UNIVERSITY OF CALIFORNIA
Santa Barbara

Corridor Location: Generating Competitive and Efficient Route Alternatives

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Geography

by

F. Antonio Medrano

Committee in charge:

Professor Richard L. Church, Chair

Professor Keith C. Clarke

Professor Michael F. Goodchild

Professor John R. Gilbert

December 2014

The dissertation of F. Antonio Medrano is approved.

_____
Michael Goodchild

_____
Keith Clarke

_____
John Gilbert

_____
Richard Church, Committee Chair

December 2014

Corridor Location: Generating Competitive and Efficient Route Alternatives

ACKNOWLEDGEMENTS

I would like to sincerely acknowledge and thank my advisor Rick Church for mentoring his knowledge and wisdom, John Krummel and Argonne National Laboratory for inspiring the topic of my research and financially supporting my work, my dissertation committee for their comments and guidance, Jack Dangermond for offering fellowships to attend conference where I could present my research to the academic community, my colleagues, my friends, and my family, and the beautiful Santa Barbara region for enriching my life while pursuing my academic endeavors.

Funding support for the research in this dissertation was provided by the Environmental Sciences Division of Argonne National Laboratories (1F-32422). I also would like to acknowledge the high-performance computing resources from the UC Santa Barbara Center for Scientific Computing, which is supported by the NSF MRSEC (DMR-1121053) and NSF CNS-0960316 grants.

PROLOGUE

"In wicked problems, the set of alternatives is too large, too diverse, and too little agreed upon. The decision-maker has difficulty even considering their range, or defining their boundaries. Further, in wicked problems the selection of a solution from among the alternatives involves, to a very large degree, resolution of conflicting goals, a function which, I have argued, should not be done by models.

But models can play a very important role in this process, by sorting out for the decision maker a subset of the alternative actions which he must consider thoroughly, and aiding in the development of his understanding of these alternatives and impacts. ... The role of optimization and modeling is the formulation of alternatives rather than the selection of one of them"

Jon C. Liebman
Some simple-minded observations on the role of optimization in public systems decision-making, (1976) *Interfaces* 6: 102-108.

"Everybody has an interest, and everyone's got an ax to grind. It's a very tough job to balance all of this."

Michael R. Peevey
President, Chino Hills Public Utilities Commission
Los Angeles Times, November 27, 2011
in discussing the Tehachapi Renewable Transmission Project

F. ANTONIO MEDRANO

Curriculum Vitae

EDUCATION

Ph.D. University of California, Santa Barbara, December 2014
Geography, emphasis on Modeling, Measurement and Computation

M.S. University of California, Santa Barbara, March 2009
Media Arts and Technology, emphasis on Multimedia Engineering

B.S. Harvey Mudd College, May 2002
Engineering, emphasis on Systems Engineering, *with honors*

REFEREED ARTICLES AND BOOK CHAPTERS

2014 Medrano, F.A. & R.L. Church, "A Simple Java Parallelization For Generating Biobjective Supported Solutions for a Network Optimization Problem." Submitted for publication, in review.

2014 Scaparra, M.P., R.L. Church & F.A. Medrano, "Corridor location: The multi-gateway shortest path model." *Journal of Geographical Systems* **16**, 287-309.

2014 Medrano, F.A. & R.L. Church, "Corridor location for infrastructure development: A fast bi-objective shortest path method for approximating the pareto frontier." *International Regional Science Review* **37**, 129-148.

2013 Medrano, F.A. & R.L. Church, "A parallel algorithm to solve near-shortest path problems on raster graphs." *In* Shi, X., Kindratenko, V. & Yang, C. eds. *Modern accelerator technologies for geographic information science.* Springer US, 83-94.

RESEARCH REPORTS AND THESES

2012 Medrano, F.A. & R.L. Church, *A new parallel algorithm to solve the near- shortest-path problem on raster graphs.* Santa Barbara: Geotrans Report, RP-01-12-01.

2011 Medrano, F.A. & R.L. Church, *Transmission corridor location: Multi-path alternative generation using the k-shortest path method.* Santa Barbara: Geotrans Report, RP-01-11-01.

2009 Medrano, F.A., *Optical position sensors with applications in servo feedback subwoofer control*. M.S. Thesis, Media Arts and Technology Program, University of California at Santa Barbara.

RESEARCH ASSISTANTSHIPS AND FELLOWSHIPS RECEIVED

2010 University of California Transportation Center Graduate Fellowship, $30,000

2010-2014 Graduate Research Assistantship, funding from Argonne National Laboratories, (Summer – Fall 2010, Fall 2011 – Winter 2013, Summer – Winter 2014)

RESEARCH INTERESTS

Combinatorial Optimization & Network Algorithms, Parallel Computing and Big Data, Spatial Optimization: Linear and Integer Programming, Multi-Objective Optimization, Location Modeling & Analysis, GIS and Geographical Analysis

ABSTRACT

Corridor Location: Generating Competitive and Efficient Route Alternatives

by

F. Antonio Medrano

The problem of transmission line corridor location can be considered, at best, a "wicked" public systems decision problem. It requires the consideration of numerous objectives while balancing the priorities of a variety of stakeholders, and designers should be prepared to develop diverse non-inferior route alternatives that must be defensible under the scrutiny of a public forum. Political elements aside, the underlying geographical computational problems that must be solved to provide a set of high quality alternatives are no less easy, as they require solving difficult spatial optimization problems on massive GIS terrain-based raster data sets.

Transmission line siting methodologies have previously been developed to guide designers in this endeavor, but close scrutiny of these methodologies show that there are many shortcomings with their approaches. The main goal of this dissertation is to take a fresh look at the process of corridor location, and develop a set of algorithms that compute path alternatives using a foundation of solid geographical theory in order to offer designers better tools for developing quality alternatives that consider the entire spectrum of viable solutions. And just as importantly, as data sets become increasingly massive and present

challenging computational elements, it is important that algorithms be efficient and able to take advantage of parallel computing resources.

A common approach to simplify a problem with numerous objectives is to combine the cost layers into a composite a priori weighted single-objective raster grid. This dissertation examines new methods used for determining a spatially diverse set of near-optimal alternatives, and develops parallel computing techniques for brute-force near-optimal path enumeration, as well as more elegant methods that take advantage of the hierarchical structure of the underlying path-tree computation to select sets of spatially diverse near optimal paths.

Another approach for corridor location is to simultaneously consider all objectives to determine the set of Pareto-optimal solutions between the objectives. This amounts to solving a discrete multi-objective shortest path problem, which is considered to be NP-Hard for computing the full set of non-inferior solutions. Given the difficulty of solving for the complete Pareto-optimal set, this dissertation develops an approximation heuristic to compute path sets that are nearly exact-optimal in a fraction of the time when compared to exact algorithms. This method is then applied as an upper bound to an exact enumerative approach, resulting in significant performance speedups. But as analytic computing continues to moved toward distributed clusters, it is important to optimize algorithms to take full advantage parallel computing. To that extent, this dissertation develops a scalable parallel framework that efficiently solves for the supported/convex solutions of a biobjective shortest path problem. This framework is equally applicable to other biobjective network optimization problems, providing a powerful tool for solving the next generation of location analysis and geographical optimization models.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. 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 (Kassakian and Schmalensee 2011). For example, tapping wind resources in remote locations requires 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, especially when a project crosses state boundaries or federal lands. 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 a transmission line 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 and reliable delivery of electricity.

A few years ago, committee chair for this dissertation Dr. Church attended a meeting in Jackson, Mississippi sponsored by the USDOT that dealt with the issues of a possible realignment of the CSX railroad. Claiborne Barnwell, a representative from the Mississippi Department of Transportation, 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 (Kishore and Singal 2014). The biggest hurdle in corridor location is to successfully finish the first phase with all agencies giving their approval after considering input from all relevant stakeholders. 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 his book *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 using a GIS, 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? Barnwell would not 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 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 (Ehlschlaeger 1998). 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 spatially from one "optimal" alignment.

For example, Electric Power Research Institute – Georgia Transmission Corporation (EPRI-GTC) published an Overhead Electric Transmission Line Siting Methodology (Houston and Johnson 2006) that is endorsed by Environmental Systems Research Institute (ESRI) and has been implemented in numerous states. It is a complicated ad-hoc methodology that can be coarsely summarized as iteratively solving straightforward single shortest paths on networks of various resolutions and objective weightings until a preferred route is selected. Corridors for refined study are generated by selecting all paths that fall in the first natural break of a histogram of path options. This selection is highly subjective to

choices of histogram bin size and the definition of a statistical break, and does not incorporate any spatial principles into the selection criteria. Since it is built-upon ESRI's ArcGIS Network Analyst tool, it also inherits the network representation errors described later in sections II.A.4 and II.A.5. While the EPRI method comes off as fundamentally sound due to its methodical procedure, technical buzzwords, and beautiful graphics; a critical spatial scientist would find numerous shortcomings throughout the methodology.

The next generation of corridor planning tools must address Barnwell's thesis, that to be successful and move to phase 2, one needs to address all competitive, but spatially different, alternatives either explicitly or implicitly in phase 1, 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 of alternatives. The work done in this PhD dissertation develops new methods of generating sets of least-cost path alternatives that follow the fundamental principals of spatial science. The variety of methods described incorporate new advances in parallel computation and sound spatial theory to give corridor location designers better tools to efficiently generate numerous path choices that meet the design criteria, justify why other paths violate design specifications, and present a set of viable options to interested agencies from which a selection can be made.

A. Historical Foundations and Related Problems

Determining the best path from one place to another can be considered one of the most fundamental of all spatial problems. Since the dawn of time, living organisms have used their knowledge and senses to cognitively search for the best route from one location to another. While formal algorithmic shortest path computation has been an active area of

research for only 60 years, variations on this shortest path problem have been studied for much longer. Within the context of cartography, John Norden produced the first known table of distances between a number of towns in Umbria (northern England) and patented the classic triangular distance table typically found in atlases and maps today in 1625 (Norden 1625). While his distances were likely calculated via manual measurements from his surveying maps along routes he selected using his local knowledge, this represents the first known instance of an all-to-all path distance computation.

Perhaps the earliest *mathematical* foundations of solving spatial routing problems came in 1735 from Leonard Euler and his study of the Seven Bridges of Königsberg problem. The city of Königsberg was set on both sides of the Pregel River, and included two large islands that were connected to each other and the mainland by seven bridges. The problem was to find a walk through the city that would cross each bridge once and only once (i.e. an edge-distinct path which must cross all bridges). Euler proved with mathematical rigor that the problem had no solution, and his proof is considered now to be the origin of graph theory and topology.

But shortest path computation doesn't involve simply the determination if there exists any vertex-distinct or edge-distinct path from one point to another. In fact, in most problems, there exist a combinatorially large number of possible paths from some origin to a destination. The shortest path problem then seeks to find the shortest distance path or the least cost path out of the enormous choices available. Mathematical methods for finding the least cost path on a network did not appear until the early 50's, which is relatively recent as compared to other similar minimum-path type problems such as the traveling salesman problem and the minimum spanning tree problem. In the traveling salesman problem (TSP),

one is given a list of points and their pairwise distances, and the task is to find a shortest possible tour that visits each point exactly once. The TSP was originally formulated verbally in a German 1832 manual for successful traveling salesmen (Schrijver 2005), then in 1931 Karl Menger defined the problem mathematically (Menger 1931). He considered the obvious brute-force algorithm for finding the optimal solution, and observed the non-optimality of the nearest neighbor heuristic, thus laying the foundation for eventually determining this problem as NP-Hard (Karp 1972).

The minimum spanning tree (MST) problem is similar to the travelling salesman problem, in that all points must be reached in the least combined distance, but in this case there is no requirement that the points be served in a vertex distinct tour. This relaxation makes the problem much easier to solve, with numerous greedy algorithms able to solve the problem to optimality in polynomial time on a pre-defined network. The first such optimal algorithm for the MST was developed in 1926 by Czech scientist Otakar Borůvka as a solution technique to The Electric Power Company of Western Moravia's request for a method to design the most economical power network (Boruvka 1926). Subsequent algorithms by Kruskal (1956) and Prim (1957) are what are most commonly used today for solving the MST problem. The planar minimal spanning tree problem, or the Steiner Tree Problem, remains an active research area. It is a more general version of the MST problem, where there is liberty to add additional vertices and interconnects to the graph in order to reduce total tree distances in reaching all nodes. In other words, one must find the underlying network (and "Steiner points") on the plane that connects points at a minimum total distance. This problem was originally made famous by Richard Karp's (1972) early

paper relating to computational complexity theory, in which the Steiner Tree Problem was one of the 21 NP-complete problems demonstrated in that publication.

These early developments laid the foundation for the theory and methods of solving network optimization and shortest path problems. The literature review in Chapter II details the origins and developments of methods specifically for corridor location with regards to solving shortest path problems. The next few sections outline the process of corridor location, and give motivation for the contributions in the remainder of this dissertation.

B. The Elements of Corridor Location and Problem Motivation

The design and implementation of a new energy source can be simplified into two spatial components: the point location of the energy source (i.e. fuel powered generation plant, wind farm, solar farm, hydroelectric plant), and the linear location of the transmission lines that deliver the energy to the demand locations. Power generation facilities are often located in remote locations, and the number of interested parties and agencies for permitting are relatively small due to the facility's impact as a singular location. On the other hand, transmission lines must cover distances much larger than the extent of the generation facility. They may traverse cities, counties, regions, and even multiple states. Applicants for new transmission lines must also obtain right-of-way permits to gain access to public land. With opposition from local communities who don't want power lines in their backyards, these approvals can take five to seven years or even longer.[1,2,3,4]

---

[1] "Wind Power Waiting on Transmission-Line Boom" *Greentech Media*, July 25, 2008. http://www.greentechmedia.com/articles/read/wind-power-waiting-on-transmission-line-boom-1181/all

[2] "Court rejects U.S. Bid to establish corridors for new electric transmission lines" *Los Angeles Times*, February 02, 2011. http://articles.latimes.com/2011/feb/02/local/la-me-electric-corridors-20110202

[3] "Giant new utility poles spark controversy in Chino Hills" *Los Angeles Times*, November 27, 2011. http://articles.latimes.com/2011/nov/27/local/la-me-powerlines-20111128

[4] "Getting 33% Renewables on the Grid, Part 3" *Greentech Media*, December 15, 2011. http://www.greentechmedia.com/articles/read/getting-33-percent-renewables-on-the-grid-part-3

When a utility decides it wants to construct a transmission line, the design process can also be split into two stages. Stage I involves determining the best compromise route based on the multiple objectives of all interested parties. Stage II involves the engineering design and eventual construction of the transmission line itself, including the locations of the poles / towers, access roads, interconnects, and so forth. This dissertation focuses on improving the methods for completing Stage I, as this is the most difficult and time-consuming stage due to the reasons stated in the previous paragraph.

In the 1970's, geographical information systems were developed with the capability of performing computer assisted transmission line location analysis of Stage I. Early thesis and dissertation work by Charles Smart (1976) at Virginia Tech, and Malcolm Potts (1975) and Ross Newkirk (1976) at the University of Western Ontario provided a procedural framework for this problem which is still applied in the present day.

1) **Identify goals and objectives** – Determine what factors are to be considered in the selection criteria for determining the best routes alternatives. Such criteria could include such factors as economic cost, environmental impact, proximity to population centers, and accessibility for maintenance.
2) **Collect relevant data** – Collect the data that corresponds to the goals and objectives. Such data could include land-use and waterways, vegetation type, slope, location of existing roads and existing transmission lines.
3) **Transform data into a network** – Classify the data layers into costs relevant to the goals and objectives, discretize the data into a raster with an appropriate scale, connect the raster cells with arcs using appropriate geometry and assign costs to the arcs.

4) **Shortest path analysis** – Compute a set of efficient alternative routes that consider the stated design goals and are differentiated enough that they can be considered distinct alternatives by a decision-making entity, and that cover the range of non-inferior solutions.

This dissertation focuses on the fourth step of this process, the shortest path analysis step, as this field alone has a vast wealth of interesting and relevant problems to be researched. The realm of shortest path analysis has already been an active area of research for the past 60 years, and the literature on the topic is covered thoroughly in the literature review of Chapter II.

The issue of shortest path analysis is not as simple as finding the single shortest path on a network. Different stakeholders will have different values, and thus what is considered the optimal path to one stakeholder may be very different to the optimal path of another. In addition, there is always uncertainty in the data used to generate a terrain network. Thus near-optimal paths should also be considered since a near-optimal path could in-fact be optimal within the error limits of the data (Ehlschlaeger 1998) or considering hidden objectives of stakeholders (Brill 1979). The selection criteria for the final route might include other implicit considerations that were not included in the analysis; therefore one must present a variety of solutions that are spatially diverse, covering the spatial range of reasonable solutions. Many methods have been developed to calculate route alternatives that address these complicated considerations, and are summarized in Table 1 on the following page. Table 1 also presents selected details about each approach including computational complexity and whether it is the subject of research presented in this dissertation, and the publication status of the dissertation work on that topic.

Table 1. Alternative Route Computation Approaches

Name	Category	Type	Complexity	Notes	Subject Of	Published
Iterative Penalty Method (IPM)	1-Objective	Iterative	Weakly Polynomial	Dismissed by the literature		
Gateway-Shortest Paths (GSP)	1-Objective	Gateway	Polynomial		Chapter IV	in prep. *
Multi-Gateway Shortest Paths	1-Objective	Gateway	Polynomial			
p-Dispersion	1-Objective	Integer Programming	NP-Hard	Depends on quality of candidate set		
kth-Shortest Paths (KSP)	1-Objective	Enumerative	Weakly Polynomial	Dominated by NSP		
Near Shortest Paths (NSP)	1-Objective	Enumerative	Weakly Polynomial		Chapter III	yes
Weighting Approach (NISE)	Multi-Obj. Supported	Iterative	Weakly Polynomial		Chapter VII	in review
Constrained Shortest Path	Multi-Obj. Unsupported	IP / Lagrangean	NP-Hard	Dominated by labeling algorithms		
Tchebysheff Method	Multi-Obj. Unsupported	Integer Programming	NP-Hard	Dominated by labeling algorithms		
Labeling Algorithms	Multi-Obj. Unsupported	Dynamic Programming	NP-Hard			
Multi-Objective KSP (GAPS)	Multi-Obj. Unsupported	Enumerative	NP-Hard			
Multi-Objective NSP	Multi-Obj. Unsupported	Enumerative	NP-Hard		Chapter VI	in prep.
Approximation Heuristics	Multi-Obj. Unsupported	Heuristic	Various		Chapter V	yes
EPRI-GTC	Ad-Hoc	Ad-Hoc	–	Uses GSP, AHP, Delphi, ArcGIS		

* The research reported in this dissertation is the subject of several papers that have been prepared for consideration in peer-reviewed journals. Material from two of these chapters has already been accepted or has recently appeared in print.

1. Single Objective Alternative Route Methods

Transmission corridor designers must factor numerous objectives when determining the optimal route alternatives to be presented to a decision maker or a group of decision makers and stakeholders. Simultaneously considering all of the dimensions of the problem is a daunting task, and numerous techniques have been developed to take all elements into consideration. The designer will have collected many layers of geographic data related to all of the objectives and the simplest approach is to appropriately classify and scale the data layers and combine them all into one composite layer by a pre-determined weighting scheme. This composite data set then has a single cost per raster cell unit, from which a network can be generated, as described in more detail in Chapter II, sections A.4 and A.5.

Solving for the single shortest path on such a network is a simple task, and can be accomplished with a shortest path algorithm such as Dijkstra's algorithm (Dijkstra 1959). But for a corridor location designer, it is necessary to calculate alternative routes as well due to the network uncertainty issues mentioned earlier, as well the need to satisfy Barnwell's dilemma by being able to demonstrate that all possibly optimal configurations were considered.

Tobler's First Law of Geography states that "Everything is related to everything else, but near things are more related than distant things" (Tobler 1970). Therefore, in order to generate distinct alternatives, they must be spatially differentiated. Methods developed to find alternative paths on roadways often use shared arc length to measure the difference between two paths (Kuby *et al.* 1997, Akgün *et al.* 2000). Lombard and Church (1993) make the case that for terrain-based raster problems, the area difference metric between paths is what should be used to evaluate the spatial divergence of two paths. It is easy to imagine in a

terrain network two paths could follow parallel adjacent routes, yet share no arcs (Figure 1a and 1b). These paths under the shared length metric would be considered completely different, but under the area difference metric are considered to be spatially similar. The two parallel paths would likely exhibit similar objective function properties, in accordance with Tobler's Law. On the other hand, using area difference as the similarity metric ensures that paths considered as dissimilar would have a true spatial difference between their routes, as in Figure 1c.

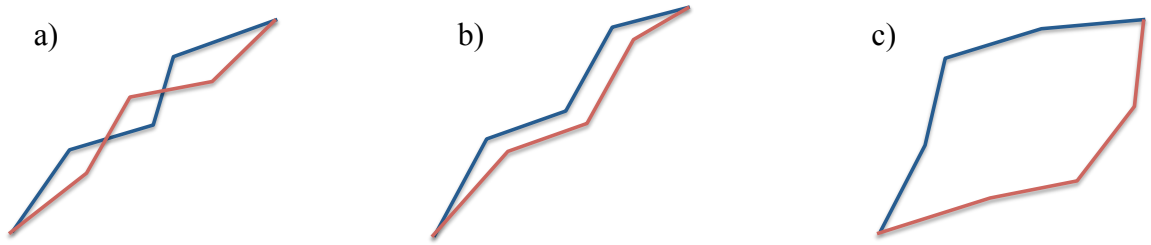
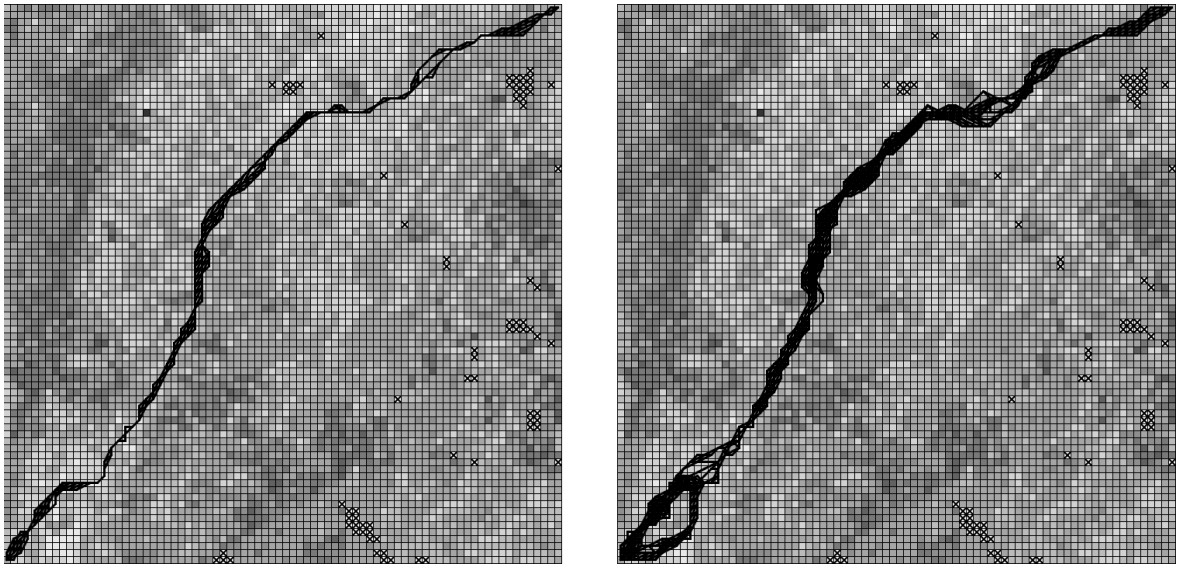


Figure 1. Comparing path alternatives: a) arc-distinct similar paths, b) arc and node distinct similar paths, c) arc and node distinct dissimilar paths

The first six algorithms in Table 1 list the main approaches used for determining single-objective shortest path alternatives. We provide short introductions to these methods in the following paragraphs, with much greater detail and citations provided in the alternative paths literature review of Chapter II section E.

The most basic approach for generating alternative routes is to enumerate all possible paths and select the best alternatives from those. While this may seem like a simple process, a complete path enumeration solution set grows factorially as the size of the network increases. But many paths are vastly inferior to the optimal path, so then it makes sense to consider instead the optimal shortest path, and a set of near-optimal paths. These can be enumerated using a k^{th}-shortest path algorithm, which generates a ranked list of k least cost paths; or a near shortest path algorithm, which generates all paths (unranked) within a

specified cost range. While these approaches will only compute paths that are high in performance, an overwhelming number of these alternatives are simply small deviations from one another rather than spatially distinct alternatives. Such alternatives are of little use to planners when facing a public forum as one must be able to convey that a wide variety of spatial configurations were considered in the analysis of alternatives. Take for example an 80x80 GIS raster grid network, which these days would be considered a very small data set. Figure 2 displays all corner-to-corner paths that are up to (a) 0.3% and (b) 0.8% more costly than the least-cost path. The first instance represents a total of 4,459,050 distinct paths, yet essentially all traverse similar routes. The second instance displays a total of 160,650,434,203 distinct paths and took more than two days to compute on a 2.8 GHz Intel Xeon server running the very fast ANSPR0 Near Shortest Paths algorithm by Carlyle and Wood (2005). Even with the over 160 billion distinct shortest paths, very few spatially distinct alternatives are generated.



(a) (b)

Figure 2. All path options within (a) 0.3% of the shortest path cost (4,459,050 paths), (b) 0.8% of the shortest path cost (160,650,434,203 paths)

A less computationally burdensome way of generating alternatives consists in modifying the graph to force some path diversity from each iteration. A simple and efficient procedure based on this concept is the Iterative Penalty Method (IPM), which solves shortest path problems sequentially, but after the generation of each new path the arcs or nodes comprised in the solution are penalized so as to discourage their use in subsequent iterations. The mechanism of penalization may take a number of user-defined forms, and thus the effectiveness of the overall methodology is hence strictly dependent on the penalization strategy in use as well as on the magnitude of the applied penalties. And even with good penalty approaches, most alternatives take the form of parallel paths that are small deviations from one another.

Even if one can generate a large enough candidate set using enumeration or IPM that there is true spatial diversity, one then must sift through the enormous solution set and somehow pick a reasonable subset of diverse alternatives for further analysis. Some approaches have been developed where the task of guaranteeing and evaluating dissimilarity is handled as a second phase after generating a large candidate set, such as the minimax method proposed by Kuby *et al.* (1997), and in the *p*-dispersion model developed by Akgün *et al.* (2000). The minimax method is a greedy approach that iteratively selects paths where each selection minimizes a linear combination of length and similarity with all the paths already inserted in the differentiated subset. The discrete *p*-dispersion model takes the set of candidate paths, and selects a subset of *p* paths that maximizes the minimum difference between any pair of selected paths. But the *p*-dispersion problem is an NP-hard problem; therefore solving such a problem on a GIS raster network requires massive computation that includes 1) enumeration of an exponentially large number of paths for a candidate set in

order to achieve any spatial diversity, followed by 2) computing some measure of difference between all pairs of the exponentially large path set, be it some simple metric such as shared length or a better metric such as area difference, followed by 3) solving the NP-hard *p*-dispersion problem on the exponentially large path set.

An approach more suitable for terrain networks is the Gateway Shortest Path (GSP) model, which is founded on the idea that each gateway shortest path is constrained to travel through a pre-specified node to generate an alternative route. Such a route can depart significantly from the shortest path, depending on the location of the selected gateway; and it carries a good guarantee of impact quality, being the best possible path going through that cell. The method consists in computing two shortest path trees, one rooted at the origin and one rooted at the destination, and overlaying them to obtain the distance and area difference values for every gateway alternative. Hence, the overall methodology only requires running a modified shortest path algorithm twice and performing label additions to generate up to $n-2$ alternative paths, where n is the number of nodes in the network. Computational experience on a real-world corridor location problem reported in Lombard and Church (1993) demonstrated the superiority of the GSP model over the IPM, both in terms of computational effort and solution quality, and gateway shortest paths are also now a common feature in modern online mapping tools (i.e. Google Maps, Bing Maps, OpenStreetMap) for interactively generating alternative routes (Luxen and Vetter 2011).

2. Multiple Objective Alternative Route Methods

Transmission line designers must consider numerous objectives when selecting the best route for a transmission line corridor. In the previous section we discussed approaches for when these objectives are combined into a single objective based on a weighted ranking of

priorities. Another method to approach such a problem is to simultaneously evaluate the various independent objectives, and to generate a set of solutions that represent the set of optimal trade-off solutions between the various objectives. Consider an example with two objectives, such as a biobjective shortest path problem, where the desire is to minimize both objective values (e.g. route cost and route environmental impact). Any proposed path can be drawn on a map, which represents the solution in decision space. That solution will have objective values for each objective corresponding to the cumulative sum of cost-surface values where the path traverses. The performance of a path with respect to the objectives can then be evaluated by plotting its objective values as a point in objective space. Figure 3 provides an example of solutions plotted in objective space, where the two objectives, z_1 and z_2, are to be minimized.

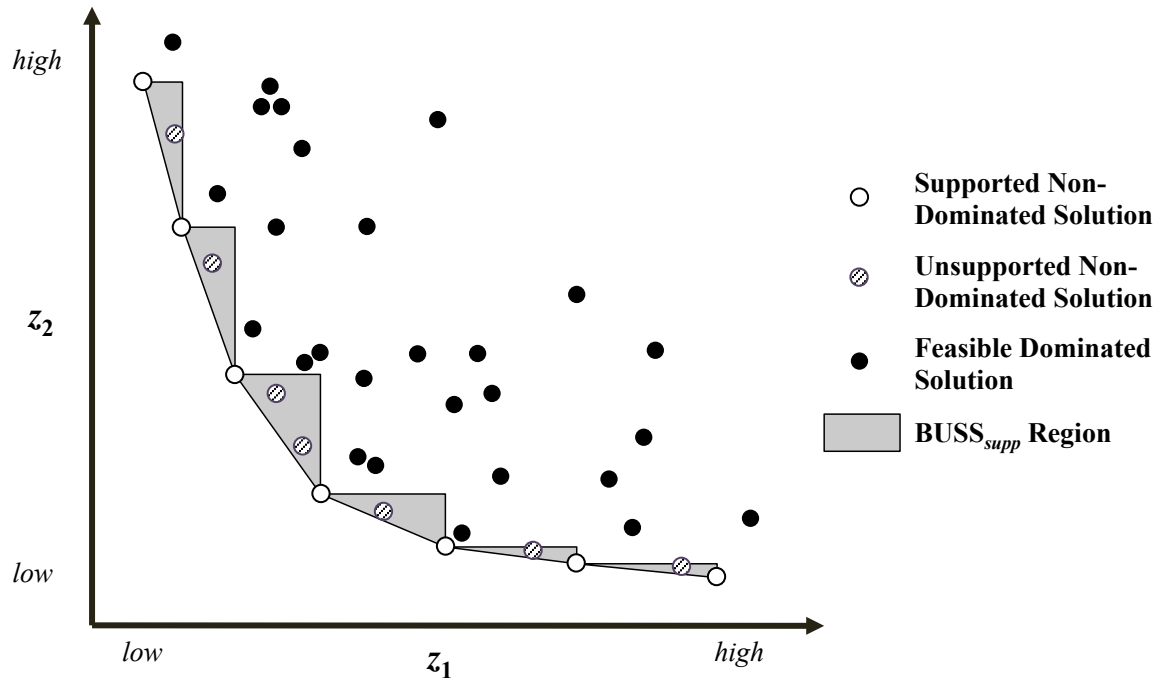


Figure 3. Categorization of Bi-Objective Solutions to a discrete problem in objective space

When numerous path configurations are analyzed in objective space, one can visually determine which paths represent better trade-offs between the objectives. The optimal trade-

off solutions are called non-dominated solutions or Pareto optimal solutions, named after the Italian economist Vilfredo Pareto (1848–1923) who used this concept in his studies of economic efficiency. A solution is Pareto optimal if there does not exist any other feasible solution with better performance with respect to one or more objectives without having to compromise on at least one other objective. These solutions are further divided into two sub-categories. Supported solutions are the vertices of the convex hull of the non-dominated solutions, and are shown as white-filled circles in Figure 3. Unsupported solutions are the remaining non-convex non-dominated subset, and are shown as diagonal-pattern-filled circles in Figure 3. Unsupported solutions exist in right-triangular regions between the supported solutions. These regions are known as duality gaps or Boundary of Unsupported Solution Search (BUSS) regions, and are depicted as gray shaded triangles in Figure 3.

Dominated solutions are solutions where there does exist another solution that is better in one or more objectives and no worse in the other objectives, and are shown as black circles in Figure 3. Graphically, in a minimization problem such as in Figure 3, a solution is dominated if there exists any other feasible solution to the lower left, since that other solution has smaller or equal objective values with respect to both z_1 and z_2.

While supported and unsupported Pareto optimal solutions are equally valuable in a multi-objective engineering problem, solving for the unsupported solutions is a far more difficult task than for the supported solutions. Methods for determining Pareto optimal solutions are covered more in-depth in the literature review in Chapter II section D, but in summary, assuming the single-objective problem can be solved in polynomial time, supported solutions can be found in weakly polynomial time as optimal solutions to a cumulative weighted single-objective problem. Unsupported solutions are equivalent to

adding a resource constraint to the problem, and are thus of the class NP-Hard. General interactive approaches for solving unsupported solutions may include adding resource constraints to all-but-one objective then solving optimally for the final objective (Current *et al.* 1990), or using a weighted Tchebycheff procedure (Steuer and Choo 1983). In a biobjective shortest path problem, specialized solution approaches have been developed for solving the complete set of unsupported solutions. These fall under two categories: labeling algorithms (Martins 1984b, Skriver and Andersen 2000) and enumeration algorithms (Coutinho-Rodrigues *et al.* 1999, Raith and Ehrgott 2009). While these specialized methods are more efficient than the general resource constraint or Tchebycheff approaches, one must keep in mind that the problem they address is an NP-Hard problem, and thus still difficult to compute for large problems.

3. The EPRI-GTC Ad-Hoc Method

The EPRI-GTC Overhead Electric Transmission Line Siting Methodology (Houston and Johnson 2006) is a comprehensive methodology for siting corridors and routes for constructing new transmission lines. It has been used in numerous projects around the United States, and has become popular as the standard methodology for these applications. It is a multi-stage method that uses ArcGIS as the underlying tool for solving numerous least-cost paths over various trade-off networks. The networks consist of different weightings of the many objectives to be considered in routing a transmission corridor, with these weighting schemes generated iteratively using the Delphi method and the analytic hierarchy process (AHP).

ArcGIS has built-in functionality for combining numerous cost layers into a single cost surface, can generate a raster network on this cost surface, and then run a shortest path

algorithm to determine a least cost path from one point to another. ArcGIS also allows for single gateway paths to be computed. On the surface, this appears to be an adequate toolset for selecting various route alternatives for a transmission corridor, but there are numerous shortcomings that arise when the details of this method are examined more closely.

The first problem is in the generation of a network from a raster cost surface, a topic discussed in much greater detail in Chapter II section A.4 of the literature review. ArcGIS generates only an $R = 1$ (queen's move) network over the raster. As shown by (Goodchild 1977), this imparts an elongation error of up to 8.2% for any solution derived from this network, and may impose higher errors from returning an optimal route that takes a different spatial path than the optimal path in a higher order network. Much improvement could be gained from the solutions generated by ArcGIS if it allowed a user to specify a higher order network.

Another shortcoming in ArcGIS corridor location is that the networks it generates use zero-width arcs. As discussed earlier in this proposal, zero-width is an unrealistic scenario, as all physical corridors will have a zone of impact with a non-zero width. Using a zero-width network may generate paths that will in the abstract, graze regions of high cost, but in reality would actually traverse through such regions. The EPRI method artificially generates corridors with some width by putting all gateway paths into a histogram, and selecting the aggregation of paths of length within the arbitrary "first statistical break" of the shortest path length. This break scheme suffers from a modifiable unit problem in the selection of the histogram bin sizes, and uses a variable and arbitrary cutoff for the edges of the corridor width.

EPRI's use of the Delphi method and analytic hierarchy process (AHP) to determine objective weightings for the different graphs is one way to incorporate stakeholder feedback into the process of generating corridors and routes while minimizing the number of shortest path solver iterations. But an alternative method is to solve for all Pareto-optimal and perhaps even near Pareto optimal solutions on a multi-objective problem that incorporates all of the trade-offs considered in the corridor design. This approach may be daunting, as solving a multi-objective problem is weakly-polynomial for the supported (convex) Pareto solutions, and NP–Hard for the unsupported (non-convex) Pareto solutions. But further advances in computing power from parallel processing allow for these larger, more complex problems to be solved. A main focus of this dissertation is thus finding new methods for harnessing parallel computation to solve these difficult multi-objective problems, so that they may be incorporated into a next generation transmission line siting methodology. Rather than using stakeholder feedback to generate a few alternatives, the approach could instead generate all optimal trade-off solutions and then use stakeholder feedback to select from those, a process called the "bottom-up approach" that is often preferred over a priori weighting (Cohon 1978). This allows a designer to be more confident that all quality alternatives were in-fact considered for final selection.

Any new methodology used for transmission corridor design should address the spatial errors and the limited range of alternatives considered in the current generation of transmission line siting methodologies. It is not the intent of this dissertation to rewrite a new methodology similar to EPRI's, as it is impossible to account for all of the political and policy decisions that went into the decision-making for that approach. But instead, the aim is to develop new tools that can be incorporated into the next generation of transmission line

siting methodologies, using modern computation techniques and spatial analysis to enable a more thorough analysis of corridor route alternatives.

C. Dissertation Contributions and Outline

The remainder of this dissertation will detail work done to develop new tools for the next generation of geographical information systems (GIS) in order to allow improved corridor location on terrain based raster networks.

Chapter II begins with a review of important definitions and theoretical concepts of the issues associated with a corridor location problem, followed by a complete literature review of the methods for solving single-objective shortest paths, multi-objective shortest paths, and spatially diverse alternative shortest paths.

Chapter III discusses new research on near shortest path enumeration. While it has already been shown that the combinatoric nature of enumeration requires an extraordinary number of solutions to be computed in order to gain spatial diversity, this work presents a parallelization of the Carlyle and Wood (2005) near shortest path algorithm in order to take advantage of parallel computing to improve the computation time of such path sets.

Chapter IV presents work that extends the Lombard and Church (1993) gateway shortest path problem, which has been shown to be capable of efficiently generating a large set of spatially-diverse near-optimal shortest paths. This research incorporates theories used in hydrology to develop a new method to automate the selection of a subset of diverse path alternatives from the gateway path set in order to quickly generate a set of solutions that are both spatially diverse and near-optimal with respect to the objectives.

Chapter V is devoted to a new algorithm for approximating the entire Pareto frontier of a biobjective shortest path problem. The method combines the NISE approach of Cohon *et al.*

(1979) for determining the exact supported solution set with the gateway shortest path approach of Lombard and Church (1993) to generate an approximation of the non-dominated solution set. Experiments show the approximation to be very near to the exact solution while computing in a fraction of the time needed for exact approaches.

Chapter VI extends the work of Chapter V by using the biobjective gateway approximation heuristic as an initial upper bound for an exact enumerative algorithm for solving a biobjective shortest path problem. Experiments then compare this improved enumeration approach to previous enumeration and labeling methods on raster grid, random, and road networks.

Chapter VII presents a new parallel framework for solving biobjective network problems for which there exist specialized solution algorithms. The framework uses the Java Fork/Join library to simplify the work distribution and synchronicity nuances that arise in a parallelization scheme. Experiments on solving a biobjective shortest path problem on a massive data set demonstrated excellent parallel performance on a variety of different operating systems and hardware configurations.

Finally, Chapter VIII concludes this dissertation with a summary and remarks for possible future work.

Each chapter in this dissertation represents a distinct contribution, and can be considered each as independent documents. Chapters III, V, and VII have been published or are pending publication in peer-reviewed journals or book chapters. The other chapters are in preparation for publication, thus all chapters contain separate introductions and literature reviews pertinent to their specific problems that are in addition to the comprehensive literature review in Chapter II.

II. Literature Review

A. Important Concepts and Basic Algorithms

1. Shortest Path Problem Formulations

Let G = (N, A) be a directed graph network with node set $N = \{u_1, u_2, \ldots, u_n\}$ and arc set $A = \{(u_1, v_1), \ldots, (u_m, v_m)\}$, and define $n = |N|$ and $m = |A|$. Each arc $(u,v) \in A$ has associated with it a positive real cost c_{uv}. Note, if an arc can be traversed in either direction, it can be represented as two directed arcs. Cost of traversal may not necessarily be symmetric. The goal of the one-to-one shortest path problem is to solve for the minimum-cost path from a source node $s \in N$ to a destination node $t \in N$. Each arc has associated with it a binary decision variable x_{uv} that is equal to 1 if it lies on the optimal shortest path, and 0 otherwise. This results in the following problem formulation:

$$\begin{aligned} \min \ & z(x) = \sum_{(u,v)\in A} c_{uv} x_{uv} \\ \text{s.t.} \ & \sum_{(u,v)\in A} x_{uv} - \sum_{(v,u)\in A} x_{vu} = \begin{cases} 1 & \text{if } u = s \\ -1 & \text{if } u = t \\ 0 & \text{if } u \neq s,t \end{cases} \\ & x_{uv} = \{0,1\} \text{ for all } (u,v) \in A \end{aligned} \qquad (1)$$

The one-to-all shortest path problem involves solving for the minimum-cost path from a source node $s \in N$ to all other nodes in the graph. This results in a shortest path tree, where the shortest path from any node may be determined by following the unique path along the tree from that node to the source node, and is formulated as follows:

$$\begin{aligned}
&\min\ z(x) = \sum_{(u,v)\in A} c_{uv}x_{uv} \\
&\text{s.t.}\quad \sum_{(u,v)\in A} x_{uv} - \sum_{(v,u)\in A} x_{vu} = \begin{cases} n-1 & \text{if } u = s \\ -1 & \text{if } u \neq s \end{cases} \\
&\qquad x_{uv} = \{0,1\} \text{ for all } (u,v) \in A
\end{aligned} \tag{2}$$

The aim of the biobjective shortest path problem is to solve for the minimum-cost paths from a source node $s \in N$ to a destination node $t \in N$ that minimizes two, often competing, objectives, z_1 and z_2. For this problem, each arc $(u,v) \in A$ has associated with it two positive real costs $c_{uv} = (c^1_{uv}, c^2_{uv})$.

$$\begin{aligned}
&\min\ z_1(x) = \sum_{(u,v)\in A} c^1_{uv}x_{uv} \\
&\min\ z_2(x) = \sum_{(u,v)\in A} c^2_{uv}x_{uv} \\
&\text{s.t.}\quad \sum_{(u,v)\in A} x_{uv} - \sum_{(v,u)\in A} x_{vu} = \begin{cases} 1 & \text{if } u = s \\ -1 & \text{if } u = t \\ 0 & \text{if } u \neq s,t \end{cases} \\
&\qquad x_{uv} = \{0,1\} \text{ for all } (u,v) \in A
\end{aligned} \tag{3}$$

While the above formulation contains two distinct objectives, supported (convex) Pareto-optimal solutions may be found by solving the weighted combined single-objective formulation, using the weight α, where $0 \leq \alpha \leq 1$.

$$\min z_C(x) = \alpha \times z_1(x) + (1-\alpha) \times z_2(x) \tag{4}$$

Different supported solutions may be computed by varying the weight, α, between the two objectives. Setting $\alpha = 1$ finds the optimal solution considering only the first objective, while setting $\alpha = 0$ finds the optimal solution with respect to the second objective, and setting α to something in between to find compromise solutions on the trade-off curve. Note that for when $\alpha = 0$ or $\alpha = 1$, the resulting solution may be *weakly* dominated by another

supported solution with an equal objective value. More detail on this phenomenon can be found in chapter VII.C.2.i.

2. Shortest Path vs. Least-Cost Path

There is a distinction to be made between solving for the shortest path, and solving for the least-cost path. In fact, the shortest path can be thought of as a special case of the least-cost path problem, where the penalty of traversing a region is solely made up of the distance traversed by taking a step through that region. Hong and Murray (2013) recently developed an improved method that is quite efficient for computing the shortest path around barriers in uniform cost Euclidian space. But when a network represents space in the real world, penalties can arise from numerous other factors. On road networks, one might want to not only solve for a point-to-point shortest distance path, but also for point-to-point least time path. In this case, factors such as average traffic speed, number of stoplights, the nature of turns (right vs. left turns at stoplights), all may have effects which result in a very different optimal least-time path compared to a least-distance path. When determining a least-cost path for constructing new transmission lines over open terrain, penalties may arise from construction cost factors (slope, soil type, vegetation cover, etc.), human and wildlife disturbance factors (e.g. penalties for traversing residential areas or nature reserves), maintenance factors (e.g. accessibility to major roadways), and much more. Once again, a route that minimizes penalties from those factors may be much different from a path that simply minimizes distance. The same algorithms are used for solving shortest paths and least-cost paths, but one must simply be explicit on exactly what is being considered the spatial penalty to be minimized, be it simply distance, or any combination of other factors.

Despite these technical differences, the words "shortest path" are still often used to reference what is actually a least-cost path.

3. 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 $O(x^4)$.

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 occur in a pathological network where the shortest path must traverse through all nodes to reach the destination. Clearly, this would rarely if ever 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 networks used, it can show that sometimes algorithms with slower theoretical worst-case efficiencies will run faster when applied to real-world problems.

The efficiencies of shortest path algorithms depend on the number of cells/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.

4. Geometric Error: Orthogonal, Queens, and Knights Moves

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 cost-distance function is based on 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) showed that significant geometric errors might arise if you do not 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 every other node, known as a complete graph. In this instance, maximum density is achieved with $m = n^2$ arcs, so the density is $m/n = n$.

Huber and Church (1985) use a radius metric called *R* to differentiate between the types of moves allowed on a raster network. *R* = 0 is when only orthogonal moves are allowed, *R* = *1* represents diagonal moves in addition to orthogonal moves, and *R* = 2 additionally allows "knight's moves".

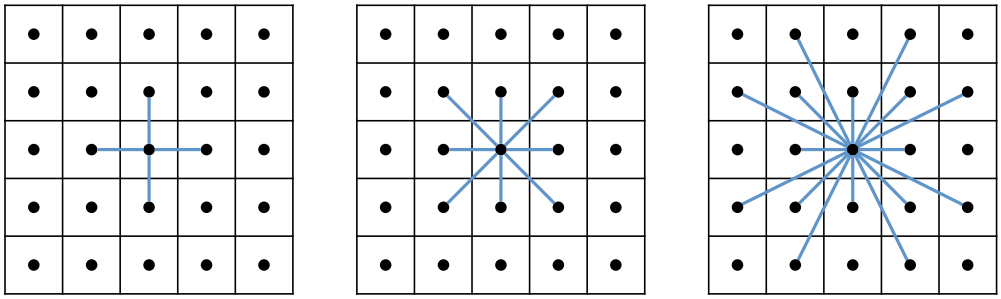
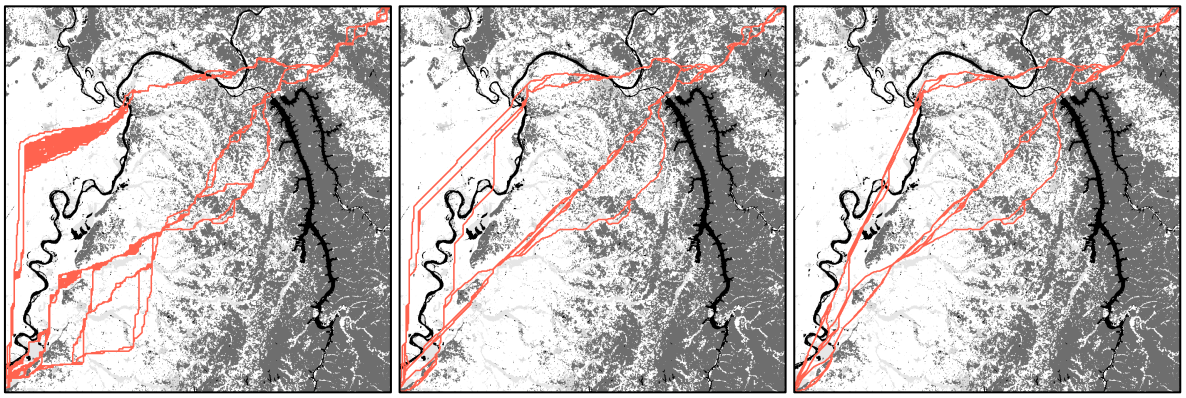


Figure 4. Interconnectivity metric on raster graphs. (a) R = 0, (b) R = 1, (c) R = 2.

Goodchild (1977) showed that an *R* = 0 network will impart a maximum of 41% elongation error to a shortest path solution. Increasing the directions (and arc density) to an *R* = 1 network reduces this error down to 8.2%, while an *R* = 2 network will have a maximum 2.8% elongation error. Anything above *R* = 2 will have an elongation error of <1%. Higher order *R* values can be used, but Huber and Church found that *R* = 2 provides the most satisfactory trade-off between accuracy and computational burden. In addition to the geometric elongation error, Huber and Church (1985) point out that path routes may be greatly affected by the radius of the network representation. Figure 5 shows the supported solutions to a biobjective shortest path problem on a million node network. These supported solutions are the convex Pareto optimal trade-off solutions when viewed in objective space. Figure 5 (a) depicts 97 distinct solutions for the *R* = 0 representation of this network, Figure 5 (b) contains the 144 distinct solutions for the *R* = 1 representation, and Figure 5 (c) shows the 281 distinct solutions for the *R* = 2 representation.



(a) (b) (c)

Figure 5. Supported solutions to a biobjective shortest path problem for (a) $R = 0$ (b) $R = 1$ (c) $R = 2$ networks, shown in decision space

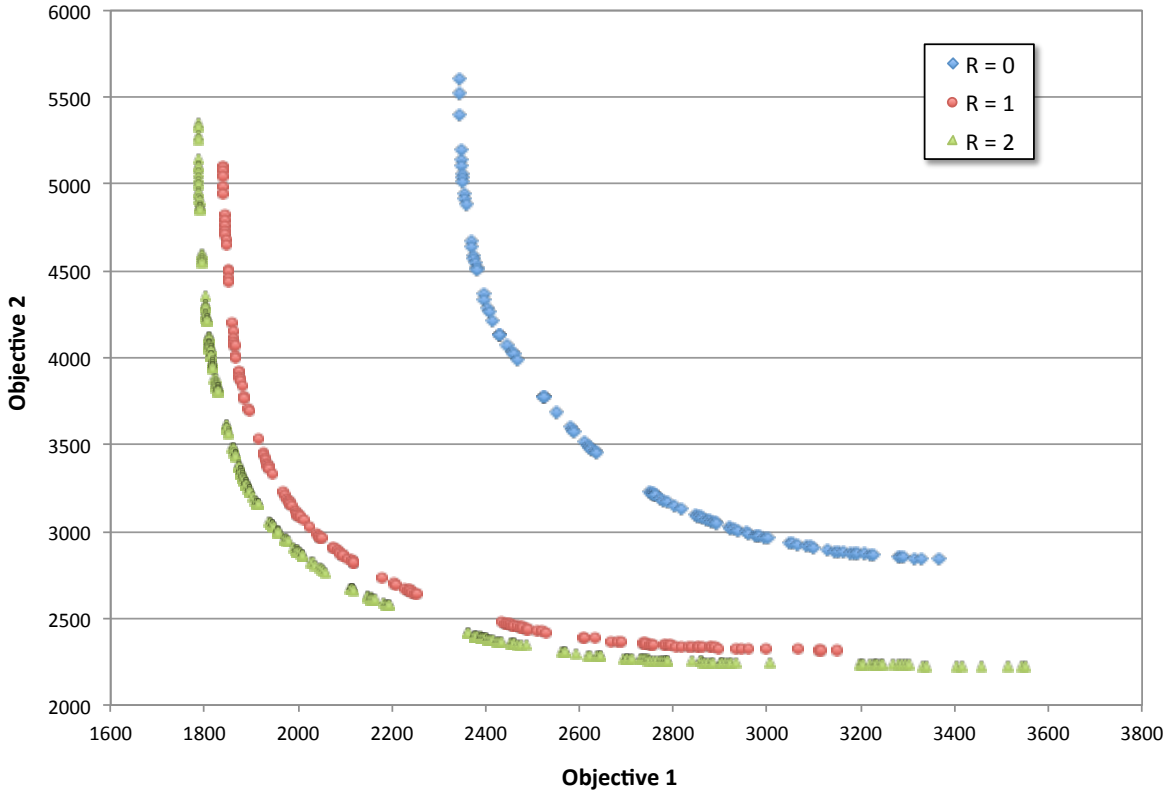


Figure 6. Supported solutions to a biobjective shortest path problem for $R = 0, 1, 2$ networks, shown in objective space

Figure 6 shows these same solutions in objective space, pointing out how the performance of the Pareto-optimal set changes for the different network representations. It is clear from these figures that when searching for spatial route alternatives, the network

representation will have a major effect on the number of alternatives, the performance of the paths, and the spatial routes those paths take.

5. Corridor Width

Any real-world corridor will have an impact zone of some width that must be considered in the design process. Unfortunately, most GIS software packages do not take this into account when generating terrain-based raster networks. For example, ArcGIS uses zero-width arcs in its path finding routines. A proper design should take width into account, which is particularly important in raster defined networks that contain abrupt changes in suitability scores between nearby cells.

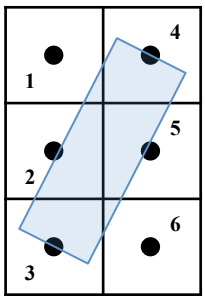


Figure 7. Depicting a corridor footprint

To properly account for impact zone, one should incorporate width into the cost function of each arc in the network (Huber 1980, Huber and Church 1985). Figure 7 gives an example of a knights move arc from node 3 to node 4. To properly calculate the cost function for an arc with width, for each cell that the arc covers one must calculate the area covered by the arc, and multiply it by the suitability score of that cell. The total arc cost then is the sum of these area weighted suitability scores. In Figure 7, this means that the arc cost is a function of all six depicted cells. By contrast, in a zero-width approach the arc cost would be equal to the sum of length-weighted suitability scores. In the example, this would

include only cells 2, 3, 4, and 5. If cells 1 or 6 happened to be highly unsuitable, a zero-width approach would not take this into account.

The process of adding width is tricky to calculate for arc widths that are much greater than the cell size, likely explaining why it is absent from most existing path finding software. One way of approximating corridor width, as suggested by Huber and Church (1985), is by smoothing the suitability score matrix. Smoothing is done by averaging cell suitability scores with the scores of neighboring cells, in essence equivalent to a low-pass filter on the raster. This minimizes abrupt changes, and adds a buffer region around areas of high impact. While not a perfect substitute for actually taking corridor width into account, smoothing will generally result in solutions more sensitive to the fact that corridors have an impact zone of greater than zero width. Smoothing is a simple and quick operation to perform, but the operation has many parameters that can affect the output. A designer should make justified decisions to approximate reality when defining the parameters of a smoothing operation.

6. Single Shortest Path: Dijkstra and A*

Computing a solution to many path problems often includes implementing a single shortest path algorithm as part of the solution process. For example, 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, or any other path problem with a single shortest path component, one must also ensure that the shortest path subroutine is optimal for the particular type of network one is analyzing.

The most widely used algorithm in solving the single-origin shortest path problem, for both its simplicity and efficiency, is Dijkstra's algorithm (Dijkstra 1959). While other shortest path algorithms did exist before his (Ford 1956, Orden 1956, Dantzig 1957, Minty 1957, Bellman 1958, Moore 1959), Dijkstra's label setting algorithm quickly became the standard one-to-one and one-to-all shortest path algorithm due to its ease of understanding and implementation, and its computational efficiency. Over the years, Dijkstra's algorithm has been amenable to improvements in efficiency through the use of advanced data structures (Dial 1969, Fredman and Tarjan 1987, Ahuja *et al.* 1990, Cormen 1990, Cherkassky *et al.* 1996) and spatial heuristics (Hart *et al.* 1968), thus maintaining its high stature even in the face of newer shortest path algorithms (Pape 1974, Glover *et al.* 1984, Pallottino 1984, Glover *et al.* 1985, Ahuja *et al.* 1990, Goldberg and Radzik 1993).

Dijkstra's 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). 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 is most often used (Ford 1956, Bellman 1958).

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 is finished 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. When a node is labeled temporary, it is included in the priority queue PQ, which in the naïve implementation is unordered, or in advanced implementation may use buckets, heaps, or some other advanced priority ordering. By using advanced data structures to store the priority queue, one can extract the minimum cost element in far less time than scanning all-the-way-through the list each time. 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 (P), and set $d(s) = 0$, and $prev(s) = s$. Add all nodes j such that $(s, j) \in E$ to the temporary node set PQ, label them as temporary T and set $prev(j) = s$ and $d(j) = c(s, j)$.

Step 2: Selecting the best T node from PQ, let node $v = (minimum(d(j)) \mid j \in PQ)$. Set the label of v to permanent P and remove it from PQ. 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 already labeled P, then continue. If j is labeled T, then compare the node's current

distance label, $d(j)$, to the distance associated with the new arc, $d(v) + c(v, j)$. Else if $d(v) + c(v, j) < d(j)$, then set $d(j) = d(v) + c(v, j)$, and set $prev(j) = v$. Also remove and re-add j into PQ. Else if node j is unlabeled U, then set j as labeled T and add it to PQ, 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 until it reaches the ending node t. For a one-to-all solution, the algorithm may be set to stop when all nodes are permanently labeled. 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 search toward the destination node t and greatly reduce the number of operations required to find the one-to-one 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 temporary nodes 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 faster than the most efficient forms of Dijkstra's algorithm when run on real road networks (Zeng and Church 2009).

7. Gateway Shortest Paths

In the early 1990's, two papers emerged from the Geography Department at UCSB that formally introduced the concept of a gateway shortest path for use as an alternative path generator in a corridor location problem (Church *et al.* 1992, Lombard and Church 1993). The 1993 paper, published in *Geographical Systems*, described the concept of a gateway path in a geographical context. The 1992 paper focused more on a simple interface programmed by the research group in order to generate and evaluate such gateway paths on a raster network.

A single-gateway path is a one-to-one shortest path, where the path is constrained to go through a selected intermediate "gateway point". To generate a single-gateway path, one must find the shortest path from the origin to the gateway point, as well as the shortest path from the destination to the gateway point. The gateway path is then the union of these two shortest paths. If a shortest path tree is calculated in advance (complexity $O(m + n \log n)$ using Fibonacci heaps), then the generation of any single-gateway path is a simple $O(n)$ operation.

There are some shortcomings with single-gateway point paths though, primarily that single-gateway paths consist of a very limited set of paths out of the total possible paths between an origin and destination. There may arise situations where a gateway path is just a

slight deviation from the shortest path (Figure 8, orange path), whereas there may exist another path just slightly longer that spatially takes a much different routing (Figure 8, green path). This slightly longer but more diverse path would likely be considered a better alternative path choice than the shorter gateway path when compared to the shortest path.

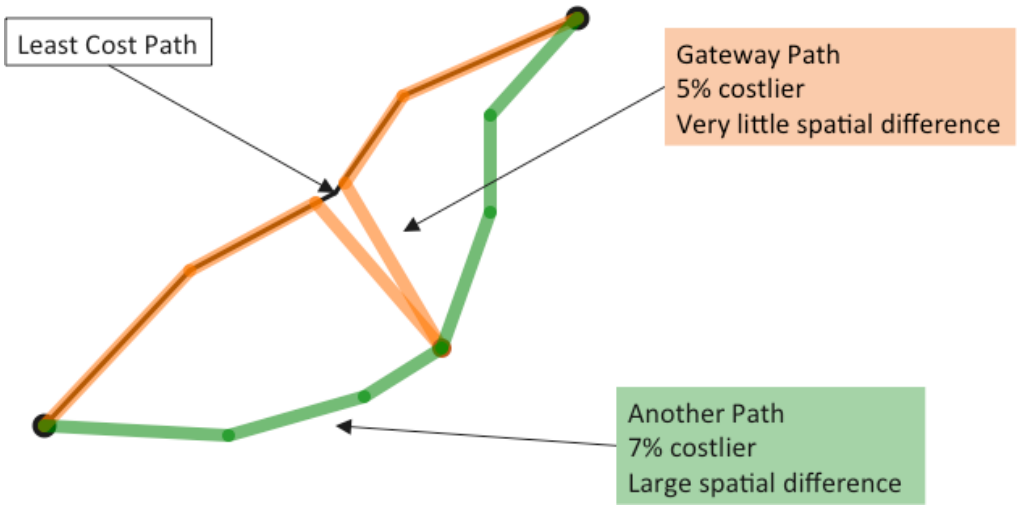


Figure 8. Example of a possible shortcoming of single-gateway paths

Scaparra *et al.* (2014) uses multiple gateway points to generate more complex paths. The general methodology is the same as single gateway, except for that now the path is constrained to go through more than one point. For example, a 2-gateway point path is composed of the shortest path from the origin to the first gateway point, the shortest path from the first gateway to the second gateway, and the shortest path from the second gateway to the destination. Multiple gateways allow the generation of paths that are not possible with only a single gateway, but the computational overhead is increased as well. However, if preprocessing is allowed then one can compute all necessary information required to generate multi-gateway paths by solving the all-to-all shortest path problem beforehand. An all-to-all shortest path solution gives the shortest paths from all points in the network to all other points in the network. The Floyd-Warshall Algorithm (Floyd 1962, Warshall 1962) is

a matrix oriented dynamic programming algorithm that solves all-to-all shortest paths in $O(n^3)$ time and $O(n^2)$ space. Alternatively, one can also solve Dijkstra's algorithm n times, resulting in a smaller $O(n(m + n \log n))$ time complexity. Both methods are highly parallelizable.

One area where running Dijkstra's algorithm repeatedly for generating multi-gateway paths may have an advantage is if there exists an upper bound on viable path lengths. Suppose in an application one wants to find path alternatives which are longer than the shortest path length L_{sp}, but shorter than some other path length D, it can be said that alternate paths lengths L must be $L_{sp} \leq L \leq D$. 1-gateway paths give you the shortest path traversing through the gateway node. If that 1-gateway path length is greater than D, then there cannot exist any multi-gateway path using that gateway point which will have length less than the 1-gateway path for that point. As such, it is not necessary to compute the shortest path tree from that node for making multi-gateway paths. This reduction takes $O(n)$ time to evaluate, and will reduce the problem size depending on how close the upper bound is to the shortest path length. When $D - L_{sp}$ is small, this may result in large computational savings.

8. K-Shortest Paths

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, up to the k^{th} shortest. Originally formulated by Bock, Kantner, and Hayes (Bock *et al.* 1957), they solved the problem using a brute force method of systematically listing all possible routes between a given origin and destination, then ranking them in order of length. This approach increases in runtime and

memory requirements factorially as the graph increases in size, so is limited in all practicality to trivially small networks. Numerous subsequent algorithms have been proposed for solving the *k*-shortest path problem using much more elegant techniques than those originally proposed by Bock et al. A few of the most relevant algorithms will be discussed in the literature review portion of this dissertation, although more in-depth reviews of various *k*-shortest path algorithms can be found in Dreyfus (1969), Eppstein (1998), Martins et al. (1998) and Hershberger et al. (2007a), 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).

9. 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 9 (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 $arc(a,b) \notin A_d$.

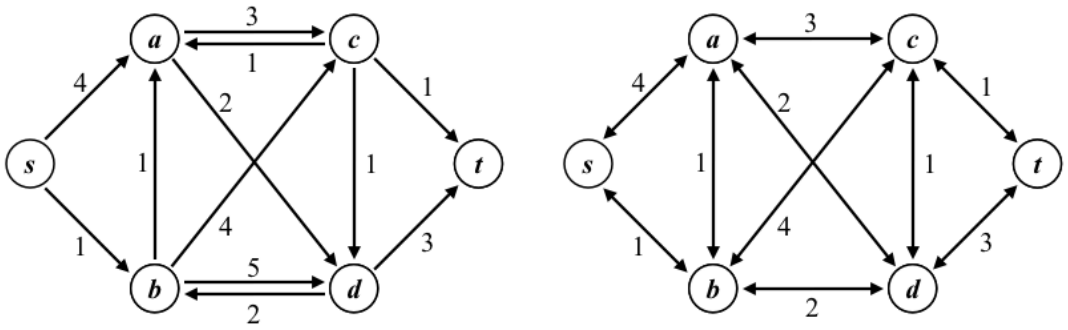


Figure 9. On the left, a directed graph (a), and on the right, an undirected graph (b)

Figure 9 (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, whereas 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. On the other hand, wayfinding on a terrain is an example where the cost is directed, since hiking uphill vs. downhill have different human impacts.

10. Simple Paths vs. Paths With Loops

By definition of what a looped path is, it's easy to realize that all valid one-to-one shortest path solutions will be without loops. Hence, looped paths are a non-issue for single shortest path algorithms. In fact, the only time one encounters loops in a single shortest path

algorithm is when there is the existence of a negative cost loop, in which case the problem becomes unsolvable.

A second, third, fourth, etc. shortest path though could involve a loop in the path. Thus, 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 contain 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 10 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, and 8.

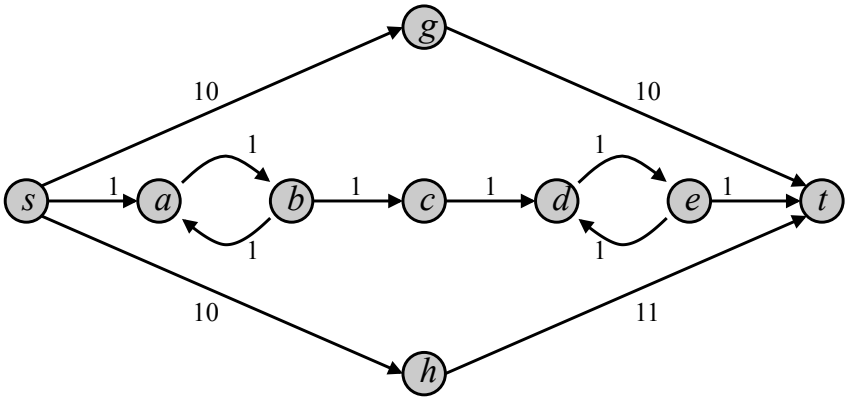


Figure 10. Network for illustrating *k*-shortest paths with and without loops

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 transmission lines, 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 simple paths.

11. 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 will still work in these cases, but only in certain conditions. If a network contains any negative loop, then the *k*-shortest path problem is impossible to solve for paths with loops. This is because a negative loop allows for a path with arbitrarily large negative length to be created, thus a ranking of paths cannot be made. In general, the shortest path between two nodes is considered to be undefined in the presence of negative cycles (Yen 1971, Fakcharoenphol and Rao 2006).

One must be careful when solving a path problem on a network with negative cost arcs, even if there are no negative cycles. Dijkstra's algorithm fails on graphs with any negative cost arcs, and it is easy to prove that adding a fixed cost to all arcs to make them all positive will not result in an equivalent optimal answer. In the presence of negative arcs, it is necessary to use another shortest path algorithm, such as the Bellman-Ford algorithm (as long as there are no negative cycles, of course). Whereas Dijkstra's algorithm uses a best-

first-search approach, Bellman-Ford uses a breadth-first search. This results in an overall slower algorithm, with a naïve complexity of $O(mn)$. Since the number of arcs m is always greater than or equal to the number of nodes n (ie. $n \leq m \leq n^2$), then $O(mn) \geq O(n^2)$. Empirically, Bellman-Ford also tends to solve less quickly than Dijkstra.

Since *k*-shortest path algorithms all use a single shortest path algorithm somewhere as a subroutine, negative cost arc considerations apply equally to those problems just as they do for one-to-one shortest path.

Corridor design models may involve conflicting objectives, some that involve maximization (e.g. maximizing safety, historical preservation), and others associated with minimization (e.g. minimizing cost, environmental impact) and depending upon a set of importance weights used, may result in arcs which have a negative composite cost value. It is important to be aware that if negative cost arcs are present, one may have to either alter the solution method, or alter the weighting scheme to ensure all positive cost arcs.

12. K-Shortest Path Lengths

The *k*-shortest path lengths (KSP*l*) problem is a slight modification of the *k*-shortest path problem. Rather than determining a set of ranked paths, it instead finds the set of distinct path lengths of those ranked paths. In essence, it overlooks the fact that there may be multiple paths of the same length, and instead simply asks what are the different lengths of the paths in the KSP set. 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.* 2001b), 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 KSP algorithms can be used to solve the KSP*l* problem. Based on these worst-case complexity and experimental results, we do not consider KSP*l* algorithms to be of any use in multi-path corridor location.

13. 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 + ε 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 experiments have been published directly comparing Carlyle and Wood's algorithm to other *k*-shortest path algorithms.

14. Parallel Computing and Performance Metrics

Since it was first declared in 1965, Moore's Law (Moore 1965) has correctly predicted that the number of transistors on an integrated circuit, and thus computational power, would double every two years. Up until 2005, the additional transistors allowed both processor clock speeds to increase and more advanced instruction-level parallelism, which resulted in overall computational processing power of single-threaded code to follow Moore's Law. But around 2005, heat dissipation became a limitation on further practical increases in processor clock speeds, and instead processor makers began using higher transistor densities to pack multiple computer processors onto a single chip, known as multi-core processors.

Innovation in multi-core processors has allowed the progress of Moore's Law to continue, starting with dual-core, then quad-core, and now in 2014 some desktop computer processors have 8 cores capable of simultaneously handling 16 threads. Even mobile phones

exist with quad-core processors. But as scientists and programmers run programs on larger and more complicated data sets, they can no longer rely on simply higher processor clock speeds to improve the performance of their codes (Sutter 2005). Instead, to continue to reap the benefits of Moore's Law, programmers must now write their programs to take advantage of multi-core processors and increasingly inexpensive parallel computing clusters. This requires looking at ways to incorporate concurrency and multi-threading into codes, so that independent control flows can be distributed over numerous processors. Some algorithms are easier to "parallelize" than others; for example, a Monte Carlo simulation entails running a model numerous times with various different initial conditions as input (Hägerstrand 1965, Clarke and Gaydos 1998). Since each model simulation is an independent calculation to every other model simulation, the individual computations can simply be assigned to separate processors without the need for any communication between the processors while computing. Unfortunately, most programs are not so simple to make parallel, and more likely a programmer will have to split data and tasks into numerous pieces, perform some distributed concurrent partial computation, communicate intermediate results between various processors and then define new task and data pieces for further partial computation, continuing until the computation is complete; a sort of wash, rinse, and repeat. Communication speeds between processors become a performance bottle-neck, trade-offs between fine-grained and coarse-grained parallelism must be calibrated for running on different hardware configurations, and race conditions and deadlocks open up a whole new Pandora's box of software bugs that must be resolved.

But aside from the nuts and bolts associated with parallel programming, some algorithms are fundamentally difficult to split into concurrent processes. For example, parallel irregular

graph traversal algorithms remain an active area of research, as these are inherently difficult to code (Bader 2008, Cong *et al.* 2008, Chhugani *et al.* 2012, Merrill *et al.* 2012). Section III of this dissertation addresses an example of one such irregular graph traversal: the parallelization of a depth-first-search (DFS) path algorithm that has been proven to be inherently sequential (Reif 1985), and thus difficult to implement in parallel.

Evaluations of parallel programs are often based on metrics such as speedup and parallel efficiency, which are used to compare the parallel performance to its serial counterpart. Letting p be the number of processors, and T_p be the execution time of a parallel algorithm on p processors, then the speedup S_p and parallel efficiency E_p are defined as

$$S_p = \frac{T_1}{T_p} \quad (5)$$

$$E_p = \frac{S_p}{p} \quad (6)$$

T_1 is the execution time for the serial (1-processor) version of the algorithm, and in the ideal scenario, $S_p = p$ and $E_p = 1$, although this rarely occurs in parallel computation applications except for trivially simple cases such as Monte-Carlo simulation. In addition to high speedup values, one also looks for a linear trend as the number of processors increases. This would indicate that a method is scalable to a very high number of processors while maintaining a good speedup. As with perfect speedup, linear speedup trends are typically not possible to maintain except in the case for very simple problems, since speedup is limited by the amount of parallelism that exists in a problem instance or program. Amdahl's Law (Amdahl 1967) states that the speedup of a parallel program is limited by the amount of work that must be done sequentially. As a consequence, parallel programs may exhibit linear speedup as the number of processors increases until the smallest level of parallel granularity

is reached, at which point the speedup will plateau to some limit no matter how many additional processors are available.

B. Single Shortest Path Literature Review

There is some debate as to who developed the first linear programming formulation of the shortest path problem. The first to *publish* a formulation was a 1956 paper by Alex Orden, where he shows that the shortest path problem can be formulated as a special case of the transshipment problem (and which he cleverly solves as a classical transportation problem) (Orden 1956). In 1957, George Dantzig published a paper in which he gives an LP formulation for numerous optimization problems, including the shortest path problem (Dantzig 1957). In that paper though, he mentions that he had originally presented the content of that paper at the 1955 ORMS National Meeting in Los Angeles. The abstracts for all presentations at that meeting are available in the November 1955 issue of the journal Operations Research (Dantzig 1955). While that abstract lists all of the other problems Dantzig later discussed in his 1957 paper, it makes no mention of the shortest path problem. It seems that a definitive answer as to who came up with the first LP formulation of the shortest path problem may forever remain a mystery.

In a response to the 1957 paper by Dantzig, George Minty (1957) proposed a clever 'analog computer' method for solving the shortest path problem. Using the same example Dantzig used in his paper of sending a package from Los Angeles to Boston, he suggests the following:

> Build a string model of the travel network, where knots represent cities and string lengths represent distances (or costs). Seize the knot 'Los Angeles' in your left hand and the knot 'Boston' in your right and pull them apart. If the model becomes entangled, have an assistant untie and re-tie knots until the entanglement is resolved. Eventually one or more paths will stretch tight – they then are alternative shortest routes.

> Dantzig's 'shortest-route tree' can be found in this model by weighting the knots and picking up the model by the knot 'Los Angeles'.
>
> It is well to label the knots since after one or two uses of the model their identities are easily confused.[5]

Soon thereafter, Richard Bellman (1958) published a dynamic programming algorithm to solve the shortest path problem, building on ideas earlier published by Lester Ford Jr. (1956). This new algorithm, now known as the Bellman-Ford algorithm, uses a label correcting breadth-first search approach, capable of finding the shortest path on directed or undirected networks with both positive and negative costs (so long as there are no negative loops). With a worst case complexity of $O(nm)$ (recall n is the number of nodes, m is the number of arcs, and $m \leq n^2$), this algorithm has a worse complexity than Dijkstra's algorithm, and in practice also runs more slowly. Even so, it is still used at times to this day because it will work on networks with negative cost arcs, and if there exist more than one shortest path it will find one with the fewest number of arcs.

In the next year, Edsger Dijkstra (1959) published his famous shortest path algorithm. His label setting, best-first search approach improves over the Bellman-Ford algorithm in terms of worst-case complexity ($O(n^2)$) and real-world runtime, while remaining simple to implement. The algorithm's only drawback is that it fails on networks that contain negative arc costs, whereas the Bellman-Ford algorithm does not. Even so, negative cost networks are rare in spatial problems, thus Dijkstra's is by far the most commonly used general-purpose shortest path algorithm.

For solving the shortest path from every point to every other point in a graph, The Floyd-Warshall algorithm was developed independently by Robert Floyd (1962) and Stephen Warshall (1962). It is a matrix oriented dynamic programming algorithm that solves all-to-

[5] This can be considered the fastest of all shortest path algorithms. Not accounting for the time to "build" the network, and assuming constant stretching speed, the problem is solved in approximately linear O(m) time!

all shortest paths in $O(n^3)$ time and $O(n^2)$ space. Alternatively, one can also solve Dijkstra's algorithm n times with Fibonacci heaps, resulting in a smaller $O(n(m + n \log n))$ worst-case time complexity.

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 A* algorithm, which uses a spatial approximation heuristic to dramatically speed up finding one-to-one shortest paths. 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, also 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.

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 subroutine, allowing the user to choose which implementation of Dijkstra they will use for their complexity analysis.

A newer shortest path algorithm called the Two-Q algorithm has shown some strong results in recent shortest path literature. Devised by Pallottino (1984), it is based on an earlier method published by Pape (1974). Both algorithms are label correcting, and so like the Bellman-Ford Algorithm, optimality is not guaranteed until the entire graph is searched. This is not a concern when searching for one-to-all shortest paths (shortest path trees), but can be a hindrance if only one-to-one shortest path queries are of interest. They essentially work by incrementally growing a graph, moving nodes efficiently between two data structures (one for unlabeled nodes, and the other for labeled nodes). The Pape algorithm uses a LIFO stack for the unlabeled nodes and a FIFO queue for the labeled, whereas the Two-Q algorithm uses (as the name implies) two queues. While they both show similar performance in practice, the Two-Q algorithm is generally preferred because the original Pape algorithm has an exponential worst-case complexity, while the Pallotino Two-Q has a polynomial worst-case complexity. Despite strong results in numerous comparative studies (see next paragraph), the Two-Q algorithm seems to be largely ignored in the textbook literature on shortest path algorithms.

Numerous studies have been published comparing the performance of various shortest path algorithms with each other (Cherkassky *et al.* 1996, Zhan and Noon 1998, Zhan and Noon 2000, Zeng and Church 2009). Cherkassky did an extensive comparison on various random and grid graphs, with conclusions that on non-negative networks, Dijkstra using double-buckets consistently performed the best. The author does note that Pape and Two-Q

perform exceedingly well on square grid problems, but suffer greatly in problems that contain many violations of the triangle inequality (i.e. non-spatial). Zhan and Noon (1998) followed up with a study using road networks. They concluded that finding one-to-all shortest paths was fastest with Pape and Two-Q, although in many cases, one-to-one shortest paths are found fastest with Dijkstra's algorithm. They clarified this statement in a later paper (Zhan and Noon 2000), where they found if the origin and destination are relatively close to each other compared to the overall size of the graph, then Dijkstra's was typically better (since it could stop once the destination was found), whereas if the origin and destination are close to spanning the size of the graph, then the incremental graph algorithms are better. In their paper, they define probabilistic cutoffs for where one or the other is faster.

One glaring hole in the Zhan and Noon studies though is that they did not include the A* algorithm in their experiments. Since road networks are spatial, A* is a valid heuristic for speeding-up the ability to calculate one-to-one shortest paths. Zeng and Church (2009) aimed to fix this by repeating Zhan and Noon's experiments but also including A* in the race. The study found that A* vastly outperformed the various implementations of Dijkstra as well as Two-Q, mainly because A* queried far fewer nodes than the other methods. This reasoning was examined much earlier by Golden and Ball (1978), where they found that on $R = 1$ (queen's move) lattice graphs, A* searches on average expanded on only 8.3% of the nodes than would a normal Dijkstra's algorithm.

A recently published paper describes a new algorithm which alters the Two-Q algorithm from being label correcting to label setting (Leng and Zeng 2009). Called MiLD-Two-Q, the changes are intended to correct the inefficiencies Two-Q has with finding one-to-one shortest paths. Their experimental results showed the new algorithm to be very successful at

this, resulting in much improved performance over Two-Q, but no research has yet been documented comparing MiLD-Two-Q experimentally with A*. Recent algorithms have also been developed for shortest path computation using parallel computing. Examples include an implementation of Dijkstra's algorithm by Crauser *et al.* (1998), a parallel algorithm for shortest path on planar graphs (Träff and Zaroliagis 2000), a parallel method for finding shortest path trees (Delling *et al.* 2013) and a Δ-stepping algorithm by Meyer and Sanders (2003) that is capable in certain instances of exhibiting linear computational complexity.

C. Near-Optimal and K^{th} Shortest Path Literature Review

The first published descriptions of applications in need of shortest path alternatives came about in the early 50's, both with regards to finding an alternate shortest route when part of the shortest route is blocked. A paper by Trueblood posed this question in the context of freeway usage (Trueblood 1952), and another by Jacobitti did likewise for automated call routing over a nationwide phone network (Jacobitti 1955). Quoting Jacobitti:

> When a telephone customer makes a long-distance call, the major problem facing the operator is how to get the call to its destination. In some cases, each toll operator has two main routes by which the call can be started towards this destination. The first-choice route, of course, is the most direct route. If this is busy, the second choice is made, followed by other available choices at the operator's discretion. When telephone operators are concerned with such a call, they can exercise choice between alternate routes. But when operator or customer toll dialing is considered, the choice of routes has to be left to a machine. Since the "intelligence" of a machine is limited to previously "programmed" operations, the choice of routes has to be decided upon, and incorporated in, an automatic alternate routing arrangement.

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.

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 1961b). This principle is the foundation of subsequent more advanced algorithms by Yen, Katoh, and others.

Pollack (1961a) 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 $O(kn^2)$. 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 Fox's algorithm run with a complexity of $O(m + kn \log n)$. The present day best-known KSP with loops 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 1984a, Azevedo *et al.* 1993, Azevedo *et al.* 1994, Martins *et al.* 1998, Jiménez and Marzal 1999, Martins *et al.* 1999a, Martins *et al.* 1999b, Martins and Pascoal 2000, Jiménez and Marzal 2003).

The next major loopless *k*-shortest path approach came when Yen (1971) presented his deviation 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. Initially, the shortest path is found, then the algorithm selects each node in that path and calculates the shortest path from the node 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 (1972) improved Yen's algorithm to cut the number of computations in half by defining criteria that prevent the redundant computation of shortest paths. 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 published by Perko (1986), and by Martins and Pascoal (2003).

Shier's algorithm for finding the *k*-shortest path *lengths* (KSP*l*) was also developed around the same time (Shier 1974, Shier 1976). Since a KSP algorithm can also be used to compute KSP*l*, Shier compared his approach to Yen's algorithm (Shier 1979). His

experiments showed though that his algorithm was faster than Yen's for only dense networks of density $m/n \geq 10$. With later advancements in implementations of Yen's algorithm, Guerriero *et al.* (2001a) 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 actual runtime.

The next major development in KSP methods came from Katoh *et al.* (1982), who 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 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 algorithm, including modifications for improvement in efficiency, some pseudocode, and suggested data structures.

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 method.

The more recent developments in the k-shortest path problem come from 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. More information on the replacement paths problem is found in Hershberger *et al.* (2007b).

D. Multi-Objective Shortest Paths Literature Review

Multi-objective optimization deals with the use of two or more objectives. For corridor location modeling, this involves the use of a path model/algorithm to handle a number of objectives. Most of past work in multi-objective path modeling is described for the use of two objectives, as they equally apply to problems of larger dimensions. Accommodating more than two objectives requires special bookkeeping that is somewhat more sophisticated than what is needed for two objectives, as some facets which intersect neighboring solutions in three or higher dimensions may not be on the boundary of the convex polytope, but

instead lie in the interior (Solanki 1986). However, aside from this issue many of the techniques used to solve for tradeoffs in two objectives can be relatively easily expanded for three or more objectives. Because of this, researchers have concentrated on the problem of resolving bi-objective problems. This dissertation takes the same tack, and restricts the discussion to bi-objective path problems.

In 1979, three different papers appeared in the published literature that addressed the problem of finding efficient points to biobjective optimization problems. Dial (1979) developed a process that involved finding up to a pre-specified number of supported points to a biobjective shortest path problem, Aneja and Nair (1979) developed an approach to find all supported points to a biobjective transportation problem, and Cohon *et al.* (1979) developed a process to find non-dominated solutions to biobjective linear programming problems. Overall, all three techniques are quite similar, but do differ in their main focus. For example, Dial's approach runs until it finds a certain number of solutions or finds the complete tradeoff curve. The choice of problems solved, and hence the resolved tradeoff curve is based upon a recursion formula taking problems in order. Aneja and Nair's approach is similar to that of Dial's except it does not stop until it has resolved all parts of the tradeoff curve. Cohon et al. (1979) show how lower and upper bounds on the tradeoff curve can be defined as supported points are added to the tradeoff curve. This allows one the opportunity to resolve at each iteration that portion of the curve with the greatest estimation error. This technique is called the Non-Inferior Solution Estimation (NISE) technique. The NISE technique will either generate all supported points on a tradeoff curve within a set estimation bound limit, or can be executed to completion to generate all supporting points as suggested by Aneja and Nair. IP problems can also have non-convex, non-inferior solutions

known as *unsupported* solutions. Unsupported solutions are much more difficult to compute, as solving for those is equivalent to adding a knapsack constraint to the problem, which has been proven to be NP-hard (Garey and Johnson 1979). Unsupported (non-convex) non-dominated solutions may exist in objective space inside of triangular regions between each supported solution. Coutinho-Rodrigues *et al.* (1999) called these viable unsupported solution regions the "duality gap", while Medrano and Church (2014) instead calls them Boundary of Unsupported Solution Search (BUSS) regions. These so-called duality gap or BUSS regions are depicted in Figure 11 as shaded triangles.

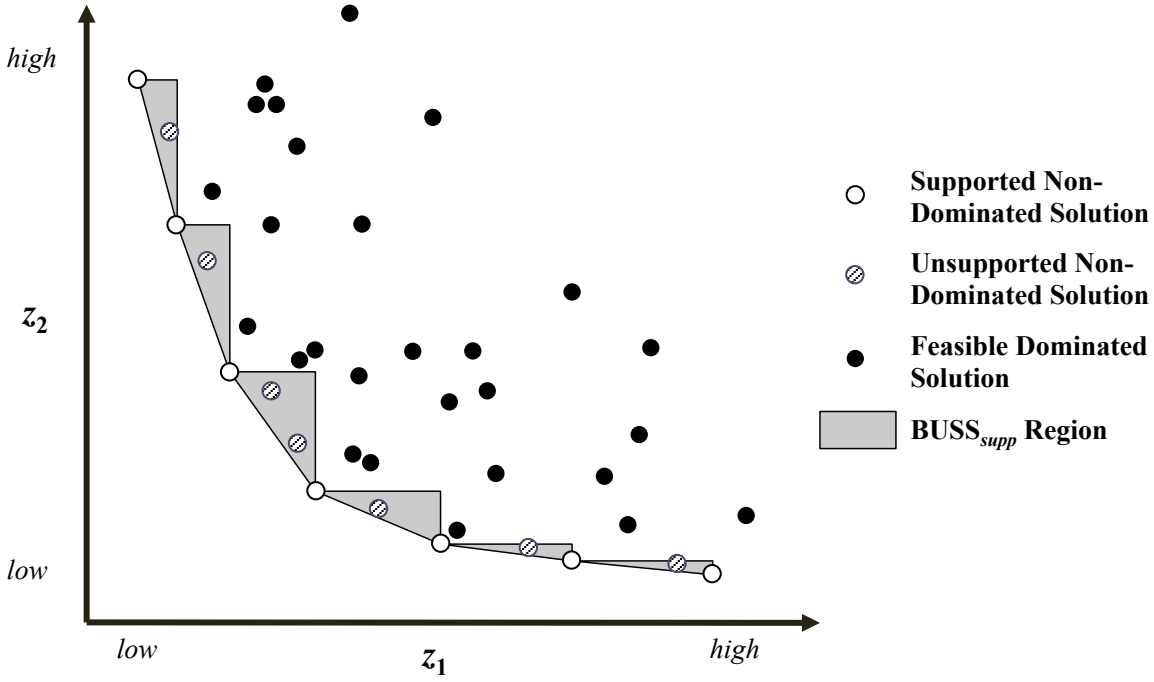


Figure 11. Categorization of Bi-Objective Solutions to a discrete problem in objective space

It is important to note that unsupported solutions still represent an optimal trade-off between the two objectives, but are much more difficult to find since they cannot be found as optima to a weighted, composite single-objective shortest path problem. Any feasible solution that does not fall in the BUSS is dominated by at least one other supported non-dominated solution. A feasible solution in the BUSS is not guaranteed to be non-dominated

as a feasible solution within the BUSS may be dominated by another solution within the same BUSS.

Early work on solving the complete Pareto-optimal solution set of a discrete bi-objective shortest path (BSP) problem includes methods using path/tree handling, parametric search, label-setting and label correcting algorithms. An extensive literature review by Raith and Ehrgott (2009) covers most of this work. One approach not covered in that review is the K^{th}-Shortest Path (KSP) method of Coutinho-Rodrigues *et al.* (1999). The Coutinho-Rodriques approach, called GAPS, is a two-phase method that first employs the NISE method of Cohon *et al.* (1979) using a classic shortest path algorithm to generate all supported non-dominated solutions. The second phase enumerates paths with a KSP algorithm in the regions between the supported solutions in order to identify all unsupported solutions. In the second phase they used a very fast augmented network KSP algorithm developed by Azevedo *et al.* (1994) that identifies both paths with loops and paths without loops. While looped paths will always be dominated in a bi-objective problem by paths without loops, path algorithms that allow loops are less restrictive than loopless path algorithms, and are thus generally faster. In their case, the choice of a fast KSP algorithm that allows for loops was based upon the premise that the cost of enumerating the paths including those with loops would outweigh the burden of eliminating them as candidates for the Pareto-optimal frontier. The reason why this premise is plausible is that any path with a loop will be dominated by at least one without a loop. Therefore, it is not necessary to detect whether a loop occurs in a path as merely comparing its objective values to other paths that have been generated can eliminate the path. The conjecture that a "paths with loops" approach in generating a Pareto-optimal frontier would be faster than using the fastest loopless KSP

algorithm has never been tested, although we suspect the cost of enumerating a combinatorially large number of near-shortest looped paths would outweigh the benefit of the fast algorithm.

Raith and Ehrgott (2009) compared the state-of-the-art biobjective shortest path algorithms of the time, including a label-correcting method by Skriver and Anderson (2000) and a label-setting method of Martins (1984b), on various types of networks. They also proposed a new approach, which was based on enumerating Near Shortest Paths using an algorithm developed by Carlyle and Wood (2005). In addition, all three approaches were cast into two-phase algorithms. Over the range of problems they analyzed, Raith and Erghott concluded that the best approaches were the two-phase methods of the label-setting and label-correcting algorithms. Another variation on the Martins (1984) label-setting approach, called NAMOA*, was developed by Mandow and Pérez de la Cruz (2005), who applied an A* heuristic to the multiobjective label-setting algorithm. As stated earlier, Raith and Ehrgott did not review or test the GAPS method nor NAMOA*. Presumably, they were unaware of those techniques at the time of their work.

Other researchers have approached the biobjective shortest path problem heuristically in order to solve the problem in polynomial time. Ghoseiri and Nadjari (2010) provide a thorough literature review of these polynomial bounded methods, so we will not review them here. In their paper, they propose an ant colony optimization heuristic, although their tests of this approach produced mixed results. Two heuristic approaches not reviewed in Ghoseiri and Nadjari that deserve mention are the interactive constrained shortest path (CSP) approach published by Current *et al.* (1990), and a Tchebysheff distance metric solution search method originally proposed by Steuer and Choo (1983). The interactive CSP

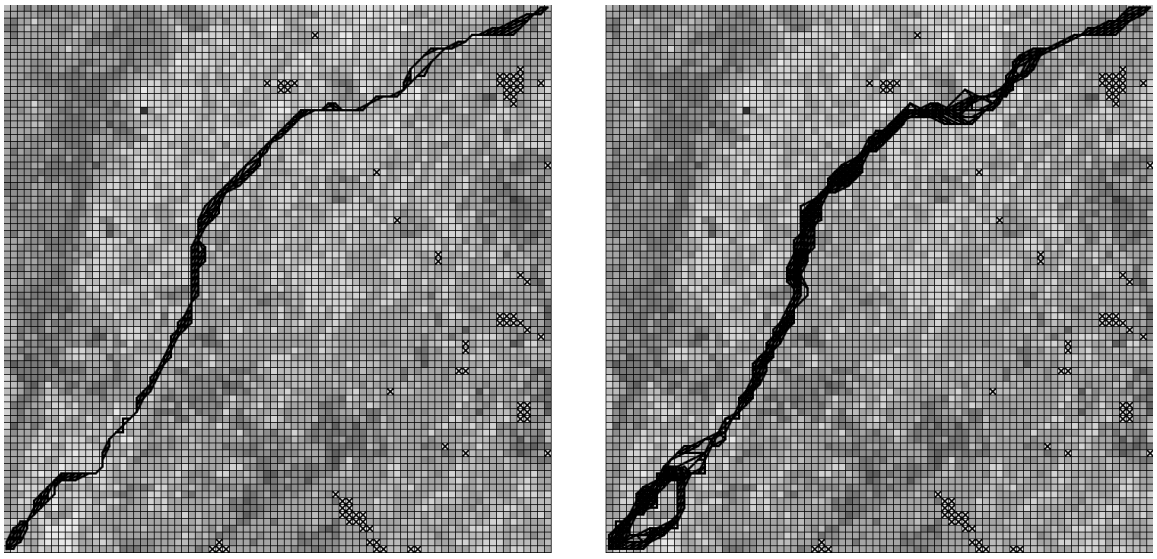
approach was designed to assist a decision-maker in generating and selecting possible paths options. After the supported solutions have been generated, the decision maker can explore possible path alternatives by the use of a constrained shortest path model solved via Lagrangian relaxation. Although generating an entire Pareto frontier for a large biobjective path problem using this approach would be computationally intensive, they assumed that a decision maker might require only a few alternatives in which to make a decision.

The Tchebysheff distance metric heuristic involves a different approach to interactively searching the objective space for exact solutions. Originally developed by Steuer and Choo (1983), it was first applied to location problems by Solanki (1991), and more recently applied to a multi-objective hazmat routing problem by Huang *et al.* (2008). The method uses a general integer-programming solver and a modified Tchebysheff objective to search for a solution in a user-selected region of the solutions space, and can be applied to a variety of MIP problems. If an augmented Tchebysheff distance matrix is used it guarantees all solutions returned are non-dominated, omitting any weakly non-dominated solutions.

Recent published work specifically on corridor location over GIS terrain networks have primarily solved a weighted sum single-objective problem, using various weight combinations to determine a subset of the supported Pareto-optimal solutions (Atkinson *et al.* 2005, Bagli *et al.* 2011). Aissi *et al.* (2012) used a polygon data representation to generate a connectivity graph of 1,356 vertices and 3,965 edges, then used the Martins (1984) approach to solve a multiobjective corridor problem. They chose origin and destination points relatively close to each other on the data set, which resulted in only three distinct non-dominated paths found.

E. Alternative Paths Literature Review

A multitude of optimization models have been proposed in the literature, which address the problem of generating several good quality alternate paths. The adequacy of each model strictly hinges on the application at hand and the goals it strives to achieve. A brute force way of generating alternative paths is via a k-shortest path algorithm, as covered in section II.C. These enumerative algorithms find large applicability in a variety of settings for small or sparse networks, such as the modeling of transportation, communication, and distribution networks. However, this approach often fails in providing paths that are truly different in terms of spatial dissimilarity. On large, dense networks encountered in terrain modeling the *k*-shortest path algorithm and related algorithms produce solutions that are only small perturbations of the optimal path. Such alternatives are of little use to planners when facing a public forum as one must be able to convey that a wide variety of spatial configurations were considered in the analysis of alternatives. As a consequence, in corridor location an overwhelmingly large number of paths must be generated before obtaining a few spatially dissimilar alternatives. For example, Figure 12 displays all paths on an 80x80 GIS raster grid that are up to (a) 0.3% and (b) 0.8% more costly than the least-cost path. The first instance represents a total of 4,459,050 distinct paths, yet essentially all traverse similar routes. The second instance displays a total of 160,650,434,203 distinct paths and took more than two days to compute on a 2.8 GHz Intel Xeon server running the very fast ANSPR0 Near Shortest Paths algorithm by Carlyle and Wood (2005). Even with the over 160 billion distinct shortest paths, very few spatially distinct alternatives are generated.



(a) (b)

Figure 12. All path options within (a) 0.3%, (b) 0.8% of the shortest path cost

A less computationally burdensome way of generating alternatives consists in modifying the objective function rather than adding constraints to the optimization model. A simple and efficient procedure based on this concept is the Iterative Penalty Method (IPM), initially developed by Ayad (1967) and Turner (1968), and later refined by Huber (1980). This approach has been used for hazardous material routing problems (Johnson *et al.* 1993), and for dynamic intelligent transportation systems (Rouphail *et al.* 1995). The Iterative Penalty Method solves shortest path problems sequentially, but after the generation of each new path, the arcs or nodes comprised in the solution are penalized so as to discourage their use in subsequent iterations. The mechanism of penalization may take a number of forms: arcs, nodes or both can be penalized; penalties can be applied by addition, multiplication or a combination of both; elements can be penalized only once or every time they are used in a solution. The effectiveness of the overall methodology is hence strictly dependent on the penalization strategy in use as well as on the magnitude of the applied penalties.

Some of the drawbacks of the *k*-shortest path approach and of the IPM can be overcome if they are used in combination with other methodologies, so that their efficiency and implementation simplicity can be exploited to generate a large selection of paths, while the task of guaranteeing and evaluating dissimilarity is handled separately. Such an idea is embodied in the minimax method proposed by Kuby *et al.* (1997), and in the *p*-dispersion model developed by Akgün *et al.* (2000). The minimax approach uses a *k*-shortest path algorithm to generate a large set of candidate paths, which are then evaluated for insertion in the "differentiated" path subset. The shortest path is first selected. Each subsequent path is chosen by solving a bicriterion optimization problem. Namely, the selected path is the one minimizing a linear combination of length and similarity with all the paths already inserted in the differentiated subset. The degree of dissimilarity between paths is measured in terms of "not shared" distances. Akgün *et al.* (2000) construct an initial set of candidate paths using a *k*-shortest path algorithm or IPM. They then extract a subset of dissimilar paths by solving a discrete *p*-dispersion model (Kuby 1987, Erkut 1990). Given a set of candidate points, the *p*-dispersion model consists in selecting a subset of *p* points that maximizes the minimum distance between any pair of selected points. When used to select spatially different paths, the *p*-dispersion model must be supplied with a notion of distance among solutions. Akgün *et al.* (2000) measure path difference through a similarity index based on shared links. Measuring path difference by shared links is acceptable when the underlying network is a road network, since typically there is an inherent spatial dispersion between roadways. Conversely, on a raster it is possible for two closely spaced parallel paths to exist with no shared links. Such paths would not be considered different alternatives when scrutinized in a public forum. For this reason, it is better to use area difference, as area

difference can account for closely spaced non-coincident paths. *P*-dispersion techniques could be expanded to differentiate based upon a combination of shared links/nodes, and area. But, whatever the method used to measure similarity between paths, the *p*-dispersion problem is an NP-hard problem; therefore solving such a problem on a massive data set will likely require extensive computation. As shown earlier in this section, enumerating paths for a candidate set on a terrain network requires generating an enormous set of paths in order to achieve any spatial diversity. The next step then requires some measure of difference between all pairs of paths to be computed, be it some simple metric such as shared length or a better metric such as area difference. Then the last step is to solve the *p*-dispersion problem on the candidate set. Overall, this requires massive computation on large and difficult-to-solve problems.

A competitive approach more suitable for terrain networks is the Gateway Shortest Path (GSP) model proposed by Church *et al.* (1992). This approach is founded on the idea that each gateway shortest path between an origin-destination pair, *i.e.* the shortest path that is constrained to travel through a pre-specified node, generates an alternative route. Such a route can depart significantly from the shortest path, depending on the location of the selected gateway; furthermore, it carries a good guarantee of impact quality, being the best possible path going through that cell. All nodes not in the shortest path define an alternative gateway path route. Church *et al.* (1992) propose a fast method to efficiently generate all such paths. The method consists in computing two shortest path trees, one rooted at the origin and one rooted at the destination, and overlaying them to obtain the distance and area difference values for every gateway alternative. Hence, the overall methodology only requires running a modified shortest path algorithm twice and performing label additions to

generate up to $N - 2$ alternative paths, where N is the number of nodes in the network. Measuring the spatial or areal difference between a gateway path and the shortest path may be accomplished at very little extra computational expense through the use of an additional area label associated with each node, The area label tracks the area "under" the path up to that node. Path differences can be calculated by subtracting the "area" of the shortest path from the sum of the area labels of the gateway path. This is detailed in Lombard and Church (1993). Even though the GSP model is a form of a constrained path problem, it overcomes the major drawbacks and inefficiencies characterizing most constrained models. Computational experience on a real-world corridor location problem reported in Lombard and Church (1993) demonstrated the superiority of the GSP model over the IPM, both in terms of computational effort and solution quality. The gateway concept was later applied in the cartographic modeling methodology by Lee and Tomlin (1997) to examine the implications of alternative routing criteria, and is also now a common feature in modern online mapping tools (i.e. Google Maps, Bing Maps, OpenStreetMap) for interactively generating alternative routes (Luxen and Vetter 2011).

III. Composite Single-Objective: Parallel Near Shortest Paths

A. Serial Near Shortest Paths

In their 2005 paper, Carlyle and Wood present two different algorithms for finding loopless Near Shortest Paths (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 (1999) 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, the algorithm can gain speed and reduce memory consumption through graph reduction: trimming unnecessary nodes from the graph that are guaranteed to not be a part of any NSP. 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), one 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 node v, if $d'(v) + d''(v) > D$, then there cannot exist any NSP that goes through that node. That node can be eliminated from the network, as well as any arcs connected to that node, thereby reducing 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.

ANSPRO Algorithm

DESCRIPTION: An algorithm to solve loopless NSP.

INPUT: A directed graph $G = (V, E)$ in adjacency list format
s = starting node
t = ending node

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

DEFINITIONS:
nextEdge(v) points to the next edge in a linked list of edges directed out of v.
nextEdge(v).reset() returns the pointer to the first edge in the linked list of edges from v.
$\tau(v)$ is 1 if vertex v is on the current subpath, 0 if otherwise.

INITIALIZATION
1) Solve one-to-all shortest path tree from node t.
2) For all nodes $v \in V$:
a. label $d'(v)$ = shortest path distance from v to t
b. $\tau(v) = 0$
c. nextEdge(v).reset()
3) D = (1 + ε) * d'(s);
4) theStack.push(s), L(s) = 0, τ(s)++

PROCEDURE
while(theStack is not empty) {
u = theStack.pop();
if(nextEdge(u).peek() ≠ null) {
uv = nextEdge(u);
if($L(u) + c(u,v) + d'(v) \le D$ and $\tau(v) == 0$)
if($v = t$)
print(theStack)
else
theStack.push(v)
$\tau(v)$++
$L(v) = L(u) + c(u, v)$
end
else
u = theStack.pop();
$\tau(u)$--
nextEdge(u).reset();
end
end
end

B. The Need for Parallelization

1. Characterizing Problem Size Growth

It is important to first establish the need for a parallelized approach to the NSP algorithm. Before considering any parallelization scheme, we wrote a serial 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 13 displays these results in a linear plot for the 20x20 network, and Figure 14 displays them on a logarithmic plot.

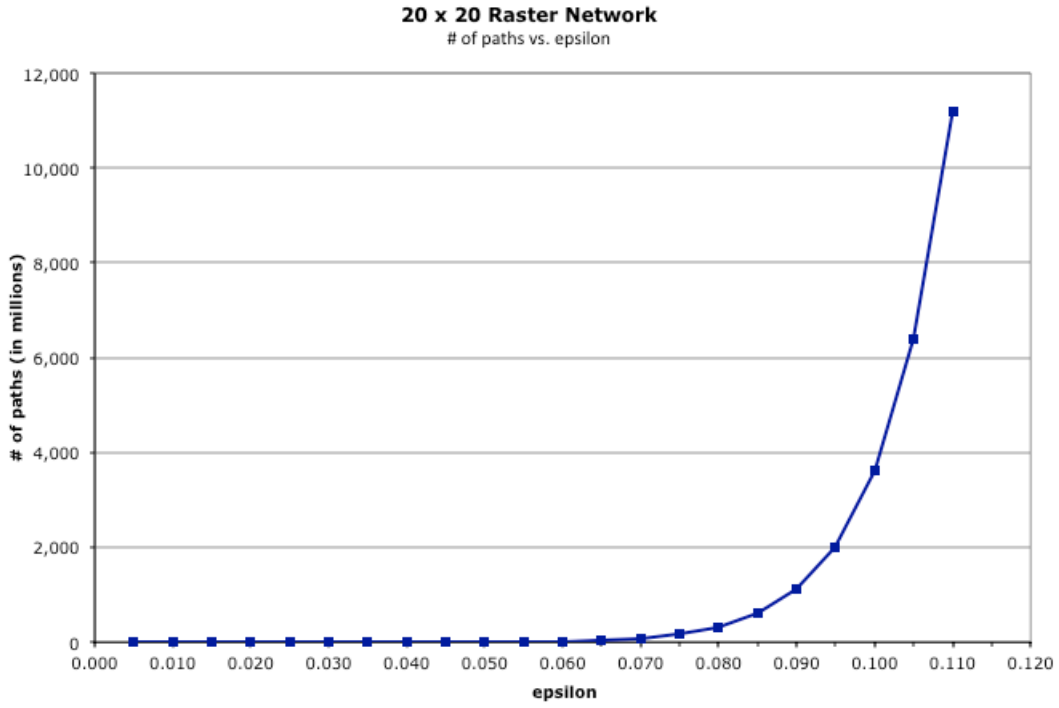


Figure 13. 20x20 network, number of paths generated by the ANSPR0 vs. epsilon

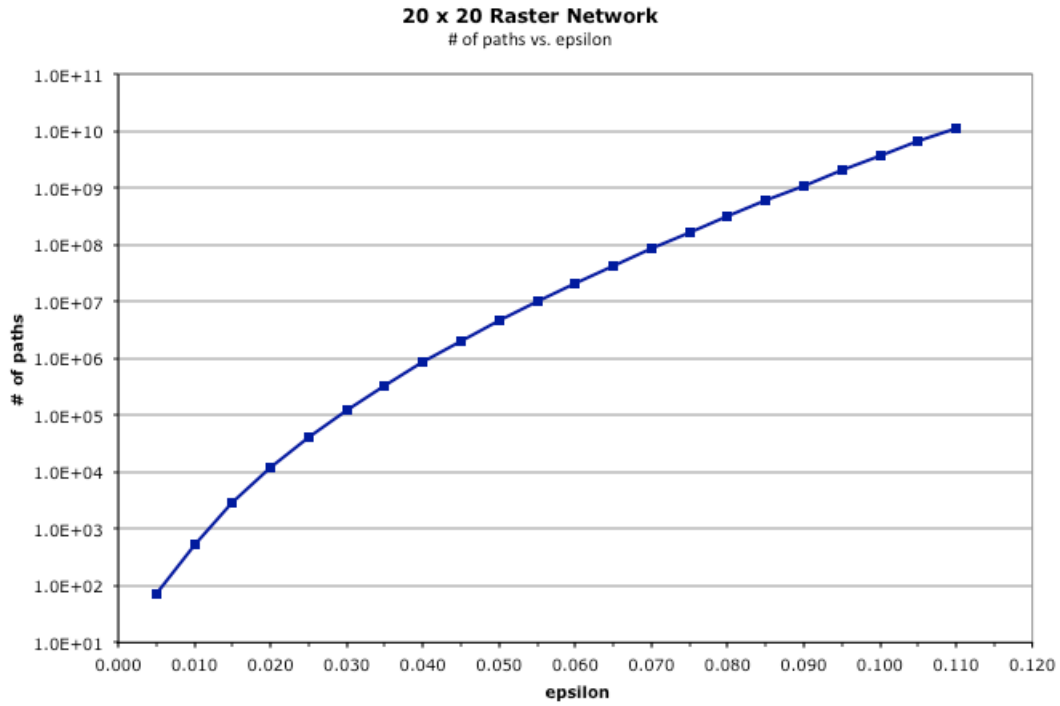


Figure 14. 20x20 network, log number of paths generated by ANSPR0 vs. epsilon

From Figure 13, 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 14, 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 the 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.

Figure 52 Figure 53 display computation time needed as a function of increasing values of ε on the same 20x20 data set. Figure 15 shows that the time to compute all paths within a threshold increases at a rapid rate as ε increases. Figure 16 plots the data on a logarithmic y-axis, and shows a straight line trend suggesting that this growth is indeed exponential in character.

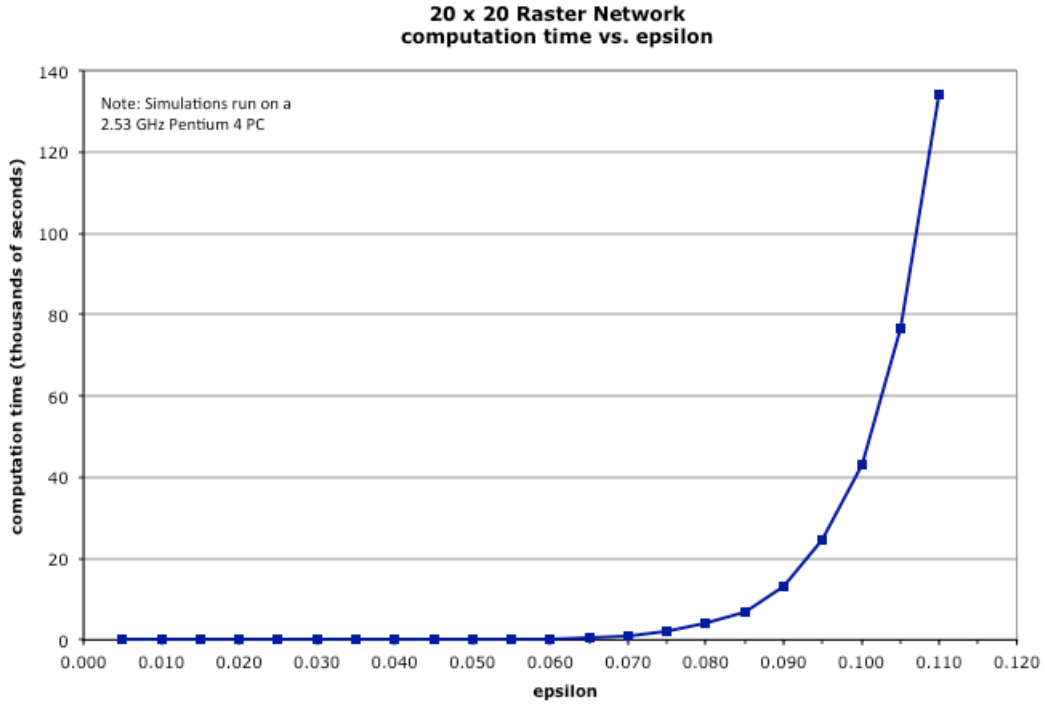


Figure 15. 20x20 network, computation time vs. epsilon

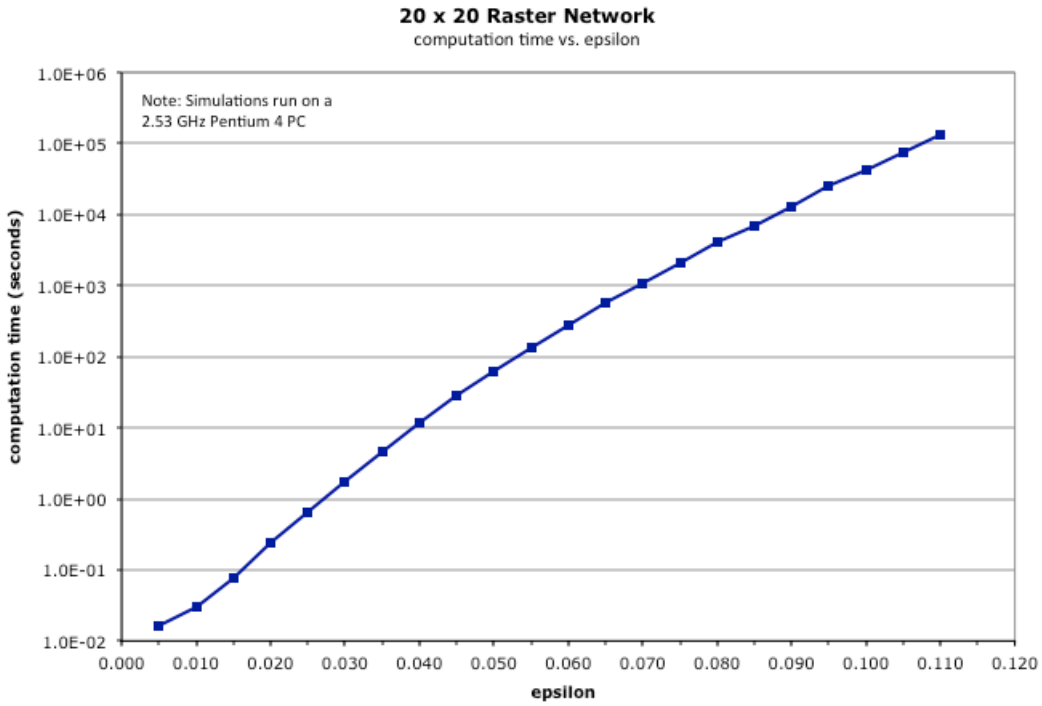


Figure 16. 20x20 network, log computation time vs. epsilon

2. 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.

C. Parallelizing Depth-First Search

The remainder of this section 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 17 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 18, 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.

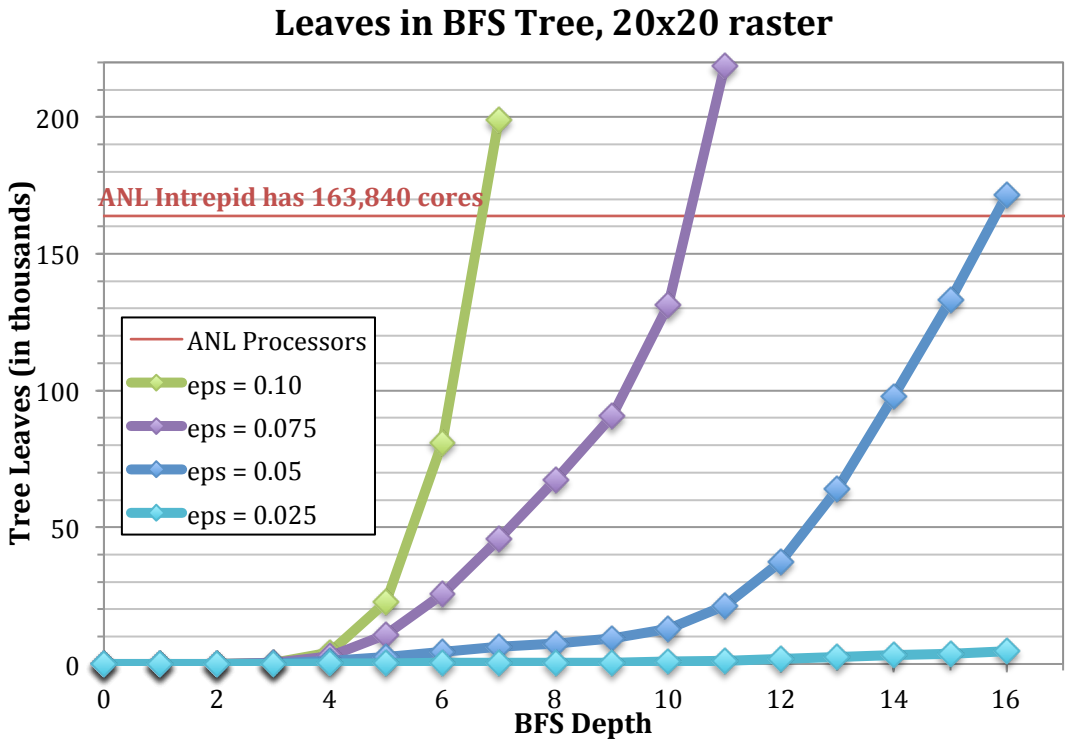


Figure 17. Leaves in BFS Tree, 20x20 raster, epsilon = 0.05

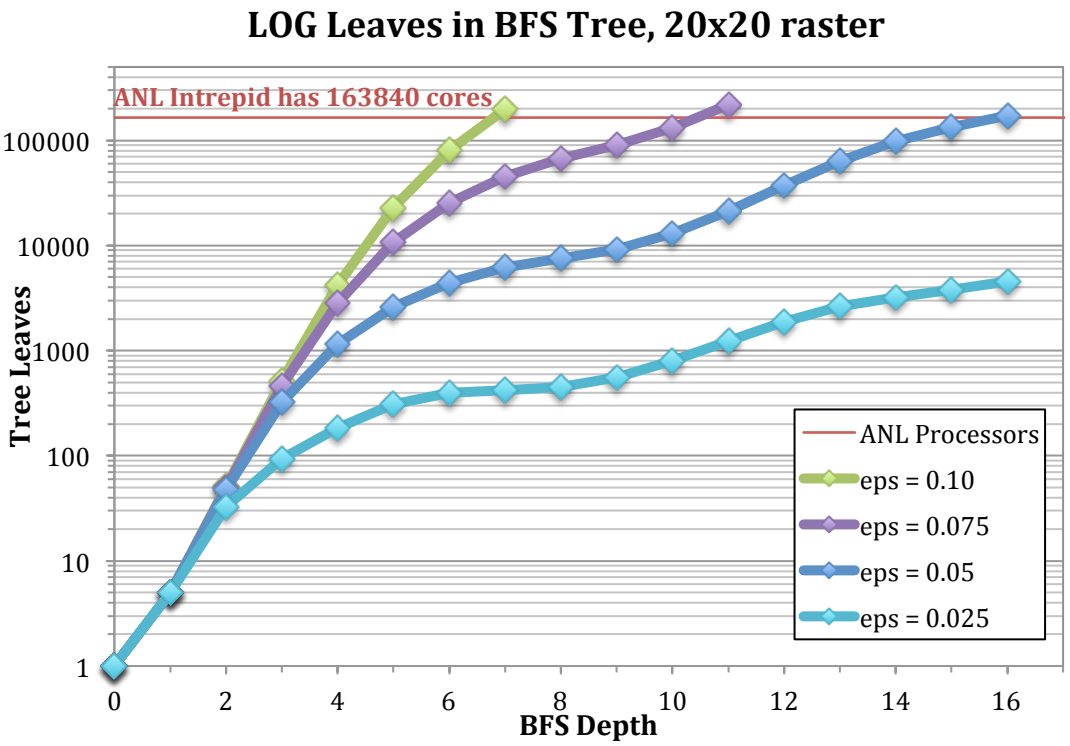


Figure 18. LOG Leaves in BFS Tree, 20x20 raster, eps = 0.05

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.

D. Analysis of Naïve BFS Work Distribution Implementation

Table 2 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 (p), 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 (S_p), and parallel efficiency (E_p).

Table 2. Naïve Parallel Algorithm Runtime Results on the 20x20 data

Epsilon	p	Time (sec)	Total Paths		Min Leaf Paths	Max Leaf Paths	S_p	E_p
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		

In the computational results shown in Table 2, 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 $\varepsilon = 0.05$, 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 3 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.

Table 3. Naïve Parallel Algorithm Runtime Results on the 80x80 data

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

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

$$parallel\ efficiency = \frac{Paths_{Total}}{p \times Paths_{Max}} \quad (7)$$

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 $Paths_{Max} \to P_{Total}/p$, then $parallel\ efficiency \to 1$. 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.

E. Distributing Workload

The load imbalances in this problem come from performing a depth-first search on a raster network, where the task workload sizes 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. The following is a description of several methods for load balancing and how well they could apply to a parallel approach of the NSP problem.

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 averages out to be somewhat similar. Adler et al. (Adler *et al.* 1995) show that when using randomized algorithms on normal or Poisson distributions of workload, 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. In a worst-case-scenario though, a large outlier could still result in an overall work imbalance.

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. This method is susceptible to the possibility of an abnormally large task being assigned last.

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, it is also far more difficult to implement than any of the other options. Even if implemented, one has to select a strategy for 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 theoretically just as effective as the

other two approaches (Blumofe and Leiserson 1994), although local communication priority is preferred in practice. This application 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, essentially a FIFO scheme.

Workload Prediction. One reason why it is difficult to balance the workload on depth-first-search irregular graph traversal algorithms is because the amount of work in each branch varies widely, and is unknown beforehand. We have considered working to identify heuristics that estimate the amount of work needed to resolve each leaf of the BFS tree. If effective, this would allow one to fathom/trim the tree in portions that have low expected work, while continuing to split leaves on portions with higher expected work. This would hopefully result in work chunks of more uniform size and avoid the inefficiencies caused by massively disparate work task sizes.

For the parallelized NSP algorithm, it appears that the first two approaches would suffer from the possibility of large work chunks superseding the benefits of random prescheduling or dynamic work scheduling. On our 80x80 data set (Table 2), even dividing the work into 38 chunks, the maximum sized still accounted for 68% of the total paths. This is far from the Gaussian or Poisson distribution that is necessary for random prescheduling to be effective. Dynamic work stealing has no theoretical drawbacks if implemented properly, but is exceedingly difficult to program for graph problems. In looking for an optimal balance between performance gains and ease of implementation, workload prediction heuristics for the purpose of developing the BFS tree only in portions with a high-expected workload

seem most promising in being an efficient method for more evenly distributing the workload across processors.

Additionally, 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 workload prediction / work stealing approach could show promise in giving a relatively even initial work distribution, then leveling task loads towards the end using dynamic work stealing. Any hybrid approach would be the most difficult to implement, as it requires developing several approaches, as well as cooperatively integrating them together.

F. Further 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.

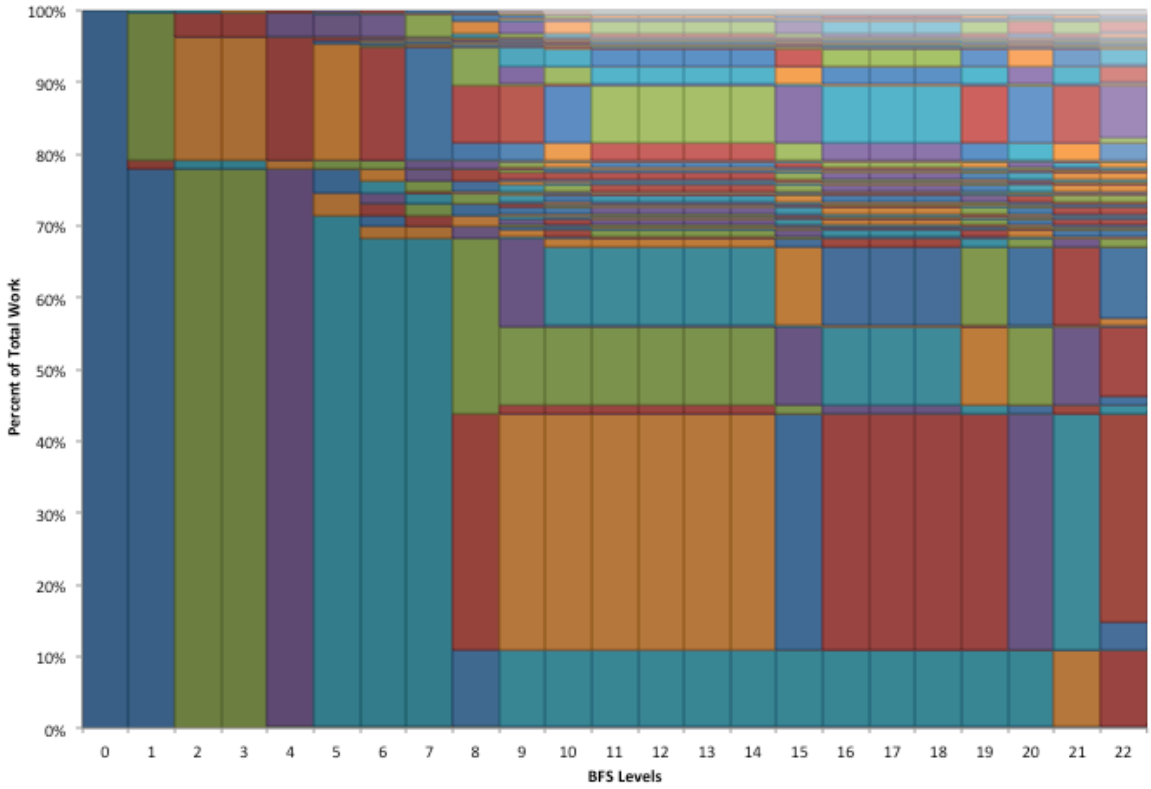


Figure 19. Work Distribution for Varying BFS Tree Levels

Figure 19 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.

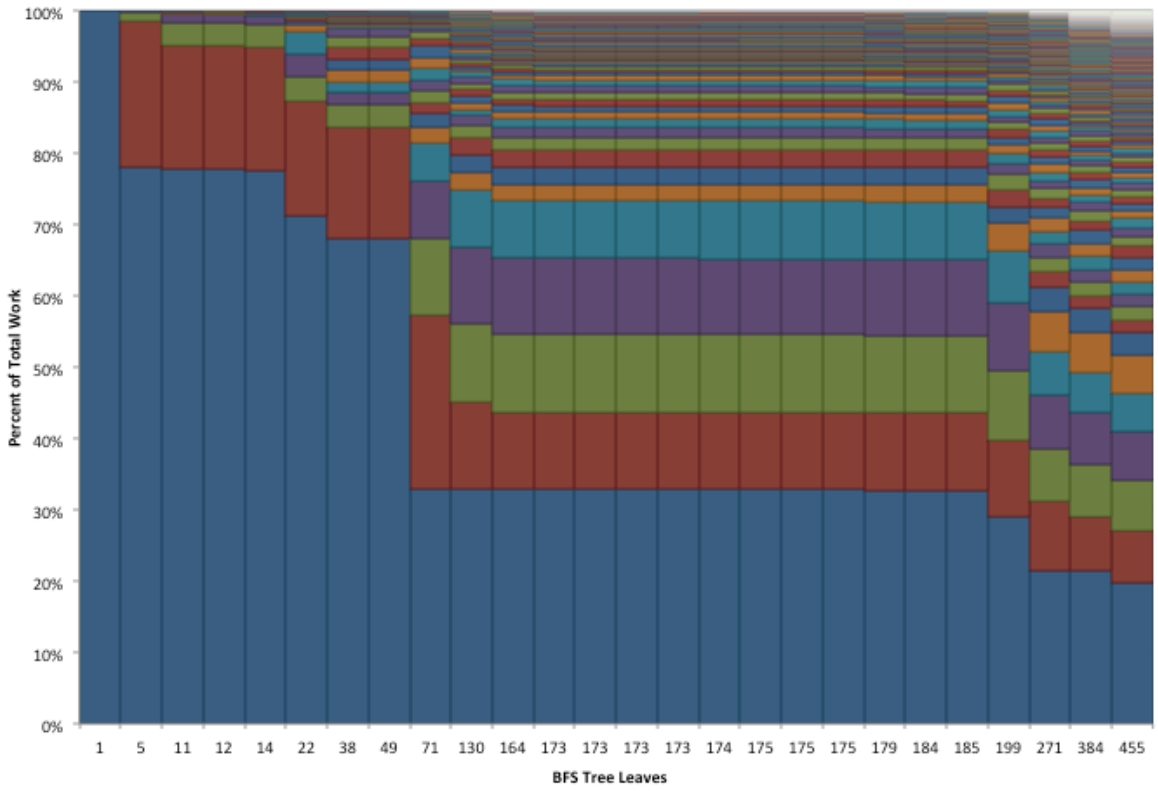


Figure 20. Sorted Work Distribution for Varying BFS Tree Levels

Figure 20 is the same as Figure 19, but with the work of each leaf for each level breakdown is sorted by size. 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 20 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.

G. Workload Prediction

It is 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:

$$L(i) + d'(i) \leq D \qquad (8)$$

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

$$slack(i) = D - L(i) - d'(i) \qquad (9)$$

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 21 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.

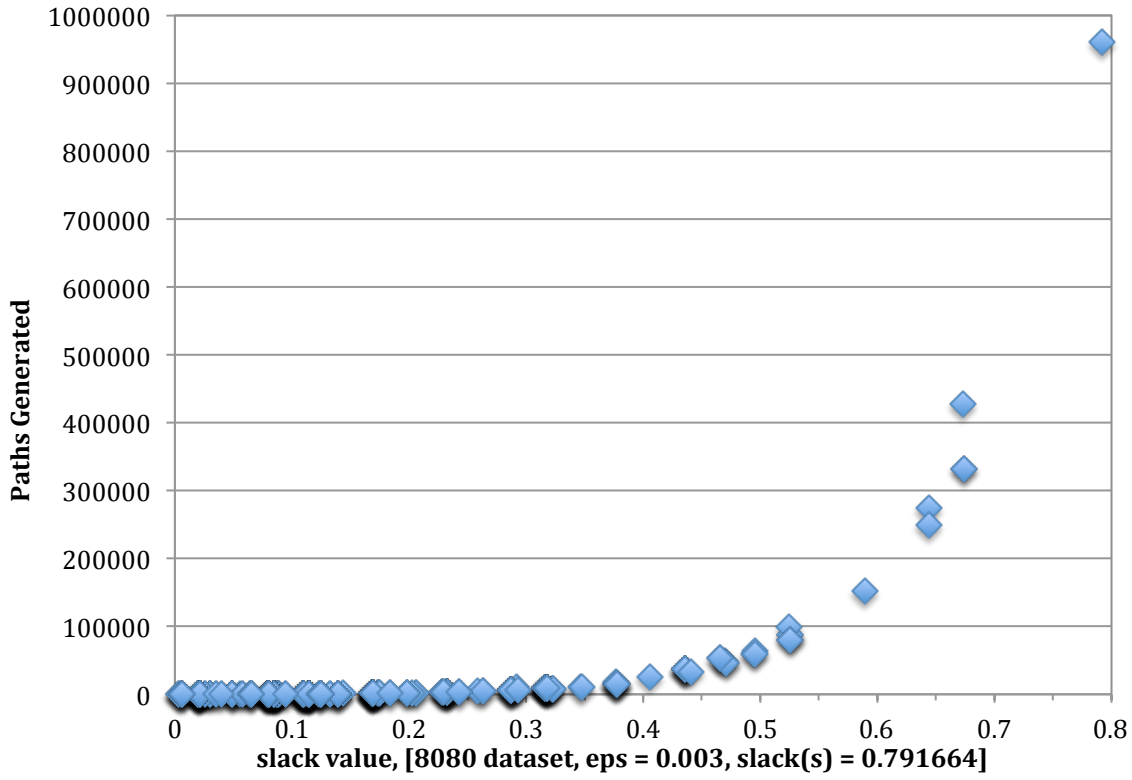


Figure 21. Paths Generated vs. Slack Value, 23 BFS levels

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.

H. 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:

$$0 \le normalized\ slack = \frac{slack(i)}{slack(s)} \le 1 \qquad (10)$$

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 22.

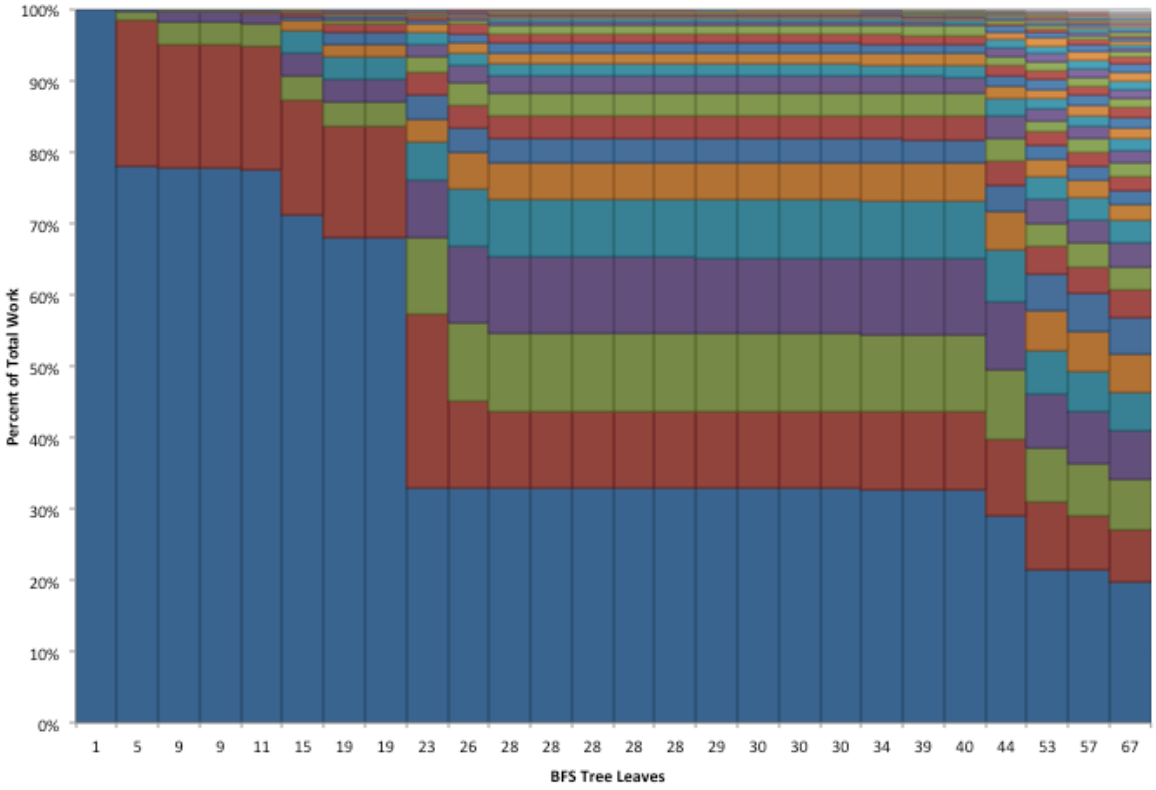


Figure 22. Sorted Work Distribution for Varying BFS Tree Levels with threshold BFS expansion

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 22, 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 20, 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 23, shows the results when using a normalized slack threshold 0.7, and the second plot in Figure 24 is associated when using normalized slack threshold of 0.8. The first graph (Figure 23) shows that 23,882 leaves were generated, and the maximum number of paths from any one leaf is 235,266. The second graph (Figure 24) 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.

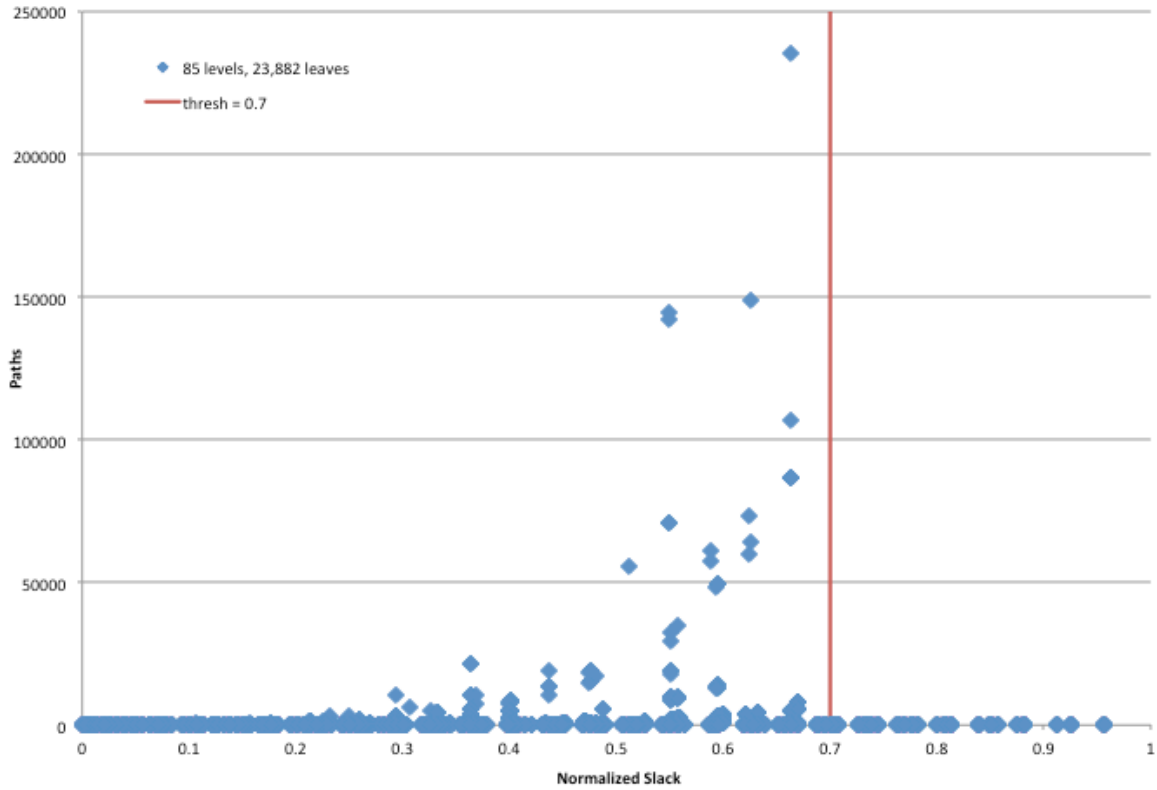


Figure 23. 80x80 data, epsilon = 0.003, 85 level BFS, norm threshold = 0.7

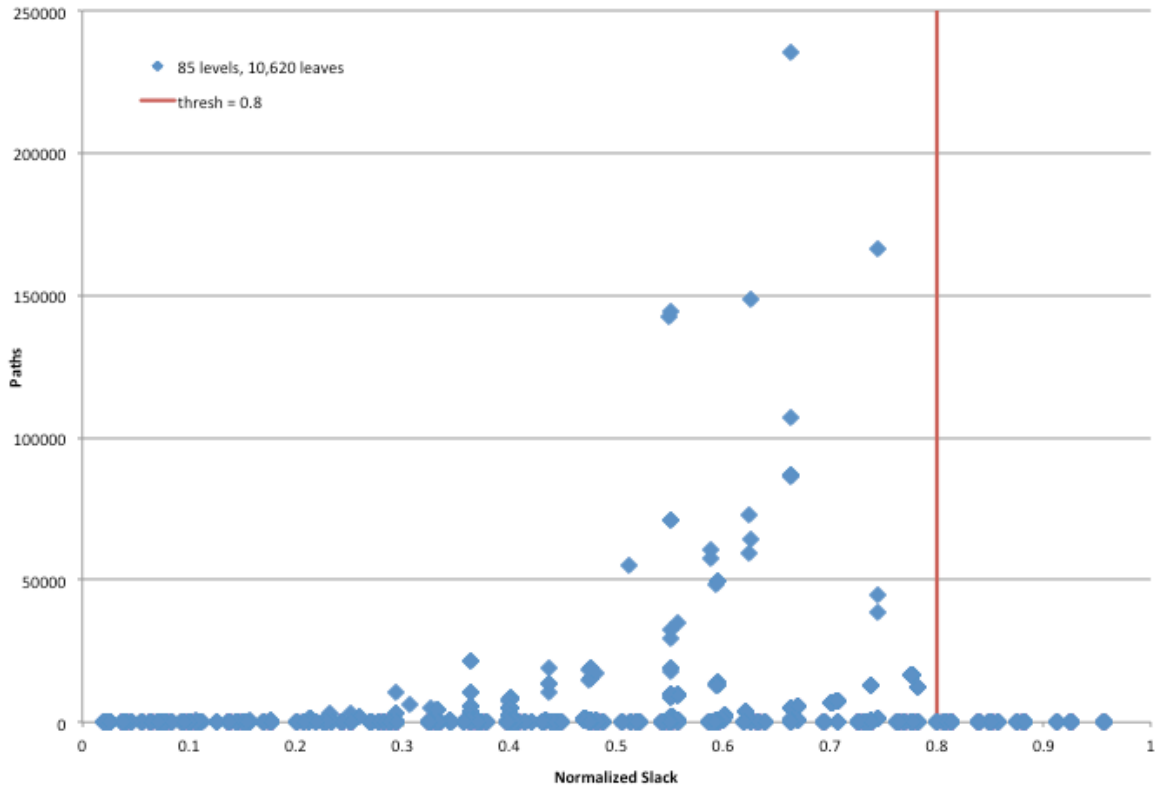


Figure 24. 80x80 data, epsilon = 0.003, 85 level BFS, norm threshold = 0.8

I. 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.

J. Concluding Remarks

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 section 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.

IV. Composite Single-Objective: Strahler Stream Order Inspired Gateway Shortest Path Subsets

A. Introduction

The gateway shortest path problem (Church *et al.* 1992, Lombard and Church 1993) has been shown to be an effective 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 specific intermediate points. While the gateway approach has shown great promise in being able to generate good paths with relatively little computational effort, thus far all techniques that have been developed to screen or review alternatives generated by the gateway model are manual approaches. Even the methodology employed by ESRI in their cost-distance model requires the user to manually review possible alternatives. What is needed is an approach that is capable of identifying good candidate gateway points given the set of gateway solutions. This candidate set would then comprise of points that are potentially "superior" in the sense that they lead to efficient, but *spatially* different solutions. In this chapter we explore a promising approach to identify such points based upon a process that was inspired by a technique in hydrology that is used to assign order to branches in a stream network.

The single gateway shortest path approach begins by the construction of two shortest path trees, one which is rooted at the origin and one which is rooted at the destination. A shortest path tree represents the shortest paths to all other nodes from a root or starting node. A shortest path tree can be easily generated by an algorithm such as Dijkstra's by starting at

a given node and stopping when all nodes have been permanently labeled by a distance from the root node. By keeping track of the precedence nodes in the path, all arcs in the tree can be retrieved and a tree constructed after the algorithm has finished. The essence of the gateway process is to generate two trees and then use the two trees to identify the route and cost for a path that travels to a gateway node (along the tree rooted at the origin) and then on to the destination (along the tree rooted at the destination). This pathway is the least cost pathway that travels from the origin to the destination and is forced to travel via the gateway point. All single gateway shortest paths can be retrieved from these two shortest path trees. In addition, it is easy to compute the distances of all gateway shortest paths by adding the distances of the two distance labels (one for each of the two trees) at each node. From this it is easy to compute a cost surface that shows the cost of travel from the origin to the destination through each possible gateway node. This is computed as a feature of the cost distance function provided in ESRI's ArcMap. An example from ArcMap is given in the following pages. Figure 25 shows a cost grid that is used to compute the shortest routes. The output of the ESRI functionality is given as two rasters: one in which the optimal route is depicted and one of the composite cost surface. Figure 26 depicts a raster indicating the source node and the destination node, as well as the optimal route in green. In this case, the origin is in the southwest corner and the destination is the node in the northeast corner. The shades of pink in the background are associated with the cost distance from the origin. Figure 27 shows the composite gateway cost surface, comprised of the sum of the cost distance from the origin and the cost distance from the destination. The ESRI functionality is designed to produce the cost surface and the shortest route, but does not provide an easy way to peruse spatially different alignment alternatives other than to view the composite costs.

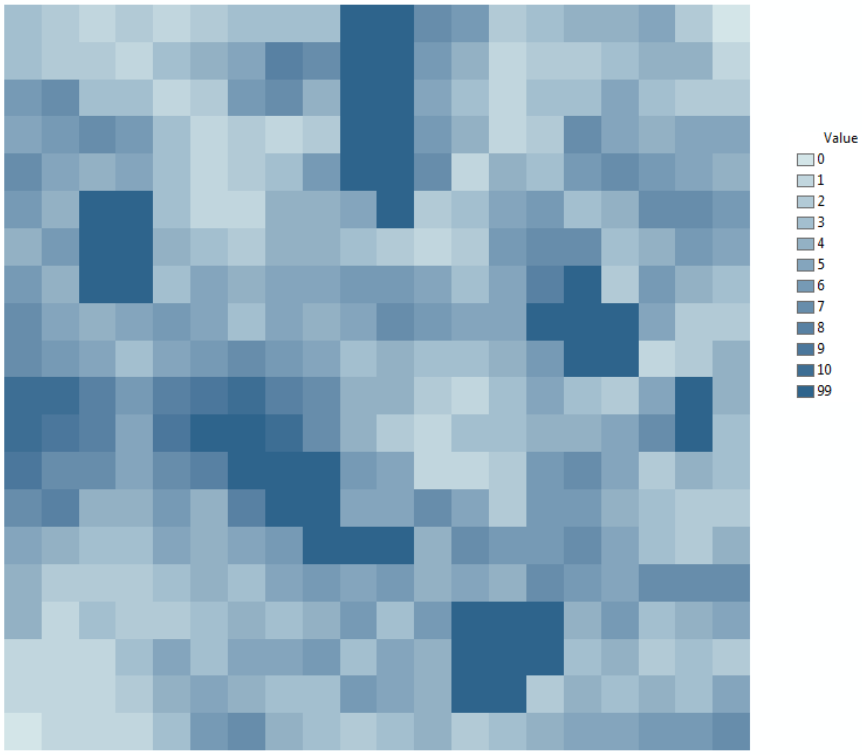


Figure 25. ArcMap 20x20 cost grid

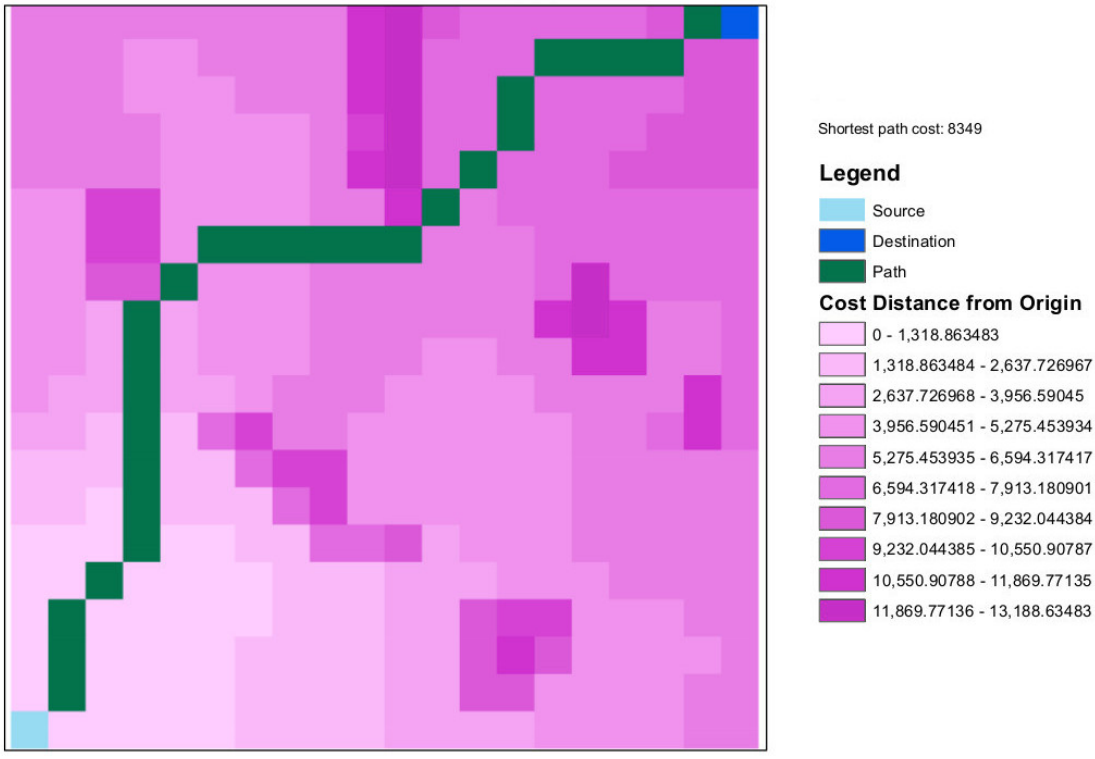


Figure 26. ArcMap 20x20 shortest path and cost distance from origin

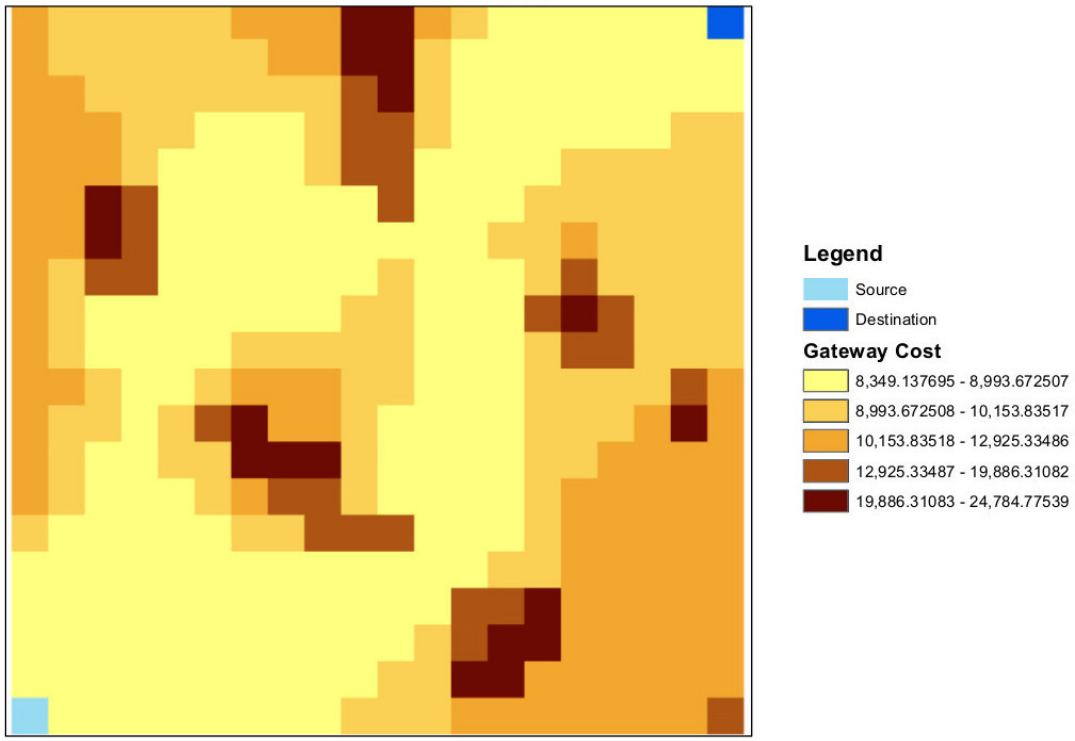


Figure 27. ArcMap 20x20 composite gateway cost surface

Church *et al.* (1992) developed an interface that allowed a user to retrieve and view any gateway path, as well as easily identify specific paths that were on the most efficient paths that were spatially different (using a Tchebychev function). In either case, the number of possible gateway path alignments to explore can be easily overwhelming. For example, a simple 80 row and 80 column raster contains 6400 cells all of which are possible gateway cells. When a more meaningful sized raster is employed in transmission routing (e.g. 1000 rows and 1000 columns), the number of gateway locations could be on the order of a million or more. As shortest path trees are "hydrologic" in structure, we have explored the use of Strahler Stream Order values as a method of identifying the principal limbs that represent potentially spatially different alternatives and sift through a potentially large number of gateway points for those that represent efficient, but spatially different alternatives. The

reason for this is that when a cost surface is non-uniform, the tree has a tendency to form "major branches" along corridors of low cost.

B. Tree Ordering Hierarchies

Strahler stream order (Strahler 1952) is used in hydrology to define stream size using the hierarchy of the tributaries. Based on an earlier stream ordering scheme by Horton (1945), Strahler's ordering modified Horton's to ensure complete objectivity in the structural composition. The ordering method as described by Strahler in his 1952 paper is as follows:

> *The smallest, or "finger-tip", channels constitute the first-order segments. A second-order segment is formed by the junction of any two first-order streams; a third-order segment is formed by the joining of any two second- order streams, etc.*

It is worth noting that other stream order schemes exist. One such scheme is the Shreve stream order (Shreve 1967). This system is simple to understand and has some nice statistical properties in terms of the distribution of order numbers, but seems to have a lesser correlation to the character of actual stream systems. Figure 28 shows an example tree organized by both Strahler and Shreve stream orders.

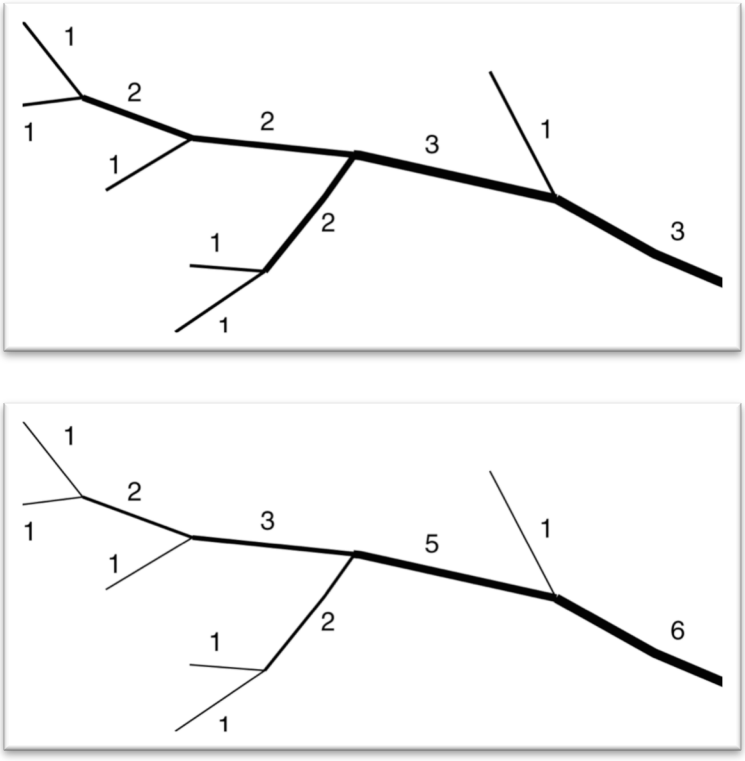


Figure 28. Strahler stream order (top) and Shreve stream order (bottom)
Credit: Wikimedia.org under the GNU Free Documentation License

Fast algorithms for calculating Strahler stream order on a tree have been developed by Lanfear (1990), and Gleyzer *et al*. (2004). Lanfear's approach uses sorting and binary search, and thus runs in approximately $O(m_t P * \log_2(m_t))$ time, where m_t is the number of arcs in the tree and P is the longest path from the "headwaters to mouth". Gleyzer's algorithm uses recursion to dramatically improve the performance of the ordering algorithm, resulting in a method that runs in $O(m_t)$ time. On a graph with n edges and m arcs, the number of edges of a spanning tree $m_t = n–1$, therefore the complexity of the Gleyzer approach is equivalent to $O(n)$. This is much faster than the $O((m+n)*\log(n))$ time required to generate the shortest path trees using our version of Dijkstra's algorithm with a binary heap priority queue. For this reason, we chose to implement Gleyzer's recursive algorithm in order to calculate Strahler order on the shortest path trees. Figure 29 contains an example of the

ordering on one such tree. On the left, is a shortest path tree from the lower-left node to all other nodes. The shortest path from the lower-left to the upper-right is highlighted in red. On the right, the tree is re-rendered with the Strahler order denoted both by arc color and arc thickness, where thick/dark arcs are high order and thin/light arcs are low order.

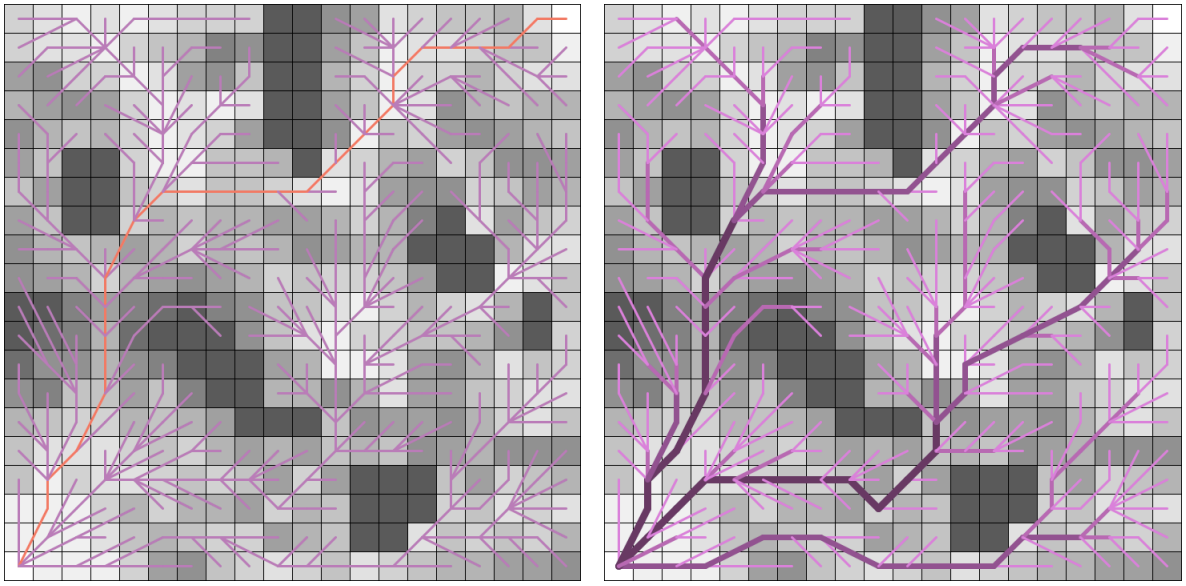


Figure 29. 20x20 Shortest path tree (left), shortest path tree with Strahler order (right)

As discussed in the introduction, a gateway shortest path is generated as the union of the shortest path from an origin to the gateway node, and the shortest path from the destination to the gateway node. All simple gateway shortest paths for all nodes on an undirected graph can be discerned from computing two shortest path trees, one from the origin and one from the destination. Both of these trees may have Strahler ordering applied to them, providing a structural hierarchy to both components of the gateway paths (see Figure 30). Section IV.D will discuss how Strahler ordering of the shortest path trees can be used for automated selection of "good" gateway points to generate quality shortest path alternatives. First though, the next section will go over evaluation criteria for shortest path alternatives.

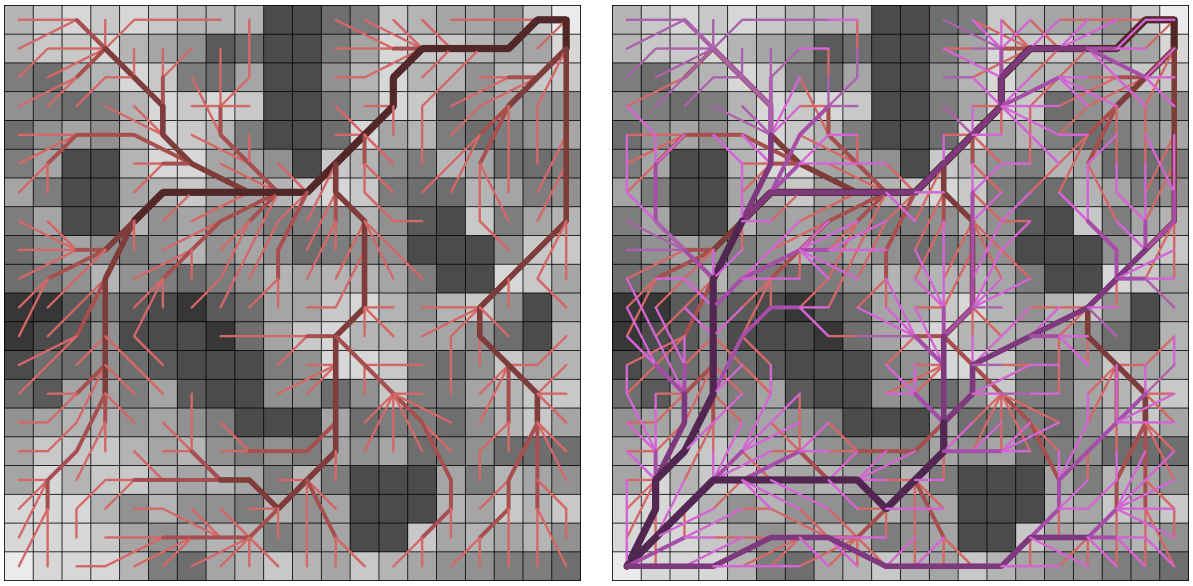


Figure 30. 20x20 reverse tree with Strahler ordering (left), and both trees overlain (right)

C. Evaluating Gateway Paths

Good alternative paths must perform well both in terms of minimizing cost, as well as being spatially different from other paths to which they are being compared. Measuring path cost is simply the sum of the cost of all arcs that compose that path. There are many ways to measure path difference, as was discussed in Chapter II.E, and in this work we used the area difference metric of comparing the alternate route to the shortest path. This is consistent with the approaches used in Lombard and Church (1993) and Scaparra *et al.* (2014). Figure 31 contains an example of such a path comparison. The image on the left highlights the shortest path by coloring the nodes of that path in orange. The image on the right highlights an alternative path using green arcs, and the area difference is displayed as the red shaded region in between the shortest path and the alternate path.

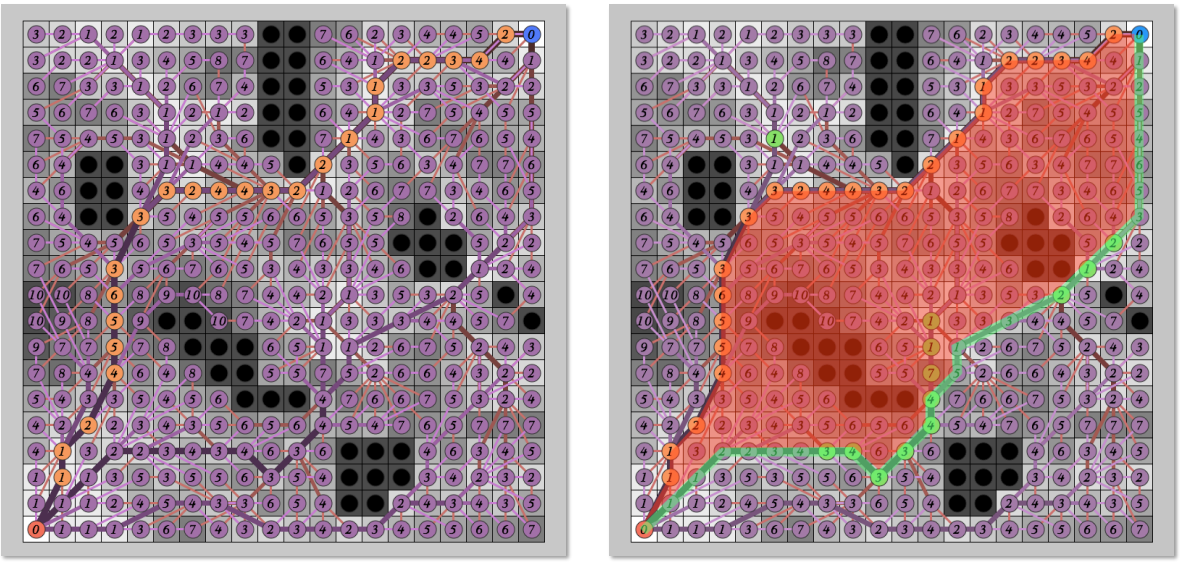


Figure 31. Shortest path (left), alternative path with area difference shaded in red (right)

Lombard and Church (1993) consider the single gateway case, where crossings between the shortest path and the gateway paths are very rare occurrences, and can therefore be disregarded. If no crossings are considered, the area computation reduces to the simple task of using area labels in the shortest path algorithm. Scaparra *et al.* (2014) instead force a gateway path through multiple gateways, in which case crossings are very likely (just consider the simple case of two gateways lying on different sides of the shortest path), and therefore cannot be neglected in the area difference computation. When paths cross, each of the polygons enclosed between the shortest path and the gateway path must be identified and its area calculated. The area computation in this case must be entirely delegated to the gateway path construction phase, since only then can the intersections of the gateway paths with the shortest path be detected. Each newly detected intersection defines a polygon, whose area can be calculated through the formula based on Green's Theorem on the plane. Namely, the area A of a non self-intersecting polygon made up of line segments between M vertices (x_i,y_i), $i = 0$ to M-1, is:

$$A = \frac{1}{2}\left|\sum_{i=0}^{M-1}(x_i y_{i+1} - x_{i+1} y_i)\right| \qquad (11)$$

In the area formula, the last vertex (x_M, y_M) is assumed to be the same as the first. Each vertex of the polygon is either a network node or an intersection point between the shortest path and the gateway path. While our approach here uses single gateway shortest paths, we have chosen to use the Green's Theorem approach for area computation, which is more precise and would apply to any future expansion to multi-gateway applications.

Evaluating alternatives is essentially a multiobjective task, as it is desired for alternatives to be both spatially different from the shortest path as well as low in objective cost. One can characterize the performance of gateway paths by plotting a point for each path in objective space, where these two competing objectives are the two axes (path length or cost and spatial difference). Figure 32 is an example of one such plot, depicting the performance of all single gateway paths from a network generated from an 80x80 subset of the Maryland Automated Geographic Information (MAGI) database. The x-axis of the plot measures area difference between the path alternative and the shortest path, and the y-axis plots the objective cost of the path alternative. Notice that the y-axis uses a reverse scale, so that both objectives are improved by moving away from the lower-left corner.

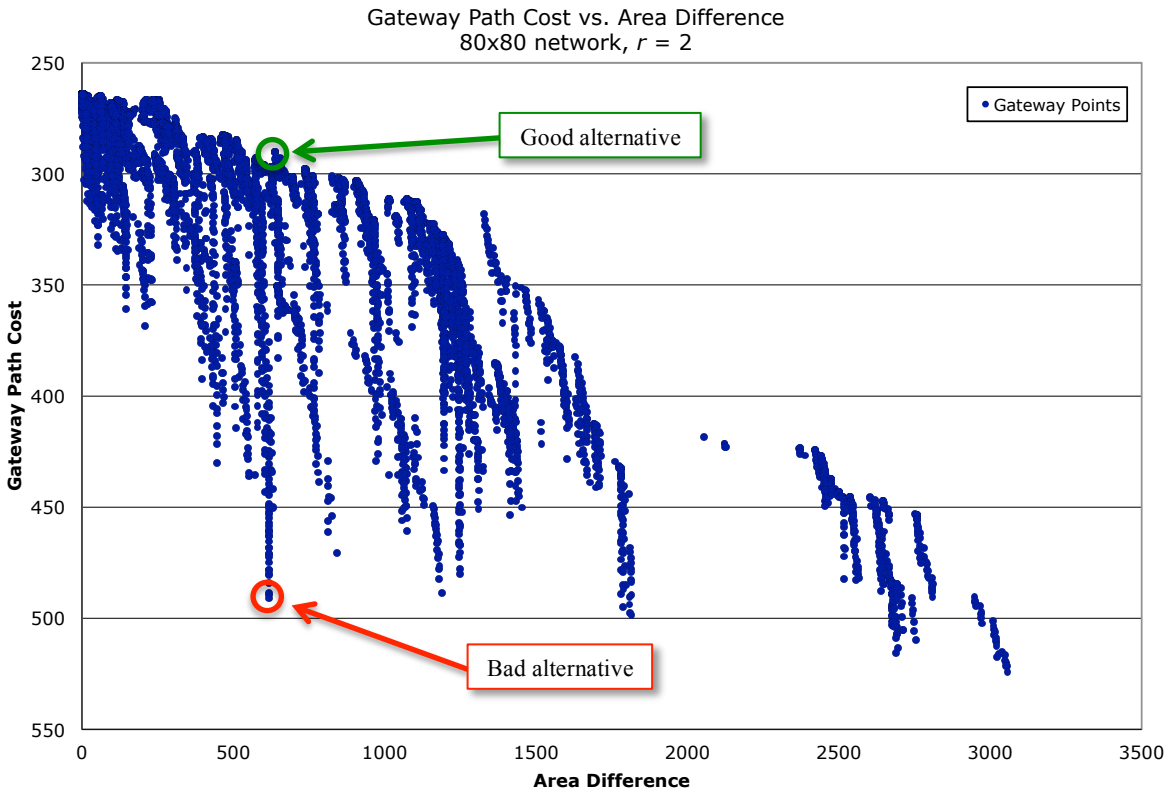


Figure 32. Objective space evaluation of gateway shortest paths

This plot allows one to compare the performance of a selected alternative from all other gateway path alternatives. For example, in Figure 32, the path represented by the point within the red circle would be considered a bad alternative. While it is spatially different from the shortest path, there exist numerous other gateway paths that have the same level of spatial difference but have a better (lower) objective cost. On the other hand, the path represented by the point in the green circle would be a good alternative, as it too is spatially different from the shortest path, but is also among the best in objective cost performance of those paths with similar area difference. In general, the best alternatives will lie on or near the top "ridge" of all the solutions in the objective space plot depicted in Figure 32.

D. Strahler Threshold Automated Alternative Path Selection

Using the above criteria for evaluating shortest path alternatives, now the question that arises is how do we automate the selection of a set of quality shortest path alternatives? Church *et al.* (1992) discuss using an interactive interface for being able to select a gateway point, view the gateway path, and also view its performance relative to other gateway paths in objective space. While we agree that exploration of path alternatives should certainly include an interactive component, present-day maps are often too large to be able to evaluate all alternatives in an interactive fashion. Even a small map such as the 80x80 example (see Figure 32) generates many hundreds of unique paths that would be cumbersome to evaluate interactively. Instead, it would be useful for the ability to have a model to suggest a small subset of these paths as worth highlighting for closer analysis.

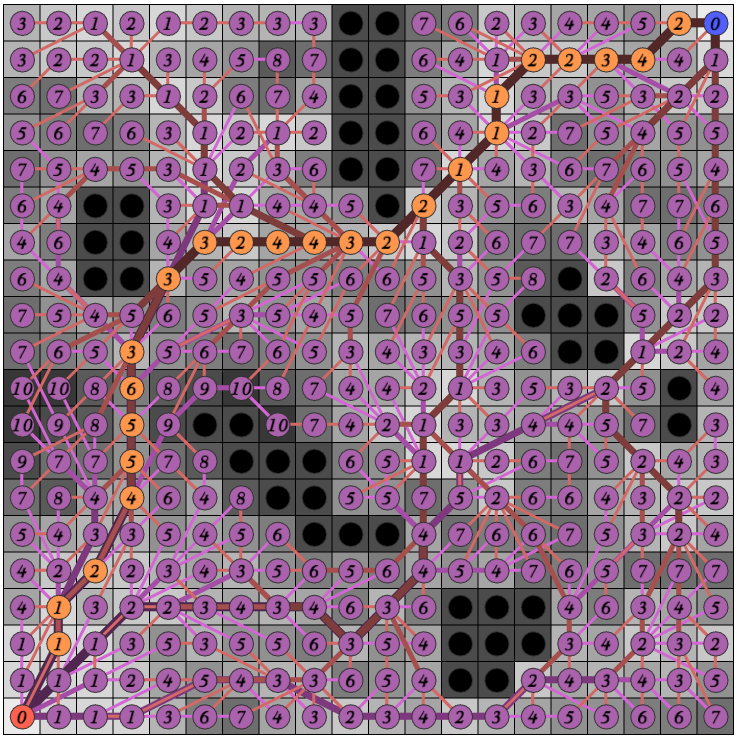
Strahler ordering enables the selection of this subset of quality paths. The idea behind it is that the paths on a shortest path tree will have a tendency to follow low-cost corridors in the data, diverging from those corridors only as necessary to reach all destinations in the network (a complete shortest path tree must cover all nodes in a network with a minimum cost tree, as measured as the sum of the cost over all nodes in the network of the path from the origin to each node). This tendency will result in major branches in the tree that will have high order when analyzed via Strahler ordering. The intersection of high-order branches from both the forward shortest path tree and the reverse shortest path tree would then indicate that a gateway path composed of those high order branches would be a low-cost alternative path. Additionally, given the inherent spacing of these large branches (one cannot have a large branch without many smaller branches feeding into it) there is some assurance that they will be among the set of spatially diverse alternatives.

Strahler stream ordering is an arc attribute, while gateway paths are typically defined by selecting a gateway point. Gateway paths may also be defined by gateway arcs (Katoh *et al.* 1982, Medrano and Church 2014), but shortest path trees do not share all of the same arcs, and thus are not suited for gateway arc path selection. To convert the Strahler arc attribute to a node attribute, we define each node order as the maximum order arc that has an endpoint at that node. Thus each node receives two Strahler order attributes, one for the forward shortest path tree, and one for the reverse.

With these attributes, criteria can be defined to highlight nodes as possible alternatives. The simplest criterion is to highlight all nodes where both forward and reverse labels are greater than or equal to a specified threshold value of t.

E. Computational Experiments

This approach was coded in the Java programming language, using the Processing API (www.processing.org) to help in visualizing the graph and algorithm results. We ran experiments on two networks used in the literature (Lombard and Church 1993, Scaparra *et al.* 2014): a 20x20 manually fabricated raster and an 80x80 subset of the Maryland Automated Geographic Information (MAGI) database. First we will discuss the results on the smaller 20x20 network, followed by a discussion of the larger 80x80 application. In the figures depicting results from the 20x20 network (Figure 33 to Figure 36), the numbers inside nodes (circles) in the decision space network represent the cost or impact of the cell in calculating the traversal impact. The 80x80 network depictions do not display numerical cost values due to display size restrictions.



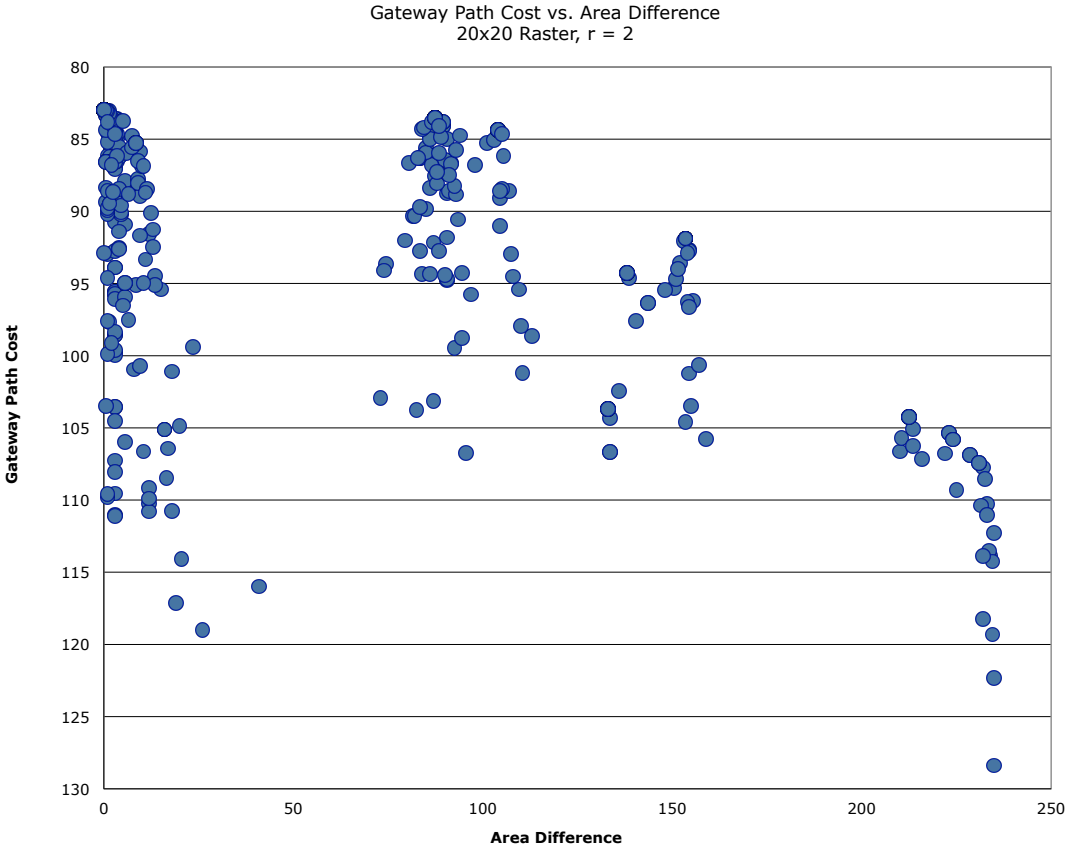
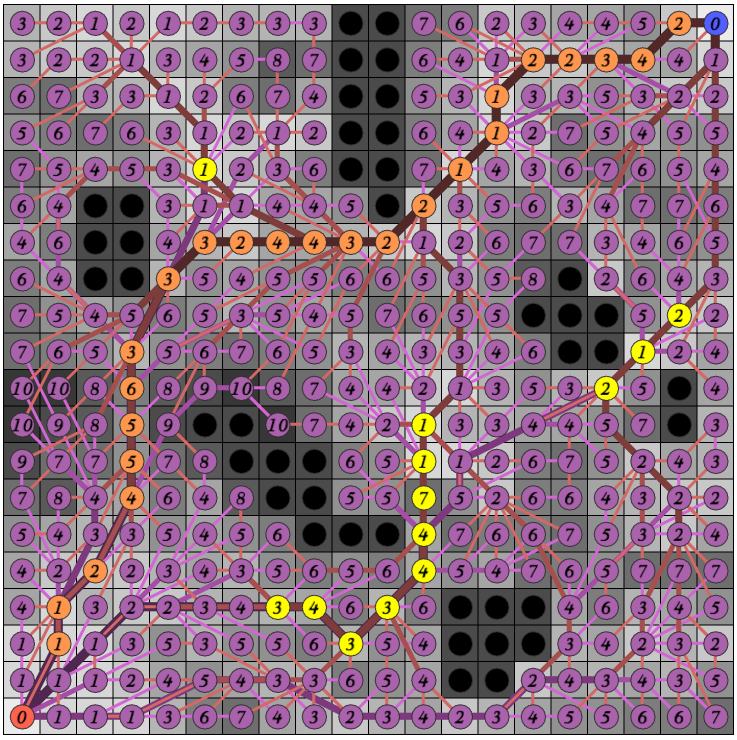


Figure 33. 20x20 r = 2 network all gateway paths: decision space (top) objective space (bottom)



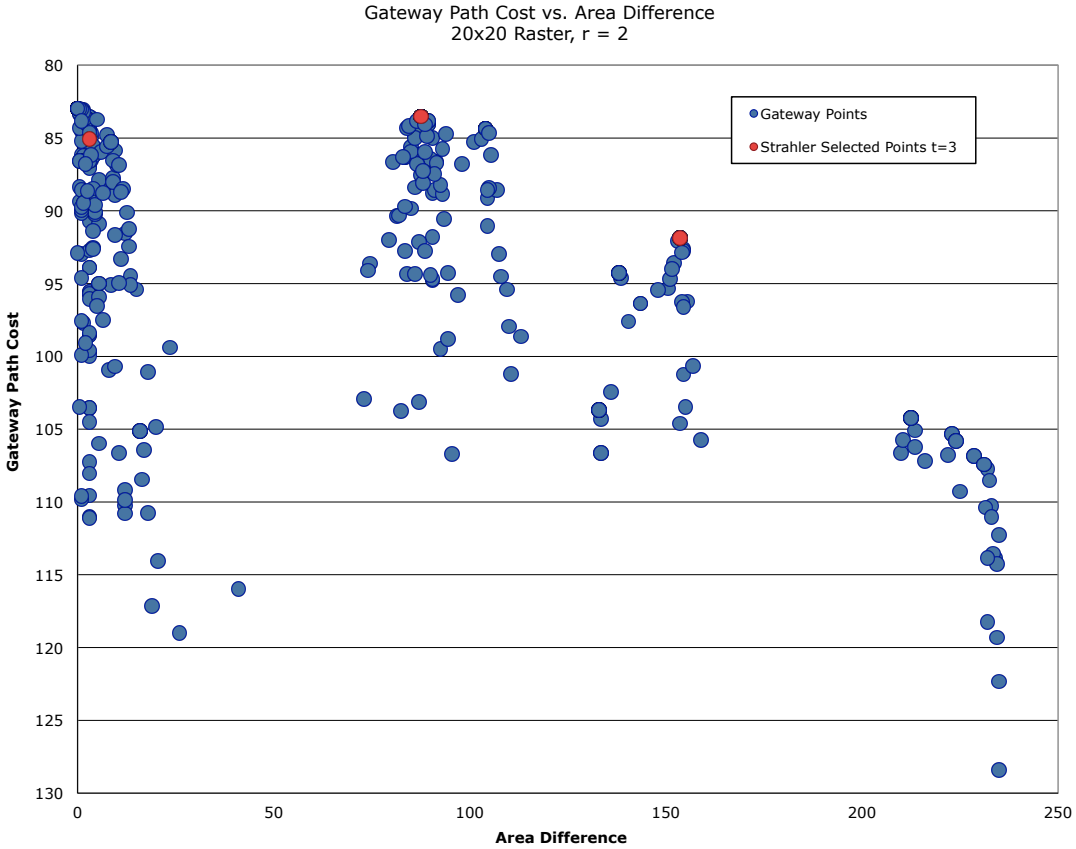
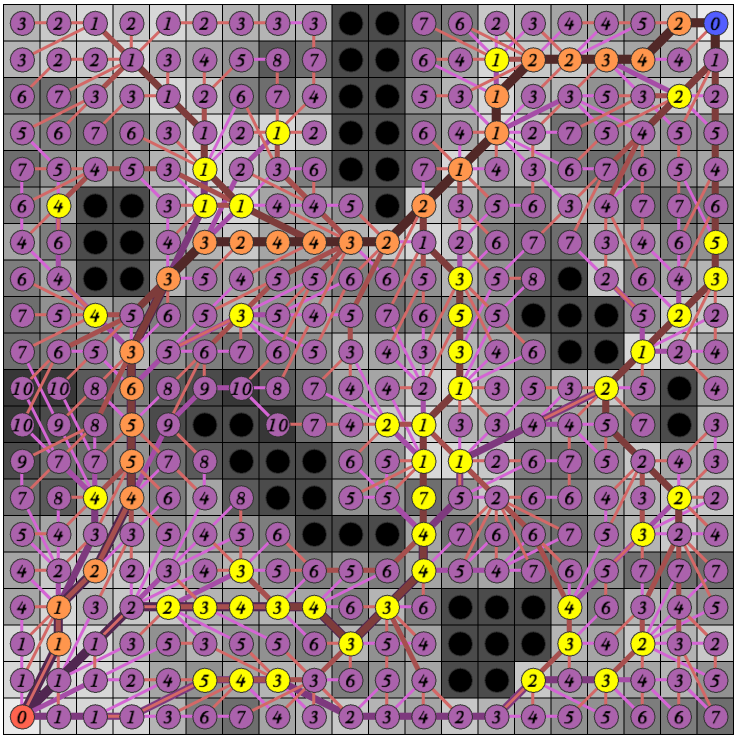


Figure 34. 20x20 network t = 3 gateways: decision space (top) objective space (bottom)



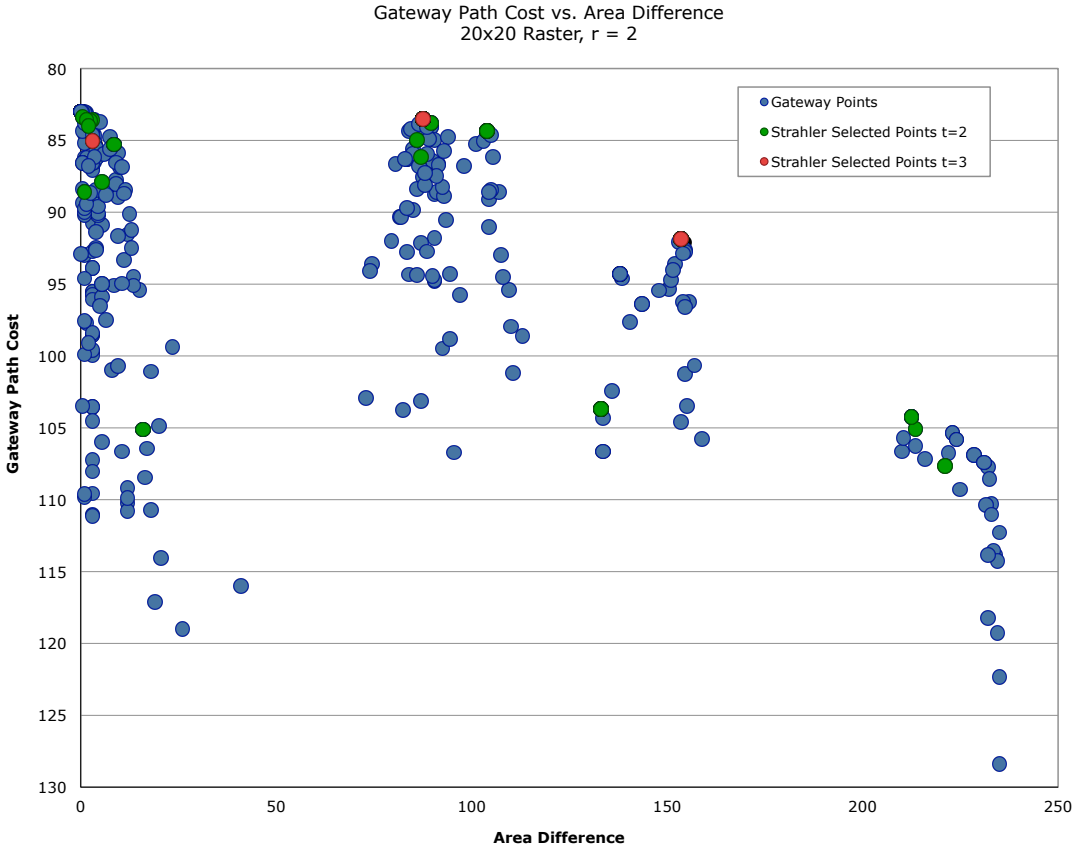
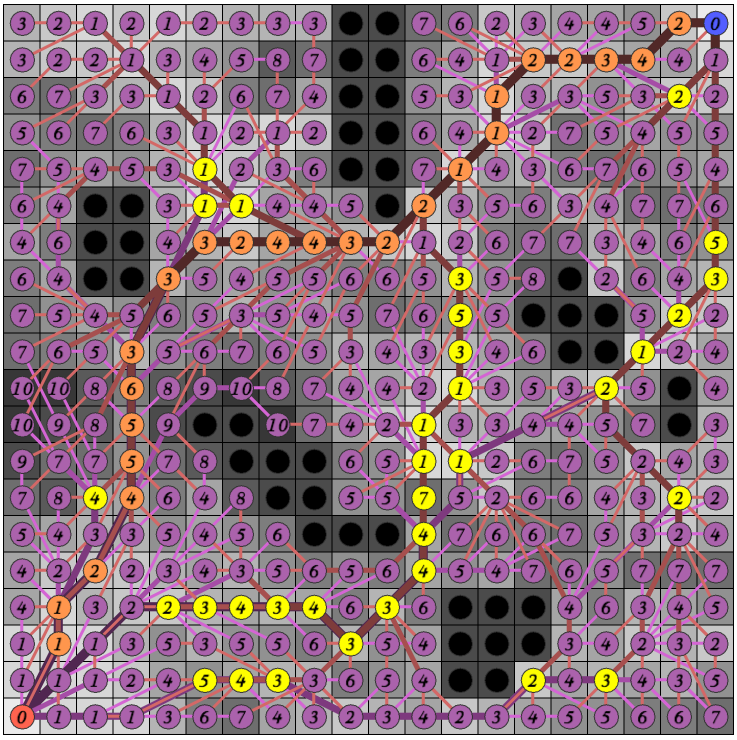


Figure 35. 20x20 network t = 2 gateways: decision space (top) objective space (bottom)



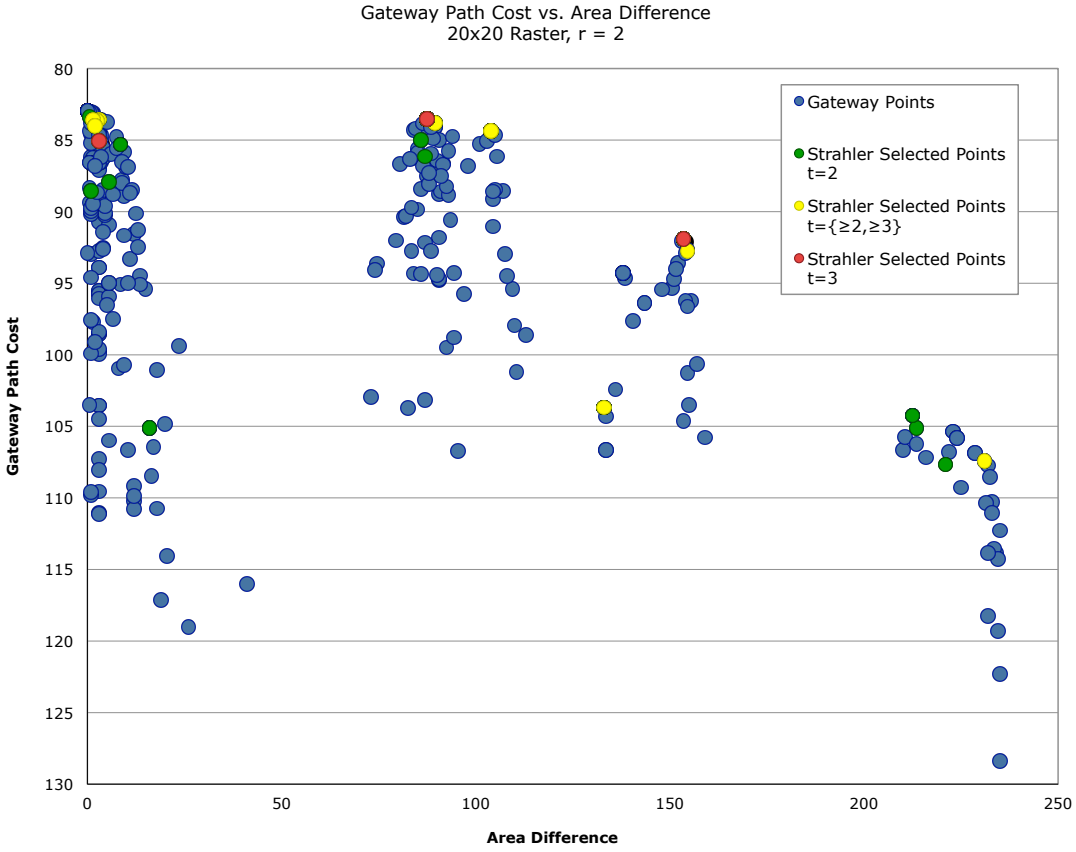


Figure 36. 20x20 network t = [[2, 3} gateways: decision space (top) objective space (bottom)

After applying the Strahler ordering on the 20x20, both the forward tree and reverse tree had branches with orders ranging from 1 to 4. Calculating the shortest path tree and applying Strahler ordering to the tree were computed instantaneously on such a small network. Figure 33 first shows the 20x20 network decision space (top) and the corresponding objective space (bottom). The bottom portion of Figure 33 displays the performance of all gateway paths with regards to cost and area difference. Figure 34 highlights the gateway nodes where the Strahler Stream Order (SSO) values of both the forward and reverse shortest path trees at that node are ≥ 3. This results in 13 gateway points being highlighted that represent 3 unique and different paths. These three paths are shown in red in the associated objective space plot. Two of the paths perform extremely well, in that they are both low in cost and also spatially different from the shortest path. The third path (represented by the most upper-left gateway point), is a small deviation from the shortest path. While it too is a low cost path, the deviation from the shortest path is minimal. Lowering the SSO threshold to ≥ 2 for each shortest path tree highlights more alternatives, as displayed in Figure 35. This highlights 44 gateway points, resulting in 20 unique gateway paths (including the three where the SSO in each direction is ≥ 3). These paths are shown in the objective space by green points alongside the t = 3 paths in red. More of the Pareto optimal paths have been selected by this lowered SSO threshold value, but more dominated paths have been highlighted as well. Perhaps some other compromise threshold might be able to maintain most of the non-dominated solutions while eliminating most of the dominated solutions. Figure 36 displays the results when trying the criteria of selecting nodes with one Strahler label ≥ 2, while the other must be ≥ 3. This combination of values falls between the stringency of each SS0 ≥ 3 and each SSO ≥ 2, and results in 33 nodes highlighted that define 12 unique paths. In the

objective space chart, these paths are colored yellow. In this case, all but one of the paths are on or near the Pareto frontier of solutions, while missing only 1 or 2 Pareto paths on the far right of the objective space chart that were caught by the $t \geq 2$ threshold (green).

Next we tested this method on the 80x80 r = 2 MAGI network. After applying the Strahler stream ordering on this network, the forward tree had branches with orders ranging from 1 to 7, while the reverse tree had branches with ranges from 1 to 6. Calculating a shortest path tree on this network took about 44 milliseconds on a Macbook Pro laptop, while the Strahler stream ordering was accomplished in 7 milliseconds. Figure 37 displays images of the forward and reverse Strahler ordered trees for the 80x80 MAGI network.

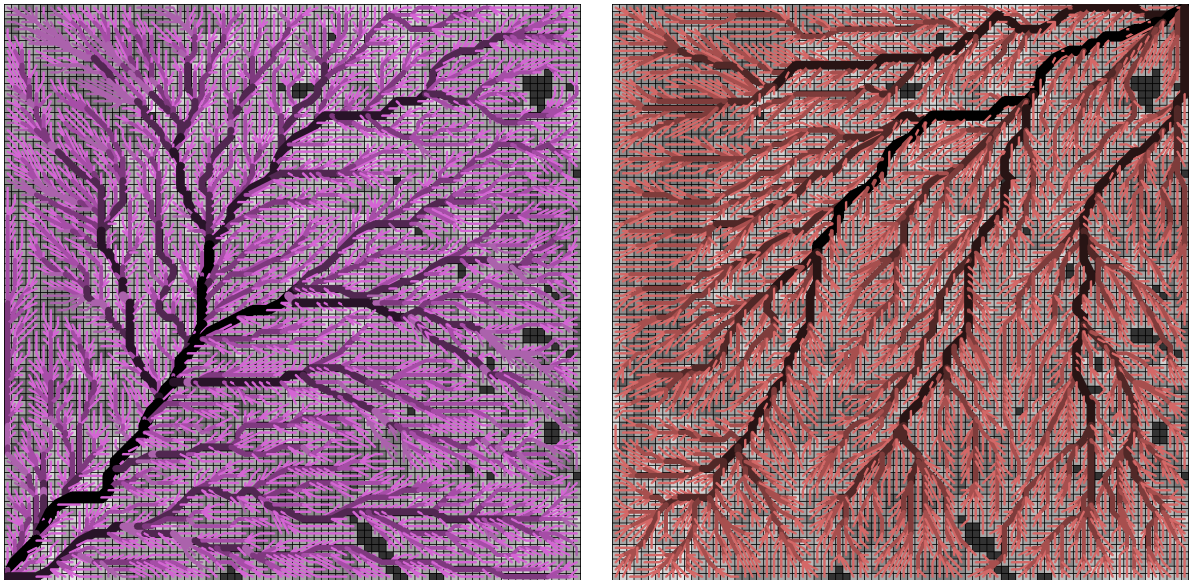
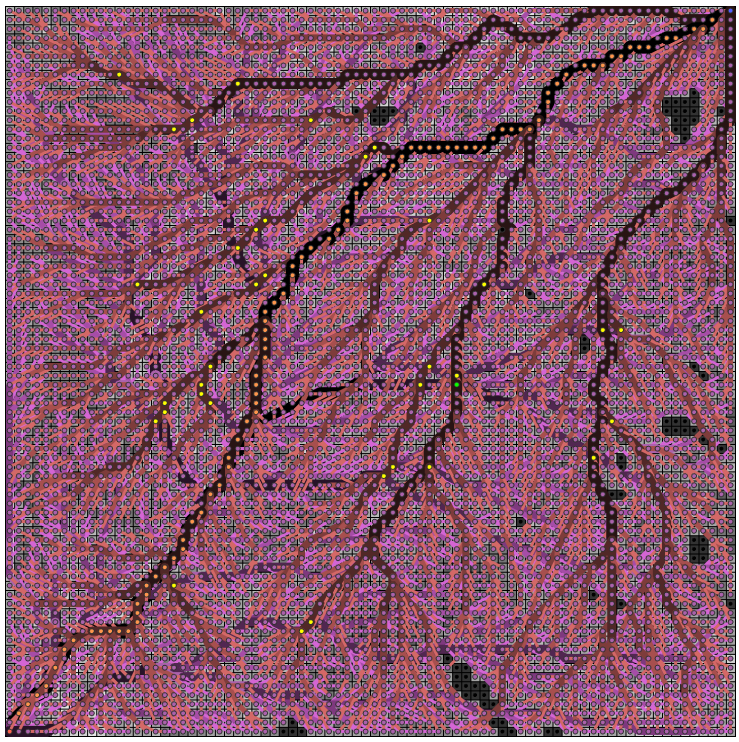


Figure 37. 80x80 shortest path trees with Strahler ordering: forward tree (left) and reverse tree (right)

For the automated path selection, there is only one path for a threshold of t = 5 in both directions. This gateway point is highlighted in green in the decision space and the objective space in Figure 38. This gateway path is near-Pareto optimal from among the gateway possibilities, and represents a reasonably good path alternative.



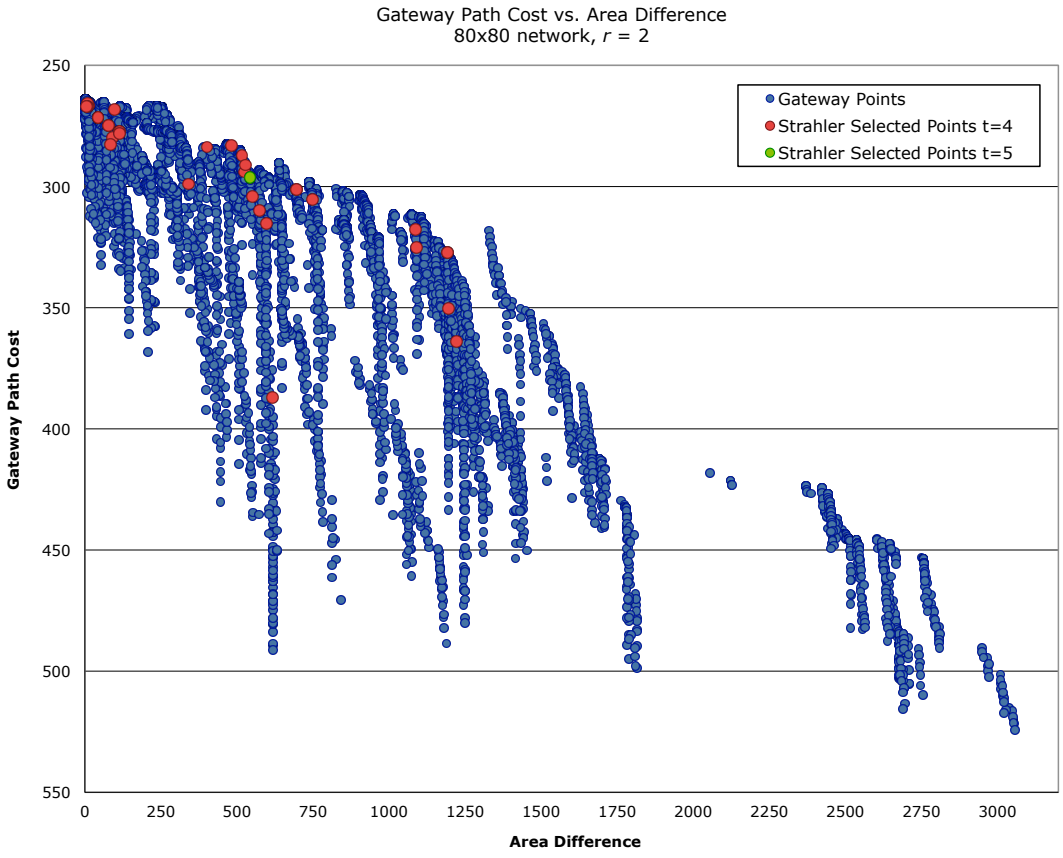
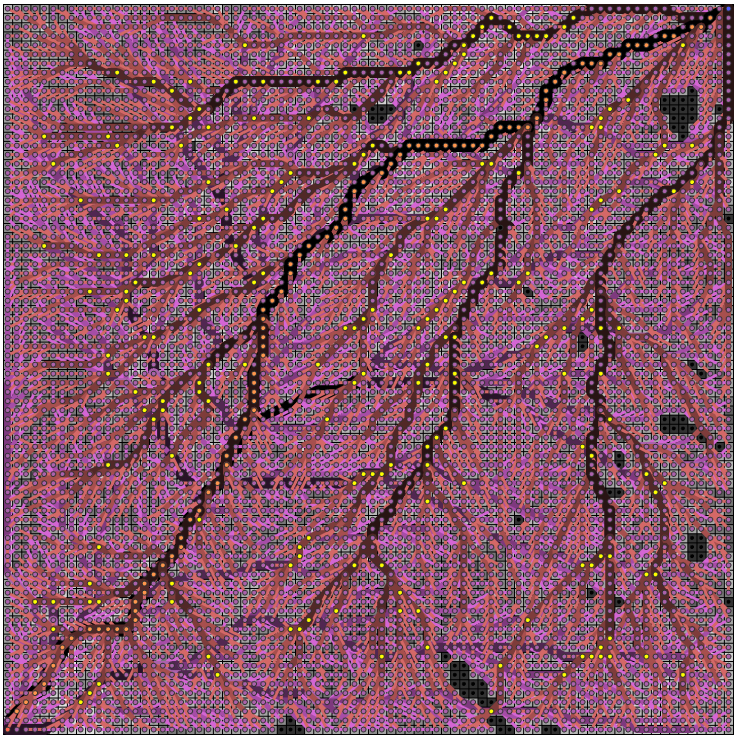


Figure 38. 80x80 network t = 5 & t = 4 gateways: decision space (top) objective space (bottom). The t = 5 gateway point in decision space is highlighted in green.



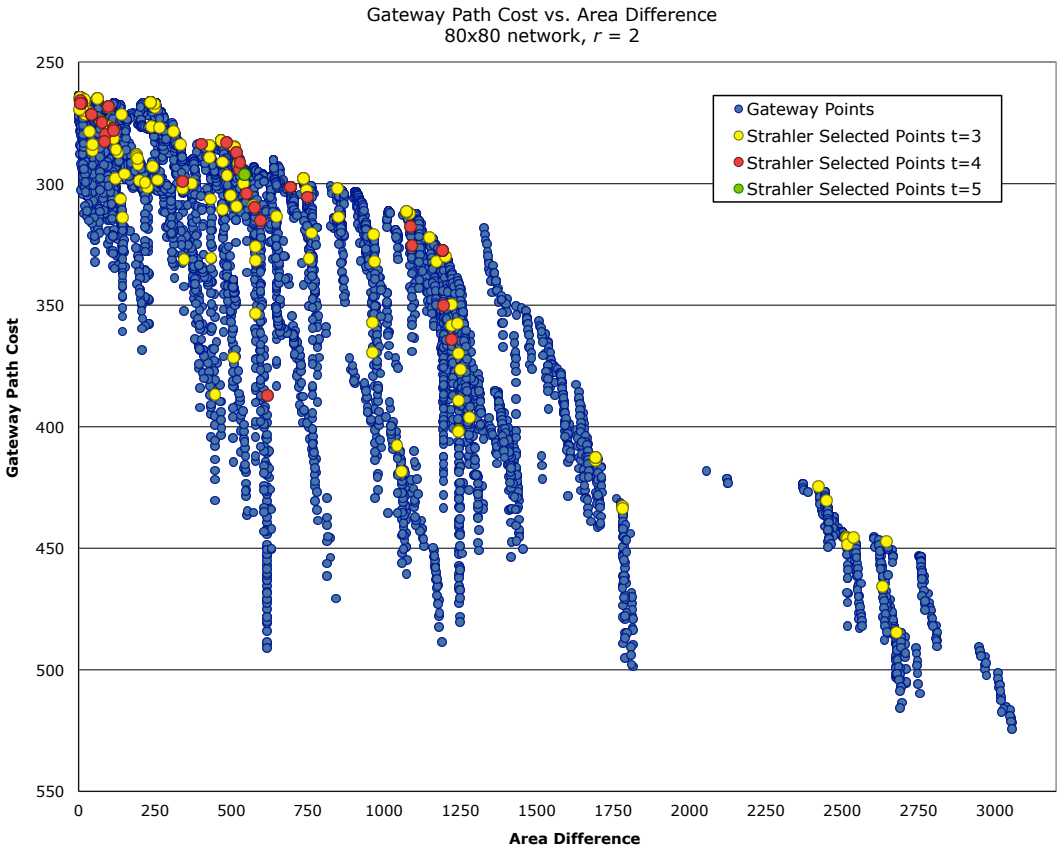


Figure 39. 80x80 network t = 3 gateways: decision space (top) objective space (bottom)

Figure 38 also highlights the points and associated paths identified by using a SSO threshold standard of $t \geq 4$. These consist of 37 gateway points that represent 27 unique gateway paths. About half of these paths are on or near the Pareto frontier, while the other half would be considered inferior. They also do not reach some of the larger area difference regions of the objective space, indicating that a less stringent threshold would be required in order to select paths of higher spatial diversity from the shortest path.

When using the less stringent SSO threshold of $t \geq 3$ for each tree as displayed in Figure 39, 203 graph nodes are highlighted. This resulted in many more unique gateway path options being identified. These gateway paths cover more of the Pareto optimal frontier in objective space, filling in many of the voids that were missed by the SSO $t \geq 4$ threshold. Additionally, much higher spatial diversity is achieved, with paths now being selected in the second tier of area difference on the far right of the objective space chart. Almost all of the higher area difference paths are on the Pareto frontier, and thus these paths would be considered excellent alternatives. On the other hand, many dominated paths were selected within the region of smaller area difference solutions.

F. Concluding Remarks

We used Strahler stream ordering to define a hierarchy for the shortest path trees in order to aid in selecting good gateway points from among all the gateway point possibilities. Each node in a network was given two Strahler stream order values based on the order hierarchy of the forward and reverse shortest path trees. Sets of path alternatives were selected by choosing gateway points with Strahler stream orders that exceeded a given threshold. These path alternatives were evaluated based upon two criteria: minimizing the path cost, as well as maximizing the path area difference relative to the shortest path.

Computation of these alternatives was extremely efficient, where the most time consuming portion of the calculation was in the generation of the shortest path tree, which has been shown to be a polynomial computation, and has an extensive literature on efficient algorithms. With Strahler stream ordering and path filtering being linear operations, this method is very fast at heuristically selecting a set of path alternatives.

On the 20x20 network, the Strahler stream order based selected alternative path sets showed excellent performance in as measured in path length vs. shortest path area difference. In the 80x80 data set, the selected alternatives also performed well for the stringent thresholds, but suffered when the threshold became too unrestrictive. The low SSO $t \geq 3$ threshold also selected far too many paths to be useful in a realtime analysis, thus indicating that it is important to select a threshold criteria suitable to the data being analyzed. It is important that this method be used interactively:

1) To ensure proper threshold values are being applied for the given data. Using this approach on various data sets will result in different magnitudes of Strahler order numbers as the size and character of the data change
2) To verify the quality of the solutions. The area difference metric is used to only compare a path alternative to the shortest path, and not to other path alternatives. Two paths with different routings in decision space may have associated points that lie very near each other in objective space.

This research was explored only within the context of selecting corridors over terrain networks. Future work should extend to other network types, using this for alternative route applications where speed of acquiring results is paramount. For example, road networks have an intrinsic hierarchy consisting of residential streets to main roads to interstate

highways, and could be particularly suitable to the selection of alternatives using a hierarchical threshold.

V. Unsupported Multi-Objective: A Biobjective Gateway Shortest Path Approximation Heuristic

A. Introduction

Multi-objective shortest path algorithms have been designed to compute the set of all Pareto-optimal (i.e. non-dominated) paths, both unsupported and supported, on a given network. Non-dominated *supported* solutions, which represent the convex subset of the non-dominated solutions, are relatively easy to identify using a weighted composite objective and a classical shortest path algorithm such as Dijkstra's algorithm (Dijkstra 1959). Unfortunately, identifying *unsupported* non-dominated shortest path solutions is a more complex task that is equivalent to solving a constrained shortest path problem, which has been proven to be NP-hard (Garey and Johnson 1979). Therefore, the combinatorial nature of the problem makes computing the exact set of non-dominated solutions on a large network a computationally intensive endeavor. Consequently, it is desirable to consider alternatives that are designed to quickly generate an approximation of the Pareto-frontier.

This section describes a new technique for determining an approximation of the non-dominated solution set for a multi-objective path problem that is designed to address large corridor location problems. This approximation is based upon identifying all optimal supported solutions and a subset of the optimal or near-optimal unsupported solutions. It is shown that this method can efficiently generate a quality set of path candidates for the unsupported non-dominated set. In the next section we review the relevant literature. This is followed by a description of the new algorithm. Computational results are provided for

various GIS-based raster networks and are compared to solutions generated by two exact, state-of-the-art algorithms.

B. Gateway Shortest Paths: Defining Gateway Node and Arc Paths

Lombard and Church (1993) defined a spatially-constrained form of the shortest path problem called the gateway shortest path. A gateway shortest path is an origin-destination shortest path that is constrained to traverse through a given intermediate node on the network. They used this construct to identify a range of path alternatives for a single-objective corridor location problem. We will call such a path a gateway node (GWN) path. We expand that definition to include gateway arcs, where the shortest path between the origin and destination must traverse a given intermediate arc. This will be called a gateway arc (GWA) path.

The process of assembling shortest gateway paths involves generating two shortest path trees, one rooted at the origin and one rooted at the destination. A shortest path tree from a particular node, u, is the set of shortest paths from u to all other nodes in N, and is generated using an efficient one-to-all shortest path algorithm. In the data used for this study, all network arcs are undirected (represented as symmetrical directed arcs), and thus a shortest path tree is able to access all nodes. The method is applicable for other graph types as well, including directed graphs and multigraphs. In a directed graph, there may not exist a feasible path to a particular gateway node/arc, in which case there is also no non-dominated path that includes that gateway node/arc. A multigraph can easily be converted into a conventional graph by adding an additional dummy-node to all parallel arcs without changing any aspect of the problem.

We used Dijkstra's algorithm (1959) with a binary heap priority queue for generating a shortest path tree. More formally, we define a shortest path tree from node u to all other nodes as $T(u)$, and a path from u to node v along $T(u)$ to be $u \xrightarrow[T(u)]{} v$. To generate all GWN or GWA paths, one begins with solving a shortest path tree from the source, $T(s)$, and from the destination, $T(t)$. From this, a GWN path constrained to pass through node $u \in N$ is defined as $s \xrightarrow[T(s)]{} u \xrightarrow[T(t)]{} t$, and a GWA path constrained to pass through arc $(u, v) \in A$ is defined as $s \xrightarrow[T(s)]{} u \rightarrow v \xrightarrow[T(t)]{} t$. In essence, each gateway node path can be generated by traversing the path from the origin s to the gateway node u on tree $T(s)$ and then traversing the reverse path from node u on tree $T(t)$ to the destination. The shortest gateway arc path is generated in a similar manner, except that after traversing the origin shortest path tree to node u, the path passes through arc (u,v), then traverses the destination tree $T(t)$ to the destination. Thus, all shortest gateway node and gateway arc paths can be simply generated by appropriate use of two shortest path trees. It should be noted that the set of all GWN paths is a subset of the GWA path set. The proof of this observation is quite simple, as the gateway node path can be found as an equivalent gateway arc path for the arc that is used on the shortest path tree touching node u on tree $T(s)$. In general, the number of gateway arc paths is significantly greater than the number of gateway node paths.

It is important to point out that gateway paths formed the basis of the fast loopless K^{th}-Shortest Path algorithm developed by Katoh *et al.* (1982). The Katoh algorithm finds the next shortest path by scanning for all GWN and GWA paths (called Type I and Type II paths in that paper) on a network with certain nodes and arcs eliminated to prevent repeat paths. Lombard and Church (1993) used GWN paths for a single objective corridor location problem, where it was desired to generate a set of spatially diverse short paths. Scaparra et

al. (2013) explore using multiple gateway nodes to generate more intricate path alternatives than what is possible with single-gateway nodes.

C. Bi-Objective Gateway Path Heuristic

Many exact algorithms for solving the bi-objective shortest path problem begin with solving the supported solutions as a set of weighted single-objective problems using the NISE algorithm. Let P be the set of $p = |P|$ supported solutions. Using NISE, computing P requires $2P + 1$ shortest path iterations in the worst case[6]. As solving a single-objective shortest path problem has polynomial complexity when using a specialized shortest path algorithm, and since the computational time of NISE is a function of the number of solutions, this makes the process weakly polynomial.

The gateway path heuristic is based upon the fact that shortest paths have already been computed by NISE to find the supported non-inferior points. In the heuristic though, rather than computing just the *one-to-one* shortest path, it is necessary instead to compute the *one-to-all* shortest path tree, $T(s)$. In practice, computing the one-to-all tree requires slightly more processing compared to a one-to-one shortest path, but both have the same overall computational complexity. In order to generate the gateway paths, an additional corresponding $T(t)$ must also be generated, followed by a scanning of all nodes or arcs to assemble the GWN or GWA paths respectively. Pseudocode for the GWN heuristic algorithm, which we call GWNH, is given below, and can be summarized by the following steps:

[6] Typically, NISE requires 2P – 1 shortest path iterations to solve all supported points; the additional runs are only necessary if initial single-objective solutions are later found to be weakly non-dominated.

Step 0: Solve $T(s)$ and $T(t)$ for the two separate single-objective shortest path problems

Step 1: Solve the appropriately weighted single-objective shortest path tree $T(s)$ problem to determine a supported point between known adjacent supported points. The weights are determined based upon the method of Cohon *et al.* (1979). Also solve the reverse shortest path tree $T(t)$. If no new supported point is found, the region between these points is fathomed.

Step 2: Scan all nodes, using the node labels of $T(s)$ and $T(t)$ to construct all GWN paths. Add each to a list of heuristic solution paths if no other path in the list dominates it. If a new path is added to the list, remove any paths from the list that are dominated by it.

Step 3: Select two adjacent un-fathomed supported points, and return to step 1. If all adjacent points are fathomed, terminate.

The GWA algorithm, called GWAH, is identical, except that the function scanGatewayNodes() is replaced by a function scanGatewayArcs(), which scans the network arcs rather than the nodes. The finer details of the NISE algorithm such as the equations for selecting the objective weights can be found in Cohon *et al*. (1979). For the bi-objective case, the weights for objective 1 (z_1) and objective 2 (z_2) are defined as a and $(1 - a)$ respectively, where $0 \leq a \leq 1$.

Overall, gateway paths are effective as a heuristic for a multi-objective problem because they represent the optimal *S*–*T* paths that traverse through some particular node or arc for a given weighted sum objective. Two iterations of a shortest path tree generating algorithm generates n gateway node paths, or m gateway arc paths, so in essence it generates far more solutions than solver iterations. In the case of gateway node paths, since one node-constrained path is generated for every node in the network, it enforces a spatial diversity in the candidate solution set, allowing for varied options to be considered for the final solution set.

Algorithm GWNH: Bi-Objective Gateway Node Path Heuristic Pseudocode

// $z(x) = a*z_1(x) + (1–a)*z_2(x)$
// $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ = the set of supported non-dominated solutions
// $Y = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_u\}$ = the set of supported and unsupported non-dominated solutions
// dij(u, a) = solves Dijkstra's algorithm with starting point u and objective weighted by a
// setA(σ_i, σ_j) selects next value of a based on the z_1 and z_2 values of σ_i and σ_j
// scanGatewayNodes(*T(u)*, *T(v),* Y) generates all gateway node (arc) paths, adds them to
// the set Y if not dominated by any $\upsilon \in Y$.
// recursiveNISE(σ_i, σ_j) performs divide and conquer to find supported solutions between
// σ_i and σ_j

function: main
$a = 0$
(σ_1, $T(s)$) = dij(s, a)
(–, $T(t)$) = dij(t, a)
Y = scanGatewayNodes(*T(s)*, *T(t),* Y)
$a = 1$
(σ_2, $T(s)$) = dij(s, a)
(–, $T(t)$) = dij(t, a)
Y = scanGatewaysNodes(*T(s)*, *T(t),* Y)
$\Sigma = \{\sigma_1\}$
Σ += recursiveNISE(σ_1, σ_2)

***function: recursiveNISE*(σ_i, σ_j)**
a = setA(σ_i, σ_j)
(σ_k, $T(s)$) = dij(s, a)
(–, $T(t)$) = dij(t, a)
Y = scanGatewayNodes(*T(s)*, *T(t),* Y)
if $\sigma_k \neq \{\sigma_i, \sigma_j\}$
  Σ += recursiveNISE(σ_i, σ_k)
  Σ += recursiveNISE(σ_k, σ_j)
  return Σ
else
  return σ_j
end

function: scanGatewayNodes(*T(u)*, *T(v),* Y)
for $\upsilon \in N$
  $\upsilon_u = s \xrightarrow[T(s)]{} u \xrightarrow[T(t)]{} t$
  if ($\upsilon_u \notin Y$ && υ_u is not dominated by any $\upsilon \in Y$)
    $Y = \{Y + \upsilon_u\}$
    $Y = \{Y - \upsilon; \upsilon \in Y$ and υ is dominated by $\upsilon_u\}$
  end
end
return Y

D. Experimental Analysis

1. Experimental Setup

The following four methods were programmed for comparison: GWNH, GWAH, the exact label correcting algorithm from Skriver and Anderson (2000), referred to as LCor, and the exact two-phase label correcting algorithm from Raith and Ehrgott (2009), referred to as 2LCor. For our shortest path sub-routine, we implemented Dijkstra's algorithm using a binary heap priority queue. All methods were programmed in the Java SE 6, and executed on a computer running OS X 10.8.2 with a 2.7 GHz Intel Core i7-3820QM processor. We evaluated both the speed and the quality of the results given on three raster networks: 1) a 20x20 manually fabricated raster, 2) an 80x80 subset of the Maryland Automated Geographic Information (MAGI) system database, and 3) a 100x160 subset of the same MAGI database. The MAGI database, funded by the Department of Energy, was developed for the state of Maryland for power plant and transmission corridor location planning. The first two networks originally included just a single objective, so a second objective cost, z_2, was randomly generated and scaled to match z_1 ranges. The 100x160 network contains two cost layers, with z_1 pertaining to an economic cost layer, and z_2 pertaining to an environmental impact layer.

Each raster was used to define networks of three r-radius types: 1) an orthogonal ($r = 0$) network, 2) a "queen's move" ($r = 1$) orthogonal and diagonal network, and 3) a "queen's plus knight's move" ($r = 2$) network. For further review of the tradeoffs between computational burden and geometric errors when generating networks from raster data the reader should consult Goodchild (1977) and Huber and Church (1985). Arc costs were generated from the node costs of the raster data in conjunction with their geometry, and

were *non-integer* in value, thus all algorithms used *double precision arithmetic*. The other studies cited in this paper have unanimously used integer cost networks for their analysis, thus our choice to use double precision arithmetic, as would be used by GIS software, incurs a penalty to overall computation speeds when compared to integer methods. Table 1 presents the general statistics of these two data sets.

Table 4. Test Networks

Raster	*r*	Nodes	Arcs	Arcs / Nodes
20x20	0	400	1,520	3.8
	1	400	2,964	7.41
	2	400	5,700	14.25
80x80	0	6,400	25,280	3.95
	1	6,400	50,244	7.85
	2	6,400	99,540	15.57
100x160	0	16,000	63,480	3.97
	1	16,000	126,444	7.90
	2	16,000	251,340	15.71

2. Evaluation of Heuristic

In addition to testing the heuristic runtimes compared to the exact algorithms, we also wanted to evaluate the quality of the solutions given by the heuristics. A number of approaches on evaluating heuristic solutions have been proposed by various authors, and are well summarized in Ghoseiri and Nadjari (2010). These include measures of average distance from the Pareto-set, average error distance, worst case distance, uniformity of the quality of the approximation set, extent of approximation, and others. We propose a new method that we think captures many of those measures in a more simplified approach. First, we define the Boundary of Unsupported Solution Search (BUSS) region of an approximate solution as being the area of the objective space where a new solution found in that region would constitute a viable candidate for a new unsupported non-dominated solution. Any new solution found outside of the BUSS region is guaranteed dominated by one or more points in the current estimate of the Pareto-solution set (see Figure 40).

An exact solution that has a set of more than one non-dominated points in objective space will have a positive $BUSS_e$ area. A heuristic solution will have a $BUSS_h \geq BUSS_e$, with the aim of those values being equal. We can define an error metric, E_{ratio} as

$$E_{ratio} = \frac{BUSS_h}{BUSS_e} \quad (12)$$

E_{ratio} is always ≥ 1, where a perfect heuristic solution would result in a value of 1. The closer the value is to a value of 1, the closer the upper bound generated by the heuristic is to approximating the exact solution frontier. In instances where $BUSS_e$ is small (i.e. the exact solution contains numerous unsupported points near the convex frontier of the supported points), a near-exact heuristic solution will be skewed toward having a large E_{ratio}. For this reason, we also introduce another metric E_{norm} which is the normalized E_{ratio}, defined as follows

$$E_{norm} = \frac{BUSS_h - BUSS_e}{BUSS_{supp} - BUSS_e} \quad (13)$$

where $BUSS_{supp}$ is the $BUSS$ region defined by only the supported solutions (also called the duality gap in Coutinho-Rodrigues *et al.* (1999)). If the heuristic solution is equal to the exact solution, then $E_{norm} = 0$. If the heuristic solution does not improve upon the supported solution set, then $E_{norm} = 1$. Anything in between means the heuristic solution set found unsupported points, where the closer to 0 the better the heuristic approximation. E_{norm} does not apply in the trivial case where exact solution set does not contain any unsupported non-dominated solutions.

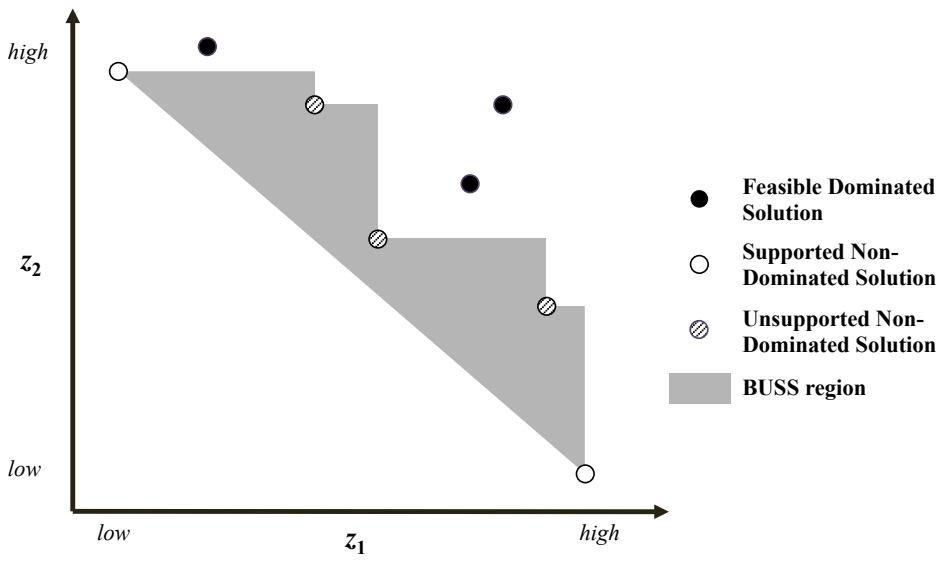


Figure 40. Defining the Boundary of Unsupported Solution Search (BUSS)

In addition to *BUSS* area, E_{ratio}, and E_{norm}, one can view the heuristic solutions in objective space to visually observe the quality of a solution set. Figure 41 a-d illustrate one heuristic solution set of Pareto paths from our experiments, in this case, the figures illustrate results from the Gateway Node Heuristic (GWNH) applied to the 20x20 $r = 2$ network.

Figure 41a displays the sixteen supported solutions to this minimization problem. Each solution is plotted in objective space, according to the z_1 and z_2 objective values of each particular solution. These supported solutions are the convex exact Pareto-optimal solutions easily found using the NISE algorithm. The white triangular regions between each supported solution are the $BUSS_{supp}$ regions. Figure 41b is a plot of all GWNH solutions found within the bounds of the extreme supported solutions. For the sake of clarity, solutions dominated by the extreme supported solutions have been eliminated and are not shown. Any of the heuristic solutions not contained within a $BUSS_{supp}$ region will be dominated by one or more supported solutions, yet many of the GWNH solutions fall inside the

$BUSS_{supp}$ areas. Figure 41c highlights with larger dark circles the Pareto-optimal subset of the GWNH candidate solution set, as well as $BUSS_{GWNH}$ defined by these solutions (the shaded regions defined by the GWNH non-dominated solutions). Compared to the $BUSS_{supp}$, the $BUSS_{GWNH}$ represents a significant reduction in area, especially within the larger $BUSS_{supp}$ regions. The larger plot in Figure 41d only shows the exact supported solutions and gives the exact $BUSS_e$ area in dark grey on top of the heuristic $BUSS_{GWNH}$ area in lighter grey. In many parts of the diagram, there is very little difference between the BUSS areas, meaning that the GWNH solution set is near, if not equal, to the exact non-inferior solution set. The inset graph shows a close-up of a particularly large $BUSS_{supp}$ area between supported points labeled A and B where there is a greater difference between the $BUSS_{GWNH}$ and $BUSS_e$ areas, highlighting where the heuristic solutions leave some room for improvement. In the inset, the exact solutions that define the $BUSS_e$ are shown.

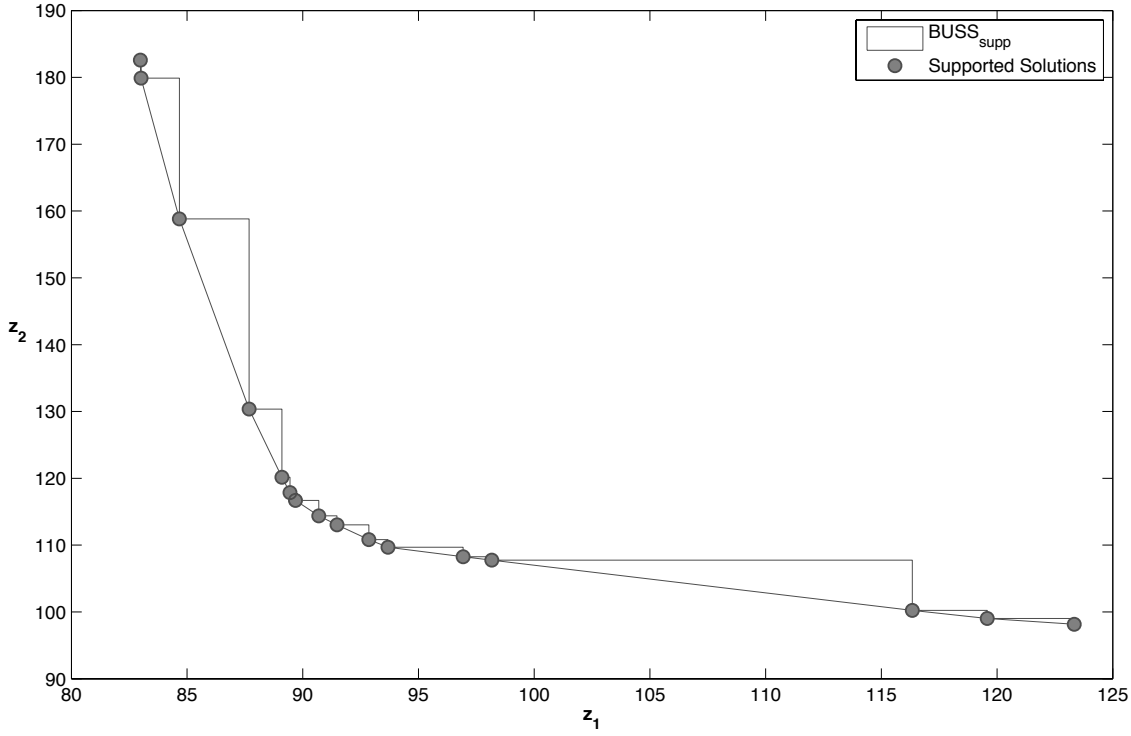


Figure 41a

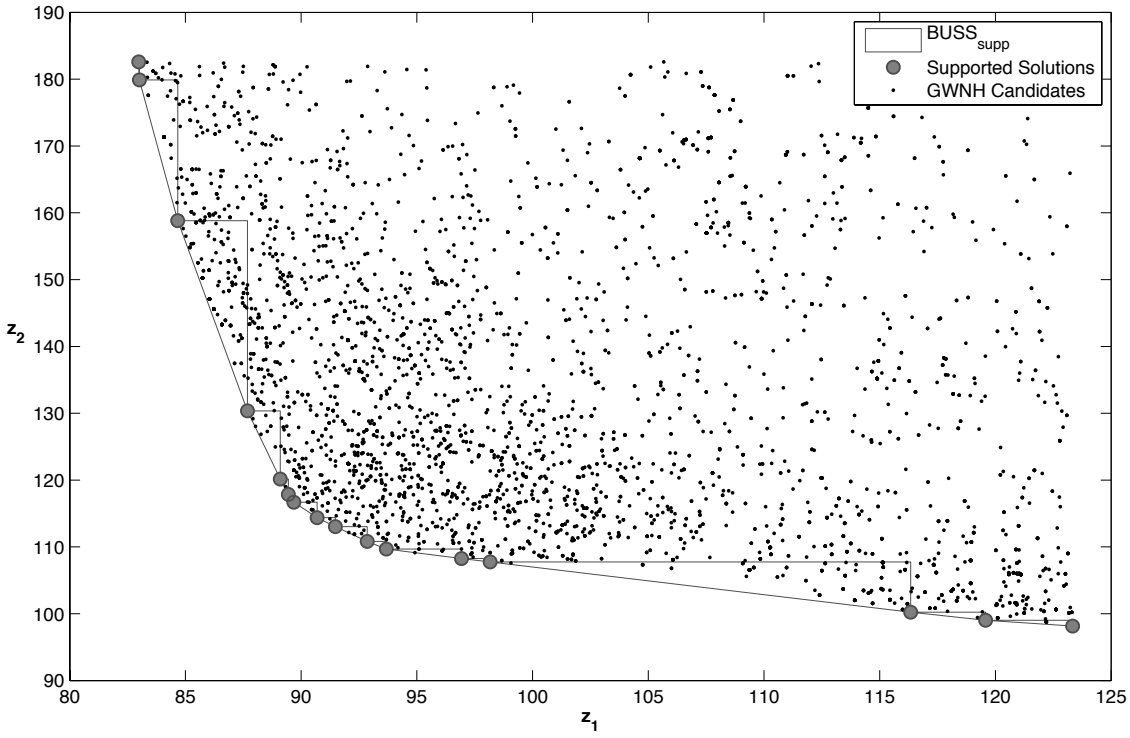


Figure 41b

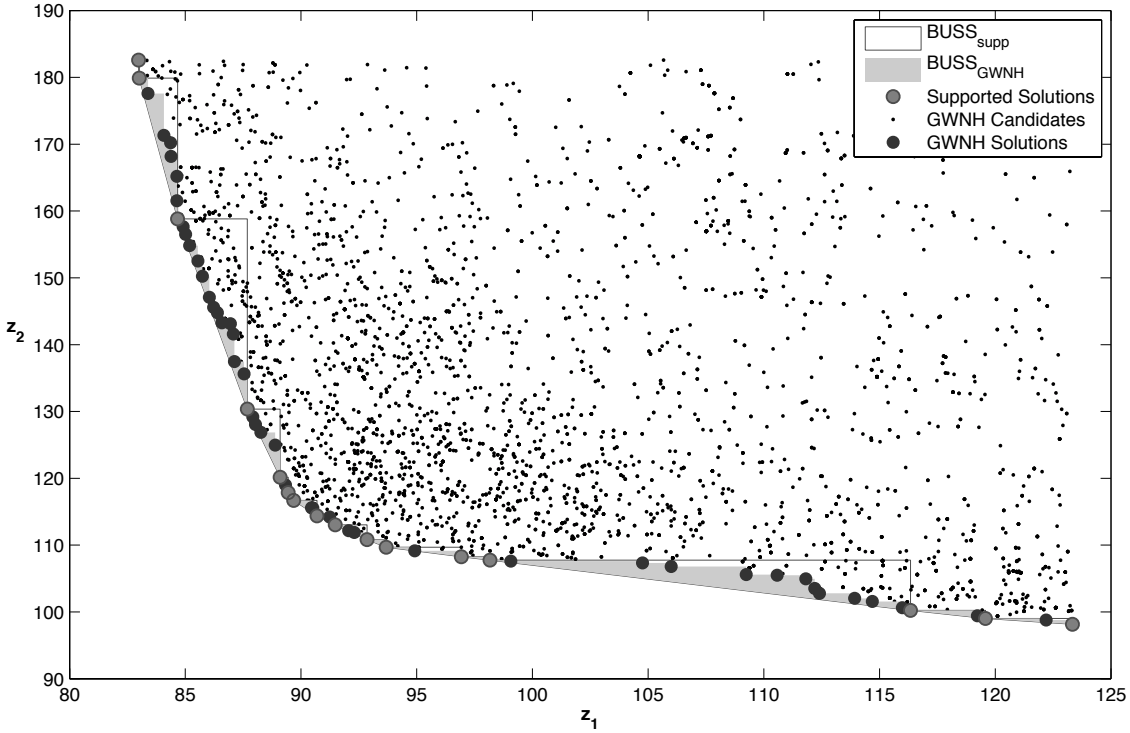


Figure 41c

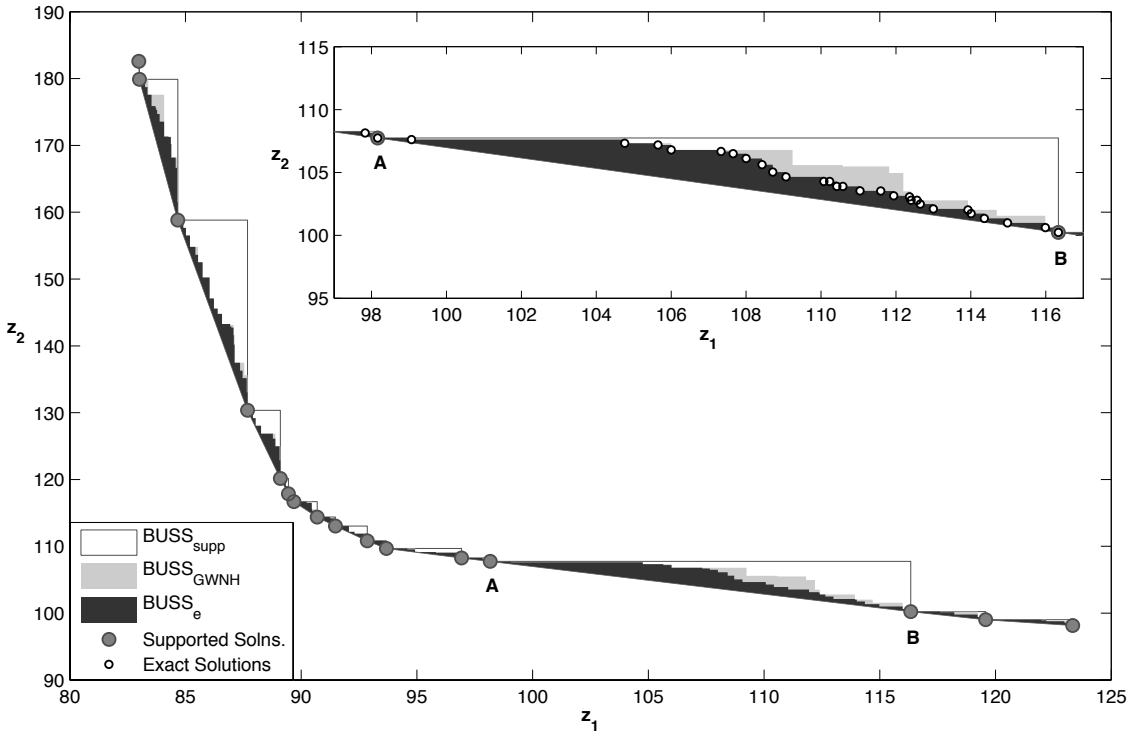


Figure 41d

Figure 41. Objective space evaluation on the 20x20 r=2 network. (a) Supported solutions only, (b) GWNH candidate solutions, (c) GWNH Pareto solutions, (d) exact vs. heuristic solutions

Solutions are depicted from the GWNH in Figure 41 rather than GWAH in order to show regions on the Pareto frontier where there is a clear difference between the $BUSS_{GWNH}$ and $BUSS_e$. In general, GWAH always gave better results due to the much greater number of candidate solutions, and therefore always had smaller differences between the exact and heuristic *BUSS* areas. In the depicted example, the ratio of the area of $BUSS_{GWNH}$ to the area of $BUSS_e$ is 1.22, indicating that heuristic frontier region is 22% larger than the exact region. Given this example, we now turn our attention to comparing exact and heuristic approaches.

E. Computational Results

Table 5 through Table 7 present the computational results of four algorithms: 1) the exact label correcting algorithm by Skriver and Andersen (LCor); 2) the two-stage label correcting of Raith and Ehrgott (2LCor); 3) the Gateway Node Heuristic (GWNH); and 4) the Gateway Arc Heuristic (GWAH). Table 5 gives the computation times in seconds for each method applied on the two different data sets using three different network definitions (orthogonal, queens, and queens plus knights). Table 6 contains the number of solutions contained in the Pareto-set for each network type, including the exact supported points and the all exact points (supported and unsupported). Table 6 also gives the number of candidate non-dominated solutions found by each of the two heuristics (solns) as well as the number of these that were exact optimal solutions (EOS) found by each heuristic. Table 7 provides a comparison of the quality of solutions between the exact approach and the two heuristic approaches, including the supported $BUSS_{supp}$ area, the exact $BUSS_e$ area area, and the

heuristic BUSS areas for each heuristic type. It also displays the error ratio, E_{ratio}, and the normalized error, E_{norm}, for each heuristic solution.

On the smallest of the 20x20 networks (400 nodes with $r = 0$ and 1), both problems were solved in a negligible amount of time by all four solution approaches. For those two cases, the simplest LCor method was fastest. While the two heuristics performed very well with small errors (all with less than 7% error), in networks of small size such as these it is best to simply use an exact approach. In the largest 20x20 network ($r = 2$), the label correcting algorithms are slower by a factor of between 2 to 3 when compared to the two heuristics. This difference in computation time can be attributed to the higher density of arcs, although exact methods continue to solve the problem in less than 0.1 seconds. The quality of the heuristic solution sets on this network as measured by the error ratio amounted to errors of 22% and 11.4% for the GWNH and GWAH, respectively, with normalized errors of 12.2% and 6.3%.

Table 5. Algorithm Computation Time Performance (in seconds)

		LCor	2LCor	GWNH	GWAH
2020	r = 0	0.008	0.018	0.012	0.014
	r = 1	0.020	0.031	0.016	0.019
	r = 2	0.070	0.065	0.024	0.038
8080	r = 0	0.233	0.413	0.265	0.300
	r = 1	1.341	1.493	0.480	0.589
	r = 2	39.756	22.577	1.387	1.757
100160	r = 0	0.119	0.403	0.346	0.366
	r = 1	3.052	1.853	0.844	1.015
	r = 2	50.710	26.468	2.474	3.183

Table 6. Number of Exact/Heuristic Solutions

		Exact Supported	Exact All	GWNH		GWAH	
		solutions	solns.	solns.	EOS	solns.	EOS
2020	r = 0	10	28	26	24	27	26
	r = 1	11	49	43	43	46	46
	r = 2	16	103	60	51	75	71
8080	r = 0	29	120	114	111	117	114
	r = 1	36	371	246	201	258	211
	r = 2	57	1339	465	317	647	503
100160	r = 0	11	22	21	21	21	21
	r = 1	19	199	129	110	134	116
	r = 2	35	718	344	292	383	322

Table 7. Quality of Exact/Heuristic Solutions

		Exact Supported	Exact All	GWNH			GWAH		
		$BUSS_{supp}$	$BUSS_e$	$BUSS_h$	E_{ratio}	E_{norm}	$BUSS_h$	E_{ratio}	E_{norm}
2020	r = 0	195.50	96.50	102.50	1.062	0.0606	99.50	1.031	0.0303
	r = 1	187.31	88.26	88.57	1.004	0.0032	88.46	1.002	0.0021
	r = 2	146.20	52.20	63.69	1.220	0.1222	58.16	1.114	0.0634
8080	r = 0	717.50	210.50	220.50	1.048	0.0197	217.50	1.033	0.0138
	r = 1	1069.60	271.86	297.42	1.094	0.0320	294.62	1.084	0.0285
	r = 2	964.22	216.02	259.86	1.203	0.0586	240.76	1.115	0.0331
100160	r = 0	87.5	49.50	50.5	1.020	0.0263	50.5	1.020	0.0263
	r = 1	907.5997	269.60	280.01	1.039	0.0163	279.25	1.036	0.0151
	r = 2	1121.629	285.49	298.99	1.047	0.0161	296.63	1.039	0.0133

For the least dense 80x80 network ($r = 0$), solution times were still fastest with LCor. This is in agreement with previous studies, which have shown that the LCor algorithm is very efficient on orthogonal grid networks. On the queen's move network ($r = 1$), the exact algorithms were more than 2x slower than the heuristics, while the heuristics yielded solutions that were both < 10% in error ratio. The real performance difference came in the most dense 80x80 ($r = 2$) network. In this case, the LCor took almost 40 seconds to solve, 2LCor improved this to 22.5 seconds, but GWNH ran much faster in 1.4 seconds and GWAH completed in less than 1.8 seconds. Despite completing in a fraction of the time, the heuristics were able to yield solutions that were within 20.3% of the exact for GWNH, and 11.5% for GWAH, with normalized errors of 5.86% for GWNH and 3.31% for GWAH. This large difference between error ratio and normalized error is due to the fact that the

exact Pareto-solution set has a very small $BUSS_e$, since in this case most of the exact unsupported points are near the convex frontier of the supported points.

On the 100x160 network, solution times followed the same trends found in solving the 80x80; with the $r = 0$ network the exact method was the fastest and the heuristic methods were faster for the higher densities ($r=1$ and $r=2$). Once again, the heuristics showed the greatest processing time improvements on the most dense $r = 2$ network, solving 10 times faster than the fastest exact methods. All heuristic methods, when run on the 100x160 network, resulted in error ratios and normalized errors of < 5%. For networks larger than those in our experiments, we expect the runtime differences to be even more pronounced, and indeed, computational results from both Skriver and Andersen, and Raith and Ehrgott found that solution times for LCor and 2LCor grew in an exponential fashion as the network size and density increased.

In summary, for the lower radius networks ($r = 0$ or $r = 1$), most if not all of the heuristic solutions were in-fact part of the exact optimal solution (EOS) set (Table 6). For the higher density networks, a somewhat smaller proportion of the heuristic solutions were EOS. For example, in the $r = 2$ 100x160 network, the exact Pareto-optimal frontier consisted of 718 different paths. The GWNH found 344 approximate Pareto-optimal solutions, of which 292 were in-fact exact Pareto-optimal solutions. Likewise, GWAH found 383 Pareto-optimal heuristic solutions, of which 322 were Pareto exact solutions. Along with the GWAH E_{ratio} result of 3.9% area difference between heuristic and exact, this indicates that the GWAH heuristic was able to produce a set of solutions where most were actually exact solutions, and those that were not represented an 3.9% upper bound above the exact solution set, all while computing in approximately one tenth of the time to compute the exact solution.

Comparing the two heuristics to each other, the GWNH always ran slightly faster than the GWAH, while GWAH always produced a closer approximation of the Pareto frontier when compared to the exact Pareto frontier. Both methods are based on computing the same number of shortest path trees, so the time difference is due to scanning fewer gateway node paths as compared to scanning the number of gateway arc paths (GWN vs. GWA paths) after the shortest path trees have been computed. The method of inserting new gateway paths into the approximate solution set employed a binary search on a sorted array list of currently non-dominated solutions. From this, the method then checks if other paths in the set are dominated by the newly inserted solution, and those are removed from the set. This process appears to consume a significant portion of the computation time, so for larger networks there could be a more pronounced tradeoff between the two heuristics in terms of computation time and solution quality.

F. Concluding Remarks

This study presents a new heuristic method for generating an approximate Pareto-optimal solution set for a biobjective shortest path problem. The new method has two variants for efficiently generating a large set of candidate paths, one making use of gateway node paths, and the other using gateway arc paths. These heuristics were compared with two exact algorithms, LCor and 2LCor, that had performed well in previous studies by Skriver and Andersen (2000), and Raith and Ehrgott (2009). When applied on small orthogonal grid networks, the exact algorithm LCor ran faster than other the methods, and made the use of a heuristic unnecessary. For larger network sizes and densities, as would be the case when solving a large-scale corridor location problem, the exact algorithms took considerably more computation time. In fact, the computational burden of the two exact methods has been

shown to be exponential as a function of problem size. Moving to extremely large problem sets, these two exact methods become intractable, and thus fast heuristic methods become necessary. In these instances, the GWN and GWA heuristics were shown to provide a high-quality solution set in a fraction of the computational effort. Additionally, if more than two objectives are considered, the complexity of the problem increases in dimension, making the exact approaches even more daunting. The heuristic approaches outlined in this chapter could extend to higher dimensions when used in conjunction with a multi-dimensional NISE approach such as the one developed by *Solanki et al.* (1993), and could offer a faster option for approximating the Pareto-front for three or more objective shortest path problems.

VI. Unsupported Multi-Objective: An Exact Biobjective Shortest Path Method with Gateway Heuristic and Supported Point Upper-Bounds

A. Introduction

Heuristic approaches are often implemented to define lower and/or upper bounds to the optimal solutions of difficult-to-solve problems (Danna *et al.* 2005, Hewitt *et al.* 2010). These bounds then restrict the solution space over which a solver computes, and can often speed up the search for the exact optimal solution. In Chapter V, we introduced a fast heuristic that approximates the Pareto frontier of a biobjective shortest path problem. This approach was shown to efficiently calculate a set of paths that closely approximate the exact Pareto optimal set when evaluated in objective space. This approximation approach represents an excellent method to generate a candidate set from which to select viable alternatives, particularly in a real-world design scenario where there exists inherent uncertainly in the spatial data, and where data sets are large and present formidable computation issues. It is logical to ask the question of whether this approximation approach could be harnessed to speed up the computation of an exact Pareto optimal solution set to a biobjective shortest path problem. This chapter introduces a new exact algorithm for solving a biobjective shortest path problem based on an enumerative approach first introduced by Raith and Ehrgott (2009), but using the approximation heuristic as an improved upper bound to speed up overall computation times. In addition, valid bounds are introduced to avoid enumerating solutions already dominated by supported solutions. This further limits the enumeration search region in hopes of improving the performance of enumerative schemes.

Specialized exact algorithms for the biobjective shortest path problem fall under two categories: labeling approaches and enumeration approaches. Raith and Ehrgott (2009) reviewed the state-of-the-art approaches of the time, and introduced two-phase variants of a label-setting algorithm by Guerriero and Musmanno (2001) and a label-correcting approach by Skriver and Andersen (2000). They also introduced a new two-phase enumerative technique called 2NSP that used the Near Shortest Path (NSP) enumeration algorithm developed by Carlyle and Wood (2005). They then evaluated the performance of those methods on three types of networks: 1) regular grid networks with random discrete uniform distribution costs, 2) random networks generated by the NetMaker network generation technique described by Skriver and Andersen (2000), and 3) road networks acquired from the Tiger Road Networks for the 9th DIMACS implementation challenge, which were originally sourced from U.S. Census data. After developing our improvements to 2NSP, we replicated the experiments on these same networks with one exception in order to compare the performance of the new approach to the existing published methods. In lieu of the random-cost grid networks, we chose to test on spatial terrain-based networks of the kind that would be used in a transmission corridor location.

B. Near Shortest Path (NSP) Biobjective Shortest Path (BSP) Algorithm

Raith and Ehrgott (2009) introduced a two-phase enumerative biobjective shortest path (BSP) algorithm based upon the Near Shortest Path (NSP) enumeration technique developed by Carlyle and Wood (2005). Earlier enumerative BSP algorithms had used *k*-shortest path subroutines (Coutinho-Rodrigues *et al.* 1999), but Carlyle and Wood showed in their paper that NSP was a much faster approach to path enumeration due to not being restricted to find the paths in ascending path-length order. The NSP algorithm searches for all paths that are

within some threshold $D = (1 + \varepsilon) \times L_{sp}$, where D is the maximum length path to be returned, ε is a positive real number, and L_{sp} is the length of the shortest path. For example, if $\varepsilon = 0.05$, then the NSP will output all paths that are within 5% of the length of the shortest path.

The Raith and Ehrgott NSP-BSP algorithm is a two-phase method, which is denoted here as 2NSP[7]. In the first phase, the supported solutions are found by solving weighted composite single-objective shortest path problems (Cohon *et al.* 1979). The remaining solutions are found in the Boundary of Unsupported Solution Search (BUSS) regions, sometimes referred to in the literature as the *duality gap*. These BUSS regions initially are defined only by the supported solutions, and consist of triangular areas between supported non-dominated (Pareto) solutions where unsupported non-dominated solutions may exist.

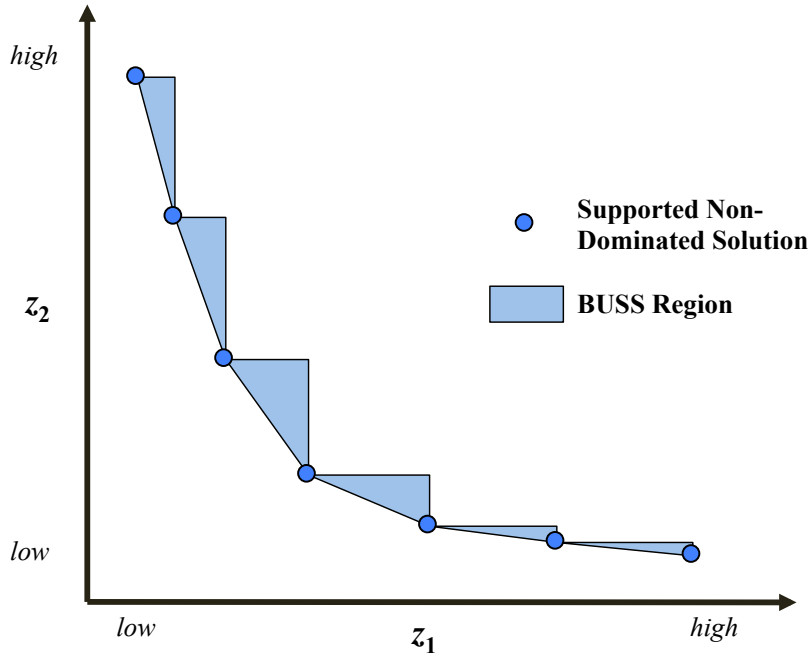


Figure 42. Initial BUSS regions between the supported non-dominated solutions

The second phase searches for the unsupported non-dominated solutions within each individual BUSS region. For each BUSS region, the composite weighting for the graph is set

[7] Raith and Ehrgott call it NSPD, where the D means it uses Dijkstra's algorithm for phase-1

so that the two supported solutions that define the specific BUSS region have equal composite objective value (points A and B in Figure 43). This defines a projection axis from which composite objective values are measured in a direction perpendicular from this projection line. Two solutions with different *component* but equal *composite* objective values for the given weight will lie on a line parallel to the projection axis.

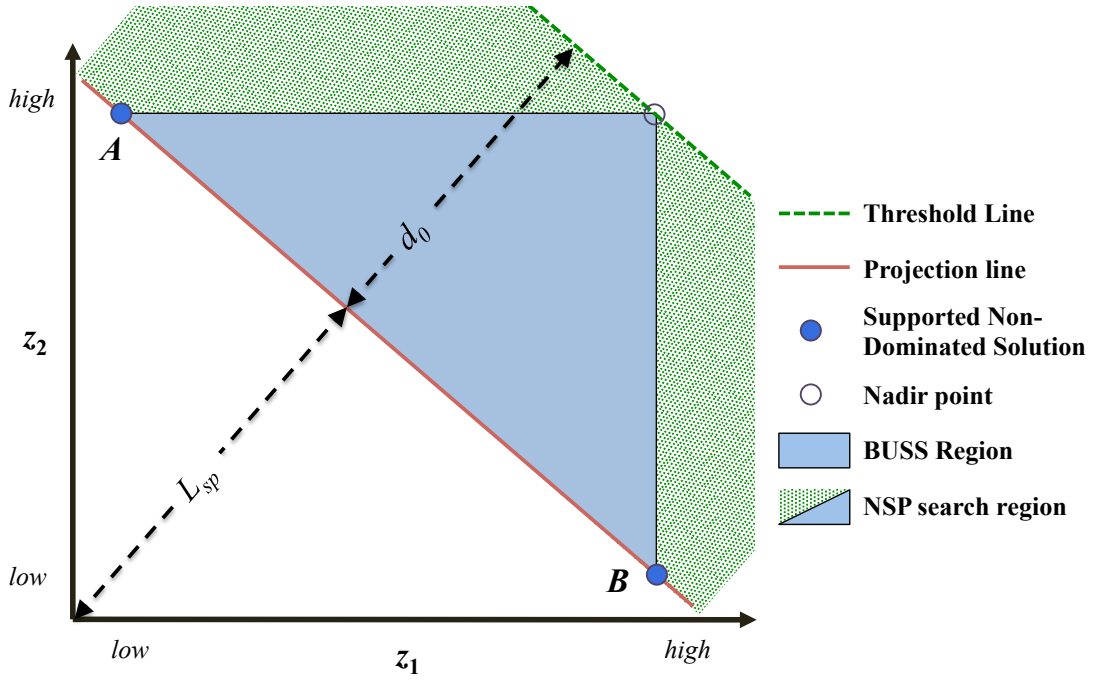


Figure 43. Setting the initial threshold for a 2NSP BUSS region. BUSS region is a subset of the NSP search region.

Recall that the NSP method enumerates all solutions within a threshold $D = (1 + \varepsilon) \times L_{sp}$. If we define $d = \varepsilon \times L_{sp}$, where d represents the amount of length greater than the shortest path length to reach the total threshold D, then this can be rewritten as $D = L_{sp} + d$. The 2NSP algorithm evaluates each BUSS region independently, and defines the initial threshold $D_0 = L_{sp} + d_0$, where d_0 is the perpendicular distance from the projection line to the most distant nadir point of the BUSS region (see Figure 43). This nadir point is initially located at the right-angle vertex of the BUSS right triangle. In objective space, this threshold is a line

parallel to the projection axis that intersects the nadir point, and defines a search region that is an infinite strip between the projection line and the threshold line, and contains the entire BUSS region. Despite an infinite search area, feasible solutions will only exist in a bounded polygon sub-region due to the convexity of the supported solutions, which represent a lower bound on the feasible solution set.

The 2NSP method begins by enumerating paths with a composite objective $z_c < D_0$. Many of the output paths may fall outside of the BUSS region, but a path found that has z_1 and z_2 objective values that place it inside the BUSS region will in-turn dominate some portion of that region, at which point the shape of the BUSS must be updated to reflect this. If the dominated region includes the nadir point that limited the value of threshold D, then the threshold is updated, further restricting the area to be searched and reducing computation to completion. This new threshold $D_{new} = L_{sp} + d_{new}$ is set to the most distant nadir point of the updated BUSS region, as referenced from the projection line (see Figure 44).

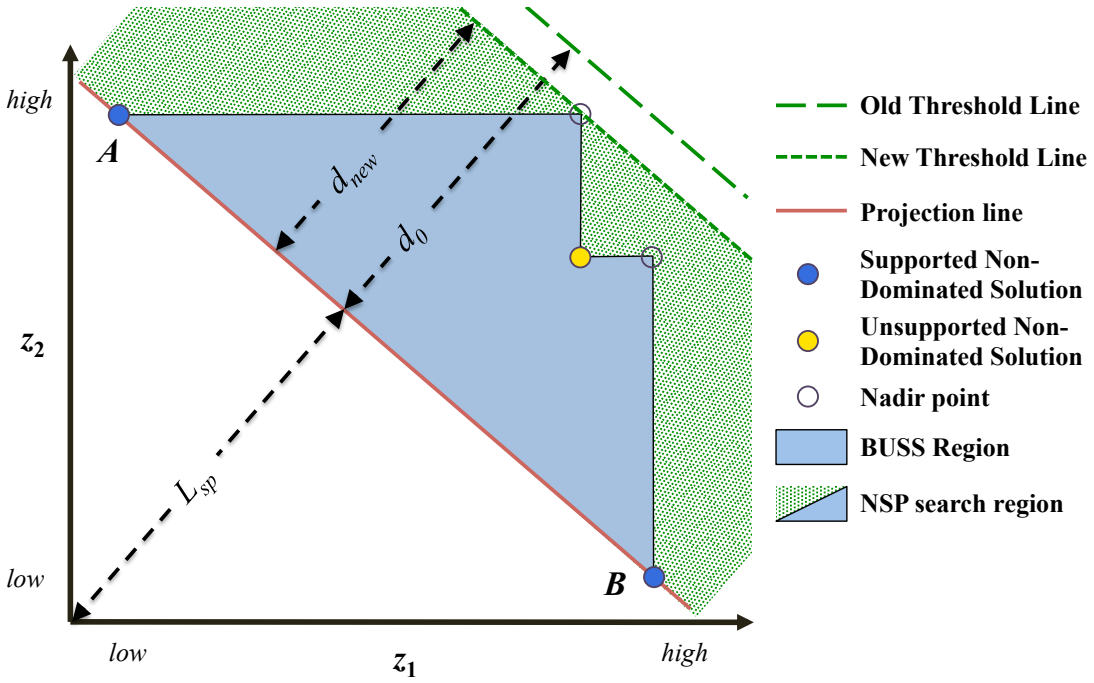


Figure 44. Updating the 2NSP threshold as solutions reduce the area of a BUSS region

Each new solution that falls into the updated BUSS region dominates some portion of the region, requiring an update to the set nadir points. A new solution may also dominate one or more of the previously found solutions, thus the program must check for this and remove any dominated solutions from the Pareto set. The NSP algorithm runs until all paths within the threshold have been enumerated, at which point all unsupported non-dominated solutions between the two supported solutions used to define the BUSS region have been found. The 2NSP method is then repeated for each adjacent pair of supported solutions until all BUSS regions have been analyzed and the entire Pareto set has been found.

C. Improvements to the NSP BSP Algorithm

1. Gateway Heuristic Upper Bound

In Chapter V, a new heuristic method was developed that was capable of efficiently computing an effective approximation of the unsupported Pareto solution set to a biobjective shortest path problem. Since this method offers excellent solution sets with little computational effort, it is proposed here to use it as an initial solution upper-bound for the biobjective NSP approach. Thus, it begins with heuristically solving for an approximation of the non-dominated points using either the biobjective gateway node (GWN) or gateway arc (GWA) heuristic. Doing so requires twice as many shortest path solver iterations for the supported points as the standard 2NSP algorithm. The heuristic solutions are then added to the candidate Pareto solution set and the NSP threshold D is updated according to the same rules as discussed in the prior page and in Figure 44. In Chapter III.B.1 we showed that NSP path enumeration (or any combinatorial path enumeration for that matter) has a tendency to grow exponentially as the threshold value increases, thus a more tightly restricted threshold value will result in significant reductions in overall computational effort. Two versions of

the 2NSP algorithm with an initial gateway upper bound were implemented: one using gateway node candidate paths (called GWN2NSP), and another using gateway arc candidate paths (called GWA2NSP). Gateway node paths generate *n* path candidates for each weighting, where *n* is the number of nodes in the network. Gateway arcs generate *m* path candidates for each weighting, where *m* is the number of arcs in the network. As path candidates are generated they are added to a lexicographically sorted list of non-dominated path candidates. When a new candidate is added to the list there must also be a check to see if the new candidate dominates any existing candidates in the list. Overall, this adds some to the computational overhead, first in doubling the number of shortest path solver iterations during the first phase of the algorithm (the NISE supported solution phase) in order to generate the gateway paths, and also with a binary search which is needed to identify if a path is non-dominated, and if so then inserting the path into the list and removing any paths dominated by the newly inserted path. If the enumerative second phase of NSP-based algorithm takes an exceptional amount of time to compute, then the tighter NSP bounds generated by the presence of an initial approximate non-dominated path set from the gateway heuristic has the potential to significantly reduce overall computation times.

2. Supported Solution Dominance Bounds

The single objective Carlyle and Wood (2005) NSP algorithm uses depth-first-search (DFS) to find all paths with length below a given threshold, *D*. The DFS uses a first-in last-out stack data structure to build paths by adding arcs to the stack so long as they could result in a path that meets that threshold criterion. A naïve approach would add arcs to the stack so long as the total path length is less than *D*, and not add an arc if adding it would cause the path length to exceed *D*. The original NSP algorithm developed by Byers and Waterman

(1984) added an elegant innovation to this idea by using information from an initial reverse shortest path tree in order to improve the efficiency of the enumeration. As an example, suppose one is enumerating near shortest paths from S to T. NSP uses a stack data structure to contain arcs that make up a path P from node S to node x, with length $L(x)$. An arc (x,y) from node x to node y will have a cost $c(x,y)$, and the shortest path from a node x to the destination node T (as computed initially by the reverse shortest path tree) has cost $d'(x)$. Let us consider a node $u \neq T$. Some path P of length $L(u)$ led us from node S to node u having a path length $\leq D$. Let us now consider adding arc (u,v) to the path P. The arc is added to the path if the following is true.

$$L(u) + c(u,v) + d'(v) \leq D \qquad \textbf{(14)}$$

This means that arc (u, v) is added only if the sum of the existing path length to u, plus the cost of the arc from u to v, plus the shortest possible path length from v to the destination, is less than the threshold D. Using the $d'(v)$ information allows the algorithm to know if there does not exist any feasible possibility of reaching the destination without exceeding the limit D. If so, it is no longer necessary then to search that portion of the graph. This approach may be considered similar to the approximation of the shortest path length from a candidate node to the destination used in the A* algorithm developed by Hart *et al.* (1968).

The biobjective variant of the algorithm, 2NSP, is the same except it uses a composite weighted single objective function z_c that is a function of the individual competing objectives z_1 and z_2, weighted by α, where $z_c = \alpha \times z_1 + (1 - \alpha) \times z_2$, and α is determined by the slope of the supported points that define the particular BUSS region being analyzed. The 2NSP method for the bi-objective case at each iteration checks if the composite function meets the threshold criterion.

$$L_c(u) + c_c(u,v) + d'_c(v) < D_c \quad (15)$$

This results in paths enumerated that all fall within the given BUSS region used to define the composite weighting. But this boundary also encompasses areas outside of the BUSS region, resulting in numerous paths enumerated that are dominated by the previously found supported non-dominated solutions (see Figure 45).

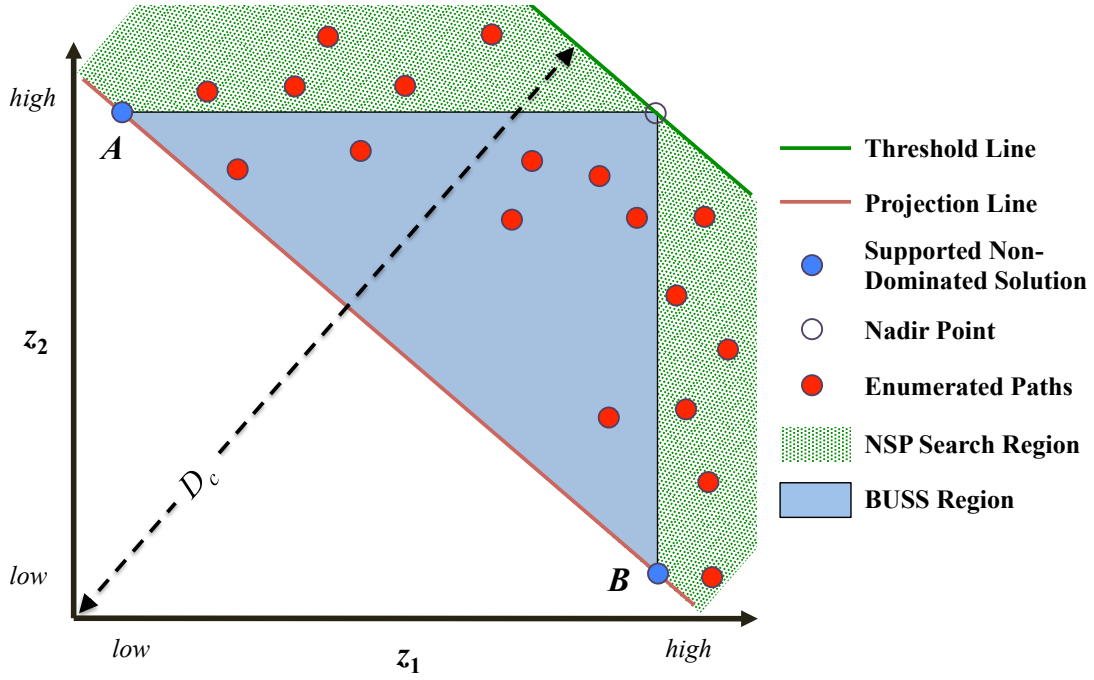


Figure 45. Paths enumerated by 2NSP, both inside and outside the BUSS region

The paths found that are outside of the BUSS region constitute wasted computational effort, as they are always dominated by a supported solution. Our tests on various raster networks have shown that a minimum of 75%, and as many as 99.88% of the paths enumerated by 2NSP were located in objective space outside of the BUSS regions.

Given this, it is logical to add a check that eliminates these paths from being found, by not only testing if a proposed arc added to the stack would exceed the threshold D_c, but also if it would result in a path dominated by either of the supported points. This requires storing

two additional labels during the initial NISE supported solution computation. Initially, biobjective NISE solves for α weightings of 0 and 1, corresponding to independent single-objective solutions that ignore the existence of other objectives. Storing the reverse shortest path tree distance for these weightings for each node gives labels for the best possible future outcome with respect to each independent objective. We call these labels $d'_1(v)$ and $d'_2(v)$ for the z_1 and z_2 objectives, respectively. Then, at the point where 2NSP queries an arc to determine if it should be added to the stack (equation 15), we add the following additional queries, where A and B are the supported points of the BUSS region.

$$L_1(u) + c_1(u,v) + d'_1(v) < z_1(B) \quad (16)$$

$$L_2(u) + c_2(u,v) + d'_2(v) < z_2(A) \quad (17)$$

This puts additional bounds on the NSP computation, dramatically reducing the size of the NSP search region to only that of the BUSS region (see Figure 46), and reducing the total number of paths enumerated. We denote this bounded algorithm as B2NSP.

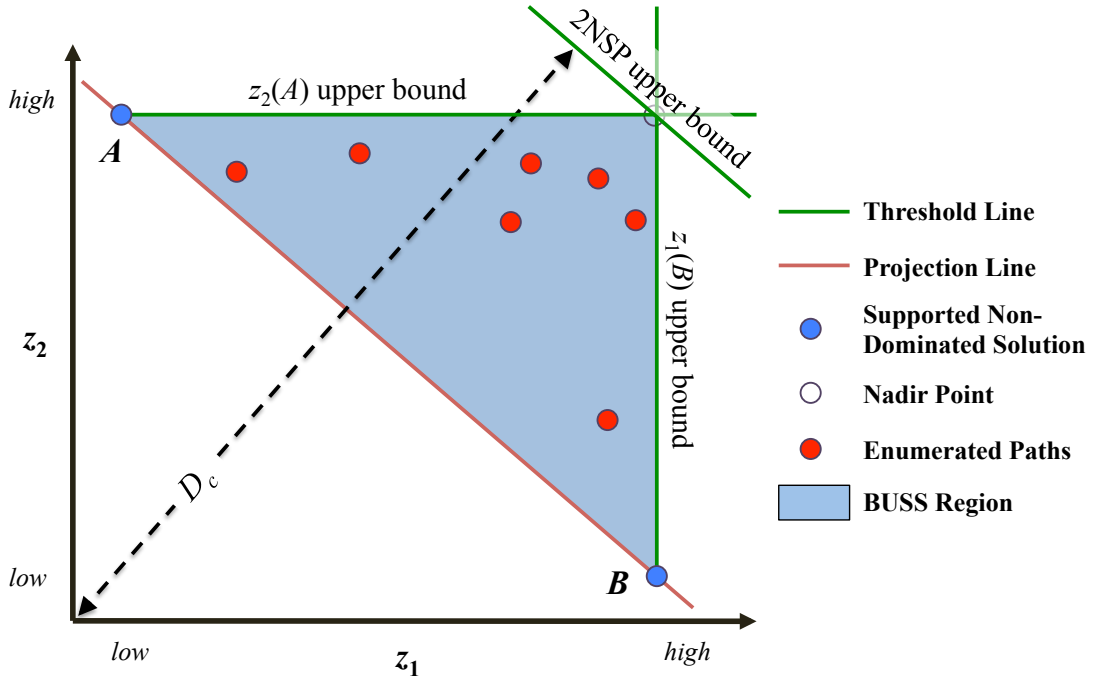


Figure 46. B2NSP supported solution bounds restricting enumerated paths to only within the initial BUSS region

3. Combining Both Improvements

The two improvement approaches described above are not mutually exclusive, and can be combined to further improve the efficiency of the enumerative method. We added the combined algorithms to our experiments, and denote the supported bounds gateway node variant as BGWN2NSP, and the supported bounds gateway arc variant as BGWA2NSP.

D. Numerical Experiments

1. Data Sets

Similar to Raith and Ehrgott (2009), the algorithms were tested on three types of networks: grid, random, and road networks. The random and road networks were the same as those used by Raith and Ehrgott, while we have changed the grid networks to be terrain-based GIS raster networks representative of the type that would be used for a transmission line corridor location application. Details on each network type are described below.

i. Raster Networks

The first type of network Raith and Ehrgott (2009) used to test their biobjective shortest path algorithms were orthogonal grid networks with random integer arc costs. Because of the main focuses of this dissertation, it was decided to test the bi-objective shortest path approach on spatial terrain-based networks of the kind that would be used in a transmission corridor location. While terrain networks are similar to grid networks in that they contain a regular connectivity pattern, they can be made more dense when they incorporate queen's move connectivity and/or knight's move connectivity between nodes (Goodchild 1977).

Such connectivity also requires the use of non-integer arc costs to account for the geometry, thus the programs we implemented use floating point arithmetic rather than integer math.

The particular terrain networks used were the same as those used in the previous chapter, namely 1) a 20x20 manually fabricated raster, 2) an 80x80 subset of the Maryland Automated Geographic Information (MAGI) system database, and 3) a 100x160 subset of the same MAGI database. The first two networks originally included just a single objective cost layer, so a second objective node cost, z_2, was randomly generated and scaled to match z_1 ranges. The 100x160 network contains two cost layers, with z_1 pertaining to an economic cost layer, and z_2 pertaining to an environmental impact layer.

Each raster was used to define networks of three r-radius types: 1) an orthogonal ($r = 0$) network, 2) a "queen's move" ($r = 1$) orthogonal and diagonal network, and 3) a "queen's plus knight's move" ($r = 2$) network. Table 8 contains pertinent statistics of these networks, including size, origin-destination node locations, r radius value, number of supported Pareto (non-dominated) solutions, and the total number of Pareto solutions.

Table 8. Raster Test Networks

Name	Dimensions and OD	r	Nodes	Arcs	Supported Pareto Solutions	All Pareto Solutions
R1	20x20	0	400	1,520	10	28
R2	SW to NE	1	400	2,964	11	49
R3		2	400	5,700	16	103
R4	80x80	0	6,400	25,280	29	120
R5	SW to NE	1	6,400	50,244	36	371
R6		2	6,400	99,540	57	1339
R7	100x160	0	16,000	63,480	11	22
R8	SW to NE	1	16,000	126,444	19	199
R9		2	16,000	251,340	35	718
R10	100x160	0	16,000	63,480	9	20
R11	NW to SE	1	16,000	126,444	17	109
R12		2	16,000	251,340	44	495

ii. Random NetMaker Networks

When Skriver and Andersen (2000) published their label correcting biobjective shortest path algorithm, they tested their approach on random networks. Originally they had tested their algorithm on NETGEN networks in order to replicate earlier work by Huarng *et al.* (1996), but they found that the simple structure of the NETGEN networks resulted in data sets with very few non-dominated paths. They then developed a new random network generator that they called NetMaker that used random Hamiltonian cycles to generate random networks with more non-dominated paths. They felt that this structure would better approximate real life problems, although it is unclear why then they did not simply try using real world data. The two objective costs for NetMaker networks are randomly assigned integers with approximately half of the arcs having a high c_{ij}^1 value and a low c_{ij}^2 value, and the other half of the arcs have a low c_{ij}^1 and a high c_{ij}^2. Full details on the structure and methods used to generate the networks can be found in Skriver and Andersen (2000).

Raith and Ehrgott (2009) also used the NetMaker networks in order to compare the different algorithms. They coded their own version of NetMaker using the same rules developed by Skriver and Anderson, and included additional features to further modify the network properties. Raith (2010) noted in a later paper that an error in their NetMaker generator code created networks with shortcuts from the origin to the destination, resulting in experiments where the computations solved instantaneously no matter what size of network was used. Their networks had very few non-dominated solutions because of the shortcut paths dominating all other possible routes. We were able to acquire the original NetMaker generator program developed by Skriver and Anderson which did not include this

error, and we used that code to generate our own errorless random networks for use in our computational experiments that are consistent with the previous literature.

The NetMaker generator program allows for certain input parameters when generating new networks. It first prompts for how many nodes in the network, and creates a random Hamiltonian cycle with all of the nodes. Then it prompts for parameters defining the maximum and minimum number of outgoing arcs that should emanate from each node. Finally, the I_{node} parameter is the node interval, i.e. the maximum allowed range of any outgoing arc. Essentially, if at node 100, and the $I_{node} = 20$, then outgoing arcs can terminate at any node within the number range 90 to 110. Table 9 lists the NetMaker networks used in our experiments, their parameters, and their properties. Networks NM1 to NM20 match the parameters of the Raith and Ehrgott experiments, where they tested on networks of up to 21,000 nodes and approximately 530,000 arcs. This work extends the experiments to networks up to 15 times larger than the Raith and Ehrgott networks (NM21 to NM30), the largest with 315,000 nodes and 7,876,183 arcs. Interestingly, the number of non-dominated solutions did not increase with network size, and seemed to correlate more with the parameters for minimum and maximum outgoing arcs per node.

Table 9. NetMaker Test Networks

Name	I_{node}	Min Outgoing Arcs	Max Outgoing Arcs	Nodes	Arcs	Supported Pareto Solutions	All Pareto Solutions
NM1	20	5	15	3,000	30,088	11	25
NM2	20	1	20	3,000	31,590	4	9
NM3	50	5	15	3,000	30,046	9	22
NM4	50	1	20	3,000	31,746	8	22
NM5	50	10	40	3,000	74,942	7	31
NM6	20	5	15	7,000	70,057	10	38
NM7	20	1	20	7,000	73,676	5	11
NM8	50	5	15	7,000	69,483	7	10
NM9	50	1	20	7,000	73,162	8	19
NM10	50	10	40	7,000	174,208	15	37
NM11	20	5	15	14,000	139,989	13	41
NM12	20	1	20	14,000	147,288	6	18
NM13	50	5	15	14,000	140,094	8	24
NM14	50	1	20	14,000	145,755	10	33
NM15	50	10	40	14,000	350,600	8	32
NM16	20	5	15	21,000	209,770	13	33
NM17	20	1	20	21,000	223,245	9	39
NM18	50	5	15	21,000	209,575	10	24
NM19	50	1	20	21,000	220,495	10	32
NM20	50	10	40	21,000	526,615	11	38
NM21	20	20	5	28,000	279,658	6	26
NM22	50	50	10	28,000	699,870	14	36
NM23	20	20	5	70,000	700,845	9	28
NM24	50	50	10	70,000	1,747,489	12	43
NM25	20	20	5	100,000	1,002,682	10	42
NM26	50	50	10	100,000	2,500,462	10	47
NM27	20	20	5	210,000	2,101,252	11	38
NM28	50	50	10	210,000	5,247,056	15	40
NM29	20	20	5	315,000	3,147,674	10	30
NM30	50	50	10	315,000	7,876,183	11	43

iii. Road Networks

The third type of network we used in the experiments reported here were road networks acquired from the Tiger Road Networks for the 9th DIMACS implementation challenge[8]. These networks were originally sourced from U.S. Census TIGER data, but then cleaned for the competition to contain only road data. The two costs for each arc are the travel time in seconds, and the travel distance in meters. The travel time was determined by DIMACS by multiplying the travel distance of an arc by one of four different road quality factors. One can see that these two objectives are correlated with each other, rather than competing, so

[8] http://www.dis.uniroma1.it/challenge9/data/tiger/

overall the Pareto frontiers have relatively few solutions for a given network size. Like Raith and Ehrgott (2009), this experiment used the Washington DC (DC), Rhode Island (RI), and New Jersey (NJ) networks. They did not list which nodes were used for their origin and destinations, so instead OD pairs were randomly selected. Table 10 lists the road networks used and their properties. It should be noted that Raith and Ehgott used versions with an artificially high-cost Hamiltonian cycle added to the network to ensure connectivity. Instead, the base network without the added cycle was used, and different OD pairs were selected if a pair turned out to be not connected.

Table 10. Road Test Networks

Name	State/ District	Nodes	Arcs	Average Pareto Solutions	Minimum Pareto Solutions	Maximum Pareto Solutions
DC1–11	Washington, DC	9,559	29,818	22.6	3	76
RI1–11	Rhode Island	53,658	138,426	20.2	2	92
NJ1–11	New Jersey	330,386	872,072	82.9	12	221

2. Computational Results

All algorithms were coded in Java 7 using the Processing library (www.processing.org) for graphical support in visualizing networks and results. The Processing library does slightly slow down code runtimes, but since it was used for all codes the relative performance between algorithms is consistent. All computational tests were performed on an Apple laptop with an Intel Core i7-3820QM, 2.7 GHz quad-core processor, 16GB of RAM, running OS X 10.9. Whenever the computation exceeded 10 hours, computation was stopped. Such instances are indicated by a dash in the table of results.

i. Raster Networks

Table 11 and Table 12 include the computational runtimes and the total number of paths enumerated for the labeling and enumeration algorithms on the raster test networks. Figure 47 displays the runtime results in a graphical chart. The first observation is the exponential runtime behavior of the enumeration methods on the raster terrain networks as the networks grew in size. Runtimes for all of the methods were all less than 0.1 seconds on all of the 20x20 networks. On R4-R6, the gateway and supported solution upper bounds both made significant improvements over 2NSP, with the best results coming from combining the two. On R4, the BGWN2NSP and BGWA2NSP methods sped up computation over 2NSP by greater than 843 times by requiring less than 0.07% of the number of paths to be enumerated. Tests on R5 showed similar significant speedup, and experiments on R6 made the problem solvable using BGWA2NSP and BGWN2NSP in 1.5 hours, while 2NSP and B2NSP were stopped incomplete after 10 hours. Despite the improvements though, all problems failed to solve after 10 hours for all enumerative methods on R7-R12, while both labeling methods (LCor and 2LCor) solved these problems in less than a minute. The GWA bounded algorithms typically performed equal to or better than their corresponding GWN versions, making the additional overhead of generating a GWA candidate set worthwhile on this type of network.

Table 11. Algorithm runtimes on raster networks

	LCor	2LCor	2NSP	B2NSP	GWN2NSP	GWA2NSP	BGWN2NSP	BGWA2NSP
	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)
R1	0.008	0.018	0.021	0.019	0.024	0.023	0.020	0.022
R2	0.018	0.031	0.028	0.025	0.024	0.027	0.021	0.027
R3	0.066	0.056	0.057	0.044	0.070	0.070	0.046	0.059
R4	0.190	0.430	7870.111	25.453	71.173	71.023	9.280	9.327
R5	1.396	1.490	59.373	35.420	12.033	12.135	7.491	7.534
R6	38.779	22.039	–	–	10073.280	9786.610	5706.956	5585.135
R7	0.132	0.388	–	–	–	–	–	–
R8	3.124	1.891	–	–	–	–	–	–
R9	50.817	25.323	–	–	–	–	–	–
R10	0.120	0.358	–	–	–	–	–	–
R11	0.415	0.941	–	–	–	–	–	–
R12	3.099	4.867	–	–	–	–	–	–

Note: results indicated with a dash did not solve in the imposed time limit of 10 hours

Table 12. Number of enumerated paths on raster networks

	LCor	2LCor	2NSP	B2NSP	GWN2NSP	GWA2NSP	BGWN2NSP	BGWA2NSP
	paths	paths	paths	paths	paths	paths	paths	paths
R1	n/a	n/a	2,213	260	814	814	145	145
R2	n/a	n/a	2,136	328	469	465	115	114
R3	n/a	n/a	11,983	2,816	6,397	5,314	1,754	1,549
R4	n/a	n/a	3,569,495,619	4,362,303	27,524,313	27,517,643	2,483,849	2,482,336
R5	n/a	n/a	37,788,197	9,418,048	6,694,418	6,694,081	1,792,104	1,792,051
R6	n/a	n/a	–	–	6,287,222,438	6,246,979,125	2,102,200,145	2,098,403,081
R7	n/a	n/a	–	–	–	–	–	–
R8	n/a	n/a	–	–	–	–	–	–
R9	n/a	n/a	–	–	–	–	–	–
R10	n/a	n/a	–	–	–	–	–	–
R11	n/a	n/a	–	–	–	–	–	–
R12	n/a	n/a	–	–	–	–	–	–

Note: results indicated with a dash did not solve in the imposed time limit of 10 hours

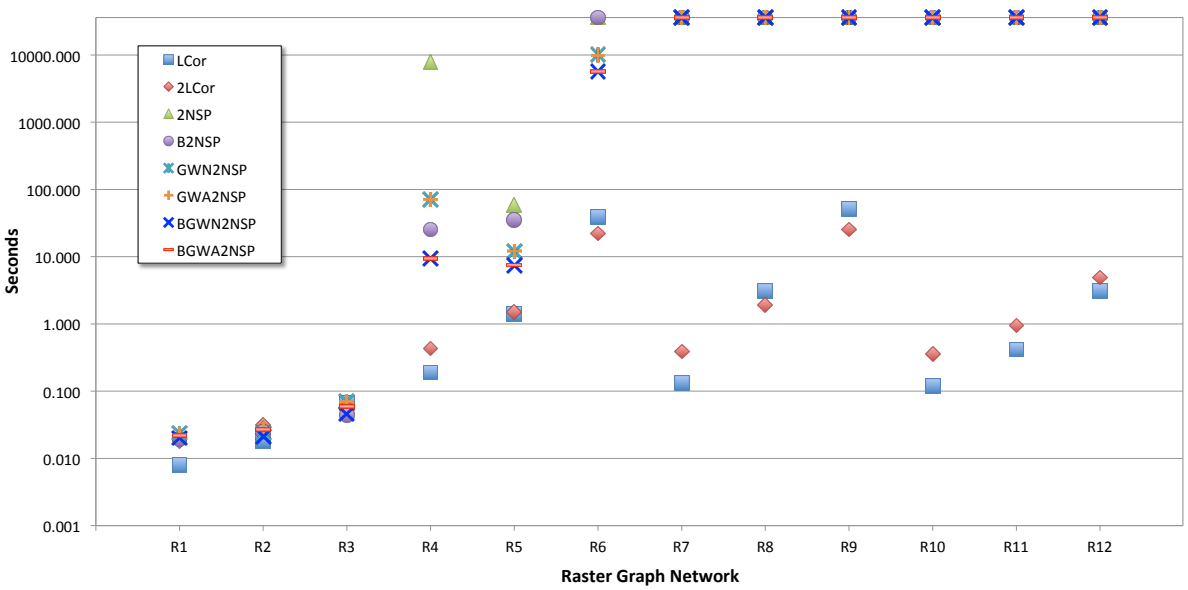


Figure 47. Algorithm runtimes on raster networks

Grid and raster networks were shown to be inherently challenging for enumerative approaches. The regular arc pattern and relatively small number of distinct values of arc costs means that as the threshold D increases linearly, the number of path combinations increase exponentially. This is exemplified by the massive increase in paths enumerated when comparing the 80x80 enumeration results to the 20x20 results, and looking at the intractability of the NSP methods on the 100x160 grids. The node labeling methods were not as encumbered as network sizes increased, since their runtimes depended more on simply the number of nodes in the network and proved to be much more efficient in these cases. Overall, for the terrain based raster networks 2LCor appears to be the best algorithm, with LCor being the second best.

ii. Random NetMaker Networks

Table 13 and Table 14 include the computational runtimes and the total number of paths enumerated for the labeling and enumeration algorithms on the NetMaker test networks. Figure 48 displays the runtime results in a graphical chart. Earlier we observed that the number of non-dominated solutions on NetMaker networks depend more on the generation parameters rather than the size of the network, and the number of paths enumerated did not seem to vary much by network size. In fact, all of the enumeration techniques generated very few paths to find the complete Pareto set, so the gateway and supported solution upper bounds did not yield much improvement to the overall computation time. The overhead required in the gateway approaches, where the number of shortest path computations is doubled in order to generate a gateway path set, always overshadowed the benefits of reduced enumeration effort. Similarly, for 2LCor, the overhead of computing the supported points first did not yield faster computation times over LCor on these networks.

Table 13. Algorithm runtimes on NetMaker networks

	LCor	2LCor	2NSP	B2NSP	GWN2NSP	GWA2NSP	BGWN2NSP	BGWA2NSP
	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)
NM1	0.281	0.418	0.149	0.139	0.230	0.251	0.231	0.252
NM2	0.248	0.277	0.050	0.050	0.149	0.150	0.154	0.176
NM3	0.243	0.359	0.129	0.130	0.217	0.233	0.222	0.232
NM4	0.256	0.335	0.100	0.100	0.202	0.226	0.213	0.225
NM5	0.559	0.614	0.184	0.183	0.294	0.347	0.300	0.401
NM6	0.538	0.735	0.268	0.284	0.494	0.618	0.510	0.559
NM7	0.525	0.612	0.139	0.148	0.304	0.340	0.313	0.349
NM8	0.489	0.513	0.068	0.073	0.378	0.422	0.388	0.421
NM9	0.496	0.634	0.222	0.222	0.430	0.547	0.442	0.557
NM10	1.411	1.607	0.550	0.542	1.140	1.374	1.170	1.374
NM11	1.049	1.615	0.737	0.724	1.514	1.676	1.516	1.648
NM12	1.138	1.286	0.268	0.271	0.827	0.892	0.843	0.900
NM13	0.863	1.146	0.488	0.488	1.066	1.150	1.087	1.159
NM14	0.952	1.343	0.563	0.537	1.247	1.360	1.252	1.369
NM15	2.309	2.445	0.660	0.646	1.761	1.937	1.795	1.943
NM16	1.446	2.446	1.254	1.275	2.649	2.826	2.725	2.868
NM17	1.598	2.113	0.865	0.875	1.904	2.091	1.986	2.146
NM18	1.274	1.992	1.069	1.028	2.148	2.228	2.158	2.240
NM19	1.487	1.952	0.939	0.911	2.216	2.414	2.253	2.434
NM20	4.045	4.692	1.575	1.577	4.093	4.519	4.064	4.561
NM21	2.221	2.548	0.828	0.819	1.912	2.083	1.934	2.104
NM22	5.730	7.159	2.953	3.061	8.066	8.853	8.207	8.808
NM23	6.545	8.915	3.569	3.548	12.105	12.900	12.372	12.944
NM24	20.173	27.049	15.825	16.025	32.980	34.841	33.816	34.956
NM25	9.952	20.043	11.344	11.042	27.094	28.302	26.951	28.257
NM26	32.009	45.246	22.671	22.136	52.077	55.031	52.063	55.087
NM27	24.385	66.291	41.079	41.257	125.815	129.540	126.357	129.552
NM28	68.948	197.415	154.987	153.856	352.190	357.076	351.135	357.642
NM29	47.199	107.098	56.539	56.603	186.031	190.220	186.075	190.510
NM30	109.945	343.840	228.322	228.316	548.658	557.424	549.109	556.172

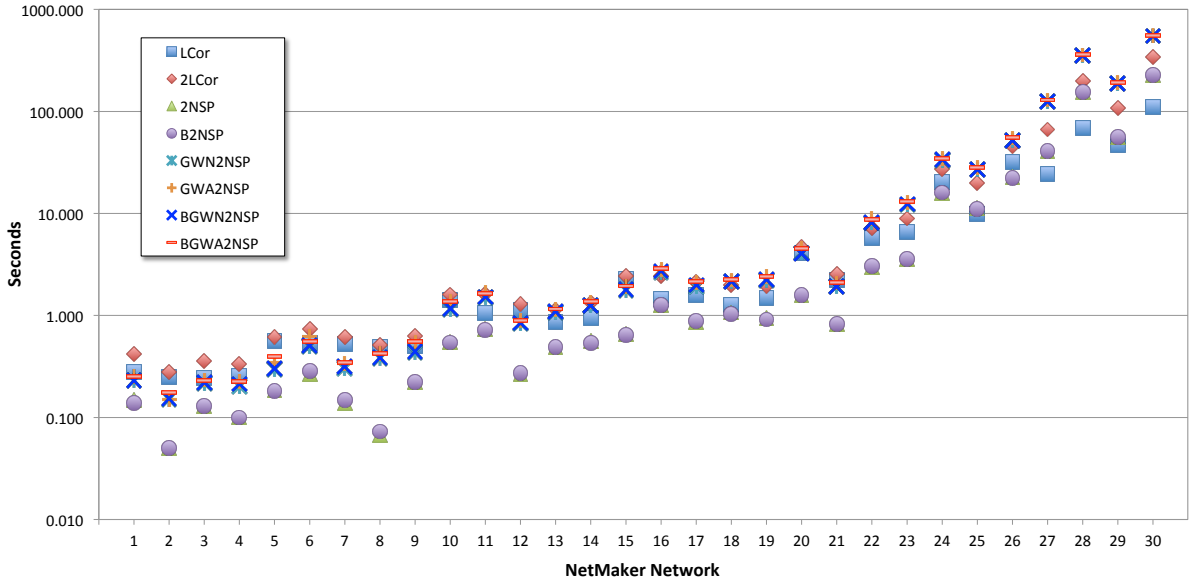


Figure 48. Algorithm runtimes on NetMaker networks

Table 14. Number of enumerated paths on NetMaker networks

	LCor	2LCor	2NSP	B2NSP	GWN2NSP	GWA2NSP	BGWN2NSP	BGWA2NSP
	paths	paths	paths	paths	paths	paths	paths	paths
NM1	n/a	n/a	287	53	187	187	44	44
NM2	n/a	n/a	68	7	46	45	7	7
NM3	n/a	n/a	192	45	151	151	39	39
NM4	n/a	n/a	275	54	158	157	46	46
NM5	n/a	n/a	1,174	423	520	491	250	247
NM6	n/a	n/a	592	98	381	324	72	63
NM7	n/a	n/a	140	68	64	47	37	23
NM8	n/a	n/a	119	16	88	88	15	15
NM9	n/a	n/a	214	48	184	184	45	45
NM10	n/a	n/a	545	64	461	410	50	44
NM11	n/a	n/a	615	138	380	343	79	76
NM12	n/a	n/a	182	52	124	123	33	32
NM13	n/a	n/a	165	41	124	124	28	28
NM14	n/a	n/a	345	78	145	145	47	47
NM15	n/a	n/a	857	314	212	177	110	90
NM16	n/a	n/a	443	68	313	272	57	53
NM17	n/a	n/a	396	67	178	169	48	48
NM18	n/a	n/a	348	77	194	189	39	39
NM19	n/a	n/a	232	38	179	179	34	34
NM20	n/a	n/a	509	86	265	234	59	54
NM21	n/a	n/a	764	201	283	283	92	92
NM22	n/a	n/a	1,591	106	958	958	88	88
NM23	n/a	n/a	823	85	400	334	60	54
NM24	n/a	n/a	1,738	377	729	619	245	216
NM25	n/a	n/a	1,476	271	649	617	125	123
NM26	n/a	n/a	1,186	287	368	358	122	117
NM27	n/a	n/a	457	82	301	277	64	62
NM28	n/a	n/a	1464	222	579	556	121	120
NM29	n/a	n/a	673	182	228	189	54	45
NM30	n/a	n/a	1334	315	313	294	116	105

On the NM1 to NM26 networks, the 2NSP and B2NSP algorithms performed equally well as the best approaches. For the largest four networks, LCor took over as the best algorithm. Since Netmaker networks do not emulate any sort of real-world network problem, we did not explore if this trend would continue for even larger networks, or for different parameters in generating the networks. If an interesting application with this network structure is found, then it may be beneficial to further explore a wider variety of parameters and their effects on computation times on the various methods.

iii. Road Networks

Table 15 and Table 16 include the computational runtimes and the total number of paths enumerated for the labeling and enumeration algorithms on the road test networks. Figure 49 displays the runtime results in a graphical chart. For the small DC road network, B2NSP was the fastest for 8 out of the 11 problems, while 2LCor was fastest for the other 3. All algorithms converged on all problems except for 2NSP, which timed out after 10 hours on one instance. For the medium RI road networks, B2NSP was the fastest for 9 out of the 11 problems, while 2LCor was fastest for the other 2. All enumeration techniques timed out on RI1; and 2NSP, GWN2NSP, and GWA2NSP timed out on RI2. For the largest NJ road networks, 2LCor was fastest for 5 problems, BGWN2NSP was fastest for 4 problems, and B2NSP was the fastest for 2 problems. On the problems where 2LCor was fastest, almost all of the enumeration techniques failed to converge in less than 10 hours. From this data we can observe that the NSP improvements, in particular the supported solution dominance bounds, provided significant gains in speed to the enumeration techniques, often resulting in the fastest computation times. But instances where the NSP methods could not solve a problem in a reasonable amount of time indicate the instability and exponential nature of enumeration for these types of road networks. On the other hand, labeling techniques were often not the fastest, but they always solved without fail in less than the cutoff time, with 2LCor always performing better than LCor. This dependability must be taken into account when selecting a best approach, so that even though B2NSP was fastest in 19 out of the 33 problems, we must recommend the 2LCor as the best method for road networks. If one can solve with two methods in parallel, then we recommend both 2LCor and B2NSP be used, since B2NSP performed best in most cases, but failed in others.

Table 15. Algorithm runtimes on road networks

	LCor	2LCor	2NSP	B2NSP	GWN2NSP	GWA2NSP	BGWN2NSP	BGWA2NSP
	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)
DC1	0.865	0.604	3.261	0.804	0.707	0.724	0.609	0.651
DC2	0.820	0.532	–	11.415	22.940	23.015	1.682	1.695
DC3	0.638	0.469	0.538	0.324	0.347	0.380	0.351	0.369
DC4	0.724	0.468	0.140	0.085	0.213	0.227	0.308	0.298
DC5	0.522	0.194	7.366	0.278	1.127	0.844	0.381	0.328
DC6	0.420	0.166	0.050	0.045	0.069	0.082	0.175	0.182
DC7	0.382	0.132	0.117	0.103	0.167	0.183	0.277	0.226
DC8	0.479	0.218	0.134	0.102	0.157	0.178	0.155	0.170
DC9	0.817	0.345	0.098	0.089	0.105	0.129	0.252	0.251
DC10	0.374	0.212	0.074	0.072	0.177	0.203	0.254	0.255
DC11	0.404	0.192	0.196	0.133	0.194	0.216	0.301	0.251
RI1	22.022	6.322	–	–	–	–	–	–
RI2	15.588	5.825	–	848.618	–	–	251.176	252.675
RI3	17.570	1.675	0.111	0.095	0.295	0.311	0.358	0.358
RI4	8.588	0.967	2.058	0.346	1.406	1.452	0.673	0.722
RI5	5.397	1.754	0.878	0.426	1.016	1.058	0.816	0.852
RI6	40.880	5.749	0.309	0.159	0.849	0.852	0.803	0.823
RI7	16.093	4.407	0.227	0.135	0.663	0.659	0.577	0.645
RI8	11.152	3.976	0.260	0.268	0.636	0.650	0.623	0.660
RI9	14.077	2.350	0.168	0.162	0.702	0.731	0.647	0.696
RI10	7.067	1.873	0.082	0.077	0.367	0.382	0.403	0.410
RI11	6.027	1.720	0.310	0.260	0.735	0.755	0.701	0.716
NJ1	922.217	214.033	–	–	–	–	–	–
NJ2	697.355	112.949	–	–	–	–	4503.596	4506.357
NJ3	301.700	107.596	–	–	–	–	–	–
NJ4	514.996	46.948	0.927	0.498	5.475	5.416	4.989	5.164
NJ5	356.477	127.828	100.460	59.99	29.775	29.256	19.264	20.121
NJ6	492.078	133.286	–	–	–	–	–	–
NJ7	579.252	47.396	–	45.849	2245.270	2252.835	24.421	25.142
NJ8	466.655	364.094	–	–	–	–	–	–
NJ9	1149.296	85.207	82.830	10.294	17.733	18.219	8.649	9.035
NJ10	527.648	171.456	139.124	60.982	99.417	99.255	44.161	45.296
NJ11	925.478	17.286	0.405	0.294	3.663	3.843	3.444	3.677

Note: results indicated with a dash did not solve in the imposed time limit of 10 hours

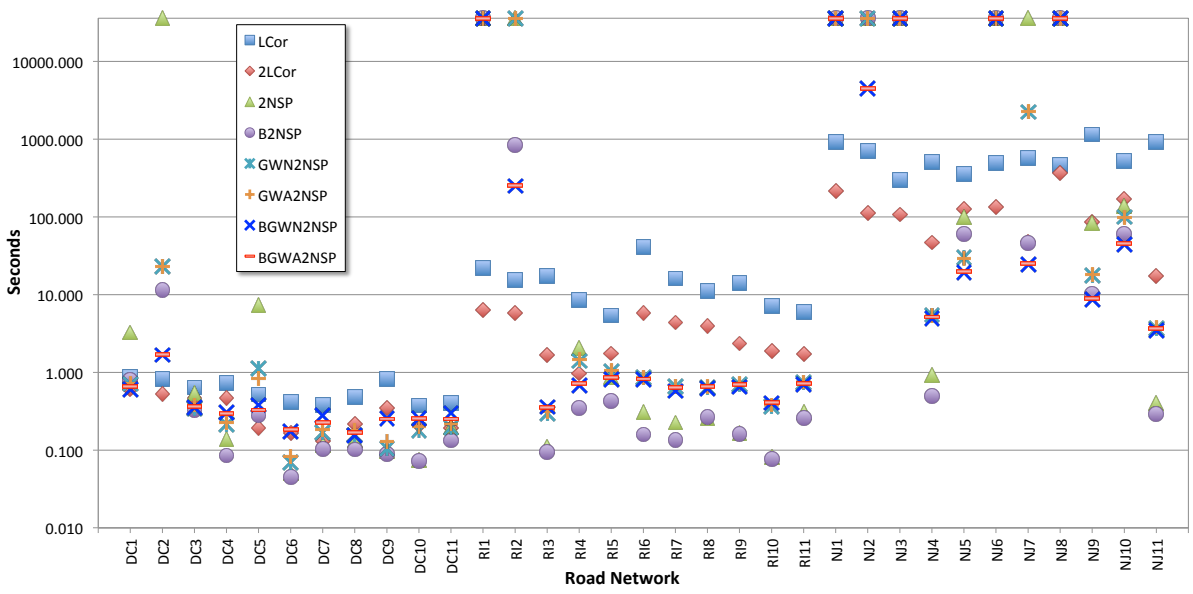


Figure 49. Algorithm runtimes on road networks

Table 16. Number of enumerated paths on road networks

	LCor	2LCor	2NSP	B2NSP	GWN2NSP	GWA2NSP	BGWN2NSP	BGWA2NSP
	paths	paths	paths	paths	paths	paths	paths	paths
DC1	n/a	n/a	1,121,078	103,548	332,083	332,067	39,253	39,249
DC2	n/a	n/a	–	596,264	8,532,377	8,532,377	89,152	89,152
DC3	n/a	n/a	72,904	2,237	18,886	18,885	1,333	1,332
DC4	n/a	n/a	16,454	137	11,493	11,492	134	134
DC5	n/a	n/a	2,805,278	736	978,939	978,939	578	578
DC6	n/a	n/a	35	2	31	31	2	2
DC7	n/a	n/a	124	8	124	124	8	8
DC8	n/a	n/a	6,336	2,978	1,239	1,239	753	753
DC9	n/a	n/a	3,054	21	2,068	2,068	21	21
DC10	n/a	n/a	234	54	126	126	25	25
DC11	n/a	n/a	10,104	425	7,139	7,139	325	325
RI1	n/a	n/a	–	–	–	–	–	–
RI2	n/a	n/a	–	15,449,205	–	–	13,742,893	13,742,893
RI3	n/a	n/a	14	0	14	14	0	0
RI4	n/a	n/a	94,443	42	92,098	92,098	29	29
RI5	n/a	n/a	126,657	5,498	106,511	106,511	5,393	5,393
RI6	n/a	n/a	5,604	162	1,601	1,601	112	112
RI7	n/a	n/a	1,089	354	991	991	322	322
RI8	n/a	n/a	32	3	32	32	3	3
RI9	n/a	n/a	248	23	162	162	22	22
RI10	n/a	n/a	22	4	21	21	4	4
RI11	n/a	n/a	28,690	2,383	16,335	10,723	2,158	2,055
NJ1	n/a	n/a	–	–	–	–	–	–
NJ2	n/a	n/a	–	–	–	–	149,462,202	149,462,202
NJ3	n/a	n/a	–	–	–	–	–	–
NJ4	n/a	n/a	391,296	524	290,031	290,031	520	520
NJ5	n/a	n/a	16,283,617	5,801,726	2,757,406	2,757,406	540,023	540,023
NJ6	n/a	n/a	–	–	–	–	–	–
NJ7	n/a	n/a	–	9,184,415	561,307,649	561,228,326	2,772,362	2,772,272
NJ8	n/a	n/a	–	–	–	–	–	–
NJ9	n/a	n/a	68,732,270	98,455	8,434,103	8,434,063	87,969	87,964
NJ10	n/a	n/a	15,834,882	1,935,798	9,162,035	9,162,023	854,286	854,284
NJ11	n/a	n/a	93,634	2,602	85,376	85,376	2,416	2,416

Note: results indicated with a dash did not solve in the imposed time limit of 10 hours

E. Concluding Remarks

In this chapter, we proposed improvements to the Raith and Ehrgott (2009) near shortest path enumeration algorithm (2NSP) for solving a biobjective shortest path problem. The improvements tightened the upper bound on the enumeration via 1) a gateway node/arc heuristic offering an initial candidate set of unsupported non-dominated solutions, 2) eliminating the computation of paths that would be dominated by the supported non-dominated solutions, and 3) a combination of both approaches. All of the enumeration approaches as well as two label correcting methods were tested on three types of network

problems: terrain-based raster grid networks, NetMaker randomly generated graph networks, and American statewide road networks. Raith and Ehrgott concluded that while 2NSP showed excellent promise in certain problem scenarios, that the exponential behavior of the enumeration made certain problems take far longer to solve with 2NSP than with labeling approaches. The additional bounds that we tested showed significant computational improvement over the original 2NSP algorithm and often resulted in being the fastest of all the methods tested, but they still displayed instances where the combinatorial nature of enumeration resulted in exceedingly large computation times. Overall then, we conclude that the labeling approaches are the most dependable algorithms for solving a biobjective shortest path problem in a reasonable amount of time.

Steiner and Radzik (2008) published some interesting work on enumeration approaches for a biobjective minimum spanning tree (MST) problem. In their case, they used a *k*-Best MST algorithm rather than a near-Best approach. The most interesting innovation of their paper was that they tried solving for more than one adjacent BUSS region at a time, since if one BUSS region is larger than another, then the algorithm may enumerate all solutions in the smaller region during the process of enumerating for the larger region. This approach for biobjective NSP enumeration may be of limited benefit since solution times for large BUSS regions take exponentially more time to solve than smaller ones, but it may be worth testing in future research. Perhaps this could breathe new life into the *k*-best enumerative biobjective shortest path approach of (Coutinho-Rodrigues *et al.* 1999).

A more promising approach for future work with biobjective NSP would be to parallelize the enumeration methods described in this chapter. Since each BUSS region is evaluated separately, they each could be analyzed in parallel via the approach described in

Chapter VII. Each individual NSP computation could also be parallelized via the method described in Chapter III. This bi-level parallelization could yield the most promising speedups, particularly if run on large-scale parallel computing clusters.

VII. Supported Multi-Objective: A Parallel Biobjective Shortest Path Algorithm

A. Introduction

Exponential growth in the capabilities of computerized data collection and analysis over the past few decades has resulted in the availability of massive data sets and networks for modeling and simulation. Traditional problems of public systems development such as corridor location for new transmission lines, pipelines, roadways and railways have always been considered a wicked optimization problem (Liebman 1976), and are now even more complicated given higher resolutions of satellite imagery for generating finer grained terrain network models. New frontiers in the analysis of large network data sets include the study of relationships between social media users (1.23 billion active Facebook users as of January 2014), and grouping by attributes within large online data repositories (Flickr contained over 8 billion photographs as of March 2013, a large portion of which are geotagged). These, and countless other recent data sources have served as the impetus for new terminology such as *big data* for working with data sets far too large to be processed by traditional database management tools, and the field of *analytics* for discovering meaningful results from these overwhelmingly large data sets.

As data sets increase in size, the computation required to do meaningful analysis on the data also increases. Moore's Law (Moore 1965) states that the number of transistors capable of being placed in an integrated circuit, and thus the computational power of a CPU, doubles every two years. This rule has held true since its inception in 1965 and until the early 2000's was mostly realized through faster processor clock speeds. In 2004, thermal limitations

prevented any further increase in processor clock speeds, creating a paradigm shift from faster clocks to multiple processor cores per CPU. Legacy programming code though cannot take advantage of multiple cores, and requires extensive rewrites to a parallel language in order to use the full capabilities of modern computers. This is not a simple task, as parallel computing introduces problems such as race conditions and deadlocks, which can result in non-deterministic behavior, infinite loops, or runtime failures. Proper implementation of low-level parallel libraries such as MPI, OpenMP, and UPC require advanced programming knowledge and sophisticated control of data transfer between processors. To address the difficulty of low-level schemes, higher-level libraries have emerged that simplify concurrent programming by hiding many of the low-level nuts and bolts. Examples include Cilk++ for C++, Grand Central Dispatch for Objective C, the Parallel Computing Toolbox for Matlab, and the concurrency libraries for Java. While these libraries do not eliminate all of the perils of concurrent programming, they do allow the programmer to focus more on big picture algorithm issues rather than the minute details of message passing schemes.

This work presents a general framework for using one such library, the Java fork/join library, for efficiently solving multi-objective network optimization problems in modern multi-core computers. Java is the only high-level language to offer a structured fork/join library optimized for divide-and-conquer algorithms, and is thus particularly suitable for the Non-Inferior Set Estimation (NISE) approach that is efficient at calculating the supported solutions of a multi-objective problem (Cohon *et al.* 1979). Section B introduces the problem and defines variables used in later pseudocode. Section C begins with a description of the serial Non-Inferior Set Estimation (NISE) algorithm (Cohon *et al.* 1979) for computing supported multi-criteria solutions, and then expands this to a proposed parallel

implementation, called pNISE. Section D presents a case study using pNISE for solving a biobjective shortest path problem on a large raster GIS network. Section E discusses some computational case study of this application to a biobjective shortest path problem. Section F presents some improvements to pNISE for instances with large number of processors. Finally, section G provides conclusions and enumerates other problems where this approach could be beneficial.

B. Background

Multiobjective optimization involves the task of determining noninferior solutions when considering multiple conflicting objectives, and is inherently more complicated than a problem's single-objective counterpart due to the added objective dimensionality. Most of past work in multi-objective modeling is first described for the use of two objectives, as this is usually the simplest case. Accommodating three or more objectives necessitates more complicated bookkeeping than what is required for two objectives, as some facets of the intersecting neighboring solutions in three or higher dimensions may lie in the interior rather than on the boundary of the convex polytope (Solanki 1986). Aside from this issue however, the fundamental theorems used to solve for tradeoffs in two objectives can be relatively easily expanded to three or more objectives. For this reason, most of the literature is concerned with the resolution of biobjective problems. This paper takes this same approach and restricts the discussion to biobjective problems as well. Further discussion of the nuances and approaches for problems with more than two objectives can be found in Przybylski *et al.* (2010).

In 1979, three different papers appeared in the published literature that addressed the problem of finding efficient solutions to biobjective optimization problems. (Dial 1979)

developed a process that involved finding up to a pre-specified number of supported points to a biobjective shortest path problem, Aneja and Nair (1979) developed an approach to find all supported points to a biobjective transportation problem, and Cohon *et al.* (1979) developed a process to find non-dominated solutions to biobjective linear programming problems. Overall, all three techniques are quite similar, but do differ in their main focus. For example, Dial's approach runs until it finds a certain number of solutions or finds the complete tradeoff curve. The choice of problems solved, and hence the resolved tradeoff curve is based upon a recursion formula taking problems in order. Aneja and Nair's approach is similar to that of Dial's except it does not stop until it has resolved all parts of the tradeoff curve. Cohon et al. (1979) show how lower and upper bounds on the tradeoff curve can be defined as supported points are added to the tradeoff curve. This allows one the opportunity to resolve at each iteration that portion of the curve with the greatest estimation error. This technique is called the Non-Inferior Solution Estimation (NISE) technique. The NISE technique will either generate all supported points on a tradeoff curve within a set estimation bound limit, or can be executed to completion to generate all supporting points as suggested by Aneja and Nair. In this paper, we adopt the NISE method of Cohon *et al.* as it can be considered the most general of the three techniques. NISE (also known by various other names) has become the standard method in the literature for solving the supported solutions of a multiobjective problem, due to its efficiency and applicability with a wide range of solver techniques (Current *et al.* 1990, Ehrgott and Wiecek 2005, Daskalakis *et al.* 2010, Clímaco and Pascoal 2012). It is used as part of the preferred approach in a wide range of multiobjective applications, including forestry and agriculture (Fischer and Church 2003, Pyke and Fischer 2005, Kasprzyk *et al.* 2009, Breschan and Heinimann 2013),

transmission and power flow systems (Salgado and Rangel Jr 2012, Soliman and Mantawy 2012, Medrano and Church 2014), industrial operations and logistics (Schilling 1982, Weber and Ellram 1993, Reklaitis 1996), and medical operations (Medaglia *et al.* 2009), just to name a few.

While NISE was originally presented within the context of biobjective linear programming problem, it can be applied to finding the supported solutions to biobjective Integer Programming (IP) or Mixed Integer Programming (MIP) problems as well. Modern first-rate MIP solvers have parallelism built-in to take advantage of multicore architectures; but specialized network optimization algorithms can often solve graph problems more efficiently than a general MIP solver. Tarapata (2007) published a comparison between solving a multiobjective shortest path problem on CPLEX vs. using a Dijkstra solver, and found that on large problems Dijkstra's computation times were 70 to 80 times faster than CPLEX.

IP problems can also have non-convex, non-inferior solutions known as *unsupported* solutions. Unsupported solutions are much more difficult to compute, as solving for those is equivalent to adding a knapsack constraint to the problem, which has been proven to be NP-hard (Garey and Johnson 1979). Some work has been published by Sanders and Mandow (2013) on a parallel biobjective shortest path algorithm for unsupported solutions, but this method introduces complicated and expensive data structures and introduce significant computational overhead in comparison to the fastest serial methods. A more recent method has been published that tries to reduce this overhead (Erb *et al.* 2014), although it is difficult to judge its effectiveness since the publication lacks any comparison with the fastest serial

methods. This chapter focuses on finding only the supported solutions of either an LP or IP problem in parallel.

The methods described in this paper are applicable to a variety of specialized network algorithms, including but not limited to biobjective variants of the minimum spanning tree problem, classical transportation problem, assignment problem, maximum flow problem, and the minimum cost flow problem. This work has chosen to apply the NISE approach though to a biobjective shortest path problem using a form of Dijkstra's shortest path algorithm (Dijkstra 1959) with a binary heap priority queue (Cherkassky *et al.* 1996) as the optimization solver. As a point of reference, we compared computation times of our Dijkstra solver implementation to the native Matlab version on a single-objective problem. The Matlab function is called `graphshortestpath()`, and also uses a binary heap priority queue. When solved on various problems on two different 1000x1000 raster network data sets, the Matlab runtimes were consistently at least 2.2x longer than our Java version.

The biobjective shortest path problem is defined as follows. Let G = (N, A) be a directed graph network with node set $N = \{u_1, u_2, \ldots, u_n\}$ and arc set $A = \{(u_1, v_1), \ldots, (u_m, v_m)\}$. Each arc $(u,v) \in A$ has associated with it two positive real costs $c_{uv} = (c^1_{uv}, c^2_{uv})$. The biobjective shortest path problem aims to solve for the minimum-cost paths from a source node $s \in N$ to a destination node $t \in N$ that minimizes two, often competing, objectives, z_1 and z_2. Each arc has associated with it a decision variable x_{uv} that is equal to 1 if it lies on the optimal shortest path, and 0 otherwise. This results in the following problem formulation:

$$
\begin{aligned}
&\min\ z_1(x) = \sum_{(u,v)\in A} c^1_{uv} x_{uv} \\
&\min\ z_2(x) = \sum_{(u,v)\in A} c^2_{uv} x_{uv} \\
&\text{s.t.} \sum_{(u,v)\in A} x_{vu} - \sum_{(v,u)\in A} x_{vu} = \begin{cases} 1 & \text{if } u = s \\ -1 & \text{if } u = t \\ 0 & \text{if } u \neq s,t \end{cases} \\
&\quad x_{uv} = \{0,1\} \text{ for all } (u,v) \in A
\end{aligned}
\qquad \textbf{(18)}
$$

While the above formulation contains two distinct objectives, supported solutions may be found by solving the weighted combined single-objective formulation, using the weight α, where $0 \leq \alpha \leq 1$.

$$
\min z_C(x) \ = \ \alpha \times z_1(x) \ + \ (1-\alpha) \times z_2(x) \qquad \textbf{(19)}
$$

Different supported solutions may be computed by varying the weight between the two objectives. Setting $\alpha = 1$ finds the optimal solution considering only the first objective, while setting $\alpha = 0$ finds the optimal solution with respect to the second objective, and setting α to something in between to find compromise solutions on the trade-off curve. While it is possible to find a number of supported solutions by iteratively stepping the weight value, the NISE method (described in the next section) specifies a procedure to find all distinct supported solutions with a minimum number of total solver iterations, or to solve for a set of supported points and stop when all points within an estimation bound have been defined.

Each solution to the combined objective of equation generates an *s*-*t* path that is a supported non-dominated solution, i.e. σ_i is an optimal solution for a given α. Additionally, let $x_{uv}(\sigma_i)$ be the value of the variable x_{uv} in the σ_i solution, where the value is 1 if arc (u, v) is on the shortest path, and 0 otherwise. For a given path solution σ_i, the $z_1(\sigma_i)$ is its objective

value with respect to the first objective, and $z_2(\sigma_i)$ is its objective value with respect to the second objective, as defined below. The term $z_C(\sigma_i, \alpha)$ represents the combined weighted objective according to the weight α.

$$z_1(\sigma_i) = \sum_{(u,v)\in A} c^1_{uv} x_{uv}(\sigma_i) \quad (20)$$

$$z_2(\sigma_i) = \sum_{(u,v)\in A} c^2_{uv} x_{uv}(\sigma_i) \quad (21)$$

$$z_C(\sigma_i, \alpha) = \alpha \times z_1(\sigma_i) + (1-\alpha) \times z_1(\sigma_i) \quad (22)$$

The set $\Psi = \{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ is the set of all supported non-dominated solutions to the problem, and form a convex Pareto frontier when plotted in objective space.

C. Supported Solution Search

1. Serial Non-Inferior Set Estimation (NISE)

The NISE method is used to find a set or subset of noninferior solutions of a biobjective linear, integer, or mixed-integer programming problem. Here, we describe this method to find all supported points of a trade-off curve. NISE begins by initially computing the single-objective solutions for each objective. In the biobjective case, these involve using weights $\alpha = 0$ and $\alpha = 1$. Once these solutions are determined, a weighting is chosen with equation 6 such that the z_c value for the two solutions are equal

$$\alpha = \frac{\left(z_2(\sigma_i) - z_2(\sigma_j)\right)}{\left(z_1(\sigma_i) - z_1(\sigma_j)\right) + \left(z_2(\sigma_i) - z_2(\sigma_j)\right)} \quad (23)$$

Figure 50 graphically depicts how the selection of α creates an objective line where the two initial solutions, σ_1 and σ_2, have equal combined objective values. With this weighting, the problem can be solved again to find a solution that minimizes this weighted combined objective, denoted by σ_3.

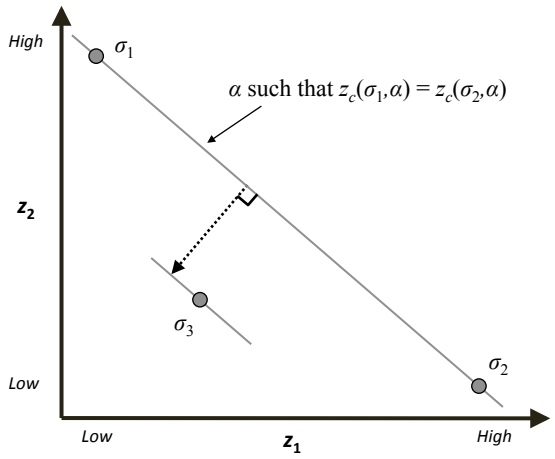


Figure 50. Objective space: σ_3 solves min $z_c(x)$ with weight α

After solving for σ_3, new weightings can be determined to find solutions that minimize the combined objective between the new adjacent supported points. Figure 51 shows a new objective line to find a solution σ_4 between σ_1 and σ_3, and another objective line for finding a solution σ_5 between σ_3 and σ_2. If a combined objective returns a solution that does not improve the combined objective from the previously found solutions, then there are no supported points that expand the convex hull between those respective solutions and the search in that region is terminated. This process continues until all adjacent points have not had any new solutions found between them, and thus all supported solutions have been found.

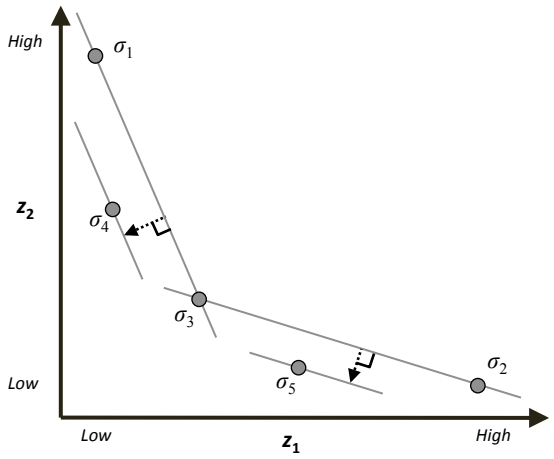


Figure 51. Objective space: supported solutions between σ_1 and σ_3, and between σ_3 and σ_2

Overall, NISE is a divide-and-conquer approach, and the general algorithm can be represented compactly with recursive function calls. The following pseudocode uses the NISE method for solving a biobjective shortest path problem using an optimal shortest path solver. The solver used in this work was Dijkstra's Algorithm with a binary heap priority queue (Cherkassky *et al.* 1996), although other specialized network algorithms could be used instead. In addition to the minimization problem presented, the code applies equally to a maximization problem by reversing the inequality in the dominance check.

Preliminary Algorithm: NISE for Biobjective Shortest Paths

```
// z_c(x) = α*z_1(x) + (1– α)*z_2(x)
// Ψ = {σ_1, σ_2,…, σ_p} = the set of supported non-dominated solutions
// Dij(α) solves a shortest s-t path with Dijkstra's algorithm using a
//    combined objective weighted by α
// SetA(σ_i, σ_j) selects next value of α based on the z_1 and z_2 values of
//    σ_i and σ_j
// RecursiveNISE(σ_i, σ_j) computes a supported solution between σ_i and σ_j

function: main
α = 1                          // minimize first objective
σ_i = Dij(α)
α = 0                          // minimize second objective
σ_j = Dij(α)
Ψ = σ_i
Ψ += RecursiveNISE(σ_i, σ_j)   // begin recursive NISE procedure

function: RecursiveNISE(σ_i, σ_j)
α = SetA(σ_i, σ_j)             // calculate alpha weighting, equation 6
σ_k = Dij(α)                   // solve composite objective
if (z_c(σ_k, α) < z_c(σ_i, α))      // if soln improves the composite
objective
Ψ += RecursiveNISE(σ_i, σ_k)
Ψ += RecursiveNISE(σ_k, σ_j)
else
Ψ += σ_j                       // else if no improvement found, return σ_j
end
return Ψ
```

The above algorithm though is a simplified version, and does not account for various anomalies that may occasionally arise. The next section lists these anomalies and how to deal with them, followed by a more comprehensive pseudocode that accounts for these scenarios.

2. NISE Anomalies

There are a few situations where one must take care in implementing the NISE method to avoid false-positive solutions or a non-terminating recursion causing a stack overflow exception. The following details these possible pitfalls, and how to avoid them.

i. Weakly Dominated Single Objective Solutions

The initial stage of the method requires solving the problem for each single objective. Oftentimes, there may exist numerous solutions that equally optimize that one objective. With regard to that objective, any of those solutions is optimal, yet they may perform quite differently from one-another when considering the other objectives in the model. In fact, in the initial single-objective base cases, an optimal solution may be returned that is weakly dominated by other equally optimal solutions. Such a solution is considered inferior, and should be omitted from the final non-dominated solution set.

For example, suppose one is minimizing $z_1(x)$ in the initial base case, as shown in Figure 52. The solver may return the solution σ_1, which is a minimum feasible solution to the problem with respect to objective 1. But there may exist another solution that was not found by the solver, σ_1', that weakly dominates σ_1, i.e. $z_1(\sigma_1) = z_1(\sigma_1')$ and $z_2(\sigma_1) > z_2(\sigma_1')$.

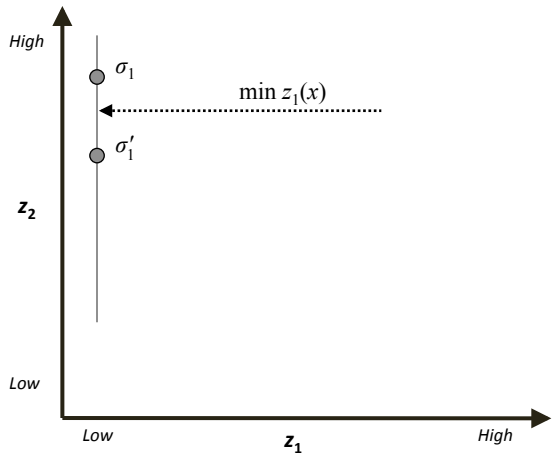


Figure 52. Weakly dominated solution that minimizes $z_1(x)$

Later in the algorithm, σ_1' will be found as the solution to a combined objective where α is very close to 1. A proper algorithm will put in place mechanisms to detect that it dominates σ_1 in order to eliminate it from the final solution.

ii. Multiple Equal Value Composite Solutions

Another anomaly arises when solving a composite objective function, i.e. $0 < \alpha < 1$, where there are numerous solutions with the same composite objective value. Figure 53 shows what this scenario would look like when plotting the solutions in objective space. In this case, for a given α, $z_C(\sigma_i,\alpha) = z_C(\sigma_j,\alpha) = z_C(\sigma_k,\alpha)$. If σ_i and σ_k were the points used to determine α, and the solution returned is σ_j, then there is no problem. σ_j is a non-dominated solution that is on the convex Pareto-frontier. While its presence does not change the shape of the convex region, i.e. it is not a corner point; it is an optimal trade-off solution that should be kept. The NISE solution approach does not guarantee finding all solutions that are not corner points, but some may be found by chance.

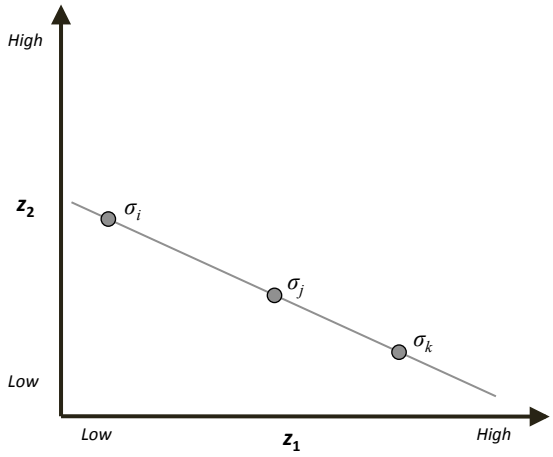


Figure 53. Multiple composite objective optimal solutions

The problem arises when σ_i and σ_j are the "outer points", i.e. `RecursiveNISE(`σ_i`, `σ_j`)`, and the solution returned is σ_k. If that point is kept, then the algorithm splits and runs `RecursiveNISE(`σ_i`, `σ_k`) and RecursiveNISE(`σ_k`, `σ_j`)`. If `RecursiveNISE(`σ_i`, `σ_k`)` returns σ_j, then there is a situation of an endless cycle alternating between those solutions. With a

recursive function, this will result in a stack overflow error, as the function will continue recursing ad infinitum until memory runs out.

In order to prevent this error and also to keep non-dominated solutions that are not corner points, rather than checking if an improvement is made to the combined objective z_c, a different criterion should be used to control if the function should recursively split. The alternative is to check if the returned solution is lexicographically in-between the outer points. If it is, then keep and split. Otherwise, the solution is lexicographically outside of the points, and the recursion ends and returns the appropriate solution. The next section revises the previous NISE pseudocode to take into account these two anomaly situations.

3. Complete NISE Pseudocode

Complete Algorithm: NISE for Biobjective Shortest Paths

```
// z_c(x) = α*z_1(x) + (1— α)*z_2(x)
// Ψ = {σ_1, σ_2,…, σ_p} = the set of supported non-dominated solutions
// Dij(α) solves a shortest s-t path with Dijkstra's algorithm using a
//    combined objective weighted by α
// SetA(σ_i, σ_j) selects next value of α based on the z_1 and z_2 values of
//    σ_i and σ_j
// RecursiveNISE(σ_i, σ_j) computes a supported solution between σ_i and σ_j

function: main
α = 1                          // minimize first objective
σ_i = Dij(α)
α = 0                          // minimize second objective
σ_j = Dij(α)
Ψ = σ_i
Ψ += RecursiveNISE(σ_i, σ_j)   // begin recursive NISE procedure
if (z_1(σ_1) == z_1(σ_2))      // if σ_2 dominates σ_1
Σ.removeFirstElement()
end

function: RecursiveNISE(σ_i, σ_j)
α = SetA(σ_i, σ_j)             // calculate alpha weighting, equation 6
σ_k = Dij(α)                   // solve composite objective
// if σ_k is lexicographically between σ_i and σ_j
if ((z_2(σ_k) < z_2(σ_i)) and (z_1(σ_k) < z_1(σ_j)))
if (z_2(σ_k) == z_2(σ_j))      // if σ_k weakly dominates σ_j
Ψ += RecursiveNISE(σ_i, σ_k)
else if (z_1(σ_k) == z_1(σ_i)) // if σ_k weakly dominates σ_i
Ψ += σ_k
Ψ += RecursiveNISE(σ_k, σ_j)
else                    // else σ_k is non-dominated
Ψ += RecursiveNISE(σ_i, σ_k)
Ψ += RecursiveNISE(σ_k, σ_j)
end
else
Ψ += σ_j                       // else if no improvement found, return σ_j
end
return Ψ
```

4. Java Fork/Join Framework

Java is a cross-platform object-oriented programming language that is ubiquitous in scientific computing, as well as in general desktop and mobile computing. It was originally released by Sun Microsystems in 1995, and is currently owned and actively developed by Oracle Corporation. One of the areas of Java language development since 2000 has been in

its concurrency libraries. In September 2004, Java 5 was released which for the first time included the java.util.concurrent application programming interface (API) that included various low-level tools for simultaneously processing numerous threads. Developers saw a further need for higher-level concurrency tools that were implicitly scalable over a wide variety of hardware configurations, and the fork/join framework was introduced by Doug Lea to address this need though the Java Community Process as a Java Specification Request, JSR 166 (Lea 2000, Lea 2003, Lea et al. 2004).

Fork/join is specifically designed to handle the difficult task of adding concurrency to recursive divide-and-conquer methods. Concurrent divide-and-conquer methods solve a problem by recursively splitting them into small subtasks, that are then solved in parallel, waiting for them to complete, and then composing results into a final answer. This approach is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g. quicksort, merge sort), multiplying large numbers, syntactic analysis (e.g. top-down parsers), convolution filters for digital image processing, and computing discrete Fourier transform (FFTs). The NISE algorithm described in this paper, used for determining the supported solutions to a biobjective optimization problem, also follows this general design paradigm.

The work breakdown of a divide-and-conquer algorithm tends to take a tree structure, where the task is split numerous times until a stopping criterion is reached, as shown graphically in Figure 54. For sorting or image processing, the stopping criteria may be dividing the problem into adequately small sub-problems; or in the case of NISE, the division stops for a specific region of the trade-off curve when no new supported solution is found in between two others. At this point, the results of the computation are sent back up the tree hierarchy, implicitly retaining the organized structure of the division, until all results

have reached the top level and the final result is complete. Fork/join task trees may be symmetrical, as is typically the case for most divide-and-conquer algorithms, but may also be asymmetrical, as is the case with NISE.

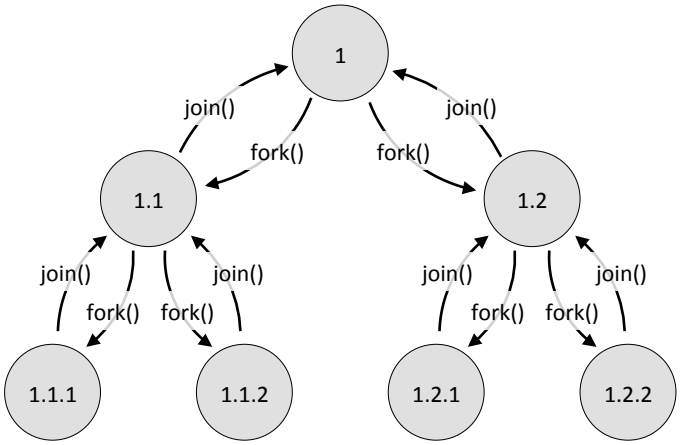


Figure 54. Fork/join task division

The Java implementation of fork/join uses a `ForkJoinPool` executor to manage the asynchronous concurrent execution of tasks. Tasks to be managed by the `ForkJoinPool` must implement the `ForkJoinTask` interface. `ForkJoinTask` objects feature two methods for performing their function: the `fork()` method launches a new task as a subtask of the one that called it, allowing it to be executed asynchronously; and, the `join()` method returns the results to the higher level task. A task cannot be joined until all of its sub-tasks have joined into it, ensuring that all computations are completed before going back up the hierarchy. The Java implementation of `ForkJoinPool` is capable of "work stealing", which actively steals and reallocates tasks when a processor is waiting for a sub-task to complete and there are other pending tasks remaining to be computed. This helps to ensure balanced workloads across processors, improving the overall parallel efficiency of the application.

D. Parallel NISE (pNise)

1. Parallel Divide and Conquer

The general usage of the fork/join design pattern takes the following form:

```
if (my portion of the work is small enough)
      do the work directly
else
      split my work into two pieces
      invoke the two pieces and wait for the results
end
```

For the purposes of NISE though, it is necessary to first run a solver iteration in order to determine whether to divide the problem once more. To accomplish this, the following modification to the design pattern is used:

```
optimize weighted composite objective
if (the problem is indivisible)
      return the result
else
      split problem into two sub-problems
      invoke the two sub-problems and wait for the results
      return list of results
end
```

Finer nuances are necessary for handling if a weakly dominated extreme point is detected, in which case then the program needs to create a single sub-problem without a split.

2. Parallel Single-Objective Extreme Points

In addition to the binary tree generated from the recursive task division, the initial base case of the NISE algorithm requires two independent runs (in the biobjective case) of a network optimization solver. These can also be set up to be run in parallel, and since this is a general iterative procedure (rather than recursive), the simplest way of doing so is with multithreading using Java's `Thread` object. In this case, the solver is initialized within two

independent threads, run simultaneously, and the `join()` method of `Thread` is used to wait until both threads have completed before proceeding with the remainder of the program.

If desired, one could avoid threads altogether, and continue using fork/join for the two base cases. While fork/join is intended for use on recursive functions, one can trick it for use on an iterative function by creating a wrapper class. Below is a pseudocode generalization of how this wrapper class is structured, called `SolverWrapper`. It takes two arguments: the first is a control boolean, and the second is the α value. One begins by calling `SolverWrapper(true,-1)`, where in this case the second argument is redundant and can take on any value. With an initial control argument of `true`, the program proceeds to split into two sub-problems, which are solved simultaneously using the fork/join functionality. Each sub-problem is given a control argument of `false` and α values of 0 and 1 respectively, corresponding to the two single-objective optimizations.

```
class SolverWrapper(boolean toggle, double a)
if (toggle == false)
      Solve(a)          // optimize with composite weight a
      return result
else
      SolverWrapper(false,0)
      SolverWrapper(false,1)
      return list of results
end
```

Experiments indicated that no significant parallel performance difference between using threads or a wrapper fork/join class for the initial base cases, possibly due to the fact that only two problems were being solved.

E. Computational Case Study

1. Test Networks

While applicable to numerous multi-criteria network problems, the motivation behind this work was to develop tools to better enable the generation of noninferior alternatives to a transmission line corridor location problem. Thus, the performance of the pNISE procedure was evaluated by running a biobjective shortest path analysis on a GIS-based raster data set assembled and used by the Eastern Interconnection States' Planning Council (EISPC). This data set is intended to facilitate the identification of potential energy sites and transmission line corridors within the EISPC region, which spans 39 eastern U.S. states, Washington D.C., and 8 Canadian provinces. The data was assembled jointly by Argonne National Laboratory, Oak Ridge National Laboratory, and the National Renewable Energy Laboratory as a part of their EISPC's Energy Zones Study (EZS) (Kuiper *et al.* 2013).

The EZS data contains numerous geographical information layers that would be used in a suitability analysis for locating new energy infrastructure, and is available through the EISPC Energy Zones Mapping Tool (EZMT, eispctools.anl.gov). The EZS includes 250 data layers, including such things as land cover type, slope, water bodies, watersheds, essential habitats, earthquake intensities, existing transmission lines, substations, rail and roadways, just to name a few. This work used a 1000x1000 raster subset of the EZS data, with a 250 square meter cell size. The region analyzed was in the Kentucky Lake region where the Tennessee River and the Cumberland River intersect the Ohio River; and includes portions of Tennessee, Kentucky, Illinois, and Missouri.

The case study involved the slope and land cover type layers for the two objectives, as these roughly correspond to the competing objectives of cost vs. environmental impact

respectively. Slope values were in percent slope, and land cover was already categorized according to the National Land Cover Database 2006 (Fry *et al.* 2011). These values and categories were converted to cell costs according to the terrain cost multipliers recommended by the Western Electricity Coordinating Council (Mason *et al.* 2012). Figure 55 displays graphics of the EISPC data maps used in the analysis, represented as 1000x1000 rasters and classified with high costs in dark colors and low costs in light colors. The left map represents the environmental impact objective, and the right map represents the construction cost objective.

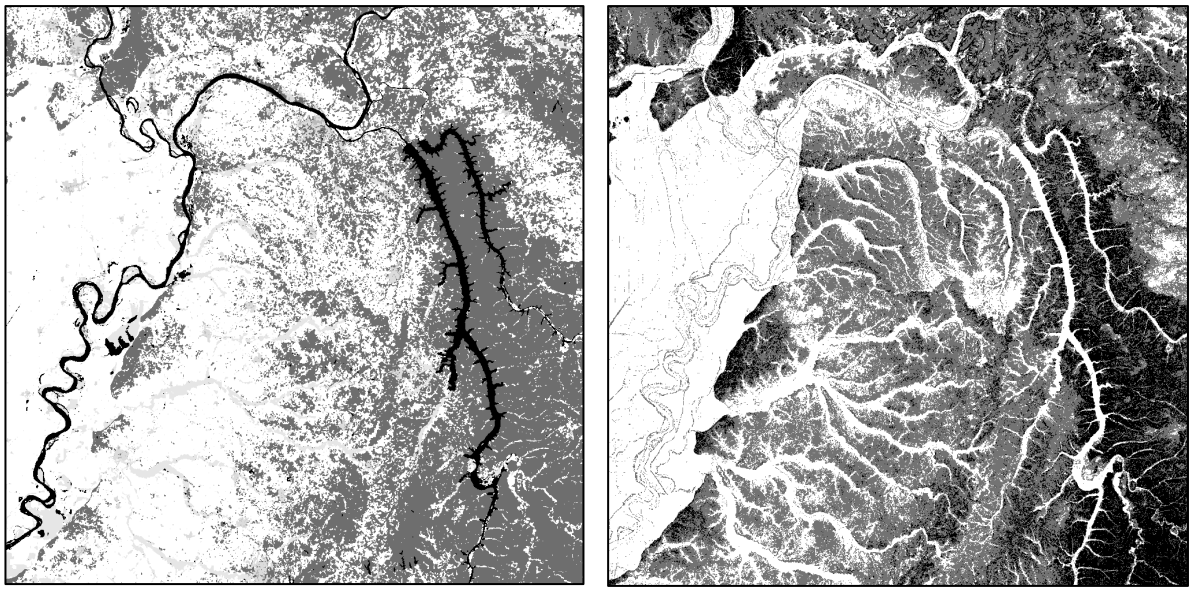


Figure 55. EISPC maps classified into two objectives: environmental impact (left), and construction cost (right)

From the raster layers, networks were created according to the guidelines of Huber and Church (1985), whereby nearby raster nodes were connected with arcs, and the arc cost labels for each objective assigned as a function of the node costs and the geometry of the arc itself. Three network versions were generated from the raster, with *r* radius values of 0, 1, and 2 respectively. The $r = 0$ network corresponds to an orthogonal grid, $r = 1$ adds diagonal "queen's moves", and $r = 2$ adds to that "rook's moves". Each higher value *r*-network

decreases the inherent geometric distortion of routes at the expense of adding more arcs and thus increasing computation time. Higher order networks are possible, but the increase in computational effort is not justified due to diminishing returns in spatial accuracy. According to Huber and Church, "the second order system ($r = 2$) appears to provide the most satisfactory trade-off between accuracy and computational burden."

Experiments were run on the 1000x1000 network on four origin/destination (OD) pairs: OD1 was from the SW corner to the NE corner, and OD2 was from the NW corner to the SE corner. The other OD pairs that were tested used starting and ending points closer to one another. Table 17 lists the networks used, and their properties including the coordinates of the OD nodes, *r*-value, number of nodes and arcs, and the number of supported noninferior solutions for that problem. Cells of the raster are referenced by the rows and columns, with the top-left corner cell being referenced as (0, 0). Row numbers increase as one heads south, and column numbers increase as one goes heading east.

Table 17. EISPC Test Networks Properties

OD Name	Origin Node	Destination Node	r	Total Nodes	Total Arcs	Supported Noninferior Solutions
1	(999, 0)	(0, 999)	0	1,000,000	3,996,000	86
			1	1,000,000	7,988,004	138
			2	1,000,000	15,964,020	266
2	(0, 0)	(999, 999)	0	1,000,000	3,996,000	89
			1	1,000,000	7,988,004	153
			2	1,000,000	15,964,020	274
3	(699, 300)	(300, 699)	0	1,000,000	3,996,000	32
			1	1,000,000	7,988,004	69
			2	1,000,000	15,964,020	114
4	(599, 400)	(400, 599)	0	1,000,000	3,996,000	23
			1	1,000,000	7,988,004	39
			2	1,000,000	15,964,020	69

2. Experimental Procedures

In order to test the efficacy of the pNISE approach, simulations were run on different hardware running Java version 7u51. One of the greatest strengths of fork/join and Java in

general is that it is cross-platform and automatically scalable, thus no modifications are necessary in order to run the code on different hardware. The first experiment compared the speedup of the pNISE versus an equivalent serial NISE implementation on a quad-core laptop running Apple OS X v10.9.2. The second experiment was a scaling experiment, evaluating the speedup and efficiency of pNISE based on the different numbers of allocated processors on a 32 core HP server running Red Hat Enterprise Linux Server release 6.2.

Metrics used to measure performenace included the speedup S_p and parallel efficiency E_p, as defined in section II.A.14. Letting p be the number of processors, and T_p be the execution time of a parallel algorithm on p processors, then T_1 is the execution time for the serial (1-processor) version of the algorithm, and in the ideal scenario, $S_p = p$ and $E_p = 1$, although this rarely occurs in parallel computation applications except for trivially simple cases such as Monte-Carlo simulation. In addition to high speedup values, one also looks for a linear trend as the number of processors increases. This would indicate that a method is scalable to a very high number of processors while maintaining a good speedup. As with perfect speedup, linear speedup trends are typically not possible to maintain except in the case for very simple problems, since speedup is limited by the amount of parallelism that exists in a problem instance or program (Amdahl 1967). In the case of pNISE, the two initial base cases must be completed before commencing the recursive portion of the algorithm. While the base cases can compute in parallel, the maximum speedup is only 2 for that portion of the calculation, since only two threads exist. Even after the recursive portion begins, the task division progresses as a binary tree (Figure 54), starting with a single level of parallelism, followed by two, then four, and so on. For smaller problems, this time with less parallelism can take a significant amount of the total computation time, so less speedup

will be expected. On the other hand, larger problems use a smaller proportion of their total computation time in these inefficient phases, and thus would have a higher expected speedup.

3. Computational Results

i. Serial NISE vs. pNISE

The first experiment tested the serial implementation of NISE to the parallel pNise. The hardware used was an Apple computer with a 3.7 GHz Intel Core i7-3820QM quad-core processor and 16GB of RAM. Results from this analysis are summarized in Table 18, comparing runtimes between a serial implementation of NISE using no concurrency, versus the fork/join pNISE approach. The results show that pNISE was able to maintain a high speedup in all cases, particularly for the largest problems (OD1 an OD2), which contain the most supported solutions. All OD1 and OD2 problems maintained speedup results between 3.32 and 3.56, with good mid-80% efficiencies. The smaller problems had a lower expected speedup due to a greater proportion of their total computation time being performed during the inefficient phases of the algorithm. This was evident with the OD3 problems having speedups of around 3.0 with mid-70% parallel efficiencies, and the smallest OD4 problems having 2.0-2.57 speedups and parallel efficiencies dropping to the 50%-65% range. In general, the larger the problem in terms of computation time and number of solutions, then the more efficient the parallelization.

Table 18. Serial NISE vs pNISE Runtimes and Speedup

OD	r	Supported Solns.	NISE T_1 (seconds)	pNISE T_4 (seconds)	S_4	E_4
1	0	86	117.490	34.394	3.416	0.854
	1	138	281.087	84.497	3.327	0.832
	2	266	950.571	267.178	3.558	0.889
2	0	89	117.319	35.245	3.329	0.832
	1	153	305.369	87.930	3.473	0.868
	2	274	981.298	281.162	3.490	0.873
3	0	32	29.748	10.019	2.969	0.742
	1	69	100.906	35.481	2.844	0.711
	2	114	285.074	95.004	3.001	0.750
4	0	23	10.484	4.075	2.573	0.643
	1	39	27.545	13.705	2.010	0.502
	2	69	87.031	36.333	2.395	0.599

ii. pNISE Scaling

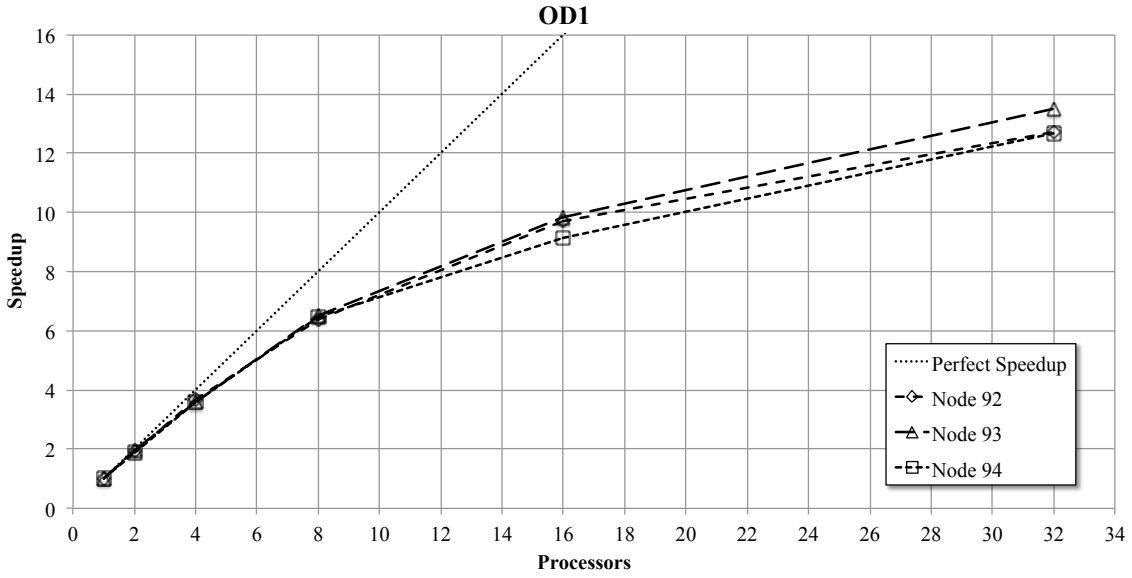
This experiment evaluated how the efficiency of the pNISE approach scales as the number of processors available is increased. `ForkJoinPool(p)` can take an optional input integer argument *p* that limits the executor service to that specified level of parallelism. All tests were run on a set of HP ProLiant DL580 Gen8 Server nodes, each with 512GB of RAM and four 2.0 GHz Intel Xeon X7550 8 core processors for a total of 32 cores per node. Results of this analysis are listed in Table 19. Only OD1 and OD2 were evaluated, as the other problems were too small to be able to fully make use of the large number of processors. Each of the cores on the nodes was approximately half as fast as a core on the Apple computer, but higher speedups were achieved since there were eight times as many cores per machine.

Table 19. pNISE Scaling on 32 core server nodes

OD1 node 92				OD1 node 93				OD1 node 94			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	1896.593	1.00	1.00	1	1866.592	1.00	1.00	1	1816.335	1.00	1.00
2	974.751	1.95	0.97	2	945.792	1.97	0.99	2	956.488	1.90	0.95
4	512.794	3.70	0.92	4	522.349	3.57	0.89	4	508.790	3.57	0.89
8	297.818	6.37	0.80	8	286.318	6.52	0.81	8	281.113	6.46	0.81
16	195.762	9.69	0.61	16	190.039	9.82	0.61	16	199.161	9.12	0.57
32	148.945	12.73	0.40	32	138.434	13.48	0.42	32	143.380	12.67	0.40

OD2 node 92				OD2 node 93				OD2 node 94			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	2007.119	1.00	1.00	1	1999.339	1.00	1.00	1	1949.368	1.00	1.00
2	1029.290	1.95	0.98	2	1021.425	1.96	0.98	2	1019.560	1.91	0.96
4	552.009	3.64	0.91	4	520.650	3.84	0.96	4	517.533	3.77	0.94
8	316.857	6.33	0.79	8	304.344	6.57	0.82	8	295.011	6.61	0.83
16	200.411	10.02	0.63	16	201.775	9.91	0.62	16	198.758	9.81	0.61
32	150.484	13.34	0.42	32	146.503	13.65	0.43	32	147.584	13.21	0.41

In general, speedups increased but the parallel efficiency decreased as the number of processors used was increased. The reasons for the drop in efficiency as more cores are used is due to a combination of the program running a longer time with less parallelism than processors, as well as the effects of increased overhead in coordinating the larger executor pool. Even so, the parallelization achieved speedups of up to 13.65, significantly reducing the overall computation time for these large problems. The speedup performance and deviation from the theoretically ideal efficiency are evident in Figure 56, which is a plot of speedup vs. processors used. It should be noted that a problem involving additional objectives or larger number of supported points will likely result in higher speedups due to their larger problem size. Overall, the results reported here demonstrate the value and ease of parallelizing the NISE algorithm.



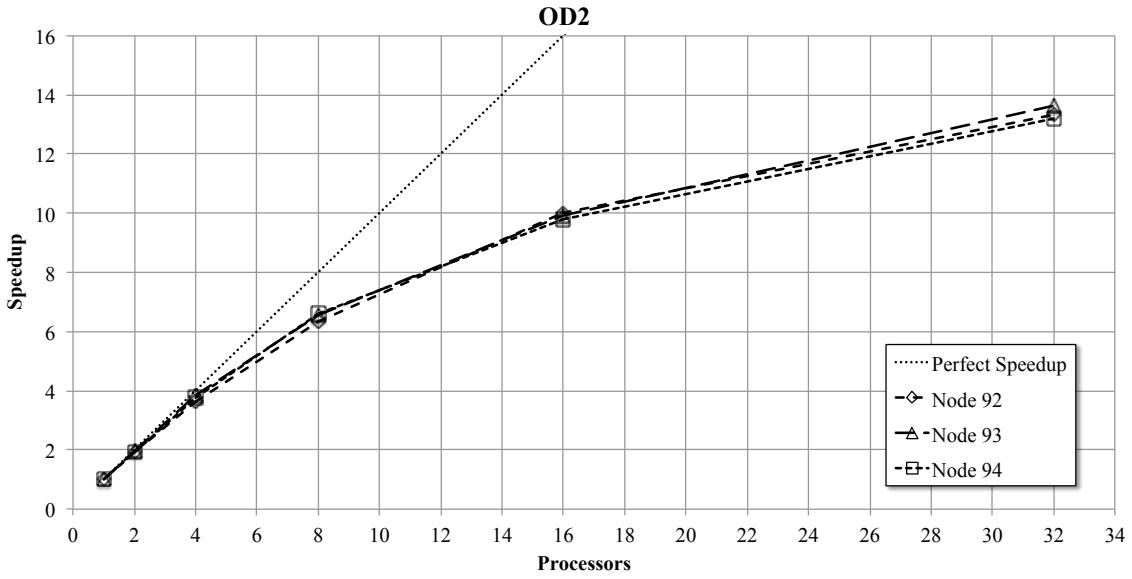


Figure 56. Scaling analysis of the speedup on 32 core nodes for OD1 (top) and OD2 (bottom)

F. Improvements to pNISE Efficiency

1. Parallelism Analysis of Basic pNISE

The previous section detailed a method for using Java's Fork/Join concurrency to parallelize the search for supported solutions to a biobjective shortest path problem. While effective at producing 13x speedups on a 32 core processor machine, there is still room for

improvement toward achieving a theoretically ideal 32x speedup. One issue is that the initial computation of the pNISE approach, where the weightings are 0 and 1, has only a level of parallelism of 2. When the solutions of these two are used to solve the next supported point, that has a parallelism of 1. Then next step then splits into two problems, with a parallelism of 2. Then 4, then 8, then 16, etc. After a few iterations, eventually there is more parallelism than there are processors and the method uses all resources efficiently from then on. But up until that point, some processors are waiting idly until enough parallelism exists to use all of the computational resources.

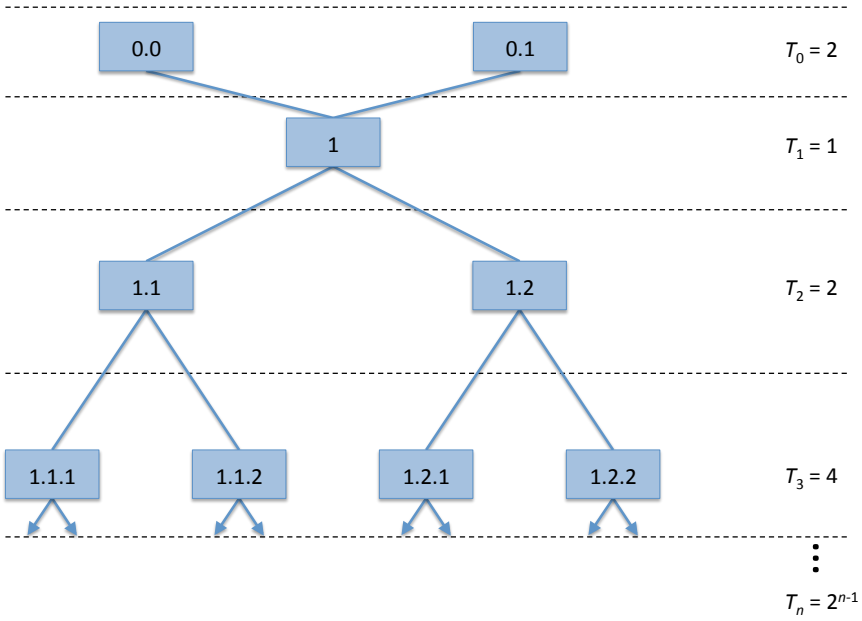


Figure 57. Parallel threads at each stage of pNISE

Figure 57 displays the parallelism at each stage in a graphical manner, showing the number of independent threads (i.e. the level of parallelism), T_i, at each stage i of the algorithm. pNise solves the weighted sum objective initially for 0 and 1 weights using two independent parallel threads. It then uses those solutions to determine the next weight, and solves a single problem. That solution is then used in conjunction with the two initial

solutions to determine two more weights, and solve those. This process continues in a binary tree manner until all branches fathom with no further supported solutions to be found. Assuming no early fathoming, each stage n after the initial has a level of parallelism of $T_n = 2^{n-1}$. This means that for a 32 core computer, six stages of the algorithm must be completed (including the initial zero stage) before there are enough threads in the thread pool to use all of the processors.

2. Improving Initial Parallelism

An improvement to the use of resources during the initial stages, rather than initially just solving the weighted objective for only values of 0 and 1, is to perform additional solver iterations on unused processors with weights in-between 0 and 1. This is done using simple threaded-parallelism, rather than a structured Fork/Join scheme. Essentially, given p processors, for each processor $i = 1 \ldots p$, execute the weighted sum objective for weight value of $z_i = (i\text{-}1) / (p\text{-}1)$ (i.e. equal intervals between 0 and 1). For a shortest path problem, these solutions take approximately the same amount of time to execute, and thus they all complete at approximately the same time. Once complete, a supported solution will have been found for each weighted graph, some of which may be repeated solutions, depending on the size of the problem and the number of processors. If there are no repeated solutions, then the p solutions can be used to initiate p-1 fork/join instances, which all get managed by the single Java thread pool in order to efficiently allocate work. From this point on, the procedure completes itself the same way as the original pNISE, in that a number of fork/join problems are solved in parallel, but the threads from all centrally handled by the Java threadpool.

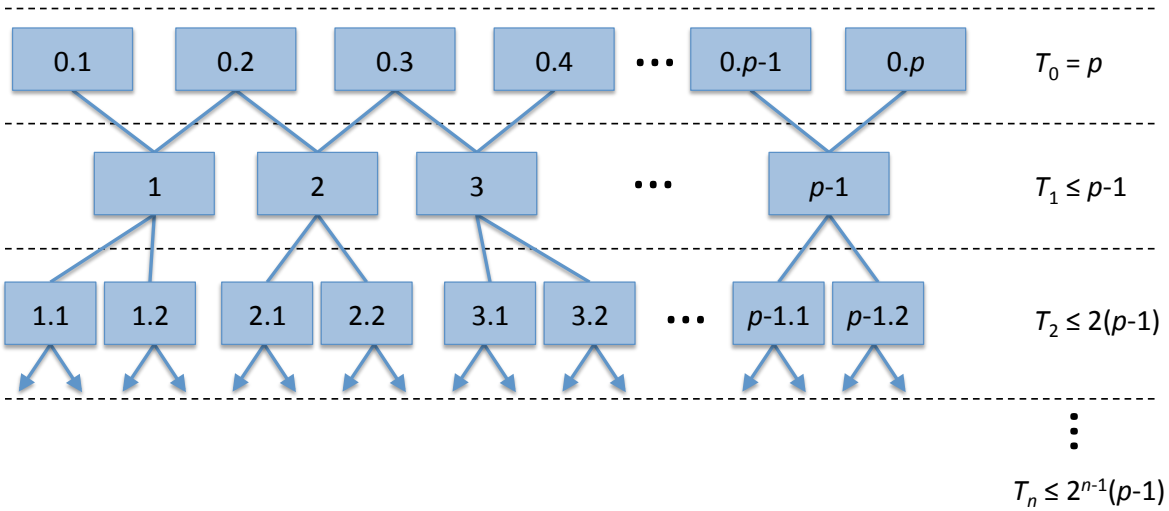


Figure 58. Parallel threads at each stage of the enhanced pNISE, making use of unused processors in the initial stages

Figure 58 displays this enhanced parallelism graphically. In the best-case scenario, the method uses all processors in the initial stage, all but one in the next stage, and all processors again for subsequent stages until computation is complete. This is in contrast with the original approach using 2 in the initial, then 1, then 2, then 4, then 8, etc. If there are repeated solutions in the initial stage, then there will be fewer than p-1 fork/join subproblems; but our experiments on the EISPC network never had fewer than 17 independent solutions in the initial stage, which means than in all cases all processors were always used by two stages later.

3. Computational Results

The same experiments were performed with the enhanced pNISE method, and compared with the simple pNISE approach. Table 20 shows the results of these computations for the two most-distant OD pairs: OD1 from the bottom-left corner to the top-right corner of the map, and OD2 from the top-left corner to the bottom-right corner on the map (see Figure 55). All computations were done on the number 92 node at UCSB's CSC for hardware

consistency. On the left side of the table are computations using the simple pNISE, and on the right are using the enhanced pNISE. As the number of processors increase, the overall speedup and efficiencies improve, in the case of the 32 core experiment going from a speedup of 12.73 to a speedup of 15.32 for OD1, and going from a speedup of 13.24 to a speedup of 15.31 for OD2.

Table 20. Comparison of simple pNISE and enhanced pNISE on 32 core server nodes

Simple pNISE – OD1 node 92				Enhanced pNISE – OD1 node 92			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	1896.593	1.00	1.00	1	1894.384	1.00	1.00
2	974.751	1.95	0.97	2	963.231	1.97	0.98
4	512.794	3.70	0.92	4	492.764	3.84	0.96
8	297.818	6.37	0.80	8	286.472	6.61	0.83
16	195.762	9.69	0.61	16	173.640	10.91	0.68
32	148.945	12.73	0.40	32	123.663	15.32	0.48

Simple pNISE – OD2 node 92				Simple pNISE – OD2 node 92			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	2007.119	1.00	1.00	1	2004.572	1.00	1.00
2	1029.290	1.95	0.98	2	1007.387	1.99	0.99
4	552.009	3.64	0.91	4	519.601	3.86	0.96
8	316.857	6.33	0.79	8	282.882	7.09	0.89
16	200.411	10.02	0.63	16	177.483	11.29	0.71
32	150.484	13.34	0.42	32	130.963	15.31	0.48

These improvements are also evident when comparing the speedups graphically as depicted in Figure 59, where the benefits of the enhanced pNISE become more pronounced as the number of processors increase.

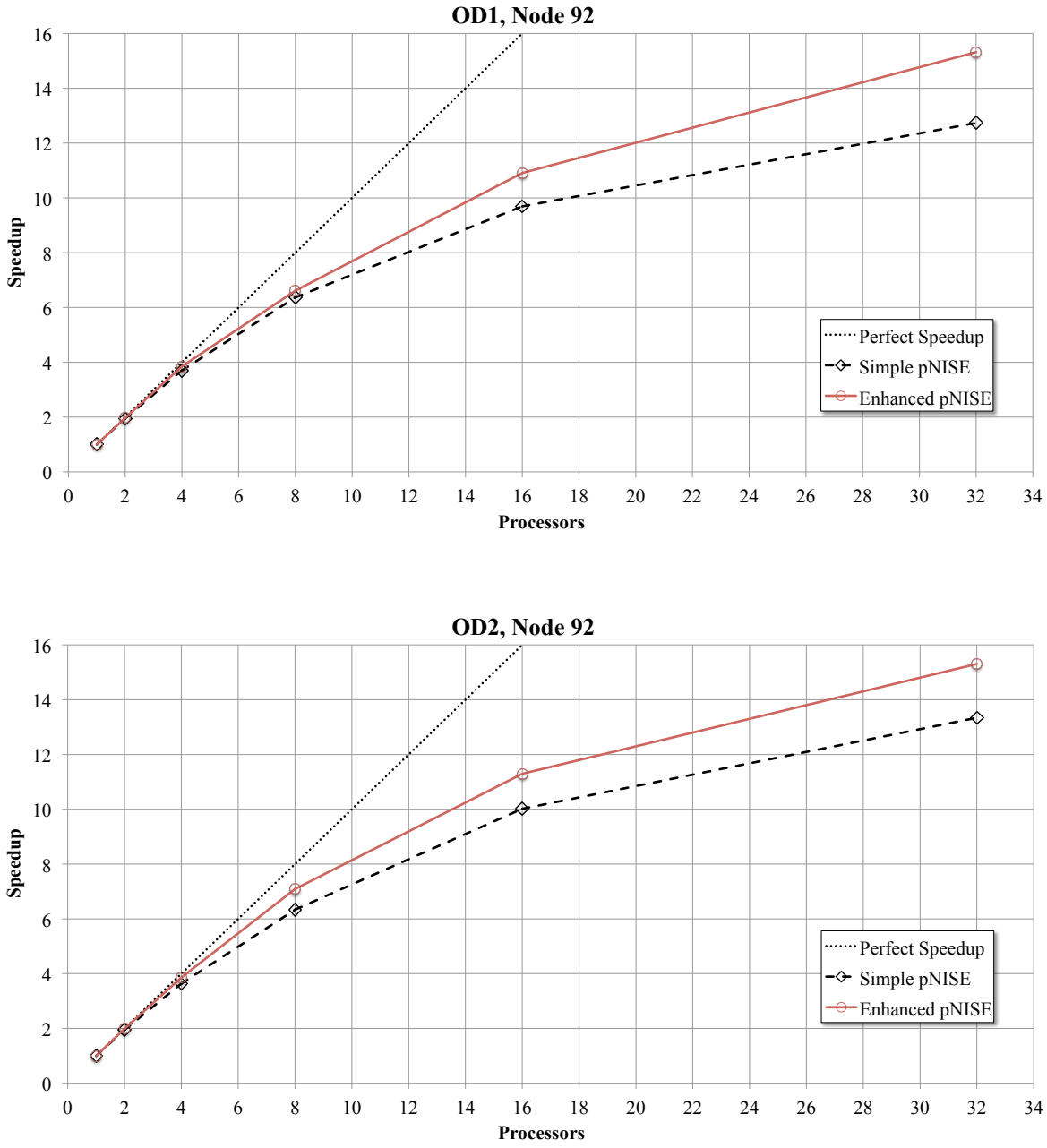


Figure 59. Speedup comparison between Simple pNISE and Enhanced pNISE for OD1 (top) and OD2 (bottom)

G. Concluding Remarks

This work developed a "simple-to-program" parallel implementation of the NISE method for computing the supported non-dominated solutions to biobjective network optimization problems, called pNISE. This method uses high-level fork/join framework within the Java 7 concurrency API to make this method parallel without the difficult complexities of traditional low-level message-passing parallel languages. After describing how to develop this algorithm, a transmission line corridor location case study using a large real-world data set demonstrated that the pNISE was effective at taking advantage of modern multi-core computer processors to significantly reduce the computation time of the biobjective supported solution set. Additional enhancements to the pNISE approach were then described, which further improved the parallel performance of the method.

The pNISE approach is applicable to all network problems where there exist fast, specialized optimal solution algorithms. In addition to the biobjective shortest path problem evaluated in this paper (Raith and Ehrgott 2009, Medrano and Church 2014), other problems that could be solved with pNISE include biobjective variants of the minimum spanning tree problem (Steiner and Radzik 2008), classical transportation problem (Aneja and Nair 1979), assignment problem (Przybylski *et al.* 2008), maximum flow problem (Royset and Wood 2007), and minimum-cost flow problem (Hamacher *et al.* 2007), just to name a few. With the continuing expansion of big data, scientists and engineers must tackle larger network problems than ever before, requiring novel tools to enable multicriteria analysis and optimization on these massive data sets using modern computing resources. Using simple general-purpose parallel tools such as pNISE to speed up computation allows a designer to

focus less time and energy on the generation of alternative solutions, and more time on model development and analysis to provide the best solutions to these challenging problems.

VIII. Conclusions

This dissertation addresses the problem of generating useful route alternatives for transmission line corridor location problems. The Electric Power Research Institute and the Georgia Transmission Corporation (EPRI-GTC) have developed the Overhead Electric Transmission Line Siting Methodology (Houston and Johnson 2006), but close scrutiny of its methodology showed that there were many shortcomings with that approach. The main goal of this dissertation was to take a fresh look at the process of corridor location, and develop a set of algorithms that could compute path alternatives using a foundation of solid geographical theory to offer designers better tools for developing alternatives that must be defended under scrutiny of a public forum. And just as importantly, as data sets become increasingly massive and present challenging computational elements, it is important that algorithms be developed to be efficient and able to take advantage of parallel computing resources.

Corridor location is a complicated problem that requires the consideration of numerous objectives while balancing the priorities of a variety of stakeholders. When preparing a GIS cost-surface for evaluating corridor path locations, various methodologies (including EPRI-GTC) use a composite single-objective a priori weighting in order to simplify the problem while still attempting to balance between the different objectives. We first examined developing routing tools for this type of composite data approach.

The most straightforward way to generate route alternatives is to enumerate all possible paths. Clearly the factorial nature of this would make such an endeavor intractable for all but the smallest networks, but perhaps enumerating a subset of near-optimal paths would yield good alternatives. For this, we implemented the Near Shortest Path (NSP) algorithm of

Carlyle and Wood (2005). While this depth-first-search algorithm is very efficient at computing all paths within some threshold of the shortest path, examples presented in this work demonstrated that this approach applied on terrain-based networks required an enormous number of paths to be computed before there was any spatial diversity in the path set. To aid this process, a parallelization of the NSP algorithm was developed that allows for faster computation on massive supercomputers, but even with the performance gains it was evident that more elegant approaches that take advantage of the spatial properties of the data would be more successful in a corridor location problem. Even so, this new parallel method could provide significant performance enhancements to other applications that use the NSP algorithm, including genetic sequencing (Waterman 1983), constrained shortest path problems (Carlyle *et al.* 2008), ship scheduling (Sigurd *et al.* 2005), and water reservoir operations (Liu *et al.* 2011).

Lombard and Church (1993) took another approach, generating alternatives by computing the set of optimal paths that are each constrained to go through a particular node in the network. This so-called Gateway Shortest Path set can be solved in polynomial time by computing two shortest path trees, and generates as many path alternatives as there are nodes in the network (although many of the paths may be duplicates). This approach was extended by automating the selection of a subset of these gateway paths that were both high performing and spatially diverse. This was accomplished by the novel use of Strahler Stream Order applied to the shortest path trees, a method used in hydrology for characterizing watershed networks. The Strahler order was used to define a hierarchical structure for the path trees that made it apparent where high quality path alternatives tend to congregate. The structures of the trees themselves implicitly force a spatial area difference between selected

paths, thus resulting in a very fast approach for generating sets of diverse but near optimal paths.

One could argue that using a composite cost-surface for corridor location limits the scope of the solution search, and that it is preferable instead to simultaneously consider all objectives to determine the set of Pareto-optimal trade-off solutions between the objectives. This is equivalent to solving a discrete multi-objective shortest path problem, which is considered to be weakly polynomial computationally when solving for the convex/supported set of non-inferior solutions, and NP-Hard for computing the non-convex/unsupported solutions. Given the difficulty of solving for the complete Pareto-optimal set, effort was directed towards developing an approximation heuristic by first solving for the supported solutions using the Non-Inferior Set Estimation (NISE) method originally developed by Cohon *et al.* (1979), and combining that with the Gateway Shortest Path methodology to generate an unsupported non-inferior candidate path set in weakly polynomial time. This approach showed excellent performance on our test networks, finding path sets that were nearly exact-optimal while computing in a fraction of the time when compared to exact algorithms.

In a slightly different direction, and because of the success of the gateway heuristic in estimating biobjective functions, the gateway approach was tested as an upper-bound for enumerative *exact* algorithms in solving for the unsupported solutions of a biobjective shortest path problem. The enumerative algorithm was developed by Raith and Ehrgott (2009), and uses NSP enumeration to compute the unsupported solutions after first using NISE to find the supported solution set. Their experiments showed the method preformed poorly on raster-like grid networks, and this new bounding technique significantly improved

the computing time performance of the enumerative method on GIS-based raster networks, although still not faster than labeling algorithm approaches for the biobjective shortest path problem. Experiments on random networks and road networks also showed major speed gains for the upper-bounded biobjective NSP method, often outperforming the labeling methods. Since the biobjective labeling algorithms performed best for the types of networks that would be used in a corridor location application, it is recommended that future work on this topic analyze possible ways to incorporate the gateway upper bound into labeling algorithms to determine if this could further speed up the labeling methods' speed performance.

The continual improvement of satellite and aerial imaging technologies have resulted in higher resolution data sets being available for corridor location design applications. While more accurate in terms of defining the relevant terrain nuances, solving complex routing problems on such large data sets is a challenge. We examined how we could solve a biobjective shortest path problem on a 1 million node, 16 million arc raster data set that is used in real transmission corridor location problems. We began by solving for the supported solutions using the NISE method, and found such problems were still tractable using conventional serial methods, but found opportunities for major speed improvements by converting them into parallel algorithms. Using the Java Fork/Join concurrency libraries (Lea 2000) which are optimized for divide-and-conquer methods, we were successful in creating a simple-to-implement parallel framework for the NISE method, called pNISE. This framework is highly scalable over a variety of platforms, showing significant speedup and efficiency on our experiments on both an Apple quad-core laptop and a Linux 32 core supercomputer. The pNISE framework is generalizable to other biobjective network

applications where there exist specialized algorithms for solving the single-objective variant, including the minimum spanning tree problem, classical transportation problem, assignment problem, maximum flow problem, and minimum-cost flow problem. Future work should also consider using pNISE in conjunction with the biobjective Gateway Shortest Path (GSP) heuristic developed in this dissertation, in order to approximate the complete Pareto path set on massive data problems. This approach, using advanced Fork/Join parallel computing techniques along with the efficiency GSP achieves by taking advantage of the spatial nature of the problem, would likely result in a powerful technique for the next generation of corridor location design tools.

IX. References

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.

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

Aissi, H., S. Chakhar & V. Mousseau, (2012). Gis-based multicriteria evaluation approach for corridor siting. *Environment and Planning B: Planning and Design,* 39, 287-307.

Akgün, V., E. Erkut & R. Batta, (2000). On finding dissimilar paths. *European Journal of Operational Research,* 121, 232-246.

Amdahl, G.M., (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS*, Atlantic City, N.J.: ACM, 483-485.

Aneja, Y.P. & K.P.K. Nair, (1979). Bicriteria transportation problem. *Management Science,* 25, 73-78.

Atkinson, D.M., P. Deadman, D. Dudycha & S. Traynor, (2005). Multi-criteria evaluation and least cost path analysis for an arctic all-weather road. *Applied Geography,* 25, 287-307.

Ayad, H.A., (1967). System evaluation by the simplified proportional assignment technique: Progress report.

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.

Bader, D., (2008). Petascale computing for large-scale graph problems. *Parallel Processing and Applied Mathematics,* 166-169.

Bagli, S., D. Geneletti & F. Orsi, (2011). Routeing of power lines through least-cost path analysis and multicriteria evaluation to minimise environmental impacts. *Environmental Impact Assessment Review,* 31, 234-239.

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

Blumofe, R.D. & C.E. Leiserson, (1994). Scheduling multithreaded computations by work stealing. *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*IEEE, 356-368.

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*.

Boruvka, O., (1926). On a minimal problem. *Prace MoraskÈ PridovedeckÈ Spolecnosti,* 3.

Breschan, J.R. & H.R. Heinimann, (2013). Ecoforest–automatic design of forest patterns attractive to wildlife in an artificial landscape. *Journal of Applied Operational Research,* 5, 125-134.

Brill, E.D., (1979). The use of optimization models in public-sector planning. *Management Science,* 25, 413-422.

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

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

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

Chhugani, J., N. Satish, C. Kim, J. Sewall & P. Dubey, (2012). Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*IEEE, 378-389.

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.

Clarke, K.C. & L.J. Gaydos, (1998). Loose-coupling a cellular automaton model and gis: Long-term urban growth prediction for san francisco and washington/baltimore. *International Journal of Geographical Information Science,* 12, 699-714.

Clímaco, J.C.N. & M.M.B. Pascoal, (2012). Multicriteria path and tree problems: Discussion on exact algorithms and applications. *International Transactions in Operational Research,* 19, 63-98.

Cohon, J.L., (1978). *Multiobjective programming and planning,* Academic Press.

Cohon, J.L., R.L. Church & D.P. Sheer, (1979). Generating multiobjective trade-offs: An algorithm for bicriterion problems. *Water Resources Research,* 15, 1001-1010.

Cong, G., S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat & T. Wen, (2008). Solving large, irregular graph problems using adaptive work-stealing. *Parallel Processing, 2008. ICPP'08. 37th International Conference on*IEEE, 536-545.

Cormen, T., (1990). *Introduction to algorithms,* 1st ed. The MIT press.

Coutinho-Rodrigues, J., J. Climaco & J. Current, (1999). An interactive bi-objective shortest path approach: Searching for unsupported nondominated solutions. *Computers & Operations Research,* 26, 789-798.

Crauser, A., K. Mehlhorn, U. Meyer & P. Sanders, (1998). A parallelization of dijkstra's shortest path algorithm. *Mathematical Foundations of Computer Science 1998,* 722.

Current, J.R., C.S. Revelle & J.L. Cohon, (1990). An interactive approach to identify the best compromise solution for two objective shortest path problems. *Computers & Operations Research,* 17, 187-198.

Danna, E., E. Rothberg & C. Le Pape, (2005). Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming,* 102, 71-90.

Dantzig, G.B., (1955). Discrete variable extremum problems. *Seventh National Meeting of the Society (ORMS)*, Los Angeles: Operations Research, 555-567.

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

Daskalakis, C., I. Diakonikolas & M. Yannakakis, (2010). How good is the chord algorithm? *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*Society for Industrial and Applied Mathematics, 978-991.

Delling, D., A.V. Goldberg, A. Nowatzyk & R.F. Werneck, (2013). Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing,* 73, 940-952.

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

Dial, R.B., (1979). A model and algorithm for multicriteria route-mode choice. *Transportation Research Part B: Methodological,* 13, 311-316.

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.

Ehlschlaeger, C.R., (1998). *The stochastic simulation approach: Tools for representing spatial application uncertanity*. University of California, Santa Barbara.

Ehrgott, M. & M.M. Wiecek, (2005). Mutiobjective programming. *Multiple criteria decision analysis: State of the art surveys.* Springer, 667-708.

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.

Erb, S., M. Kobitzsch & P. Sanders, (2014). Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. *In* Gudmundsson, J. & Katajainen, J. eds. *Experimental algorithms.* Springer International Publishing, 111-122.

Erkut, E., (1990). The discrete p-dispersion problem. *European Journal of Operational Research,* 46, 48-60.

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.

Fischer, D.T. & R.L. Church, (2003). Clustering and compactness in reserve site selection: An extension of the biodiversity management area selection model. *Forest Science,* 49, 555-565.

Floyd, R.W., (1962). Algorithm 97: Shortest path. *Communications of the ACM,* 5, 345.

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.

Fry, J.A., G. Xian, S. Jin, J.A. Dewitz, C.G. Homer, Y. Limin, C.A. Barnes, N.D. Herold & J.D. Wickham, (2011). Completion of the 2006 national land cover database for the conterminous united states. *Photogrammetric Engineering and Remote Sensing,* 77, 858-864.

Garey, M.R. & D.S. Johnson, (1979). *Computers and intractability,* Freeman San Francisco, CA.

Ghoseiri, K. & B. Nadjari, (2010). An ant colony optimization algorithm for the bi-objective shortest path problem. *Applied Soft Computing,* 10, 1237-1246.

Gleyzer, A., M. Denisyuk, A. Rimmer & Y. Salingar, (2004). A fast recursive gis algorithm for computing strahler stream order in braided and nonbraided networks. *Journal of the American Water Resources Association,* 40, 937-946.

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, (2001). Label correcting methods to solve multicriteria shortest path problems. *Journal of Optimization Theory and Applications,* 111, 589-613.

Guerriero, F., R. Musmanno, V. Lacagnina & A. Pecorella, (2001a). A class of label-correcting methods for the k shortest paths problem. *Operations Research,* 423-429.

Guerriero, F., R. Musmanno, V. Lacagnina & A. Pecorella, (2001b). 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.

Hägerstrand, T., (1965). A monte carlo approach to diffusion. *European Journal of Sociology,* 6, 43-67.

Hamacher, H.W., C.R. Pedersen & S. Ruzika, (2007). Multiple objective minimum cost flow problems: A review. *European Journal of Operational Research,* 176, 1404-1422.

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, (2007a). Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Trans. Algorithms,* 3, 45.

Hershberger, J., S. Suri & A. Bhosle, (2007b). On the difficulty of some shortest path problems. *ACM Transactions on Algorithms (TALG),* 3, 1-15.

Hewitt, M., G.L. Nemhauser & M.W. Savelsbergh, (2010). Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem. *INFORMS Journal on Computing,* 22, 314-325.

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.

Hong, I. & A.T. Murray, (2013). Efficient measurement of continuous space shortest distance around barriers. *International Journal of Geographical Information Science,* 1-17.

Horton, R.E., (1945). Erosional development of streams and their drainage basins; hydrophysical approach to quantitative morphology. *Geological Society of America Bulletin,* 56, 275.

Houston, G. & C. Johnson, (2006). Epri-gtc overhead electric transmission line siting methodology. *Technical Report,* 198.

Huang, B., P. Fery, L. Xue & Y. Wang, (2008). Seeking the pareto front for multiobjective spatial optimization problems. *International Journal of Geographical Information Science,* 22, 507-526.

Huarng, F., P. Pulat & L. Shih, (1996). A computational comparison of some bicriterion shortest path algorithms. *Journal of the Chinese Institute of Industrial Engineers,* 13, 121-125.

Huber, D.L., (1980). Alternative methods in corridor routing. *unpublished master's thesis,* 205.

Huber, D.L. & R.L. Church, (1985). Transmission corridor location modeling. *Journal of Transportation Engineering-Asce,* 111, 114-130.

Jacobitti, E., (1955). Automatic alternate routing in a 4a crossbar system. *Bell Laboratories Record,* 141-145.

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.

Johnson, P., D. Joy, D. Clarke & J. Jacobi, (1993). *Highway 3.1: An enhanced highway routing model: Program description, methodology, and revised users manual*.

Karp, R.M., (1972). Reducibility among combinatorial problems. *In* Miller, R.E. & Thatcher, J.W. eds. *Complexity of computer computations.* Springer.

Kasprzyk, J.R., P.M. Reed, B.R. Kirsch & G.W. Characklis, (2009). Managing population and drought risks using many-objective water portfolio planning under uncertainty. *Water Resources Research,* 45.

Kassakian, J.G. & R. Schmalensee, (2011). *The future of the electric grid: An interdisciplinary mit study*.

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

Kishore, T. & S. Singal, (2014). Optimal economic planning of power transmission lines: A review. *Renewable and Sustainable Energy Reviews,* 39, 949-974.

Kruskal, J.B., (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society,* 7, 48-50.

Kuby, M., Z.Y. Xu & X.D. Xie, (1997). A minimax method for finding the k best "differentiated" paths. *Geographical Analysis,* 29, 298-313.

Kuby, M.J., (1987). Programming models for facility dispersion: The p-dispersion and maxisum dispersion problems. *Geographical Analysis,* 19, 315-329.

Kuiper, J., D.P. Ames, D. Koehler, R. Lee & T. Quinby, (2013). *Web-based mapping applications for solar energy project planning.* Idaho National Laboratory, Preprint, INL/CON-13-28372.

Lanfear, K.J., (1990). A fast algorithm for automatically computing strahler stream order. *Water Resources Bulletin,* 26, 977-981.

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.

Lea, D., (2000). A java fork/join framework. *Proceedings of the ACM 2000 conference on Java Grande*ACM, 36-43.

Lea, D., 2003. *Concurrency jsr-166 interest site* [online]. http://gee.cs.oswego.edu/dl/concurrency-interest/index.html.

Lea, D., J. Bowbeer, D. Holmes, B. Goetz & T. Peierls, 2004. *Jsr 166: Concurrency utilities* [online]. Java Community Process. Available from: http://www.jcp.org/en/jsr/detail?id=166.

Lee, B. & C. Tomlin, (1997). Automate transportation corridor allocation. *GIS WORLD,* 10, 56-61.

Leng, J. & W. Zeng, (2009). An improved shortest path algorithm for computing one-to-one shortest paths on road networks. *icise,* 1979-1982.

Liebman, J.C., (1976). Some simple-minded observations on the role of optimization in public systems decision-making. *Interfaces,* 6, 102-108.

Liu, P., X. Cai & S. Guo, (2011). Deriving multiple near‐optimal solutions to deterministic reservoir operation problems. *Water Resources Research,* 47.

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

Luxen, D. & C. Vetter, (2011). Real-time routing with openstreetmap data. *ACM SIGSPATIAL GIS '11*, Chicago, IL, 513-516.

Mandow, L. & J.L. Pérez De La Cruz, (2005). A new approach to multiobjective a* search. *International Joint Conference on Artificial Intelligence*, 218-223.

Martins, E. & M. Pascoal, (2000). *An algorithm for ranking optimal paths.* Coimbra, Portugal.

Martins, E., M. Pascoal & J. Santos, (1999a). *An algorithm for ranking loopless paths.* Coimbra, Portugal.

Martins, E., M. Pascoal & J. Santos, (1999b). Deviation algorithms for ranking shortest paths. *International Journal of Foundations of Computer Science,* 10, 247-262.

Martins, E.D.V., (1984a). An algorithm for ranking paths that may contain cycles. *European Journal of Operational Research,* 18, 123-130.

Martins, E.Q.V., (1984b). On a multicriteria shortest path problem. *European Journal of Operational Research,* 16, 236-245.

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*.

Mason, T., T. Curry & D. Wilson, (2012). *Capital costs for transmission and substations.* Black & Veatch prepared for WECC, Proj. No. 176322.

Mcharg, I.L. & American Museum of Natural History., (1969). *Design with nature,* [1st ed. Published for the American Museum of Natural History [by] the Natural History Press.

Medaglia, A.L., J.G. Villegas & D.M. Rodríguez-Coca, (2009). Hybrid biobjective evolutionary algorithms for the design of a hospital waste management network. *Journal of Heuristics,* 15, 153-176.

Medrano, F.A. & R.L. Church, (2014). Corridor location for infrastructure development: A fast bi-objective shortest path method for approximating the pareto frontier. *International Regional Science Review,* 37, 129-148.

Menger, K., (1931). Some applications of point-set methods. *The Annals of Mathematics,* 32, 739-760.

Merrill, D., M. Garland & A. Grimshaw, (2012). Scalable gpu graph traversal. *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*ACM, 117-128.

Meyer, U. & P. Sanders, (2003). Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms,* 49, 114-152.

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.

Moore, G.E., 1965. Cramming more components onto integrated circuits. McGraw-Hill New York, NY, USA.

Newkirk, R.T., (1976). *A computer based planning system to optimize environmental resource allocations when locating utilities*. Ph.D. The University of Western Ontario (Canada).

Norden, J., (1625). *England, an intended guide for english travellers*.

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

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., (1961a). The kth best route through a network. *Operations Research,* 9, 578-580.

Pollack, M., (1961b). 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*. Thesis (MA)--University of Western Ontario.

Prim, R.C., (1957). Shortest connection networks and some generalizations. *Bell system technical journal,* 36, 1389-1401.

Przybylski, A., X. Gandibleux & M. Ehrgott, (2008). Two phase algorithms for the bi-objective assignment problem. *European Journal of Operational Research,* 185, 509-533.

Przybylski, A., X. Gandibleux & M. Ehrgott, (2010). A recursive algorithm for finding all nondominated extreme points in the outcome set of a multiobjective integer programme. *INFORMS Journal on Computing,* 22, 371-386.

Pyke, C.R. & D.T. Fischer, (2005). Selection of bioclimatically representative biological reserve systems under climate change. *Biological Conservation,* 121, 429-441.

Raith, A., (2010). Speed-up of labelling algorithms for biobjective shortest path problems. *Proceedings of the 45th annual conference of the ORSNZ. Auckland, New Zealand*, 313-322.

Raith, A. & M. Ehrgott, (2009). A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research,* 36, 1299-1331.

Reif, J.H., (1985). Depth-first search is inherently sequential. *Information Processing Letters,* 20, 229-234.

Reklaitis, G.V., (1996). Overview of scheduling and planning of batch process operations. *Batch processing systems engineering.* Springer, 660-705.

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.

Rouphail, N.M., S.R. Ranjithan, W. El Dessouki, T. Smith & E.D. Brill, (1995). A decision support system for dynamic pre-trip route planning. *Applications of Advanced Technologies in Transportation Engineering (1995)*ASCE, 325-329.

Royset, J.O. & R.K. Wood, (2007). Solving the bi-objective maximum-flow network-interdiction problem. *INFORMS Journal on Computing,* 19, 175-184.

Salgado, R. & E. Rangel Jr, (2012). Optimal power flow solutions through multi-objective programming. *Energy,* 42, 35-45.

Sanders, P. & L. Mandow, (2013). Parallel label-setting multi-objective shortest path search. *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*IEEE, 215-224.

Scaparra, M.P., R.L. Church & F.A. Medrano, (2013). Corridor location: The multi-gateway model. *WP, Department of Geography, University of California, Santa Barbara, CA*.

Scaparra, M.P., R.L. Church & F.A. Medrano, (2014). Corridor location: The multi-gateway shortest path model. *Journal of Geographical Systems* 16, 287-309.

Schilling, D.A., (1982). Strategic facility planning: The analysis of options. *Decision Sciences,* 13, 1-14.

Schrijver, A., (2005). On the history of combinatorial optimization (till 1960). *Handbooks in operations research and management science,* 12, 1-68.

Sedgewick, R. & J. Vitter, (1986). Shortest paths in euclidean graphs. *Algorithmica,* 1, 31-48.

Shier, D., (1979). On algorithms for finding the k shortest paths in a network. *Networks,* 9, 195-214.

Shreve, R.L., (1967). Infinite topologically random channel networks. *The Journal of Geology,* 178-186.

Sigurd, M.M., N.L. Ulstein, B. Nygreen & D.M. Ryan, (2005). Ship scheduling with recurring visits and visit separation requirements. *Column generation.* Springer, 225-245.

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.

Skriver, A.J.V. & K.A. Andersen, (2000). A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research,* 27, 507-524.

Smart, C.W., (1976). *A computer-assisted technique for planning minimum impact transmission right of way routes*. Virginia Polytechnic Institute and State University.

Solanki, R., (1991). Generating the noninferior set in mixed integer biobjective linear programs: An application to a location problem. *Computers & Operations Research,* 18, 1-15.

Solanki, R.S., (1986). *Techniques for approximating the noninferior set in linear multiobjective programming problems with several objectives*. Ph.D. Johns Hopkins University.

Solanki, R.S., P.A. Appino & J.L. Cohon, (1993). Approximating the noninferior set in multiobjective linear programming problems. *European Journal of Operational Research,* 68, 356-373.

Soliman, S.a.-H. & A.-a.H. Mantawy, (2012). Optimal power flow. *Modern optimization techniques with applications in electric power systems.* Springer, 281-346.

Steiner, S. & T. Radzik, (2008). Computing all efficient solutions of the biobjective minimum spanning tree problem. *Computers & Operations Research,* 35, 198-211.

Steuer, R.E. & E.-U. Choo, (1983). An interactive weighted tchebycheff procedure for multiple objective programming. *Mathematical programming,* 26, 326-344.

Strahler, A.N., (1952). Hypsometric (area-altitude) analysis of erosional topography. *Geological Society of America Bulletin,* 63, 1117.

Sutter, H., (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal,* 30, 202-210.

Tarapata, Z., (2007). Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *International Journal of Applied Mathematics and Computer Science,* 17, 269-287.

Tobler, W.R., (1970). A computer movie simulating urban growth in the detroit region. *Economic geography,* 46, 234-240.

Träff, J.L. & C.D. Zaroliagis, (2000). A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *Journal of Parallel and Distributed Computing,* 60, 1103-1124.

Trueblood, D.L., (1952). Effect of travel time and distance on freeway usage. *Highway Research Board Bulletin*.

Turner, A.K.F., (1968). *Computer-assisted procedures to generate and evaluate regional highway alternatives,* Purdue University.

Warshall, S., (1962). A theorem on boolean matrices. *Journal of the ACM (JACM),* 9, 11-12.

Waterman, M.S., (1983). Sequence alignments in the neighborhood of the optimum with general application to dynamic programming. *Proceedings of the National Academy of Sciences,* 80, 3123-3124.

Weber, C.A. & L.M. Ellram, (1993). Supplier selection using multi-objective programming: A decision support system approach. *International Journal of Physical Distribution & Logistics Management,* 23, 3-14.

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. & C. Noon, (2000). A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths. *Journal of Geographic Information and Decision Analysis,* 4, 1-11.

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