A Comparative Study of k-Shortest Path Algorithms

A. W. Brander
M. C. Sinclair
Dept. of Electronic Systems Engineering, University of Essex,
Wivenhoe Park, Colchester, Essex C04 3SQ.
Tel: 01206-872477; Fax: 01206-872900; email: mcs@essex.ac.uk

Abstract
Efficient management of networks requires that the shortest route from one point (node) to another is known; this is termed the shortest path. It is often necessary to be able to determine alternative routes through the network, in case any part of the shortest path is damaged or busy. The k-shortest paths represent an ordered list of the alternative routes available. Four algorithms were selected for more detailed study from over seventy papers written on this subject since the 1950's. These four were implemented in the 'C' programming language and, on the basis of the results, an assessment was made of their relative performance.

1 The Background

The shortest path through a network is the least cost route from a given node to another given node, and this path will usually be the preferred route between those two nodes. When the shortest path between two nodes is not available for some reason, it is necessary to determine the second shortest path. If this too is not available, a third path may be needed. The series of paths thus derived are known collectively as the k-shortest paths (KSP), and represent the first, second, third, ..., kth paths typically of least length from one node to another. In obtaining the KSPs, it is normally necessary to determine independently the shortest path ($k = 1$) between the two given nodes before computation of the remaining $k - 1$ shortest paths can be carried out.

Computation of shortest paths has been the subject of much research since the 1950s. It is important to realise that the term 'shortest' does not just apply to the distance between two nodes, but can involve any single component made up of one or more factors, including cost, safety or time, that put a weighting on the route. KSP algorithms are thus widely used in the fields of telecommunications, operations research, computer science and transportation science.

2 The Problem

The work presented here was driven by the desire to find a faster algorithm to calculate the KSPs between nodes in a network than that, Yen [1], used to date by the second author. It was realised that Lawler [2] had investigated this problem and improved on Yen [1], and this raised the question of whether there were even faster methods.

3 Definitions

In order to follow the solutions for the KSP problem, it is necessary to understand the basic ideas associated with many areas of modern mathematics, in particular graph theory. The following definitions explain some of the terms used in this paper.

The network is represented as a graph $G = (V, E)$ where V is a finite set of n *nodes* (or vertices) $V(G)$ and E is a finite set of m *edges* (*i.e.* links or arcs) $E(G)$ that connect the nodes. An edge is often represented as $e_p(i, j)$ with a certain weight, where i and j are the nodes at the endpoints of the edge e_p. A simple graph representation of a network is shown in Figure 1, where the nodes are shown as numbered discs and the edges as lines linking the nodes. A graph is said to be *connected* if there is a path between every pair of nodes in the network.

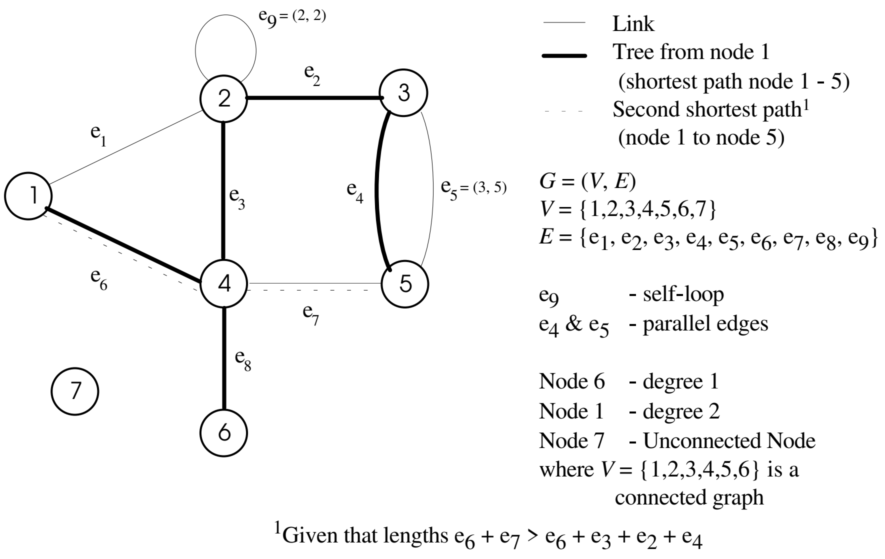

Figure 1: Network drawn as a graph

The terms *path* and *route* may be defined as the sequence of nodes and edges that need to be alternately followed in order to move from a start node to an end node. An *undirected* network is one where all edges in the network are bi-directional, such that travel is possible along the edge in both directions, and generally $e_p(i,j) = e_p(j,i)$. It is possible that there may be two edges between some pairs of nodes and these edges are referred to as *parallel* (or multiple) edges. A *self-loop* is an edge that connects a node to itself.

The *hop count* is a count of the number of edges (or hops) in a path. An *elementary path* or *simple path* is a path containing no *loops*, such that each node is visited at most once and the maximum path hop count possible is given by the total number of nodes minus one (*i.e.* $n-1$). The weight associated with an edge may be positive or negative, but in general loops of overall negative length may not exist, as the tendency would be for the algorithms to go round the loop indefinitely, to minimise the total length of the path.

In order to calculate the KSPs it is necessary to force the algorithms to choose different routes through the network. This is conveniently done by *marking* the nodes and edges as not being allowed in further path calculations (between the two specified nodes). This operation is often referred to as *removal of a node* and *removal of an edge*, respectively. Removal of an edge e_p from a graph G is simply the operation that results in the subgraph consisting of all the nodes and all the edges in G except the removed edge e_p. Removal of a node i from a graph G is the operation that results in the subgraph consisting of all the nodes and edges in G except node i and those edges incident on node i. Edge marking is implemented by setting edges to have infinite length in some algorithms. This means that they will never be selected to form a path (or if they are, all valid paths have already been found); a path of infinite length is useless.

In many algorithms it is necessary to split a path into two sections in order to calculate other paths; these are known as the *root* and *spur* respectively. The root path is often taken from a section of a previously found [k-]shortest path and a new spur found by calculation.

Depending on the weights assigned to the edges in a network, it is possible that the graph of that network will not obey the rules of *Euclidean space*. Euclidean space requires that one side of a triangle is smaller than (or at the extreme, equal to) the sum of the other two sides. This means that the shortest route is also the most direct route. However the weights applied to the edges are not necessarily just concerned with the distance and therefore may not obey these rules. This is an important point when mentally visualising the expected operation of a KSP algorithm.

The *binary minimum heap* [3] is a convenient method for storing data that is entered in a random order and output in an ordered fashion, and it is used here to store a record of the paths awaiting selection as the next shortest. This heap has at most two (binary) children for each element and the top element (1) has the smallest (minimum) value, and thus here is the next shortest path. It is known that *Fibonacci heaps* [3] provide a more efficient means of data storage, but for simplicity the binary minimum heap has been retained.

4 Assumptions

In order to limit the search for appropriate KSP algorithms, it proved necessary to make some assumptions about the networks that the algorithms would be applied to. Given that our interest was primarily telecommunications networks, the following assumptions were made: networks are undirected, with no parallel edges, no self-loops and no negative edges (although the latter are permitted by some algorithms). Further it was assumed that we are only interested in finding elementary paths in the network.

Algorithms may calculate the KSPs from one node to another node (1–1), from one node to all nodes (1–all) or vice versa (all–1), or from all nodes to all nodes (all–all). In telecommunications we are interested in all of these, but of primary interest is the single node to node calculation and it is this that we shall concentrate on.

5 Literature Survey

A literature survey was conducted to find previous work on KSPs, revealing some 70 papers from many different branches of modern mathematics. Four algorithms [1, 2, 4, 5] from among these were chosen for implementation and speed comparison.

6 Algorithm Selection

The criteria for algorithm selection was based purely on the expected speed of operation of the algorithm in a network meeting the assumptions above. A strong indicator used during the literature survey to provide an insight into algorithm speed, was an algorithm's computational complexity. However, this could only be used as a rough guide, as our interest was in average (rather than worst case) performance on 'real' networks, and actual implementation would thus be the only way to determine the best algorithm.

The two original algorithms, Yen [1] and Lawler [2] were implemented to provide a reference to the expected speed and improvement available. Katoh [4] was included as it represented a comparatively recent update and modification to Yen [1]. The fourth algorithm, Hoffman [5], was implemented after further study, as it was felt that it had the potential to outperform the other algorithms.

Yen [1] is known to have a computational complexity of $O(kn^3)$, where $O(n^2)$ is due to the shortest-path calculation. Lawler [2] improves this by a constant factor, but the computational complexity remains the same. Katoh [4] (approximately) and Hoffman [5] both have a computational complexity of $O(kn^2)$, but use widely different means to derive the KSPs[1].

[1]Katoh himself [4] claims $O(kc(n,m)) \leq O(k \min[n^2, m \log n])$

7 Implementation

The algorithms were coded in 'C' on the department's UNIX workstations. Steps were taken to ensure that the coding of all the algorithms was done as efficiently as possible and such that none of the algorithms were particularly disadvantaged with respect to another. For simplicity, the network was represented as an adjacency matrix; arrays and pointers were used extensively; and particular care was taken in the management of dynamic memory.

7.1 Yen

The shortest path (containing $p \leq n$ nodes) is found using a standard shortest-path algorithm (*e.g.* Dijkstra's [6]) and placed in the results list (Yen's list A). Yen [1] takes every node in the shortest path, except the terminating node and calculates another shortest path (spur) from each selected node to the terminating node. For each such node, the path from the start node to the current node is the root path. Two restrictions are placed on the spur path: 1) It must not pass through any node on the root path (*i.e.* loopless) and 2) It must not branch from the current node on any edge used by a previously found [k-]shortest path with the same root. Node and edge marking is used to prevent the spur paths from looping or simply following the route of a previous [k-]shortest path. If a new spur path is found it is appended to the root path for that node, to form a complete path from start to end node, which is then a candidate for the next KSP. All such paths are stored (Yen's list B) and the shortest remaining unselected path is selected as the next KSP, and transferred to the results list (Yen's list A). The same process is repeated, calculating a spur path from each node in each new KSP, until the required number of KSP have been found.

Various improvements to Yen's original algorithm have been commented on in papers over the years. The most significant improvement can be made with the use of a heap to store Yen's list B, giving an improved performance although the overall computational complexity does not change. A further improvement is checking for non-existent spur paths (*i.e.* where the root path exists, but all spur paths from that root have been used previously).

If list B contains enough shortest paths all of the same next minimum length, to produce all the required paths, it is not necessary to extract each path in turn and perform the above calculations; only to extract the required number of paths and place them directly in list A, as no other shorter paths will be found.

7.2 Lawler

Lawler [2] presents a modification to Yen's algorithm, such that rather than calculating and then discarding any duplicate paths, they are simply never calculated.

The extra paths occur due to the calculation of spur paths from nodes in the root of the KSPs. Lawler points out that it is only necessary to calculate new paths from nodes that were on the spur of the previous KSP. Consider a node on the root of a path; Yen calculates a spur path from this node every time a new KSP is required. The path calculated each time will be the same provided that no extra edge marking has taken place at this node. Extra edge marking will only take place if a KSP has been chosen that branches from this node *i.e.* this node is on the spur. When one of these paths becomes a KSP, then all paths from the root of that KSP will have already been calculated and stored in the heap of prospective KSPs (Yen's list B).

It is therefore necessary to keep a record of the node where each path branched from its parent. This node marks the point from where the calculation of spur paths starts. The increased efficiency of Lawler can be explained therefore as, when finding more than two KSPs, the next KSP will on average branch from the middle of the path and so approximately a 50% improvement in speed is achieved over Yen.

7.3 Katoh

This algorithm [4] is claimed to be faster than Yen due to the way that the previous [k-]shortest path is broken down for the calculation of the next KSP. This path is broken into three sections rather than the ($p-1 \leq n-1$) sections for Yen. Two shortest-path trees are calculated for each section, one from the start node and another to the end node.

The three sections used in the algorithm are derived from the previous KSP by recording (exactly as in Lawler's algorithm) the point that each path branches from its parent. The sections are: 1) all nodes after the branch, 2) the branch node and 3) all nodes before the branch (defined as P_a, P_b and P_c respectively in the paper). The algorithm then calculates one path from each of these sections. Restrictions are placed on this path such that it must deviate from its parent at some point and must not follow other previous KSPs. Each path is calculated by generating two shortest path trees: from the specially calculated start node in each section and from the given end node. The possible shortest paths are represented by paths across these two trees, where each path goes between the trees via a common node or a single edge.

7.4 Hoffman

Hoffman's algorithm [5] requires that the shortest path between the two nodes has been found and that the shortest-path tree from all nodes to the terminal node is known.

In Yen and Lawler, the spur paths are calculated from each node $O(n)$ on the previous [k-]shortest path in turn, using a shortest-path algorithm $O(n^2)$ with node marking, making a total time of $O(n^3)$. In contrast, Hoffman calculates the shortest-path tree from all nodes to the end node $O(n^2)$ at the start of execution. Then, to find the next shortest path, it is only necessary to search from each node $O(n)$ on the previous KSP spur to every other node $O(n)$, making a total time of $O(n^2)$. Each path is made up of three sections: 1) the start node to the selected (*i.e.* branching) node, 2) the edge from the selected node to the new node and 3) the branch of the shortest-path tree from the new node to the terminating node. Edge marking does not need to be used, as edges can be ignored when searching through those from the selected node. The shortest path from each selected node is placed in the heap. The next KSP is then the path with the shortest total length from the start node along the root, via the edge and along the tree to the end node. As the shortest-path tree is known beforehand, it is only a matter of searching at most $n \times n$ edges to determine each KSP.

An important point to note in this path generation, is that the paths evolve by the addition of exactly one edge to the existing root and spur tree paths. This gives rise to a complication, that is automatically excluded by Yen's and Lawler's algorithms, such that it is possible (and quite likely) that looping paths will be generated. These paths are essential to the operation of the algorithm and must be kept like any other previous or prospective KSP. The looping path obtained will follow a previous KSP, but with the addition of a loop making the path slightly longer. The importance is that elementary paths may evolve from the looping section, again by the addition of one edge. The way looping paths are generated is determined by the geometry of the network and whether it obeys the rules of Euclidean geometry. The generation of looping paths adds an extra overhead to this algorithm, which is not easily quantifiable, but will depend on the network features.

A further speed improvement is available with this algorithm, as it is not necessary to calculate the full path details (*i.e.* creating the route) until the path is finally selected as a KSP and removed from the heap. Additionally, the same shortest-path tree to the terminal node can be used for all–1 KSP calculations.

8 Results

In determining the performance of the algorithms, it is useful to summarise their main characteristics. The majority of time taken to calculate the KSPs occurs in the call to a shortest-path algorithm. This algorithm is used not only to find the first KSP (*i.e.* the shortest path), but also several times to calculate each KSP ($k > 1$), except in Hoffman. Katoh generates a maximum of three prospective KSPs from any previous [k-]shortest path, while Yen generates a number of paths according to the path length (*i.e.* the number of hops). Lawler generates some smaller number (approximately 50%) than Yen, depending where the path branched from its parent. For any realistically sized network therefore, Katoh will normally generate significantly fewer paths than either Yen or Lawler. However, in comparison Hoffman only generates one shortest-path tree for all KSP.

The number of paths calculated is not the only concern; the method of calculation and its speed is also important. Both Yen and Lawler calculate the shortest path, while Katoh calculates shortest-path trees. It can be shown that on average a shortest-path tree takes twice as long to calculate as a single shortest path. In order for Katoh to be faster than another algorithm, that algorithm must make at least twelve ($2 \times 2 \times 3$) calls to the shortest-path algorithm for each KSP calculated (although Katoh may not always calculate P_a or P_c). This means that for Yen's algorithm the path lengths must be an average of at least 13 nodes long (12 hops), but for Lawler's algorithm the spur section of the paths must be at least 13 nodes long, and hence on average the path must be about 25 nodes long. For networks to have average path lengths this long, it is necessary for the network to be very large and have a low connectivity. Path hops this long are unlikely to exist in general telecommunications networks and so Katoh does not present a viable algorithm. A typical comparison of path (and tree) calculations is shown in Figure 2.

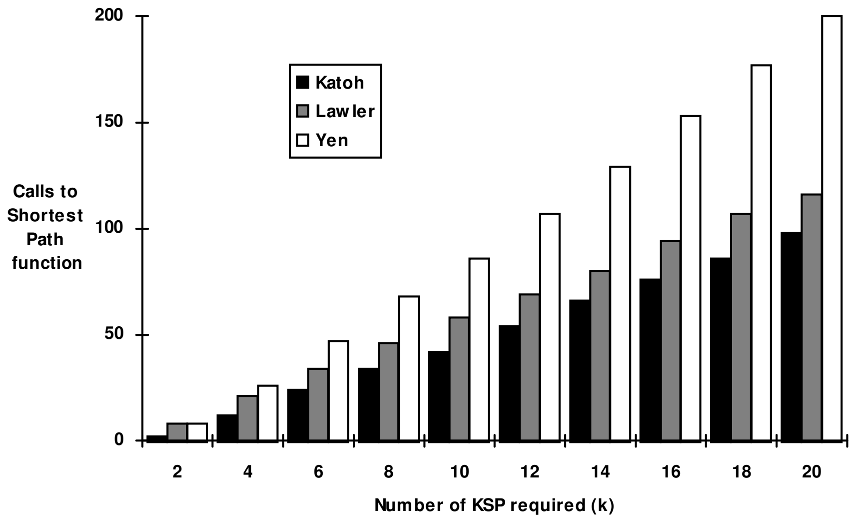

Figure 2: Number of KSP *shortest path* function calls made

However, while this is a better indication of the performance than the computational complexity, it does not demonstrate the required proof of overall best speed. This can only be achieved by comparing the run-times of the algorithms in tests on real networks. The results of one such series of runs on the COST 239 European Optical Network [7] are shown in Figure 3.

This figure demonstrates a staggering improvement in the performance of the algorithms, between Katoh and Hoffman; and yet Katoh is the later, more complicated algorithm. The improvement from Yen to Lawler is clearly indicated.

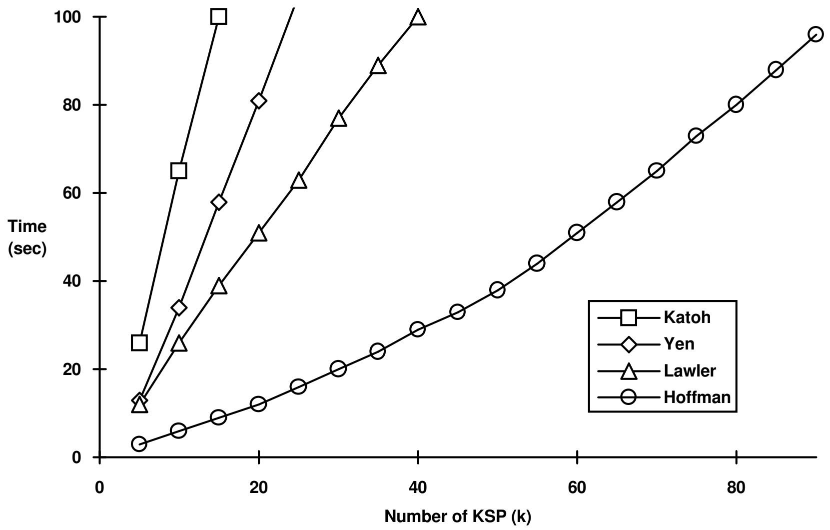

Figure 3: Times for the European Optical Network from London to all nodes

9 Conclusions and Further Work

Lawler's algorithm [2] has been shown to provide the expected speed improvement over Yen's algorithm [1].

However, the best overall speed for the network tested was achieved using Hoffman's algorithm [5]. The results indicate that the performance advantage of Hoffman's algorithm may decay with problem size, but this will require further investigation.

The KSP algorithms studied are very dependent on the efficient implementation of the shortest-path [tree] algorithm and any improvements to this may alter the performance balance.

In general the telecommunications networks of interest to us will be [well] connected. However, the algorithms will also work on unconnected networks, but their efficiency will be improved if the subnetworks are analysed separately.

With algorithms available from so many different branches of mathematics, it is difficult to compare them all without implementing them and as such, the following papers present alternative techniques from other fields that may yet perform faster than Hoffman's algorithm: Shier [8], Carraresi [9], and Boffey [10]. However, these do not all meet the restrictions of loopless and non-disjoint paths, so extra precautions will have to be taken.

10 Acknowledgements

The research that is described in this paper was undertaken by the first author (supervised by the second) as part of an M.Sc. in Telecommunications and Information Systems at the Department of Electronics Systems Engineering, University of Essex. The first author was supported by an EPSRC Advanced Studentship.

References

[1] Yen JY. Finding the k shortest loopless paths in a network. Management Science 1971; 17:712–716

[2] Lawler EL. A procedure for computing the k best solutions to discrete optimisation problems and its application to the shortest path problem. Management Science, Theory Series 1972; 18:401–405

[3] Ahuja RK, Magnanti TL, Orlin JB. Network Flows: Theory, Algorithms, and Applications. Prentice Hall, New Jersey, 1993

[4] Katoh N, Ibaraki T, Mine H. An efficient algorithm for k shortest simple paths. Networks 1982; 12:411–427

[5] Hoffman W, Pavley R. A method for the solution of the nth best path problem. Journal of the Association for Computing Machinery (ACM) 1959; 6:506–514

[6] Dijkstra EW. A note on two problems in connexion with graphs. Numerische Mathematik 1959; 1:269–271

[7] O'Mahony MJ, Sinclair MC, Mikac B. Ultra-high capacity optical transmission network: European research project COST 239. Information, Telecommunications, Automata Journal 1993; 12:33-45

[8] Shier DR. Iterative methods for determining the k shortest paths in a network. Networks 1976; 6:205–229.

[9] Carraresi P, Sodini C. A binary enumeration tree to find k shortest paths. Methods of Operations Research (Germany) 7th Symposium on Operations Research, St. Gallen, Switzerland 1983; 177–188

[10] Boffey B. The all-to-all alternative route problem. Operation Research – Rairo-Recherche Operationelle 1993; 27:375–387