# Transmission Corridor Location: Multi-Path Alternative Generation Using the *K*-Shortest Path Method

F. Antonio Medrano and Richard L. Church

Project 301CR, GeoTrans RP-01-11-01
January 2011

Photos courtesy of DOE/NREL

University of California, Santa Barbara
Department of Geography
1832 Ellison Hall
Santa Barbara, CA 93106
medrano@geog.ucsb.edu
church@geog.ucsb.edu

# Transmission Corridor Location: Multi-Path Alternative Generation Using the *K*-Shortest Path Method

F. Antonio Medrano
Department of Geography
University of California at Santa Barbara
medrano@geog.ucsb.edu

Richard L. Church
Department of Geography
University of California at Santa Barbara
church@geog.ucsb.edu

January 2011

**Abstract**
The corridor location problem for determining good route alternatives for developing new infrastructure often requires generating many viable alternatives which may be later evaluated in public discourse. One method for generating large sets of least cost paths is by solving the k-shortest path problem on a terrain network, but its usage in the literature on finding route alternatives has been limited to routing over sparse road or communication networks. We explored the use of k-shortest path techniques on the denser graph structures of GIS terrain data to examine if they are an efficient approach for corridor location problems. We found that the exponential nature of the k-shortest path problem makes this an overwhelming approach, both in terms of memory and computation time. Future work should focus on finding smart methods of reducing the problem size while retaining optimal solutions, as well as modifying the algorithms to run on high performance supercomputers in order to solve problems of the scale necessary for transmission corridor design. We have promising ideas for such methods, and hope to continue this research in future work.

## Table of Contents


Introduction 4

Choosing a K-Shortest Path Algorithm 6
- *Single Shortest Path* 7
- *Directed and Undirected Networks* 9
- *Simple Paths vs. Paths With Loops* 10
- *Networks with Negative Arcs* 11
- *K-Shortest Path Lengths* 12
- *Near-Shortest Paths* 12

K-Shortest Path Algorithms: Literature Review 13

Detailed Descriptions of Optimal Loopless KSP Algorithms 16
- *Hoffman and Pavley's Algorithm* 16
- *Yen's Algorithm / Lawler's Improvement* 17
- *Katoh's Algorithm* 18
- *Carlyle's Near-Shortest Path Algorithm* 19
- *Comparison of the Algorithms* 22
- *Summary* 24

Implementation 25
- *Memory Performance* 25
- *Time Performance* 28
- *Conclusions* 29

Conclusions and Future Work 30
- *High Performance Computing for KSP/NSP* 30
- *Explore p-Dispersion* 31
- *Gateway Paths / Hybrid NSP-Gateway Approach* 31

Acknowledgements 33

Bibliography 34

## Introduction

It is commonly acknowledged that the existing electrical grid in the U.S. needs to be expanded to meet the needs of growth and change, growth in terms of electrical demand and change associated with using new sources of renewable energy generation like wind and solar. For example, tapping wind resources in remote locations will require new transmission capacity to deliver it to load centers. One of the biggest challenges faced by the electrical power industry in the US is associated with increasing capacity and interties of the existing electrical grid. The crux of the matter is that it is difficult to locate new transmission corridors that are efficient, keep environmental impacts low, and are politically acceptable. This problem is contentious at best and what many would define as a wicked public problem. The basis for such a difficult planning problem is that no one wants one per se close to where they live and no one wants them where they obstruct views on the landscape. Simply put, most people want transmission corridors as far away from them as possible, but at the same time rely on them every day for safe-reliable delivery of electricity.

A few years ago at a meeting in Jackson, Mississippi sponsored by the USDOT that dealt with the issues of a possible realignment of the CSX railroad, Claiborne Barnwell described a nagging problem associated with making new road alignments. In essence, he described a scenario where a plan had been developed and was then being presented to an open public forum. In that forum, he thought it entirely possible that some locally observant citizen would stand up and ask why the route for the alignment didn't "go over there" as the citizen broadly traces a route on the map. Mr. Barnwell stated that if such an alignment hadn't been studied, then he would be hard pressed to answer the question. Technically, he said, he would be forced to go back to the drawing board to study that as well. What he seemed to suggest is that all alignments need to be studied, so that an answer is quick and the citizen can be satisfied. But how can this be done to the satisfaction of everyone at hand? To understand this in perspective, let us first describe the process of corridor location planning.

There are two principal phases in corridor location planning. The first phase involves the determination of the general route or alignment for the corridor. The second phase deals with the engineering design and route refinement. The biggest hurdle in corridor location is to successfully finish the first phase with all agencies giving their approval. A corridor location model integrated within a Geographical Information System (GIS) typically supports that phase. A GIS allows data from different sources and of different types to be merged into a composite map that represents the combined impacts and costs with a corridor traversing the landscape. This planning approach harks back to Ian McHarg (1969) and *Design with Nature*. McHarg described an approach in which a composite cost map can be produced, where high impact/cost areas are colored dark and low impact/cost areas are very light in color. He then suggested taking this composite map and tracing a route that avoids the darker areas and uses the lighter areas as much as possible. Today, this approach is accomplished in a more automated fashion, from developing the composite cost map to using an algorithm to find the path of least impact/cost from the origin to the destination. The result of this approach is the least cost/impact path across the landscape. All other paths (assuming importance weights of various objectives are kept constant) will have a higher impact/cost. In essence, Claiborne Barnwell's concern is addressed in that he could say to the citizen who had asked the question: By the fact that the routes

generated by the computer algorithm are optimal means that the particular route in question is not as good as what we have determined. Thus, end of story, or is it? You see, Barnwell wouldn't be able to say how much better the proposed route is compared to what the citizen was suggesting. In fact, the proposed route impact might not be very different from the citizen's route based upon the data and impact and cost functions. Furthermore, there may be errors in the data that if fixed would favor the citizen's route over the proposed route. Finally, it may take only slight changes in importance weights for the computerized algorithm to select an alignment that differs from what was originally proposed as well as differs from what was suggested by the citizen. Perhaps the dilemma of Barnwell is now better understood. The best solution may not be that much better than other good alternatives, and it is desirable to understand exactly where good, competitive alignments exist. That is, one should search for competitive, near optimal alignments that differ from the optimal alignment.

Corridor location planning tools have not been designed to generate significantly different alternatives, but rely instead on a straightforward application of the single shortest path algorithm (Turner 1968, Potts 1975, Owens 1975, Newkirk 1976, Smart 1976). Barnwell's thesis is that to be successful and move to phase 2, one needs to take the task seriously in phase 1 and address all alternatives either explicitly or implicitly, as well as be ready to state how much better one is over the other in a public setting. Addressing this requires new, improved corridor-planning tools, based upon a model that is capable of performing a comprehensive spatial search in alternatives. One of the possible approaches for searching a large set of paths is the *k*-shortest paths algorithm. The *k*-shortest path algorithm explicitly searches for the shortest path, the second shortest path, the third shortest path, and so on. It has been widely studied and numerous algorithms have been developed for solving this problem on a network. The basic premise in this particular report is that the *k*-shortest path algorithm may be of value in searching for alternate configurations of a corridor.

This report is organized as follows. First, we provide a general introduction to solving the shortest path problem. Then we delve deeper into specific nuances of shortest path algorithms and k-shortest path algorithms in specific, with the purpose of selecting the best candidate for application in corridor alternatives generation. We then discuss issues of implementation and present several algorithmic tests that were performed as a part of this work. This is followed by a short discussion of alternatives to the *k*-shortest path algorithm in searching for route alternatives. Finally, we present a plan in which the *k*-shortest path algorithm can be harnessed to search for corridor alternatives.

## Choosing a *K*-Shortest Path Algorithm

The *k*-shortest path (KSP) problem is a computational problem that aims to determine not only the shortest path on a network from a given origin to a destination, but also the second shortest, third shortest, fourth shortest, and so on. While the general problem is relatively simple to comprehend, there are many nuances that must be considered before choosing which of the numerous available algorithms to use in-order to solve for a specific application.

Since the early 50's, the shortest path and *k*-shortest path problem have been of interest to computer scientists and network engineers, and an extensive literature exists on these topics. This report summarizes the results of a literature review on this topic for the purpose of determining which algorithm might be best suited for a corridor routing search process. More general reviews can be found for the single shortest path problem in Dreyfus (1969), Cherkassky et al. (1996), and Zeng and Church (2009). For the k-shortest path problem, reviews can be found in Dreyfus (1969), Eppstein (1998), Martins et al. (1998) and Hershberger et al. (2007b), as well as Eppstein's exhaustive online bibliography of *k* shortest path papers up to the year 2001 (Eppstein 2001, http://www.ics.uci.edu/~eppstein/bibs/kpath.bib).

Before choosing a *k*-shortest path algorithm, one has to consider numerous aspects of the problem that can affect whether a specific algorithm can be used or whether another might be more appropriate to use. The following sections will describe the points one must consider, and suggest an appropriate algorithms for each instance. Additionally, numerous *k*-shortest path algorithms call a single shortest path algorithm numerous times as part of their implementation, so we will also include a brief review of single shortest path algorithms as well.

### *Efficiency: Worst-Case Complexity vs. Experimental Results*

The computer science literature tends to analyze the efficiency of algorithms in terms of worst-case complexity. This represents the number of operations it takes for an algorithm to run in the worst possible case of a problem instance of a given size. Typically, it is represented in big "O" notation, which describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions.

The formal definition of big O notation is $f(x) = O(g(x))$ as $x \to \infty$, if and only if, for sufficiently large values of $x$, $f(x)$ is at most a constant multiplied by $g(x)$ in absolute value. For example, suppose you have a sorting algorithm which sorts an array of numbers of length $x$. Let's say that to sort this array given a worst-case problem instance, it takes $6x^4 - 2x^3 + 5$ comparison operations. In big "O" notation, this means that for extremely large arrays, the algorithm is dominated by the 4th order polynomial term, hence its efficiency is $O(n^4)$.

The efficiencies of shortest path algorithms depend on the number of nodes $n$ and the number of edges/arcs $m$ in the network. Just about all networks contain more arcs than nodes, but a sparse network will contain fewer arcs as compared to a dense network. For road networks, the arc to node ratio ($m/n$) is typically in the range of 2.6−3.3. This is considered a sparse network.

Terrain is typically modeled as a raster network, with each pixel of the topological terrain image representing a node on the network. Neighboring nodes are then connected with arcs where each arc is given a cost that represents a weighted sum of impacts and costs associated with routing a corridor at that location. For example, the ArcGIS system assumes a network node at the center of each raster cell. Then arcs connect each node to its eight nearest neighboring cells (nodes) in the four orthogonal directions and the four diagonal directions, which results in a more dense network of $m/n = 8$. Goodchild (1977) and Huber and Church (1985) showed that significant geometric errors might arise if you don't also include knight's moves on a raster representation. While error is reduced, it also doubles the arc-to-node ratio to a more dense value of 16. In the extreme case, every node in a network may be connected to ever other node. In this instance, maximum density is achieved with $m = n^2$ arcs, so the density is $m/n = n$.

For many algorithms, worst-case complexity can be computed as a precise mathematical function, which is useful to know how fast something will run in the absolute worst case. But, worst-case scenarios are often unrealistic. For example, the worst case for most shortest path algorithms occurs in a contrived network where the shortest path must traverse through all nodes to reach the destination. Clearly, this would never occur in a real-world situation. This has led some investigators to include computational tests in which a comparison of an algorithm's performance on random networks and real-world networks is made. While this doesn't give a definitive boundary for how fast an algorithm runs in all instances, as results will vary depending on the network used, it can show that sometimes algorithms with slower theoretical worst-case efficiencies can sometimes run faster in real-world applications. It is somewhat common in the literature to find papers that make refinements to an existing *k*-shortest path algorithm that results in the same worst-case complexity, but speeds up the real-world runtime by some amount (Azevedo *et al.* 1994, Jiménez and Marzal 2003, Martins and Pascoal 2003).

*Single Shortest Path*

Many *k*-shortest path algorithms have as a subroutine a single shortest path function to find the shortest path after modifying the network in order to prevent a previously found path from being identified again. Therefore, when implementing an efficient k-shortest path routine, one must also ensure that the single shortest path routine is optimal for the particular type of network one is analyzing.

The most widely used algorithm for the identifying shortest path problem starting at an origin and reaching the destination, for both its simplicity and efficiency, is Dijkstra's algorithm (Dijkstra 1959). The original naive implementation of the algorithm has a complexity of $O(n^2)$, but implementations using modern data structures such as Fibonacci heaps, double buckets, and approximate buckets have lowered the complexity to $O(m + n \log n)$ and have significantly reduced experimental runtimes (Cherkassky *et al.* 1996, Zhan and Noon 1998, Zeng and Church 2009). As a side note, the terms "shortest path" and "least cost path" can be used interchangeably to represent the path of least penalty. Nomenclature depends on what costs are represented in the network (distance vs. some other penalty value). It is important to note that Dijkstra's algorithm can only be used in networks that do not contain negative arc costs. For networks with negative arcs (but no negative cycles), the less efficient Bellman-Ford Algorithm most often used.

Simply put, Dijkstra's algorithm is a label setting algorithm that uses best-first-search to expand a shortest path tree from the origin node. The algorithm completes when the tree reaches the destination node. At that point, the shortest path can be traced back along the tree from the destination back to the origin.

In more detail, Dijkstra's algorithm works as follows. Let $G(V, E)$ represent a directed network with $n=|V|$ nodes and $m = |E|$ arcs. Each arc is represented as $(i, j)$, where $i$ is the origin node and $j$ is the destination node. Each arc has a cost associated with traversing it, denoted as $c(i, j)$. The algorithm creates a directed tree, $DT$, rooted at the starting node $s$, and at each iteration it adds an arc and node to the tree. The algorithm stops when the tree reaches the destination node $t$. The shortest path from the origin to any specific node on the tree is unique and can easily be traced from that node by backtracking along the tree, branch by branch, until reaching the root or starting node.

For each node $v \in V$, let $d(v)$ be total cost of the path from $s$ to $v$ in $DT$. Initially, all $d(v)$ values should be set to infinity. Each node is labeled with one of three labels: unlabeled ($U$), temporary ($T$), and permanent ($P$). All nodes begin as unlabeled. A node becomes permanently labeled when it becomes part of the least cost path tree $DT$. When that happens, it stores the previous node in the tree as $prev(v)$. The least cost path from any node to the root of the tree can be back-tracked by following the chain of $prev(v)$ until reaching node $s$. The algorithm runs as follows:

**Step 1:** Set $s$ as permanently labeled, and set $d(s) = 0$, and $prev(s) = s$. Set all nodes $j$ such that $(s, j) \in E$ to be temporary labeled ($T$) and set $prev(j) = s$ and $d(j) = c(s, j)$.

**Step 2:** Scanning all nodes $j$ labeled $T$, let node $v = (minimum(d(j)) \mid j \in T)$. Set the label of $v$ to permanent ($P$). If $v = t$, the shortest path to the destination has been found, and conclude the program.

**Step 3:** Find all arcs $(v, j) \in E$ where $v$ is the newly labeled $P$ node. If $j$ is labeled $P$, then eliminate the arc by removing it from $E$. If $j$ is labeled $T$, then compare the node's current distance label, $d(j)$, to the distance associated with the new arc, $d(v) + c(v, j)$. If $d(v) + c(v, j) < d(j)$, then set $d(j) = d(v) + c(v, j)$, delete arc $(prev(j), j)$ from $E$ and set $prev(j) = v$. Otherwise, delete arc $(v, j)$ from $A$. If node $j$ is unlabeled ($U$), then set $j$ as labeled $T$, set $prev(j) = v$, and set $d(j) = d(v) + c(v, j)$. Return to step 2.

The general character of Dijkstra's algorithm is that it creates a shortest path tree that spreads out blindly in all directions until it reaches the ending node $t$. If the network in question has a spatial definition such that the location of the nodes corresponds to a physical position (which is the case with GIS-sourced data), then one can take advantage of this information to steer the shortest path tree toward the destination node $t$ and greatly reduce the number of operations required to find the shortest path without loss of optimality. This is the basis of the A* algorithm (Hart *et al.* 1968) which can be viewed as a generalized version of Dijkstra's algorithm. In each iteration, Dijkstra's algorithm scans the set of $T$ (the nodes with temporary labels) and picks that node $j$ that has the lowest distance label (i.e. $d(j) = d(i) + c(i, j)$). The A* algorithm differs in that it picks that node $j$ in T with the lowest label value of $d(j)$ where $d(j) = d(i) + c(i, j) + h(j)$. When $h(j) = 0$, Dijkstra's algorithm and A* are exactly the same. For A*, the value of $h(j)$ represents an estimate to complete the path to the destination. So long as the $h(j)$ function is always equal to or

less than the actual path distance from $j$ to $t$, then the A* algorithm will guarantee an optimal solution. Typically, Euclidian distance is used for $h(j)$, since a straight line distance is always less than or equal to the distance of any path from a node to the destination.

Adding a "spatial guide" element to path routing, like A*, can greatly reduce the number of temporary nodes that an algorithm looks at before reaching the destination (Golden and Ball 1978). As part of a $k$-shortest path routine, Skiscim and Golden (1987, 1989) showed that on Euclidian networks, using A* can reduce the number of nodes the algorithm must consider by 99%, and dramatically improve runtimes. Sedgewick and Vittel (1986) showed that the A* algorithm on Euclidian graphs can find a shortest path with worst case complexity of $O(n)$. More recent published experimental results have shown A* to be the fastest shortest path algorithm on real road networks (Zeng and Church 2009).

Zhan and Noon (1998) note in their paper that if one is searching for the shortest path from a single origin to all other nodes in a network instead of the shortest route between two specific nodes, label correcting incremental graph algorithms by Pape-Levit and Pallotino perform extremely well in experiments using real road networks. This fact is somewhat surprising given that the theoretical worst-case complexity of either algorithm is worse than Dijkstra. A recently published algorithm based on Pallotino's TWO-Q method called MiLD-TWO-Q has shown some promise in producing very fast one-to-one shortest paths, but no research has yet been documented comparing it experimentally with A* (Leng and Zeng 2009). In both papers, no mention was made of the performance of these algorithms when implemented on raster networks.

*Directed and Undirected Networks*

Networks or graphs upon which one might apply a shortest path algorithm may be either directed or undirected. A directed graph will contain arcs that are directional (directed from one node to another), and may represent different penalty values depending on the direction of travel. Figure 1(a) shows an example of a directed graph $G_d = (N_d, A_d)$, containing nodes $N_d$ and arcs $A_d$. Note that in this example, it is possible to move directly from $a$ to $c$ and from $c$ to $a$, but that the cost in moving between nodes $a$ and $c$ is different based upon direction: *i.e.* $cost(a,c) = 3$, while $cost(c,a) = 1$, thus $cost(a,c) \neq cost(c,a)$. Another example of directionality in graph (a) is that while arc$(b,a) \in A_d$, the reverse direction arc$(a,b) \notin A_d$.

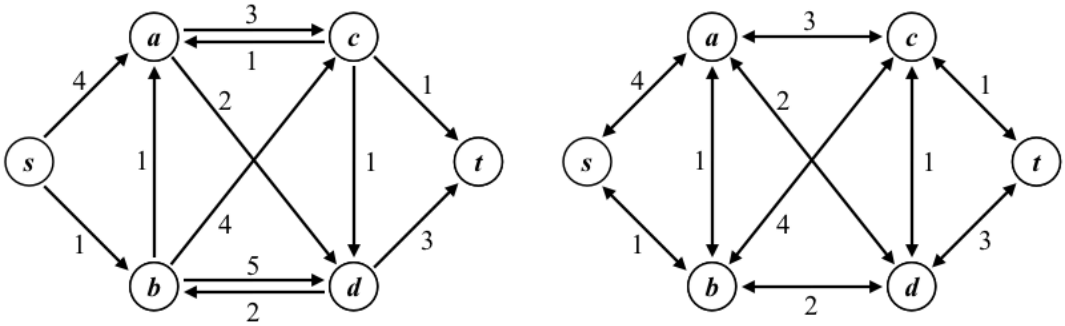


Figure 1 - An illustration of a directed graph (a), and an undirected graph (b)

Figure 1 (b) depicts an undirected graph $G_u = (N_u, A_u)$, containing nodes $N_u$ and arcs $A_u$. Unlike the directed graph, the undirected graph is symmetric. The cost to travel on any arc is the same in both directions. Or in other words, for all arcs $(x,y) \in A_u$, $cost(x,y) = cost(y,x)$. This also means that the shortest path from $s$ to $t$ has the same length as, and coincides with, the shortest path from $t$ to $s$.

Some path algorithms are designed to work on both type of networks, directed and undirected, whereas others are restricted to one form. Dijkstra's algorithm and A* both give correct results on both kinds of graphs, but as we see in the next section, some $k$-shortest path algorithms may work on only one or the other. For a corridor location problem, it is safe to assume that GIS generated networks are undirected, where the impacts from the construction of power lines or the efficiency of power transmission will not be affected by traveling one way or the opposite between two nodes.

*Simple Paths vs. Paths With Loops*

Algorithms for finding $k$-shortest paths can be divided into two principal sets: those that allow for loops in the paths, and those that do not. Paths that do not allow loops are called simple, and are defined by the rule that they do not allow for the repetition of any nodes or arcs. The directed network in Figure 2 illustrates the differences in finding the $k$-shortest paths when allowing paths with loops as opposed to only simple paths. One can see that the three simple shortest paths have lengths 6, 20, and 21, respectively. Without the simplicity constraint, paths may use the cycles ($a$, $b$, $a$) and ($d$, $e$, $d$). For this case the three shortest paths that involve cycles have lengths 6, 8, 10.

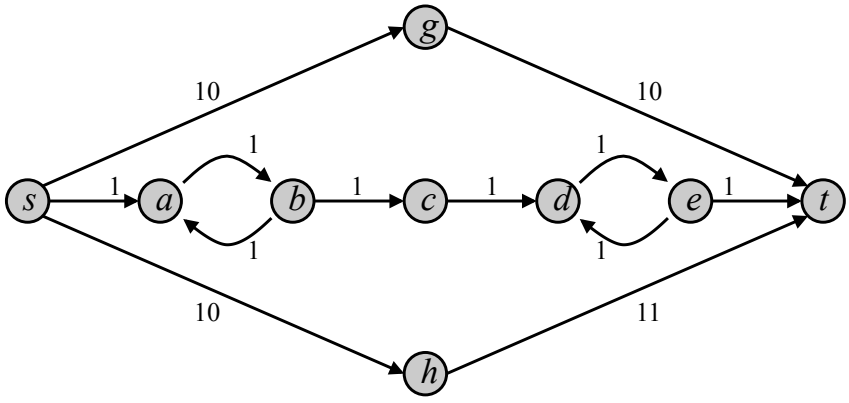


Figure 2 - Network for illustrating $k$-shortest paths with and without loops

Fox (1975) developed an early algorithm that found the $k$-shortest paths between an origin and destination allowing loops in $O(m + kn \log n)$ time. Eppstein (1998) improved upon that using an algorithm that creates a heap data structure of shortest paths, from which each path could be output in $O(n)$ time. This resulted in achieving a theoretical worst-case complexity of $O(m + n \log n + k)$ for this problem. Jiménez and Marzal (2003) later modified Eppstein's algorithm in a way that retained the same worst-case running time, but resulted in major improvements in practical performance.

Very few real-world applications have a use for paths with loops, since they just create redundancy in a network and do not add to the robustness of a path. In the case of locating a corridor for a power line, including loops in a path would not provide any benefits, yet they would add to the overall cost of the network. For this reason, it does not make sense to use Eppstein's algorithm for corridor location. Rather, it is better to look at algorithms that generate only the *k*-shortest simple paths.

On directed graphs, the fastest *k*-shortest simple paths algorithm is Yen's Algorithm (Yen 1971), which runs in $O(kn\ c(n,m))$ time, where $c(n,m)$ is the cost of running Dijkstra on the network. Yen's algorithm works on undirected graphs as well, but Katoh, Ibaraki et al.(1982) later presented an algorithm specifically for undirected graphs which improved the complexity to $O(k\ c(n,m))$. We discuss both of these algorithms in greater detail later in this report.

*Networks with Negative Arcs*

There are instances when a network may contain negative arcs. This situation would arise if there were a negative penalty (positive incentive) to traverse through some section of the network. Many shortest path and *k*-shortest path algorithms still work in this case, but only in certain conditions. If a network contains any negative loop, then the shortest path problem is impossible to solve for paths with loops. This is because a negative loop allows for any arbitrary path length to be created, thus a ranking of paths cannot be made. Even for simple paths, there does not exist an algorithm for finding a shortest path in the presence of negative cycles (Yen 1971, Fakcharoenphol and Rao 2006).

Even when a negative cost cycle does not exist, some KSP algorithms fail in the presence of negative cost arcs. As mentioned earlier, *k*-shortest path algorithms often use a single-shortest-path algorithm as a subroutine. Typically, Dijkstra's is used for this purpose, but Dijkstra's algorithm fails on graphs with any negative cost arcs. Although the Bellman-Ford algorithm can run on networks with negative arcs (but not negative cost cycles), it runs at a less efficient complexity of $O(mn)$. Implemented as a part of Yen's algorithm, this results in an overall KSP algorithm of complexity $O(kmn^2)$, as compared to $O(kn\ (m + n \log n))$ for positive networks using Dijkstra with Fibonacci heaps.

In the real-world case of power-line corridor location, building a power line system through some region will never cost less than building nothing at all, therefore it is safe to assume that these networks will never have negative arcs. That being the case, we will then assume that some form of the Dijkstra algorithm will be used for finding single shortest paths. However, it is important to note that corridor design models may involve conflicting objectives, some that involve maximization (e.g. maximizing compatibility), and others associated with minimization (e.g. minimizing cost) and depending upon a set of importance weights used, may result in arcs which have a negative composite cost value. This means that it may be necessary in some circumstances to restrict the definition of arc costs so that negative composite costs do not occur.

*K-Shortest Path Lengths*

There may be a situation where one desires to know the *k*-shortest path lengths instead of the *k*-shortest paths. The *k*-shortest path lengths problem overlooks the fact that there may be multiple paths of the same length. Therefore this approach may omit numerous alternate paths that would be found in a conventional *k*-shortest paths algorithm. A number of algorithms for finding the *k*-shortest path lengths can be found in a paper by Shier (1979).

In his paper, Shier performs computational experiments with a number of proposed algorithms, and concludes that for dense networks ($m/n \geq 10$), his label-setting double sweep algorithm is the fastest. A later publication (Guerriero *et al.* 2001), which used modern data structures for label-setting and label-correcting algorithms, found that for all instances except fully dense graphs (*i.e.* all nodes connected to all other nodes), that the double sweep was the slowest algorithm for finding the *k*-shortest path lengths. Guerrero's article does not cite a paper published just before his, (Rink *et al.* 2000), in which an improvement on the complexity of Shier's algorithms is shown to reduce the worst case complexity from $O(n^3k^3)$ to $O(n^3k^2)$. Rink's article does not report any computational experiments though, so it is unknown if their improvement would result in any real-world differences in runtime.

The results of Guerrero and Rink show that in almost all cases, the older double-sweep algorithm is far slower than other *k*-shortest path length algorithms, and that the newer double-sweep still has a worse complexity than the best *k*-shortest path algorithms. Since then, the literature is almost non-existent of new developments on algorithms for finding the *k*-shortest path lengths. We propose that this may be due to little practical interest in the problem of shortest path *lengths*, as well as the fact that faster *k*-shortest path algorithms can be used to find simply the shortest path lengths. Based on these worst-case complexity and experimental results, we did not consider KSP*l* algorithms for use in multi-path corridor location.

*Near-Shortest Paths*

Rather than searching for a set of *k* paths ranked in order of length, another way of generating all paths that are close in length to the shortest path is to solve the Near Shortest Path problem. Near shortest paths are defined as paths whose length are within a factor of $1 + \varepsilon$ of the shortest path length for some user-specified $\varepsilon \geq 0$. This generates an unknown number of paths that are of a length within some threshold of the shortest path. The first to formulate this problem were Byers and Waterman (1984), who developed an algorithm to solve this problem for paths with loops to investigate evolutionary relationships between two DNA sequences.

Carlyle and Wood (2005) modified the Byers and Waterman algorithm to constrain the results to only loopless near-shortest paths. Their algorithm could then be used along with a binary search tree to solve the *k*-shortest path problem with a worst case complexity of $O(kn\, c(n,m)\, (\log n + \log c_{max}))$, where $c(n,m)$ is the cost of running Dijkstra and $c_{max}$ is the largest edge length. Carlyle and Wood compared runtimes of their KSP algorithm to that of Katoh's algorithm as implemented by Hadjiconstantinou and Christofides (1999), and declared theirs to be far superior despite using different networks and faster computers for their study. No third party experiments

have been published comparing Carlyle and Wood's algorithm to other *k*-shortest path algorithms.

## K-Shortest Path Algorithms: a review of the literature

The first algorithm published for solving the *k*-shortest loopless path problem was by Bock, Kantner, and Hayes (1957), and essentially worked by systematically listing all possible routes between a given origin and destination, then ranking them in order of length. This brute-force method increases runtime and memory requirements factorially as the graph increases in size, so is limited in all practicality to very small networks.

Most *k*-shortest path algorithms incorporate some form of a single shortest path routine or a shortest path tree routine in order to find their paths, thus another foundation of the KSP literature is Dijkstra's publication of his algorithm for finding the single shortest path (Dijkstra 1959). While other shortest path algorithms did exist before (Ford 1956, Dantzig 1957, Minty 1957, Bellman 1958, Moore 1959), Dijkstra's label setting algorithm immediately became the standard one-to-one and one-to-all shortest path algorithm upon its publication due to its ease of understanding and implementation, its applicability to all sorts of networks, and its computational efficiency. Over the years, Dijkstra's algorithm has been receptive to improvements in efficiency through the use of advanced data structures (Dial 1969, Fredman and Tarjan 1987, Ahuja *et al.* 1990, Corman *et al.* 1990, Cherkassky *et al.* 1996) and spatial heuristics (Hart *et al.* 1968), thus maintaining its high stature even in the face of later developments in shortest path algorithms (Pape 1974, Glover *et al.* 1984, Pallottino 1984, Glover *et al.* 1985, Ahuja *et al.* 1990, Goldberg and Radzik 1993).

The first practical *k*-shortest path algorithm for problems of any size was presented by Hoffman and Pavley (1959), and is relevant for numerous reasons. It is noteworthy as being the first algorithm to realize that successively longer routes are the result of deviations from a shorter route (Pollack 1961a). This principle is the foundation of subsequent more advanced algorithms by Yen, Katoh, and others. The algorithm is also vastly overlooked and misunderstood in the literature, for reasons that will be explained later in more detail in the algorithms section.

Pollack (1961b) published an algorithm which finds the $k^{th}$ shortest path by eliminating all combinations of one arc from each prior $k - 1$ shortest paths and running a shortest path algorithm for each scenario. This results in a running time of $O(n^k)$. For small values of $k$, this method is reasonably fast and easy to implement, but it quickly becomes impractical for large values of $k$.

The first algorithm for *k*-shortest paths with loops was developed in the late 60's by Dreyfus (1969) with complexity of $O(kn^2)$. Being a less constrained problem than finding simple paths, there was great interest in finding even faster algorithms for KSP with loops. The next advancement was published by Fox (1975), in which he presented an algorithm based on Dijkstra's algorithm. Modern day implementations using priority queue data structures make this algorithm run with a complexity of $O(m + kn \log n)$. The present day best-known algorithm in terms of worst-case complexity was introduced by Eppstein (1998). His algorithm was shown to

be the theoretically best possible worst-case complexity of $O(m + n \log n + k)$, solving the problem by generating a tree of shortest paths stored in a heap data structure, from which the *k*-shortest paths can be extracted very efficiently. It has been shown that this worst-case complexity cannot be improved upon, yet development of algorithms for finding looped paths has remained an active field, with numerous algorithms claiming to be computationally faster than Eppstein's in certain applications (Martins 1984, Azevedo *et al.* 1993, Azevedo *et al.* 1994, Martins *et al.* 1998, Jiménez and Marzal 1999, Martins and Esteves dos Santos 1999, Martins *et al.* 1999, Martins and Pascoal 2000, Jiménez and Marzal 2003).

For single shortest path algorithms, the late 60's brought new developments that greatly improved the efficiency of Dijkstra's algorithm. For spatial graphs, Hart et al. (1968) published the aforementioned A* algorithm. Shortly thereafter, Dial (1969) presented a new method of implementing Dijkstra's algorithm using a "bucket" data structure for storing values of temporarily labeled arcs. This structure reduces the number of nodes scanned during each iteration, dramatically improving runtimes. While the worst case complexity of Dijkstra's naive algorithm is $O(n^2)$, the bucket implementation is $O(m + nC)$, where $C$ is the maximum arc length in a network. Cherkassky et al. (1996) would later extend Dial's buckets idea to more complicated bucket data structures such as buckets with an overflow bag, approximate buckets, and double buckets, which further improved runtimes.

The next major *k*-shortest path algorithm came when Yen (1971) presented his algorithm. Unlike Hoffman's algorithm with uncertain computational bounds, the efficiency of Yen's algorithm could easily be determined for all graphs. Like Hoffman's algorithm, Yen's algorithm generates *k*-shortest paths as deviations from a shorter path. Once the shortest path has been found, the algorithm calculates a shortest path from each node in the shortest path to the destination, eliminating certain arcs so as to not repeat a previously found shortest path. All found paths are stored in a list, and the shortest of the found paths is the next shortest path. The process is then repeated on all shortest paths until $k$ paths have been found. Using a naive shortest path algorithm, Yen's algorithm was found to be of complexity $O(kn^3)$. Later implementations using advanced data structures reduced the complexity to $O(kn\,(m + n \log n))$ using Fibonacci heaps.

Shortly after Yen published his method, Eugene Lawler presented a modification to Yen's algorithm that cut the number of computations in half by defining criteria that prevent the redundant computation of shortest paths (Lawler 1972). While this does not affect the worst-case complexity, his modified algorithm is now considered the standard efficient implementation for Yen's Algorithm. Further refinements on Yen's algorithm have been presented by Perko (1986), and by Martins and Pascoal (2003).

Shier's algorithm for finding the *k*-shortest path *lengths* was also developed around the same time (Shier 1974, Shier 1976). His experiments showed though that his algorithm was faster than Yen's for only dense networks of density $m/n \geq 10$ (Shier 1979). With later advancements in implementations of Yen's algorithm, Guerriero, et al. (2001) showed that Shier's algorithm was slower than Yen's algorithm for all but complete graphs ($m/n = n$). Rink, et al. (2000) published an improvement on Shier's algorithm which has been shown to reduce the worst case complexity from $O(n^3k^3)$ to $O(n^3k^2)$. The article does not report any computational experiments though, so it is unknown if their improvement would result in any measurable differences in runtime.

The next major development in computing *k*-shortest paths came when Katoh, et al. (1982) published an algorithm that applies only to undirected networks, solving with a worst case complexity of $O(k\,c(n,m))$, where $c(n,m)$ is the cost of running Dijkstra's shortest path algorithm. Like Yen's algorithm, it also finds the next shortest path as a deviation of a previous shortest path. But, whereas Yen's algorithm runs a shortest path routine up to $n$ times to find the next shortest path, once for each node on the previous path; the Katoh et al. algorithm runs a shortest path routine exactly 3 times each time one seeks the next shortest path. This is equivalent to a theoretical $O(n)$ improvement over Yen's algorithm. A paper by Hadjiconstantinou and Christofides (1999) gives the most complete explanation of how to implement the Katoh et al. algorithm, including modifications for improvement in efficiency, some pseudocode, and suggested data structures.

Fredman and Tarjan (1987) published a paper that introduced a new data structure for storing variables that they called Fibonacci heaps (named after the Fibonacci numbers used to prove the running time analysis). This structure allowed the worst case complexity of Dijkstra's algorithm to be reduced from $O(n^2)$ to $O(m + n \log n)$. It is also independent of the arc cost values (unlike buckets), therefore computer scientists tend to use this when analyzing new algorithms that use Dijkstra as a subroutine. Except for contrived worst-case graph scenarios, Dijkstra's algorithm with Fibonacci heaps has been found to be slower than most other implementations (Cherkassky *et al.* 1996, Zhan and Noon 1998). Because of this, some researchers use the term $c(n,m)$ to denote the runtime-cost of running Dijkstra when they evaluate the complexity of an algorithm which uses Dijkstra as a subrouting, allowing the user to determine which complexity for Dijkstra they will use.

A new way of looking at the *k*-shortest simple path problem was presented by Carlyle and Wood (2005) in their paper about the Near Shortest Path problem. Based on an earlier paper by Byers and Waterman(1984) which defined the Near Shortest Path Problem for paths with loops, Carlyle and Wood modified that algorithm to find only simple paths. Instead of looking for $k$ paths in order of length, the near shortest path algorithm finds all paths within a factor of $(1 + \varepsilon)$ of the length of the shortest path, where the user defines some $\varepsilon \geq 0$. Without the restriction of having to find paths in order of length, they use a depth-first search routine to enumerate their paths very quickly. Their algorithm calls Dijkstra only once, similar to the Hoffman and Pavley algorithm.

The most recent major development in the *k*-shortest path problem is a paper by Hershberger, et al. (2007a), which presents a new algorithm for finding the *k*-shortest simple paths on a directed network in $O(k\,c(n,m))$ time. This is the same complexity as the Katoh et al. algorithm, which works only on undirected graphs, and is an $O(n)$ improvement over Yen's algorithm. Based on the Katoh et al. algorithm, it uses a "replacement path algorithm" to find alternate routes when links are removed from the graph. The method they publish though is prone to failure in rare instances, thus it cannot be considered a "correct" algorithm. It can detect instances of failure though, and switch over to a slower correct algorithm (such as Yen's algorithm) when that occurs. For more information on the replacement paths problem, please refer to Hershberger, et al. (2007b).

## Detailed Descriptions of Optimal Loopless KSP Algorithms

*Hoffman and Pavley's Algorithm*

The complexity of Hoffman and Pavley's algorithm (Hoffman and Pavley 1959) is difficult to specify, yet in practice it appears that it can be a very efficient algorithm (Brander and Sinclair 1995). It only requires a single instance of running a shortest path algorithm to generate the shortest path tree. From that point on, all subsequent paths are calculated by scanning all nodes on a found shortest path, generating all 1-arc deviations from each node on that path, then following the shortest path tree the rest of the way back to the origin. Each path then consists of a concatenation of a portion of a previous shortest path, a deviation arc, and a portion of the shortest path tree.

What makes it hard to predict the number of operations this algorithm requires to run is that in the process of calculating the $k$-shortest simple paths, it will also generate some unknown number of paths with loops. The algorithm can be used for finding looped paths as well as simple paths. To find simple paths only, one detects loops by checking if the deviation node is repeated in the path sequence, as it is guaranteed that if the path has loops, then the deviation node will appear twice in the path. It is necessary to check to see if a path that deviates from a looped path contains the old loop, as well as see if any new loops are created. Even when looking for simple paths, it is necessary to generate the looped paths as part of the algorithm. Because of the additional looped paths generated in the process, the algorithm must iterate $h$ times, where $h \geq k$. Unfortunately, an exact value of $h$ cannot be computed since it will depend on the network topology. These looped paths are omitted in the program output, but are essential to the operation of the algorithm.

Figure 3 shows an example of how it is necessary in this algorithm to find looped paths. Figure 3 (a) shows an undirected network with four nodes and five arcs; with the arc costs displayed alongside each arc. To find the $k$-shortest simple paths from A to D, first we find the shortest path, ACD, with a cost of 2, as shown in Figure 3 (b). To find the second shortest path, one possible deviation is to deviate at node C via arc CB. The new path, Figure 3 (c), is made up of part of the 1$^{st}$ shortest path (CD), the deviation arc (CB), and the shortest path from B to A (BCA). Taking these sections in reverse, we form the overall path of ACBCD, with repeated node C. This path turns out to be the second shortest path from A to D, with a cost of 4, but it contains a loop. Next, we search all deviations from the shortest path, and also from the looped second shortest path. One deviation from the looped path is to deviate at node B via arc BA. This creates a new path from the previous path (BCD), a deviation arc (BA), and since the deviation arc ends at A, there is no further path. The new path is ABCD, with a cost of 7, and it is easy to see that it is the second shortest simple path, and the fourth shortest overall path (the third is ACBCBCD with length 6). It is also evident that it is impossible to generate ABCD as a single deviation of ACD, since ABCD is two deviations from ACD.

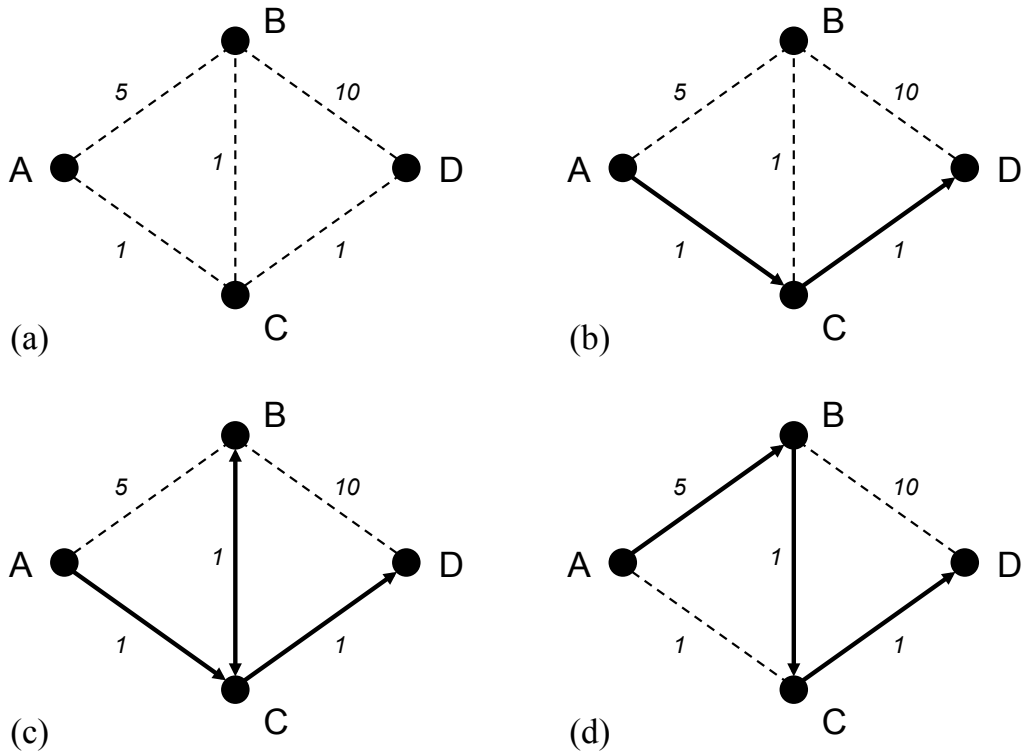


Figure 3 - Generating looped paths are essential to Hoffman & Pavley's algorithm

We calculate that the overall runtime of this algorthm as $O(c(n,m) + hm)$, where $c(n,m)$ is the cost to run Dijkstra's algorithm. But if it is used on a complete graph where $m = n^2$, then the runtime becomes $O(c(n,m) + hn^2) = O(hn^2)$. In his online KSP bibliography (Eppstein 2001), David Eppstein claims that for certain pathological networks the complexity becomes $O(c(n,m) + k^2m)$. Unfortunately, he does not describe an example of such a network, so we are unaware of the structure for which that complexity would occur.

Brander and Sinclair presented the only comparison that we could find of Hoffman and Pavley's algorithm to other KSP algorithms. In their paper, they ran experiments between the algorithms of Hoffman and Pavley, Yen, Lawler, and Katoh et al., tested on optical telecommunications networks. In their implementation, they found that Hoffman and Pavley's method greatly outperformed the other algorithms. It must be noted though that Brander's study was done prior to Hadjiconstantinou and Christofides (1999) publishing a definitive paper on how to efficiently implement Katoh's algorithm, which included slight modifications for improving the computational efficiency of that algorithm. Even so, we suspect that Hoffman and Pavley's method would still outperform Katoh's algorithm.

*Yen's Algorithm / Lawler's Improvement*

Similar to Hoffman and Pavley's algorithm, Yen's algorithm (Yen 1971) is also based on the idea that the $n$-th shortest path is a deviation of the ($n$-1)-th shortest path. Rather than deviating paths using deviation arcs and the shortest path tree, it instead calculates a shortest path for every

node on the previous path, temporarily eliminating key arcs to generate alternate routes. This approach eliminates the possibility of any looped paths ever appearing, at the expense of additional computation required to find each path.

Yen's algorithm starts with finding the shortest path $P_1$ on a directed or undirected graph G = ($V$, $E$), from source node $S$ to destination node $T$ using one of the many shortest path algorithms available. $P_1$ is composed of the nodes $\{v_1^1, v_2^1, \ldots, v_{Q1}^1\} \in V$, where $Q1$ is the number of nodes in the path $P_1$. Next, it calculates a set of candidate paths $A_i^2$, $i$ = 1, 2, … , ($Q$1-1) as deviations of the shortest path $P_1$. Each candidate path $A_i^2$ is formed as the union of two subpaths: the root $R_i^2$, and the spur $S_i^2$. The root, $R_i^2$, is the subpath of P1 consisting of $\{v_1^1, v_2^1, \ldots, v_i^1\}$. The spur, $S_i^2$ is the shortest path from deviation node $n = \{v_i^1 \mid n \in V\}$ to the destination node T, where the arc $e = \{(v_i^1, v_{i+1}^1) \mid e \in E\}$ is temporarily deleted from the network. Arc $e$ is also stored for node $n$, so that it will be omitted every subsequent time node $n$ is again used as a deviation node.

As they are calculated, each $A_i^2$ candidate path is added to a list *LIST*, where it is stored until all candidate paths are calculated. When all $A_i^2$ paths have been generated, the shortest path in *LIST* is selected as $P_2$, the second shortest path. It is removed from *LIST*, but all other calculated paths remain. To find the third shortest path, $P_3$, the process is repeated by finding all $A_i^3$ deviation paths deviating from $P_2$, adding them to *LIST*, and selecting the shortest from that. When adding a path to *LIST*, one must check if that path is already contained in *LIST*. If it is a repeated path, it is simply discarded.

To calculate the worst-case complexity of this algorithm, one can observe that a path $P_k$ may contain up to $n$ nodes (all nodes). Therefore, to calculate $P_{k+1}$, one must calculate $n$ deviation paths, each taking $O(c(n, m))$ time to calculate using some shortest path algorithm, where $c(n, m)$ is the time to run a shortest path routine such as Dijkstra's. This must be done $K$ times to find K-shortest paths, thus the overall complexity is $O(Kn\ c(n, m))$.

Shortly after Yen's algorithm was published, Eugene Lawler published a simple modification (Lawler 1972), which results in a major improvement in the runtime of the algorithm. He observed that when computing the candidate paths $A_i^{k+1}$ as deviations of the path $P_k$, it is only necessary to calculate deviations of the spur portion of $P_k$. This is because paths deviating from the root portion of $P_k$ will have already been calculated at a previous point, and thus are already included in LIST. This observation essentially cuts the number of iterations in half. While this has no effect on the overall worst-case complexity, it is a major improvement in average efficiency.

*Katoh's Algorithm*

For finding the $K$-shortest loopless paths in the case of an undirected graph, the Katoh et al. algorithm has the best-known worst-case complexity (Katoh *et al.* 1982). Like other $K$-shortest path algorithms, this algorithm calculates candidate paths as deviations from the previous shortest route. Unlike Yen's algorithm though, which executes up to $n$ iterations of Dijkstra's algorithm to find the next shortest path, the Katoh et al. algorithm performs only up to a constant 3 iterations each time, thereby achieving a worst-case time complexity of $O(K\ c(n, m))$.

The basis of the algorithm is that the next shortest path is found by determining the first shortest gateway-path[1] that is not part of any previous shortest path. An *S*-*T* gateway shortest path on the undirected graph G = (*V*, *E*) is the path starting at node *S*, ending at node *T*, in which the route is constrained to go through some intermediate node $n \in V$, known as the gateway node, or though an intermediate arc $a \in E$. A gateway shortest path through node *n* can be found by finding the shortest path from *S* to *n*, and finding the reverse shortest path from *T* to *n*. Gateway paths through arcs are found in a similar fashion. The path cost of a gateway shortest path is the sum of the forward path and the reverse path costs to and from the constraining element, plus the cost of the gateway arc if it is an arc gateway path.

Gateway shortest paths for all gateway nodes and arcs can be computed by generating the complete forward shortest path tree from *S* and the complete reverse-shortest path tree from *T*. This step takes $O(c(n, m))$ time. To find the shortest gateway path, one then scans the gateway path costs of all nodes $O(n)$ and arcs $O(m)$, and keeps the shortest one. Since $O(c(n, m)) > O(m) \geq O(n)$ for all graphs, the shortest path tree portion is the limiting factor in this step.

Using the same notation as above in describing Yen's algorithm, to find $P_k$, the Katoh et al. algorithm performs three forward-shortest path tree iterations of the above gateway path calculation, once on the spur portion of $P_{k-1}$, once on the spur portion of $P_{k-2}$, and once on the root shared by both $P_{k-1}$ and $P_{k-2}$. Each gateway iteration may have key elements of the graph removed to ensure mutually disjoint candidates. All iterations use the same reverse-path tree from the destination node *T*, calculated at the start of the program. For the three iterations, the shortest gateway path found is saved as a candidate path $A_a^k$, $A_b^k$, and $A_c^k$ respectively. It then simply follows that $P_k = \min\{A_a^k, A_b^k, A_c^k\}$.

Missing from the Katoh et al. publication was any implementation of the algorithm, with runtimes and comparisons to other algorithms. Hadjiconstantinou and Christofides (1999) published an excellent paper which gives a thorough treatment of how to implement the Katoh et al. algorithm, including pseudocode for all the relevant functions, and suggested data structures for efficiently storing and accessing the shortest paths. They also provided some suggestions for minor tweaks to the algorithm for some slight speed enhancements. Hadjiconstantinou and Christofides implemented the Katoh et al. algorithm and show results that prove that the running time is a linear function with respect to the number of paths found, *k*, and polynomial with respect to graph size, *n*. They do not compare their implementations to other algorithms.

*Carlyle & Woods' Near-Shortest Path Algorithm*

The Near-Shortest Path (NSP) problem is a slight variation on the *K*th-Shortest Path (KSP) problem. Unlike the KSP, which finds a set of *K* paths ranked in order of length, the NSP finds all paths less than a specified length. More specifically, near shortest paths are defined as paths whose lengths are within a factor of $(1 + \varepsilon)$ of the shortest path length for some user-specified $\varepsilon \geq$

[1] Katoh's paper does not use the term "gateway shortest path", referencing them instead as deviation paths. The concept of gateway paths was more formally and thoroughly explored in a later paper by Lombard and Church (1993), hence this is the terminology that we use in the description above.

0. This generates an unknown number of paths that are of a length within some threshold of the shortest path. The first to formulate this problem were Byers and Waterman (1984).
Carlyle and Wood (2005) modified the Byers and Waterman algorithm to constrain the results to only loopless near-shortest paths. They compared runtimes of their KSP algorithm to that of the Katoh et al. algorithm as implemented by Hadjiconstantinou and Christofides (1999), and declared theirs to be far superior despite using different networks and faster computers for their study. While no other experiments have been published that compare Carlyle and Wood's algorithm to other *k*-shortest path algorithms, in another publication by Carlyle, Royset and Wood (Carlyle *et al.* 2008), they argue that "enumerating paths in order of length requires undue computational effort, storage and algorithmic complexity", and if it is not necessary to use KSP, then NSP is far quicker.

In their 2005 paper, Carlyle and Wood present two different algorithms for finding loopless NSPs. The first one, ANSPR0, is directly based on the Byers and Waterman method, except for a slight modification to output only loopless paths. While it has an exponential worst-case complexity, it takes a pathological example to create this slow a scenario. Otherwise, the algorithm runs extremely fast. Their other algorithm, ANSPR1, has a better worst-case complexity, but when implemented runs slower than ANSPR0. Combined with a binary search tree, they showed that the ANSPR1 algorithm could be modified to solve the KSP problem with worst case complexity of $O(Kn\, c(n,m)\, (\log n + \log c_{max}))$, where $c(n,m)$ is the cost of running Dijkstra and $c_{max}$ is the largest edge length. They called this modified version AKSPR1, and when implemented it ran much faster than the Hadjiconstantinou and Christofides implemention of the Katoh et al. Algorithm.

The real insight in this paper is the ANSPR0 algorithm. When the problem doesn't require a KSP output, and NSP will suffice, the ANSPR0 algorithm runs many orders of magnitude faster than a traditional KSP algorithm. While the only comparison in this paper was with the Katoh et al. Algorithm, we suspect that ANSPR0 would run far faster than all other KSP algorithms as well.

On the next page, we have printed a pseudocode description of the ANSPR0 algorithm. The general idea of ANSPR0 is that it uses depth first search to find all paths of length $\leq D$ on the network, where D = (1 + ε) × (Shortest Path Length). First it solves the reverse shortest path tree (from destination to origin) to acquire the shortest path cost from any node to the destination, *t*. This is the only time that a shortest path algorithm is solved. It then builds NSP's by adding nodes to a first-in last-out stack, *theStack*. When a vertex *v* is added to *theStack*, its $\tau(v)$ is set to 1, denoting that it is in the stack. This prevents it from being added again to the stack, satisfying the loopless criteria. After initializing by pushing the starting node, *s*, onto *theStack*, it peeks at the top node, *u*, in the stack (in the first case, the just placed starting node), and starts iterating through all edges $(u, v)$ from that node. For each edge, it evaluates the sum of the path cost of the path in the stack up to that point, $L(u)$, plus the arc cost $c(u, v)$, plus the shortest path cost (acquired from the shortest path tree) from the arc's end node $d'(v)$, and checks if it's less than the max path cost *D*. If $L(u) + c(u,v) + d'(v) \leq D$ and $\tau(v) = 0$ (meaning *v* is not in the stack yet), then *v* is added to the stack, $L(v)$ and $\tau(v)$ are updated, and the process repeats. This continues until the stack path reaches the destination node *t*. When that occurs, the path is output or saved, and the top node in *theStack* is popped (removed). The new top node is "peeked", and the remaining arcs that did not get evaluated after finding the first one that fit the ≤ D threshold are

evaluated until one meets the criteria. If no other arcs meet the $\leq D$ requirement, then the top node in *theStack* is again popped, and the process is repeated until a $\leq D$ arc is found. The path then moves forward again until reaching the destination, then again backtracks, and so on, until all possible paths that are $\leq D$ have been found.

This approach is very efficient because the depth-first approach consists of fast addition/comparison operations, and never has to repeat any calculations in the process of generating paths. Also, not having to store a list of candidate path lengths to determine the next-shortest path length saves time and memory. Overall, this approach is a very streamlined and efficient method of quickly enumerating an enormous set of paths.

**ANSPR0 Algorithm**

DESCRIPTION: An algorithm to solve loopless NSP.

INPUT: A directed graph $G = (V, E)$ in adjacency list format, $\tau$ , $s$, $t$, $\mathbf{c} \geq \mathbf{0}$, and $\varepsilon \geq 0$.
"firstEdge($v$)" points to the first edge in a linked list of edges directed out of $v$.

OUTPUT: All $s$-$t$ paths (may include loops), whose lengths are within a factor of $1 + \varepsilon$ of being shortest.

```
{ /* A single shortest-path calculation evaluates all d'(v) in the next step. */
  for (all v ∈ V ) { d'(v) ← shortest-path distance from v to t; }
  D ← (1 + ε) d'(s);
  for (all v ∈ V) { nextEdge(v) ← firstEdge(v); }
  theStack ← s; L(s) ← 0;
  /* τ(v) denotes whether the vertex v appears on the current subpath. */
  τ(s) ← 1;
  for (all v ∈ V − s ) { τ (v) ← 0; }
  while( theStack is not empty ) {
    u ← vertex at the top of theStack;
    if( nextEdge(u) ≠ null ) {
      (u, v) ← the edge pointed to by nextEdge(u);
      increment nextEdge(u);
      if( L(u) + c(u,v) + d'(v) ≤ D and τ(v) = 0) {
        if( v = t ) {
          print( theStack ∪ t );
        } else {
          push v on theStack;
          τ(v) ← τ(v)+1;
          L(v) ← L(u) + c(u, v);
        }
      }
    } else {
      Pop u from theStack;
      τ(u) ← τ(u)−1;
      nextEdge(u) ← firstEdge(u);
    }
  }
}
```

*Comparison of the Algorithms*

Brander and Sinclair (1995) wrote the only paper with any comparison of most of the algorithms discussed in this section. In that paper, they compared implementations of Hoffman and Pavley's Algorithm, Yen/Lawler's Algorithm, and the Katoh et al. Algorithm. Below are the results they found on running the various algorithms on their example network, where they found that the Hoffman and Pavley algorithm performed far faster than any of the other algorithms, followed by Lawler, Yen, and Katoh.

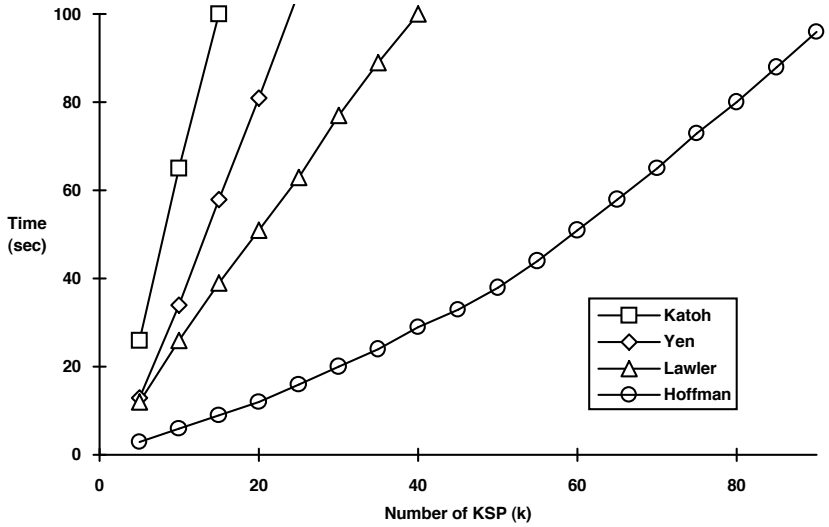


Figure 4 - Comparative KSP algorithm results from Brander and Sinclair (1995)

Unfortunately, the extent of their computational tests was very limited, as they only ran experiments on a single network. The network they tested was the COST 239 European Optical Network (O'Mahony 2002), a 20-node optical telecommunications network. This network is far smaller than those that would be used for a corridor location problem, and thus the results are by no means definitive.

It would be expected that Lawler's improvement would make it twice as fast as Yen's, and the results show that this is true. Even though the Katoh et al. algorithm has a far better worst-case complexity than Yen/Lawler's, they found it to run slower. The cause of this is because they used such a small network. Brander and Sinclair note that while the Yen and Lawler methods calculate shortest-paths, the Katoh et al. algorithm calculates shortest-path trees. On average, it takes twice as long to calculate a shortest-path trees than it does to compute a shortest-path, which means that for the Katoh et al. algorithm to be faster than another algorithm, that algorithm must make at least six calls to the shortest-path algorithm being used for each KSP calculated.[2] This means that the Katoh et al. algorithm will be faster than Lawler's version of Yen's algorithm if the paths are longer than 12 nodes long. For their 20 node network though, this was rarely, if ever, the case.

[2] Brander and Sinclair calculated 6 shortest path trees for each path in Katoh's algorithm, but one only needs to calculate 3 each time, since the reverse tree is always the same. This means their claim that an algorithm must take at least twelve calls to be slower than Katoh is false. It should in-fact be six.

The only comparison of algorithms between Carlyle and Wood's NSP algorithm and any KSP algorithm is found in Carlyle and Wood's (2005) paper. As mentioned before, they found that both their KSP modified ANSPR0 and ANSPR1 algorithms (called AKSPR0 and AKSPR1, respectively) ran orders of magnitude faster than the Katoh et al. Algorithm. Paraphrasing their words:

> Hadjiconstantinou and Christofides indicate a run time for their KSPR algorithm of over 1400 seconds when $K = 103$ for a test graph having 1000 vertices and an edge-to-vertex density of 4. This graph is of roughly the same size and structure as our Grid 40 × 25, which we solve in 0.59 seconds with the polynomial-time variant of **AKSPR1**; and the $D_\infty$ variant (**AKSPR0**) solves this problem so quickly that it does not register with the time functions in the standard C library. Indeed, our best algorithm produces $10^7$ paths in about one 40th of the time in which their algorithm produces $10^3$ paths. Even taking our faster computer into account, it is safe to conclude that our best algorithms are several orders of magnitude more efficient than theirs.

Clearly, they are confident that this approach is vastly superior to the Katoh et al. algorithm. It would make sense to run definitive experiments comparing Hoffman and Pavley's KSP algorithm to Carlyle and Wood's NSP algorithm, as these appear to be the two fastest overall. We suspect, though, that the NSP algorithm would still run faster because of the extra overhead needed in the KSP to store candidate paths and the need to check for the existence of loops.

*Summary*

Based upon our thorough literature review of the algorithms available for generating sets of *K*-shortest paths and near shortest paths, we selected four for closer review (Yen, Katoh et al, Carlyle and Wood, and Hoffman and Pavley). This section has described these algorithms in detail, and a comparison of these four algorithms is summarized in the table below.

| **A Summary of the Most Promising Loopless KSP and NSP Algorithms** | | | |
|---|---|---|---|
| **Algorithm** | **Worst Case Complexity** | **Improvements** | **Comments** |
| Hoffman & Pavley (1959) | $O(K^2m)$ (Eppstein, 2001) | none | Potentially very fast, but mostly untested in the literature. |
| Yen (1971) | $O(Kn\ c(n, m))$ | Lawler (1972)<br>Perko (1986)<br>Martins & Pascoal (2003) | Easiest KSP algorithm to implement, but slower than some others. |
| Katoh, et al.(1982) | $O(K\ c(n, m))$ | Hadjiconstantinou and Christofides (1999) | Best worst-case complexity and fast runtime on all but the smallest graphs. Only works on undirected graphs. |
| Carlyle & Wood (2005) | ANSPR0: exponential<br>ANSPR1: polynomial<br>AKSPR1: $O(Kn\ c(n,m)(\log n + \log c_{max}))$ | none (yet) | All variants are fairly easy to implement. These are likely the fastest algorithms for generating sets of shortest paths. |

In the next section we discuss an implementation of the ANSPR0 version of the Carlyle and Wood near shortest paths algorithm. We found this algorithm to be simple to implement, and that it shows promise of generating paths faster than any other algorithm. Generating path alternatives for corridor location does not necessitate having the set of paths be in ranked order of length, so the savings in processor time and memory were found to be especially welcome when generating sets of $>10^7$ paths.

## Implementation

We implemented the Carlyle & Wood ANSPR0 algorithm in Java using the Processing libraries (processing.org) to augment and simplify the visualization capabilities. This implementation was run on the following sample data networks:

1) 20 x 20 manually fabricated raster. This network contains 400 nodes and 2,850 arcs using an R = 2 arc set[3]
2) 80 x 80 subset of the Maryland Automated Geographic Information System (MAGI) database. This network contains 6,400 nodes and 49,770 arcs using an R = 2 arc set[3]

Both of these networks are smaller than what would be used in a real-world corridor location application, but they are adequate for use in our present "proof of concept" stage of development. Larger data sets would require more powerful computing assets than what we have available to generate viable path alternatives in a reasonable amount of time. The approaches used though are easily scalable for implementation on supercomputing resources. As it stands now, our program will accept networks of any size as a raster of node costs in the standard .asc file format, and generate the arcs for an R = 0, 1, or 2 system.

*Memory Performance*

We ran tests on both networks for numerous values of ε to see how the number of paths increased as we increased the threshold value ε. Figure 5 displays these results in a linear plot for the 20x20 network, and Figure 6 displays them on a logarithmic plot.

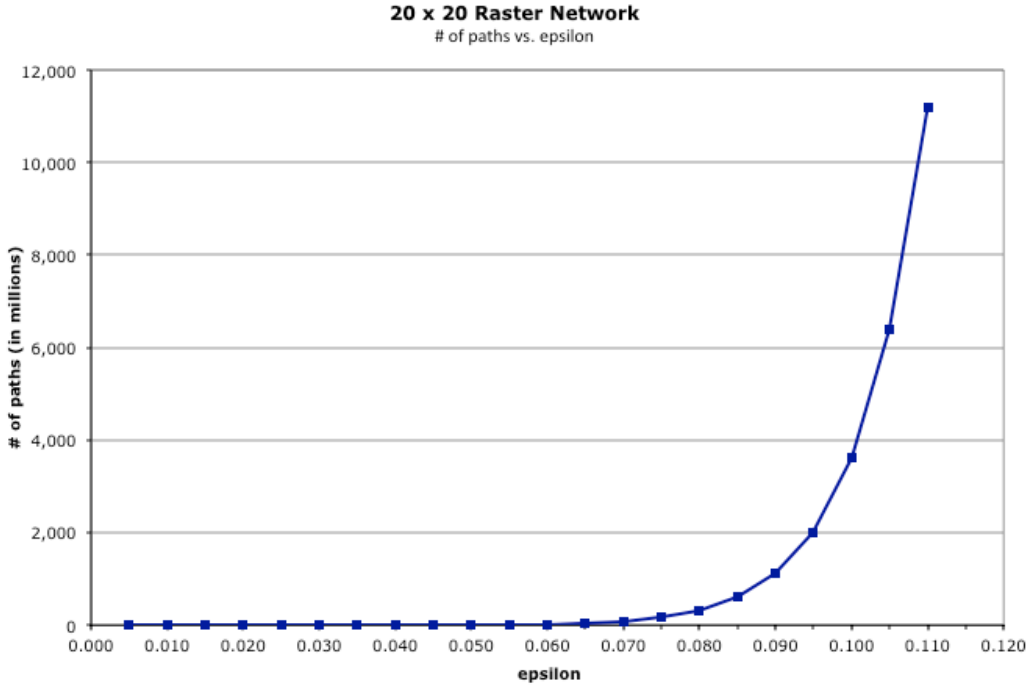


Figure 5 - 20x20 network, number of paths vs. epsilon

[3] For information on the R-value for arcs on a raster network, please refer to Huber and Church (1985).

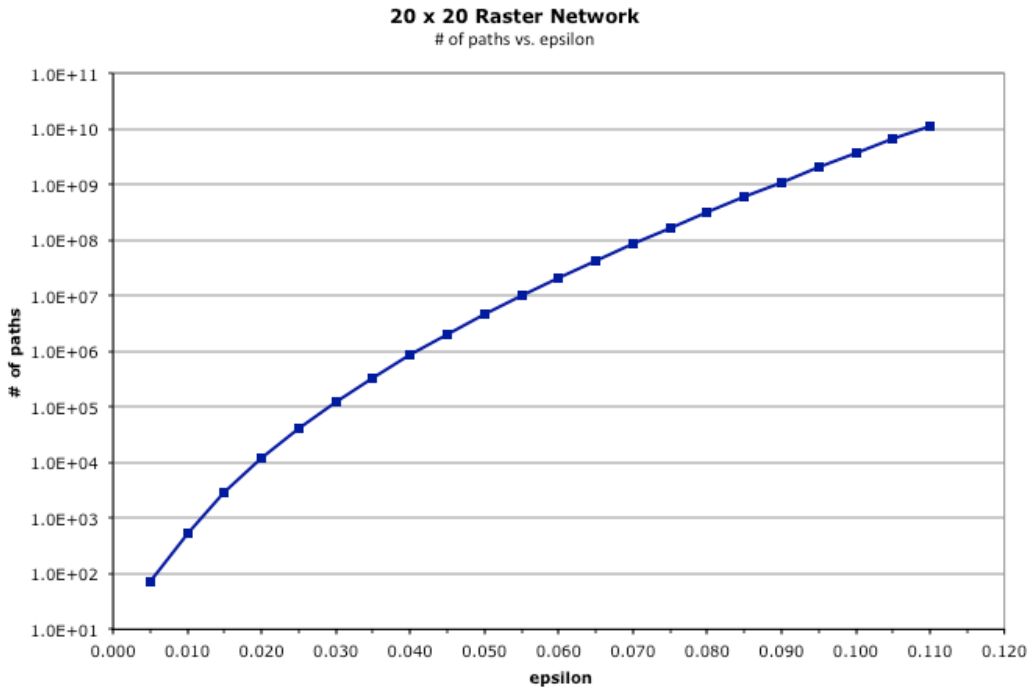


Figure 6 - 20x20 network, log number of paths vs. epsilon

Figure 5 shows the rapid rate of growth of the number of paths found as a function of the threshold parameter ε, while Figure 6 shows the same data plotted on a logarithmic y-axis. In Figure 6, after an initial ramp-up, the curve flattens out into a straight line, suggesting an exponential rate of growth in the number of paths. Eventually, we would expect the curve to flatten out as we approach finding all possible combinations of paths. The range of epsilons we used come nowhere near this boundary, and we would expect the trend we plot to continue for epsilon values at least a couple orders of magnitude higher than those which we computed.

We ran the same experiment for the 80x80 data, and found similar results. Because the number of paths for each given value of ε is much higher for the 80x80 as compared to the 20x20, the range of epsilons used in our 80x80 experiments are an order of magnitude smaller than those in our 20x20 experiments. For example, for ε =0.005, on the 20 x 20 data there are 73 paths, but for the 80x80 there are 510,343,616 paths. In Figure 7, we again see the rapid rate of growth in the number of paths as ε increases. The data is plotted on a logarithmic plot in Figure 8, again showing a linear trend after an initial ramp-up, confirming the exponential growth of the problem.

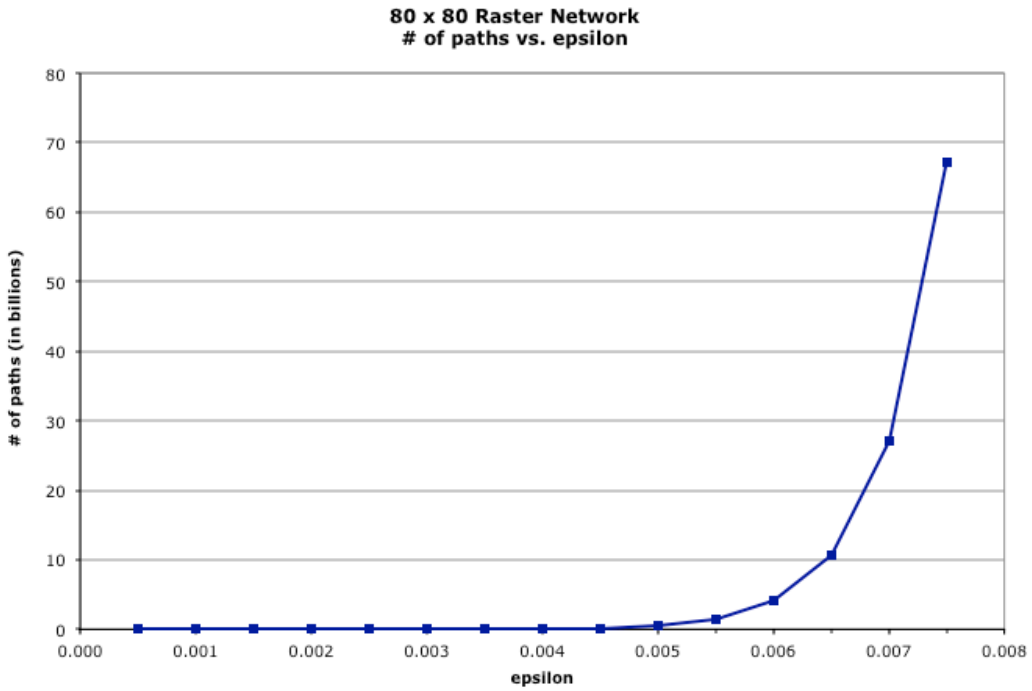


Figure 7 - 80x80 network, number of paths vs. epsilon

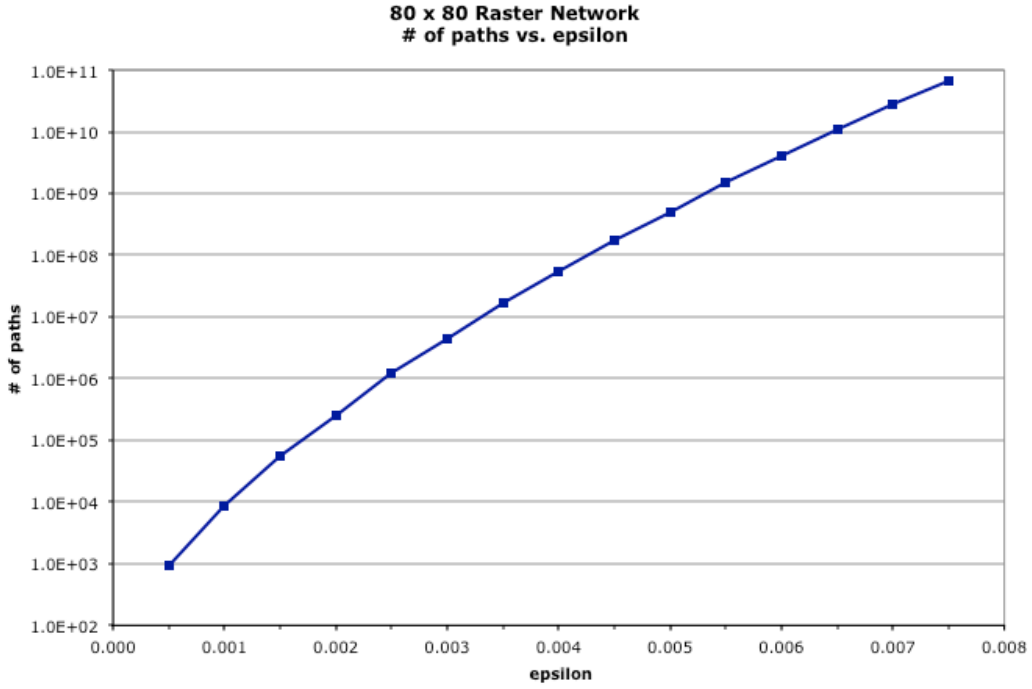


Figure 8 - 80x80 network, log number of paths vs. epsilon

For both networks, the number of paths generated grew exponentially as we increased the threshold value ε. This presents problems in data storage, as the memory requirement to store these paths is proportional to the number of paths generated. It is easy to see how this could present a challenge as the problem size increases to something more on the order in scale of what would be used in a real-life corridor location application.

*Time Performance*

We were also interested in seeing the relationship in the amount of time it took to compute paths as we increased the threshold value ε. It was necessary to run the computations on the same computer with nothing else running in the background that could disrupt computations, so we used a Windows XP computer with a Pentium 4 single core processor running at 2.53 GHz and 512 MB of RAM. Newer computers would run the code faster by some ratio, but the overall data profile would be the same.

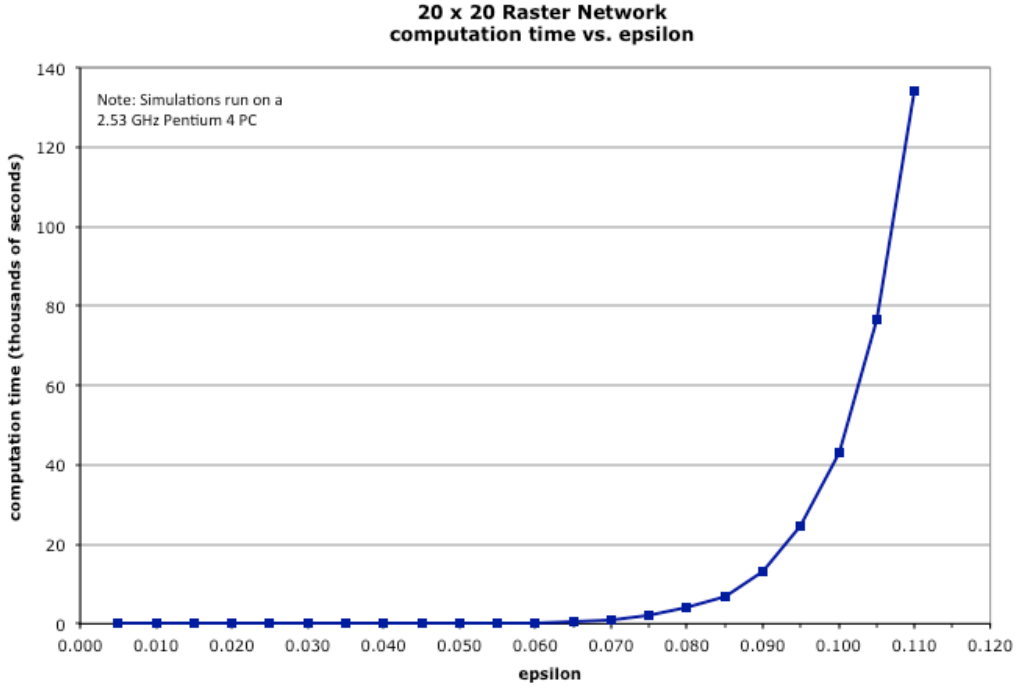


Figure 9 - 20x20 network, computation time vs. epsilon

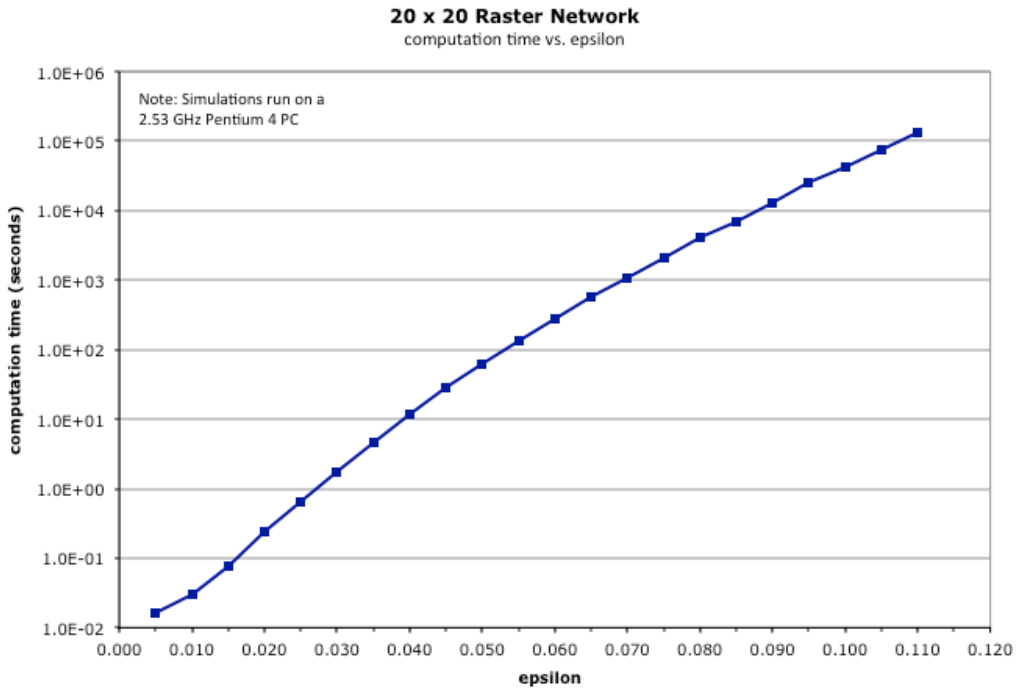


Figure 10 - 20x20 network, log computation time vs. epsilon

Similar to the memory performance, Figure 9 shows that the time to compute all paths within a threshold increases at a rapid rate as ε increases. Figure 10 plots the data on a logarithmic y-axis, and shows a straight line trend suggesting that this growth is exponential. These results are analogous to what we found in the number of paths found as a function of ε, in that both show an exponential increase as we increase the value of ε.

*Conclusions*

Generating sets of NSPs quickly grows to an overwhelming size as we increase the network size or increase the value of ε. Generating all paths within 0.75% of the shortest path length on our 80x80 network took a new Intel Core i7 desktop a little over 4 days to solve at a rate of 185,000 paths per second. While we make no claims that our code cannot be further optimized, the reality is that even with the best code, generating all paths within 10% of the shortest path on a 100 megapixel raster is beyond the reach of any commercial off-the-shelf computer available today. A discussion on possible approaches on how to solve such a problem can be found in the Future Work section at the end of this report.

## Conclusions and Future Work

The purpose of this study was to review the existing literature on *K*-Shortest Path algorithms, evaluate their capability of being implemented on large raster networks, and examine their efficacy in finding viable alternative routes for corridor location of transmission lines. We have examined the literature on KSP algorithms, from its origins in the 50's to the modern day state of the art, and identified a handful that stand out as being particularly well suited for finding loopless paths on larger raster networks. In particular, we believe that the Near Shortest Paths approach by Carlyle and Wood (2005), which solves an unranked relaxation of the KSP, is the top candidate.

Even with efficient algorithms such as the Carlyle and Wood's NSP algorithm, the combinatorial nature of shortest path sets continues to make the time and memory requirements for generating path sets a daunting task. We found there to be exponential growth in both time and memory as we increased both the network size and the path-length threshold. Our experiments were performed on small 20 x 20 and 80 x 80 networks, and yet many of the simulations took more than a day to run. To be useful as a tool for corridor location, solution methods must be able to run on a 10,000 x 10,000 network. In the remainder of this section, we outline future directions and areas of research that require further development in order to expand these approaches to scale necessary in making this a valuable tool for transmission line planners.

### *High Performance Computing for KSP/NSP*

Even on our small test networks, the computation time of finding all NSP of a useful threshold value grew at a rapid rate. It became clear that in order to perform this task on problems of real size (say a 10,000 x 10,000 raster network), supercomputers would have to be used. Supercomputers present unique challenges, as they use specialized programming languages such as MPI, UPC, OpenMP, or Cilk, which allow for dividing up the computation tasks among any number of processors or threads. These languages bring with them new challenges for the programmer, bringing up issues of deciding how to divide the data and computation.

Fortunately, the depth-first search approach used by NSP can be split into any number of independent threads in a fairly simple manner, making it highly suitable for translation into a parallel computing environment. One can begin by implementing the ANSPR0 algorithm in a breadth-first search fashion, building a "trie" data structure (Hadjiconstantinou and Christofides 1999) of all vertex combinations of NSPs. The program would run in this fashion until the trie has the same number of leaves as the number of processors available for computation. When that point is reached, the problem is then split into numerous sub-problems, with each leaf node as the new starting node, and solved independently on separate processors. If some processors finish earlier than others, load balancing can occur by splitting up existing threads mid-flight, with the process continuing with all resources until the final solution set has been found. While conceptually the idea is fairly simple, considerable work will need to be invested in crafting this method for a supercomputer environment. We have begun learning the MPI programming language in order to prepare to implement this on the ANL supercomputing resources.

*Explore p-Dispersion*

After generating a large set of paths through some KSP or NSP algorithm, one may want to extract from that set a smaller set of "best" spatially different paths. The *p*-dispersion method is an optimization model that selects a set of *p* items out of a larger set, which maximizes the total difference between the *p* items using some difference metric. This model, using a path area difference metric, appears to be one alternative to extract a representative set of spatially distinct best paths from the larger set. Unfortunately, the *p*-dispersion problem is NP-complete, and thus computationally very difficult to solve to optimality. Current methods of approximating a solution use a semi-greedy heuristic, followed by a local search to improve the initial solution. It is unknown how well this approach would work on a problem of the scale of selecting the 10 or 100 best paths out of a set of 1 trillion paths; so further work must be done to see if this solution heuristic is appropriate for such an approach.

*Explore Gateway Paths / Hybrid NSP-Gateway Approach*

Even with the use of supercomputers, it may be necessary to reduce the size of a problem to bring it down to a manageable size. The gateway shortest path problem (Church *et al.* 1992, Lombard and Church 1993) is another method for efficiently generating sets of alternative routes on a raster network. Essentially a form of the constrained shortest path problem, a gateway path is the shortest path from an origin to a destination, constrained to also traverse through one or more particular intermediate points. While the gateway approach has shown great promise in being able to generate good paths with relatively little amounts of computation, refining this method for automated selection of gateway points to generate "good path sets" is an ongoing topic of research that we are currently involved.

We also propose that we can use gateway paths as a smart method of reducing a NSP problem size by numerous orders of magnitude. Essentially, this method would subdivide a raster network into macro-cells of some pre-determined size, and connect these macro cells by macro-arcs composed of the shortest paths between the lowest-cost gateway nodes contained in each of the macro-cells. We believe that this approach would retain the meso-scale topological features of the original network while dramatically reducing the size of the network upon which the NSP algorithm would run. As an example, Figure 12 shows a 16 x 16 network on the left. Using the r = 2 arc connectivity metric, this gives us a raster network with 256 nodes and 1,770 arcs. Dividing the network into 4 x 4 macro-cells, connecting each macro-cell by the r = 1 macro-arc consisting of the shortest path between low-cost gateway nodes contained in each macro-cell, we are able to reduce this network to one of 16 nodes and 42 arcs.

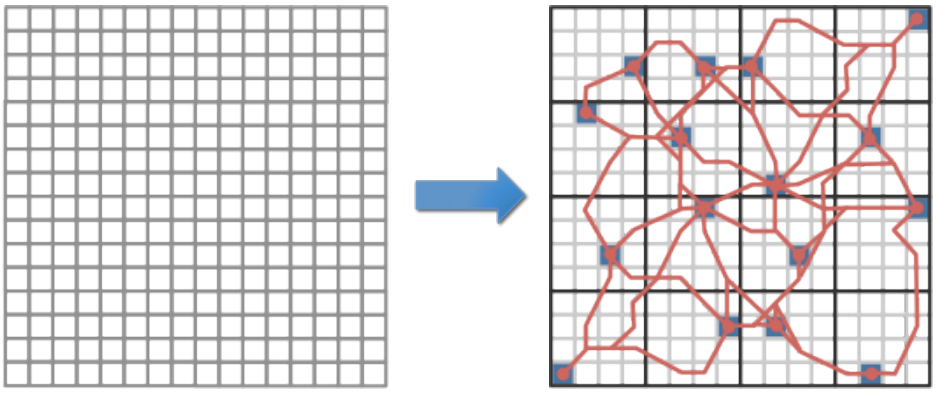

Figure 12 - Using gateway paths to reduce the size of a grid of 256 nodes and 1,770 arcs (r = 2) to a grid with 16 nodes and 42 arcs

On larger networks, these macro-cells could be larger, such as 100 x 100 or even more, especially if the terrain is relatively homogenous, resulting in even greater reductions on problem size. Irregular meshes could be used as well, in order to retain high detail in regions if major change, while saving space in regions of little change. We hypothesize that any of these approaches would retain the essential path options necessary for finding "good" paths; yet dramatically reduce the computational requirements for finding these paths using the NSP algorithm. An implementation of this, with thorough testing and analysis is still necessary to determine if this is indeed true.

## Acknowledgements

We would like to thank Argonne National Laboratories for providing the funding to conduct the initial part of this research. We also wish to acknowledge the support of the University of California Transportation Center for providing fellowship support for Mr. Medrano during part of this research as well.

## Bibliography


Ahuja, R., K. Mehlhorn, J. Orlin & R. Tarjan, (1990). Faster algorithms for the shortest path problem. *Journal of the ACM (JACM),* 37, 213-223.

Azevedo, J.A., M.E.O.S. Costa, J.J.E.R.S. Madeira & E.Q.V. Martins, (1993). An algorithm for the ranking of shortest paths. *European Journal of Operational Research,* 69, 97-106.

Azevedo, J.A., J.J.E.R.S. Madeira, E.Q.V. Martins & F.M.A. Pires, (1994). A computational improvement for a shortest paths ranking algorithm. *European Journal of Operational Research,* 73, 188-191.

Bellman, R.E., (1958). On a routing problem. *Q. Applied Math,* 16, 87-90.

Bock, F., H. Kantner & J. Haynes, (1957). An algorithm (the r-th best path algorithm) for finding and ranking paths through a network. *Research report, Armour Research Foundation of Illinois Institute of Technology, Chicago, Illinois*.

Brander, A. & M. Sinclair, (1995). A comparative study of k-shortest path algorithms. *Proc. 11th UK Performance Engineering Workshop for Computer and Telecommunications Systems*Citeseer.

Byers, T. & M. Waterman, (1984). Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. *Operations Research,* 32, 1381-1384.

Carlyle, W. & R. Wood, (2005). Near-shortest and k-shortest simple paths. *Networks,* 46, 98-109.

Carlyle, W.M., J.O. Royset & R.K. Wood, (2008). Lagrangian relaxation and enumeration for solving constrained shortest-path problems. *Networks,* 52, 256-270.

Cherkassky, B., A. Goldberg & T. Radzik, (1996). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming,* 73, 129-174.

Church, R.L., S.R. Loban & K. Lombard, (1992). An interface for exploring spatial alternatives for a corridor location problem. *Computers & Geosciences,* 18, 1095-1105.

Corman, T., C. Leiserson & R. Rivest, (1990). *Introduction to algorithms*: McGraw-Hill, New York.

Dantzig, G., (1957). Discrete-variable extremum problems. *Operations Research,* 266-277.

Dial, R., (1969). Algorithm 360: Shortest-path forest with topological ordering [h]. *Communications of the ACM,* 12, 632-633.

Dijkstra, E.W., (1959). A note on two problems in connexion with graphs. *Numerische Mathematik,* 1, 269-271.

Dreyfus, S., (1969). An appraisal of some shortest-path algorithms. *Operations Research,* 17, 395-412.

Eppstein, D., (1998). Finding the k shortest paths. *Siam Journal on Computing,* 28, 652-673.

Eppstein, D., 2001. *Bibliography on k shortest paths and other "k best solutions" problems* [online]. http://www.ics.uci.edu/~eppstein/bibs/kpath.bib.

Fakcharoenphol, J. & S. Rao, (2006). Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences,* 72, 868-889.

Ford, L., (1956). Network flow theory. *Rand Corporation Technical Report,* P-932.

Fox, B., (1975). K-th shortest paths and applications to the probabilistic networks. *ORSA/TIMS Joint National Meeting 23*, B263.

Fredman, M. & R. Tarjan, (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM),* 34, 596-615.

Glover, F., R. Glover & D. Klingman, (1984). Computational study of an improved shortest path algorithm. *Networks,* 14, 25-36.
Glover, F., D. Klingman & N. Phillips, (1985). A new polynomially bounded shortest path algorithm. *Operations Research,* 33, 65-73.
Goldberg, A. & T. Radzik, (1993). A heuristic improvement of the bellman-ford algorithm. *Applied Mathematics Letters,* 6, 3-6.
Golden, B. & M. Ball, (1978). Shortest paths with euclidean distances: An explanatory model. *Networks,* 8, 297-314.
Goodchild, M., (1977). An evaluation of lattice solutions to the problem of corridor location. Environment and Planning A, 9, 727-738.
Guerriero, F., R. Musmanno, V. Lacagnina & A. Pecorella, (2001). A class of label-correcting methods for the k shortest paths problem. *Operations Research,* 423-429.
Hadjiconstantinou, E. & N. Christofides, (1999). An efficient implementation of an algorithm for finding k shortest simple paths. *Networks,* 34, 88-101.
Hart, P., N. Nilsson & B. Raphael, (1968). A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics,* 4, 198.
Hershberger, J., M. Maxel & S. Suri, (2007). Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Trans. Algorithms,* 3, 45.
Hershberger, J., S. Suri & A. Bhosle, (2007). On the difficulty of some shortest path problems. *ACM Transactions on Algorithms (TALG),* 3, 1-15.
Hoffman, W. & R. Pavley, (1959). A method for the solution of the n th best path problem. *Journal of the ACM (JACM),* 6, 506-514.
Huber, D.L. & R.L. Church, (1985). Transmission corridor location modeling. *Journal of Transportation Engineering-Asce,* 111, 114-130.
Jiménez, V. & A. Marzal, 1999. Computing the k shortest paths: A new algorithm and an experimental comparison. *Algorithm engineering.* 15-29.
Jiménez, V. & A. Marzal, 2003. A lazy version of eppstein's k shortest paths algorithm. *Experimental and efficient algorithms.* 179-191.
Katoh, N., T. Ibaraki & H. Mine, (1982). An efficient algorithm for k shortest simple paths. *Networks,* 12, 411-427.
Lawler, E.L., (1972). A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science,* 18, 401-405.
Leng, J. & W. Zeng, (2009). An improved shortest path algorithm for computing one-to-one shortest paths on road networks. *icise,* 1979-1982.
Lombard, K. & R. Church, (1993). The gateway shortest path problem: Generating alternative routes for a corridor location problem. *Geographical Systems,* 1, 25-45.
Martins, E. & J.L. Esteves Dos Santos, 1999. *A new shortest paths ranking algorithm.* Coimbra, Portugal.
Martins, E. & M. Pascoal, 2000. *An algorithm for ranking optimal paths.* Coimbra, Portugal.
Martins, E., M. Pascoal & J. Santos, (1999). Deviation algorithms for ranking shortest paths. *International Journal of Foundations of Computer Science,* 10, 247-262.
Martins, E.D.V., (1984). An algorithm for ranking paths that may contain cycles. *European Journal of Operational Research,* 18, 123-130.
Martins, E.Q.V. & M.M.B. Pascoal, (2003). A new implementation of yen's ranking loopless paths algorithm. *4OR: A Quarterly Journal of Operations Research,* 1, 121-133.

Martins, E.Q.V., M.M.B. Pascoal & J.L. Esteves Dos Santos, (1998). The k shortest paths problem. *International Journal of Foundations of Computer Science*.
Mcharg, I.L. & American Museum of Natural History., (1969). *Design with nature,* [1st ed. Garden City, N.Y.,: Published for the American Museum of Natural History [by] the Natural History Press.
Minty, G., (1957). A comment on the shortest-route problem. *Operations Research,* 5, 724-724.
Moore, E., (1959). The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching*, Harvard University, 285-292.
Newkirk, R.T. (1976). A computer based planning system to optimize environmental resource allocations when locating utilities, PhD dissertation, University of Western Ontario, London, Ontario.
O'Mahony, M., (2002). Results from the cost 239 project. Ultra-high capacity optical transmission networks, IEEE, 11-18.
Owens, G.B. (1975). Evaluating a highway locational model: The London 402 controversy, Masters thesis, University of Western Ontario, London Ontario.
Pallottino, S., (1984). Shortest-path methods: Complexity, interrelations and new propositions. *Networks,* 14, 257-267.
Pape, U., (1974). Implementation and efficiency of moore-algorithms for the shortest route problem. *Mathematical programming,* 7, 212-222.
Perko, A., (1986). Implementation of algorithms for k shortest loopless paths. *Networks,* 16, 149-160.
Pollack, M., (1961). The kth best route through a network. *Operations Research,* 9, 578-580.
Pollack, M., (1961). Solutions of the kth best route through a network--a review. *Journal of Mathematical Analysis and Applications,* 3, 547-559.
Potts, J.M. (1975) "Concept and location algorithm for a multi-purpose communications corridor," Masters thesis, University of Western Ontario, London, Ontario.
Rink, K., E. Rodin & V. Sundarapandian, (2000). A simplification of the double-sweep algorithm to solve the k-shortest path problem* 1. *Applied Mathematics Letters,* 13, 77-85.
Sedgewick, R. & J. Vitter, (1986). Shortest paths in euclidean graphs. *Algorithmica,* 1, 31-48.
Shier, D., (1974). Computational experience with an algorithm for finding the k shortest paths in a network. *Journal of Research,* 78.
Shier, D., (1976). Iterative methods for determining the k shortest paths in a network. *Networks,* 6, 205-229.
Shier, D., (1979). On algorithms for finding the k shortest paths in a network. *Networks,* 9, 195-214.
Skiscim, C. & B. Golden, (1987). Computing k-shortest path lengths in euclidean networks. *Networks,* 17, 341-352.
Skiscim, C. & B. Golden, (1989). Solving k-shortest and constrained shortest path problems efficiently. *Annals of Operations Research,* 20, 249-282.
Smart, C. W. (1976). A computer-assisted technique for planning minimum impact transmission right of way routes, PhD dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA.
Turner, A.K. (1968). Computer-assisted procedures to generate and evaluate regional highway alternatives, PhD dissertation, Purdue University, Lafayette, IN.
Yen, J.Y., (1971). Finding the k shortest loopless paths in a network. *Management Science,* 17,

712-716.

Zeng, W. & R.L. Church, (2009). Finding shortest paths on real road networks: The case for a*. *International Journal of Geographical Information Science,* 23, 531-543.

Zhan, F.B. & C.E. Noon, (1998). Shortest path algorithms: An evaluation using real road networks. *Transportation Science,* 32, 65-73.