Shortest Path Algorithms in Transportation Networks

V.V.S.Chandra Mouli[1], S.Meena Kumari[2], N.Geethanjali[3]

GJCST Computing Classification C.2.1 & C.2.5

***Abstract*- Shortest Path problems are among the most studied network flow optimization problems with interesting applications in a wide range of fields. One such application is in the field of GPS routing systems. These systems need to quickly solve large shortest path problems but are typically embedded in devices with limited memory and external storage. Conventional techniques for solving shortest paths within large networks cannot be used as they are either too slow or require huge amounts of storage. In this project we have tried to reduce the runtime of conventional techniques by exploiting the physical structure of the road network and using network pre-processing techniques. Our algorithms may not guarantee optimal results but can offer significant savings in terms of memory requirements and processing speed. Our work uses heuristic estimates to bind the search and directs it towards a destination. We also associate a radius with each node that gives a measure of importance for roads in the network. The farther we get from either the origin or destination the more selective we become about the roads we travel with greater importance (i.e. roads with larger radii).By using these techniques we were able to dramatically reduce the runtime performance compared to conventional techniques while still maintaining an acceptable level of accuracy.**

Keywords- Routing, Shortest Path, Network, Radius.

I INTRODUCTION

We consider a long-studied generalization of the shortest path problem, in which not one but several short paths must be produced. The k shortest paths problem is to list the k paths connecting a given source-destination pair in the digraph with minimum total length. Our techniques also apply to the problem of listing all paths shorter than some given threshold length. 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. Unfortunately, prior research does not provide a clear direction for choosing an algorithm when one faces the problem of computing shortest paths on real road networks. Past research in testing different shortest path algorithms suggests that Dijkstra's implementation with double the best algorithm for networks with nonnegative arc lengths [1, 2]. However like most popular papers on Shortest Path algorithms, they have concentrated their focus on algorithms that guarantee optimality and have worked on tuning data structures used in implementing these algorithms. Since no "best" algorithm currently exists for every kind of transportation problem, research in this field has recently moved to the design and implementation of "heuristic" shortest path procedures, which are able to capture the peculiarities of the problem under consideration and improve the run time performance of a search, but at the cost of not guaranteeing optimality. As it is impossible to cover all search implementations, we use Dijkstra's algorithm as a building block to create an efficient search algorithm that implements an artificial intelligence approach to the routing problem that may not guarantee optimal results but gives significant savings in terms of memory requirements and processing speed. In the version of these problems studied here, cycles of repeated vertices are allowed. We first present a basic version of our algorithm, which is simple enough to be suitable for practical implementation while losing only a logarithmic factor in time complexity. We then show how to achieve optimal time (constant time per path once a shortest path tree has been computed) by applying Frederickson's algorithm for finding the minimum k elements in a heap-ordered tree.

Manuscript received "Feb 21, 2010 at 12:06 PM GMT"

1st V.V.S.Chandra Mouli, & 2nd S.Meena Kumari, Assistant Professor, G.Pulla Reddy Engineering College, Kurnool , A.P, India
(Email: chandu2527@gmail.com)
3rd N.Geethanjali, Associate Professor, Computer Science And Technology Department,Sri Krishna Devaraya University ,Anantapur,A.P,India
(Email: anjali.csd@yahoo.com)

II DIFFERENT SEARCH ALGORITHMS

In the following subsections we discuss about different searching techniques.

A. *Intelligent Transport System*

To fully appreciate the merits of a search technique it is important to understand the commercial environment in which these techniques are implemented. Many route finding systems are currently in development worldwide and the majority form part of much larger systems to our paper manage and operate the road network more efficiently. These management infrastructures are known as Intelligent Transport Systems (ITS) and vary in complexity and size. These systems fall into two main categories, centralized and decentralized systems [3]. Centralized systems are linked to an information centre which collates and processes traffic and network information. Typically a driver requests a particular route from onboard electronics. The route is then relayed to a central location that carries out all the processing of the route. Decentralized systems on the other hand offer information to the driver which is computed onboard using local information sources. Typically such

systems contain road network information on optical storage devices and electronics to feed a GPS.

B. Network Definitions

Before continuing let us introduce some notation and formally define the shortest path Problem. A network is a graph G = (N, A) consisting of a unique indexed set of nodes N With n = |N| and a spanning set of directed arcs A with m = |A|. Each arc a is represented as an ordered pair of nodes, in the form "from node i to j", denoted by a = (i,j). Each arc (i,j) has an associated numerical value l_{ij}, which represents the Distance, time or cost incurred by traversing the arc. Each node i has a set of successors S(i) (i.e. the set of all nodes j: (i,j)∈A) and predecessors P(i) (i.e. set of all nodes j: (j,i) ∈A).

C. Search Algorithms

One possible approach to solving shortest path problems would be to pre-calculate and store the shortest path from every node to every possible other node, which would allow us to answer a shortest path query in constant time. Unfortunately the required storage size and computation time grows with the square of the number of nodes. With realistic road networks in mind this processing would take years if not decades and be impossible to store. Hence to overcome this problem we require real time search techniques. From previous studies [1, 2, 4] we know that the implementation of labeling algorithms are the fastest for one-to-one searches.

Two aspects are particularly important to the shortest path algorithms discussed in this project:

i. The strategies used to select the next node to be visited during a search, and
ii. The data structures utilized to maintain the set of previously visited nodes.

A number of data structures can be used to manipulate the set of nodes in order to support search strategies. These data structures include arrays, singly and doubly linked lists, stacks, heaps, buckets and queues. Detailed definitions and operations related to these data structures are standard knowledge and are well documented. Past research has concentrated mainly on the issue of data structures, which can be manipulated and bounded to form clever techniques in creating priority queues for selecting nodes to be scanned. A good example of this is the Dijkstra implementation with double buckets [1]. In a labeling algorithm, the number of visited nodes during a search is a good indication of the size of the search space. This means that a search strategy which visits fewer nodes during a search is generally more efficient in terms of processing speed. The number of nodes visited depends on the depth d (i.e. the number of arcs on the optimal path) of the destination from the origin, and the branching factor b. For a 'best first search' the number of nodes explored during a search is of the order O(bd) [3]. This exponential growth in the number of explored nodes is known as "combinatorial explosion" and is the main obstacle in computing shortest paths in large networks. (Note that even though Dijkstra's algorithm is polynomial in the number of nodes n in the graph, this bound is no restriction on how the number of nodes visited varies with d). For general search this exponential growth with depth makes many problems unsolvable on current hardware, as memory is soon exhausted and a solution may take an unreasonable time to compute. These effects can be lessened by using artificial intelligence (heuristic type) techniques which will be discussed later. However let us first define and implement Dijkstra's labeling algorithm.

D. Dijkstra's Naive Implementation

Your Dijkstra's labeling method is a central procedure in shortest path algorithms. The output of the labeling method is an out-tree from a source node s, to a set of nodes L. An out-tree is a tree originating from the source node to other nodes to which the shortest distance from the source node is known. This out-tree is constructed iteratively, and the shortest path from s to any destination node t in the tree is obtained upon termination of the method.

Three pieces of information are required for each node i in the labeling method while constructing the shortest path tree:

i. The distance label, d (i),
ii. The parent-node/predecessor p (i),
iii. The set of permanently labeled nodes L.

The distance label d(i) stores an upper bound on the shortest path distance from s to i, while p(i) records the node that immediately precedes node i in the out-tree. If a node has not yet been added to the out-tree, it is considered 'unreached'. Normally the distance label of an unreached node is set to infinity. When we know that the shortest path from node s to node i is also the absolute shortest path, then node i is called permanently labeled. When further improvement is expected to be made on the distance from the origin to node i, then node i is considered only temporarily labeled. It follows that d(i) is an upper bound on the shortest path distance to node i if node i is temporarily labeled, and d(i) represents the final optimal shortest path distance to node i if the node is permanently labeled [1,2]. By iteratively adding a temporarily labeled node with the smallest distance label d(i) to the set of permanently labeled nodes L, Dijkstra's algorithm guarantees optimality. One advantage with Dijkstra's labeling algorithm is that the algorithm can be terminated when the destination node is permanently labeled. Most other algorithms guarantee optimal shortest paths only upon termination when the entire shortest path tree has been explored.

E. Symmetrical Dijkstra Algorithm

Pohl adapted Dijkstra's shortest path algorithm to decrease the size of the search space [1]. Pohl's algorithm was the first to use a bi-directional search method. This algorithm consists of a forward search from an origin node to the destination node and a backwards search from the destination node to the origin node. This was done in an attempt to reduce the search complexity to $O\ (b^{d/2})$

compared to $O(b^d)$ as with Dijkstra's algorithm. This search method assumes that the two searches grow symmetrically and will meet in some middle area. Sometimes this might not be the case, and as a worst-case scenario, this might instead become two $O(b^d)$ searches. The symmetrical or Bi-directional Dijkstra's algorithm by Pohl grows 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 iteratively add one 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. By selecting the new permanently labeled node from either the forward or backward phases we maintain the Dijkstra criterion required for optimality.

F. A Search*

So far we have examined search techniques that can be generalized 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 shall 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 and Symmetrical Dijkstra algorithms. This is usually at the expense of optimality; solutions are now not guaranteed to be the best. Typically when solving problems on directly based or variations on Dijkstra's labeling algorithm

.The A* algorithm by Hart and Nilsson [2] formalized 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 (a heuristic estimate) [4]. To build a shortest path from the origin s to the destination t, we use the original distance from s accumulated along the edges (as in Dijkstra's algorithm) Plus an estimate of the distance to t. Thus we use global information about our network to guide the search for the shortest path from s to t. This algorithm places more importance on paths leading towards t than paths moving away from t. In essence 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.

There are several ways to estimate the lower bound from a leaf node in the search tree to the destination node. These estimations are carried out by so called "evaluation" functions [3]. The closer this estimate is to a tight lower bound on the distance to the estimation, the better the quality of the A* Search. Hence the merits of an A* search depends highly on the evaluation function h(i,j). There are two main evaluation functions used in the A* search. A true lower bound between two points is the length of a straight line between those two points (i.e. the Euclidean distance):

$$H_E(i,t) = \text{sqrt}[(x(i)-x(t))^2 + (y(i)-y(t))^2]$$

where x(i), y(i) and x(t), y(t) are the coordinates for node i and the destination node t respectively. The other commonly used evaluation function is the Manhattan distance hM. In this case the estimated lower bound distance is the sum of distance in the x and y coordinates.

$$H_M(i,t) = |x(i) - x(t)| + |y(i) - y(t)|$$

The Manhattan distance is not the true lower bound between two points and hence will typically yield non-optimal results. By using time as a measure of cost, the network becomes near-Euclidean. This is because of the varying speeds of roads in the network. Roads of similar lengths might have different times associated with using those roads. If the network is not strictly

Euclidean but near-Euclidean then our selection criteria for the next node to label permanently will not yield optimal results. By using the A* search, the shortest path tree should now grow towards t (unlike Dijkstra's algorithm where the tree grows approximately radially). As before, the search for the shortest path is terminated as soon as t is added to the shortest path tree. Earlier we discussed the problem of combinatorial explosion with a blind search time complexity in the order of $O(b^d)$. With A* search this is reduced to $O(b_e^d)$ where b_e is the effective branching factor. The A* search reduces the search space by reducing the number of node expansions. Although A* is still susceptible to the problem of combinatorial explosion, it decreases the effect by reducing the size of the base in the complexity term.

G. Weighted A Search*

By choosing an appropriate multiplicative factor we can increase the contribution of the estimated component in calculating the label of a vertex (i.e. increase the contribution of the evaluation function) [4]. 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 parameterized 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 over do 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.

H. Radius Search

To eliminate or minimize the effects of combinatorial explosion we need to adopt a search technique similar to the way humans approach navigation problems. So far we have not implemented any intelligence once within a search which can filter out roads that are less likely to be traveled on. This type of intelligence requires some form of historical knowledge about the network. Since the road network does not change very often it is possible to calculate auxiliary information in a pre-processing step. Perhaps the most obvious way to classify the roads in the network is to identify the class of each road (i.e. motorways, highways, local roads etc), and then to exploit these classes in the search. This is similar to the way humans approach routing problems and is known as Hierarchical Search [3,5]. Hierarchical methods offer the prospect of greatly reducing the size of any search by simplifying the search through a series of simplified levels, where each of these levels is an abstraction of the previous level. These abstractions reduce the overall size of the search space that an algorithm addresses and thus the complexity of any search is reduced. For route finding, hierarchical levels are constructed in which higher speed roads are placed higher up in the hierarchy. However by introducing these arbitrary hierarchies the path optimality is often lost [3].

The hierarchical algorithm uses a discrete number of hierarchy levels. A Radius search is a hierarchical search with a continuous range of hierarchy levels. A Radius search takes advantage of the fact that the fastest path between two junctions is more likely to use a highway than a local road, especially if the two junctions are far apart. In this method each node i has an associated radius r(i). Before we consider how r(i) is calculated, we first examine how radii can be used to restrict a search. When looking for a shortest path from s to t, a node i is considered as a possible node to include in the search only if s or t lies inside a circle of radius r(i) cantered at node i. If both distances are greater than the node radius, the node is simply ignored [5].For any given origin and destination node, we can immediately simplify the network by removing all the nodes (and associated arcs) whose radii do not encircle the origin or destination nodes. The radius search is not a search algorithm by itself, but an independent mechanism of reducing search complexity. Hence the radius concept can be used in conjunction with any search algorithm.

The optimal radius for a node i is the smallest radius r(i) for which the radius centred at node i encircles either the origin or destination node for all optimal paths that include node i. If the radii are calculated as a maximum over all such shortest paths, then it is guaranteed that the radius search algorithm is exact (i.e. guaranteed optimality). The radii are also minimal since with any smaller radius at least one optimal shortest path will not be found. One possible difficulty is that the calculation of the radii by examining all paths over a particular node takes much too long since every possible shortest path in the network has to be calculated at least once. Instead we implemented a heuristic approach to calculate these radii [5]. In the first phase of this heuristic approach we divide the network into overlapping grids of approximately 2000 nodes and initialize all node radii to be 0. We then select a random starting node s from all possible nodes N and a random destination node t within the same grid as s. Using the Symmetric Dijkstra algorithm we solve for the shortest path R from s to t. We continue this process of selecting random starting and destination nodes and updating the radii of nodes in the shortest path as many times as possible.

If we do not generate enough random paths in the first phase then the radii of some nodes will never have been updated and hence will still be 0. However if a node is a 'closed node' (i.e. the node is only used in a shortest path if it is either the origin or destination of that shortest path) then it will never be part of a shortest path unless we start or finish at that node. Hence the radii of closed nodes will always be 0. In the second phase of this modified algorithm we go through all nodes in the network and examine their radii. If a node is not closed and has 0 radius, then we conduct shortest path searches in the vicinity of the nodes that generate a reasonable lower bound on its radius. We do this in the second phase by creating a sub graph of 200 of the closest nodes and associated arcs GSUB2 to the node with 0 radius and solve all-to-all shortest paths on GSUB2. This should force some shortest paths R through this node and give it a better radius lower bound than 0. So far in the first two phases we have calculated shortest paths within grids. Hence the radii are no larger than the grids they are created in. As a result, after the first two phases we have a fairly good coverage of local radii only (i.e. these radii only restrict a search for shortest paths within grids). If we were to use these radii to restrict a search over a large distance (i.e. over several grids) then we would not be able to find a path because no nodes exist which have radii greater than the size of a single grid. To travel over large distances we need to calculate radii of roads such as highways and motorways

III CONCLUSION

By exploiting the physical structure of road networks, the A* algorithm is able to bias its search towards a goal and reduce the search space. By using the concept of radii as a measure of importance of nodes, we are able to incorporate pre-processing within our shortest path algorithm to further restrict the search space. This dramatically reduces the search complexity in terms of the run time performance while still maintaining an acceptable level of inaccuracy. For a one to one shortest path or the shortest paths from one to some, it may be worthwhile to consider one of the Dijkstra 's implementations. But Dijkstra implementations depend on the maximum size of the network arc lengths Dijkstra approximate buckets implementation (DIKBA) is recommended for less arc length. For problems with a maximum arc length greater than 1500, the Dijkstra double buckets (DIKBD) implementation should also be considered since it appears to be less sensitive to problems in data set 1

with very large arc lengths. The Bellman Ford Moore implementations with parent checking (BFP) have serious difficulties on large networks. So this algorithm is not recommended for road network and for being coded in a GIS package. This system can efficiently generate less similar paths and provide users more wide choices than other system. Because of the simplicity of the topological structure and the k-shortest path algorithm, the developer can also easily develop a rich featured user interface for displaying and setting.

IV References

1) Coello, C. A. C.,(1999). An Updated Survey of Evolutionary Multiobjective Optimization Techniques: State of the Artand Future Trends,(1999). Proceedings of the Congress on Evolutionary Computation, 1, 3-13, IEEE Press,6-9
2) Cormen T.H, Introduction to Algorithms. MIT Press, Massachusetts, USA. Cherkassy B V, Goldberg A V and Radzik T. (1993) Shortest Paths Algorithms:
3) Theory and Experimental Evaluation. Research project, Department of Computer Science, Cornell and Stanford Universities and Krasikova Institute for Economics and Mathematics.
4) Hart P E and Nilson N J. (1968) A formal basis of the heuristic determination of minimum cost paths. IEEE Transactions of Systems Science and Cybernetics 4(2), 100-107. 45 Pearsons J. (1998) Heuristic Search in Route Finding. Master's Thesis, University of Auckland.
5) Sedgewick R and Vitter J S. (1986) Shortest Paths in Euclidean Graphs Algorithmica 1, 31-48.
6) Ertl G.(1996) Optimierung und Kontrolle, Shortest Path Calculations in Large Road Networks. Project in Discrete Optimisation, Karl-Franzens-Universitat Graz .
7) Aggarwal, B. Schieber, and T. Tokuyama (1993). Finding a minimum weight K-link path in graphs with Monge property and applications. Proc. 9th Symp. Computational Geometry, pp. 189–197. Assoc. for Computing Machinery.
8) R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. (1990) Faster algorithms for the shortest path problem. J. Assoc. Comput. Mach. 37:213–223. Assoc. for Computing Machinery.
9) Azevedo, M. E. O. Santos Costa, J. J. E. R. Silvestre Madeira, and E. Q. V. Martins (1993). An algorithm for the ranking of shortest paths. Eur. J. Operational Research 69:97–106.