Application of Order-Based Genetic Algorithms
to Network Path Searching and Location Estimation

by

Walter T. Baugh

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

APPROVED

__________________________
J. S. Bay, Chairman

__________________________  ______________________
H. F. VanLandingham          C. E. Nunnally

June 1994

Blacksburg, Virginia
Application of Order-Based Genetic Algorithms
to Network Path Searching and Location Estimation

by

Walter Thomas Baugh

Dr. John S. Bay, Chairman
Electrical Engineering

Abstract

Given an incompletely connected network of nodes, populations of multiple redundant paths from each node to every other node are sought. Examples of such a scenario include telecommunications network routing, inventory control, and VLSI routing. The problem of location estimation for a network of nonstationary nodes is considered in detail as a framework for developing a solution. In particular, the problem of estimation of the actual physical positions of a system of mobile units is considered. The problem is solved by using order-based genetic algorithms, hybridized for the particular application of searching for multiple redundant subtours, increasing accuracy and robustness. With genetic algorithms performing the network routing, the modular approach used throughout is demonstrated by using four different schemes for location estimation. Implementation details are considered, with an emphasis toward eventual implementation on embedded controllers. The genetic algorithm operators are customized for the application by considering the subtour nature of the problem, as opposed to traditional applications of genetic algorithms which consider complete tours containing every node in the network. The customized genetic algorithm operators demonstrate greatly improved performance over traditional operators, both in terms of speed and robustness in the face of communication loss.
Acknowledgments

I thank my wife, Bunny, for her patience and loving endurance throughout. I also appreciate the support and encouragement of the members of my advisory committee, Dr. Bay, Dr. Nunnally, and Dr. VanLandingham. I also am deeply grateful for the assistance of Dr. Kenneth Knowles and Dr. Thomas Bechert, currently professors at the United States Naval Academy, and Major Paul McGraw, USMC, who made it possible for me to attend graduate studies as an Air Force Laboratory Graduate Fellow.
# Table of Contents

Table of Figures ................................................ vii

Chapter 1 - Introduction ........................................ 1
  1.1 - Problem Statement .................................... 1
  1.2 - Solution Concept .................................... 3
  1.3 - Chapter Organization .................................... 4

Chapter 2 - Background .......................................... 6
  2.1 - Genetic Algorithms .................................... 6
    2.1.1 - History .......................................... 7
    2.1.2 - Functional Description ........................ 8
    2.1.3 - Selection and Replacement ..................... 9
    2.1.4 - Order-Based Genetic Algorithms ............... 10
    2.1.5 - Hybridization .................................... 11
  2.2 - Applications .......................................... 13
    2.2.1 - Network Switching ............................ 13
    2.2.2 - Inventory Control ............................ 14
    2.2.3 - Mobile Vehicles ............................ 15
    2.2.4 - VLSI/PCB Routing ............................ 15
  2.3 - Definition of Terms .................................... 16

Chapter 3 - A First Approximation .................................... 19
  3.1 - Model and Notation .................................... 20
    3.1.1 - Nodes, Links, Paths, and Measurements ............ 20
    3.1.2 - Distance .................................... 21
  3.2 - Estimation Calculus .................................... 22
    3.2.1 - Composite Observations ....................... 22
    3.2.2 - Location Estimates ............................ 23
  3.3 - Simulation Results .................................... 25
    3.3.1 - Simulator Architecture ....................... 25
    3.3.2 - Static Line .................................... 27
    3.3.3 - Dynamic Line .................................... 28
    3.3.4 - Grid and Circle .................................... 29
    3.3.5 - Static Pallet .................................... 29
    3.3.6 - Error Rejection vs. Accepted Scrambling ...... 30
    3.3.7 - Gaussian Error vs. Step Error ................. 31
  3.4 - Performance Models .................................... 33
    3.4.1 - Communications Bandwidth ..................... 33
    3.4.2 - CPU Consumption ............................. 34
    3.4.3 - Memory Consumption ............................ 35
  3.5 - Interim Conclusions .................................... 36
Table of Figures

3.1 - An example of links, paths, composite observations, and distances ....... 20
3.2 - Static Line ............................................................................. 27
3.3 - Dynamic Line ........................................................................ 28
3.4 - Static Pallet .......................................................................... 30
3.5 - Rejection versus Scrambling ............................................. 31
3.6 - Gaussian versus Step Noise ............................................. 32

4.1 - Example for application of analogous crossover .................. 43
4.2 - Example of the outside cross matching operator .................. 48
4.3 - Example of the inside cross matching operator ................... 50
4.4 - Example of the modified crossover operator ..................... 52
4.5 - Example of the linear inversion operator ......................... 54
4.6 - Example of the circular inversion operator ....................... 55
4.7 - Example of the randomized reduction operator ............... 56

5.1 - Simple trigonometry example ........................................... 63
5.2 - Plot of means versus variance .......................................... 66
5.3 - 3D plot of transformed polar probability density function ......... 67
5.4 - Plot of 200 sample measurements ..................................... 68
5.5 - Example of combining biases .......................................... 69

6.1 - Simulator measurement model ........................................... 78

7.1 - An example of a wide field pattern ................................. 94
7.2 - An example of a deep field pattern ................................. 95
7.3 - Comparison of operator sets on population size in a wide field 100
7.4 - Comparison of operator sets on targets found in a wide field 101
7.5 - Comparison of operator sets on population size in a deep field 102
7.6 - Comparison of operator sets on targets found in a deep field 103
7.7 - Comparison of sensor noise on average error in a wide field 104
7.8 - Comparison of sensor noise on path fitness in a wide field 105
7.9 - Comparison of sensor noise on average error in a deep field 106
7.10 - Comparison of sensor noise on path fitness in a deep field 107
7.11 - Comparison of communication loss on population size in a wide field 108
7.12 - Comparison of communication loss on targets found in a wide field 109
7.13 - Comparison of communication loss on population size in a deep field 110
7.14 - Comparison of communication loss on targets found in a deep field 111
7.15 - Comparison of various estimators ................................. 112
7.16 - Deep field pattern used to compare operator sets 2 and 3 .... 113
Chapter 1 - Introduction

This work discusses application of one artificial intelligence tool, genetic algorithms, to the problem of network path finding. Specifically, multiple redundant near-optimal paths are sought as a means to increase location estimate accuracy given a database of measurements representing connections between nodes.

1.1 - Problem Statement

Given a system of mobile, dispersed elements, such as surface ships, or perhaps armored vehicles and infantry units, estimates of location to each element from every other element are sought. This situation can be considered to be a network of incompletely connected nodes, where the nodes are the elements themselves and the connections are measurements from sources such as radar contact, laser rangefinders, or transponder time delays. Incompletely connected, as used throughout, means that not every node is directly connected to all other nodes. However, all nodes are members of one network, and not two or more coexisting, but unconnected, networks. In other words, paths from any node to any other node exist.

The network structure provides information which is sufficient to form the desired location estimates. To exploit this information, populations of multiple, redundant paths to any particular element through the network structure are sought. These populations of paths, once found, provide improvements which depend on redundancy either for accuracy, robustness, or solubility, or a combination of these. These improvements can be obtained by using the fact that each edge, or connection between nodes, has associated with it a quantity, or weighting, which affects the quality of paths containing that edge. One
example of such a quantity is distance between nodes, although direct measurements will be corrupted by noise. As a result, reasonable models concerning physical sources of disturbances must be considered in the solution.

To provide a foundation for discussion, a simple example of a number of vehicles with primitive, and thus noisy and of limited range, sensors will be considered. This system of vehicles can not rely upon outside sources of information such as base stations, surveyed checkpoints, or similar aids other than a common bearing reference, such as magnetic north. In addition, the system possesses a broadcast net capable of transmitting information to all units, even those beyond direct measurement. However, it is assumed that this broadcast net is subject to random noise which corrupts measurement transmissions. One possible scenario is that of ultrasonic sensors and an RF broadcast net. In such a scenario, units in broadcast range of each other may not be able to make measurements due to the limited range of the ultrasonic transducers.

This thesis presents a method which can produce location estimates in such a situation despite sensor noise, communication loss, and limited sensor range. In addition, the techniques used are applicable to other problems, as will be discussed later.
1.2 - Solution Concept

Genetic algorithms, described in detail in Chapter 2, and in particular order-based genetic algorithms, described in Chapter 4, will be used to generate and maintain the required populations of multiple redundant paths throughout the network. In general, genetic algorithms are an optimization technique which iteratively forms successively better solutions to a problem by combining previous solutions. Usually, solution parameters are coded as binary strings. Order-based genetic algorithms are a variant of genetic algorithms in which the information is coded as a list, such as a list of nodes in a network.

The implementation of the genetic algorithms used in this thesis will pay particular attention to the subtour, or incomplete tour, nature of the problem. In other words, rather than finding complete tours, or circuits, through the network, the solution concerns paths which contain less than all the nodes in the system. To this end, the order-based genetic algorithm operators will be customized for the problem at hand.

The implementation of this system will be modular, facilitating maintenance. One advantage of this structure is that changes in one functional area have no effect on the other. Indeed, throughout development, many changes were effected by simply “unplugging” one module and substituting another in its place. As a demonstration of this modularity, four estimation techniques will be implemented, each of which constructs location estimates based on a measurement database and the latest population of paths. While any of these four estimation techniques would provide reasonable estimates, the goal is to show that the paths produced by the system are independent of the estimation technique.
1.3 - Chapter Organization

Chapter 2 presents background information concerning genetic algorithms and applications of the techniques presented in this thesis. Specifically covered in this chapter are the relevant history of genetic algorithms and a discussion of their basics. Also introduced are the concepts of order-based genetic algorithms and hybridization of genetic algorithms. This chapter concludes with a sampling of potential applications for the methods discussed.

Chapter 3 lays the groundwork for application of these techniques to one problem, that of location estimation for an incompletely connected network of mobile vehicles. In addition, intermediate simulation results are presented that motivate further development in later chapters.

Chapter 4 presents a more detailed treatment of order-based genetic algorithms than that given in Chapter 2. First, relevant literature is reviewed for applicability, and alternative approaches are justified. Next, the eight order-based genetic algorithm operators specifically used in later simulations are described in detail. This chapter concludes with a brief discussion of the applicability of simulated annealing to the problem.

Chapter 5 is devoted to an exploration of estimation techniques. While this chapter specifically relates to the location estimation problem discussed in Chapter 3, it is also an example of the modular approach used in the solution to the location estimation problem. A three-level estimator hierarchy is used, demonstrating the decoupled elements of the solution, with order-based genetic algorithms linking the first layer with the two remaining estimator layers. This chapter concludes with a derivation of the bias
encountered when converting from polar coordinates to rectangular coordinates, a necessary operation in the case of location estimation.

Chapter 6 covers the more practical concerns of implementing a genetic algorithm approach to the location estimation problem described in Chapter 3. While the genetic algorithm operators themselves were detailed in Chapter 4, this chapter builds a conceptual structure around them suitable for practical implementation. Also included in this structure are estimator implementation details. The chapter also includes a brief discussion of the implementation of a simulator.

Chapter 7 presents simulation results. First, key issues requiring simulation are discussed, along with model definitions used in the simulation. Simulation results are presented which complement those given in Chapter 3. Then, the simulation results are interpreted.

Chapter 8 summarizes the lessons learned throughout the development of these techniques. First, the performance of various genetic algorithm operators are considered. Next, the performance of the estimators are discussed. Finally, practical issues surrounding implementation of these techniques are covered, along with recommendations for future work.
Chapter 2 - Background

Before delving in great detail into the particular approach to the problem at hand, a brief discussion of genetic algorithms and possible applications is in order.

2.1 - Genetic Algorithms

The invention of genetic algorithms, a diverse branch of artificial intelligence, is credited to John Holland [8]. One of his students, David Goldberg, published what many consider to be the definitive tutorial on genetic algorithms, *Genetic Algorithms in Search, Optimization, and Machine Learning* [10]. In this text, Goldberg presents a detailed description of traditional genetic algorithms along with several modifications that significantly improve performance. In addition, Goldberg supplements this material with several examples from real applications, including his own work with the control of natural gas pipelines [10].

The other key text in the field, *Handbook of Genetic Algorithms*, edited by Lawrence Davis, is an excellent collection of applications of genetic algorithms to a variety of disciplines [8]. In addition, Davis includes a brief tutorial on more advanced topics, which nicely complements Goldberg’s text. One topic, hybridization, discussed in detail by Davis, is a key theme that played a large role in the development of the techniques described in this thesis.

Genetic algorithms are iterative tools for solving optimization problems that are difficult, if not impossible, to solve by more traditional means. The central concept is to iteratively form hypotheses about the solution to a problem and test those solutions for
fitness, or accuracy. Then, form new hypotheses based on combinations of the better recent hypotheses and repeat the cycle. As more iterations are completed, the population of hypotheses, or "genes", tends to improve as better genes, and better combinations, are created [8][10].

2.1.1 - History

John Holland is credited with creating genetic algorithms as they exist today [8]. In the early 1960s, while teaching courses in adaptive systems at the University of Michigan, he noticed that iterative solutions of function optimization problems primarily used and produced single solutions to continuous functions by using derivative, or gradient, information. An example is Newton's Method for root-finding. This algorithm, in general, takes as an argument a single value, and produces a single value as root of a smooth function. However, these techniques are ill-equipped to handle discontinuous functions. Further, such methods as hill-climbing and simplex tend to converge on themselves, often in a local, rather than global, minimum [17].

The conceptual leap taken by Holland was recognizing that the theories of natural selection and evolution may apply to function optimization [10]. That is, instead of relying on foreknowledge of the optimization problem under consideration, such as the derivatives required by Newton's Method, for example, a more robust optimization method should be based on the actual fitness, or function evaluation, of the system, treated as a "black box." Then, from those tentative solutions that best satisfy the problem, new solutions are created using the natural processes of crossover and mutation. Viewed this way, simplex methods are just a special case of crossover.
The term "genetic algorithm" is credited to J. Bagley, a student of Holland, who used crossover and mutation operators for optimizing a hexapawn problem [10]. In addition, Bagley also introduced the inversion operator that creates new genes by inverting the order of a segment of a gene. The inversion operator is one simple, and relatively ineffectual in the current case, unary operator. Other more powerful unary operators will be introduced later in Chapter 4.

2.1.2 - Functional Description

Genetic algorithms are tools for function optimization. While other definitions abound, for the current discussion this definition suffices. Traditional genetic algorithms take a black-box approach to function optimization by generating a set of arguments for an unknown function, then examining the performance of that set of arguments [10]. In practice, the assumption of an unknown function is usually realistic. Although one may have identified models for the system in question, that model may not be complete nor even accurate. This ability of genetic algorithms to tackle problems without a complete model is one aspect attracting a great deal of attention, especially for defense-related work [11].

As an example, consider the telecommunication switching problem discussed by Cox, et. al., in Davis [8]. While the authors take great pains to develop models capturing the dynamic nature of this problem, disturbances such as weather, sun spots, vandalism, etc., will have effects on the system that are difficult, if not impossible, to adequately capture in a model. As a result, attempting to deterministically assign network paths based on such a model could lead to disappointing results.

Fortunately, genetic algorithms provide a measure of robustness to systems in which they are employed. An adaptive technique that can sense changes in a system's
dynamic response to commands, genetic algorithms, in their traditional sense, perform quite well without a detailed model [8] . In the example above, should rioting disrupt operations by having microwave antennas and associated equipment carted off by looters, a genetic algorithm could detect that previously useful commands are no longer effective.

The concept behind genetic algorithms is simple. First, generate possible solutions to a problem. Next, implement those solutions and examine the results. Finally, use the best solutions and attempt to create better solutions [10] .

Traditional genetic algorithms implement this concept by working with solutions coded as binary strings [10] . Generally, these binary strings are divided into parameter fields. Then, using reproduction functions known as operators, these strings are combined with one another to form new strings in an identical format. Typically, the binary strings are then assigned fitness values based on the quality of the output provided by the encoded solution. The strings with better fitness values are then used to create new strings.

2.1.3 - Selection and Replacement

A necessary component of any genetic algorithm is a method for selecting strings for reproduction. The many reproduction selection techniques available represent a completely separate topic. One popular method is roulette-wheel selection [10] . In this method, each member of the population is assigned a probability of selection proportional to fitness. In other words, each member is assigned a slot on a roulette wheel, with the size of the slot proportional to fitness. Then, a random number is generated, and the member whose slot is chosen is allowed to reproduce. Although this is a popular technique, it entails certain risks. The major risk in this technique is that it is possible,
although not likely, that the new population will contain members that are the offspring of the worst member of the population.

Another issue concerning reproduction is the method of replacing old members of the population with new members. In traditional applications, the entire generation is replaced by offspring, a technique appropriately known as *generational replacement* [8]. Unfortunately, this technique is also risky. Again, although not likely, it is possible that an extremely effective solution is lost by producing inferior offspring. If the entire generation is replaced by the offspring, then disaster results.

2.1.4 - Order-Based Genetic Algorithms

The traditional genetic algorithms discussed above depend on binary coded strings. The particular variant of genetic algorithm used here is order-based genetic algorithms. The term *order-based* implies that, rather than coding information as integers, information is coded by lists of ordered elements [8]. These ordered lists can be thought of as representing tours, or closed paths through a network. The first part of Chapter 4 is devoted exclusively to issues associated with order-based genetic algorithms. In turn, the material in Chapter 4 is based on material presented in Chapter 3. The basics of order-based genetic algorithms are discussed here, with a detailed discussion left for Chapter 4.

Order-based genetic algorithms are so named because the location of information in each string is important. In more general genetic algorithms, parameter A may be encoded before or after parameter B. However, with order-based genetic algorithms, each string is an ordered list of possibly unique elements. Each list in the current
application is a set of instructions describing how to reach a target node by stepping from one intermediate node to another.

As a result of the special nature of order-based genetic algorithms, many of the traditional operators do not directly apply. Although a great deal of modification of these operators is required, new opportunities arise for creating other operators not possible with conventional genetic algorithms. These issues will be handled more completely in Chapter 4. One special consideration with this particular implementation is that the entire system, for consistency, requires that each gene be a complete list of nodes, where each node is represented exactly once. This restriction aids implementation of the operators described below, as well as simplifies memory management.

One example of an application suited for order-based genetic algorithms is network routing [8]. A natural way of coding a path through the network is simply to list the nodes in the order that they appear along the path. However, with this coding, the traditional operators such as crossover lose effectiveness, particularly if the coding for a node is split by poor selection of a crossover point. For example, if an eight-bit coding were used for nodes in a network with two hundred nodes, crossover may create a path with nonexistent node names. As a result, operators must be carefully crafted with the application in mind, a departure from traditional uses of genetic algorithms.

2.1.5 - Hybridization

Such crafting of operators to suit the application is known as hybridization, a practice strongly advocated by Davis [8]. Genetic algorithm purists, abhorring hybridization, insist that the conceptual strength of genetic algorithms is their application independence. In other words, genetic algorithms may be readily applied to problems as
diverse as air-injected hydrocyclone optimization or DNA mapping, with little or no change to the underlying software. Unfortunately, as with any generic technique, failure to fit the solution to the problem often results in mediocre performance as well.

Jog, et. al., refer to hybridized genetic algorithms as Heuristic Genetic Algorithms, or HGA, if problem specific heuristics are incorporated in the operators [12]. While this is probably a more precise term than simply hybridization, both terms embrace the same concept. As a result, the term hybridization is used throughout this thesis.

The solution presented here involves a great deal of hybridization. In particular, the reproduction operators are designed with the goal of first finding any path to a target, and then fleshing out and improving the population. A robust set of operators will be discussed in Chapter 4. As will be shown in Chapter 7, the traditional operators are woefully inadequate.
2.2 - Applications

The techniques developed in this thesis apply to a wide variety of problems. A few of these applications will be discussed in this section. The solution to the path search problem, as described in Chapters 3 and 4, resembles a network routing problem, which indeed it is. Be the network a commercial telecommunications grid, a Local Area Network (LAN), or simply an embedded controller network in an automobile, for example, many of the macroscopic concepts apply. One advantage of the method employed here is the robustness of the solution in the face of random node failure. In fact, as will be apparent, often the system is prepared with many redundant solutions to fill the breach caused by failure.

Other applications require redundant approaches through a network to even operate. One example that comes to mind is a recent proposal for inventory management. RF transponders, attached to pallets, may communicate with neighbors in the immediate vicinity, making range-only measurements where possible. With only single paths through the network, unambiguous location estimates are impossible. Multiple redundant paths are necessary to pinpoint locations in two or three dimensions, similar to the concept behind the Global Positioning System (GPS).

2.2.1 - Network Switching

Telecommunication networks can benefit from these techniques, especially when connecting trunks have varying capacities and availabilities. Although these techniques were developed with measurement lengths in mind, remaining capacity on a trunk could represent a form of "distance". In addition, certain connections could be given more or
less weighting than others. Then, routing could be optimized with regard to various measures, such as equalizing remaining capacity on all trunks, maintaining peak usage of more expensive connections, reserving capacity on critical trunks, etc.

A further benefit to telecommunication networks concerns transient disconnections, such as the vandalism example given earlier. Many redundant path solutions to any target node can be maintained in the background, allowing immediate traffic switching around unplanned disconnections.

2.2.2 - Inventory Control

Another application of these techniques is inventory control. Large organizations, especially government agencies, are vulnerable to loss and/or pilferage with regard to warehousing of large amounts of material. As any experienced librarian will admit, the easiest way to lose a book is to store it in the wrong place. Similarly, moving cryptically-marked containers around in a large warehouse is a sure recipe for loss.

One current proposal for these techniques concerns a U. S. Army contract for inventory control in government warehouses. The concept involves "checking-in" pallets by attaching a transponder and logging the pallet contents and transponder code into a database. During storage, the system would periodically form a model of the pallet locations using the transponder network. When a particular item is desired, the system is queried for the appropriate pallet location. Then, the desired pallet can be retrieved and "checked-out." In this application, the system goal is to form location estimates based on time delays through various paths. As a result, multiple redundant paths to each target are necessary, an inherent feature of the techniques developed in this thesis. Fortunately, accuracy on the order of meters is sufficient.
2.2.3 - Mobile Vehicles

Another possible application of this material involves location estimation for a population of mobile vehicles. Similar to the inventory control application above, multiple paths to each target node are necessary to form location estimates with sub-meter accuracy. Chapter 3 introduces some of the techniques used here based on an example of this application.

2.2.4 - VLSI/PCB Routing

Very Large Scale Integration (VLSI) and Printed Circuit Board (PCB) design software depends heavily on effective routing algorithms. The techniques discussed here are applicable, with some modifications. As a distance measure, the number of vias and/or layers in a path could be used along with the line distance to minimize maze routes. The greatest benefit of these methods is its parallel nature, routing all nets simultaneously rather than serially. For example, Tango Route, a PCB routing tool, routes nets around previously routed nets, whereas a parallel router could find global optimums that a serial router could not.
2.3 - Definition of Terms

Terms commonly used throughout this thesis are defined below.

*Composite Observation* - An interim location estimate formed by combining a set of observations corresponding to paths in a population.

*Edge* - A connection between nodes in a network. An edge may represent a measurement from one node to another, for example, or may be a physical link, such as a fiber optic connection or a circuit board trace.

*Equality Rejection* - A method of preventing saturation of a population by detecting and discarding identical members.

*Fitness* - A measure of the quality of a hypothesized solution used in a genetic algorithm.

*Generational Replacement* - A popular genetic algorithm replacement technique in which the new generation completely displaces the old generation, regardless of relative fitness values.

*Genetic Algorithms* - An optimization technique that seeks to improve solutions by iteratively generating hypotheses. Then, better solutions are sought by creating new hypotheses as combinations of old hypotheses.

*Hybridization* - The adaptation, or customization, of genetic algorithm operators to the specific problem under consideration.
**Measurement** - A raw sensor quantity that is subject to noise.

**Network** - A collection of elements, or nodes, linked together with edges.

**Node** - A point in a network, or an individual element in a distributed system.

**Observation** - An interim location estimate obtained by the vector sum of measurements along a path.

**Operator** - A genetic algorithm reproduction function that creates new hypotheses by combining old hypotheses in a particular way. For example, an inversion operator creates new hypotheses by reversing, or inverting the order, of elements of old hypotheses.

**Order-Based Genetic Algorithms** - A version of genetic algorithms in which the hypothesized solutions are formulated as lists of discrete elements rather than numeric data, for example lists of nodes in a network.

**Path** - A series of edges forming a route through a network connecting two or more nodes in series.

**Population** - In general, a set of hypothetical solutions used by a genetic algorithm. For this particular application, a set of paths to a particular node.

**Rejection** - Discarding data based on a detected error in a transmission.

**Roulette Wheel Selection** - A popular genetic algorithm selection technique in which hypotheses are chosen for reproduction based on a normalized fitness score share across the entire population.
**Shorth** - Also the shortest half, that subset, containing half the elements, of an ordered set of data that has the minimum distance from the first element to the last element. Used in mode estimators.

**Simplex** - An optimization technique that seeks to improve a solution by forming a multidimensional box around a local minimum or maximum. Then, a better solution is sought by shrinking the box toward the minimum or maximum.

**Simulated Annealing** - An analogy to the metallurgical process of annealing applied to optimization problems.

**Source** - The node from which a path originates.

**Subtour** - A path through a network that is a subset of the entire set of nodes.

**Target** - The node to which a path is sought.

**Tour** - A closed path through a network containing the complete set of nodes.
Chapter 3 - A First Approximation

The context of this chapter is a population of mobile vehicles [19][20], but the underlying concepts are applicable to other contexts, such as telecommunication networks, etc. The application at hand involves a number of mobile vehicles that, to perform their assigned tasks, require knowledge of the location of each of their peers [23]. These vehicles are equipped with ultrasonic sensors of limited range, which are used to obtain noisy range and bearing measurements to each neighbor within range [1].

Due to sensor range limitations and obstructions, not every vehicle will be able to measure each neighbor [1]. At regular intervals, each vehicle transmits these measurements with the entire community on a broadcast net, presumably RF, which itself is subject to some degree of noise. The protocol used on the common net is unspecified, but provides a facility for error detection, but not correction, as well as allowing each vehicle to, in turn, transmit its set of observations to the community. Each vehicle must be equipped with an on-board facility, such as a compass, to maintain a common bearing frame of reference. However, there does not exist a well-surveyed network of base stations, as required by some schemes [13][15], nor even a single surveyed base station.

It is necessary that each vehicle can uniquely identify each neighbor during the measurement process via a handshaking process and that each vehicle will in turn broadcast all observations to every other vehicle in the system. Receipt of each broadcasted measurement is degraded by communications noise, causing receiving vehicles to identify and reject, but not correct, a certain percentage of corrupted observations. It is unnecessary for the vehicles to have a priori knowledge of the error statistics.
3.1 - Model and Notation

Given a set of \( n \) vehicles, we uniquely identify each by a symbol, in this case a subscripted \( H \). Select one vehicle to be the observer/estimator, and another to be the target, identified by \( H_0 \) and \( H_T \), respectively. Note that the choices for observer/estimator and target are completely arbitrary, as each vehicle will compute location estimates for every vehicle in the system, even those out of direct observation. The goal of the estimation process is to determine a new estimate based on a combination of the previous estimate and a composite observation derived from the latest set of communicated observations. A composite observation may be formed by calculating a weighted average of the direct observation, if it exists, from \( H_0 \) to \( H_T \), with indirect observations from \( H_0 \) to \( H_T \) formed by a path of observation weaving through other vehicles, as will be discussed shortly.

![Figure 3.1](image)

**Figure 3.1** - An example of links, paths, composite observations, and distances.

3.1.1 - Nodes, Links, Paths and Measurements

Any observation, or measurement, will be converted, prior to storage in the on-board measurement database, from local polar coordinates into an observation vector in
global Cartesian coordinates. Note that any bearing error due to an error in orientation can be absorbed into the measurement bearing error. Let an observation vector, or link, from any vehicle, or node, $H_x$ to any other node $H_y$ be denoted by $L_{xy}$. Let any particular ordered set $A$ of such links, or a path $A$, between the node $H_O$ to the target node $H_T$ be denoted by $P_{OT,A}$. Note that a path may consist of simply the single link $L_{OT}$ or the vector sum of a number of intervening links $L_{Oj} + L_{jk} + \cdots + L_{mn} + L_{nT}$. The previous paths may be denoted by $P_{OT,A}$ and $P_{OT,b}$, respectively. In the current model, we allow the path to touch each node only once, i.e., a path containing links such as $\{ \cdots, L_{xj}, L_{jk}, L_{kj}, L_{jy}, \cdots \}$ is disallowed. There may be many possible paths from $H_O$ to $H_T$. See Figure 3.1 for an example.

3.1.2 - Distance

Let the distance function $d(P_{xy,A})$ produce a scalar integer representing the number of links in any particular path $P_{xy,A}$. Thus, $d(P_{xy,A})$ maps any particular path $P_{xy,A}$ onto the integers $1 \leq d(P_{xy,A}) < n$. For any distance $d$, there may be many possible distinct paths, depending on the number of available observations.
3.2 - Estimation Calculus

This section introduces techniques initially used to form location estimates based on a measurement database.

3.2.1 - Composite Observations

By the principle of maximum likelihood, to form the best composite observation $Z_k$ for any iteration $k$, the system must apply a weighted average inversely proportional to the variance of the estimate error associated with the individual paths [22], as given by (3.1).

$$Z_k = \frac{\sum w_i P_{OT,i}}{\sum w_i} , \quad w_i = \frac{1}{\sigma_{P_{OT,i}}^2}$$  \hspace{1cm} (3.1)

From any unit's perspective, the variance of the noise affecting the sensors is unknown, and may change depending upon the application. Therefore, the variance of the error in the paths, that is the sum of the variances of the error in each of the links, cannot be determined. However, if we assume that the sensors are identical, then the variance of the error in any path is simply the product of the distance $d(P_{OT,i})$ and the variance of the error in any link. With this knowledge, the link variances cancel, leaving (3.2):

$$Z_k = \frac{\sum w_i P_{OT,i}}{\sum w_i} , \quad w_i = \frac{1}{d(P_{OT,i})}$$  \hspace{1cm} (3.2)
Therefore, it is possible for each unit to apply a weighted average of paths to form a composite observation \( Z_k \) independent of the distribution of the sensor noise. From time to time there may be no existing path from one node to another, should communication noise destroy critical links. Should this occur, in the current model, the system will simply not update the estimate, but will wait for the next pass of observations to fill in the gaps. As demonstrated later, simulations indicate that these lapses have little effect. This method has the advantage of allowing asynchronous updates, further increasing the robustness of the method.

### 3.2.2 - Location Estimates

At this point, composite observations, the most recent combinations of measurement data, have been calculated to each unit, as described above. The next step in the estimation process is to form a new position estimate to each unit, \( X_{k+1} \), based upon a combination of the previous estimate, \( X_k \), with the composite observation, \( Z_{k+1} \). The method used is adapted from Lewis [14], where the \( b \) parameter represents a weighting factor and will be discussed below.

\[
X_{k+1} = X_k + \left( \frac{1}{b+1} \right) (Z_{k+1} - X_k)
\]  

(3.3)

In the static case, where the vehicles are not moving but are simply collecting observations, it can be shown that the optimum choice for \( b \) in (3.3) is \( k \), the number of the current iteration, resulting in a simple mean over time [14]. As will be demonstrated later, setting \( b = k \) is insufficient for realistic applications. The equivalent discrete time system, which is simply the z-transform of (3.3) is:
Examination of this transfer function (3.4) will give some insight into the expected performance of the estimator. As indicated by this transfer function, the pole moves from \( z=0 \) to \( z=1 \) with \( b \) increasing from 0 to \( \infty \), decreasing the effect of new information on the estimates.

As expected from analysis of the first order dynamics, simulations have shown that incrementing \( b \) with every iteration, which is equivalent to setting \( b = k \) in (3.3), results in very poor estimates in the dynamic case, where the vehicles are moving, since as time elapses the system responds less and less to new data. A better choice for \( b \) in the dynamic case is a constant, which allows new data to effect the estimates equally at all times. Also, lower values of \( b \) produce greater responsiveness, yet more susceptibility to noise. Simulations have shown that values of \( 10 < b < 50 \) produce acceptable results, with values near the lower end producing the best compromise between static and dynamic performance.

Another perspective on the influence of the parameter \( b \) is to consider it as a memory parameter. Lower values of \( b \) tend to place greater emphasis upon the validity of more recent measurements with respect to the latest estimate. Conversely, higher values of \( b \) favor the validity of the previous estimate over the more recent measurements.
3.3 - Simulation Results

To illustrate the effects of various conditions, we will consider several scenarios. Some scenarios consider the static case in which the vehicles are assumed to have halted, for whatever reason. Other scenarios consider the dynamic case, in which the vehicles are moving toward various objectives, which may or may not be a common goal. In each scenario, the average of 100 trials is presented. The goal is to provide a more accurate indication of what the system can accomplish rather than report the results of a single trial, which may not truly represent the performance of the system.

3.3.1 - Simulator Architecture

The simulation program operates in the Microsoft Windows™ environment. This simulator enables the user to graphically and interactively create and destroy vehicles to test responsiveness. In addition, the user can move existing vehicles and change simulator parameters without recompilation. In addition, several repeatable test patterns are available to determine the average response of the system to repeated trials with the same parameters. It is possible to select overlays presenting raw observations, single unit estimates formed by direct observations, and group estimates formed by composite observations, which contributes greatly to an intuitive feel for the effect of various conditions. Also, the message processing protocol required by Windows™ emulates the expected protocol to be used in an application, increasing the portability of code from the simulator to working models.

In the simulator, the vehicles are modeled as 0.5 meters square, on a field of approximately 50.0 meters by 40.0 meters. The anticipated use of ultrasonic sensors employing a handshaking protocol resulted in the estimation of the noise parameters listed
below as well as the range limitation of 20 meters [1][2]. Obviously, some applications will require modification of the sensor apparatus, but these parameters are considered sufficiently conservative to provide encouraging results. Several simulator options were implemented to account for various scenarios:

**Communication Noise:**

None (Clear Communication)
- Detect errors in and reject 5% or 20% of transmitted measurements
- Accept and use 5% or 20% of scrambled measurements

**Parameter b:**

10, 20, 50, or \( k \)

**Test Patterns:** The following arrangements of nodes are available in the simulator.

- Circle: 9 units, radius 15.0 meters
- Grid: 9 units, 3x3 grid, 12.0 meter spacing
- Line: 9 units, 6.0 meter horizontal spacing, 4.0 meter vertical spacing
- Pallet: 8 units at the border of a 3 meter pallet, each 0.5 meters from the centerline.
- Interactive: mouse-driven interface to allow user selectable number and placement

**Measurement Error Statistics:** Based on references [1] and [2], the following statistics represent realistic assumptions.

**Gaussian Polar**

\[
\mu_{\text{range error}} = \mu_{\text{bearing error}} = 0 \\
\sigma_{\text{range error}} = 0.5 \text{ and } \sigma_{\text{bearing error}} = \pi/24
\]

**Step Polar:** Quantization to discrete values
- Range: 2.0 meter increments
- Bearing: 16 compass directions
At each increment in the figures to follow, the following steps occur. First, each unit measures all other units in range. Then, in turn, the units transmit their measurements to all other units. Finally, using equations (3.2) and (3.3), the units form location estimates to all other units, including those out of direct measurement range.

3.3.2 - Static Line

This scenario, in which the vehicles are arranged in a line, each observing only one or two neighbors to each side, can occur in a variety of applications. Some of the most common involve mine sweeping, or anti-submarine warfare. The goal of the units is to orient themselves to the location of all neighbors. The major difficulty of this arrangement is the lack of redundant paths to remote neighbors, making all measurements highly sensitive to wildly inaccurate observations by very few units. All trial runs in this condition are from the perspective of one of the end units, using the Gaussian Polar error model and 5% communication rejection. The following graph, and all that follow, show the error in meters, averaged over all trials and all units for each iteration.

![Static Line Graph](image)

**Figure 3.2** - Static Line. All vehicles are fixed in position and are attempting to estimate the location of each other.
Experimental results, given as Figure 3.2, show that after 40 or so iterations, all values of $b$ except for 50 produce an estimate within 0.5 meters per unit for the entire system. Also, as expected, the value $b = k$ outperforms all other options.

3.3.3 - Dynamic Line

This scenario is a more realistic extension of the static case. Shortly after arriving in the target area, the vehicles begin to move to new locations. Although they could very well transmit crude course and speed information, which could then be used to transform the estimate to the correct value, various factors affect the accuracy of this data. For example, measurement granularity, wheel slippage, and terrain grade may all contribute to errors in course and speed.

![Dynamic Line Graph](image)

**Figure 3.3** - Dynamic Line. Each vehicle is shifted 0.05 meters every two observation cycles between cycle 30 and cycle 90.
To model these errors, each vehicle is shifted 0.05 meters once every two iterations, each in a different but constant direction, after 30 static iterations. This shifting occurs 30 times, after which the positions are held constant to demonstrate how various methods track the new static condition. Again, the trials are from the perspective of one of the end units, using Gaussian Polar error and 5% rejection.

The experimental results, Figure 3.3, show a marked difference in the ability to track a shifting target, as predicted earlier. In this context, setting the parameter $b = k$ produces clearly unacceptable results, while the value $b = 10$ tracks fairly well. Note that the error peaks occur at the last position shift. The response given by $b = 10$ is nearly indistinguishable from the normal "bobble" in the static case, maintaining roughly 0.5 meters error per unit.

3.3.4 - Grid and Circle

Although not shown, experimental runs with the grid and circle patterns (common geometries used in sector searches) show similar results for the static and dynamic cases.

3.3.5 - Static Pallet

This scenario, as the name implies, arises when a number of vehicles, in this case eight, cooperate to move a cargo pallet. It has been shown [19] that the forces detected via the pallet are sufficient to achieve cooperation. This case is presented as a comparison. Note that, ideally, the constraint of the pallet prevents any relative motion since the units remain fixed relative to one another. Also, in this case the vehicles are more closely spaced than in any other case considered in this chapter.
Figure 3.4 - Static Pallet. The vehicles are moving a 3x3 meter pallet.

The simulation results in Figure 3.4 are similar, except in scale, to those obtained for the static line. Note that the static line error of 0.5 meters for \( b = 10 \) has been reduced to just over 0.07 meters. This improvement of nearly an order of magnitude supports the intuitive notion that as the spacing of vehicles decreases, so should the estimation error. A large component of the error is due to bearing noise, which has less effect upon the Cartesian coordinate error components as the range decreases.

3.3.6 - Error Rejection vs. Accepted Scrambling

The load on the communications bandwidth is highly dependent on the protocol used to transmit the data. One question that must be answered is whether error detection and/or correction is necessary. The simulator implements models based on detection and rejection of errors versus acceptance of scrambled data to demonstrate the difference in performance of the two models. In the next figure, one hundred trials of rejection of 20\% of the measurements versus scrambling and using 5\% of the undetected erroneous measurements are shown for values of \( b = 10 \) and \( b = k \), using the static grid model.
As shown in Figure 3.5, acceptance of 5% scrambled data is a bad decision, regardless of the parameter $b$. On the same scale, the differences between $b = 10$ and $b = k$ for 20% rejection are barely distinguishable. Obviously, the key point is to detect and reject bad data. Note that the lack of 20% of the measurements had far less impact than a smaller percentage of accepted bad data. This discussion indicates that error detection is mandatory, although error correction is unnecessary, allowing a simpler implementation.

3.3.7 - Gaussian Error vs. Step Error

The final simulation demonstrates the differences encountered between a Gaussian Polar model and a Step Polar model. In the Step Polar model, all bearings are resolved to one of sixteen discrete values, and all ranges are resolved to 2.0 meter increments, after the addition of a small Gaussian noise component. One motivation for this model is an optimization of the communication load by packing all observation data into a single byte.
In addition, there exist published attempts to collect information by rings of fixed sensors and/or compasses [2][13].

Figure 3.6 - The measurement error, in polar coordinates, is assumed to be either Gaussian or truncated to discrete steps.

As shown in Figure 3.6, the estimates for the "pure" Gaussian model are roughly twice as accurate as the estimates generated by the fewer discrete values allowed by the Step Polar method. For some applications, the loss of precision may be acceptable, although in the general case the transmission of more high resolution observations is preferred, regardless of the selection of the $b$ parameter. All four of the variations shown in Figure 3.6 were run using the static circle model.
3.4 - Performance Models

In this section, performance models are developed that concern communications bandwidth, CPU consumption, and memory consumption. These models are helpful in identifying areas in which effort invested in improvement will be the most profitable.

3.4.1 - Communications Bandwidth

As always, cost is a primary design issue. As a result, communications bandwidth is the least extensible resource in the system, since extra bandwidth is usually increasingly expensive. Therefore, emphasis should be placed on methods that do not rely upon error correcting codes and/or retransmission. In one possible scenario, each unit transmits a packet of data consisting of a single byte for an identification header, one byte total for course and speed, and up to $n-1$, range dependent, five-byte observation subpackets. Each five-byte subpacket consists of a one-byte identification header plus a sixteen bit range and a sixteen bit bearing to each neighbor. Therefore, each unit transmits, in the worst case, $5n-3$ bytes each observation cycle. The total load on the network is then $n(5n-3)$ bytes per cycle.

Assuming an additional 25% overhead for framing and error detection, the system could support up to 18 units at 9600 baud with the arbitrary selection of one observation cycle per second. However, lack of direct measurement in many cases, as well as selection of a longer observation period, will increase the available capacity. Nonetheless, the load imposed upon the communication subsystem increases as approximately $5n^2$. Note that this derivation is dependent upon the ability of each unit to broadcast its measurements to the
entire group. Should the application require a retransmission capability to relay information, this would, of course, reduce the available capacity.

3.4.2 - CPU Consumption

The second most precious resource in the system, CPU consumption, increases at a horrific rate with moderately larger $n$. Our current model assumes that CPU consumption is directly proportional to the number of links examined. The following discussion presents a worst-case view, assuming that all possible paths to each node are examined. Database management techniques as well as the employment of floating point vector math supports this assumption. As path selection is essentially a study in combinatorics, the following derivation applies.

With $n$ units, there are $x = n - 2$ candidate nodes for selecting as links in the path from $H_O$ to $H_T$. Of these candidates, a unit may choose $i = d(P_{O,T,A}) - 1$ members of path $A$ for any length $d$, $i$ ranging from 0 to $x$. Of these $i$ members, there are $i!$ ways to order them, each unique ordering being a distinct path of length $1 + i$ in this context. This leads directly to the result:

$$\text{Execution time} \propto \sum_{i=0}^{x} \binom{x}{i} (i!)(i+1)$$

$$= \sum_{i=0}^{x} \frac{x!(i+1)i!}{(x-i)!i!}$$

$$= x! \sum_{i=0}^{x} \frac{i+1}{(x-i)!}$$

(3.5)

Consider the three largest terms in the above sum, $i = x, x - 1, \text{and } x - 2$:  

34
Execution time \( \propto x! \left[ \frac{x+1}{0!} + \frac{x}{1!} + \frac{x-1}{2!} \right] \)

\[ = x! \left[ \frac{5x+1}{2} \right] \approx 2.5x(x!) \quad (3.6) \]

As shown in (3.6), CPU consumption, in the worst case, increases as roughly \( 2.5x(x) \). In reality, missing observation links reduce the number of possible paths. In the current simulator, each unit only searches for paths up to 3 links longer than the shortest path found to any target node, further reducing the search space. Despite these optimizations, major gains in performance stand to be made, especially for large numbers of units, by improving the search algorithm, as will be discussed in later chapters.

3.4.3 - Memory Consumption

Memory is the cheapest and least utilized resource in the system. In the current model, each unit stores \( n \) sets of measurements, including the one set of direct measurements and the \( n - 1 \) sets received from the other units, to \( n - 1 \) neighbors, as well as \( n \) sets of course and speed information and \( n - 1 \) estimates. The actual storage is highly dependent upon the word length and format used, although the memory consumption clearly increases as \( n^2 \). Unlike communications bandwidth, however, additional memory is cheap.
3.5 - Interim Conclusions

3.5.1 - Progress Summary

The preceding results indicate that a system of mobile vehicles can, by sharing information, form accurate location estimates of one another despite a great deal of noise from a variety of sources. Based on these results, any application of these methods should be designed with the following factors in mind: First, while a great deal of sensor precision is unnecessary, bearing accuracy should be emphasized over range accuracy, and the sensors should be nearly identical for our model to apply. Second, the communication protocol should support error identification and rejection, as attempted filtering and use of corrupted data with the current model is a fatal error. Error correction or retransmission is unnecessary, as the next sweep of measurements provides the missing information, preventing the decrease in bandwidth caused by correction or retransmission schemes. Third, for many applications, values of the $b$ parameter of approximately 10 provide a robust response in many situations.

The performance of the system discussed herein is encouraging. The multitude of measurements available in the system provides the basis for precise estimates despite sensor and communication noise. It is expected that these results will apply equally well to navigation, a discipline that has traditionally relied upon exact measurements [21].
3.5.2 - Projected Improvements

While developing this method, a variety of opportunities for improvement emerged. One of the planned improvements, and one expected to yield the most benefits, is the optimization of the path search algorithm. As discussed earlier, the time consumed by a unit searching for a chain of observations to every other unit in the system is the performance bottleneck. A variety of methods exist to increase performance in this area, the most promising are those that trade memory consumption for speed.
Chapter 4 - Improving the Path Search with Order-Based Genetic Algorithms

As noted in the previous chapter, the path search problem consumes, by far, the greatest amount of computational resources in the system for systems of moderate size. This chapter describes a method by which memory is indeed traded for speed. This method revolves around the use of order-based genetic algorithms.

4.1 - Customizing Order-Based Genetic Algorithms

Chapter 2 introduced order-based genetic algorithms and compared them to traditional genetic algorithms. This chapter introduces a particular variant of the order-based genetic algorithms customized for this particular problem. The best way to describe this customization for the current problem is to discuss a typical application of order-based genetic algorithms, the traveling salesman problem, and then compare this problem to the current problem.

4.1.1 - Genetic Algorithms and the Traveling Salesman

The traveling salesman problem is a popular optimization problem [17]. Assume a salesman has to visit $N$ cities, in any order he pleases, during a business trip, returning to his point of origin at the end of the tour. His goal, since time is money and he detests driving, is to visit all $N$ cities only once while driving the minimum distance possible.

Literature abounds with proposed solutions to this seemingly simple problem, which has $N!$ possible solutions, as a bit of careful reflection will show [17]. For small
$N$, a brute-force exhaustive search of all possible solutions is appropriate. However, larger $N$ prohibit such exhaustive searches of the solution space. As a result, most approaches to this problem rely on some means of generating nonexhaustive lists of cities, or nodes, which are then compared with other lists for fitness. The problem rapidly begins to resemble the problem addressed in this thesis. The resemblances continue.

In some variants of this problem, it is assumed that the network of $N$ cities is completely connected, that is, a direct route exists from any city to all other cities [12][24], a physically unrealistic scenario in even the most fantastic Eisenhowerian vision. Telecommunication companies would also scoff at the idea of a completely connected network. This lack of complete connection complicates the problem, since many of the possible paths through the network will be invalid.

Although many similarities exist between the traveling salesman problem and the network path problem considered here, there are two major differences. First, the traveling salesman wants to visit all the cities and return home. In the problem considered here, subtour paths are sought to particular nodes. That is, it is not necessary to try to weave an optimal path through the entire network. The second difference is that the traveling salesman seeks the single optimal path. However, for the network problem, many redundant near-optimal paths are sought to each node. These differences require modification of normal order-based genetic algorithm operators.

4.1.2 - Some Order-Based Operators

Many attempts have been made to apply genetic algorithm techniques to the traveling salesman problem. Accordingly, new operators have been invented that attempt to intelligently produce new paths in an effort to optimize the distance traveled. The
interested reader is referred to references [12] and [24] for more detailed descriptions. In the paragraphs that follow, order-based operators will be dissected for application to the current problem.

The first order-based operators considered here were introduced at the 1985 International Conference on Genetic Algorithms and Their Applications and are summarized by Goldberg [10]. One such operator is the partially matched crossover (PMX) operator. Using Goldberg's example, consider the following parent paths, with crossover sites marked with 'I':

A:
9: 8: 4 | 5: 6: 7 | 1: 3: 2:10
B:
8: 7: 1 | 2: 3:10 | 9: 5: 4: 6

Now, create two offspring by swapping 5 and 2, 6 and 3, and 7 and 10, which are the elements found in the center section:

C:
9: 8: 4 | 2: 3:10 | 1: 6: 5: 7
D:
8:10: 1 | 5: 6: 7 | 9: 2: 4: 3

While this operator seems effective in experiments with the traveling salesman problem, it has the effect in the current problem of creating almost randomly generated paths, since only short subtours are of interest. Also, most of the manipulation performed by this operator affects the post-target region of the list, which is irrelevant for subtours. Goldberg also summarizes two other order-based crossover operators, order crossover (OX) and cycle crossover (CX) [10].

Like PMX, OX and CX tend to produce new paths that are essentially random when subtours are the goal. For example, using the same parents as above, order
crossover produces the following children by moving the swap regions to the front of the list and packing any remaining nodes at the end:

\[
\begin{array}{c}
\text{C:} & 5:6:7 & 2:3:10 & 1:9:8:4 \\
\text{D:} & 2:3:10 & 5:6:7 & 9:4:8:1 \\
\end{array}
\]

Cycle crossover produces children by a complicated selection process that, for the current application, usually scrambles the pre-target subtour. As a result, it will not be considered here.

To reiterate, PMX, OX and CX are useful operators when considering a complete tour of nodes. Elements of PMX and OX were used to craft the crossover operator described later. However, for the current problem, operators that optimize subtours are required for peak performance.

Whitley, et. al., recognized the inadequacy of the PMX, OX, and CX operators when considering incompletely connected networks, since the previously discussed operators manipulate nodes rather than connections, or edges. The key issue is discussed concisely in the following quote from [24]:

\[
\ldots \text{an ideal operator should construct an offspring tour by exclusively using links present in the two parent structures} \ldots \text{Thus, an operator that preserves edges will exploit a maximal amount of information from the parent structures. Operators that break links introduce unwanted mutation. This mutation can be thought of as a kind of 'leak' in the search process.}
\]
To prevent such mutations, Whitley and his colleagues created the edge recombination operator [24]. Use of this operator requires that the problem be encoded as a database of edges. Modifying Whitley's example, consider a network of six nodes $A$ through $F$ with the following set of connections $\{ab, ac, ad, ae, bc, bd, bf, cd, ce, cf, df, ef\}$, where a pair of lowercase letters represent a bi-directional connection between the corresponding uppercase nodes. Arranging the connections in a table, the tour $ABCDFE$ can be represented as:

<table>
<thead>
<tr>
<th></th>
<th>ab</th>
<th>ac</th>
<th>ad</th>
<th>ae</th>
<th>be</th>
<th>bd</th>
<th>bf</th>
<th>cd</th>
<th>ce</th>
<th>cf</th>
<th>df</th>
<th>ef</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that any valid tour contains a binary string with $N$ ones, with each node represented exactly twice in the list of connections. Unfortunately, in all cases with a completely connected network, and for some incompletely connected networks, these characteristics are not sufficient to establish a valid tour, a fact omitted in reference [24].

For this example, with a completely connected network, the edge set $\{ab, ac, bc, de, df, ef\}$ would satisfy the above criteria, yet represent two unconnected subtours, an invalid situation for the traveling salesman problem. However, this representation forms the basis for the theoretically useful edge recombination operator.

The edge recombination operator uses the above binary representation to form offspring by choosing one bit from each parent, as demonstrated below with parents $A$ and $B$, producing child $C$:

<table>
<thead>
<tr>
<th></th>
<th>ab</th>
<th>ac</th>
<th>ad</th>
<th>ae</th>
<th>bc</th>
<th>bd</th>
<th>bf</th>
<th>cd</th>
<th>ce</th>
<th>cf</th>
<th>df</th>
<th>ef</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B:</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C:</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
However, a bit of experimenting with the above example will show that the above approach produces mostly invalid results if the binary strings are recombined blindly. Although this approach has implementation problems, the concept of manipulating edges is important, since edges, or links, are the building blocks of the desired solution. As noted by Whitley, *et. al.*, edge recombination, when considered as binary strings, represents schema manipulation, a key theoretical ingredient of genetic algorithms [24].

Another order-based operator considered is Yuval Davidor's analogous crossover operator, formulated in the context of robot trajectory generation [7]. However, Davidor also incorporated knowledge from scheduling production processes, which is similar to robot trajectory generation, except that some production processes can proceed in parallel where manipulator trajectories are purely serial processes [7].

Assume two possible manipulator trajectories are considered, as shown in Figure 4.1.

![Figure 4.1](image)

*Figure 4.1* - An example of two separate paths crossing at two nodes, 4 and 9. These nodes are ideal locations for application of the analogous crossover.

As shown in Figure 4.1, two paths $A$ and $B$ are considered which cross at nodes 4 and 9. The analogous crossover operator generates new paths by swapping segments defined by the crossing points. The paths $A$ and $B$ are listed below, along with all possible offspring paths $C$, produced by the analogous crossover operator:
The analogous crossover operator seems directly applicable to the current case. One major advantage of this method is that all offspring are guaranteed to be valid paths, since all their component links exist in the parents. As a result, the analogous crossover operator was adapted to form two new operators, the outside cross matching operator and the inside cross matching operator, discussed later in this chapter.

The last order-based operators of interest are the subtour-swap operator and the subtour-chunk operators discussed by Cleveland and Smith and created by Grefenstette. The interested reader is directed to reference [6] for details. As with most of the operators discussed thus far, these operators do not directly apply to the current problem since most of their manipulation affects post-target nodes. Despite the prefix, these operators modify complete tours. As a result, the children produced by these operators have almost random pre-target subtours.

4.1.3 - Selection and Replacement

Chapter 2 discussed selection and replacement along with their associated hazards. In this section, those selection and replacement techniques are modified for use throughout this thesis. As mentioned earlier, the major risk in the roulette wheel selection technique is that it is possible, although not likely, that the new population will contain members that are the offspring of the worst member of the population. To avoid this risk in the current application, the top thirty members of the population always
reproduce, since multiple redundant solutions are sought. This goal is a departure from traditional genetic algorithm applications, where a single best solution is sought.

As another twist, the operators are applied in a batch fashion, where the entire population is reproduced by the first operator enabled, then the next, and so on until the entire set of operators has been applied. While seemingly inefficient, this method removes any difficulties associated with parameter tuning, such as the rate of operator selection. In addition, efficiency is not a major issue for two reasons: 1) integer string manipulation is faster than floating-point arithmetic and 2) as will be seen in Chapter 7, proper selection of operator suites will provide exceptional performance under these conditions. This paradigm is another departure from traditional genetic algorithms, where the mutation operator, while important, is best used sparingly [10][18]. Since the ultimate goal is accurate estimates, effort invested in searching for a robust population of paths pays accuracy dividends later.

Another issue concerning reproduction is the method of replacement. The popular technique known as generational replacement was discussed [8]. As a review, generational replacement replaces the entire population every generation with the new generation. Unfortunately, this technique is also risky. Again, although not likely, it is possible that an extremely effective solution is lost by producing inferior offspring and then replacing the entire generation by the offspring. To avoid this problem, a superset population is created by temporarily combining the new members with their offspring. Then, this superset population is sorted by fitness and the best members selected [8]. Using this technique, the population is guaranteed to not "devolve" under constant problem conditions. Generational replacement has no such guarantees.
4.2 - Glossary of Order-Based Genetic Algorithm Operators

This section discusses, in some detail, the various methods available in the system for producing new paths from the current population, along with the strengths and weaknesses of each technique. Although this section is a primer on the mechanics of each technique, the actual implementation is detailed where appropriate. The eight operators that follow are adaptations of various techniques.

4.2.1 - Direct Search

The direct search operator forms new paths by two means. First, the database is searched for a direct connection between the base node and the current target node. If this direct measurement exists, a new path is created. This path contains the current target node header in the first position, with all remaining node headers following in no particular order, to maintain consistency of string length. This portion of the reproduction cycle can only produce one offspring. To make maximum use of the remainder of the reproduction cycle, the direct search operator also searches the database for all paths of length 2. That is, all possible (n-2) paths consisting of one intervening node are tested.

In most cases, the direct search operator, even with the second phase of searching for all paths with one intervening node, will still not produce a full population of new offspring. To correct this deficiency, the concept could be extended to searching for all length 3 paths, etc. However, this approach begins to resemble a combinatorial search, which is surpassed in efficiency by other methods, most notably outside cross matching, described later. The combinatorial search method rapidly loses efficiency because most
of the paths considered are unviable. As the direct search operator requires no parents, it is well suited for initializing, or seeding, a population, although it does not provide a great deal of diversity, nor does it find paths to many nodes in most realistic scenarios.

4.2.2 - Random Search

The random search operator, as the name suggests, simply creates randomly ordered node lists. This operator does not require parents. Although it is useful in many genetic algorithm applications for infusing a population with new “ideas” that may have been removed from the population over the generations, it is not very useful in this context. In the current problem, the network is very loosely connected. Therefore, like a raw combinatorial search, most node lists produced by this operator will be unviable. As such, it is a very inefficient search method, and should be used sparingly. It is included primarily for comparison purposes, since it resembles the traditional genetic algorithm mutation operator.

4.2.3 - Outside Cross Matching

The outside cross matching operator is probably the most powerful operator in the suite. This operator exploits progress made in finding paths to other nodes, allowing discoveries to rapidly propagate throughout the system. The basic concept follows. Assume a path is desired to node \( A \). By searching the database for all measurements to node \( A \), the base node discovers a measurement to node \( A \) by node \( B \). If a path exists to node \( B \), then a path can be created to node \( A \) by copying the path to node \( B \) and appending the measurement \( BA \). This operator was initially developed for this thesis as a
curiosity, but rapidly became respected for its utility for the current problem, despite the simplicity of the concept.

While most of the operators consider each population of paths to each individual node to be separate "communities", this operator effectively combines paths from several communities. In a sense, this operator, an adaptation of dynamic programming [3] as well as analogous crossover, is a directed combinatorial search, since a large portion of unviable paths are never considered. One drawback of this operator is that it cannot unilaterally create paths. However, when combined with the direct search operator, the outside cross matching operator will find the shortest path to any node, provided the path exists and the database is static.

Consider the following example:

![Diagram](image)

**Figure 4.2** - An example of the outside cross matching operator. A path already exists to node 4. A path to the target T is created by appending the link 4T.

Assume we have, in our population to node 4, the path (S,1,2,3,4), but we are looking for a path to node T. If node 4 can "see" node T, we can readily form the path (S,1,2,3,4,T). It is easy to see how this operator can rapidly "grow" paths to other nodes. In fact, this operator, combined with the direct search operator, will find a path, if it exists, to any node in less than (N-1) reproduction cycles, where N is the number of units in the system. This guarantee is supported by considering the fact that any path contains,
at most, \((N-1)\) nodes. If this maximum length path exists, a path to the first node, and possibly a path to the second node, will be found in the first cycle by the direct search operator. Regardless, a path to the second node will be found during the second cycle by the outside cross matching operator. This growth of paths proceeds until the path to the last node is found in the \((N-1)th\) reproduction cycle.

In most situations, the outlook is even better. Except in very unusual circumstances, where the nodes are dispersed at maximum ranges along a line, for example, paths much shorter than \((N-1)\), if any, will exist to any node. As a result, we can expect to find paths with this operator in very few reproduction cycles.

In situations in which the measurement database is static, the outside cross matching operator resembles Dijkstra's minimum-path algorithm [9]. This popular path-search technique is applied in parallel to all populations at once, rather than recalculated at each iteration. With a static database, the outside cross matching operator produces a minimum-path spanning arborescence, rooted at the source node.

4.2.4 - Inside Cross Matching

The **inside cross matching** operator is a variation on the outside cross matching operator. This operator also exploits progress made in finding paths to other nodes, again allowing discoveries to rapidly propagate throughout the system. The basic concept follows. Assume a path exists to node \(A\). Choose an intervening node, node \(B\), for example, along this path. Let the length of this subpath from the source node to node \(B\) be \(L_{BA}\). Now, examine the best path in the population of paths to node \(B\). If this path is shorter than \(L_{BA}\), replace the subpath to node \(A\) via node \(B\) with the best path to node \(B\). This new path is now placed into the population to node \(A\).
A concrete example of the inside cross matching operator follows.

Old Path to node 9: 0:1:5:7:8:3:4:9:2:6
Best Path to node 8: 0:6:8:2:1:4:3:5:7:9

Note the preservation of the complete set of nodes. At first glance, this operator seems interesting, but not very powerful. With a static database, this is certainly the case. However, consider the diagram below:

Figure 4.3 - An example of the inside cross matching operator. The path to node $T$ is shortened by bridging nodes 1 and 4.

Assume that, initially, the measurement (1,4) is invalid. Perhaps a tree is in the way. However, the path represented by the solid line is valid, and is the best path in the database to node $T$. Now, for whatever reason, the measurement (1,4) arrives in the database. If only the outside cross matching operator is available, the optimized path $(S,1,4)$ would be found to node 4 in the next reproduction cycle, the path $(S,1,4,5)$ would be found to node 5 in the second reproduction cycle, and so forth until the path $(S,1,4,5,6,7,T)$ is found in the fifth reproduction cycle. However, assume inside cross matching is available. Now, in the second reproduction cycle, the subpath $(S,1,2,3,4)$ in the path $(S,1,2,3,4,5,6,7,T)$ would be replaced by $(S,1,4)$ forming $(S,1,4,5,6,7,T)$, the same result as before, only in two cycles instead of five, a significant savings.
As demonstrated above, the inside cross matching operator is a good complement to the outside cross matching operator, providing the flexibility needed in some dynamic situations. While we can count on the outside cross matching operator to find good paths eventually, the inside cross matching operator can rapidly exploit gains.

The question now arises, how do we choose which intervening element to examine. Since we want to create thirty or so offspring each cycle, and the paths are generally much shorter than thirty nodes, a good method is to work backwards from the next-to-last node, examining all but the first node. This method means that longer subpaths will be replaced first, increasing the chance of producing better paths if the path is longer than the number of offspring available.

4.2.5 - Crossover

The crossover operator, the only purely binary technique considered, forms pairs of offspring from pairs of parent paths, attempting to combine the best features of both parents in at least one of the offspring. Each path contains the same number of distinct elements, each representing a system node, and all nodes are represented on each path with no repetitions. Offspring produced by the crossover must maintain this representation. Each parent path contains small, high quality subtours that must be preserved during the crossover while scrambling of paths must be kept to a minimum. In particular, the leading elements of the paths should be preserved exactly, when possible. The current discussion assumes that the crossover point \( p \) and length \( l \) are predetermined for paths of length \( n \).
Modified Crossover Algorithm Psuedocode

I. Reserve storage for two offspring $C_1$ and $C_2$ of length $(n+l)$

II. Copy from $P_1[0:p-1]$ to $C_1[0:p-1]$ and
    from $P_2[0:p-1]$ to $C_2[0:p-1]$
    
    Copy from $P_1[p:p+l-1]$ to $C_2[p:p+l-1]$ and
    from $P_2[p:p+l-1]$ to $C_1[p:p+l-1]$
    
    Copy from $P_1[p+l:n-1]$ to $C_1[p+l:n-1]$ and
    from $P_2[p+l:n-1]$ to $C_2[p+l:n-1]$
    
    Copy from $P_1[p:p+l-1]$ to $C_2[n:n+l-1]$ and
    from $P_2[p:p+l-1]$ to $C_1[n:n+l-1]$

III. Scan $C_1$ and $C_2$ to remove repeated nodes, packing to the left.

IV. Return $C_1$ and $C_2$ as the result.

Consider the following diagram.

Figure 4.4 - An example of the modified crossover operator. List consistency is maintained by guaranteeing no duplicate or missing nodes.
To enhance understanding of this technique, the following is presented as an example of the crossover algorithm:


II. C1: 0:9:5:4:8:3:2:4:3:7:6:8:1

III. C1: 0:9:5:4:8:3:2:7:6:1

The advantage of crossover is that small effective subpaths from two proven paths can be combined. Disadvantages of this method include the time consumed by removing redundancies, as well as the extra memory consumed by the extra end elements. A further disadvantage of crossover is that if both parents are similar, although not exact copies, the offspring may be exact copies of the parents. This technique is most useful in situations where deep searches are necessary to find valid paths, as in extended patterns such as linear or ring patterns. Crossover may also be useful for tracking a moving target through the population, although the cross matching operators look very promising in this regard.

4.2.6 - Linear Inversion

The **linear inversion** operator, a unary reordering technique, forms a single offspring for each parent considered. Start and end points for the inversion are selected, with the end point greater than the start point. The offspring is identical to the parent except that the nodes in the inversion zone are reversed.
The following simple example illustrates the technique:

\[
\begin{align*}
P &: 0:9:5:6:8:1:2:4:3:7 \\
\end{align*}
\]

One advantage of linear inversion is that it is fast. Another advantage is that it, in most cases, allows searches of a broad field, by reordering near nodes, while maintaining a short valid stub, which may be necessary if the source cannot directly observe many other units. Should the target be near the center of the inversion zone, this technique rotates in fresh nodes from the outer zone. A disadvantage of linear inversion is that it loses its utility when many very short valid paths exist, as inverting nodes beyond the target serves no useful purpose. However, in these situations the combinatorial search technique will yield excellent results.

4.2.7 - Circular Inversion

The **circular inversion** operator, another unary reordering technique, forms a single offspring for each parent considered, similar to the linear inversion technique discussed above. To perform circular inversion, start and end points for the inversion are selected. If the end point is greater than the start point, the technique is identical to the linear inversion. However, if the end point is less than the start point, the path is
considered to be formed in a circle. Now, the inversion can extend past the ends of the path. The offspring is identical to the parent except that the nodes in the inversion zone are reversed.

![Figure 4.6](image)

**Figure 4.6** - An example of the circular inversion operator. The path is treated as a closed loop, with an arc inverted.

The following simple example illustrates the technique:

\[
\begin{align*}
P &: \ 0:9:5:6:8:1:2:4:3:7 \\
C &: \ 7:3:4:6:8:1:2:5:9:0
\end{align*}
\]

Circular inversion shares many of the advantages and disadvantages of linear inversion. In fact, linear inversion is merely a special case of circular inversion. However, an advantage of circular inversion is that, for very long paths, it is possible to rotate in nodes from deep in the list. Another advantage of circular inversion is that it is likely to replace an overused near stub. This replacement may actually be a disadvantage in special cases.

### 4.2.8 - Randomized Reduction

The **randomized reduction** operator attempts to shorten valid paths by randomly removing nodes from the path prior to the target, replacing them at the end of the path to maintain consistency. This method is an attempt to eliminate middlemen, the idea being
that two nodes separated by only one node may be able to observe one another directly. The concept also applies to removing short segments of nodes, which may be fruitful in broad fields.

![Figure 4.7 - An example of the randomized reduction operator. A randomly selected section of the path prior to the target is relocated to the end of the list, shortening the pre-target subtour.](image)

The following example assumes node 8 is the target node:

\[
\begin{align*}
P & : 0 : 9 : 5 : 6 : 8 : 1 : 2 : 4 : 3 : 7 \\
C & : 0 : 5 : 6 : 8 : 1 : 2 : 4 : 3 : 7 : 9
\end{align*}
\]

The greatest advantage of randomized reduction is that it is the method most likely to improve broad field paths, \textit{i.e.} those that weave back and forth in a dense pattern of units. This method completely fails only in rare extended patterns. But, in such situations, many of the methods discussed also fail.
4.3 - Relevant Concepts from Simulated Annealing

No discussion of order-based path searching would be complete without a discussion of simulated annealing. As the name suggests, simulated annealing is an analogy to the metallurgical process of annealing to optimization problems [17].

The metallurgical process of annealing strengthens the crystalline structure of a substance by slowly cooling a heated mass, allowing the molecules to settle into low energy states. Similarly, simulated annealing is an attempt to mimic this natural process by making successively smaller perturbations to a proposed solution, testing the resulting solutions for improved fitness.

The application of simulated annealing to the path searching problem is centered around the idea of perturbing the population in decreasing amounts, allowing the path to settle, or relax, into a low energy state, where “energy” in this sense is defined by the path fitness function. As with order-based genetic algorithms, simulated annealing is often demonstrated with use of the traveling salesman problem [17].

Only a few operators, crossover, inversion, and reduction, as defined here, allow the use of simulated annealing concepts. As the population matures, progressively smaller sections of parent paths are chosen, regardless of the operator used. As an illustration, consider the randomized reduction operator. If the population is relatively young, simulated annealing principles suggest that one attempt to remove relatively large path subsections, i.e. four or five nodes, while searching for better paths. On the other hand, when the population is more mature, one should remove single nodes while searching for better paths.
Similar concepts apply for the crossover and inversion operators. In the current problem, the subtours are usually so short that directly controlling the size of the perturbation is difficult. To avoid this difficulty, the size of the perturbation zone is randomly selected for each application of an operator. So, the larger perturbations are more effective initially, while the smaller are more effective later.
Chapter 5 - Estimation Techniques

The previous chapter presented, in great detail, the genetic algorithms used to find multiple redundant paths to each node in the system. This chapter describes how those paths are combined with the database of measurements to produce location estimates.

5.1 - An Estimator Hierarchy

Keeping true to the concept of modularity, the location estimates are obtained by a hierarchy of estimators. First, the raw sensor data from each node is filtered within the node to obtain measurements that are communicated over the broadcast net. Then, these measurements are combined with the current paths to form observations. Finally, these observations are combined over time to form location estimates. Implementation details for these techniques are given in Section 6.4, Estimator Implementation.

5.1.1 - Measurement Estimation

The measurements are prefiltered, as polar quantities, at each unit before being broadcast to the population. This approach will be justified in Section 5.2, The Case for Prefiltering. In other words, each node filters the range and bearing measurements to each node separately with a simple low-pass filter. The filtered quantities could be combined to form Cartesian coordinates local to each node, using simple trigonometry, and then broadcast throughout the system. The primary advantage to this approach is that it reduces bias in the measurements, which tends to “contract” the location estimates toward the observer if the raw measurements are used in later stages. Section 5.2
discusses this phenomenon in detail. Prefiltering on-board each unit prior to transmission also distributes the processing load, gaining a degree of parallelism.

5.1.2 - Composite Observation Estimation

Once valid paths are found to any given node, the prefiltered measurements in the database are used as links in those paths, resulting in a vector sum of the measurements terminating somewhere in the vicinity of the node in question. Let this termination point, or vector sum, be defined as an *observation*. In most cases, the population database will contain many valid paths to any node, with an observation associated with each path. Combining these multiple observations into a single quantity, defined as a *composite observation*, provides an opportunity to reduce the error inherent in any single observation. Four well-known estimation methods, a mean estimator, a median estimator, a mode-mean estimator, and a mode-median estimator, are considered for computing composite observations from a population of observations. The following paragraphs, as well as Section 6.4, discuss the particular implementation of these common techniques.

The mean estimator is simply an adaptation of the common weighted mean. A median estimator, as the name suggests, produces an estimate by selecting the middle quantity in a sorted list of quantities. This method is based on the idea that the true value should be near the center of the group of noisy values, which is the expected result if the noise is zero mean. A weighted variation of this approach is to sort the quantities, *x* coordinates, for example, and then choose the quantity having half the total weight to either side. The weighted version is used in the simulator, where the weights are the inverse fitness values of the respective paths.
The two remaining estimators, the mode-mean and the mode-median, form estimates by trying to find the densest grouping of measured noisy values. Before discussing each in detail, it is necessary to discuss a quantity known as the shorth, or shortest-half, a central concept in mode-estimators. Given a sorted list of $N$ quantities, $x$ coordinates, for example, find the group of $D$ quantities that minimizes the distance between the smallest and largest quantities in the group, where $D = 1 + \text{integer}(N/2)$. This group is then the shorth of the list of quantities. An example is in order. Consider the ordered list of 11 numbers below:

\[
\{ 1.1, 1.2, 1.4, 1.6, 1.6, 1.7, 1.7, 1.8, 2.1, 2.5, 3.7 \}
\]

Since $N = 11$, $D = 1 + \text{integer}(11/2) = 6$. We must now examine the list and compare “lengths” of groups of 6 numbers. For clarity, define this length

\[
d((x_0, x_1, \cdots, x_6)) = x_n - x_0
\]

Then,

\[
d(\{1.1, 1.2, 1.4, 1.6, 1.6, 1.7\}) = 0.6, \\
d(\{1.2, 1.4, 1.6, 1.6, 1.7, 1.7\}) = 0.5, \\
d(\{1.4, 1.6, 1.6, 1.7, 1.7, 1.8\}) = 0.4, \\
d(\{1.6, 1.6, 1.7, 1.7, 1.8, 2.1\}) = 0.5, \\
d(\{1.6, 1.7, 1.7, 1.8, 2.1, 2.5\}) = 0.9, \\
d(\{1.7, 1.7, 1.8, 2.1, 2.5, 3.7\}) = 2.0
\]

In this example, the set \{1.4, 1.6, 1.6, 1.7, 1.7, 1.8\} forms the shorth. In rare cases, two groups will have the same length. This case represents a multimodal distribution of values. In such cases, especially when the underlying distribution is known to be unimodal, it is best to accept a “supergroup” consisting of all elements lying
between the minimum of the leftmost group and the maximum of the rightmost group. While not technically fulfilling the definition of the shorth, this approach does fulfill the intent, which is to find the densest group of values. Another approach, when the underlying distribution is not known to be unimodal, is to break the tie by considering the two $D-1$ subgroups in each group, accepting the smallest of those four groups. The "supergroup" approach is used in the simulator.

Now that the shorth is defined, the mode-mean and mode-median estimators may be discussed. A mode-mean estimator calculates either the simple mean or a weighted mean of the elements in the shorth. A weighted average is used in the simulator. As with the weighted median estimator described earlier, the weights are simply the inverse fitness values of the respective paths. A mode-median estimator calculates the median, either direct or weighted, of the shorth. Again, a weighted mode-median estimator is used in the simulator.

5.1.3 - Location Estimation

The location estimator, the third level in the estimator hierarchy, finally forms the location estimate based on the last location estimate and the latest composite observation. The measurement estimator, the prefilter described earlier, is designed to reduce bias in the location estimates. The composite observation estimator, described in the previous section, attempts to locate the densest group of observations. The location estimator updates the location estimate with a simple low-pass filter, similar to the method discussed in Chapter 3. Implementation details are given in Section 6.4, Estimator Implementation.
5.2 - The Case for Prefiltering

As discussed in the previous section, the raw sensor data is prefiltered on-board each node prior to dissemination. This section justifies that approach by first presenting a derivation of the polar probability density function, and then computing the expected value of the estimators, revealing an unsavory bias.

5.2.1 - Derivation of Polar Probability Density Function

Consider the drawing below, centered at a source node and oriented arbitrarily:

![Diagram of a simple trigonometry example](image)

*Figure 5.1 - Simple trigonometry example.*

It is clear, from simple trigonometry, that the location of the node at point P may be described equally well in Cartesian coordinates or in polar coordinates. Also, note that the Cartesian coordinates may be easily transformed by selection of appropriate bases such that the $y$ coordinate is zero and the $x$ coordinate is equal to $r$. 
Periodically, the nodes make noisy polar measurements of one another. That is, measurements are composed of pairs of independent random variables, \( R \) and \( \Theta \). The assumption of independence, while not proven, is nonetheless reasonable, since a bearing measurement is probably independent of the corresponding range measurement. Further, assume the polar random variables are each Gaussian random variables with \( \mu_R = r_0 \), \( \mu_\Theta = 0 \) and variances \( \sigma_R^2 \) and \( \sigma_\Theta^2 \). Now, we wish to find the joint probability density function \( f_{XY}(x,y) \), from which we may determine whether the measurements are biased. The following derivation is based on McGillem and Cooper [5]. Recall that

\[
f_{XY}(x,y) = \frac{f_{R\Theta}(r,\theta)_{r=g_1(x,y),\theta=g_2(x,y)}}{|J|} \tag{5.2}
\]

where \( J \) is the Jacobian of the coordinate transformation and is found by

\[
J = \begin{bmatrix}
\frac{\partial x}{\partial r} & \frac{\partial x}{\partial \theta} \\
\frac{\partial y}{\partial r} & \frac{\partial y}{\partial \theta}
\end{bmatrix} = \begin{bmatrix}
\frac{\partial (r \cos \theta)}{\partial r} & \frac{\partial (r \cos \theta)}{\partial \theta} \\
\frac{\partial (r \sin \theta)}{\partial r} & \frac{\partial (r \sin \theta)}{\partial \theta}
\end{bmatrix}
\tag{5.3}
\]

\[
= \begin{bmatrix}
\cos \theta & -r \sin \theta \\
\sin \theta & r \cos \theta
\end{bmatrix}
\]

So,

\[
|J| = r \cos^2 \theta + r \sin^2 \theta = r \tag{5.4}
\]

Now, since the polar random variables are independent,
Recognizing that \( r = \sqrt{x^2 + y^2} \) and \( \theta = \arctan\left(\frac{y}{x}\right) \), the joint probability density function becomes:

\[
f_{XY}(x, y) = \frac{1}{r} f_R(r) g_1(x, y) f_\Theta(\theta) g_2(x, y) = \frac{1}{r} \left( \frac{1}{\sqrt{2\pi\sigma_R^2}} e^{-\frac{(r-\mu_R)^2}{2\sigma_R^2}} \right) \left( \frac{1}{\sqrt{2\pi\sigma_\Theta^2}} e^{-\frac{\theta^2}{2\sigma_\Theta^2}} \right)
\]

\[\text{(5.5)}\]

At this point we may be tempted to apply the familiar substitution \( x = y \tan(\theta) \) and attempt to reduce this expression to a more palatable form. However, that route leads to an equally ugly expression that is just as difficult to integrate. Fortunately, the above equation yields its secrets readily enough, when subjected to numerical integration [17].
5.2.2 - Expected Value and Bias

Now that we have an expression for the joint density function \( f_{XY}(x,y) \), we may uncover a few facts about the noisy measurements. The graph in Figure 5.2 below plots \( E[X] \) and \( E[Y] \) for several values of \( \sigma_\theta^2, \sigma_R^2 = 1 \) and \( \mu_R = 20 \).

![Figure 5.2 - Plot of the means of X and Y versus the variance of \( \theta \) when converted from polar to rectangular coordinates.](image)

Note that \( E[Y] \) is zero for all \( \sigma_\theta^2 \), while \( E[X] \) decreases almost linearly from the mean 20 with increasing \( \sigma_\theta^2 \).

As a sample of the shape of the joint probability density function, consider the plot in Figure 5.3 on the next page, generated for \( \sigma_\theta^2 = 0.20 \), or a standard deviation of roughly 25 degrees, \( \sigma_R^2 = 1 \) and \( \mu_R = 20 \).
Figure 5.3 - 3D plot of transformed polar probability density function.

Note that the plot in Figure 5.3 resembles a two-dimensional Gaussian distribution warped around the origin, which is exactly what one would expect. With the plot above in mind, it is easy to see that the expected value of $y$ should be zero, while the expected value of $x$ is somewhat less than the mean of the radius. The degree of disparity between the mean of $x$ and the mean of the radius depends on the amount of weight in the curved arms of the distribution, that depends in turn on the variance of $\theta$. 
The following plot shows 200 samples of measurements with the same statistics as before:

![Plot of 200 sample measurements.](image)

**Figure 5.4** - Plot of 200 sample measurements.

The above plot shows that while the distribution about $y$ is roughly symmetric, the $x$ distribution about the mean, 20, is clearly biased toward the origin.
Consider the selection of basis vectors discussed earlier. The assumption of the zero mean, or expected value, of the bearing measurement implies that the expected value of $y = r \sin(\theta)$ would be zero. However, the nodes will be taking measurements to other nodes in several directions at once. Of course, this implies that a portion of the bias will be distributed into $x$ and $y$ components, which we may consider a small error vector pointing back toward the source from the target. Since the location estimate to each node will, in general, be based on the vector sum of a series of measurements, we can consider the overall bias to be, for our purposes, the vector sum of the individual bias vectors. The component of this overall bias vector along the direct line from source to target will certainly be nonzero, while the components of the individual biases, and thus the overall bias, orthogonal to the direct line will tend to cancel. The following figure illustrates this situation:

![Figure 5.5 - Example of combining biases.](image)

Note that the overall bias is generally, although not perfectly, in the direction of the source from the target. The effect of this bias is that all the location estimates in the
system will tend to be contracted toward the source, and the degree of contraction is nearly linear with regard to the variance of the bearing errors. The vector sum of the biases, which depends on the variance of the bearing error, represents, as an approximation, the variance of the overall measurement, since the variance of a sum is simply the sum of the variances.

This discussion leads to the conclusion that the overall estimates will have an unavoidable bias toward the source, and that this bias is almost linearly related to the variance in the bearing measurements. Any method, \textit{i.e.} prefiltering, that can reduce the bearing measurement variance, will also reduce the bias in the overall estimates. Furthermore, the bias could be estimated, and thus reduced, if the variance of the bearing measurement could be made available to the estimator. Kalman filtering of the raw bearing measurement data, as one approach, could accomplish both of these objectives. However, since the communication bandwidth is assumed to be limited, transmission of the variance is impractical, and since the computational power is also assumed to be limited, it is equally impractical that each unit should calculate the variances after receiving the raw measurement data.

The obvious compromise, under these circumstances, is to prefilter the raw measurement data on board the sensing node, and then transmit only the most recent estimate of the true bearing and range, without any other statistics. Experimentation has shown that this method is effective, even with a simple low-pass filter. Another approach, not implemented here, would be to add a small offset, proportional to the calculated bearing variance, to the estimated range, and then transmit this quantity. However, for this method to be effective, one must be certain that the bearing measurement itself is unbiased.
Chapter 6 - Implementation and Algorithms

Now that the theoretical groundwork is in place, the concepts can be tested. As no suitable hardware network is available for study, a simulator was constructed. In the simulator, each functional area is maintained in separate modules, facilitating maintenance and portability.

6.1 - Simulator Architecture

Before discussing the implementation of the system, a brief overview of the simulator architecture is in order.

6.1.1 - Operating System and Tools

The simulator was developed to run under the Windows 3.1™ operating system, primarily because the event-driven nature of the system is preserved under Windows [4]. Other considerations, such as the interactive user interface and simplicity of command automation served secondary, but important goals. The C programming language was chosen over C++ because of a concern for code and data structure size, since the final product is to be programmed into embedded controllers. Most of the well-publicized features of C++, such as data encapsulation and abstraction, are readily emulated in C. Techniques to accomplish this degree of sophistication were adapted primarily from McConnell [16]. Some routines for the user interface were adapted from Conger [4].
6.1.2 - Error Models

In general, the simulator is designed to emulate as many real-world difficulties as possible, including occlusion, communication error, and measurement error. Where possible, the simulator exceeds expected levels of noise and other errors [1][2]. This design is intended to not only validate the processes used, but also increase the probability of successful operation in more benign environments. The following paragraphs discuss the simulated errors in more detail.

The first source of error considered is measurement error. All sensors are noisy, and the range and bearing sensors on our vehicles are no exception. A Gaussian noise model is considered. Three noise intensity models are available, detailed in Chapter 7. The maximum effective range of the sensors is assumed to be 20.0 meters [1].

The next source of error illustrated in the simulator is communication error. It is presumed that a certain percentage of transmissions will become garbled, for any of a number of reasons. However, many of these garbled transmissions can be detected by error-checking algorithms such as Cyclic Redundancy Check (CRC) [17]. A design goal of the system is that the estimator will function correctly with an incomplete set of measurements. Therefore, the communication error is simulated by randomly discarding a certain percentage of measurements. The user may choose to discard a percentage of measurements from the following options: 5%, 20% or 40%. These percentages are far greater than those expected in reality.

The final source of error considered in the simulator is occlusion. Occlusion occurs when one unit blocks measurements between a pair of units. This form of simulated error allows accurate representation of a dynamic scenario. For plotting purposes, the units as displayed are twice the size indicated by the display scale. Even so,
this exaggerated size is used for determining the shadow zone, again providing a comfortable operating margin.

6.1.3 - Consideration of Embedded Application

The ultimate goal of this work is to port the refined code to embedded controllers in actual working models. This goal places several constraints on the software design. First, the C programming language, arguably the most common high level language supported by compilers targeted for embedded controllers, increases the degree of flexibility in choosing hardware. Next, the software was designed in functional modules, keeping the operating system dependent code and the algorithmic code independent where possible, simplifying porting. Finally, the entire system is based on the concept that the various measurements will arrive in no particular order. This concept supports interrupt-driven operation, typical of many embedded control applications.
6.2 - System Design

The previous section briefly discussed the background requirements for porting to embedded controllers. The following sections expand those ideas in more detail.

6.2.1 - Asynchronous System Operation

As there will be no global time base, the entire system must operate asynchronously. Units will be broadcasting measurements via a global broadcast net. The solution detailed herein is from the perspective of only one unit in the system. However, this process is identical on every unit in the system.

Before discussing the method used to integrate the data into robust estimates, it is necessary to first discuss the on-board free-running timer. This timer, an integer-valued variable incremented periodically, does not necessarily correspond to the current time in the traditional sense. Instead, its sole purpose is to provide an internal frame of reference for coordinating various activities. Each unit maintains its own free-running timer, but, for our purposes, it is not necessary that they be coordinated.

Several options exist for constructing a free-running timer. One example is the on-chip capability by that same name found on the Motorola MC68HC11 family of microcontrollers. Another example is an interrupt triggered periodically by a traditional microcontroller timer. The interrupt service routine would simply increment a counter and return. It is not necessary to change the counter value outside the interrupt service routine. In fact, it is not possible to ever change the counter value in microcontrollers such as the MC68HC11, other than by resetting the processor.
Regardless of the means used to implement the free-running timer, the current discussion assumes that a sixteen-bit free-running timer exists, and that its value is available to the system through access routines. Each increment of the timer constitutes a time slice in the current discussion. In an actual system, various actions occur independently of the timer. For instance, message passing would trigger other interrupts, regardless of the actions of the free-running timer. However, to test the various concepts, the simulator uses the timer to initiate processes such as genetic algorithm reproduction or location estimation based on the current database. While it would be possible to start an external process whose sole function is to drive the simulation, there is no significant difference in the operation of the system if those actions are driven by the internal timer.

At each time slice, certain actions occur, depending on the state of the task-manager. The task-manager allocates processor resources to the main processes: measurements, communication, database management, population management, estimator, and in the simulator, plotting. These activities are not necessarily allocated processor time in turn. Instead, for example, the task-manager may allocate more time slices to the population management process should there be a deficiency in valid paths to a large portion of the target nodes. Each of the processes may then allocate a portion of their time slices to their subprocesses.

6.2.2 - Time Stamping

Measurement transmissions arrive essentially at random. A simple implementation would store the measurement in a database, calculating estimates from the current measurement database. However, it is necessary for the system to distinguish between a valid, current measurement and leftover values for a unit no longer in contact.
It is convenient to speak of measurement validity in terms of existence. If the measurement exists in the database, then it is valid. One way of establishing the validity of a measurement is to assign it a validity tag. When the system is initialized, all measurement storage locations can be initialized as invalid. Then, as measurements arrive at random, the measurement can be stored and tagged valid. To detect aged and no longer useful measurements, it is necessary to also assign a time stamp, in this case a sixteen-bit integer, to each measurement.

As measurements, both those taken by on-board sensors and those received from other units, arrive at the database manager, the measurements are assigned a time-stamp corresponding to the current value of the free-running timer. This time stamp allows the system to invalidate measurements residing in the database after a certain period, represented by $K$ increments of the free-running timer, has elapsed.

As each measurement is accessed, either by the genetic algorithm module or the estimation module, its time-stamp is compared to the current value of the free-running timer. If the difference is greater than the predetermined value $K$, the measurement is tagged as invalid and not used since the measurement has aged past the predetermined limit. If the difference is less than $K$, the measurement is accepted. Thus, all measurements accessed by other modules will be compared for validity at the time of access. This process will be referred to as access validation, and can be easily made transparent to other modules.

Access validation is insufficient to guarantee that no measurements are outdated. It is also necessary to periodically scrub the entire database for outdated measurements, preferably twice or more during each timer cycle, to prevent integer rollover confusion. One scrubbing per timer cycle is insufficient, as a measurement may be received within $K$ increments prior to the scrub and accepted as valid. Now, if the first access occurs after
any integer number of timer cycles later but before the scrubbing process, it will be erroneously used for an estimate, although this measurement is quite old.

Scrubbing does not have to be accomplished all at once, interfering with other time-critical processes. It is sufficient to scrub a very small subset of the database at regular intervals, so long as the entire database is scrubbed at least twice per timer cycle.

Another benefit accrues from the use of a validity tag. The genetic algorithm module does not use the actual measurement data until a valid path is found and a fitness score is calculated. During the search for valid paths, only the validity tag is used to indicate measurements existing in the database. As a result, accesses by the genetic algorithm operators only require the validity tag to establish valid paths, rather than accessing two floating point values, range and bearing.

The predetermined value $K$ is compiled into the current simulator to allow a measurement to remain valid for only two measurement sweeps by all the units. This choice of two sweeps is purposely low to demonstrate how the genetic algorithm operators handle communication loss, discussed in Chapter 7.

6.2.3 - Measurement Simulation

The measurement model given in Figure 6.1 is used to simulate the physical measurement process. Since all units function identically, it is only necessary to show the actions of one particular unit. The following discussion concerning this model is from the perspective of only one unit, the source. Keep in mind that this discussion concerns only the simulation of the measurement process.
For each unit in the system, its absolute Cartesian coordinates as well as polar coordinates from the source are maintained in a simulator database containing omniscient information not available to a real system. Not shown in Figure 6.1 is the state of the sensors, that can be disabled by the user, or the current communication loss model. If neither disabled sensors nor a randomly selected communication loss prevents a measurement, the polar range is examined. If the polar range is greater than the maximum range, then the potential for a measurement is rejected, and the next unit is considered.

The next discrimination performed is the check for occlusion. The simulator database is searched for all units with a range less than the range of the current target. For any unit nearer the source than the current target, the bearing of the corners of the nearer target are calculated. If the target bearing is not either greater than all or less than
all of the corner bearings of the nearer unit, then the target is occluded and the possibility of a measurement is rejected.

Once the preceding checks have established that the target unit can be measured by the source, the noise specified by the current noise model is added to the true polar data to generate a noisy measurement. This measurement is then "transmitted" to the system by being stored in the measurement database with an updated time stamp.

6.2.4 - Data Structures

The current set of measurements in the database are the raw materials from which more abstract quantities will be constructed. At this point, it is necessary to review some terminology. Any given measurement from one unit to another is termed a network edge, or a link. A chain of such links, or vectors, from one unit to another through a number of intervening units is termed a path. A path may simply be the direct measurement, or link, from the observer to the target, or, in rare cases, may include all units in the system. The path length is simply the magnitude of the vector sum of the individual links. A path can be represented by a list of nodes, or unit identifiers. Note, however, that this list is merely a representation of the path, while the path itself is the actual trace of the vectors, with their associated errors, from the observer to the target. Associated with each path are several other quantities, such as fitness score, validity tag, weight, and others. These other quantities will be described in detail later. Also, it is necessary for reproduction purposes to require that each list representing a path contain all the nodes in the system, although only those prior to the target are considered for estimation purposes.
A variety of data structures forms the basis for most data storage in the system. The structures chosen for this simulator, along with their uses, are discussed in the paragraphs that follow. The structures are directly implementable in a working system.

**Vectors and coordinates** - All vectors, including measurements, and floating point coordinates are represented by a `FCOORD` data type.

```c
typedef struct tagFCOORD
{
    float x;
    float y;
} FCOORD;
```

For readers unfamiliar with C, this structure defines a new data type containing two single-precision floating point variables, x and y, that contain the components of a vector. The last three lines define pointer references, which improves readability in function prototypes, a practice familiar to most Windows programmers [4].

**Nodes** - Although not technically a structure, each node is represented by a `NODE` data type. This data type is defined as either an integer or character, depending upon the application and the machine representation. In a small system with less than 256 units, an unsigned 8-bit quantity would suffice. In the current simulator, a 16-bit quantity is used, not only because the resources exist, but also to stress the coding technique to expose weaknesses and require dynamic allocation.

```c
typedef int NODE;
typedef NODE *PNODE;
typedef NODE NEAR *NPNODE;
typedef NODE FAR *LPNODE;
typedef NODE _huge *HPNODE;
```
Paths - Paths are more than just lists of nodes.

typedef struct {
    HPNODE hpnodeNodeList;
    NODE nodeTargetNode;
    int nTargetIndex;
    float fFitness;
    float fWeight;
    FCOORD fxyPathVector;
    BOOL bIsValid;
} PATH;

typedef PATH *PPATH;
typedef PATH NEAR *NPPATH;
typedef PATH FAR *LPPOPUL;
typedef PATH __huge *HPPATH;

As the above structure shows, a path contains not only a pointer to a list of nodes, but also a target node, an index to the target in the list and a floating point fitness score. To ease manipulation, a path also contains a floating point weight, a FCOORD structure representing the latest path vector, and a Boolean variable indicating whether or not the path is valid, based on the current measurement database.

Populations - Populations, in turn, are more than lists of paths.

typedef struct {
    HPPATH ahppathPaths[POPULATION_SIZE];
    NODE nodeTargetNode;
    FCOORD fxyObservation;
    FCOORD fxyEstimate;
    int nNumValidPaths;
    float fTotalFitness;
    BOOL bValidEstimate;
    BOOL bValidObservation;
} POPUL;

typedef POPUL *PPPOPUL;
typedef POPUL NEAR *NPPOPUL;
typedef POPUL FAR *LPPOPUL;
As indicated by the structure definition on the previous page, a population of paths to each target node contains a pointer to an array of paths, defined earlier. In addition, a population contains a target node identifier, a FCOORD structure representing the latest composite observation based on the population of paths, and a FCOORD structure representing the latest location estimate. Also included for ease of manipulation are an integer variable representing the number of valid paths in the population, a floating point sum of the fitness in the population, and Boolean variables indicating the state of the estimate and observations.
6.3 - Genetic Algorithm Implementation

With the groundwork laid for implementation of the system, details applicable for the use of genetic algorithms will now be discussed. First, the implementation of a fitness scoring module is discussed, followed by a discussion of the population maintenance implementation. Finally, equality rejection is discussed.

6.3.1 - Fitness and Validity

The genetic algorithm fitness scoring module assigns a fitness score and a validity tag to each path based on the number of nodes in the path prior to the target, the summed per-link length of the path, and the validity of the path. The path is considered valid if the chain of nodes to the target are composed of links that exist in the measurement database. If one or more of these links are not present in the measurement database, the path is invalid. Validity is a far stronger description than fitness, for, except in certain cases, fitness is incalculable for invalid paths.

The fitness score is an indication of how accurately the path represents the true location of the target. In general, a lower fitness score represents a more accurate path, with two factors influencing this accuracy. First, the number of nodes in the path prior to the target represents, to a large degree, the variance of the composite measurement represented by the vector sum of the path. Each link in the path can be considered a random variable with a particular, yet unknown, variance. Likewise, the vector sum of these links can be considered another random variable, with variance equivalent to the sum of the individual variances of the component links. Therefore, the fewer the links,
the smaller the variance, and the more accurate the composite measurement represented by the particular path.

To a lesser degree, the fitness score is determined by the scalar sum of the magnitude of the individual links. The variance in the Cartesian coordinates of each link vector is a function of the length of the vector. As range increases, any bearing measurement error is magnified in the error in the Cartesian coordinates. So, the fitness score must take this effect into account.

Let $F_T$ be the total path fitness, $N$ the number of intervening nodes in the path, $L_i$ the scalar magnitude of each component link, and $R$ the maximum effective range of the sensors. Then, the following fitness function accommodates both of the above considerations:

$$F_T = \sum_{i=0}^{N} 1 + \left( \frac{L_i}{2R} \right) = N + 1 + \frac{1}{2R} \sum_{i=0}^{N} L_i$$ (6.1)

This function has the following properties, assuming that no measurement will have a range twice $R$:

- The minimum, or best, fitness score is 1, but is unattainable in practice.
- The maximum fitness score is less than twice the total number of units in the system.
- All links contribute between 1.0 and 2.0 to the total fitness score.
- Longer links increase, or worsen, the fitness score more than shorter links.

The factor of 2 in the denominator was chosen heuristically. A larger factor places more emphasis on producing paths with fewer nodes, while a smaller factor places
more emphasis on paths with fewer kinks. The choice of 2 represents an effort to simultaneously satisfy both of these goals.

Given the above discussion, one may be tempted to combine the fitness function with the validity tag by assigning invalid paths very large fitness scores, i.e. one million. This temptation should be resisted. As further discussion will illustrate, the fitness scores will be used for many purposes. Assigning arbitrary tag values will needlessly complicate these processes. In addition, assigning and checking a separate Boolean variable is far faster than decoding a floating point value. Finally, some processes require both a validity tag and a “leftover” fitness value from past performance for proper operation, as explained later.

6.3.2 - Population Maintenance

During operation, estimates to each target are based on the combination of a population of paths. In this case, thirty paths to each node are maintained [18]. Each population of thirty paths is “grown” over time through a variety of means, as discussed later. During each reproduction cycle, thirty new, tentative paths are produced by the reproduction module. These new paths are added to the original thirty paths in memory. Then, the fitness scores of all sixty paths are determined by the scoring module, discussed earlier.

Once the group of sixty paths are sorted by fitness values, the top thirty paths are retained, with the memory consumed by the remaining thirty being released to the system. This method, a departure from traditional genetic algorithm approaches that simply replace the entire population each cycle, is guaranteed to not reduce the overall
population fitness during each cycle [8]: in the worst case where all thirty new paths are poor, the original population is left unchanged.

In some circumstances, a valid path may become temporarily invalid. Such a circumstance could arise from communication noise, or from a moving obstacle temporarily obscuring an observation link. If the population is weak, offhandedly discarding a path could waste a large amount of effort relocating this path once the obstacle passes. If the population is strong, this loss may be insignificant. In any case, a facility for saving previously valid paths represents a potentially powerful source of replenishing a population with a minor investment in memory and bookkeeping. Some references [10] suggest using recessive traits, and the resultant doubling of memory consumption, to implement this capability. However, the operators discussed in Chapter 4 perform so well that this capability is not essential. Although memory is cheap, it should not be wasted implementing an unnecessary feature.

6.3.3 - Offspring Production Methods

All eight of the reproduction methods discussed in Chapter 4 are available in the simulator. However, as we shall see in Chapter 7, three of these operators, direct search, outside cross matching, and inside cross matching, form a fairly robust team that should be included, as a minimum, in the final working model. The reproduction operators are implemented directly as discussed in Chapter 4, so, to avoid redundancy, the implementation will not be discussed here. The interested reader is directed back to that earlier chapter, using the data structures listed in Section 6.2 and the code in Appendix A as reference materials.
6.3.4 - Equality Rejection

The maintenance of a valid population requires that duplicate paths be removed from the population of paths to each target node. A brute force node-by-node comparison of the population is inefficient, and in fact could miss matches prior to the target, as nodes past the target are irrelevant. A more efficient, and effective, hierarchical method is described below:

Step 1: Compare target lengths. If identical, continue to step 2.
Step 2: Compare fitness values. If identical, continue to step 3.
Step 3: Compare node-by-node to the targets. If identical, the paths are identical.

Once duplicates have been identified, many options exist for removing the repeated elements. If the population is of sufficient quality, the most expedient option is to tag the path as invalid and continue. However, if the population is weak, a more fruitful option is to search for further paths by random generation, as well as any of the unary methods given above, operating on the repeated path.

This situation is likely to happen only when the combinatorial search has repeated itself. Otherwise, the population would be strong, since duplicates are most likely to occur when the population is finding the best paths available. In this situation, the unary operators are likely to produce another duplicated path.
An efficient algorithm for scrubbing the population for repeated paths is as follows:

Step 1: Sort the population by fitness.
Step 2: Accept the first path as unique.
Step 3: Check the next path for validity. If valid, continue to Step 4. If invalid, repeat until no more paths exist, then continue at Step 6.
Step 4: Compare the path selected in Step 3 to valid paths preceding the current path in the sort order. If no duplicates are found, accept the path as unique and repeat Step 3. If a match exists, continue to Step 5. If no more valid paths exist, continue at Step 6.
Step 5: Attempt to remedy the duplication by either tagging the path as invalid or generating a new path. Repeat Step 3 with this path as the next path. Note: a possibility of an infinite loop exists here. Implement an attempt counter to prevent infinite loops and tag the path as invalid if the counter limit is exceeded, then continue at Step 3.
Step 6: Sort population by fitness and accept the best paths as the new population.
6.4 - Estimator Implementation

The remaining portion of the system requires taking the paths produced by the genetic algorithms and producing location estimates. This process will now be discussed.

6.4.1 - Prefilter

Earlier, in Chapter 5, the case was made for prefiltering the measurements on board each unit before transmitting that data. The prefilter chosen for this application is a simple low-pass filter. Pseudocode for the prefilter algorithm follows:

Collect a new Range\textsubscript{measured} and Bearing\textsubscript{measured} from the sensors
If there is a valid older measurement in the database
\{
  \text{Range\textsubscript{new}} = \text{OLD\_WEIGHT} \times \text{Range\textsubscript{old}} + \text{NEW\_WEIGHT} \times \text{Range\textsubscript{measured}}
  \text{Bearing\textsubscript{new}} = \text{OLD\_WEIGHT} \times \text{Bearing\textsubscript{old}} + \text{NEW\_WEIGHT} \times \text{Bearing\textsubscript{measured}}
  \text{where } \text{OLD\_WEIGHT} + \text{NEW\_WEIGHT} = 1
\}
otherwise
\{
  \text{Range\textsubscript{new}} = \text{Range\textsubscript{measured}}
  \text{Bearing\textsubscript{new}} = \text{Bearing\textsubscript{measured}}
\}
\text{X} = \text{Range\textsubscript{new}} \times \cos(\text{Bearing\textsubscript{new}})
\text{Y} = \text{Range\textsubscript{new}} \times \sin(\text{Bearing\textsubscript{new}})
\text{Transmit X and Y to the group}

In the above implementation, OLD\_WEIGHT was heuristically chosen to be 0.6 and NEW\_WEIGHT was chosen to be 0.4. However, more sophisticated estimators could be used instead. The choice of estimator is discussed further in Chapter 8.
6.4.2 - Composite Observation Estimators

Location estimates to other units in the system are calculated by a two-level process. First, the vector sums of measurements along each path in the population are combined to form a composite observation, as discussed in Chapter 3. Next, the composite observations are combined with previous estimates, if any, to form new estimates. This section deals with the former, while the next section handles the latter.

In the simulator, four composite observation estimators are available: mean, median, mode-mean, and mode-median. Each of these estimators will be discussed in the following paragraphs.

The mean estimator simply forms a weighted average of the path vectors, where the weight of each path is the inverse of the fitness score for each path. As a result, shorter, better paths are weighted more heavily than longer, potentially more erratic paths.

Modifying equation (3.2) yields:

\[
Z_k = \frac{\sum_{i} w_i P_{OT,i}}{\sum_{i} w_i}, \quad w_i = \frac{1}{fitness(P_{OT,i})}
\]  

(6.2)

The median estimator forms a composite observation by using a weighted median estimate, discussed in Section 5.1.2.
The weighted median is accomplished by the following algorithm:

Step 1: Assign each path a weight equal to the inverse of the fitness score.
Step 2: Calculate the total weight of the population and divide by 2.0.
Step 3: Sort the population according to $X$ components of the path vectors.
Step 4: Starting at one end of the population, add the individual weights until the total is equal to or greater than half the total weight.

Step 5: Accept the $X$ component of the current path vector as the $X$ component of the composite observation. Alternatively, one could interpolate between the two bounding values of $X$ for the $X$ component of the composite observation. The former was implemented in the simulator.

Step 6: Repeat Steps 3-5 for the $Y$ component.
Step 7: Repeat Steps 3-5 for the $Z$ component, if working in 3D.

The mode-mean and mode-median estimators first extract the shorth, or shortest half, of the population. Then, the population shorth is passed to the mean or median estimator module, as appropriate, to form a weighted estimate as described above. The population shorth is extracted by the following algorithm:

Step 1: Sort the population of $N$ paths by coordinate.
Step 2: Calculate the shorth length $D = 1 + N/2$.
Step 3: For each of $C = 1 + N - D$ candidate shorths, calculate the distance between the first element and the last element.
Step 4: Accept the ordered subset of paths with the smallest distance as the shorth.
6.4.3 - Location Estimator

The upper level of estimation calculates the location estimate based on a low pass filter, similar to the filter used for the measurement prefilter. The following pseudocode illustrates the technique:

```
Calculate a new composite observation
If there is a valid older estimate in the database
{
    Estimate_{new} = OLD_WEIGHT * Estimate_{old} + NEW_WEIGHT*Observation
}
otherwise
{
    Estimate_{new} = Observation
}
```

As before, the OLD_WEIGHT and NEW_WEIGHT were selected heuristically, this time 0.4 and 0.6, respectively. The preceding comments concerning more sophisticated estimation techniques apply to this estimate as well.

This chapter has detailed the implementation of the system, suitable for implementation on actual hardware. In addition, simulator-specific issues have been addressed. In the next chapter, the results of simulation of the system are presented.
Chapter 7 - Simulation Results

In this chapter, the performance of the order-based genetic algorithms will be discussed. By using the simulator, the system will be exercised in several scenarios. First, the design of the scenarios is discussed. Then, the simulation results are presented.

7.1 - Scenario Design and Objectives

7.1.1 - Key Issues

In Chapter 3, results from early simulations were presented that demonstrated performance in various patterns, along with comparisons of static and dynamic cases. Also, the effects of error rejection were compared to the effects of accepting scrambled data. In addition, the effect of the Gaussian density noise model was compared to published step noise models based on sensor rings. As those issues are invariant with addition of the genetic algorithms, they will not be readdressed in this chapter.

In this chapter, the performance of the order-based genetic algorithms is the prime issue. Particularly addressed is how well these algorithms handle different unit layouts and various degrees of sensor noise and communication loss. Also considered is the utility of the various estimators discussed in Chapter 6.

7.1.2 - Scenario Model Definitions

The simulation models listed on the next page were used to create the various scenarios.
Model Definitions

Field Type
Operator Sets
Sensor Model
Communication Loss Model
Estimator Model

Each of these models must be discussed in detail before presenting the simulation results.

Field Type

Two field types were used, wide and deep. The wide field model simulates a situation in which the units are arranged in a compact pattern. The characteristic feature of a wide field model is that there is an abundance of redundant paths to each unit. The wide field model presents a situation in which the system can attain very robust performance despite degradation in communication loss and other errors. The figure below shows an example of a wide field model.

![Wide Field Pattern](image)

Figure 7.1 - An example of a wide field pattern.
In the above example, the source unit is located at the edge of the pattern. An even more extreme example of a wide field model would have the source surrounded by other units. However, since even dense patterns have units at the edge, the source unit is placed at the edge of the pattern during simulations.

The deep field model represents a situation in which the units are more dispersed, such as along a line. However, units in a geometrically compact group can still represent a deep field model, as shown in Figure 7.2.

![Figure 7.2 - An example of a deep field pattern.](image)

If sensors on the units marked with an X are either damaged or disabled, the group that results is topologically equivalent to a ragged line pattern, as shown by the dashed line. This is the pattern used for deep field simulations below. Note that estimates can still be formed to the disabled units, which are not invisible, but merely have their sensors disabled.

The deep field pattern increases the number of iterations necessary to find paths to all units, as well as increases the average estimate error. This increase in the average
estimate error is due to two factors: fewer redundant paths exist to each unit, and the paths to each unit contain more intervening units, increasing the variance, as discussed in Chapter 5.

**Operator Sets**

The genetic algorithm reproduction operators discussed in Chapter 4 are grouped into three sets, listed below:

Set 1
- Random Production
- Crossover
- Linear Inversion
- Circular Inversion

Set 2
- Direct Search
- Outside Cross Matching

Set 3
- Direct Search
- Outside Cross Matching
- Inside Cross Matching
- Randomized Reduction

Set 1 represents the traditional genetic algorithm operators. Set 2 represents the barest minimum set composed of the domain-specific operators, in terms of a subtour search, while Set 3 is a superset of the domain-specific operators.

**Sensor Model**

Three sensor noise models are considered, *Good*, *Fair*, and *Poor*. The statistics related to each sensor model are listed on the next page.
Good

\[ \sigma_{\text{Range}} = 0.1 \text{ meters} \]
\[ \sigma_{\text{Bearing}} = 2.5 \text{ degrees} \]

Fair

\[ \sigma_{\text{Range}} = 0.5 \text{ meters} \]
\[ \sigma_{\text{Bearing}} = 7.5 \text{ degrees} \]

Poor

\[ \sigma_{\text{Range}} = 1.0 \text{ meters} \]
\[ \sigma_{\text{Bearing}} = 15.0 \text{ degrees} \]

Note that the *Fair* noise model has five times the range deviation and three times the bearing deviation of the *Good* model, and that the *Poor* model has twice the deviation of the *Fair* model.

*Communication Loss Model*

Four communication loss models are considered.

- Lossless
- Five percent loss
- Twenty percent loss
- Forty percent loss.

*Estimator Model*

Four estimator models are considered.

- Mean
- Median
- Mode-Mean
- Mode-Median
7.1.3 - Data Collection

With the above options available, there are \((2)(3)(3)(4)(4) = 288\) possible scenarios. Fortunately, only a few of these are necessary to explore the utility of the various methods. The data presented below was obtained by holding various parameters constant, and varying others, as explained in each section. In this way, the relevant issues can be concisely addressed. In Chapter 3, the results of one hundred trials over one hundred and twenty iterations were displayed. Fortunately, the performance of the algorithms have improved to the point that only thirty iterations are necessary in this chapter. Each trace on the charts on the pages to follow was obtained by averaging ten sets of data, which reveals more fluctuation than the previous average of one hundred sets.

The genetic algorithms demonstrated throughout this chapter were based on a network size of forty units. The base population size of thirty paths is maintained using the superset replacement technique.
7.2 - Results

In this section, the results of simulation are presented. First, sets of operators are compared for ability to produce populations. Then, the effects of varying amounts of sensor noise and communication loss are compared. Finally, the various estimator models are compared.

7.2.1 - Operator Comparison

In this first round of comparisons, the three sets of operators are compared. In all cases considered here, the sensors are assumed noise-free and the communication is assumed lossless. Also, the median estimator is used.
Wide Field

The graph below shows the average population size, in terms of number of valid paths per unit, for each of the three operator sets working in a wide field.

Figure 7.3 - Comparison of the effect of operator set on the average population size in a wide field.

Note that in Figure 7.3, Set 1, the traditional operators, barely gets started, while Sets 2 and 3 make immediate gains. Also, Set 3, a superset of Set 2, rises faster than set 2, although both sets quickly form nearly 27 population members per unit.
The graph below shows the average number of units found, a maximum of 39, for the same experiment:

![Graph showing operator comparison in a wide field](image)

**Figure 7.4** - Comparison of the effect of operator set on the number of target units found in a wide field.

As before, set 1 barely gets started, finding, on average, only the four or five nearest units by iteration thirty. On the other hand, by iteration four or five, sets 2 and 3 have estimates to all the units in the system, with set 3 having a slight edge. Since operator set 1 is so inept at finding paths, it will not be considered further.
Deep Field

One would expect some degradation in performance in the deep field case compared to the wide field case. The primary cause of this degradation is that longer paths, in the sense that the paths contain more intervening units, must be found to each target. The graph below shows the average population size per unit for operator sets 2 and 3 working in a deep field, along with the wide field results of operator 3.

![Operator Comparison, Deep Field](image)

**Figure 7.5** - Comparison of the effect of operator set on the average population size in a deep field.

In this case, the performance of sets 2 and 3 are barely distinguishable. Note that set 2 at times has a slight edge over set 3, the reverse of the wide field results. Also, it is seen from the above graph that the system finds fewer paths to all units, which can be expected from the deep field layout. This result primarily affects the nearer units, for which fewer redundant paths are now available. The graph on the next page shows the average number of estimates found for the same experiment.
Figure 7.6 - Comparison of the effect of operator set on the number of target units found in a deep field.

Again, the performance of sets 2 and 3 are so similar as to be considered identical. Note that by iteration nine, in all cases, all the units in the system have been found.

Overall, it is obvious that the traditional genetic algorithm operators do not handle order-based applications, such as this one, well at all. However, the domain-specific operators used in sets 2 and 3 perform exceptionally well. In a pinch, the two operators in set 2, *direct search* and *outside crossmatching*, could be used exclusively without a serious degradation of performance.

7.2.2 - Sensor Noise

In this section, the effect of sensor noise will be examined. In all cases considered here, operator set 3 is used, along with the median estimator. In addition, lossless communication is assumed. Since sensor noise has no effect on population generation,
and thus the existence of estimates, the relevant ideas now revolve around population fitness and estimate quality.

**Wide Field**

The effect of sensor noise on estimate quality in a wide field is plotted below in Figure 7.7.

![Noise Comparison, Wide Field](image)

**Figure 7.7** - Comparison of the effect of sensor noise on the average error per unit in a wide field.

Recall that the *Poor* model has twice the deviation of the *Fair* model, which in turn has five times the range deviation of the *Good* model. As predicted by the analysis in Chapter 5, the resulting error seems linearly related to the deviation in the noise, despite the warping of the density function. In the plots above, it is visually apparent that the *Fair* model has only half the average error of the *Poor* model, while the *Good* model
has roughly twenty percent of the error of the *Fair* model. Recall from Chapter 5 that the greatest majority of this error is bias introduced by the warped density function.

The effect of sensor noise on fitness values in a wide field is plotted below.

![Noise Comparison, Wide Field](image)

**Figure 7.8** - Comparison of the effect of sensor noise on the average path fitness in a wide field.

As shown above, the path fitness values seem independent of the sensor noise model. Therefore, improving the measurements by reducing the variance, and hence the bias, should have little effect on the performance of the genetic algorithms. This discovery enhances the modular aspect of the system in that changes in one module have little to no effect on other modules.

As explained in Chapter 6, lower fitness values are better. However, the above plot shows the fitness values increasing initially from the baseline fitness average produced by only finding the nearest units. Referring back to the operator comparison plots, the population increases initially and then slowly decreases, reaching a steady state.
During this period of expanding populations, the average fitness values naturally increase as the system locates more distant units. Once all the units have been located, the fitness values gradually improve, as shown above.

Deep Field

The effect of sensor noise on estimate quality in a deep field is shown below.

![Graph showing noise comparison in a deep field](image)

**Figure 7.9** - Comparison of the effect of sensor noise on the average error per unit in a deep field.

In this plot, the error is a bit more wild, although the same relationships between the sensor noise models remain. Also note that the errors are larger than in the wide field case, which is expected since the paths are in general longer, and thus contain the sum of more individual measurement variances.
The effect of sensor noise on fitness values in a deep field is plotted below.

![Noise Comparison, Deep Field](image)

**Figure 7.10** - Comparison of the effect of sensor noise on the average path fitness in a deep field.

As before in the wide field case, the fitness values are generally independent of the noise model. Also note that, as before, the fitness values initially increase until all units are found. Then, the fitness values gradually fall as more fit paths are found.

### 7.2.3 - Communication Loss

In this section, the effects of communication loss are examined. In all cases considered, operator set 3 is used, and the sensors are assumed to be noise-free. Again, the median estimator is used. Communication loss should have the greatest impact on population size and number of targets found. Therefore, the effect of the communication loss models on these values are considered. As discussed in Chapter 6, the system can be given a degree of memory to combat the effect of communication loss. In the examples that follow, the system only retains measurements for a maximum of two iterations.
While this value could be increased to improve performance, a small value better illustrates the relevant concepts.

Wide Field

Figure 7.11 compares the effect of communication loss on population size in a wide field.

![Comm Loss Comparison, Wide Field](image)

**Figure 7.11** - Comparison of the effect of communication loss on the average population size in a wide field.

The chart above indicates that there is a nearly linear relationship between population loss and communication loss. In other words, a twenty percent loss in communication causes, roughly, a twenty percent loss in population size, and so on. According to the plot above, one might surmise that the system should be able to handle even an eighty percent communication loss and still have some population members left. Unfortunately, this is not the case, as demonstrated later in Figure 7.13.
The graph below shows the effect of communication loss on number of targets found in a wide field.

![Comm Loss Comparison, Wide Field](image)

**Figure 7.12** - Comparison of the effect of communication loss on number of target units found in a wide field.

Note the degradation in number of target units found with forty percent communication loss. However, the system seems remarkably tolerant of twenty percent loss. Comparing this chart to the previous chart, it seems that the effect of increasing communication loss is to first degrade population size, and then lose track of the outlying units. This situation is exacerbated in the deep field case.
Deep Field

The graph below compares the effect of communication loss on population size in a deep field.

![Comm Loss Comparison, Deep Field](image)

**Figure 7.13** - Comparison of the effect of communication loss on the average population size in a deep field.

Compared to the wide field case, the deep field case roughly doubles the population loss. The previously mentioned linear relationship still seems to hold, although the magnitude of loss has increased.
The graph below shows the effect of communication loss on number of targets found in a deep field.

![Comm Loss Comparison, Deep Field](image)

**Figure 7.14** - Comparison of the effect of communication loss on number of target units found in a deep field.

In the wide field case, twenty percent communication loss had little effect on the number of target units found. As shown in Figure 7.14, even five percent communication loss has some effect. Increasing the memory parameter to greater than two iterations would decrease this effect. However, the intuitive notion that more disperse patterns are more sensitive to communication loss is supported by these results.
7.2.4 - Estimator Comparison

In this section, the effect on estimate quality of the various estimators is examined. Operator set 3 was used with the Fair sensor noise model and twenty percent communication loss in a wide field.

![Estimator Comparison](image)

**Figure 7.15** - Comparison of the effect of various estimators.

As shown in Figure 7.15, the mean and median estimators seem to perform best. Since the mean estimator is provably the optimal estimator in the mean-square sense, this result is comforting. Note that in all cases, bias rules the day, preventing a decrease in the error to zero.

7.2.5 - Transient Obstructions

So far, operator sets 2 and 3 have shown similar performance. If this is the case, then what advantage accrues from using operator set 3? The following demonstration will show that the inside cross matching operator improves performance when countering transient obstructions.
Consider the modified deep field pattern below:

![Figure 7.16 - Deep field pattern used to compare operator sets 2 and 3.]

Again, assume sensors on the nodes marked with an X are either damaged or disabled, except for the node also marked with a box, the intermittent node, whose sensors intermittently function, simulating the effect of transient obstacles. For the discussion that follows, we wish to find multiple paths to the node marked with a circle, the target node.

The procedure used for comparing operator sets 2 and 3 is as follows. Starting with a "lobotomized" path population, leave the intermittent node turned off for ten cycles. At the end of ten cycles, enable the intermittent node and count the cycles until the first path to the target through the intermittent node. For fun, leave the intermittent node turned on for a total of ten cycles, and then turn it off again, observing the reconstruction of the path population to the target node. For this comparison, communication was set to lossless to isolate the effect of the different operators.
Using the above procedure with operator set 2 produced the following results: The target node was first located after six iterations. After enabling the intermittent node, four iterations elapsed before a path was formed through the intermittent node. After disabling the intermittent node, the path population remained constant for one iteration, demonstrating the memory effect discussed in Chapter 6. On the second iteration after disabling the intermittent node, the population reverted to taking the S-shaped route to the target, showing no loss of estimation despite the loss of paths through the intermittent node.

Using the above procedure with operator set 3 produced the following results: The target node was first located, again, after six iterations. After enabling the intermittent node, only two iterations elapsed before a path was formed through the intermittent node. After disabling the intermittent node, the path population remained constant for one iteration, again demonstrating the memory effect discussed in Chapter 6. On the second iteration after disabling the intermittent node, as with operator set 2, the population reverted to taking the S-shaped route to the target, showing no loss of estimation despite the loss of paths through the intermittent node.

The lesson to be learned from this example is that the inside cross matching operator found a shortcut to the target twice as fast, two iterations as compared to four, as the outside cross matching operator.

However, the inside cross matching operator had no effect on the forward search for paths, the specialty of the outside cross matching operator. In both cases, the memory effect designed into the system performed as desired, showing a robust response to failure. These examples demonstrate that the preferred operator set is set 3: direct search, outside cross matching, inside cross matching, and randomized reduction.
Chapter 8 - Conclusions

In this final chapter, the conclusions drawn from the previous material are presented. First, the performance of the genetic algorithm operators will be discussed. Next, the issue of location estimators will be considered, followed by a discussion of practical considerations when implementing order-based genetic algorithms in an embedded application.

8.1 - Performance of Genetic Algorithm Operators

As seen in Chapter 7, the traditional genetic algorithm operators, crossover and mutation, perform poorly in comparison to the hybridized genetic algorithm operators discussed in Chapter 4. A minimum set of order-based genetic algorithm operators for subtour applications consists of the direct search and the outside cross matching operators. These two operators are sufficient for effective forward path searching, regardless of the node topology.

For enhanced performance in situations where the nodes are intermittently able to sense their neighbors, the inclusion of the inside cross matching operator enhances the robustness of the system. Although the example cited in Chapter 7 showed a doubling of search speed, the improvement is geometry dependent, and is often four or more times faster for large populations. Another factor influencing intermittent sensors and/or communications loss is the memory factor assigned to the measurement database. All the examples used in this thesis were based on a two-iteration limit to stress the system. A larger memory limit does not affect the quality of the estimates as long as new information arrives and displaces old data, but it does enable the system to maintain good path populations in the event of transient losses.
8.2 - Performance of Estimators

Location estimation, of secondary importance in this work, was included to demonstrate the application of a path population. Four estimators, mean, median, mode-mean, and mode-median, were compared for effectiveness. As expected, the mean estimator provided the best results, although not by a very large margin.

The effect of the estimators is independent of the path search module. In fact, an entire chapter, Chapter 3, was devoted to the issue of location estimation without the aid of genetic algorithms, although this chapter motivated their development. As shown in Chapter 3, motion plays a large role in the effectiveness of the estimate quality, as does sensor noise, as seen in Chapter 7. Chapter 5 discussed the effect of the sensor noise variance on the bias in the final estimates. To reduce bias in the location estimates, it is necessary to reduce the variance in the sensor noise. Although one obvious solution is to improve the sensor quality, another acceptable approach is to prefilter the measurements at each node, transmitting a filtered version.

Since the measurement noise variance has a predictable effect on the estimate bias, that of linearly contracting the estimate net, two solutions, not covered in this thesis, emerge. Both approaches would use a prefilter, such as a Kalman filter, that provides an estimate of the variance in the measurements. The first approach would apply an expansion factor, based on the variance, to the measurement estimate before transmission, canceling the bias in the location estimates. The second approach would transmit the variance estimate along with the measurement estimate. Then, the variance estimate could be used to properly weight the composite observations used to form the location estimates. The second approach would reduce the bias by a small amount, while the first approach is more likely to make a significant reduction of the location estimate bias by actively canceling the estimation bias caused by the polar-to-Cartesian conversion.
8.3 - Practical Considerations

In this section, practical considerations for embedded application are discussed. First, issues involving asynchronous application are covered, followed by a discussion of the implementation of the genetic algorithms. Finally, some thoughts on hardware selection are offered.

Chapter 6 discussed asynchronous system operation in detail. Implementation of a measurement time stamp is a key element in increasing system robustness. However, care must be taken to scrub the measurement database, as discussed in that chapter, at least two, and preferably more, times per stamp period to prevent rollover confusion. Ideally, the stamp period should be long enough to allow several, on the order of a dozen, system iterations, where a system iteration is defined as the transmission of measurements by the entire network.

Due to the modular nature of the system, servicing of the path populations by the genetic algorithms is completely independent of the rate of arrival of the measurements. Observation of the simulator has shown that, in general, the genetic algorithm modules will far outpace the communications bandwidth of the system for network sizes of a few dozen nodes. However, this relative performance is dependent on the particular hardware chosen. In any event, it is unlikely, unless ridiculously large path population sizes are used, that the genetic algorithm modules will be the bottleneck in the system, a goal implied at the end of Chapter 3.

Implementation of the genetic algorithms offers a great deal of flexibility. As most of the manipulations, other than the few that involve fitness scores, involve integer strings, a math coprocessor is not necessary. In addition, the path population sizes may be adjusted, dynamically if necessary, to suit the available memory. Use of the superset
population replacement concept, discussed in Chapter 4, is highly recommended over the traditional generational replacement technique. Attempts to save a few bytes here would be penny wise and pound foolish, considering the guarantee against devolution offered by the superset method.

As each node processes a path population for each other node, the system is naturally parallel. But, since the cross matching operators spy on other populations, the path processing is not independent. Regardless, the genetic algorithms offer an excellent opportunity to implement some sort of parallel processing scheme.

The system as used here implemented each path as an integer string of length equal to the number of nodes in the system. However, as each path is essentially a subtour of length much less than the number of nodes in the system, a considerable memory savings is possible. However, the complexity added by requiring dynamic memory allocation, deallocation, and packing may make addition of more memory the cheapest and easiest solution. Certainly, this task should not be undertaken by an operating system novice, since the memory management must be absolutely leakproof, a rare thing even today. With this concern in mind, the system as implemented is more likely to succeed than a system destabilized by attempts to improve an already acceptable solution.

The genetic algorithms manipulate primarily integer strings. If one were comfortable with using only a handful of operators, such as direct search, the cross matching operators, and randomized reduction, some of the steps in processing these operators, especially the shifting and copying, could be implemented in hardware. However, as the genetic algorithms are certainly fast enough in software, this approach may be a solution in search of a problem.
As the system was developed in well-defined modules, interrupt enabling and disabling can occur in an orderly fashion at module entry and exit points. However, in an actual application, measurement transmissions will arrive essentially at random. To prevent loss, extra measures not used in the Windows non-preemptive environment must be considered. One possible solution to the competition for resources is to split the burden among several processors. The key body of information in the system, and the temporal hazard, is the measurement database. Adequate management of this database could require three separate processors: 1) a measurement receptionist that handles communications and buffers incoming measurements against loss, 2) a database supervisor that controls access to the database as well as scrubs the time stamps, and 3) the rest of the system, that manipulates the path populations and generates estimates.

Overall, the system is robust and effective. Because of the redundant path population, the system is able to form relatively precise location estimates, despite a great deal of sensor noise. The performance of the genetic algorithms exceeded my expectations, especially since the customized operators performed much better than the traditional operators.
Appendix - GENES: The Genetic Algorithm Simulator

This appendix contains the source code for the simulator, listed in its entirety. The simulator source code consists of 6757 lines of code in 64 files. The following files are listed in alphabetical order in the pages which follow.

<table>
<thead>
<tr>
<th>File</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>about.c</td>
<td>122</td>
</tr>
<tr>
<td>bmpmgr.c</td>
<td>122</td>
</tr>
<tr>
<td>bmpmgr.h</td>
<td>125</td>
</tr>
<tr>
<td>child.c</td>
<td>125</td>
</tr>
<tr>
<td>child.h</td>
<td>128</td>
</tr>
<tr>
<td>dialog.h</td>
<td>129</td>
</tr>
<tr>
<td>estimate.c</td>
<td>129</td>
</tr>
<tr>
<td>estimate.h</td>
<td>135</td>
</tr>
<tr>
<td>expmt.c</td>
<td>136</td>
</tr>
<tr>
<td>expmt.h</td>
<td>141</td>
</tr>
<tr>
<td>fcoord.h</td>
<td>141</td>
</tr>
<tr>
<td>genedlg1.dlg</td>
<td>141</td>
</tr>
<tr>
<td>genedlg1.h</td>
<td>142</td>
</tr>
<tr>
<td>genedlg2.dlg</td>
<td>142</td>
</tr>
<tr>
<td>genedlg2.h</td>
<td>142</td>
</tr>
<tr>
<td>genedlg3.dlg</td>
<td>142</td>
</tr>
<tr>
<td>genedlg3.h</td>
<td>143</td>
</tr>
<tr>
<td>genedlg4.dlg</td>
<td>143</td>
</tr>
<tr>
<td>genedlg4.h</td>
<td>144</td>
</tr>
<tr>
<td>genedlg5.dlg</td>
<td>144</td>
</tr>
<tr>
<td>genedlg5.h</td>
<td>144</td>
</tr>
<tr>
<td>genedlgs.c</td>
<td>145</td>
</tr>
<tr>
<td>genedlgs.h</td>
<td>153</td>
</tr>
<tr>
<td>genemenu.c</td>
<td>154</td>
</tr>
<tr>
<td>genemenu.h</td>
<td>156</td>
</tr>
<tr>
<td>geneplot.c</td>
<td>156</td>
</tr>
<tr>
<td>geneplot.h</td>
<td>160</td>
</tr>
<tr>
<td>genes.def</td>
<td>160</td>
</tr>
<tr>
<td>genes.h</td>
<td>161</td>
</tr>
<tr>
<td>genes.mak</td>
<td>161</td>
</tr>
<tr>
<td>genes.rc</td>
<td>168</td>
</tr>
<tr>
<td>genetype.h</td>
<td>170</td>
</tr>
<tr>
<td>glblopts.c</td>
<td>170</td>
</tr>
</tbody>
</table>
glblopts.h .......................................................... 174
instance.h .......................................................... 176
listmgr.c .......................................................... 176
listmgr.h .......................................................... 181
metrics.h .......................................................... 181
msmtdb.c .......................................................... 182
msmtdb.h .......................................................... 185
msmtmkrl.c ......................................................... 185
msmtmkrl.h .......................................................... 190
numunits.h .......................................................... 190
paintmgr.c .......................................................... 191
paintmgr.h .......................................................... 191
pathdb.c .......................................................... 197
pathdb.h .......................................................... 197
pathmgr.c .......................................................... 197
pathmgr.h .......................................................... 208
plotbmp.c .......................................................... 209
plotbmp.h .......................................................... 210
populmgr.c .......................................................... 210
populmgr.h .......................................................... 219
random.c .......................................................... 219
random.h .......................................................... 221
resource.h .......................................................... 221
timeslc.c .......................................................... 221
timeslc.h .......................................................... 225
timestamp.c .......................................................... 225
timestamp.h .......................................................... 226
vector.c .......................................................... 226
vector.h .......................................................... 227
winmain.c .......................................................... 227
wndproc.c .......................................................... 228
about.c

#define STRICT
#define NOCOMM
#include <Windows.h>
#include <windowsx.h>
#include "genes.h"
#pragma warning (disable:4100)

BOOL CALLBACK _export
About (HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg) {
    case WM_INITDIALOG:
        return TRUE;
    case WM_COMMAND:
        if (wParam == IDOK || wParam == IDCANCEL)
        {
            EndDialog (hdlg, TRUE);
            return TRUE;
        }
        break;
    }
    return FALSE;
}

bmpmgr.c

/* bmpmgr.c - Bitmap Manager */

#define STRICT
#define NOCOMM
#include <Windows.h>
#include <windowx.h>
#include "bmpmgr.h"

#define NUMBER_BITMAPS 21

static HBITMAP hBitmap[NUMBER_BITMAPS];

void OpenBitmaps( HINSTANCE hInstance )
{
    hBitmap[0] = LoadBitmap( hInstance, "SOURCE_BMP" );
    hBitmap[1] = LoadBitmap( hInstance, "A_QE_PA_BMP" );
    hBitmap[2] = LoadBitmap( hInstance, "A_QE_PM_BMP" );
    hBitmap[3] = LoadBitmap( hInstance, "A_QE_PF_BMP" );
    hBitmap[4] = LoadBitmap( hInstance, "A_QF_PA_BMP" );
    hBitmap[5] = LoadBitmap( hInstance, "A_QF_PM_BMP" );
    hBitmap[6] = LoadBitmap( hInstance, "A_QF_PF_BMP" );
    hBitmap[7] = LoadBitmap( hInstance, "A_QF_PA_BMP" );
    hBitmap[8] = LoadBitmap( hInstance, "A_QF_PM_BMP" );
    hBitmap[9] = LoadBitmap( hInstance, "A_QF_PF_BMP" );
    hBitmap[10] = LoadBitmap( hInstance, "A_QN_PF_BMP" );
    hBitmap[11] = LoadBitmap( hInstance, "I_QE_PA_BMP" );
    hBitmap[12] = LoadBitmap( hInstance, "I_QE_PM_BMP" );
    hBitmap[13] = LoadBitmap( hInstance, "I_QE_PF_BMP" );
    hBitmap[14] = LoadBitmap( hInstance, "I_QF_PA_BMP" );
    hBitmap[15] = LoadBitmap( hInstance, "I_QF_PM_BMP" );
void CloseBitmaps( void )
{
    int i;
    for( i = 0; i < NUMBER_BITMAPS; i++ )
    {
        DeleteObject( hBitmap[i] );
    }
}

HBITMAP GetSourceBitmapHandle( void )
{
    return hBitmap[0];
}

HBITMAP GetTargetBitmapHandle( BOOL bSensorsActive, int nEstimateState, int nPopulationState )
{
    if( bSensorsActive == TRUE )
    {
        switch( nEstimateState )
        {
        case ESTIMATE_STATE_EXCELLENT:
            switch( nPopulationState )
            {
            case POPULATION_STATE_ALL:
                return hBitmap[1];
            case POPULATION_STATE_MANY:
                return hBitmap[2];
            case POPULATION_STATE_FEW:
                return hBitmap[3];
            case POPULATION_STATE_NONE:
                return hBitmap[10];
            }
        case ESTIMATE_STATE_FAIR:
            switch( nPopulationState )
            {
            case POPULATION_STATE_ALL:
                return hBitmap[4];
            case POPULATION_STATE_MANY:
                return hBitmap[5];
            case POPULATION_STATE_FEW:
                return hBitmap[6];
            case POPULATION_STATE_NONE:
                return hBitmap[10];
            }
        case ESTIMATE_STATE_POOR:
            switch( nPopulationState )
            {
            }
        }
else
    case POPULATION_STATE_ALL:
        return hBitmap[7];
    case POPULATION_STATE_MANY:
        return hBitmap[8];
    case POPULATION_STATE_FEW:
        return hBitmap[9];
    case POPULATION_STATE_NONE:
        return hBitmap[10];

    case ESTIMATE_STATE_NONE:
        return hBitmap[10];
    }
else
    {
    switch( nEstimateState )
    {
    case ESTIMATE_STATE_EXCELLENT:
        switch( nPopulationState )
        {
        case POPULATION_STATE_ALL:
            return hBitmap[11];
        case POPULATION_STATE_MANY:
            return hBitmap[12];
        case POPULATION_STATE_FEW:
            return hBitmap[13];
        case POPULATION_STATE_NONE:
            return hBitmap[20];
        }
    case ESTIMATE_STATE_FAIR:
        switch( nPopulationState )
        {
        case POPULATION_STATE_ALL:
            return hBitmap[14];
        case POPULATION_STATE_MANY:
            return hBitmap[15];
        case POPULATION_STATE_FEW:
            return hBitmap[16];
        case POPULATION_STATE_NONE:
            return hBitmap[20];
        }
    case ESTIMATE_STATE_POOR:
        switch( nPopulationState )
        {
        case POPULATION_STATE_ALL:
            return hBitmap[17];
        case POPULATION_STATE_MANY:
            return hBitmap[18];
        case POPULATION_STATE_FEW:
return hBitmap[19];

case POPULATION_STATE_NONE:
    return hBitmap[20];
}

case ESTIMATE_STATE_NONE:
    return hBitmap[20];
}

bmpmgr.h

/* bmpmgr.h - Header file for Bitmap Manager */

#ifndef BMPMGR_H
#define BMPMGR_H

#define STRICT
#define NOCOMM

#include <Windows.h>
#include <Windowsx.h>
#include "child.h"
#include "bmpmgr.h"
#include "populmgr.h"
#include "msmtdb.h"
#include "genedlgs.h"

#define WS_UNITCHILD ( WS_CHILD | WS_BORDER | WS_CLIPSIBLINGS )
#define CHILD_X 16
#define CHILD_Y 16

static BOOL  bIsTracking = FALSE;
static POINT  ptPickup, ptChild;
static RECT   rParent;
static int nChildIndex;

void Child_MouseHandler( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam );
void Child_PaintHandler( HWND hwnd );
void PaintUnitBitmap( HDC hdc, HBITMAP hBitmap );

#define ESTIMATE_STATE_EXCELLENT 900
#define ESTIMATE_STATE_FAIR 901
#define ESTIMATE_STATE_POOR 902
#define ESTIMATE_STATE_NONE 903
#define POPULATION_STATE_ALL 920
#define POPULATION_STATE_MANY 921
#define POPULATION_STATE_FEW 922
#define POPULATION_STATE_NONE 923

#endif

child.c
void GetChildCoordFromParent( HWND hParent, HWND hChild, LPPOINT ptChildCoord );
void Child_LoadPoint( LPPOINT ptPoint, int nX, int nY );
void Child_LoadPointFromLong( LPPOINT ptPoint, LONG lLong );
void Child_PrepareForTracking( HWND hwnd, LONG lParam );
void Child_CancelTracking( void );

BOOL RegisterUnitChildClass( HANDLE hInstance, LPSTR szChildClassName )
{
    WNDCLASS wc;
    wc.style = NULL;
    wc.lpfnWndProc = (WNDPROC) ChildWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = (HBRUSH) (COLOR_APPWORKSPACE + 1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = szChildClassName;
    return RegisterClass( &wc );
}

HWND CreateUnit( HWND hwnd, HANDLE hInst, LPSTR szClsName, LPPOINT xyPos )
{
    HWND hwndChild;
    hwndChild = CreateWindow( szClsName, NULL, WS_UNICHILD, xyPos->x, xyPos->y, CHILD_X, CHILD_Y, hwnd, NULL, hInst, NULL );
    ShowWindow( hwndChild, SW_SHOW );
    return hwndChild;
}

LRESULT CALLBACK _export ChildWndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_CREATE:
        UpdateWindow( hwnd );
        break;

    case WM_RBUTTONDOWN:
    case WM_LBUTTONDOWN:
    case WM_LBUTTONUP:
    case WM_MOUSEMOVE:
        Child_MouseHandler( hwnd, msg, wParam, lParam );
        break;

    case WM_PAINT:
        Child_PaintHandler( hwnd );
        break;

    case WM_DESTROY:
        break;
    }
default: return DefWindowProc (hwnd, msg, wParam, lParam);
}
return (0L);
}

void Child_PaintHandler( HWND hwnd )
{
PAINTSTRUCT ps;
BITMAP hBitmap;

BeginPaint( hwnd, &ps );
hBitmap = GetChildBitmapHandleFromHandle( hwnd );
PaintUnitBitmap( ps.hdc, hBitmap );
EndPaint( hwnd, &ps );
}

void PaintUnitBitmap( HDC hDC, HBITMAP hBitmap )
{
HDC hMemDC;

hMemDC = CreateCompatibleDC( hDC );
SelectObject( hMemDC, hBitmap );
BitBlt( hDC, 0, 0, 14, 14, hMemDC, 0, 0, SRCCOPY );
DeleteDC( hMemDC );
}

void Child_MouseHandler( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
int nNewX, nNewY;
POINT ptMouse;

switch (msg)
{
case WM_RBUTTONDOWN:
    ToggleSensorStateFromHandle( hwnd );
    InvalidateRect( hwnd, NULL, TRUE );
    break;

case WM_LBUTTONDOWN:
    if( (wParam & MK_SHIFT) == 0 )
    {
        Child_PrepareForTracking( hwnd, lParam );
    }
    SetCapture( hwnd );
    break;

case WM_LBUTTONUP:
    if( bIsTracking == TRUE )
    {
        Child_CancelTracking();
    }
    else
    {
        AgentOptionsDialogHandler( hwnd );
    }
    ReleaseCapture();
    break;

case WM_MOUSEMOVE:
    if( bIsTracking == TRUE )
    {
        Child_LoadPointFromLong( &ptMouse, lParam );
        nNewX = ptMouse.x-tpPickup.x+ptChild.x;
        nNewY = ptMouse.y-tpPickup.y+ptChild.y;
        if( nNewX < 0 ) nNewX = 0;
        if( nNewY < 0 ) nNewY = 0;
        if( nNewX > rParent.right ) nNewX = rParent.right;
        if( nNewY > rParent.bottom ) nNewY = rParent.bottom;
    }
void Child_LoadPoint( &ptChild, nNewX, nNewY );
MoveWindow( hwnd, nNewX, nNewY, CHILD_X, CHILD_Y, TRUE );

StoreScreenCoordinates( nChildIndex, &ptChild );
ConvertScreenToTrueAndSave( nChildIndex );
}
break;

default:
break;
}

void Child_CancelTracking( void )
{
    bisTracking = FALSE;
}

void Child_PrepareForTracking( HWND hwnd, LONG lParam )
HWND hParent;
    bisTracking = TRUE;
Child_LoadPointFromLong( hwnd, lParam );
    hParent = GetParent( hwnd );
    GetClientRect( hParent, &rParent );
rParent.right -= CHILD_X;
rParent.bottom -= CHILD_Y;
    GetChildCoordFromParent( hParent, hwnd, &ptChild );
nChildIndex = GetChildIndexFromHandle( hwnd );
}

void GetChildCoordFromParent( HWND hParent, HWND hChild, LPPOINT ptChildCoord )
    POINT ptParentScreen, ptChildScreen;
    ptParentScreen.x = 0;
    ptParentScreen.y = 0;
    ClientToScreen( hParent, &ptParentScreen );
    ptChildScreen.x = 0;
    ptChildScreen.y = 0;
    ClientToScreen( hChild, &ptChildScreen );
    ptChildCoord->x = ptChildScreen.x - ptParentScreen.x;
    ptChildCoord->y = ptChildScreen.y - ptParentScreen.y;
}

void Child_LoadPoint( LPPOINT ptPoint, int nX, int nY )
    ptPoint->x = nX;
    ptPoint->y = nY;
}

void Child_LoadPointFromLong( LPPOINT ptPoint, LONG lLong )
    ptPoint->x = LOWORD( lLong );
    ptPoint->y = HIWORD( lLong );
}

child.h

BOOL RegisterUnitChildClass( HANDLE hinstance, LPSTR szChildClassName );
HWND CreateUnit( HWND hwnd, HANDLE hInst, LPSTR szClsName, LPPOINT xyPos );
LRESULT CALLBACK _export ChildWndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);
dialog.h

#define IDD_ABOUTBOX 20

estimate.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include "estimate.h"
#include "pathdb.h"
#include "genetype.h"
#include "populmgr.h"
#include "glblopts.h"

#define FLOAT_ZERO ((float)0.0)
#define FLOAT_ONE  ((float)1.0)
#define FLOAT_TWO  ((float)2.0)
#define NEW_WT  ((float)0.6)
#define OLD_WT  ((FLOAT_ONE-NEW_WT)

/* used only to retrieve vectors */
static FCOORD fxyVectors[POPULATION_SIZE];

static float fRawX[POPULATION_SIZE];
static float fRawY[POPULATION_SIZE];
static float fRawScores[POPULATION_SIZE];
static float fSortWeights[POPULATION_SIZE];

BOOL Estimate_GetPathVectors( float *fx, float *fy, int *nNumElements, int nIndex );
BOOL Estimate_GetWeights( float *fWeights, int nIndex, int nNumExpectedElements );
BOOL Estimate_ExecuteMeanEstimator( int nIndex, FCOORD *fxyObservation );
BOOL Estimate_ExecuteModeMeanEstimator( int nIndex, FCOORD *fxyObservation );
BOOL Estimate_ExecuteModeMedianEstimator( int nIndex, FCOORD *fxyObservation );
void Estimate_SortFloatArrayWithWeight( float *fCoordinates, float *fWeights, int nNumElements );
void Estimate_ComputeMeanEstimate( float *fOut, float *fin, float *fWeights, int nNumElements );
void Estimate_ComputeMedianEstimate( float *fOut, float *fin, float *fWeights, int nNumElements );
BOOL Estimate_IsArgEven( int nArg );
BOOL Estimate_SumArray( float *fSum, float *fArray, int nNumElements );
void Estimate_LinearlyInterpolate( float *fEstimate, float fM1, float fM2, float fW1, float fW2 );
void Estimate_CopyArray( float *fOut, float *fIn, int nNumElements );
void Estimate_FindShorth( int *nShorthSize, float *fCoordinates, float *fWeights, int nNumElements );
int Estimate_CalculateShorthLength( int nN );

BOOL ProcessEstimateSlice( int nIndex )
{
    BOOL bIsValid,
    FCOORD fxyNewEstimate, fxyOldEstimate, fxyObservation;

    bIsValid = GetObservationFromIndex( nIndex, &fxyObservation );
    if( bIsValid == FALSE )
    {
        InvalidateEstimateFromIndex( nIndex );
        UpdateEstimateStateToNoneFromIndex( nIndex );
    }
BOOL UpdateChildBitmapHandleFromIndex(int nIndex);
return FALSE;
}

bIsValid = GetEstimateFromIndex(nIndex, &fxyOldEstimate);
if( bIsValid == FALSE )
{
    fxyNewEstimate.x = fxyObservation.x;
    fxyNewEstimate.y = fxyObservation.y;
}
else
{
    fxyNewEstimate.x = (NEW_WT*fxyObservation.x) + (OLD_WT*fxyOldEstimate.x);
    fxyNewEstimate.y = (NEW_WT*fxyObservation.y) + (OLD_WT*fxyOldEstimate.y);
}

StoreEstimateFromIndex(nIndex, &fxyNewEstimate);
UpdateEstimateStateFromIndex(nIndex, &fxyNewEstimate);
UpdateChildBitmapHandleFromIndex(nIndex);
return TRUE;

BOOL ProcessObservationSlice(int nIndex)
{
    BOOL bValid;
    FCOORD fxyObservation;
    int nEstimator;
    nEstimator = GetEstimator();
    switch(nEstimator)
    {
        case ESTIMATOR_MEAN:
            bValid = Estimate_ExecuteMeanEstimator(nIndex, &fxyObservation);
            break;
        case ESTIMATOR_MEDIAN:
            bValid = Estimate_ExecuteMedianEstimator(nIndex, &fxyObservation);
            break;
        case ESTIMATOR_MODE_MEAN:
            bValid = Estimate_ExecuteModeMeanEstimator(nIndex, &fxyObservation);
            break;
        case ESTIMATOR_MODE_MEDIAN:
            bValid = Estimate_ExecuteModeMedianEstimator(nIndex, &fxyObservation);
            break;
        default:
            break;
    }
    if( bValid == FALSE ) return FALSE;
    StoreObservationFromIndex(nIndex, &fxyObservation);
    return TRUE;
}

BOOL Estimate_ExecuteMeanEstimator(int nIndex, FCOORD *fxyObservation)
{
    int nNumElements;
    BOOL bIsValid;
    float fX, fY;
    bIsValid = Estimate_GetPathVectors(&fRawX[0], &fRawY[0], &nNumElements, nIndex);
    if( bIsValid == FALSE ) return FALSE;
    if( nNumElements == 0 )
    {
        InvalidateObservationFromIndex(nIndex);
        return FALSE;
    }
    }
bIsValid = Estimate_GetWeights( &fRawWeights[0], nIndex, nNumElements );
if( bIsValid == FALSE ) return FALSE;

/* Estimate X coordinate */
Estimate_ComputeMeanEstimate( &fX, &fRawX[0], &fRawWeights[0], nNumElements );
fxObservation->x = fX;

/* Estimate Y coordinate */
Estimate_ComputeMeanEstimate( &fY, &fRawY[0], &fRawWeights[0], nNumElements );
fxObservation->y = fY;
return TRUE;

BOOL Estimate_ExecuteMedianEstimator( int nIndex, FCOORD *fxObservation )
{
    int nNumElements;
    BOOL bIsValid;
    float fx, fy;

    bIsValid = Estimate_GetPathVectors( &fRawX[0], &fRawY[0], &nNumElements, nIndex );
    if( bIsValid == FALSE ) return FALSE;
    if( nNumElements == 0 )
    {
        InvalidateObservationFromIndex( nIndex );
        return FALSE;
    }

    bIsValid = Estimate_GetWeights( &fRawWeights[0], nIndex, nNumElements );
    if( bIsValid == FALSE ) return FALSE;

    /* Estimate X coordinate */
    Estimate_CopyArray( &fSortWeights[0], &fRawWeights[0], nNumElements );
    Estimate_ComputeMedianEstimate( &fX, &fRawX[0],
    &fSortWeights[0], nNumElements );
    fxObservation->x = fX;

    /* Estimate Y coordinate */
    Estimate_CopyArray( &fSortWeights[0], &fRawWeights[0], nNumElements );
    Estimate_ComputeMedianEstimate( &fY, &fRawY[0],
    &fSortWeights[0], nNumElements );
    fxObservation->y = fY;
return TRUE;
}

BOOL Estimate_ExecuteModeMeanEstimator( int nIndex, FCOORD *fxObservation )
{
    int nNumElements, nShorthSize;
    BOOL bIsValid;
    float fx, fy;

    bIsValid = Estimate_GetPathVectors( &fRawX[0], &fRawY[0], &nNumElements, nIndex );
    if( bIsValid == FALSE ) return FALSE;
    if( nNumElements == 0 )
    {
        InvalidateObservationFromIndex( nIndex );
        return FALSE;
    }

    bIsValid = Estimate_GetWeights( &fRawWeights[0], nIndex, nNumElements );
    if( bIsValid == FALSE ) return FALSE;

    /* Estimate X coordinate */
    Estimate_CopyArray( &fSortWeights[0], &fRawWeights[0], nNumElements );
    Estimate_FindShorth( &nShorthSize, &fRawX[0], &fSortWeights[0], nNumElements );
    Estimate_ComputeMeanEstimate( &fX, &fRawX[0], &fSortWeights[0], nShorthSize );
bool fxyObservation->x = fx;

/* Estimate Y coordinate */
Estimate_CopyArray( &fSortWeights[0], &fRawWeights[0], nNumElements );
Estimate_FindShorth( &nShorthSize, &fRawY[0], &fSortWeights[0], nNumElements );
Estimate_ComputeMeanEstimate( &fY, &fRawY[0], &fSortWeights[0], nShorthSize );
fxyObservation->y = fY;

return TRUE;

void Estimate_ExecuteModeMedianEstimator( int nIndex, FCOORD *fxyObservation )
{
    BOOL bIsValid;
    float fx, fy;

    bIsValid = Estimate_GetPathVectors( &fRawX[0], &fRawY[0], &nNumElements, nIndex );
    if( bIsValid == FALSE ) return FALSE;
    if( nNumElements == 0 )
    {
        InvalidateObservationFromIndex( nIndex );
        return FALSE;
    }

    bIsValid = Estimate_GetWeights( &fRawWeights[0], nIndex, nNumElements );
    if( bIsValid == FALSE ) return FALSE;

    /* Estimate X coordinate */
    Estimate_CopyArray( &fSortWeights[0], &fRawWeights[0], nNumElements );
    Estimate_FindShorth( &nShorthSize, &fRawX[0], &fSortWeights[0], nNumElements );
    Estimate_ComputeMedianEstimate( &fX, &fRawX[0], &fSortWeights[0], nShorthSize );
    fxyObservation->x = fX;

    /* Estimate Y coordinate */
    Estimate_CopyArray( &fSortWeights[0], &fRawWeights[0], nNumElements );
    Estimate_FindShorth( &nShorthSize, &fRawY[0], &fSortWeights[0], nNumElements );
    Estimate_ComputeMedianEstimate( &fY, &fRawY[0], &fSortWeights[0], nShorthSize );
    fxyObservation->y = fY;

    return TRUE;
}

void Estimate_FindShorth( int *nShorthSize, float *fCoordinates,
float *fWeights, int nNumElements )
{
    int i, nShorthLength, nNumTests, nBestShorth;
    int nLeft, nRight;
    float fBestLength, fLength;
    BOOL bMultipleShorths;

    if( nNumElements <= 2 )
    {
        *nShorthSize = nNumElements;
        return;
    }

    Estimate_SortFloatArrayWithWeight( fCoordinates, fWeights, nNumElements );
    nShorthLength = Estimate_CalculateShorthLength( nNumElements );
    nNumTests = nNumElements - nShorthLength + 1;
    nBestShorth = 0;
    fBestLength = fCoordinates[nShorthLength-1] - fCoordinates[0];
    bMultipleShorths = FALSE;
    nLeft = 0;

}
for( i = l; i < nNumTests; i++ )
{
    fLength = fCoordinates[i+nShorthLength-1] - fCoordinates[i];
    if( fLength == fBestLength )
    {
        bMultipleShorths = TRUE;
        nRight = i+nShorthLength-1;
    }
    if( fLength < fBestLength )
    {
        fBestLength = fLength;
        nBestShorth = i;
        nLeft = i;
        bMultipleShorths = FALSE;
    }
}

if( bMultipleShorths == TRUE )
{
    nShorthLength = nRight - nLeft + l;
}

for( i = 0; i < nShorthLength; i++ )
{
    fCoordinates[i] = fCoordinates[i+nBestShorth];
    fWeights[i] = fCoordinates[i+nBestShorth];
}
*nShorthSize = nShorthLength;

void Estimate_ComputeMeanEstimate( float *fOut, float *fIn,
float *fWeights, int nNumElements )
{
    int i;
    float fCoordinateSum, fWeightSum, fEstimate;

    fCoordinateSum = FLOAT_ZERO;
    fWeightSum = FLOAT_ZERO;

    for( i = 0; i < nNumElements; i++ )
    {
        fCoordinateSum += fIn[i]*fWeights[i];
        fWeightSum += fWeights[i];
    }
    fEstimate = fCoordinateSum/fWeightSum;
    *fOut = fEstimate;
}

void Estimate_ComputeMedianEstimate( float *fOut, float *fIn,
float *fWeights, int nNumElements )
{
    int i;
    float fTotalWeight, fHalfWeight, fSum, fEstimate;
    float fMl, fm2, fW1, fW2;

    Estimate_SortFloatArrayWithWeight( fIn, fWeights, nNumElements );
    Estimate_SumArray( &fTotalWeight, fWeights, nNumElements );
    fHalfWeight = fTotalWeight/FLOAT_TWO;

    fSum = fWeights[0];
    if( fSum >= fHalfWeight )
    {
        *fOut = fIn[0];
        return;
    }

    for( i = 1; i < nNumElements; i++ )
    {
        fSum += fWeights[i];
        if( fSum == fHalfWeight )
        {
            fEstimate = fIn[i];
            break;
        }
    }
}
if( fSum > fHalfWeight )
{
    fM1 = fin[i-1];
    fM2 = fin[i];
    fW1 = fWeights[i-1];
    fW2 = fWeights[i];
    Estimate_LinearlyInterpolate( &fEstimate, fM1, fM2, fW1, fW2 );
    break;
}
*fOut = fEstimate;

Estimate_SortFloatArrayWithWeight( float *fCoordinates, float *fWeights, int nNumElements )
{
    int i, j, k;
    float fTempCoordinate, fTestCoordinate, fTempWeight;
    for( i = 1; i < nNumElements; i++ )
    {
        fTempCoordinate = fCoordinates[i];
        for( j = 0; j < i; j++ )
        {
            fTestCoordinate = fCoordinates[j];
            if( fTempCoordinate > fTestCoordinate ) continue;

            fTempWeight = fWeights[i];
            for( k = i; k > j; k-- )
            {
                fWeights[k] = fWeights[k-1];
                fCoordinates[k] = fCoordinates[k-1];
            }
            fWeights[j] = fTempWeight;
            fCoordinates[j] = fTempCoordinate;
            break;
        }
    }
}

BOOL Estimate_SumArray( float *fSum, float *fArray, int nNumElements )
{
    float fTotal;
    int i;
    *fSum = FLOAT_ZERO;
    if( nNumElements <= 0 ) return FALSE;
    if( nNumElements > POPULATION_SIZE ) return FALSE;

    fTotal = FLOAT_ZERO;
    for( i = 0; i < nNumElements; i++ )
    {
        fTotal += fArray[i];
    }
    *fSum = fTotal;
    return TRUE;
}

BOOL Estimate_GetPathVectors( float *fX, float *fY, int *nNumElements, int nIndex )
{
    int i, nNumPaths;
    BOOL bPathsValid;
    bPathsValid = GetPathVectorsFromIndex( nIndex, &fxyVectors[0], &nNumPaths );
    if( bPathsValid == FALSE ) return FALSE;
    if( nNumPaths < 0 ) return FALSE;
    if( nNumPaths > POPULATION_SIZE ) return FALSE;
    *nNumElements = nNumPaths;
```c
for( i = 0; i < nNumPaths; i++ )
{
    fX[i] = fxyVectors[i].x;
    fY[i] = fxyVectors[i].y;
}
return TRUE;

BOOL Estimate_GetWeights( float *fWeights, int nIndex, int nNumExpectedElements )
{
    int i, nNumScores;
    BOOL bScoresValid;
    float fScore;
    bScoresValid = GetFitnessScoresFromIndex( nIndex, fWeights, &nNumScores );
    if( bScoresValid == FALSE ) return FALSE;
    if( nNumExpectedElements != nNumScores ) return FALSE;
    for( i = 0; i < nNumScores; i++ )
    {
        fScore = fWeights[i];
        if( fScore <= FLOAT_ZERO ) return FALSE;
        fWeights[i] = FLOAT_ONE/fScore;
    }
    return TRUE;
}

void Estimate_LinearlyInterpolate( float *fEstimate, float fM1, float fM2,
                float fW1, float fW2 )
{
    float fResult;
    fResult = (fM1*fW1 + fM2*fW2)/(fW1 + fW2);
    *fEstimate = fResult;
}

int Estimate_CalculateShorthLength( int nN )
{
    int nD;
    nD = 1 + nN/2;
    return nD;
}

BOOL Estimate_IsArgEven( int nArg )
{
    int nHalfArg, n2xHalfArg;
    nHalfArg = nArg/2;
    n2xHalfArg = nHalfArg + nHalfArg;
    if( n2xHalfArg == nArg ) return TRUE;
    return FALSE;
}

void Estimate_CopyArray( float *fOut, float *fin, int nNumElements )
{
    int i;
    for( i = 0; i < nNumElements; i++ )
    {
        fOut[i] = fin[i];
    }
}

estimate.h

#ifndef ESTIMATE_H
#define ESTIMATE_H

BOOL ProcessEstimateSlice( int nIndex );
BOOL ProcessObservationSlice( int nIndex );

#endif
```
expmt.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <Windowsx.h>
#include <stdio.h>
#include "expmt.h"
#include "resource.h"
#include "numunits.h"
#include "populmgr.h"
#include "glblopts.h"
#include "pathdb.h"
#define NUMBER_OF_ITERATIONS 30
#define NUMBER_OF_SETS 10

static BOOL bCollectingData = FALSE;
static int nSetNumber = 0;
static int nIterationNumber = 0;
static double dAvgPopSize[NUMBER_OF_ITERATIONS], dAvgFitness[NUMBER_OF_ITERATIONS];
static double dAvgEstError[NUMBER_OF_ITERATIONS], dNumEstimates[NUMBER_OF_ITERATIONS];
static int nExperimentType;

void Expmt_ActivateAllUnits( void );
void Expmt_CreateDeepPattern( void );
void Expmt_SendMessageToUnit( int nindex, WORD wMsg, WORD wParam, LONG lParam );
void Expmt_OperatorSet1( void );
void Expmt_OperatorSet2( void );
void Expmt_OperatorSet3( void );
void Expmt_DeselectAllOperators( void );
void Expmt_SaveExperimentData( void );

void CollectData( HWND hwnd )
{
    double dPopSize, dFitness, dError;
    int nNumEstimates;

    if( bCollectingData == FALSE ) return;
    GetAveragePopulationSize( &dPopSize );
    dAvgPopSize[nIterationNumber] += dPopSize;
    GetAverageFitnessScore( &dFitness );
    dAvgFitness[nIterationNumber] += dFitness;
    GetAverageEstimateError( &dError, &nNumEstimates );
    dAvgEstError[nIterationNumber] += dError;
    dNumEstimates[nIterationNumber] += ((double)nNumEstimates);
    nIterationNumber++;
    if( nIterationNumber == NUMBER_OF_ITERATIONS )
    {
        SendMessage( hwnd, WM_COMMAND, IDM_LOBOTOMY, 0L );
        nSetNumber++;
        nIterationNumber = 0;
    }
    if( nSetNumber == NUMBER_OF_SETS )
    {
        Expmt_SaveExperimentData();
        bCollectingData = FALSE;
        MessageBox( hwnd, "Complete", "Experiment Status", MB_OK );
        InvalidateRect( hwnd, NULL, FALSE );
        UpdateWindow( hwnd );
    }
    UpdatePacifer( hwnd );
}
void UpdatePacifer( HWND hwnd )
{
    HDC hDC;
    char szPacifer[64];

    if( bCollectingData == FALSE ) return;
    sprintf( szPacifer, "Set \d of \d Iteration \d of \d ",
             (nSetNumber+1),NUMBER_OF_SETS,(nIterationNumber+1),NUMBER_OF_ITERATIONS );
    hDC = GetDC( hwnd );
    SetTextColor( hDC, RGB( 128, 128, 128) );
    SetBkColor( hDC, RGB( 0, 0, 0) );
    TextOut( hDC, 0, 0, szPacifer, lstrlen( szPacifer ) );
    ReleaseDC( hwnd, hDC );
}

void Expmt_SaveExperimentData( void )
{
    int i;
    double dPopSize, dFitness, dEstError, dAvgNumEstimates;
    FILE *fpOutFile;
    char szOutString[128];

    sprintf( szOutString, "gene%d.dat", nExperimentType );
    fpOutFile = fopen( szOutString, "w" );
    fprintf(fpOutFile,""Experiment %d\n", nExperimentType);
    fprintf(fpOutFile, "\Pop Size\","\Fitness\","Est Error","Num Estimates\n"");
    for( i = 0; i < NUMBER_OF_ITERATIONS; i++ )
    {
        dPopSize = dAvgPopSize[i]/((double)NUMBER_OF_SETS);
        dFitness = dAvgFitness[i]/((double)NUMBER_OF_SETS);
        dEstError = dAvgEstError[i]/((double)NUMBER_OF_SETS);
        dAvgNumEstimates = dNumEstimates[i]/((double)NUMBER_OF_SETS);
        fprintf( fpOutFile, "%.1f,%.1f,%.1f,%.1f\n",
                  dPopSize, dFitness, dEstError, dAvgNumEstimates );
    }
    fclose(fpOutFile);
}

void ExperimentOptionsHandler( HWND hwnd, WPARAM wParam )
{
    int i;
    BOOL bIsPaused;

    nExperimentType = ((int)wParam);
    Expmt_ActivateAllUnits();
    Expmt_OperatorSet1();
    SetCommLossToNone();
    SetErrorScaleToGood();
    SetEstimatorToMedian();
    switch( wParam )
    {
    case IDM_EXPMT_OWS1:
        Expmt_OperatorSet1();
        break;
    case IDM_EXPMT_OWS2:
        Expmt_OperatorSet2();
        break;
    case IDM_EXPMT_OWS3:
        break;
    case IDM_EXPMT_ODS2:
        Expmt_CreateDeepPattern();
        Expmt_OperatorSet2();
        break;
    }
case IDM_EXPMT_ODS3:
    Expmt_CreateDeepPattern();
    break;

case IDM_EXPMT_SWPOOR:
    SetErrorScaleToPoor();
    break;

case IDM_EXPMT_SWFAIR:
    SetErrorScaleToFair();
    break;

case IDM_EXPMT_SWGOOD:
    break;

case IDM_EXPMT_SDPOOR:
    Expmt_CreateDeepPattern();
    SetErrorScaleToPoor();
    break;

case IDM_EXPMT_SDFAIR:
    Expmt_CreateDeepPattern();
    SetErrorScaleToFair();
    break;

case IDM_EXPMT_SDGOOD:
    Expmt_CreateDeepPattern();
    break;

case IDM_EXPMT_CW5:
    SetCommLossTo5Percent();
    break;

case IDM_EXPMT_CW20:
    SetCommLossTo20Percent();
    break;

case IDM_EXPMT_CW40:
    SetCommLossTo40Percent();
    break;

case IDM_EXPMT_CD5:
    Expmt_CreateDeepPattern();
    SetCommLossTo5Percent();
    break;

case IDM_EXPMT_CD20:
    Expmt_CreateDeepPattern();
    SetCommLossTo20Percent();
    break;

case IDM_EXPMT_CD40:
    Expmt_CreateDeepPattern();
    SetCommLossTo40Percent();
    break;

case IDM_EXPMT_EST1:
    SetErrorScaleToFair();
    SetCommLossTo20Percent();
    SetEstimatorToMean();
    break;

case IDM_EXPMT_EST2:
SetErrorScaleToFair();
SetCommLossTo20Percent();
break;

case IDM_EXPMT_EST3:
    SetErrorScaleToFair();
    SetCommLossTo20Percent();
    SetEstimatorToModeMean();
    break;

case IDM_EXPMT_EST4:
    SetErrorScaleToFair();
    SetCommLossTo20Percent();
    SetEstimatorToModeMedian();
    break;

case IDM_EXPMT_MOTION:
    break;

bCollectingData = TRUE;
nSetNumber = 0;
nIterationNumber = 0;

for( i = 0; i < NUMBER_OF_ITERATIONS; i++ )
{
    dAvgPopSize[i] = 0.0;
    dAvgFitness[i] = 0.0;
    dAvgEstError[i] = 0.0;
    SendMessage( hwnd, WM_COMMAND, IDM_LOBOTOMY, 0L );
    bIsPaused = GetGlobalPausedState();
    if( bIsPaused == TRUE )
        SendMessage( hwnd, WM_COMMAND, IDM_PAUSE, 0L );
}
UpdatePacifer( hwnd );

void Expmt_OperatorSet1( void )
{
    Expmt_DeselectAllOperators();
    ToggleReproductionRandom();
    ToggleReproductionCrossover();
    ToggleReproductionLinear();
    ToggleReproductionCircular();
}

void Expmt_OperatorSet2( void )
{
    Expmt_DeselectAllOperators();
    ToggleReproductionDirect();
    ToggleReproductionOutside();
}

void Expmt_OperatorSet3( void )
{
    Expmt_DeselectAllOperators();
    ToggleReproductionDirect();
    ToggleReproductionInside();
    ToggleReproductionOutside();
    ToggleReproductionReduction();
}

void Expmt_DeselectAllOperators( void )
{
    BOOL bState;
    bState = GetReproductionDirectState();
    if( bState == TRUE )
void ToggleReproductionDirect();

bState = GetReproductionRandomState();
if( bState == TRUE )
    ToggleReproductionRandom();

bState = GetReproductionInsideState();
if( bState == TRUE )
    ToggleReproductionInside();

bState = GetReproductionOutsideState();
if( bState == TRUE )
    ToggleReproductionOutside();

bState = GetReproductionCrossoverState();
if( bState == TRUE )
    ToggleReproductionCrossover();

bState = GetReproductionLinearState();
if( bState == TRUE )
    ToggleReproductionLinear();

bState = GetReproductionCircularState();
if( bState == TRUE )
    ToggleReproductionCircular();

bState = GetReproductionReductionState();
if( bState == TRUE )
    ToggleReproductionReduction();

void Expmt_ActivateAllUnits( void )
{
    int i;
    BOOL bSensorState;

    for( i = 1; i < NUMBER_OF_UNITS; i++ )
    {
        GetSensorStateFromIndex( i, &bSensorState );
        if( bSensorState == FALSE )
            Expmt_SendMessageToUnit( i, WM_RBUTTONDOWN, 0, 0L );
    }
}

void Expmt_CreateDeepPattern( void )
{
    Expmt_SendMessageToUnit( 2, WM_RBUTTONDOWN, 0, 0L );
    Expmt_SendMessageToUnit( 10, WM_RBUTTONDOWN, 0, 0L );
    Expmt_SendMessageToUnit( 18, WM_RBUTTONDOWN, 0, 0L );
    Expmt_SendMessageToUnit( 26, WM_RBUTTONDOWN, 0, 0L );
    Expmt_SendMessageToUnit( 13, WM_RBUTTONDOWN, 0, 0L );
    Expmt_SendMessageToUnit( 21, WM_RBUTTONDOWN, 0, 0L );
    Expmt_SendMessageToUnit( 29, WM_RBUTTONDOWN, 0, 0L );
    Expmt_SendMessageToUnit( 37, WM_RBUTTONDOWN, 0, 0L );
void Expmt_SendMessageToUnit( int nIndex, WORD wMsg, WORD wParam, LONG lParam )
{
    HWND hwndUnit;
    hwndUnit = GetChildHandleFromIndex( nIndex );
    if( hwndUnit == NULL ) return;
    SendMessage( hwndUnit, wMsg, wParam, lParam );
}

expmt.h

#ifndef EXPMT_H
#define EXPMT_H

void CollectData( HWND hwnd );
void ExperimentOptionsHandler( HWND hwnd, WPARAM wParam );
void UpdatePacifer( HWND hwnd );
#endif

fcoord.h

#ifndef FCOORD_H
#define FCOORD_H

/* Structure typing for a floating point x,y coordinate */
typedef struct tagFCOORD
{
    float x;
    float y;
} FCOORD;

typedef FCOORD *PFCOORD;
typedef FCOORD NEAR *NPFCOORD;
typedef FCOORD FAR *LPFCOORD;
#endif

genedlg1.dlg

DLGINCLUDE RCDATA DISCARDABLE
BEGIN
    "GENEDLG1.H\0"
END

IDD_DISPLAY DIALOG 109, 54, 131, 86
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Display Options"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL "Estimates", IDD_DISPLAY_ESTIMATES, "Button",
        BS_AUTOCHECKBOX | WS_TABSTOP, 39, 8, 46, 10
    CONTROL "Observations", IDD_DISPLAY_OBSERVATIONS, "Button",
        BS_AUTOCHECKBOX | WS_TABSTOP, 39, 21, 58, 10
    CONTROL "Measurements", IDD_DISPLAY_MEASUREMENTS, "Button",
        BS_AUTOCHECKBOX | WS_TABSTOP, 39, 34, 62, 10
    CONTROL "Paths", IDD_DISPLAY_PATHS, "Button", BS_AUTOCHECKBOX |
        WS_TABSTOP, 39, 47, 40, 10
    CONTROL "OK", IDOK, "Button", WS_TABSTOP, 45, 66, 40, 14
END
definedlg1.h

#define IDD_DISPLAY 1100
#define IDD_DISPLAY_ESTIMATES 1101
#define IDD_DISPLAY_PATHS 1103
#define IDD_DISPLAY_OBSERVATIONS 1102
#define IDD_DISPLAY_MEASUREMENTS 1104

definedlg2.dlg

DLGINCLUDE RCDATA DISCARDABLE
BEGIN
"GENEDLG2.H"
END

IDD_ERROR DIALOG 100, 44, 160, 169
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Error Parameters"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL
        "0.1 meters / 2.5°", IDD_NOISE_GOOD, "Button", BS_AUTORADIOBUTTON | WS_GROUP, 36, 23, 64, 10
        "0.5 meters / 7.5°", IDD_NOISE_FAIR, "Button", BS_AUTORADIOBUTTON, 36, 33, 66, 10
        "1.0 meters / 15°", IDD_NOISE_POOR, "Button", BS_AUTORADIOBUTTON, 36, 43, 64, 10
        "None", IDD_LOSS_NONE, "Button", BS_AUTORADIOBUTTON | WS_GROUP, 36, 89, 30, 10
        "5 Percent", IDD_LOSS_5, "Button", BS_AUTORADIOBUTTON, 36, 99, 44, 10
        "20 Percent", IDD_LOSS_20, "Button", BS_AUTORADIOBUTTON, 36, 109, 48, 10
        "40 Percent", IDD_LOSS_40, "Button", BS_AUTORADIOBUTTON, 36, 119, 48, 10
        "Noise Parameters", IDD_NOISE, "Button", BS_GROUPBOX | WS_GROUP, 21, 5, 117, 58
        "Loss Parameters", IDD_LOSS, "Button", BS_GROUPBOX, 21, 71, 117, 70
        "OK", IDOK, "Button", WS_TABSTOP, 60, 149, 40, 14
END

definedlg2.h

#define IDD_ERROR 1200
#define IDD_NOISE_FAIR 1212
#define IDD_NOISE_POOR 1213
#define IDD_LOSS_NONE 1221
#define IDD_LOSS_5 1222
#define IDD_LOSS_20 1223
#define IDD_LOSS_40 1224
#define IDD_NOISE_GOOD 1211
#define IDD_LOSS 1220
#define IDD_NOISE 1210

definedlg3.dlg

DLGINCLUDE RCDATA DISCARDABLE
BEGIN
"GENEDLG3.H\0"

END

IDD_OPERATORS DIALOG 141, 109, 163, 143
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Genetic Algorithm Operator Suite"
FONT 8, "MS Sans Serif"
BEGIN
CONTROL "Direct Search", IDD_OPERATORS_DIRECT, "Button",
BS_AUTOCHECKBOX | WS_TABSTOP, 35, 15, 59, 10
CONTROL "Random Generation", IDD_OPERATORS_RANDOM, "Button",
BS_AUTOCHECKBOX | WS_TABSTOP, 35, 24, 81, 10
CONTROL "Inside Matching", IDD_OPERATORS_INSIDE, "Button",
BS_AUTOCHECKBOX | WS_TABSTOP, 35, 37, 66, 10
CONTROL "Outside Matching", IDD_OPERATORS_OUTSIDE, "Button",
BS_AUTOCHECKBOX | WS_TABSTOP, 35, 50, 71, 10
CONTROL "Crossover", IDD_OPERATORS_CROSSOVER, "Button",
BS_AUTOCHECKBOX | WS_TABSTOP, 35, 63, 47, 10
CONTROL "Linear Inversion", IDD_OPERATORS_LINEAR, "Button",
BS_AUTOCHECKBOX | WS_TABSTOP, 35, 76, 67, 10
CONTROL "Circular Inversion", IDD_OPERATORS_CIRCULAR, "Button",
BS_AUTOCHECKBOX | WS_TABSTOP, 35, 89, 72, 10
CONTROL "Randomized Reduction", IDD_OPERATORS_REDUCTION,
"Button", BS_AUTOCHECKBOX | WS_TABSTOP, 35, 102, 92, 10
CONTROL "OK", IDOK, "Button", WS_TABSTOP, 61, 123, 40, 14
END

#define IDD_OPERATORS 1400
#define IDD_OPERATORS_DIRECT 1401
#define IDD_OPERATORS_RANDOM 1402
#define IDD_OPERATORS_INSIDE 1403
#define IDD_OPERATORS_OUTSIDE 1404
#define IDD_OPERATORS_CROSSOVER 1405
#define IDD_OPERATORS_LINEAR 1406
#define IDD_OPERATORS_CIRCULAR 1407
#define IDD_OPERATORS_REDUCTION 1408

genedlg3.h

genedlg4.dlg
CONTROL "Show No Paths", IDD_AGENT_PATHS_NONE, "Button", BS_AUTORADIOBUTTON, 125, 13, 64, 10
CONTROL "Show 1 Path", IDD_AGENT_PATHS_1, "Button", BS_AUTORADIOBUTTON, 125, 23, 64, 10
CONTROL "Show 5 Paths", IDD_AGENT_PATHS_5, "Button", BS_AUTORADIOBUTTON, 125, 33, 64, 10
CONTROL "Show 15 Paths", IDD_AGENT_PATHS_15, "Button", BS_AUTORADIOBUTTON, 125, 43, 64, 10
CONTROL "Show All Paths", IDD_AGENT_PATHS_ALL, "Button", BS_AUTORADIOBUTTON, 125, 53, 63, 10
CONTROL "OK", IDOK, "Button", BS_GROUPBOX, 106, 106, 7, 100, 62
CONTROL "OK", IDOK, "Button", BS_GROUPBOX, 111, 7, 100, 62

#define IDD_AGENT 1500
#define IDD_AGENT_PATHS_NONE 1521
#define IDD_AGENT_PATHS_1 1522
#define IDD_AGENT_PATHS_5 1523
#define IDD_AGENT_PATHS_15 1524
#define IDD_AGENT_PATHS_ALL 1525
#define IDD_AGENT_ESTIMATES 1511
#define IDD_AGENT_OBSERVATIONS 1512
#define IDD_AGENT_MEASUREMENTS 1513
#define IDD_AGENT_STATISTICS 1530

#define IDD_ESTIMATOR 1600
#define IDD_EST_MEAN 1601
#define IDD_EST_MEDIAN 1602
#define IDD_EST_MODE_MEAN 1603
#define IDD_EST_MODE_MEDIAN 1604

DLGINCLUDE RCDATA DISCARDABLE
BEGIN
"GENEDLGS.H\0"
END

IDD_ESTIMATOR DIALOG 100, 44, 160, 169
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Location Estimator"
FONT 8, "MS Sans Serif"
BEGIN
CONTROL "Mean", IDD_EST_MEAN, "Button", BS_AUTORADIOBUTTON | WS_GROUP, 36, 23, 64, 10
CONTROL "Median", IDD_EST_MEDIAN, "Button", BS_AUTORADIOBUTTON | WS_GROUP, 36, 33, 64, 10
CONTROL "Mode-Mean", IDD_EST_MODE_MEAN, "Button", BS_AUTORADIOBUTTON, 36, 43, 64, 10
CONTROL "Mode-Median", IDD_EST_MODE_MEDIAN, "Button", BS_AUTORADIOBUTTON, 36, 53, 64, 10
CONTROL "OK", IDOK, "Button", WS_TABSTOP, 60, 149, 40, 14
END

#define IDD_E garnished text
```c
#define STRICT
#define NOCOMM
#include <windows.h>
#include <Windowsx.h>
#include "genedlgs.h"
#include "genedlg1.h"
#include "genedlg2.h"
#include "genedlg3.h"
#include "genedlg4.h"
#include "genedlg5.h"
#include "glibopts.h"
#include "instance.h"
#include "populmgr.h"

void GeneDlgs_UpdateDialogDisplayOptions(HWND hdlg);
void GeneDlgs_UpdateDialogErrorOptions(HWND hdlg);
void GeneDlgs_UpdateDialogEstimatorOptions(HWND hdlg);
void GeneDlgs_UpdateDialogNoiseOptions(HWND hdlg);
void GeneDlgs_UpdateDialogLossOptions(HWND hdlg);
void GeneDlgs_UpdateDialogOperatorOptions(HWND hdlg);
void GeneDlgs_UpdateDialogAgentOptions(HWND hdlg);

#pragma warning (disable:4100)
static HWND ghChild = NULL;

void DisplayOptionsDialogHandler(HWND hwnd)
{
  DLGPROC lpProc;
  HINSTANCE hInstance;

  hInstance = GetWindowInstance(hwnd);
  lpProc = (DLGPROC)MakeProcInstance((FARPROC)DisplayDialogBoxProc, hInstance);
  DialogBox(hInstance, MAKEINTRESOURCE(IDD_DISPLAY), hwnd, lpProc);
  FreeProcInstance((FARPROC)lpProc);
}

void ErrorOptionsDialogHandler(HWND hwnd)
{
  DLGPROC lpProc;
  HINSTANCE hInstance;

  hInstance = GetWindowInstance(hwnd);
  lpProc = (DLGPROC)MakeProcInstance((FARPROC)ErrorDialogBoxProc, hInstance);
  DialogBox(hInstance, MAKEINTRESOURCE(IDD_ERROR), hwnd, lpProc);
  FreeProcInstance((FARPROC)lpProc);
}

void EstimatorOptionsDialogHandler(HWND hwnd)
{
  DLGPROC lpProc;
  HINSTANCE hInstance;

  hInstance = GetWindowInstance(hwnd);
  lpProc = (DLGPROC)MakeProcInstance((FARPROC)EstimatorDialogBoxProc, hInstance);
  DialogBox(hInstance, MAKEINTRESOURCE(IDD_ESTIMATOR), hwnd, lpProc);
  FreeProcInstance((FARPROC)lpProc);
}

void OperatorOptionsDialogHandler(HWND hwnd)
{
  DLGPROC lpProc;
  HINSTANCE hInstance;
```
void hinstance = GetWindowInstance( hwnd );
lpProc = (DLGPROC)MakeProcInstance( (FARPROC)OperatorDialogBoxProc, hinstance );
DialogBox( hinstance, MAKEINTRESOURCE(IDD_OPERATORS), hwnd, lpProc );
FreeProcInstance((FARPROC) lpProc );
}

void AgentOptionsDialogHandler( HWND hChild )
{
    DLGPROC lpProc;
    HINSTANCE hinstance;

    ghChild = hChild;
    hinstance = GetWindowInstance( hChild );
    lpProc = (DLGPROC)MakeProcInstance( (FARPROC)AgentDialogBoxProc, hinstance );
    DialogBox( hinstance, MAKEINTRESOURCE(IDD_AGENT), hChild, lpProc );
    FreeProcInstance((FARPROC) lpProc );
}

BOOL CALLBACK _export
DisplayDialogBoxProc( HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch (msg)
    {
        case WM_INITDIALOG:
            GeneDlgs_UpdateDialogDisplayOptions( hdlg );
            return TRUE;

        case WM_COMMAND:
            switch( wParam
            {
                case IDD_DISPLAY_ESTIMATES:
                    ToggleGlobalEstimateDisplay();
                    return TRUE;

                case IDD_DISPLAY_OBSERVATIONS:
                    ToggleGlobalObservationDisplay();
                    return TRUE;

                case IDD_DISPLAY_PATHS:
                    ToggleGlobalPathDisplay();
                    return TRUE;

                case IDD_DISPLAY_MEASUREMENTS:
                    ToggleGlobalMeasurementDisplay();
                    return TRUE;

                case IDOK:
                    EndDialog( hdlg, TRUE );
                    return TRUE;
            }
    return FALSE;
}

BOOL CALLBACK _export
ErrorDialogBoxProc( HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch (msg)
    {
        case WM_INITDIALOG:
            GeneDlgs_UpdateDialogErrorOptions( hdlg );
            return TRUE;
    }
}
case WM_COMMAND:
    switch (wParam) {
        case IDD_NOISE_GOOD:
            SetErrorScaleToGood();
            return TRUE;
        case IDD_NOISE_FAIR:
            SetErrorScaleToFair();
            return TRUE;
        case IDD_NOISE_POOR:
            SetErrorScaleToPoor();
            return TRUE;
        case IDD_loss_NONE:
            SetCommLossToNone();
            return TRUE;
        case IDD_loss_5:
            SetCommLossTo5Percent();
            return TRUE;
        case IDD_loss_20:
            SetCommLossTo20Percent();
            return TRUE;
        case IDD_loss_40:
            SetCommLossTo40Percent();
            return TRUE;
        case IDOK:
            EndDialog(hdlg, TRUE);
            return TRUE;
    }
    return FALSE;

BOOL CALLBACK _export
EstimatorDialogBoxProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg) {
        case WM_COMMAND:
            switch (wParam) {
                case IDD_EST_MEAN:
                    SetEstimatorToMean();
                    return TRUE;
                case IDD_EST_MEDIAN:
                    SetEstimatorToMedian();
                    return TRUE;
            }
    }
    return FALSE;
}

147
case IDD_EST_MODE_MEDIAN:
    SetEstimatorToModeMedian();
    return TRUE;

    case IDOK:
        EndDialog( hdlg, TRUE );
        return TRUE;
    }
    return FALSE;
}

BOOL CALLBACK _export
OperatorDialogBoxProc( HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch (msg)
    {
        case WM_INITDIALOG:
            GeneDlg_UpdateDialogOperatorOptions( hdlg );
            return TRUE;

        case WM_COMMAND:
            switch( wParam
            {
                case IDD_OPERATORS_DIRECT:
                    ToggleReproductionDirect();
                    return TRUE;

                case IDD_OPERATORS_RANDOM:
                    ToggleReproductionRandom();
                    return TRUE;

                case IDD_OPERATORS_INSIDE:
                    ToggleReproductionInside();
                    return TRUE;

                case IDD_OPERATORS_OUTSIDE:
                    ToggleReproductionOutside();
                    return TRUE;

                case IDD_OPERATORS_CROSSOVER:
                    ToggleReproductionCrossover();
                    return TRUE;

                case IDD_OPERATORS_LINEAR:
                    ToggleReproductionLinear();
                    return TRUE;

                case IDD_OPERATORS_CIRCULAR:
                    ToggleReproductionCircular();
                    return TRUE;

                case IDD_OPERATORS_REDUCTION:
                    ToggleReproductionReduction();
                    return TRUE;

                case IDOK:
                    EndDialog( hdlg, TRUE );
                    return TRUE;

            }
    }

    return FALSE;
}
BOOL CALLBACK __export
AgentDialogBoxProc( HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch (msg)
    {
    case WM_INITDIALOG:
        GeneDlgs_UpdateDialogAgentOptions( hdlg );
        return TRUE;
    case WM_COMMAND:
        switch( wParam
        return FALSE;
    case IDD_AGENT_ESTIMATES:
        ToggleIndividualEstimateDisplayFromHandle( ghChild );
        return TRUE;
    case IDD_AGENT_OBSERVATIONS:
        ToggleIndividualObservationDisplayFromHandle( ghChild );
        return TRUE;
    case IDD_AGENT_MEASUREMENTS:
        ToggleIndividualMeasurementDisplayFromHandle( ghChild );
        return TRUE;
    case IDD_AGENT_PATHS_NONE:
        SetIndividualPathDisplayToNoneFromHandle( ghChild );
        return TRUE;
    case IDD_AGENT_PATHS_1:
        SetIndividualPathDisplayTo1FromHandle( ghChild );
        return TRUE;
    case IDD_AGENT_PATHS_5:
        SetIndividualPathDisplayTo5FromHandle( ghChild );
        return TRUE;
    case IDD_AGENT_PATHS_ALL:
        SetIndividualPathDisplayToAllFromHandle( ghChild );
        return TRUE;
    case IDD_AGENT_STATISTICS:
        MessageBox( hdlg, "Statistics Here",
                    "Agent Statistics", MB_OK );
        return TRUE;
    case IDOK:
        EndDialog( hdlg, TRUE );
        return TRUE;
    }
    return FALSE;
}

void GeneDlgs_UpdateDialogDisplayOptions( HWND hdlg )
{
    BOOL bState;

    bState = GetGlobalEstimateDisplayState();
    if( bState == TRUE )
void CheckDlgButton( hdlg, IDD_DISPLAY_ESTIMATES, MF_CHECKED );
else
void CheckDlgButton( hdlg, IDD_DISPLAY_ESTIMATES, MF_UNCHECKED );

bState = GetGlobalObservationDisplayState();
if( bState == TRUE )
{ 
    CheckDlgButton( hdlg, IDD_DISPLAY_OBSERVATIONS, MF_CHECKED );
} else
{ 
    CheckDlgButton( hdlg, IDD_DISPLAY_OBSERVATIONS, MF_UNCHECKED );
}

bState = GetGlobalMeasurementDisplayState();
if( bState == TRUE )
{ 
    CheckDlgButton( hdlg, IDD_DISPLAY_MEASUREMENTS, MF_CHECKED );
} else
{ 
    CheckDlgButton( hdlg, IDD_DISPLAY_MEASUREMENTS, MF_UNCHECKED );
}

bState = GetGlobalPathDisplayState();
if( bState == TRUE )
{ 
    CheckDlgButton( hdlg, IDD_DISPLAY_PATHS, MF_CHECKED );
} else
{ 
    CheckDlgButton( hdlg, IDD_DISPLAY_PATHS, MF_UNCHECKED );
}

void GeneDlgs_UpdateDialogErrorOptions( HWND hdlg )
{ 
    GeneDlgs_UpdateDialogNoiseOptions( hdlg );
    GeneDlgs_UpdateDialogLossOptions( hdlg );
}

void GeneDlgs_UpdateDialogEstimatorOptions( HWND hdlg )
{ 
    int nState;
    nState = GetEstimator();
    switch( nState )
    {
    case ESTIMATOR_MEAN:
        CheckRadioButton( hdlg, IDD_EST_MEAN, IDD_EST_MODE_MEDIAN, 
            IDD_EST_MEAN );
        break;
    case ESTIMATOR_MEDIAN:
        CheckRadioButton( hdlg, IDD_EST_MEAN, IDD_EST_MODE_MEDIAN, 
            IDD_EST_MEDIAN );
        break;
    case ESTIMATOR_MODE_MEAN:
        CheckRadioButton( hdlg, IDD_EST_MEAN, IDD_EST_MODE_MEDIAN, 
            IDD_EST_MODE_MEAN );
        break;
    case ESTIMATOR_MODE_MEDIAN:
        CheckRadioButton( hdlg, IDD_EST_MEAN, IDD_EST_MODE_MEDIAN, 
            IDD_EST_MODE_MEDIAN );
        break;
    }
void GeneDlgs_UpdateDialogLossOptions(HWND hdlg)
{
    int nState;

    nState = GetCommLoss();
    switch(nState)
    {
    case COMM_LOSS_NONE:
        CheckRadioButton(hdlg, IDD_LOSS_NONE, IDD_LOSS_40, IDD_LOSS_NONE);
        break;
    case COMM_LOSS_5_PERCENT:
        CheckRadioButton(hdlg, IDD_LOSS_NONE, IDD_LOSS_40, IDD_LOSS_5);
        break;
    case COMM_LOSS_20_PERCENT:
        CheckRadioButton(hdlg, IDD_LOSS_NONE, IDD_LOSS_40, IDD_LOSS_20);
        break;
    case COMM_LOSS_40_PERCENT:
        CheckRadioButton(hdlg, IDD_LOSS_NONE, IDD_LOSS_40, IDD_LOSS_40);
        break;
    }
}

void GeneDlgs_UpdateDialogNoiseOptions(HWND hdlg)
{
    int nState;

    nState = GetErrorScale();
    switch(nState)
    {
    case ERROR_SCALE_GOOD:
        CheckRadioButton(hdlg, IDD_NOISE_GOOD, IDD_NOISE_POOR, IDD_NOISE_GOOD);
        break;
    case ERROR_SCALE_FAIR:
        CheckRadioButton(hdlg, IDD_NOISE_GOOD, IDD_NOISE_POOR, IDD_NOISE_FAIR);
        break;
    case ERROR_SCALE_POOR:
        CheckRadioButton(hdlg, IDD_NOISE_GOOD, IDD_NOISE_POOR, IDD_NOISE_POOR);
        break;
    }
}

void GeneDlgs_UpdateDialogOperatorOptions(HWND hdlg)
{
    BOOL bState;

    bState = GetReproductionDirectState();
    if( bState == TRUE )
    {
        CheckDlgButton(hdlg, IDD_OPERATORS_DIRECT, MF_CHECKED);
    }
    else
    {
        CheckDlgButton(hdlg, IDD_OPERATORS_DIRECT, MF_UNCHECKED);
    }

    bState = GetReproductionRandomState();
    if( bState == TRUE )
    {
        CheckDlgButton(hdlg, IDD_OPERATORS_RANDOM, MF_CHECKED);
    }
    else
    {
{ CheckDlgButton( hdlg, IDD_OPERATORS_RANDOM, MF_UNCHECKED );
}

bState = GetReproductionInsideState();
if( bState == TRUE )
{ CheckDlgButton( hdlg, IDD_OPERATORS_INSIDE, MF_CHECKED );
} else
{ CheckDlgButton( hdlg, IDD_OPERATORS_INSIDE, MF_UNCHECKED );
}

bState = GetReproductionOutsideState();
if( bState == TRUE )
{ CheckDlgButton( hdlg, IDD_OPERATORS_OUTSIDE, MF_CHECKED );
} else
{ CheckDlgButton( hdlg, IDD_OPERATORS_OUTSIDE, MF_UNCHECKED );
}

bState = GetReproductionCrossoverState();
if( bState == TRUE )
{ CheckDlgButton( hdlg, IDD_OPERATORS_CROSSOVER, MF_CHECKED );
} else
{ CheckDlgButton( hdlg, IDD_OPERATORS_CROSSOVER, MF_UNCHECKED );
}

bState = GetReproductionCircularState();
if( bState == TRUE )
{ CheckDlgButton( hdlg, IDD_OPERATORS_CIRCULAR, MF_CHECKED );
} else
{ CheckDlgButton( hdlg, IDD_OPERATORS_CIRCULAR, MF_UNCHECKED );
}

bState = GetReproductionReductionState();
if( bState == TRUE )
{ CheckDlgButton( hdlg, IDD_OPERATORS_REDUCTION, MF_CHECKED );
} else
{ CheckDlgButton( hdlg, IDD_OPERATORS_REDUCTION, MF_UNCHECKED );
}


void GeneDlgs_UpdateDialogAgentOptions( HWND hdlg )
{
    BOOL bState;
    int nState;

    bState = GetIndividualEstimateDisplayStateFromHandle( ghChild );
    if( bState == TRUE )
    { CheckDlgButton( hdlg, IDD_AGENT_ESTIMATES, MF_CHECKED );
    } else
    { CheckDlgButton( hdlg, IDD_AGENT_ESTIMATES, MF_UNCHECKED );
    }
}
bState = GetIndividualObservationDisplayStateFromHandle( ghChild );
if( bState == TRUE )
{
    CheckDlgButton( hdlg, IDD_AGENT_OBSERVATIONS, MF_CHECKED );
}
else
{
    CheckDlgButton( hdlg, IDD_AGENT_OBSERVATIONS, MF_UNCHECKED );
}

bState = GetIndividualMeasurementDisplayStateFromHandle( ghChild );
if( bState == TRUE )
{
    CheckDlgButton( hdlg, IDD_AGENT_MEASUREMENTS, MF_CHECKED );
}
else
{
    CheckDlgButton( hdlg, IDD_AGENT_MEASUREMENTS, MF_UNCHECKED );
}

nState = GetIndividualPathDisplayFromHandle( ghChild );
switch( nState )
{
    case SHOW_NO_PATHS:
        CheckRadioButton( hdlg, IDD_AGENT_PATHS_NONE,
                         IDD_AGENT_PATHS_ALL, IDD_AGENT_PATHS_NONE );
        break;
    case SHOW_1_PATH:
        CheckRadioButton( hdlg, IDD_AGENT_PATHS_NONE,
                         IDD_AGENT_PATHS_ALL, IDD_AGENT_PATHS_1 );
        break;
    case SHOW_5_PATHS:
        CheckRadioButton( hdlg, IDD_AGENT_PATHS_NONE,
                         IDD_AGENT_PATHS_ALL, IDD_AGENT_PATHS_5 );
        break;
    case SHOW_15_PATHS:
        CheckRadioButton( hdlg, IDD_AGENT_PATHS_NONE,
                         IDD_AGENT_PATHS_ALL, IDD_AGENT_PATHS_15 );
        break;
    case SHOW_ALL_PATHS:
        CheckRadioButton( hdlg, IDD_AGENT_PATHS_NONE,
                         IDD_AGENT_PATHS_ALL, IDD_AGENT_PATHS_ALL );
        break;
    default:
        break;
}

genedlgs.h

#ifndef GENEDLGS_H
#define GENEDLGS_H

void DisplayOptionsDialogHandler( HWND hwnd );
void ErrorOptionsDialogHandler( HWND hwnd );
void EstimatorOptionsDialogHandler( HWND hwnd );
void OperatorOptionsDialogHandler( HWND hwnd );
void AgentOptionsDialogHandler( HWND hChild );

BOOL CALLBACK __export
DisplayDialogBoxProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK __export
ErrorDialogBoxProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK __export
EstimatorDialogBoxProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK __export
OperatorDialogBoxProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK __export
AgentDialogBoxProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);
#endif

#define STRICT
#define NOCOMM
#include <Windows.h>
#include <windowsx.h>
#include "genes.h"
#include "genemenu.h"
#include "resource.h"
#include "instance.h"
#include "genedlgs.h"
#include "gblopts.h"
#include "pathdb.h"
#include "timeslc.h"
#include "msmtdb.h"
#include "populmgr.h"
#include "expmt.h"

#include "glblopts.h"
#include "pathdb.h"
#include "timeslc.h"
#include "msmtdb.h"
#include "populmgr.h"
#include "expmt.h"

#define CHECK_IT
#define UNCHECK_IT
MF_BYCOMMAND
MF_CHECKED)
MF_UNCHECKED

void GeneMenu_ShowAboutBox(HWND hwnd);
void GeneMenu_UpdatePause(HMENU hMenu);

void GeneMenuHandler(HWND hwnd, WPARAM wParam)
{
    switch(wParam)
    {
        case IDM/about:
            GeneMenu_ShowAboutBox(hwnd);
            break;

        case IDM/LOBOTOMY:
            SetCursor(LoadCursor(NULL, IDC_WAIT));
            LobotomizePopulation();
            LobotomizeMeasurementDatabase();
            ResetTimeSlice(hwnd);
            ResetAllAgentDisplays();
            SetCursor(LoadCursor(NULL, IDC_ARROW));
            break;

        case IDM/PAUSE:
            ToggleGlobalPausedState();
            break;

        case IDM/EXIT:
            DestroyWindow(hwnd);
            break;

        case IDM OPTIONS DISPLAY:
            break;
    }
}
```c
void DisplayOptionsDialogHandler( hwnd );
break;

case IDM_OPTIONS_ERROR:
    ErrorOptionsDialogHandler( hwnd );
    break;

case IDM_OPTIONS_ESTIMATOR:
    EstimatorOptionsDialogHandler( hwnd );
    break;

case IDM_OPTIONS_OPERATORS:
    OperatorOptionsDialogHandler( hwnd );
    break;

case IDM_EXPMT_OWS1:
case IDM_EXPMT_OWS2:
case IDM_EXPMT_OWS3:
case IDM_EXPMT_ODS2:
case IDM_EXPMT_ODS3:
case IDM_EXPMT_SWPOOR:
case IDM_EXPMT_SWFAIR:
case IDM_EXPMT_SWGOOD:
case IDM_EXPMT_SDPOOR:
case IDM_EXPMT_SDFAIR:
case IDM_EXPMT_SDGOOD:
case IDM_EXPMT_CW5:
case IDM_EXPMT_CW20:
case IDM_EXPMT_CW40:
case IDM_EXPMT_CD5:
case IDM_EXPMT_CD20:
case IDM_EXPMT_CD40:
case IDM_EXPMT_EST1:
case IDM_EXPMT_EST2:
case IDM_EXPMT_EST3:
case IDM_EXPMT_EST4:
case IDM_EXPMT_MOTION:
    ExperimentOptionsHandler( hwnd, wParam );
    break;

default:
    break;
}

UpdateGeneMenu( hwnd );

void UpdateGeneMenu( HWND hwnd )
{
    HMENU hMenu;
    hMenu = GetMenu( hwnd );

    GeneMenu_UpdatePause( hMenu );
}

void GeneMenu_UpdatePause( HMENU hMenu )
{
    BOOL bState;

    CheckMenuItem( hMenu, IDM_PAUSE, UNCHECK IT );
    bState = GetGlobalPausedState();
    if( bState == TRUE )
        CheckMenuItem( hMenu, IDM_PAUSE, CHECK IT );
}
```
void GeneMenu_ShowAboutBox( HWND hwnd )
{
    DLGPROC lpProcAbout;
    HINSTANCE hInstance;

    hInstance = GetWindowInstance (hwnd);
    lpProcAbout = (DLGPROC) MakeProcInstance ((FARPROC) About, hInstance);
    DialogBox (hInstance, MAKEINTRESOURCE (IDD_ABOUTBOX), hwnd, lpProcAbout);
    FreeProcInstance ((FARPROC) lpProcAbout);
}

genemenu.h

#ifndef GENEMENU_H
#define GENEMENU_H

void GeneMenuHandler( HWND hwnd, WPARAM wParam );
void UpdateGeneMenu( HWND hwnd );

#endif

geneplot.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include "geneplot.h"
#include "fcoord.h"
#include "plotbmp.h"
#include "numunits.h"
#include "populmgr.h"
#include "vector.h"
#include "msmtdb.h"
#include "metrics.h"
#include "glibopt.h"
#include "pathdb.h"
#include "genetype.h"

static FCOORD fxyObs;

void GenePlot_ConvertTrueToScreen( POINT *xyScreen, FCOORD *fxyTrue );
void GenePlot_PlotPath( HDC hDC, int nToIndex, int nPathNum, DWORD dwColor );
void GenePlot_DrawCross( HDC hDC, POINT *xyPlot );

void PlotMeasurements( HWND hParent, int nFromIndex )
{
    int i;
    HBIMAP hBitmap;
    HDC hDC, hMemDC;
    HPEN hPen;
    BOOL bPlotOn, bValid;
    FCOORD fxyTrue, fxyMeasurement, fxyPlot;
    POINT xyPlot;

    bPlotOn = GetGlobalMeasurementDisplayState();
    if( bPlotOn == FALSE ) return;

    GetIndividualMeasurementDisplayStateFromIndex( nFromIndex, &bPlotOn );
    if( bPlotOn == FALSE ) return;

156
bValid = GetTrueCoordinates( nFromIndex, &fxyTrue );  
if( bValid == FALSE ) return;

hBitmap = GetBackgroundBitmapHandle();  
hDC = GetDC( hParent );  
hMemDC = CreateCompatibleDC( hDC );  
SelectObject( hMemDC, hBitmap );  
hPen = CreatePen( PS_SOLID, 2, RGB( 255, 0, 0 ) );  
SelectObject( hMemDC, hPen );  

for( i = 0; i < NUMBER_OF_UNITS; i++ )  
{  
if( i == nFromIndex ) continue;

bValid = GetMeasurement( nFromIndex, i, &fxyMeasurement );  
if( bValid == FALSE ) continue;

SumVectors( &fxyPlot, &fxyTrue, &fxyMeasurement );  
GenePlot_ConvertTrueToScreen( &xyPlot, &fxyPlot );

MoveTo( hMemDC, xyPlot.x-1, xyPlot.y );  
LineTo( hMemDC, xyPlot.x, xyPlot.y );

SelectObject( hMemDC, GetStockObject( BLACK_PEN ) );  
DeleteObject( hPen );  
DeleteDC( hMemDC );  
ReleaseDC( hParent, hDC );

}

void GenePlot_ConvertTrueToScreen( POINT *xyScreen, FCOORD *fxyTrue )
{
float fTrueX, fTrueY;

fTrueX = fxyTrue->x*((float)PIXELS_PER_METER);  
fTrueY = ((float)0.0) - fxyTrue->y*((float)PIXELS_PER_METER);

xyScreen->x = ((int)fTrueX);  
xyScreen->y = ((int)fTrueY);
}

void PlotEstimates( HWND hParent, int nToIndex )
{
HBITMAP hBitmap;  
HPEN hPen;  
HDC hDC, hMemDC;  
BOOL bPlotOn, bValid;  
FCOORD fxyEstimate, fxyFrom, fxyTrue;  
POINT xyPlot;

bPlotOn = GetGlobalEstimateDisplayState();  
if( bPlotOn == FALSE ) return;

bValid = GetIndividualEstimateDisplayStateFromIndex( nToIndex, &bPlotOn );  
if( bValid == FALSE ) return;  
if( bPlotOn == FALSE ) return;

bValid = GetEstimateFromIndex( nToIndex, &fxyEstimate );  
if( bValid == FALSE ) return;

bValid = GetTrueCoordinates( 0, &fxyFrom );  
if( bValid == FALSE ) return;
SumVectors( &fxyTrue, &fxyFrom, &fxyEstimate );  
GenePlot_ConvertTrueToScreen( &xyPlot, &fxyTrue );

hBitmap = GetBackgroundBitmapHandle();
hDC = GetDC( hParent );
hMemDC = CreateCompatibleDC( hDC );
SelectObject( hMemDC, hBitmap );
hPen = CreatePen( PS_SOLID, 2, RGB( 0, 0, 255 ) );
SelectObject( hMemDC, hPen );

/**< Plot Estimate Here */
GenePlot_DrawCross( hMemDC, &xyPlot );
SelectObject( hMemDC, GetStockObject( BLACK_PEN ) );
DeleteObject( hPen );
DeleteDC( hMemDC );
ReleaseDC( hParent, hDC );

void PlotPaths( HWND hParent, int nToindex )
{
   int i, nPlotState;
   HBITMAP hBitmap;
   HDC hDC, hMemDC;
   BOOL bPlotOn, bValid;

   bPlotOn = GetGlobalPathDisplayState();
   if( bPlotOn == FALSE ) return;

   bValid = GetIndividualPathDisplayFromIndex( nToIndex, &nPlotState );
   if( bValid == FALSE ) return;

   hBitmap = GetBackgroundBitmapHandle();
   hDC = GetDC( hParent );
   hMemDC = CreateCompatibleDC( hDC );
   SelectObject( hMemDC, hBitmap );

   switch( nPlotState )
   {
   case SHOW_ALL_PATHS:
      for( i = 15; i < POPULATION_SIZE; i++ )
      {
         GenePlot_PlotPath( hMemDC, nToIndex, i, RGB(255,0,0) );
      }
   case SHOW_15_PATHS:
      for( i = 5; i < 15; i++ )
      {
         GenePlot_PlotPath( hMemDC, nToIndex, i, RGB(255,255,0) );
      }
   case SHOW_5_PATHS:
      for( i = 1; i < 5; i++ )
      {
         GenePlot_PlotPath( hMemDC, nToIndex, i, RGB(0,255,0) );
      }
   case SHOW_1_PATH:
      GenePlot_PlotPath( hMemDC, nToIndex, 0, RGB(0,0,255) );
      break;
   case SHOW_NO_PATHS:
      default:
      break;
   }
   DeleteDC( hMemDC );
   ReleaseDC( hParent, hDC );
}

void GenePlot_PlotPath( HDC hDC, int nToindex, int nPathNum, DWORD dwColor )
{
   HPEN hPen;

   158
```c
int i, nFrom, nTo, nTargetIndex;
FCOORD fxyFrom, fxyTo, fxyMeasurement;
BOOL bIsValid, bPathValid;
POINT xyPlot;
LPPOPUL ppopPopul;
HPNODE hpnodeList;

if( nPathNum < 0 ) return;
if( nPathNum >= POPULATION_SIZE ) return;

bIsValid = GetPopulationPointerFromIndex( nToIndex, &ppopPopul );
if( bIsValid == FALSE ) return;

bPathValid = ppopPopul->ahppathPaths[nPathNum]->bIsValid;
if( bPathValid == FALSE ) return;

nTargetIndex = ppopPopul->ahppathPaths[nPathNum]->nTargetIndex;
hpnodeList = ppopPopul->ahppathPaths[nPathNum]->hpnodeNodeList;

/* Get Start Point */
bIsValid = GetTrueCoordinates( 0, &fxyFrom );
if( bIsValid == FALSE ) return;

hPen = CreatePen( PS_SOLID, 1, dwColor);
SelectObject( hDC, hPen );

GenePlot_ConvertTrueToScreen( &xyPlot, &fxyFrom );
MoveTo( hDC, xyPlot.x, xyPlot.y );
nFrom = 0;
for( i = 0; i <= nTargetIndex; i++ )
{
    nTo = (int)hpnodeList[i];
    /* Get Measurement */
bIsValid = GetMeasurement( nFrom, nTo, &fxyMeasurement );
    if( bIsValid == FALSE ) break;

    SumVectors( &fxyTo, &fxyFrom, &fxyMeasurement );
    GenePlot_ConvertTrueToScreen( &xyPlot, &fxyTo );
    LineTo( hDC, xyPlot.x, xyPlot.y );
    fxyFrom.x = fxyTo.x;
    fxyFrom.y = fxyTo.y;
    nFrom = nTo;
}

SelectObject( hDC, GetStockObject( BLACK_PEN ) );
DeleteObject( hPen );
}

void GenePlot_DrawCross( HDC hDC, POINT *xyPlot )
{
    MoveTo( hDC, xyPlot->x, xyPlot->y-5 );
    LineTo( hDC, xyPlot->x, xyPlot->y+5 );
    MoveTo( hDC, xyPlot->x-5, xyPlot->y );
    LineTo( hDC, xyPlot->x+S, xyPlot->y );
}

void PlotObservations( HWND hParent, int nToIndex )
{
    HBMP hBitmap;
    HPG hPen;
    HDC hDC, hMemDC;
    BOOL bPlotOn, bValid;
    FCOORD fxyFrom, fxyTrue;
    POINT xyPlot;
}
bPlotOn = GetGlobalObservationDisplayState();
if( bPlotOn == FALSE ) return;

bValid = GetIndividualObservationDisplayStateFromIndex( nToIndex, &bPlotOn );
if( bValid == FALSE ) return;
if( bPlotOn == FALSE ) return;

bValid = GetTrueCoordinates( 0, &fxyFrom );
if( bValid == FALSE ) return;

bValid = GetObservationFromIndex( nToIndex, &fxyObs );
if( bValid == FALSE ) return;

hBitmap = GetBackgroundBitmapHandle();
hDC = GetDC( hParent );
hMemDC = CreateCompatibleDC( hDC );
SelectObject( hMemDC, hBitmap );
hPen = CreatePen( PS_SOLID, 1, RGB( 0, 255, 0 ) );
SelectObject( hMemDC, hPen );

SumVectors( &fxyTrue, &fxyFrom, &fxyObs );
GenePlot_ConvertTrueToScreen( &xyPlot, &fxyTrue );
GenePlot_DrawCross( hMemDC, &xyPlot );
SelectObject( hMemDC, GetStockObject( BLACK_PEN ) );
DeleteObject( hPen );
DeleteDC( hMemDC );
ReleaseDC( hParent, hDC );
}

geneplot.h

#ifndef GENEPLOT_H
#define GENEPLOT_H

void PlotMeasurements( HWND hParent, int nFromIndex );
void PlotEstimates( HWND hParent, int nToIndex );
void PlotPaths( HWND hParent, int nToIndex );
void PlotObservations( HWND hwnd, int nIndex );

#endif

genes.def

; Module-definition file for the Genetic Algorithm simulator

NAME GENES WINDOWAPI
DESCRIPTION 'Genetic Algorithm Location Estimation, Copyright Walter T. Baugh, 1993'
STUB 'WINSTUB.EXE'
EXETYPE WINDOWS 3.1
PROTMODE
STACKSIZE 5120
HEAPSIZE 15024
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
EXPORTS
MainWndProc @1 ; name of main window function
About @2 ; name of About dialog function
ChildWndProc @3 ; name of Child window function
DisplayDialogBoxProc @4 ; name of Display dialog function
genes.h

#define SZCLASSNAME 60
#define SZWINDOWTITLE 60
#define WS_GENES (WS_OVERLAPPEDWINDOW)
#define WMU_TIME_SLICE WM_USER+210

int PASCAL WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
LRESULT CALLBACK _export MainWndProc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK _export About(HWND, UINT, WPARAM, LPARAM);

#include "resource.h"
#include "dialog.h"

genes.mak

ORIGIN = PWB
ORIGIN_VER = 2.0
PROJ = GENES
PROJFILE = GENES.MAK
DEBUG = 1

CC = cl
CFLAGS_G = /AL /W4 /G2 /GA /Gef /Zp /BATCH
CFLAGS_D = /f /Ot /zi /Gs
CFLAGS_R = /f- /Os /G/ /Ga /Gs
CXX = cl
CXXFLAGS_G = /G2 /W2 /GA /Gef /Zp /BATCH
CXXFLAGS_D = /f /zi /Oo /Gs
CXXFLAGS_R = /f- /Oe /Os /Gs
MAPFILE_D = $(PROJ).map
MAPFILE_R = NUL
LFLAGS_G = /BATCH /ONERROR:NOEXE
LFLAGS_D = /CO /MAP:FULL /NOF /NOPACKC
LFLAGS_R = /NOF
LLIBS_G = LIBW.LIB
LINKER = link
ILINK = ilink
LRF = echo > NUL
ILFLAGS = /a /e
RC = ic
LLIBS_R = /NOD:LLIBCE LLIBCEW
LLIBS_D = /NOD:LLIBCE LLIBCEW

FILES = ABOUT.OBJ BMPMGR.C CHILD.C GENEMENU.C GENEPLOT.C GENES.DEF GLBLOPTS.C\ MSMTDB.C MSMTMKR.C PAINTMGR.C PATHDB.C POPULMGR.C RANDOM.C TIMESLCE.C\ TIMESTAMP.C VECTOR.C WINMAIN.C WNDPROC.C PLOTBMP.C GENES.RC PATHMGR.C\ LISTMGR.C ESTIMATE.C GENEDLGS.C EXPMT.C
DEF_FILE = GENES.DEF
OBJ_EXT = ABOUT.OBJ

OBJS = BMPMGR.obj CHILD.obj GENEMENU.obj GENEPLOT.obj GLBLOPTS.obj MSMTDB.obj\ MSMTMKR.obj PAINTMGR.obj PATHDB.obj POPULMGR.obj RANDOM.obj\ TIMESLCE.obj TIMESTAMP.obj VECTOR.obj WINMAIN.obj WNDPROC.obj\ PLOTBMP.obj PATHMGR.obj LISTMGR.obj ESTIMATE.obj GENEDLGS.obj\ EXPMT.obj $(OBJS_EXT)
RESS = GENES.res

all: $(PROJ).exe

.SUFFIXES: .obj .res .c .rc

BMPMGR.obj : BMPMGR.C bmpmgr.h
  IF $(DEBUG)
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_G) /FoBMPMGR.obj BMPMGR.C
  ELSE
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_R) /FoBMPMGR.obj BMPMGR.C
  ENDIF

CHILD.obj : CHILD.C child.h bmpmgr.h populmgr.h msmtdb.h genedlgs.h fcoord.h timesamp.h
  IF $(DEBUG)
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_G) /FoCHILD.obj CHILD.C
  ELSE
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_R) /FoCHILD.obj CHILD.C
  ENDIF

GENEMENU.obj : GENEMENU.C genes.h genemenu.h resource.h instance.h genedlgs.h\
    glblopts.h pathdb.h timeslc.h msmtdb.h populmgr.h expmt.h dialog.h\
    genetype.h timesamp.h fcoord.h
  IF $(DEBUG)
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_G) /FoGENEMENU.obj GENEMENU.C
  ELSE
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_R) /FoGENEMENU.obj GENEMENU.C
  ENDIF

GENEPLOT.obj : GENEPLOT.C geneplot.h fcoord.h plotbmp.h numunits.h populmgr.h\
    vector.h msmtdb.h metrics.h glblopts.h pathdb.h genetype.h timesamp.h
  IF $(DEBUG)
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_G) /FoGENEPLOT.obj GENEPLOT.C
  ELSE
    @$(CC) @<<$(PROJ).rsp
    $(CFLAGS_R) /FoGENEPLOT.obj GENEPLOT.C
  ENDIF
GLBLOPTS.obj : GLBLOPTS.C glblopts.h
!IF $(DEBUG)
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_D) /FoGLBLOPTS.obj GLBLOPTS.C
<<
!ELSE
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_R) /FoGLBLOPTS.obj GLBLOPTS.C
<<
!ENDIF

MSMTDB.obj : MSMTDB.C numunits.h msmtdb.h metrics.h timestamp.h fcoord.h
!IF $(DEBUG)
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_D) /FoMSMTDB.obj MSMTDB.C
<<
!ELSE
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_R) /FoMSMTDB.obj MSMTDB.C
<<
!ENDIF

MSMTMKR.obj : MSMTMKR.C fcoord.h msmtmkr.h msmtdb.h populmgr.h numunits.h\metrics.h random.h glblopts.h timestamp.h
!IF $(DEBUG)
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_D) /FoMSMTMKR.obj MSMTMKR.C
<<
!ELSE
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_R) /FoMSMTMKR.obj MSMTMKR.C
<<
!ENDIF

PAINTMGR.obj : PAINTMGR.C paintmgr.h plotbmp.h
!IF $(DEBUG)
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_D) /FoPAINTMGR.obj PAINTMGR.C
<<
!ELSE
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_R) /FoPAINTMGR.obj PAINTMGR.C
<<
!ENDIF

PATHDB.obj : PATHDB.C pathdb.h genetype.h numunits.h listmgr.h msmtdb.h\fcoord.h timestamp.h
!IF $(DEBUG)
   @$(CC) @<<$(PROJ).rsp
/c $(CFLAGS_G)
$(CFLAGS_D) /FoPATHDB.obj PATHDB.C
<<
!ELSE
   @$$(CC) @<<$(PROJ).rsp

$(CFLAGS_D) /FoVECTOR.obj VECTOR.C
<<
!ELSE
  @$$(CC) @<<$(PROJ).rsp
  /c $$(CFLAGS_G)
$$ (CFLAGS_R) /FoVECTOR.obj VECTOR.C
<<
!ENDIF

WINMAIN.obj : WINMAIN.C genes.h instance.h resource.h dialog.h
  !IF $(DEBUG)
    @$$(CC) @<<$(PROJ).rsp
    /c $$(CFLAGS_G)
$$ (CFLAGS_D) /FoWINMAIN.obj WINMAIN.C
    <<
    !ELSE
    @$$(CC) @<<$(PROJ).rsp
    /c $$(CFLAGS_G)
$$ (CFLAGS_R) /FoWINMAIN.obj WINMAIN.C
    <<
  !ENDIF

WNDPROC.obj : WNDPROC.C genes.h timeslc.h bmpmgr.h instance.h paintmgr.h\
   child.h populgr.h genemenu.h random.h msmtdb.h msmtmkr.h numunits.h\
   plotbmp.h pathdb.h glibropts.h resource.h dialog.h fcoord.h timestamp.h\
   genotype.h
  !IF $(DEBUG)
    @$$(CC) @<<$(PROJ).rsp
    /c $$(CFLAGS_G)
$$ (CFLAGS_D) /FoWNDPROC.obj WNDPROC.C
    <<
    !ELSE
    @$$(CC) @<<$(PROJ).rsp
    /c $$(CFLAGS_G)
$$ (CFLAGS_R) /FoWNDPROC.obj WNDPROC.C
    <<
  !ENDIF

PLOTBMP.obj : PLOTBMP.C plotbmp.h
  !IF $(DEBUG)
    @$$(CC) @<<$(PROJ).rsp
    /c $$(CFLAGS_G)
$$ (CFLAGS_D) /FoPLOTBMP.obj PLOTBMP.C
    <<
    !ELSE
    @$$(CC) @<<$(PROJ).rsp
    /c $$(CFLAGS_G)
$$ (CFLAGS_R) /FoPLOTBMP.obj PLOTBMP.C
    <<
  !ENDIF

GENES.res : GENES.RC resource.h dialog.h gendlg1.h gendlg2.h gendlg3.h\
  gendlg4.h gendlg5.h c:\source\win\icons\genetic.ico\n  c:\source\win\bitmaps\box14\btn14.bmp\n  c:\source\win\bitmaps\box14\bl4aqepa.bmp\n  c:\source\win\bitmaps\box14\bl4aqepm.bmp\n  c:\source\win\bitmaps\box14\bl4aqepf.bmp\n  c:\source\win\bitmaps\box14\bl4aqfpa.bmp\n  c:\source\win\bitmaps\box14\bl4aqfpm.bmp\n  c:\source\win\bitmaps\box14\bl4aqfpf.bmp\n  c:\source\win\bitmaps\box14\bl4aqppa.bmp\n  c:\source\win\bitmaps\box14\bl4aqppm.bmp\n  c:\source\win\bitmaps\box14\bl4aqppf.bmp\n
PATHMGR.obj : PATHMGR.C pathmgr.h genotype.h pathdb.h numunits.h random.h listmgr.h msmtdb.h D:\C700\INCLUDE\math.h populmgr.h glblopts.h fcookie.h timestamp.h
!IFDEF $(DEBUG)
  @$@$(CC) @<<$(PROJ).rsp /c $(CFLAGS) /FoPATHMGR.obj PATHMGR.C
<<
!ELSE
  @$@$(CC) @<<$(PROJ).rsp /c $(CFLAGS) /FoPATHMGR.obj PATHMGR.C
<<
!ENDIF

LISTMGR.obj : LISTMGR.C genotype.h listmgr.h random.h numunits.h fcookie.h
!IFDEF $(DEBUG)
  @$@$(CC) @<<$(PROJ).rsp /c $(CFLAGS) /FoLISTMGR.obj LISTMGR.C
<<
!ELSE
  @$@$(CC) @<<$(PROJ).rsp /c $(CFLAGS) /FoLISTMGR.obj LISTMGR.C
<<
!ENDIF

ESTIMATE.obj : ESTIMATE.C estimate.h pathdb.h genotype.h populmgr.h glblopts.h fcookie.h
!IFDEF $(DEBUG)
  @$@$(CC) @<<$(PROJ).rsp /c $(CFLAGS) /FoESTIMATE.obj ESTIMATE.C
<<
!ELSE
  @$@$(CC) @<<$(PROJ).rsp /c $(CFLAGS) /FoESTIMATE.obj ESTIMATE.C
<<
!ENDIF

GENEDLGS.obj : GENEDLGS.C genedlgs.h genedlg1.h genedlg2.h genedlg3.h genedlg4.h genedlg5.h glblopts.h instance.h populmgr.h fcookie.h
!IFDEF $(DEBUG)
  @$@$(CC) @<<$(PROJ).rsp /c $(CFLAGS) /FoGENEDLGS.obj GENEDLGS.C
<<
!ENDIF

166
!ELSE
  @$(CC) @<<$(PROJ).rsp
  /c $(CFLAGS_G)
  $(CFLAGS_R) /FoGENEDLGs.obj GENEDLGs.C
<<
!ENDIF

EXPMT.obj : EXPMT.C expmt.h resource.h numunits.h populmgr.h glblopts.h
           pathdb.h fcoord.h genotype.h
!IF $(DEBUG)
  @$(CC) @<<$(PROJ).rsp
  /c $(CFLAGS_G)
  $(CFLAGS_D) /FoEXPMT.obj EXPMT.C
<<
!ELSE
  @$(CC) @<<$(PROJ).rsp
  /c $(CFLAGS_G)
  $(CFLAGS_R) /FoEXPMT.obj EXPMT.C
<<
!ENDIF

$(PROJ).exe : $(DEF_FILE) $(OBS) $(RESS)
!IF $(DEBUG)
  $(LRF) @<<$(PROJ).lrf
  $(RT_OBJS: =) $(OBS: =) $@
  $(MAPFILE_D)
  $(LIBS: =) +
  $(LLIBS_G: =) +
  $(LLIBS_D: =) +
  $(DEF_FILE) $(LFLAGS_G) $(LFLAGS_D);
<<
!ELSE
  $(LRF) @<<$(PROJ).lrf
  $(RT_OBJS: =) $(OBS: =) $@
  $(MAPFILE_R)
  $(LIBS: =) +
  $(LLIBS_G: =) +
  $(LLIBS_R: =) +
  $(DEF_FILE) $(LFLAGS_G) $(LFLAGS_R);
<<
!ENDIF
  $(LINKER) @$(PROJ).lrf
  $(RC) $(RCFLAGS2) $(RESS) $@

.c.obj :
!IF $(DEBUG)
  @$(CC) @<<$(PROJ).rsp
  /c $(CFLAGS_G)
  $(CFLAGS_R) /Fo$@ $<
genes.rc

#include <windows.h>
#include "resource.h"
#include "dialog.h"
#include "genedlg1.h"
#include "genedlg2.h"
#include "genedlg3.h"
#include "genedlg4.h"
#include "genedlgS.h"

GENE ICON
SOURCE BMP

ICON c:\source\win\icons\genetic.ico

SOURCE BMP
BITMAP c:\source\win\bitmaps\box14\btn14.bmp

A_QE_PA_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqepa.bmp
A_QE_PM_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqepm.bmp
A_QE_PF_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqepf.bmp
A_QP_PA_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqfpa.bmp
A_QP_PM_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqfpm.bmp
A_QP_PF_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqfpf.bmp
A_QN_PA_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqppa.bmp
A_QN_PM_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqppm.bmp
A_QN_PF_BMP
BITMAP c:\source\win\bitmaps\box14\b14aqppf.bmp

I_QE_PA_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqls.bmp
I_QE_PM_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqfpa.bmp
I_QE_PF_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqfpf.bmp
I_QP_PA_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqls.bmp
I_QP_PM_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqfpm.bmp
I_QP_PF_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqfpf.bmp
I_QN_PA_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqls.bmp
I_QN_PM_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqfpm.bmp
I_QN_PF_BMP
BITMAP c:\source\win\bitmaps\box14\b14iqfpf.bmp

IDM_GENESMENU MENU
BEGIN
POPUP "&Command"
BEGIN
MENUTITEM "&Pause" IDM_PAUSE
MENUTITEM SEPARATOR
MENUTITEM "&Lobotomize" IDM_LOBOTOMY

168
 includes "genes.dlg"
#include "genedlg1.dlg"
#include "genedlg2.dlg"
#include "genedlg3.dlg"
#include "genedlg4.dlg"
#include "genedlg5.dlg"
#ifndef GENETYPE_H
#define GENETYPE_H

#include "fcoord.h"

#define POPULATION_SIZE

typedef int NODE;
typedef NODE *PNODE;
typedef NODE NEAR *NNODE;
typedef NODE FAR *LNODE;
typedef NODE _huge *HNODE;

typedef struct tagPATH
    { HNODE hnodeNodeList;
        NODE nodeTargetNode;
        int nTargetIndex;
        float fFitness;
        float fWeight;
        FCOORD fxyPathVector;
        BOOL bIsValid;
    } PATH;

typedef PATH *PPATH;
typedef PATH NEAR *NNPATH;
typedef PATH FAR *LPATH;
typedef PATH _huge *HPATH;

#define NOCOMM

typedef struct tagPOPUL
    { HPATH ahpathPaths[POPULATION_SIZE];
        NODE nodeTargetNode;
        FCOORD fxyObservation;
        FCOORD fxyEstimate;
        int nNumValidPaths;
        float fTotalFitness;
        BOOL bValidEstimate;
        BOOL bValidObservation;
        POPUL POPUL;
    } POPUL;

typedef POPUL *PPPOPUL;
typedef POPUL NEAR *NPOPUL;
typedef POPUL FAR *LPPOPUL;
typedef POPUL _huge *HPPOPUL;
#endif

#endif

#define STRICT

#include <windows.h>
#include <windowsx.h>
#include "gblopts.h"

static int nNoiseModel = NOISE_MODEL_GAUSSIAN;
static int nCommLoss = COMM_LOSS_5_PERCENT;
static int nErrorScale = ERROR_SCALE_FAIR;
static BOOL bShowEstimates = TRUE;

#include "gblopts.h"

/* array of pointers */

gblopts.c
static BOOL bShowObservations = TRUE;
static BOOL bShowMeasurements = FALSE;
static BOOL bShowPaths = TRUE;
static BOOL bIsPaused = TRUE;

static BOOL bReproductionDirect = TRUE;
static BOOL bReproductionRandom = FALSE;
static BOOL bReproductionInside = TRUE;
static BOOL bReproductionOutside = TRUE;
static BOOL bReproductionCrossover = FALSE;
static BOOL bReproductionLinear = FALSE;
static BOOL bReproductionCircular = FALSE;
static BOOL bReproductionReduction = FALSE;

static int nEstimator = ESTIMATOR_MEDIAN;

void ToggleGlobalEstimateDisplay( void )
{
    if( bShowEstimates == TRUE )
    {
        bShowEstimates = FALSE;
    }
    else
    {
        bShowEstimates = TRUE;
    }
}

BOOL GetGlobalEstimateDisplayState( void )
{
    return bShowEstimates;
}

void ToggleGlobalObservationDisplay( void )
{
    if( bShowObservations == TRUE )
    {
        bShowObservations = FALSE;
    }
    else
    {
        bShowObservations = TRUE;
    }
}

BOOL GetGlobalObservationDisplayState( void )
{
    return bShowObservations;
}

void ToggleGlobalPausedState( void )
{
    if( bIsPaused == TRUE )
    {
        bIsPaused = FALSE;
    }
    else
    {
        bIsPaused = TRUE;
    }
}

BOOL GetGlobalPausedState( void )
{
    return bIsPaused;
}

void ToggleGlobalMeasurementDisplay( void )
{
    if( bShowMeasurements == TRUE )
    {
        bShowMeasurements = FALSE;
    }
    else
    {
        bShowMeasurements = TRUE;
    }
}
BOOL GetGlobalMeasurementDisplayState( void )
{
    return bShowMeasurements;
}

void ToggleGlobalPathDisplay( void )
{
    if( bShowPaths == TRUE ) bShowPaths = FALSE;
    else bShowPaths = TRUE;
}

BOOL GetGlobalPathDisplayState( void )
{
    return bShowPaths;
}

void SetNoiseModelToGaussian( void )
{
    nNoiseModel = NOISE_MODEL_GAUSSIAN;
}

void SetNoiseModelToCauchy( void )
{
    nNoiseModel = NOISE_MODEL_CAUCHY;
}

int GetNoiseModel( void )
{
    return nNoiseModel;
}

void SetCommLossToNone( void )
{
    nCommLoss = COMM_LOSS_NONE;
}

void SetCommLossTo5Percent( void )
{
    nCommLoss = COMM_LOSS_5_PERCENT;
}

void SetCommLossTo20Percent( void )
{
    nCommLoss = COMM_LOSS_20_PERCENT;
}

void SetCommLossTo40Percent( void )
{
    nCommLoss = COMM_LOSS_40_PERCENT;
}

int GetCommLoss( void )
{
    return nCommLoss;
}

void SetErrorScaleToGood( void )
{
    nErrorScale = ERROR_SCALE_GOOD;
}

void SetErrorScaleToFair( void )
{
    nErrorScale = ERROR_SCALE_FAIR;
}

void SetErrorScaleToPoor( void )
{
    nErrorScale = ERROR_SCALE_POOR;
}

int GetErrorScale( void )
{
    return nErrorScale;
}

void ToggleReproductionDirect( void )
{
    if( bReproductionDirect == TRUE )
BOOL bReproductionDirect = FALSE;
    }
else
    {
    bReproductionDirect = TRUE;
    }

BOOL GetReproductionDirectState( void )
{
return bReproductionDirect;
}

void ToggleReproductionRandom( void )
{
if( bReproductionRandom == TRUE )
    {
    bReproductionRandom = FALSE;
    }
else
    {
    bReproductionRandom = TRUE;
    }

BOOL GetReproductionRandomState( void )
{
return bReproductionRandom;
}

void ToggleReproductionInside( void )
{
if( bReproductionInside == TRUE )
    {
    bReproductionInside = FALSE;
    }
else
    {
    bReproductionInside = TRUE;
    }

BOOL GetReproductionInsideState( void )
{
return bReproductionInside;
}

void ToggleReproductionOutside( void )
{
if( bReproductionOutside == TRUE )
    {
    bReproductionOutside = FALSE;
    }
else
    {
    bReproductionOutside = TRUE;
    }

BOOL GetReproductionOutsideState( void )
{
return bReproductionOutside;
}

void ToggleReproductionCrossover( void )
{
if( bReproductionCrossover == TRUE )
    {
    bReproductionCrossover = FALSE;
    }
else
    {
    bReproductionCrossover = TRUE;
    }

BOOL GetReproductionCrossoverState( void )
{
return bReproductionCrossover;
}
void ToggleReproductionLinear( void )
{ if( bReproductionLinear == TRUE )
    { bReproductionLinear = FALSE; }
else
    { bReproductionLinear = TRUE; }
}

BOOL GetReproductionLinearState( void )
{ return bReproductionLinear; }

void ToggleReproductionCircular( void )
{ if( bReproductionCircular == TRUE )
    { bReproductionCircular = FALSE; }
else
    { bReproductionCircular = TRUE; }
}

BOOL GetReproductionCircularState( void )
{ return bReproductionCircular; }

void ToggleReproductionReduction( void )
{ if( bReproductionReduction == TRUE )
    { bReproductionReduction = FALSE; }
else
    { bReproductionReduction = TRUE; }
}

BOOL GetReproductionReductionState( void )
{ return bReproductionReduction; }

void SetEstimatorToMean( void )
{ nEstimator = ESTIMATOR_MEAN; }

void SetEstimatorToMedian( void )
{ nEstimator = ESTIMATOR_MEDIAN; }

void SetEstimatorToModeMean( void )
{ nEstimator = ESTIMATOR_MODE_MEAN; }

void SetEstimatorToModeMedian( void )
{ nEstimator = ESTIMATOR_MODE_MEDIAN; }

int GetEstimator( void )
{ return nEstimator; }

glblopts.h

#ifndef GLBLOPTS_H

#define GLBLOPTS_H

#endif

174
#define GLBLOPTS_H

#define NOISE_MODEL_GAUSSIAN 1100
#define NOISE_MODEL_CAUCHY 1101

#define COMM_LOSS_NONE 1110
#define COMM_LOSS_5_PERCENT 1111
#define COMM_LOSS_20_PERCENT 1112
#define COMM_LOSS_40_PERCENT 1113

#define ERROR_SCALE_GOOD 1120
#define ERROR_SCALE_FAIR 1121
#define ERROR_SCALE_POOR 1122

#define ESTIMATOR_MEAN 1130
#define ESTIMATOR_MEDIAN 1131
#define ESTIMATOR_MODE_MEAN 1132
#define ESTIMATOR_MODE_MEDIAN 1133

void ToggleGlobalEstimateDisplay( void);
BOOL GetGlobalEstimateDisplayState( void);
void ToggleGlobalObservationDisplay( void);
BOOL GetGlobalObservationDisplayState( void);
void ToggleGlobalMeasurementDisplay( void);
BOOL GetGlobalMeasurementDisplayState( void);
void ToggleGlobalPathDisplay( void);
BOOL GetGlobalPathDisplayState( void);
void SetNoiseModelToGaussian( void);
void SetNoiseModelToCauchy( void);
int GetNoiseModel( void);
void SetCommLossToNone( void);
void SetCommLossTo5Percent( void);
void SetCommLossTo20Percent( void);
void SetCommLossTo40Percent( void);
int GetCommLoss( void);
void SetErrorScaleToGood( void);
void SetErrorScaleToFair( void);
void SetErrorScaleToPoor( void);
int GetErrorScale( void);
void ToggleGlobalPausedState( void);
BOOL GetGlobalPausedState( void);

void ToggleReproductionDirect( void);
BOOL GetReproductionDirectState( void);
void ToggleReproductionRandom( void);
BOOL GetReproductionRandomState( void);
void ToggleReproductionInside( void);
BOOL GetReproductionInsideState( void);
void ToggleReproductionOutside( void);
BOOL GetReproductionOutsideState( void);
void ToggleReproductionCrossover( void);
BOOL GetReproductionCrossoverState( void);
void ToggleReproductionLinear( void);
BOOL GetReproductionLinearState( void);
void ToggleReproductionCircular( void);
BOOL GetReproductionCircularState( void);
void ToggleReproductionReduction( void);
BOOL GetReproductionReductionState( void);

void SetEstimatorToMean( void);
void SetEstimatorToMedian( void);
void SetEstimatorToModeMean( void);
void SetEstimatorToModeMedian( void);
int GetEstimator( void );
#endif

HANDLE GetParentInstanceHandle( void );

instance.h

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include "genetype.h"
#include "listmgr.h"
#include "random.h"
#include "numunits.h"

#define LIST_SIZE (NUMBER_OF_UNITS-1)
#define TEMP_LIST_SIZE (2*(LIST_SIZE))
#define LIST_EMPTY ((NODE)-1)
#define DELETED ((NODE)-1)
#define MAKENODE(a) ((NODE)(a))

static NODE __huge hpnodeTempList1[TEMP_LIST_SIZE];
static NODE __huge hpnodeTempList2[TEMP_LIST_SIZE];

void ListMgr_CopyList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nLength );
void ListMgr_RemoveListRedundancies( HPNODE hpnodeOut, HPNODE hpnodeIn, int nLength );

void CreateRandomList( HPNODE hpnodeList )
{ int i, nNewIndex;
  BOOL bDirection;

  /* Set up Lists for scrambling */
  for( i = 0; i < LIST_SIZE; i++ )
  { hpnodeTempList1[i] = MAKENODE(i+1);
    hpnodeList[i] = LIST_EMPTY;
  }

  /* Scramble lists */
  for( i = 0; i < LIST_SIZE; i++ )
  { nNewIndex = GetUniformInteger( LIST_SIZE-1 );
    bDirection = TestBinaryProbability( ((float)0.5) );
    while( hpnodeList[nNewIndex] != LIST_EMPTY )
    {
      if( bDirection == TRUE ) nNewIndex++;
      else nNewIndex--;
      if( nNewIndex < 0 ) nNewIndex = LIST_SIZE-1;
      if( nNewIndex == LIST_SIZE ) nNewIndex = 0;
    }
    hpnodeList[nNewIndex] = hpnodeTempList1[i];
  }
}
void

BOOL{

CreateDirectList( HPNODE hpnodeList, NODE nodeTarget )
{
int i, nListIndex;
NODE nodeNewNode;

hpnodeList[0] = nodeTarget;
nListIndex = 1;
for ( i = 0; i < LIST_SIZE; i++ )
{
    nodeNewNode = MAKE_NODE(i+1);
    if ( nodeNewNode == nodeTarget ) continue;
    hpnodeList[nListIndex++] = nodeNewNode;
}
}

BOOL RandomLinearListInversion( HPNODE hpnodeOut, HPNODE hpnodeIn, int nSize, int nMax)
{
int nEnd, nStart, nRange;
BOOL bIsValid;

if ( nSize <= 0 )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}
if ( nSize >= LIST_SIZE )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}
if ( nMax <= 0 )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}
if ( nMax >= LIST_SIZE )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}

nRange = nMax - nSize;
if ( nRange <= 0 )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}

nStart = GetUniformInteger( nRange );
nEnd = nStart + nSize;

bIsValid = LinearlyInvertList( hpnodeOut, hpnodeIn, nStart, nEnd );
return bIsValid;
}

BOOL LinearlyInvertList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nStart, int nEnd)
{
int i, nIndex;
NODE nodeCurrent;

if ( nStart < 0 )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}
if ( nEnd < 0 )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}
if ( nEnd < nStart )
{

}
BOOL
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}

if( nStart >= LIST_SIZE )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}

if( nEnd >= LIST_SIZE )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    return FALSE;
}

ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );

for( i = nStart; i <= nEnd; i++ )
{
    nodeCurrent = hpnodeIn[i];
    nIndex = nStart + nEnd - i;
    hpnodeOut[nIndex] = nodeCurrent;
}

return TRUE;
}

BOOL RandomCircularListInversion( HPNODE hpnodeOut, HPNODE hpnodeIn, int nSize )
{
    int nEnd, nStart;
    BOOL bIsValid;

    if( nSize <= 0 )
    {
        ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
        return FALSE;
    }

    if( nSize >= LIST_SIZE )
    {
        ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
        return FALSE;
    }

    nStart = GetUniformInteger( LIST_SIZE-1 );
    nEnd = nStart + nSize;
    if( nEnd >= LIST_SIZE ) nEnd -= LIST_SIZE;

    bIsValid = CircularlyInvertList( hpnodeOut, hpnodeIn, nStart, nEnd );
    return bIsValid;
}

BOOL CircularlyInvertList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nStart, int nEnd )
{
    int i, nSwapSize, nFromIndex, nToIndex;
    NODE nodeCurrent;
    BOOL bResult;

    if( nStart < 0 )
    {
        ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
        return FALSE;
    }

    if( nEnd < 0 )
    {
        ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
        return FALSE;
    }

    if( nStart >= LIST_SIZE )
    {
        ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
        return FALSE;
    }

    if( nEnd >= LIST_SIZE )
    {
        ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );
    }

    return bResult;
}
return FALSE;
}

ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE );

if( nEnd >= nStart )
{
    bResult = LinearlyInvertList( hpnodeOut, hpnodeIn, nStart, nEnd );
    return bResult;
}

nSwapSize = LIST_SIZE + 1 - nStart + nEnd;
for( i = 0; i < nSwapSize; i++ )
{
    nFromIndex = i + nStart;
    if( nFromIndex >= LIST_SIZE ) nFromIndex -= LIST_SIZE;
    nToIndex = nEnd - i;
    if( nToIndex < 0 ) nToIndex += LIST_SIZE;
    nodeCurrent = hpnodeIn[nFromIndex];
    hpnodeOut[nToIndex] = nodeCurrent;
}
return TRUE;

}

void CreateRandomIndirectList( HPNODE hpnodeList, NODE nodeTarget, int nTargetIndex )
{
    int nListIndex;
    NODE nodeNewNode;

    CreateRandomList( hpnodeList );
    nodeNewNode = hpnodeList[nTargetIndex];
    if( nodeNewNode == nodeTarget ) return;

    FindTargetNode( hpnodeList, &nListIndex, nodeTarget );
    hpnodeList[nTargetIndex] = nodeTarget;
    hpnodeList[nListIndex] = nodeNewNode;
}

BOOL FindTargetNode( HPNODE hpnodeList, int *nTargetIndex, NODE nodeTarget )
{
    int i;
    NODE nodeTestNode;

    for( i = 0; i < LIST_SIZE; i++ )
    {
        nodeTestNode = hpnodeList[i];
        if( nodeTestNode == nodeTarget )
        {
            *nTargetIndex = i;
            return TRUE;
        }
    }
*ntargetIndex = 0;
return FALSE;
}

BOOL CrossoverLists( HPNODE hpnodeOut1, HPNODE hpnodeOut2,
                     HPNODE hpnodeIn1, HPNODE hpnodeIn2, int nSwapSize )
{
    int nStart, nLastElement, nCopy, nTotalLength;

    nStart = GetUniformInteger( LIST_SIZE-1-nSwapSize );

    ListMgr_CopyList( hpnodeTempList1, hpnodeIn1, LIST_SIZE );
    ListMgr_CopyList( hpnodeTempList2, hpnodeIn2, LIST_SIZE );
    nLastElement = LIST_SIZE-1;
    /* Set up copy length */
BOOL ReduceList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nTargetIndex )
{
    int ndxReduction, nCopy;
    NODE nodeReduced;

    if( nTargetIndex == 0 )
    {
        CreateRandomList( hpnodeOut );
        return FALSE;
    }

    if( nTargetIndex == 1 )
    {
        ndxReduction = 0;
    }
    else
    {
        ndxReduction = GetUniformInteger( nTargetIndex );
        if( ndxReduction == nTargetIndex ) ndxReduction = 0;
    }

    nodeReduced = hpnodeIn[ndxReduction];
    nCopy = ndxReduction;
    ListMgr_CopyList( hpnodeOut, hpnodeIn, nCopy );
    nCopy = LIST_SIZE-ndxReduction-1;
    ListMgr_CopyList( &hpnodeOut[ndxReduction], &hpnodeIn[ndxReduction+1], nCopy );
    hpnodeOut[LIST_SIZE] = nodeReduced;
    return TRUE;
}

void CopyList( HPNODE hpnodeOut, HPNODE hpnodeIn )
{
    ListMgr_CopyList( hpnodeOut, hpnodeIn, LIST_SIZE);
}

void ListMgr_CopyList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nLength )
{
    int i;

    if( nLength <= 0 ) return;

    for( i = 0; i < nLength; i++ )
    {
        hpnodeOut[i] = hpnodeIn[i];
    }
}
```c
void ListMgr_RemoveListRedundancies( HPNODE hpnodeOut, HPNODE hpnodeIn, int nLength )
{
    int i, nAhead, nGood;
    NODE nodeTestNode;
    nGood = 0;
    for( i = 0; i < nLength; i++ )
    {
        nodeTestNode = hpnodeIn[i];
        if( nodeTestNode == DELETED ) continue;
        hpnodeIn[i] = DELETED;
        hpnodeOut[nGood++] = nodeTestNode;
        for( nAhead = i+1; nAhead < nLength; nAhead++ )
        {
            if( hpnodeIn[nAhead] == nodeTestNode )
            {
                hpnodeIn[nAhead] = DELETED;
                break;
            }
        }
    }
}

listmgr.h
#endif
#include "genetype.h"
void CreateRandomList( HPNODE hpnodeList );
BOOL FindTargetNode( HPNODE hpnodeList, int *nTargetIndex, NODE nodeTarget );
BOOL CrossoverLists( HPNODE hpnodeOut1, HPNODE hpnodeOut2,
    HPNODE hpnodeIn1, HPNODE hpnodeIn2, int nSwapSize );
BOOL ReduceList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nTargetIndex );
void CreateDirectList( HPNODE hpnodeList, NODE nodeTarget );
void CreateRandomIndirectList( HPNODE hpnodeList, NODE nodeTarget, int nTargetIndex );
void CopyList( HPNODE hpnodeOut, HPNODE hpnodeIn );
BOOL RandomLinearListInversion( HPNODE hpnodeOut, HPNODE hpnodeIn, int nSize, int nMax );
BOOL LinearlyInvertList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nStart, int nEnd );
BOOL RandomCircularListInversion( HPNODE hpnodeOut, HPNODE hpnodeIn, int nSize );
BOOL CircularlyInvertList( HPNODE hpnodeOut, HPNODE hpnodeIn, int nStart, int nEnd );
#endif

metrics.h
#define PIXELS_PER_METER 10.0
#define METERS_PER_PIXEL 0.1
#define UNIT_HALFSIZE_X 1.0
#define UNIT_HALFSIZE_Y 1.0
#define UNIT_DIAMETER 1.0
#define MAXIMUM_RANGE ( (float)20.0 )
#endif
```
# Data Arrays */
POINT axyScreenCoordinates[NUMBER_OF_UNITS];
FCOORD axyTrueCoordinates[NUMBER_OF_UNITS];
FCOORD _huge axyMeasurement[NUMBER_OF_UNITS][NUMBER_OF_UNITS];
TMSTMP atmstmpMeasurementTimeStamp[NUMBER_OF_UNITS][NUMBER_OF_UNITS];
BOOL abMeasurementValid[NUMBER_OF_UNITS][NUMBER_OF_UNITS];

/* Function Declarations */
BOOL MsmtDB_ValidateIndex( int nIndex );

/* Function Code */
BOOL InitializeMeasurementDatabase( HWND hParent )
{
    HWND hTemp;
    TMSTMP tmstmpCurrent;
    int i, j;

    /* Debug only */
    FCOORD _huge * fxyPtr;
    hTemp = hParent;

    tmstmpCurrent = GetCurrentTimeStamp();
    for( i = 0; i < NUMBER_OF_UNITS; i++ )
    {
        for( j = 0; j < NUMBER_OF_UNITS; j++ )
        {
            atmstmpMeasurementTimeStamp[i][j] = tmstmpCurrent;
            abMeasurementValid[i][j] = FALSE;
        }
    }
    return TRUE;
}

void LobotomizeMeasurementDatabase( void )
{
    TMSTMP tmstmpCurrent;
    int i, j;

    tmstmpCurrent = GetCurrentTimeStamp();
    for( i = 0; i < NUMBER_OF_UNITS; i++ )
    {
        for( j = 0; j < NUMBER_OF_UNITS; j++ )
        {
            atmstmpMeasurementTimeStamp[i][j] = tmstmpCurrent;
            abMeasurementValid[i][j] = FALSE;
        }
    }
}

BOOL DestroyMeasurementDatabase( HWND hParent )
{
    HWND hTemp;
    hTemp = hParent;
    return TRUE;
}

182
BOOL MsmtDB_ValidateIndex( int nIndex )
{
    if( nIndex < 0 ) return FALSE;
    if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
    return TRUE;
}

BOOL StoreScreenCoordinates( int nIndex, POINT *xyScreen )
{
    BOOL bTest;
    bTest = MsmtDB_ValidateIndex( nIndex );
    if( bTest == FALSE ) return FALSE;
    axyScreenCoordinates[nIndex].x = xyScreen->x;
    axyScreenCoordinates[nIndex].y = xyScreen->y;
    return TRUE;
}

BOOL GetScreenCoordinates( int nIndex, POINT *xyScreen )
{
    BOOL bTest;
    bTest = MsmtDB_ValidateIndex( nIndex );
    if( bTest == FALSE ) return FALSE;
    xyScreen->x = axyScreenCoordinates[nIndex].x;
    xyScreen->y = axyScreenCoordinates[nIndex].y;
    return TRUE;
}

BOOL StoreTrueCoordinates( int nIndex, FCOORD *fxyTrue )
{
    BOOL bTest;
    bTest = MsmtDB_ValidateIndex( nIndex );
    if( bTest == FALSE ) return FALSE;
    afxyTrueCoordinates[nIndex].x = fxyTrue->x;
    afxyTrueCoordinates[nIndex].y = fxyTrue->y;
    return TRUE;
}

BOOL GetTrueCoordinates( int nIndex, FCOORD *fxyTrue )
{
    BOOL bTest;
    bTest = MsmtDB_ValidateIndex( nIndex );
    if( bTest == FALSE ) return FALSE;
    fxyTrue->x = afxyTrueCoordinates[nIndex].x;
    fxyTrue->y = afxyTrueCoordinates[nIndex].y;
    return TRUE;
}

BOOL StoreMeasurement( int nFromIndex, int nToIndex, FCOORD *fxyMeasurement )
{
    BOOL bTest;
    TMSTMP tmstmpCurrent;
    bTest = MsmtDB_ValidateIndex( nFromIndex );
    if( bTest == FALSE ) return FALSE;
    bTest = MsmtDB_ValidateIndex( nToIndex );
    if( bTest == FALSE ) return FALSE;
    tmstmpCurrent = GetCurrentTimeStamp();
BOOL GetMeasurement( int nFromIndex, int nToIndex, FCOORD *fxyMeasurement)
{
    BOOL bTest;
    TMSTMP tmstmpTest;

    /* Check validity of indices */
    bTest = MsmtDB_ValidateIndex( nFromIndex );
    if( bTest == FALSE ) return FALSE;
    bTest = MsmtDB_ValidateIndex( nToIndex );
    if( bTest == FALSE ) return FALSE;

    /* Check validity flag */
    bTest = abMeasurementValid[nFromIndex][nToIndex];
    if( bTest == FALSE ) return FALSE;

    /* Test for expired time stamp */
    tmstmpTest = atmstmpMeasurementTimeStamp[nFromIndex][nToIndex];
    bTest = ValidateTimeStamp( tmstmpTest );
    if( bTest == FALSE )
    {
        abMeasurementValid[nFromIndex][nToIndex] = FALSE;
        return FALSE;
    }

    fxyMeasurement->x = afxyMeasurement[nFromIndex][nToIndex].x;
    fxyMeasurement->y = afxyMeasurement[nFromIndex][nToIndex].y;

    return TRUE;
}

BOOL ConvertScreenToTrueAndSave( int nIndex)
{
    BOOL bTest;
    float fTrueX, fTrueY, fScreenX, fScreenY;

    bTest = MsmtDB_ValidateIndex( nIndex );
    if( bTest == FALSE ) return FALSE;

    fScreenX = (float) axyScreenCoordinates[nIndex].x;
    fScreenY = (float) axyScreenCoordinates[nIndex].y;

    fTrueX = ((float)UNIT_HALFSIZE_X) + fScreenX*(float)METERS_PER_PIXEL;
    fTrueY = -((float)UNIT_HALFSIZE_Y) - fScreenY*(float)METERS_PER_PIXEL;

    afxyTrueCoordinates[nIndex].x = fTrueX;
    afxyTrueCoordinates[nIndex].y = fTrueY;

    return TRUE;
}

BOOL ConvertTrueToScreenAndSave( int nIndex)
{
    BOOL bTest;
    float fTrueX, fTrueY, fScreenX, fScreenY;

    bTest = MsmtDB_ValidateIndex( nIndex );
    if( bTest == FALSE ) return FALSE;
fTrueX = afxyTrueCoordinates[nIndex].x;
fTrueY = afxyTrueCoordinates[nIndex].y;

fScreenX = ((float)PIXELS_PER_METER)*(fTrueX - ((float)UNIT_HALFSIZE_X));
fScreenY = ((float)PIXELS_PER_METER)*(-fTrueY + ((float)UNIT_HALFSIZE_Y));

axyScreenCoordinates[nIndex].x = (int) fScreenX;
axyScreenCoordinates[nIndex].y = (int) fScreenY;

return TRUE;

msmtdb.h

#ifndef MSMTDB_H
#define MSMTDB_H

#include "timestmp.h"
#define FCOORD_H
include "fcoord.h"

/* Function Declarations */

BOOL LobotomizeMeasurementDatabase( void);
BOOL InitializeMeasurementDatabase( HWND hParent );
BOOL DestroyMeasurementDatabase( HWND hParent );
void StoreScreenCoordinates( int nIndex, POINT *xyScreen );
BOOL GetScreenCoordinates( int nIndex, POINT *xyScreen );
BOOL StoreTrueCoordinates( int nIndex, FCOORD *fxyTrue );
BOOL GetTrueCoordinates( int nIndex, FCOORD *fxyTrue );
BOOL StoreMeasurement( int nFromIndex, int nToIndex, FCOORD *fxyMeasurement );
BOOL GetMeasurement( int nFromIndex, int nToIndex, FCOORD *fxyMeasurement );
void ConvertScreenToTrueAndSave( int nIndex );
BOOL ConvertTrueToScreenAndSave( int nIndex );

#endif

msmtmkr.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include <math.h>

#include "fcoord.h"
#include "msmtmkr.h"
#include "msmtdb.h"
#include "populmgr.h"
#include "numunits.h"
#include "metrics.h"
#include "random.h"
#include "glblopts.h"

#define TWO_PI 6.283185308
#define BEARING_SCALE_NICE 0.0261799 /* PI/120 */
#define BEARING_SCALE_GOOD 0.1308997 /* PI/24 */
#define BEARING_SCALE_BAD 0.2617994 /* PI/12 */
#define RANGE_SCALE_NICE 0.1 /* Meters */
#define RANGE_SCALE_GOOD 0.5 /* Meters */
#define RANGE_SCALE_BAD 1.0  /* Meters */
#define FLOAT_ONE ((float)1.0)
#define NEW_WT ((float)0.4)
#define OLD_WT (FLOAT_ONE - NEW_WT)

float __huge afRanges[NUMBER_OF_UNITS][NUMBER_OF_UNITS];
float __huge afBearings[NUMBER_OF_UNITS][NUMBER_OF_UNITS];
BOOL abWasVisible[NUMBER_OF_UNITS][NUMBER_OF_UNITS];

static FCOORD afxyUnit[NUMBER_OF_UNITS];
static float afUnitRange[NUMBER_OF_UNITS];
static float afUnitThetaCenter[NUMBER_OF_UNITS];
static float afUnitThetaLeft[NUMBER_OF_UNITS];
static float afUnitThetaRight[NUMBER_OF_UNITS];
static BOOL abIsNowVisible[NUMBER_OF_UNITS];
static FCOORD fxyBase;

BOOL MsmtMkr_GetBaseCoordinates( int nIndex);
void MsmtMkr_GetOtherUnitCoordinatesFromBase( int nIndex);
void MsmtMkr_DetermineShadowed( int nIndex);
void MsmtMkr_MakeNoisyMeasurements( int nPromIndex);
BOOL MsmtMkr_DetermineCommLoss( int nCommLoss);
void MsmtMkr_AddNoiseToPolar( LPFCOORD fxyNoisyPolar, LPFCOORD fxyTruePolar, int nNoiseModel, int nErrorScale);
BOOL MsmtMkr_DetermineBracketing( float fLeft, float fCenter, float fRight);
void MsmtMkr_GetPolarFromXY( LPFCOORD fxyPolar, LPFCOORD fxyXY);
void MsmtMkr_GetXYFromPolar( LPFCOORD fxyXY, LPFCOORD fxyPolar);

void InitializeMeasurementMakerFilters( void )
{
  int i, j;
  for( i = 0; i < NUMBER_OF_UNITS; i++ )
    { for( j = 0; j < NUMBER_OF_UNITS; j++ )
      {
        abWasVisible[i][j] = FALSE;
      } }

BOOL ProcessMeasurementSlice( int nIndex )
{
  /* Ensure Sensors are active */
  BOOL bSensorState, bValid;
  bValid = GetSensorStateFromIndex( nIndex, &bSensorState );
  if( bValid == FALSE ) return FALSE;
  if( bSensorState == FALSE ) return TRUE;
  bValid = MsmtMkr_GetBaseCoordinates( nIndex );
  if( bValid == FALSE ) return FALSE;
  MsmtMkr_GetOtherUnitCoordinatesFromBase( nIndex );
  MsmtMkr_DetermineShadowed( nIndex );
  MsmtMkr_MakeNoisyMeasurements( nIndex );
  return TRUE;
}

BOOL MsmtMkr_GetBaseCoordinates( int nIndex )
{
  return GetTrueCoordinates( nIndex, &fxyBase );
}
void MsmtMkr_GetOtherUnitCoordinatesFromBase( int nIndex )
{
    int i;
    FCOORD fxyTrueXY, fxyTempXY, fxyTempPolar;
    float fDeltaTheta;
    BOOL bValid;
    for( i = 0; i < NUMBER_OF_UNITS; i++ )
    {
        abIsNowVisible[i] = FALSE;
        if( i == nIndex ) continue;
        bValid = GetTrueCoordinates( i, &fxyTrueXY );
        if( bValid == FALSE ) continue;
        fxyTempXY.x = fxyTrueXY.x - fxyBase.x;
        if( fxyTempXY.x > MAXIMUM_RANGE ) continue;
        if( fxyTempXY.x < -MAXIMUM_RANGE ) continue;
        fxyTempXY.y = fxyTrueXY.y - fxyBase.y;
        if( fxyTempXY.y > MAXIMUM_RANGE ) continue;
        if( fxyTempXY.y < -MAXIMUM_RANGE ) continue;
        MsmtMkr_GetPolarFromXY( &fxyTempPolar, &fxyTempXY );
        if( fxyTempPolar.x > MAXIMUM_RANGE ) continue;
        /* Definitely in range, so store it */
        abIsNowVisible[i] = TRUE;
        afxyUnit[i].x = fxyTempXY.x;
        afxyUnit[i].y = fxyTempXY.y;
        afUnitRange[i] = fxyTempPolar.x;
        afUnitThetaCenter[i] = fxyTempPolar.y;
        fDeltaTheta = ((float) atan2(UNIT_DIAMETER, ((double) (fxyTempPolar.x)) ));
        afUnitThetaLeft[i] = fxyTempPolar.y + fDeltaTheta;
        afUnitThetaRight[i] = fxyTempPolar.y - fDeltaTheta;
    }
}

void MsmtMkr_DetermineShadowed( int nIndex )
{
    int i, j;
    float fRange, fLeft, fRight;
    float fTestRange, fCenter;
    BOOL bBracketed, bIsVisible;
    for( i = 0; i < NUMBER_OF_UNITS; i++ )
    {
        if( i == nIndex ) continue;
        bIsVisible = abIsNowVisible[i];
        if( bIsVisible == FALSE ) continue;
        fRange = afUnitRange[i];
        fLeft = afUnitThetaLeft[i];
        fRight = afUnitThetaRight[i];
        for( j = 0; j < NUMBER_OF_UNITS; j++ )
        {
            if( i == j ) continue;
            if( j == nIndex ) continue;
            bIsVisible = abIsNowVisible[j];
            if( bIsVisible == FALSE ) continue;
            fTestRange = afUnitRange[j];
if ( fTestRange <= fRange ) continue;

fCenter = afUnitThetaCenter[j];
bBracketed = MsmtMkr_DetermineBracketing( fLeft, fCenter, fRight );
if ( bBracketed == TRUE )
{
    abIsNowVisible[j] = FALSE;
}

void MsmtMkr_MakeNoisyMeasurements( int nFromIndex )
{
    int i, nCommLoss, nNoiseModel, nErrorScale;
    BOOL bisVisible, bWasVisible, bCommLoss;
    FCOORD fxyMeasurement, fxyTruePolar, fxyNoisyPolar, fxyPrefiltered;

    nCommLoss = GetCommLoss();
    nNoiseModel = GetNoiseModel();
    nErrorScale = GetErrorScale();
    for ( i = 0; i < NUMBER_OF_UNITS; i++ )
    {
        if ( i == nFromIndex ) continue;
        bisVisible = abIsNowVisible[i];
        if ( bisVisible == FALSE ) continue;

        fxyTruePolar.x = afUnitRange[i];
        fxyTruePolar.y = afUnitThetaCenter[i];
        MsmtMkr_AddNoiseToPolar( &fxyNoisyPolar, &fxyTruePolar,
                                nNoiseModel, nErrorScale );

        bWasVisible = abWasVisible[nFromIndex][i];
        if ( bWasVisible == FALSE )
        {
            else
            fxyPrefiltered.x = fxyNoisyPolar.x;
            fxyPrefiltered.y = fxyNoisyPolar.y;
        }
        else
        {
            fxyPrefiltered.x = (NEW_WT*fxyNoisyPolar.x) +
                               (OLD_WT*afRanges[nFromIndex][i]);
            fxyPrefiltered.y = (NEW_WT*fxyNoisyPolar.y) +
                               (OLD_WT*afBearings[nFromIndex][i]);
        }
        afRanges[nFromIndex][i] = fxyPrefiltered.x;
        afBearings[nFromIndex][i] = fxyPrefiltered.y;

        MsmtMkr_GetXYFromPolar( &fxyMeasurement, &fxyPrefiltered );
        bCommLoss = MsmtMkr_DetermineCommLoss( nCommLoss );
        if ( bCommLoss == TRUE ) continue;
        StoreMeasurement( nFromIndex, i, &fxyMeasurement );
    }

    for ( i = 0; i < NUMBER_OF_UNITS; i++ )
    {
        bisVisible = abIsNowVisible[i];
        abWasVisible[nFromIndex][i] = bisVisible;
    }
}

BOOL MsmtMkr_DetermineCommLoss( int nCommLoss )
{
    double dRand, dMax;

    switch( nCommLoss )
void
  
case COMM_LOSS_5_PERCENT:
    dMax = 0.05;
    break;
  
case COMM_LOSS_20_PERCENT:
    dMax = 0.20;
    break;
  
case COMM_LOSS_40_PERCENT:
    dMax = 0.40;
    break;
  
default:
    return FALSE;
    break;
}

GetUnitUniform( &dRand );
if( dRand >= dMax ) return FALSE;
return TRUE;

MsmtMkr_AddNoiseToPolar( LPFCOORD fxyNoisyPolar, LPFCOORD fxyTruePolar,
  int nNoiseModel, int nErrorScale )
{
  float fRangeNoise, fBearingNoise;
  double dRand1, dRand2;

  switch( nNoiseModel )
  {
    case NOISE_MODEL_CAUCHY:
      GetUnitCauchy( &dRand1 );
      GetUnitCauchy( &dRand2 );
      break;
    
    case NOISE_MODEL_GAUSSIAN:
      GetStandardGaussian( &dRand1 );
      GetStandardGaussian( &dRand2 );
      break;
    
    default:
      dRand1 = 0.0;
      dRand2 = 0.0;
      break;
  }

  switch( nErrorScale )
  {
    case ERROR_SCALE_GOOD:
      fRangeNoise = (float)( RANGE_SCALE_NICE*dRand1 );
      fBearingNoise = (float)( BEARING_SCALE_NICE*dRand2 );
      break;
    
    case ERROR_SCALE_FAIR:
      fRangeNoise = (float)( RANGE_SCALE_GOOD*dRand1 );
      fBearingNoise = (float)( BEARING_SCALE_GOOD*dRand2 );
      break;
    
    case ERROR_SCALE_POOR:
      fRangeNoise = (float)( RANGE_SCALE_BAD*dRand1 );
      fBearingNoise = (float)( BEARING_SCALE_BAD*dRand2 );
      break;
    
    default:
      

BOOL BOOL
{
void

fRangeNoise = ((float)0.0);
fbearingNoise = ((float)0.0);
break;
}

fxyNoisyPolar->x = fxyTruePolar->x + fRangeNoise;
fxyNoisyPolar->y = fxyTruePolar->y + fbearingNoise;

BOOL MsmtMkr_DetermineBracketing( float fLeft, float fCenter, float fRight )
{
    /* Determine normal bracketing */
    if( (fCenter < fLeft) && (fCenter > fRight) ) return TRUE;

    /* Determine end bracketing */
    if( fCenter > 0 ) fCenter -= TWO_PI;
    else fCenter += TWO_PI;
    if( (fCenter < fLeft) && (fCenter > fRight) ) return TRUE;
    return FALSE;
}

void MsmtMkr_GetPolarFromXY( LPFCOORD fxyPolar, LPFCOORD fxyXY )
{
double dRange, dTheta;

dRange = _hypot( ((double)(fxyXY->x)), ((double)(fxyXY->y)) );
fxyPolar->x = (float) dRange;

dTheta = atan2( ((double)(fxyXY->y)), ((double)(fxyXY->x)) );
fxyPolar->y = (float) dTheta;
}

void MsmtMkr_GetXYFromPolar( LPFCOORD fxyXY, LPFCOORD fxyPolar )
{
float fX, fY;

fX = fxyPolar->x * ((float) cos( ((double) fxyPolar->y) ) );
fY = fxyPolar->x * ((float) sin( ((double) fxyPolar->y) ) );

fxyXY->x = fX;
fxyXY->y = fY;

msmtmkr.h

#ifndef MSMTMKR_H
#define MSMTMKR_H

BOOL ProcessMeasurementSlice( int nIndex);
void InitializeMeasurementMakerFilters( void );
#endif

numunits.h

#ifndef NUMUNITS_H
#define NUMUNITS_H

#define UNIT_ROWS 5
#define UNIT_COLUMNS 8
#define NUMBER_OF_UNITS (UNIT_ROWS*UNIT_COLUMNS)
#endif
paintmgr.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include "paintmgr.h"
#include "plotbmp.h"
#include "expmt.h"

BOOL PaintScreenHandler( HWND hwnd )
/* WM_PAINT handler function for graphical display */
{
    HDC hMemDC;
    PAINTSTRUCT ps;
    HBITMAP hBitmap;
    int nSizeX, nSizeY, nUpper, nLeft;

    hBitmap = GetForegroundBitmapHandle();
    BeginPaint( hwnd, &ps );
    nSizeX = ps.rcPaint.right - ps.rcPaint.left + 1;
    nSizeY = ps.rcPaint.bottom - ps.rcPaint.top + 1;
    nUpper = ps.rcPaint.top;
    nLeft = ps.rcPaint.left;

    hMemDC = CreateCompatibleDC( ps.hdc );
    SelectObject( hMemDC, hBitmap );
    BitBlt( ps.hdc, nLeft, nUpper, nSizeX, nSizeY, hMemDC, nLeft, nUpper, SRCCOPY );
    DeleteDC( hMemDC );
    EndPaint( hwnd, &ps );
    UpdatePacifer( hwnd );
    return TRUE;
}

paintmgr.h

#ifndef PAINTMGR_H
#define PAINTMGR_H

BOOL PaintScreenHandler( HWND hwnd );
#endif

#define STRICT
#define NOCOMM
#include <Windows.h>
#include <windowsx.h>
#include <math.h>
#include "pathdb.h"
#include "genetype.h"
#include "numunits.h"
#include "listmgr.h"

#include "msmtdb.h"
/* Index 0 represents spare path set */
static POPUL apopPop[NUMBER_OF_UNITS];
static GLOBALHANDLE ghndNodes;

pathdb.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include <math.h>

#include "pathdb.h"
#include "genotype.h"
#include "numunits.h"
#include "listmgr.h"
/* included only for data collection purposes */
#include "msmtdb.h"

/* Index 0 represents spare path set */
static POPUL apopPop[NUMBER_OF_UNITS];
static GLOBALHANDLE ghndNodes;
static HPNODE hugenodePtr;
static GLOBALHANDLE ghndPaths;
static HPPATH hugepathPtr;

void PathDB_GetNodeAllocationSize( DWORD *dwNodeAlloc );
void PathDB_GetPathAllocationSize( DWORD *dwPathAlloc );
BOOL PathDB_AllocateAndLockMemory( void );
void PathDB_DistributeMemory( void );
void PathDB_InitializeParameters( void );

BOOL InitializePathDatabase( void )
{
    BOOL bMemOK;
    bMemOK = PathDB_AllocateAndLockMemory();
    if( bMemOK == FALSE ) return FALSE;
    PathDB_DistributeMemory();
    PathDB_InitializeParameters();
    return TRUE;
}

BOOL DestroyPathDatabase( void )
{
    GlobalFree( ghndNodes );
    GlobalFree( ghndPaths );
    return TRUE;
}

void PathDB_InitializeParameters( void )
{
    int nPopindex, nPathindex;
    HPNODE hpnodeList;
    for( nPopindex = 0; nPopIndex < NUMBER_OF_UNITS; nPopIndex++ )
    {
        for( nPathIndex = 0; nPathIndex < POPULATION_SIZE; nPathIndex++ )
        {
            apopPop[nPopIndex].ahppathPaths[nPathIndex]->nodeTargetNode = (NODE) nPopIndex;
            apopPop[nPopIndex].ahppathPaths[nPathIndex]->nTargetIndex = 0;
            apopPop[nPopIndex].ahppathPaths[nPathIndex]->bIsValid = FALSE;
            hpnodeList = apopPop[nPopIndex].ahppathPaths[nPathIndex]->hpnodeNodeList;
            CreateDirectList( hpnodeList, (NODE) nPopIndex );
        }
        apopPop[nPopIndex].nodeTargetNode = (NODE) nPopIndex;
        apopPop[nPopIndex].bValidEstimate = FALSE;
        apopPop[nPopIndex].bValidObservation = FALSE;
        apopPop[nPopIndex].nNumValidPaths = 0;
        apopPop[nPopIndex].fTotalFitness = ((float)0.0);
    }
}

BOOL GetPathVectorsFromIndex( int nIndex, PCOORD *fxyVectors, int *nNumPaths )
{
    int i, nTempNum;
    BOOL bIsValid;
    if( nIndex <= 0 ) return FALSE;
    if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
    nTempNum = 0;
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        bIsValid = apopPop[nIndex].ahppathPaths[i]->bIsValid;
        if( bIsValid == FALSE ) break;
        nTempNum++;
    }
    return TRUE;
}
BOOL fxyVectors[i].x = apopPop[nIndex].ahppathPaths[i]->fxyPathVector.x;
fxvectors[i].y = apopPop[nIndex].ahppathPaths[i]->fxyPathVector.y;

*nNumPaths = nTempNum;
return TRUE;

BOOL GetFitnessScoresFromIndex( int nIndex, float *fScores, int *nNumPaths )
{
    int i, nTempNum;
    BOOL bIsValid;
    if( nIndex <= 0 ) return FALSE;
    if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
    nTempNum = 0;
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        bIsValid = apopPop[nIndex].ahppathPaths[i]->bIsValid;
        if( bIsValid == FALSE ) break;
        nTempNum++;
        fScores[i] = apopPop[nIndex].ahppathPaths[i]->fFitness;
    }
    *nNumPaths = nTempNum;
    return TRUE;
}

BOOL GetPathListFromIndex( int nIndex, HPPATH *ahpathPtr )
{
    int i;
    if( nIndex <= 0 ) return FALSE;
    if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        ahpathPtr[i] = apopPop[nIndex].ahppathPaths[i];
        ahpathPtr[i+POPULATION_SIZE] = apopPop[0].ahppathPaths[i];
    }
    return TRUE;
}

BOOL SavePathListFromIndex( int nIndex, HPPATH *ahpathPtr )
{
    int i;
    if( nIndex <= 0 ) return FALSE;
    if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        apopPop[nIndex].ahppathPaths[i] = ahpathPtr[i];
        apopPop[0].ahppathPaths[i] = ahpathPtr[i+POPULATION_SIZE];
    }
    return TRUE;
}

BOOL GetPopulationPointerFromIndex( int nIndex, LPPOPUL *ppopPop )
{
    if( nIndex <= 0 ) return FALSE;
    if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
    *ppopPop = &apopPop[nIndex];
    return TRUE;
}

void PathDB_DistributeMemory( void )
{
    int nPopIndex, nPathIndex;
    long lPathElement, lNodeElement;
    HPNODE hnodePtr;
    HPPATH hpathPtr;
BOOL lPathElement = (long)0;
BOOL lNodeElement = (long)0;
for( nPopIndex = 0; nPopIndex < NUMBER_OF_UNITS; nPopIndex++)
{
    for( nPathIndex = 0; nPathIndex < POPULATION_SIZE; nPathIndex++)
    {
        hpathPtr = &hugepathPtr[lPathElement];
        apopPop[nPopIndex].ahppathPaths[nPathIndex] = hpathPtr;
        hnodePtr = &hugenodePtr[lNodeElement];
        apopPop[nPopIndex].ahppathPaths[nPathIndex]->hpnodeNodeList = hnodePtr;
        lNodeElement += (long)(NUMBER_OF_UNITS);
        lPathElement++;
    }
}

BOOL PathDB_AllocateAndLockMemory( void
{  DWORD dwNodeSize, dwPathSize;
    PathDB_GetNodeAllocationSize( &dwNodeSize);
    PathDB_GetPathAllocationSize( &dwPathSize);

    ghndNodes = GlobalAlloc( GMEM_MOVEABLE, dwNodeSize);
    if( ghndNodes == NULL
    {    return FALSE;

    ghndPaths = GlobalAlloc( GMEM_MOVEABLE, dwPathSize);
    if( ghndPaths == NULL
    {    GlobalFree( ghndNodes);
        return FALSE;
    }

    hugenodePtr = (HPNODE) GlobalLock( ghndNodes);
    hugepathPtr = (HPPATH) GlobalLock( ghndPaths);
    return TRUE;
}

void PathDB_GetNodeAllocationSize( DWORD *dwNodeAlloc
{  DWORD dwNumUnits, dwNodeSize, dwPopulationSize;
    dwNumUnits = ((DWORD)( NUMBER_OF_UNITS ));
    dwNodeSize = ((DWORD)sizeof( NODE ));
    dwPopulationSize = ((DWORD)( POPULATION_SIZE ));
    *dwNodeAlloc = dwNumUnits*dwNumUnits*dwNodeSize*dwPopulationSize;
}

void PathDB_GetPathAllocationSize( DWORD *dwPathAlloc
{  DWORD dwNumUnits, dwPathSize, dwPopulationSize;
    dwNumUnits = ((DWORD)( NUMBER_OF_UNITS ));
    dwPathSize = ((DWORD)sizeof( PATH ));
    dwPopulationSize = ((DWORD)( POPULATION_SIZE ));
    *dwPathAlloc = dwNumUnits*dwPathSize*dwPopulationSize;
}

BOOL GetObservationFromindex( int nIndex, FCOORD *fxyObservation
{  BOOL bIsValid;
    if( nIndex <= 0 ) return FALSE;
BOOL StoreObservationFromIndex( int nIndex, FCOORD *fxyObservation )
{
if( nIndex <= 0 ) return FALSE;
if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
apopPop[nIndex].fxyObservation.x = fxyObservation->x;
apopPop[nIndex].fxyObservation.y = fxyObservation->y;
apopPop[nIndex].bValidObservation = TRUE;
return TRUE;
}

BOOL InvalidateObservationFromIndex( int nIndex )
{
if( nIndex <= 0 ) return FALSE;
if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
apopPop[nIndex].bValidObservation = FALSE;
return TRUE;
}

BOOL GetEstimateFromIndex( int nIndex, FCOORD *fxyEstimate )
{
BOOL bIsValid;
if( nIndex <= 0 ) return FALSE;
if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
bIsValid = apopPop[nIndex].bValidEstimate;
if( bIsValid == FALSE ) return FALSE;
fxyEstimate->x = apopPop[nIndex].fxyEstimate.x;
fxyEstimate->y = apopPop[nIndex].fxyEstimate.y;
return TRUE;
}

BOOL StoreEstimateFromIndex( int nIndex, FCOORD *fxyEstimate )
{
if( nIndex <= 0 ) return FALSE;
if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
apopPop[nIndex].fxyEstimate.x = fxyEstimate->x;
apopPop[nIndex].fxyEstimate.y = fxyEstimate->y;
apopPop[nIndex].bValidEstimate = TRUE;
return TRUE;
}

BOOL InvalidateEstimateFromIndex( int nIndex )
{
if( nIndex <= 0 ) return FALSE;
if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
apopPop[nIndex].bValidEstimate = FALSE;
return TRUE;
}

void LobotomizePopulation( void )
{
PathDBInitializeParameters();
}
void GetAveragePopulationSize( double *dPopSize )
{
    int i, nPopSum;
    double dResult;

    nPopSum = 0;
    for( i = 1; i < NUMBER_OF_UNITS; i++ )
    {
        nPopSum += apopPop[i].nNumValidPaths;
    }
    dResult = ((double)nPopSum)/((double)(NUMBER_OF_UNITS-1));
    *dPopSize = dResult;
}

void GetAverageFitnessScore( double *dFitness )
{
    int i, j, nNumPaths;
    double dResult;
    float fFitnessSum;
    BOOL bPathValid;

    fFitnessSum = ((float)0.0);
    nNumPaths = 0;
    for( i = 1; i < NUMBER_OF_UNITS; i++ )
    {
        for( j = 0; j < POPULATION_SIZE; j++ )
        {
            bPathValid = apopPop[i].ahppathPaths[j]->bIsValid;
            if( bPathValid == FALSE ) continue;
            fFitnessSum += apopPop[i].ahppathPaths[j]->fFitness;
            nNumPaths++;
        }
    }
    if( nNumPaths != 0 )
    {
        dResult = ((double)fFitnessSum)/((double)(nNumPaths));
    }
    else
    {
        dResult = 100.0;
    }
    *dFitness = dResult;
}

void GetAverageEstimateError( double *dEstError, int *nNumEstimates )
{
    int i, nEstimates;
    double dDeltaX, dDeltaY;
    FCORRD fxyTrue, fxySource;
    BOOL bIsValid;
    double dDistance, dEstErrorSum, dResult;

    dEstErrorSum = 0.0;
    nEstimates = 0;
    GetTrueCoordinates( 0, &fxySource );
    for( i = 1; i < NUMBER_OF_UNITS; i++ )
    {
        GetTrueCoordinates( i, &fxyTrue );
        bIsValid = apopPop[i].bValidEstimate;
        if( bIsValid == FALSE ) continue;
        dDeltaX = ((double)(fxyTrue.x - fxySource.x - apopPop[i].fxyEstimate.x));
        dDeltaY = ((double)(fxyTrue.y - fxySource.y - apopPop[i].fxyEstimate.y));
        dDistance = _hypot( dDeltaX, dDeltaY );
        dEstErrorSum += dDistance;
        nEstimates++;
    }
    if( nEstimates != 0 )
    {
        dResult = dEstErrorSum/((double)(nEstimates));
    }
else
{
    dResult = 100.0;
}
*dEstError = dResult;
*nNumEstimates = nEstimates;

#endif

#define STRICT
#define NOCOMM
#include <windows.h>
#include <Windowsx.h>
#include "pathmgr.h"
#include "genetype.h"
#include "pathdb.h"
#include "numunits.h"
#include "random.h"
#include "listmgr.h"
#include "msmtdb.h"
#include "math.h"
#include "populmgr.h"
#include "glblopts.h"

#define SCORES_SCALE  (((float)0.0025)
#define FLOAT_ZERO   (((float)0.0)

static HPPATH ahpPaths[2*POPULATION_SIZE];
static NODE nodeTarget;
void PathMgr_RescoreOldPaths( void );

pathmgr.c
void PathMgr_CreateRandomPath( HPATH hpPath );
void PathMgr_ExecuteRandomization( void );
void PathMgr_ExecuteCrossover( void );
void PathMgr_ExecuteLinearInversion( void );
void PathMgr_ExecuteCircularInversion( void );
void PathMgr_ExecuteReduction( void );
void PathMgr_ExecuteOutsideCrossMatching( int nIndex );
void PathMgr_ExecuteInsideCrossMatching( void );
void PathMgr_ScorePopulation( int nIndex );
void PathMgr_ScorePath( HPATH hpPath );
void PathMgr_SortPathList( void );
int PathMgr_GetSwapSize( void );
BOOL PathMgr_GetLinkLength( int nFromIndex, int nToIndex, float *fLength, FCOORD *fxyVector );
void PathMgr_RemoveRedundantPaths( void );
BOOL PathMgr_CreateDirectPath( HPATH hpPath );
void PathMgr_CreateRandomIndirectPath( HPATH hpPath );
void PathMgr_CreateInsidePath( HPATH hpNewPath, HPATH hpOldPath, HPATH hpTestPath, int nSubTargetIndex );
uint ReproducePathPopulation( int nIndex );

void ReproducePathPopulation( int nIndex )
{
    BOOL bState;
    if( nIndex <= 0 ) return;
    if( nIndex >= NUMBER_OF_UNITS ) return;

    bState = GetReproductionDirectState();
    if( bState == TRUE )
    {
        PathMgr_ReproducePopulation( nIndex, REPRODUCTION_DIRECT );
    }

    bState = GetReproductionRandomState();
    if( bState == TRUE )
    {
        PathMgr_ReproducePopulation( nIndex, REPRODUCTION_RANDOMIZE );
    }

    bState = GetReproductionOutsideState();
    if( bState == TRUE )
    {
        PathMgr_ReproducePopulation( nIndex, REPRODUCTION_CROSSMATCH_OUTSIDE );
    }

    bState = GetReproductionInsideState();
    if( bState == TRUE )
    {
        PathMgr_ReproducePopulation( nIndex, REPRODUCTION_CROSSMATCH_INSIDE );
    }

    bState = GetReproductionCrossoverState();
    if( bState == TRUE )
    {
        PathMgr_ReproducePopulation( nIndex, REPRODUCTION_CROSSOVER );
    }

    bState = GetReproductionLinearState();
    if( bState == TRUE )
    {
        PathMgr_ReproducePopulation( nIndex, REPRODUCTION_LINEAR_INVERSION );
    }

    bState = GetReproductionCircularState();
    if( bState == TRUE )
void PathMgr_ReproducePopulation( nindex, REPRODUCTION_CIRCULAR_INVERSION );

bState = GetReproductionReductionState();
if( bState == TRUE )
{
    PathMgr_ReproducePopulation( nIndex, REPRODUCTION_REDUCTION );
}

PathMgr_ScorePopulation( nIndex );

void PathMgr_ReproducePopulation( int nIndex, int nMethod )
{
    if( nIndex <= 0 ) return;
    if( nIndex >= NUMBER_OF_UNITS ) return;

    nodeTarget = (NODE)nIndex;
    GetPathListFromIndex( nIndex, &ahpPaths[0] );

    switch( nMethod )
    {
    case REPRODUCTION_RANDOMIZE:
        PathMgr_ExecuteRandomization();
        break;

    case REPRODUCTION_CROSSOVER:
        PathMgr_ExecuteCrossover();
        break;

    case REPRODUCTION_DIRECT:
        PathMgr_ExecuteDirectSearch();
        break;

    case REPRODUCTION_LINEAR_INVERSION:
        PathMgr_ExecuteLinearInversion();
        break;

    case REPRODUCTION_CIRCULAR_INVERSION:
        PathMgr_ExecuteCircularInversion();
        break;

    case REPRODUCTION_REDUCTION:
        PathMgr_ExecuteReduction();
        break;

    case REPRODUCTION_CROSSMATCH_OUTSIDE:
        PathMgr_ExecuteOutsideCrossMatching( nIndex );
        break;

    case REPRODUCTION_CROSSMATCH_INSIDE:
        PathMgr_ExecuteInsideCrossMatching();
        break;

    default:
        return;
    }

    PathMgr_ScrubPathPopulation();
    SavePathListFromIndex( nIndex, &ahpPaths[0] );
}

void PathMgr_RescoreOldPaths( void )
{
    int i;
}
for( i = 0; i < POPULATION_SIZE; i++ )
{
    PathMgr_ScorePath( ahpPaths[i] );
}

void PathMgr_ScrubPathPopulation( void )
{
    PathMgr_RescoreOldPaths();
    PathMgr_SortPathList();
    PathMgr_RemoveRedundantPaths();
    PathMgr_SortPathList();
}

void PathMgr_RemoveRedundantPaths( void )
{
    int i, j;
    BOOL bMatches;
    BOOL bisValid;
    for( i = 1; i < 2*POPULATION_SIZE; i++ )
    {
        bIsValid = ahpPaths[i]->bIsValid;
        if( bIsValid == FALSE ) return;
        for( j = 0; j < i; j++ )
        {
            bisValid = ahpPaths[j]->bIsValid;
            if( bisValid == FALSE ) continue;
            bMatches = PathMgr_ComparePathsForMatch(ahpPaths[j], ahpPaths[i]);
            if( bMatches == TRUE )
            {
                ahpPaths[j]->bIsValid = FALSE;
            }
        }
    }
}

BOOL PathMgr_ComparePathsForMatch( HPPATH hpPathl, HPPATH hpPath2 )
{ int i, nTargetIndex1, nTargetIndex2;
  float fFitnessl, fFitness2;
  NODE node1, node2;
  /* Compare target lengths */
  nTargetIndex1 = hpPathl->nTargetIndex;
  nTargetIndex2 = hpPath2->nTargetIndex;
  if( nTargetIndex1 != nTargetIndex2 ) return FALSE;
  /* compare fitness values */
  fFitness1 = hpPathl->fFitness;
  fFitness2 = hpPath2->fFitness;
  if( fFitness1 != fFitness2 ) return FALSE;
  /* compare node-by-node */
  for( i = 0; i < nTargetIndex1; i++ )
  {
      node1 = hpPathl->nodeTargetNode;
      node2 = hpPath2->nodeTargetNode;
      if( node1 != node2 ) return FALSE;
  }
  /* Paths match */
  return TRUE;
}

void PathMgr_CreateDirectPath( HPPATH hpPath )
{ HPNODE hpnodeList;
  hpnodeList = hpPath->hpnodeNodeList;
}
CreateDirectList( hpnodeList, nodeTarget );
hpPath->nTargetIndex = 0;
PathMgr_ScorePath( hpPath );
}

void PathMgr_CreateRandomIndirectPath( HPATH hpPath )
{ HNODE hpnodeList;
  hpnodeList = hpPath->hpnodeNodeList;
  CreateRandomIndirectList( hpnodeList, nodeTarget, 1 );
  hpPath->nTargetIndex = 1;
  PathMgr_ScorePath( hpPath );
}

void PathMgr_ExecuteDirectSearch( void )
{ int i;
  PathMgr_CreateDirectPath( ahpPaths[POPULATION_SIZE] );
  for( i = 1; i < POPULATION_SIZE; i++ )
  { PathMgr_CreateRandomIndirectPath( ahpPaths[i+POPULATION_SIZE] );
  }
}

void PathMgr_ExecuteRandomization( void )
{ int i;
  for( i = 0; i < POPULATION_SIZE; i++ )
  { PathMgr_CreateRandomPath( ahpPaths[i+POPULATION_SIZE] );
  }
}

void PathMgr_ExecuteCrossover( void )
{ int i, nTargetIndex, nHalfPopulation, nSwapSize,
  HNODE hpnodeIn1, hpnodeIn2, hpnodeOut1, hpnodeOut2,
  BOOL bIsValid;
  nHalfPopulation = POPULATION_SIZE/2;
  nSwapSize = PathMgr_GetSwapSize();
  for( i = 0; i < nHalfPopulation; i++ )
  { bIsValid = ahpPaths[2*i]->bIsValid;
    if( bIsValid == FALSE ) break;
    bIsValid = ahpPaths[2*i+1]->bIsValid;
    if( bIsValid == FALSE ) break;
    hpnodeIn1 = ahpPaths[2*i]->hpnodeNodeList;
    hpnodeIn2 = ahpPaths[2*i+1]->hpnodeNodeList;
    hpnodeOut1 = ahpPaths[2*i+POPULATION_SIZE]->hpnodeNodeList;
    hpnodeOut2 = ahpPaths[2*i+1+POPULATION_SIZE]->hpnodeNodeList;
  }
CrossoverLists( hpnodeOut1, hpnodeOut2, hpnodeIn1, hpnodeIn2, nSwapSize );
FindTargetNode( hpnodeOut1, &nTargetIndex, nodeTarget );
ahpPaths[2*i+POPULATION_SIZE]->nTargetIndex = nTargetIndex;
FindTargetNode( hpnodeOut2, &nTargetIndex, nodeTarget );
ahpPaths[2*i+1+POPULATION_SIZE]->nTargetIndex = nTargetIndex;
PathMgr_ScorePath( ahpPaths[2*i+POPULATION_SIZE] );
PathMgr_ScorePath( ahpPaths[2*i+1+POPULATION_SIZE] );

void PathMgr_ExecuteLinearInversion( void )
{
    int nSize, nMax, i, nTargetIndex;
    HPNODE hpnodeIn, hpnodeOut;
    BOOL bIsValid;
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        bIsValid = ahpPaths[i]->bIsValid;
        if( bIsValid == FALSE ) break;

        hpnodeIn = ahpPaths[i]->hpnodeNodeList;
        hpnodeOut = ahpPaths[i+POPULATION_SIZE]->hpnodeNodeList;
        nMax = ahpPaths[i]->nTargetIndex;
        nSize = GetUniformInteger( nMax/2 );

        RandomLinearListInversion( hpnodeOut, hpnodeIn, nSize, nMax );
        FindTargetNode( hpnodeOut, &nTargetIndex, nodeTarget );
        ahpPaths[i+POPULATION_SIZE]->nTargetIndex = nTargetIndex;
        PathMgr_ScorePath( ahpPaths[i+POPULATION_SIZE] );
    }
}

void PathMgr_ExecuteCircularInversion( void )
{
    int nSize, i, nTargetIndex;
    HPNODE hpnodeIn, hpnodeOut;
    BOOL bIsValid;
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        bIsValid = ahpPaths[i]->bIsValid;
        if( bIsValid == FALSE ) break;

        hpnodeIn = ahpPaths[i]->hpnodeNodeList;
        hpnodeOut = ahpPaths[i+POPULATION_SIZE]->hpnodeNodeList;
        nSize = GetUniformInteger( 10 );

        RandomCircularListInversion( hpnodeOut, hpnodeIn, nSize );
        FindTargetNode( hpnodeOut, &nTargetIndex, nodeTarget );
        ahpPaths[i+POPULATION_SIZE]->nTargetIndex = nTargetIndex;
        PathMgr_ScorePath( ahpPaths[i+POPULATION_SIZE] );
    }
}

void PathMgr_ExecuteReduction( void )
{
    int i, nTargetIndex;
    HPNODE hpnodeIn, hpnodeOut;
    BOOL bIsValid;
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        bIsValid = ahpPaths[i]->bIsValid;
        if( bIsValid == FALSE ) break;

        hpnodeIn = ahpPaths[i]->hpnodeNodeList;
        hpnodeOut = ahpPaths[i+POPULATION_SIZE]->hpnodeNodeList;
        nMax = ahpPaths[i]->nTargetIndex;
        nSize = GetUniformInteger( nMax/2 );

        RandomLinearListInversion( hpnodeOut, hpnodeIn, nSize, nMax );
        FindTargetNode( hpnodeOut, &nTargetIndex, nodeTarget );
        ahpPaths[i+POPULATION_SIZE]->nTargetIndex = nTargetIndex;
        PathMgr_ScorePath( ahpPaths[i+POPULATION_SIZE] );
    }
}
void PathMgr_ExecuteOutsideCrossMatching( int nIndex )
{
    int i, j, nPathIndex;
    LPPOPUL lppopPop;
    HPPATH hpCurrentPath, hpTestPath;
    BOOL bIsValid;

    nPathIndex = 0;
    hpCurrentPath = ahpPaths[nPathIndex+POPULATION_SIZE];
    for( j = 0; j < 5; j++ )
    {
        for( i = 1; i < NUMBER_OF_UNITS; i++ )
        {
            if( i == nIndex ) continue;
            GetPopulationPointerFromIndex( i, &lppopPop );
            hpTestPath = lppopPop->ahppathPaths[j];
            bIsValid = PathMgr_MatchOutsidePaths( hpCurrentPath, hpTestPath );
            if( bIsValid == TRUE )
            {
                PathMgr_ScorePath( hpCurrentPath );
                nPathIndex++;
                if( nPathIndex == POPULATION_SIZE ) return;
                hpCurrentPath = ahpPaths[nPathIndex+POPULATION_SIZE];
            }
        }
    }
}

BOOL PathMgr_MatchOutsidePaths( HPPATH hpNewPropPath, HPPATH hpOldPath )
{
    int nOldTargetIndex, nNewTargetIndex, nDisplacedIndex;
    int nFromIndex, nToIndex;
    NODE nodeDisplacedNode;
    HPNODE hpnodeOldList, hpnodeNewList;
    BOOL bIsValid;
    FCOORD fxyDummy;

    /* If old path is invalid, give up */
    if( bIsValid == hpNewPropPath->bIsValid; return TRUE;

    /* If old path includes target node, give up */
    hpnodeOldList = hpOldPath->hpnodeNodeList;
    FindTargetNode( hpnodeOldList, &nNewTargetIndex, nodeTarget );
    nOldTargetIndex = hpOldPath->nTargetIndex;
    if( nNewTargetIndex < nOldTargetIndex ) return FALSE;

    /* If old path target cannot see new target, give up */
    nFromIndex = ((int)hpNewPropPath->nodeTargetNode);
    nToIndex = ((int)nodeTarget);
    bIsValid = GetMeasurement( nFromIndex, nToIndex, &fxyDummy );
    if( bIsValid == FALSE ) return FALSE;

    /* Candidate old path is good for matching, so replicate */
    hpnodeNewPropPath = hpNewPropPath->hpnodeNodeList;
    CopyList( hpnodeNewPropPath, hpnodeOldList );

    return TRUE;
}
nDisplacedIndex = nOldTargetIndex+1;
nodDisplacedNode = hpnodeNewList[nDisplacedIndex];
hpnodeNewList[nDisplacedIndex] = nodeTarget;
hpnodeNewList[nNewTargetIndex] = nodDisplacedNode;

hpNewPath->nodeTargetNode = nodeTarget;
hpNewPath->nTargetIndex = nDisplacedIndex;
hpNewPath->bIsValid = FALSE;
return TRUE;

void PathMgr_ExecuteInsideCrossMatching( void)
{
    int i, j, nPathIndex, nTargetIndex, nSubTargetIndex, nTestPathIndex;
    NODE nodeSubNode;
    LPPOPUL lppopPop;
    HPPATH hpOldPath, hpNewPath, hpTestPath;
    BOOL bIsValid;

    nPathIndex = 0;
    for( j = 0; j < POPULATION_SIZE; j++ )
    {
        hpOldPath = ahpPaths[j];
        bIsValid = hpOldPath->bIsValid;
        if( bIsValid == FALSE ) continue;

        hpNewPath = ahpPaths[nPathIndex+POPULATION_SIZE];

        nTargetIndex = hpOldPath->nTargetIndex;
        for( i = 1; i < nTargetIndex; i++)
        {
            nodeSubNode = hpOldPath->hpnodeNodeList[i];
            nTestPathIndex = ((int)nodeSubNode);
            GetPopulationPointerFromIndex( nTestPathIndex, &lppopPop );

            /* Get best path to subnode */
            hpTestPath = lppopPop->ahppathPaths[0];
            bIsValid = hpTestPath->bIsValid;
            if( bIsValid == FALSE ) continue;

            nSubTargetIndex = hpTestPath->nTargetIndex;
            if( nSubTargetIndex == i ) continue;

            bIsValid = PathMgr_CreateInsidePath( hpNewPath, hpOldPath,
                                               hpTestPath, i );
            if( bIsValid == TRUE )
            {
                PathMgr_ScorePath( hpNewPath );
                nPathIndex++;
                if( nPathIndex == POPULATION_SIZE ) return;
                hpNewPath = ahpPaths[nPathIndex+POPULATION_SIZE];
            }
        }
    }
}

BOOL PathMgr_CreateInsidePath( HPPATH hpNewPath, HPPATH hpOldPath,
                              HPPATH hpTestPath, int nSubTargetIndex )
{/* Assume # is the subtarget node terminated in Test */
 /* Assume # is the target node terminated in Old */
 /* Old = {AAAAABBCCCC} */
 /* Test = {DDDEEEEEE} */
New = {DDDBBFFFGG}

where \{G\} = \{C\} - \{C&D\} and \{F\} = \{A\} - \{A&D\} * /

#if *

/* Calculate some substring lengths */

nLenA = nSubTargetIndex+1;

nLenB = nTargetIndexOld+1-nLenA;

nLenC = NUMBER_OF_UNITS-1-nLenA-nLenB;

nLenD = nTargetIndexTest+1;

nLenE = NUMBER_OF_UNITS-1-nLenD;

hpnOld = hpOldPath->hpnodeNodeList;

hpnNew = hpNewPath->hpnodeNodeList;

hpnTest = hpTestPath->hpnodeNodeList;

/* Find any common nodes between B & D. If any, quit */

for( i = 0; i < nLenB; i++ )
{
    nodeOld = hpnOld[nLenA+i];
    for( j = 0; j < nLenD; j++ )
    {
        nodeTest = hpnTest[j];
        if( nodeTest == nodeOld )
            return FALSE;
    }
}

/* Copy substring D into New */

for( i = 0; i < nLenD; i++ )
{
    nodeNew = hpnTest[i];
    hpnNew[i] = nodeNew;
}

/* Copy substring B into New */

for( i = 0; i < nLenB; i++ )
{
    nodeNew = hpnOld[nLenA+i];
    hpnNew[nLenD+i] = nodeNew;
}

/* Copy \{A\}-\{A&D\} into New */

nLenF = 0;

nBase = nLenD+nLenB;

for( i = 0; i < nLenA; i++ )
{
    nodeOld = hpnOld[i];
    bMatch = FALSE;
    for( j = 0; j < nLenD; j++ )
    {
        nodeTest = hpnTest[j];
        if( nodeTest == nodeOld )
            bMatch = TRUE;
        break;
    }
if (bMatch == FALSE) {
    hpnNew[nBase+nLenF] = nodeOld;
    nLenF++;
}

/* Copy {C}-{C@D} into New */
nLenG = 0;
nBase = nLenD+nLenB+nLenF;
for (i = 0; i < nLenC; i++) {
    nodeOld = hpnOld[nLenA+nLenB+i];
    bMatch = FALSE;
    for (j = 0; j < nLenD; j++) {
        nodeTest = hpnTest[j];
        if (nodeTest == nodeOld) {
            bMatch = TRUE;
            break;
        }
    }
    if (bMatch == FALSE) {
        hpnNew[nBase+nLenG] = nodeOld;
        nLenG++;
    }
}

/* Check lengths for consistency */
if (nLenB+nLenF+nLenG) != (nLenE) return FALSE;
nodeNew = hpnOld[nTargetIndexOld];
hpNewPath->nodeTargetNode = nodeNew;

nTargetIndexNew = nTargetIndexTest+nLenB;
hpNewPath->nTargetIndex = nTargetIndexNew;
hpNewPath->bIsValid = FALSE;
return TRUE;

void PathMgr_ScorePath( HPPATH hpPath )
{
    float fScore, fLength, fTotalFitness;
    BOOL bIsValid;
    int i, nFromIndex, nToIndex, nTargetIndex;
    FCOORD fxyVector, fxySubVector;

    hpPath->bIsValid = FALSE;
    nTargetIndex = hpPath->nTargetIndex;
    if (nTargetIndex < 0)
    {
        hpPath->bIsValid = FALSE;
        return;
    }
    if (nTargetIndex >= NUMBER_OF_UNITS)
    {
        hpPath->bIsValid = FALSE;
        return;
    }

    hpPath->nodeTargetNode = nodeTarget;

    /* Check for uninitialized node list and path */
    if (nodeTarget != hpPath->hpnodeNodeList[nTargetIndex]) return;

    nFromIndex = 0;
    fxyVector.x = FLOAT_ZERO;
    fxyVector.y = FLOAT_ZERO;
    fTotalFitness = ((float)(nTargetIndex+1));
```c
for( i = 0; i <= nTargetIndex; i++ )
{
    nToIndex = ((int)hpPath->hpnodeNodeList[i]);
    bIsValid = PathMgr_GetLinkLength( nFromIndex, nToIndex,
        &fLength, &fxySubVector );
    if( bIsValid == FALSE )
    {
        hpPath->bIsValid = FALSE;
        return;
    }
    fxyVector.x += fxySubVector.x;
    fxyVector.y += fxySubVector.y;
    fScore = fLength*SCORE_SCALE;
    fTotalFitness += fScore;
    nFromIndex = nToIndex;
}

hpPath->bIsValid = TRUE;
hpPath->fxyPathVector.x = fxyVector.x;
hpPath->fxyPathVector.y = fxyVector.y;
hpPath->fFitness = fTotalFitness;

BOOL PathMgr_GetLinkLength( int nFromIndex, int nToIndex,
float *fLength, FCOORD *fxyVector )
{
    double dLength;
    BOOL bValid;
    bValid = GetMeasurement( nFromIndex, nToIndex, fxyVector );
    if( bValid == FALSE ) return FALSE;
    dLength = _hypot( (double)fxyVector->x), (double)fxyVector->y );
    *fLength = (float)dLength);
    return TRUE;
}

void PathMgr_SortPathList( void )
/*
    Optimized Insertion Sort   */
{
    int i, j, k;
    HPATH hpTempPath;
    float fTempScore, fTestScore;
    BOOL bTempValid, bIsBetter, bTestValid;
    for( i = l; i < 2*POPULATION_SIZE; i++ )
    {
        hpTempPath = ahpPaths[i];
        bTempValid = hpTempPath->bIsValid;
        if( bTempValid == FALSE ) continue;
        fTempScore = hpTempPath->fFitness;
        for( j = 0; j < i; j++ )
        { 
            bIsBetter = FALSE;
            bTestValid = ahpPaths[j]->bIsValid;
            if( bTestValid == TRUE )
            {
                fTestScore = ahpPaths[j]->fFitness;
                if( fTestScore > fTempScore ) bIsBetter = TRUE;
            }
            else
            { 
                bIsBetter = TRUE;
            }
            if( bIsBetter == FALSE ) continue;
            for( k = i; k > j; k-- )
            { 
                ahpPaths[k] = ahpPaths[k-1];
            }
        }
}


```c
void
ahpPaths[j] = hpTempPath;
break;
}
}

void PathMgr_ScorePopulation( int nIndex )
{
    int i, nNumPaths;
    float fTotalFitness, fFitness;
    LPPOPUL ppopPop;
    HPPATH hpPath;
    BOOL bIsValid;

    nNumPaths = 0;
    fTotalFitness = FLOAT_ZERO;
    GetPopulationPointerFromIndex( nIndex, &ppopPop );
    for( i = 0; i < POPULATION_SIZE; i++ )
    {
        hpPath = ppopPop->ahppathPaths[i];
        bIsValid = hpPath->bIsValid;
        if( bIsValid == TRUE )
        {
            nNumPaths++;
            fFitness = hpPath->fFitness;
            fTotalFitness += fFitness;
        }
        else
        {
            break;
        }
    }
    ppopPop->nNumValidPaths = nNumPaths;
    ppopPop->fTotalFitness = fTotalFitness;
    UpdatePopulationStateFromIndex( nIndex, nNumPaths );
}

int PathMgr_GetSwapSize( void )
{
    int nSize;
    nSize = GetUniformInteger( 8 );
    return nSize;
}
```

### pathmgr.h

```c
#ifndef PATHMGR_H
#define PATHMGR_H

#define REPRODUCTION_RANDOMIZE 3001
#define REPRODUCTION_CROSSOVER 3002
#define REPRODUCTION_LINEAR_INVERSION 3003
#define REPRODUCTION_CIRCULAR_INVERSION 3004
#define REPRODUCTION_REDUCTION 3005
#define REPRODUCTION_CROSSMATCH_OUTSIDE 3006
#define REPRODUCTION_CROSSMATCH_INSIDE 3007
#define REPRODUCTION_COMBINATORIAL 3008
#define REPRODUCTION_DIRECT 3009

void ReproducePathPopulation( int nIndex );

#endif
```
#define STRICT
#define NOCOMM
#include <windows.h>
#include <Windowsx.h>
#include "plotbmp.h"
#define BACKGROUND_COLOR BLACK_BRUSH

static HBITMAP hForegroundBitmap;
static HBITMAP hBackgroundBitmap;
static HWND hParentWnd;
static int nSizeX, nSizeY;

void CreatePlottingBitmaps(HWND hwnd)
{
    HDC hDC, hMemDC;
    nSizeX = GetSystemMetrics(SM_CXFULLSCREEN);
    nSizeY = GetSystemMetrics(SM_CYFULLSCREEN);
    hParentWnd = hwnd;
    hDC = GetDC(hParentWnd);
    hMemDC = CreateCompatibleDC(hDC);
    SelectObject(hMemDC, GetStockObject(BACKGROUND_COLOR));
    hForegroundBitmap = CreateCompatibleBitmap(hDC, nSizeX, nSizeY);
    SelectObject(hMemDC, hForegroundBitmap);
    Rectangle(hMemDC, 0, 0, nSizeX, nSizeY);
    hBackgroundBitmap = CreateCompatibleBitmap(hDC, nSizeX, nSizeY);
    SelectObject(hMemDC, hBackgroundBitmap);
    Rectangle(hMemDC, 0, 0, nSizeX, nSizeY);
    DeleteDC(hMemDC);
    ReleaseDC(hParentWnd, hDC);
}

void DestroyPlottingBitmaps(void)
{
    DeleteObject(hForegroundBitmap);
    DeleteObject(hBackgroundBitmap);
}

void SwapPlottingBitmaps(void)
{
    HBITMAP hTemp;
    HDC hDC, hMemDC;
    hTemp = hForegroundBitmap;
    hForegroundBitmap = hBackgroundBitmap;
    hBackgroundBitmap = hTemp;
    hDC = GetDC(hParentWnd);
    hMemDC = CreateCompatibleDC(hDC);
    SelectObject(hMemDC, GetStockObject(BACKGROUND_COLOR));
    SelectObject(hMemDC, hBackgroundBitmap);
    Rectangle(hMemDC, 0, 0, nSizeX, nSizeY);
    DeleteDC(hMemDC);
    ReleaseDC(hParentWnd, hDC);
}
HBITMAP GetForegroundBitmapHandle( void )
{
    return hForegroundBitmap;
}

HBITMAP GetBackgroundBitmapHandle( void )
{
    return hBackgroundBitmap;
}

plotbmap.h

#ifndef PLOTBMP_H
#define PLOTBMP_H

void CreatePlottingBitrnaps( HWND hwnd);
void DestroyPlottingBitrnaps( void );
void SwapPlottingBitrnaps( void );
HBITMAP GetForegroundBitrnapHandle( void );
HBITMAP GetBackgroundBitrnapHandle( void );

#endif

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include "bmpmgr.h"
#include "child.h"
#include "populmgr.h"
#include "numunits.h"
#include "msmtdb.h"
#include "populmgr.h"
#include "genotype.h"
#include "math.h"

#define EXCELLENT_ERROR ((float)0.5)
#define FAIR_ERROR ((float)2.0)

typedef struct tagUNIT
{
    HWND hwndWindowHandle;
    HBITMAP hBitmap;
    int nEstimateState;
    int nPopulationState;
    BOOL bSensorsActive;
    BOOL bDisplayEstimates;
    BOOL bDisplayObservations;
    BOOL bDisplayMeasurements;
    int nDisplayPathsState;
    UNIT;
} UNIT,

typedef *PUNIT;
typedef *NPUNIT;
typedef *LPUNIT;

define PLOTBMP_H

define PLOTBMP_H

populmgr.c

#include <windows.h>
#include <windowsx.h>
#include "bmpmgr.h"
#include "child.h"
#include "populmgr.h"
#include "numunits.h"
#include "msmtdb.h"
#include "populmgr.h"
#include "genotype.h"
#include "math.h"

#define EXCELLENT_ERROR ((float)0.5)
#define FAIR_ERROR ((float)2.0)

typedef struct tagUNIT
{
    HWND hwndWindowHandle;
    HBITMAP hBitmap;
    int nEstimateState;
    int nPopulationState;
    BOOL bSensorsActive;
    BOOL bDisplayEstimates;
    BOOL bDisplayObservations;
    BOOL bDisplayMeasurements;
    int nDisplayPathsState;
    UNIT;
} UNIT,

typedef *PUNIT;
typedef *NPUNIT;
typedef *LPUNIT;

210
static UNIT aunitUnits[NUMBER_OF_UNITS];
static HWND ahwndUnits[NUMBER_OF_UNITS];
static int an_indices[NUMBER_OF_UNITS];

BOOL PopulMgr_IsEven( int nTest );
int PopulMgr_GetIndexFromHandle( HWND hwnd );
void PopulMgr_PrepareHandleIndexList( void );
int PopulMgr_SearchSubPopulationForHandle( HWND hwnd, int nLowIndex, int nTopIndex );
BOOL PopulMgr.ValidateIndex( int nIndex );
void PopulMgr_ForceChildRepaint( int nIndex );

void CreatePopulation( HWND hwndParent, HINSTANCE hInstance, LPSTR szUnitClsName )
{
    int i, j, nIndex;
    int nDX, nDY, nMaxX, nMaxY, nFourthX, nFourthY, nHalfX, nOffsetX;
    BOOL bIsEven;
    HBITMAP hBitmap;
    POINT xyUnit;
    HWND hwndUnit;

    nMaxX = GetSystemMetrics( SM_CXFULLSCREEN );
    nMaxY = GetSystemMetrics( SM_CYFULLSCREEN );

    nDX = nMaxX/UNIT_COLUMNS;
    nDY = nMaxY/UNIT_ROWS;

    nHalfX = nDX/2;
    nFourthX = nDX/4;
    nFourthY = nDY/4;

    nIndex = 0;
    for( i = 0; i < UNIT_ROWS; i++ )
    {
        bIsEven = PopulMgr_IsEven( i );
        nOffsetX = 0;
        if( bIsEven == FALSE ) nOffsetX = nHalfX;
        for( j = 0; j < UNIT_COLUMNS; j++ )
        {
            xyUnit.x = nFourthX + nOffsetX + j*nDX;
            xyUnit.y = nFourthY + i*nDY;
            hwndUnit = CreateUnit(hwndParent,hInstance,szUnitClsName,&xyUnit);
            aunitUnits[nIndex].hwndWindowHandle = hwndUnit;
            aunitUnits[nIndex].nEstimateState = ESTIMATE_STATE_NONE;
            aunitUnits[nIndex].nPopulationState = POPULATION_STATE_NONE;
            aunitUnits[nIndex].bSensorsActive = TRUE;
            UpdateChildBitmapHandleFromIndex( nIndex );
            aunitUnits[nIndex].bDisplayObservations = TRUE;
            aunitUnits[nIndex].bDisplayMeasurements = TRUE;
            aunitUnits[nIndex].nDisplayPathsState = SHOW_NO_PATHS;

        }
    }

    /* Set Source Bitmap */
    hBitmap = GetSourceBitmapHandle();
    aunitUnits[0].hBitmap = hBitmap;
    PopulMgr_PrepareHandleIndexList();
}

void SetIndividualPathDisplayToNoneFromHandle( HWND hwnd )
{
    int nIndex;

    /* Set Source Bitmap */
    hBitmap = GetSourceBitmapHandle();
    aunitUnits[0].hBitmap = hBitmap;
    PopulMgr_PrepareHandleIndexList();
}
void nindex = PopulMgr_GetIndexFromHandle( hwnd );
if( nIndex <= 0 ) return;

aunitUnits[nIndex].nDisplayPathsState = SHOW_NO_PATHS;
}

void SetIndividualPathDisplayTo1FromHandle( HWND hwnd )
{
    int nIndex;
    nindex = PopulMgr_GetIndexFromHandle( hwnd );
    if( nIndex <= 0 ) return;

    aunitUnits[nIndex].nDisplayPathsState = SHOW_1_PATH;
}

void SetIndividualPathDisplayTo5FromHandle( HWND hwnd )
{
    int nIndex;
    nindex = PopulMgr_GetIndexFromHandle( hwnd );
    if( nIndex <= 0 ) return;

    aunitUnits[nIndex].nDisplayPathsState = SHOW_5_PATHS;
}

void SetIndividualPathDisplayTo15FromHandle( HWND hwnd )
{
    int nIndex;
    nindex = PopulMgr_GetIndexFromHandle( hwnd );
    if( nIndex <= 0 ) return;

    aunitUnits[nIndex].nDisplayPathsState = SHOW_15_PATHS;
}

void SetIndividualPathDisplayToAllFromHandle( HWND hwnd )
{
    int nIndex;
    nindex = PopulMgr_GetIndexFromHandle( hwnd );
    if( nIndex <= 0 ) return;

    aunitUnits[nIndex].nDisplayPathsState = SHOW_ALL_PATHS;
}

int GetIndividualPathDisplayFromHandle( HWND hwnd )
{
    int nIndex, nState;
    nindex = PopulMgr_GetIndexFromHandle( hwnd );
    if( nIndex <= 0 ) return FALSE;

    nState = aunitUnits[nIndex].nDisplayPathsState;
    return nState;
}

BOOL GetIndividualPathDisplayFromIndex( int nIndex, int *nState )
{
    BOOL bValid;

    bValid = PopulMgr_VALIDATEINDEX( nIndex );
    if( bValid == FALSE )
    {
        *nState = SHOW_NO_PATHS;
        return FALSE;
    }

    *nState = aunitUnits[nIndex].nDisplayPathsState;
    return TRUE;
}
void DisableIndividualEstimateDisplayFromHandle(HWND hwnd)
{
    int nIndex;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return;
    aunitUnits[nIndex].bDisplayEstimates = FALSE;
}

void EnableIndividualEstimateDisplayFromHandle(HWND hwnd)
{
    int nIndex;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return;
    aunitUnits[nIndex].bDisplayEstimates = TRUE;
}

void ToggleIndividualEstimateDisplayFromHandle(HWND hwnd)
{
    int nIndex;
    BOOL bState;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return;

    bState = aunitUnits[nIndex].bDisplayEstimates;
    if (bState == TRUE)
    {
        aunitUnits[nIndex].bDisplayEstimates = FALSE;
    }
    else
    {
        aunitUnits[nIndex].bDisplayEstimates = TRUE;
    }
}

BOOL GetIndividualEstimateDisplayStateFromHandle(HWND hwnd)
{
    int nIndex;
    BOOL bState;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return FALSE;

    bState = aunitUnits[nIndex].bDisplayEstimates;
    return bState;
}

BOOL GetIndividualEstimateDisplayStateFromIndex(int nIndex, BOOL *bState)
{
    BOOL bValid;

    bValid = PopulMgr_VerifyIndex(nIndex);
    if (bValid == FALSE)
    {
        *bState = FALSE;
        return FALSE;
    }

    *bState = aunitUnits[nIndex].bDisplayEstimates;
    return TRUE;
}

void DisableIndividualObservationDisplayFromHandle(HWND hwnd)
{
    int nIndex;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return;

    aunitUnits[nIndex].bDisplayObservations = FALSE;
}
void EnableIndividualObservationDisplayFromHandle(HWND hwnd) {
    int nIndex;
    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return;
    aunitUnits[nIndex].bDisplayObservations = TRUE;
}

void ToggleIndividualObservationDisplayFromHandle(HWND hwnd) {
    int nIndex;
    BOOL bState;
    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return;
    bState = aunitUnits[nIndex].bDisplayObservations;
    if (bState == TRUE) {
        aunitUnits[nIndex].bDisplayObservations = FALSE;
    } else {
        aunitUnits[nIndex].bDisplayObservations = TRUE;
    }
}

BOOL GetIndividualObservationDisplayStateFromHandle(HWND hwnd) {
    int nIndex;
    BOOL bState;
    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex <= 0) return FALSE;
    bState = aunitUnits[nIndex].bDisplayObservations;
    return bState;
}

BOOL GetIndividualObservationDisplayStateFromIndex(int nIndex, BOOL *bState) {
    BOOL bValid;
    bValid = PopulMgr_validateIndex(nIndex);
    if (bValid == FALSE) {
        *bState = FALSE;
        return FALSE;
    }
    *bState = aunitUnits[nIndex].bDisplayObservations;
    return TRUE;
}

void DisableIndividualMeasurementDisplayFromHandle(HWND hwnd) {
    int nIndex;
    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex < 0) return;
    aunitUnits[nIndex].bDisplayMeasurements = FALSE;
}

void EnableIndividualMeasurementDisplayFromHandle(HWND hwnd) {
    int nIndex;
    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if (nIndex < 0) return;
    aunitUnits[nIndex].bDisplayMeasurements = TRUE;
}
void ToggleIndividualMeasurementDisplayFromHandle(HWND hwnd)
{
    int nIndex;
    BOOL bState;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if( nIndex < 0 ) return;

    bState = aunitUnits[nIndex].bDisplayMeasurements;
    if( bState == TRUE )
    {
        aunitUnits[nIndex].bDisplayMeasurements = FALSE;
    }
    else
    {
        aunitUnits[nIndex].bDisplayMeasurements = TRUE;
    }
}

BOOL GetIndividualMeasurementDisplayStateFromHandle(HWND hwnd)
{
    int nIndex;
    BOOL bState;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if( nIndex < 0 ) return FALSE;
    bState = aunitUnits[nIndex].bDisplayMeasurements;
    return bState;
}

BOOL GetIndividualMeasurementDisplayStateFromIndex(int nIndex, BOOL *bState)
{
    BOOL bValid;
    bValid = PopulMgr_UpdateIndex(nIndex);
    if( bValid == FALSE )
    {
        *bState = FALSE;
        return FALSE;
    }
    *bState = aunitUnits[nIndex].bDisplayMeasurements;
    return TRUE;
}

void ToggleSensorStateFromHandle(HWND hwnd)
{
    int nIndex;
    BOOL bSensorsActive;

    nIndex = PopulMgr_GetIndexFromHandle(hwnd);
    if( nIndex <= 0 ) return;

    bSensorsActive = aunitUnits[nIndex].bSensorsActive;
    if( bSensorsActive == TRUE )
    {
        aunitUnits[nIndex].bSensorsActive = FALSE;
    }
    else
    {
        aunitUnits[nIndex].bSensorsActive = TRUE;
    }
    UpdateChildBitmapHandleFromIndex(nIndex);
}

BOOL GetSensorStateFromIndex(int nIndex, BOOL *bState)
{
    BOOL bValid;
    bValid = PopulMgr_UpdateIndex(nIndex);
    if( bValid == FALSE )
    {
        *bState = FALSE;
        return FALSE;
    }
}
void UpdateChildBitmapHandleFromIndex( int nIndex )
{
    int nEstimateState, nPopulationState;
    BOOL bSensorsActive;
    HBITMAP hBitmap;
    nEstimateState = aUnitUnits[nIndex].nEstimateState;
    nPopulationState = aUnitUnits[nIndex].nPopulationState;
    bSensorsActive = aUnitUnits[nIndex].bSensorsActive;
    hBitmap = GetTargetBitmapHandle(bSensorsActive, nEstimateState, nPopulationState);
    aUnitUnits[nIndex].hBitmap = hBitmap;
    PopulMgr_ForceChildRepaint( nIndex );
}

void PopulMgr_ForceChildRepaint( int nIndex )
{
    HWND hChild;
    RECT rClient;
    hChild = aUnitUnits[nIndex].hwndWindowHandle;
    GetClientRect( hChild, &rClient );
    InvalidateRect( hChild, &rClient, FALSE );
    UpdateWindow( hChild );
}

HBITMAP GetChildBitmapHandleFromHandle( HWND hwnd )
{
    int nIndex;
    HBITMAP hBitmap;
    nIndex = PopulMgr_GetIndexFromHandle( hwnd );
    if( nIndex < 0 ) return NULL;
    hBitmap = aUnitUnits[nIndex].hBitmap;
}

void PopulMgr_PrepareHandleIndexList( void )
/* Optimized Insertion Sort */
{
    HWND hNew;
    ahwndUnits[0] = aUnitUnits[0].hwndWindowHandle;
    anIndices[0] = 0;
    for( i = 1; i < NUMBER_OF_UNITS; i++ )
    {
        hNew = aUnitUnits[i].hwndWindowHandle;
        ahwndUnits[i] = hNew;
        anIndices[i] = i;
        for( j = 0; j < i; j++ )
        {
            if( ahwndUnits[j] < hNew ) continue;
            for( k = i; k > j; k-- )
            {
                ahwndUnits[k] = ahwndUnits[k-1];
                anIndices[k] = anIndices[k-1];
            }
        }
    }
ahwndUnits[j] = hNew;
anIndices[j] = i;
break;
}
}

int GetChildIndexFromHandle(HWND hwnd)
{
    return PopulMgr_GetIndexFromHandle(hwnd);
}

HWND GetChildHandleFromIndex(int nIndex)
{
    if (nIndex >= NUMBER_OF_UNITS) return NULL;
    if (nIndex < 0) return NULL;
    return ahwndUnits[nIndex];
}

int PopulMgr_GetIndexFromHandle(HWND hwnd)
{
    if (hwnd == ahwndUnits[0]) return anIndices[0];
    if (hwnd == ahwndUnits[NUMBER_OF_UNITS - 1]) return anIndices[NUMBER_OF_UNITS - 1];
    return PopulMgr_SearchSubPopulationForHandle(hwnd, 0, NUMBER_OF_UNITS - 1);
}

int PopulMgr_SearchSubPopulationForHandle(HWND hwnd, int nLowIndex, int nTopIndex)
int nMidIndex, nDifference;
HWND hTemp;

nDifference = nTopIndex - nLowIndex;
if (nDifference == 1) return -1;

nMidIndex = nLowIndex + nDifference / 2;
hTemp = ahwndUnits[nMidIndex];
if (hwnd == hTemp) return anIndices[nMidIndex];
if (hTemp > hwnd)
    return PopulMgr_SearchSubPopulationForHandle(hwnd, nLowIndex, nMidIndex);
else
    return PopulMgr_SearchSubPopulationForHandle(hwnd, nMidIndex, nTopIndex);

void UpdatePopulationStateFromIndex(int nIndex, int nNumPaths)
{
    if (nIndex >= NUMBER_OF_UNITS) return;
    if (nIndex < 0) return;
    if (nNumPaths == POPULATION_SIZE)
    {
        aunitUnits[nIndex].nPopulationState = POPULATION_STATE_ALL;
        return;
    }
    if (nNumPaths == 0)
    {
        aunitUnits[nIndex].nPopulationState = POPULATION_STATE_NONE;
        return;
    }
    if (nNumPaths > (POPULATION_SIZE / 2))
    {
        aunitUnits[nIndex].nPopulationState = POPULATION_STATE_MANY;
        return;
    }
    aunitUnits[nIndex].nPopulationState = POPULATION_STATE_FEW;
}
void UpdateEstimateStateToNoneFromIndex( int nIndex )
{
    if( nIndex >= NUMBER_OF_UNITS ) return;
    if( nIndex < 0 ) return;
    aunitUnits[nIndex].nEstimateState = ESTIMATE_STATE_NONE;
}

void UpdateEstimateStateFromIndex( int nIndex, FCOORD *fxyEstimate )
{
    FCOORD fxyTrueSource, fxyTrueTarget;
    float fXError, fYError, fRadiusError;
    BOOL bIsValid;

    if( nIndex >= NUMBER_OF_UNITS ) return;
    if( nIndex < 0 ) return;
    bIsValid = GetTrueCoordinates( 0, &fxyTrueSource );
    if( bIsValid == FALSE ) return;
    bIsValid = GetTrueCoordinates( nIndex, &fxyTrueTarget );
    if( bIsValid == FALSE ) return;

    fXError = fxyTrueSource.x + fxyEstimate->x - fxyTrueTarget.x;
    fYError = fxyTrueSource.y + fxyEstimate->y - fxyTrueTarget.y;
    fRadiusError = (_hypot( (double)fXError), (double)fYError));

    if( fRadiusError < EXCELLENT_ERROR )
    {
        aunitUnits[nIndex].nEstimateState = ESTIMATE_STATE_EXCELLENT;
        return;
    }

    if( fRadiusError < FAIR_ERROR )
    {
        aunitUnits[nIndex].nEstimateState = ESTIMATE_STATE_FAIR;
        return;
    }

    aunitUnits[nIndex].nEstimateState = ESTIMATE_STATE_POOR;
}

BOOL PopulMgr_IsEven( int nTest )
{
    int nHalf, nFull;

    nHalf = nTest/2;
    nFull = 2*nHalf;
    if( nFull == nTest ) return TRUE;
    return FALSE;
}

BOOL PopulMgr_ValidateIndex( int nIndex )
{
    if( nIndex < 0 ) return FALSE;
    if( nIndex >= NUMBER_OF_UNITS ) return FALSE;
    return TRUE;
}

BOOL ResetAllAgentDisplays( void )
{
    int nIndex;

    for( nIndex = 1; nIndex < NUMBER_OF_UNITS; nIndex++ )
    {
        aunitUnits[nIndex].nEstimateState = ESTIMATE_STATE_NONE;
        aunitUnits[nIndex].nPopulationState = POPULATION_STATE_NONE;
        UpdateChildBitmapHandleFromIndex( nIndex );
    }
    return TRUE;
}
#ifndef POPULMGR_H
#define POPULMGR_H

#include "fcoord.h"

#define SHOW_NO_PATHS 1000
#define SHOW_1_PATH 1001
#define SHOW_5_PATHS 1002
#define SHOW_15_PATHS 1003
#define SHOW_ALL_PATHS 1004

HWND GetChildHandleFromIndex( int nIndex );
BOOL ResetAllAgentDisplays( void );
void CreatePopulation( HWND hwndParent, HINSTANCE hInstance, LPSTR szUnitClsName );
void UpdateChildBitmapHandleFromIndex( int nIndex );
void ToggleSensorStateFromHandle( HWND hwnd );
BOOL GetSensorStateFromIndex( int nIndex, BOOL *bState );
HBITMAP GetChildBitmapHandleFromHandle( HWND hwnd );
void DisableIndividualEstimateDisplayFromHandle( HWND hwnd );
void EnableIndividualEstimateDisplayFromHandle( HWND hwnd );
void ToggleIndividualEstimateDisplayFromHandle( HWND hwnd );
BOOL GetIndividualEstimateDisplayStateFromHandle( HWND hwnd );
BOOL GetIndividualEstimateDisplayStateFromIndex( int nIndex, BOOL *bState );
void DisableIndividualObservationDisplayFromHandle( HWND hwnd );
void EnableIndividualObservationDisplayFromHandle( HWND hwnd );
void ToggleIndividualObservationDisplayFromHandle( HWND hwnd );
BOOL GetIndividualObservationDisplayStateFromHandle( HWND hwnd );
BOOL GetIndividualObservationDisplayStateFromIndex( int nIndex, BOOL *bState );
void DisableIndividualMeasurementDisplayFromHandle( HWND hwnd );
void EnableIndividualMeasurementDisplayFromHandle( HWND hwnd );
void ToggleIndividualMeasurementDisplayFromHandle( HWND hwnd );
BOOL GetIndividualMeasurementDisplayStateFromHandle( HWND hwnd );
BOOL GetIndividualMeasurementDisplayStateFromIndex( int nIndex, BOOL *bState );
void SetIndividualPathDisplayToNoneFromHandle( HWND hwnd );
void SetIndividualPathDisplayTo1FromHandle( HWND hwnd );
void SetIndividualPathDisplayTo5FromHandle( HWND hwnd );
void SetIndividualPathDisplayTo15FromHandle( HWND hwnd );
int GetIndividualPathDisplayFromHandle( HWND hwnd );
int GetIndividualPathDisplayFromIndex( int nIndex, int nState );
int GetChildIndexFromHandle( HWND hwnd );
void UpdatePopulationStateFromIndex( int nIndex, int nNumPaths );
void UpdateEstimateStateToNoneFromIndex( int nIndex );
void UpdateEstimateStateFromIndex( int nIndex, FCOORD *fxyEstimate );
#endif

random.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include <stdlib.h>
#include <math.h>
#include "random.h"
#include "time.h"

#define A_RAND
#define N_RAND

RAND_MAX

4

219
#define S_RAND_SEED 3.141592654
#define PI

static double dGaussAdd, dGaussFac;

void InitializeRandomNumbers( void )
{
    int stime;
    long ltime;

    dGaussAdd = sqrt( 3.0 * (double) (N_RAND) );
    dGaussFac = 2.0 * dGaussAdd / ((double) (N_RAND) * (double) (A_RAND));
    ltime = time(NULL);
    stime = (unsigned) ltime / 2;
    srand( S_RAND_SEED );
}

void GetStandardGaussian( double *dRand )
{
    double dSum, i, nRand;

    dSum = 0.0;
    for( i = 0; i < N_RAND; i++ )
    {
        nRand = rand();
        dSum += (double) nRand;
    }
    *dRand = dGaussFac * dSum - dGaussAdd;
}

void GetUnitUniform( double *dRand )
{
    int nRand;

    nRand = rand();
    *dRand = ((double) nRand) / ((double) RAND_MAX);
}

void GetUnitCauchy( double *dRand )
{
    double dUniform, dArg;

    GetUnitUniform( &dUniform );
    dArg = (PI) * (dUniform - 0.5);
    *dRand = tan( dArg );
}

BOOL TestBinaryProbability( float fProbLevel )
{
    double dRandom;

    if( fProbLevel <= ((float) 0.0) ) return FALSE;
    if( fProbLevel >= ((float) 1.0) ) return TRUE;
    GetUnitUniform( &dRandom );
    if( ((float) dRandom) > fProbLevel ) return FALSE;
    return TRUE;
}

int GetUniformInteger( int nMax )
{
    int nRand;
    double dRand, dMax;

    GetUnitUniform( &dRand );
    dMax = ((double) nMax);
    nRand = ((int) (dMax * dRand));
    if( nRand > nMax ) nRand = nMax;
    return nRand;
}
random.h

#ifndef RANDOM_H
#define RANDOM_H

void InitializeRandomNumbers( void);
void GetStandardGaussian( double *dRand);
void GetUnitUniform( double *dRand);
void GetUnitCauchy( double *dRand);
BOOL TestBinaryProbability( float fProbLevel);
int GetUniformInteger( int nMax );

#endif

resource.h

#define IDM GENESMENU 10
#define IDM_ABOUT 11
#define IDM_EXIT 17
#define IDM_PAUSE 18
#define IDM_LOBOTOMY 20

#define IDM_OPTIONS_DISPLAY 80
#define IDM_OPTIONS_ERROR 81
#define IDM_OPTIONS_OPERATORS 82
#define IDM_OPTIONS_ESTIMATOR 83

#define IDM_EXPMT_OWS1 1800
#define IDM_EXPMT_OWS2 1801
#define IDM_EXPMT_OWS3 1802
#define IDM_EXPMT_OSS2 1810
#define IDM_EXPMT_OSS3 1811
#define IDM_EXPMT_SNPOOR 1821
#define IDM_EXPMT_SNFAIR 1822
#define IDM_EXPMT_SNGOOD 1823
#define IDM_EXPMT_SDPOOR 1830
#define IDM_EXPMT_SDFAIR 1831
#define IDM_EXPMT_SDGODD 1832
#define IDM_EXPMT_CN5 1840
#define IDM_EXPMT_CW20 1841
#define IDM_EXPMT_CW40 1842
#define IDM_EXPMT_CD5 1850
#define IDM_EXPMT_CD20 1851
#define IDM_EXPMT_CDO40 1852
#define IDM_EXPMT_EST1 1861
#define IDM_EXPMT_EST2 1862
#define IDM_EXPMT_EST3 1863
#define IDM_EXPMT_EST4 1864
#define IDM_EXPMT_MOTION 1870
#define IDS_CLASSNAME 16
#define IDS_WINDOWTITLE 17

timeslc.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include "timeslc.h"


```c
#include "timestamp.h"
#include "msmtmk.h"
#include "numunits.h"
#include "plotbmp.h"
#include "geneplot.h"
#include "pathmgr.h"
#include "estimate.h"
#include "random.h"
#include "expmt.h"

#define SLICE_INITIAL 2000
#define SLICE_REPRODUCTION 2001
#define SLICE_OBSERVATION 2002
#define SLICE_ESTIMATE 2003
#define SLICE_PLOT1 2004
#define SLICE_PLOT2 2005
#define SLICE_SOURCE 2006
#define SLICE_MEASUREMENT 2007
#define SLICE_COLLECT_DATA 2008

#define LIST_EMPTY -1

static int nCurrentSlice = SLICE_INITIAL;
static int nCurrentIndex = 0;
static int nIndices[NUMBERS_OF_UNITS];
static int nCounter = 0;

void TimeSlc_CollectData( HWND hwnd );
void TimeSlc_UpdateSliceIndex( void);
void TimeSlc_ExecuteLocationEstimates( HWND hwnd, int nIndex);
void TimeSlc_ExecuteMeasurements( HWND hwnd, int nIndex);
void TimeSlc_ExecuteReproduction( HWND hwnd, int nIndex);
void TimeSlc_ExecuteObservationEstimates( HWND hwnd, int nIndex);
void TimeSlc_SwapBitmaps( void);
void TimeSlc_RandomizeIndices( void);

void ProcessTimeSlice( HWND hwnd )
{
    switch( nCurrentSlice )
    {
        case SLICE_INITIAL:
            break;

        case SLICE_MEASUREMENT:
            IncrementTimeStamp();
            TimeSlc_ExecuteMeasurements( hwnd, nCurrentIndex );
            break;

        case SLICE_REPRODUCTION:
            TimeSlc_ExecuteReproduction( hwnd, nCurrentIndex );
            break;

        case SLICE_OBSERVATION:
            TimeSlc_ExecuteObservationEstimates( hwnd, nCurrentIndex );
            break;

        case SLICE_ESTIMATE:
            TimeSlc_ExecuteLocationEstimates( hwnd, nCurrentIndex );
            break;

        case SLICE_PLOT1:
            TimeSlc_SwapBitmaps();
            break;
    }
}
```
case SLICE_PLOT2:
    TimeSlc_PlotSituation( hwnd );
    break;

case SLICE_SOURCE:
    TimeSlc_ExecuteMeasurements( hwnd, 0 );
    break;

case SLICE_COLLECT_DATA:
    TimeSlc_CollectData( hwnd );
    break;

default:
    return;
}

TimeSlc_UpdateSliceIndex();

void TimeSlc_CollectData( HWND hwnd )
{
    CollectData( hwnd );
}

void ResetTimeSlice( HWND hwnd )
{
    nCurrentSlice = SLICE_INITIAL;
    TimeSlc_RandomizeIndices();
    TimeSlc_SwapBitmaps();
    TimeSlc_SwapBitmaps();
    TimeSlc_PlotSituation( hwnd );
}

void TimeSlc_ExecuteMeasurements( HWND hwnd, int nIndex )
{
    ProcessMeasurementSlice( nIndex );
    PlotMeasurements( hwnd, nIndex );
}

void TimeSlc_ExecuteReproduction( HWND hwnd, int nIndex )
{
    ReproducePathPopulation( nIndex );
    PlotPaths( hwnd, nIndex );
}

void TimeSlc_ExecuteObservationEstimates( HWND hwnd, int nIndex )
{
    ProcessObservationSlice( nIndex );
    PlotObservations( hwnd, nIndex );
}

void TimeSlc_ExecuteLocationEstimates( HWND hwnd, int nIndex )
{
    ProcessEstimateSlice( nIndex );
    PlotEstimates( hwnd, nIndex );
}

void TimeSlc_PlotSituation( HWND hwnd )
{
    InvalidateRect( hwnd, NULL, FALSE );
    UpdateWindow( hwnd );
}

void TimeSlc_SwapBitmaps( void )
{
    SwapPlottingBitmaps();
}
void TimeSlc_UpdateSliceIndex( void )
{
    switch( nCurrentSlice )
    {
        case SLICE_INITIAL:
            nCurrentSlice = SLICE_SOURCE;
            nCounter = 0;
            nCurrentIndex = nIndices[nCounter];
            break;

        case SLICE_MEASUREMENT:
            nCurrentSlice = SLICE_REPRODUCTION;
            break;

        case SLICE_REPRODUCTION:
            nCurrentSlice = SLICE_OBSERVATION;
            break;

        case SLICE_OBSERVATION:
            nCurrentSlice = SLICE_ESTIMATE;
            break;

        case SLICE_ESTIMATE:
            nCurrentSlice = SLICE_MEASUREMENT;
            nCounter++;
            nCurrentIndex = nIndices[nCounter];
            if( nCounter == NUMBER_OF_UNITS )
            {
                nCurrentSlice = SLICE_PLOT1;
                nCounter = 0;
                nCurrentIndex = nIndices[nCounter];
            }
            break;

        case SLICE_PLOT1:
            nCurrentSlice = SLICE_PLOT2;
            nCounter = 0;
            nCurrentIndex = nIndices[nCounter];
            break;

        case SLICE_PLOT2:
            nCurrentSlice = SLICE_COLLECT_DATA;
            nCounter = 0;
            nCurrentIndex = nIndices[nCounter];
            break;

        case SLICE_COLLECT_DATA:
            nCurrentSlice = SLICE_SOURCE;
            nCounter = 0;
            nCurrentIndex = nIndices[nCounter];
            break;

        case SLICE_SOURCE:
            nCurrentSlice = SLICE_MEASUREMENT;
            nCounter = 1;
            nCurrentIndex = nIndices[nCounter];
            break;

        default:
            return;
    }
}

void TimeSlc_RandomizeIndices( void )
{
    int i, nNewIndex;
BOOL bDirection;

/* Set up Lists for scrambling */
for( i = 1; i < NUMBER_OF_UNITS; i++ )
{ 
nIndices[i] = LIST_EMPTY;
}

/* Scramble lists */
for( i = 1; i < NUMBER_OF_UNITS; i++ )
{ 
nNewIndex = GetUniformInteger( NUMBER_OF_UNITS-1 );

bDirection = TestBinaryProbability( ((float)0.5) );
while( nIndices[nNewIndex] != LIST_EMPTY )
{ if( bDirection == TRUE ) nNewIndex++;
else nNewIndex--;
if( nNewIndex < 1 ) nNewIndex = NUMBER_OF_UNITS-1;
if( nNewIndex == NUMBER_OF_UNITS ) nNewIndex = 1;
}
nIndices[nNewIndex] = i;
}

# ifndef TIMESLC_H
# define TIMESLC_H
void ProcessTimeSlice( HWND hwnd );
void ResetTimeSlice( HWND hwnd );
#endif

#include <windows.h>
#include <windowsx.h>
#include "timestmp.h"
#define DEFAULT_VALID_OFFSET 80

static TMSTMP tmstmpCurrentTimeStamp = 0;
static TMSTMP tmstmpValidOffset = DEFAULT_VALID_OFFSET;

void SetTimeStampValidOffset( TMSTMP tmstmpOffset )
{ 
tmstmpValidOffset = tmstmpOffset;
}

void SetDefaultTimeStampValidOffset( void )
{ 
tmstmpValidOffset = DEFAULT_VALID_OFFSET;
}

void IncrementTimeStamp( void )
{ 
tmstmpCurrentTimeStamp++;
}

TMSTMP GetCurrentTimeStamp( void )
{ 
return tmstmpCurrentTimeStamp;
}
BOOL ValidateTime Stam p(TM ST MP tmstmp Test Stamp )
{
    TMST MP tmstmp Delta Time;
    tmstmp Delta Time = tmstmp Current Time Stamp - tmstmp Test Stamp;
    if( tmstmp Delta Time > tmstmp Valid Offset ) return FALSE;
    else return TRUE;
}

timestmp.h

#ifndef TIMESTMP_H
#define TIMESTMP_H

typedef unsigned int TMST MP;

void SetTimeStampValidOffset( TMST MP tmstmp Offset);
void SetDefaultTimeStampValidOffset( void);
void IncrementTimeStamp( void);
TMST MP GetCurrentTimeStamp( void);
BOOL ValidateTime Stam p(TM ST MP tmstmp Test Stamp );

#endif

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include "vector.h"

vector.c

BOOL CopyVector( LPFCOORD fxyTarget, LPFCOORD fxySource )
{
    fxyTarget->x = fxySource->x;
    fxyTarget->y = fxySource->y;
    return TRUE;
}

BOOL ScaleVector( LPFCOORD fxyVector, float fScale )
{
    float fScaleX, fScaleY;
    fScaleX = fxyVector->x * fScale;
    fScaleY = fxyVector->y * fScale;
    fxyVector->x = fScaleX;
    fxyVector->y = fScaleY;
    return TRUE;
}

BOOL SumVectors( LPFCOORD fxySum, LPFCOORD fxyArg1, LPFCOORD fxyArg2 )
{
    float fSumX, fSumY;
    fSumX = fxyArg1->x + fxyArg2->x;
    fSumY = fxyArg1->y + fxyArg2->y;
    fxySum->x = fSumX;
    fxySum->y = fSumY;
    return TRUE;
}

BOOL LoadNullVector( LPFCOORD fxyNull )
{
fxyNull->x = ((float)(0.0));
fxyNull->y = ((float)(0.0));
return TRUE;
}

vector.h

#ifndef VECTOR_H
#define VECTOR_H
#include "fcoord.h"
BOOL LoadNullVector( LPFCOORD fxyNull );
BOOL SumVectors( LPFCOORD fxySum, LPFCORD fxyArg1, LPFCORD fxyArg2 );
BOOL ScaleVector( LPFCORD fxyVector, float fScale );
BOOL CopyVector( LPFCORD fxyTarget, LPFCORD fxySource );
#endif

winmain.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <Windowsx.h>
#include "genes.h"
#include "instance.h"
#pragma warning (disable:4100)
#define BACKGROUND_COLOR BLACK_BRUSH

static HANDLE ghInstance;

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow )
{
    MSG msg;
    HWND hwnd;
    char szClassName[50];
    char szWindowTitle[50];
    WNDCLASS wc;

    ghInstance = hInstance; /* If no prev. instance */
    if ( !hPrevInstance )
    {
        if ( !LoadString( hInstance, IDS_CLASSNAME, &szClassName[0], sizeof szClassName ) )
            return FALSE;
        wc.style = 0;
        wc.lpfnWndProc = (WNDPROC) MainWndProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;
        wc.hinstance = hInstance;
        wc.hIcon = LoadIcon( hInstance, "GENE_ICON" );
        wc.hCursor = LoadCursor( NULL, IDC_ARROW );
        wc.hbrBackground = GetStockObject( BACKGROUND_COLOR );
        wc.lpszMenuName = MAKEINTRESOURCE(IDM_GENESMENU);
        wc.lpszClassName = &szClassName[0];

        if ( !RegisterClass(&wc) ) return FALSE;
    }

    return TRUE;
}
if( !LoadString( hInstance, IDS_CLASSNAME, &szClassName[0], sizeof szClassName ) )
    return FALSE;
if( !LoadString( hInstance, IDS_WINDOWTITLE, &szWindowTitle[0], sizeof szWindowTitle ) )
    return FALSE;

hwnd = CreateWindow( &szClassName[0],
    &szWindowTitle[0],
    WS_GENES,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL );

if( !hwnd ) return FALSE;
ShowWindow(hwnd, nCmdShow);
for(; ; )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
            {
                if( msg.message == WM_QUIT )
                    break;
                else
                    {
                        TranslateMessage( &msg );
                        DispatchMessage( &msg );
                    }
            }
        else
            {
                SendMessage( hwnd, WMU_TIME_SLICE, 0, 0L );
            }
    return msg.wParam;
}
HANDLE GetParentInstanceHandle( void )
    { return ghInstance; }

wndproc.c

#define STRICT
#define NOCOMM
#include <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include "genes.h"
#include "timeslc.h"
#include "bmpmgr.h"
#include "instance.h"
#include "paintmgr.h"
#include "child.h"
#include "populmgr.h"
#include "genemenu.h"
#include "random.h"
#include "msmdb.h"
#include "msmntmr.h"
#include "numunits.h"
#include "plotbmp.h"
#include "pathdb.h"
#include "glblopts.h"

static char szUnitClsName[] = "UnitChild";

void WndProc_Temp_TestMeasurementDatabase(HWND hwnd) {
    int i, j;
    FCOORD fxyTest;
    float fi, fj;

    for( i = 0; i < NUMBER_OF_UNITS; i++ )
        fi = (float) i;
        for( j = 0; j < NUMBER_OF_UNITS; j++ )
            fj = (float) j;
            fxyTest.x = fi;
            fxyTest.y = fj;
            StoreMeasurement( i, j, &fxyTest );

    for( i = 0; i < NUMBER_OF_UNITS; i++ )
        fi = (float) i;
        for( j = 0; j < NUMBER_OF_UNITS; j++ )
            fj = (float) j;
            fxyTest.x = fi;
            fxyTest.y = fj;
            StoreMeasurement( i, j, &fxyTest );

    WndProc_Temp_TestMeasurementDatabase(HWND hwnd);
{ 
    fj = (float) j;
    GetMeasurement( i, j, &fxyTest );
    if( fj != fxyTest.x )
    {
        MessageBox( hwnd, "Memory Error",
                   "Error in Measurement Database", MB_OK );
        return;
    }
    if( fj != fxyTest.y )
    {
        MessageBox( hwnd, "Memory Error",
                   "Error in Measurement Database", MB_OK );
        return;
    }
}

BOOL WndProc_Initialize( HWND hwnd )
{
    HANDLE hInstance;
    hInstance = GetParentInstanceHandle();
    CreatePlottingBitmaps( hwnd );
    OpenBitmaps( hInstance );
    RegisterUnitChildClass( hInstance, (LPSTR) szUnitClsName );
    CreatePopulation( hwnd, hInstance, (LPSTR) szUnitClsName );
    InitializeRandomNumbers();
    InitializeMeasurementDatabase( hwnd );
    InitializeMeasurementMakerFilters();
    InitializePathDatabase();
    UpdateGeneMenu( hwnd );
    SendMessage ( hwnd, WM_COMMAND, IDM_ABOUT, 0L );
    UpdateWindow( hwnd );
    ResetTimeSlice( hwnd );

    /* WndProc_Temp_TestMeasurementDatabase( hwnd ); */
    return TRUE;
}

BOOL WndProc_Destroy( HWND hwnd )
{
    CloseBitmaps();
    DestroyPlottingBitmaps();
    DestroyMeasurementDatabase( hwnd );
    DestroyPathDatabase();
    return TRUE;
}
Bibliography


The vita has been removed from the scanned document