University of California Santa Barbara
Santa Barbara, California
June 2015

The Life Cycle Energy-Water Usage Efficiency of Artificial Groundwater Recharge Via the Reuse of Treated Wastewater

A Dissertation presented
by
Eric Daniel Fournier
to
The Bren School of Environmental Science & Management

in partial fulfillment of the requirements for the degree
of
Doctor of Philosophy

in the subject
of
Environmental Science & Management

This Dissertation is Approved
by

Committee Chair

Signature:_____ Date: _____
Dr. Arturo A. Keller

Committee Member

Signature: _____ Date: _____
Dr. James Frew

Committee Member

Signature: _____ Date: _____
Dr. Roland Geyer

This dissertation is dedicated to the memory of Julio Arnold Muniz.
He was my best friend.

Curriculum Vitae of Eric Daniel Fournier

June 2015

Education

Bachelor of Science in Environmental Science, Bucknell University, Lewisburg, PA, June 2008, (Magna Cum Laude, Phi Beta Kappa)

Master of Environmental Science, Yale School of Forestry and Environmental Science, New Haven, CT, June 2010

Master of Arts in Geography, University of California Santa Barbara, Santa Barbara, CA, March 2015

Employment

2009-2010: Teaching Assistant, Geographic Information Science, Yale School of Forestry and Environmental Science, New Haven, CT

2010-2013: Teaching Assistant, Biogeochemistry, Landscape Ecology, Geographic Information Science, University of California at Santa Barbara, Santa Barbara, CA

Publications

Automating the Preprocessing of Heterogeneous Spatial Data Inputs to Multi-Criteria Site Suitability Analyses, *Environmental Software and Modeling*, Submitted 2015.

MOGADOR revisited: improving a genetic approach to multi-objective corridor search, *Journal of Environmental Planning and Design* B, 2015.

Minimizing impacts of land use change on ecosystem services using multi-criteria heuristic analysis, *Journal of Environmental Management*, 2015.

Spectroscopic investigations of Fe^{2+} complexation on nontronite clay, *Langmuir*, 2007.

Fellowships

Walton Family Foundation Sustainable Water Markets Fellow, Toyota Motor Company Doctoral Fellow, Forestry Scholarship, McKenna Foundation Research Fellow

Specializations

GIS, LCA, Genetic Algorithms, Stochastic Processes, Combinatoric Optimization

The Life Cycle Energy-Water Usage Efficiency of Artificial Groundwater Recharge Via the Reuse of Treated Wastewater

Abstract

This dissertation investigates the dynamic energy-water usage efficiency of civil engineering projects involving the recharge of subsurface groundwater aquifers via the reuse of treated municipal wastewater. For this purpose, a three-component integrated assessment model has been developed. The first component uses a cartographic modeling technique known as Weighted Overlay Analysis (WOA) to determine the location and extent of sites that are suitable for the development of groundwater recharge basins given a regional geographic context. The second component uses a novel Genetic Algorithm (GA) to address the multi-objective spatial optimization problem associated with locating corridors for the support infrastructure required to physically transport water from the treatment facility to the recharge site. The third and final component takes data about the anticipated recharge treatment source location, reuse destination location, and proposed infrastructure corridor location and uses them to populate a spatially explicit Life Cycle Inventory (LCI) model capturing all of the process energy consumption and material inputs to the reuse system. Five case studies involving the planning of new basin scale artificial recharge systems within the state of California are presented and discussed.

Contents

Listing of figures

Acknowledgments

I WOULD LIKE TO THANK my father Paul, my mother Patricia, my brother Bryan, and my girlfriend Tessa for their continued support of my education. I would also like to thank my PhD Committee members, Dr. Arturo Keller, Dr. James Frew, and Dr. Roland Geyer for their respective contributions to the development and successful completion of this project. Finally, I would like to thank Dr. Charles Dana Tomlin for inspiring me to pursue a doctorate in this field of study through his work, teaching, eloquence, and integrity.

We have arranged a civilization in which most crucial elements profoundly depend upon science and technology.

Carl Sagan (1934-1996)

0

Introduction

0.1 Research Objectives

The overarching research objective of this dissertation can be stated as follows: develop an integrated modeling framework, incorporating data depicting local geographic context, that is capable of quantifying the life-cycle energy-water usage efficiency of proposed new infrastructural systems supporting the reuse of treated municipal wastewater for the purpose of artificial groundwater recharge. In order to achieve this goal the following three canonical problems were identified and addressed in sequential order:

1. Given multiple independent objectives – propose a systematic method for selecting sites that are suitable for the development of new artificial groundwater recharge infrastructure.

2. Given multiple independent objectives – propose a systematic method for routing optimal corridors for new water distribution infrastructure linking a designed treated effluent source to a designated recharge destination.

3. Given a fixed influent flow rate, a set of mean influent pollutant concentrations, a set of maximum effluent pollutant concentrations, and the topological structure of the effluent distribution network – propose a systematic method for determining the life-cycle energy-water usage efficiency of an integrated wastewater reuse system

The remainder of Chapter (0) provides an in-depth background discussion of the both the academic and social relevance of the stated research objective. Subsequent Chapters (1-3)

are organized with respect to the various independent research activities that were undertaken to address each of the three canonical problems listed above. The final Chapter (4) provides an in depth analysis of a set five case study implementations that were developed to assess the framework's efficacy in satisfying the stated research objective.

0.2 The Energy-Water Nexus

Nearly all modern industrialized societies rely upon energy generation technologies which are derivative of a thermodynamic process known as a heat engine. In a typical heat engine the chemical energy stored within a fuel source such as coal, petroleum, or natural gas, must be first be released as ambient thermal energy through the process of combustion. The heat engine then, by virtue of its design, converts this ambient thermal energy into mechanical energy for the purpose of performing some sort of meaningful work – i.e. generating electricity.

The history of the advancement of the human species is a story which can be cast in terms of the progressive discovery new, higher density chemical energy stores and their enhanced exploitation via the development of new, ever more sophisticated heat engines[25,26]. Interestingly however, is the fact that nowhere in our history has there ever occurred a single substantial deviation in the choice of the working fluid that actually performs the crucial energy conversion process within a heat engine: water. For all of the advances which have been made in terms of improved fuel processing, boiler and combustion chamber design, etc., water has and likely will continue to remain stubbornly positioned as a critical component of nearly all major commercial scale energy systems; and, in particular, those involved

with the production of electricity.

Another technological system which can also be viewed as a foundation pillar of modern industrialized societies is the mechanized disposal of human and animal wastes via engineered sewage conveyance and treatment processes[3]. Here again, the advancement of the human species might alternatively be cast in terms of the progressive improvement in the reliability and efficiency of these systems over time. And, what is more, just as the advancement of our energy system appears to be bounded by the physical properties of the water, so too has the advancement of wastewater management been similarly constrained.

For example, in terms of water treatment processes, the upper bound on process efficiency is determined by the Second Law of Thermodynamics. This law states that, for any closed system, there is a tendency for the entropy of that system to increase over time[32]. *Ceteris paribus* wastewater, a heterogeneous mixture, possesses higher entropy than pure water does. This means that any attempt to purify a polluted wastewater stream must necessarily incur the cost of some energy input to facilitate the requisite reduction in entropy.

In terms of water distribution processes, the key physical determinant of energy efficiency stems from the density of water. At 1000 kg/m^3, considerable energy must be expended whenever large quantities of water must be moved over a distance or lifted against an elevation gradient. What is more, many treatment processes utilize pressurized sieving techniques where water is physically pushed through a porous membrane in order to forcibly remove dissolved pollutant species. As a consequence of this practice, the energy inputs required to overcome the previously mentioned entropic gradient are often supplied for the immediate purpose of moving quantities of water from place to place.

The energy-water nexus is a term which has emerged from within the academic research

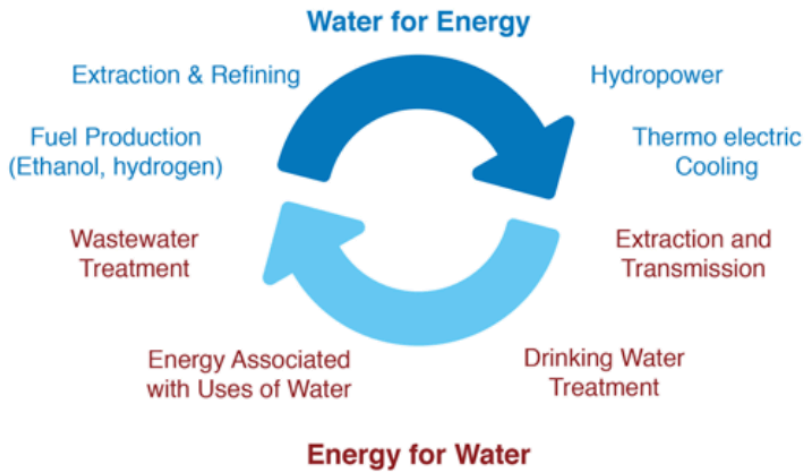


Figure 1: Perspectives on the Energy-Water Nexus [45]

community to describe these types of dynamic interrelationships which are inherent to our energy and water systems. There are two perspectives from which the energy-water nexus can be alternatively studied. The first emphasizes the *water for energy* dimension, and is generally concerned with the study of processes and technologies that are involved with the direct withdrawal and consumption of water for the production of both primary and final energy resources. The second of these perspectives, focuses alternatively on the *energy for water* dimension; investigating processes and technologies which consume energy for the purpose of transmitting or purifying freshwater resources.

Here in the United States, 50% of total annual freshwater withdrawals are used for the cooling of thermoelectric power plants. Alternatively, 4% of the nation's total energy consumption is dedicated to the transmission and purification of water and wastewater. While these national figures speak to the overall significance of this issue, the situation becomes more acute when one begins to consider different regional contexts.

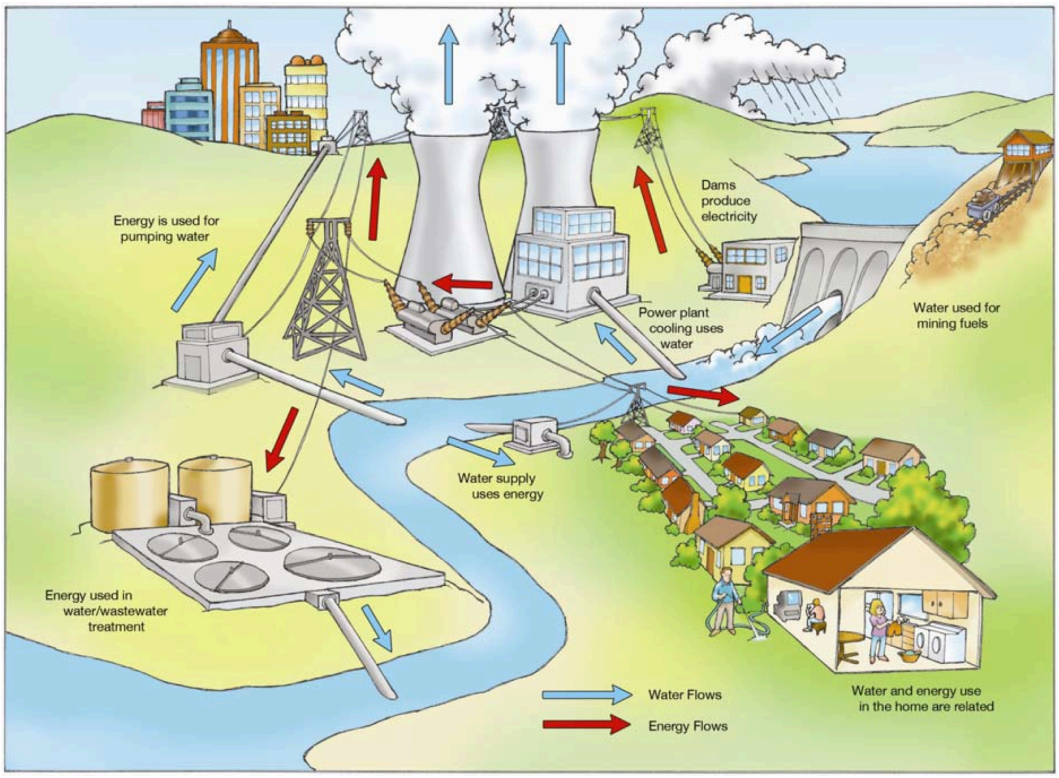


Figure 2: The dimensions of the energy-water nexus[45]

The criticality of the energy-water nexus becomes greatly exacerbated in those areas where either water is scarce, energy is scarce, or both. Unfortunately, the state of California suffers from both of these conditions to varying degrees, making the energy-water nexus a frequent source of regional interest within both the academic research and socio-political circles. For example, in a 2005 report published by the California Energy Commission (CEC) it was found that 19% of the electricity and 32% of the natural gas consumed within the entire state were used for purpose directly related to the supply and treatment of fresh-water resources.

0.3 Water Distribution Systems

One of the main drivers for of this tremendous energy consumption within the state of California is the large scale transfer of freshwater resources between physically distinct hydrologic basins. California is crossed latitudinally by a massive network of interconnected hydraulic engineering projects including pipelines, aqueducts, reservoirs, and pump stations. These systems, which have been funded by a mixture of Federal, State, and Local agencies, were designed to reconcile discontinuities between the spatial and temporal distributions of the supply and demand for freshwater resources within the state.

Inter-basin transfers typically involve the movement of water against a considerable elevation gradient. Due to water's previously mentioned high density, there are substantial energetic costs associated with operating the infrastructure required to facilitate these transfers. For example, the bar graph to the right of Figure 4 compares the energy intensity of several different sources of municipal water within the state of California. According to this research water resources which are supplied via inter-basin transfer, either through branches of the State Water Project or through the Colorado River Aqueduct, rank very poorly in terms of energy usage efficiency relative to a number of other water supply systems.

0.4 Wastewater Treatment

The term *wastewater treatment* refers to a set of physical and chemical processes that have been individually designed and are collectively composed for the purpose of removing a suite of physical, chemical, and/or biological contaminants from a quantity of water. The operational goal of a wastewater treatment process is typically defined in terms of a set of

Figure 3: The geographic extent of water distribution infrastructure in the state of California [51]

desired effluent pollutant concentrations that are sufficiently low to enable that effluent to be used for a specific end-use application. Crucially implicit in this definition therefore, is the notion that the precise combination and configuration of the individual, atomic treat-

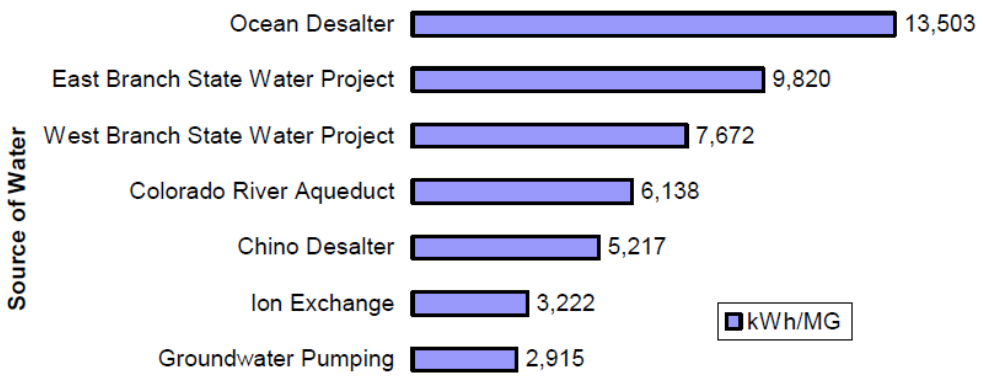


Figure 4: The geographic extent of water distribution infrastructure in the state California [33]

ment processes, will vary on the basis of the purity requirements associated with the anticipated end-use application. Figure 5 provides a high level view of the separate treatment processes that are typical to modern wastewater treatment facilities; assigning them to a somewhat informal hierarchy consisting of: primary treatment, secondary treatment, and tertiary treatment.

The collection of processes described in Figure 5 represent the required suite of treatment operations necessary to produce effluent water that is suitable for groundwater recharge applications (also known as: indirect potable reuse) given a normal stream of influent municipal wastewater. Typically, wastewater treatment plants (WWTPs) – normal in the sense that they are producing treated effluent from generic municipal sources for release into natural surrounding environs – are only required to provide up to a secondary treatment level. This level of treatment, on the diagram provided in Figure 5, corresponds to the fourth row of processes from the top. As the figure shows, providing treated effluent for reuse applications necessitates the implementation of as many a seven additional layers of tertiary treat-

ment in order to manage issues such as suspended colloids, disinfection, and phosphorous removal.

This hierarchy of treatment processes depicted in 5 is only meant to be representative of the types of processes that are typically required for reuse applications. In reality however, neither the rigid segregation of treatment processes nor the association of each treatment level with a set of designated end-uses have been precisely codified into a coherent regulatory framework at the Federal level. In general, primary and secondary treatment are Federally mandated as part of the generic WWTP under provisions of the Clean Water Act. However, tertiary treatment processes and requirements for their application in the context of different desired end-use applications, are, at present, regulated at the state or local levels in a more ad-hoc manner. Figure 6 attempts to illustrate the complexity of this landscape by mapping treatment levels to different end-use types, with annotations where additional restrictions may apply.

0.5 Wastewater Recycling and Reuse

Over the past ten years the reuse of treated wastewater has emerged as the fastest growing source of new water supply for municipal water districts in the Western United States. There are a variety of reasons for this growth trend. For example, some districts enjoy the degree of self determinism that ownership of a reuse system affords them; particularly in cases where they are beholden to a the whims of a third party water wholesalers or are junior water right holders within their basin. In general however, the primary driving force behind the increased interest in reuse has been down to the fact that treated municipal wastewater is perceived to be an efficient means of new water supply for a number of low

quality end use cases.

If one looks at existing reuse systems, these assumptions regarding system efficiencies appear to be valid. This is because the vast majority of reuse systems which have been implemented to date have taken advantage of favorable circumstances such as where the destination location for the treated effluent is positioned in fairly close proximity to the WWTP. An illustrative example of such a situation might be the use of wastewater which had been subjected to basic secondary treatment for the irrigation of a nearby cemetery or golf course. These types of reuse systems can be thought of as *low hanging fruit* because the only additional energy process based energy expenditures associated with their operation come from whatever tertiary treatment processes must be added into the WWTP operations to meet the effluent purity requirements associated with the designated end-use.

More recently however, many municipalities have begun to actively investigate the possibility of expanding both the scale and extent of their reuse operations: capturing larger fractions of their WWTP plants' throughput and distributing the treated effluent to a more diverse portfolio of end-use recipients. Among these proposed new end-use applications, perhaps the most attractive has been for artificial groundwater recharge.

An artificial groundwater recharge operation is one which takes some stream of input water, it need to necessarily be treated wastewater, and either passively or forcefully introduces it to the subsurface aquifer. The goal of this process is typically undertaken to remedy a condition of overdraft wherein the aquifer has been subjected to increased rates of withdrawals, decreases rates of natural infiltration, or both. Typically artificial groundwater recharge systems adopt one of two approaches in terms of the mechanism by which they physically deliver the water back to the subsurface. The first approach is passive – in-

volving the construction of one or more large scale infiltration basins into which water is pumped and then allowed to percolate into the subsurface through a porous media under the force of gravity. The second approach is active – involving the construction of one or more pump wells through which water is forced into the subsurface under the action of a mechanized pump.

Due to the considerable cost associated with pump operation as well as the limited recharge capacity associated with each individual wellhead, recharge basins have, up until this point, proven to be far and away the more attractive of the two options in terms of existing facilities. The primary exemption to this rule being locations where the reason that the recharge operations have been undertaken is create a target increase in subsurface hydrostatic pressure for the purpose of mitigating the encroachment of brackish water into the aquifer – which can often occur in coastal regions – or for the purpose of halting or redirecting the movement of some subsurface groundwater pollutant plume.

Despite the fact that artificial recharge basins take advantage of the force of gravity to introduce the water back into the subsurface they can still be expected to be associated with fairly large operational energy requirements. This is because, for a host of practical engineering as well as political and economic constraints, recharge basins tend not to be constructed as close to the source of their supply water as one might initially think. As a result, considerable amounts of energy must be continually invested to pump water, often against an elevation gradient, from its source of production to the recharge basin where it will ultimately be consumed for the purpose of artificial groundwater recharge.

0.6 The Energy-Water Feedback Loop

The suggestion that all forms of artificial groundwater recharge are likely to be associated with non-trivial process based energy demands leads to an interesting question regarding the net life-cycle energy utilization efficiency of reuse systems. This question stems from an understanding of the water usage intensity of various electricity generation technologies can vary significantly and is often much high than one mights initially expect.

For example, Figure 7 plots the water usage intensity – measured in terms of units of water consumed per unit of electric power produced – for a suite of electricity generation technologies that make up a substantial portion of the municipal energy generation portfolio here in the United States. As Figure 7 illustrates, depending upon the details of how electricity is produced in a given region, artificial groundwater recharge operations that utilize heavily treated municipal wastewater as their source feedstock have the potential to be associated with significant process energy related water consumption levels at the point of electricity production.

In light of the presence of this significant feedback loop, the hypothesis that this dissertation has been designed to test is whether or not there exists a specific set of circumstances in which it may be possible for municipal wastewater reuse systems involving a substantial artificial groundwater recharge components to produce a situation where they are saving some water locally, but at the cost of consuming more water regionally. This hypothesized negative net water savings, at the regional scale, would therefore be the result of water consumption which is embedded in the energy that must be imported to facilitate the reuse and recharge operations.

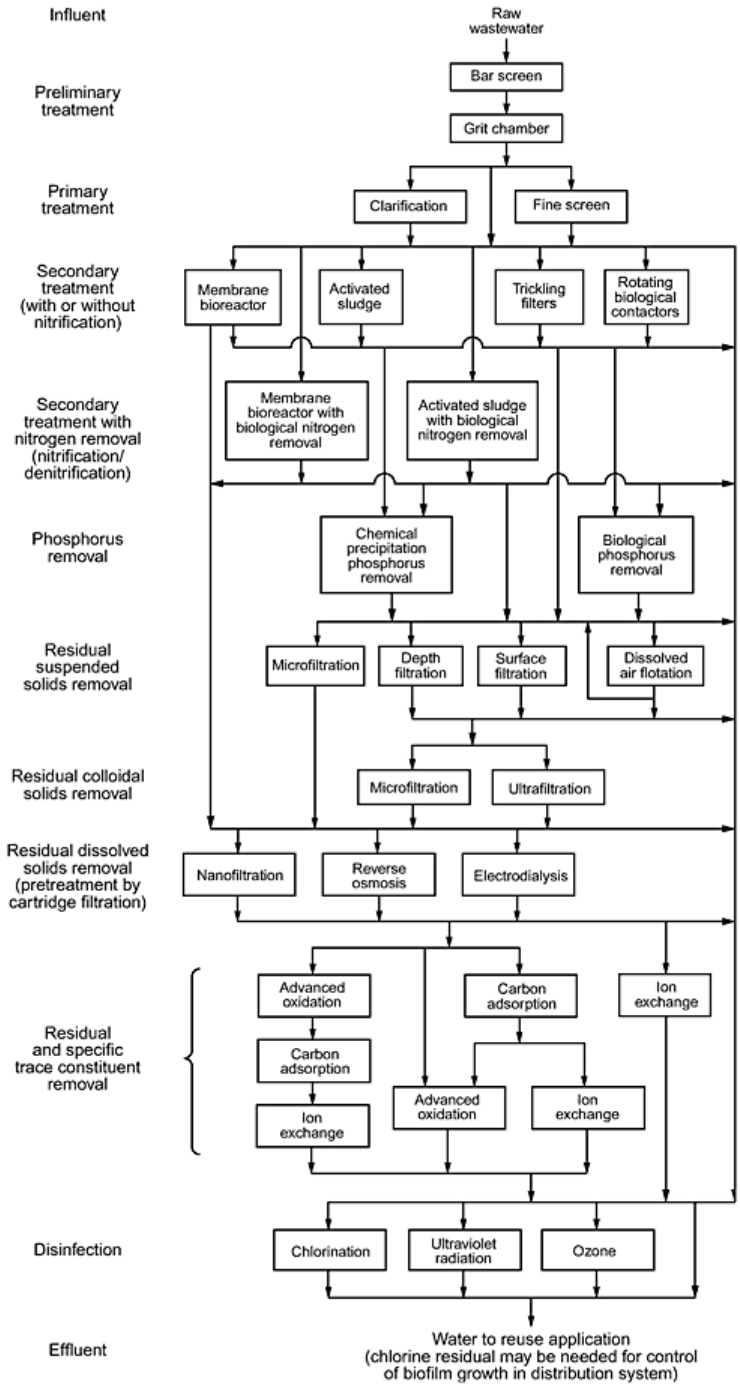


Figure 5: Process flow diagram of various wastewater treatment methods [4]

Types of Use	*Treatment Level*		
	Disinfected Tertiary	*Disinfected Secondary*	*Undisinfected Secondary*
Urban Uses and Landscape Irrigation			
Fire protection	☑		
Toilet & urinal flushing	☑		
Irrigation of parks, schoolyards, residential landscaping	☑		
Irrigation of cemeteries, highway landscaping		☑	
Irrigation of nurseries		☑	
Landscape impoundment	☑	☑*	
Agricultural Irrigation			
Pasture for milk animals		☑	
Fodder and fiber crops			☑
Orchards (no contact between fruit and recycled water)			☑
Vineyards (no contact between fruit and recycled water)			☑
Non-food bearing trees			☑
Food crops eaten after processing		☑	
Food crops eaten raw	☑		
Commercial/Industrial			
Cooling & air conditioning - w/cooling towers	☑	☑*	
Structural fire fighting	☑		
Commercial car washes	☑		
Commercial laundries	☑		
Artificial snow making	☑		
Soil compaction, concrete mixing		☑	
Environmental and Other Uses			
Recreational ponds with body contact (swimming)	☑		
Wildlife habitat/wetland		☑	
Aquaculture	☑	☑*	
Groundwater Recharge			
Seawater intrusion barrier	☑*		
Replenishment of potable aquifers	☑*		

**Restrictions may apply*

Figure 6: Appropriate End Use Categories for Different Levels of Treated Wastewater [33]

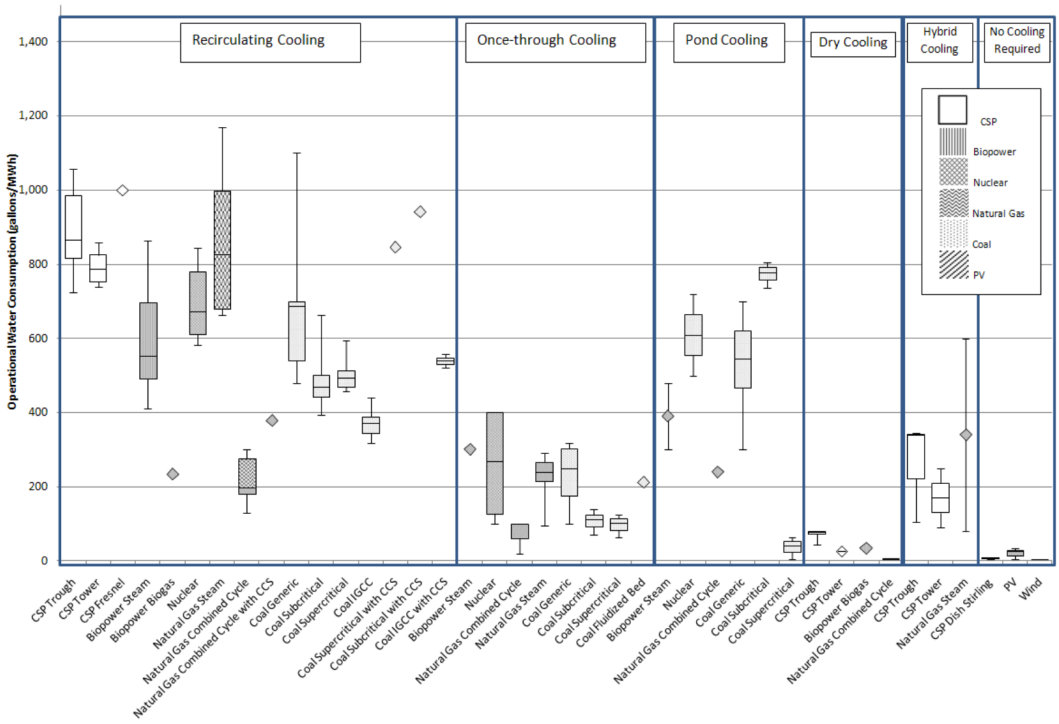


Figure 7: The water consumption intensity, measured in terms of water consumed per unit of electric power produced, for various electricity generation technologies [6]

Do not lose your faith – for a mighty fortress is our mathematics. It shall rise to the occasion. It always has.

Stanislaw Ulam (1909-1984)

1

Selecting Suitable Sites

1.1 Selecting Suitable Sites for Artificial Groundwater Recharge Basins

The first principle challenge which must be overcome in an attempt to quantify the life-cycle energy water usage efficiency of a hypothetical water reuse system involving artificial groundwater recharge is the need to develop of a systematic method for selecting sites that are suitable for the construction of the requisite recharge basins. This problem of *selecting suitable sites*, relative to one or more criteria of suitability, is one which is extremely common within the domain of Geographic Information Science (GIScience). And indeed, the need to develop consistent methodologies and capable tools for achieving this purpose were among the core research goals which initially stimulated the early development of the field.

1.2 Multi-Criteria Site Suitability Analyses

One methodology which has emerged as a reliable means of approaching this type problem, and the one which was adopted for the purposes of this dissertation, is a procedure known as multi-criteria site suitability (MCSS) analysis. MCSS analyses mathematically combine two or more input geographic data layers that each correspond to some independent measure of site suitability for a given landuse application[3] . The output of this MCSS computation is a single geographic data layer in which the value at each location represents a composite measure of overall site suitability relative to all of the independent criteria, simultaneously[28,18].

MCSS analyses are typically conducted using geographic information that has been

stored in a continuous raster format. This means that prior to conducting this type of analysis each of the input geographic data layers that are to be used must be preprocessed relative to some reference raster format so as to ensure the feasibility and consistency of the MCSS computation. For example, in order for the computation to be feasible: all of the input data layers must have the same number of cells and occupy that same geographic extent. Similarly, in order for the computation to be consistent: the ordinality and the scaling of the values in each raster must accurately reflect the relative weighting and directionality of each independent suitability criterium.

Geographic information system (GIS) software packages are commonly used to conduct both these types of data preprocessing operations as well as the MCSS computation itself[?,?]. This is because they provide pre-built functions which facilitate the import of spatial data layers from disparate sources as well as the manipulation of geographic data layers such that they satisfy the previously mentioned feasibility and consistency constraints. Often, the MCSS analyses itself is not the endpoint goal of a given research effort however. Many times, MCSS analyses are used as inputs to some other, more complex, numerical optimization model[14]. This situation is frequently encountered in the fields of operations research (OR) and location science (LS) where MCSS outputs are used to derive network topologies or linear programming constraints for optimization problems related to vehicle routing, flow maximization, or facility location[30,15].

1.3 The MCSS Data Preprocessing Workflow

The generic data preprocessing workflow for MCSS analysis involves one or more of the three phases illustrated conceptually in Figure 1.1. First, all of the input spatial data layers

must be checked to determine whether or not they possess the same coordinate system. If they do not, a single reference coordinate system must be chosen by the analyst to function as the standard for all of the input layers. The choice of this reference coordinate system is generally driven by the scale of spatial domain involved as well as the desired tradeoff between distance versus areal measurement error. The output of this *Reproject* operation is a set of duplicate data layers that are all projected as the reference coordinate system.

The second phase of the workflow involves clipping the various input data layers on the basis of their overlap with some reference spatial extent. This reference extent may correspond to the boundaries of a single input layer or may be arbitrarily designated by the analyst. The output of this *Clip* operation is a set of duplicate data layers that all have the same spatial extent as the reference extent.

The third and final phase of the workflow involves rasterizing all of the input layers such that they have the same cell size and cell alignment. During this phase, input data layers that are stored using a different geographic data model must be algorithmically converted into a raster based representation. As part of this algorithmic conversion process, a reference cell size, usually corresponding to the largest cell size contained within the input data layers, is used as a reference. The output of this *Rasterize* operation is a set of duplicate layers that all share the same cell size and cell alignment as the designated reference.

1.4 The MCSS Computational Workflow

Instead of executing the MCSS data preprocessing steps manually for each of the five case study regions that were going to be investigated as part of this dissertation, the decision was made to implement a more generic data preprocessing software framework which enable

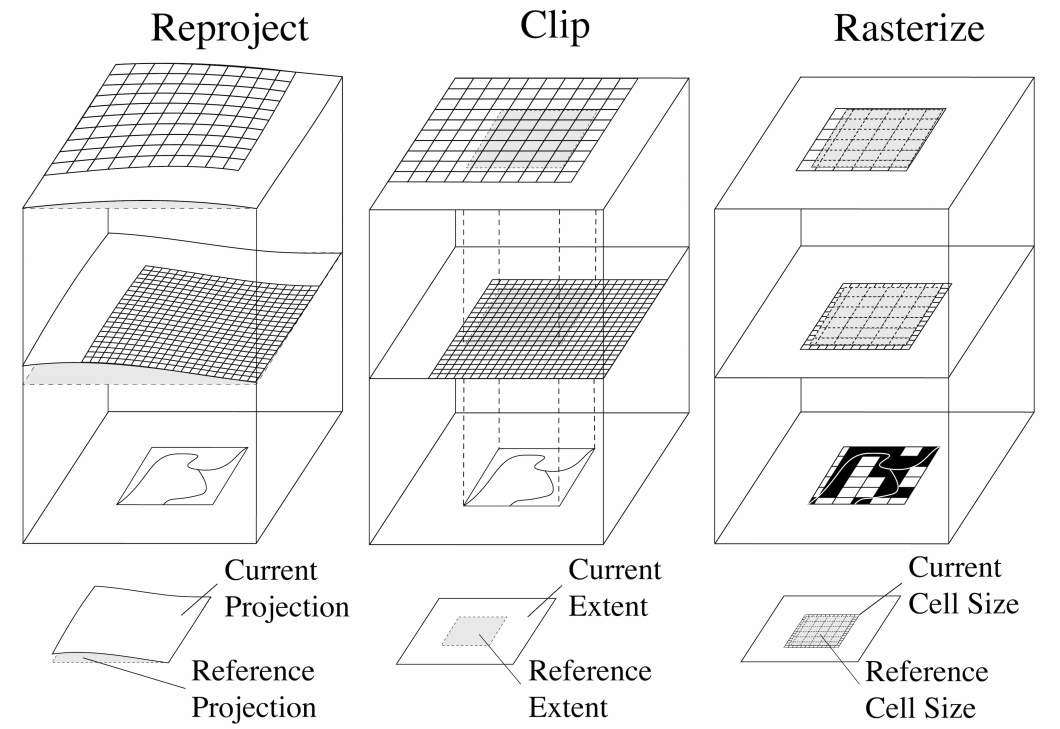


Figure 1.1: Conceptual illustration of the input data preprocessing operations and workflow phases typical to most MCSS analyses

any researcher – not necessarily the author of this dissertation – to conduct a similar analysis for any region in which sufficient input data was available. These tools, which shall be described in subsequent sections, were implemented in the the MATLAB® programming environment and have been made publicly available as a source code repository hosted at: https://github.com/ericdfournier/WOSS. The control flow logic which guided their development is described in the pseudocode in Figure 1.2.

Prior to the initiation of any MCSS analysis the researcher must provide a set of raw spatial data input files and designate a set of spatial reference criteria. Once these requirements are met, all of the input spatial data files are then subjected to a sequence of conditional

statements. Depending upon the result of these conditional statement evaluations various transformation functions are then sequentially applied to the input data files so as to produce a set of outputs whose projection, spatial extent, cell size, and cell alignment all match a set of designated spatial reference criteria.

```
1:  procedure PREPROCESS INPUT DATA
2:      for all input do
3:          if input_prj ≠ reference_prj then
4:              input = Reproject(input)
5:          else if input_ext ≠ reference_ext then
6:              input = Clip(input)
7:          else if isRaster(input_type) ≠ True then
8:              input = Rasterize(input)
9:          else if input_cs ≠ reference_cs then
10:             input = Resize(input)
11:         end if
12:         output = input
13:     end for
14:     return output
15: end procedure
```

Figure 1.2: Control logic for MCSS input data preprocessing workflow.

The current version of the toolset only supports the reprojection of input spatial data layers that are represented using geographic coordinates – i.e. data stored in latitude & longitude coordinate space). This constraint not only limits the directionality of the reprojection operation but also greatly simplifies the spatial interpolation routines required for the rasterization process. The authors plan to lift this restriction in future versions of the toolset as the MATLAB® language's native support for forward map projection as well as the automated parsing of standard formatted spatial reference data strings improves.

Following the preprocessing of the spatial data inputs, the next phase of the MCSS mod-

eling process is the user guided reclassification of the data values in each layer into a quantitative measure of suitability for the land use application in question. Routines are provided in the toolset to facilitate this process. Among these are an automated histogram equalization based reclassification procedure which assigns suitability values ensuring an even distribution of all the values contained within some range across all of the areas within the spatial data layer. Other tools allow the user to manually specify the range of the bins used for the reclassification of raw input data values to site suitability rankings.

1.5 The Software Toolset Repository Architecture

The WOSS toolset is publicly available via the GitHub repository hosted at: `https://github.com/ericdfournier/WOSS`. The directory structure of this repository is illustrated in Figure 1.3. Below the top level root directory are four standalone files: (1) a *LICENSE.md* file, (2) a *README.md* file, (3) a *GUI.m* file and (4) *GUI.fig* file. The first two files contain the software license and general repository usage guidance, respectively. The third and fourth, are the source code files supporting the standalone graphical user interface that visually guides a user through this same data preprocessing workflow.

Also below the top level root directory are the following three subdirectories: (1) *src/*, (2) *smp/*, (3) *output/*. The *src/* directory contains the MATLAB® source code m-files comprising the toolset's various functions. The *smp/* directory contains MATLAB® .m-file scripts that can optionally be called to automate the execution of multiple data preprocessing workflows. The */smp/data/* directory contains two sub-directories: *vector/* and *raster/*. Each of these houses the corresponding sub-directories, one for each vector and raster based raw input spatial data files provided by the user. The tiles used for each of these **(filename)*

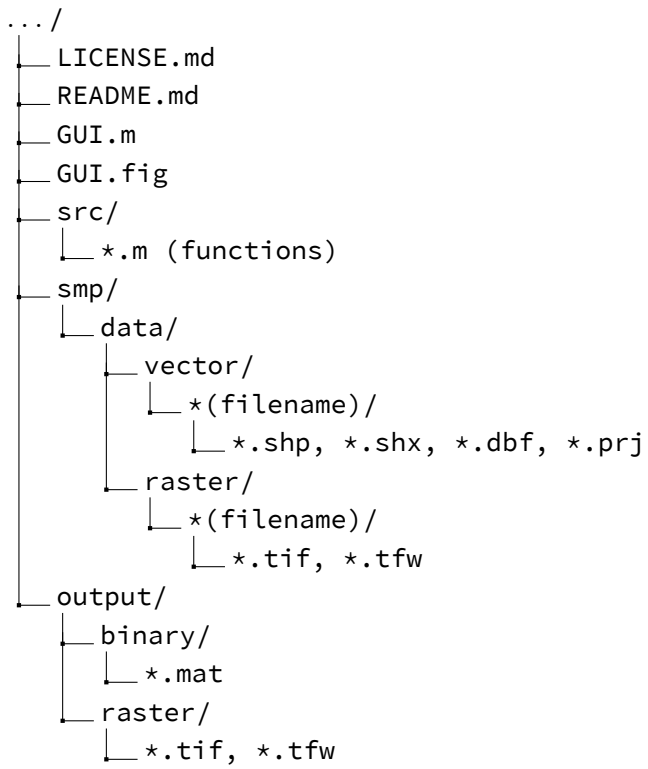


Figure 1.3: Directory tree structure for the toolset repository. Filetypes required for input data and automatically generated as output data are shown.

sub-directories are automatically assigned to the outputs generated by the tool. The supported vector input filetype is the ESRI shapefile format. Alternatively, the supported raster input file type is the open source GeoTiff format. Finally, the *output/* directory contains two sub-directories: *binary/* and *raster/*. These subdirectories comprise the default destination locations for all of the outputs generated by the toolset tools. Outputs can be produced in either the mat-file MATLAB® ASCII-binary format or in the same GeoTiff format as the input raster data.

The toolset supports the use of composite raster data sets which are made up of multiple, possibly overlapping, individual raster data tiles. It does this by performing a bounding box

intersection test for each input raster data tile with the reference spatial domain. For those tiles whose bounding boxes are found to intersect that of the reference domain, values are iteratively compiled into a new composite mosaic data layer made up of, potentially several, individual tiles. This makes it possible to use input raster data layers that are of arbitrarily high resolution covering large geographic domains.

1.6 An Example Implementation

The raw input datasets which were selected for the example implementation were collected from several publicly available sources. A brief topical description of each source as well as a link to its source web repository is given in Figure 1.4. In addition to these raw input data sources, a number of derived data products are generated automatically from the digital elevation model (DEM) for use in this particular case study analysis. These derived products include: slope & aspect.

1.7 The Geographic Unit of Analysis

The geographic unit of analysis selected for this example implementation is the US Geologic Survey (USGS) Hydrologic Unit Code (HUC) level five watershed. Specifically, the level five watershed areas contained within the administrative boundaries of the state of California. According to the USGS:

> *The United States is divided and sub-divided into successively smaller hydrologic units which are classified into four levels: regions, sub-regions, accounting units, and cataloging units. The hydrologic units are arranged or nested within each other, from the largest geographic area (regions) to the smallest geographic*

Type	Category	Source
Vector	Resource Areas	Cal-Atlas
Vector	County Boundaries	Cal-Atlas
Vector	Surface Geology	Cal-Atlas
Vector	Road Network	Cal-Atlas
Vector	STATSGO Soils	USGS
Vector	State Park Boundaries	Cal-Atlas
Vector	Stream Reaches	National Map
Vector	Street Network	Cal-Atlas
Vector	Surface Water Storage	Cal-Atlas
Raster	Crop Data Layer	USDA
Raster	Digital Elevation Model	National Map
Raster	NLCD Landcover	National Map

Figure 1.4: Table of input data sources used in the case study MCSS model for artificial groundwater recharge applications.

> *area (cataloging units). Each hydrologic unit is identified by a unique HUC consisting of two to twelve digits based on the levels of classification in the hydrologic unit system.*[52]

The level five designation within this HUC framework is comprised of closed contiguous regions possessing an average area of 588 square kilometers. These level five HUC designated areas are often referred to as the HUC-10 watersheds because of their use of a ten digit unique numerical identification code. Within the state of California, there are 1,040 individual HUC-10 watersheds. These watersheds are non-overlapping and have been derived algorithmically from the national elevation dataset by USGS scientists according to the method described by[52].

1.8 The WOSS Model Outputs

Figure 1.5 illustrates a set of sample outputs that were generated by the data preprocessing components of the toolset for an example HUC-10 reference boundary. The reference boundary can either be selected manually, by calling a function which prompts the user to click on map with all of the HUC-10 boundaries drawn on it, or automatically, by specifying the 10-digit code corresponding to the desired HUC-10 watershed. The toolset processes generate an output layer stack – the individual component layers of which are illustrated in the colored inset map panels. Only layers for which there is at least one non-empty data value are included in the generated outputs. Thus, the number of output components may vary depending upon the coverage of the input data layers relative to the domain of the reference boundary. For vector data inputs, the values which are contained in the output are those corresponding to a single attribute field selected by the user. This field must be either a real or coded numeric data type as the raster data format does not support the native representation of categorical variables.

The toolset structure allows for the automated repetition of this data preprocessing workflow for a large number of reference boundaries. In this case study implementation, for example, the toolset was used to prepare a single such output layers stack for each of the 1,040 individual HUC-10 reference boundaries contained within the state of California. With these outputs, a corresponding the MCSS analysis could then be easily conducted for any or every such HUC-10 watershed in the State.

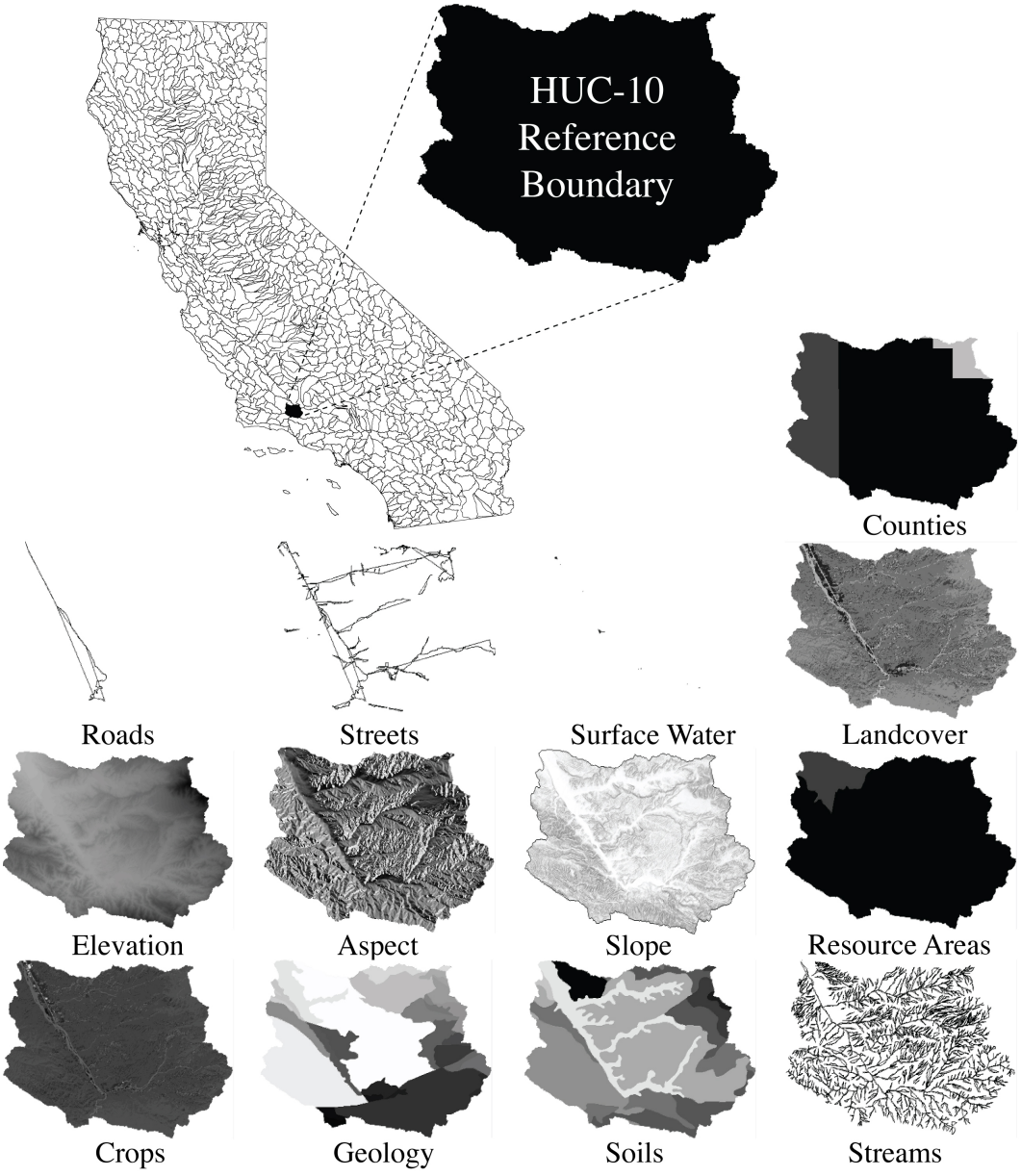


Figure 1.5: Graphical Illustration of the WOSS Model Outputs for Reference HUC-10 Boundary

[A computer] takes these very simple-minded instructions – 'go fetch a number, add it to this number, put the result there, perceive if it's greater than this other number' – but executes them at a rate of, let's say, 1,000,000 per second. At 1,000,000 per second, the results are indistinguishable from magic.

Steve Jobs (1955-2011)

2

Locating Optimal Corridors

2.1 Locating Optimal Corridors for Water Distribution Infrastructure

The source of location for the delivery of treated wastewater for artificial groundwater recharge applications is always the Wastewater Treatment Plant (WWTP). WWTPs are overwhelming located at low elevation points within the hydrological basins that they serve; as this feature ensures a minimum energy input requirement to deliver wastewater from its distributed sites of generation (i.e. the many homes and businesses distributed geographically throughout the basin) to a single centralized point of treatment. In this way, a system design based upon a single centralized WWTP is best able to take advantage of the use of gravity to provide the motive force for the wastewater's journey through the system. As a side note, an question which may become of substantial future interest is how a more prominent role of reuse may alter our thinking about the optimal design of water treatment collection and conveyance infrastructure. Specifically, it remains an open research question as to whether or not the existing paradigm, with large centrally location treatment facilities, would persist as the most favorable solution if one were charged with designing an integrated wastewater treatment and reuse system from the oft.

The salient output of the first model component - WOSS - is the designation of one or more suitable sites for the placement of either a gravity fed surface spreading infiltration basin or a pump driven subsurface recharge well. In this way, we are left with one or many (in the case of multiple suitable sites) combinations of point source and destination locations. The next modeling challenge which must be overcome therefore is the development of a scheme for generating plausible pathways which connect the source locations to the various destinations. The implicit assumption here being, that in order to deliver the

treated wastewater from its source of production, the WWTP, to the end point of use, the recharge site, new water conveyance infrastructure must be constructed.

At this stage it is important to pause to consider whether or not the construction of new water conveyance infrastructure is in fact a necessary condition of all or any water reuse projects. There are two perspectives from which a response to this objection can be framed. The first is technical in nature. If no new conveyance structure is implemented, two fundamental technical requirements must be met. The first is that the point of reuse must be situated favorably with respect to the existing potable water conveyance system such that it can be readily be connected to, and withdraw from it, large quantities of water for recharge purposes. The second is that the treated wastewater which is to be reused must be returned to a sufficiently high standard of quality such that it can be reincorporated directly back into the potable water supply. This second constraint, while technically feasible given sufficient financial resources, leads naturally to the other category of potential objections to the development of a reuse project without the addition of new conveyance infrastructure: namely, the social stigma associated with co-mingling so-called reclaimed "black water," with the potable freshwater supply. There is a considerable body of research in the social sciences which suggests that a majority of people harbor a very basic, if somewhat irrational, prejudice against the direct reuse of reclaimed water for potable applications.

2.2 The Multi-Objective Corridor Location Problem

The multi-objective corridor location problem can be formally written as Equation 2.1[64]. The problem involves the simultaneous minimization of the sums of w independent objective functions $\hat{O}_w$ evaluated at the set of discrete locations x_n comprising a corridor of

length n. A valid corridor x_n is subject to the constraint that all of its nodes must be contained with the feasible search domain Ω. Additional, optional constraints, as in Equation 2.2, are often imposed upon the structure of x_n and shall be discussed in greater detail in subsequent sections.

$$Minimize : \prod_{n=1}^{n} \left\{ \hat{O}_1(x_n), \ldots, \hat{O}_w(x_n) \right\} \tag{2.1}$$

$$S.t. : x_n \in \Omega \tag{2.2}$$

Where:

x_n = The set of discrete row column indices defining a corridor of length n

Ω = The set of discrete row column indices defining the feasible decision space

O_w = The true but unknown forms of w continuous objective functions

$\hat{O}_w$ = The estimates of O_w defined over the discrete set Ω

As a subset of SPPs, corridor location problems tend to be defined in the context of networks with large numbers of nodes and highly structured topologies[24]. These shared characteristics arise from the fact that corridor location problems are typically posed in the context of continuous geographic space – a feature which requires that the requisite underlying network structure be generated algorithmically[54]. In practice, this is frequently accomplished by automating the conversion of a geographically referenced raster grid into a set of nodes by referencing the the centroids of the cells within the raster in a process similar to that described by[30]. Once the nodes in the network have been created they can then con-

nected to one another by automatically generating arcs using some standard mode of node connectivity; again, using methods similar to those described by[15].

2.3 Genetic Algorithms

Genetic Algorithms (GAs) are a family of search heuristics that mimic the process of natural selection to derive one or more near optimal solutions to a given optimization problem.[22] GAs constitute a subset of Evolutionary Algorithms (EAs) which encode solutions using data structures that are analogous to biological chromosomes[20]. This feature allows the search for new, better solutions to be accomplished via the iterative application of genetic operations such as crossover, mutation, selection, etc.[23] GAs have been developed and profitably used in a wide variety of problem domains from engineering and economics to chemistry and physics. General purpose reviews of the application of GAs to various problems are available from the following references.[21,66,17] For a more specialized reviews regarding state of the art applications of multi-objective genetic algorithms see the excellent book from Coello & Lamont and, more recently, from Zhou *et al.*[16,64]

2.4 The MOGADOR Algorithm

MOGADOR is an acronym stands for "Multiple Objective Genetic Algorithm for Corridor Selection Problems."[63] The algorithm was introduced as a novel genetic approach to the problem of multi-objective corridor search.[40,63] It development was intended to service need for a robust method of siting optimal corridors relevant to a variety of environmental planning and design applications.[7,13,65] A need which persisted despite numerous previous efforts to adapt general purpose, deterministic, SSP solution techniques for use in corridor

location.[27,31,38] The continued need for refined algorithmic approaches to the location of optimal corridors within a broad range of application domains is evidenced by the continued appearance of closely related publications during the intervening years.[2,41,42,48,50,59]

Relative to other traditional shortest path finding routines, the MOGADOR algorithm has been observed to possess a number of favorable characteristics. First among these is the ability of MOGADOR to accommodate large problem statements (large Ω) and or those which require the simultaneous evaluation of large numbers of independent objectives (large w)[?,63]. Another useful feature of the MOGADOR algorithm is its ability to generate an entire solution set from a single run – as opposed to the single solution per run which is typical of traditional SPP algorithms[?,?,?,63]. Lastly, but perhaps most importantly, is that when properly parameterized, each of the solutions generated by the MOGADOR algorithm can be said to be non-inferior to every other[?]. In this way, the output solution set approximates the so called Pareto optimal solution set for the given problem statement[?].

In order to proceed with our discussion of the MOGADOR algorithm, some basic terminology must first be defined. In the context of the MOGADOR algorithm a single gene x is comprised of a pair of row column indices (r, c) to a geographically referenced 2-D array comprising the feasible search domain Ω. An individual I_m is comprised of a sequence of row column index vectors x_n which collectively form a valid pathway between a predefined set of source ${}^{s}\mathrm{x}$ and destination ${}^{d}\mathrm{x}$ locations. In this way, each individual represents a feasible solution to the proposed corridor location problem. A population P_g is comprised of m individuals. And, finally, an evolution E_b is comprised of g populations.

Figure 2.1 provides a pseudocode description of the MOGADOR algorithm. Its structural components are fairly typical among GAs in general. The search process begins with

a stochastic routine for generating of an initial seed population P_1. Following the initialization of this seed population, the fitness F_1 of each individual in the seed population is computed by summing the objective function scores corresponding to each set of nodes comprising each individual. Upon the completion of this initialization phase, the algorithm then enters a loop wherein successive genetic operations are applied to the initial seed population. In the case of MOGADOR, these operators include: the selection individuals for reproduction on the based upon their fitness, the crossover of selected individuals that share at least one common feature, and finally, the mutation of crossed over individuals so as to maintain a degree of random variation within the population. At the end of each loop iteration a new population is generated, its fitness evaluated, and a convergence parameter computed on the basis of the observed rate of improvement in population fitness across previous iterations. If convergence is achieved ($c_g < {}^t c$) the loop is broken and the algorithm terminates returning: the evolutionary history of the population $P_{1:g}$, the corresponding fitness values each individual within the population $F_{1:g}$, and the convergence parameter history $C_{1:g}$.

Figure 2.1: MOGADOR Algorithm Pseudocode

2.5 The MOGADOR Data Structure

The optimal data structure for use in concert with the MOGADOR algorithm is a nested list of lists. Such a list based data structure is well suited to this context as there can be a high degree of variability in the number of elements produced by the different stochastic genetic operators.1 Figure 2.2 illustrates a small but valid example such a nested list of lists

Algorithm 1

procedure MOGADOR
    $g = 1 \leftarrow$ initialize loop iterator
    $c_g = 0 \leftarrow$ initialize convergence parameter
    $\mathrm{P}_g = \mathit{initializeSeedPopulation}(m, \Omega)$
    $\mathrm{F}_g = \mathit{computePopulationFitness}(\mathrm{P}_g, \hat{O}_w)$
    while $c_g < {}^{t}c$ do:
        $g+ = 1 \leftarrow$ update loop iterator
        $\mathrm{S}_g = \mathit{selectIndividualsFromPopulation}(\mathrm{P}_g)$
        $\mathrm{X}_g = \mathit{crossoverSelectedIndividuals}(\mathrm{S}_g)$
        $\mathrm{P}_g = \mathit{mutateCrossoverIndividuals}(\mathrm{X}_g)$
        $\mathrm{F}_g = \mathit{computePopulationFitness}(\mathrm{P}_g)$
        $c_g = \mathit{computeConvergenceParameter}(\mathrm{F}_{1:g})$
        end
    end while
    return: $\mathrm{P}_{1:g}$, $\mathrm{F}_{1:g}$, $\mathrm{C}_{1:g}$
end procedure

data structure as used in the context of the MOGADOR algorithm. Note, for example, how the number of row column index vectors n in the first individual I_1 equals 5, while the number of index vectors in subsequent individuals $\mathrm{I}_{2:4}$ belonging to the same population, ranges between 3 and 6. The source of this variation has to do with the stochasticity inherent to the population initialization routine. Similarly, note how the number of populations g in the first evolution E_1 is 3 while the number of populations in the second evolution is 4. The source of this variation has to do with the stochasticity in the crossover and mutation processes. Indeed, the only level of the data structure's hierarchy at which a constant number of elements is required is at the population level. Meaning, in other words, that each population P_g contained within evolutions E_b must all possess the same number of m individuals. This requirement ensures that the behavior of the genetic operators is consistent between separate evolutions.

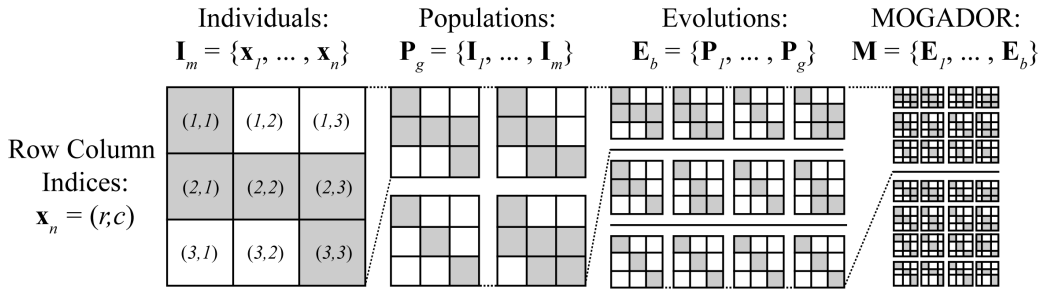


Figure 2.2: MOGADOR Algorithm Data Structure

2.6 Initializing the MOGADOR Algorithm

A novel population initialization procedure has been developed for use in conjunction with the MOGADOR algorithm which improves the global quality of the output solution set while simultaneously reducing overall computational effort. At its core, this novel pseudo-random walk algorithm works by repeatedly sampling a dynamically parameterized bivariate Gaussian distribution. The generic form of the probability density function for the bivariate Gaussian distribution can be written as Equation 2.3[3]. Here, the bivariate Gaussian PDF $f(\tau)$ is function of two inputs. [1] The first is a mean vector μ, comprised of the means (μ_1, μ_2) of two continuous random variables (τ_1, τ_2). [2] The second is a covariance matrix Σ, comprised of the pairwise covariances σ for all possible combinations of the continuous random variables (τ_1, τ_2).

$$f(\tau) = \frac{1}{\sqrt{(2\pi)^2 \mid \Sigma \mid}} e^{\frac{-1}{2}(\quad - \quad)^T \Sigma^{-1} (\quad - \quad)} \tag{2.3}$$

Where:

$$\tau = \text{The set of correlated continuous random variables } (\tau_1, \tau_2)$$

$$\mu = \text{The set of mean values } (\mu_1, \mu_2) \text{ for } (\tau_1, \tau_2)$$

$$\Sigma = \text{The pairwise covariance structure } \begin{bmatrix} (\tau_1, \tau_1) & (\tau_1, \tau_2) \\ (\tau_2, \tau_1) & (\tau_2, \tau_2) \end{bmatrix} \text{ for } (\tau_1, \tau_2)$$

Each sampled value for τ_1, τ_2 can be reduced to a unit vector and interpreted as a set of row column index deltas. The repeated sampling of the distribution therefore provides a simple yet powerful technique for generating randomized positional changes within a 2-D lattice. Additionally, as shall be discussed in the subsequent sections, the ability to dynamically adjust the parameters of the bivariate Gaussian PDF at any time during the sampling process provides a mechanism by which one is able to functionally constrain the randomness of the walk; hence the term: pseudo-random walk.

Figure 2.3 provides a pseudocode representation of the proposed pseudo-random walk procedure. Structurally, the routine consists of two nested while loops. At each iteration h of the outer loop a single step x_n along an individual walk I_m is taken. This loop continues until the location of the current step is equal to that of the destination ${}^{d}\mathrm{x}$. At each iteration u of the inner loop a candidate next step $\Delta\mathrm{x}_u$ is produced by random sampling the parameterized bivariate Gaussian PDF $f(\mathrm{x}_u)$. Candidate next steps are only considered valid if they are contained within the current valid connected set V_n. The current valid connected set is comprised of all neighboring nodes that not been previously visited and that are inclusive to the search domain Ω. If the candidate step is found to be valid the inner loop terminates, the outer loop iterates, and the walk process continues. If the valid set is ever found to be empty the outer loop iterator h is reset and the entire process is restarted.

Figure 2.3: Pseudo-Random Walk Algorithm Pseudocode

Algorithm 2

procedure PSEUDO-RANDOM WALK
    $n = 1 \leftarrow$ initialize outer loop iterator
    $\mathrm{x}_n = {}^{s}\mathrm{x} \leftarrow$ initialize individual at source
    $\mathrm{I}^* = \mathit{computeEuclideanShortestPath}({}^{s}x, {}^{d}\mathrm{x})$
    while $\mathrm{x}_n \neq {}^{d}\mathrm{x}$ do:
        $\mathrm{V}_n = \mathit{computeValidConnectedSet}(\mathrm{x}_{1:n}, \Omega)$
        if $\mathrm{V}_n = \emptyset$ then:
            $n = 1 \leftarrow$ reset outer loop iterator
            continue
        end if
        $u = 1 \leftarrow$ initialize inner loop iterator
        $\mu_n = \mathit{computeOrientationVector}(\mathrm{x}_n, {}^{d}\mathrm{x})$
        $d_n = \mathit{computeMinimumBasisDistance}(\mathrm{x}_n, \mathrm{I}^*)$
        $n = n + 1 \leftarrow$ update outer loop iterator
        while $\mathrm{x}_n \notin \mathrm{V}_n$ do:
            $\Sigma_u = \mathit{computeCovarianceMatrix}(d_n, u)$
            $\Delta\mathrm{x}_u = \mathit{sampleBivariateGaussianDistribution}(\mu_n, \Sigma_u)$
            $\mathrm{x}_n = \mathrm{x}_{n-1} + \Delta\mathrm{x}_u$
            $u = u + 1 \leftarrow$ update inner loop iterator
        end while
    end while
    return: $\mathrm{x}_{1:n}$
end procedure

In the process of sampling the parameterized bivariate Gaussian distribution the following three pieces of information are used to functionally constrain the probabilities associated with each candidate next step Δx_u. [1] At the start of each walk the set of array indices corresponding to the Euclidean shortest path I^* from the source to the destination are generated using Bresenham's line algorithm[3] . Using this set of indices the minimum distance d_n from the current position to the nearest point along this Euclidean shortest path is determined. [2] Next, at each step an orientation vector is computed indicating the orientation of the destination location relative to the current position. [3] Finally, each time the bivariate Gaussian PDF is sampled a counter variable u is iterated.

The orientation vector is interpreted as the mean of the bivariate Gaussian $PDF(\mu_n)$. Alternatively, the minimum distance and iteration variables (d_n, u) are processed as inputs to a generator function which produces the covariance matrix Σ_u. This composition of this generator function ensures that the degree of randomness inherent to the selection of each next step is directly related to number of iterations while at the same time being inversely related to the minimum distance from the current location to the Euclidean shortest path.

2.7 An Example Pseudo-Random Walk

In an effort to make this pseudo-random walk procedure more comprehensible, particularly with regards to the parameterization of the bivariate Gaussian PDF $f(x_u)$, an example implementation is provided in Figure 2.4. On the far left of Figure 2.4. the current status of an arbitrary pseudo-random walk is shown midway through completion. Also drawn, as a broken line, is an abstract representation of the Euclidean shortest path I^* connecting the source location ${}^s x$ to the destination location ${}^d x$. Show at bottom is the current value of

the distance parameter $d_n \approx 7.1$. Just to the right of this, in the zoom inset area, the current valid connected set V_n is drawn with the previously visited indices (x_n, x_{n-1}) greyed out to illustrate their elimination from consideration as valid next steps in the walk process. Also shown in this inset are the row column unit vector deltas Δx_u associated with movement to each of the seven nodes contained within the current valid connected set. The small arrow pointing downwards and to the right depicts the current state of the orientation vector μ_n which describes the position of the destination location ${}^{d}x$ relative to the current walk location x_n. The current value of this vector $(\mu_n = [1, 1])$ can be thought of as indicating the row and column deltas associated the next step possible step most directly leading towards the destination.

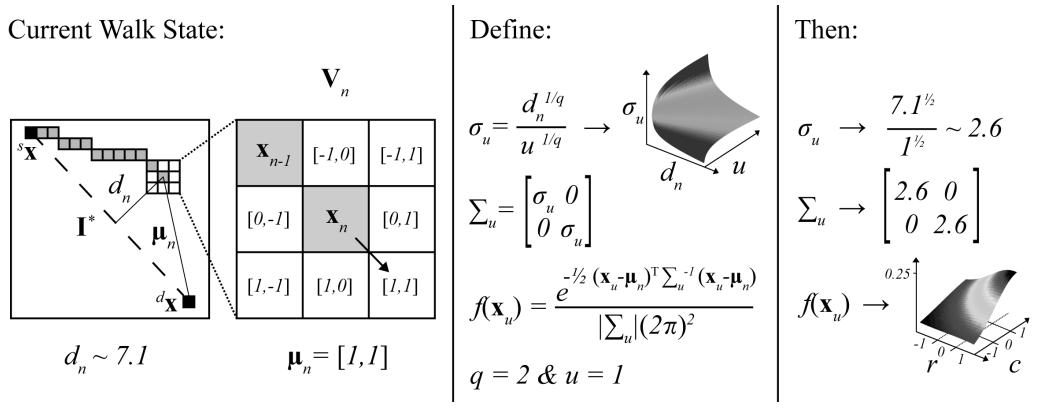


Figure 2.4: Pseudo-Random Walk Example

Continuing on to the right within Figure 2.4, three functions and two parameter values are defined. [1] First, the covariance term σ_u for the current sample iteration u is specified using a covariance generator function whose form enforces the relationship between distance, iteration count, and covariance previously described. A small demonstration plot of this function's form is provided. [2] Next, the covariance matrix Σ_u is defined by insert-

ing the covariance term σ_u to the diagonal elements of an empty square 2-D matrix. This repeated use of the same covariance term guarantees that for any value of σ_u the output covariance matrix Σ_u will be positive definite. A square, symmetric, and positive definite covariance matrix is a hard requirement for the evaluation of the parameterized bivariate Gaussian PDF $f(\mathrm{x}_u)$. The final two variable definitions (q, u) are provided for the sake of computing illustrative values for the other parameters. Numerical evaluations of these expressions are given on the far right portion of the figure. Here again, a small demonstration plot showing the form of the evaluated bivariate Gaussian PDF $f(\mathrm{x}_u)$ is provided.

One aspect of this process which warrants further discussion is the role of the fixed parameter q in determining the degree of randomness exhibited by a given pseudo-random walk. The degree of randomness can be quantitatively defined as the range and extent to which the bivariate Gaussian PDF $f(\mathrm{x}_u)$ deviates from its uniform bivariate counterpart. To illustrate this concept consider for example the characteristics of the PDF that would be required to produce a simple random walk using a similarly structured procedure. In such a case the value of $f(\mathrm{x}_u)$ would have to be equal for all possible values of $\Delta\mathrm{x}_u$. Due to the way in which the covariance generator function has been proposed, the q parameter can therefore be used to determine the maximum range of variation in σ_u which can be produced from any combination of (d_n, u) input values. In this way, q does not alter the structure of the bivariate Gaussian PDF $f(\mathrm{x}_u)$ but rather only its magnitude. As a result, while the value of q must always be greater than zero to produce real outputs from the covariance generator function, its value is inversely related to the degree of walk randomness.

2.8 Initializing Problems with Large Decisions Spaces

While the pseudo-random walk procedure can be used to generate an initial seed population for any MOGADOR problem statement; a number of circumstances have been identified in which the performance of the population initialization procedure can be further refined. [1] The first such situation involves problems with extremely large decision spaces – defined as being thousands grid cells or more on a side. [2] The second problem specifications where it is known, a-priori, that the Euclidean path connecting the source to the destination is not entirely feasible.

Historically, corridor location problems which have been posed in the context of extremely large decisions spaces (large Ω) have been considered infeasible both for conventional deterministic SPP optimization techniques as well as for heuristic approaches such as MOGADOR. With regards to MOGADOR, the source of this infeasibility stems from the huge runtime commitment associated with generating and processing populations containing a sufficient number of individuals so as to ensure that a sufficient amount of genetic diversity can be captured during the initialization phase to conduct a global search.

One strategy which can be employed to ensure enough genetic diversity is produced within the initial seed population without having to generate populations of unduly large size or populations with individuals characterized by a high degree of randomness, is the generation of so called multi-part pseudo random walks. This procedure can be thought of as somewhat analogous to orthogonal statistical sampling techniques such as Latin Hypercube sampling which are used to generate samples from a non-uniformly distributed population by first dividing it into equally probable subspaces (McKay et al 1979, Ye 1998).

The implicit assumption here being that the fitness distribution of all possible corridors connecting a typical source and destination pair is similarly non-uniform.

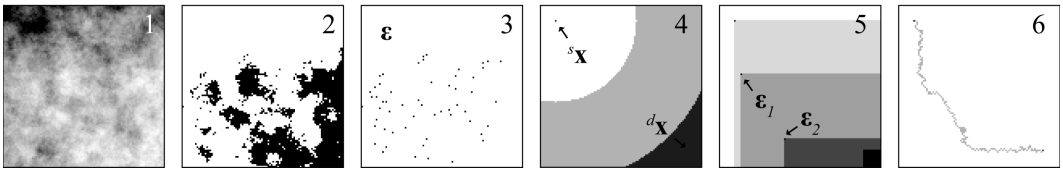


Figure 2.5: Conceptual Illustration of a Multi-Part Pseudo-Random Walk

The multi-part walk process is described by the sequence of panels moving from left to right within Figure 2.5. [1] The process begins with the far left panel which plots the value of the objective variables within the a square 2-D search domain. [2] The first step is to create a binary mask of feasible nodes by selecting objective surface values less than some arbitrary threshold. [3] The next step involves determining the cell indices for the set of centroid nodes ε computed from the connected components within this binary mask. [4] After this, these centroids are assigned rankings on the basis of their inclusion in bins of progressive Euclidean distance from the source location. [5] Next, the procedure requires the iterative selection of centroids ε_1, ε_2, one from each successive distance bin, until the bin containing the destination location is reached. The centroid selection process can be unstructured, random within each bin, or structured (as shown), where the centroids considered eligible for selection each iteration is restricted to those orientated positively in the direction of the destination. [6] Finally, the source is connected to the destination a series of pseudo-random walks constructed for sequential pairs of the selected centroids.

2.9 Initializing Problems with Concave Decision Spaces

Another circumstance which has the potential to dramatically effect the performance of the pseudo-random walk procedure are problem specifications in which all or a portion of the Euclidean path connecting the source to the destination falls outside the feasible area of the search domain. Such a circumstance can be described as a concave problem, as the source and destination locations are not convex to one another within the boundaries of the decision space Ω. Concave problem statements have the potential to create a situation where at each iteration n of the pseudo-random walk process large values of u must be attained before the covariance term is relaxed enough to allow for the sampling process to generate a Δx_u that is contained within the valid set V_n. In a worst case scenario, the entire runtime improvement associated with the pseudo-random walk based population initialization procedure might be lost.

An approach which has been developed to address such cases is the so called concave multi-part pseudo random walk. It is similar to the standard multi-part walk in that the final walk is composed of a collection of pseudo-random walk sections. However, it differs from the standard walk procedure in that rather than partitioning the space on the basis of distance bands, it iteratively divides the decision space into a series of convex subregions. Here again, these convex sub region contain the centroids associated with connected regions of low objective variable values. The procedure is illustrated conceptually by the sequence of panels contained in Figure 2.6.

The concave multi-part walk process is described by the sequence of panels moving from left to right within Figure 2.6. [1] In the first panel, on the far left, we can see a problem that

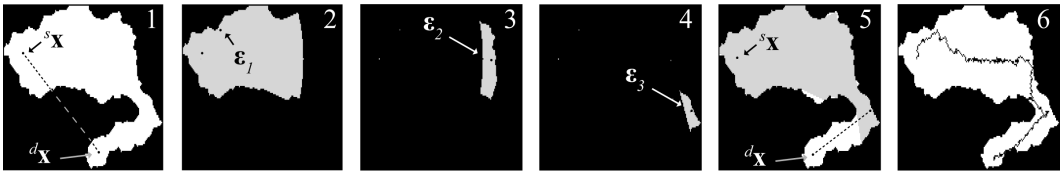


Figure 2.6: Conceptual Illustration of a Convex Multi-Part Pseudo-Random Walk

has been posed in such a way that the Euclidean path connecting the source to the destination is not feasible as it exits the boundaries of the search domain. [2] For brevity, the two subsequent steps are not shown as they are identical to steps 2-3 in the standard multi-part pseudo random walk procedure. These omitted steps involve the generation of candidate centroids from the connected components within the objective surface. The next illustrated step, shown in the panel second from left, involves computing all of the row column indices that are convex to the source location.Within this convex region a the first centroid ε_1 is selected. Here again, the process can be unstructured, where the entire convex region is searched, or structured (as shown), where the initial convex region is restricted to some maximum distance from the source. [3-4] From here, additional non-overlapping convex subregions are computed and candidate centroids iteratively selected from within them. [5] The centroid selection process concludes when the current convex region contains the destination location. [6] Finally, the source is connected to the destination, as in the simple multi-part case, by a series of pseudo-random walks constructed for sequential pairs of the selected centroids.

2.10 Measuring Initialization Performance

The stochastic processes inherent to the pseudo-random walk procedure, as well as to many other components of the MOGADOR algorithm, make it difficult to analytically derive performance characteristics. As a consequence, with MOGADOR, as with many other GAs, features such as runtime performance must be evaluated through empirical observation. The following sections introduce the results obtained from several such empirical investigations related to the performance of the pseudo-random walk based population initialization procedure for the MOGADOR algorithm. For reference, all computations were performed using a desktop class hardware possessing a 2.3 GHz Intel Quad Core i7 processor (2nd Gen.) with 16 GB of system RAM.

The first of these investigations seeks to understand the role of the fixed parameter q, embedded in the pseudo-random walk covariance generator function, in determining the structural characteristics of output populations. In order to study this issue, a synthetic problem statement was created. This problem statement involves a square search domain of 100 nodes on a side – resulting in a total problem size of $\Omega = 10,000$ nodes. The value of the estimated objective function $\hat{O}_w$ used to evaluate fitness was set as constant for all of the nodes in the search domain. In this way, the objective score is roughly equivalent to individual walk length in Euclidean space. Using this problem statement, twenty seed populations, all containing $m = 100$ individuals, were generated using monotonically increasing values of q beginning with $q = 0.5$ and concluding with $q = 10$. For each the generated populations, the average objective scores for all individuals as well as the standard deviation of the objective scores among individuals were evaluated. The results of this empirical

investigation are presented in Figure 2.7.

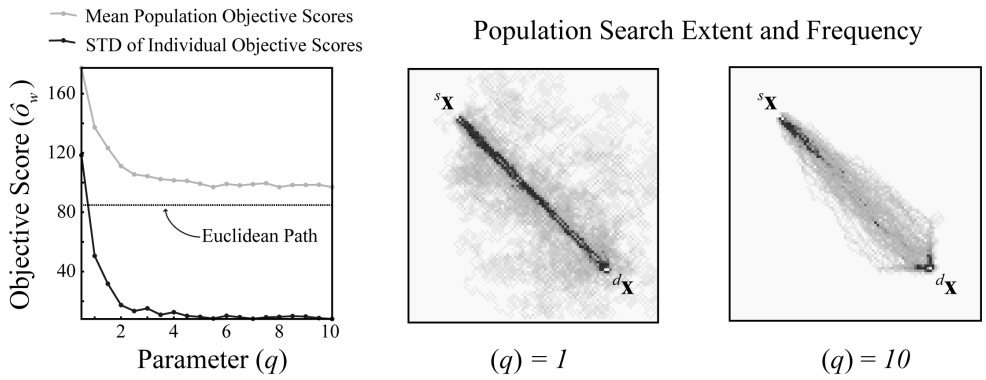


Figure 2.7: Observing the Role of the Fixed Parameter q in Determining Path Randomness

As Figure 2.7 illustrates, increasing the value of the fixed parameter q causes the individuals within the a population to more closely approximate the Euclidean shortest path between the source and destination. Similarly, relaxation of this parameter results in an increase in the perceived randomness among the individual walks within a population. These attributes can be clearly observed in the two images to the right of Figure 2.7 which show the frequency with which every node within the search domain has been visited by any individual within two populations generated from different values of q.

Another issue warranting empirical investigation is the relationship between runtime performance of the proposed initialization procedure and problem size. In order to study this issue a series of ten synthetic problem statements were constructed with near identical structural components; differing from on another only in terms of problem size. In each of these problem statements the source location was positioned one fifth of the way down and to the right from the top left of the search domain and, likewise, the destination location was positioned a constant one fifth of the way up from the bottom right corner of the

search domain. For all of the different populations generated, the value of the fixed parameter q was set relatively high $q = 10$ to reduce computational effort.

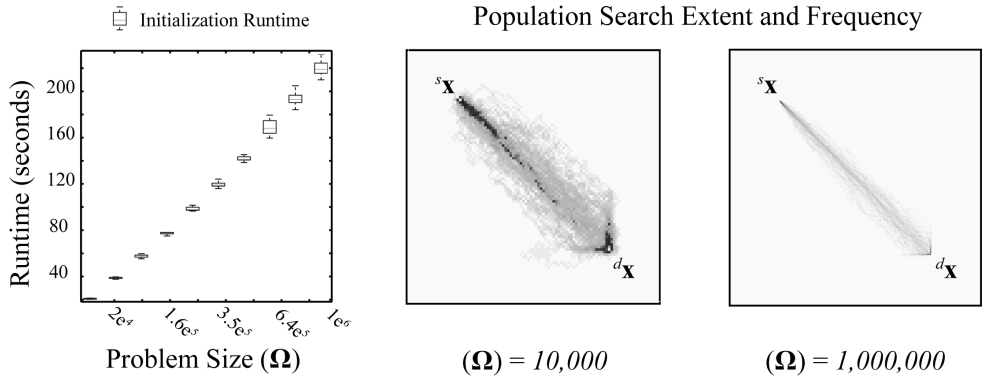


Figure 2.8: Observing the Role of Problem Size on Initialization Runtime

The plot to the left of Figure 2.8 illustrates the the distributional properties of the runtimes required to generate ten replicate populations for the ten problem statements of progressively increasing size – resulting in a total of $g = 100$ unique populations, each comprised of $m = 100$ individuals. Such repeated simulation is necessary as the proposed initialization routine is based upon a stochastic sampling process which can and will deliver variable runtimes for the repeated applications to the same problem context. One feature of note in this plot is the roughly linear relationship between the mean runtime and problem size for this type of pseudo-random walk based approach to the problem initialization procedure for the range of problem sizes considered. The two images to the right illustrate the effective search extent and frequency for the smallest and the largest populations generated during this investigation.

One of the considerations previously discussed related to the initialization of the MO-GADOR algorithm in the context of large problem statements was the need to ensure suf-

ficient diversity within the seed population for the search process to be conducted at the global level. This problem is clearly evident in the population search extent and frequency image contained on the far right of Figure 2.8. In this example, the population clearly fails to explore a sufficiently large portion of the decision space to be considered as a form of global search. The solution which was previously proposed to this problem involved generating so called multi-part pseudo-random walks. The subsequent investigation therefore, compares the statistical characteristics of a set of populations generated from standard pseudo-random walk to another set of population generated from multi-part pseudo random walks. The results of this investigation are provided in Figure 2.9.

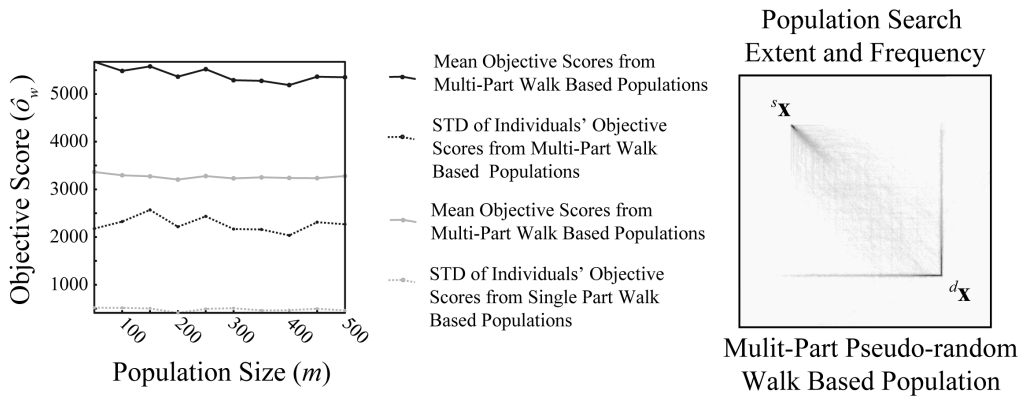


Figure 2.9: Observing the Characteristics of Multi-Part Pseudo-Random Walks

The runtime reductions which can be achieved from the use of the multi-part pseudo random walk procedure will principally occur in the context of relatively high value settings for the fixed parameter q. This is because while the various component segments of a multi-part walk may not deviate significantly from the Euclidean shortest path however, the randomized connection of multiple such segments produces composite individuals that are significantly more diverse.

2.11 The MOGADOR Algorithm Outputs

The output of the MOGADOR algorithm is the state of population at the time in which the algorithm has either achieved convergence or has been terminated upon reaching the specified limit for evolutionary iterations. This population contains the row-column subscripts for each individual corridor as well as the corresponding fitness values at each of the row column subscripts along the entire length of each individual corridor. These individuals are ranked on the basis of their aggregate cumulative fitness values. These values are computed by summing each objective value at each subscript location along the length of each individual corridor. Because this is a multi-objective context two individuals may have equivalent aggregate cumulative fitness scores yet the distribution of these scores across the various objectives may vary. This would represent a situation where those two individuals – as independent solutions to the mulit-objective corridor location problem – are *Pareto Optimal* or, alternatively stated, inclusive of the *non-dominated set*.

Either the rest of the world can't live like the developed world or we need, as a society, to think more about the technology of providing these services with less intensive use of at least certain resources. We need to do a more diligent job of good housekeeping.

Thomas E. Graedel (1940-)

3

Quantifying Life-Cycle Energy-Water Resource Utilization

3.1 Quantifying Life-Cycle Energy-Water Resource Usage Efficiency

The third major challenge associated with this dissertation's stated research objective involves the development of a systematic method for quantifying the life-cycle energy-water resource utilization efficiency of water reuse systems involving significant artificial groundwater recharge components. For this purpose, the decision was made to build upon a substantial body of previous research on this subject by utilizing an existing life-cycle inventory (LCI) database comprising energy and material process flow data collected from a large sample of existing WWTP operations and reuse facilities[55,56,57]. The novel contribution of this dissertation to this body of work appears in the form of a new set of methods for the programmatic parameterization of the LCI model based upon the geographic context and spatial layout of the proposed reuse project under evaluation.

For example, by first determining suitable sites for recharge infrastructure and then elucidating near-optimal corridors for the distribution network supplying the treated wastewater from the WWTP, the research analyst is then able to gain a deeper and more fundamental understanding of the systems behaviors. For example, with a detailed corridor specification in hand, it then becomes possible to estimate process energy demands of the system from first principles rather than having to use vague reference data aggregated from a wide distribution of empirical measurements[33]. As we shall demonstrate in subsequent sections, these new capabilities go a long way towards improving both the accuracy and the precision of the model's outputs.

3.2 Life Cycle Assessment and Inventory Modeling

Life-cycle assessment (LCA) is an environmental accounting framework that was developed to systematically quantify the material and energy inputs and outputs from a product, process, or system throughout all its stages of life. It uses a cradle-to-grave or, in some cases, cradle-to-cradle perspective, to evaluate the design processes as well as the entire supply chain associated with manufacturing, transportation, the use phase, and waste management[47,46]. Practically, process based LCA analyses, which shall be the focus of the remainder of this discussion, incorporate two distinct modeling phases. The first involves the development of a Life-cycle Inventory model is a cumulative record of all of the materials and energy flows required to deliver a single functional unit of the product or process in question. The second, optional, phase of LCA analyses is the Life-cycle Impact Assessment (LCIA) component. An LCIA model attempts to translate the raw energy and material flows contained within the LCI into different categories of environmental impacts such as global warming potential, ocean acidification, freshwater eutrophication, etc.

If we recall, the overarching goal of this dissertation project was to quantify the energy-water usage efficiency of artificial groundwater recharge projects involving the reuse of treated municipal wastewater. In order to accomplish this goal customized LCI models will be developed for five case study regions in which the distribution networks responsible for transporting the treated wastewater for its point of origin, the WWTP, to its destination point of consumption, an artificial groundwater infiltration basin positioned at a designated destination location, typically upstream within the regional watershed. The raw data supporting the creation of each of these custom LCI models was derived from the UC

Berkeley supported Web Water-Energy Sustainability Tool (WWEST)[55]. In each case study, the scope of this LCI modeling exercise was limited to the construction and operational requirements associated with the WWTP plant, the treated water distribution network, and the infiltration basin. This system boundary has been defined in such a way as to emphasize the dynamic contribution of the treated wastewater distribution to the LCI of a given functional unit of treated wastewater delivered to the sub-surface in the face of variable geographic context.

3.3 Wastewater Treatment Processes

The phrase wastewater treatment encompasses a wide variety of different processes and operational facilities depending upon: the quality of the influent water, the volume of the influent water, and the desired quality/end-use application for the treated effluent water[35]. In the United States the operation of WWTPs are regulated at both the State and Federal levels[34]. At the Federal level the principle regulatory agency is the United States Environmental Protection Agency (USEPA) and the principle regulatory program is the National Pollutant Discharge Elimination System (NPDES)[60]. According to the legal mandate of the NPDES program, WWTP operators – as well as a wide variety of other entities – are required to apply, at regular time intervals, for discharge permits which provide them with the legal right to release waters containing limited concentrations of regulated pollutants into the environment. Also under this mandate, the USEPA is required to distribute these permits and enforce non-compliance with their terms.

Since the inception of the NPDES program, the USEPA has worked to make readily accessibly a centralized database of all registered permit holders within the United States.

This database in interesting for the purposes of this project in that it contains spatially referenced information about the operational aspects of every operating WWTP in the U.S.[43] Crucially, this information includes data on maximum daily permitted flow rates and total maximum daily loads that can be used to parameterize the type of process based LCI model facilitated by the WWEST tool.

In terms of their basic physical layout and operational requirements, WWTPs are typically constructed with a tiered layout; comprising primary treatment, secondary treatment, and sometimes, various so called tertiary processes[53,1,44]. Both primary and secondary treatment are terms that come with narrow legal definitions and are implemented at nearly all WWTP plants handling municipal sewage discharges. Tertiary treatment processes are more loosely defined and encompass a suite of advanced treatment processes that are so costly that they tend to only be implemented at a minority of WWTP that are subject to a unique circumstances in terms of influent pollutant loadings/composition or requirements associated with designed high purity effluent end-use applications[12,29].

Primary treatment encompasses processes and equipment dedicated to the physical separation of non-soluble waste constituents present in the influent wastewater stream[39]. Within a WWTP, a number of distinct processes are often lumped together as being part of the primary treatment. For example, when water first enters the WWTP it is guided through a series of progressively refined grates to screen out bulk pollutants such as anthropogenic trash or natural plant and animal detritus. Following from this bulk screening phase, the water is guided into a series of settling basins where its movement is slowed to crawl to facilitate the settlement of suspended pollutant materials such as sediment[39]. Due to the slow rate at which this settling process proceeds, the physical infrastructure which

supports it can comprise a significant fraction of the overall footprint of a WWTP; particularly for those with high flow volume processing requirements.

Secondary treatment encompasses processes and equipment dedicated to the biological (and sometimes chemical) degradation of soluble waste constitutes present in the influent wastewater stream[39]. In most municipal WWTP secondary treatment is accomplished through a passively aspirated, aerobic biological digestion reactors. In these reactors large colonies of bacterial species are cultivated on high surface area media using the organic components of the influent wastewater stream as a feedstock for the continued growth[39]. At the end of their life cycle, the bacteria fall to base of the reactor tank and must be continuously removed in the form of a product known as activated sludge.

Tertiary treatment encompasses processes and equipment dedicated to the removal of soluble inorganic and some organic chemical species (including some viruses and pharmaceutical agents) present within the influent wastewater stream[39]. At present, tertiary treatment processes are not mandatory for all WWTP facilities regulated under the NPDES program. In general, they tend to only be implemented at those specific locations in which a last and credible threat to public or environmental health has been identified and for which a targeted tertiary treatment process exists to address. In this way, mandates for tertiary treatment are typically instigated at the state or local level and done so on a case by case basis. Among the most common tertiary treatment processes include: reverse osmosis filtration, batch irradiation with ultra-violent light, the application of specialized chemical amendments, de-nitrification processes, and others[39].

For the purposes of this analysis and the customized LCI models which shall be constructed as part of the case study investigations, only primary and secondary treatment pro-

cesses shall be included in the scope. This decision has been made to eliminate a substantial bias in the inventory models process flows that might be associated with the inclusion of specialized tertiary treatment procedures.

3.4 Water Distribution Infrastructure

The immediate delivery and reuse of treated wastewater for various municipal and agricultural end-use applications is still a relatively new phenomenon[5,8]. As such, the regulatory landscape surrounding such practices is still not well defined at the Federal level here in the United States[4]. Consequently, what regulations due exist, typically have been enacted at the State and local levels, with the most advanced frameworks, unsurprisingly, existing in those states such as Florida, California, and Arizona where the popularity of reuse as viable alternative source of freshwater supply, has been surging in recent years[11,37,49,62].

In all of the locations within the United States for which solid regulatory frameworks surrounding reuse currently exist, there are strong constraints governing the use of existing water distribution infrastructure for the transportation of treated wastewater from its point of origin, the WWTP, to its point of end-use[19,61,10]. These regulations, without exception, stipulate that treated wastewater, even if returned to a level of quality consistent with requirements for potable use, cannot be conveyed using existing distribution infrastructure carrying potable water for human consumption in municipal areas. Due to this regulatory constraint, all treated wastewater destined for some sort of municipal reuse must be carried through dedicated parallel distribution infrastructure[11]. In California, this infrastructure is easily identified at locations where treated wastewater is being reused due to the bright purple color of all the pipes. This color encoding is meant to be a strong visual reminder that

the water being carried within them has not been deemed, from a regulatory perspective, as being fit for direct human consumption[9,60].

The requirement that treated wastewater, regardless of its standard of treatment and anticipated end use application, be transported using a separate parallel distribution network is expected to be a crucial factor in determining the overall life-cycle energy-water usage efficiency of large scale water reuse systems feeding into artificial groundwater recharge basins. The reasoning behind this expectation is based upon the interaction of the following two key factors. Firstly, water is a dense material, and thus it is very energy intensive to transport it over long distances and against steep elevation gradients. Secondly, artificial groundwater recharge basins typically require fairly large amounts of contiguous land area that are situated in fairly close proximity to highly developed urban and suburban communities. Municipal water resource management agencies are tightly constrained in terms of the operating budgets from which they are able to draw funds to procure new land holdings for the purpose of constructing artificial recharge basins. Thus, artificial recharge basins , primarily to economic constraints, are typically located fairly far afield from the WWTPs which feed them.

3.5 WWEST Recycled Water Reuse Life Cycle Inventory Model

The WWEST Recycled Water Reuse Life-cycle Inventory Model refers to an integrated life-cycle inventory database and modeling framework for understanding the environmental impacts of wastewater treatment and reuse processes that was developed by a team of academic researchers operating out of the University of California at Berkeley College of Engineering[58]. The principal investigators behind the project are Doctor Arpad Horvath, Pro-

fessor of Civil and Environmental Engineering, and Doctor Jennifer Stokes. Doctor Stokes completed the initial development work for the WWEST model as part of her doctoral dissertation research in the area of environmental impact assessment of civil infrastructure systems.

The LCI database underlying the WWEST toolset contains process flow information for the manufacture and operation of equipment and facilities involved in the supply, treatment, and distribution of municipal wastewater for the purpose of non-potable reuse. Figure 3.1 provides a schematic overview of the WWEST database model components and their respective input data sources.

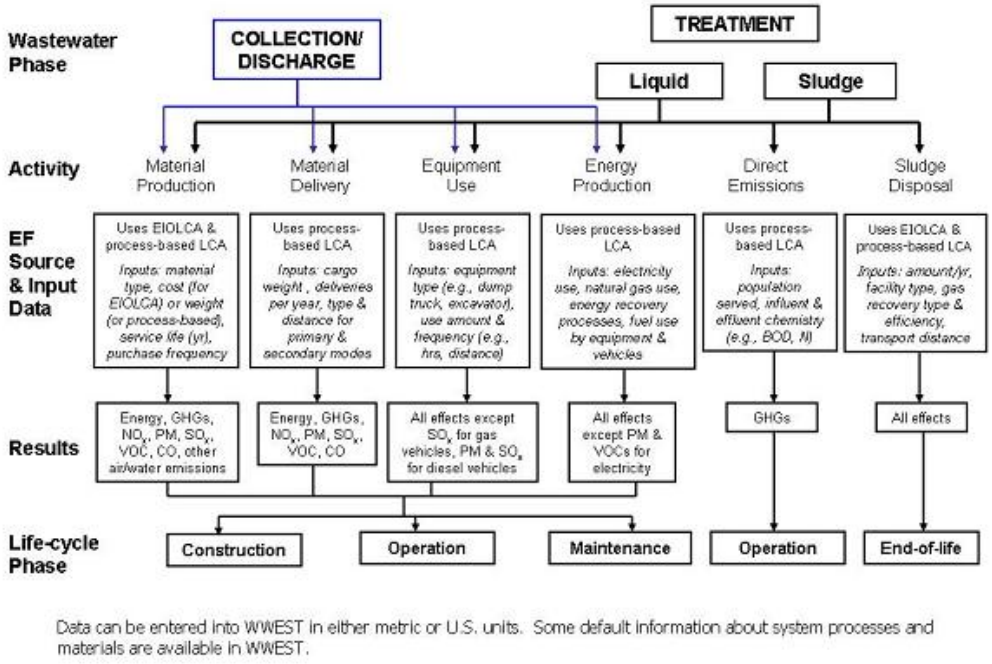


Figure 3.1: WWEST Model System Boundaries

The WWEST LCI database can be accessed in two ways. The first, which provides full access to all of the attribute fields contained within the database, is via an excel spreadsheet that has been distributed with a built-in set of computational macros. The second method exposes a more limited range of the data values contained within the database and is accessible via a streamlined web application, also known as WWESTweb. The principle difference between the WWEST excel model and the web application is in the extent to which each allows the user to customize the various parameters associated with a hypothetic water reuse project and its supporting infrastructure. In this sense, the excel based model is more appropriate for building an LCI model to quantify the process flows attributable to an existing facility or which is in a very advanced stage of design planning. Conversely, the WWESTweb framework is more appropriate for less well defined scenario based planning exercises, where the primary goal is to assess the relative impact of major system design alternatives that have only roughly been fleshed out.

3.6 Parameterizing Pump Energy Requirements

The WWEST model requires a number of key input parameters to be supplied by the user before it can be run. These parameters correspond to various attributes of the proposed reuse system including: pipe length and material information derived from the distribution network topology, estimates of distribution process energy consumption figures, and process based chemical inputs associated with any required tertiary treatment phases.

In order to facilitate the systematic parameterization of the WWEST model a programmatic workflow was developed which automates the calculation of a number of parameters from two input data sources: (1) the topology of the distribution network (i.e. the corridor

solution from the MOGADOR optimization model) and (2) the expected daily flow rate derived from the maximum permitted flow value assigned to each WWTP facility through the NPDES permitting program.

To illustrate how this workflow functions, take for example the process of calculating the expected energy requirements associated with the distribution of the treated wastewater from its source at that WWTP to its destination at the reuse facility. Estimating the instantaneous power output associated with the operation of a pipeline based water distribution system begins with Equation 3.1[36].

$$P = \frac{Q * H_t * g * \varrho}{E} \tag{3.1}$$

Where:

$$P = \text{The instantaneous pump energy } (W)$$
$$Q = \text{The instantaneous flow rate } (m^3/s)$$
$$H_t = \text{The total head } (m)$$
$$g = \text{The gravitational constant } (m/s^2)$$
$$\varrho = \text{The density of the fluid } (Kg/m^3)$$
$$E = \text{The pump efficiency factor } (unitless)$$

The first term in this expression (Q), the instantaneous flow rate, can be computed by dividing the annual volume of processes by the WWTP by the number of seconds in a year. The second term (H_t), the total system head, can be computed summing the individual static and dynamic head components as in the following Equations 3.2 and 3.3[36]. The last

three terms ($g, \varrho, \& E$) are constants that are characteristic to the system.

$$H_s = e_i - e_o \tag{3.2}$$

Where:

$$H_s = \text{The static head component}$$
$$e_i = \text{The elevation at pipeline inlet}$$
$$e_o = \text{The levation at pipeline output}$$

Equation 3.2 describes how the static head can be computed as the difference between the elevations at the inlet and the outlet locations for the pipeline. When the corridor specification is known, this difference can be straightforwardly assessed by referencing the first and last subscript indices of the corridor to a raster based digital elevation model.

Computing the dynamic component (H_d) of the total system head is a significantly more complicated process; however, it can, nonetheless, be similarly automated from the same detailed knowledge regarding the pipeline corridor specification and its underlying elevation profile. Equations 3.3 through 3.7 illustrate the sequence of operations by which dynamic head (H_d) can be computed[36].

$$H_d = \frac{K_t * V^2}{2g} \tag{3.3}$$

Where:

$$H_d = \text{The dynamic head component}$$
$$K_t = \text{The total system losses}$$

$$V = \text{The flow velocity}$$

The dynamic head term is a representation of the frictional forces that arise from the movement of water through the pipeline. These cumulative effective of these forces are represented by the term (K_t), total losses, and are multiplied by the square of the velocity at which the water is flowing. The (K_t) can be parsed into two separate components, $(K_p \& K_f)$, as shown in Equation 3.4[36]. These two terms represent the relative contribution of the friction associated with the movement of water over the textured surface of the pipeline interior walls and the friction associated with the movement of water through a tortuous pipeline that has various connective fittings such as elbow joints and flow control valves.

$$K_t = K_p + K_f \tag{3.4}$$

Where:

$$K_p = \text{The pipe loss component } (\textit{unitless})$$
$$K_f = \text{The fitting loss component } (\textit{unitless})$$

Typically, the velocity term (V) is computed from the product of a specific flow rate (Q) and pipeline cross sectional area (A) as in Equation 3.5. However, in the case of this analysis, the pipeline cross sectional area was solved for by specifying a maximum permissible flow velocity $V_{max} = (10 m/s)$ relative to some designated flow rate[36].

$$V = \frac{Q}{A} \tag{3.5}$$

Where:

$$A = \text{The cross sectional area of the pipe } (m^2)$$

The component of the total losses attributable to the cumulative friction encounter along the pipeline's pipe section walls (K_p) can be calculated from Equation 3.6 as the product of a friction coefficient (f) and the pipeline length (L) divided by the diameter of the pipeline pipe sections (D)[36].

$$K_p = \frac{f * L}{D} \tag{3.6}$$

Where:

$$f = \text{The friction coefficient of the pipe } (\textit{unitless})$$
$$L = \text{The cumulative length of all the pipe sections } (m)$$
$$D = \text{The diameter of each pipe section } (m)$$

The friction component (f) of the pipeline loss term (K_p) can be computed from the empirically derived Equation 3.7 where (k) is a roughness factor that is characteristic to the pipe section material construction and (Re) is the Reynolds number which is a dimensionless quantity that this associated to the smoothness with which a fluid flows and can be derived from the fluid's characteristic kinematic viscosity (υ) as in Equation 3.8[36].

$$f = \frac{0.25}{\left\{\log_{10} \frac{k}{3.75 * D} + \frac{5.74}{Re^{0.9}}\right\}^2} \tag{3.7}$$

Where:

$$k = \text{The roughness factor of the pipe material } (m)$$
$$Re = \text{The Reynolds Number of the fluid } (\textit{unitless})$$

$$Re = \frac{V * D}{\upsilon} \tag{3.8}$$

Where:

$$\upsilon = \text{The kinematic viscosity of the fluid } (m^2/s)$$

The fitting losses (K_f), which makes up the second component of the total losses expression (K_t) and which ultimately feeds into the calculation of dynamic head (H_d), can be computed by using the corridor specification to cumulatively assess the need for various fitting components to facilitate pipeline deviations and flow control systems. The contribution of each fitting component to the total (K_f) factor is empirically defined and can be iterative summing the contribution of each fitting (K_n), multiplied by its appropriate fitting loss factor (K'_n), along the entire length of the pipeline corridor as in Equation 3.9[36].

$$K_f = \sum_{i=1}^{n} \{(K_1 * K'_1) + \cdots + (K_n * K'_n)\} \tag{3.9}$$

Where:

$$K_n = \text{The individual fitting loss component } (\textit{unitless})$$
$$K'_n = \text{The individual fitting loss factor } (\textit{unitless})$$

3.7 Parameterizing Construction Material Requirements

Once the pump energy associated with the corridor specification has been computed according to method laid out in Equations 3.1 - 3.9 the next step is to estimate the volume of

reinforced concrete that must be poured to facilitate the construction of the requisite infrastructural components of the proposed reuse system. It is assumed for the purpose of this analysis that these new infrastructural components will principally be associated with the need to install one or more pumping houses which will be situated along the length of the corridor[36]. The determination of the size of each of these facilities and their concomitant material footprints will be determined on the basis of the total pump energy required for each corridor and the structure of the different elevation profiles. Consideration will be given to economies of scale in terms of the efficiency that can be achieved through the operation of larger pump systems.

3.8 Parameterizing Chemical Consumption Rates

The third key set of parameters that must be provided to the WWEST model are the volumes of any consumable chemical substances that must be provided to facilitate the tertiary treatment processes associated with the proposed reuse system. Ideally, the determination of these parameters would be made on the basis of the relative quality of the influent wastewater to that of the effluent treated water. However, the data requirements for this level of specificity in the model parameterization are significant and thus were left outside the scope of this analysis. As a result, all of the input parameter settings for the chemical consumption rates of the tertiary treatment processes were set to be equal to average values in the literature for a standard suite of treatment processes. These parameter settings can be found in the Parameter input tables contained within Appendix ??.

3.9 Estimating Net Water Usage Efficiency

The final step towards the research program's overarching goal of estimating the life-cycle energy-water usage efficiency of proposed new reuse systems involves converting life-cycle energy consumption into predicted water consumption. This conversion can be accomplished by applying known water usage efficiency factors for a suite of energy generation technologies to the local grid mix responsible for producing the energy that will be supplied to the reuse system over its life-cycle. For the purpose of this analysis, the calculated metric will focus on the consumptive use of water for energy production as opposed to the non-consumptive use. This designation is important as the difference between the consumptive and non-consumptive water use profiles for a number of energy generation, and in particular, cooling technologies, can be non-trivial.

While the distribution of energy generation technologies is known from published statistics on the local grid mix associated with each of the five case study regions, a more granular breakdown of the precise thermoelectric cooling technology used by each category of energy producer is not known. In the absence of this detailed information a range of energy usage efficiencies – corresponding to the min-max range of cooling technology efficiencies for each production method – will be generated by the analysis, providing bounds to our uncertainty in the final calculated values.

3.10 The WWEST Model Outputs

The final output of the WWEST based calculation will involve three numbers. The first corresponds to the volume of wastewater that is to be treated and recycled by the proposed

reuse operation over the course of a given year. The second corresponds to the life-cycle energy requirements attributable to this volume of water reuse each year at the site in question. The third number will be the quantity of water that must be consumed in order to generate the energy associated with the system's annualized life-cycle energy requirements. By computing the ratio of the second number to the third we will thus be left with an estimated ratio depicting the life-cycle energy-water usage efficiency of the proposed reuse system. If this ratio is greater than one it means that the proposed system is highly efficient, with more water being saved within the local basin each year than is consumed – likely elsewhere, outside the basin – to produce the energy associated with its operation. Similarly, if this ratio is greater than zero but less than one, the system is only partly efficient. And finally, if this ratio is less than zero, the system is highly inefficient, with more water being expended to the produce the energy required for the reuse operation that is saved by the operation of the reuse system. This third condition would essentially amount to a situation where water was being virtually imported into the basin in the form of the energy consumed by the reuse system.

In god we trust. All others bring data.

William Edwards Demming (1900-1993)

4

Case Study Results

4.1 Santa Barbara Region

The first case study region, comprising the coastal southern portion of Santa Barbara County, was selected to reflect the local interests of the institution supporting this dissertation research. Hydrologically, this case study region is distinctly enclosed by a steep coastal mountain range to the north and the pacific ocean to the south. This case study area is not connected to any of the major inter-basin water transfer projects within the state (i.e. The State Water Project, the Los Angeles aqueduct, etc.). As such, Santa Barbara municipal water managers must be both creative and self reliant in terms of their long term municipal water supply strategies.

Fortunately, from a freshwater management perspective, the region's unique physical geography also functions to limit the possibilities for increased population growth and urban development. Thus, the prospects for severe water shortages due to steep increases in demand are fairly unlikely. Despite this fact however, the recent drought condition throughout the state have lead to high wholesale water costs for the Santa Barbara district. This is because they are in competition with regional agricultural interests with long term investments in costly orchard based crops that cannot be left to fallow.

In terms of alternative water supply options within the region, Santa Barbara has recently renewed talks for the development of a local seawater desalination plant that had been put on hold following the 2008 economic recession. This willingness to reconsider a high cost desalination based alternative freshwater supply strategy suggests that large scale municipal water reuse may also be put forth as a feasible alternative in the near term future and thus, that such a prospective analysis of the tradeoffs associated with such a system

would indeed be valuable exercise.

4.1.1 Regional Context

- HUC-8 Code: 18060013
- Total Area: 1, 173.6 km^2
- Maximum Elevation: 1, 376.7 m
- Minimum Elevation: -0.7 m
- Mean Slope: 13.98 %
- Standard Deviation of Slope: 11.07 %
- Dominant Soil Composition: Hydrologic Soil Group - B: 10 – 20% clay, 50 – 90% sand, 35% rock fragments

Figure 4.1: Santa Barbara Region Overview (Filled in Black)

4.1.2 Search Domain

The search domain used for both the weighted overlay site suitability analysis as well as the corridor location problem specification is depicted in Figure 4.2. The extent and dimensions of this search domain is depicted in the statistics below.

- Grid Dimensions: 363 *cells* x 1351 *cells*
- Grid Cell Resolution: 100 *m* x 100 *m* (1 *ha*)
- Feasible Grid Cells: 117, 363 *cells*

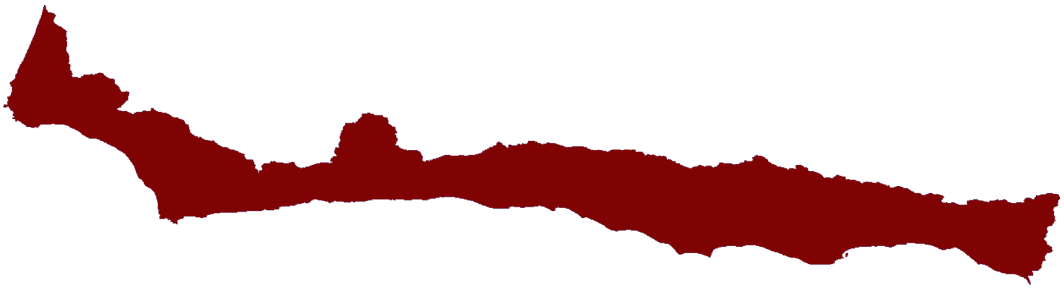


Figure 4.2: Santa Barbara Region Search Domain (Filled in Red)

4.1.3 Destination Search Inputs

There are three key inputs to the weighted overlay analysis used to determine the location and extent of suitable sites for the implementation of artificial groundwater recharge basins within the region. The four layers which were generated as the discrete inputs to the WOA procedure are depicted in Figures 4.3 through 4.5. The first layer gives each cell in the search domain a score between 1 and 10 on the basis of the suitability of its slope for the implementation of a artificial groundwater recharge basin. Areas with steep slopes are given lower suitability scores. Areas with shallower slopes are given high suitability scores.

The second input to the WOA destination search process is based upon the permeability of the surface geology as shown in Figure 4.4. Permeability is a crucial parameter in determining the rate of infiltration that can be achieved by a recharge basin and thus the requisite size of a basin for the purpose achieving a specified total rate of recharge. The geology score layer gives each cell in the search domain a ordinal score between 1 and 10 on the basis of the underlying surface geology layer's permeability constant.

The final input to the WOA destination search process is based upon the existing landuse as shown in Figure 4.5. The existing landuse can be a proxy measure of both the cost of procurement for the landholdings required to implement the artificial recharge basin as well as the regulatory and engineering difficulty associated with artificial recharge basin implementation. Here again, these scores are have been pegged to a 1 to 10 ordinal scale that aligns with those assigned to each of the other two score layers.

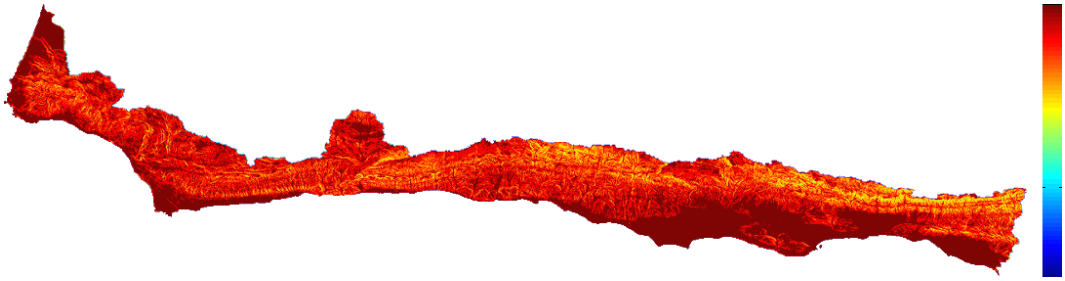


Figure 4.3: Santa Barbara Region Destination Search Inputs: Slope Score (Blue:Low, Red:High)

4.1.4 Destination Search Outputs

The raw output of the WOA destination search process is a composite layer of which depicts a measure of overall suitability for the given landuse application on an ordinal scale as in Figure 4.6. This single composite suitability layer is then thresholded, selecting only

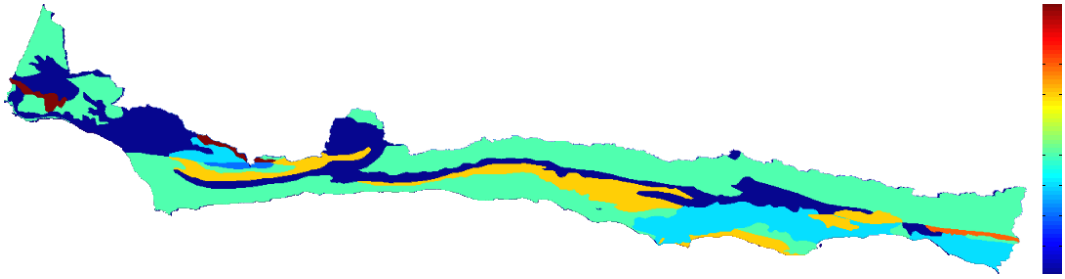


Figure 4.4: Santa Barbara Region Destination Search Inputs: Geology Score (Blue:Low, Red:High)

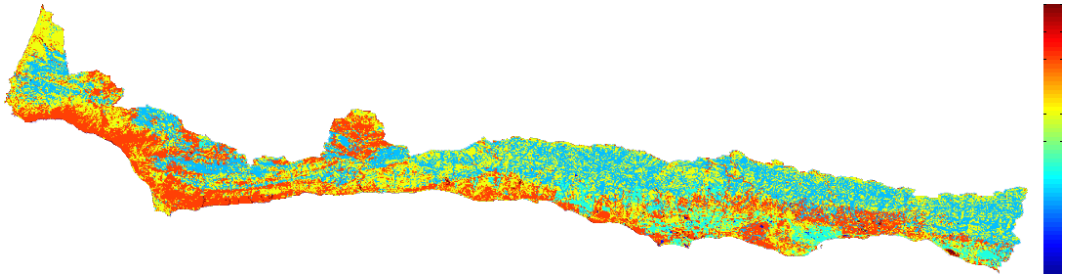


Figure 4.5: Santa Barbara Region Destination Search Inputs: Landuse Score (Blue:Low, Red:High)

those areas that have the highest composite suitability scores as shown in Figure 4.7. A set of morphological operations is applied to this threshold mask which ranks each connected area of high suitability in terms of its size. Larger connected areas of high suitability are considered better in this process and thus, in this way, a single destination location for the corridor search process can be automatically selected as the single largest area of high aggregate suitability with the study area.

4.1.5 Proposed Corridor Endpoints

For the Santa Barbara case study region, the final output of the WOA analysis is shown in Figure 4.8 in red and mapped relative to the location of the source location for the corridor location analysis that corresponds to the location of the largest WWTP within the basin, in

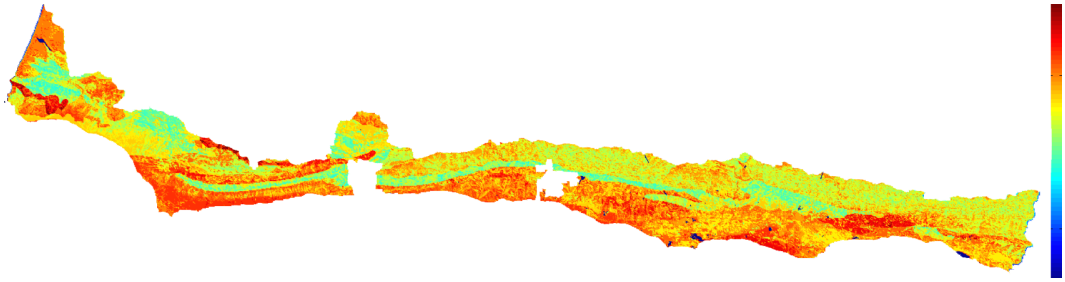


Figure 4.6: Santa Barbara Region Destination Search Outputs: Composite Scores (Blue:Low, Red:High)

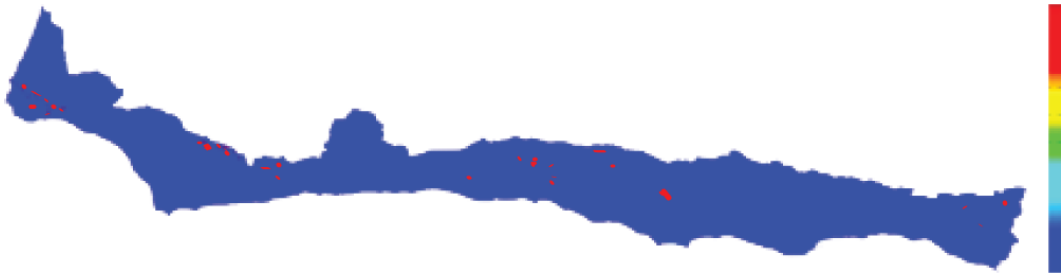


Figure 4.7: Santa Barbara Region Destination Search Outputs: Candidate Regions

green. These two points, plus the extent of the search domain, form the basis of the corridor location problem specification that is to be discussed in further detail in the subsequent section.

- Start Location: $(313, 1083)$
- End Destination: $(248, 886)$
- Shortest Euclidean Path Distance: $20,745\ m\ (21\ km)$

4.1.6 Proposed Objective Layers

For the corridor location problem specification used as the input to the MOGADOR algorithm, four key pieces of information are required. The first three correspond to the source

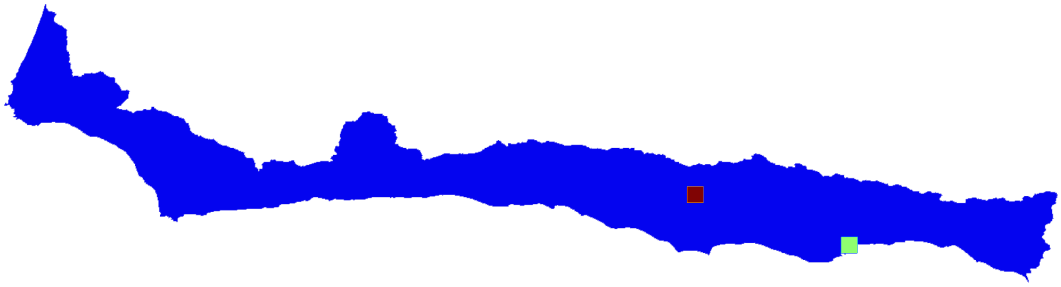


Figure 4.8: Santa Barbara Region Proposed Corridor Endpoints (Green:Source, Red:Destination

location, the destination location, and the search domain boundaries that have been previously described. The forth key input category corresponds to the objective score layers which capture the cost associated with routing sections of a corridor over each grid cell in the search domain. For this analysis, the following three distinct objectives were developed.

The first objective category is based upon the accessibility of each location for the purposes of constructing and maintaining the water distribution infrastructure that the corridor is designed to support and is shown in Figure 4.9. It is fundamentally easier to get materials and people to a locations that are positioned along road networks. As a result, the underlying road network topology was used to encode a continuous objective score layer with values ranging from 1 to 10 that can be described as a measure of "Accessibility" and which favors those locations that are on and around roads.

The second objective category is based upon the existing land use regime within the regional search domain. The idea behind the composition of this objective can be thought of as somewhat of the converse of Accessibility in the sense that, regions which are already heavily developed are likely to be socially, politically, or economically challenging to implement corridors for large scale water distribution pipeline infrastructure. Using standardized USGS based land use classification, each grid cell in the search domain is given a nominal

objective score value from 1 to 10 corresponding the relative level of "Disturbance" that would be associated with routing a corridor across it. This objective layer is depicted in the layer plotted in Figure 4.10.

The third objective category is derived from the underlying slope within the search domain. Steeper slopes are assigned a higher ordinal score, ranging from 1 to 10. This objective reflects the desire for corridors to be shorter in length and minimally accumulate slopes over their length. In this way, the slope score provides a mechanism for the corridor routing algorithm to preferentially favor corridors that would have minimal energy requirements in terms of the operational energy requirements of the anticipated water distribution infrastructure. This slope score objective layer is depicted in Figure 4.11.

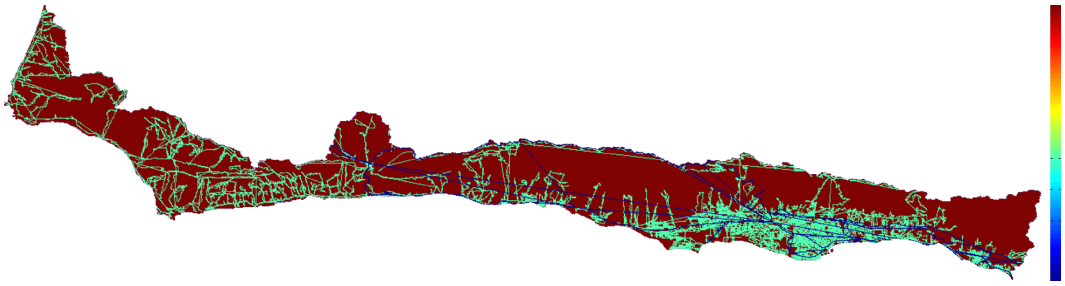


Figure 4.9: Santa Barbara Region Accessibility Based Objective Scores (Blue:Low, Red:High)

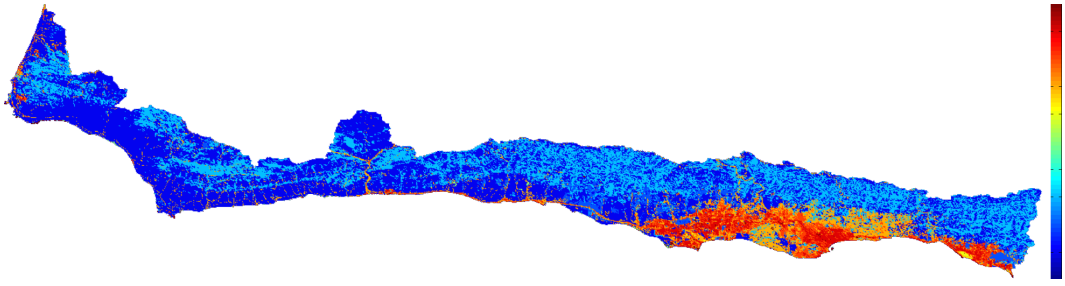


Figure 4.10: Santa Barbara Region Land Use Disturbance Based Objective Scores (Blue:Low, Red:High)

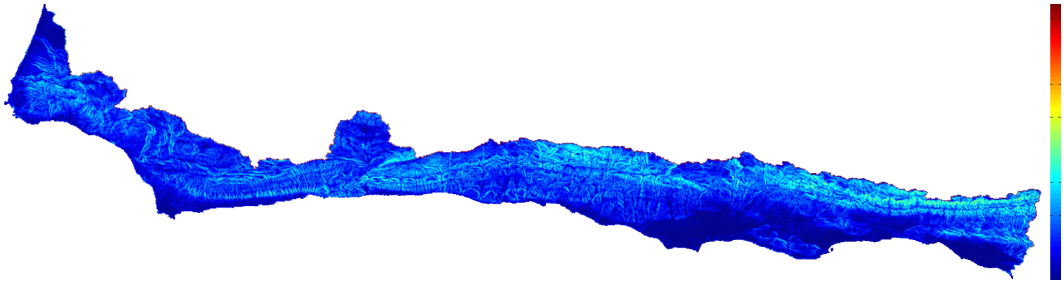


Figure 4.11: Santa Barbara Region Slope Based Objective Scores (Blue:Low, Red:High)

4.1.7 Proposed Corridor Solutions

Shown in Figure 4.12 are the outputs of a series of three runs of the MOGADOR algorithm for the Santa Barbara region problem specification. These three runs differ solely in terms of the number of individuals contained within the seed population. The size of this seed population determines the extent with which the input search domain is search and, consequently, the degree to which the output solution corridor is likely to approximate the global optimal solution. The figure contains six panes made up of three rows and three columns. The columns depict, from left to right, and plan view of the final output solution set, and line plot of the break down of objective scores for the top 100 ranked individuals in the final output solution set, and, finally, a histogram plot of the frequency of total aggregate objective scores among the same top 100 individuals. Alternatively, the rows, moving from top to bottom, reflect the changing results as the population size is increased from 1,000 to 10,000 to 100,000.

As the histogram plots of the aggregate objective scores illustrate, with a population size of 100,000 the aggregate objective scores are quite low, and the quality of the final output solution set is very high. This improvement in solution quality comes at the expense of

processing time/effort. This tradeoff shall be discussed in greater detail and illustrated comparatively across all of the five case studies at the end of this chapter.

One interesting feature of this exercise which can be readily appreciated from this set of plots is the source of the improvement in the aggregate objective scores between the different runs. For example, note the height of the data series depicted by the blue line, corresponding to the accessibility score, in the three plots in the middle column. The progressive decrease in the values associated with this line indicates that the reduction in aggregate objective scores between the three runs can be attributed to a reduction in the Accessibility score. This is tantamount to saying that the search process is able to provide better solutions as it "Finds the road network." And indeed, this conclusion is reflected from a simple visual inspection of the output corridors plotted in the panels contained in the first column. Here it can be seen that in the 100,000 population size solution set, the pathway sections have become much more linear, and appear to correspond with the layout of different road segments which occupy the area in between the source and the destination.

Figure 4.14 provides an illustration of the top ranked final output corridor solution presented in the context of the full search domain. For all of the case studies the highest quality solution was developed by the model run containing the largest seed population. This was fully expected however and is in good agreement with the theoretical discussion of the role of the population initialization procedure in the behavior of the MOGADOR algorithm described in Chapter 3. l

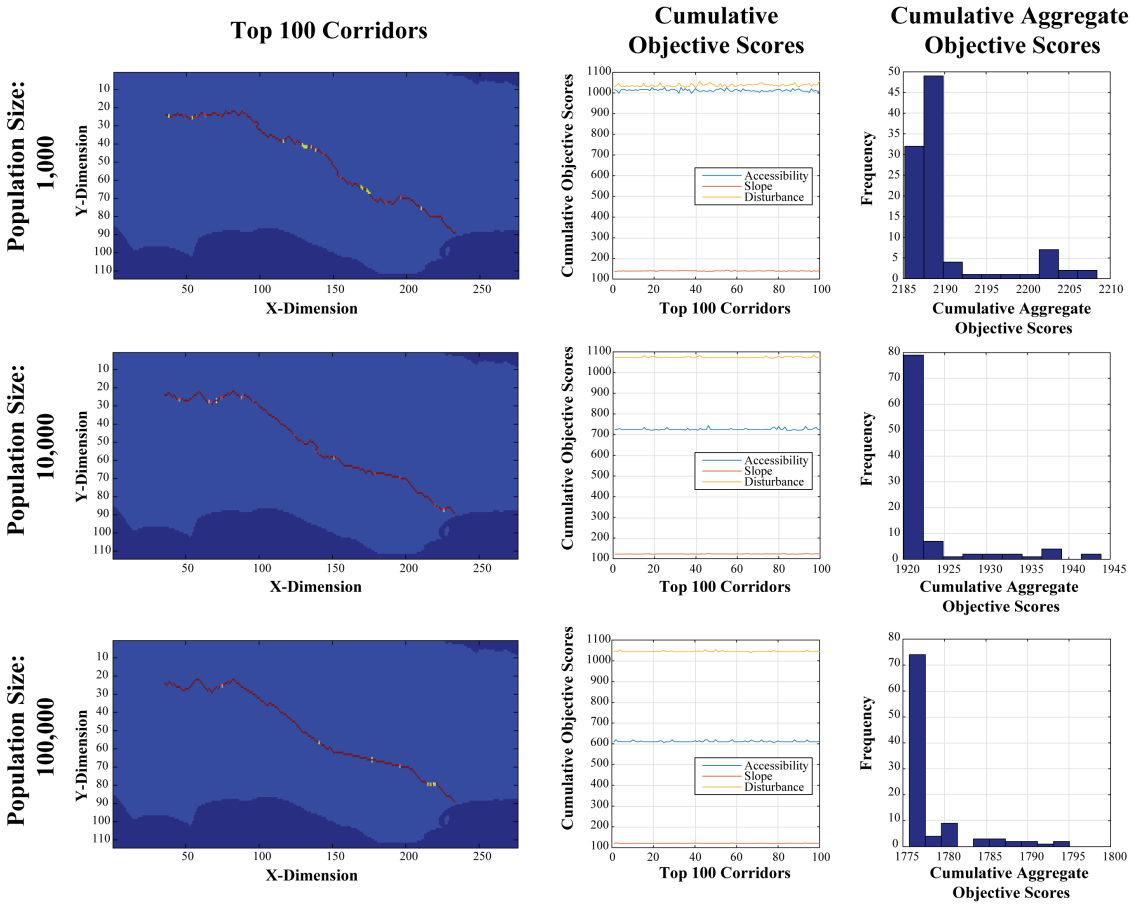


Figure 4.12: Santa Barbara Region Corridor Analysis Results

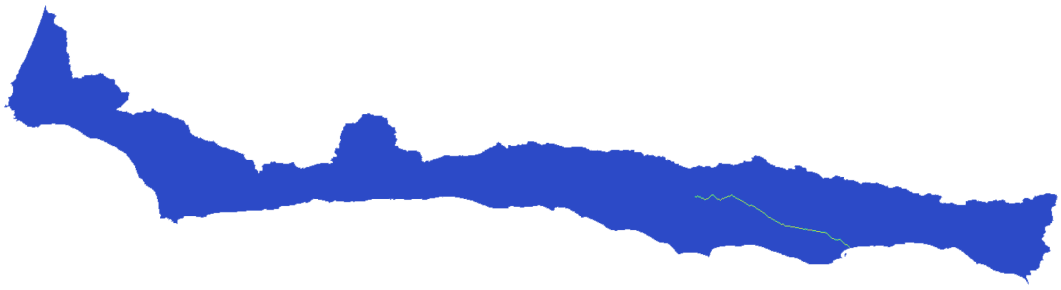


Figure 4.13: Santa Barbara Region Top 100 Corridors (Pop Size: 100,000) Basin Wide Overview

4.1.8 Along-Corridor Elevation Profile

Figure 4.14 illustrates the along-corridor elevation profile that can be generated by superimposing the output corridor solution on top of a regional digital elevation model for the Santa Barbara region. As the Figure shows the total elevation gain between the source and the destination location is a modest 200 meters across a distance spread of roughly 25 kilometers. While it may appear that the corridor has a significant amount of vertical fluctuations, these are minor in absolute terms, and stem from the fact that the slope score – the objective most directly related to the corridor elevation profile structure – was but only one of three in the multi-objective problem statement. These elevation fluctuations therefore can be thought of as the result of a profitable tradeoff between the accumulation of smoother slopes and more favorable values for the other two objectives.

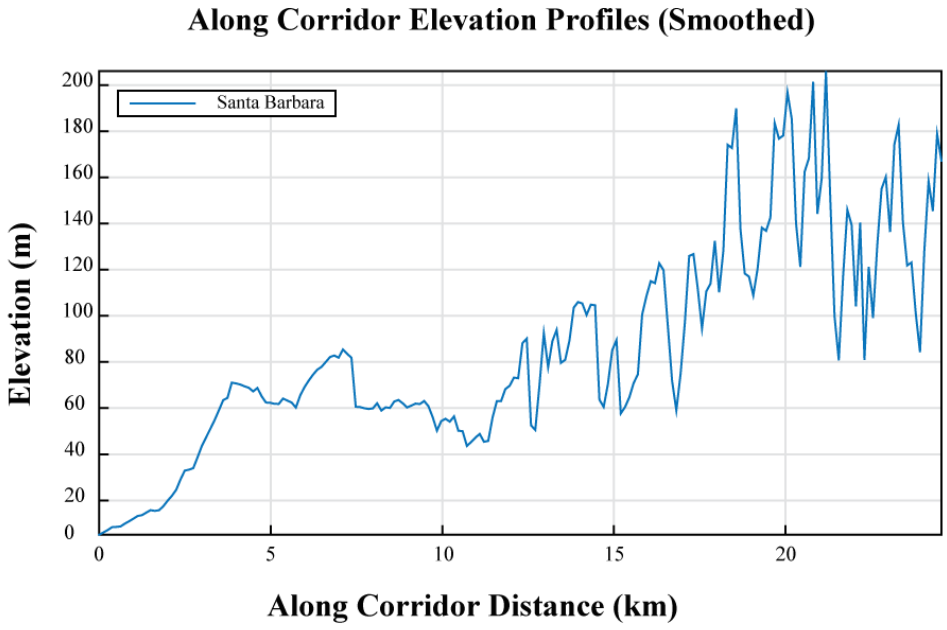


Figure 4.14: Santa Barbara Region Proposed Corridor Elevation Profile

4.2 Oxnard Region

The second case study region consists of the HUC-8 zone containing the Oxnard plain and its immediate surrounding territories stretching as far inland as Ojai. This second case study region is located nearly adjacent to the first – separated only by a single HUC-8 basin. The reason for the choice of two case study regions in such close physical proximity is down to the recent implementation of a functioning large scale water reuse system by the municipal water management district there.

The majority of this HUC-8 zone's area is a comprised of a broad low lying alluvial plain. This plain has found rich application within the agricultural sector, supporting the production of a wide variety of row crops as well as high value orchard stands. Over the past three decades, the region has also experience significant population growth with sprawling suburban communities encroaching into the more marginal farmlands or those held by smaller independent farmers. The combined freshwater demands of these two sectors have conspired to create a persistent imbalance between freshwater supply and demand in this coastal region.

Oxnard's struggle with freshwater management issues can be traced as far back as 1937 when the USGS identified that sustained groundwater pumping to support the irrigation of surface crops was contributing to the depletion of the underlying aquifer and inviting the intrusion of brackish seawater into the subsurface hydrologic strata. In response to this issue, the local municipal water authority enacted a program in which a portion of the regions' agricultural water was diverted towards a series of subsurface injection wells – strategically positioned along the coast – through which freshwater would be pumped to created

an artificial pressure head barrier to prevent further intrusion of seawater, and thus further contamination of the aquifer.

This program has operated successfully for a number of decades now; achieving a functional equilibrium between the level of groundwater pumping occurring within the basin and the amount of water the is delivered to the artificial intrusion barrier. More recently however, decreases in available freshwater supply due to a persistent statewide drought have forced municipal water managers in this region to thing more proactively about developing alternative sources of water supply. This process began with the creation of a plant to substitute potable freshwater for reclaimed brackish water for use in the subsurface barrier injection wells.

The successful operation of this plant for a number of years inspired enough confidence among the water resource management authorities in this area to pursue and very recently achieve a goal of implementing a facility capable of reclaiming and reusing the growing volume of municipal wastewater being generated within the basin. This new facility, commissioned just this year, provides the capability to treat 100% of the wastewater generated in the basin to a potable standard through a complex treatment chain incorporating a sophisticated chain of tertiary treatment processes including: advanced micro-filtration, reverse osmosis, ultraviolet filtration, and ozonation. The long term plan for the water currently being produced by this facility is for groundwater recharge at higher elevation locations within the basin. As such, this locale represents the ideal candidate for evaluation in this study.

4.2.1 Regional Context

- HUC-8 Code: 18070102
- Total Area: $5,188.3\ km^2$
- Maximum Elevation: $2,664.4\ m$
- Minimum Elevation: $-0.05\ m$
- Mean Slope: 15.54 %
- Standard Deviation of Slope: 11.11 %
- Dominant Soil Composition: Hydrologic Soil Group - B: $10-20\%$ clay, $50-90\%$ sand, 35% rock fragments

Figure 4.15: Oxnard Region Overview (Filled in Black)

4.2.2 Search Domain

The search domain comprising the Oxnard study region is described in the statistics below and depicted graphically in the map panel contained within 4.16. Relative to the total land area contained within the Santa Barbara study region, the Oxnard domain is quite large, being nearly four times its total size.

- Grid Dimensions: 677 *cells* x 1586 *cells*
- Grid Cell Resolution: 100 *m* x 100 *m* (1 *ha*)
- Feasible Grid Cells: 518, 834 *cells*

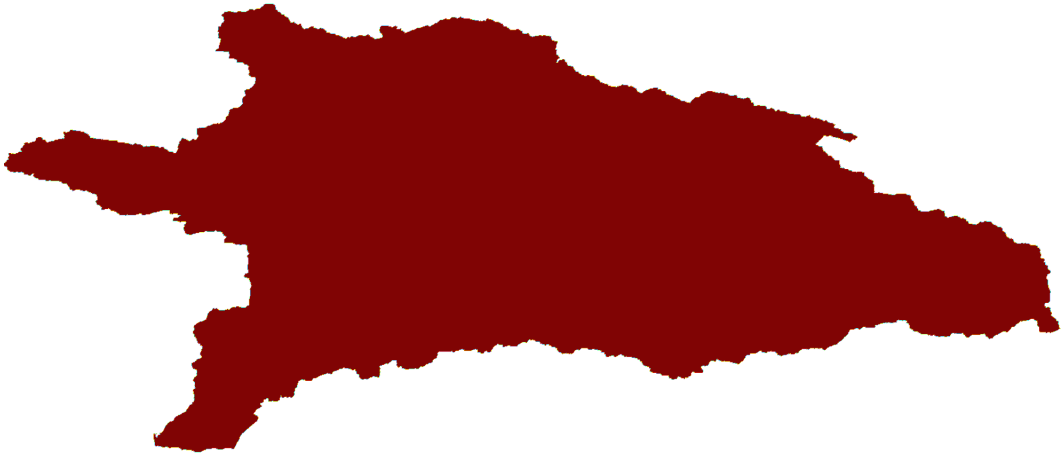


Figure 4.16: Oxnard Region Search Domain (Filled in Red)

4.2.3 Destination Search Inputs

In Figures 4.17 through 4.19 the three key inputs to the Oxnard reuse destination search process are shown. A visual inspection of these three layers reveals that there is an obvious

band of continuously high suitability stretching from the foot of the basin (at the lower left) along its lower portion nearly across its breadth (to the lower right). This corridor is flat low lying river bed. It possesses a highly permeable surface geology, a very shallow slope profile, and relatively low intensity land use applications.

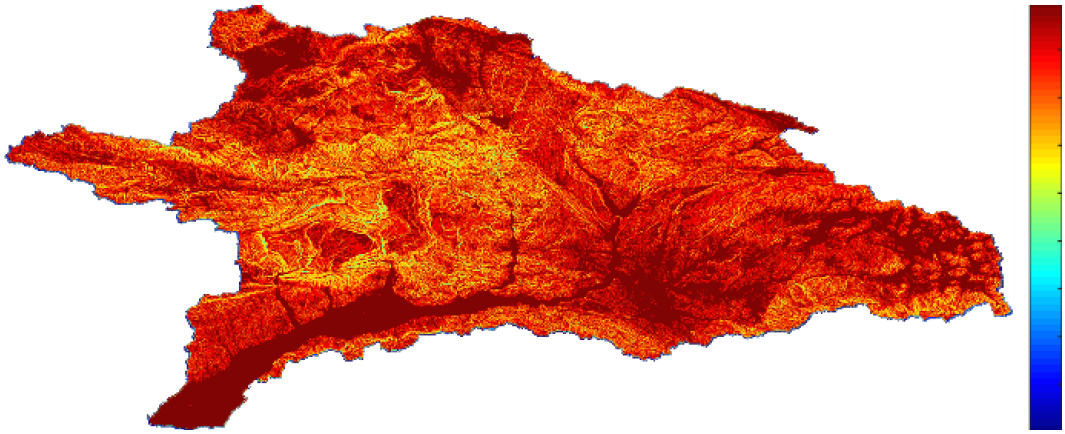


Figure 4.17: Oxnard Region Destination Search Inputs: Slope Score (Blue:Low, Red:High)

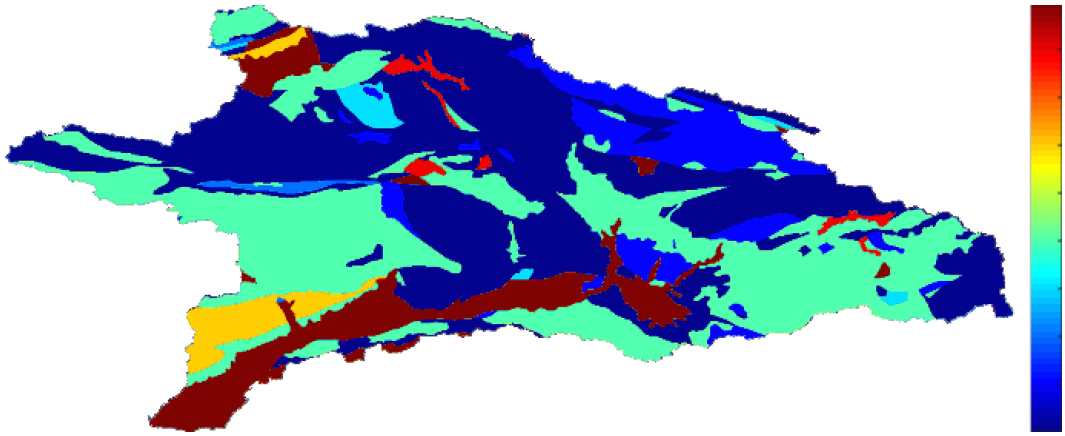


Figure 4.18: Oxnard Region Destination Search Inputs: Geology Score (Blue:Low, Red:High)

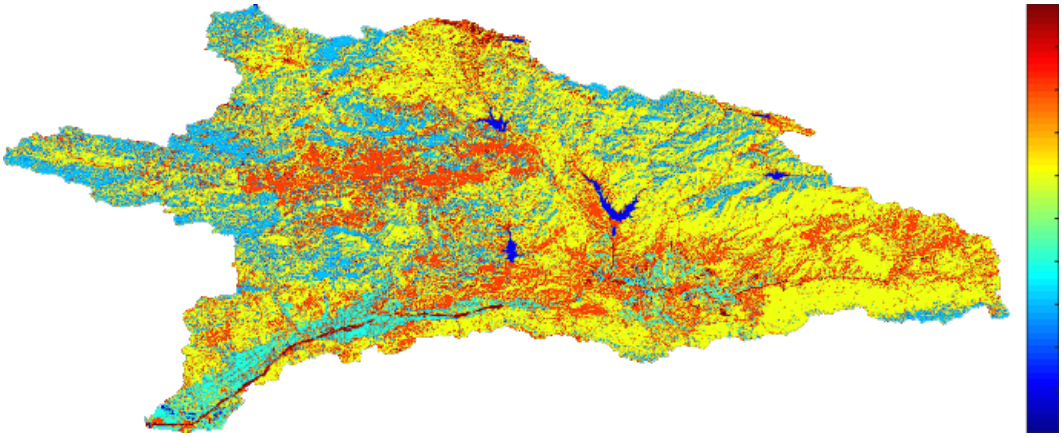


Figure 4.19: Oxnard Region Destination Search Inputs: Landuse Score (Blue:Low, Red:High)

4.2.4 Destination Search Outputs

The layer showing the composite suitability of each cell within the study site for the site of a destination artificial water reuse facility is shown in Figure 4.20. As this figure shows, the area with the highest composite suitability is that which was mentioned previously as being clearly visible within each of the individual input suitability layers. The majority of the top ranked areas of contiguous high suitability are contained within this region, as shown in Figure 4.21.

4.2.5 Proposed Corridor Endpoints

For the Oxnard case study region, the final output of the WOA analysis is shown in Figure 4.22 in red and mapped relative to the location of the source location for the corridor location analysis that corresponds to the location of the largest WWTP within the basin, in green. These two points, plus the extent of the search domain, form the basis of the corridor location problem specification that is to be discussed in further detail in the subsequent

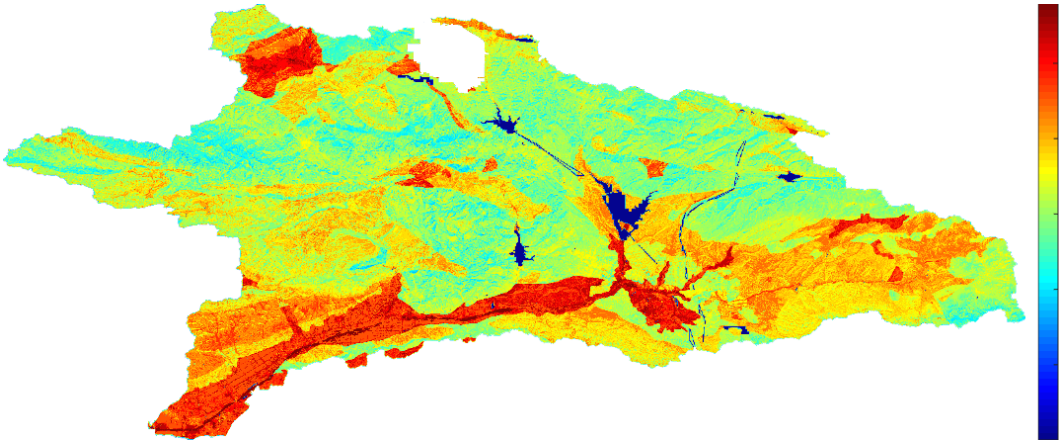


Figure 4.20: Oxnard Region Destination Search Outputs: Composite Scores (Blue:Low, Red:High)

Figure 4.21: Oxnard Region Destination Search Outputs: Candidate Regions

section.

- Start Location: $(656, 236)$
- End Destination: $(513, 532)$
- Shortest Euclidean Path Distance: $32,873\ m\ (32\ km)$

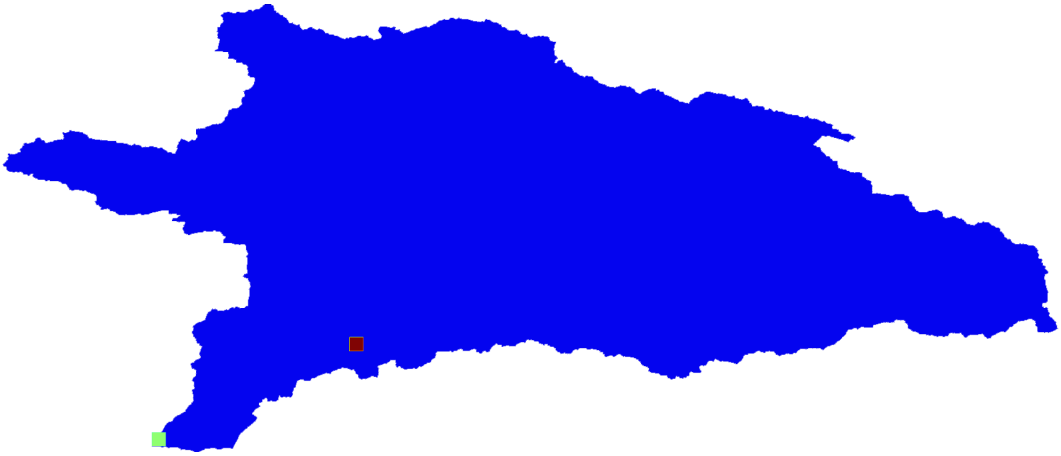


Figure 4.22: Oxnard Region Proposed Corridor Endpoints

4.2.6 Proposed Objective Layers

In Figures ?? through 4.25 the three independent objective layers used as inputs to the MOGADOR problem specification for the Oxnard study site are shown. These three layers correspond to the categories of landuse disturbance, accessibility, and slope described previously for the Santa Barbara case study region and used for all of the other case studies in the analysis.

4.2.7 Proposed Corridor Solutions

Figure 4.26 presents a figure panel containing the outputs of the three separate MOGADOR algorithm runs for the Oxnard study site problem specification using three different population sizes. As this figure panel illustrates, the first algorithm run, with a population size of 1,000, delivered a set of 100 top output corridor solutions with aggregate objective score values ranging from 3585 to 3615. With the second run of the algorithm, where the population

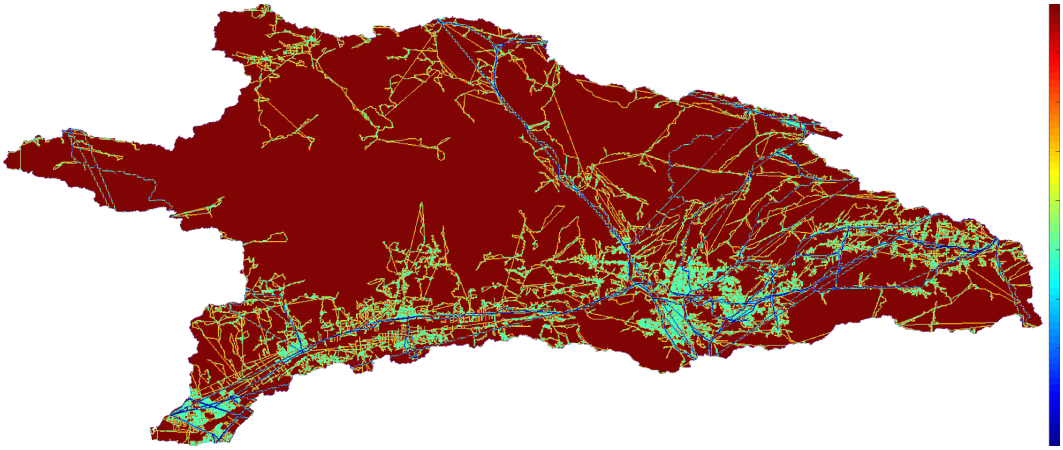


Figure 4.23: Oxnard Region Accessibility Based Objective Scores (Blue:Low, Red:High)

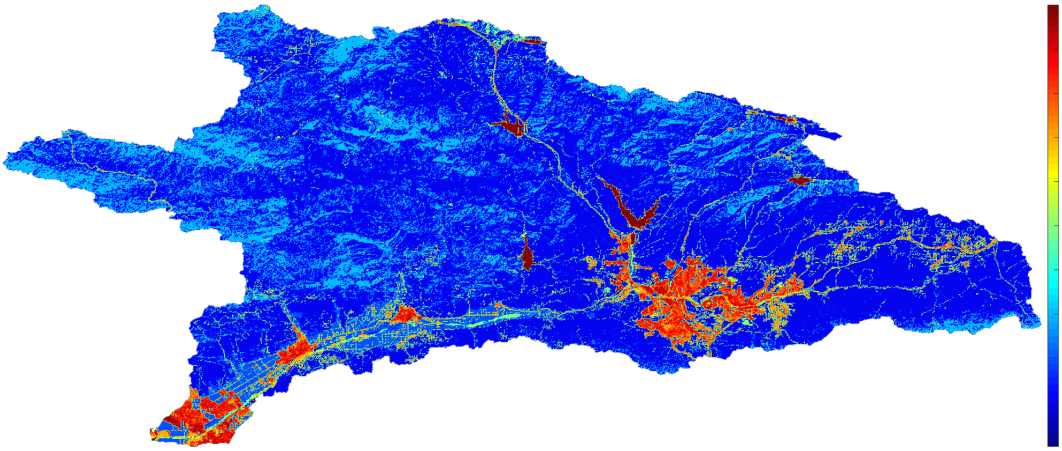


Figure 4.24: Oxnard Region Land Use Based Disturbance Objective Scores (Blue:Low, Red:High)

size was increased to a 10,000, the top 100 output corridor solutions' aggregate objective scores can be observed to have improved markedly, covering a range from 2910 to 2935. The line plot in the center of the figure attests to the fact that this improvement came from reductions in both the accessibility and disturbance scores associated with the new output corridor set.

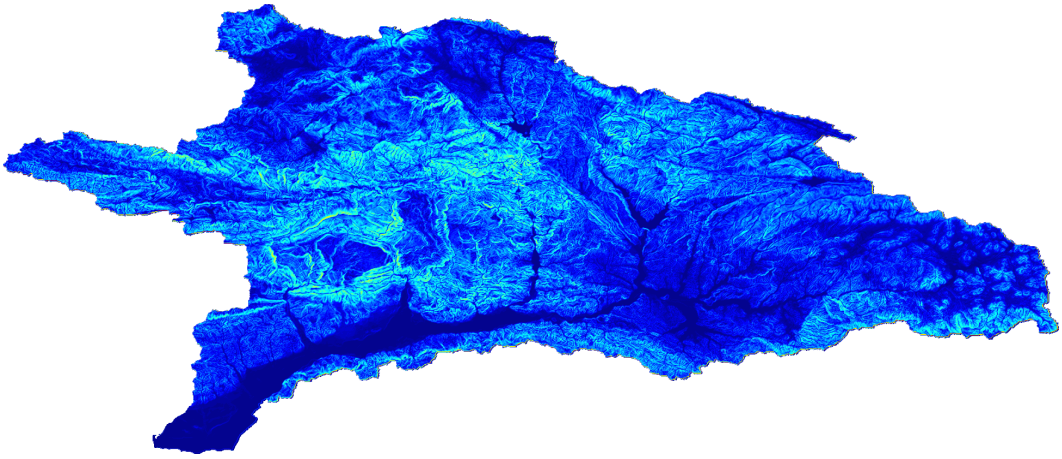


Figure 4.25: Oxnard Region Slope Based Objective Scores (Blue:Low, Red:High)

The final corridor solution set, generated from a MOGADOR model run where the input population size was fixed at a value of 100,000, produced an output set of top 100 solutions with an even lower range of composite objective scores: ranging from between 2740 to 2760. The improvement in this overall composite objective scores, as shown by the line plot at the lower center portion of the figure panel, can be observed as being attributable to marginal reductions in the accessibility and disturbance objective scores. This can be interpreted as the algorithm locating corridors which route around areas with high intensity land uses and routing along the more highly accessible transportation network.

Figure 4.27 provides a broad plan overview illustration of the final output corridors produced by the MOGADOR solution run where the population size was set to a value of 100,000.

The corresponding elevation profile for the solution illustrated in Figure 4.27 can be observed plotted in Figure 4.28. This elevation profile reals that there is only a modest amount of elevation change along the length of the proposed corridor solution (roughly 130 me-

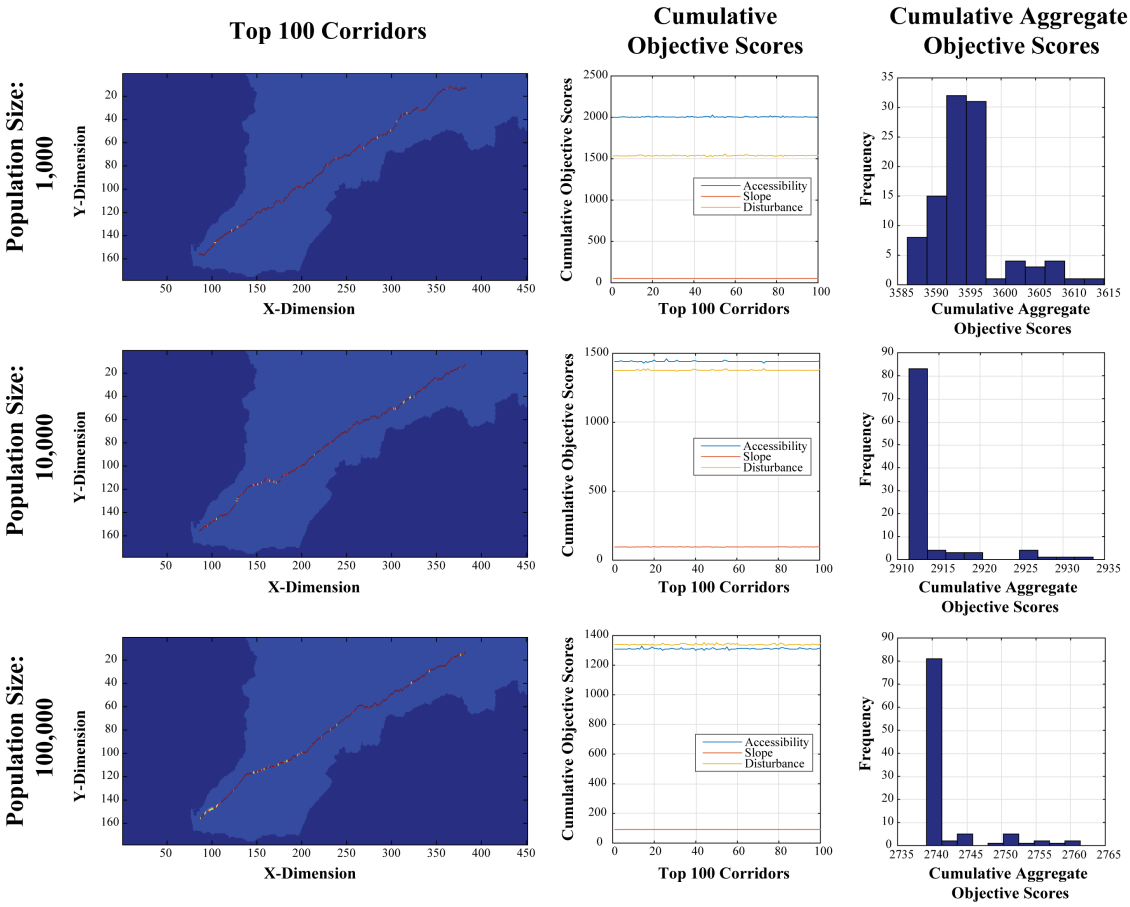


Figure 4.26: Oxnard Region Corridor Analysis Results

ters). It also shows that this elevation gain is discontinuous along the length of the proposed corridor solution with there being to two modest hills – each roughly 40 meters in height – that must be ascended and descended before the destination is reached.

Figure 4.27: Oxnard Region Top 100 Corridors (Pop Size: 100,000) Basin Wide Overview

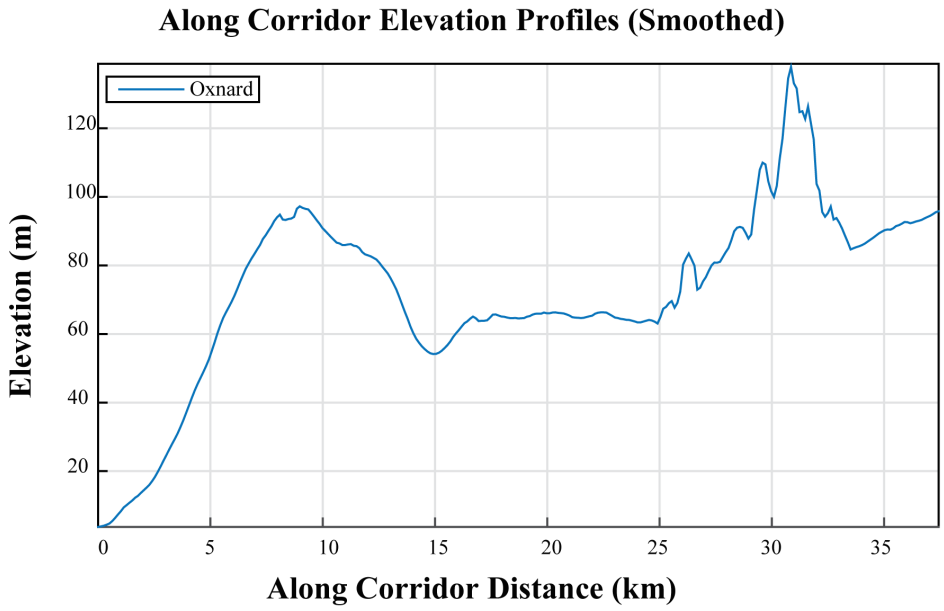


Figure 4.28: Oxnard Region Proposed Corridor Elevation Profile

4.3 San Diego Region

The third case study region investigated as part of this dissertation consists of the HUC-8 basin comprising the city of San Diego and its adjacent metropolitan districts. This basin is situated at the southwestern most tip of the State of California and adjoins the international border between the United States and Mexico as shown by the black filled area plotted in Figure 4.29.

In terms of its water resource management history, San Diego has become world renowned as a leader in freshwater management for its innovative approaches to demand management policy and advanced technological solutions for providing alternatives supply. As is often the case, necessity has been the mother of this innovation, with the San Diego region experiencing explosive growth in population growth and associated freshwater demand over the past 50 years while at the same time being cutoff from major statewide inter-basin water transfer projects.

For example, from 2009 to 2013 the San Diego Municipal Water District embarked upon a large scale demonstration project to determine whether the advanced tertiary water treatment systems that would be necessary to facilitate large scale indirect, or possibly even direct, potable reuse could be implemented effectively and reliably at scale. In this project, purified water was blended with imported water supplies in the San Vicente Reservoir before going to the standard drinking water treatment plant. Due in large part to this success of the pilot program, the San Diego city council recently unanimously approved a three and a half billion dollar direct potable reuse project and plant that is to be constructed over the next decade. This facility is being planned in conjunction with another large scale desali-

nation plant in a bid to build a portfolio of alternative freshwater supply and groundwater recharge capacity just as the state enters the fourth year of a crippling drought condition.

The San Diego case study region is unique in the context of the other case study regions evaluated as part of this dissertation in that the destination location to be used for the corridor location problem specification has been designated as the location of the municipal drinking water treatment plant. As such, no destination search process was undertaken for this case study region.

4.3.1 Regional Context

- HUC-8 Code: 18070304
- Total Area: $4,338.1\ km^2$
- Maximum Elevation: $1,977\ m$
- Minimum Elevation: $-0.7\ m$
- Mean Slope: $9.38\ \%$
- Standard Deviation of Slope: $8.77\ \%$
- Dominant Soil Composition: Hydrologic Soil Group - B: $10 - 20\%$ clay, $50 - 90\%$ sand, 35% rock fragments

4.3.2 Search Domain

The search domain comprising the San Diego study region is described in the statistics below and depicted graphically in the map panel contained within 4.30.

- Grid Dimensions: 798 *cells* x 898 *cells*

Figure 4.29: San Diego Region Overview (Filled in Black)

- Grid Cell Resolution: 100 m x 100 m (1 ha)
- Feasible Grid Cells: 433, 808 *cells*

4.3.3 Proposed Corridor Endpoints

The proposed endpoints to be used in the MOGADOR algorithm specification are shown in Figure 4.31. The source location was determined by the location of the largest NPDES permitted WWTP in the basin while the destination location, in this case, was pre-determined as the location at which an artificial groundwater recharge basin has already been implemented.

- Start Location: (635, 42)
- End Destination: (453, 363)

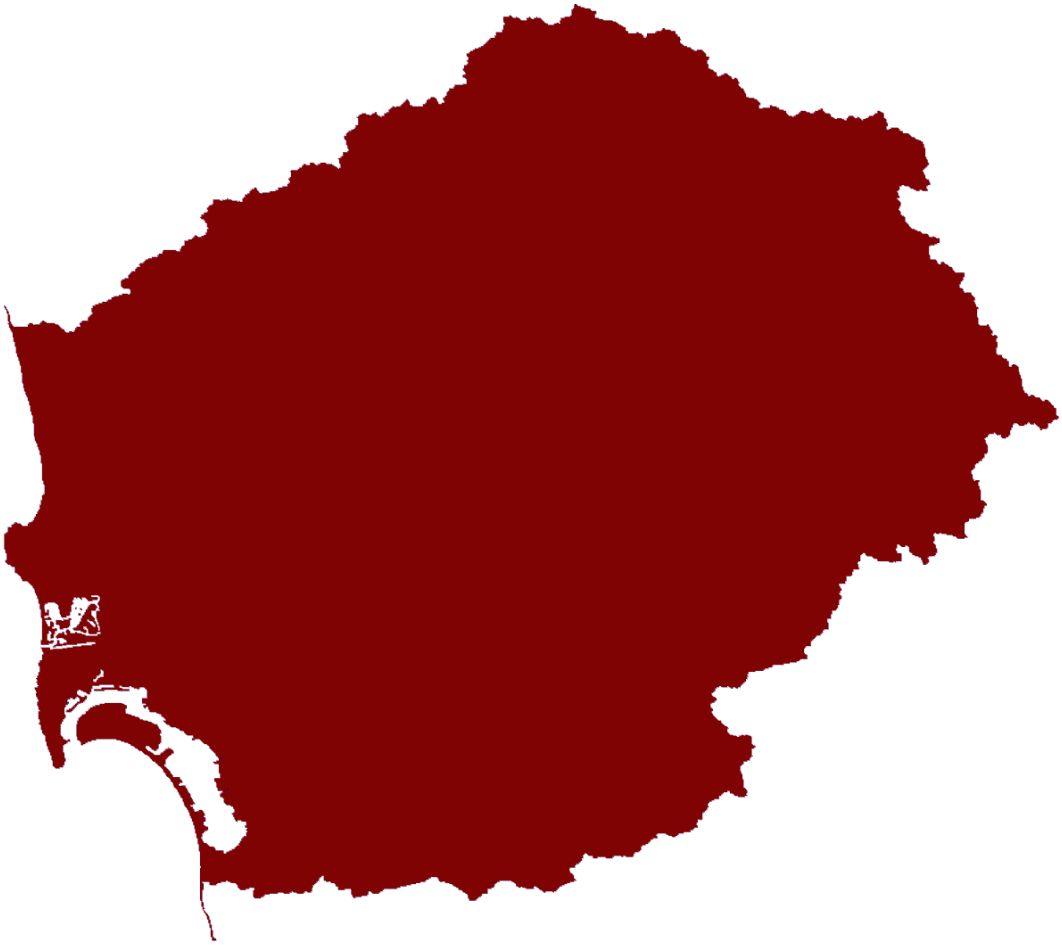


Figure 4.30: San Diego Region Search Domain (Filled in Red)

- Shortest Euclidean Path Distance: $36,901\ m\ (36\ km)$

4.3.4 Proposed Objective Layers

The three proposed objective layers which round out the MOGADOR algorithm problem specification and depicted in Figures 4.33 through 4.34, consist of the same accessibility, landuse disturbance, and slope based data layers as those described previously for Santa

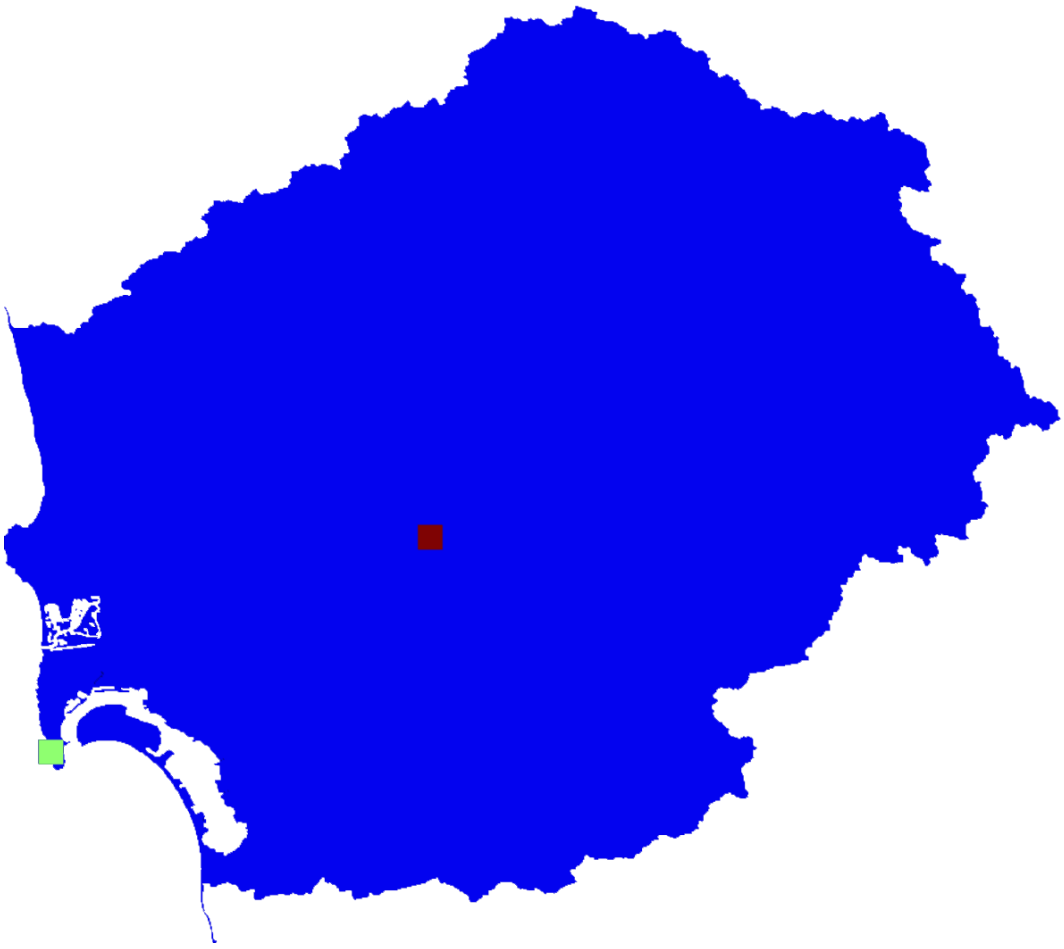


Figure 4.31: San Diego Region Proposed Corridor Endpoints

Barbara & Oxnard, and used for all of the case studies included in this analysis. In terms of the structure of these objectives in the San Diego region, the coastal areas tend to be very highly developed with large land use disturbance scores as well as a dense road network providing favorable accessibility values. The basin does not contain an extreme amount of topographic relief as evidenced by the fairly homogeneous distribution of slopes shown in Figure 4.34.

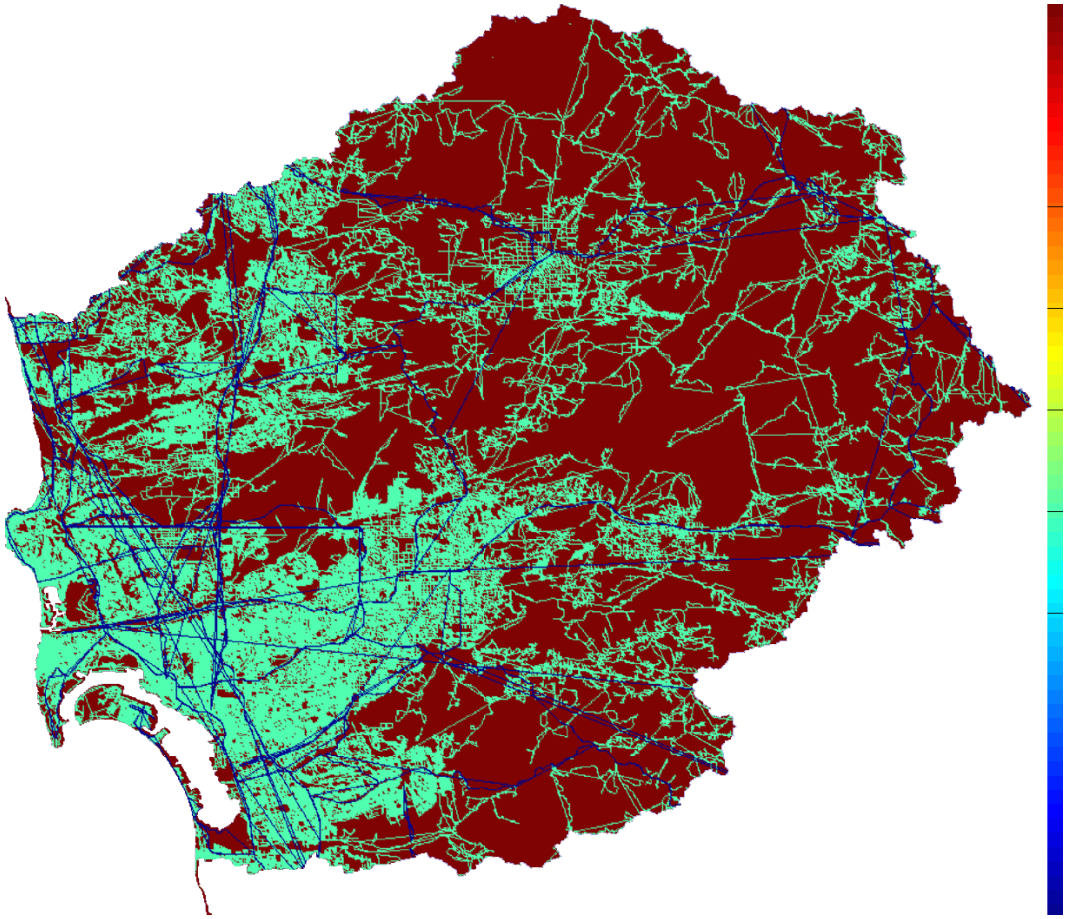


Figure 4.32: San Diego Region Accessibility Based Objective Scores (Blue:Low, Red:High)

4.3.5 Proposed Corridor Solutions

The results of the three runs of the MOGADOR algorithm for the San Diego region case study corridor location analysis are presented in Figure ?? and reflect the same three variations on the seed population size described for the previous case studies. Here again, the highest quality solution set was produced by the MOGADOR run using the largest population size with the minimum cumulative objective scores for the top 100 output solutions

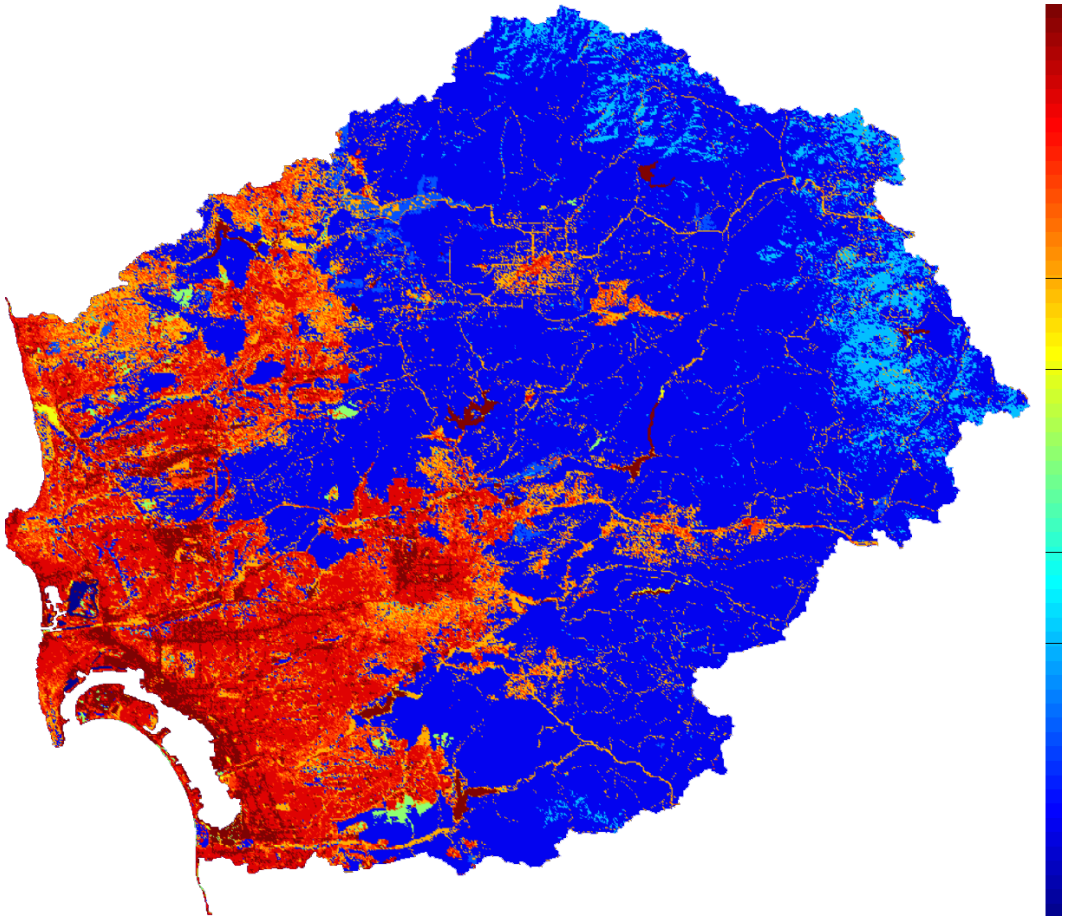


Figure 4.33: San Diego Region Land Use Disturbance Based Objective Scores (Blue:Low, Red:High)

ranging from 3840 to 3865. Between the three algorithm runs, the majority of the aggregate objective score improvement came from reductions in the accessibility and disturbance scores; this, again, reflecting the algorithm's iterative discovery of those corridor sections running in and along road network sections while avoiding areas with higher intensity landuse.

The top output solution for the three runs is depicted in the context of the entire search domain in Figure 4.36. As this Figure shows the final corridor routes from the location of

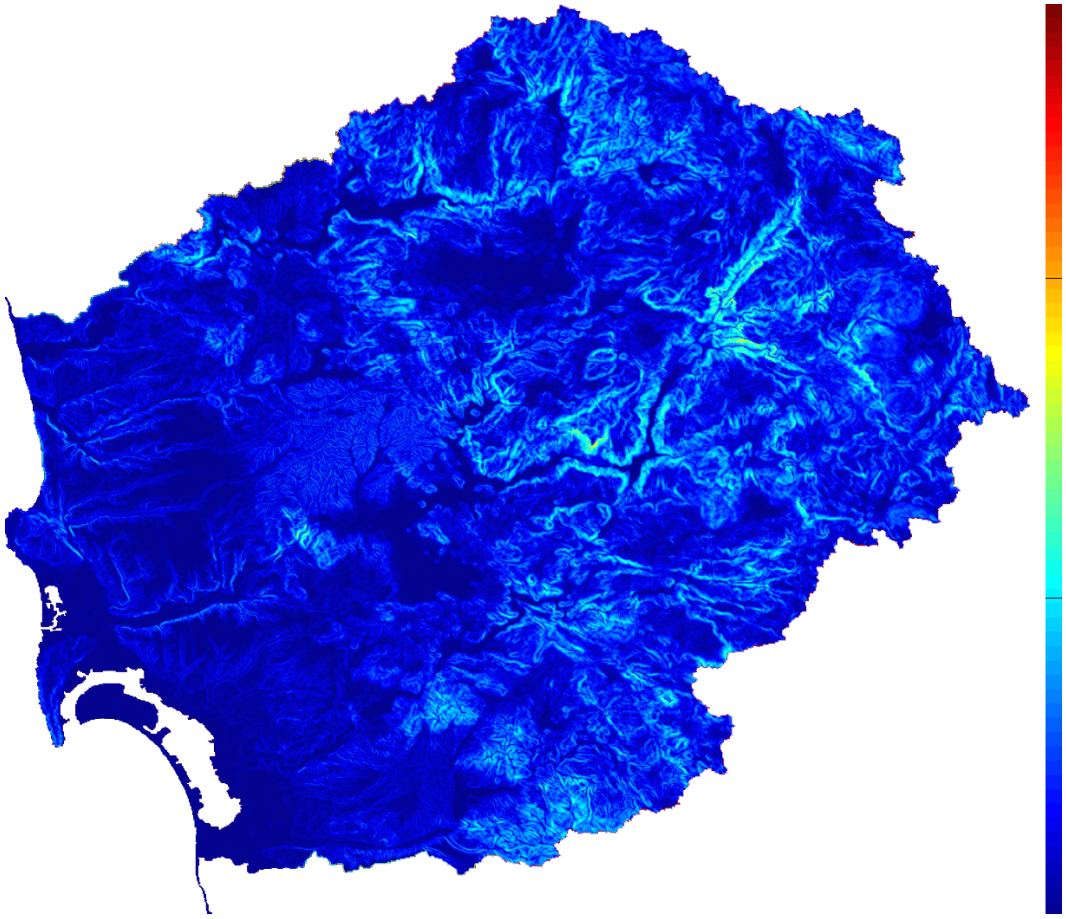


Figure 4.34: San Diego Region Slope Based Objective Scores (Blue:Low, Red:High)

the coastal WWTP treatment plant, around the large harbor area at the Southwest portion of the search domain, and up towards the existing groundwater recharge facility located at the heart of the basin.

Figure 4.37 plots the elevation profile of the land surface along the length of the proposed corridor solution. The maximum elevation gain between the endpoints of the corridor is roughly 230 meters, however there is a considerable amount of ups and downs along the corridor's length. The jaggedness of the elevation profile reflects the extremely high den-

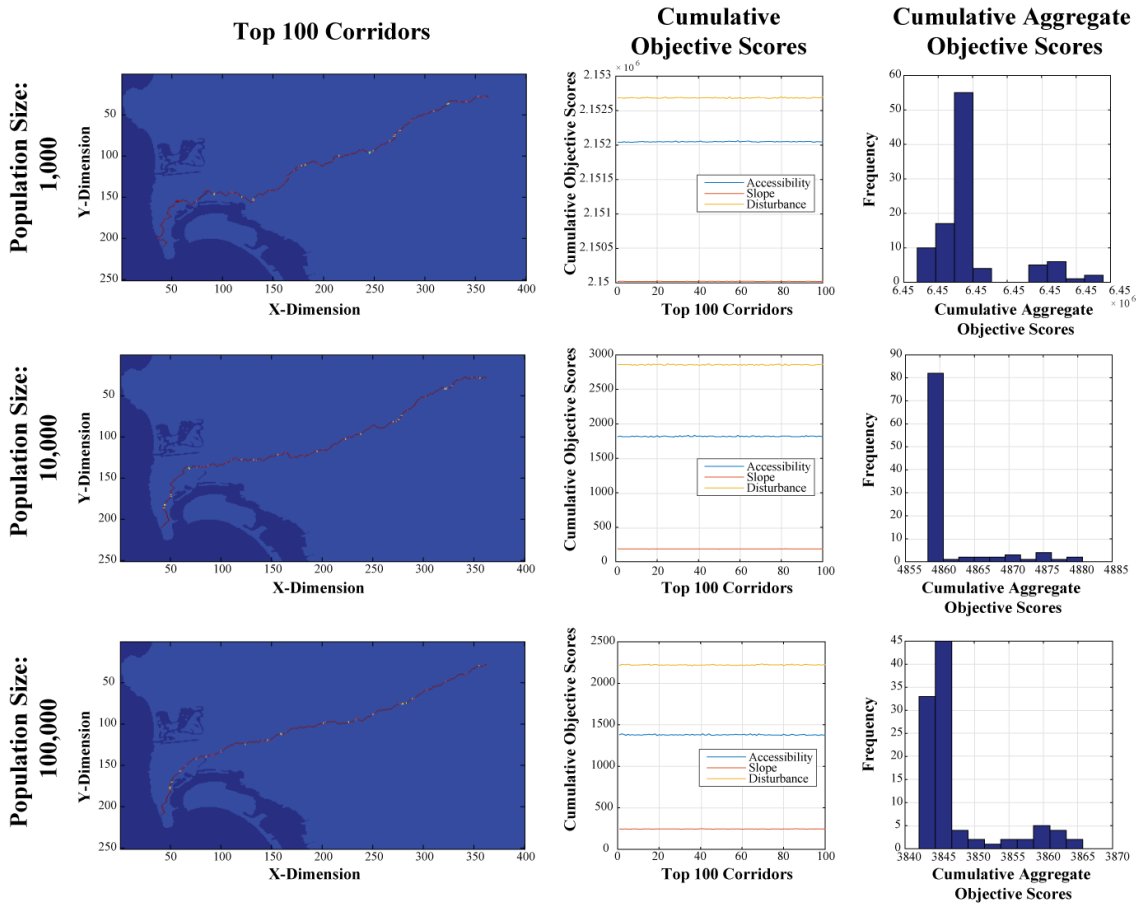


Figure 4.35: San Diego Region Corridor Analysis Results

sity of the urban environment in the area immediately inland from the coastal WWTP. It reflects a necessary tradeoff between the accumulation of slope and the need to route around areas with high intensity existing landuse.

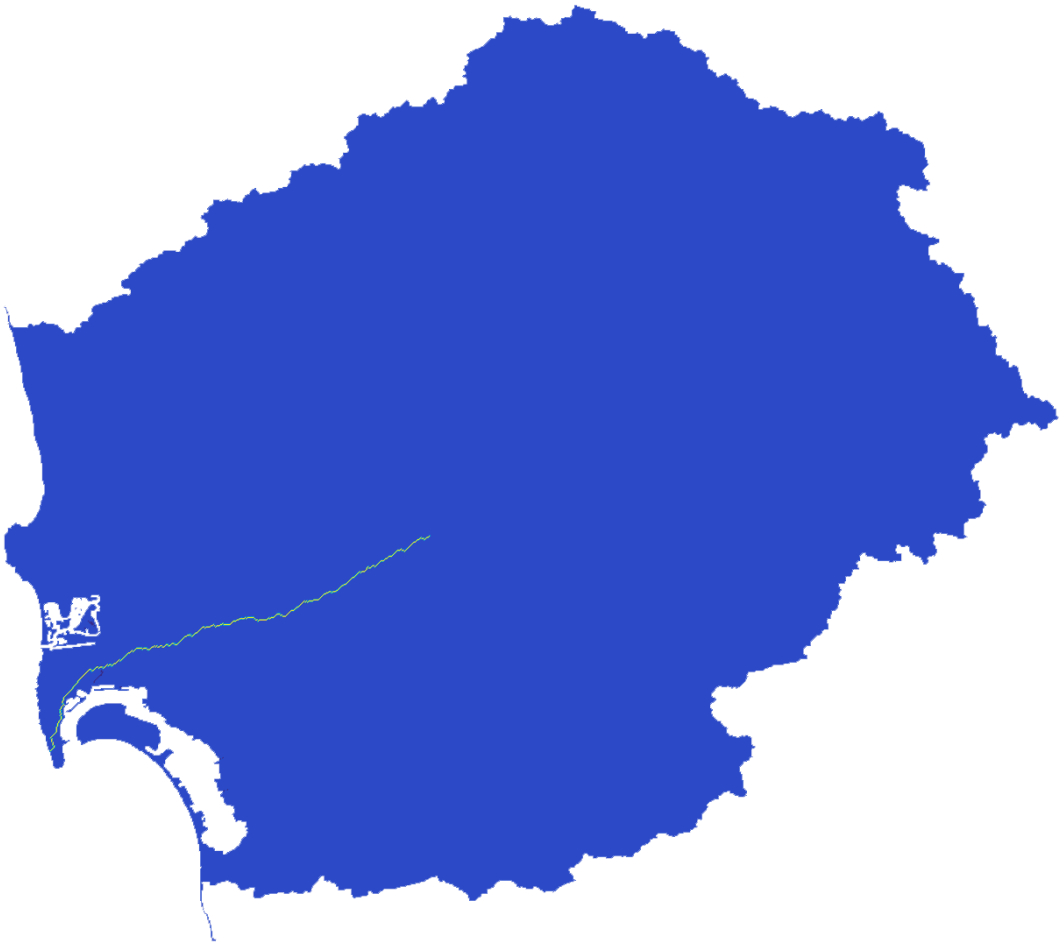


Figure 4.36: San Diego Region Top 100 Corridors (Pop Size: 100,000) Basin Wide Overview

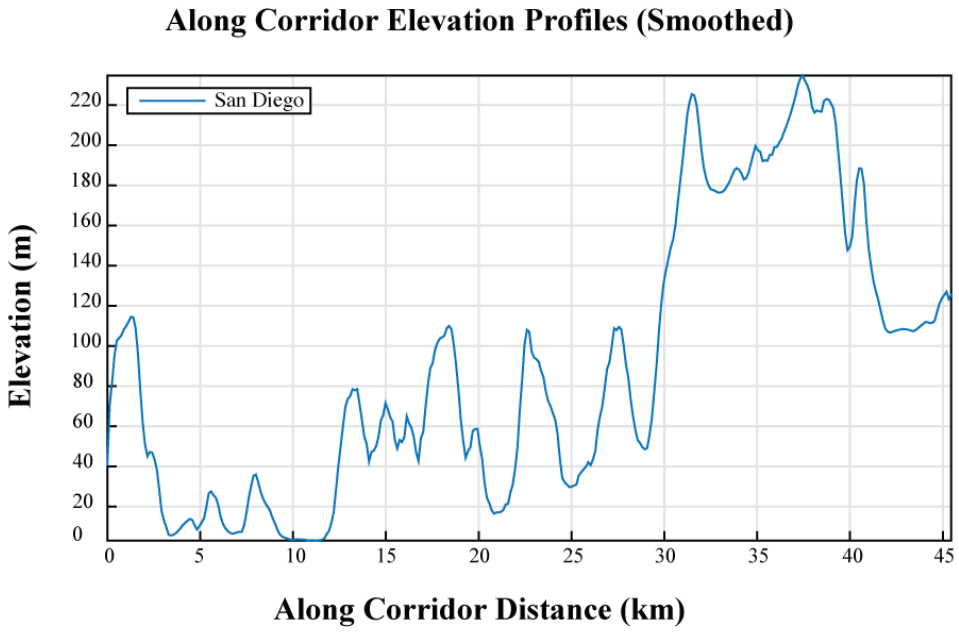


Figure 4.37: Santa Diego Region Proposed Corridor Elevation Profile

4.4 Santa Ana – San Bernadino Region

The Santa Ana – San Bernadino case study region, filled in black in Figure 4.38, is comprised of another coastal HUC-8 basin that is situated north of San Diego and south of Oxnard. The majority of the basin's area is positioned inland with a small strip of land stretching westward towards the coast around the bed of the Santa Ana river. In terms of total area, the Santa Ana – San Bernadino is the second largest case study region being investigated as part of this dissertation analysis. It is almost four times the size of the Santa Barbara case study basin and is only marginally smaller in size that the largest of the five study sites: Fresno – Tulare.

The Santa Ana – San Bernadino region, which is positioned squarely within Orange County, shares a number of hydrologic similarities to the San Diego region and as a result, it too has been forced to adopt a whole suite of innovative water resource management policies and technological solutions. In fact, this region is home to the first large scale commercial municipal wastewater recycling and reuse installation in the United States; called Water Factory 21. This facility takes raw sewage as influent as used a cutting edge treatment process chain to return that water to levels of purity that are of near potable standard. This reclaimed water is then pumped uphill to a series of interconnected recharge basins positioned along the bed of the Santa Ana river where it is allowed to infiltrate back into the subsurface aquifer, providing a crucial source of artificial recharge.

The Santa Ana – San Bernadino case study site provides a unique opportunity to benchmark the results of this modeling framework against an existing reuse facility that incorporates a significant component of artificial groundwater recharge. In this way, we can do

things like compare the layout of this corridor solution proposed for this region to that implemented in the real world, as well as, hopefully in the future, evaluate the estimates for the water-energy usage efficiency associated with the proposed systems specification to that experienced by the Water Factory 21 facility.

4.4.1 Regional Context

- HUC-8 Code: 18070203
- Total Area: $5,375.9\ km^2$
- Maximum Elevation: $3,461.3\ m$
- Minimum Elevation: $-0.7\ m$
- Mean Slope: 10.56 %
- Standard Deviation of Slope: 12.21 %
- Dominant Soil Composition: Hydrologic Soil Group - B: $10-20\%$ clay, $50-90\%$ sand, 35% rock fragments

4.4.2 Search Domain

The search domain comprising the San Diego study region is described in the statistics below and depicted graphically in the map panel contained within 4.39.

- Grid Dimensions: 854 *cells* x 1463 *cells*
- Grid Cell Resolution: 100 m x 100 m (1 *ha*)
- Feasible Grid Cells: $537,587$ *cells*

Figure 4.38: Santa Ana – San Bernadino Region Overview (Filled in Black)

4.4.3 Destination Search Inputs

In Figures 4.40 through 4.42 the three key inputs to the Santa Ana – San Bernadino region case study reuse destination search process are shown. Here again, a visual inspection of these three layers reveals that there is a large central plain of highly suitable areas in the left central portion of the basin. This area of high suitability is connected to the coast region, where the WWTP is located by a thin strip of land area that is marginally suitable according to the three separated suitability layers which runs along the bed of the Santa Ana river.

4.4.4 Destination Search Outputs

The output of the weighted overlay analysis used to engage in the search for suitable sites for the application of the artificial groundwater recharge surface infiltration basin are shown

Figure 4.39: Santa Ana – San Bernadino Region Search Domain (Filled in Red)

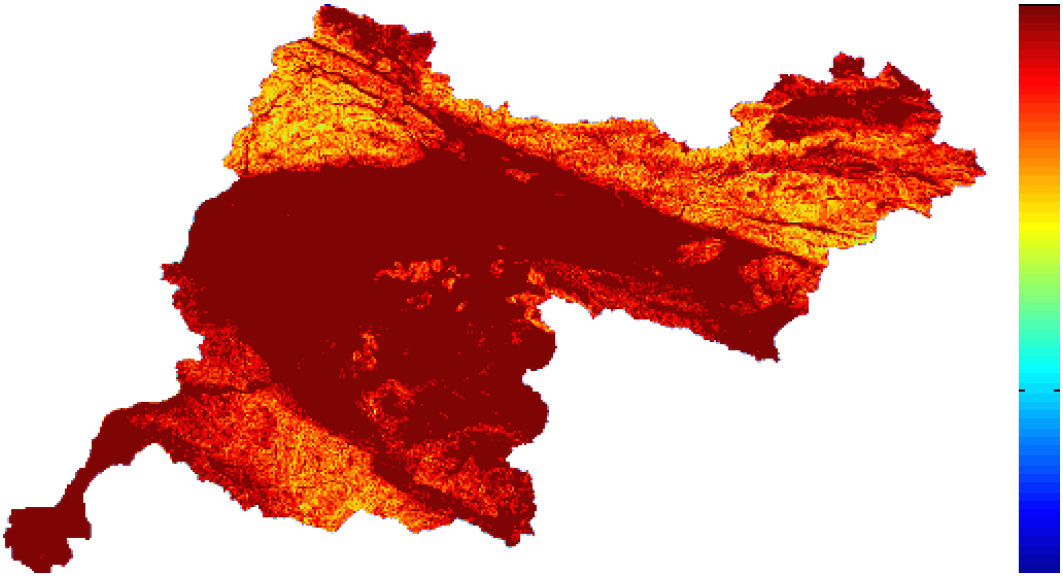


Figure 4.40: Santa Ana – San Bernadino Region Destination Search Inputs: Slope Score (Blue:Low, Red:High)

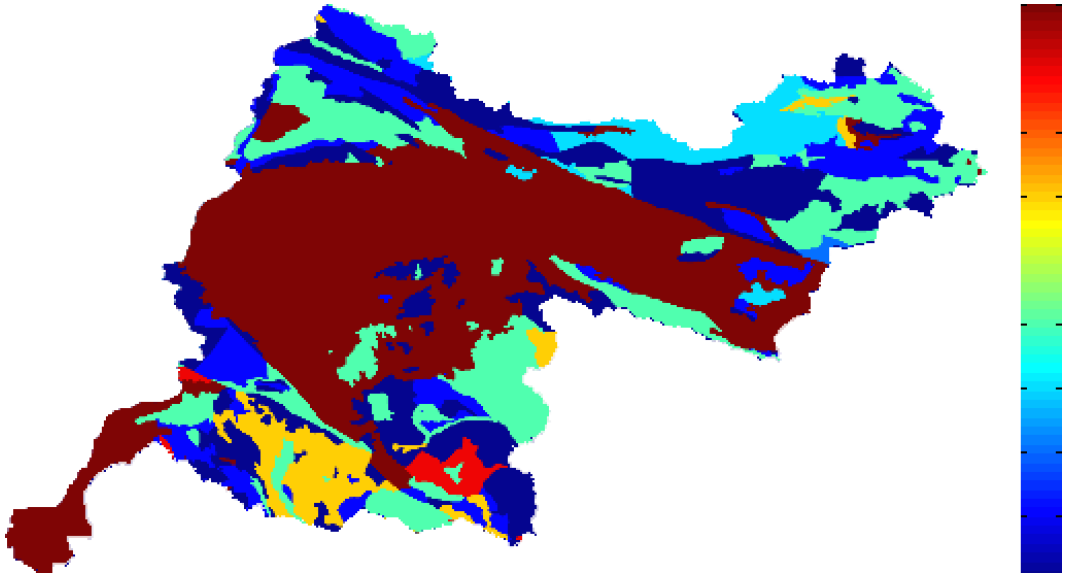


Figure 4.41: Santa Ana – San Bernadino Region Destination Search Inputs: Geology Score (Blue:Low, Red:High)

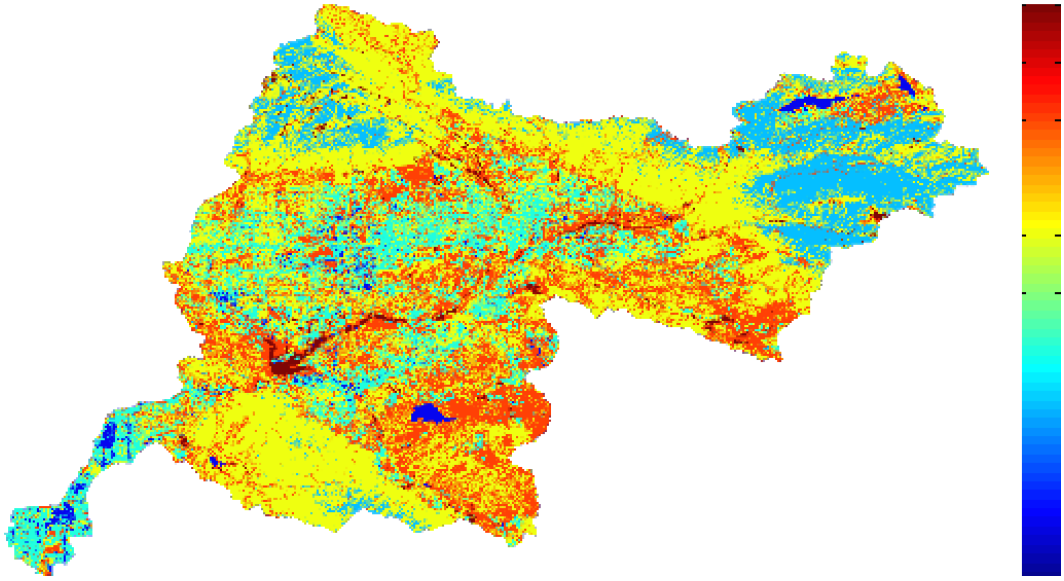


Figure 4.42: Santa Ana – San Bernadino Region Destination Search Inputs: Landuse Score (Blue:Low, Red:High)

in the composite site suitability layer depicted in Figure ??. The largest patches of contiguous high suitability are highlighted in the red portions of Figure ??. The obvious best candidate for a recharge basin site within this search domain can been seen as the large red area positioned along the left center edge of the basin.

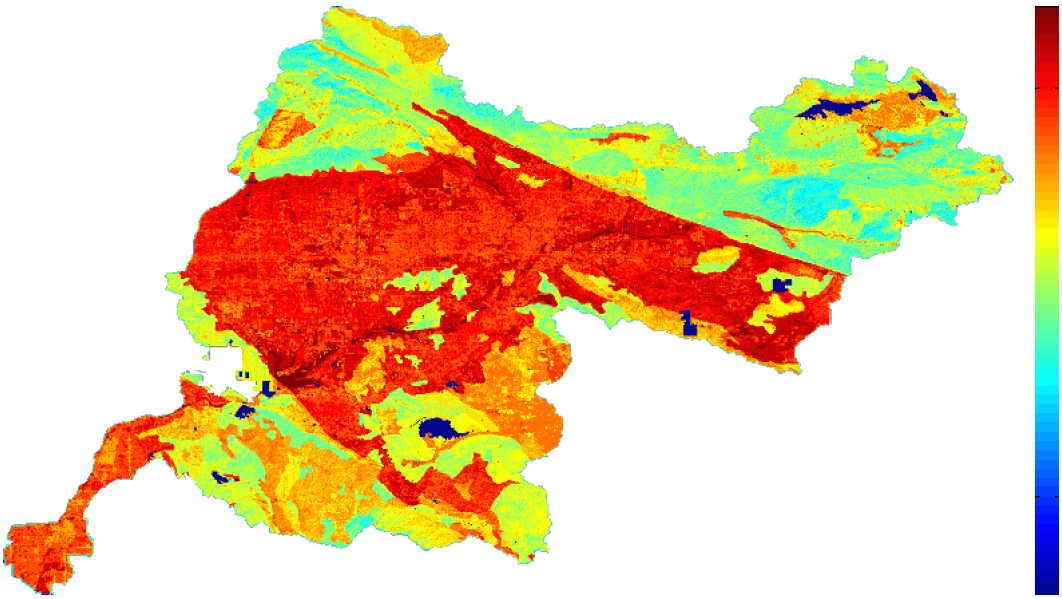


Figure 4.43: Santa Ana – San Bernadino Region Destination Search Outputs: Composite Scores (Blue:Low, Red:High)

4.4.5 Proposed Corridor Endpoints

The proposed endpoints for the Santa Ana – San Bernadino corridor location problem specification to be delivered to the MOGADOR algorithm are plotted in Figure 4.45. The location of the destination site has been selected as the centroid of the large contiguous area of high suitability referenced in the previous section.

- Start Location: (840, 48)

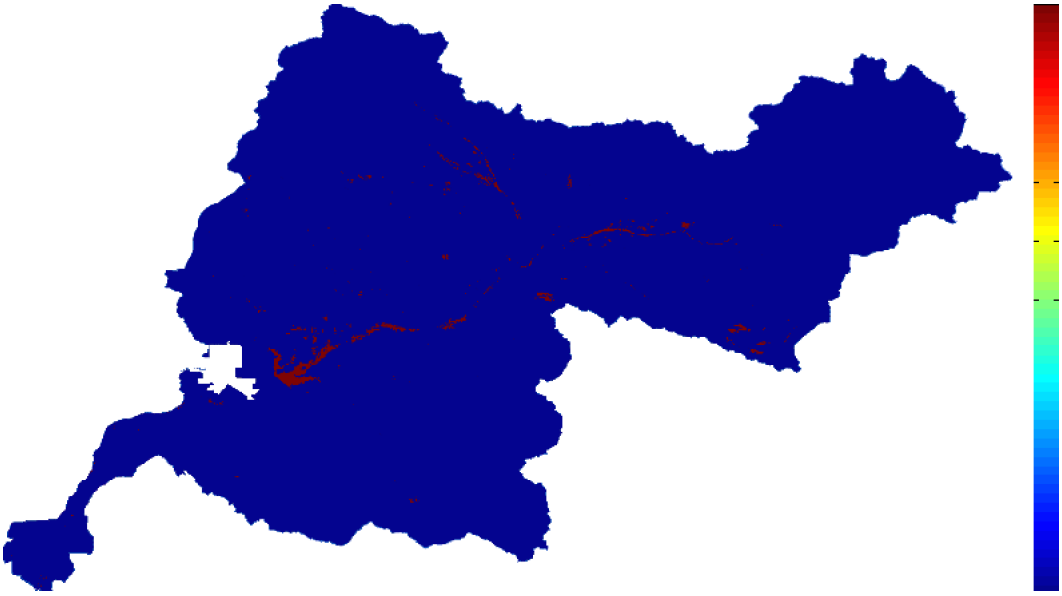


Figure 4.44: Santa Ana – San Bernadino Region Destination Search Outputs: Candidate Regions

- End Destination: (528, 430)
- Shortest Euclidean Path Distance: 49, 322 m (49 km)

4.4.6 Proposed Objective Layers

The three proposed objective layers to be used as part of the MOGADOR algorithm corridor location problem specification are illustrated graphically in Figures ?? through 4.47. Structurally, these objective layers were generated according to the same procedures used to generate the corresponding objective layers for each one of the other five case study sites. As the figures show, the coastal area is a flat, low lying spit with a high average landuse intensity and a very dense road network. This coastal region is largely separated from other populated areas in the basin's interior by coastal mountain range with only a narrow passage having been cut by the Santa Ana River.

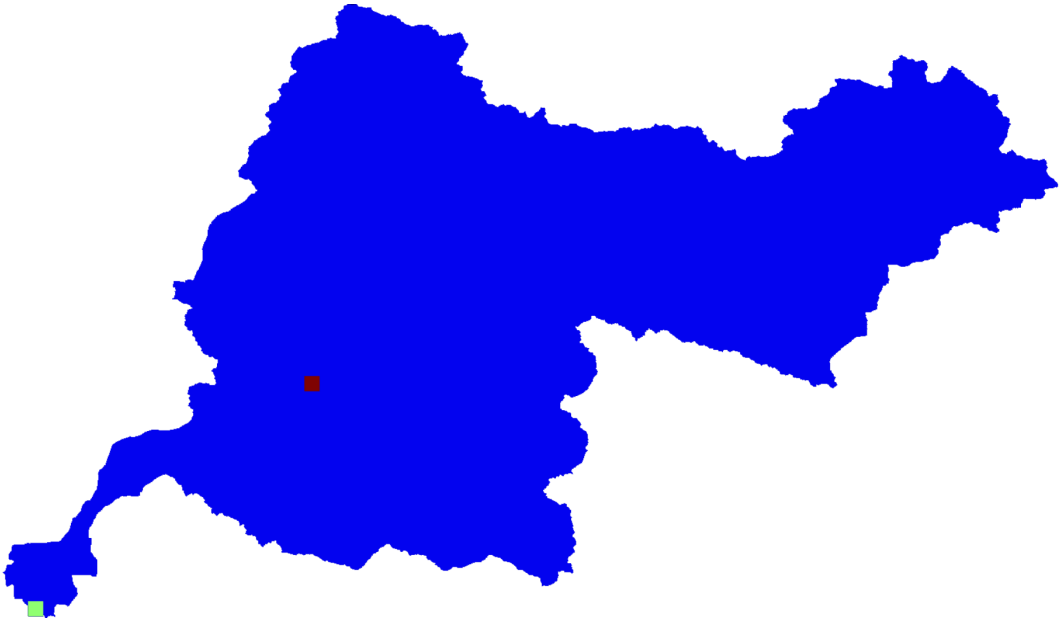


Figure 4.45: Santa Ana – San Bernadino Region Proposed Corridor Endpoints

4.4.7 Proposed Corridor Solutions

As illustrated in Figure 4.49 The proposed corridor solutions for the three MOGADOR model runs with initial population sizes of 1,000, 10,000, and 100,000 show some interesting results. For example, with a population size of 1,000, the algorithm is not able to explore enough of the decision space to produce an output solution set which does not exit the search domain boundary. As a result of this, the range of cumulative aggregate objective function values for the top 100 solutions produced by this run of the algorithm are enormous in size, reflecting the arbitrarily high objective scores assigned to all grid cells outside the feasible search domain for all of the objectives.

As the population size is increased however, we can see that the output solution sets begin to respect the search domain boundary and that the composite aggregate objective

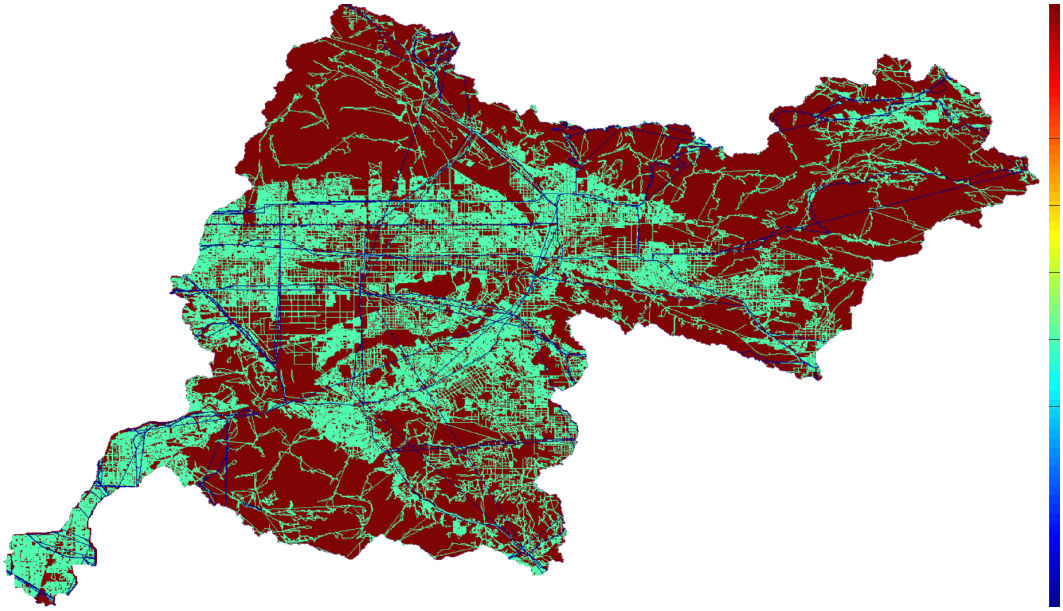


Figure 4.46: Santa Ana – San Bernadino Region Accessibility Based Objective Scores (Blue:Low, Red:High)

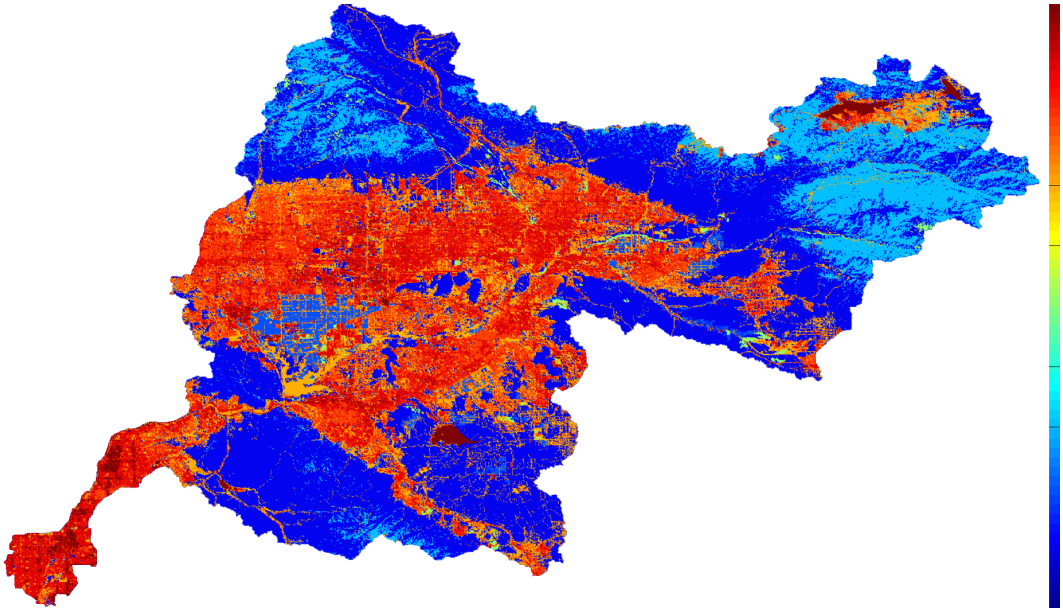


Figure 4.47: Santa Ana – San Bernadino Region Land Use Disturbance Based Objective Scores (Blue:Low, Red:High)

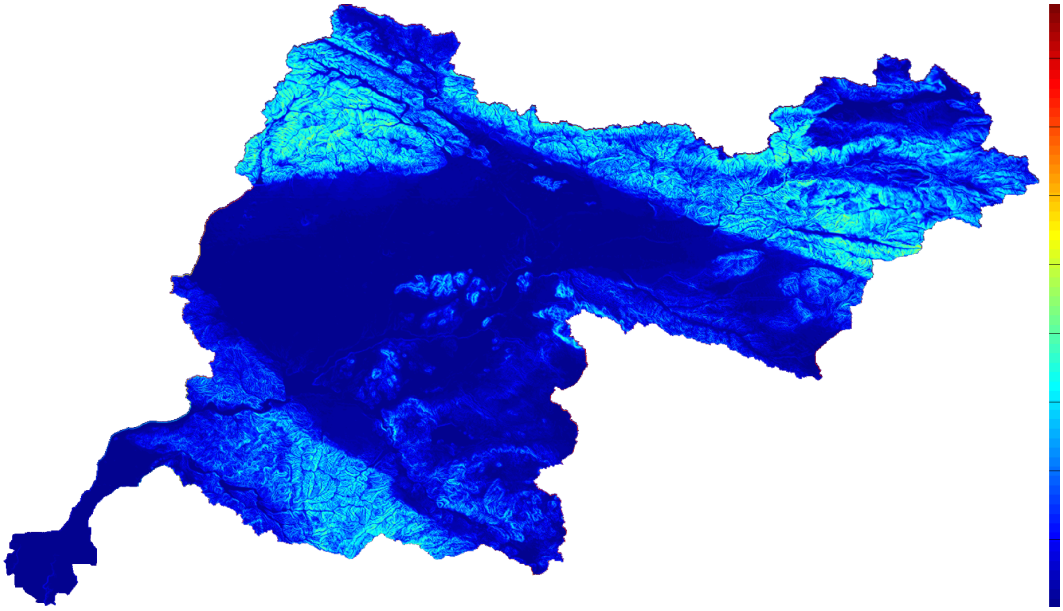


Figure 4.48: Santa Ana – San Bernadino Region Slope Based Objective Scores (Blue:Low, Red:High)

scores for the top 100 solutions in each set decrease dramatically. The top 100 solutions for the MOGADOR run with a population size of 100,000 can be seen to have relatively lower disturbance scores compared to the top 100 solutions generated by the algorithm run with a population size of 10,000. This difference reflects that ability of the run with the larger population size to route corridors that minimally disturb areas within the study site with intensive or otherwise sensitive existing landuse types.

The top 100 solutions generated by the MOGADOR algorithm run with a population of 100,000 are plotted relative to the entire study site's search domain in Figure 4.50.

Figure 4.51 plots the along corridor elevation profile for the best corridor solution produced as an output of the MOGADOR algorithm. This elevation profile reveals that the Santa Ana – San Bernadino study site has the largest net elevation gradient of all of the case

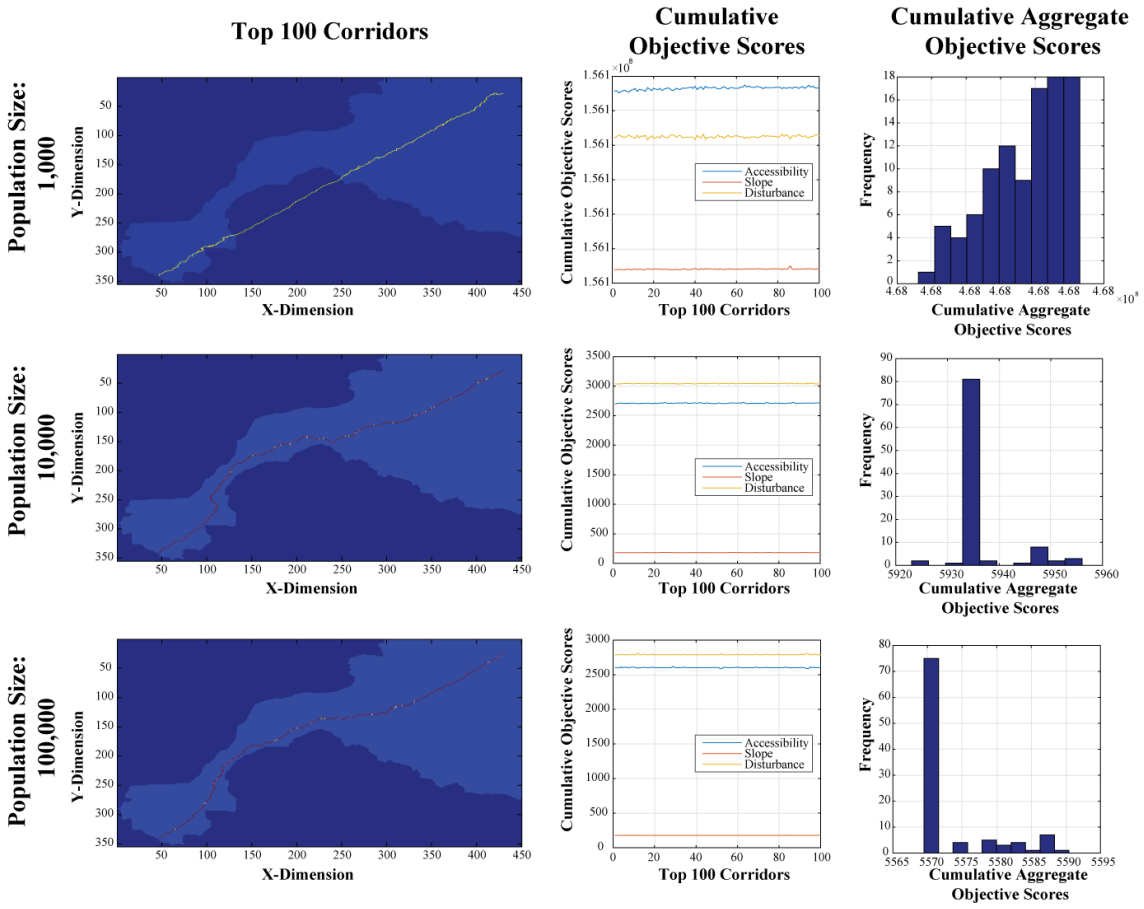


Figure 4.49: Santa Ana – San Bernadino Region Corridor Analysis Results

study regions: a total of 350 meters. It also shows that the accumulation of elevation along the length of the corridor is fairly continuous with two small declines which must be navigated at the very end section of the proposed corridor.

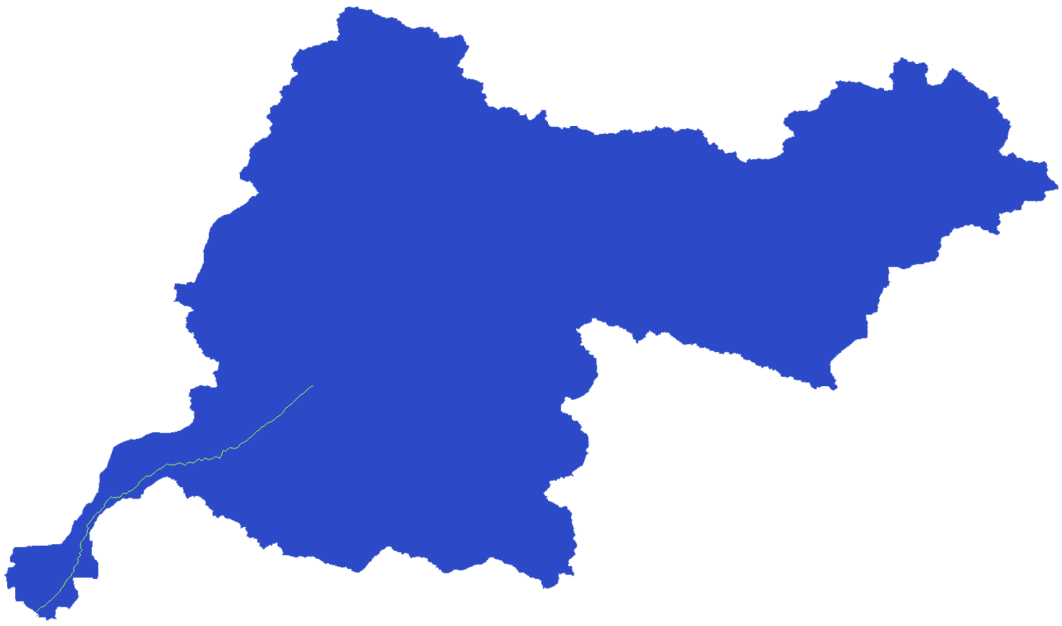


Figure 4.50: Santa Ana – San Bernadino Region Top 100 Corridors (Pop Size: 100,000) Basin Wide Overview

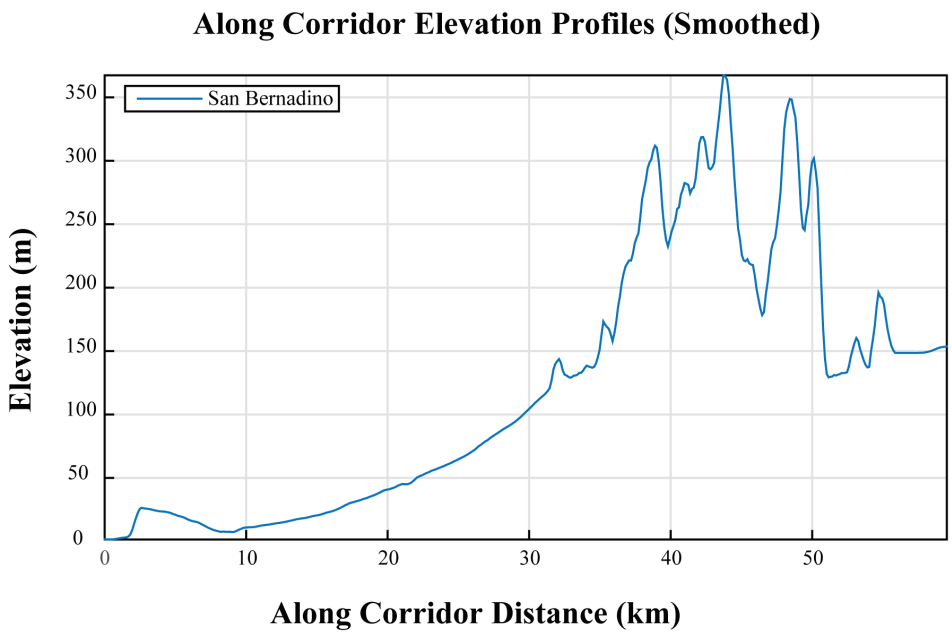


Figure 4.51: San Bernadino Region Proposed Corridor Elevation Profile

4.5 Fresno – Tulare Region

4.5.1 Regional Context

- HUC-8 Code: 18030009
- Total Area: 6, 943.6 km^2
- Maximum Elevation: 1, 536.6 m
- Minimum Elevation: 0 m
- Mean Slope: 2.16 %
- Standard Deviation of Slope: 6.24 %
- Dominant Soil Composition: Hydrologic Soil Group - B: 10 – 20% clay, 50 – 90% sand, 35% rock fragments

Figure 4.52: Fresno – Tulare Region Overview (Filled in Black)

4.5.2 Search Domain

- Grid Dimensions: 1018 *cells* x 1459 *cells*
- Grid Cell Resolution: 100 *m* x 100 *m* (1 *ha*)
- Feasible Grid Cells: 694, 365 *cells*

Figure 4.53: Fresno – Tulare Region Search Domain (Filled in Red)

Figure 4.54: Fresno – Tulare Region Destination Search Inputs: Slope Score (Blue:Low, Red:High)

4.5.3 Destination Search Inputs

4.5.4 Destination Search Outputs

4.5.5 Proposed Corridor Endpoints

- Start Location: $(435, 1037)$
- End Destination: $(421, 387)$
- Shortest Euclidean Path Distance: $65,015\ m$ $(65\ km)$

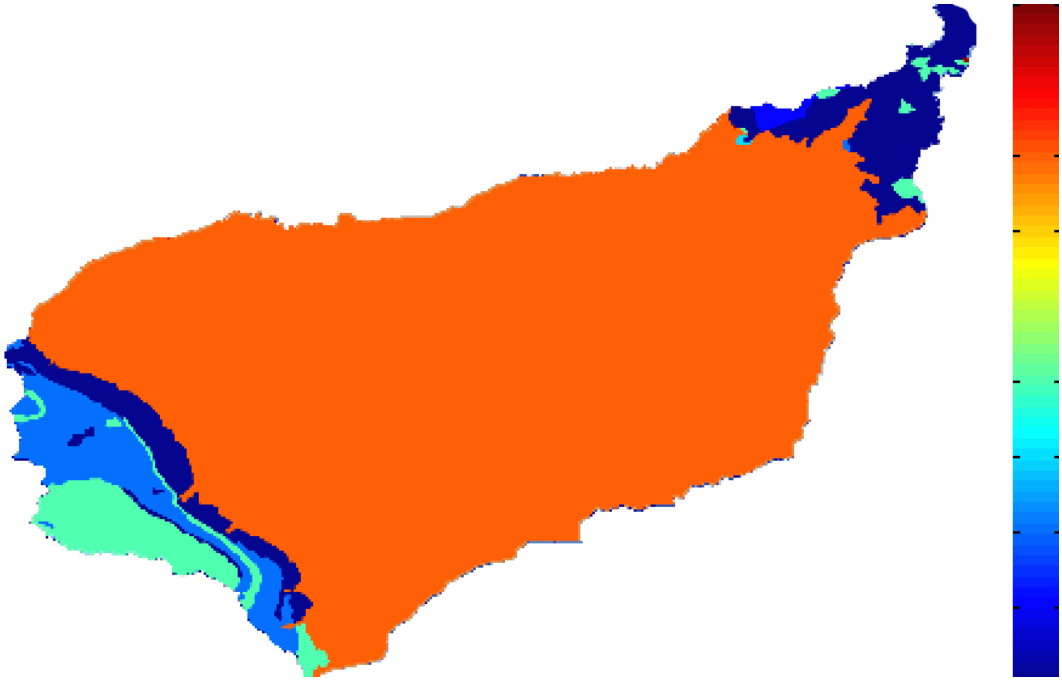


Figure 4.55: Fresno – Tulare Region Destination Search Inputs: Geology Score (Blue:Low, Red:High)

4.5.6 Proposed Objective Layers

4.5.7 Proposed Corridor Solutions

4.5.8 Anticipated Distribution of Life Cycle Energy Usages and Net Water Savings

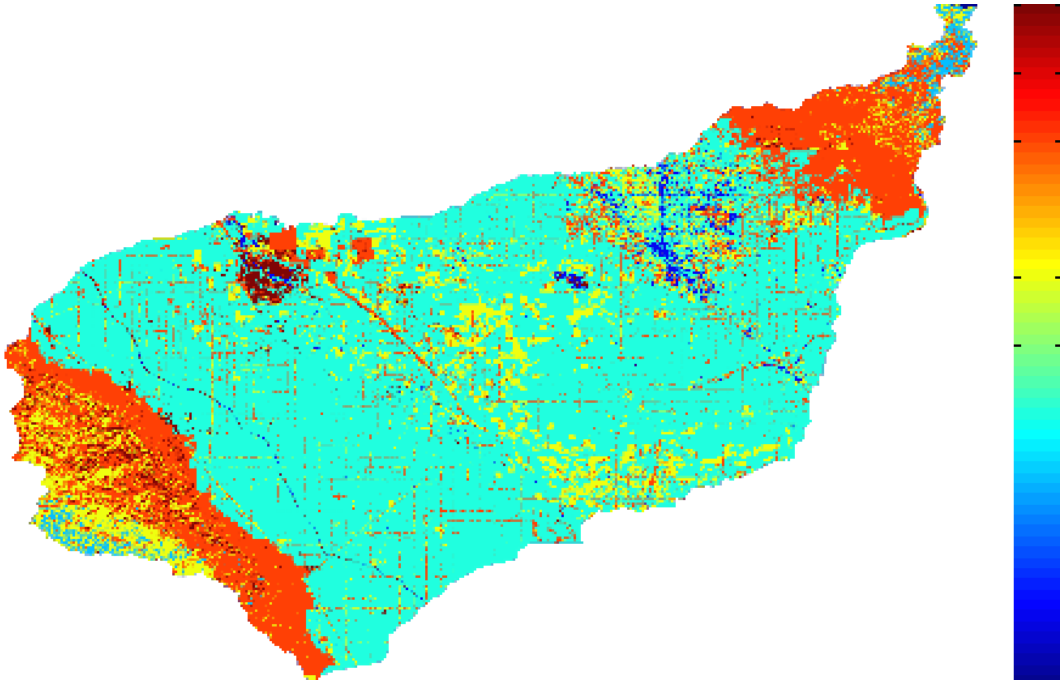


Figure 4.56: Fresno – Tulare Region Destination Search Inputs: Landuse Score (Blue:Low, Red:High)

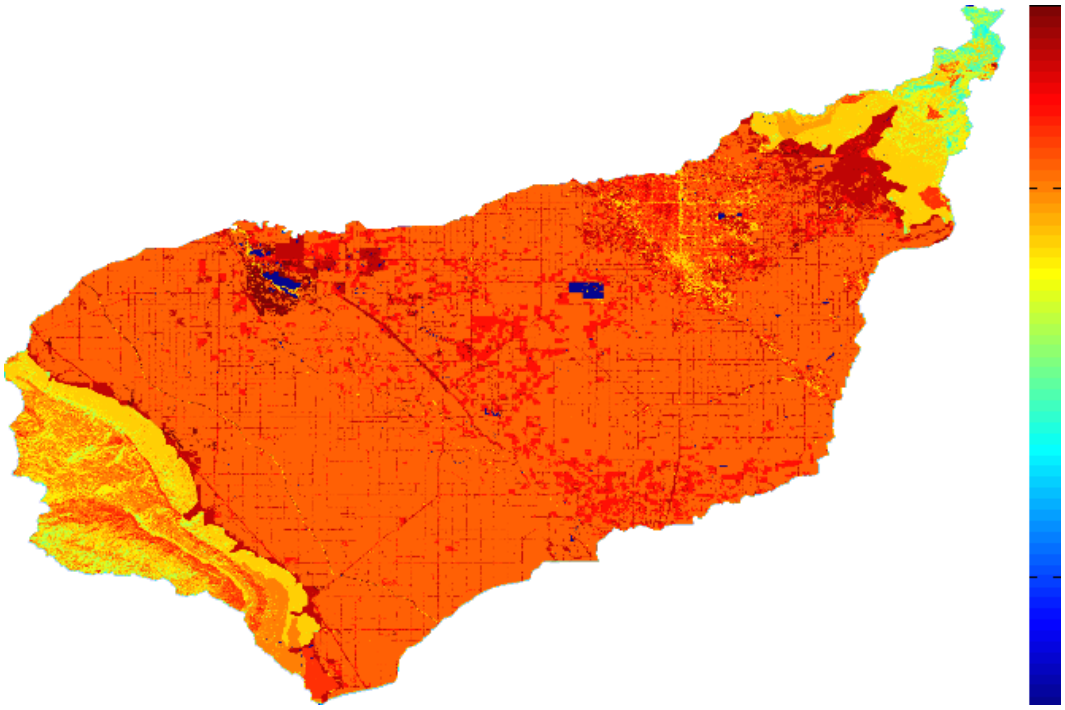


Figure 4.57: Fresno – Tulare Region Destination Search Outputs: Composite Scores (Blue:Low, Red:High)

Figure 4.58: Fresno – Tulare Region Destination Search Outputs: Candidate Regions

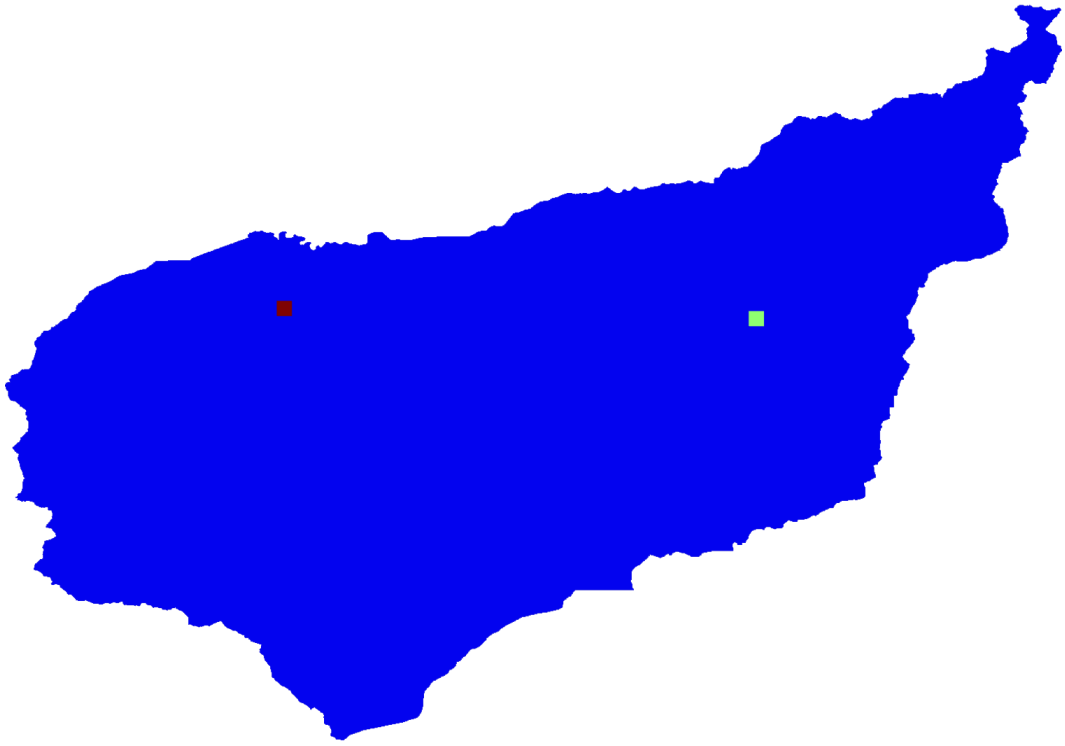


Figure 4.59: Fresno – Tulare Region Proposed Corridor Endpoints

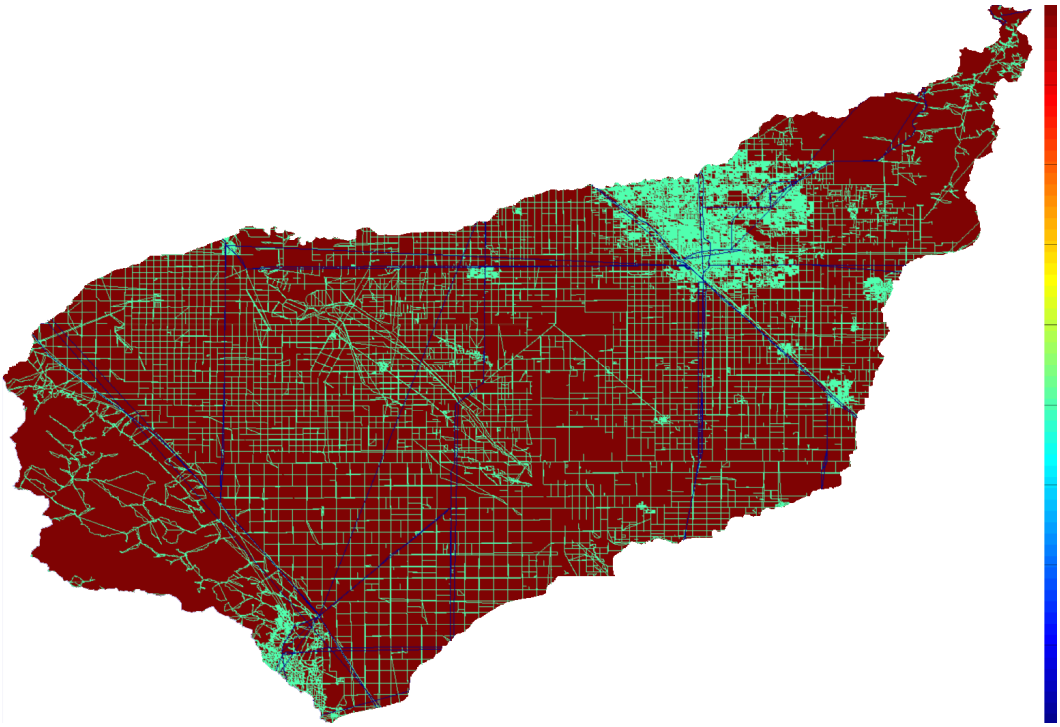


Figure 4.60: Fresno – Tulare Region Accessibility Based Objective Scores (Blue:Low, Red:High)

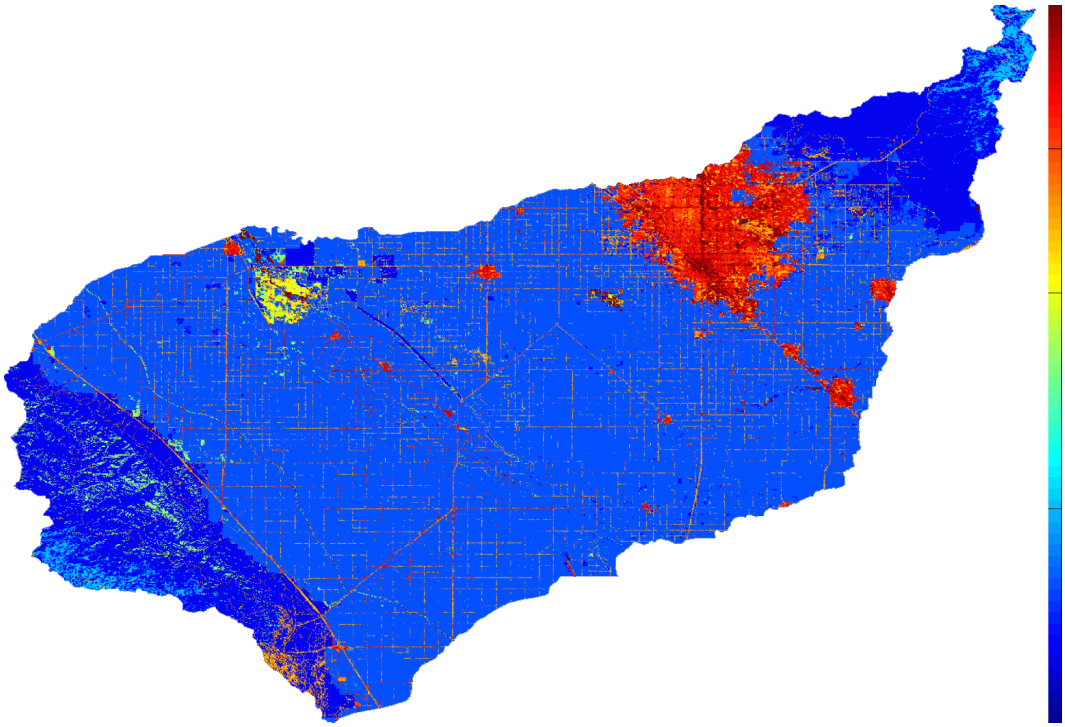


Figure 4.61: Fresno – Tulare Region Land Use Disturbance Based Objective Scores (Blue:Low, Red:High)

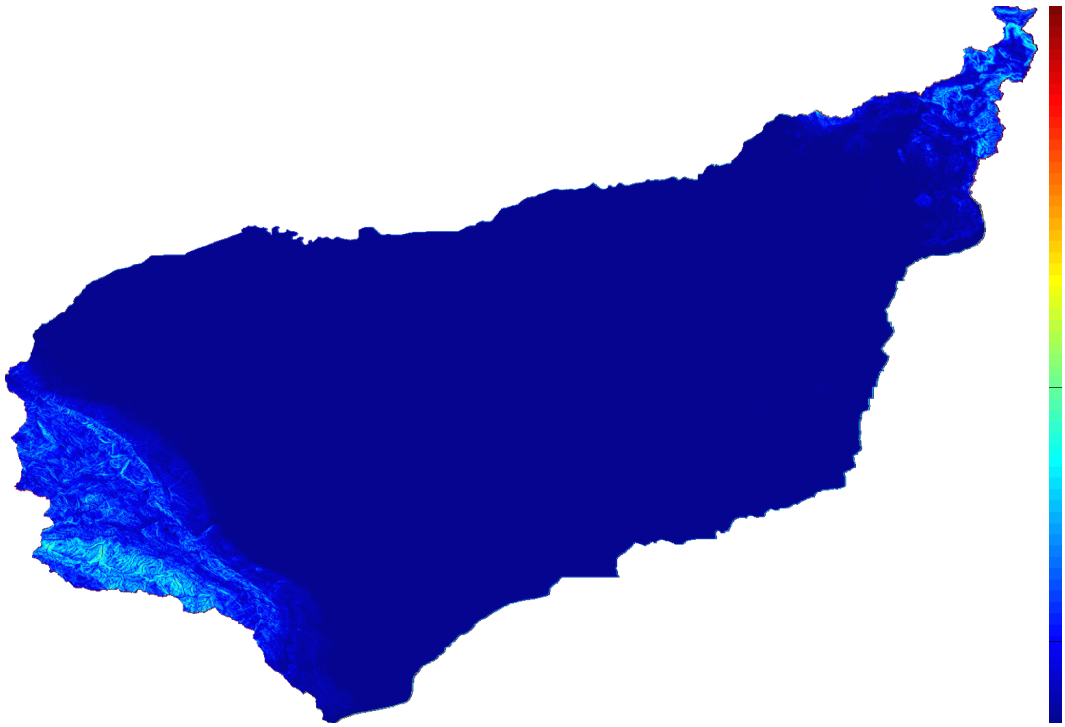


Figure 4.62: Fresno – Tulare Region Slope Based Objective Scores (Blue:Low, Red:High)

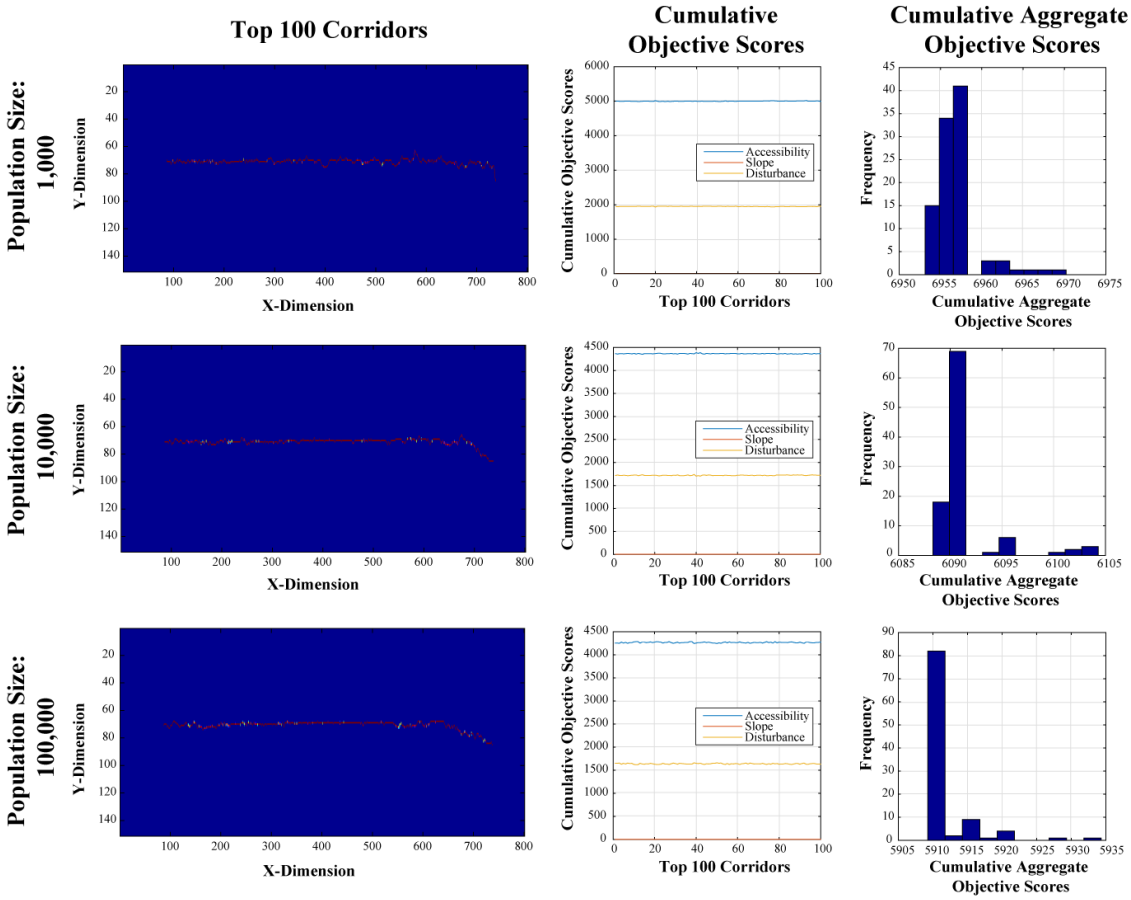


Figure 4.63: Fresno Region Corridor Analysis Results

Figure 4.64: Fresno Region Top 100 Corridors (Pop Size: 100,000) Basin Wide Overview

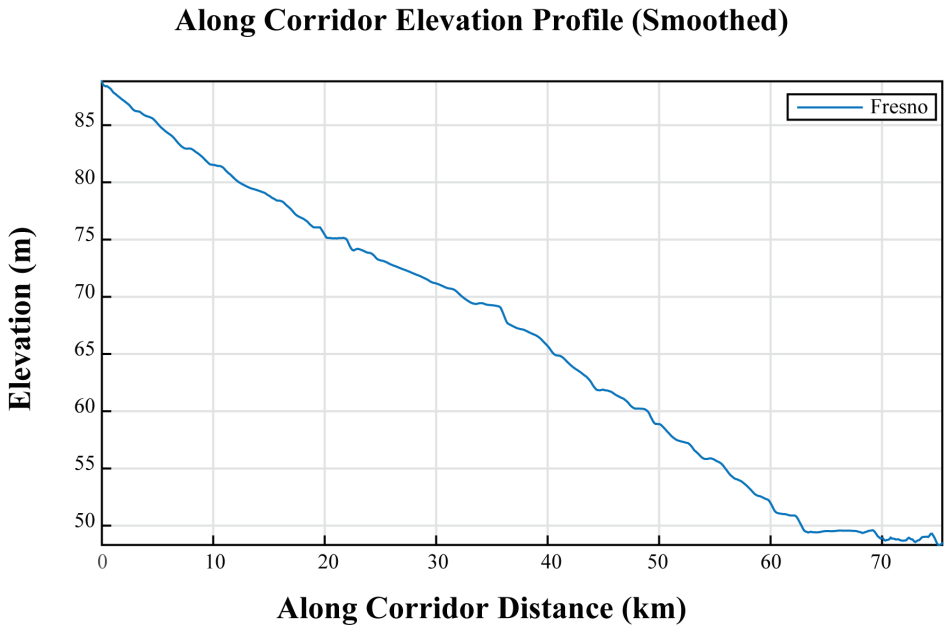


Figure 4.65: Fresno Region Proposed Corridor Elevation Profile

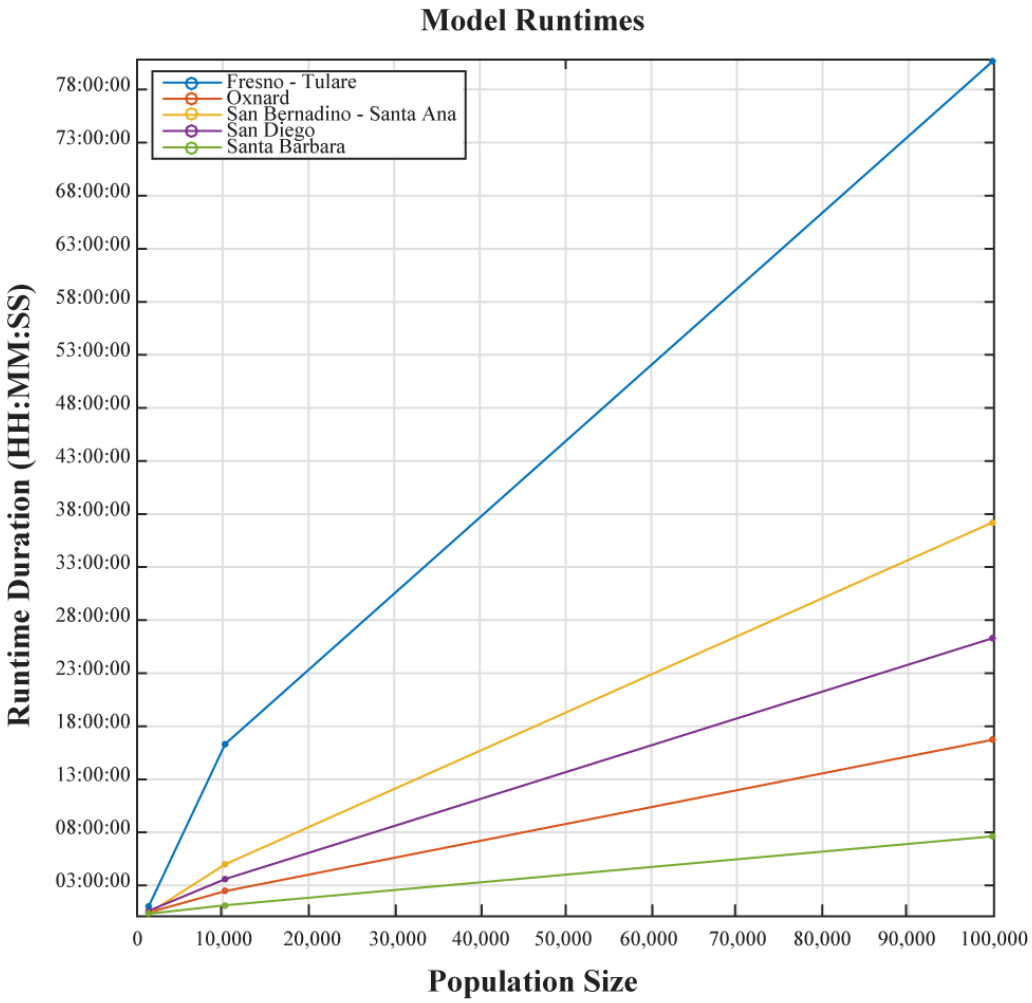


Figure 4.66: Algorithm Runtime Performance for Each of the Five Case Study Regions for Three Population Sizes

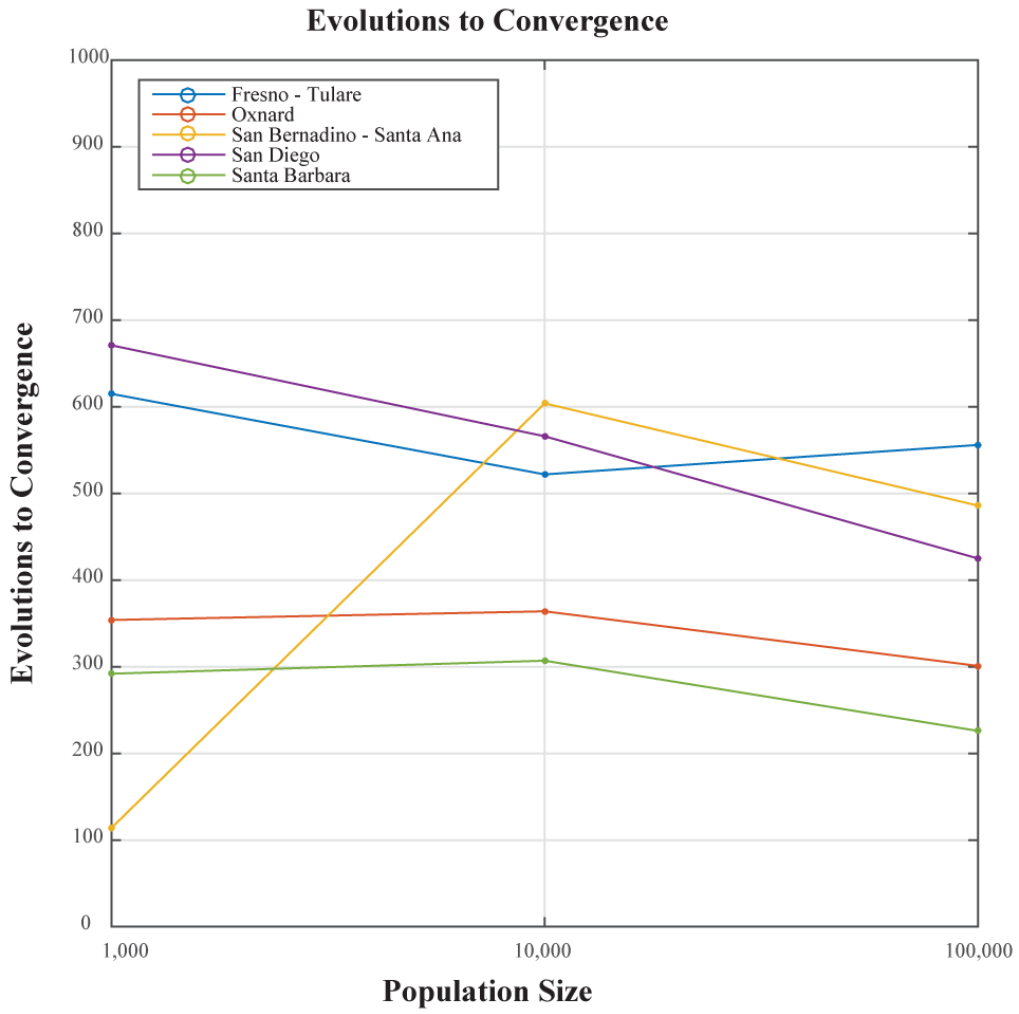


Figure 4.67: Algorithm Convergence Rates for Each of the Five Case Study Regions for Three Population Sizes

4.7 Evaluating Algorithm Solution Quality

4.8 Evaluating Corridor Elevation Profiles

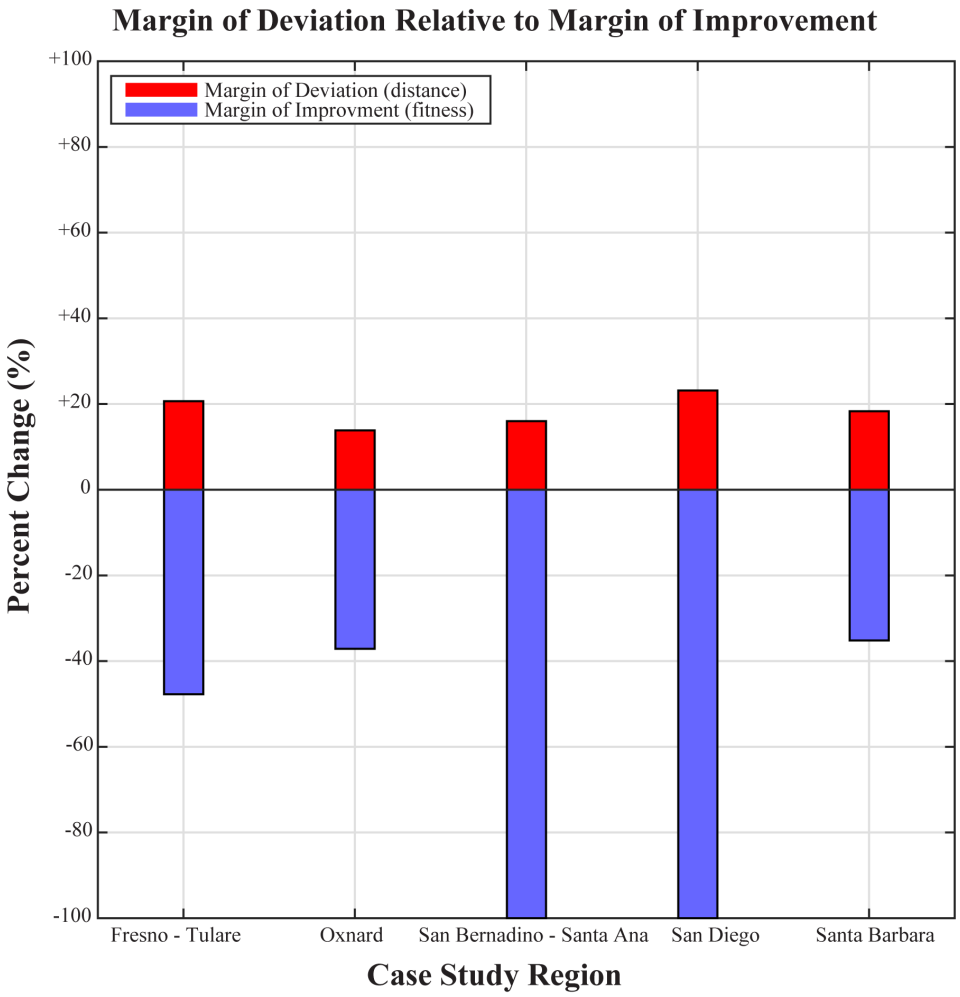


Figure 4.68: Comparison of the Along Path Distance and Cumulative Objective Scores between the Solution Corridors and the Euclidean Shortest Corridors

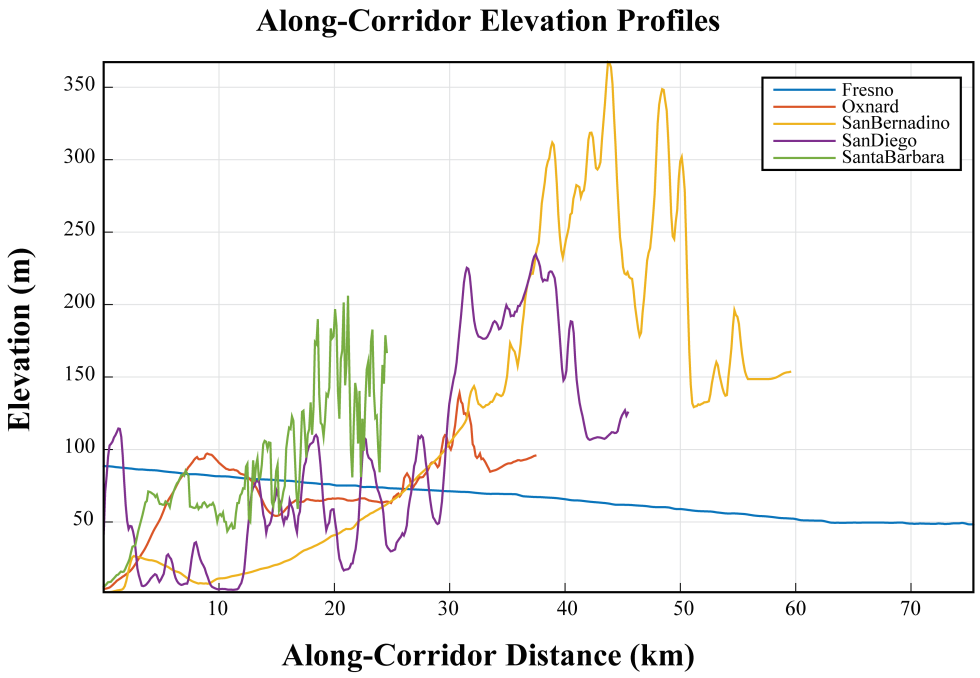


Figure 4.69: Comparison of the Along Corridor Elevation Profiles for each of the Solution Corridors

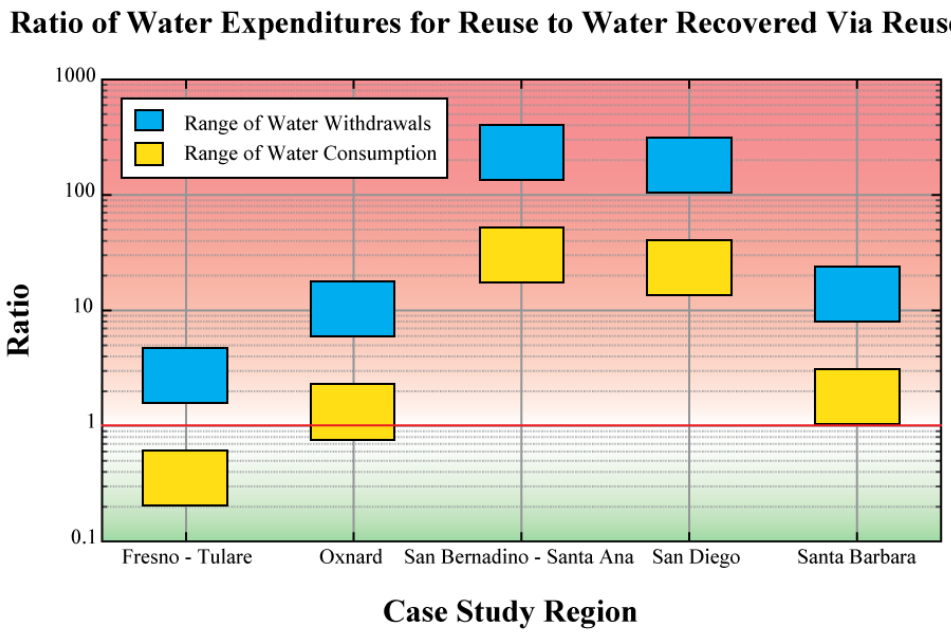


Figure 4.70: Comparison of the Net Water Usage Efficiencies of Reuse for each of the Case Study Regions Measured in Terms of Both the Withdrawals and Consumption of Water for the Production of Energy Required for Reuse

A

Go Language Source Code Repository for a Parallel + Concurrent Implementation of the MOGADOR Algorithm for the

`https://github.com/ericdfournier/corridor`

<LICENSE>

<initialize.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package corridor

import (
        "errors"
        "fmt"
        "math"
        "runtime"
        "sort"

        "github.com/gonum/diff/fd"
        "github.com/gonum/matrix/mat64"
        "github.com/satori/go.uuid"
)

// new problem parameters function
func NewParameters(sourceSubscripts,
    destinationSubscripts []int,
    populationSize, evolutionSize int,
    randomnessCoefficient float64) *Parameters {

        // set default mutation count
        mutationCount := 1

        // set defacult mutation fraction
        mutationFraction := 0.2

        // set selection fraction
        selectionFraction := 0.5

        // set selection probability
        selectionProbability := 0.8

        // get concurrency size
        maxConcurrency := runtime.NumCPU()

        // return output
        return &Parameters{
                SrcSubs: sourceSubscripts,
                DstSubs: destinationSubscripts,
                RndCoef: randomnessCoefficient,
                PopSize: populationSize,
```

```
                SelFrac: selectionFraction,
                SelProb: selectionProbability,
                MutaCnt: mutationCount,
                MutaFrc: mutationFraction,
                EvoSize: evolutionSize,
                ConSize: maxConcurrency,
        }
}

// new domain initialization function
func NewDomain(domainMatrix *mat64.Dense) *Domain {

        // get domain size
        rows, cols := domainMatrix.Dims()

        // compute band count
        bandCount := 2 + (int(math.Floor(math.Sqrt(math.Pow(float64(rows), 2.0)+math.Pow(float64(cols), 2.0)))) / 142)

        //return output
        return &Domain{
                Rows:   rows,
                Cols:   cols,
                Matrix: domainMatrix,
                BndCnt: bandCount,
        }
}

// new objective initialization function
func NewObjective(identifier int, fitnessMatrix *mat64.Dense) *Objective {

        // return output
        return &Objective{
                Id:     identifier,
                Matrix: fitnessMatrix,
        }
}

// new basis solution initialization function
func NewBasis(sourceSubs, destinationSubs []int, searchDomain *Domain) *Basis {

        // compute all minimum euclidean distances for search domain
        allMinimumDistances := AllMinDistance(sourceSubs, destinationSubs, searchDomain.Matrix)

        // generate subscripts from bresenham's algorithm
        subs := Bresenham(sourceSubs, destinationSubs)

        // compute maximum permitted chromosome length
```

```
        maxLength := len(subs) * 10

        // return output
        return &Basis{
                Matrix: allMinimumDistances,
                Subs:   subs,
                MaxLen: maxLength,
        }
}

// new chromosome initialization function
func NewChromosome(searchDomain *Domain,
    searchParameters *Parameters,
    searchObjectives *MultiObjective) *Chromosome {

        // generate node subscripts
        nodeSubs := NewNodeSubs(searchDomain, searchParameters)

        // generate subscripts from directed walk procedure
        subs := MultiPartDirectedWalk(nodeSubs, searchDomain, searchParameters)

        // initialize empty fitness place holders
        fitVal := make([][]float64, searchObjectives.ObjectiveCount)
        for i := 0; i < searchObjectives.ObjectiveCount; i++ {
                fitVal[i] = make([]float64, len(subs))
        }
        totFit := make([]float64, searchObjectives.ObjectiveCount)
        var aggFit float64 = 0.0

        // generate placeholder variables
        uuid := uuid.NewV4()

        // return output
        return &Chromosome{
                Id:               uuid,
                Subs:             subs,
                Fitness:          fitVal,
                TotalFitness:     totFit,
                AggregateFitness: aggFit,
        }
}

// new empty chromosome initialization function
func NewEmptyChromosome(searchDomain *Domain, searchObjectives *MultiObjective) *Chromosome {

        // initialize subscripts
        subs := make([][]int, 0)
```

```
        // generate placeholder id
        uuid := uuid.NewV4()

        // initialize empty fitness place holders
        fitVal := make([][]float64, searchObjectives.ObjectiveCount)
        for i := 0; i < searchObjectives.ObjectiveCount; i++ {
                fitVal[i] = make([]float64, len(subs))
        }
        totFit := make([]float64, searchObjectives.ObjectiveCount)
        var aggFit float64 = 0.0

        // return output
        return &Chromosome{
                Id:               uuid,
                Subs:             subs,
                Fitness:          fitVal,
                TotalFitness:     totFit,
                AggregateFitness: aggFit,
        }
}

// new population initialization function
func NewPopulation(identifier int,
    searchDomain *Domain,
    searchParameters *Parameters,
    searchObjectives *MultiObjective) *Population {

        // initialize communication channel
        chr := make(chan *Chromosome, searchParameters.PopSize)

        // initialize new empty chromosome before entering loop
        newChrom := NewEmptyChromosome(searchDomain, searchObjectives)

        // initialize concurrency limit channel
        conc := make(chan bool, searchParameters.ConSize)

        // generate chromosomes via go routines
        for i := 0; i < searchParameters.PopSize; i++ {

                // write to control channel
                conc <- true

                // launch chromosome initialization go routines
                go func(searchDomain *Domain, searchParameters *Parameters, searchObjectives *MultiObjective) {
                        defer func() { <-conc }()
                        newChrom = NewChromosome(searchDomain, searchParameters, searchObjectives)
```

```
                        chr <- ChromosomeFitness(newChrom, searchObjectives)
                }(searchDomain, searchParameters, searchObjectives)
        }

        // cap parallelism at concurrency limit
        for j := 0; j < cap(conc); j++ {
                conc <- true
        }

        // initialize fitness placeholder
        meanFit := make([]float64, searchObjectives.ObjectiveCount)
        var aggMeanFit float64 = 0.0

        // return output
        return &Population{
                Id:                  identifier,
                Chromosomes:         chr,
                MeanFitness:         meanFit,
                AggregateMeanFitness: aggMeanFit,
        }

}

// new empty population initialization function
func NewEmptyPopulation(identifier int, searchObjectives *MultiObjective) *Population {

        // initialize empty chromosomes channel
        chr := make(chan *Chromosome)

        // initialize fitness placeholder
        meanFit := make([]float64, searchObjectives.ObjectiveCount)
        var aggMeanFit float64 = 0.0

        // return output
        return &Population{
                Id:                  identifier,
                Chromosomes:         chr,
                MeanFitness:         meanFit,
                AggregateMeanFitness: aggMeanFit,
        }
}

// new empty evolution initialization function
func NewEmptyEvolution(searchParameters *Parameters) *Evolution {

        // initialize empty population channel
        popChan := make(chan *Population, searchParameters.EvoSize)
```

```
        // initialize empty fitness gradient
        gradFit := make([]float64, searchParameters.EvoSize)

        // return output
        return &Evolution{
                Populations:     popChan,
                FitnessGradient: gradFit,
        }
}

// new evolution initialization function
func NewEvolution(searchParameters *Parameters,
    searchDomain *Domain,
    searchObjectives *MultiObjective) *Evolution {

        // initialize seed population identifier
        var popID int = 0

        // initialize population channel
        popChan := make(chan *Population, 1)

        // print initialization status message
        fmt.Println("Initializing Seed Population...")

        // initialize seed population
        seedPop := NewPopulation(popID, searchDomain, searchParameters, searchObjectives)
        popChan <- PopulationFitness(seedPop, searchParameters, searchObjectives)

        // initialize raw fitness data slice
        rawAggMeanFit := make([]float64, searchParameters.EvoSize)

        // initialize fitness gradient variable
        gradFit := make([]float64, searchParameters.EvoSize)

        // enter loop
        for i := 0; i < searchParameters.EvoSize; i++ {

                // perform population evolution
                newPop := PopulationEvolution(<-popChan, searchDomain, searchParameters, searchObjectives)

                // compute population fitness
                newPop = PopulationFitness(newPop, searchParameters, searchObjectives)

                // write aggregate mean fitness value to vector
                rawAggMeanFit[i] = newPop.AggregateMeanFitness
```

```
// generate inline fitness gradient function
var fitnessGradFnc = func(n float64) float64 { return rawAggMeanFit[int(n)] }

// compute fitness gradient
gradFit[i] = fd.Derivative(fitnessGradFnc, float64(i), nil)

// skip gradient check on first iteration
if i < 1 {

        // return new population to channel
        popChan <- newPop

        // increment progress bar
        fmt.Println("Evolution: ", i+1)

} else if i >= 1 && i < (searchParameters.EvoSize-1) {

        if gradFit[i] > 0 {

                // return current population to channel
                popChan <- newPop

                // close population channel
                close(popChan)

                // print success message
                fmt.Println("Convergence Achieved, Evolution Commplete!")

                // break loop
                break

        } else if gradFit[i] <= 0 {

                // return new population to channel
                popChan <- newPop

                // increment progress bar
                fmt.Println("Evolution: ", i+1)

        }

} else if i == searchParameters.EvoSize-1 {

        // return new population to channel
        popChan <- newPop

        // close population channel
```

```
                        close(popChan)

                        // print termination message
                        fmt.Println("Convergence Not Achieved, Maximum Number of Evolutions Reached...")

                        // break loop
                        break
                }
        }

        // return output
        return &Evolution{
                Populations:     popChan,
                FitnessGradient: gradFit,
        }
}

// function to return copies of a user specified fraction of
// the individual chromosomes within a population ranked in terms
// of individual aggregate fitness
func NewEliteFraction(inputFraction float64,
    inputPopulation *Population) (outputChromosomes []*Chromosome) {

        // count input chromosomes
        chromCount := cap(inputPopulation.Chromosomes)

        // initialize aggregate score slice
        chromFrac := int(math.Ceil(inputFraction * float64(chromCount)))

        // initialize chromosome map
        chromMap := make(map[float64]*Chromosome)

        // initialize chromosome map key slice
        chromKey := make([]float64, chromCount)

        // initialize output slice
        output := make([]*Chromosome, chromFrac)

        // loop through channel to populate slice
        for i := 0; i < chromCount; i++ {
                curChrom := <-inputPopulation.Chromosomes
                chromMap[curChrom.AggregateFitness] = curChrom
                chromKey[i] = curChrom.AggregateFitness
        }

        // sort on aggregate fitness keys
        sort.Float64s(chromKey)
```

```
        // loop through and generate output slice faction
        for j := 0; j < chromFrac; j++ {
                output[j] = chromMap[chromKey[j]]
        }

        // return output
        return output
}

// function to return copies of a user specified number of
// unique individual chromosomes from within a population
// with each chromosome being ranked in terms of its
// individual aggregate fitness
func NewEliteSet(inputCount int,
    inputPopulation *Population,
        inputParameters *Parameters) (outputChromosomes []*Chromosome) {

        // check band count against population size
        if inputCount >= int(math.Floor((0.5 * float64(inputParameters.PopSize)))) {
                err := errors.New("Input elite set count must be less than 1/2 the input population size \n")
                panic(err)
        }

        // count input chromosomes
        chromCount := cap(inputPopulation.Chromosomes)

        // initialize chromosome map
        chromMap := make(map[float64]*Chromosome)

        // initialize chromosome map key slice
        chromKey := make([]float64, chromCount)

        // initialize output slice
        output := make([]*Chromosome, inputCount)

        // loop through channel to populate slice from channel
        for i := 0; i < chromCount; i++ {
                curChrom := <-inputPopulation.Chromosomes
                chromMap[curChrom.AggregateFitness] = curChrom
                chromKey[i] = curChrom.AggregateFitness
                inputPopulation.Chromosomes <- curChrom
        }

        // sort on aggregate fitness keys
        sort.Float64s(chromKey)
```

```
        // initalize iteration counter
        var iter int = 0

        // loop through and generate output slice set
        for j := 0; j < chromCount; j++ {

                // deal with initial state
                if j == 0 {
                        output[iter] = chromMap[chromKey[j]]
                        iter += 1
                        continue
                }

                // get uuids
                prevUuid := chromMap[chromKey[j-1]].Id.String()
                curUuid := chromMap[chromKey[j]].Id.String()

                // impose uniqueness constraint
                if prevUuid != curUuid {
                        output[iter] = chromMap[chromKey[j]]
                        iter += 1
                } else {
                        continue
                }

                // stop if inputCount reached
                if iter == inputCount {
                        break
                }
        }

        // return output
        return output
}
```

<io.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package corridor

import (
        "encoding/csv"
        "fmt"
        "os"
        "strconv"
        "time"

        "github.com/gonum/matrix/mat64"
)

// function to write an input comma separated value
// file's contents to an output domain structure
func CsvToSubs(inputFilepath string) (outputSubs []int) {

        // open file
        data, err := os.Open(inputFilepath)

        // parse error if file not found
        if err != nil {
                fmt.Println(err)
                return
        }

        // close file on completion
        defer data.Close()

        // generate new reader from open file
        reader := csv.NewReader(data)

        // set reader structure field
        reader.FieldsPerRecord = -1

        // use reader to read raw csv data
        rawCSVdata, err := reader.ReadAll()

        // parse csv file formatting errors
        if err != nil {
                fmt.Println(err)
                os.Exit(1)
```

```
        }

        // initialize empty row and column counts
        rows := len(rawCSVdata)
        cols := len(rawCSVdata[0])

        // initialize output
        output := make([]int, 2)

        // loop through and extract values
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {

                        // get string value and convert to integer
                        strVal := rawCSVdata[i][j]
                        intVal, err := strconv.Atoi(strVal)

                        // shift value by one to account for buffer boundaries
                        output[j] = intVal + 1

                        // parse error
                        if err != nil {
                                fmt.Println(err)
                                os.Exit(1)
                        }
                }
        }

        // return output
        return output
}

// function to write an input comma separated value
// file's contents to an output domain structure
func CsvToDomain(inputFilepath string) (outputDomain *Domain) {

        // open file
        data, err := os.Open(inputFilepath)

        // parse error if file not found
        if err != nil {
                fmt.Println(err)
                return
        }

        // close file on completion
        defer data.Close()
```

```
// generate new reader from open file
reader := csv.NewReader(data)

// set reader structure field
reader.FieldsPerRecord = -1

// use reader to read raw csv data
rawCSVdata, err := reader.ReadAll()

// parse csv file formatting errors
if err != nil {
        fmt.Println(err)
        os.Exit(1)
}

// initialize empty row and column counts
rows := len(rawCSVdata)
cols := len(rawCSVdata[0])

// initialize domain matrix
domMat := mat64.NewDense(rows+2, cols+2, nil)

// write values from rawCSVdata to domain matrix
for i := 0; i < rows+2; i++ {
        for j := 0; j < cols+2; j++ {
                // create a 1 pixel boundary buffer of zeros
                if i == 0 {
                        domMat.Set(i, j, 0.0)
                } else if i == rows+1 {
                        domMat.Set(i, j, 0.0)
                } else if j == 0 {
                        domMat.Set(i, j, 0.0)
                } else if j == cols+1 {
                        domMat.Set(i, j, 0.0)
                } else {

                        // get string value and convert to integer
                        strVal := rawCSVdata[i-1][j-1]
                        fltVal, err := strconv.ParseFloat(strVal, 64)

                        // parse error
                        if err != nil {
                                fmt.Println(err)
                                os.Exit(1)
                        }
```

```
                        // write value to matrix
                        domMat.Set(i, j, fltVal)
                    }
                }
        }

        // initialize new domain
        output := NewDomain(domMat)

        // return output
        return output
}

// function to write an input comma separated value
// file's contents to an output objective structure
func CsvToObjective(identifier int, inputFilepath string) (outputObjective *Objective) {

        // open file
        dataFile, err := os.Open(inputFilepath)

        // parse error if file not found
        if err != nil {
                fmt.Println(err)
                return
        }

        // close file on completion
        defer dataFile.Close()

        // generate new reader from open file
        reader := csv.NewReader(dataFile)

        // set reader structure field
        reader.FieldsPerRecord = -1

        // use reader to read raw csv data
        rawCSVdata, err := reader.ReadAll()

        // parse csv file formatting errors
        if err != nil {
                fmt.Println(err)
                os.Exit(1)
        }

        // initialize empty row and column counts
        rows := len(rawCSVdata)
        cols := len(rawCSVdata[0])
```

```
        // initialize domain matrix
        objMat := mat64.NewDense(rows+2, cols+2, nil)

        // write values from rawCSVdata to domain matrix
        for i := 0; i < rows+2; i++ {
                for j := 0; j < cols+2; j++ {

                        // create a 1 pixel boundary buffer of zeros
                        if i == 0 {
                                objMat.Set(i, j, 0.0)
                        } else if i == rows+1 {
                                objMat.Set(i, j, 0.0)
                        } else if j == 0 {
                                objMat.Set(i, j, 0.0)
                        } else if j == cols+1 {
                                objMat.Set(i, j, 0.0)
                        } else {

                                // get string value and convert to float
                                strVal := rawCSVdata[i-1][j-1]
                                fltVal, err := strconv.ParseFloat(strVal, 64)

                                // parse error
                                if err != nil {
                                        fmt.Println(err)
                                        os.Exit(1)
                                }

                                // write matrix value
                                objMat.Set(i, j, fltVal)
                        }
                }
        }

        // initialize new domain
        output := NewObjective(identifier, objMat)

        // return output
        return output
}

// function to write a set of input comma separated value
// files' contents to an output multiobjective structure
func CsvToMultiObjective(inputFilepaths ...string) (outputMultiObjective *MultiObjective) {

        // get variadic input length
```

```
        objectiveCount := len(inputFilepaths)

        // initialize objective slice
        objectiveSlice := make([]*Objective, objectiveCount)

        // initialize objectives identifier
        var objectiveID int = 0

        // loop through and extract objectives
        for i := 0; i < objectiveCount; i++ {

                // read CSV data to objective
                objectiveSlice[i] = CsvToObjective(objectiveID, inputFilepaths[i])

                // increment objective identifier
                objectiveID += 1

        }

        // return multiObjective output
        return &MultiObjective{
                ObjectiveCount: objectiveCount,
                Objectives:     objectiveSlice,
        }
}

// function to write the values from an input
// chromosome structure to an output csv file
func ChromosomeToString(inputChromosome *Chromosome) (outputRawString [][]string) {

        // get input chromosome length
        chromLen := len(inputChromosome.Subs)

        // count input chromosome objectives
        objCount := len(inputChromosome.TotalFitness)

        // intitialize raw output string slice
        rawCSVdata := make([][]string, objCount+2)

        // loop through and format values as strings for output encoding
        for j := 0; j < objCount+2; j++ {

                // allocate inner slice
                rawCSVdata[j] = make([]string, chromLen)

                for i := 0; i < chromLen; i++ {
```

```
                        // transpose subs by one to account for boundary buffer
                        if j == 0 {
                                rawCSVdata[j][i] = strconv.Itoa(inputChromosome.Subs[i][0] - 1)
                        } else if j == 1 {
                                rawCSVdata[j][i] = strconv.Itoa(inputChromosome.Subs[i][1] - 1)
                        } else {
                                rawCSVdata[j][i] = strconv.FormatFloat(inputChromosome.Fitness[j-2][i], 'f', 2, 64)
                        }
                }
        }

        // return output
        return rawCSVdata
}

// function to write the values from an input elite set
// to an output csv file
func EliteSetToCsv(inputEliteSet []*Chromosome, outputFilepath string) {

        // open file
        csvfile, err := os.Create(outputFilepath)

        // parse file opening errors
        if err != nil {
                fmt.Println("Error:", err)
                return
        }

        // close file on completion
        defer csvfile.Close()

        // get chromosome count
        chromCount := len(inputEliteSet)

        // initialize rawCSVdata and chromosome string structures
        var chromString, rawCSVdata [][]string

        // loop through chromsomes and generate composite string structure
        for i := 0; i < chromCount; i++ {
                chromString = ChromosomeToString(inputEliteSet[i])
                rawCSVdata = append(rawCSVdata, chromString...)
        }

        // initialize writer object
        writer := csv.NewWriter(csvfile)

        // write data or get error
```

```
        err = writer.WriteAll(rawCSVdata)

        // parse errors
        if err != nil {
                fmt.Println("Error:", err)
                return
        }

        // flush writer object
        writer.Flush()
}

// function to write the runtime parameters from an evolution
// to an output csv file
func RuntimeLogToCsv(inputEvolution *Evolution,
    inputRuntime time.Duration,
    outputFilepath string) {

        // open file
        csvfile, err := os.Create(outputFilepath)

        // parse file opening errors
        if err != nil {
                fmt.Println("Error:", err)
                return
        }

        // close file on completion
        defer csvfile.Close()

        // initialize rawCSVdata structure
        var rawCSVdata []string

        // populate string slice
        rawCSVdata = append(rawCSVdata, inputRuntime.String())

        // initalize evolutionary counter
        var evo int = 0

        // count required evolutions
        for i := 0; i < len(inputEvolution.FitnessGradient); i++ {
                if inputEvolution.FitnessGradient[i] != 0 {
                        evo += 1
                } else {
                        continue
                }
        }
```

```
        // convert to string
        rawCSVdata = append(rawCSVdata, strconv.Itoa(evo))

        // initialize writer object
        writer := csv.NewWriter(csvfile)

        // write data or get error
        err = writer.Write(rawCSVdata)

        // parse errors
        if err != nil {
                fmt.Println("Error:", err)
                return
        }

        // flush writer object
        writer.Flush()
}
```

<lib.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.package main

package corridor

import (
        "errors"
        "math"

        "github.com/gonum/matrix/mat64"
)

// compute euclidean distance for a pair of subscript indices
func Distance(aSubs, bSubs []int) (dist float64) {

        // initialize variables
        var x0 float64 = float64(aSubs[0])
        var x1 float64 = float64(bSubs[0])
        var y0 float64 = float64(aSubs[1])
        var y1 float64 = float64(bSubs[1])
        var pow float64 = 2.0
        var dx float64 = x1 - x0
        var dy float64 = y1 - y0

        // compute distance
        var output float64 = math.Sqrt(math.Pow(dx, pow) + math.Pow(dy, pow))

        // return final output
        return output
}

// alldistance computes the distance from each location with the input
// search domain and a given point defined by an input pair of row
// column subscripts
func AllDistance(aSubs []int,
    searchDomainMatrix *mat64.Dense) (allDistMatrix *mat64.Dense) {

        // get matrix dimensions
        rows, cols := searchDomainMatrix.Dims()

        // initialize new output matrix
        output := mat64.NewDense(rows, cols, nil)

        // initialize destination point subscript slice
```

```
        bSubs := make([]int, 2)

        // loop through all values and compute minimum distances
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {
                        bSubs[0] = i
                        bSubs[1] = j
                        output.Set(bSubs[0], bSubs[1], Distance(aSubs, bSubs))
                }
        }

        // return output
        return output

}

// compute the minimum distance between a given input point and
// the subscripts comprised of a line segement joining two other
// input points
func MinDistance(pSubs, aSubs, bSubs []int) (minDist float64) {

        // initialize variables
        var x float64 = float64(pSubs[0])
        var y float64 = float64(pSubs[1])
        var x0 float64 = float64(aSubs[0])
        var y0 float64 = float64(aSubs[1])
        var x1 float64 = float64(bSubs[0])
        var y1 float64 = float64(bSubs[1])

        // compute difference components
        a := x - x0
        b := y - y0
        c := x1 - x0
        d := y1 - y0

        // compute dot product of difference components
        dot := a*c + b*d
        lenSq := c*c + d*d

        // initialize parameter
        var param float64 = -1.0

        // if zero length condition
        if lenSq != 0 {
                param = dot / lenSq
        }
```

```
        // initialize transform variables
        var xx, yy float64

        // switch transform mechanism on orientation
        if param < 0 {
                xx = x0
                yy = y0
        } else if param > 1 {
                xx = x1
                yy = y1
        } else {
                xx = x0 + param*c
                yy = y0 + param*d
        }

        // execute transform
        var dx float64 = x - xx
        var dy float64 = y - yy

        // generate output
        output := math.Sqrt(dx*dx + dy*dy)

        // return final output
        return output
}

// allmindistance computes the distance from each location within the
// input search domain and to the nearest subscript located along the
// line formed by the two input subscripts
func AllMinDistance(aSubs, bSubs []int,
    searchDomainMatrix *mat64.Dense) (allMinDistMatrix *mat64.Dense) {

        // get matrix dimensions
        rows, cols := searchDomainMatrix.Dims()

        // initialize new output matrix
        output := mat64.NewDense(rows, cols, nil)

        // initialize slice
        pSubs := make([]int, 2)

        // loop through all values and compute minimum distances
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {
                        pSubs[0] = i
                        pSubs[1] = j
                        curMinDist := MinDistance(pSubs, aSubs, bSubs)
```

```
                        output.Set(pSubs[0], pSubs[1], curMinDist)
                }
        }

        // return final output
        return output
}

// distancebands recodes a distance matrix computed from a single
// source location to ordinal set of bands of increasing distance
func DistanceBands(bandCount int, distanceMatrix *mat64.Dense) (bandMatrix *mat64.Dense) {

        // get matrix dimensions
        rows, cols := distanceMatrix.Dims()

        // check band count against input distance matrix size
        if bandCount > rows || bandCount > cols {
                err := errors.New("Input band count too large for input distance matrix \n")
                panic(err)
        }

        // initialize output
        output := mat64.NewDense(rows, cols, nil)

        // generate band range
        minDist := distanceMatrix.Min()
        maxDist := distanceMatrix.Max()

        // initialize band interval unit and slice
        bandUnit := (maxDist - minDist) / float64(bandCount+1)
        bandInt := make([]float64, bandCount+1)

        // generate band intervals
        for i := 0; i < bandCount+1; i++ {
                if i == 0 {
                        bandInt[i] = 0
                } else {
                        bandInt[i] = bandInt[i-1] + bandUnit
                }
        }

        // perform conversion to the appropriate band interval
        for i := 0; i < len(bandInt)-1; i++ {
                for j := 0; j < rows; j++ {
                        for k := 0; k < rows; k++ {
                                if distanceMatrix.At(j, k) >= bandInt[i] && distanceMatrix.At(j, k) < bandInt[i+1] {
                                        output.Set(j, k, float64(i))
```

```
                        } else if distanceMatrix.At(j, k) >= bandInt[i+1] {
                                output.Set(j, k, float64(i+1))
                        }
                }
        }
    }

    // return output
    return output
}

// bandmask selects the elements in a distance band matrix
// corresponding to a specified input band identification number
// and outputs a binary matrix of the same dimensions as the distance
// band matrix with the values at those locations encoded as ones
// and all other locations encoded as zeros
func BandMask(bandValue float64, bandMatrix *mat64.Dense) (binaryBandMat *mat64.Dense) {

    // get row column dimensions of band matrix
    rows, cols := bandMatrix.Dims()

    // initialize output
    output := mat64.NewDense(rows, cols, nil)

    // loop through matrix values and perform binary encoding
    for i := 0; i < rows; i++ {
        for j := 0; j < cols; j++ {

            // perform elementwise equality test
            if i == 0 || i == rows-1 || j == 0 || j == cols-1 {
                output.Set(i, j, 0.0)
            } else {
                if bandValue == bandMatrix.At(i, j) {
                    output.Set(i, j, 1.0)
                } else {
                    output.Set(i, j, 0.0)
                }
            }
        }
    }

    // return output
    return output
}

// nonzerosubs returns a 2-D slice containing the row column indices
// of all nonzero elements contained wihtin a given input matrix
```

```
func NonZeroSubs(inputMatrix *mat64.Dense) (nonZeroSubs [][]int) {

        // get matrix dimensions
        rows, cols := inputMatrix.Dims()

        // initialize output
        output := make([][]int, 1)
        output[0] = make([]int, 2)

        // initialize iterator and current subscript slice
        var iter int = 0

        // loop through and check values
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {

                        // test for non-zero values
                        if inputMatrix.At(i, j) != 0.0 {
                                if iter == 0 {
                                        output[iter] = []int{i, j}
                                        iter += 1
                                } else if iter > 0 {
                                        output = append(output, []int{i, j})
                                        iter += 1
                                }
                        }
                }
        }

        // return output
        return output
}

// findsubs returns a 2-D slice containing the row column indices
// of all of the elements contained wihtin a given input matrix
// that are equal in value to some provided input value
func FindSubs(inputValue float64, inputMatrix *mat64.Dense) (foundSubs [][]int) {

        // get matrix dimensions
        rows, cols := inputMatrix.Dims()

        // initialize output
        output := make([][]int, 1)
        output[0] = make([]int, 2)

        // initialize iterator and current subscript slice
        var iter int = 0
```

```
        // loop through and check values
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {

                        // test for equality
                        if inputMatrix.At(i, j) == inputValue {
                                if iter == 0 {
                                        output[iter] = []int{i, j}
                                        iter += 1
                                } else if iter > 0 {
                                        output = append(output, []int{i, j})
                                        iter += 1
                                }
                        }
                }
        }

        // return output
        return output
}

// orientation accepts as inputs a pair of point subscripts
// and returns a binary vector indicating the relative orientation
// of the first point to the second in binary terms
func Orientation(aSubs, bSubs []int) (orientationVector []int) {

        // initialize output
        output := make([]int, 2)

        // generate reference orientation row parameter
        if aSubs[0]-bSubs[0] < 0 {
                output[0] = 1
        } else if aSubs[0]-bSubs[0] == 0 {
                output[0] = 0
        } else if aSubs[0]-bSubs[0] > 0 {
                output[0] = -1
        }

        // generate reference orientation column parameter
        if aSubs[1]-bSubs[1] < 0 {
                output[1] = 1
        } else if aSubs[1]-bSubs[1] == 0 {
                output[1] = 0
        } else if aSubs[1]-bSubs[1] > 0 {
                output[1] = -1
        }
```

```
        // return output
        return output
}

// orientation mask returns a binary encoded matrix for
// a given point where all points orientated towards
// a given second point are encoded as 1 and all points
// orientated away from the given second point as 0
func OrientationMask(aSubs, bSubs []int,
    searchDomainMatrix *mat64.Dense) (orientationMask *mat64.Dense) {

        // generate matrix dimensions
        rows, cols := searchDomainMatrix.Dims()

        // initialize output matrix
        output := mat64.NewDense(rows, cols, nil)

        // generate reference orientation vectors
        sRefOrientVec := Orientation(aSubs, bSubs)
        dRefOrientVec := Orientation(bSubs, aSubs)

        // initialize current subs and orientation vectors
        curSubs := make([]int, 2)
        sOrientVec := make([]int, 2)
        dOrientVec := make([]int, 2)

        // loop through domain matrix and generate orientation matrix values
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {

                        // compute current orientation
                        curSubs[0] = i
                        curSubs[1] = j
                        sOrientVec = Orientation(curSubs, bSubs)
                        dOrientVec = Orientation(curSubs, aSubs)

                        // check for match and assign values
                        if sOrientVec[0] == sRefOrientVec[0] && sOrientVec[1] == sRefOrientVec[1] {
                                if dOrientVec[0] == dRefOrientVec[0] && dOrientVec[1] == dRefOrientVec[1] {
                                        output.Set(i, j, 1.0)
                                }
                        }
                }
        }

        // return output
```

```
        return output
}

// bresenham generates the list of subscript indices corresponding to the
// euclidean shortest paths connecting two subscript pairs in discrete space
func Bresenham(aSubs, bSubs []int) (lineSubs [][]int) {

        // initialize variables
        var x0 int = aSubs[0]
        var x1 int = bSubs[0]
        var y0 int = aSubs[1]
        var y1 int = bSubs[1]

        // check row differential
        dx := x1 - x0
        if dx < 0 {
                dx = -dx
        }

        // check column differential
        dy := y1 - y0

        // if differential is negative flip
        if dy < 0 {
                dy = -dy
        }

        // initialize stride variables
        var sx, sy int

        // set row stride direction
        if x0 < x1 {
                sx = 1
        } else {
                sx = -1
        }

        // set column stride direction
        if y0 < y1 {
                sy = 1
        } else {
                sy = -1
        }

        // calculate error component
        err := dx - dy
```

```
        // initialize output 2D slice vector
        dist := math.Ceil(Distance(aSubs, bSubs))
        maxLen := int(dist)
        output := make([][]int, 0, maxLen)

        // loop through and generate subscripts
        for {
                var val = []int{x0, y0}
                output = append(output, val)
                if x0 == x1 && y0 == y1 {
                        break
                }
                e2 := 2 * err
                if e2 > -dy {
                        err -= dy
                        x0 += sx
                }
                if e2 < dx {
                        err += dx
                        y0 += sy
                }
        }

        // return final output
        return output
}

// function to return the subscript indices of the cells corresponding to the
// queens neighborhood for a given subscript pair
func NeighborhoodSubs(aSubs []int) (neighSubs [][]int) {

        // initialize output slice
        output := make([][]int, 0)

        // write neighborhood subscript values
        output = append(output, []int{aSubs[0] - 1, aSubs[1] - 1})
        output = append(output, []int{aSubs[0] - 1, aSubs[1]})
        output = append(output, []int{aSubs[0] - 1, aSubs[1] + 1})
        output = append(output, []int{aSubs[0], aSubs[1] - 1})
        output = append(output, []int{aSubs[0], aSubs[1]})
        output = append(output, []int{aSubs[0], aSubs[1] + 1})
        output = append(output, []int{aSubs[0] + 1, aSubs[1] - 1})
        output = append(output, []int{aSubs[0] + 1, aSubs[1]})
        output = append(output, []int{aSubs[0] + 1, aSubs[1] + 1})

        // return output
        return output
```

```
}

// function to validate an input sub domain for use in generating
// a chromosomal mutation via the random walk procedure
func ValidateSubDomain(subSource, subDestin []int, subMat *mat64.Dense) bool {

        // initialize output
        var output bool

        // generate sub source neighborhood
        sNeigh := NeighborhoodSubs(subSource)

        // generate sub destination neighborhood
        dNeigh := NeighborhoodSubs(subDestin)

        // generate center row
        centerRow := subMat.RowView(2)

        // generate center column
        centerCol := subMat.ColView(2)

        // initialize summation variables
        var sSum float64 = 0.0
        var dSum float64 = 0.0
        var rSum float64 = 0.0
        var cSum float64 = 0.0

        // enter for loop for start and destination sums
        for i := 0; i < 9; i++ {
                sSum = sSum + subMat.At(sNeigh[i][0], sNeigh[i][1])
                dSum = dSum + subMat.At(dNeigh[i][0], dNeigh[i][1])
        }

        // enter for loop for row column sums
        for j := 0; j < 5; j++ {
                rSum = rSum + centerRow.At(j, 0)
                cSum = cSum + centerCol.At(j, 0)
        }

        // check conditions to validate neighborhood
        if sSum <= 1.0 || dSum <= 1.0 || rSum == 0.0 || cSum == 0.0 {
                output = false
        } else {
                output = true
        }

        //return final output
```

```
        return output
}

// function validate the tabu neighborhood of an input pair of
// row column subscripts
func ValidateTabu(currentSubs []int, tabuMatrix *mat64.Dense) bool {

        // initialize output
        var output bool

        // initialize tabu neighborhood sum
        var tSum int = 0

        // generate neighborhood subscripts
        tNeigh := NeighborhoodSubs(currentSubs)

        // loop through and compute sum
        for i := 0; i < 9; i++ {
                if i != 4 {
                        tSum += int(tabuMatrix.At(tNeigh[i][0], tNeigh[i][1]))
                }
        }

        // write output boolean
        if tSum == 0 {
                output = false
        } else {
                output = true
        }

        // return output
        return output
}

// function to count the number of digits in an input integer as
// its base ten logarithm
func DigitCount(input int) (digits int) {

        // compute digits as the log of the input
        output := int(math.Floor(math.Log10(float64(input))))

        // return output
        return output
}
```

<lib_test.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.package main

package corridor

import (
        "math"
        "testing"

        "github.com/gonum/matrix/mat64"
)

// test Distance function
func TestDistance(t *testing.T) {

        // initialize test case
        t.Log("Distance Test: Expected Distance = 10")

        // initialize expected values
        var expValue float64 = 10.0

        // initialize test case variables
        var aSubs = []int{0, 10}
        var bSubs = []int{0, 20}

        // perform test case
        testCase := Distance(aSubs, bSubs)

        // log test result
        if testCase == expValue {
                t.Log("Distance Test: Computed Distance =", testCase)
        } else {
                t.Error("Distance Test: Computed Distance =", testCase)
        }
}

// test AllDistance function
func TestAllDistance(t *testing.T) {

        // initialize test case
        t.Log("AllDistance Test: Expected Matrix =
            {{3 3 3 [1.4142135623730951 1 1.4142135623730951 1 0 1 1.4142135623730951 1 1.4142135623730951]} 3 3}")

        // initialize expected value
```

```
        var expValueVector = []float64{
                math.Sqrt(2.0), 1.0, math.Sqrt(2.0),
                1.0, 0, 1.0,
                math.Sqrt(2.0), 1.0, math.Sqrt(2.0)}
        expValueMatrix := mat64.NewDense(3, 3, expValueVector)

        // initialize test case variables
        var aSubs = []int{1, 1}
        searchDomainMatrix := mat64.NewDense(3, 3, nil)

        // perform test case
        testCase := AllDistance(aSubs, searchDomainMatrix)

        // log test result
        if testCase.Equals(expValueMatrix) == true {
                t.Log("AllDistance Test: Computed Matrix =", *testCase)
        } else {
                t.Error("AllDistance Test: Computed Matrix =", *testCase)
        }
}

// test MinDistance function
func TestMinDistance(t *testing.T) {

        // initialize test case
        t.Log("MinDistance Test: Expected Value = 1.4142135623730951")

        // initialize expected value
        var expValue float64 = math.Sqrt(2.0)

        // initialize test case variables
        var pSubs = []int{0, 2}
        var aSubs = []int{0, 0}
        var bSubs = []int{2, 2}

        // perform test case
        testCase := MinDistance(pSubs, aSubs, bSubs)

        // log test result
        if testCase == expValue {
                t.Log("MinDistance Test: Computed Value =", testCase)
        } else {
                t.Error("AllDistance Test: Computed Value =", testCase)
        }
}

// test AllMinDistance function
```

```
func TestAllMinDistance(t *testing.T) {

        // initialize test case
        t.Log("AllMinDistance Test:
            Expected Matrix = {{3 3 3 [0 1 1.4142135623730951 1 0 1 1.4142135623730951 1 0]} 3 3}")

        // initialize expected value
        var expValueVector = []float64{
                0.0, (math.Sqrt(2.0) / 2.0), math.Sqrt(2.0),
                (math.Sqrt(2.0) / 2.0), 0.0, (math.Sqrt(2.0) / 2.0),
                math.Sqrt(2.0), (math.Sqrt(2.0) / 2.0), 0.0}
        expValueMatrix := mat64.NewDense(3, 3, expValueVector)

        // initialize test case variables
        var aSubs = []int{0, 0}
        var bSubs = []int{2, 2}
        searchDomainMatrix := mat64.NewDense(3, 3, nil)

        // peform test case
        testCase := AllMinDistance(aSubs, bSubs, searchDomainMatrix)

        // log test result
        if testCase.Equals(expValueMatrix) {
                t.Log("AllMinDistance Test: Computed Matrix =", testCase)
        } else {
                t.Error("AllMinDistance Test: Computed Matrix =", testCase)
        }
}

// test DistanceBands
func TestDistanceBands(t *testing.T) {

        // initialize test case
        t.Log("DistanceBands Test: Expected Matrix = {{3 3 3 [0 1 2 1 1 2 2 2 2]} 3 3}")

        // initialize expected value
        var expValueVector = []float64{
                0.0, 1.0, 2.0,
                1.0, 1.0, 2.0,
                2.0, 2.0, 2.0}
        expValueMatrix := mat64.NewDense(3, 3, expValueVector)

        // initialize test case variables
        var aSubs = []int{0, 0}
        var bandCount int = 2
        searchDomainMatrix := mat64.NewDense(3, 3, nil)
```

```
        // compute distance matrix !! dependent on AllDistance test result !!
        distanceMatrix := AllDistance(aSubs, searchDomainMatrix)

        // perform test case
        testCase := DistanceBands(bandCount, distanceMatrix)

        // log test result
        if testCase.Equals(expValueMatrix) {
                t.Log("DistanceBands Test: Computed Matrix =", *testCase)
        } else {
                t.Error("DistanceBands Test: Computed Matrix =", *testCase)
        }
}

// test BandMask
func TestBandMask(t *testing.T) {

        // initialize test case
        t.Log("BandMask Test: Expected Matrix = {{3 3 3 [0 0 0 0 1 0 0 0 0]} 3 3}")

        // initialize expected value
        var expValueVector = []float64{
                0.0, 0.0, 0.0,
                0.0, 1.0, 0.0,
                0.0, 0.0, 0.0}
        expValueMatrix := mat64.NewDense(3, 3, expValueVector)

        // initialize test case variables
        var aSubs = []int{0, 0}
        var bandCount int = 2
        var bandValue float64 = 1.0
        searchDomainMatrix := mat64.NewDense(3, 3, nil)

        // compute distance matrix !! dependent on AllDistance test result !!
        distanceMatrix := AllDistance(aSubs, searchDomainMatrix)

        // compute band matrix !! dependent on DistanceBands test result!!
        bandMatrix := DistanceBands(bandCount, distanceMatrix)

        // perform test case
        testCase := BandMask(bandValue, bandMatrix)

        // log test results
        if testCase.Equals(expValueMatrix) {
                t.Log("BandMask Test: Computed Matrix =", *testCase)
        } else {
                t.Error("BandMask Test: Computed Matrix =", *testCase)
```

```
        }
}

// test NonZeroSubs
func TestNonZeroSubs(t *testing.T) {

        // initialize test case
        t.Log("NonZeroSubs Test: Expected Vector = [[1 1]]")

        // initialize expected value
        expValueVector := make([][]int, 1)
        expValueVector[0] = []int{1, 1}

        // initialize test case variables
        var aSubs = []int{0, 0}
        var bandCount int = 2
        var bandValue float64 = 1.0
        searchDomainMatrix := mat64.NewDense(3, 3, nil)

        // compute distance matrix !! dependent on AllDistance test result !!
        distanceMatrix := AllDistance(aSubs, searchDomainMatrix)

        // compute band matrix !! dependent on DistanceBands test result!!
        bandMatrix := DistanceBands(bandCount, distanceMatrix)

        // compute band mask !! dependent on BandMask test result!!
        bandMask := BandMask(bandValue, bandMatrix)

        // perform test case
        testCase := NonZeroSubs(bandMask)

        // log test results
        if testCase[0][0] == expValueVector[0][0] && testCase[0][1] == expValueVector[0][1] {
                t.Log("NonZeroSubs Test: Computed Vector =", testCase)
        } else {
                t.Error("NonZeroSubs Test: Computed Vector =", testCase)
        }
}

// test FindSubs
func TestFindSubs(t *testing.T) {

        // initialize test case
        t.Log("FindSubs Test: Expected Vector = [[0 0]]")

        // initialize expected value
        expValueVector := make([][]int, 1)
```

```
        expValueVector[0] = []int{0, 0}

        // initialize test case variables
        var inputValue float64 = 0.0
        var aSubs = []int{0, 0}
        searchDomainMatrix := mat64.NewDense(3, 3, nil)

        // compute distance matrix !! dependent on AllDistance test result !!
        distanceMatrix := AllDistance(aSubs, searchDomainMatrix)

        // perform test case
        testCase := FindSubs(inputValue, distanceMatrix)

        // log test results
        if testCase[0][0] == expValueVector[0][0] && testCase[0][1] == expValueVector[0][1] {
                t.Log("FindSubs Test: Computed Vector =", testCase)
        } else {
                t.Error("FindSubs Test: Computed Vector =", testCase)
        }
}

// test Orientation
func TestOrientation(t *testing.T) {

        // initialize test case
        t.Log("Orientation Test: Expected Vector = [1 1]")

        // initialize expected value
        var expValueVector = []int{1, 1}

        // initialize test case variables
        var aSubs = []int{0, 0}
        var bSubs = []int{2, 2}

        // perform test case
        testCase := Orientation(aSubs, bSubs)

        // log test results
        if testCase[0] == expValueVector[0] && testCase[1] == expValueVector[1] {
                t.Log("Orientation Test: Computed Vector =", testCase)
        } else {
                t.Error("Orientation Test: Computed Vector =", testCase)
        }
}

// test OrientationMask
func TestOrientationMask(t *testing.T) {
```

```
        // initialize test case
        t.Log("OrientationMask Test: Expected Matrix = {{3 3 3 [0 0 0 0 1 0 0 0 0]} 3 3}")

        // initialize expected value
        var expValueVector = []float64{
                0.0, 0.0, 0.0,
                0.0, 1.0, 0.0,
                0.0, 0.0, 0.0}
        expValueMatrix := mat64.NewDense(3, 3, expValueVector)

        // initialize test case variables
        var aSubs = []int{0, 0}
        var bSubs = []int{2, 2}
        searchDomainMatrix := mat64.NewDense(3, 3, nil)

        // perform test case
        testCase := OrientationMask(aSubs, bSubs, searchDomainMatrix)

        // log test results
        if testCase.Equals(expValueMatrix) {
                t.Log("OrientationMask Test: Computed Matrix =", *testCase)
        } else {
                t.Error("OrientationMask Test: Computed Matrix =", *testCase)
        }
}

// test Bresenham
func TestBresenham(t *testing.T) {

        // initialize test case
        t.Log("Bresenham Test: Expected Vector = [[0 0] [1 1] [2 2]]")

        // initialize expected value
        expValueVector := make([][]int, 1)
        expValueVector[0] = []int{0, 0}
        expValueVector = append(expValueVector, []int{1, 1})
        expValueVector = append(expValueVector, []int{2, 2})

        // initialize test case variables
        var aSubs = []int{0, 0}
        var bSubs = []int{2, 2}
        var testBool bool = true

        // perform test case
        testCase := Bresenham(aSubs, bSubs)
```

```
        // examine test results
        for i := 0; i < len(testCase); i++ {
                if testCase[i][0] != expValueVector[i][0] || testCase[i][1] != expValueVector[i][1] {
                        testBool = false
                        break
                }
        }

        // log test results
        if testBool == true {
                t.Log("Bresenham Test: Computed Vector =", testCase)
        } else {
                t.Error("Bresenham Test: Computed Vector =", testCase)
        }
}

// test NeighborhoodSubs
func TestNeighborhoodSubs(t *testing.T) {

        // initialize test case
        t.Log("NeighborhoodSubs Test: Expected Vector = [[0 0] [0 1] [0 2] [1 0] [1 1] [1 2] [2 0] [2 1] [2 2]]")

        // initialize expected value
        expValueVector := make([][]int, 0)
        for i := 0; i < 3; i++ {
                for j := 0; j < 3; j++ {
                        expValueVector = append(expValueVector, []int{i, j})
                }
        }

        // initialize test ccase variables
        var aSubs = []int{1, 1}
        var testBool bool = true

        // perform test case
        testCase := NeighborhoodSubs(aSubs)

        // examine test results
        for k := 0; k < len(testCase); k++ {
                if testCase[k][0] != expValueVector[k][0] || testCase[k][1] != expValueVector[k][1] {
                        testBool = false
                        break
                }
        }

        // log test results
        if testBool == true {
```

```
                t.Log("NeighborhoodSubs Test: Computed Vector =", testCase)
        } else {
                t.Error("NeighborhoodSubs Test: Computed Vector =", testCase)
        }
}

// test ValidateSubDomain
func TestValidateSubDomain(t *testing.T) {

        // initialize test case
        t.Log("ValidateSubDomain: Expecte Value = true")

        // initialize expected values
        var invalidVector = []float64{
                0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 0.0, 1.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0}
        invalidMatrix := mat64.NewDense(5, 5, invalidVector)
        var validVector = []float64{
                0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0}
        validMatrix := mat64.NewDense(5, 5, validVector)

        // initialize test case variables
        var subSource = []int{1, 1}
        var subDestin = []int{2, 3}
        var testCase1 bool
        var testCase2 bool

        // perform test cases
        testCase1 = ValidateSubDomain(subSource, subDestin, invalidMatrix)
        testCase2 = ValidateSubDomain(subSource, subDestin, validMatrix)

        // log test results
        if testCase1 == false && testCase2 == true {
                t.Log("ValidateSubDomain Test: Computed Value =", true)
        } else {
                t.Error("ValidateSubDomain Test: Computed Value =", false)
        }
}

// test ValidateTabu
```

```
func TestValidateTabu(t *testing.T) {

	// initialize test case
	t.Log("ValidateTabu Test: Expected Value = true")

	// initialize expected values
	var invalidVector = []float64{
		0.0, 0.0, 0.0, 0.0, 0.0,
		0.0, 0.0, 0.0, 0.0, 0.0,
		0.0, 0.0, 0.0, 0.0, 0.0,
		0.0, 0.0, 0.0, 0.0, 0.0,
		0.0, 0.0, 0.0, 0.0, 0.0}
	invalidMatrix := mat64.NewDense(5, 5, invalidVector)
	var validVector = []float64{
		0.0, 0.0, 0.0, 0.0, 0.0,
		0.0, 1.0, 1.0, 1.0, 0.0,
		0.0, 1.0, 1.0, 1.0, 0.0,
		0.0, 1.0, 1.0, 1.0, 0.0,
		0.0, 0.0, 0.0, 0.0, 0.0}
	validMatrix := mat64.NewDense(5, 5, validVector)

	// initialize test case variables
	var currentSubs = []int{2, 2}
	var testCase1 bool
	var testCase2 bool

	// perform test cases
	testCase1 = ValidateTabu(currentSubs, invalidMatrix)
	testCase2 = ValidateTabu(currentSubs, validMatrix)
	testBool := testCase1 == false && testCase2 == true

	// log test results
	if testBool {
		t.Log("ValidateTabu Test: Computed Value =", testBool)
	} else {
		t.Error("ValidateTabu Test: Computed Value =", testBool)
	}
}

// test DigitCount
func TestDigitCount(t *testing.T) {

	// initialize test case
	t.Log("DigitCount Test: Expected Value = 10")

	// initialize expected values
	var expValue int = 10
```

```
        // initialize test case variables
        input := 1000000000

        // peform test case
        testCase := DigitCount(input)

        // log test results
        if testCase == expValue {
                t.Log("DigitCount Test: Computed Value =", testCase)
        } else {
                t.Error("DigitCount Test: Computed Value =", testCase)
        }
}
```

<operators.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.package main

package corridor

import (
        "math"
        "math/rand"
        "time"

        "github.com/gonum/matrix/mat64"
)

// fitness function to generate the total fitness and chromosome
// fitness values for a given input chromosome
func ChromosomeFitness(inputChromosome *Chromosome,
    inputObjectives *MultiObjective) (outputChromosome *Chromosome) {

        // get chromosome length
        chromLen := len(inputChromosome.Subs)

        // clear current chromosome fitness values
        inputChromosome.Fitness = make([][]float64, inputObjectives.ObjectiveCount)
        for i := 0; i < inputObjectives.ObjectiveCount; i++ {
                inputChromosome.Fitness[i] = make([]float64, len(inputChromosome.Subs))
        }

        // initialize current & aggregate fitness
        var aggFit float64 = 0.0
        var curFit float64 = 0.0

        // evaluate chromosome length and objectives to compute fitnesses
        for i := 0; i < inputObjectives.ObjectiveCount; i++ {
                for j := 0; j < chromLen; j++ {
                        curFit =
                            inputObjectives.Objectives[i].Matrix.At(inputChromosome.Subs[j][0],
                                inputChromosome.Subs[j][1])
                        inputChromosome.Fitness[i][j] = curFit
                        inputChromosome.TotalFitness[i] = inputChromosome.TotalFitness[i] + curFit
                }

                // compute aggregate fitness
                aggFit = aggFit + inputChromosome.TotalFitness[i]
        }
```

```
        // calculate aggregate fitness
        inputChromosome.AggregateFitness = aggFit

        // return outputs
        return inputChromosome
}

// fitness function generate the mean fitness values for all of the chromosomes
// in a given population
func PopulationFitness(inputPopulation *Population,
    inputParameters *Parameters,
    inputObjectives *MultiObjective) (outputPopulation *Population) {

        // initialize output
        var cumFit float64 = 0.0
        var aggMeanFit float64 = 0.0

        // iterate over the different objectives and drain channel to compute fitness
        for i := 0; i < inputObjectives.ObjectiveCount; i++ {
                for j := 0; j < inputParameters.PopSize; j++ {

                        // read current chromosome from channel
                        curChrom := <-inputPopulation.Chromosomes

                        // compute cumulative fitness
                        cumFit = cumFit + curChrom.TotalFitness[i]

                        // recieve from channel
                        inputPopulation.Chromosomes <- curChrom
                }

                // compute mean from cumulative
                inputPopulation.MeanFitness[i] = cumFit / float64(inputParameters.PopSize)

                // compute aggreage mean fitness
                aggMeanFit = aggMeanFit + inputPopulation.MeanFitness[i]
        }

        // write aggregate mean fitness to output
        inputPopulation.AggregateMeanFitness = aggMeanFit

        // return output
        return inputPopulation
}

// selection operator selects between two chromosomes with a
```

```
// probability of the most fit chromosome being selected
// determined by the input selection probability ratio
func ChromosomeSelection(chrom1, chrom2 *Chromosome,
    selectionProb float64) (selectedChrom *Chromosome) {

        // initialize output
        output := chrom1

        // get current time for random number seed
        rand.Seed(time.Now().UnixNano())

        // generate random number to determine selection result
        dec := rand.Float64()

        // perform conditional selection
        if dec > selectionProb { // normal
                if chrom1.AggregateFitness > chrom2.AggregateFitness {
                        output = chrom1
                } else {
                        output = chrom2
                }
        } else { // inverted
                if chrom1.AggregateFitness > chrom2.AggregateFitness {
                        output = chrom2
                } else {
                        output = chrom1
                }
        }

        // return output
        return output
}

// population selection operator selects half of the input
// population for reproduction based upon comparative
// fitness and some randomized input selection fraction
func PopulationSelection(inputPopulation *Population,
    inputParameters *Parameters) (selection chan *Chromosome) {

        // initialize selection channel size
        selSize := int(math.Floor(float64(cap(inputPopulation.Chromosomes)) * inputParameters.SelFrac))

        // initialize selection channel
        output := make(chan *Chromosome, selSize)

        // initialize selection probability
        selProb := inputParameters.SelProb
```

```
        // initialize selection loop
        for i := 0; i < selSize; i++ {
                chrom1 := <-inputPopulation.Chromosomes
                chrom2 := <-inputPopulation.Chromosomes

                go func(chrom1, chrom2 *Chromosome) {
                        // write selection to output channel
                        output <- ChromosomeSelection(chrom1, chrom2, selProb)
                }(chrom1, chrom2)
        }

        // return selection channel
        return output
}

// intersection determines whether or not the subscripts
// associated with two input chromosomes share any in
// values in common and reports their relative indices
func ChromosomeIntersection(subs1, subs2 [][]int) (subs1Indices, subs2Indices []int) {

        // initialize output index slice
        output1 := make([]int, 0)
        output2 := make([]int, 0)

        // initialize subscript lengths
        len1 := len(subs1)
        len2 := len(subs2)

        // enter intersection loop
        for i := 0; i < len1; i++ {
                for j := 0; j < len2; j++ {
                        if subs1[i][0] == subs2[j][0] && subs1[i][1] == subs2[j][1] {
                                output1 = append(output1, i)
                                output2 = append(output2, j)
                        }
                }
        }

        // return intersection output
        return output1, output2

}

// crossover operator performs the single point crossover
// operation for two input chromosomes that have
// previously been selected from a source population
```

```
func ChromosomeCrossover(chrom1Ind, chrom2Ind []int,
    chrom1Subs, chrom2Subs [][]int) (crossoverChrom [][]int) {

        // initialize maximum length
        maxLen := len(chrom1Subs) + len(chrom2Subs)

        // initialize output
        output := make([][]int, 0, maxLen)

        // get current time for random number seed
        rand.Seed(time.Now().UnixNano())

        var r int

        // generate random number to determine selection result
        // while screening out initial source index match
        for {
                r = rand.Intn(len(chrom1Ind) - 1)
                if r == 0 {
                        continue
                } else {
                        break
                }
        }

        // generate subscript slice 1
        for i := 0; i < (chrom1Ind[r] + 1); i++ {
                output = append(output, chrom1Subs[i])
        }

        // generate subscript slice 2
        for j := (chrom2Ind[r] + 1); j < len(chrom2Subs); j++ {
                output = append(output, chrom2Subs[j])
        }

        // return output
        return output

}

// selection crossover operator performs a single part
// crossover on each of the individuals provided in an
// input selection channel of chromosomes
func SelectionCrossover(inputSelection chan *Chromosome,
    inputParameters *Parameters,
    inputObjectives *MultiObjective,
    inputDomain *Domain) (crossover chan *Chromosome) {
```

```
        // initialize crossover channel
        output := make(chan *Chromosome, inputParameters.PopSize)

        // initialize crossover loop
        for i := 0; i < inputParameters.PopSize; i++ {
                for {
                        // extract chromosomes
                        chrom1 := <-inputSelection
                        chrom2 := <-inputSelection

                        // initialize empty index slices
                        var chrom1Ind []int
                        var chrom2Ind []int

                        // initialize empty chromosome
                        empChrom := NewEmptyChromosome(inputDomain, inputObjectives)

                        // check for valid crossover point
                        chrom1Ind, chrom2Ind = ChromosomeIntersection(chrom1.Subs, chrom2.Subs)

                        // resample chromosomes if no intersection
                        if len(chrom1Ind) > 2 {
                                empChrom.Subs = ChromosomeCrossover(chrom1Ind, chrom2Ind, chrom1.Subs, chrom2.Subs)
                                empChrom = ChromosomeFitness(empChrom, inputObjectives)
                                output <- empChrom
                                inputSelection <- chrom1
                                inputSelection <- chrom2
                                break
                        } else {
                                inputSelection <- chrom2
                                inputSelection <- chrom1
                                continue
                        }
                }
        }

        // return output
        return output
}

// mutationLocus to randomly select a mutation locus and return the adjacent
// loci along the length of the chromosome
func MutationLoci(inputChromosome *Chromosome) (previousLocus,
    mutationLocus, nextLocus []int,
    mutationIndex int) {
```

```
        // compute chromosome length
        lenChrom := len(inputChromosome.Subs)

        // seed random number generator
        rand.Seed(time.Now().UnixNano())

        // randomly select mutation index
        mutIndex := rand.Intn(lenChrom-4) + 2

        // get mutation locui subscripts from mutIndex
        mutLocus := inputChromosome.Subs[mutIndex]
        prvLocus := inputChromosome.Subs[mutIndex-1]
        nxtLocus := inputChromosome.Subs[mutIndex+1]

        // return output
        return prvLocus, mutLocus, nxtLocus, mutIndex
}

// mutation sub domain returns the subdomain to be used for the mutation
// specific directed walk procedure
func MutationSubDomain(previousLocus,
    mutationLocus, nextLocus []int,
    inputDomain *mat64.Dense) (outputSubDomain *mat64.Dense) {

        // generate mutation locus neighborhood indices
        nInd := NeighborhoodSubs(mutationLocus)

        // initialize iterator
        var n int = 0

        // initialize sub domain matrix
        subMat := mat64.NewDense(5, 5, nil)

        // clean sub domain
        for i := 0; i < 5; i++ {
                for j := 0; j < 5; j++ {
                        if i == 0 {
                                subMat.Set(i, j, 0.0)
                        } else if i == 4 {
                                subMat.Set(i, j, 0.0)
                        } else if j == 0 {
                                subMat.Set(i, j, 0.0)
                        } else if j == 4 {
                                subMat.Set(i, j, 0.0)
                        } else if nInd[n][0] == previousLocus[0] && nInd[n][1] == previousLocus[1] {
                                subMat.Set(i, j, 1.0)
                                // iterate counter
```

```
                        n += 1
                } else if nInd[n][0] == nextLocus[0] && nInd[n][1] == nextLocus[1] {
                        subMat.Set(i, j, 1.0)
                        // iterate counter
                        n += 1
                } else {
                        subMat.Set(i, j, inputDomain.At(nInd[n][0], nInd[n][1]))
                        // iterate counter
                        n += 1
                }
        }
    }

    // return output
    return subMat
}

// function to generate a mutation within a given chromosome at a specified
// number of mutation loci
func ChromosomeMutation(inputChromosome *Chromosome,
    inputDomain *Domain,
    inputParameters *Parameters,
    inputObjectives *MultiObjective) (outputChromosome *Chromosome) {

    // compute chromosome length
    lenChrom := len(inputChromosome.Subs)

    // initialize output chromosome
    output := inputChromosome

    // initialize reference domain matrix
    refDomain := mat64.NewDense(inputDomain.Rows, inputDomain.Cols, nil)

    // clone input domain matrix
    refDomain.Clone(inputDomain.Matrix)

    // block out cells on current chromosome
    for k := 0; k < lenChrom; k++ {
            refDomain.Set(inputChromosome.Subs[k][0], inputChromosome.Subs[k][1], 0.0)
    }

    // enter unbounded mutation search loop
    for {
            // generate mutation loci
            prvLocus, mutLocus, nxtLocus, mutIndex := MutationLoci(inputChromosome)

            // first check if deletion is valid, else perform mutation
```

```
if Distance(prvLocus, nxtLocus) < 1.5 {

        // perform simple deletion of mutation index
        output.Subs = append(output.Subs[:mutIndex], output.Subs[(mutIndex+1):]...)

        // loop over objective and remove fitness values
        for r := 0; r < inputObjectives.ObjectiveCount; r++ {
                output.Fitness[r] =
                    append(output.Fitness[r][:mutIndex], output.Fitness[r][(mutIndex+1):]...)
        }
        break
} else {

        // generate mutation subdomain
        subMat := MutationSubDomain(prvLocus, mutLocus, nxtLocus, refDomain)

        // generate sub source and sub destination
        subSource := make([]int, 2)
        subDestin := make([]int, 2)
        subSource[0] = prvLocus[0] - mutLocus[0] + 2
        subSource[1] = prvLocus[1] - mutLocus[1] + 2
        subDestin[0] = nxtLocus[0] - mutLocus[0] + 2
        subDestin[1] = nxtLocus[1] - mutLocus[1] + 2

        // generate subdomain from sub matrix and generate sub basis
        subDomain := NewDomain(subMat)
        subParams := NewParameters(subSource, subDestin, 1, 1, inputParameters.RndCoef)
        subBasis := NewBasis(subSource, subDestin, subDomain)

        // check validity of sub domain
        subDomainTest := ValidateSubDomain(subSource, subDestin, subMat)

        // resample if subdomain is invalid
        if subDomainTest == false {
                continue
        } else {

                // generate directed walk based mutation
                subWlk, tabuTest :=
                    MutationWalk(subParams.SrcSubs, subParams.DstSubs, subDomain, subParams, subBasis)

                // if tabu test fails abort mutation and restart
                if tabuTest == false {
                        subWlk = make([][]int, 1)
                        continue
                } else {
```

```
                                        subLen := len(subWlk)
                                        subFit := make([][]float64, inputObjectives.ObjectiveCount)

                                        // translate subscripts and evaluate fitnesses
                                        for i := 0; i < inputObjectives.ObjectiveCount; i++ {

                                                // initialize subfit section
                                                subFit[i] = make([]float64, subLen)

                                                // translate subscripts and compute sub walk fitness
                                                for j := 0; j < subLen; j++ {
                                                        if i == 0 {
                                                                subWlk[j][0] = subWlk[j][0] - 2 + mutLocus[0]
                                                                subWlk[j][1] = subWlk[j][1] - 2 + mutLocus[1]
                                                        }

                                                        subFit[i][j] =
                                                            inputObjectives.Objectives[i].Matrix.At(subWlk[j][0],
                                                                subWlk[j][1])
                                                }

                                                // delete mutation locus from fitnesses
                                                output.Fitness[i] = append(output.Fitness[i][:mutIndex],
                                                    output.Fitness[i][(mutIndex+1):]...)

                                                // insert sub walk section into original chromosome fitnesses
                                                output.Fitness[i] = append(output.Fitness[i][:mutIndex-1],
                                                    append(subFit[i], output.Fitness[i][mutIndex+1:]...)...)
                                        }

                                        // delete mutation locus from subs
                                        output.Subs = append(output.Subs[:mutIndex], output.Subs[(mutIndex+1):]...)

                                        // insert new sub walk subscripts into subs
                                        output.Subs =
                                            append(output.Subs[:mutIndex-1],
                                                append(subWlk, output.Subs[mutIndex+1:]...)...)
                                        break
                                }
                        }
                }

        }

        // return output
        return output
}
```

```
// function to generate multiple mutations on multiple separate loci on the same
// input chromosome
func ChromosomeMultiMutation(inputChromosome *Chromosome,
    inputDomain *Domain,
    inputParameters *Parameters,
    inputObjectives *MultiObjective) (outputChromosome *Chromosome) {

        // loop through mutation count
        for i := 0; i < inputParameters.MutaCnt; i++ {
                inputChromosome = ChromosomeMutation(inputChromosome, inputDomain, inputParameters, inputObjectives)
        }

        // return output
        return inputChromosome
}

// function to generate mutations within a specified fraction of an input
// population with those chromosomes being selected at random
func PopulationMutation(inputChromosomes chan *Chromosome,
    inputParameters *Parameters,
    inputObjectives *MultiObjective,
    inputDomain *Domain) (outputChromosomes chan *Chromosome) {

        // calculate the total number of chromosomes that are to receive mutations
        mutations := int(math.Floor(float64(inputParameters.PopSize) * float64(inputParameters.MutaFrc)))

        // seed random number generator
        rand.Seed(time.Now().UnixNano())

        // initialize mutation selection test variable and iteration counter variable
        var iter int
        var mutTest int

        // initialize throttle size
        conc := make(chan bool, inputParameters.ConSize)

        // initialize selection loop
        for j := 0; j < inputParameters.PopSize; j++ {

                // get current chromosome from channel
                curChrom := <-inputChromosomes

                // generate random mutation selection binary integer
                mutTest = rand.Intn(2)

                // screen on mutation indices
```

```
                if mutTest == 1 {

                        // write to control channel
                        conc <- true

                        // launch go routines
                        go func(curChrom *Chromosome,
                            inputDomain *Domain,
                            inputParameters *Parameters,
                            inputObjectives *MultiObjective) {
                                defer func() { <-conc }()
                                curChrom = ChromosomeMultiMutation(curChrom, inputDomain, inputParameters, inputObjectives)
                        }(curChrom, inputDomain, inputParameters, inputObjectives)

                        // update iterator
                        iter += 1

                        // return current chromosome back to channel
                        inputChromosomes <- curChrom

                } else {

                        // return current chromosome back to channel
                        inputChromosomes <- curChrom
                }

                // break once the desired number of mutants has been generated
                if iter == mutations {
                        break
                }
        }

        // cap parallelism at concurrency limit
        for j := 0; j < cap(conc); j++ {
                conc <- true
        }

        // return selection channel
        return inputChromosomes
}

// population evolution operator generates a new population
// from an input population using the selection and crossover operators
func PopulationEvolution(inputPopulation *Population,
    inputDomain *Domain,
    inputParameters *Parameters,
    inputObjectives *MultiObjective) (outputPopulation *Population) {
```

```
        // initialize new empty population
        output := NewEmptyPopulation(inputPopulation.Id+1, inputObjectives)

        // perform population selection
        popSel := PopulationSelection(inputPopulation, inputParameters)

        // perform selection crossover
        selCrs := SelectionCrossover(popSel, inputParameters, inputObjectives, inputDomain)

        // fill empty population
        popMut := PopulationMutation(selCrs, inputParameters, inputObjectives, inputDomain)

        // assign channel to output population
        output.Chromosomes = popMut

        // return output
        return output
}
```

<random.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package corridor

import (
        "math"
        "math/rand"
        "time"

        "github.com/gonum/matrix/mat64"
)

// multivariatenormalrandom generates pairs of bivariate normally distributed
// random numbers given an input mean vector and covariance matrix
func MultiVariateNormalRandom(mu *mat64.Dense, sigma *mat64.SymDense) (rndsmp *mat64.Dense) {

        // initialize vector slices
        o := make([]float64, 2)
        n := make([]float64, 2)

        // generate random numbers from normal distribution, prohibit [0,0]
        // combinations
        rand.Seed(time.Now().UnixNano())

        // enter loop
        for i := 0; i < 2; i++ {
                n[i] = rand.NormFloat64()
        }

        // convert to matrix type
        rnd := mat64.NewDense(2, 1, n)
        output := mat64.NewDense(2, 1, o)

        // perform cholesky decomposition on covariance matrix
        lower := mat64.NewTriDense(2, false, nil)
        lower.Cholesky(sigma, true)

        // compute output
        output.Mul(lower, rnd)
        output.Add(output, mu)

        //return final output
        return output
```

```
}

// fixmultivariatenormalrandom converts an input vector of bivariate normally
// distributed random numbers into a version where the values have been fixed
// to a [-1, 0 ,1] range
func FixMultiVariateNormalRandom(rndsmp *mat64.Dense) (fixsmp *mat64.Dense) {

        // initialize vector slice
        o := make([]float64, 2)

        // write up down movement direction
        if rndsmp.At(0, 0) > 1.0 {
                o[0] = 1
        } else if rndsmp.At(0, 0) >= -1.0 && rndsmp.At(0, 0) <= 1.0 {
                o[0] = 0
        } else if rndsmp.At(0, 0) < -1.0 {
                o[0] = -1
        }

        // write left right movement direction
        if rndsmp.At(1, 0) > 1.0 {
                o[1] = 1
        } else if rndsmp.At(1, 0) >= -1.0 && rndsmp.At(1, 0) <= 1.0 {
                o[1] = 0
        } else if rndsmp.At(1, 0) < -1.0 {
                o[1] = -1
        }

        // convert to matrix type
        output := mat64.NewDense(1, 2, o)

        // return final output
        return output
}

// newrandom repeatedly generates a new random sample from mvrnd and then fixes
// it using fixrandom until the sample is comprised of a non [0, 0] case
func NewRandom(mu *mat64.Dense, sigma *mat64.SymDense) (newRand []int) {

        // initialize rndsmp and fixsmp and output variables
        rndsmp := mat64.NewDense(2, 1, nil)
        fixsmp := mat64.NewDense(1, 2, nil)

        // generate random vectors prohibiting zero-zero cases
        for {
                rndsmp = MultiVariateNormalRandom(mu, sigma)
                fixsmp = FixMultiVariateNormalRandom(rndsmp)
```

```
                if fixsmp.At(0, 0) == 0 && fixsmp.At(0, 1) == 0 {
                        continue
                } else {
                        break
                }
        }

        // initialize output
        output := make([]int, 2)

        // write output values
        output[0] = int(fixsmp.At(0, 0))
        output[1] = int(fixsmp.At(0, 1))

        // return final output
        return output
}

// newmu generates a matrix representation of mu that reflects the
// spatial orentiation between the input current subscript and the
// destination subscript
func NewMu(curSubs, dstSubs []int) (mu *mat64.Dense) {

        // compute mu as the orientation vector
        orientVec := Orientation(curSubs, dstSubs)

        // convert mu to float
        var muVec = []float64{float64(orientVec[0]), float64(orientVec[1])}

        // initialize matrix output
        output := mat64.NewDense(2, 1, muVec)

        // return final output
        return output
}

// newsigma generates a matrix representation of sigma that reflects the
// number of iterations in the sampling process as well as the distance
// from the basis euclidean solution
func NewSigma(iterations int, randomness, distance float64) (sigma *mat64.SymDense) {

        // impose lower bound on distance
        if distance < 1 {
                distance = 1.0
        }

        // set numerator
```

```
        var num float64 = 1.0

        // initialize covariance
        var cov float64

        // compute covariance
        if distance == 1.0 {
                cov = 1.0
        } else {
                cov = math.Pow(distance, (num/randomness)) / math.Pow(float64(iterations), (num/randomness))
        }

        // initialize matrix output
        output := mat64.NewSymDense(2, nil)

        // set values
        output.SetSym(0, 0, cov)
        output.SetSym(0, 1, 0.0)
        output.SetSym(1, 0, 0.0)
        output.SetSym(1, 1, cov)

        // return final output
        return output
}

// newsubs generates a feasible new subscript value set within the
// input search domain
func NewSubs(curSubs, destinationSubs []int,
    curDist float64,
    searchParameters *Parameters,
    searchDomain *Domain) (subs []int) {

        // initialize iteration counter
        var iterations int = 1

        // initialize output
        output := make([]int, 2)

        // generate and fix a bivariate normally distributed random vector
        // prohibit all zero cases and validate using the search domain
        for {

                // generate mu and sigma values
                mu := NewMu(curSubs, destinationSubs)
                sigma := NewSigma(iterations, searchParameters.RndCoef, curDist)

                // generate fixed random bivariate normally distributed numbers
```

```
                try := NewRandom(mu, sigma)

                // write output
                output[0] = curSubs[0] + try[0]
                output[1] = curSubs[1] + try[1]

                // DEBUG
                // test if currentIndex is forbidden
                if searchDomain.Matrix.At(output[0], output[1]) == 0.0 {
                        iterations += 1
                        continue
                }

                // test if currentIndex inside search domain
                if output[0] > searchDomain.Rows-1 ||
                    output[1] > searchDomain.Cols-1 ||
                    output[0] < 0 || output[1] < 0 {
                            iterations += 1
                            continue
                } else {
                        break
                }
        }

        // return final output
        return output
}

// directedwalk generates a new directed walk connecting a source subscript to a
// destination subscript within the context of an input search domain
func DirectedWalk(sourceSubs, destinationSubs []int,
    searchDomain *Domain,
    searchParameters *Parameters,
    basisSolution *Basis) (subs [][]int) {

        // initialize chromosomal 2D slice with source subscript as first element
        output := make([][]int, 1, basisSolution.MaxLen)
        output[0] = make([]int, 2)
        output[0][0] = sourceSubs[0]
        output[0][1] = sourceSubs[1]

        // enter unbounded for loop
        for {

                // initialize new tabu matrix
                tabu := mat64.NewDense(searchDomain.Rows, searchDomain.Cols, nil)
                for i := 0; i < searchDomain.Rows; i++ {
```

```
        for j := 0; j < searchDomain.Cols; j++ {

                if i == 0 || i == searchDomain.Rows-1 || j == 0 || j == searchDomain.Cols-1 {
                        tabu.Set(i, j, 0.0)
                } else {
                        tabu.Set(i, j, 1.0)
                }
        }
}

//tabu.Clone(searchDomain.Matrix)
tabu.Set(sourceSubs[0], sourceSubs[1], 0.0)

// initialize current subscripts, distance, try, and iteration counter
curSubs := make([]int, 2)
var curDist float64
var try []int

// enter bounded for loop
for i := 0; i < basisSolution.MaxLen; i++ {

        // get current subscripts
        curSubs = output[len(output)-1]

        // validate tabu neighborhood
        if ValidateTabu(curSubs, tabu) == false {
                break
        }

        // compute current distance
        curDist = basisSolution.Matrix.At(curSubs[0], curSubs[1])

        // generate new try
        try = NewSubs(curSubs, destinationSubs, curDist, searchParameters, searchDomain)

        // apply control conditions
        if try[0] == destinationSubs[0] && try[1] == destinationSubs[1] {
                output = append(output, try)
                break
        } else if tabu.At(try[0], try[1]) == 0.0 {
                continue
        } else {
                output = append(output, try)
                tabu.Set(try[0], try[1], 0.0)
        }
}
```

```
            // repeat walk if destination not reached
            if output[len(output)-1][0] == destinationSubs[0] &&
                output[len(output)-1][1] == destinationSubs[1] {

                    // break unbounded for loop
                    break
            } else {

                    // re-initialize chromosomal 2D slice with source subscript as first element
                    output := make([][]int, 1, basisSolution.MaxLen)
                    output[0] = make([]int, 2)
                    output[0][0] = sourceSubs[0]
                    output[0][1] = sourceSubs[1]

                    // restart process
                    continue
            }
        }

        // return final output
        return output
}

// mutationwalk generates a new directed walk connecting a source subscript to a
// destination subscript within the context of an input mutation search domain
func MutationWalk(sourceSubs, destinationSubs []int,
    searchDomain *Domain,
    searchParameters *Parameters,
    basisSolution *Basis) (subs [][]int, tabuTest bool) {

        // initialize chromosomal 2D slice with source subscript as first
        // element
        output := make([][]int, 1, basisSolution.MaxLen)
        output[0] = make([]int, 2)
        output[0][0] = sourceSubs[0]
        output[0][1] = sourceSubs[1]

        // initialize new tabu matrix
        tabu := mat64.NewDense(searchDomain.Rows, searchDomain.Cols, nil)
        tabu.Clone(searchDomain.Matrix)
        tabu.Set(sourceSubs[0], sourceSubs[1], 0.0)

        // initialize current subscripts, distance, try, and iteration counter
        curSubs := make([]int, 2)
        var curDist float64
        var try []int
        var test bool
```

```
        // enter un-bounded for loop
        for {

                // get current subscripts
                curSubs = output[len(output)-1]

                // compute current distance
                curDist = basisSolution.Matrix.At(curSubs[0], curSubs[1])

                // generate new try
                try = NewSubs(curSubs, searchParameters.DstSubs, curDist, searchParameters, searchDomain)

                // apply control conditions
                if try[0] == destinationSubs[0] && try[1] == destinationSubs[1] {
                        output = append(output, try)
                        break
                } else if tabu.At(try[0], try[1]) == 0.0 {
                        continue
                } else {
                        output = append(output, try)
                        tabu.Set(try[0], try[1], 0.0)
                }

                // validate tabu matrix
                test = ValidateSubDomain(try, destinationSubs, tabu)

                // reset if tabu is invalid
                if test == false {
                        break
                }
        }

        // return final output
        return output, test
}

// newnodesubs generates an poutput slice of new intermediate destination nodes
// that are progressively further, in terms of euclidean distance, from
// a given input source location and are orientation towards a given
// destination location
func NewNodeSubs(searchDomain *Domain, searchParameters *Parameters) (nodeSubs [][]int) {

        // initialize output
        output := make([][]int, 1)
        output[0] = searchParameters.SrcSubs
```

```
// check band count against input distance matrix size
if searchDomain.BndCnt < 3 {

        // asign node subscripts
        output = append(output, searchParameters.DstSubs)
} else if searchDomain.BndCnt >= 3 {

        // generate distance matrix from source subscripts
        distMat := AllDistance(searchParameters.SrcSubs, searchDomain.Matrix)

        // encode distance bands
        bandMat := DistanceBands(searchDomain.BndCnt, distMat)

        if bandMat.At(searchParameters.SrcSubs[0], searchParameters.SrcSubs[1]) ==
            bandMat.At(searchParameters.DstSubs[0], searchParameters.DstSubs[1]) {

                // asign node subscripts
                output = append(output, searchParameters.DstSubs)
        } else {

                // seed random number generator
                rand.Seed(time.Now().UnixNano())

                // loop through band vector and generate band value subscripts
                for i := 1; i < searchDomain.BndCnt-1; i++ {

                        // generate band mask
                        bandMaskMat := BandMask(float64(i), bandMat)

                        // break loop if the destination is in the current band mask
                        if bandMaskMat.At(searchParameters.DstSubs[0], searchParameters.DstSubs[1]) == 1.0 {
                                break
                        }

                        // generate orientation mask
                        orientMaskMat :=
                            OrientationMask(output[i-1], searchParameters.DstSubs, searchDomain.Matrix)

                        // initialize final mask
                        finalMaskMat := mat64.NewDense(searchDomain.Rows, searchDomain.Cols, nil)

                        // compute final mask through elementwise multiplication
                        finalMaskMat.MulElem(bandMaskMat, orientMaskMat)

                        // generate subs from final mask
                        finalSubs := NonZeroSubs(finalMaskMat)
```

```
                        // generate random number of length interval
                        randInd := finalSubs[rand.Intn(len(finalSubs))]

                        // break out of loop if final mask is empty
                        if randInd[0] == 0 && randInd[1] == 0 {
                                break
                        }

                        // extract randomly selected value and write to output
                        output = append(output, randInd)
                }

                // set the final subscript to the destination
                output = append(output, searchParameters.DstSubs)
        }
    }

    // return output
    return output
}

// multipartdirectedwalk generates a new multipart directed walk from a given set
// of input problem parameters
func MultiPartDirectedWalk(nodeSubs [][]int,
    searchDomain *Domain,
    searchParameters *Parameters) (subs [][]int) {

    // generate basis solution
    basisSolution := NewBasis(nodeSubs[0], nodeSubs[1], searchDomain)

    // initialize output
    output := make([][]int, basisSolution.MaxLen)

    // catch single part walk case
    if len(nodeSubs) == 2 {

            // generate output as a single part directed walk
            output =
                DirectedWalk(nodeSubs[0], nodeSubs[1], searchDomain, searchParameters, basisSolution)

    } else if len(nodeSubs) > 2 {

            // generate output as multi part directed walk
            output =
                DirectedWalk(nodeSubs[0], nodeSubs[1], searchDomain, searchParameters, basisSolution)

            // loop through the band count to generate sub walk parts
```

```
		for i := 1; i < len(nodeSubs)-1; i++ {

			// generate basis solution
			basisSolution = NewBasis(nodeSubs[i], nodeSubs[i+1], searchDomain)

			// generate initial output slice and then append subsequent slices
			curWalk :=
			    DirectedWalk(nodeSubs[i], nodeSubs[i+1], searchDomain, searchParameters, basisSolution)

			/*
				The debug section below is attempting to deal with possible cases
				where two parts of two different path sections overlap in the final
				multipart pathway. Attempts to deal with this by iteratively precluding
				path sections from the search domain have lead to infinite loop conditions.
				More work is needed to resolve this issue.
			*/

			// DEBUG
			//// mask walk section from search domain
			//for j := 0; j < len(curWalk); j++ {
			//	searchDomain.Matrix.Set(curWalk[j][0], curWalk[j][1], 0.0)
			//}

			// append subscripts to output
			for j := 1; j < len(curWalk); j++ {
				output = append(output, curWalk[j])
			}
		}
	}

	// return output
	return output
}
```

<random_test.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package corridor

import (
        "math"
        "testing"

        "github.com/gonum/matrix/mat64"
)

// test multivariaterandomnormal
func TestMultiVariateRandomNormal(t *testing.T) {

        // initialize test case
        t.Log("MultiVariateRandomNormal Test: Expected Value = [(1 +- 0.1) (1 +- 0.1)]")

        // initialize expected values
        var muVec = []float64{1, 1}
        mu := mat64.NewDense(2, 1, muVec)
        var sigmaVec = []float64{1, 0, 0, 1}
        sigma := mat64.NewSymDense(2, sigmaVec)

        // initialize test case variables
        testMat := mat64.NewDense(10000, 2, nil)
        testCase := make([]float64, 2)
        var testBool bool
        var aSum float64 = 0.0
        var bSum float64 = 0.0

        // perform test case
        for i := 0; i < 10000; i++ {
                curVal := MultiVariateNormalRandom(mu, sigma)
                testMat.Set(i, 0, curVal.At(0, 0))
                testMat.Set(i, 1, curVal.At(1, 0))
        }

        // compute test result
        for i := 0; i < 10000; i++ {
                aSum += testMat.At(i, 0)
                bSum += testMat.At(i, 1)
        }
        testCase[0] = aSum / 10000.0
```

```
        testCase[1] = bSum / 10000.0
        testBool = (math.Abs(1-testCase[0]) < 0.1) && (math.Abs(1-testCase[1]) < 0.1)

        // log test results
        if testBool {
                t.Log("MultiVariateRandomNormal Test: Computed Mean =", testCase)
        } else {
                t.Error("MultiVariateRandomNormal Test: Computed Mean =", testCase)
        }
}

// test fixmultivariatenormalrandom
func TestFixMultiVariateNormalRandom(t *testing.T) {

        // initialize test case
        t.Log("FixMultiVariateNormalRandom Test: Expected Value = [(0.47 +- 0.1) (0.47 +- 0.1)]")

        // initialize expected values
        var muVec = []float64{1, 1}
        mu := mat64.NewDense(2, 1, muVec)
        var sigmaVec = []float64{1, 0, 0, 1}
        sigma := mat64.NewSymDense(2, sigmaVec)

        // initialize test case variables
        testMat := mat64.NewDense(10000, 2, nil)
        testCase := make([]float64, 2)
        var testBool bool
        var aSum float64 = 0.0
        var bSum float64 = 0.0

        // generate fixed random samples
        for i := 0; i < 10000; i++ {
                curRnd := MultiVariateNormalRandom(mu, sigma)
                curFix := FixMultiVariateNormalRandom(curRnd)
                testMat.Set(i, 0, curFix.At(0, 0))
                testMat.Set(i, 1, curFix.At(0, 1))
        }

        // compute test result
        for i := 0; i < 10000; i++ {
                aSum += testMat.At(i, 0)
                bSum += testMat.At(i, 1)
        }
        testCase[0] = aSum / 10000.0
        testCase[1] = bSum / 10000.0
        testBool = (math.Abs(0.47-testCase[0]) < 0.1) && (math.Abs(0.47-testCase[1]) < 0.1)
```

```
	// log test results
	if testBool {
		t.Log("FixMultiVariateNormalRandom Test: Computed Mean =", testCase)
	} else {
		t.Error("FixMultiVariateNormalRandom Test: Computed Value =", testCase)
	}
}

// test newrandom
func TestNewRandom(t *testing.T) {

	// initialize test case
	t.Log("NewRandom Test: Expected Value = [(0.615 +- 0.1) (0.615 +- 0.1)]")

	// initialize expected values
	var muVec = []float64{1, 1}
	mu := mat64.NewDense(2, 1, muVec)
	var sigmaVec = []float64{1, 0, 0, 1}
	sigma := mat64.NewSymDense(2, sigmaVec)

	// initialize test case variables
	testMat := mat64.NewDense(10000, 2, nil)
	testCase := make([]float64, 2)
	var testBool bool
	var aSum float64 = 0.0
	var bSum float64 = 0.0

	// generate random samples
	for i := 0; i < 10000; i++ {
		curVal := NewRandom(mu, sigma)
		testMat.Set(i, 0, float64(curVal[0]))
		testMat.Set(i, 1, float64(curVal[1]))
	}

	// compute test result
	for i := 0; i < 10000; i++ {
		aSum += testMat.At(i, 0)
		bSum += testMat.At(i, 1)
	}
	testCase[0] = aSum / 10000.0
	testCase[1] = bSum / 10000.0
	testBool = (math.Abs(0.615 - testCase[0]) < 0.1) && (math.Abs(0.615 - testCase[1]) < 0.1)

	// log test results
	if testBool {
		t.Log("NewRandom Test: Computed Value =", testCase)
	} else {
```

```
                t.Error("NewRandom Test: Computed Value =", testCase)
        }
}

// test newmu
func TestNewMu(t *testing.T) {

        // initialize test case
        t.Log("NewMu Test: Expected Value = {{2 1 1 [1 1]} 2 1}")

        // initialize expected values
        var expVal = []float64{1.0, 1.0}

        // initialize test case variables
        var curSubs = []int{10, 10}
        var dstSubs = []int{100, 100}

        // perform test case
        testCase := NewMu(curSubs, dstSubs)

        // log test results
        if testCase.At(0, 0) == expVal[0] && testCase.At(1, 0) == expVal[1] {
                t.Log("NewMu Test: Computed Value =", *testCase)
        } else {
                t.Error("NewMu Test: Computed Value =", *testCase)
        }
}

// test newsigma
func TestNewSigma(t *testing.T) {

        // initialize test case
        t.Log("NewSigma Test: Expected Matrix = {{2 1 [10 0 0 10]} 121}")

        // initialize expected values
        var expValVec = []float64{10.0, 0.0, 0.0, 10.0}
        expValMat := mat64.NewDense(2, 2, expValVec)

        // initialize test case variables
        var iterations int = 1
        var randomness float64 = 1.0
        var distance float64 = 10.0

        // perform test case
        testCase := NewSigma(iterations, randomness, distance)

        // log test results
```

```
        if testCase.At(0, 0) == expValMat.At(0, 0) &&
                testCase.At(0, 1) == expValMat.At(0, 1) &&
                testCase.At(1, 0) == expValMat.At(1, 0) &&
                testCase.At(1, 1) == expValMat.At(1, 1) {
                t.Log("NewSigma Test: Computed Matrix =", *testCase)
        } else {
                t.Error("NewSigma Test: Compute Matrix =", *testCase)
        }
}

// test newsubs
func TestNewSubs(t *testing.T) {

        // initialize test case
        t.Log("NewSubs Test: Expected Value = [1 2] or [2 1]")

        // initialize expected values
        var expVal1 = []int{1, 2}
        var expVal2 = []int{2, 1}

        // initialize test case variables
        var curSubs = []int{1, 1}
        var dstSubs = []int{3, 3}
        var curDist float64 = 0.0
        testParams := NewParameters(curSubs, dstSubs, 10, 10, 1.0)
        var domainVec = []float64{
                0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 1.0, 0.0, 1.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0}
        domainMat := mat64.NewDense(5, 5, domainVec)
        testDomain := NewDomain(domainMat)

        // perform test case
        testCase := NewSubs(curSubs, dstSubs, curDist, testParams, testDomain)

        // log test results
        if (testCase[0] == expVal1[0] && testCase[1] == expVal1[1]) ||
                (testCase[0] == expVal2[0] && testCase[1] == expVal2[1]) {
                t.Log("NewSubs Test: Computed Value =", testCase)
        } else {
                t.Error("NewSubs Test: Computed Value =", testCase)
        }
}

// test directedwalk
```

```
func TestDirectedWalk(t *testing.T) {

	// initialize test case
	t.Log("DirectedWalk Test: Expected Value = [[1 1] [1 2] [2 3] [3 3]]")

	// initialize expected values
	expVal := make([][]int, 5)
	expVal[0] = []int{1, 1}
	expVal[1] = []int{1, 2}
	expVal[2] = []int{2, 3}
	expVal[3] = []int{3, 3}

	// initialize test case variables
	var sourceSubs = []int{1, 1}
	var destinationSubs = []int{3, 3}
	testParams := NewParameters(sourceSubs, destinationSubs, 10, 10, 1.0)
	var domainVec = []float64{
		0.0, 0.0, 0.0, 0.0, 0.0,
		0.0, 1.0, 1.0, 0.0, 0.0,
		0.0, 0.0, 0.0, 1.0, 0.0,
		0.0, 0.0, 0.0, 1.0, 0.0,
		0.0, 0.0, 0.0, 0.0, 0.0}
	domainMat := mat64.NewDense(5, 5, domainVec)
	testDomain := NewDomain(domainMat)
	testBasis := NewBasis(sourceSubs, destinationSubs, testDomain)
	var testBool bool

	// perform test case
	testCase := DirectedWalk(sourceSubs, destinationSubs, testDomain, testParams, testBasis)

	// evaluate test results
	for i := 0; i < 4; i++ {
		if testCase[i][0] == expVal[i][0] && testCase[i][1] == expVal[i][1] {
			testBool = true
		} else {
			testBool = false
			break
		}
	}

	// log test results
	if testBool {
		t.Log("DirectedWalk Test: Computed Value =", testCase)
	} else {
		t.Error("DirectedWalk Test: Computed Value =", testCase)
	}
}
```

```
// test mutationwalk
func TestMutationWalk(t *testing.T) {

	// initialize test case
	t.Log("MutationWalk Test: Expected Value = [[1 1] [2 1] [3 2] [3 3]] or [[1 1] [1 2] [2 3] [3 3]]")

	// initialize test case variables
	var sourceSubs = []int{1, 1}
	var destinationSubs = []int{3, 3}
	testParams := NewParameters(sourceSubs, destinationSubs, 10, 10, 1.0)
	var domainVec = []float64{
		0.0, 0.0, 0.0, 0.0, 0.0,
		0.0, 1.0, 1.0, 1.1, 0.0,
		0.0, 1.0, 0.0, 1.0, 0.0,
		0.0, 1.0, 1.0, 1.0, 0.0,
		0.0, 0.0, 0.0, 0.0, 0.0}
	domainMat := mat64.NewDense(5, 5, domainVec)
	testDomain := NewDomain(domainMat)
	testBasis := NewBasis(sourceSubs, destinationSubs, testDomain)
	var testBool bool
	var testCase [][]int

	// perform test case
	for {
		testCase, testBool = MutationWalk(sourceSubs,
		    destinationSubs, testDomain, testParams, testBasis)
		if testBool == true {
			break
		} else {
			continue
		}
	}

	// log test results
	if testBool {
		t.Log("MutationWalk Test: Computed Value =", testCase)
	} else {
		t.Error("MutationWalk Test: Computed Value =", testCase)
	}
}

// test newnodesubs
func TestNewNodeSubs(t *testing.T) {

	// initialize test case
	t.Log("NewNodeSubs Test: Expected Value = [[1 1] [2 2] [3 3]]")
```

```
        // initialize expected values
        expVal := make([][]int, 3)
        expVal[0] = []int{1, 1}
        expVal[1] = []int{2, 2}
        expVal[2] = []int{3, 3}

        // initialize test case variables
        var sourceSubs = []int{1, 1}
        var destinationSubs = []int{3, 3}
        testParams := NewParameters(sourceSubs, destinationSubs, 10, 10, 1.0)
        var domainVec = []float64{
                0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 1.0, 1.1, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0}
        domainMat := mat64.NewDense(5, 5, domainVec)
        testDomain := NewDomain(domainMat)
        testDomain.BndCnt = 3

        // perform test case
        testCase := NewNodeSubs(testDomain, testParams)
        testBool := (testCase[0][0] == expVal[0][0] &&
                testCase[0][1] == expVal[0][1] &&
                testCase[1][0] == expVal[1][0] &&
                testCase[1][1] == expVal[1][1] &&
                testCase[2][0] == expVal[2][0] &&
                testCase[2][1] == expVal[2][1])

        // log test results
        if testBool {
                t.Log("NewNodeSubs Test: Computed Value =", testCase)
        } else {
                t.Error("NewNodeSubs Test: Computed Value =", testCase)
        }
}

// test multipartdirectedwalk
func TestMultiPartDirectedWalk(t *testing.T) {

        // initialize test case
        t.Log("MultiPartDirectedWalk: Expected Value = [[1 1]...[3 3]]")

        // initialize expected value

        // initialize test case variables
```

```
        var sourceSubs = []int{1, 1}
        var destinationSubs = []int{3, 3}
        testParams := NewParameters(sourceSubs, destinationSubs, 10, 10, 1.0)
        var domainVec = []float64{
                0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 0.0, 0.1, 0.0,
                0.0, 1.0, 1.0, 0.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0}
        domainMat := mat64.NewDense(5, 5, domainVec)
        testDomain := NewDomain(domainMat)
        testDomain.BndCnt = 3
        nodeSubs := make([][]int, 3)
        nodeSubs[0] = []int{1, 1}
        nodeSubs[1] = []int{3, 1}
        nodeSubs[2] = []int{3, 3}

        // perform test case
        testCase := MultiPartDirectedWalk(nodeSubs, testDomain, testParams)

        // evaluate test case
        t.Log("MultiPartDirectedWalk: Computed Value =", testCase)
}
```

<samples.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package corridor

import (
        "math"
        "math/rand"
        "runtime"
        "time"

        "github.com/gonum/matrix/mat64"
)

// new sample parameters initialization function
func NewSampleParameters(searchDomain *Domain) *Parameters {

        // initialize variables
        sourceSubscripts := make([]int, 2)
        sourceSubscripts[0] = 3
        sourceSubscripts[1] = 3
        destinationSubscripts := make([]int, 2)
        destinationSubscripts[0] = searchDomain.Rows - 3
        destinationSubscripts[1] = searchDomain.Cols - 3
        randomnessCoefficient := 1.0
        populationSize := 1000
        selectionFraction := 0.5
        selectionProbability := 0.8
        mutationCount := 1
        mutationFraction := 0.2
        evolutionSize := 1000
        maxConcurrency := runtime.NumCPU()

        // return output
        return &Parameters{
                SrcSubs: sourceSubscripts,
                DstSubs: destinationSubscripts,
                RndCoef: randomnessCoefficient,
                PopSize: populationSize,
                SelFrac: selectionFraction,
                SelProb: selectionProbability,
                MutaCnt: mutationCount,
                MutaFrc: mutationFraction,
                EvoSize: evolutionSize,
```

```
                ConSize: maxConcurrency,
        }
}

// new sample domain initialization function
func NewSampleDomain(rows, cols int) *Domain {

        // initialize empty matrix
        domainSize := rows * cols
        mat := make([]float64, domainSize)
        domainMatrix := mat64.NewDense(rows, cols, mat)

        // loop through index values togo define domain
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {
                        if i == 0 {
                                domainMatrix.Set(i, j, 0.0)
                        } else if i == rows-1 {
                                domainMatrix.Set(i, j, 0.0)
                        } else if j == 0 {
                                domainMatrix.Set(i, j, 0.0)
                        } else if j == cols-1 {
                                domainMatrix.Set(i, j, 0.0)
                        } else {
                                domainMatrix.Set(i, j, 1.0)
                        }
                }
        }

        // compute band count
        bandCount := 2 +
            (int(math.Floor(math.Sqrt(math.Pow(float64(rows), 2.0)+math.Pow(float64(cols), 2.0)))) / 142)

        // return output
        return &Domain{
                Rows:   rows,
                Cols:   cols,
                Matrix: domainMatrix,
                BndCnt: bandCount,
        }
}

// new sample mutation domain initialization function
func NewSampleMutationDomain() *Domain {

        // fix domain size
        var rows int = 5
```

```
        var cols int = 5

        // initialize empty matrix
        domainSize := rows * cols
        mat := make([]float64, domainSize)
        domainMatrix := mat64.NewDense(rows, cols, mat)

        // loop through index values togo define domain
        for i := 0; i < rows; i++ {
                for j := 0; j < cols; j++ {
                        if i == 0 {
                                domainMatrix.Set(i, j, 0.0)
                        } else if i == rows-1 {
                                domainMatrix.Set(i, j, 0.0)
                        } else if j == 0 {
                                domainMatrix.Set(i, j, 0.0)
                        } else if j == cols-1 {
                                domainMatrix.Set(i, j, 0.0)
                        } else {
                                domainMatrix.Set(i, j, 1.0)
                        }
                }
        }

        // eliminate center
        domainMatrix.Set(2, 2, 0.0)

        // set band count to nil
        var bandCount int = 2

        // return output
        return &Domain{
                Rows:   rows,
                Cols:   cols,
                Matrix: domainMatrix,
                BndCnt: bandCount,
        }
}

// new sample objective initialization function
func NewSampleObjectives(rows, cols, objectiveCount int) *MultiObjective {

        // initialize matrix dimensions
        objectiveSize := rows * cols
        var objectiveId int = 0

        // seed random number generator
```

```
        rand.Seed(time.Now().UnixNano())

        // initialize empty objective slice
        objSlice := make([]*Objective, objectiveCount)

        // loop through matrix indices and assign random objective values
        for k := 0; k < objectiveCount; k++ {

                // initialize empty objective matrix
                mat := make([]float64, objectiveSize)
                objMat := mat64.NewDense(rows, cols, mat)

                // write random objective values
                for i := 0; i < rows; i++ {
                        for j := 0; j < cols; j++ {
                                objMat.Set(i, j, math.Abs(rand.Float64()))
                        }
                }

                // write to objective slice
                objSlice[k] = NewObjective(objectiveId, objMat)

                // iterate objective id
                objectiveId += 1
        }

        return &MultiObjective{
                ObjectiveCount: objectiveCount,
                Objectives:     objSlice,
        }
}
```

<samples_test.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package corridor

import (
        "fmt"
        "runtime"
        "testing"
)

// small problem benchmark
func BenchmarkSmall(b *testing.B) {

        // set max processing units
        runtime.GOMAXPROCS(1)

        // initialize domain
        sampleDomain := NewSampleDomain(20, 20)
        sampleDomain.BndCnt = 3

        // initialize objectives
        objectiveCount := 3
        sampleObjectives := NewSampleObjectives(sampleDomain.Rows, sampleDomain.Cols, objectiveCount)

        // initialize parameters
        sampleParameters := NewSampleParameters(sampleDomain)

        // evolve populations
        toyEvolution := NewEvolution(sampleParameters, sampleDomain, sampleObjectives)

        // extract output population
        finalPop := <-toyEvolution.Populations

        // view output population
        ViewPopulation(sampleDomain, sampleParameters, finalPop)

        // print top individual fitness
        fmt.Println("Population Mean Fitness =")
        fmt.Println(finalPop.MeanFitness)
}

// parallel small problem benchmark
func BenchmarkParallelSmall(b *testing.B) {
```

```
        // set max processing units
        cpuCount := runtime.NumCPU()
        runtime.GOMAXPROCS(cpuCount)

        // initialize domain
        sampleDomain := NewSampleDomain(20, 20)
        sampleDomain.BndCnt = 3

        // initialize objectives
        objectiveCount := 3
        sampleObjectives := NewSampleObjectives(sampleDomain.Rows, sampleDomain.Cols, objectiveCount)

        // initialize parameters
        sampleParameters := NewSampleParameters(sampleDomain)

        // evolve populations
        toyEvolution := NewEvolution(sampleParameters, sampleDomain, sampleObjectives)

        // extract output population
        finalPop := <-toyEvolution.Populations

        // view output population
        ViewPopulation(sampleDomain, sampleParameters, finalPop)

        // print top individual fitness
        fmt.Println("Population Mean Fitness =")
        fmt.Println(finalPop.MeanFitness)
}

// medium problem benchmark
func BenchmarkMedium(b *testing.B) {

        // set max processing units
        runtime.GOMAXPROCS(1)

        // initialize domain
        sampleDomain := NewSampleDomain(20, 20)
        sampleDomain.BndCnt = 3

        // initialize objectives
        objectiveCount := 3
        sampleObjectives := NewSampleObjectives(sampleDomain.Rows, sampleDomain.Cols, objectiveCount)

        // initialize parameters
        sampleParameters := NewSampleParameters(sampleDomain)
        sampleParameters.PopSize = 10000
```

```
        // evolve populations
        toyEvolution := NewEvolution(sampleParameters, sampleDomain, sampleObjectives)

        // extract output population
        finalPop := <-toyEvolution.Populations

        // view output population
        ViewPopulation(sampleDomain, sampleParameters, finalPop)

        // print top individual fitness
        fmt.Println("Population Mean Fitness =")
        fmt.Println(finalPop.MeanFitness)
}

// parallel medium problem benchmark
func BenchmarkParallelMedium(b *testing.B) {

        // set max processing units
        cpuCount := runtime.NumCPU()
        runtime.GOMAXPROCS(cpuCount)

        // initialize domain
        sampleDomain := NewSampleDomain(20, 20)
        sampleDomain.BndCnt = 3

        // initialize objectives
        objectiveCount := 3
        sampleObjectives := NewSampleObjectives(sampleDomain.Rows, sampleDomain.Cols, objectiveCount)

        // initialize parameters
        sampleParameters := NewSampleParameters(sampleDomain)
        sampleParameters.PopSize = 10000

        // evolve populations
        toyEvolution := NewEvolution(sampleParameters, sampleDomain, sampleObjectives)

        // extract output population
        finalPop := <-toyEvolution.Populations

        // view output population
        ViewPopulation(sampleDomain, sampleParameters, finalPop)

        // print top individual fitness
        fmt.Println("Population Mean Fitness =")
        fmt.Println(finalPop.MeanFitness)
}
```

```
// large problem benchmark
func BenchmarkLarge(b *testing.B) {

        // set max processing units
        runtime.GOMAXPROCS(1)

        // initialize domain
        sampleDomain := NewSampleDomain(20, 20)
        sampleDomain.BndCnt = 3

        // initialize objectives
        objectiveCount := 3
        sampleObjectives := NewSampleObjectives(sampleDomain.Rows, sampleDomain.Cols, objectiveCount)

        // initialize parameters
        sampleParameters := NewSampleParameters(sampleDomain)
        sampleParameters.PopSize = 100000

        // evolve populations
        toyEvolution := NewEvolution(sampleParameters, sampleDomain, sampleObjectives)

        // extract output population
        finalPop := <-toyEvolution.Populations

        // view output population
        ViewPopulation(sampleDomain, sampleParameters, finalPop)

        // print top individual fitness
        fmt.Println("Population Mean Fitness =")
        fmt.Println(finalPop.MeanFitness)
}

// parallel large problem benchmark
func BenchmarkParallelLarge(b *testing.B) {

        // set max processing units
        cpuCount := runtime.NumCPU()
        runtime.GOMAXPROCS(cpuCount)

        // initialize domain
        sampleDomain := NewSampleDomain(20, 20)
        sampleDomain.BndCnt = 3

        // initialize objectives
        objectiveCount := 3
        sampleObjectives := NewSampleObjectives(sampleDomain.Rows, sampleDomain.Cols, objectiveCount)
```

```
        // initialize parameters
        sampleParameters := NewSampleParameters(sampleDomain)
        sampleParameters.PopSize = 100000

        // evolve populations
        toyEvolution := NewEvolution(sampleParameters, sampleDomain, sampleObjectives)

        // extract output population
        finalPop := <-toyEvolution.Populations

        // view output population
        ViewPopulation(sampleDomain, sampleParameters, finalPop)

        // print top individual fitness
        fmt.Println("Population Mean Fitness =")
        fmt.Println(finalPop.MeanFitness)
}
```

<types.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package corridor

import (
        "github.com/gonum/matrix/mat64"
        "github.com/satori/go.uuid"
)

// parameters are comprised of fixed input avlues that are
// unique to the problem specification that are referenced
// by the algorithm at various stage of the solution process
type Parameters struct {
        SrcSubs []int
        DstSubs []int
        RndCoef float64
        PopSize int
        SelFrac float64
        SelProb float64
        MutaCnt int
        MutaFrc float64
        EvoSize int
        ConSize int
}

// domains are comprised of boolean arrays which indicate the
// feasible locations for the search algorithm
type Domain struct {
        Rows   int
        Cols   int
        Matrix *mat64.Dense
        BndCnt int
}

// objectives are comprised of matrices which use location
// indices to key to floating point fitness values within the
// search domain
type Objective struct {
        Id     int
        Matrix *mat64.Dense
}

// multiObjective objects are comprised of a channel of individual
```

```
// independent objectives that are used for the evaluation of
// chromosome and population level fitness values
type MultiObjective struct {
        ObjectiveCount int
        Objectives     []*Objective
}

// a basis solution is comprised of the subscript indices forming
// the euclidean shortest path connecting the source to the dest
type Basis struct {
        Matrix *mat64.Dense
        Subs   [][]int
        MaxLen int
}

// chromosomess are comprised of genes which are distinct row column
// indices to some spatially reference search domain.
type Chromosome struct {
        Id               uuid.UUID
        Subs             [][]int
        Fitness          [][]float64
        TotalFitness     []float64
        AggregateFitness float64
}

// populations are comprised of a fixed number of chromosomes.
// this number corresponds to the populationSize.
type Population struct {
        Id                   int
        Chromosomes          chan *Chromosome
        MeanFitness          []float64
        AggregateMeanFitness float64
}

// evolutions are comprised of a stochastic number of populations.
// this number is determined by the convergence rate of the
// algorithm.
type Evolution struct {
        Populations     chan *Population
        FitnessGradient []float64
}
```

<visualize.go>

```
// Copyright ©2015 The corridor Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.package main

package corridor

import (
	"fmt"

	"github.com/gonum/matrix/mat64"
)

// function to print the properties of a search domain to the command line
func ViewDomain(searchDomain *Domain) {

	// get search domain matrix dimensions
	rows, _ := searchDomain.Matrix.Dims()

	// print domain values to command line
	fmt.Printf("Search Domain Values = \n")
	for i := 0; i < rows; i++ {
		rawRowVals := searchDomain.Matrix.RawRowView(i)
		fmt.Printf("%1.0f\n", rawRowVals)
	}
}

// function to print the properties of a basis solution to the command line
func ViewBasis(basisSolution *Basis) {

	// get basis solution matrix dimensions
	rows, _ := basisSolution.Matrix.Dims()

	// print domain values to command line
	fmt.Printf("Basis Solution Values = \n")
	for i := 0; i < rows; i++ {
		rawRowVals := basisSolution.Matrix.RawRowView(i)
		fmt.Printf("%1.0f\n", rawRowVals)
	}
}

// function to print the properties of a chromosome to the command line
func ViewChromosome(searchDomain *Domain, searchParameters *Parameters, inputChromosome *Chromosome) {

	// get search domain matrix dimensions and empty value slice
	domainSize := searchDomain.Rows * searchDomain.Cols
```

```
        v := make([]float64, domainSize)

        // allocate new empty matrix
        blankMat := mat64.NewDense(searchDomain.Rows, searchDomain.Cols, v)

        // assign chromosome values to the empty matrix
        for i := 0; i < len(inputChromosome.Subs); i++ {
                blankMat.Set(inputChromosome.Subs[i][0], inputChromosome.Subs[i][1], 1.0)
        }

        // print chromosome values to command line
        fmt.Printf("Chromosome = \n")
        for i := 0; i < searchDomain.Rows; i++ {
                rawRowVals := blankMat.RawRowView(i)
                fmt.Printf("%1.0f\n", rawRowVals)
        }

        // print output to the command line
        fmt.Printf("Chromosome Length = %d\n", len(inputChromosome.Subs))
        fmt.Printf("Chromosome Total Fitness = %1.5f\n", inputChromosome.TotalFitness)
}

// functions to print the frequency of chromosomes in a search domain to the command line
func ViewPopulation(searchDomain *Domain, searchParameters *Parameters, inputPopulation *Population) {

        // allocate new empty matrix
        mat := mat64.NewDense(searchDomain.Rows, searchDomain.Cols, nil)

        // accumulated visited subscripts in new empty matrix
        for i := 0; i < searchParameters.PopSize; i++ {

                // extract current chromosome from channel
                curChrom := <-inputPopulation.Chromosomes
                curInd := curChrom.Subs
                lenCurInd := len(curInd)

                // iterate over subscript indices
                for j := 0; j < lenCurInd; j++ {
                        curSubs := curInd[j]
                        curVal := mat.At(curSubs[0], curSubs[1])
                        newVal := curVal + 1
                        mat.Set(curSubs[0], curSubs[1], newVal)
                }

                // repopulate channel
                inputPopulation.Chromosomes <- curChrom
        }
```

```
        // print matrix values to command line
        fmt.Printf("Population Size = %d\n", searchParameters.PopSize)
        fmt.Printf("Population Frequency = \n")
        for q := 0; q < searchDomain.Rows; q++ {
                rawRowVals := mat.RawRowView(q)
                fmt.Printf("%*.0f\n", DigitCount(searchParameters.PopSize)+1, rawRowVals)
        }
}
```

References

[1] Adams, B., Dajani, J., & Gemmell, R. (1973). On the Centralization of Wastewater Treatment Facilities. *Journal of the American Water Resources Association*, 9(5), 1065–1065.

[2] Aissi, H., Chakhar, S., & Mousseau, V. (2012). GIS-based multicriteria evaluation approach for corridor siting. *Environment and Planning B: Planning and Design*, 39(2), 287–307.

[3] Angelakis, A. N. & Rose, J. B. (2014). *Evolution of Sanitation and Wastewater Technologies Through the Centuries*. IWA Publishing.

[4] Asano, T., Burton, F., Leverenz, H., Tsuchihashi, R., & Tchobanoglous, G. (2007). *Water Reuse: Issues, Technologies, and Applications*. Boston, MA: McGraw-Hill Professional, 1 edition edition.

[5] Association, W. R. (2011). *National Database of Water Reuse Facilities Summary Report*. Technical report.

[6] Averyt, K., Macknick, J., Rogers, J., Madden, N., Fisher, J., Meldrum, J., & Newmark, R. (2013). Water use for electricity in the United States: an analysis of reported and calculated water use information for 2008. *Environmental Research Letters*, 8(1), 015001.

[7] Bennett, D. A., Xiao, N., & Armstrong, M. P. (2004). Exploring the geographic consequences of public policies using evolutionary algorithms. *Annals of the Association of American Geographers*, 94(4), 827–847.

[8] Bixio, D., Thoeye, C., Wintgens, T., Ravazzini, a., Miska, V., Muston, M., Chikurel, H., Aharoni, a., Joksimovic, D., & Melin, T. (2008). Water reclamation and reuse: implementation and management issues. *Desalination*, 218(1-3), 13–23.

[9] California Department of Health Services Division of Drinking Water and Environmental Management (2001). *Guidelines for the Preparation of an Engineering Report for the Production, Distribution and use of Recycled Water*. Technical report, State of California-Health and Human Services Agency, Sacramento, California.

[10] California Department of Health Services Division of Drinking Water and Environmental Management (2011). *Title 22*. Technical report, State of California-Health and Human Services Agency, Sacramento, California.

[11] California Department of Water Resources Recycled Water Task Force (2003). *Water Recycling 2030*. Technical report, California Department of Water Resources.

[12] CH2M Hill (2004). *Recycled Water Project Implementation Strategies Technical Memorandum*. Technical report, The United States Bureau of Land Reclamation, Santa Ana, California.

[13] Chakhar, S. & Martel, J.-M. (2003). Enhancing geographical information systems capabilities with multi-criteria evaluation functions. *Journal of Geographic Information and Decision Analysis*, 7(2), 47–71.

[14] Church, R. L. (2002). Geographical information systems and location science. *Computers & Operations Research*, 29(6), 541–562.

[15] Church, R. L., Loban, S. R., & Lombard, K. (1992). An interface for exploring spatial alternatives for a corridor location problem. *Computers & Geosciences*, 18(8), 1095–1105.

[16] Coello, C., Lamont, G., & Veldhuisen, D. V. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*. New York, NY: Springer, second edi edition.

[17] Coello, C. A. C., Aguirre, A. H., & Zitzler, E. (2001). Evolutionary multi-criterion optimization.

[18] Collins, M. G., Steiner, F. R., & Rushman, M. J. (2001). Land-use suitability analysis in the United States: Historical development and promising technological achievements. *Environmental Management*, 28(5), 611–621.

[19] Daughton, C. G. (2004). Ground Water Recharge and Chemical Contaminants: Challenges in Communicating the Connections and Collisions of Two Disparate Worlds. *Ground Water Monitoring & Remediation*, 24(2), 127–138.

[20] Deb, K. (2001). *Multi-Objective Optimization using Evolutionary Algorithms*. New York, New York, USA: Wiley.

[21] Fonseca, C. M. & Fleming, P. J. (1995). An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary computation*, 3(1), 1–16.

[22] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, volume Addison-We of *Artificial Intelligence*. Addison-Wesley.

[23] Goldberg, D. E. & Holland, J. H. (1988). Genetic Algorithms and Machine Learning. *Machine Learning*, 3(2), 95–99.

[24] Goodchild, M. F. (1977). An evaluation of lattice solutions to the problem of corridor location. *Environment and Planning A*, 9(7), 727–738.

[25] Grübler, A. (2003). *Technology and global change*. Cambridge University Press.

[26] Grübler, A. (2010). *Technological change and the environment*. Routledge.

[27] Hallam, C., Harrison, K. J., & Ward, J. A. (2001). A multiobjective optimal path algorithm. *Digital Signal Processing*, 11(2), 133–143.

[28] Hopkins, L. D. (1977). Methods for Generating Land Suitability Maps: A Comparative Evaluation. *Journal of the American Institute of Planners*, 43(4), 386–400.

[29] Horton, M. B. (2009). *Treatment Technology Report for Recycled Water*. Technical Report 805, State of California-Health and Human Services Agency, Sacramento, California.

[30] Huber, D. L. & Church, R. L. (1985). Transmission corridor location modeling. *Journal of Transportation Engineering*, 111(2), 114–130.

[31] Jankowski, P. (1995). Integrating geographical information systems and multiple criteria decision-making methods. *International journal of geographical information systems*, 9(3), 251–273.

[32] Jaynes, E. (1988). The evolution of Carnot's principle. *Maximum-entropy and bayesian methods in science ...*, 1, 1–17.

[33] Klein, G., Krebs, M., Hall, V., O'Brien, T., & Blevins, B. B. (2005). *California's Water – Energy Relationship*. Technical Report November, California Energy Commission (CEC), Sacramento, California.

[34] Lautze, J., Stander, E., Drechsel, P., da Silva, A. K., & Keraita, B. (2014). *Global experiences in water reuse*. International Water Management Institute (IWMI); CGIAR Research Program on Water, Land and Ecosystems (WLE).

[35] Levine, A. D. & Asano, T. (2004). Recovering Sustainable Water from Wastewater. *Environmental science & technology*, 38(11), 201A–208A.

[36] Liu, H. (2003). *Pipeline engineering*. CRC Press.

[37] Lofman, D., Petersen, M., & Bower, A. (2010). Water, Energy and Environment Nexus: The California Experience. *International Journal of Water Resources Development*, 18(1), 73–85.

[38] Lombard, K. & Church, R. L. (1993). The gateway shortest path problem: generating alternative routes for a corridor location problem. *Geographical systems*, 1(1), 25–45.

[39] Metcalf, E., Eddy, H. P., & Tchobanoglous, G. (1991). Wastewater engineering: treatment, disposal, and reuse. *Water Resources and Environmental Engineering*.

[40] Mooney, P. & Winstanley, A. (2006). An evolutionary algorithm for multicriteria path optimization problems. *International Journal of Geographical Information Science*, 20(4), 401–423.

[41] Mousseau, V., Aissi, H., & Chakhar, S. (2010). A Three-Phase Approach and a Fast Algorithm to Compute Efficient Corridors within GIS Framework. *CER*, 10(7), 1–18.

[42] Neema, M. & Ohgai, a. (2010). Multi-objective location modeling of urban parks and open spaces: Continuous optimization. *Computers, Environment and Urban Systems*, 34(5), 359–376.

[43] NRC (2008). *Prospects for Managed Underground Storage of Recoverable Water*. National Academies Press.

[44] NRC (2012). *Water Reuse: Potential for Expanding the Nation's Water Supply Through Reuse of Municipal Wastewater*. National Academies Press.

[45] Pate, R., Hightower, M., Cameron, C., & Einfeld, W. (2007). *Overview of Energy-Water Interdependencies and the Emerging Energy Demands on Water Resources*. Technical Report March 2007, Sandia National Laboratory, Albuquerque, New Mexico.

[46] Pennington, D. W., Potting, J., Finnveden, G., Lindeijer, E., Jolliet, O., Rydberg, T., & Rebitzer, G. (2004). Life cycle assessment part 2: current impact assessment practice. *Environment International*, 30(5), 721–39.

[47] Rebitzer, G., Ekvall, T., Frischknecht, R., Hunkeler, D., Norris, G., Rydberg, T., Schmidt, W.-P., Suh, S., Weidema, B. P., & Pennington, D. W. (2004). Life cycle assessment part 1: framework, goal and scope definition, inventory analysis, and applications. *Environment international*, 30(5), 701–20.

[48] Roberts, S. a., Hall, G. B., & Calamai, P. H. (2010). Evolutionary Multi-objective Optimization for landscape system design. *Journal of Geographical Systems*, 13(3), 299–326.

[49] Rodrigo, D., Calva, E. J. L., Cannan, A., Roesner, L., & O'Conner, T. P. (2012). *Total Water Management*. Technical Report July, United States Environmental Protection Agency: National Risk Management Research Laboratory Office of Research and Development, Cincinnati, Ohio.

[50] Scaparra, M. P., Church, R. L., & Medrano, F. A. (2014). Corridor location: the multi-gateway shortest path model. *Journal of Geographical Systems*, 16(3), 287–309.

[51] Schwarzenegger, A., Chrisman, M., & Snow, L. A. (2005). *California Water Plan Update 2005: Highlights*. Technical report, California Department of Water Resources, Sacramento, California.

[52] Seaber, P. R., Kapinos, F. P., & Knapp, G. L. (1987). *Hydrologic Unit Maps: US Geological Survey Water Supply Paper 2294*. USGS.

[53] Stander, A. G. J., Vuuren, L. R. J. V., & Stander, G. J. (1969). The Reclamation of Potable Water from Wastewater. *Journal (Water Pollution Control Federation)*, 41(3), 355–367.

[54] Stefanakis, E. & Kavouras, M. (1995). On the determination of the optimum path in space. In *Spatial Information Theory A Theoretical Basis for GIS* (pp. 241–257). Springer.

[55] Stokes, J. & Horvath, A. (2006). LCA Methodology and Case Study Life Cycle Energy Assessment of Alternative Water Supply Systems. *International Journal of Life Cycle Assessment*, 11(5), 335–343.

[56] Stokes, J. & Horvath, A. (2011). Life-Cycle Assessment of Urban Water Provision : Tool and Case Study in California. *Journal of Infrastructure Systems*, (March), 15–24.

[57] Stokes, J. R., Hendrickson, T. P., & Horvath, A. (2014). Save water to save carbon and money: developing abatement costs for expanded greenhouse gas reduction portfolios. *Environmental science & technology*, 48(23), 13583–91.

[58] Stokes, J. R. & Horvath, A. (2010). Supply-chain environmental effects of wastewater utilities. *Environmental Research Letters*, 5(1), 014015.

[59] Tsai, C.-C., Huang, H.-C., & Chan, C.-K. (2011). Parallel elite genetic algorithm and its application to global path planning for autonomous robot navigation. *Industrial Electronics, IEEE Transactions on*, 58(10), 4813–4821.

[60] USEPA (2012). *2012 Guidelines for Water Reuse*. Technical Report September, United States Environmental Protection Agency, Office of Wastewater Management, Washington D.C.

[61] World Health Organization (2006). *WHO Guidelines for the Safe Use of Wastewater, Excreta and Greywater: Volume 1, Policy and regulatory aspects*. Technical report, World Health Organization.

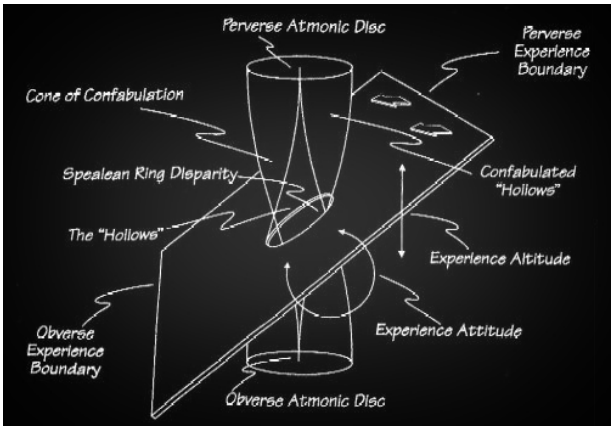
[62] Younos, T. & Caitlin A, G., Eds. (2014). *Potable Water: Emerging Global Problems and Solutions*. Springer.

[63] Zhang, X. & Armstrong, M. P. (2008). Genetic algorithms and the corridor location problem: multiple objectives and alternative solutions. *Environment and Planning B: Planning and Design*, 35(1), 148–168.

[64] Zhou, A., Qu, B.-Y., Li, H., Zhao, S.-Z., Suganthan, P. N., & Zhang, Q. (2011). Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1), 32–49.

[65] Zhou, J. & Civco, D. (1996). Using genetic learning neural networks for spatial decision making in GIS. *Photogrammetric Engineering and Remote Sensing*, 62(11), 1287–1295.

[66] Zitzler, E. & Thiele, L. (1998). Multiobjective optimization using evolutionary algorithms—a comparative case study. In *Parallel problem solving from nature—PPSN V* (pp. 292–301).: Springer.



We, amnesiacs all, condemned to live in an eternally fleeting present, have created the most elaborate of human constructions, memory, to buffer ourselves against the intolerable knowledge of the irreversible passage of time and the irretrieveability of its moments and events.

- Geoffrey Sonnabend