higher its priority. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and h values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any other node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic. If the actual shortest path is desired, the algorithm may also update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal node. [6]

ii). IA* Algorithm

The original A* has been modified to improve the running time. We propose mainly three techniques to reduce the running time of the A* algorithm.

i. Database projection.

ii. Remapping of nodes.

iii. Bucket sorting with indexing.

Database Projection

We use a database projection technique to reduce the number of nodes which are processed by the algorithm. Once we obtain the source-node and destination-node, we fetch a set of nodes by forming an ellipse around the source-node and destination-node. The two nodes will form the focal points of the ellipse which is drawn. For doing this we need to have latitude and longitude positions of each of the node in the road network. This is where GPS comes into play. When we map the road network data, we also map the latitude and longitude of the road network. We call the ellipse as MBE (Minimum bounding ellipse). An example figure of ellipse which is used for filtering nodes is shown below.

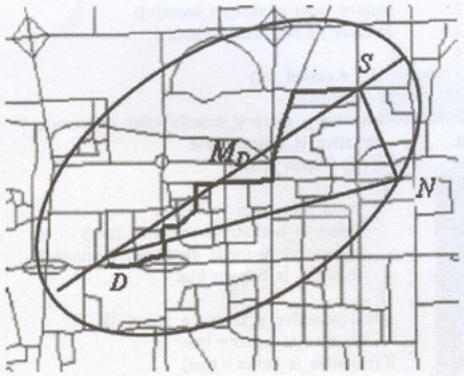

Figure 1: MBE for node extraction

In Figure 1, S is the source, D is the destination and M_D, the major diagonal. The node N is a node in the vicinity of the ellipse. We are using the following equation to find all the nodes which comes inside the ellipse. Assuming that (X_d, Y_d) be the latitude and longitude of the destination point and (X_s, Y_s) be the latitude and longitude of the source point. For a given node N, let (X_n, Y_n) be the latitude and longitude positions in the node. A node N is said to be inside the ellipse with eccentricity η if the following equation is satisfied. We define DBP(Point p_1,Point p_2) as the distance between points function which will give the straight line distance between two points for points p_1 and p_2.

DBP(S, N) + DBP(D, N) <= η*DBP(S, D)

DBP(Point p_1, Point p_2)

{

return Sqrt(Square(p_1.latitude-p_2.latitude) +Square (p_2.longitude-p_1.longitude))

}

The lower the η value the more will be the number of nodes that will be eliminated in this phase. When the value of η is selected it should be selected in such a way that there should be a path between the source and destination through the nodes which are not eliminated. Selecting the η depends on the type of graph generally. If the graph is very dense i.e. so many nodes are available in a very small space (Eg:- city maps) a lower value of η would suffice. But if the map is sparse we will have to select higher η values. From our studies we have found out that the values of η varies from 2 to 5.

This database projection technique exploits the fact that the shortest path between any two points will be almost a straight line path and therefore in such cases there is no need to consider nodes which does not come anywhere near the straight line joining source and destination nodes.

Remapping of Nodes

To improve the processing speed and reduce the memory consumption of the dataset, we use the following techniques. We remap all the nodes to continuous integer space. For example, if the nodes obtained after the database projection phase are {100,200,234,388}, then it is remapped as {1, 2, 3, 4} i.e. 1 instead of 100, 2 instead of 200 and so on. This will reduce the adjacency matrix size to a great extent. If we use the items directly then we will have to use an adjacency matrix of size 388 whereas after remapping the matrix size reduces to 4. This in turn will reduce the memory access time. For real time road networks where number of nodes are in lakhs or crores, the time saved is really significant.

Also, instead of using the normal adjacency matrix for representing whether two nodes are connected, here we use a modified approach. In normal adjacency matrix approach, if node k is related to node x ,y ,z then *adjacency_matrix*[k][x] = 1, *adjacency_matrix* [k][y] =1, *adjacency_matrix*[k][z]=1. In our approach if nodes x,y,z are related with node k then *adjacency_matrix*[k][*1*] will be set as x, *adjacency_matrix* [k][*2*] will be set as y and *adjacency_matrix*[k][*3*] will be set as z. This would again save lot of memory consumption and memory access time for huge databases.

Bucket Sorting

A* algorithm spends most of its processing time to sort the nodes from open set and obtain the node having the minimum cost. Since this is done each time a new element is inserted, which heavily influences the running time of the algorithm. To improve the running time of the sorting procedure we use bucket sorting. It has been found that this reduces the running time to a great extent. The method used for bucket sorting is explained below.

In our implementation, we are using single level buckets. When we start the execution of the algorithm, we initialize the bucket size. It is done based on the perpendicular distance between the source and the destination. If that distance increases, then the bucket size also increases proportionally. If the distance between source and destination is D in terms of latitude & longitude values, then the bucket size, BS, shall be D/0.01. We are using 0.01 because 0.01 in latitude & longitude corresponds to 1 km. This can be modified by considering the density/sparsity of the road network. Each

element in the bucket shall be a sorted singly linked list. A linked list only requires O(1) time to complete an operation in each distance update in the bucket data structure. These operations include:

1) Checking if a bucket is empty

2) Deleting an element from a bucket.

3) Retrieving the top element from the bucket.

The insertion operation will take O(n) on an average where n=N/BS where N is the total number of nodes and BS denotes the number of buckets available. All nodes with distances ranging between i*(D/BS) and (i+1)*(D/BS) (where D is the distance between source and destination and BS the bucket size) shall be inserted in the i^{th} bucket.

To improve the performance of the bucket sorting technique, we are using a variable named 'index' which will contain the index of the smallest indexed bucket which is not null. Whenever an element is inserted or removed from the buckets, the 'index' variable is updated. This helps us to avoid scanning of the bucket each time when we need to find the smallest element. This optimization is not shown in the algorithm given below as it is simple and understandable.

ALGORITHM 2: IA* Algorithm

Input: database D, source-node, destination-node Output: shortest path from source-node to destination-node

// The heuristic we are considering here is straight line distance heuristic.

```
1.  IA* (source, destination)
2.  {
3.    closed_set = F
4.    Initialize open_set
5.    bucket_size=heuristic_distance(source, destination)
6.    Initalize bucket[size]
7.    g_score[source] = 0
8.     h_score[source]=heuristic_distance(source,destination)
9.    f_score[source]=g_score[source]+h_score[source]
10.    insert_to_bucket(f_score[source], source);
11.   while (open_set != F)
12.   {
13.     for(i=1 to bucket_ size)
14.     {
15.       If(bucket[i]!=null)
16.       x=first_element(bucket[i]);
17.     }
18.     if (x = goal)
19.         return generate_path(came_from,destination)
20.         open_set =  open_set\x
21.         closed_set = closed_set U x
22.         remove_from_bucket(x,f_score[x])
23.         foreach( y € neighbor_nodes(x))
24.         {
25.           If (y € closed_set)
26.              continue
27.           tentative_g_score=g_score[x]+dist_between(x,y)
28.           tentative_is_better = false
29.           if (!(y € open_set))
30.           {
31.               open_set = open_set U y
32.               insert_to_bucket(f_score[start], start)
33.               h_score[y]=heuristic_distance(y, destination)
34.               tentative_is_better = true
35.           }
36.           else if (tentative_g_score < g_score[y])
37.               tentative_is_better = true
38.           if (tentative_is_better = true)
39.           {
40.             remove_from_bucket(y,f_score[y]);
41.           came_from[y] = x
42.             g_score[y] = tentative_g_score
43.             f_score[y] = g_score[y] + h_score[y]
44.             insert_to_bucket(f_score[y],y);
45.         }
46.   }
47.   return failure
48. }
49. generate_path(came_from,curr_node)
50. {
51.   if came_from[current_node] != -1
52.   {
53.     p=generate_path(came_from,came_from[curr_node]);
54.     return (p + current_node);
55.   }
56.   else
57.     return null;
58. }

59. **insert_to_bucket(f_score, node)
60. {
61.   for(i = 1 to size)
62.   {
63.     if(f_score>i*D/BS && f_score<(i+1)*D/BS)
64.     {
65.         insert_to(bucket[i]);
66.     }
67.   }
68. }

69. **remove_from_bucket (node, f_score)
70. {
71.   for(i = 1 to size)
72.   {
73.     if(f_score>i*D/BS && f_score<(i+1)*D/BS)
74.     {
75.       remove_from(bucket[i]);
76.     }
77.   }
78. }
```

** Each element of the bucket will contain a linked list which is always sorted. The element is always inserted to the linked list at the exact position. The smallest element will always be on the top.

Algorithm Explanation

In the above algorithm we do not explain the filtering of nodes using the MBE ellipse technique, since it has been explained clearly in the previous section. We assume that the nodes considered here contain only the set of nodes contained in the MBE ellipse.

The algorithm takes as input the source node and the destination node. The *closed_set* is initialized to NULL (line no 3) and the *open_set* is initialized to set of nodes which fell inside the MBE (line no 4). The *bucket_size* is initialized to the heuristic distance between source and destination and all the buckets are initialized (line no 5 & 6). The *f_score* for source node is calculated (line no 9). The *g_score* for source is zero since distance from source node to source node is zero (line no 7). The *h_score* for source is the heuristic distance between the source node and destination node (line no 8). The source node is now inserted into the bucket (line no 10).

In the next loop, check whether the *open_set* is NULL to make sure if there is any more nodes left to which the distance need to be found (line no 11). If this loop exits before we find the path to the destination node, then it means that there is no path through the existing set of MBE filtered nodes, and the size of MBE needs to be increased so as to take into account other nodes and other possible paths.We scan the buckets to find the smallest valued node in the set of nodes which has been inserted. Since the buckets are in such a way that the lowest element will always be the first element in the first non-empty bucket, we only need to fetch that (line no 15 & 16). If the returned node itself is the goal then we have found the path already and need to generate the path from source to destination. This is done by calling the *generate_path* function (line no 19). If it is not the destination node then the returned node x is removed from *open_set* and added to the *closed_set* (line no 20 - 21) and also removed from the bucket. This is done because we have already found the best available path for x from source. If y is an element of *closed_set* then the shortest path to it has been already been found so no need to update the *f_score*(line no 26). If y is not an element of *open_set* then *f_score* of y need to be added to the bucket and y is added to *open_set* (line no 31). If y is an element of *open_set* then *f_score* of y need to be updated (line no 33-35). So we calculate the new *f_score* and update it if it is lower than the previously calculated *f_score* in the bucket (line no 43-44).

The *generate_path* function generates the path through which the vehicle should travel to reach the destination from source. The path is actually stored in an array known as came_from. In the came_from array if came_from[p]=q then it means that q is the ancestor of p. The root node r will have came_from[r]=-1. This method of representation is similar to array representation of sets. So starting from the destination node d we will check came_from[d]. If came_from[d]=e, then we set e as the ancestor of d (line no 44). We now check came_from[e] and go on finding ancestors recursively (line no 43) until we reach the source node s where the came_from(s) will be equal to -1 (line no 51).

The insert_to_bucket function takes the *f_score* and the node id as arguments. It inserts the node_id into the required bucket by checking the value of *f_score* against the range of values of each bucket (line no 63-65). The remove_from_bucket function takes the f_score and the node id as arguments. It removes the node_id from the bucket in which it was previously inserted by checking the value of f_score against the range of values of each bucket (line no 73-75). The space complexity of the algorithm is linear in terms of the number of nodes i.e. O(k) where k is the number of nodes.

IV IMPLEMENTATION AND RESULT ANALYSIS

We have implemented our method using C# language and the code was compiled using 32-bit Microsoft Studio 2005 C# compiler. We have implemented Dijkstra's algorithm, A* algorithm and our IA* algorithm. We have used a real road network for comparing the algorithms and therefore feel that the comparison results are fair and can be trusted over artificial data sets. We have used the Banglore road network database for our study and implementation. Our results obtained are shown below. We have considered a network containing almost one lakh nodes and are trying to find the distance between points of varying straight line distances.

Table 1: Running Time Analysis

Straight Distance in Kilometers	Distance Calculated by A*Star	Time(ms) A*Star	Time(ms) IA*	Time(ms) Dijkstra (With buckets)
5.811	8.505	31	<1	15
13.962	15.699	359	16	31
16.985	18.537	1032	47	63
23.383	24.884	3671	125	250
26.187	29.039	5734	210	422
32.279	37.214	6984	250	547

We have done a large number of experiments and shall present only representative results here. Table 1 shows the runtime performance of various algorithms along with proposed Modified A* algorithm. From the results shown, it is clear that the IA* algorithm outperforms the existing A* and Dijkstra algorithm which are its major competitors. Figure 2 shows the comparison graph of our modified A* algorithm against IA* algorithm whose sorting has been done using buckets. Figure 3 shows the performance of A* algorithm against Dijkstra Algorithm. In both the cases IA* algorithm performs better than its counterparts.

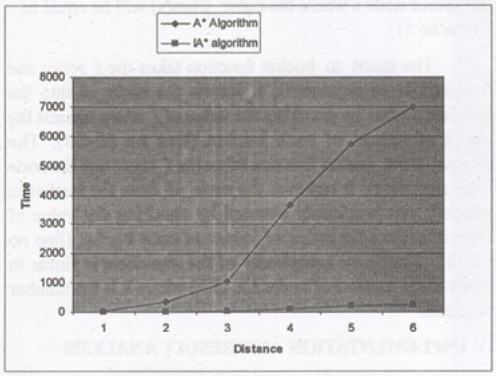

Figure 2 : Runtime of the Dijkstra algorithm vs Modified A* algorithm

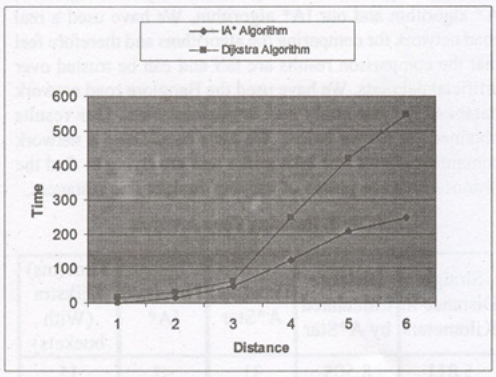

Figure 3: Runtime of the A* algorithm vs Modified A* algorithm

V CONCLUSION & FUTURE WORK

We have proposed IA* algorithm which uses projected databases for reducing the processing space. We also make use of bucket sorting and indexing technique to improve the running time of our algorithm. Our technique is computationally efficient in terms of space, time and processor usage. We are working on to further optimize the running time and to make the memory consumption more efficient. In future, we plan to modify the algorithm to support dynamic route prediction taking into account issues like traffic blocks, road blocks etc.

ACKNOWLEDGEMENTS

We would like to thank GeoEdge Technologies Pvt. Ltd. for providing us with the valuable infrastructure and the maps databases which helped us test our algorithm in a real time environment. We would like to thank Mr. Ganesh Kumar Ponnuswamy, System Analyst, GeoEdge Technologies Pvt. Ltd. for giving us invaluable input in the form of suggestions and the guidance he offered us during the development of this algorithm and its implementation.

REFERENCES

[1] F. Benjamin Zhan," Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures", Journal of Geographic Information and Decision Analysis, vol.1, no.1, pp. 69-82,2001.

[2] Wim Pijls,Henk Posty," A new bidirectional algorithm for shortest paths", Econometric Institute Report EI ,2008

[3] Feng Lu and Poh-Chin Lai," A Shortest Path Searching Method with Area Limitation Heuristics", Lecture notes in Computer Science, 2006, NUMB 3991, pages 884-887, ISSN 0302-9743.

[4] Thomas H.Cormen ,Charles E.Leiserson,Ronald L.Rivest, and Clifford Stein."Introduction to Algorithms",Second Edition.MIT Press and McGraw-Hill,2001.ISBN 0-262-03293-7.Section 24.3:Dijkstra's algorithm, pp.595-601.

[5] F. Benjamin Zhan and Charles E. Noon ," Shortest Path Algorithms: An Evaluation using Real Road Networks",Journal of Transportation Science Vol. 32,

No. 1, February 1998 .

[6] Heung Suk Hwang, "GIS-Based VRP Solver for Supply Chain Network - 2-D and 3-D GIS Vehicle Routing Model ", LSCM2006 (International Conference on Logistics and Supply Chain Management), Hong Kong, 5-7, Jan, 2006.

[7] Zhang Ke ,LIiuXiao ming ,WANG Xiao jing , "Research on Route Planning System for Vehicle Automatic Navigation",Systems Engineering,2001.

[8] M. YANG, "Application design and implementation of GPS-GPRS location system vehicle terminals," Telecommunication Engineering, March, 2004, pp. 103-106.

[9] E. W. Dijkstra." A note on two problems in connection with graphs. Numerische Mathematik" pp. 269-271, 1959.

[10] Peter E.Hart ,Nils J.Nilsson,"A formal Basis for the heuristic Determination of Minimum Cost Paths ",IEEE Transactions on Systems Science and Cybernatics, July 1968.

[11] Decher,Rina,Judea Pearl," Generalized best-first search strategies and the optimality of A*",Journal of the ACM, Volume 32 Issue 3, July 1985.

[12] Lu Feng, Zhou Chenghu, Wan Qing, 2000, An improved Dijkstra's shortest path algorithm based on quad heap priority queue and MBR searching method, Proceedings of the 9th Spatial Data Handling Symposium, 2000,6b:3-13.

[13] Ahuja R.K.,Mehlhorn K.,Orlin J.B. and Tarjan R.E.," Faster algorithms for the shortest path problem", Journal of the ACM ,Volume 37 Issue 2, April 1990.

[14] Amit's A *pages,URL http://theory.stanford.edu/~amitp/Gameprogramming

[15] A *search algorithm ,URL http://en.wikipedia.org/wiki/A*_search_algorithm

[16] Lester P,A*pathFinding for Beginners,URL:http://www.policyalmanac.org/games/aStarTutorial.htm