JOURNAL OF MULTI-CRITERIA DECISION ANALYSIS
J. Multi-Crit. Decis. Anal. **22**, 244–259 (2015)
Published online 7 April 2015 in Wiley Online Library
(wileyonlinelibrary.com) DOI: 10.1002/mcda.1541

A Parallel Computing Framework for Finding the Supported Solutions to a Biobjective Network Optimization Problem

FERNANDO ANTONIO MEDRANO* and RICHARD LEE CHURCH
Department of Geography, University of California, Santa Barbara, California, USA

ABSTRACT

Solving multi-objective combinatorial optimization problems can quickly become computationally challenging when applied to large networks generated from big data. Present-day first-rate Mixed Integer Programming (MIP) solvers have parallelism built-in to take advantage of multicore architectures; but specialized network optimization algorithms that can often solve graph problems more efficiently than a general MIP solver are typically programmed serially. Thus, these network algorithm implementations do not take full advantage of modern multicore computer capabilities. Many parallel computing languages include a fork/join framework into their concurrency packages that allow for simple conversion of recursive serial algorithms to operate in parallel without having to use complicated low-level message-passing interfaces. Fork/join is particularly well suited to divide and conquer algorithms such as Non-Inferior Set Estimation (NISE) method for computing the supported Pareto front of a multi-objective optimization problem. This work develops a simple general parallel NISE (pNISE) implementation using the Java 7 concurrency tools, and then offers results from a specific implementation of solving a bi-objective shortest path problem. The results indicate that this method is capable of sizeable speed-ups on both small-scale personal computers as well as large-scale shared memory supercomputers. Finally, other network problems suitable to this method are discussed, including multi-objective variants of the minimum spanning tree problem, transportation problem, assignment problem, maximum flow problem, and the minimum-cost flow problem.

KEY WORDS: multicriteria analysis; biobjective shortest paths; parallel computing; GIS; network algorithms

1. 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 modelling and simulation. Traditional problems of public systems development such as corridor location for new transmission lines, pipelines, roadways and railways have always been considered wicked optimization problems (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 other types 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 central processing unit (CPU), doubles every two years. This rule has held true since its inception in 1965, and until the early 2000s 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.

*Correspondence to: Department of Geography, UC Santa Barbara, 1832 Ellison Hall, Santa Barbara, California 93106-4060, USA. E-mail: medrano@geog.ucsb.edu

Received 01 April 2014
Accepted 17 February 2015

Legacy programming codes though cannot take advantage of multiple cores, and require 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 behaviour, infinite loops or runtime failures. Proper implementation of low-level parallel libraries such as Message Passing Interface, Open Multi-Processing (OpenMP) and Unified Parallel C 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 between processor cores.

This work presents a general framework for using one such model, the Java fork/join library, for efficiently solving multi-objective network optimization problems in modern multicore computers. Fork/join is particularly suitable for use in parallelizing the Non-Inferior Set Estimation (NISE) approach that is efficient at calculating the supported solutions of a multi-objective problem (Cohon *et al*., 1979). Java is not the only high-level language to offer a structured fork/join library optimized for divide-and-conquer algorithms, and in fact fork/join can also be found in such languages as OpenMP, Task Parallel Library for .NET, and both Cilk Plus and Threading Building Blocks for C++. We chose Java for the fact that its fork/join concurrency libraries are well developed and highly optimized, but by no means do we suggest that Java is the only way to implement this framework. In fact, we encourage the application of the methods in this paper to be used with other parallel languages that contain fork/join functionality. Section 2 introduces the problem and defines variables used in later pseudocode. Section 3 begins with a description of the serial NISE algorithm, and then expands this to a proposed parallel implementation, called parallel Non-Inferior Set Estimation (pNISE). Section 4 presents a case study using pNISE for solving a biobjective shortest path problem on a large raster geographic information system (GIS) network. Finally, Section 5 discusses the results of this application, and enumerates other problems where this approach could be beneficial.

2. BACKGROUND

Multiobjective optimization involves the task of determining non-inferior 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 the past work in multi-objective modelling 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 neighbouring 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 trade-off curve. The choice of problems solved, and hence the resolved trade-off 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 trade-off curve. Cohon *et al.* (1979) show how lower and upper bounds on the trade-off curve can be defined as supported points are added to the trade-off curve. This allows one the opportunity to resolve at each iteration that portion of the curve with the greatest

DOI: 10.1002/mcda

estimation error. This technique is called the Non-Inferior Solution Estimation (NISE) technique. The NISE technique will either generate all supported points on a trade-off 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, because of 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 (LP) 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 versus using a Dijkstra solver, and found that on large problems, Dijkstra's computation times were 70 to 80 times faster than CPLEX.

Integer Programming 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 Non-deterministic Polynomial-time hard (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 implements complicated and expensive data structures that 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 because the publication lacks any comparison with the fastest serial methods. This paper 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 parallel NISE approach 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 Java 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 1000×1000 raster network data sets, the MATLAB runtimes were consistently at least 2.2× 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 $c_{uv} = \left(c^1_{uv}, c^2_{uv}\right)$. 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:

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

While the aforementioned formulation contains two distinct objectives, supported solutions may be found by solving the weighted composite single-

DOI: 10.1002/mcda

objective formulation, using the weight α, where $0 \leq \alpha \leq 1$.

$$minz_C(x) \quad = \quad \alpha \times z_1(x) + (1-\alpha) \times z_2(x) \qquad (2)$$

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 in-between values 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 composite objective of Equation (2) generates an *s-t* path that is a supported non-dominated solution, that is, 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 function $z_1(\sigma_i)$ 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 in the following equations. The term $z_C(\sigma_i, \alpha)$ represents the composite weighted objective according to the weight α.

$$z_1(\sigma_i) = \sum_{(u,v) \in A} c^1_{uv} x_{uv}(\sigma_i) \qquad (3)$$

$$z_2(\sigma_i) = \sum_{(u,v) \in A} c^2_{uv} x_{uv}(\sigma_i) \qquad (4)$$

$$z_C(\sigma_i, \alpha) = \alpha \times z_1(\sigma_i) + (1-\alpha) \times z_1(\sigma_i) \qquad (5)$$

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.

3. SUPPORTED SOLUTION SEARCH

3.1. Serial non-inferior set estimation

The NISE method is used to find a set or subset of non-inferior solutions of a biobjective linear, integer or MIP 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.

$$\alpha = \frac{\left(z_2(\sigma_i) - z_2(\sigma_j)\right)}{\left(z_1(\sigma_i) - z_1(\sigma_j)\right) + \left(z_2(\sigma_i) - z_2(\sigma_j)\right)} \qquad (6)$$

Figure 1 graphically depicts how the selection of α creates an objective line where the two initial solutions, σ_1 and σ_2, have equal composite objective values. With this weighting, the problem can be solved again to find a solution that minimizes this weighted composite objective, denoted by σ_3.

After solving for σ_3, new weightings can be determined to find solutions that minimize the composite objective between the new adjacent supported points. Figure 2 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 solution is found that does not improve the composite 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 fathomed (terminated). This process continues until all adjacent points have fathomed, and thus all supported solutions have been found.

Overall, NISE can be considered a divide-and-conquer approach, and the general algorithm can be represented compactly with recursive function calls.

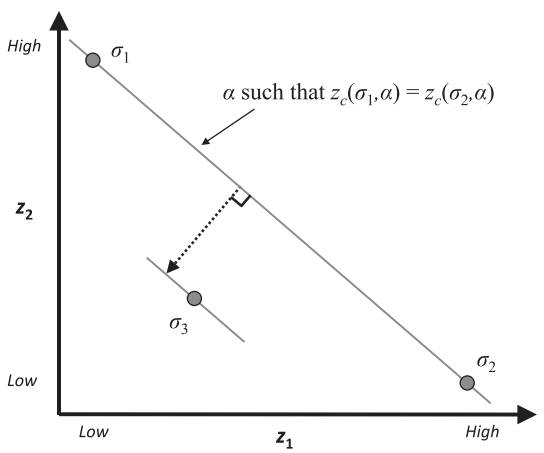


Figure 1. Objective space: σ_3 solves min $z_c(x)$ with weight α.

DOI: 10.1002/mcda

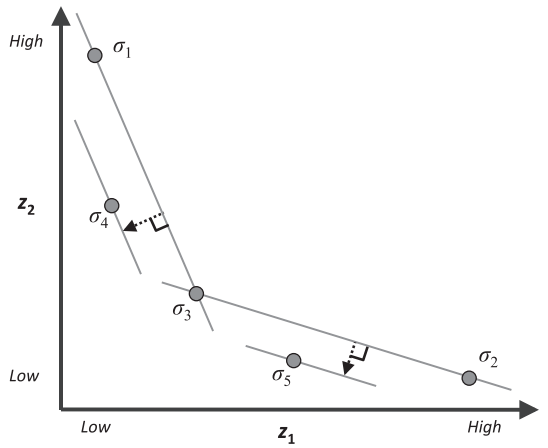


Figure 2. Objective space: supported solutions between σ_1 and σ_3, and between σ_3 and σ_2.

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
// zc(x) = α*z1(x) + (1− α)*z2(x)
// Ψ = {σ1, σ2,…, σp} = the set of supported non-dominated solutions
// Dij(α) solves a shortest s-t path with Dijkstra's algorithm using a
//composite objective weighted by α
// SetA(σi, σj) selects next value of α based on the z1 and z2 values of
//σi and σj
// RecursiveNISE(σi, σj) computes a supported solution between σi and σj

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

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

The previous algorithm though is a simplified version, and does not account for particular 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.

3.2. Non-inferior set estimation anomalies

One must take care in implementing the NISE method to avoid situations that may result in false-positive solutions or a non-terminating recursion causing a stack overflow exception. The following details these possible pitfalls, and how to avoid them.

3.2.1. 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 3. The solver may return the solution σ_1, which is a minimum feasible solution to the problem with

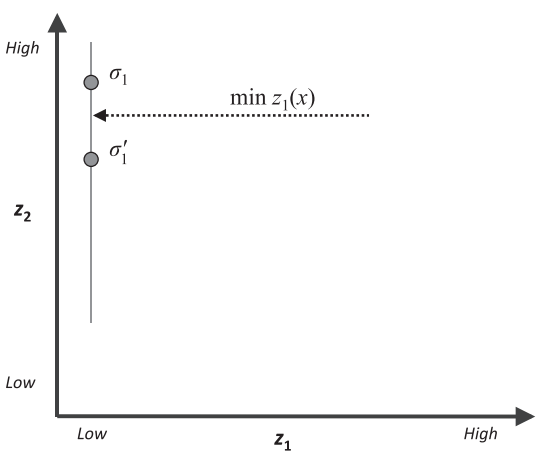


Figure 3. Weakly dominated solution that minimizes $z_1(x)$.

DOI: 10.1002/mcda

respect to objective 1. But there may exist another solution that was not found by the solver σ_1', that weakly dominates σ_1, that is, $z_1(\sigma_1) = z_1(\sigma_1')$ and $z_2(\sigma_1) > z_2(\sigma_1')$.

Later in the algorithm, σ_1' will be found as the solution to a composite objective where α is very close to 1. A proper NISE implementation will put in place mechanisms to detect this scenario, and eliminate σ_1 (in this case) from the final solution.

3.2.2. Multiple equal value composite solutions. Another anomaly arises when solving a composite objective function, that is, $0 < \alpha < 1$, where there are numerous solutions with the same composite objective value. Figure 4 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 σ_j is a supported solution that is on the convex Pareto-frontier but it is not an extreme (corner) point. While its presence does not change the shape of the convex region, it is an optimal trade-off solution that should be kept. The NISE solution approach does not guarantee finding all solutions that are not extreme, but some may be found by chance.

The problem arises when σ_i and σ_j are the 'outer points', that is, `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.

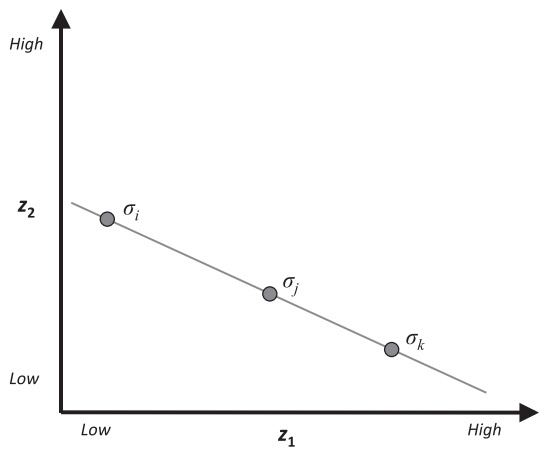


Figure 4. Multiple composite objective optimal solutions.

In order to prevent this error yet also to keep supported non-extreme solutions, rather than checking if an improvement is made to the composite 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.

3.2.3. Complete non-inferior set estimation pseudocode.

```
Complete Algorithm: NISE for Biobjective Shortest Paths
// zC(x) = α*z1(x) + (1− α)*z2(x)
// Ψ = {σ1, σ2,…, σp} = the set of supported non-dominated solutions
// Dij(α) solves a shortest s-t path with Dijkstra's algorithm using a
//composite objective weighted by α
// SetA(σi, σj) selects next value of α based on the z1 and z2 values of
//σi and σj
// RecursiveNISE(σi, σj) computes a supported solution between σi and σj

function: main
α = 1          // minimize first objective
σi = Dij(α)
α = 0          // minimize second objective
σj = Dij(α)
Ψ = σi
Ψ += RecursiveNISE(σi, σj)   // begin recursive NISE procedure
if (z1(σ1) == z1(σ2))
// if σ2 dominates σ1
    Σ.removeFirstElement()
end

function: RecursiveNISE(σi, σj)
α = SetA(σi, σj)     // calculate alpha weighting, equation 6
σk = Dij(α)          // solve composite objective
// if σk is lexicographically between σi and σj
if ((z2(σk) < z2(σi)) and (z1(σk) < z1(σj)))
  if (z2(σk) == z2(σj))  // if σk weakly dominates σj
    Ψ += RecursiveNISE(σi, σk)
  else if (z1(σk) == z1(σi))   // if σk weakly dominates σi
    Ψ += σk
    Ψ += RecursiveNISE(σk, σj)
  else    // else σk is non-dominated
    Ψ += RecursiveNISE(σi, σk)
    Ψ += RecursiveNISE(σk, σj)
  end
else
  Ψ += σj     // else if no improvement found, return σj
end
return Ψ
```


DOI: 10.1002/mcda

3.3. Java fork/join framework

Java is a cross-platform object-oriented programming language that is ubiquitous in scientific, enterprise, 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 primary 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 through 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 solved in parallel. Once a set of subtasks is complete, the results are recombined 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 transforms. 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 5. 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.

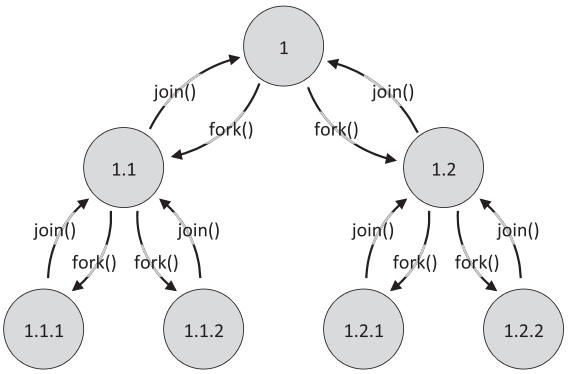


Figure 5. Fork/join task division.

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, which are as follows: 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 and improves the overall parallel efficiency of the application.

3.4. Parallel non-inferior set estimation

3.4.1. 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:

DOI: 10.1002/mcda

```
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
```

3.4.2. Parallel single-objective extreme points. In addition to the binary tree generated from the recursive task division, the initial base case of a biobjective NISE algorithm requires two independent runs of a network optimization solver. These can also be set up to be run in parallel, and because 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 programme.

3.5. Additional improvements to parallel non-inferior set estimation efficiency

3.5.1. Parallelism analysis of basic parallel non-inferior set estimation. The previous section detailed a method for using fork/join concurrency to parallelize the search for supported solutions using the NISE algorithm. While computational results in the next section show major speedup using this basic approach, there is still room for improvement towards trying to achieve the theoretical perfect efficiency. One issue is that the initial computation of the pNISE approach, where

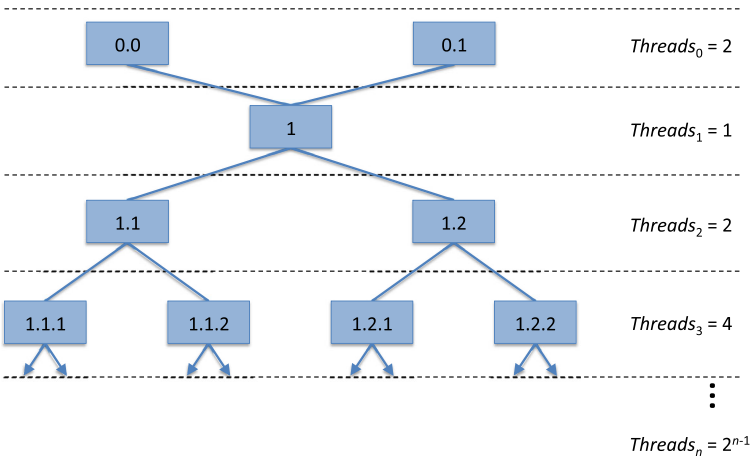


Figure 6. Parallel threads at each stage of parallel non-inferior set estimation.

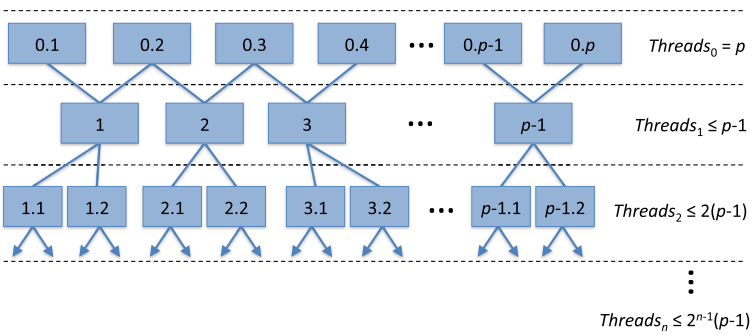


Figure 7. Parallel threads at each stage of the enhanced parallel non-inferior set estimation, making use of unused processors in the initial stages.

DOI: 10.1002/mcda

the weightings are 0 and 1, has only a level of parallelism of 2. When the two solutions 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, and so on. After a few iterations, eventually, there is more parallelism than there are processors and the approach 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. This is essentially a manifestation of Amdahl's law (Amdahl, 1967), which roughly states that a parallel algorithm's speedup is limited by the portion of the work that has less parallelism than there are processors.

Figure 6 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 to divide in a binary 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 before there are enough threads in the thread pool to use all of the processors.

3.5.2. 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 performed 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 $\alpha_i = (i\text{-}1)/(p\text{-}1)$ (i.e. equal intervals between 0 and 1). Once complete, a supported solution will have been found for each weighting, 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 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, with the threads centrally handled by the Java threadpool.

Figure 7 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

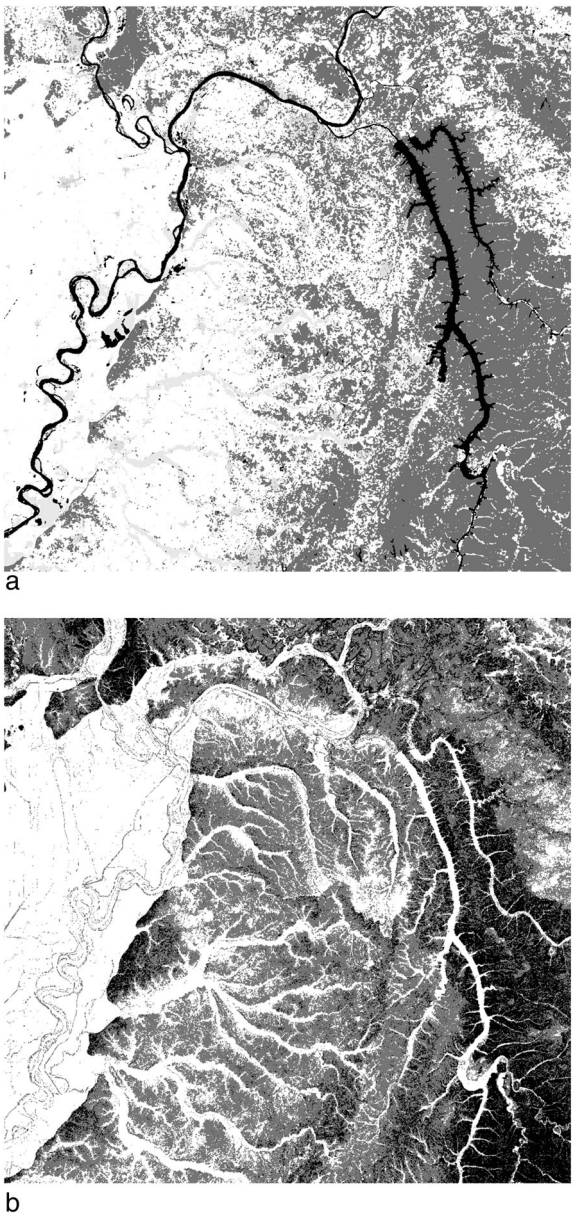


Figure 8. Eastern Interconnection States' Planning Council maps classified into two objectives: environmental impact (top) and construction cost (bottom).

DOI: 10.1002/mcda

Table I. Eastern Interconnection States'Planning Council test networks properties

OD name	Origin node	Destination node	r	Total nodes	Total arcs	Supported non-inferior 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

stage, and all processors again for subsequent stages until computation is complete. This is in contrast with the original approach, which required a few stages to achieve full parallelism. If there are repeated solutions in the initial stage, then there will be fewer than p-1 fork/join sub-problems; but in our experiments with the enhanced pNISE all processors were always used after two levels of recursion.

4. COMPUTATIONAL CASE STUDY

4.1. 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 non-inferior 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 US 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 (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 1000×1000 raster subset of the EZS data, with a 250 square metre cell size. The region analysed 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 versus environmental impact, respectively. Slope values were in percent slope, and land cover was 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 8 graphically displays the EISPC data maps used in the analysis, represented as 1000×1000 rasters and classified with high costs in dark colours and low costs in light colours. The left map represents the environmental impact objective, and the right map represents the construction cost objective.

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 versions were generated, with r radius values of 0, 1 and 2,

DOI: 10.1002/mcda

Table II. Serial non-inferior set estimation (NISE) versus parallel non-inferior set estimation (pNISE) runtimes and speedup

OD	r	Supported solutions	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

respectively. The $r=0$ network corresponds to an orthogonal grid, $r=1$ adds diagonal 'queen's moves', and $r=2$ adds to that 'knight'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 (Goodchild, 1977). Higher-order networks are possible, but the increase in computational effort is not justified because of 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 1000×1000 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 1 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 non-inferior 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

Table III. Parallel non-inferior set estimation 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

DOI: 10.1002/mcda

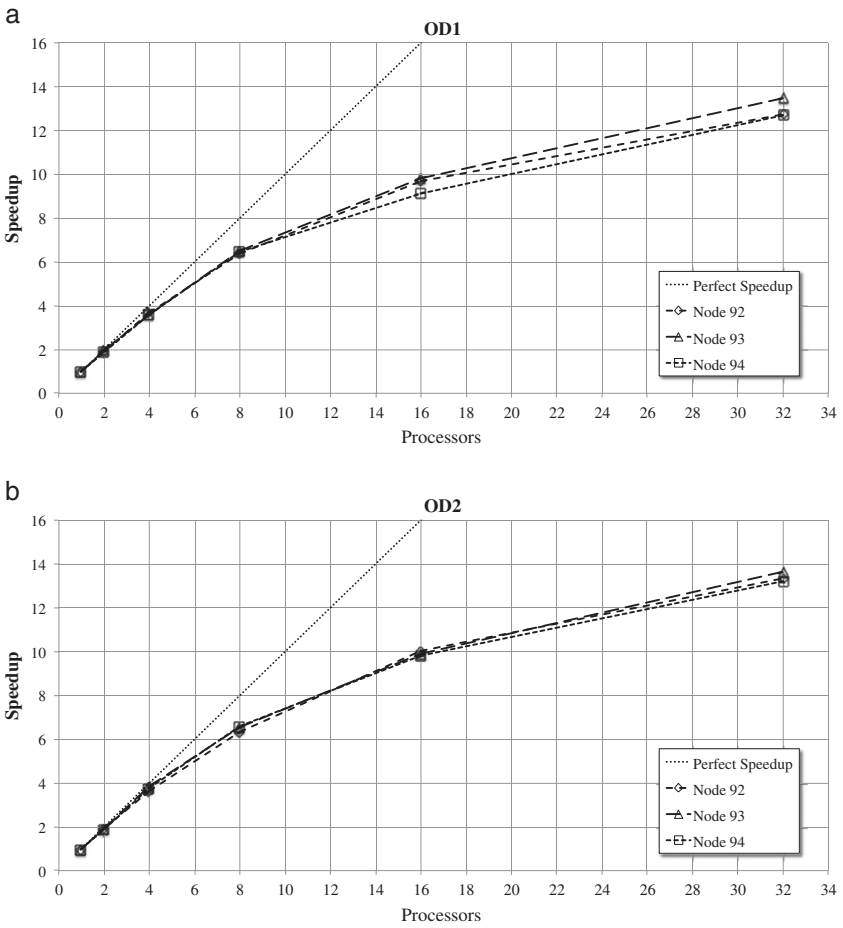


Figure 9. Scaling analysis of the speedup on 32 core nodes for origin/destination OD1 (top) and OD2 (bottom).

heads south, and column numbers increase as one goes heading east.

4.2. 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. The third experiment was run on the same 32-core hardware, and compared the basid pNISE approach to the enhanced pNISE method that uses all processors in the initial stage.

Evaluations of parallel programmes 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

$$S_p = \frac{T_1}{T_p} \tag{7}$$

$$E_p = \frac{S_p}{p} \tag{8}$$

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

DOI: 10.1002/mcda

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, because speedup is limited by the amount of parallelism that exists in a problem instance or programme.

4.3. Computational results

4.3.1. Serial non-inferior set estimations versus parallel non-inferior set estimation. The first experiment tested the serial implementation of NISE to pNISE. The hardware used was an Apple computer with a 3.7 GHz Intel Core i7-3820QM quad-core processor and 16 GB of RAM. Results from this analysis are summarized in Table 2. 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.

4.3.2. Parallel non-inferior set estimation scaling. This experiment evaluated how the efficiency of the pNISE approach scales as the number of processors available is increased. In Java, `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 provided by the University of California Santa Barbara (UCSB) Center for Scientific Computing (CSC), each with 512 GB 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 3. Only OD1 and OD2 were evaluated, as the other problems were too small 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 because there were eight times as many cores per machine.

In general, speedups increased but the parallel efficiency decreased as the number of processors used was increased. The reasons for the drop in

Table IV. Comparison of simple parallel non-inferior set estimation (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				**Enhanced 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

DOI: 10.1002/mcda

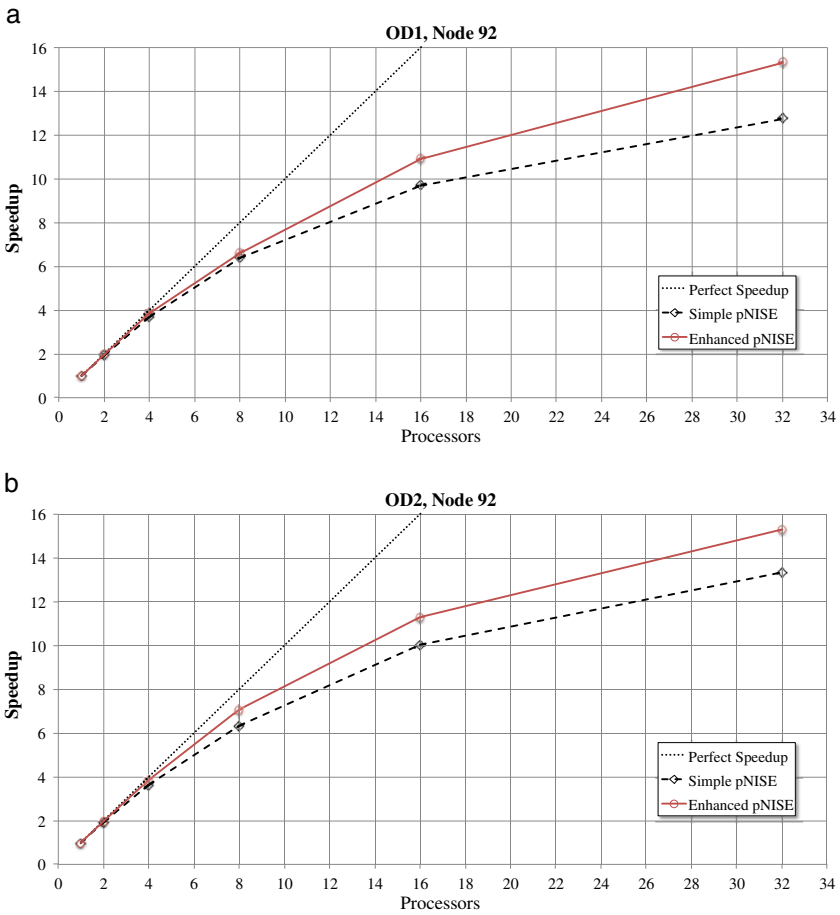


Figure 10. Speedup comparison between simple parallel non-inferior set estimation (pNISE) and enhanced pNISE for origin/destination OD1 (top) and OD2 (bottom).

efficiency as more cores are used is due to a combination of the programme 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 9, which is a plot of speedup versus processors used. Overall, the results reported here demonstrate the value of parallelizing the NISE algorithm.

4.3.3. Parallel non-inferior set estimation versus enhanced parallel non-inferior set estimation. The same scaling experiments as in the previous section were performed with the enhanced pNISE method that uses the parallelized initial solution search and were compared with the simple pNISE approach. All computations were performed on the number 92 node at UCSB's CSC for hardware consistency, and using the same OD1 and OD2 origin-destination pairs.

Table 4 shows the results of these experiments, where the left side displays computation times using the simple pNISE, and on the right using the enhanced pNISE. As the number of processors increase, the overall speedup and efficiencies improve. In the case of the 32 core experiment, the enhanced pNISE was able to improve the speedup from 12.73 to 15.32 for OD1 and from 13.24 to 15.31 for OD2. These improvements are also evident when comparing the speedups graphically as depicted in Figure 10, where the benefits of the enhanced pNISE become more pronounced as the number of processors increase.

DOI: 10.1002/mcda

5. CONCLUDING REMARKS

This work develops a 'simple-to-programme' parallel implementation of the NISE method for computing the supported non-dominated solutions to biobjective network optimization problems, called pNISE. This method uses a high-level fork/join concurrency framework to make this method parallel without the difficult complexities of traditional low-level message-passing parallel languages. Additional enhancements to the pNISE approach were then described that increase the parallelism in early stages of the algorithm, which further improved the parallel performance of the method. 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 multicore computer processors to significantly reduce the computation time of the biobjective supported solution set.

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 expanding presence of big data, scientists and engineers must tackle larger network problems than ever before, requiring novel tools to enable multicriteria analysis 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 for the best decision-making on these challenging problems.

ACKNOWLEDGEMENTS

The authors would like to extend their appreciation to John Krummel and the Environmental Sciences Division of Argonne National Laboratories for providing the funding to conduct this research (1 F-32422). We 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. Finally, we would like to thank the reviewers and Carlos Baez for their helpful comments and feedback.

REFERENCES

Amdahl GM 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS*, ACM: Atlantic City, N.J.; 483–485.

Aneja YP, Nair KPK 1979. Bicriteria transportation problem. *Management Science* **25**: 73–78.

Breschan JR, Heinimann HR 2013. Ecoforest–automatic design of forest patterns attractive to wildlife in an artificial landscape. *Journal of Applied Operational Research* **5**: 125–134.

Cherkassky BV, Goldberg AV, Radzik T 1996. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming* **73**: 129–174.

Clímaco JCN, Pascoal MMB 2012. Multicriteria path and tree problems: discussion on exact algorithms and applications. *International Transactions in Operational Research* **19**: 63–98.

Cohon JL, Church RL, Sheer DP 1979. Generating multiobjective trade-offs: an algorithm for bicriterion problems. *Water Resources Research* **15**: 1001–1010.

Current JR, Revelle CS, Cohon JL 1990. An interactive approach to identify the best compromise solution for two objective shortest path problems. *Computers & Operations Research* **17**: 187–198.

Daskalakis C, Diakonikolas I, Yannakakis M. 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.

Dial RB 1979. A model and algorithm for multicriteria route-mode choice. *Transportation Research Part B: Methodological* **13**: 311–316.

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

Ehrgott M, Wiecek MM 2005. Mutiobjective programming. In *Multiple Criteria Decision Analysis: State of the art Surveys*, Figueira J, Greco S, Ehrgott M (eds). Springer Science+Business Media, Inc.: New York, NY; 667–708.

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

Fischer DT, Church RL 2003. Clustering and compactness in reserve site selection: an extension of the biodiversity management area selection model. *Forest Science* **49**: 555–565.

DOI: 10.1002/mcda

Fry JA, Xian G, Jin S, Dewitz JA, Homer CG, Limin Y, Barnes CA, Herold ND, Wickham JD 2011. Completion of the 2006 national land cover database for the conterminous united states. *Photogrammetric Engineering and Remote Sensing* **77**: 858–864.

Garey MR, Johnson DS 1979. *Computers and Intractability*, Freeman San Francisco: CA.

Goodchild M 1977. An evaluation of lattice solutions to the problem of corridor location. *Environment and Planning A* **9**: 727–738.

Hamacher HW, Pedersen CR, Ruzika S 2007. Multiple objective minimum cost flow problems: a review. *European Journal of Operational Research* **176**: 1404–1422.

Huber DL, Church RL 1985. Transmission corridor location modeling. *Journal of Transportation Engineering-Asce* **111**: 114–130.

Kasprzyk JR, Reed PM, Kirsch BR, Characklis GW 2009. Managing population and drought risks using many-objective water portfolio planning under uncertainty. *Water Resources Research* **45**: 7 pages.

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

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]. Available at http://gee.cs.oswego.edu/dl/concurrency-interest/index.html.*

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

Liebman JC 1976. Some simple-minded observations on the role of optimization in public systems decision-making. *Interfaces* **6**: 102–108.

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

Medaglia AL, Villegas JG, Rodríguez-Coca DM 2009. Hybrid biobjective evolutionary algorithms for the design of a hospital waste management network. *Journal of Heuristics* **15**: 153–176.

Medrano FA, Church RL 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.

Moore GE 1965. *Cramming More Components onto Integrated Circuits*, McGraw-Hill New York: NY, USA.

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

Przybylski A, Gandibleux X, Ehrgott M 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 CR, Fischer DT 2005. Selection of bioclimatically representative biological reserve systems under climate change. *Biological Conservation* **121**: 429–441.

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

Reklaitis GV. 1996. Overview of scheduling and planning of batch process operations. In *Batch Processing Systems Engineering*, Reklaitis GV, Sunol AK, Rippin DWT, Hortaçsu Ö (eds). Springer Berlin Heidelberg: Berlin, Germany; 660–705.

Royset JO, Wood RK 2007. Solving the bi-objective maximum-flow network-interdiction problem. *INFORMS Journal on Computing* **19**: 175–184.

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

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

Schilling DA 1982. Strategic facility planning: the analysis of options. *Decision Sciences* **13**: 1–14.

Solanki RS. 1986. Techniques for approximating the noninferior set in linear multiobjective programming problems with several objectives. Ph.D. Johns Hopkins University.

Soliman Sa-H, Mantawy A-aH. 2012. Optimal power flow. *Modern Optimization Techniques with Applications in Electric Power Systems*. Springer: New York, NY; 281–346.

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

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.

Weber CA, Ellram LM 1993. Supplier selection using multi-objective programming: a decision support system approach. *International Journal of Physical Distribution & Logistics Management* **23**: 3–14.

Article

International Regional Science Review
2014, Vol. 37(2) 129-148

DOI: 10.1177/0160017613507772
irx.sagepub.com

Corridor Location for Infrastructure Development: A Fast Bi-objective Shortest Path Method for Approximating the Pareto Frontier

F. Antonio Medrano[1] and Richard L. Church[1]

Abstract

Corridor location for developing new infrastructure such as transmission lines, roadways, and pipelines over terrain must consider numerous factors when determining the set of optimal route candidates. Previous researchers have cast this problem as a multiobjective least cost path problem, where competing objectives represent cost, environmental impact, and other major noncommensurate objectives. To fully characterize the optimal trade-off solution set, one must be able to generate both supported and unsupported nondominated solutions. Fortunately, the supported, nondominated solutions are relatively easy to identify by using a single-objective shortest path algorithm in conjunction with the noninferior set estimation method of Cohon, Church, and Sheer. But, finding the unsupported nondominated solutions can be nondeterministic polynomial time hard. This article proposes a heuristic approach that is capable of determining a Pareto frontier that is very near exact in polynomial time. This heuristic approach uses gateway node and gateway arc paths to generate a large set of spatially diverse locally optimal candidate

[1] Department of Geography, University of California, Santa Barbara, CA, USA

Corresponding Author:
F. Antonio Medrano, Department of Geography, University of California, 1832 Ellison Hall, Santa Barbara, CA 93106, USA.
Email: medrano@geog.ucsb.edu

solutions with few shortest path solver iterations, which are shown to represent a solution set that approximates to a high degree, the exact Pareto set. The method is described within the context of a bi-objective corridor location problem and applied to several data sets that have been used in past work. A comparison between this new approach and existing exact algorithms is provided. Overall, the new method is shown to be effective at finding a sizable number of the unsupported nondominated solutions in a very small amount of computational time.

Keywords
network analysis, spatial analysis, methods, location models, optimization, other methods, multiobjective and multicriteria analysis, other spatial analysis, geographic information science

Introduction

Regional development rests on the provision of needed infrastructure, including roads, pipelines, communication lines, and transmission facilities. Large corridor projects, like that of the proposed high-speed rail project in California, are often controversial as such development is never devoid of impacts. This means that public sector planning models must be capable of generating significantly different alternatives, as this exploration would likely reveal solutions that are optimal within the context of stated and unstated objectives (Brill 1979). Models must also incorporate numerous objectives, as many factors including construction cost, environmental impact, access for maintenance, proximity to population, and other issues must be simultaneously evaluated for determining a corridor route. Consequently, it is necessary to consider approaches that generate significantly different alternatives with respect to stated objectives, and identify a range of Pareto-optimal solutions. Pareto-optimal solutions are by definition those solutions in which an objective value cannot be improved without one or more other objective values being degraded. In this article, we concentrate on the issue of identifying those corridor solutions that comprise the Pareto-optimal set.

The location of a corridor across a heterogeneous terrain for the purposes of developing new infrastructure such as transmission power lines, pipelines, and roads is a complex problem, as it involves the consideration of factors affecting numerous ecological, social, and economic parameters and a multitude of stakeholders and interested parties. Corridor location is typically defined where the landscape is represented as a raster (or grid cells), and each cell centroid represents a possible center point for a corridor (Church and Clifford 1979). In a simple implementation, a network is then defined where nodes represent cell centers and arcs are used to connect nodes of neighboring cells. A cost for each arc is defined for each objective based upon the attributes of the cells that the arc crosses. Finally, a shortest path algorithm

is used to find the route of least cost or impedance between an origin and a destination. When more than one objective exists, then a multiobjective solution approach must be used. In this article, we assume that there exist at least two competing objectives.

Multiobjective shortest path algorithms have been designed to compute the set of all Pareto-optimal (i.e., nondominated) paths, both unsupported and supported, on a given network. Nondominated *supported* solutions, which represent the convex subset of the nondominated solutions, are relatively easy to identify using a weighted composite objective and a classical shortest path algorithm such as Dijkstra's (1959) algorithm. Unfortunately, identifying *unsupported* nondominated shortest path solutions is a more complex task that is equivalent to solving a constrained shortest path (CSP) problem, which has proved to be nondeterministic polynomial time hard (NP-hard; Garey and Johnson 1979). Therefore, the combinatorial nature of the problem makes computing the exact set of nondominated 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.

In this article, we propose a new technique for determining an approximation of the nondominated solution set for a multiobjective 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 nondominated 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 geographic information system (GIS)-based raster networks and are compared to solutions generated by two exact, state-of-the-art algorithms.

Background

Multiobjective 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 multiobjective 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 that 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 biobjective problems. We will take that same tack and restrict our discussion to bi-

objective path problems for the remainder of the article, except for our final comments.

Let $G = (N, A)$ be a directed graph network with a set of nodes $N = \{1, 2, \ldots, n\}$ and a set of arcs $A = \{(i_1, j_1), \ldots, (i_m, j_m)\}$. Each arc $(i,j) \in A$ has associated with it two positive real costs $c_{ij} = (c_{ij}^1, c_{ij}^2)$. The bi-objective shortest path (BSP) problem aims to solve for paths from a source node $s \in N$ to a destination node $t \in N$ according to the following formulation:

$$\begin{aligned} &min \;\; z_1(x) = \sum_{(i,j)\in A} c_{ij}^1 x_{ij} \\ &min \;\; z_2(x) = \sum_{(i,j)\in A} c_{ij}^2 x_{ij} \\ &\text{s.t.} \quad \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{if } i = t \\ 0 & \text{if } i \neq s,t \end{cases} \\ &\qquad x_{ij} = \{0, 1\} \text{ for all } (i,j) \in A. \end{aligned} \tag{1}$$

While the above formulation contains two distinct objectives, when solving the BSP problem, solution methods often treat the problem as a single objective composed of the weighted sum of these two competing objectives. Since the objective function can vary according to the weights applied, the goal is to find all of the solutions to this optimization problem that represent the optimal trade-offs between all possible nonnegative weights of the two objectives. These solutions are known as *nondominated* or *Pareto-optimal solutions* and are depicted in Figure 1. Continuous, linear bi-objective optimization problems have a continuous, convex Pareto frontier when observed in objective space. Solving for this frontier exactly is simple to do in weakly polynomial time using the noninferior set estimation (NISE) algorithm (Cohon, Church, and Sheer 1979), which was developed originally for continuous variable problems. This approach was later applied to integer bi-objective problems (Solanki 1991) in order to find the discrete set of supported (convex) nondominated solutions. These solutions though do not make up the complete set of Pareto-optimal solutions in a discrete problem and thus there remain triangular regions between each supported solution where unsupported (nonconvex) nondominated solutions may exist. Coutinho-Rodrigues, Climaco, and Current (1999) called these viable unsupported solution regions the *duality gap*. These so-called *gap regions* are also depicted in Figure 1 as shaded triangles. We prefer to call these triangular regions the Boundary of Unsupported Solution Search ($BUSS_{supp}$) regions, with the *supp* subscript denoting that they are defined by the supported nondominated solutions. Our terminology is further discussed in Evaluation of Heuristic subsection.

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

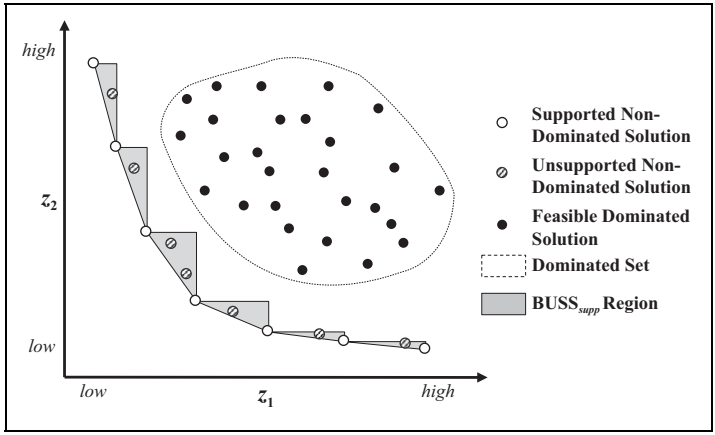


Figure 1. Categorization of bi-objective solutions to a discrete problem.

be found as optima to a weighted, composite single-objective shortest path problem. Any feasible solution that does not fall in the BUSS$_{supp}$ is dominated by at least one other supported nondominated solution. A feasible solution in the BUSS$_{supp}$ is not guaranteed to be nondominated, as a feasible solution within the BUSS$_{supp}$ may be dominated by another solution within the same BUSS$_{supp}$.

Early work on solving the complete Pareto-optimal solution set of a discrete 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, Climaco, and Current (1999). The Coutinho-Rodriques approach, called *GAPS*, is a two-phase method that first employs the NISE method of Cohon, Church, and Sheer (1979) using a classic shortest path algorithm to generate all supported nondominated 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 BSP algorithms of the time, including a label-correcting method by Skriver and Anderson (2000) and a label-setting method of Martins (1984), 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 in the Martins (1984) label-setting approach, called *New Algorithm for Multi-Objective A* (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 BSP 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 article, 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 CSP approach published by Current, Revelle, and Cohon (1990), and a Tchebycheff distance metric solution search method originally 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 CSP model solved via Lagrangian relaxation. Although generating an entire Pareto frontier for a large bi-objective 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 Tchebycheff 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 multiobjective hazmat routing problem by Huang et al. (2008). This method uses a general integer-programming solver and a modified Tchebycheff objective to search for a solution in a user-selected region of the solutions space, and can be applied to a variety of *mixed integer programming* (MIP) problems. If an augmented Tchebycheff distance matrix is used, it guarantees all solutions returned are nondominated, otherwise weakly nondominated solutions may also be found.

Our heuristic, described in the next section, uses a specialized shortest path algorithm for determining the heuristic Pareto set, and thus in exchange for a loss in

generality can solve a shortest path problem much more efficiently than a general MIP solver. In addition, our heuristic is able to generate many solutions per solver iteration, allowing for a greater number of nondominated solutions to be found than shortest path solver iterations. While all the solutions are not always exact nondominated, our experiments show that this method generates a set of near-Pareto-optimal solutions that very closely approximate the exact frontier.

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, Geneletti, and Orsi 2011). Aissi, Chakhar, and Mousseau (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 nondominated paths found.

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 (GWA), where the shortest path between the origin and destination must traverse a given intermediate arc. This will be called a *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 nondominated path that includes that GWN/GWA. 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 (1959) algorithm 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 $u \underset{T(u)}{\longrightarrow} v$. 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 $s \underset{T(s)}{\longrightarrow} u \underset{T(t)}{\longrightarrow} t$, and a GWA path constrained to pass through arc $(u, \mathrm{v}) \in A$ is defined as $s \underset{T(s)}{\longrightarrow} u \rightarrow v \underset{T(t)}{\longrightarrow} t$. In essence, each GWN path can be generated by traversing the path from the origin s to the GWN u on tree $T(s)$ and then traversing the reverse path from node u on tree $T(t)$ to the destination. The shortest GWA 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 GWN and GWA 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 GWN path can be found as an equivalent GWA path for the arc that is used on the shortest path tree touching node u on tree $T(s)$. In general, the number of GWA paths is significantly greater than the number of GWN paths.

It is important to point out that gateway paths formed the basis of the fast loopless KSP algorithm developed by Katoh, Ibaraki, and Mine (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 article) 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, Church, and Medrano (2013) explore using multiple GWNs to generate more intricate path alternatives than what is possible with single GWNs.

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.[1] 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 noninferior 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 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 Gateway Node Heuristic (GWNH), is given in Figure 2 below, and can be summarized by the following steps:

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, Church, and Sheer (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 unfathomed supported points, and return to step 1. If all adjacent points are fathomed, terminate.

The GWA algorithm, called *Gateway Arc Heuristic* (*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, Church, and Sheer (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 multiobjective 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 GWN paths, or m GWA paths, so in essence it generates far more solutions than solver iterations. In the case of GWN 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.

Experimental Analysis

Experimental Setup

We programmed both GWNH and GWAH, and for comparison, also encoded 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 subroutine, we implemented Dijkstra's algorithm using a binary heap priority queue. All methods were programmed

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 u $\in N$
  $\upsilon_u = s \xrightarrow[T(s)]{} u \xrightarrow[T(t)]{} t$
  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

Figure 2. Bi-objective gateway node path heuristic pseudocode.

Table 1. Test Networks.

Raster	r	Nodes	Arcs	Arcs/nodes
20 × 20	0	400	1,520	3.8
	1	400	2,964	7.41
	2	400	5,700	14.25
80 × 80	0	6,400	25,280	3.95
	1	6,400	50,244	7.85
	2	6,400	99,540	15.57
100 × 160	0	16,000	63,480	3.97
	1	16,000	126,444	7.90
	2	16,000	251,340	15.71

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 20 × 20 manually fabricated raster, (2) an 80 × 80 subset of the Maryland Automated Geographic Information (MAGI) system database, and (3) a 100 × 160 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 100 × 160 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 trade-offs 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 *noninteger* in value, thus all algorithms used *double precision arithmetic*. Table 1 presents the general statistics of these two data sets.

Evaluation of Heuristic

In addition to testing the heuristic runtimes compared to the exact algorithms, we 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 BUSS region of an

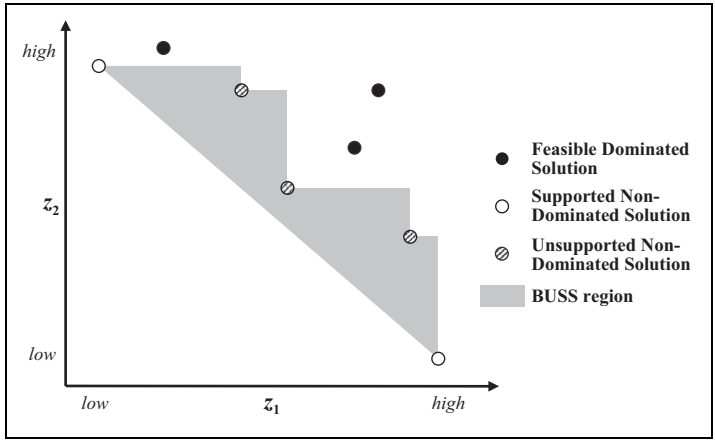


Figure 3. Defining the boundary of unsupported solution search (BUSS).

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 nondominated 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 3).

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

$$E_{\text{ratio}} = \frac{\text{BUSS}_h}{\text{BUSS}_e}. \tag{2}$$

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

$$E_{\text{norm}} = \frac{\text{BUSS}_h - \text{BUSS}_e}{\text{BUSS}_{supp} - \text{BUSS}_e}, \tag{3}$$

where BUSS_{supp} is the BUSS region defined by only the supported solutions (also called the *duality gap* in Coutinho-Rodrigues, Climaco, and Current 1999). If the heuristic solution is equal to the exact solution, then $E_{\text{norm}} = 0$. If the heuristic solution does not improve upon the supported solution set, then $E_{\text{norm}} = 1$. Anything in

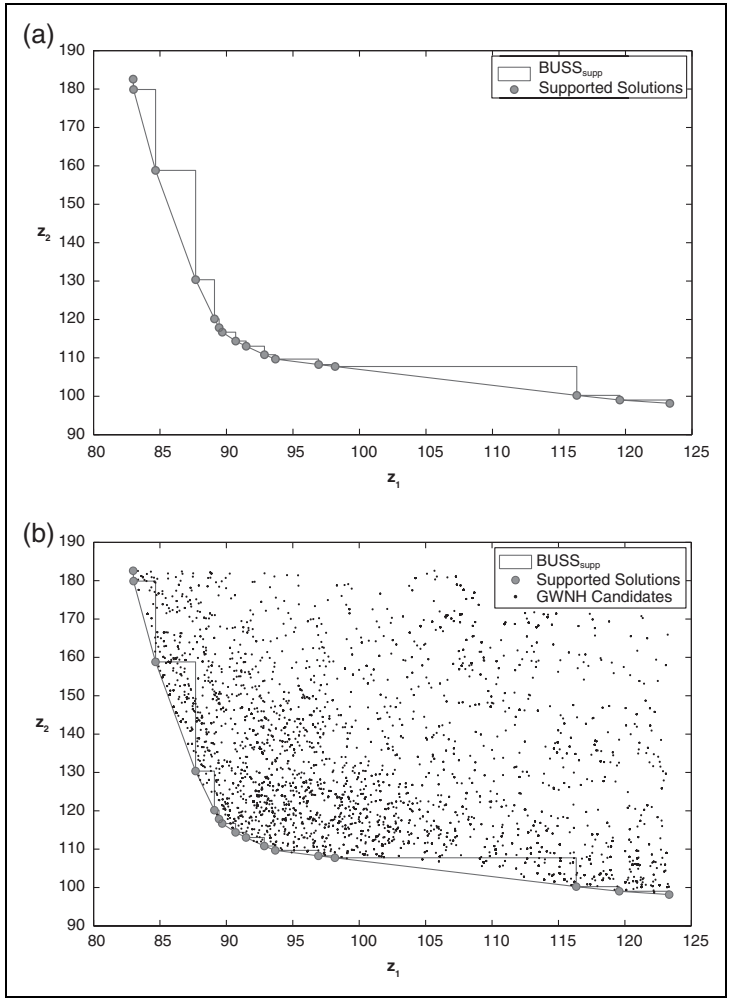


Figure 4. Objective space evaluation on the 20 × 20 $r = 2$ network. (a) Supported solutions only, (b) GWNH candidate solutions, (c) GWNH Pareto solutions, (d) exact versus heuristic solutions. GWNH = Gateway Node Heuristic.

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 nondominated solutions.

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 4 a–d illustrate one heuristic solution set of Pareto paths from our experiments, in this case, the figures illustrate results from the GWNH applied to the 20 × 20 $r = 2$ network.

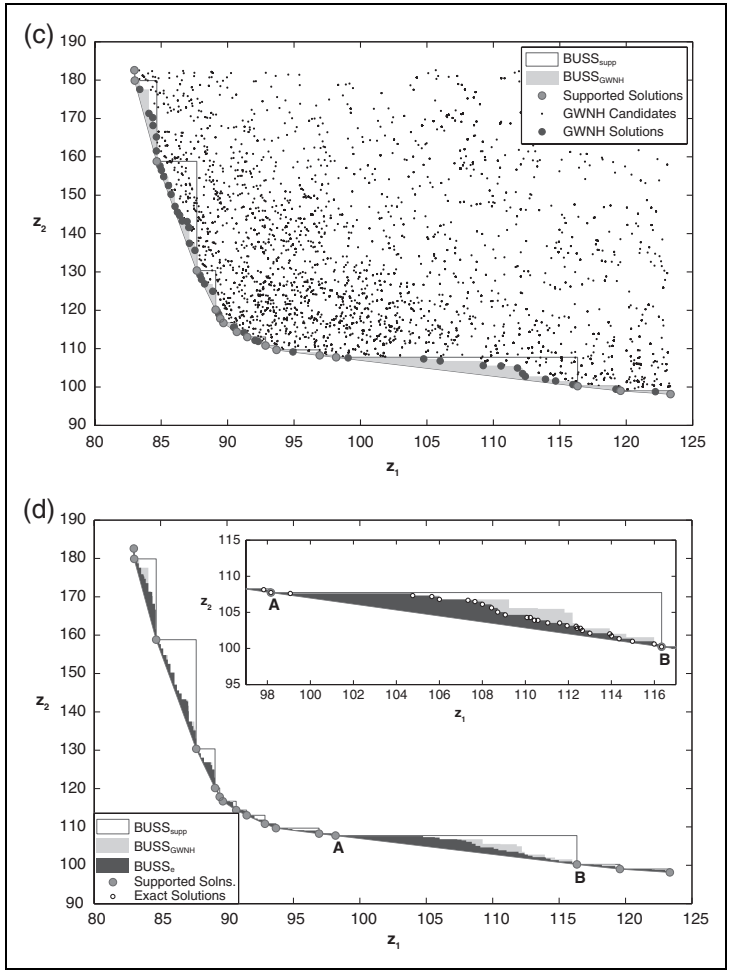


Figure 4. (continued)

Figure 4a 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 4b 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 a one or more supported solutions; yet, many of the GWNH solutions fall inside the BUSS$_{supp}$ areas. Figure 4c

Table 2. Algorithm Computation Time Performance (in Seconds).

		LCor	2LCor	GWNH	GWAH
20 × 20	$r = 0$	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
80 × 80	$r = 0$	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
100 × 160	$r = 0$	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

Note: GWAH = Gateway Arc Heuristic; GWNH = Gateway Node Heuristic.

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 nondominated 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 4d only shows the exact supported solutions and gives the exact $BUSS_e$ area in dark gray on top of the heuristic $BUSS_{GWNH}$ area in lighter gray. 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 noninferior 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.

We depicted solutions from the GWNH in Figure 4 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 percent larger than the exact region. Given this example, we now turn our attention to comparing exact and heuristic approaches.

Computational Results

Tables 2 through 4 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 GWNH; and (4) the GWAH. Table 2 gives the computation times in seconds for each method applied on the two different data sets using three different network definitions (orthogonal, queens, and

Table 3. Number of Exact/Heuristic Solutions.

		Exact supported	Exact all	GWNH		GWAH	
		Solutions	Solutions	Solutions	EOS	Solutions	EOS
20 × 20	$r = 0$	10	28	26	24	27	26
	$r = 1$	11	49	43	43	46	46
	$r = 2$	16	103	60	51	75	71
80 × 80	$r = 0$	29	120	114	111	117	114
	$r = 1$	36	371	246	201	258	211
	$r = 2$	57	1339	465	317	647	503
100 × 160	$r = 0$	11	22	21	21	21	21
	$r = 1$	19	199	129	110	134	116
	$r = 2$	35	718	344	292	383	322

Note: EOS = exact optimal solutions; GWAH = Gateway Arc Heuristic; GWNH = Gateway Node Heuristic.

Table 4. 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}
20 × 20	$r = 0$	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
80 × 80	$r = 0$	717.50	210.50	220.50	1.048	0.0197	217.50	1.033	0.0138
	$r = 1$	1,069.60	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
100 × 160	$r = 0$	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
	$r = 2$	1,121.629	285.49	298.99	1.047	0.0161	296.63	1.039	0.0133

Note: BUSS = Boundary of Unsupported Solution Search; GWAH = Gateway Arc Heuristic; GWNH = Gateway Node Heuristic.

queens plus knights). Table 3 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 3 also gives the number of candidate nondominated 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 4 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, 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 20 × 20 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 percent error), in networks of small size such as these it is best to simply use an exact approach. In the largest 20 × 20 network ($r = 2$), the label correcting algorithms are slower by a factor of between 2 and 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 percent and 11.4 percent for the GWNH and GWAH, respectively, with normalized errors of 12.2 percent and 6.3 percent.

For the least dense 80 × 80 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 2 × slower than the heuristics, while the heuristics yielded solutions that were both <10 percent in error ratio. The real performance difference came in the most dense 80 × 80 ($r = 2$) 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 percent of the exact for GWNH, and 11.5 percent for GWAH, with normalized errors of 5.86 percent for GWNH and 3.31 percent 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 100 × 160 network, solution times followed the same trends found in solving the 80 × 80; 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 ten times faster than the fastest exact methods. All heuristic methods, when run on the 100 × 160 network, resulted in error ratios and normalized errors of <5 percent. 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 EOS set (Table 3). For the higher-density networks, a somewhat smaller proportion of the heuristic solutions were EOS. For example, in the $r = 2$ 100 × 160 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 percent 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 percent 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 GWN paths as compared to scanning the number of GWA 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 nondominated 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 trade-off between the two heuristics in terms of computation time and solution quality.

Conclusion

This study presents a new heuristic method for generating an approximate Pareto-optimal solution set for a BSP problem. The new method has two variants for efficiently generating a large set of candidate paths, one making use of GWN paths and the other using GWA 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 article could extend to higher dimensions when used in conjunction with a multidimensional NISE approach such as the one developed by Solanki, Appino, and Cohon (1993), and could offer a faster option for approximating the Pareto front for three or more objective shortest path problems.

Acknowledgments

We would also like to acknowledge helpful comments of Anita Schöbel when an earlier version of this article was presented at the ISOLDE XII in meetings in Kyoto, Japan, 2012. And finally, we also would like to thank the reviewers, whose comments and insights were of great help.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: The authors would like to thank the Environmental Sciences Division of Argonne National Laboratories for providing the funding to conduct this research (1F-32422).

Note

1. Typically, noninferior set estimation (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 nondominated.

References

Aissi, H., S. Chakhar, and V. Mousseau. 2012. "Gis-based Multicriteria Evaluation Approach for Corridor Siting." *Environment and Planning B: Planning and Design* 39:287–307.

Atkinson, D. M., P. Deadman, D. Dudycha, and S. Traynor. 2005. "Multi-criteria Evaluation and Least Cost Path Analysis for an Arctic All-weather Road." *Applied Geography* 25:287–307.

Azevedo, J. A., J. J. E. R. S. Madeira, E. Q. V. Martins, and F. M. A. Pires. 1994. "A Computational Improvement for a Shortest Paths Ranking Algorithm." *European Journal of Operational Research* 73:188–91.

Bagli, S., D. Geneletti, and F. Orsi. 2011. "Routeing of Power Lines through Least-cost Path Analysis and Multicriteria Evaluation to Minimise Environmental Impacts." *Environmental Impact Assessment Review* 31:234–39.

Brill, E. D. 1979. "The Use of Optimization Models in Public-sector Planning." *Management Science* 25:413–22.

Carlyle, W. M., and R. K. Wood. 2005. "Near-shortest and k-shortest Simple Paths." *Networks* 46:98–109.

Church, R. L., and T. J. Clifford. 1979. "Discussion of Environmental Optimization of Power Lines." *Journal of the Environmental Engineering Division* 105:438–439.

Cohon, J. L., R. L. Church, and D. P. Sheer. 1979. "Generating Multiobjective Trade-offs: An Algorithm for Bicriterion Problems." *Water Resources Research* 15:1001–10.

Coutinho-Rodrigues, J., J. Climaco, and J. Current. 1999. "An Interactive Bi-objective Shortest Path Approach: Searching for Unsupported Nondominated Solutions." *Computers & Operations Research* 26:789–98.

Current, J. R., C. S. Revelle, and J. L. Cohon. 1990. "An Interactive Approach to Identify the Best Compromise Solution for Two Objective Shortest Path Problems." *Computers & Operations Research* 17:187–98.

Dijkstra, E. W. 1959. "A Note on Two Problems in Connexion with Graphs." *Numerische Mathematik* 1:269–71.

Garey, M. R., and D. S. Johnson. 1979. *Computers and Intractability*. San Francisco, CA: Freeman.

Ghoseiri, K., and B. Nadjari. 2010. "An Ant Colony Optimization Algorithm for the Bi-objective Shortest Path Problem." *Applied Soft Computing* 10:1237–46.

Goodchild, M. 1977. "An Evaluation of Lattice Solutions to the Problem of Corridor Location." *Environment and Planning A* 9:727–38.

Huang, B., P. Fery, L. Xue, and Y. Wang. 2008. "Seeking the Pareto Front for Multiobjective Spatial Optimization Problems." *International Journal of Geographical Information Science* 22:507–26.

Huber, D. L., and R. L. Church. 1985. "Transmission Corridor Location Modeling." *Journal of Transportation Engineering-Asce* 111:114–30.

Katoh, N., T. Ibaraki, and H. Mine. 1982. "An Efficient Algorithm for k Shortest Simple Paths." *Networks* 12:411–27.

Lombard, K., and R. Church. 1993. "The Gateway Shortest Path Problem: Generating Alternative Routes for a Corridor Location Problem." *Geographical Systems* 1:25–45.

Mandow, L., and J. L. Pérez De La Cruz. 2005. "A New Approach to Multiobjective A* Search." International Joint Conference on Artificial Intelligence, 218–223.

Martins, E. Q. V. 1984. "On a Multicriteria Shortest Path Problem." *European Journal of Operational Research* 16:236–45.

Raith, A., and M. Ehrgott. 2009. "A Comparison of Solution Strategies for Biobjective Shortest Path Problems." *Computers & Operations Research* 36:1299–331.

Scaparra, M. P., R. L. Church, and F. A. Medrano. 2013. "Corridor Location: The Multi-gateway Model." Working Paper, Department of Geography, University of California, Santa Barbara, CA.

Skriver, A. J. V., and K. A. Andersen. 2000. "A Label Correcting Approach for Solving Bicriterion Shortest-path Problems." *Computers & Operations Research* 27:507–24.

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." PhD thesis from Johns Hopkins University.

Solanki, R. S., P. A. Appino, and J. L. Cohon. 1993. "Approximating the Noninferior Set in Multiobjective Linear Programming Problems." *European Journal of Operational Research* 68:356–73.

Steuer, R. E., and E.-U. Choo. 1983. "An Interactive Weighted Tchebycheff Procedure for Multiple Objective Programming." *Mathematical Programming* 26:326–44.

Chapter 7
A Parallel Algorithm to Solve Near-Shortest Path Problems on Raster Graphs

F. Antonio Medrano and Richard L. Church

Abstract The Near-Shortest Path (NSP) algorithm (Carlyle and Wood, Networks 46(2): 98–109, 2005; Medrano and Church, GeoTrans RP-01-12-01, UC Santa Barbara, 2012) has been identified as being effective at generating sets of good route alternatives for designing new infrastructure. While the algorithm itself is faster than other enumerative shortest path set approaches including the Kth-shortest path problem, the solution set size and computation time grow exponentially as the problem size or parameters increase, and requires the use of high-performance parallel computing to solve for real-world problems. We present a new breadth-first-search parallelization of the NSP algorithm. Computational results and future work for parallel efficiency improvements are discussed.

Keywords Parallel algorithms • Near-shortest paths • kth-shortest paths • Shortest path algorithms

7.1 Introduction

Since it was first declared in 1965, Moore's Law has correctly predicted that the number of transistors on an integrated circuit, and thus computational power, would double every 2 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

F.A. Medrano (✉) • R.L. Church
Department of Geography, University of California at Santa Barbara,
Santa Barbara, CA, USA
e-mail: medrano@geog.ucsb.edu; church@geog.ucsb.edu

X. Shi et al. (eds.), *Modern Accelerator Technologies for Geographic Information Science*,
DOI 10.1007/978-1-4614-8745-6_7,

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, starting with dual-core, then quad-core, to now in 2013 where some processors have 8-cores capable of simultaneously handling 16-threads, have allowed the progress of Moore's Law to continue. 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 their 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. Since each model simulation is an independent calculation to every other model simulation, then 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 et al. 2008; Cong et al. 2008; Chhugani et al. 2012; Merrill et al. 2012). This chapter 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. While the approach described here does make some progress in being able to make this path algorithm concurrent, our results suggest that there is considerable room for improvement, and we suggest future research plans and their associated challenges toward the end of this chapter. While this is a difficult problem, the development of new parallel path algorithm approaches is an important need when facing complex tasks such as robot operations and corridor planning on increasingly large and more complex networks.

Single objective shortest or least cost path tools are common in present day GIS software packages. These tools are useful in computing an optimal route over a

terrain or road network, in which some objective such as distance or cost is to be minimized. Real-world problems are often more complicated than simply minimizing a single objective, and thus designers often need to solve more complex path problems such as the resource constrained shortest path (minimize cost A while not exceeding some quantity of cost B) or the multi-objective shortest path (minimize a weighted sum of numerous costs). The constrained shortest path and related problems are NP-Complete, and are thus quite difficult to solve.

Various methods using enumeration algorithms have been published for solving the resource constrained shortest path (Beasley and Christofides 1989; Carlyle et al. 2008; Handler and Zang 1980) and multi-objective shortest paths (Clímaco and Coutinho-Rodrigues 1988; Coutinho-Rodrigues et al. 1999; Raith and Ehrgott 2009). Most of these methods involve enumeration of some sort, either solving a Kth Shortest Path (KSP) problem, which returns a ranked list of shortest paths from an origin to a destination; or solving a Near-Shortest Path (NSP) problem, which returns a set of paths from an origin to a destination such that all are less than some defined cost. For loopless paths, algorithms for solving the NSP problem are much more efficient than those for solving KSP problems, and it has been shown that it is faster to solve the KSP by first solving an NSP and then post-processing the output (Carlyle and Wood 2005). This being the case, our research focuses on parallelizing the fastest known NSP algorithm (Carlyle and Wood 2005). Parallelization is necessary to solve large-scale problems, since we show that the solution set size will grow exponentially as problem sizes increases.

7.2 Background

Shortest path algorithms defined for network problems have been an active area of computational research since the 1950s. While shortest path routing has been a fundamental human problem since the dawn of time, Alex Orden in 1956 (Orden 1956) was the first to formulate mathematically the shortest path problem, which he did as a linear program. This was followed shortly by a number of different algorithms developed to solve the shortest path problem, including the well-known Bellman-Ford Algorithm (Bellman 1958) and Dijkstra's Algorithm (Dijkstra 1959). Since then, there have been many advances and refinements in shortest path algorithms, and recent comparisons between various methods can be found in Cherkassky et al. (1996), Zeng and Church (2009), and Zhan and Noon (2000).

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

The Near-Shortest Path (NSP) Problem, originally formulated by Byers and Waterman (Byers and Waterman 1984), is a slight variation of the KSP problem. Unlike the KSP, which returns a ranked list of the k shortest paths on the network, the NSP problem returns all distinct paths on the network between an origin and destination longer than the shortest path within a prescribed threshold, ε, expressed as a decimal fraction. If the shortest path has length L_{sp}, then the NSP returns all paths of length $\leq (1+\varepsilon) \times L_{sp}$.

7.3 Carlyle and Wood's Near-Shortest Path Algorithm

In their 2005 paper, Carlyle and Wood (2005) present two different algorithms for finding loopless NSPs, ANSPR0 and ANSPR1 (Algorithm Near Shortest Paths Restricted 0 and 1 respectively). ANSPR0 is based on the Byers and Waterman (1984) method with a modification to output only loopless paths. The general idea is to find all paths of length $\leq D$ on the network, where $D = (1+\varepsilon) \times L_{sp}$. First, it solves the reverse shortest path tree (all shortest paths from the destination to all other nodes on the network) to acquire the shortest path cost from any node to the destination, t. This is the only time that a traditional shortest path algorithm is used. It then uses depth-first search (DFS) and a first-in last-out stack to generate the set of NSP's. This approach is very efficient because the DFS uses only fast addition/comparison operations, and never has to repeat any shortest path calculations in the process of generating paths. While it has an exponential worst-case complexity, it takes a pathological example to create such behavior. ANSPR1, has a better worst-case complexity, but when implemented was shown to run slower than ANSPR0. Combined with a binary search tree, Carlyle and Wood showed that the ANSPR1 algorithm could be modified to solve the KSP problem much faster than the fastest loopless KSP method of Katoh et al. (1982) as implemented by Hadjiconstantinou and Christofides (1999). While no other experiments have been published that compare Carlyle and Wood's algorithm to other KSP algorithms, Carlyle et al. (2008) argue that enumerating paths in order of length requires undue computational effort, and if it is not necessary to use KSP then the NSP is far superior.

7.4 The Need for Parallelization

To demonstrate the need for a parallelized approach to the NSP algorithm, we first wrote a serial JAVA implementation of the NSP algorithm and tested it on two networks:

1. 20×20 manually fabricated raster. This network contains 400 nodes and 2,850 undirected arcs.
2. 80×80 subset of the Maryland Automated Geographic Information System (MAGI) database. This network contains 6,400 nodes and 49,770 undirected arcs.

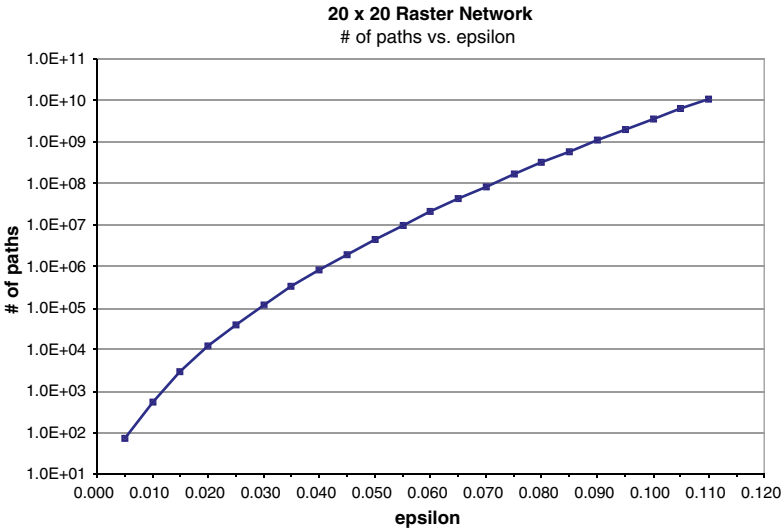


Fig. 7.1 20×20 network, log number of paths generated by the ANSPR0 vs. epsilon

Both of these data sets were first used by Huber and Church (1985). Both networks have undirected "queen's and knight's move" arcs emanating from each node, as this raster network model has been shown to offer a good compromise between accuracy and computational burden (Huber and Church 1985).

To characterize the solution set growth rate, we plotted the number of paths output by NSP for numerous values of ε on both networks on a log-linear plot. Figure 7.1 shows this plot for the 20×20 network. The logarithmic y-axis is of the number of paths output, and the x-axis is the ε value. The result is a straight-line trend, indicating an exponential growth in the number of paths as ε increases. Similarly, as seen in Fig. 7.2, the computation time as a function of ε was found to be proportional to solution size growth, and when plotted on the log-linear axis also showed exponential growth with respect to ε.

The NSP algorithm generated nearly 4 billion solutions on the 20×20 raster region when the epsilon value was set at 0.10, i.e. within 10 % of the optimal path. Because the number of paths for each given value of ε is much higher for the 80×80 network as compared to the 20×20, the range of epsilons used in our 80×80 experiments were an order of magnitude smaller than those in our 20×20 experiments. For example, for ε=0.005, on the 20×20 data this generated a solution set of 73 near-shortest paths; but for the 80×80 with the same ε=0.005, there were 510,343,616 such paths.

These experiments demonstrated that generating a set of NSPs can be an enormous task and may overwhelm computational resources as one increases the network size or increases the value of ε. Generating all paths within 0.75 % of the shortest path length on the 80×80 network took more than 4 days using an Intel

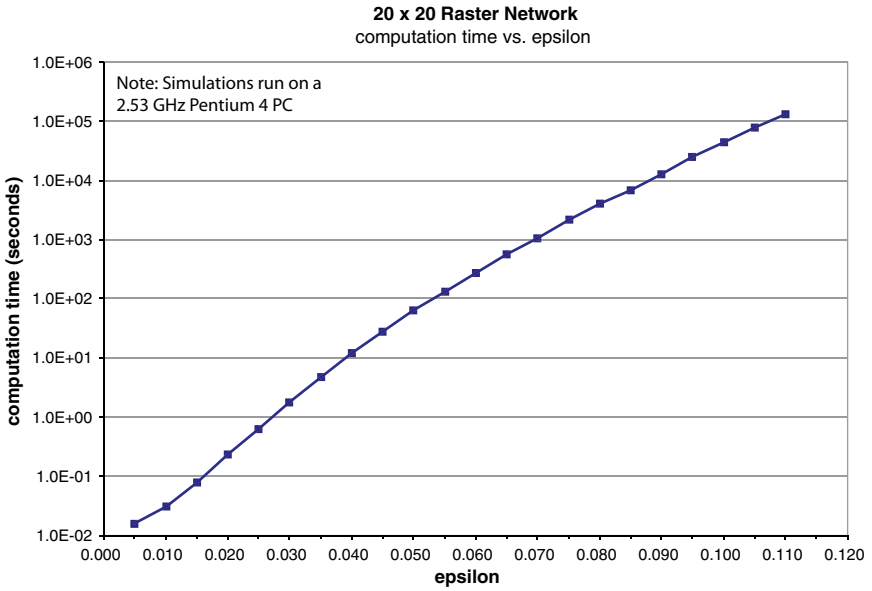


Fig. 7.2 20×20 network, log computation time of ANSPR0 vs. epsilon

Core i7 desktop and employing a serial JAVA implementation, even though this application was able to generate 185,000 paths per second! The quad-core processor used was capable of running two threads per core, yet the serial code was using only one out of the eight potential threads of this processor. While we make no claims that the serial code cannot be further optimized, the reality is that even with the best serial 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 running a serial NSP code. Instead, in order to have any hope of making this computation, one must take advantage of the full parallel capabilities of modern processors.

7.5 Parallelizing Depth-First-Search

Our proposed technique of converting the Carlyle and Wood depth-first-search NSP algorithm into a parallel algorithm begins with performing an initial breadth-first-search (BFS) generation of all NSPs emanating from the origin point. Rather than completely building one NSP at a time with DFS, the BFS simultaneously builds the start of all NSPs. BFS naturally forms a tree structure, with concurrent paths sharing branches until they deviate, and the end nodes of the various path stems resulting as different leaves on the tree. We stored the BFS paths in a "*trie*" data structure (Hadjiconstantinou and Christofides 1999; Aho et al. 1983), which is an efficient

data structure for storing sets of paths which partially overlap. If the entire computation were performed as a BFS, then every leaf of the final *trie* would be the destination node of the network. In our method, the BFS runs until there are as many leaves on the BFS tree as there are processors available for computation, at which point each processor is then tasked with running the traditional DFS NSP algorithm using a *trie* leaf as a starting point, finding all paths from that point which are less than the threshold length minus the path-length from the origin to the leaf starting node.

7.6 Analysis of Implementation

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.

Table 7.1 contains data collected from various runs of our first generation parallel code implementation on the 20×20 data set on the Triton nodes. The columns show epsilon value, number of processors, total NSP runtime in seconds, total paths found, paths found on the leaf of fewest paths (Min Paths Leaf), paths found on the leaf of most paths (Max Paths Leaf), speedup, and parallel efficiency. 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 hopefully use all processors efficiently with no idle time and reach a parallel efficiency of 1.0 overall, although this rarely happens for all but the most trivial algorithms.

In the computational results shown in Table 7.1 we see that good parallel efficiency was achieved when the ratio between the maximum number of paths found on a leaf and the minimum number of paths found on a leaf is not too great. For example, when $\varepsilon = 0.05$, and 5 processors were employed (BFS depth = 1), the ratio was approximately 6:1 "max to min paths". This resulted in a very respectable 0.61 value of parallel efficiency. With $\varepsilon = 0.05$ and employing 48 processes (BFS depth = 2), the "max to min paths" ratio was approximately 50,000:1. The minimum path leaf quickly finished its work in 0.2 ms, while the maximum path leaf took 2.5 s to complete. This significant amount of relative idle time resulted in a lesser parallel efficiency of 0.18.

Table 7.2 gives results for computational tests on the 80×80 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 an unimpressive speedup of 1.61 when using 38 processors, which is equivalent to a parallel efficiency of 0.04.

As a result of the independent computation of each leaf, we found that that the expected overall parallel efficiency followed this relationship:

Table 7.1 Parallel NSP runtime results on 20×20 network

Epsilon	Number of processors	Time (s)	Total paths		Min paths leaf	Max paths leaf	Speed-up	Parallel efficiency
0.05	1	21.73	4,601,053	Paths	4,601,053		1.00	1.00
				Time	21.729			
				Paths/s	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/s	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/s	56,403	225,750		
0.07	1	392.37	86,384,393	Paths	86,384,393		1.00	1.00
				Time	392.373			
				Paths/s	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/s	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/s	85,091	236,353		

Table 7.2 Parallel NSP runtime results on the 80×80 network

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

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

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

7.7 Implementation Challenges: 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 our parallel approach to the NSP problem. For further details, please refer to Medrano and Church (2012).

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 ($\gg p$), 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 80×80 data set (Table 7.2), 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.

7.8 Conclusions

The goal of this research was to develop an efficient parallel implementation of the fastest NSP algorithm. The approach described here uses breadth-first-search to split the work up in a pleasingly parallel fashion, and was able to show a significant speedup when computing with multiple processors. Large variances in the work-chunk sizes though prevented this approach from running at theoretically optimal parallel efficiencies. Further work is needed in exploring approaches to more evenly spread workload across various processors. We described possible methods for further improvements in parallel efficiency, and of those we recommend devising a predictive metric that could be used to estimate the work on each portion of the BFS tree, and stunting tree growth where small work would be expected, followed by a work-stealing scheme during the computation. We expect that the former would result in a more consistent set of work-chunks, while the latter would even-out any remaining imbalances, leading to improved overall efficiency and performance of the parallel code. Further research would aim to develop an effective GIS tool able to solve larger and more complicated path routing problems.

Acknowledgements We would like to thank the Environmental Sciences Division of Argonne National Laboratories for providing the funding to conduct this research (1F-32422).

References

Adler, M., et al.: Parallel randomized load balancing. In: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, 238–247 ACM City (1995)

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

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

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

Beasley, J.E. and N. Christofides: An Algorithm for the Resource Constrained Shortest-Path Problem. Networks, 19(4), 379–394 (1989)

Bellman, R.E.: On a routing problem. Q. Applied Math, 1687–90 (1958)

Blumofe, R.D. and C.E. Leiserson: Scheduling multithreaded computations by work stealing. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. IEEE. (1994)

Bock, F., H. Kantner, and J. Haynes: 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, (1957)

Byers, T. and M. Waterman: Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. Operations Research, 32(6), 1381–1384 (1984)

Carlyle, W.M. and R.K. Wood: Near-shortest and K-shortest simple paths. Networks, 46(2), 98–109 (2005)
Carlyle, W.M., J.O. Royset, and R.K. Wood: Lagrangian Relaxation and Enumeration for Solving Constrained Shortest-Path Problems. Networks, 52(4), 256–270 (2008)
Chhugani, J., et al.: Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE. (2012)
Cherkassky, B.V., A.V. Goldberg, and T. Radzik: Shortest paths algorithms: theory and experimental evaluation. Mathematical programming, 73(2), 129–174 (1996)
Clímaco, J. and J. Coutinho-Rodrigues: On an interactive bicriteria shortest path algorithm. Lisbon, Portugal. (1988)
Cong, G., et al.: Solving large, irregular graph problems using adaptive work-stealing. In: Parallel Processing, 2008. ICPP'08. 37th International Conference on. IEEE. (2008)
Coutinho-Rodrigues, J., J. Climaco, and J. Current: An interactive bi-objective shortest path approach: searching for unsupported nondominated solutions. Computers & Operations Research, 26(8), 789–798 (1999)
Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271 (1959)
Hadjiconstantinou, E. and N. Christofides: An efficient implementation of an algorithm for finding K shortest simple paths. Networks, 34(2), 88–101 (1999)
Handler, G.Y. and I. Zang: A dual algorithm for the constrained shortest path problem. Networks, 10(4), 293–309 (1980)
Hoffman, W. and R. Pavley: A Method for the Solution of the N th Best Path Problem. Journal of the ACM (JACM), 6(4), 506–514 (1959)
Huber, D.L. and R.L. Church: Transmission Corridor Location Modeling. Journal of Transportation Engineering-Asce, 111(2), 114–130 (1985)
Katoh, N., T. Ibaraki, and H. Mine: An efficient algorithm for k shortest simple paths. Networks, 12(4), 411–427 (1982)
Medrano, F.A. and R.L. Church: A New Parallel Algorithm to Solve the Near-Shortest-Path Problem on Raster Graphs. GeoTrans RP-01-12-01, UC Santa Barbara (2012)
Medrano, F.A. and R.L. Church: Transmission Corridor Location: Multi-Path Alternative Generation Using the K-Shortest Path Method. GeoTrans RP-01-11-01, UC Santa Barbara (2011)
Merrill, D., M. Garland, and A. Grimshaw: Scalable GPU graph traversal. In: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. ACM. (2012)
Raith, A. and M. Ehrgott: A comparison of solution strategies for biobjective shortest path problems. Computers & Operations Research, 36(4), 1299–1331 (2009)
Reif, J.H.: Depth-first search is inherently sequential. Information Processing Letters, 20(5), 229–234 (1985)
Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 30(3), 202–210 (2005)
Orden, A.: The transhipment problem. Management Science, 2(3), 276–285 (1956)
Yen, J.Y.: Finding the K Shortest Loopless Paths in a Network. Management Science, 17(11), 712–716 (1971)
Zeng, W. and R.L. Church: Finding shortest paths on real road networks: the case for A*. International Journal of Geographical Information Science, 23(4), 531–543 (2009)
Zhan, F. and C. Noon: A Comparison Between Label-Setting and Label-Correcting Algorithms for Computing One-to-One Shortest Paths. Journal of Geographic Information and Decision Analysis, 4(2), 1–11 (2000)

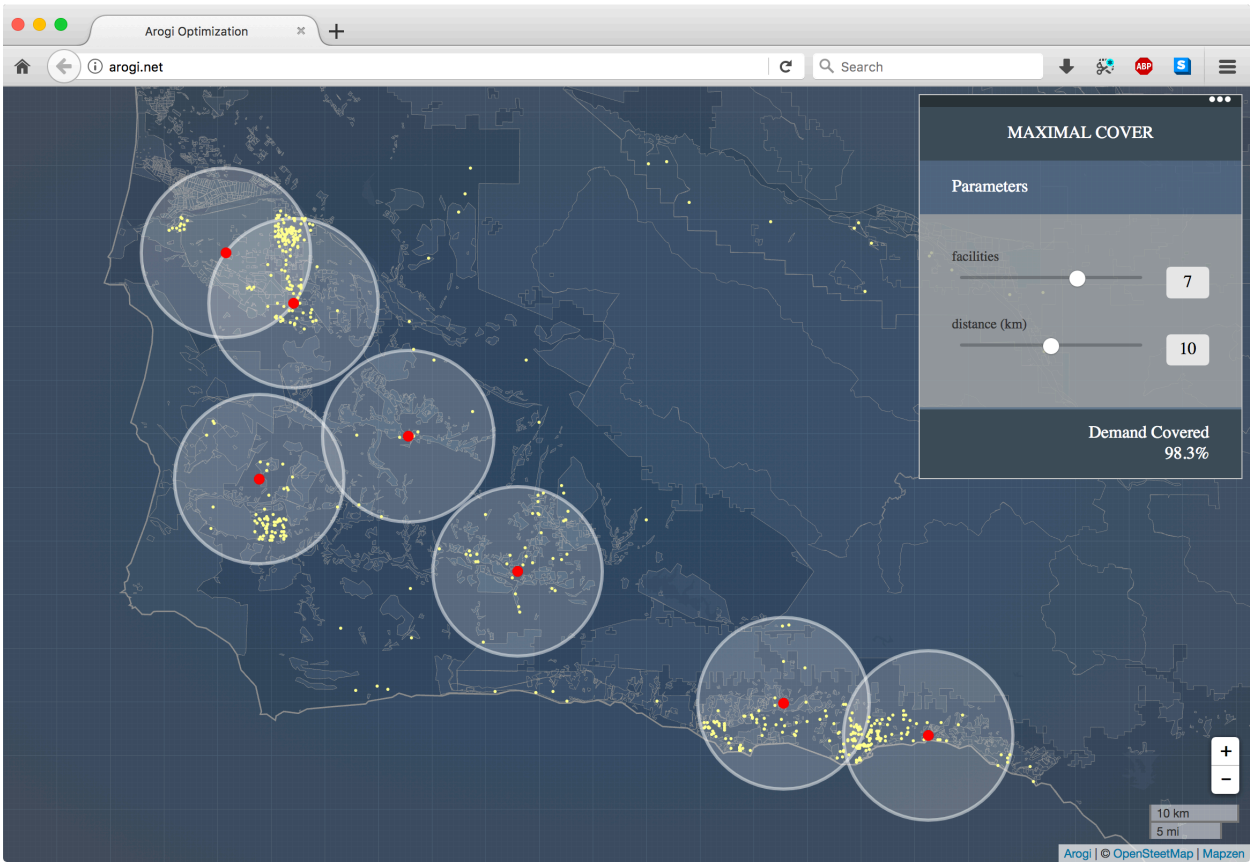
MAP AND VISUALIZATION PORTFOLIO — *F. Antonio Medrano*

Most of my spatial data visualizations have been created programmatically from scratch, as I my research has mostly centered around being more of a GIS toolmaker rather than a GIS tool user. The following visualizations were created to demonstrate certain properties of spatial data which could not be performed out-of-the box on common GIS software.

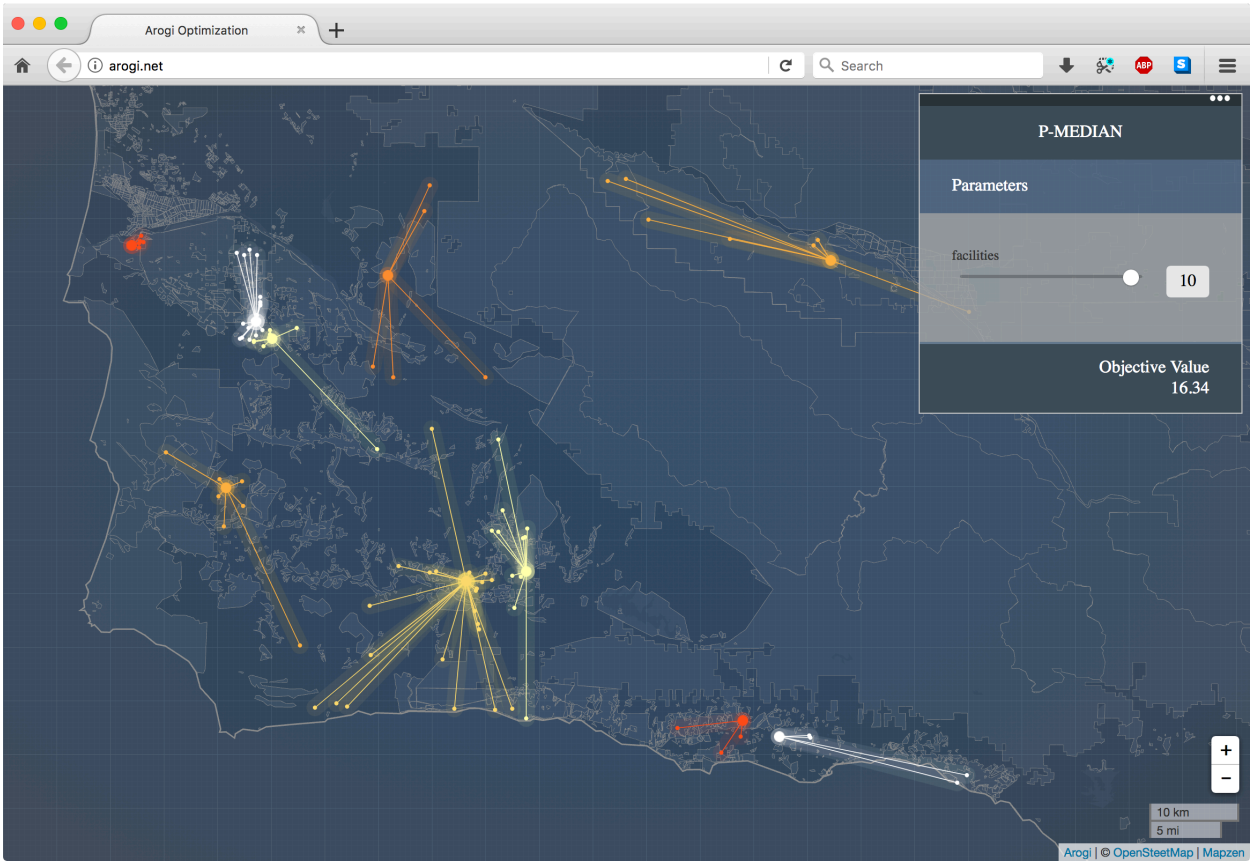
1. Online spatial optimization using open-source software.

The following visualizations were created as a part of the R&D for my start-up company's SBIR Phase I grant. They demonstrate, as a proof-of-concept, the ability to assemble a variety of open-source tools to solve complex spatial optimization problems. These visualizations used Tangram map rendering with a Leaflet map interface on the front-end; Mapzen Valhalla for road-network distances and routing, the OR-Tools mathematical programming solver from Google, and open source geospatial libraries GDAL/OGR, GEOS on the back-end; and open-data from OpenStreetMap. The front-end was coded in JavaScript and the back-end was coded in Python.

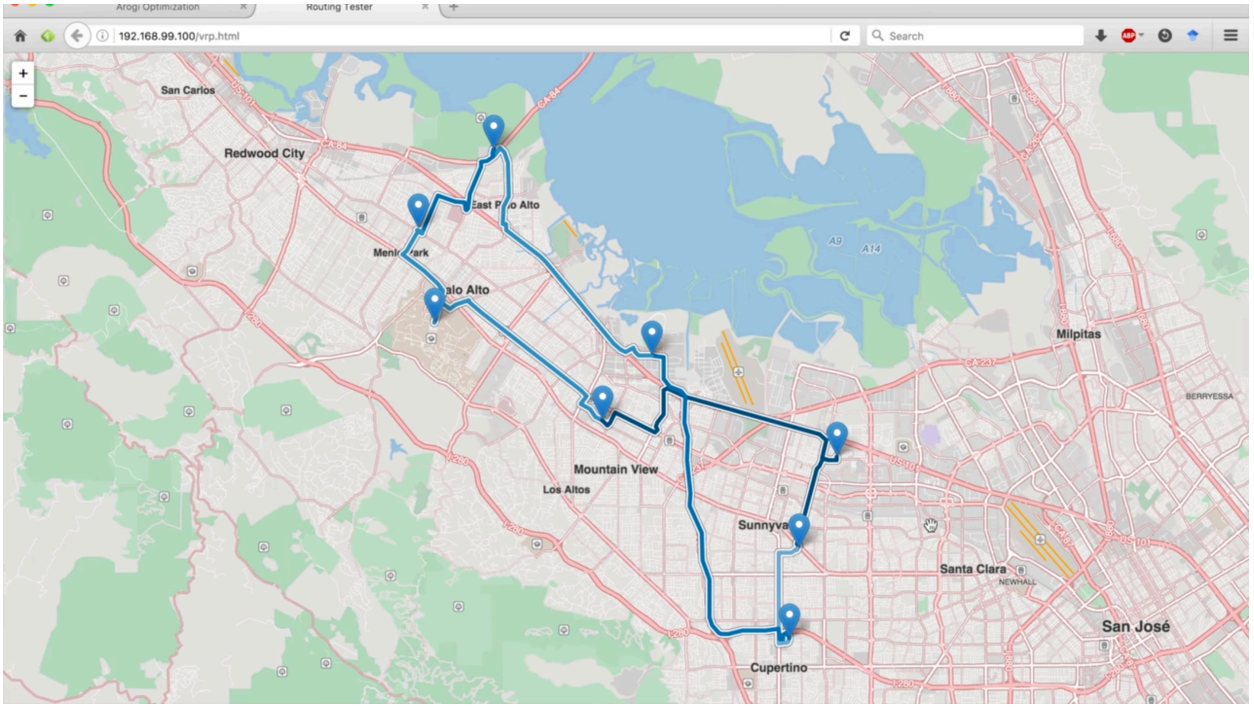
Solving a Maximum Cover Location Problem (MCLP)



Solving a P-Median Problem (PMP)

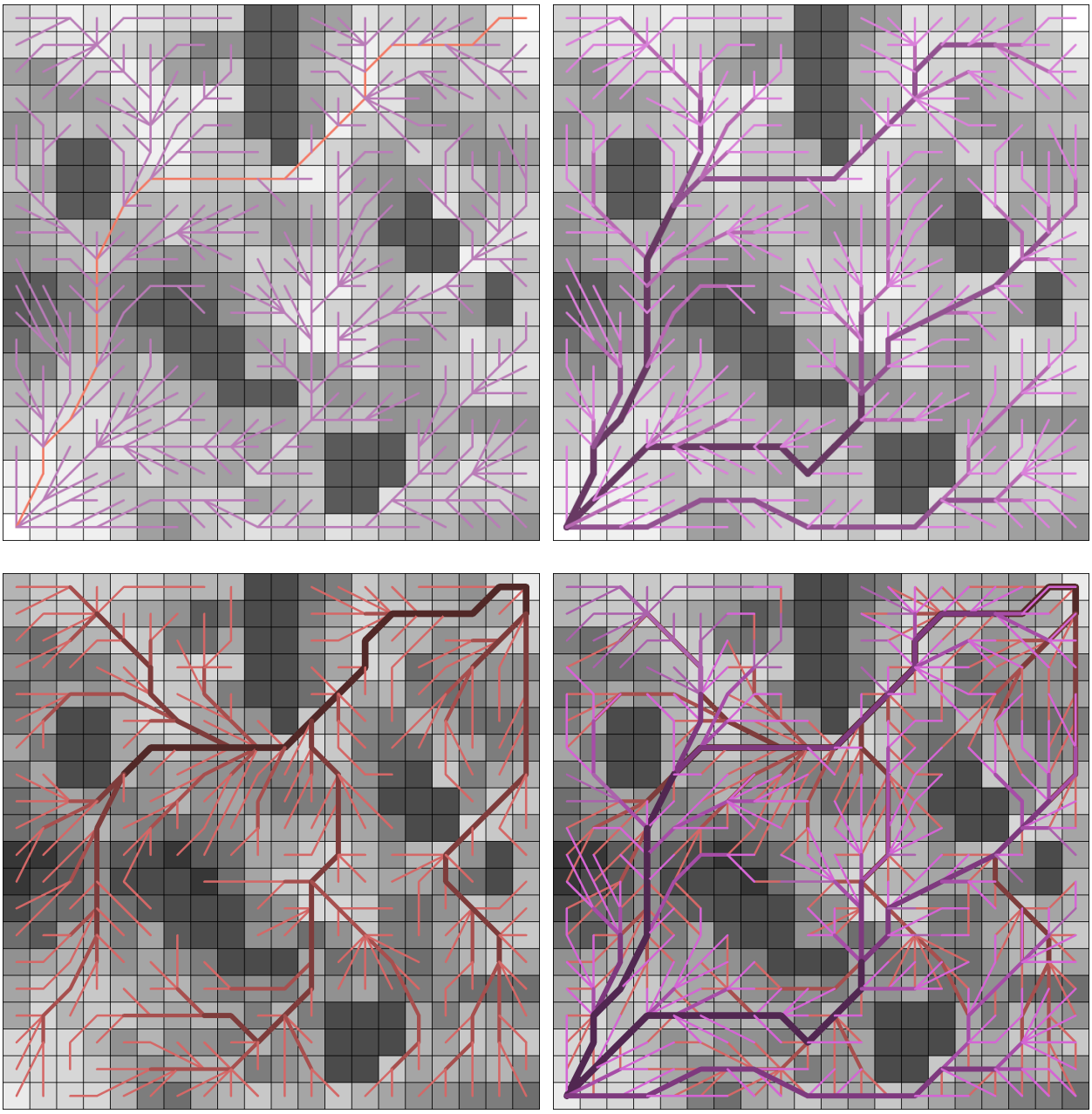


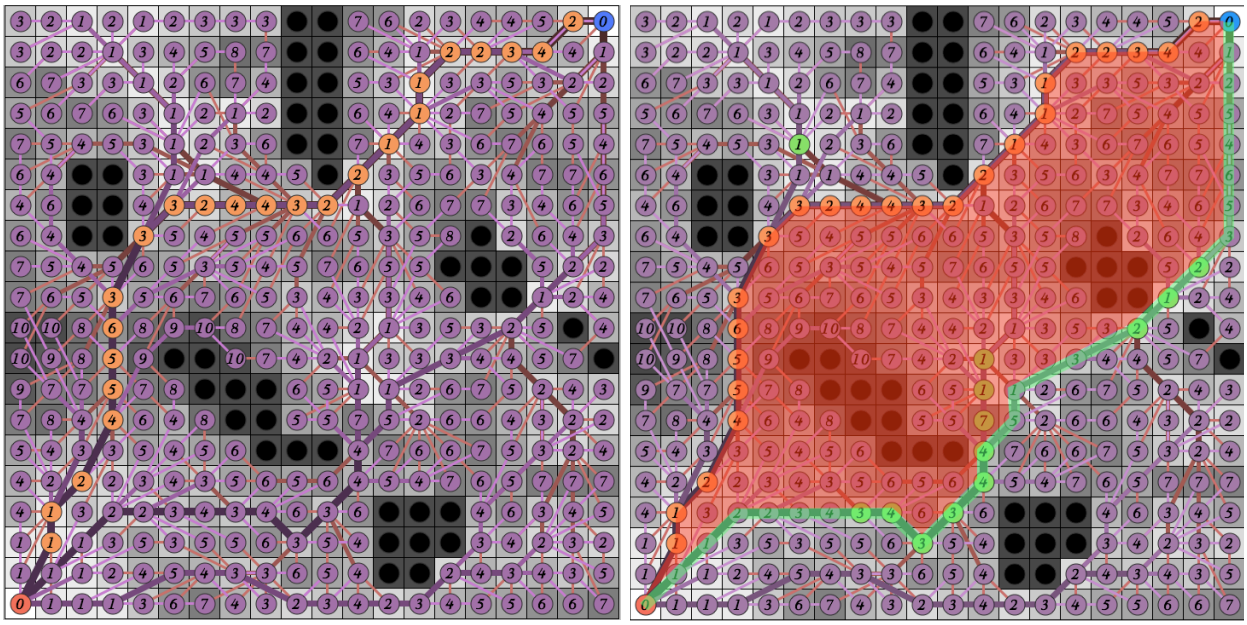
Solving a Vehicle Routing Problem (VRP) on the road network



2. Automated Near-Shortest Path Selection using Strahler Stream Order

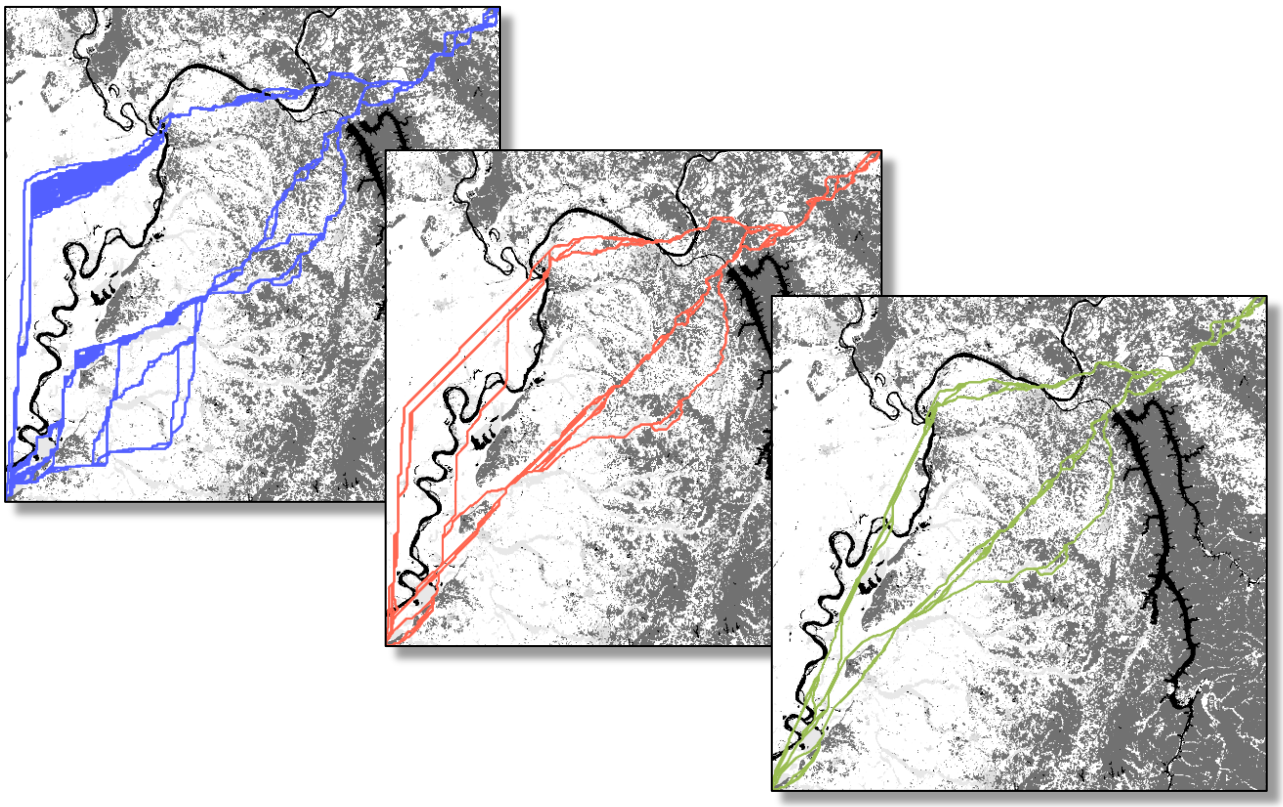
The following images were programmed in Java using the Processing library for visualizations to demonstrate the using Strahler stream ordering to give structure to the forward and reverse shortest path trees, then using the trees to generate quality alternative near-shortest paths. In the final image, the shaded region represents the area difference between the shortest path and the automatically-computed spatially-diverse near-shortest-path.





3. Demonstrating Distortions of Shortest Paths due to Raster Network Connectivity

The following images were programmed in Java and Processing to demonstrate the geometric distortions of shortest paths when generating networks from a raster using orthogonal, queen's move, and queen + knight's move connectivities. These visualizations show all supported non-dominated solutions to a bi-objective shortest path problem on a 1000x1000 raster network.



4. Showing that enumeration is not good for calculating alternate shortest paths

The following image was programmed in Java and Processing, showing the enumeration of all paths that are within 0.8% in length of the shortest path on an 80x80 raster network with queen's + knight's move connectivity using a Near Shortest Path Algorithm (Carlyle & Wood 2005). This contains of the overlay of 160,650,434,203 distinct paths, all within 0.8% in length of the shortest path (less than 1 percent!). While the rendering does not have much aesthetic, it is a stark visual demonstration that the combinatoric aspect of path enumeration prevents it from being an effective method for generating spatially diverse alternate shortest paths.