A New Parallel Algorithm to Solve the Near-Shortest-Path Problem on Raster Graphs

F. Antonio Medrano and Richard L. Church

Project 301CR, GeoTrans RP-01-12-01
January 2012

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

A New Parallel Algorithm to Solve the Near-Shortest-Path Problem on Raster Graphs

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 2012

Abstract

The Near Shortest Path algorithm (Carlyle and Wood 2005) has been identified as being effective at generating sets of good route alternatives for developing new infrastructure. While the algorithm itself is faster than other shortest path set approaches such as solving the *k*-th shortest path problem, the solution set size and computation time grows exponentially as the problem size increases and requires the use of high-performance parallel computing to solve real world corridor location problems. We identified a new breadth-first-search parallelization of the Near Shortest Path algorithm, although its efficiency was limited by large discrepancies in workloads from each processing thread. In an effort to more equally distribute work, we defined a metric that can be used to predict workloads from different parts of the breadth first search tree reducing the overall variability of workload between threads. This resulted in much improved algorithm performance and parallel computing efficiencies. Future work should focus on refining this new approach, and developing guidelines for implementing this method over a variety of datasets.

Table of Contents

Introduction

The shortest path problem has been defined on a network and on a continuous space domain. For example, one might want to find the shortest route on a road network or the least time sailing route across an ocean given currents and wind conditions. The problem that we address here involves the use of a network. The network for the purposes of this research represents all possible directions for an electrical transmission corridor as it traverses its way across the landscape. This network is described in more detail below. The main objective is to find corridor routes that are Pareto optimal or nearly Pareto optimal, given several incommensurate, competing objectives. To do this requires finding the shortest path along with paths that are close to being the shortest, in order to generate, characterize, and compare all competitive paths. One of the ways to meet this objective is to use a near shortest path algorithm.

In this report we describe what we believe is the fastest near shortest path program that has been developed to date. We then provide data generated from a test of this algorithm and show that for even small networks, the number of alternative routes can be exceedingly large. As the execution time of the algorithm is, in part, a function of the number of alternate routes generated, the execution time can be quite large as well. The main alternative to increasing the speed of such an algorithm is to develop a parallel algorithm for use on a computer with multiple processors. This report addresses specific issues in developing a parallelized near shortest path algorithm, with the objective of maximizing parallel efficiency.

In the next section we provide a short history in the development of methods to solve the shortest path problem and near short path problems. We then describe in more detail the near shortest path algorithm of Carlyle and Woods. This algorithm has been shown to be the most efficient method developed to date to solve for near shortest paths. We then discuss some details concerning the network on which a corridor transmission location problem is defined. Whereas road networks have 2.6 to 3.3 arcs per network node, transmission planning networks require 16 to 32 arcs per network node. This level of arc density is required to address on-the-ground variations in topography, costs, and impacts. Unfortunately, this level of arc density increases the overall work load of the near shortest path algorithm and the time it takes to complete its task. We provide details associated with the nature of this problem and discuss the needs to develop a parallelized version of the near shortest path problem. We show that a straightforward implementation of parallelizing this algorithm results in a very low level of parallel efficiency. We then discuss methods that have been developed as a part of this project to increase this efficiency and show that there is a promising approach to making a substantial improvement in efficiency when used on a massively parallel machine.

Background

Shortest path algorithms defined for network problems have been an active area of computational research since the 50's, with the shortest path problem initially being formulated as a linear programming program by Orden as a special form of a transshipment problem and then by Dantzig as a direct path problem (Orden 1956, Dantzig 1957). Although Orden published the earliest formal paper, Dantzig wrote about the shortest path problem as one of several problems that he presented at a conference in 1955. The work of Dantzig and Orden led to a number of different approaches being proposed in the literature. These included the Bellman-Ford Algorithm (Ford 1956, Bellman 1958) and Dijkstra's Algorithm (Dijkstra 1959, Moore 1959).

The *K*-th Shortest Path (KSP) Problem is an extension of the shortest path problem on a network, where the goal is to return the 1st, 2nd, 3rd, …, kth shortest paths that exist from a prespecified origin to a prespecified destination. Initially formulated by Bock, Kantner, and Haynes (1957), good algorithms for solving this problem (for loopless paths) have been developed by Hoffman and Pavley (1959), Yen (1971), and Katoh et al. (1982). A review of this literature can be found in Medrano and Church (2011).

The Near Shortest Path (NSP) Problem is a slight variation of the KSP problem, initially formulated by Byers and Waterman (1984). Unlike the KSP, which returns a ranked list of the k shortest paths on the network (between a specified origin-destination pair), the NSP returns all distinct paths on the network between a specified origin and destination longer than the shortest path within a prescribed percentage threshold, ε, expressed as a decimal. In other words, if the shortest path has length L_{sp}, then the NSP returns all paths of length L_{sp} $(1+\varepsilon)$. One key difference between the NSP and the KSP is that before running the NSP, it is not known how many paths the NSP algorithm will find. Whereas the KSP generates paths in order of length, paths are returned by the NSP simply as they are found.

Carlyle and Wood (2005) developed an algorithm to quickly solve the NSP for loopless paths. This algorithm is based on a depth-first-search (DFS) routine, which adds nodes that belong to a NSP to a stack data structure until the destination is reached. To find the next NSP, nodes are popped from the stack until a deviation is found which results in a different NSP. This continues until all NSPs have been found. While this algorithm has an exponential worst-case complexity, this only occurs in the most pathological of examples. Carlyle and Woods' computational results on both random graphs and road networks found that this method could find sets of NSPs very quickly, and by sorting the output, this method also solved the KSP problem faster than previous KSP algorithms.

While the NSP algorithm is quite fast, the fundamental combinatorial nature of the problem makes solving for sets of NSPs on large graph sizes or for a large threshold parameter (ε) a difficult task. On a given graph, as ε increases linearly, the number of NSPs found tend to increase exponentially. Also, as the size of the graph increases, the number of NSPs returned will increase factorially. Even with efficient algorithms, the problem quickly grows to a scale that is difficult to solve with a single computer. The only practical approach to solve a large NSP problem in a reasonable amount of time is to employ parallel computing techniques, which is the subject of this report

In the next section of this report, we provide details that define the network used in corridor planning. This is generally based on the use of raster or gridded data. After the basic network is described, we then describe the NSP algorithm of Carlyle and Wood. We follow that description with an example application which demonstrates the need for developing a parallel algorithmic approach. The remaining sections of this report discuss a simple, naïve parallel approach along with a more sophisticated method that has been developed to increase parallel efficiency.

Raster Graph Data

The efficiencies of graph 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 are added which connect each node to its eight nearest neighboring cells (nodes), four in the orthogonal directions and four in the diagonal directions, which results in a more dense network of $m/n = 4$ (while each node has 8 arcs emanating, every arc connects to two nodes). Goodchild (1977) 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 8. In the extreme case, every node in a network may be connected to every other node, known as a complete graph. In this instance, maximum density is achieved with $m = (n^2 - n)/2$ undirected arcs, so the arc-node ratio is $m/n = (n-1)/2 = O(n)$.

Huber and Church (1985) use a metric called *R* to differentiate between the types of moves allowed on a raster network. $R = 0$ is used to define the case when only orthogonal moves are allowed, $R = 1$ is used to define the case where diagonal moves can be used in addition to orthogonal moves, and $R = 2$ is used to define the case when "knight's moves" are included in addition to $R = 1$ moves. One can think of the value of R representing a buffer width of cells surrounding a target cell. For $R = 1$, case *b* in Figure 1, arcs are directed towards all cells in the ring of cells about the center cell. For $R = 2$, arcs are defined in such a manner that a direct straight line path from a cell to all neighbors in a 2 cell buffer ring about a cell are defined (see Figure 1c). Moving from one level $R = k$ to subsequent level $R = k+1$ requires a doubling of the needed arcs.

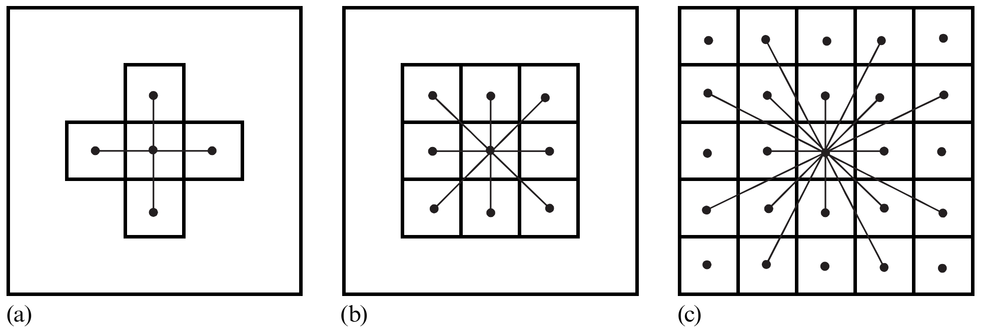

Figure 1 - Interconnectivity metric on raster graphs. (a) *R* = 0, (b) *R* = 1, (c) *R* = 2. Diagram is from Zhang and Armstrong (2008).

Goodchild showed that moving from $R = 0$ to $R = 1$ improves accuracy of distance measurement by 30%, while moving from $R = 1$ to $R = 2$ improves accuracy an additional 7%. Higher order R values can be used, adding even more arcs to the network, but Huber and Church found that $R = 2$ provides the most satisfactory trade-off between accuracy and computational burden. This is the metric we used in our research.

Thus far, we have tested our implementation of the NSP algorithm 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 and was originally used by Huber and Church.
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.

Both of these data sets have been used by Huber and Church (1985), Lombard and Church (1993), and by Church, Loban and Lombard (1992).

The program that has been developed as a part of this research project 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.

Carlyle & Wood’s 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 $\varepsilon \geq 0$, an acceptable percentage increase from the shortest route distance between a given origin and destination. The number of such near-short paths is not known in advance and can be identified only when such a search has been completed. 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. (1982) 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 the NSP is far quicker.

In their 2005 paper, Carlyle and Wood present two different algorithms for finding loopless NSPs. The first one, ANSPR0 (Algorithm Near Shortest Paths Restricted 0), 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 (Algorithm Near Shortest Paths Restricted 1), has a better worst-case complexity, but when implemented it tends to run 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 implementation of the Katoh et al. KSP algorithm.

In the following pages, we present a verbal description, as well as a pseudo code 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 $D = (1 + \varepsilon) \times L_{sp}$, and L_{sp} is the 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 acceptable 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 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 $\leq D$ 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 shortest path calculations in the process of generating paths. Also, unlike Yen's KSP Algorithm or Katoh's KSP Algorithm, not having to store a list of candidate path lengths to determine the next-shortest path length saves time and memory. Additionally, we can further optimize the algorithm by trimming unnecessary nodes from the graph in the following manner.

Before initiating the depth first search, Carlyle and Wood use Dijkstra's algorithm, starting at the destination node, to determine the shortest path length $d'(v)$ from each node to the destination. By also solving Dijkstra's algorithm in the forward direction (from the origin node), you can solve the shortest path length from the origin to each node, which we'll call $d''(v)$. With the information from both shortest path trees, one rooted at the origin and the other rooted at the destination, you can now easily determine the shortest path length possible from origin to destination if it is constrained to go through a specific node v. For each node, if $d'(v) + d''(v) > D$, then there cannot exist any NSP that goes through that node v. We can then eliminate that node from the network and any arcs connected to that node, and thereby reduce the overall problem size and memory requirements and speeding up the algorithm runtime.

Overall, this approach is a very streamlined and efficient method of quickly enumerating a large set of paths.

ANSPR0 Algorithm

DESCRIPTION: An algorithm to solve loopless NSP.
INPUT: A directed graph $G = (V, E)$ in adjacency list format, τ , 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 $v \in V$ ) { $d'(v) \leftarrow$ shortest-path distance from $v$ to $t$; }
  $D \leftarrow (1 + \varepsilon)\, d'(s)$;
  **for** (all $v \in V$) { nextEdge($v$) $\leftarrow$ firstEdge($v$); }
  theStack $\leftarrow s$; $L(s) \leftarrow 0$;
  /* $\tau(v)$ denotes whether the vertex $v$ appears on the current subpath. */
  $\tau(s) \leftarrow 1$;
  **for** (all $v \in V - s$ ) { $\tau(v) \leftarrow 0$; }
  **while**( theStack is not empty ) {
    $u \leftarrow$ vertex at the top of theStack;
    **if**( nextEdge($u$) $\neq$ null ) {
      $(u, v) \leftarrow$ the edge pointed to by nextEdge($u$);
      increment nextEdge($u$);
      **if**( $L(u) + c(u,v) + d'(v) \leq D$ and $\tau(v) = 0$) {
        **if**( $v = t$ ) {
          print( theStack $\cup\, t$ );
        } **else** {
          push $v$ on theStack;
          $\tau(v) \leftarrow \tau(v)+1$;
          $L(v) \leftarrow L(u) + c(u, v)$;
        }
      }
    } **else** {
      Pop $u$ from theStack;
      $\tau(u) \leftarrow \tau(u)-1$;
      nextEdge($u$) $\leftarrow$ firstEdge($u$);
    }
  }
}
```

The Need for Parallelization

Characterizing Problem Size Growth

It is important to first establish the need for a parallelized approach to the NSP algorithm. Before we considered any parallelization scheme, we wrote a JAVA implementation of the NSP algorithm. We ran tests on both networks (*i.e.* 20x20 and 80x80 raster defined networks) for numerous values of ε to see how the number of paths increased as we increased the threshold value ε. Figure 2 displays these results in a linear plot for the 20x20 network, and Figure 3 displays them on a logarithmic plot.

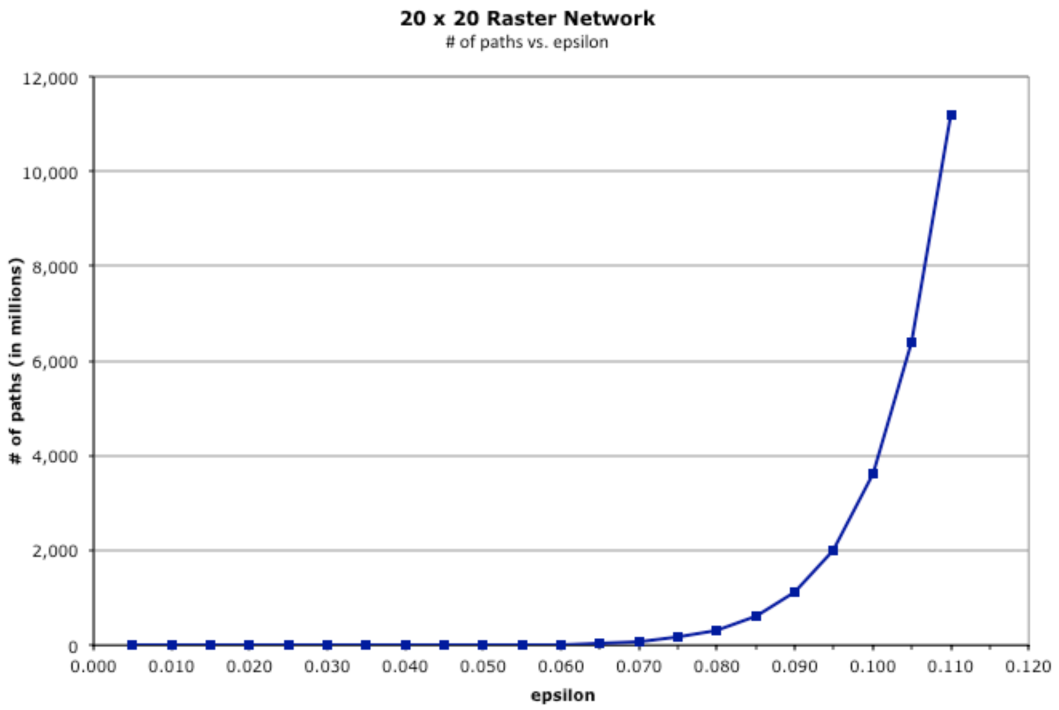

Figure 2 - 20x20 network, number of paths generated by the ANSPR0 vs. epsilon

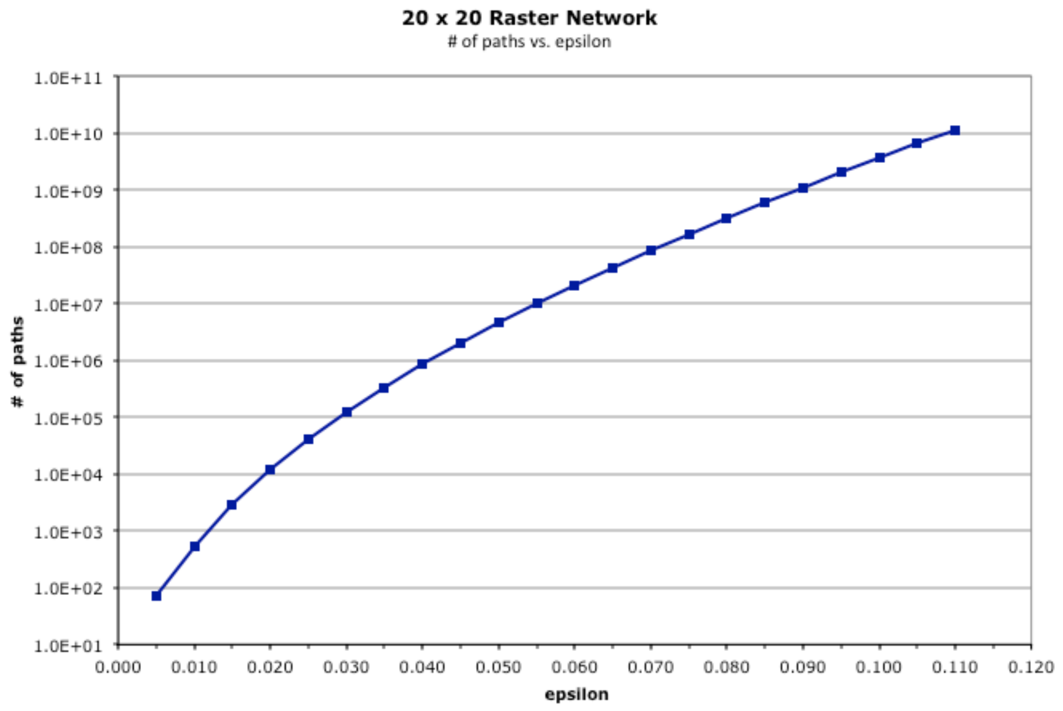

Figure 3 - 20x20 network, log number of paths generated by ANSPR0 vs. epsilon

From figure 2, we can see that the Near Shortest Path Algorithm generated nearly 4 billion solutions on the 20x20 raster region when the epsilon value was set at 0.10. This means that there exist nearly 4 billion paths that had a length that was within 10 percent of optimal path. In Figure 3, after an initial ramp-up, the curve flattens out into a straight line, suggesting an exponential growth rate in the number of paths generated as a function of epsilon. Eventually, we would expect the curve to flatten out as we approach the condition of finding all possible combinations of paths, although the range of epsilon values used is not anywhere close to this boundary or upper limit. We would expect observed trend to continue for epsilon values of at least a couple orders of magnitude higher than the values that we tested.

We ran the same computational experiment for the 80x80 raster data and R=2 network, 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 raster, the range of epsilons used in our 80x80 experiments were an order of magnitude smaller than those in our 20x20 experiments. For example, for ε =0.005, on the 20x20 data there were 73 near shortest paths, but for the 80x80 there were 510,343,616 such paths.

Figures 4 and 5 display computation time needed as a function of increasing values of ε on the same 20x20 data set. Figure 4 shows that the time to compute all paths within a threshold increases at a rapid rate as ε increases. Figure 5 plots the data on a logarithmic y-axis, and shows a straight line trend suggesting that this growth is indeed exponential in character.

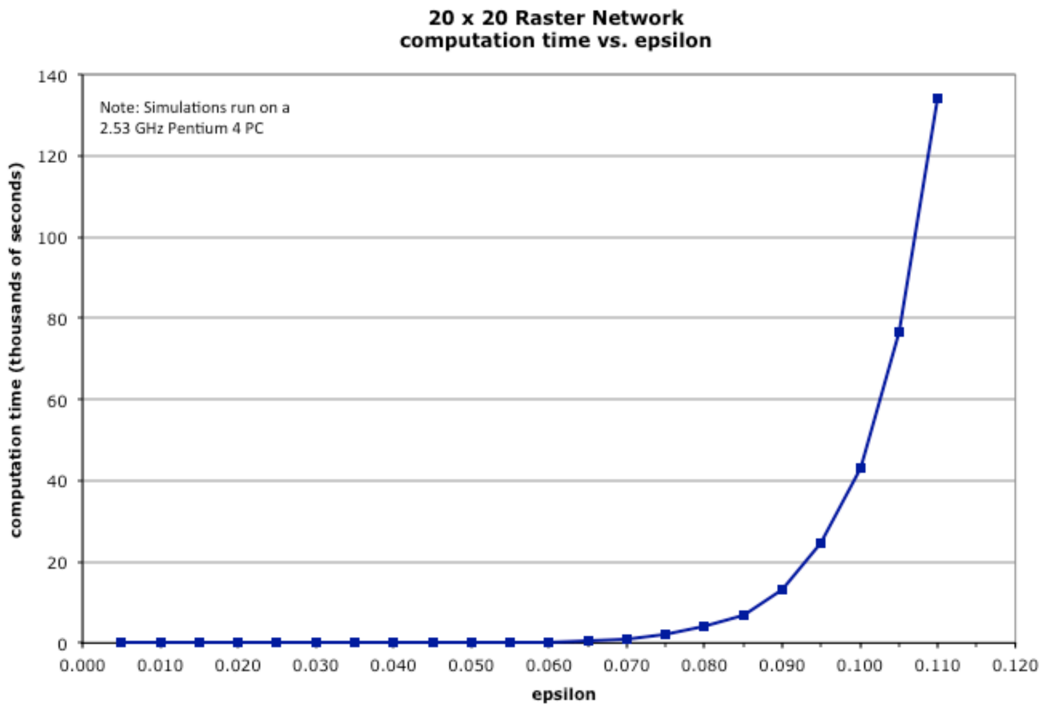

Figure 4 -20x20 network, computation time vs. epsilon

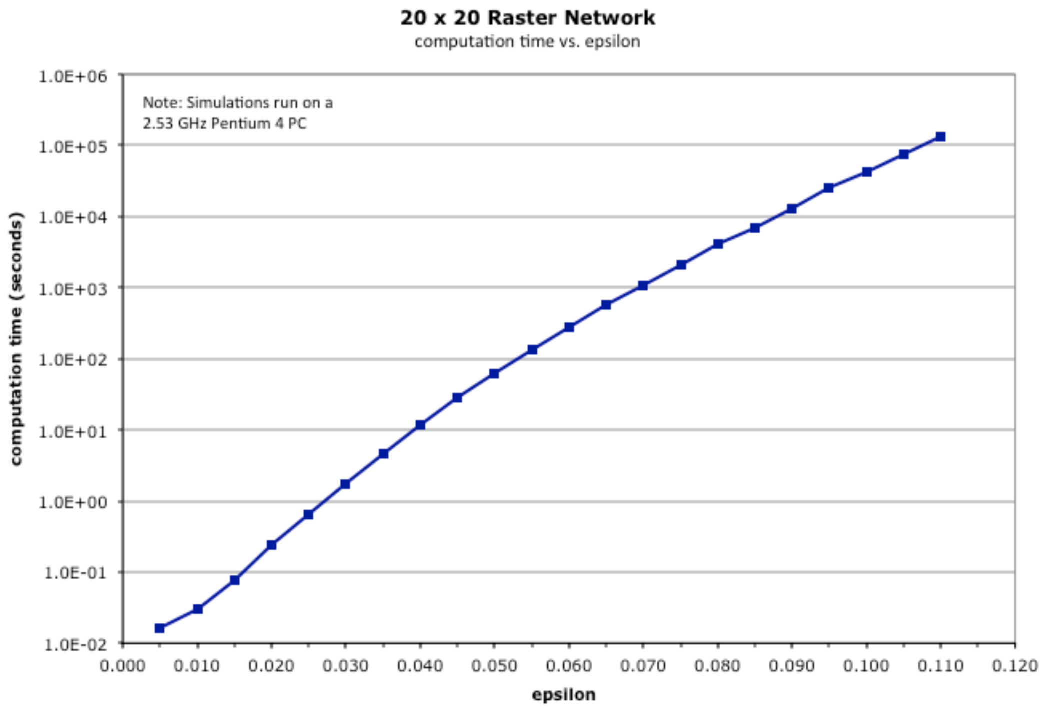

Figure 5 -20x20 network, log computation time vs. epsilon

Problem Size Conclusions

Generating a set of Near Shortest Paths can be an enormous task and may overwhelm computational resources as we increase the network size or increase the value of ε. Generating all paths within 0.75% of the shortest path length on the 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 on a serial JAVA implementation. While we make no claims that the serial code cannot be further optimized (indeed, since running these tests, we have been able to improve our program's speed by approximately 5%), 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. To even consider solving such a problem to completion would require the use of parallel computing techniques and a large number of processors.

Parallelizing Depth-First Search

The remainder of this report discusses two approaches to parallelizing the near shortest path algorithm. Prototyping was done using UCSD's Triton Supercomputer. Triton consists of 256 gB222X Appro blade nodes, each containing 2 quad-core Intel Nehalem 2.4 GHz processors, 24 GB of memory, and is capable of a peak processing power of 20 TeraFlops. Our code was written in C++ using the MPI extension to communicate between the different processors/nodes.

Our initial technique of converting the NSP algorithm into a parallel algorithm was to begin with a breadth-first-search (BFS) on all NSPs emanating from the origin point. This approach naturally results in a tree structure, with concurrent paths sharing branches until they deviate, and the end nodes of the paths found as the leaves on the trees. The BFS paths are stored in a "trie" data structure (Aho *et al.* 1983, Hadjiconstantinou and Christofides 1999). The BFS runs until there are as many leaves on the tree as there are processors available for computation, at which point each processor is then tasked with running the DFS NSP algorithm using the leaf as its starting point, finding all paths from that point which are less than the threshold length minus the path-length to the leaf starting node.

Because we restricted our BFS to only nodes that are guaranteed to have at least one NSP on it, and because the number of nodes with NSPs on them varies as a function of the threshold ε, the number of leaves at each level of the BFS tree varied also as a function of ε. The chart given in Figure 6 plots the number of tree leaves as a function of the number of BFS levels, showing that the leaves grow somewhat exponentially per level (as expected). Each curve on this figure is associated with a specific value of epsilon. Note we have also listed as a point of reference the number of processors employed in the Argonne National Laboratory supercomputer called Intrepid. In Figure 7, there is a general flattening of the curves between levels 6 and 8. This is due to the presence of a small impenetrable barrier on the 20x20 raster at this distance from the origin, showing that the topological features of the data itself have an influence on the rate of growth, in addition to the ε parameter.

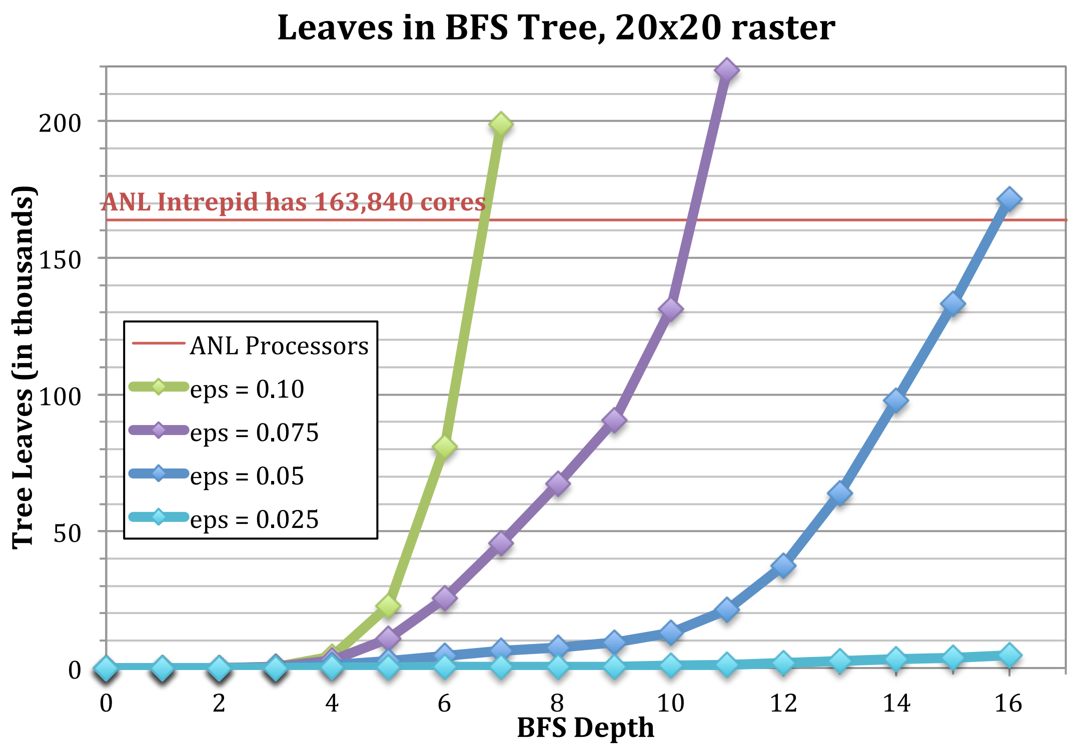

Figure 6 - Leaves in BFS Tree, 20x20 raster, epsilon = 0.05

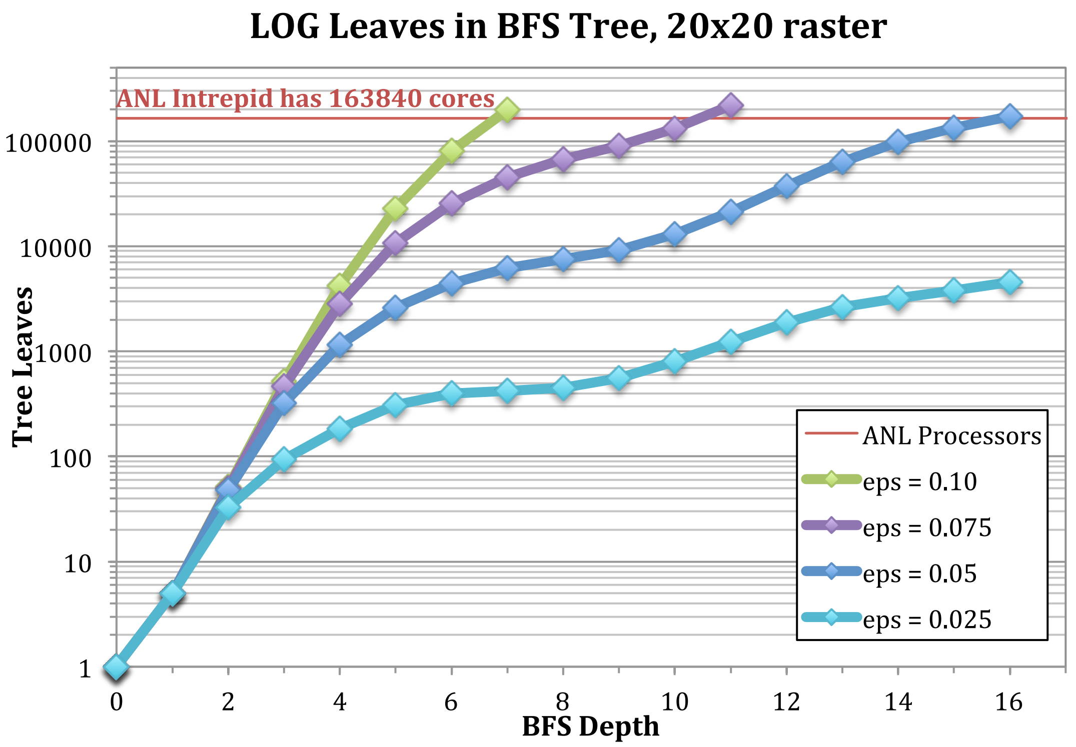

Figure 7 - LOG Leaves in BFS Tree, 20x20 raster, eps = 0.05

For a period of time, we considered storing the BFS paths as a Directed Acyclic Graph (DAG). A DAG structure allows paths to diverge and rejoin, so that a single leaf can represent a set of paths with the same possible suffixes. In a DAG, every node appears only once in the graph, as opposed to nodes appearing numerous times in a tree structure. By doing this, we thought we could possibly run a single NSP thread for more than one path prefix at a time, theoretically giving us the potential for a super-linear speedup. Although this line of reasoning seemed promising, we found that this would not work, because different prefixes required blocking out different nodes as being "already added to the stack". Therefore, what would be a possible candidate for one prefix might not be a feasible candidate for another prefix. While the thought of super-linear speed-up was enticing, we had to accept that our approach for achieving this would not produce a correct algorithm.

Analysis of Implementation

Table 1 below shows some data collected from various runs of our first parallel code implementation on the 20x20 data set on the Triton nodes. The columns show epsilon value, number of processors, total NSP runtime in seconds, total paths found, paths found on the leaf of fewest paths (Min Paths Leaf), paths found on the leaf of most paths (Max Paths Leaf), speedup, and parallel efficiency.

Epsilon	Number of Processors	Time (sec)	Total Paths		Min Paths Leaf	Max Paths Leaf	Speedup	Parallel Efficiency
0.05	1	21.73	4,601,053	paths	4,601,053		1.00	1.00
				time	21.729			
				paths/sec	211,747			
0.05	5	7.07	4,601,053	paths	247,446	1,530,887	3.07	0.61
				time	1.30064	7.07048		
				paths/sec	190,249	216,518		
0.05	48	2.51	4,601,053	paths	11	565,901	8.67	0.18
				time	0.000195	2.50676		
				paths/sec	56,403	225,750		
0.07	1	392.37	86,384,393	paths	86,384,393		1.00	1.00
				time	392.373			
				paths/sec	220,159			
0.07	5	119.94	86,384,393	paths	5,782,131	26,620,106	3.27	0.65
				time	27.3463	119.939		
				paths/sec	211,441	221,947		
0.07	50	34.39	86,384,393	paths	137	8,129,092	11.41	0.23
				time	0.00161	34.3939		
				paths/sec	85,091	236,353		

Table 1 - 1st Parallel Algorithm Runtime Results on the 20x20 data

In the computational results shown in Table 1, we see that good parallel efficiency was achieved when the ratio between the maximum number of paths found on a leaf and the minimum number of paths found on a leaf is not too great. For example, when ε = 0.05, and 5 processors and threads were employed (1 BFS level), the ratio was approximately 6:1 "max to min paths". This resulted in a very respectable 0.61 value of parallel efficiency. With epsilon set at 0.05 and when employing 48 processes though (BFS level 2), the "max to min paths" ratio was 50,000:1. The min leaf quickly finished its work in 0.2 milliseconds, while the max leaf took 2.5 seconds to complete. This significant amount of relative idle time resulted in a much worse parallel efficiency of 0.18.

Table 2 gives results for computational tests on the 80x80 data using the same parallel implementation. This experiment produced even larger discrepancies between the number of paths found in the "max" sized leaf and the number of paths found in the "min" sized leaf, resulting in a truly abysmal speedup of 1.61 when using 38 processors, which is equivalent to a parallel efficiency of 0.04. Note here that, when using multiple processors, the closer the value of parallel efficiency is to 1.00 the better. By definition, when one uses only one processor, it will be rated at 100% efficiency for that one processor. The main objective in parallelizing a routine is to use all processors efficiently with no idle time and reach a parallel efficiency of 1.0 overall.

Epsilon	Number of Processors	Time (sec)	Total Paths		Min Leaf Path	Max Leaf Paths	Speedup	Parallel Efficiency
0.003	1	33.21	4,459,050	paths	4,459,050		1.00	1.00
				time	33.21			
				paths/sec	134,253			
0.003	5	25.51	4,459,050	paths	3,462	3,475,928	1.30	0.26
				time	0.03324	25.5127		
				paths/sec	104,152	136,243		
0.003	11	25.49	4,459,050	paths	852	3,472,466	1.30	0.12
				time	0.01286	25.4856		
				paths/sec	66,251	136,252		
0.003	12	25.50	4,459,050	paths	852	3,472,466	1.30	0.11
				time	0.01262	25.5003		
				paths/sec	67,501	136,174		
0.003	14	24.29	4,459,050	paths	600	3,462,254	1.37	0.10
				time	0.00817	24.2906		
				paths/sec	73,475	142,535		
0.003	22	21.95	4,459,050	paths	300	3,175,358	1.51	0.07
				time	0.00408	21.9474		
				paths/sec	73,511	144,680		
0.003	38	20.68	4,459,050	paths	216	3,033,530	1.61	0.04
				time	0.00527	20.68		
				paths/sec	41,018	146,714		

Table 2 - 1st Parallel Algorithm Runtime Results on the 80x80 data

Due to the independent computation of each leaf, we found that that the expected overall parallel efficiency follows this relationship:

$$parallel\ efficiency = \frac{Paths_{Total}}{p \times Paths_{Max}}$$

where $Paths_{Total}$ is the total number of paths found for the given input parameters and data, and $Paths_{Max}$ is the maximum number of paths found by one processor, and p is the number of processors used. Additionally, as $Paths_{Max} \rightarrow P_{Total}/p$, then $parallel\ efficiency \rightarrow 1$. This is essentially an example of Amdahl's Law (Amdahl 1967) in action, which states that the potential parallelism available in any program is limited by the amount of work that must be run sequentially. Clearly, there is a need to try to distribute the work more evenly in order to make the most efficient use of all processors.

Distributing Workload

The load imbalances in this problem come from performing depth-first search on a raster network. This is due to the fact that task costs are completely unknown until after execution is completed. Therefore, offline partitioning or scheduling algorithms cannot be used beforehand, as there is not enough information available in order to make use of such schemes.

In this project we considered several existing methods for a load balancing. The following is a description of these methods and how well they would apply to our NSP path problem.

Approach #1 – Randomized Task Distribution

As it stood before, the code distributed the work by running a BFS algorithm until the tree had as many leaves as there were processors, then assigned one leaf to each processor for it to run to completion. The drawback was that some leaves contained far more work than others, resulting in lots of idle processor time for some of the processors.

A randomized task distribution approach would be based on generating far more leaves than processors, then assign these tasks randomly to each processor. By randomly distributing sufficient work chunks of unknown size to numerous processors, the hope is that overall work for each processor average out to be somewhat similar. Adler et al. (1995) show that when using randomized algorithms, in order to get a "good" balance one must generate at the very least p log p tasks, where p is the number of processors.

Pros of this approach:
- Easy to implement

Cons of this approach:
- No guarantee of perfect distribution
- In a worst-case-scenario, a large outlier could still result in an overall work imbalance

Approach #2 – Dynamic Centralized Scheduling

Centralized scheduling uses an as-needed approach for assigning tasks. Like randomized task distribution, centralized scheduling first generates a list of tasks (>> *p*), then assigns the first *p* tasks to the various processors to compute. When a processor completes a task, it asks the scheduler for another task. The scheduler assigns a new task, removes it from the list, and this process would continue until all tasks have been assigned and computed.

Pros of this approach:

- A little more difficult to implement than option #1, but still relatively easy to code

Cons of this approach:

- Susceptible to the possibility of an abnormally large task being assigned last
- If communication is expensive, then this could be slow

Approach #3 – Dynamic Work Stealing

Dynamic work stealing is an approach that assigns all work to all processors at the start; then when one processor completes its tasks, it steals part of a task from another processor in order to have more work to do. This approach is certainly viable for a DFS algorithm; and if one does not consider communication time between processors, it has the possibility of producing the best theoretical results. Unfortunately, is far more difficult to implement than the other options. There are also many approaches one can take in selecting which processor to steal work from, including asynchronous round robin, global round robin, and random polling/stealing. It has been proven that a random polling/stealing approach is just as effective as the other two approaches, which is ideal because it is also easier to implement. In our case, we could use a worker queue that at each time assigns work from the processor at the front of the queue. Any time a processor steals work or gets stolen from, it then gets placed at the back of the queue, ensuring it won't get stolen from immediately afterward.

Pros of this approach:

- Best theoretical worst-case scenario

Cons of this approach:

- Far more difficult to implement than approaches #1 and #2
- If communication is expensive, then this could be slow

Approach #4 – Combined Approach

The best way to use the strengths and hide the weaknesses of any approach is to combine it with another complimentary approach. For example, a hybrid randomized distribution / work stealing approach could show promise in giving a relatively even initial work distribution from the randomized approach, then leveling things out toward the end using dynamic work stealing. It's easy to see though that any hybrid approach would be the most difficult to implement, as it requires developing several approaches, as well as integrating them to cooperate together.

Existing Approaches: Conclusions

After considering the options, we were not quite satisfied with any of them for our application. The first two approaches suffered from major drawbacks due to the possibility of large work chunks superseding the benefits of random prescheduling or dynamic work scheduling. Efficient dynamic work stealing appears to be promising, but is outside the realm of our programming expertise at this moment. Future collaborations with expert parallel programmers could incorporate dynamic work stealing to aid in the workload distribution. Without this as an option yet, what we needed was a way to be able to predict the amount of work that would be generated from each leaf of the BFS tree. That would enable the program to either break down those portions into smaller pieces, or use a pre-scheduler to optimally distribute the work evenly. Further, such information could potentially be useful in any of the parallel strategies listed above.

Analysis of Naïve BFS Work Distribution

Before being able to predict the variation of work distribution, we analyzed what that distribution was when using the simple BFS tree approach. We ran the NSP algorithm numerous times to completion, each time running the BFS component to a different number of levels, and evaluated how many paths were generated from each leaf of the BFS tree at each given level.

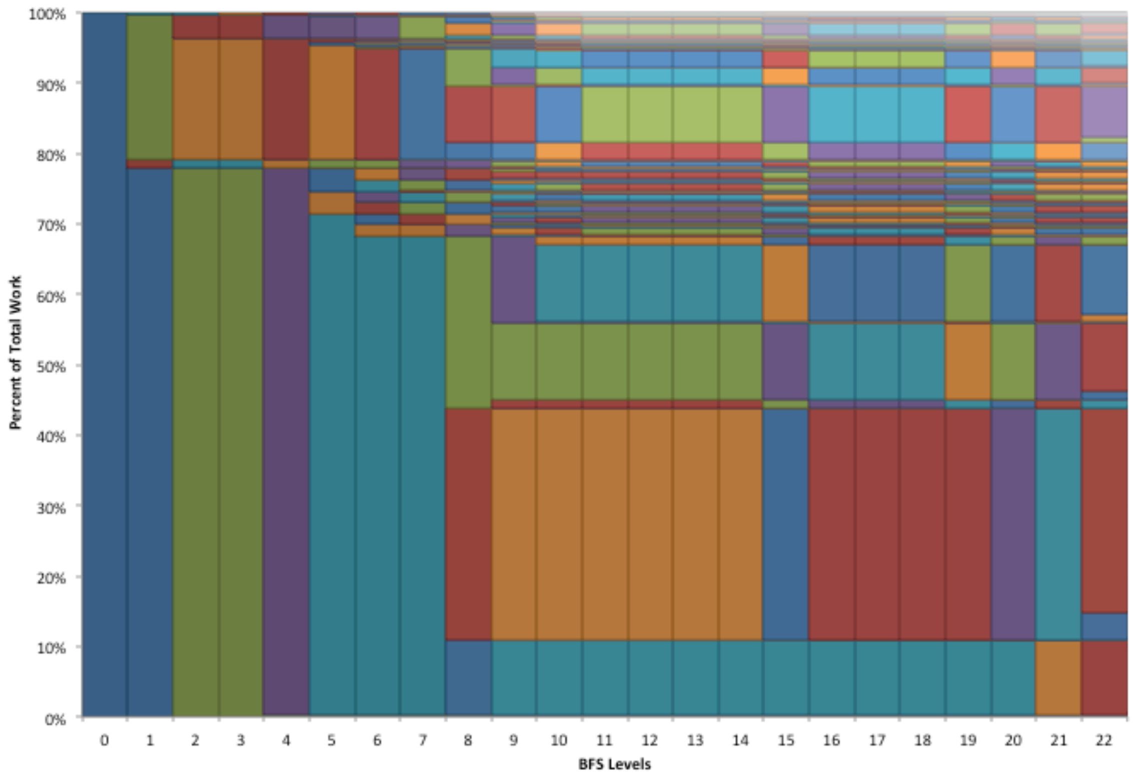

Figure 8 - Work Distribution for Varying BFS Tree Levels

Figure 8 shows the work distribution for varying BFS tree levels. With zero levels, 100% of the work is done by one process, with 1 level, there are 5 leaves, but one leaf accounts for almost 80% of the work, another for about 20% of the work, with just a tiny amount of work remaining for the three other leaves. As the BFS tree propagates, all of the chunks continue to shrink, some more quickly than others. While this visualization is useful at seeing how the tree physically propagates, another useful way to view the data is to sort the work chunks by size. Figure 9 is the same as Figure 8, but with the work of each leaf for each level breakdown is sorted by size.

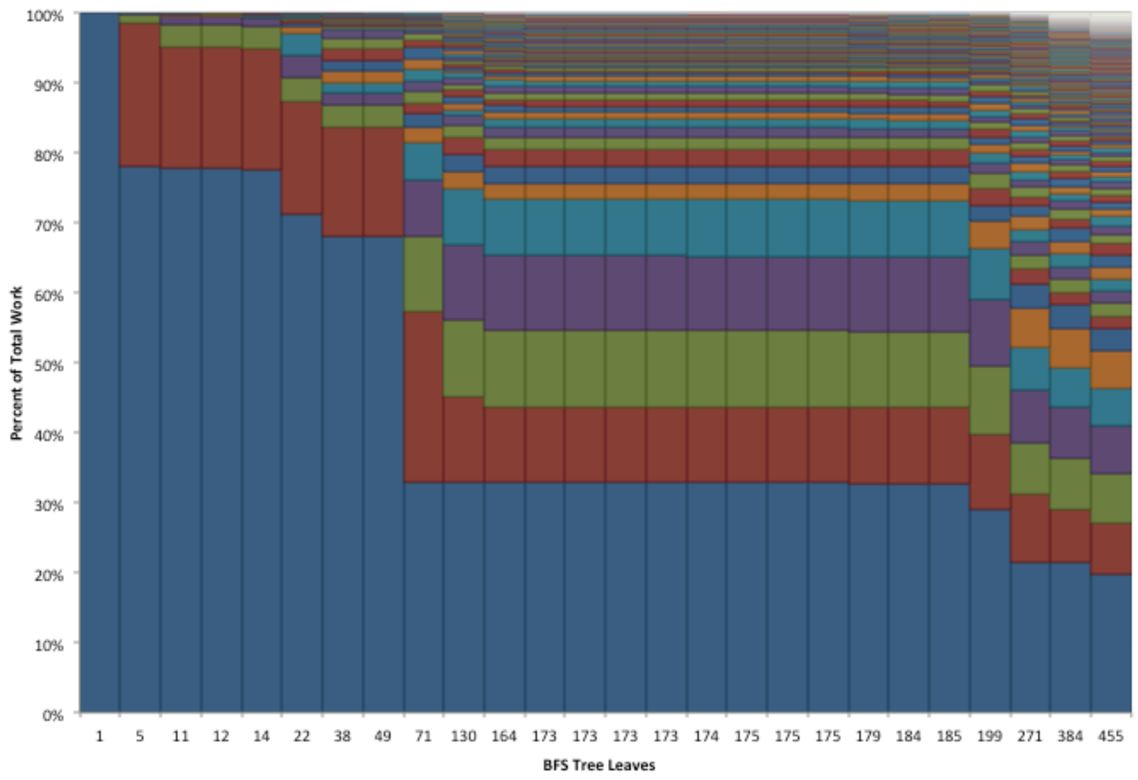

Figure 9 - Sorted Work Distribution for Varying BFS Tree Levels

The x-axis, rather than labeled by level, now shows the number of leaves in that particular level. The each column from left to right still represents one more level of depth in the BFS tree. For each column, the components of a column of the chart in Figure 9 represent the work in each leaf sorted from smallest (at the top of the column) to the largest (at the bottom of the column). This presentation helps to convey an idea of how the largest chunks (i.e. the limiting chunks) break down over time, as well as give an idea of how the overall distribution of work among the leaves break out. The final level is level 25, with 455 leaves. Even with so many leaves, there exists 1 leaf that accounts for approximately 20% of the work. This points to a need for finding a way to generate fewer miniscule leaves, and at the same time break down the larger leaves before processing.

Workload Prediction

It was clear that what was holding back the ability to generate an even workload distribution was the inability to predict how much work would emanate from a particular leaf of the BFS tree. In response to this need, we developed a model that predicts how many paths will emanate from a particular leaf of the BFS tree.

Let $d'(i)$ be the shortest path length from node i to the destination node t. Thus the shortest path length from the origin s to the destination t can be represented by $d'(s)$.
Recall that the NSP algorithm finds all paths of length $\leq D$ on the network, where $D = (1 + \varepsilon) \times d'(s)$. Let $L(i)$ be the path length along the BFS tree from the origin node s to node i (which may

not necessarily be the shortest path from s to i). Then for any node i on any NSP, the following invariant is always true:

$$L(i) + d'(i) \leq D$$

Now define the slack to be how much "wiggle room" we have left to generate non-optimal paths from any leaf of the BFS tree. Initially, at the origin, $slack(s) = \varepsilon \times d'(s)$. This is the total amount of slack available for any path to be considered a near shortest path. For any point i along a near shortest path, the slack can be calculated as

$$slack(i) = D - L(i) - d'(i)$$

This value gives a measure of the amount of distance deviation from the shortest path the remaining path branches emanating from i are allowed to have, and yet still be counted as an NSP. Given a particular number of BFS levels, we were able to calculate the slack value for all leafs of the BFS tree at that point, then run the algorithm to completion to see how many paths were generated from each of those leaves. Figure 10 below shows these values plotted against each other for a 23 level BFS on the 80x80 dataset. Note that for this data set and ε value, $slack(s) = 0.791664$.

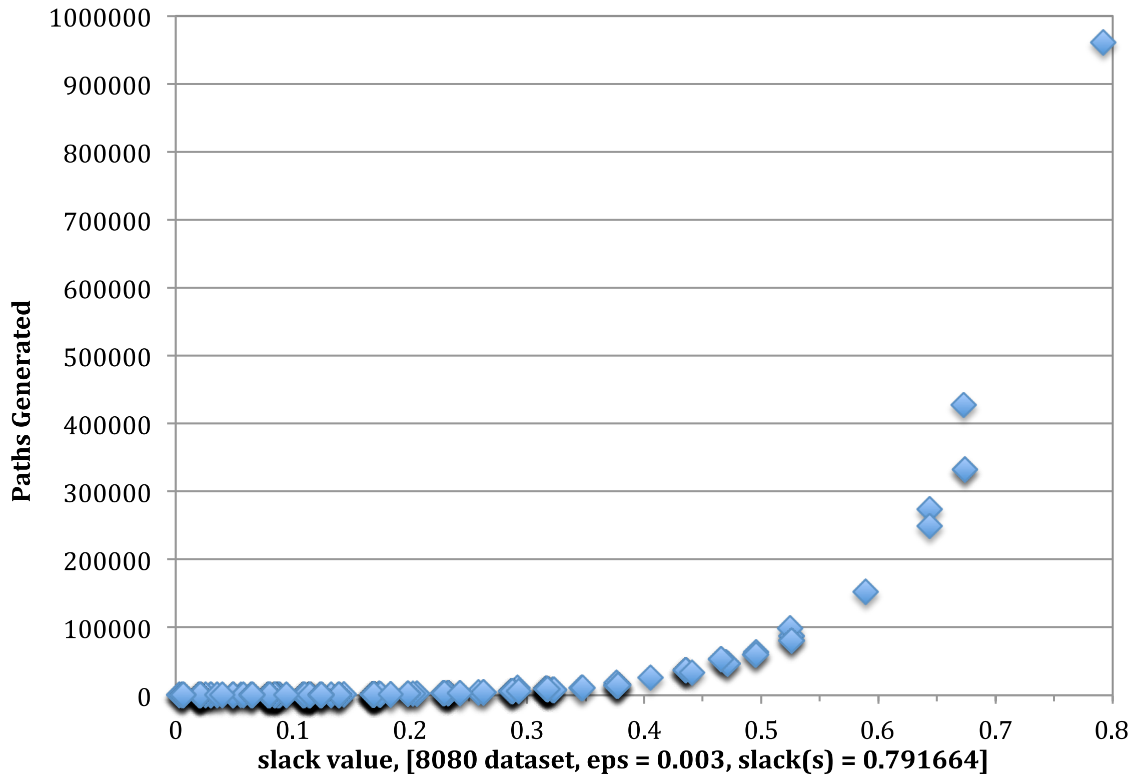

Figure 10 - Paths Generated vs. Slack Value, 23 BFS levels

What we can observe here is that there is a strong relation between the slack value of a leaf and the number of paths generated from that leaf. This same curve shape was found for other BFS levels as well. Since there appears to be a strong relationship between slack and the number of alternate paths generated from that leaf, then this means that the slack value can be used as a predictor for which leaves will generate more work than others.

Threshold BFS Expansion

Using the slack value information, we modified the BFS tree expansion to expand only nodes with a higher than average-expected path count. Rather than expanding all leaves at each level, we selected a threshold value as a cutoff, expanding only BFS leaves that had a higher slack value than the cutoff. To generalize the cutoff value, we defined a normalized slack as:

$$0 \le normalized\ slack = \frac{slack(i)}{slack(s)} \le 1$$

This normalization allows us to have a slack range between 0 and 1. Originally, with the BFS expansion, all leaves on the BFS tree had the same depth. This new BFS expansion results in a BFS tree where the branches have different depths, essentially fathoming once they have a normalized slack value below the cutoff value. If we again plot the sorted work distribution as the BFS tree iterates through a new distribution of work emerges, as depicted in Figure 11.

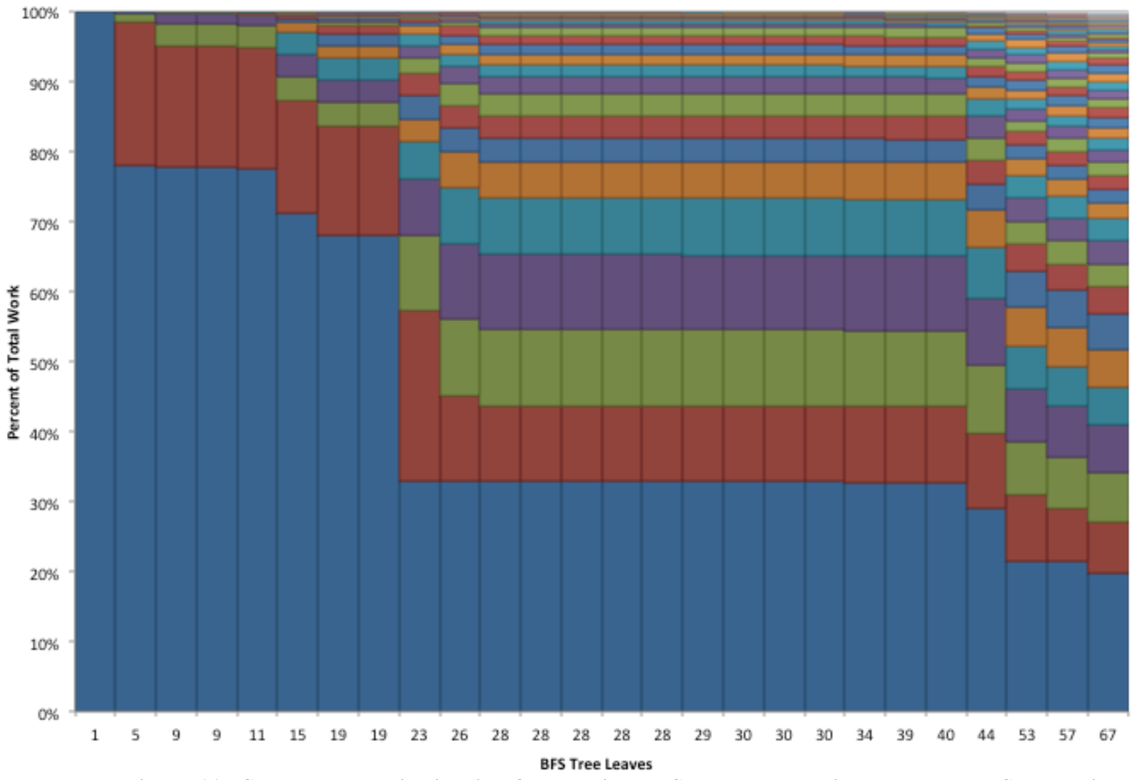

Figure 11 - Sorted Work Distribution for Varying BFS Tree Levels with threshold BFS expansion

As expected, the large work chunks remain the same as the naïve BFS expansion, as those are split up the same as before. The difference is that the small chunks are not further divided. This results in far fewer chunks after the same 25 BFS expansions (67 leaves in this case vs. 455

before), allowing the tree to develop far deeper when generating the same number of leaves, and breaking up the larger chunks while not wasting time breaking up the small ones. As an example, in Figure 11, the largest work chunk at level 25 contains about 20% of the processing work, yet only 67 leaves/threads have been created, whereas in Figure 9, after 25 levels the largest work chunk still contains about 20% of the total processing work, yet 455 leaves/threads have been created. If our goal was to generate 450 threads, we could continue developing the 67 leaf tree, and likely reduce the largest work chunk to below 20%.

One aspect that must be considered in this threshold strategy is the selection of a threshold value. For example, Figures 12 and 13 depict the number of paths from each leaf vs. normalized slack value, with the tree developed to 85 levels on the 80x80 dataset. The first plot, given in Figure 12, shows the results when using a normalized slack threshold 0.7, and the second plot in Figure 13 is associated when using normalized slack threshold of 0.8. The first graph (Figure 12) shows that 23,882 leaves were generated, and the maximum number of paths from any one leaf is 235,266. The second graph (Figure 13) shows that 10,620 leaves were generated; yet the maximum paths from one leaf are still 235,266. In essence, picking the lower threshold generated a tree over twice as large, thereby using much more memory; yet still had the same limiting maximum work chunk size as what was given by the higher threshold. At this point, we have not generated any guidelines on how to select an optimal threshold value, but this is certainly something that is worth exploring in the future.

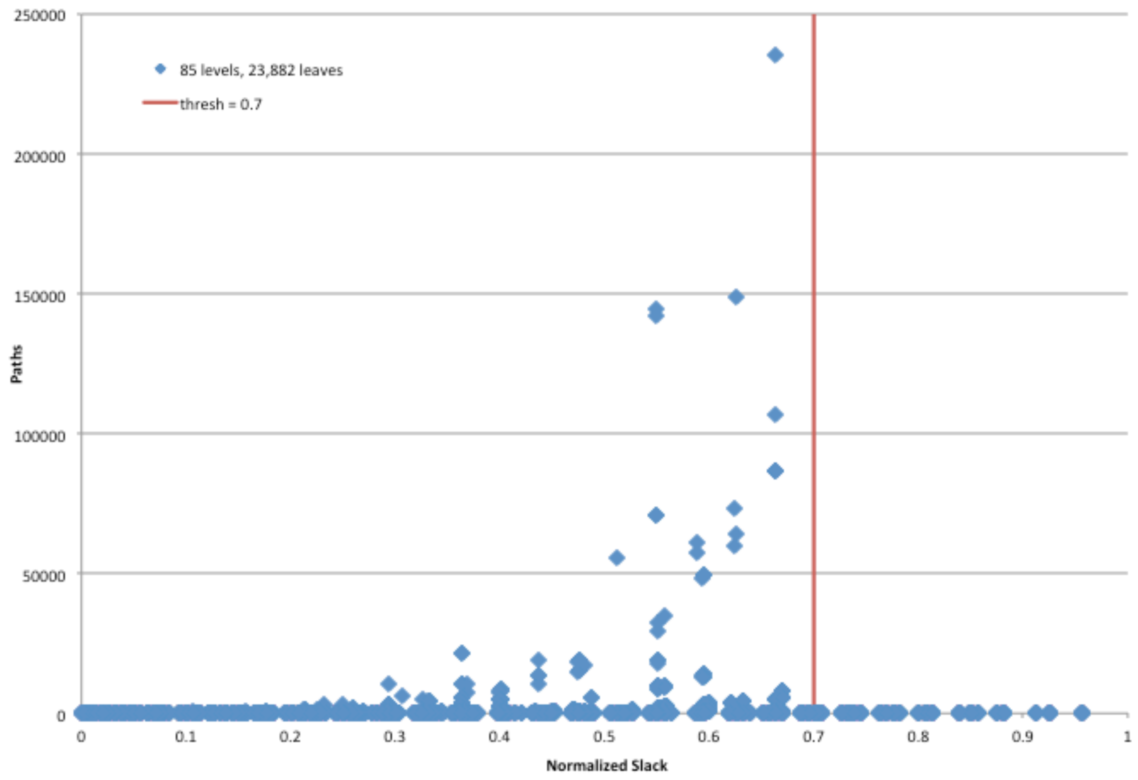

Figure 12 - 8080 data, epsilon = 0.003, 85 level BFS, norm threshold = 0.7

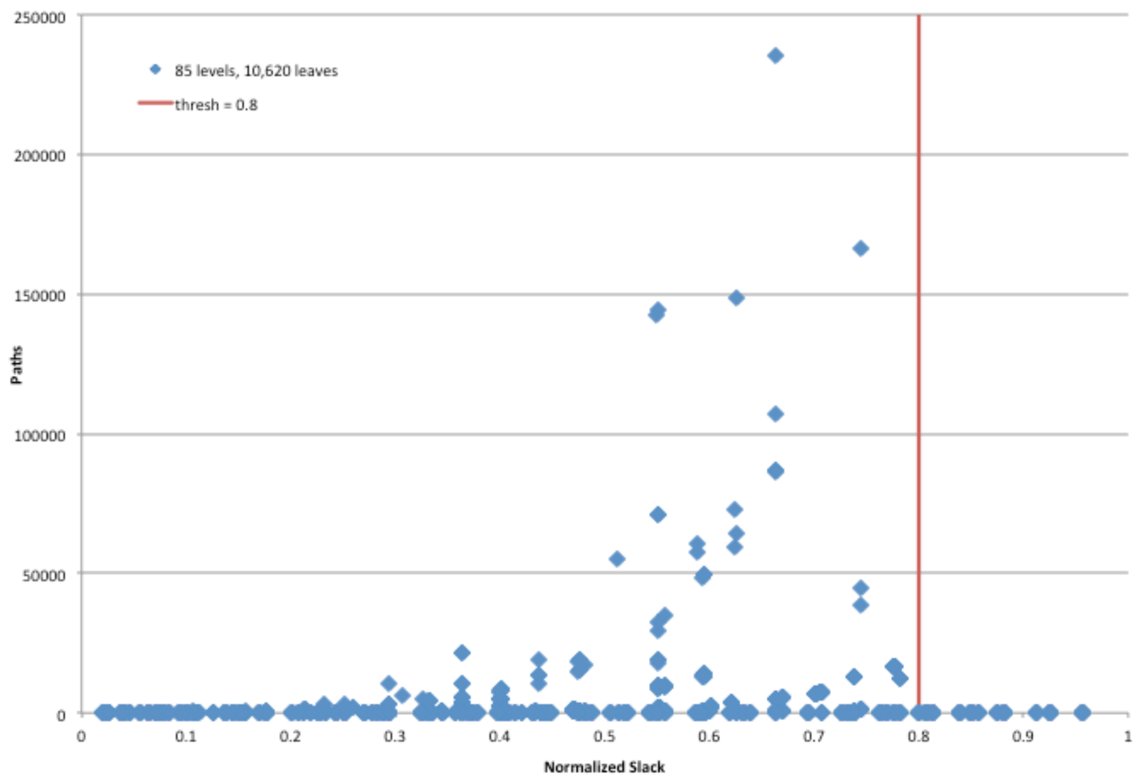

Figure 13 - 8080 data, epsilon = 0.003, 85 level BFS, norm threshold = 0.8

Tree Trimming BFS: Computational Results

We applied the new tree trimming BFS Near Shortest Path algorithm on a computer with 8 processors, expanding the BFS tree to 6 levels using a normalized trimming threshold of 0.8. This test ran with 8 processors in 20.722 seconds, only 0.042 seconds slower than the naïve 6-level BFS that used 38 processors. By achieving the same runtime and speedup with far fewer processors, we were able to improve the parallel efficiency from 0.04 to 0.20, which is a substantial gain in efficiency.

Further improvement can be generated when the BFS tree is grown to a larger depth. When run to 25 levels (and 67 leaves), while using 8 processors, the computation was completed in 7.587 seconds, producing a speedup of 4.378 and a parallel efficiency of 0.547. While still not perfect in parallel efficiency, this example demonstrates how this tree splitting approach can very effectively distribute computational work more evenly based upon normalized path slack.

Conclusions

This project set out to develop a parallel implementation of a near shortest paths algorithm. The original approach split the work up in a pleasingly parallel fashion, but because of large variances in the work-chunk sizes, most processors spent much of their time sitting idle, and overall performance suffered accordingly. We developed a new approach in splitting up the work, devising a predictive metric that could be used to estimate the work on each portion of the BFS tree, and to then expand the tree in portions with high-expected amounts of work. This approach succeeded in returning a more consistent set of work-chunks, resulting in improved overall performance of the parallel code.

It is important to note that work remains in exploring the properties of this new approach, including developing guidelines for optimal threshold values, and determining how far one should develop the tree before the time to calculate a new BFS level outweighs the benefits from further splitting up the work. These are issues that will be considered in order to fully develop the theory behind this approach.

Overall, the preliminary results generated here show tremendous promise for using the near shortest path algorithm in a parallel mode for large networks. Although further testing and analysis is required in order to fully develop and tune this approach, all results thus far point to this being effective at improving the division of work, resulting in better speedup and parallel efficiency. The transmission corridor location problem is a complex and controversial problem in a public setting. The development reported in this document is one of the tools that are envisioned to play a key role in corridor planning, by generating both fine-scale and gross-scale alternatives that ensure all competing close to optimal alternatives are generated.

Acknowledgements

We would like to thank the Environmental Sciences Division of Argonne National Laboratories for providing the funding to conduct the initial part of this research under Project #301CR. We would also like to acknowledge John Krummel for his encouragement and guidance.

Bibliography

Adler, M., S. Chakrabarti, M. Mitzenmacher & L. Rasmussen, (1995). Parallel randomized load balancing. *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing.* Las Vegas, Nevada, United States: ACM, 238-247.

Aho, A.V., J.E. Hopcroft & J. Ullman, (1983). *Data structures and algorithms,* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Amdahl, G.M., (1967). Validity of the single processor approach to achieving large scale computing capabilitiesACM, 483-485.

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*.

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.

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.

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

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

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

Goodchild, M., (1977). An evaluation of lattice solutions to the problem of corridor location. *Environment and Planning A,* 9, 727-738.

Hadjiconstantinou, E. & N. Christofides, (1999). An efficient implementation of an algorithm for finding k shortest simple paths. *Networks,* 34, 88-101.

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.

Katoh, N., T. Ibaraki & H. Mine, (1982). An efficient algorithm for k shortest simple paths. *Networks,* 12, 411-427.

Lombard, K. & R. Church, (1993). The gateway shortest path problem: Generating alternative routes for a corridor location problem. *Geographical Systems,* 1, 25-45.

Medrano, F.A. & R.L. Church, (2011). *Transmission corridor location: Multi-path alternative generation using the k-shortest path method.* Santa Barbara: Geotrans.

Moore, E., (1959). The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching*, Harvard University, 285-292.

Orden, A., (1956). The transhipment problem. *Management Science,* 2, 276-285.

Yen, J.Y., (1971). Finding the k shortest loopless paths in a network. *Management Science,* 17, 712-716.

Zhang, X.D. & M.P. Armstrong, (2008). Genetic algorithms and the corridor location problem: Multiple objectives and alternative solutions. *Environment and Planning B-Planning & Design,* 35, 148-168.