ISSN:0975-9646

Analysis Of Optimal Route Algorithms Under Constraint Conditions

M.V.Mawale[1], Dr. Y.B.Gandole[2]

[1] Department of Computer Science, [2] Department of Electronics
Adarsha Science J.B.Arts and Birla Commerce, Mahavidyalaya
Dhamangaon Rly-444709 (India)

***Abstract*— The shortest path problem is the problem of finding a path between two vertices (or nodes) such that the sum of the weights of its constituent edges is minimized. Clearly, the choice of shortest-path algorithm for a particular problem will involve complex tradeoffs between flexibility, scalability, performance, and implementation complexity. The comparison made by us developed provides a basis for evaluating these tradeoffs. It was found that although factoring in this optimization to larger degrees did lead to significant imperfections, a balanced level was located where not only were perfect or near-perfect paths were found, but they were also found in the shortest time. the paper analyses the existing optimal route algorithms on a specific problem under constraint conditions with optimal factors based on the Dijkstra algorithm, Bellman Ford Algorithms and A* Algorithms. The paper also gives a method of the algorithm design in detail, and analyses of its time complexity and space complexity.**

Keywords— Optimal Route, Dijkstra algorithm, Bellman Ford Algorithms, A* Algorithms.

I. INTRODUCTION

The optimal route has many meanings, not only the shortest distance in the general geography, but also the shortest time, the least expense, the route utilization and so on. Regardless of using which judgment standard, its core problem is the shortest-route algorithm. The optimal route category may include: the longest route, the most reliable route, the maximum capacity route, accessibility evaluation and various route distributions [1-4]. In graph theory, the shortest path problem (Optimal Route) is the problem of finding a path between two vertices (or nodes) such that the sum of the weights of its constituent edges is minimized. An example is finding the quickest way to get from one location to another on a road map; in this case, the vertices represent locations and the edges represent segments of road and are weighted by the time needed to travel that segment. The problem is also sometimes called the single-pair shortest path problem, to distinguish it from the following generalizations:

- *single-source shortest path problem*, in which to find shortest paths from a source vertex *v* to all other vertices in the graph.
- *single-destination shortest path problem*, in which find shortest paths from all vertices in the graph to a single destination vertex *v*. This can be reduced to the single-source shortest path problem by reversing the edges in the graph.
- *all-pairs shortest path problem*, in which to find shortest paths between every pair of vertices *v*, *v'* in the graph.

These generalizations have significantly more efficient algorithms than the simplistic approach of running a single-pair shortest path algorithm on all relevant pairs of vertices.
Clearly, the choice of shortest-path algorithm for a particular problem will involve complex tradeoffs between flexibility, scalability, performance, and implementation complexity. The comparison made by us developed provides a basis for evaluating these tradeoffs. It was found that although factoring in this optimization to larger degrees did lead to significant imperfections, a balanced level was located where not only were perfect or near-perfect paths were found, but they were also found in the shortest time [6-16].This paper is focus on the comparison of three different shortest path algorithms. The problem is an important problem in graph theory and has applications in communications, transportation, and electronics problems. It is interesting because analysis shows that three algorithms can be optimal in different circumstances, depending on tradeoffs between computation and communication costs.

II OPTIMAL ROUTE ALGORITHMS

1. Dijkstra's Algorithm

Dijkstra's algorithm solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.
This problem is related to the spanning tree one. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. For a graph [5],

$G = (V,E)$ (1)

Where V is a set of vertices and E is a set of edges.Time Complexity $= V \log (E)$ (2)

Space Complexity $= V*(V+E) \log (V)$ (3)

Procedure:

- Step 1 [Initialization]
  - T = {s} Set of nodes so far incorporated
  - L(n) = w(s, n) for n ≠ s

- initial path costs to neighboring nodes are simply link costs
- Step 2 [Get Next Node]
  - find neighboring node not in T with least-cost path from s
  - incorporate node into T
  - also incorporate the edge that is incident on that node and a node in T that contributes to the path
- Step 3 [Update Least-Cost Paths]
  - L(n) = min[L(n), L(x) + w(x, n)] for all n Ï T
  - f latter term is minimum, path from s to n is path from s to x concatenated with edge from x to n

It perform following function

- finds shortest paths from given source node s to all other nodes
- by developing paths in order of increasing path length
- algorithm runs in stages each time adding node with next shortest path
- algorithm terminates when all nodes processed by algorithm (in set T)

Bellman Ford Algorithm

The Bellman–Ford algorithm, a label correcting algorithm, computes single-source shortest paths in a weighted digraph (where some of the edge weights may be negative). Dijkstra's algorithm solves the same problem with a lower running time, but requires edge weights to be non-negative. Thus, Bellman–Ford is usually used only when there are negative edge weights. If the graph does contain a cycle of negative weights, Bellman-Ford can only detect this; Bellman-Ford cannot find the shortest path that does not repeat any vertex in such a graph. This problem is at least as hard as the NP-complete longest path problem.

Bellman Ford Algorithm Procedure

Bellman–Ford runs in O(|V|·|E|) time, where |V| and |E| are the number of vertices and edges respectively. This implementation takes in a graph, represented as lists of vertices and edges, and modifies the vertices so that their distance and predecessor attributes store the shortest paths.

Procedure Bellman Ford (list vertices, list edges, vertex source)

```
Step 1: Initialize graph
            for each vertex v in vertices:
              if v is source then v.distance := 0
              else v.distance := infinity
              v.predecessor := null
Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
  for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if v.distance > u.distance + uv.weight:
      v.distance := u.distance + uv.weight
      v.predecessor := u
Step 3: check for negative-weight cycles
  for each edge uv in edges:
    u := uv.source
    v := uv.destination
        if v.distance > u.distance + uv.weight:
        error "Graph contains a negative-weight cycle"
```

Time Complexity = V*E * log(V*L)

Space Complexity= V

A* Algorithm

1. Create a search graph G, consisting solely of the start node, no. Put no on a list called OPEN.
2. Create a list called CLOSED that is initially empty.
3. If OPEN is empty, exit with failure.
4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Called this node n.
5. If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to no in G. (The pointers define a search tree and are established in Step 7.)
6. Expand node n, generating the set M, of its successors that are not already ancestors of n in G. Install these members of M as successors of n in G.
7. Establish a pointer to n from each of those members of M that were not already in G (i.e., not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member, m, of M that was already on OPEN or CLOSED, redirect its pointer to n if the best path to m found so far is through n. For each member of M already on CLOSED, redirect the pointers of each of its descendants in G so that they point backward along the best paths found so far to these descendants.
8. Reorder the list OPEN in order of increasing f values. (Ties among minimal f values are resolved in favor of the deepest node in the search tree.)
9. Time Complexity=Log (H*V)
10. Space Complexity= V+E

III IMPLEMENTATION AND TESTING

This section discussed the use and testing of application. Testing process conducted by performing the test route search using existing network path finding algorithms with multiple destinations at once and by providing constraints

Implementation of Dijkstra's Algorithm

The Routing Implementation of Dijkstra's Algorithm for the network is shown in Fig. 1.

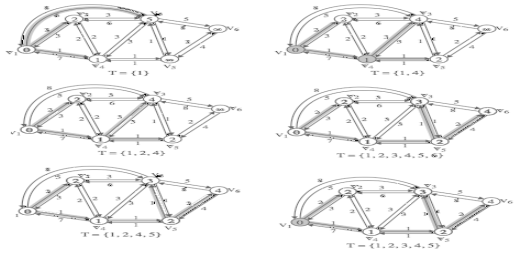

Fig. 1 Routing Implementation of Dijkstra's Algorithm

Implementation of bellman Ford Algorithm

The Routing Implementation of bellman Ford Algorithm for the network is shown in Fig. 2.

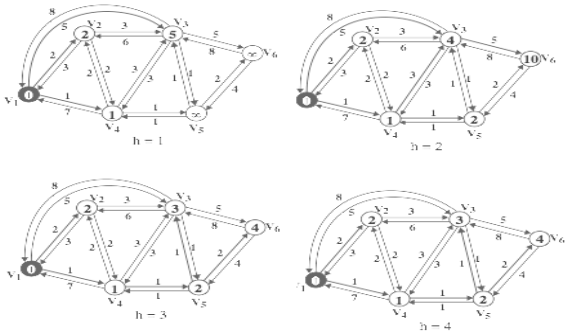

Fig. 2 Routing Implementation of bellman Ford Algorithm

IV EXPERIMENTAL RESULT ANALYSIS

The experimental result analysis for Dijkstra's Algorithm and Bellman ford Algorithm are shown in Figure 1 and 2 respectively.

Table 1 Experimental Result Analysis of Dijkstra's Algorithm

Iteration	T	L(2)	Path	L(3)	Path	L(4)	Path	L(5)	Path	L(6)	Path
1	{1}	2	1–2	5	1-3	1	1–4	∞	-	∞	-
2	{1,4}	2	1–2	4	1-4-3	1	1–4	2	1-4–5	∞	-
3	{1, 2, 4}	2	1–2	4	1-4-3	1	1–4	2	1-4–5	∞	-
4	{1, 2, 4, 5}	2	1–2	3	1-4-5–3	1	1–4	2	1-4–5	4	1-4-5–6
5	{1, 2, 3, 4, 5}	2	1–2	3	1-4-5–3	1	1–4	2	1-4–5	4	1-4-5–6
6	{1, 2, 3, 4, 5, 6}	2	1-2	3	1-4-5-3	1	1-4	2	1-4–5	4	1-4-5-6

Table 2 Experimental Result analysis of Bellman ford Algorithm

h	Lh(2)	Path	Lh(3)	Path	Lh(4)	Path	Lh(5)	Path	Lh(6)	Path
0	∞	-	∞	-	∞	-	∞	-	∞	-
1	2	1-2	5	1-3	1	1-4	∞	-	∞	-
2	2	1-2	4	1-4-3	1	1-4	2	1-4-5	10	1-3-6
3	2	1-2	3	1-4-5-3	1	1-4	2	1-4-5	4	1-4-5-6
4	2	1-2	3	1-4-5-3	1	1-4	2	1-4-5	4	1-4-5-6

Efficiency

1. Dijkstra's algorithm

The efficiency of Dijkstra's varies depending on |V|=n Delete Min sand |E| updates for priority queues that were used. If a Fibonacci heap was used then the complexity is O(| E | + | V | log | V |) ,which is the best bound. The Delete Mins operation takes O(log|V|).If a standard binary heap is used then the complexity is O(| E |log |E|),| E | log |E|. If the set used is a priority queue then the complexity is O(|E|+|V|2) where O(|V|2) is from |V| scans of the unordered set New Frontier to find the vertex with the least Distance value.

2. Bellman-Ford algorithm

Routing algorithms based on the distributed Bellman-Ford algorithm (DBF) suffer from exponential message complexity in some scenarios. The message complexity of the first algorithm, called the multiplicative approximation algorithm, is O(nm log (nΔ)), where Δ is the maximum length over all edges of an n-nodes m-edges network. The message complexity of the second algorithm, called the additive approximation algorithm, is O(Δ/δ nm), where is δ is the minimum length over all edges in the network.

3. A* Search Algorithm

A* search has been implemented in the UMOP multi-agent planning framework to study its performance characteristics. The time limit was 600 seconds and the memory limit was 450 MB. For UMOP and DOP the number allocated BDD nodes of the BDD-package and the number of partitions in the disjunctive partitioning were hand-tuned for best performance.

Big-O Notation for complexity and is expressed in terms of the order of magnitude of frequency count gives in the table 3. The graphical comparison for efficiency in terms of speed is shown in Fig. 3

Table 3 Efficiency in terms of speed

Run Time(N)	Dijkstra	Bellman Ford	A*	Speed(F)
400	5.490	4.770	3.246	0.799
500	6.862	5.962	4.058	0.999
600	8.235	7.155	4.869	1.198
700	9.607	8.347	5.681	1.398
800	10.980	9.540	6.493	1.598
900	12.353	10.732	7.304	1.798
1000	13.725	11.925	8.116	1.998

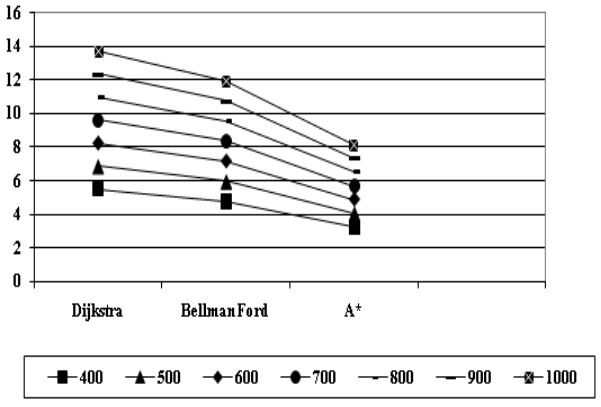

Fig.3 : Comparison of efficiency in terms of speed

Time Complexity

1. Dijkstra algorithm

A implementation of the priority queue gives a run time complexity O(V²), where V is the number of vertices. Implementing the priority queue with a Fibonacci heap makes the time complexity O(E + V log V), where E is the number of edges.

2. Bellman ford algorithm

Its worst case time complexity is the same as the original Dijkstra Algorithm i.e. O(| V | 2) , where | V | is the number

of nodes in the network. Unlike QoSRBF it is not ensured that QoSRDKS can always find a rout satisfying multiple QoS requirements, even when there exists one. The simulation results show, however that QoSRDKS is also very efficient.

3. A* search algorithm

A* algorithm generally is polynomial however it can become exponential when it's exponential have it's complexity as O(x^n) where n is the maximum length of any path in the revealed area and x is a is the average number of edges per node.

The table 4 gives that the Time Complexity of the shortest path algorithms depends on the number of vertices, number of edges , edge length and the heuristic in case of the A* algorithm. It is observed that the time complexity of the Dijkstra's Algorithm depends on the number of vertices and is inversely proportional to the number of vertices. The comparison of Time complexity is shown in Fig. 4. For the graph of six vertices and nine edges The Dijsktra's algorithm gives time complexity of 7.0036 which is far below than that of Bellman Ford because the number of vertices is less and the Bellman Ford algorithm depends on the edges length as well

Table4 : Analysis of time Complexity

Algorithm	Formula	Vertex (V)	Edges (E)	Length (L)	Heuristic(H)	Time Complexity
Dijkstra's	V log (E)	6	9	10	-	7.0036
Bellman Ford	V*E* log(V*L)	6	9	10	-	96.00
A*	Log (H*V)	6	9	10	100	2.77

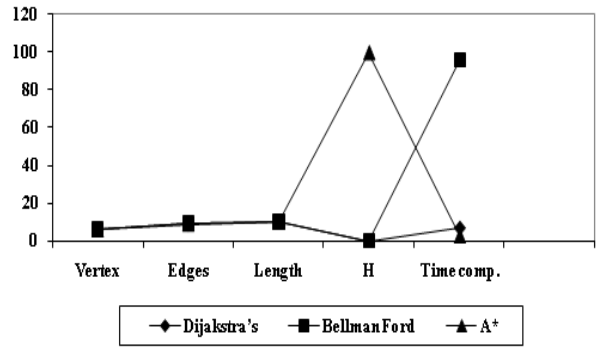

Fig.4 Comparision of Time Complexity

Computational Cost

1. Dijkstra's algorithm

Computational cost of Dijkstra's algorithm using an array or list to store the labeled vertices is $O(V^2 + E) = O(V^2)$

2. Bellman Ford algorithm

In the worst case computational cost of Bellman Ford algorithm uses O(V3) time in order to find single-source shortest paths. This is not very efficient. By a slight modification it can find all-pairs shortest paths in the same time.

3. A* search algorithm

The complexity for A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the heuristic function h meets the following condition:| h(x) − h * (x) | = O(logh * (x))

where h * is the optimal heuristic,

Table 5 shows the numerical analysis and Fig.5 shows the graphical analysis of the computational cost performed for the graph with six vertices and nine edges.

Table 5 Analysis of Computational Cost

Method	Cost Complexity		Communication Number		Communication Volume	
Dijkstra's	FN3/P	28.56	0	0	0	0
Bellman Ford	N3/P	24	N log(P)	5.72	N2 log(P)	34.352
A*	N3/P	24	N log (P)	5.72	N2 log(P)/sqrt(P)	11.45

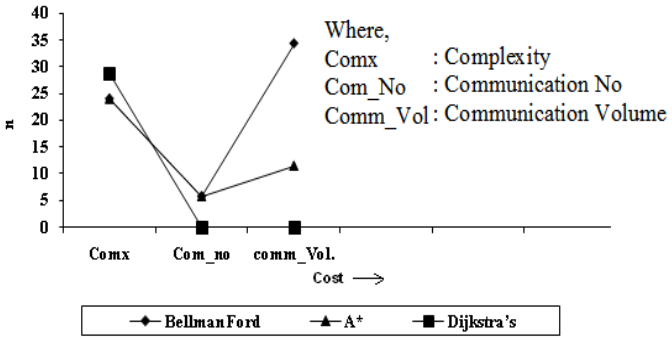

Fig. 5 Comparison of Cost Complexity

Space Complexity

The space complexity of Dijkstra's algorithm is more for the lower values of the vertex as it is inversely proportional to the vertex. The Bellman Ford space complexity is lower for the lower number of vertices and is directly proportional to the cardinality of the vertices.

Table 6 Analysis of Space Complexity

Algorithm	Formula	Vertex(V)	Edges(E)	Space Complexity
Dijkstra's	V*(V+E) log (V)	6	9	70.033
Bellman Ford	V	6	9	6
A*	V+E	6	9	15

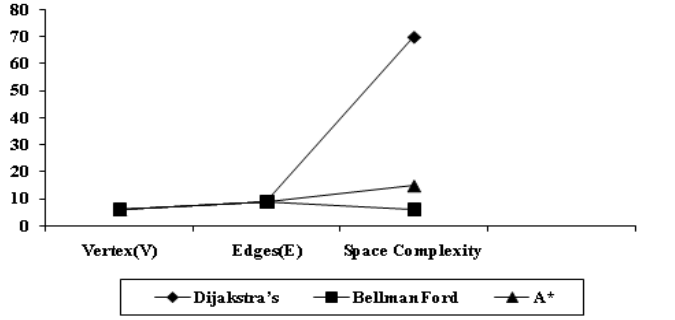

Fig. 6 Comparison of space complexity

Fig. 6 indicates the A* being a search algorithm basically gives near to constant space complexity and has negligible effect for the number of vertices where as Dijkstra's algorithm and Bellman Ford as more variations for small changes of the number of vertices .

Performance and Efficiency

Performance and Efficiency is depending on the previously compared parameters that is Computation Cost, Speed and Space and Time complexity. It is observed that lower number

of nodes in Bellman Ford algorithm is Better and for the Higher number of nodes the Dijkstra's algorithm is more efficient. The A* algorithm is best suited for the searching for the nodes amongst the High number of nodes.

Debugging Activities

1. Dijkstra's algorithm

Assertions are used for the implementation of the algorithm and it vary from case to case. In this case it is capable to handle the graphs with more than 10000 vertices. Hence included an assertion that will fail if the input violates this assumption.

2. Bellman ford algorithm

Debugging equation for Bellman-Ford when Label routers i=A, B, C, … and D(i,j) = distance for best route from i to remote j and d(i,j) = distance from router i to neighbor j then it set to infinity if i=j or i and j not immediate neighbors.

3. A* search algorithm

There are number of simple optimizations or implementation details that can significantly affect the performance of an A* implementation. The first detail is that the way the priority queue handles ties and has a significant effect on performance on applications. If ties are broken so the queue behaves in a LIFO manner, A* will behave like Depth-first search among equal cost paths.

Better Algorithms under specific application

Dijkstra's algorithm can be implemented in O(|V|^2) or O(|E|*log(|V|)) time depending on how the next node to consider is chosen. It is observed that as |E| increases, it approaches |V|^2, so choosing the right programming representation requires knowing something about the potential inputs to the application area.

On the other hand the Bellman Ford algorithm that handles negative weighted edges runs in O(|E| * |V|) time, which is significantly slower -- for so-called dense graphs with many edges, it can approach a cubic time of O(|V|*|E|) as |E| is O(|V|^2).

A* algorithm is the best algorithm for searching the path at least cost. It uses a distance-plus-cost heuristic function (usually denoted f(x)) to determine the order in which the search visits nodes in the tree.

V RESULT AND DISCUSSION

While analysing the result for the application specific comparison of three shortest path problem solving algorithms namely Dijkstra's, Bellman Ford and A* Algorithms. It is noted that Dijkstra's and Bellman Ford algorithms solve the single source shortest path problem. The primary difference in the function of the two algorithms is that Dijkstra's algorithm cannot handle negative edge weights. Bellman-Ford's algorithm can handle some edges with negative weight. It is noted that if there is a negative cycle there is no shortest path. Following points gives the result analysis of the different algorithm for some application specific.

Computational Cost Summery

- The maximum time over all process depends on number of vertex and edges.
- The cost complexity for Dijkstra's algorithm is highest than Bellman Ford algorithm.
- The cost complexity of Bellman Ford algorithm is almost same

Efficiency in terms of speed

- The Bellman Ford Algorithm was faster than the Dijkstra's algorithm.
- Speed advantage for Bellman Ford algorithm was less than reported value, F = 1.19
- Speed for Dijkstra's is highest for same number of process run.
- With increasing number of process, Dijkstra's algorithm eventually becomes faster because no communication occurs

Time complexity

- Time Complexity of the shortest path algorithms depends on the number of vertices, number of edges , edge length and the heuristic in case of the A* algorithm.
- It observed that the time complexity of the Dijkstra's Algorithm depends on the number of vertices and is inversely proportional to the number of vertices.
- Time complexity was highest for Bellman Ford than Dijkstra's and A*algorithm.
- Time complexity was worst for A* algorithm.

Performance and Efficiency

Performance and Efficiency

It found that for lower number of nodes the Bellman Ford algorithm is better and for the higher number of nodes the Dijkstra's Algorithm is more efficient.

- The A* algorithm is best suited for the searching for the nodes amongst the large number of nodes.

Debugging Activities

- The Dijkstra's algorithm is easy to implement and thus the debugging of it is also simpler.
- The Bellman Ford algorithm is most difficult to implement and is basically finds its use in the network routers thus is difficult to debug as hardware constraints are also to be considered while implementing.
- A* Algorithm is the variation of the Bellman ford but is basically used as searching algorithm debugging depends on the amount of the data to search among.

VI CONCLUSION

In the result of the study it found that none of the algorithm is capable to handle the bottleneck in network if the number of nodes increases or the network traffic increases.

Following conclusion are drawn from the study for each algorithm

- **Dijkstra's Algorithms**
  - Dijkstra's As per the numerical analysis performed conclude that Dijkstra's algorithm is comparatively more efficient if the number of nodes is more and the implementation is also easier. The applications where

the Network traffic is more the Dijkstra's algorithm based applications are better but for the low traffic networks the Bellman Ford based Systems perform faster.

- Study recommended that implementation of Dijkstra's algorithm best because it is easier to code and will be about as efficient as A*.
- Dijkstra's algorithm radiates out from the initial node.

- **Bellman Ford Algorithms**
  - Efficiency dependent on processing time of algorithms
  - Space and Time complexity depends on amount of information required from other nodes
  - Its implementation is specific and depends on application implementation.
  - It converge under static topology and costs
  - It converge to same solution
  - If link costs depend on traffic, which depends on routes chosen, may have feedback instability.
- **A* Algorithms**
  - A* search algorithm is having better utility if it is be for the searching of the nodes and thus the optimized search applications should use it for better result.
  - The A* algorithm directs its search towards the destination basically, with a heuristic is equivalent to A* except for a couple of technical fact.
  - A* use less memory.
  - The effectiveness of the A* algorithm depends on the heuristic h(x)
  - It run in polynomial time

REFERENCE

[1] SI Lian-fa, WANG Wen-jing.Realization of Optimal Algorithm for Fast Dijkstra Latest Path. *Bulletin of Surveying and Mapping*.2005,(8) pp .15-18.

[2] ZHANG Chi-jun,YANG Yong-jian,ZHAO Hong-bo. Improvement and Realization of the Shortest Path Algorithm Based on Path-dependent.*Computer Engineering and Applications.* 2006,25(2) ,pp.56-58.

[3] WANG Kai-yi,ZHAO Chun-jiang,XU Gui-xian,etc.A High-efficiency Realization Way of the Shortest Path Search Problem in GIS Field.*Journal of Image and Graphics*. 003,8(8) pp. 951-956.

[4] WANG Jin-feng,LI Lian-fa,GE Yong,etc.A Theoretic Framework for Spatial Analysis.*Acta Geographica Sinica.* 2001,55(1) , pp.92-102.

[5] Dijkstra, E. W. "A Note on Two Problems in Connection with Graphs." Numerische Math. 1, 269-271, 1959.

[6] Whiting, P. D. and Hillier, J. A. "A Method for Finding the Shortest Route through a Road Network." Operational Res. Quart. 11, 37-40, 1960.

[7] Richard Bellman(1958) "On a Routing Problem", in Quarterly of Applied Mathematics,16(1),pp.87-90

[8] Lestor R. Ford jr., D. R. Fulkerson(1962.) "Flows in Networks", Princeton University Press,

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein(2001) "Introduction to Algorithms", Second Edition. MIT Press and McGraw-Hill,. ISBN 0-262-03293-7. Section 24.1: The Bellman-Ford algorithm, pp.588–592.

[10] Mongkol Ekpanyapong, Thaisiri Waterwai Sung, Kyu Lim(2006) "Statistical Bellman-Ford algorithm with an application to retiming" Asia and South Pacific Design Automation Conference archive Proceedings of the 2006 Asia and South Pacific Design Automation Conference Pages: 959 - 964

[11] Andrew V. Goldberga, and Tomasz Radzikb,(1993), "A heuristic improvement of the Bellman-Ford algorithm", Applied Mathematics Letters,Volume 6, Issue 3, May 1993, Pages 3-6

[12] Robert Sedgewick. Algorithms in Java. Third Edition. ISBN 0-201-36121-3. Section 21.7: Negative Edge Weights. http://safari.oreilly.com/0201361213/ch21lev1sec7

[13] Jin Y. Yen. "An algorithm for Finding Shortest Routes from all Source Nodes to a Given Destination in General Network", Quart. Appl. Math., 27, 1970, 526-530.

[14] Richard Bellman: On a Routing Problem, in Quarterly of Applied Mathematics, 16(1), pp.87-90, 1958.

[15] Lestor R. Ford jr., D. R. Fulkerson: Flows in Networks, Princeton University Press, 1962.

[16] 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 24.1: The Bellman-Ford algorithm, pp.588–592. Problem 24-1, pp.614–615.