Various Ways of Shortest Path Algorithm

Pawankumar I. Patil
Network Infrastructure Mgmt System,
VJTI
Mumbai, India

Prakash Teltumde
Network Infrastructure Mgmt System,
VJTI
Mumbai, India

C.Shreeharsha
Network Infrastructure Mgmt System,
VJTI
Mumbai, India

Abstract-- Shortest path problem is of most important in the networks and of most interest to study. For example we can consider we want to send packet from one node to another to achieve throughput we required to follow the shortest path available between two nodes and also that path should not be down. We require the systems should quickly solve large shortest path problems. Conventional techniques for solving shortest paths within large networks are either too slow or require huge amounts of storage so are not used. Here we discuss various algorithm like Dijkstra's algorithm, Two-queue algorithm, A*algorithm, restricted algorithm, etc for finding shortest path in large networks. We represent them in another way so that they work efficiently. And also we consider the time performance of the algorithms. And finally we got the A* algorithm is faster as compare to other algorithm for finding the effective shortest path.

Keywords-*Shortest Path Algorithm; Dijkstra's Algorithm; Two-Queue; DKD;DKA,Restricted Search,A* algorithm.*

Corresponding Author: Pawankumar I. Patil

I. INTRODUCTION

An algorithm is a finite set of well-defined instructions that takes some set of values as input and produces some set of values as output to solve a problem. Generally we perform the analysis of algorithm to determine the space and time required by the algorithm and to determine whether the algorithm is working properly that is it is giving proper output? Shortest path algorithm is of most important, since we required it at our day-to-day life like finding shortest way to the place, while sending data over network and etc. Due to the nature of routing applications, we need flexible and efficient shortest path procedures, both from a processing time point of view and also in terms of the memory requirements. We do not have any documentation on selecting the best algorithm. Past research suggests that Dijkstra's implementation with double buckets is the best algorithm for networks with nonnegative arc lengths and implementation of labeling algorithms are fastest for one-to-one searches. Here we will discuss different algorithms like Dijkstra's algorithm, Dijkstra's Navie implementation, Dijkstra's Symmetrical algorithm, Two-Queue algorithm, MildTwo-queue algorithm and there comparision. Then we go for Dijkstra's algorithm implemented with approximate Buckets and Double Buckets, Restricted search, A* search and weighted A* search. Finally we draw some conclusion.

II. LITERATURE SURVEY

1) Dijkstra's algorithm

Dijkstra's shortest path algorithm is used to solve the single source shortest path problem. For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

Pseudocode:

```
function Dijkstra(Graph, source):
for each vertex v in Graph:          // Initializations
dist[v] := infinity ;       // Unknown distance function
from source to v
previous[v] := undefined ;          // Previous node in optimal path from source
end for ;
dist[source] := 0 ;           // Distance from source to source
Q := the set of all nodes in Graph ;
// All nodes in the graph are unoptimized - thus are in Q
while Q is not empty:                // The main loop
u := vertex in Q with smallest dist[] ;
if dist[u] = infinity:
break ;                  // all remaining vertices are inaccessible from source
end if ;
remove u from Q ;
for each neighbor v of u:       // where v has not yet been removed from Q.
alt := dist[u] + dist_between(u, v) ;
if alt < dist[v]:            // Relax (u,v,a)
dist[v] := alt ;
previous[v] := u ;
end if ;
end for ;
end while ;
return dist[] ;
  end Dijkstra
```

2) Dijkstra's Naive Implementation

Many shortest path algorithms have Dijkstra's labeling method as central procedure. The labeling method produces an out-tree as output. An out-tree is a tree, which starts from the source node to nodes whose shortest distance from the source node is known. The out-tree is constructed iteratively, and the shortest path from source to any destination node in the tree is obtained by terminating the method. The information required for each node *i* in the labelling method while constructing the shortest path tree:

- the distance label, *d(i)*
- the parent-node/predecessor *p(i)*
- the set of permanently labeled nodes .

The distance label *d(i)* stores an upper bound on the shortest path distance from source to *i*, while *p(i)* records the node that immediately precedes node *i* in the out-tree. If a node has not been added to the out-tree, then it is considered unreached node. The distance label of an unreached node is set to infinity. The absolute shortest path from source node to node *i* is called permanently labeled. A node is temporarily labeled if there are changes to be made in the node. Dijkstra's algorithm guarantees optimality by repeated addition of a temporarily labeled node with the smallest distance label *d(i)* and the set of permanently labeled nodes.

3) Symmetrical Dijkstra Algorithm

Dijkstra's shortest path algorithm is adapted by Pohl to decrease the size of the search space. Pohl was the first to use a bi-directional search method. The algorithm has forward search from an origin node to the destination node and a backward search from the destination node to the origin node. It is done to reduce the search complexity compared to the Dijkstra's algorithm. The search method assumes that the two searches grow symmetrically and will meet in some middle area. The Symmetrical or Bi-directional Dijkstra's algorithm by Pohl makes two search trees, one from the origin, giving a tree spanning a set of nodes *LF* for which the minimum distance/time from the origin is known, and a second from the destination that gives a tree spanning a set of nodes *LB* for which the minimum distance/time to the destination is known. We goes on adding node to either *LF* or *LB* until there exists an arc crossing from *LF* to *LB*. Like Dijkstra's algorithm Pohl's bi-directional search chooses the node with the smallest cost label to label permanently. Dijkstra criterion required for optimality is maintained by selecting the new permanently labeled node from either the forward or backward phases.

4) Two-Queue

Pallottino introduces the graph growth algorithm implemented with two queues (TQQ) in 1984. TQQ is an improved version of the growth graph implementation developed by Pape (PAP) in 1974.

The four basic operations involved in shortest path tree constructing procedure are:

Queue_Initialization(Q) initialize queue Q;

Queue_Removal(Q, i) remove node i from queue Q;

Queue_Insertion(Q, j) insert node j into queue Q;
Q = Null? check whether queue Q is empty

The major difference between TQQ and PAP is in the Queue Insertion(Q, j) operation. In the implementations of PAP and TQQ, nodes are partitioned into two sets, first set contains the nodes whose current distance labels are not used to find a shortest path and the second includes the remaining nodes. The first set of nodes is maintained by a priority queue Q. Nodes in the second set are further split into two categories, first is the unreached nodes which are never entered Q, i.e., nodes having distance labels as infinite, and second is labeled nodes, i.e., the

nodes which are inserted into Q at least once, and the nodes whose current distance labels are already used.

A stack is used as a priority queue in the PAP implementation. A PAP algorithm is enhance by replacing stack with a FIFO queue. Here new data structure is constructed called two-queue. Because both Q' and Q" are queues in the two-queue data structure, nodes can be inserted at the end of Q' and Q", and they can be removed from the head of Q' and Q".

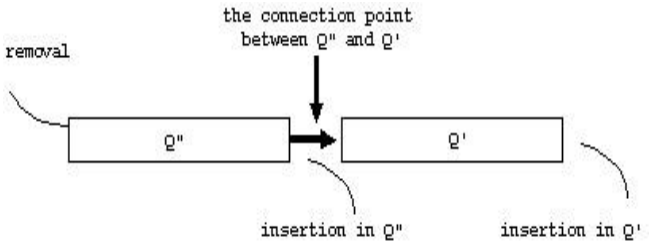

Fig. 1 Two-queue data structure (Q) consisting of Q" and Q'

The node is inserted at the end of Q' if it is unreached, or the node is inserted at the end of Q" if it is temporarily labeled. So the change in the Queue insertion(Q, j) operation of the PAP implementation are as follow and other operations remains same.

Queue_Insertion(Q, j)

> For any node j that is not already in Q,insert the node at the end of Q' if the node is unreached, i.e., if S(j) = unreached or insert the node at the end of Q" if the node is temporarily labeled.

5) MiLDTWO-Q

The main steps of the new algorithm are the same as TWOQ, and one variable (*min_label*) is added in the new algorithm to trace the minimum label.

The four basic operations involved in this procedure are:

Initialization(): (same as TWO-Q with one instruction added in the end)
min_label=0;

Extract(i): (the same as TWO-Q)

Scan(i)：(the same as TWO-Q)

Compare(i)：
if（dest_node.state!=UNREACHED&&i.label== in_label）
{
min_label= dest_node.label;
for all node p, p∈ V and p.state==IN_QUEUE

```
{
if(p.label< min_label)
min_label= p.label;
}
if(min_label >=dest_node.label)
the algorithm stops;
}
```

MiLD-TWO-Q can stop in time, the searching space is reduced. But sometimes it requires extra work of updating the *min_label.*

6) Dijkstra's algorithm implemented with Approximate Buckets(DKA)

In Dijkstra's original algorithm, temporarily labeled nodes are treated as a non ordered list. This is of course a bottleneck operation because all nodes in Q required to be visited at each iteration in order to select the node with the minimum distance label. A enhancement of the original Dijkstra algorithm is to maintain the labeled nodes in a data structure in such a way that the nodes are sorted by distance labels. The bucket data structure is just one of those structures. Buckets are sets arranged in a sorted fashion as shown in fig. Nodes contained in each bucket can be represented with a doubly linked list. A doubly linked list only requires O(1) time to complete an operation in each distance update in the bucket data structure. These operations include checking if a bucket is empty, adding an element to a bucket, and deleting an element from a bucket.

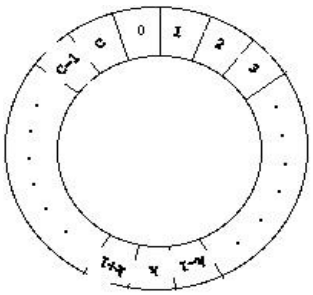

Fig. 2 Bucket data structure

Dial(1969) was the first to implement the Dijkstra's algorithm using buckets. In Dial's implementation, bucket m contains all temporarily labeled nodes whose distance labels are equal to m. Buckets 0, 1, 2, 3, .. checked sequentially until the first nonempty bucket is identified. The first entry of nonempty bucket has the node with minimum distance label by definition. The nodes with the minimum distance label become permanently labeled and are deleted from the bucket during the scanning process. The position of a temporarily labeled node in the buckets is updated accordingly as the distance label of a node changes. This is repeated until all nodes are permanently labeled. Dial's original implementation of the Dijkstra algorithm (DKB) requires *nC+1* buckets in the worst case, where *C* is the maximum arc length of a network. However, it has been proven that for a network with a maximum arc length *C*, only *C+1* buckets are needed to maintain all temporarily labeled.

It can be seen that this algorithm trades speed for space. Each node can be scanned more than once, but a node cannot be scanned more than b times.

7) Dijkstra's algorithm implemented with double buckets (DKD)

The double bucket implementation of the Dijkstra's algorithm (DKD) combines the ideas of two algorithms DKM and DKA. Two levels of buckets, high-level and low-level, are maintained in the DKD implementation. A total of *d* buckets in the low-level buckets are used. A bucket *i* in the high-level buckets contains all nodes whose distance labels are within the range of [*i*d, (i+1)* d-1*]. In addition, a nonempty bucket with the smallest index *L* is also maintained in the high-level buckets. A low-level bucket *d(j)-L*d* maintains nodes whose distance labels are within the range of [*L*d, (L+1)* d-1*]. Nodes in the low-level buckets are examined during the scanning process. After all nodes in the low-level buckets are scanned, the value of *L* is increased. When the value of *L* increases, nodes in the nonempty high-level buckets are moved to its corresponding low-level buckets, and the next cycle of scanning process begins.

8) Restricted search

The Dijkstra's algorithm starts search from the start point and spreads as a circle until the radius arrives the destination. Most searches at the area opposite the direction of destination are useless. M Fu et al described an optimal approach to find shortest path for Vehicle Navigation System by physically cutting off area within which the shortest path is not supposed to appear. Instead of searching the entire circle, the Restricted Search Method only searches the small area of the remaining part.

The Dijkstra's algorithm is used as the same way. However, instead of relaxing all adjacent nodes in each iteration, the algorithm filters out the nodes beyond the restricted area by checking if they are out of range.

9) A Search*

So far we have examined search techniques that can be generalised for any network (as long as it does not contain negative length cycles). However the physical nature of real road networks motivates investigation into the possible use of heuristic solutions that exploit the near-Euclidean network structure to reduce solution times while hopefully obtaining near optimal paths. For most of these heuristics the goal is to bias a more focused search towards the destination. As we see incorporating heuristic knowledge into a search can dramatically reduce solution times. When the underlying network is Euclidean or approximately Euclidean as is the case of road networks, then it is possible to improve the average case run time of the Dijkstra's and Symmetrical Dijkstra's algorithms. This is usually at the expense of optimality; solutions are now not guaranteed to be the best. Typically when solving problems on such networks the inherent geometric information is ignored by algorithms that are directly based or variations on Dijkstra's labelling algorithm.

The A* algorithm by Hart and Nilsson formalised the concept of integrating a heuristic into a search procedure. Instead of choosing the next node to label permanently as that with the least cost (as measured from the start node), the choice of node is based on the cost from the start node plus an estimate of proximity to the destination. To build a shortest path from the origin to the destination, we use the original distance from source accumulated along the edges plus an estimate of the distance to destination. Thus we use global information about our network to guide the search for the shortest path from source to destination. This algorithm places more importance on paths leading towards *t* than paths moving away from *t*.

The A* algorithm combines two pieces of information:

i) the current knowledge available about the upper bounds (given by the distance labels *d(i)*), and

ii) an estimate of the distance from a leaf node of the search tree to the destination.

The A* algorithm integrates a heuristic into a search procedure. Instead of choosing the next node with the least cost (as measured from the start node), the choice of node is based on the cost from the start node plus an estimate of proximity to the destination (a heuristic estimate). F. Engineer described this approach to solve the problem of optimal path finding.

The A* Search algorithm:

```
for each u G:
d[u] = infinity;
parent[u] = NIL;
End for
d[s] = 0;
f(V) = 0;
H = {s};
while NotEmpty(H) and targetNotFound:
u = Extract_Min(H);
label u as examined;
for each v adjacent to u:
if d[v] > d[u] + w[u, v] , then
d[v] = d[u] + w[u, v];
p[v] = u;
f(v) = d[v] + h(v, D);
DecreaseKey[v, H];
```

10) Weighted A Search*

By choosing appropriate multiplicative factor we can increase the contribution of estimated component in calculating label of a vertex. From an intuitive standpoint this corresponds to further biasing the forward search towards the destination and the backward search towards the origin. The heuristic is parameterised by the multiplicative factor termed the "overdo" parameter used to weight the evaluation function. This modification will generally not yield optimal paths, but we would expect it to further reduce the search space. The aim is to find an optimal multiplicative or overdo factor for which the running time is significantly improved while the solution quality is still acceptable. Thus there will be an empirical time/performance trade-off as a function of the overdo parameter.

III. COMPARISION

From this graph it is seen that Symmetric Dijkstra's algorithm is two times faster than Dijkstra's algorithm, and A* algorithm is three times faster than Dijkstra's algorithm.

Both the Dijkstra and the Symmetrical Dijkstra algorithm guarantee optimality, hence we would expect them to have 0 PIR. However our implementations of the A* and Radius algorithms do not guarantee optimal results.

We can see that the average PIR of the A* algorithm is almost level at approximately 0.5, the largest PIR for our network is within 5%.

The running time and accuracy of the algorithms with respect to different distances

Distance	5009	11262	15164	21498	34250	50227
Dijkstra's	25⁄20	55⁄50	100⁄80	180⁄120	401⁄300	410⁄300
A*	24⁄19	45⁄40	70⁄60	130⁄100	310⁄250	410⁄325
Accuracy	1.0	1.0	1.0	1.0	1.0	1.0
Restricted	12⁄10	20⁄15	40⁄30	68⁄40	100⁄70	170⁄150
Accuracy	1.28	1.02	1.26	1.2	1.2	1.05

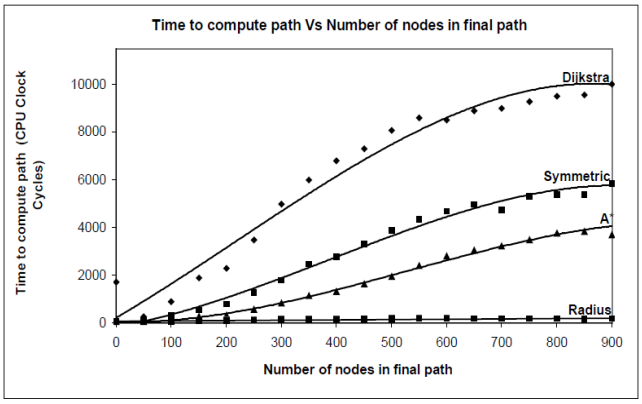

Fig. 3 Time to compute paths using different Algorithms

The time complexity of Dijkstra's algorithm is O((V + E)*log(V) = O(E* log(V)). As the number of nodes in a graph increases, the running time of the applied algorithm will become longer and longer. When the shortest path is relatively short, MiLD-TWO-Q runs much more quickly than TWO-Q and the runtime of MiLD-TWO-Q increases with increasing shortest path distance. MiLD-TWO-Q stops in time and cut down the number of scans.

Although there has been considerable reported research related to the evaluation of the performance of shortest path algorithms, there has been no clear answer as to which algorithm or a set of algorithms runs fastest on real road networks in the literature. A recent evaluation of shortest path algorithms using real road networks has identified The Graph Growth Algorithms implemented with two queues (TQQ), The Dijkstra's algorithm implemented with approximate buckets (DKA), and The Dijkstra's algorithm implemented with double buckets (DKD) are fastest.

A* algorithm uses the same approach as Dijkstra's except that it uses accumulated cost of edge plus the Euclidean distance from current node to the destination. A* algorithm does not improve worst case time complexity, but it improves average time complexity. The run time of A* is much shorter than the Dijkstra's algorithm.

The restricted algorithm can find the optimal path
within linear time but the restricted area has to be carefully selected. The selection actually depends on the graph itself.

IV. CONCLUSION

Finally we come at the conclusion as the A* algorithm with Eucledian heuristic functional is fastest amoung all the algorithms. It also guarantee to find shortest path.

The performance of A* heuristic search algorithm can be increased by memorizing the path cost of distance from source to node and the estimated distance from node to destination.For finding shortest path between new nodes the algorithm will first checks the data stored at that node and find the nodes having shortest path from it then we proceed further.

REFERENCES

[1] Chabini and B. Dean, "Shortest path problems in discrete time dynamic networks: Complexity, algorithm, and implementations", Massachusetts Institute of Technology, Cambridge, MA, Tech. Resp. 1999.

[2] Fast Shortest Path Algorithms for large Road networks by Faramroze engineer, Department of Engineering Science, University of Auckland, New Zeland.

[3] Ahuja, R. K., Mehlhorn, K., Orlin, J. B., and Tarjan, R. E. (1990) Faster algorithms for the Shortest Path Problem. Journal of Association of Computing Machinery, 37, 213-223.

[4] F. B. Zhan, "Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures" Journal of Geographic Information and Decision Analysis, vol. 1, pp. 69-82, 1997

[5] Hung, M. H., and Divoky, J. J. (1988) A Computational Study of Efficient Shortest Path Algorithms. Computers & Operations Research, 15, 567-576.

[6] Faramroze Engineer. Fast Shortest Path Algorithms for Large Road Networks.Department of Engineering Science. University of Auckland, New Zealand.

[7] Thomas Willhalm. Speed Up Shortest-Path Computations. January 27, 2005.

[8] Pearsons J. (1998) Heuristic Search in Route Finding. Master's Thesis, University of Auckland.

[9] I/O-Efficiency of Shortest Path Algorithms: An Analysis, Bin Jiang, Department of Computer Science, Information Systems-Databases ETH Zurich CH-8092 Zurich, Switzerland.

[10] A Performance Analysis of Hierarchical Shortest Path Algorithms, A. Fetterer Oracle Corporation 500 Oracle Parkway Redwood Shores, CA 94065 S. Shekhar Department of Computer Science University of Minnesota Minneapolis, MN, 55455.