1996

Using Genetic Algorithms to Solve the Geometric Traveling Salesperson Problem

Michael Criswell

College of Saint Benedict/Saint John's University

Follow this and additional works at: https://digitalcommons.csbsju.edu/honors_theses

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.csbsju.edu/honors_theses/745

Available by permission of the author. Reproduction or retransmission of this material in any form is prohibited without expressed written permission of the author.
Using Genetic Algorithms to Solve the Geometric Traveling Salesperson Problem

A THESIS
The Honors Program
College of St. Benedict/St. John's University

In Partial Fulfillment
of the Requirements for "Departmental Distinction"
and the Degree Bachelor of Arts
in the Department of Computer Science

Advisor
Dr. Michael Gass

by
Michael Andrew Criswell
May 9, 1996
PROJECT TITLE: Using Genetic Algorithms to Solve the Traveling Salesperson Problem

Approved by:
Dr. Mike Gass
Chair, Department of Mathematics
Dr. Lynn Ziegler
Chair, Department of Computer Science
Dr. J. Andrew Holey
Assistant Professor of Computer Science
Margaret Cook
Director, Honors Thesis Program
Anthony Cunningham
Director, Honors Program
Introduction

There is a class of problems in Computer Science that do not respond to classical methods of solution. They are referred to as NP problems. One such problem, the Traveling Salesperson Problem, has been shown to be in the class of NP problems. Various methods have been devised to attack such problems and one such method is called the Genetic Algorithm. Genetic Algorithms simulate Darwinian natural selection and their unorthodox methods have been used to solve such difficult problems like the Traveling Salesperson Problem. The difficulty with using Genetic Algorithms is that there are many variables that affect its performance and, as in real life, it is difficult to know how each one should be set so as to use Genetic Algorithms in the most effective way. We will explore a particular Genetic Algorithm to gain insight about the effects of specific variables on its ability to solve the Traveling Salesperson Problem.

Traveling Salesperson Problem

The Traveling Salesperson Problem, or TSP, is a relatively simple problem for humans to solve for just a few of cities. It consists of a practical problem that sales professions typically have dealt with at one point or other: What “tour” does a person use to reach \( x \) number of cities in the shortest distance traveled while only visiting each city once? For just a few cities, humans can simply pick up a map and use their index finger to trace a path in mere seconds. Computers as powerful as a desktop PC using what is termed an exhaustive search method [4] can also find a best tour for just a few cities. The difficulty that arises is, as the number of cities that need to be visited grows large, the best tour becomes incredibly difficult to determine. The human
cannot trace an adequate number of possible optimal paths in a reasonable time to ensure the best path has been found. The desktop computer has a speed advantage when it comes to the sheer number of paths it can evaluate per second. However, it would take the computer a lifetime to find all the possible tours for a large number of cities. As humans, we have a distinct advantage over computers with our “common sense” that enables us not even to consider paths that go obviously too far out of the way or make no sense. While the human could use his or her “common sense” to make an educated guess, the tour through the cities can’t be guaranteed to be the best without actually going through all the similar possibilities with a hand calculator. This may find a path that the human never even noticed. Hence the reason that computers are employed to find the “best” path by checking every possibility. However, with twenty or more cities, the problem cannot be solved in a reasonable amount of time by even the most powerful computers. The reason for such poor performance in even the fastest supercomputers is that an exhaustive search has been used in attempting to solve the TSP.

**Exhaustive Search Method**

What an exhaustive search [4] does is start at the beginning city and create a list of all cities that can be reached directly from it. It then systematically takes each of these paths and finds all the possible cities it could go to next. If we are traveling between x cities, then starting at city #1 we find there is a maximum of x-1 paths from it. Therefore x(x-1) or x²-x paths must be considered at this stage. As the process continues, we find that (x-1)! paths must be computed. Obviously, as the number of cities (x) increases the total number of paths that must be considered increases well beyond the computing capacity of either a human being or computer. Several ways to deal with this problem have been developed such as “Breadth First
Search," "Shortest Path," and "Adjacency Representation" [4] but none have been able to solve
the problem in a reasonable amount of time as the number of cities increases. Today, finding the
best solution for a tour through 100 cities is computationally impossible.

**Nondeterministic Methods**

The problem with the above technique appears to lie in the linear method itself. Several
nondeterministic methods have been employed to solve the TSP within a reasonable amount of
time. Among these new methods are Genetic Algorithms or "GA's." GA's are a type of
GA's from other AI methods of programming is the fact that GA's do not really know anything
about what they are working with. The algorithms works with strings of computer data in much
the same way a human cell works with DNA. All cells reproduce and copy DNA information in
mass quantities through mitosis. The information contained in the peptide bonds of DNA
includes the entire blueprint for a human body. This is an incredible amount of information that
is being constantly updated, copied, and changed. New cells created through the process of
growth live or die depending on rules of evolution or heredity. If characteristics of a cell lead to
a healthier existence for that cell, it will more than likely survive to do what cells do best: create
more cells. Some cells with detrimental characteristics will survive by chance and pass on their
information right alongside the healthy ones. Most of the time, however, survival of the fittest is
the rule.

**Genetic Algorithms**

A man by the name of John Holland brought the characteristics of Darwinian natural
selection into computer science [11]. The idea was to use a binary string in the place of a peptide
chain in DNA to represent an individual. Then the functions of the cell that affect DNA coding were reproduced in the computer: crossover and mutation. Crossover is an event that sometimes occurs in human cells when they divide. A diploid chromosome (a chromosome with two copies of DNA) will sometimes break apart during cell division with pieces from either or both copies of DNA information. Simply considered, a diploid chromosome will separate into two single strands of DNA but each strand will contain pieces from copy A and pieces from copy B. The other cell function that affects coding is mutation. Mutation, like crossover, happens infrequently. When mutation occurs, one or several “genes” or codes are randomly changed. This is caused by random protons floating through the universe or radiation from man-made sources. These protons knock a protein off the strand of DNA which is fixed by the cell. Mutation is simply stated as an unforeseen and random change in the code of a chromosome. With the application of crossover, mutation, and selection we have a codable algorithm that optimizes in much the same way as genetics work. Hence the name “Genetic Algorithms.”

**Fundamentals**

In order to follow the same processes as evolution and heredity, GA’s go through a multiple-step process: Generate or initialize a population. Evaluate every member of that population. Select a group of individuals from the population based upon their evaluation of fitness. Alter the selected population using operations such as crossover and mutation. Go back to step 2 and continue [3,9].

**Population**

A GA will create a “population” of strings of data or chromosomes in the initialization phase. It is in this phase that random codes or predefined codes are produced in large
numbers[7,8,12]. This hopefully results in an immense number of individuals each with a unique code.

**Evaluation**

The second phase, evaluation, gives each “chromosome” a rating of fitness. [1] This fitness value is calculated by a heuristic equation that is unique to the problem to be solved (see TSP GA below).

**Selection**

Once each chromosome has a fitness value, a number of individuals are selected from the population based on fitness. GA’s use the concept of survival of the fittest when dealing with individuals in a population. Many unique ways exist of selecting individuals from a population to aid the development of even better individuals. The method chosen by this researcher is the “Roulette Wheel” selection heuristic[3,9]. Basically, the fitness for each chromosome to be selected is computed and slots for a certain size of fitness are created in a roulette wheel. A roulette wheel that allows more slots for higher fitness values is spun once for each of the chromosomes in the population. If, in a matter of speaking, the number of a chromosome comes up that chromosome is selected. More than one chromosome can be selected, but with this method the best are copied, the average stay even, and the worst fitness levels are never chosen[18]. Whichever method is chosen for selection, several important things must be accomplished by it. A heuristic must be used to decide which individuals are selected. If selection is done by picking individuals at random, the GA will work no better than an exhaustive search method. The most “fit” individuals should be selected. Without the most “fit” individuals, the best solutions would be eliminated before they had a chance to participate in
creating better individuals. Selection has a very significant effect upon how the GA performs. It must keep the best individuals to provide good "breeding" stock but also keep enough average individuals so that the following generations do not converge and lose diversity.

**Genetic Operators**

The next phase changes or adds to the population using crossover and mutation. The chromosomes chosen through selection are then used to create a new "generation" of chromosomes using crossover and mutation. [11] Two parents are chosen and crossed-over to create a new child or children with "genes" from either parent in the hope that the child will have some of the positive characteristics of both parents. The intensity or number of individuals selected for crossover is set at a predetermined rate. Similarly, individuals are selected for mutation at a predetermined rate. Mutation, in GA’s, mimics natural phenomena that occur in most living things. Occasionally an individual may be created with a noticeably different genetic makeup. If that newly created difference allows the individual a better chance to stay alive, this new characteristic could be passed on to any offspring. A rabbit may develop a new color of fur to match its surroundings. A cockroach may develop that is genetically immune to a certain pesticide. GA’s attempt to imitate this occurrence by randomly changing a small piece of an individual. All of the mutated or crossed-over individuals are then given a fitness value and joined with the remaining population.

**Parallelism**

This cycle then repeats over and over for a specified number of generations etc. The resulting "best" chromosome represents the optimized or most likely solution depending on the application. GA’s are extremely easy to use in parallel systems. Parallel systems solve problems
by splitting it up into several pieces and working on each piece simultaneously until the solution is reassembled. The reason GA's are so easy to use in parallel systems is their nonlinear construction. A population can be broken into several smaller sub-populations without any loss of data[3]. Furthermore, immigrants can migrate between sub-populations at different rates. For example, a parallel system can have four sub-populations cycling simultaneously but only three of those sub-populations can have migration. Two of those sub-populations can migrate 2% of their individuals while the third migrates 25% of its individuals. The fourth sub-population can exist in much the same way as the Galapagos Islands, it neither receives nor sends individuals to any of the other sub-populations.

**Applying Genetic Algorithms to the TSP**

While the application to simulation of evolution and heredity is an appealing use for GA's [10], their real potential lies in applying them to nondeterministic problems. The aforementioned Traveling Salesperson Problem is an interesting and useful application of Genetic Algorithms. The connection between a chromosome of data and the tour of a salesperson might seem like comparing apples with oranges. After all, how are strings of 1's and 0's used to find the best tour through a number of cities? This representation issue is easily resolved by using the strings of 1's and 0's to represent strings of cities[18,7]. Part of the string can be used to specify a number in binary. Using a two bit representation we can get a four-city tour in the following way: 00 can represent city 0, 01 can represent city 1, 10 can represent city 2, and 11 can represent city 3. Giving characteristics such as a name, position, etc. to each of these cities can be quite easy. Now we can use the order of the cities in the string to give us the order of cities to visit in a tour. For example, the string of {11001001} represents {3, 0, 2, 1} which
means that the path we take through the cities is to visit #3 first, #0 second, #2 third, and #1 last. We can now vary the bit-length of the string to allow for a greater number of cities in the tour. A string where 32 bits determine a city can be used for a tour between approximately four billion cities. There will quite probably never be the need to visit this many cities in a single tour.

One of several problems that may appear at this point is how to apply crossover and mutation to a tour of cities. Some of the methods include partially mapped crossover (PMX), matrix crossover (MX), edge recombination (ER), order crossover (OX), and cycle crossover (CX). [18,13,12]. This area is where most of my research has been applied. I have experimented with several different methods to determine which works best under different circumstances (see below). Basically, to keep crossover from scrambling the binary numbers in the strings, crossover points are selected between each city number. For example, if 4-bit numbers are represented in the binary string, crossover points can be selected in 4-bit increments. The 4-bit tour of “00011001001101110 . . .” may have two crossover points selected in these two places:”00011001001101110 . . .” so that the two cities between these crossover points (1100 and 1001) are crossed over. The matter of maintaining the integrity of the crossed over chromosomes so that city numbers aren’t repeated or lost is handled specifically by the crossover method used. (See below) Another problem that appears is how to apply mutation. Normally, mutation simply flips a random bit from “0” to “1". This could, again, compromise the integrity of the tour. One of several solutions to this is a simple swapping of cities. First, two different cities in the tour are picked at random. The positions of these two chosen cities within the tour are swapped. This way, the change is kept small and duplicates or corrupt city numbers are not created. Another problem that could be seen is how to compute the fitness of each tour. The
method used in my research is to create a two-dimensional graph of all the cities and compute the distances traveled between each city in the tour and assign the total mileage to the individual as fitness. The GA will then select the smallest fitness numbers for optimization. Now instead of trying to find every possible path in a tour of thousands of cities using an exhaustive search, we can have hundreds of different paths through those cities constantly being optimized by the GA. What is even better is that we can have several different sub-populations of tours optimizing simultaneously if we use parallel GA's.

**DGENESIS**

The programming package used here is Distributed GENESIS developed by Erick Cantu-Paz [3] and John J. Grefenstette[9]. This package contains several small modules that work every aspect of the GA. Separate modules achieve crossover and mutation plus modules that control population initialization. The package requires the user to create his or her own evaluation function that will be used to compute fitness for every chromosome in the population. DGENESIS can also run in parallel on several machines simultaneously with user-specified levels of migration and population sizes. The number of processes that can be run on each computer is also user-defined. After running an experiment with certain crossover and mutation rates, reports are created with the best individuals listed, mean, the variance, time of the run, and other relevant data. This package has been optimized to run on the Silicon Graphics computers located on the campus-wide UNIX network. The mutation, crossover, initialization, and evaluation modules have all been rewritten to work the Traveling Salesperson Problem.

DGENESIS provides a number of variables that affect the operation of a GA. Several variables were selected to test and obtain results that would suggest a beneficial or detrimental
effect on the performance of the GA: Crossover Rate, Crossover Method, Mutation Rate, Processes, Population Size, and City Topology. All other variables were kept the same: Processor Topology, Generation Gap, Scaling Window, Minimum Rank, and Initial Population. By testing each of these variables independently, the ideal setup and clues to the reasons why each setup succeeds or fails will emerge.

The challenge began with rewriting DGENESIS so that it will work with the TSP and maintain tour integrity in the individuals. DGENESIS was written using the classical methods of handling binary strings in crossover, mutation, and evaluation. The program supplied by Cantu-Paz for population initialization (Appendix A: initpop.c) needed extensive rewriting in order to successfully read in an initial population from a text file to start each run of the GA with the same population. The initialization program was rewritten to perform the following steps:
1) Beginning process opens the init.foo file.
2) Read in number of individuals specified in the in.foo file for the beginning process.
3) Open a socket for another process and send the necessary number of individuals from the init.foo file across the socket.
4) Repeat step 3 until all the processes receive their specified number of individuals from the init.foo file.
5) Close the init.foo file and fill the remaining processes with randomly generated individuals if necessary. (Appendix B, initpop.c)

Crossover Routines

Once this task was completed, a crossover program needed to be written so that individuals that were crossed over did not compromise order or create multiple city visits. Since
this was an experimental variable, several different crossover programs were written using three methods described by Michalewicz [18]: Partially Mapped Crossover or PMX, Order Crossover or OX, and Cycle Crossover or CX.

PMX is a method that uses mappings created by the crossed-over pieces of the individuals to maintain integrity in the non-crossed-over pieces. When two individuals are selected for crossover, two points in those individuals are selected at random. The method of selecting points between binary representations of cities has been discussed (see above). The two parents use the correlating positions in the crossed-over piece to set a series of mappings. With p1 = (1,2,3,4,5,6,7,8) and p2 = (4,1,3,6,2,5,8,7) We select two random crossover points at positions 3 and 6: (xx|xxx|xx). By matching the corresponding positions from p1 and p2 we get several mappings: 3<->3, 4<->6, 5<->2, and 6<->5. These mappings are used to swap the cities to prevent duplicates and missing cities. Using Goldberg’s pseudocode[6] for applying mappings, we start at the first crossover point in p1 and use the mapping to swap cities in p1 so that 3 gets swapped with itself. Next we move on to the next city after the first crossover point and swap 4 and 6 in p1. This process continues until we reach the second crossover point. Now we have a properly crossed-over c1 chromosome. Using the original p1 we do the same for p2 so that the two children created are c1 = (14|3625/78) and c2 = (21|3456|87). As can be seen, this method of crossover creates children where the crossed-over pieces are the only pieces with the path from one of the parents. A deficiency of the method is that the non-crossed over parts are essentially scrambled to retain the path integrity.

CX is a method that uses a cycle to retain the positions of certain parts of the parents. The two parents are placed on top of each other, and the mappings between them are used to set a
cycle within those two parents. The positions included in that cycle are kept in place while the
genes outside the cycle are directly crossed over. For the above parents: p1 = (1,2,3,4,5,6,7,8)
and p2 = (4,1,3,6,2,5,8,7) we take the first position in p1: 1. Since 1 maps to 4, we keep 1 and 4
from p1. Since 4 maps to 6, we keep 1,4 and 6. Since 6 maps to 5, we keep 1,4,6, and 5. And so
on, until we eventually get a mapping to a position that has already been kept. So we keep
1,2,4,5,and 6 in the same position for both p1 and p2. The rest of the parents that are not kept are
crossed over with one another to produce the children: c1 = (1,2,3,4,5,6,8,7) and c2 =
(4,1,3,6,2,5,7,8). This method retains the positions of cities from both parents, but it only
exchanges the positions of those individuals not in the cycle. A deficiency of the method is that
the original order of cities is not guaranteed to be retained.

OX crossover appears to have a solution to the deficiencies of both PMX and CX. OX
selects two crossover points and directly crosses over those individuals between them. It also
repositions the non-crossed-over parts to retain their order as well. It achieves this by first
crossing over the two parts between the crossover points. Then, at the gene just after the second
crossover point, it produces a list of the parent’s genes in order. This list is then edited to remove
those cities already in the child from the crossover part. The list is read into the child from the
first position to end, skipping over the crossed over part. For our example above, p1 =
(1,2,4,5,6,7,8) and p2 = (4,1,3,6,2,5,8,7), we get our two children c1 = (x,x,3,6,2,5,1,x,x) and
c2 = (x,x,3,4,5,6,1,x,x). Next we make a list for c1 from the second crossover point of p1 that is:
(7,8,1,2,3,4,5,6). We remove the cities already in the crossed-over part of c1: (7,8,1,4). This list
is fed into c1 to fill in the remaining positions = (7,8,3,6,2,5,1,4). This same procedure is
applied to the second child to produce c2 = (8,7,3,4,5,6,1,2). This method not only preserves
the exact order of pieces from one of the parents, but it also preserves part of the order of the second parent so that better copies of the two parents are passed on to the new individuals.

**Mutation Routine**

Once these three crossover methods were programmed (Appendix B: ecxross.c, pmxcross.c, oxcross.c), a mutation method needed to be developed. The simple mutation that consisted of choosing two genes at random and swapping them was used. This was very simple to code and became the new mutation program (Appendix B: mutate.c).

**Evaluation Routine**

An evaluation function was created that could follow the path representation of cities and give a fitness value equal to the total distance traveled. The Pythagorean formula was used to compute the straight-line distances between each city in the tour. The cities' coordinates were hard coded in the evaluation function itself in an array. By using the path representation discussed earlier, the distance between each city was computed and totaled. The final program passed the total back as the evaluation value for the individual passed in. (Appendix B: tsp.c)

**Initial Population Creation Routine**

Another program was created to aid in creation of the initial population in init.foo. (Appendix B: initmaker.c) This program created an array of individuals given information such as number of individuals, experiment name, and number of cities. This program then swapped two random genes in each of the new individuals one-hundred times to create a sufficiently random initial population. This population was then stored in a file: init.foo. The extension, ".foo" is the name of the experiment in keeping with the DGENESIS convention.

Several other areas of DGENESIS also needed work to function correctly (Appendix B:
migrate.c and evaluate.c).

### Running DGENESIS

DGENESIS requires a very specific setup to work properly. Once compiled and placed in the proper directory, a file must be created named, “GAhosts” to specify specific computers that will be used in the parallel operations and how many processes each can run simultaneously. All of the computers on the CSB/SJU UNIX network were listed in this file with the more powerful computers (e.g., Indy’s and r4000's) at the top of the list. The Sun workstations were not listed due to incompatibility problems. At this point the evaluation file (Appendix B: tsp.c) was placed in the proper directory and compiled into the system. An experiment was created using the “setup” program provided with DGENESIS that creates three files that contain all the variables used in the experiment. Information stored in the in.foo file contains the crossover and mutation rates along with the total population and the number of evaluations to execute before stopping the system. The proc.foo file contains population sizes for each process running the experiment. The template.foo file contains chromosome specific information. Each gene in the chromosome is given a maximum and minimum value and granularity or number of values between. The “links” program provided with DGENESIS is used next to create a topological mapping for the number of processors used. This program makes a file named, “link.foo” that lists, for each process, the process number, the process being communicated to, the number of individuals sent during a migration, and the number of generations between migrations. The “links” program provides hypercube, ring, complete, and mesh topologies. The user can bypass the “links” program and create his or her own link.foo file with a different topology than ones provided.
Once all these files have been created, the system can be started. The program "rgo" is used to start the system given two parameters: the name of the evaluation function and the name of the experiment variables. For example, with the above files created we would type, "rgo tsp foo". The system will send out a call to all computers listed in the "GAhosts" file and use the computers that answer back first. The system then reads in the initial population stored in the file, "init.foo". The beginning process will read in enough individuals to fill its own population and then send individuals to the participating computers to fill their populations. (Appendix B: initpop.c) Once all the computers have the necessary data, DGENESIS begins to run the GA. (Appendix A: generate.c) For the first generation, it evaluates all the individuals in the population. The individuals are stored in a multidimensional array. The population contains a number of chromosomes equivalent to the size of the population. Each chromosome has a binary representation, an evaluation value, and a flag to cause a reevaluation due to either crossover, mutation or migration. The binary representation is packed or unpacked to allow either a binary or numeric string representation. (Appendix A: convert.c) For every generation, fit individuals are selected from the old population (see roulette wheel above), mutation and crossover are applied, migration is handled if necessary, and the new individuals are evaluated. DGENESIS collects data on the best individuals from every process, measures the system performance with a mean, variance, and average value and stores information about convergence of the GA. The program then checks to see if it should stop by calling the "done()" function which gives a true or false answer. Done() checks to see if the total number of evaluations has been run, if no new individuals have been created, or if better individuals are not created. If any of these criteria are met, Done() will stop the GA.
As DGENESIS stops its genetic cycle, it stops all the computers used to run the GA and then creates a report of the entire run (rep.foo). Many other files are created through the run of DGENESIS. Each process creates two files xmin.foo and xout.foo. Both files are used to keep track of all the individuals created in each process, their evaluation value, and when they were created. Much of the data stored in these two files is used to create the final report (rep.foo). Another file, reg.foo, contains the names and process numbers of all computers used in the experiment.

Experimentation and Results

Experiments were run using a unit-square topology of cities. The optimal tour was computed and the GA was allowed to run until the optimal tour was found. This method of testing performance may not be accurate when applied to real-life situations. (See below: City Topology) The same initial population was used for each experiment which was created at random by using the initmaker.c program. (Appendix B: initmaker.c) Each experimental variable was tested six times and then averaged.

Effect of Crossover Rate

The first variable to be tested between crossover methods was the crossover rate. For each crossover method, the crossover rate was varied between 10% and 100% each at 10% increments. PMX produced a very strange curve between crossover rates of 40% and 70%. These two rates are quite good, but between them the number of generations necessary to find the solution jumps up to almost the same number required for 100%. The reason for this is quite difficult to understand. It concerns PMX itself since the results of the other two methods show nothing similar. It is unrelated to parallelizing or any other variables. Apparently, as long as it is
working with two thirds or one third of the population, it will function properly. However, approaching half the population is detrimental to the number of necessary generations. OX yields a much better average of generations for the solution than PMX. The graph OX creates when varying the crossover rate is similar to maximum population graphs for certain areas of the wilderness. This graph shows a large number of generations necessary that drops down to its minimum at 30% and then levels off at a plateau halfway between the minimum and maximum. CX has a similar graph, although slightly more generations are needed than OX. CX is also better than PMX. CX traces a graph similar to that of OX where extremely low crossover requires many generations to produce a solution. This then drops down to its minimum at around 60% and then plateaus between the minimum and maximum. Some reasons for the differences in behaviors of these methods can be traced back to how they function. PMX will preserve only the order of the crossed-over pieces and the rest go through a kind of scrambling similar to mutation. CX preserves the positioning of part of both parents so that both of the parents are represented in the offspring rather than just one parent and a jumbled mess. So CX should be better than PMX because of this. OX takes one step further than CX by preserving the order and the positioning from parts of both parents.
Effect of Mutation Rate

To find an efficient mutation rate for DGENESIS, several runs were made at five different mutation rates: 0.1%, 0.5%, 1%, 5%, and 10%. Each of the crossover methods were used to examine how crossover affected each one individually. Surprisingly, CX worked better in the higher rates than did PMX or OX. The best rate for CX turned out to be 0.5% instead of 0.1% like PMX and OX. This suggests that more mutation is needed in CX due to its methodology. CX, unlike PMX and OX maintains the actual positions of cities from both parents. Not many positions are being switched other than what is copied from either parent.

![Average Generation of First Solution](image)

What a higher mutation rate does for CX, especially the mutation method used here, is move many genes to places that would not normally get placed there with just CX crossover. PMX especially does not need a lot of mutation simply because the mappings are supplying enough random change in the non-crossed-over parts of the children. OX achieves random change by reading back the gene sequence from the second crossover point rather than from the beginning. OX can also maintain the sequence as it does this. A more specific reason would be how a gene changes position. In CX, a gene has to wait until a parent with the same gene in a different place comes along and does not include that gene in the cycle. In PMX a gene can move either within the crossover segment or because a mapping affects it. In OX a gene moves less inside the crossover segment than through the act of rereading the outside parts back into the individual. By listing the genes back in starting from the
second crossover point, a gene frequently travels from one side of the crossover segment to the other. However, since the genes are read back into the individual in order, getting away from neighboring genes is difficult for a single gene. The latter explains better why PMX is in less need of mutation than is OX. The 10% mutation rate causes all three crossover types to work extremely poorly. An explanation for this drop in performance is related to what mutation does for the GA as it nears an optimal solution. As a GA nears an optimal solution, the population contains a large percentage of individuals that are very close to optimal. When one of these individuals mutates, two cities that may be from opposite ends of the tour are swapped. When a GA is just beginning it needs this kind of small but radical movement of cities within the tour. However, when a GA has so many near-optimal individuals, this kind of change is more likely to change it for the worse. With this higher mutation rate, the GA will find a medium level of progress relatively quickly, but it will take a lot longer to refine that medium solution to the best solution because mutation is destroying the better solutions.

Effect of the Number of Processors

The number of processors seems to have a minor effect upon the performance of the GA. By testing each crossover method in four separate numbers of processors in a hypercube topology, it can be seen that the difference of performance from one to sixteen processes is a good leap. The difference between four and sixteen is not as obvious. While this particular
parallel scheme has its advantages, it would be foolish to assume a substantial increase in performance by continuing to increase the number of processors. The topology would appear to make the biggest difference here. The reason for this could be luck. This particular experiment kept the population size the same for each processor - no matter how many processors were used. If there had been no migration allowed between the processors they would have all performed exactly like the single processor. Since the multiple-processor runs allowed migration at 20 generation intervals, the best individuals from each processor were shared with the others. Only a few individuals were shared to keep from annihilating the “native” population but allow better genetic stock. Since each population was equally sized and affected by the same rates for mutation and crossover, each processor worked autonomously except for the occasional migration. By adding more processors to the GA all that is added is an extra few individuals that get migrated. The only effect adding processors would have on the GA is the added luck of one of those processors applying just the right crossover and mutation to create a solution quickly. This may explain the slight increase in performance rather than an immense increase.

**Effect of Population Size**

Population size is a very important variable for *any* crossover method. The population size has to provide enough variation of individuals while being small enough with which to work. In smaller tours, such as four and eight city tours, a large enough population can be randomly created to contain most of the possible permutations of tours. Four cities only have six possible permutations of tours while eight cities have 5040. A population of 12 could supply a four-city GA with enough individuals for there to be a solution in the first generation. A population of 400 individuals could create a good enough base of individuals for an eight-city
GA to get a solution quickly using only a few generations. With an eight-city tour it is quite possible to create an initial population of six thousand or more to encompass all the possibilities. Since these smaller tours can be evaluated easily enough with an exhaustive search, genetic algorithms are not really needed. Sixteen, thirty-two, and sixty-four city tours begin to approach the borderline for exhaustive search methods.

**Effect of City Topology**

The arrangement of the cities in the tours creates an interesting effect. When the cities are arranged in a grid formation, the GA required fewer generations to find the solution than it did for a more specific arrangement (e.g., in a circle or random scattering of points). This could be the result of the number of equivalent solutions a particular topology allows. Since a grid can have many different tours that are equivalent solutions, the GA will have many different tours approaching the optimal solution. With more structured arrangements, like what exists in real life, the GA has only one or two equivalent solutions to a particular topology. The GA would need more time to change its population towards one or two equivalent solutions than it would ten or twenty different equivalent solutions.

**Conclusion**

In conclusion, the DGENESIS package allows a cornucopia of experimental parameters. Effective values for these parameters are difficult to discover. The information suggested by
these tests give a good clue. OX appears to be the best method of crossover among the three used here. Its maintenance of parental order works quite well to combine two good tours to create a better tour. An extremely low mutation rate would be recommended given the data discovered here. The size of the population is relative to the number of total cities. The population size must be big enough to provide a large search space but small enough to take less time than an exhaustive search. The use of many processors is recommended. However, too many can slow DGENESIS due to the amount of communication overhead from its heavy use of Berkeley sockets. Sixteen could be an adequate number of processors. Much experimentation still needs to be done. One issue that is yet to be resolved is the topology of parallel processing[3]. This could have an interesting effect upon how individuals are shared with different populations that develop diverse individuals. Newer crossover methods can be developed to create better offspring. GA's still have a long way to go to reach the promised land, but perhaps enough maps are being made to get there.
Appendix A:

THE DGENESIS SYSTEM

This appendix includes code created by J.J. Grefenstette and Erick Cantu-Paz. Only code relevant to this paper has been included. The complete original code can be found at the Genetic Algorithm source code library at:

---CONVERT.C---

#include "extern.h"
static char BIT[CHARSIZE] = { '200', '100', '040', '020',
                              '010', '004', '002', '001'};

/* Itoc and Cto1 translate ints to strings and vice versa */
unsigned long int Cto1(instring, length)
    char *instring;    /* string representation */
    int length;       /* length of instring */
{
    register int i;    /* loop control */
    unsigned long n;  /* accumulator for return value */

    n = (unsigned long) 0;
    for (i=0; i<length; i++)
    {
        n <<= 1;
        n += (*instring++ - (int)'0');
    }
    return(n);
}

void Itoc(n, outstring, length)
    unsigned long int n;    /* input int value */
    char *outstring;         /* string representation */
    int length;             /* length of outstring */
{
    register int i;    /* loop control */
    for (i=length-1; i>=0; i--)
    {
outstring[i] = '0' + (n & 1);
    n >>= 1;
}

/* Pack and Unpack translate between strings and (packed) bit arrays */
void Pack(instring, outstring, length)
{
    char *instring;    /* string representation */
    char *outstring;   /* packed representation of instring */
    int length;        /* length of instring */

    static firstflag = 1;
    static full;       /* number of fully used bytes in outstring */
    static slop;       /* number of bits used in outstring's last byte */
    register i,j;      /* loop control */

    if (firstflag)
    {
        full = length / CHARSIZE;
        slop = length % CHARSIZE;
        firstflag = 0;
    }

    for (i=0; i<full; i++, outstring++)
    {
        *outstring = '0';
        for (j=0; j < CHARSIZE; j++)
        {
            if (*instring++ == '1') *outstring |= BIT[j];
        }
    }

    if (slop)
    {
        *outstring = '0';
        for (j=0; j < slop; j++)
        {
            if (*instring++ == '1') *outstring |= BIT[j];
        }
    }
}

void Unpack(instring, outstring, length)
{
    char *instring;    /* packed bit representation */
    char *outstring;   /* string representation of instring */
    int length;        /* length of outstring */

    static firstflag = 1;
    static full;       /* number of fully used bytes in instring */
    static slop;       /* number of bits used in instring's last byte */
    register i,j;      /* loop control */

    if (firstflag)
    {
        full = length / CHARSIZE;
        slop = length % CHARSIZE;
        firstflag = 0;
    }

    for (i=0; i<full; i++, instring++)
    {

24
Appendix A

for (j=0; j < CHARSIZE; j++)
    if (*instring & BIT[j])
        *outstring++ = '1';
    else
        *outstring++ = '0';

if (slop)
{
    for (j=0; j < slop; j++)
        if (*instring & BIT[j])
            *outstring++ = '1';
        else
            *outstring++ = '0';
}

*outstring = '\0';

/* Translations between fixed point ints and reflected Gray code */
void Gray(instring, outstring, length)
char *instring; /* string representing fixed point int */
char *outstring; /* string representing Gray coded value */
register int length; /* length of strings */
{
    register int i;
    register char last;
    last = '0';
    for (i=0; i<length; i++)
    {
        outstring[i] = '0' + (instring[i] != last);
        last = instring[i];
    }
}

void Degray(instring, outstring, length)
char *instring; /* string representing Gray coded int */
char *outstring; /* string representing fixed point int */
register int length; /* length of strings */
{
    register int i;
    register int last;
    last = 0;
    for (i=0; i<length; i++)
    {
        if (instring[i] == '1')
            outstring[i] = '0' + (!last);
        else
            outstring[i] = '0' + last;
        last = outstring[i] - '0';
    }
}

/* Translations between string representation and floating point vectors */
Appendix A

void FloatRep(instring, vect, length)
  char instring[]; /* string representation */
  double vect[]; /* floating point representation */
  int length; /* length of vect (output array) */
{
  register int i; /* loop control */
  unsigned long int n; /* decoded int value */
  register int pos; /* position to start decoding */
  char tmpstring[80]; /* used for gray code interpretation */
  pos = 0;
  for (i=0; i < length; i++)
  {
    if (Grayflag)
    {
      Degray(&instring[pos], tmpstring, Gene[i].bitlength);
      n = Ctoi(tmpstring, Gene[i].bitlength);
    }
    else
    {
      n = Ctoi(&instring[pos], Gene[i].bitlength);
    }
    vect[i] = Gene[i].min + n*Gene[i].incr;
    pos += Gene[i].bitlength;
  }
}

void StringRep(vect, outstring, length)
  double *vect; /* floating point representation */
  char *outstring; /* string representation */
  int length; /* length of vect */
{
  register int i; /* loop control */
  unsigned long int n; /* index of vect[i] within legal range */
  register int pos; /* next position for filling outstring */
  char tmpstring[80]; /* used for gray code translation */
  pos = 0;
  for (i=0; i < length; i++)
  {
    /* convert floating value to an index */
    n = (int) ((vect[i] - Gene[i].min) / Gene[i].incr + 0.5);
    /* encode n in char string */
    if (Grayflag)
    {
      /* convert to Gray code */
      Itoc(n, tmpstring, Gene[i].bitlength);
      Gray(tmpstring, &outstring[pos], Gene[i].bitlength);
    }
    else
    {
      Itoc(n, &outstring[pos], Gene[i].bitlength);
    }
  }
}
Appendix A

pos += Gene[i].bitlength;
}
outstring[pos] = '0';
}

----------------------------------------------- EVAL.C -----------------------------------------------

double eval(str, length, vect, genes)
char str[];    /* string representation */
int length;   /* length of bit string */
double vect[]; /* floating point representation */
int genes;    /* number of elements in vect */
{
    register int i;
    double sum;
    sum = 0.0;
    for (i = 0; i < genes; i++)
        sum += vect[i] * vect[i];
    return (sum);
}

----------------------------------------------- GENERATE.C -----------------------------------------------

/*
* GENESIS Copyright (c) 1986, 1990 by John J. Grefenstette
* This program may be freely copied for educational
* and research purposes. All other rights reserved.
* file: generate.c
* purpose: One generation consists of
* (1) forming a new population of structures.
* (2) evaluating the population.
* (3) gathering performance statistics.
* modified: 7 feb 86
* 2 dec 86: call Measure() before Done() so that
* we can quit upon convergence. Measure() nows
* calls Converge().
* 1 jul 93: add communication point. (ECP)
*/
/*
* Distributed GENESIS
* 27
Appendix A

* Erick Cantu-Paz
* ecantu@lamport.rhon.itam.mx
*
* Instituto Tecnologico Autonomo de Mexico
* 1993
*
*------------------------------------------------------------------------
*/
#include "extern.h"
#include "timeb.h"
#include <errno.h>
extern int Sockets[];
void Generate()
{
    static int rsflag = 1; /* flag cleared after restart */
    STRUCTURE *temp; /* for swapping population pointers */
    register int i; /* for marking structures */
    register int j; /* loop control */
    static int nNewMigrants = 0; /* migrants received in this gen */
    int migflag; /* Migset already made in this gen */
    if (Traceflag)
        printf("Gen %d\n", Gen);
    Trace("Generate entered");
    /* create a new population */
    if (Restartflag && rsflag)
    {
        /* this is a restart so read checkpoint file */
        Restart();
        rsflag = 0; /* disable local restart flag. */
        Converge();
    }
    else if (Gen == 0) { /* this is a fresh experiment */
        Initialize(); /* form an initial population */
        nNewMigrants = 0;
        if (My_id)
           RecvStart();
        else
            SendStart();
        Spin++;
        if (Processes > 1)
            for (i = 1; i < MAXSOCKETS; i++)
                ClearSocket(Sockets[i]);
    }
    else { /* form a new population from */
        /* the old one via genetic operators */
        Select(nNewMigrants);
        nNewMigrants = 0;
        Mutate();
        Crossover();
        if (Eliteflag)
            Elitist();
        if (Allflag) /* mark structures for evaluation */
            for (i = 0; i < Popsize; i++)
Appendix A

New[j].Needs_evaluation = 1;

ifUNIX

if (Processes > 1) {
    /* communication point */
    migflag = 1;
    for (j = 0; j < Links; j++) {
        /* send migrants if comm point reached */
        if (!Gen % Linktable[j].migration_int == 0) {
            if (migflag)
                MakeMigset();
            migflag = 0;
            SendMigrants();
        }
    }
    /* receive migrants from other processes */
    if (Syncflag)
        nNewMigrants = ReceiveMigrantsSync();
    else
        nNewMigrants = ReceiveMigrantsAsync();
    if (nNewMigrants) {
        /*printf("(\%d) nNewMigrants \%dn", My_id, nNewMigrants);*/
        AppendMigrants(nNewMigrants);
    }
}

/* evaluate the newly formed population */
Evaluate(nNewMigrants);
/* gather performance statistics */
Measure(nNewMigrants);
/* check termination condition for this experiment */
Doneflag = Done();
/* checkpoint if appropriate */
if (Num_dumps && Dump_freq && Gen % Dump_freq == 0) {
    if (Num_dumps > 1) {
        sprintf(Dumpfile, "dump.%d", Curr_dump);
        Curr_dump = (Curr_dump + 1) % Num_dumps;
        Checkpoint(Dumpfile);
    }
    Checkpoint(Ckptfile);
} else {
    if (Doneflag) {
        if (Lastflag)
            Checkpoint(Ckptfile);
        else if (Savesize)
            Printbest();
    }
}
Appendix A

/* swap pointers for next generation */
temp = Old;
Old = New;
New = temp;
/* update generation counter */
Gen++;
Trace("Generate completed");
}

-------------------------------------------------------------------------MAIN.C -------------------------------------------------------------------------

/*@GENESIS Copyright (c) 1986, 1990 by John J. Grefenstette
* This program may be freely copied for educational
* and research purposes. All other rights reserved.
*/
* file: main.c
*
* purpose: main program for genesis.
*
* modified: 28 mar 86
* 15 sep 90 - to use display routines
* 15 aug 93 - added FreeMem to release dynamic memory used. (ECP)
*/
#include "global.h"
extern PROCESS *ProcTable;
extern int Sockets[];
void FreeMem()
{
    register int i;  /* loop control */
    LINKSTRUCT *aux, *auxnext;
    Trace("FreeMem entered");
    if (Vector) free(Vector);
    if (Bitstring) free(Bitstring);
    /* free populations */
    for (i=0; i<Popsize + Migreccv; i++){
        if (Old[i].Gene) free(Old[i].Gene);
        if (New[i].Gene) free(New[i].Gene);
    }
    if (Old) free(Old);
    if (New) free(New);
    /* free migration structures */
    for (i=0; i<Migreccv; i++)
        if (Migrants[i].Gene) free(Migrants[i].Gene);
    if (Migrants) free(Migrants);
    for (i=0; i<Migsend; i++)
        if (Migset[i].Gene) free(Migset[i].Gene);
    if (Migset) free(Migset);
    if (Window) free(Window);
Appendix A

for (i=0; i<SaveSize; i++)
    if (Bestset[i].Gene) free(Bestset[i].Gene);
if (Bestset) free(Bestset);
if (Gene) free(Gene);
/* free process table */
if (My_id == 0 && ProcTable){
    for (i=0; i<Processes; i++){
        aux = ProcTable[i].links;
        while (aux){
            auxnext = aux->next_link;
            if (aux) free(aux);
            aux = auxnext;
        }
    }
    free(ProcTable);
}
/* free link tables */
if (Linktable) free(Linktable);
if (LinkRecTable) free(LinkRecTable);
Trace("FreeMem completed");

main(argc,argv)
int argc;
char *argv[];
{
    FILE *fp;
    long clock;
    long time();
    char *ctime();
    void die();
    charErrMsg[80];
    int i;
/* see input.c for the use of command line args */
Input(argc,argv);
if (Displayflag) {
    initscr();
    signal(SIGINT, die);
    clear();
    refresh();
    if (Interflag)
        Interactive(); /* never returns */
    /* this point is reached only if InterFlag is OFF */
    move(1,0);
    printf("run until Trials = %d", Totaltrials);
    move(1, 35);
    printf("executing: ");
    refresh();
}
    /* one experiment */
}
Appendix A

if (Traceflag)
    printf("Experiment %d\n", Experiment);
do /* see generate.c for main GA loop */
{
    Generate();
}
while (!Doneflag);
if (Traceflag)
    printf("Online %e Offline %e Best %e\n", 
            Online, Offline, Best);
/* accumulate performance measurements */
Totonline += Online;
Totoffline += Offline;
Totbest += Best;
/* get ready for next experiment */
Experiment++;
Gen = 0;

#if UNIX
    if (Processes > 1){
        SendByeLinks();
        if (My_id) SendEnd();
        elseRecvEnd();
    }
#endif

while (Experiment < Totalexperiments); /* compute and print final performance measures */
    Totonline /= Totalexperiments;
    Totoffline /= Totalexperiments;
    Totbest /= Totalexperiments;
if (Onlnflag)
    printf("Online %e\n", Totonline);
if (Offlnflag)
    printf("Offline %e\n", Totoffline);
if (Bestflag)
    printf("Best %e\n", Totbest);
if (Logflag && My_id == 0)
{
    fp = fopen(Logfile, "a");
    fprintf(fp, "Online %e \n", Totonline);
    fprintf(fp, "Offline %e \n", Totoffline);
    fprintf(fp, "Best %e\n", Totbest);
    time(&clock);
    fprintf(fp, "%s\n", ctime(&clock));
    fclose(fp);
}
if (Displayflag) {
    move(23,0);
    die();
}
Appendix A

/* free dynamic memory used */
FreeMem();
if (My_id == 0){
    if ((fp = fopen(Endfile, "w")) == NULL){
        fprintf(ErrMsg, "main: can't create Endfile %s", Endfile);
        IOError(ErrMsg);
    }
    fclose(fp);
}
#endif
for (i=0; i<MAXSOCKETS; i++) close(Sockets[i]);
return 0;
}

------------------------------------------MIGRATE.C ------------------------------------------

/*
 */
/* ------------------------------------------MIGRATE.C ------------------------------------------
 * Distributed GENESIS
 * Erick Cantu-Paz
 * ecantu@lamport.rhon.itam.mx
 * Instituto Tecnologico Autonomo de Mexico
 * 1993
 */
/*
 * file: migrate.c
 * purpose: migration of individuals between processes
 */
#include <sys/time.h>
#include "extern.h"
extern int FirstMigrant, LastMigrant;
extern int Sockets[];
/*
 * copy the best individuals from the population to Migset
 */
void MakeMigset()
{
    SortPopulation(New);
    bcopy((char *) New, (char *) Migset, (int) (sizeof(STRUCTURE) * Migsend));
}
Appendix A

/*
 * Sends a migration header with number of migrants, then sends migrants
 */
void SendMigrants(link)
int link;
{
    register int i, j;  /* loop control */
    int migrants;  /* number of individuals to migrate */
    char buffer[MAXMSG];  /* send / receive buffer */
    Trace("SendMigrants entered");

    if (!Linktable[link].in_use)
        return;
    migrants = (int)(Linktable[link].migration_rate * Popsize);
    /* sends migration header */
    bzero(buffer, MAXMSG);
    sprintf(buffer, "%s %d %d", MIGHEAD, migrants, Gen);
    if (Send(Sockets[0], buffer, strlen(buffer), &Linktable[link].addr) < 0)
        IOnError("SendBest: can't send request");
    /* sends migrants */
    for (i = 0; i < migrants; i++) {
        bzero(buffer, MAXMSG);
        Unpack(Migset[i].Gene, Bitstring, Length);
        sprintf(buffer, "%s %d", Bitstring, Migset[i].Perf);
        if (Send(Sockets[0], buffer, strlen(buffer), &Linktable[link].addr) < 0)
            IOnError("SendBest: can't send migrants");
    }
    Trace("SendMigrants completed");
}

/*
 * Receives migrants from any process ** asynchronously **
 * Return value: number of received migrants.
 */
int ReceiveMigrantsSync()
{
    register int i;  /* loop control */
    fd_set rfds;  /* candidate links */
    int nfds;  /* number of file descriptors (for select) */
    struct timeval tv;  /* timeout for select */
    int status;  /* return value from select, recv */
    int nNewMigrants = 0;  /* migrants received */
    Trace("ReceiveMigrantsSync entered");
    nfds = gettablesize();
    FD_ZERO(&rfds);
    for (i = 1; i < Linkrecv; i++)
        FD_SET(Sockets[i], &rfds);
    tv.tv_sec = 0;
    tv.tv_usec = 0;
    status = select(nfds, &rfds, (fd_set *) 0, (fd_set *) 0, &tv);
    if (status > 0)
**Appendix A**

```c
for (i = 1; i < Linkrecv; i++)
    if (FD_ISSET(Sockets[i], &rfd))
        nNewMigrants += ReceiveIndividuals(i - 1);
Trace("ReceiveMigrantsAsync completed");
return nNewMigrants;
}

/*
* Receives migrants from any process ** SYNONRONOUSLY **
* Return value: number of received migrants.
*/
int ReceiveMigrantsSync()
{
    int nNewMigrants; /* migrants received */
    int status, retry, cont, total, sum, j;
    fd_set migset; /* set of active links */
    struct timeval tv; /* better than polling */
    char flag[MAXSOCKETS]; /* true if something was received from each socket */
    Trace("ReceiveMigrantsSync entered");
    nNewMigrants = 0;
    tv.tv_sec = 0;
    tv.tv_usec = 500000;
    sum = 0;
    retry = 0;
    cont = total = 0;
    for (j = 0; j < MAXSOCKETS; j++)
        flag[j] = 0;
    /* total = number of active links */
    for (j = 0; j < Linkrecv; j++)
        if ((Gen & LinkRecTable[j].migration_int == 0) & LinkRecTable[j].in_use)
            total ++;
    do {
        /* create the set of candidate links */
        FD_ZERO(&migset);
        for (j = 0; j < Linkrecv; j++)
            if ((Gen & LinkRecTable[j].migration_int == 0) &
                !flag[j] & LinkRecTable[j].in_use) {
                FD_SET(Sockets[j + 1], &migset);
            }
        /* wait to receive something */
        status = select(getdtablesizer(), &migset, NULL, NULL, &tv);
        if (status == 0)
            perror("select");
        else {
            /* receive the migrants */
            for (j = 0; j < Linkrecv; j++)
                if (FD_ISSET(Sockets[j + 1], &migset)) {
                    flag[j] = 1;
                    nNewMigrants += ReceiveIndividuals(j);
                }
        }
    }
```
Appendix A

/* if retry == 1, we have to wait for more migrants */
contain 0;
for (j = 0; j < Linkrecv; j++)
    if (flag[j])
        contain++;
    if (contain == total)
        retry = 0;
    else
        retry = 1;
} while (retry);
Trace("ReceiveMigrantsSync completed");
return nNewMigrants;

/*
* Receives individuals from 'active' process into 'Migrants' queue
* Return value : the number of received migrants.
*/
int ReceiveIndividuals(active)
int active; /* active link from LinkRecTable */
{
    register int i, j; /* loop control */
    struct sockaddr_in aux_addr;
    int status; /* return value from select */
    int remgen; /* remote generation */
    int nMigrantsRecv = 0; /* migrants received */
    int nMigrants = 0; /* migrants sent by remote process (from mig header) */
    char buffer[MAXMSG];
    struct timeval tv; /* now unnecessary! */
    Trace("ReceiveIndividuals entered");
    if (active < 0 || active > Linkrecv - 1)
        Error("ReceiveIndividuals: bad link number");
    /* is the link still active? */
    if (!LinkRecTable[active].in_use)
        return 0;
    /* receive a message; header or ENDEXP */
bzero(buffer, MAXMSG);
    status = RecvTO(Sockets[active + 1], buffer, MAXMSG, &aux_addr, NULL);
    if (status < 0) {
        perror("timeout:");
        printf("%d %d)TIMEOUT!! esperando a %s %d
", My_id, FindPorts(),
            inet_ntoa(LinkRecTable[active].addr.sin_addr),
            ntohs(LinkRecTable[active].addr.sin_port));
        return 0;
    }
    /* did I receive the message from the right process? */
    if (strcmp(inet_ntoa(aux_addr.sin_addr), inet_ntoa(LinkRecTable[active].addr.sin_addr)) ||
        aux_addr.sin_port != LinkRecTable[active].addr.sin_port) {
        printf("Link %d wrong %s %d vs. %s %d
", active,
            inet_ntoa(aux_addr.sin_addr), ntohs(aux_addr.sin_port),
            inet_ntoa(LinkRecTable[active].addr.sin_addr), ntohs(LinkRecTable[active].addr.sin_port));
    }
Appendix A

/* did I receive ENDEXP?, deactivate the link if true */
if (strstr(buffer, ENDEXP)) {
    LinkRecTable[active].in_use = 0;
    return 0;
}

/* received a header, get the number of migrants and their gen */
if (strstr(buffer, MIGHEAD)) {
    sscanf(buffer, "%s %d %d", &nMigrants, &remgen);
} else {
    printf("Should received MIGHEAD: \n%\n", buffer);
    return 0;
}

/* migrants from the right generation? */
if (Syncflag && Gen != remgen) {
    printf("Should receive migrants from gen %d, received %d\n", Gen, remgen);
    printf("%\n", buffer);
    return 0;
}

tv.tv_sec = 100;
tv.tv_usec = 0;

/* copy arriving migrants to Migrants array */
for (i = 0; i < nMigrants; i++) {
if (LastMigrant == Migrecv)
    printf("ReceiveIndividuals: migrants buffer full\n");
    bzero(buffer, MAXMSG);
    if (RecvTO(Sockets[active + 1], buffer, MAXMSG, &aux_addr, NULL) < 0)
        continue;
    sscanf(buffer, "%s %d", Bitstring, &Migrants[LastMigrant].Perf);
    Pack(Bitstring, Migrants[LastMigrant].Gene, Length);
    Migrants[LastMigrant].Needs_evaluation = 0;
    nMigrantsRecv++;
    LastMigrant++;
}

Trace("ReceiveIndividuals completed");
return nMigrantsRecv;

/*
* Appends migrants to 'New' population, called by generate() after each
* generation is new migrants have arrived
*/
void AppendMigrants(nNewMigrants)
int nNewMigrants;
{
    register int i,j;
    if (nNewMigrants != LastMigrant)
        printf("AppendMigrants: something strange is going on\n");
    /* append the migrants to New population */
    for (i = 0; i < nNewMigrants; i++) {
        bcopy((char *)&Migrants[i].Gene, (char *)&New[Popsize+i].Gene, Length);
    }
}
Appendix A

New[Popsizel+1].Perf = Migrants[i].Perf;
New[Popsizel+1].Needs_evaluation = 1;
}
LastMigrant = 0;
}

-----------------------------------------MUTATE.C-----------------------------------------

/*
 * GENESIS Copyright (c) 1986, 1990 by John J. Grefenstette
 * This program may be freely copied for educational
 * and research purposes. All other rights reserved.
 * *
 * file: mutate.c
 *
 * purpose: Perform mutation on the current population.
 * *
 * The global variable Mu_next indicates the position of
 * next mutation, treating the entire population as a linear
 * array of positions.
 * *
 * modified: 7 feb 98
 *
 * 12 nov 86: pass length to Pack() and Unpack()
 */
#include "extern.h"
void Mutate()
{
    static int bits; /* number of bits per pop */
    register int i; /* index of the Mutated structure */
    register int j; /* position within the structure */
    register char k; /* a random allele */
    register int open; /* currently Unpacked Gene */
    static int firstflag = 1;
    Trace("Mutate entered");
    Dtrace("mutation");
    if (firstflag)
    {
        bits = Gapsize*Psize*Length + 0.5;
        firstflag = 0;
    }
    if (M_rate > 0.0)
    {
        open = -1;
        while (Mu_next<bits)
        {
            i = Mu_next/Length; /* Mutated structure */
            j = Mu_next % Length; /* Mutated position */
        }
Appendix A

if (open != i) /* need to Unpack structure i */
{
    Unpack (New[i].Gene, Bitstring, Length);
    open = i;
}
/* choose a random allele */
if (Randint(0,1))
    k = 'I';
else
    k = 'O';
if (k != Bitstring[i]) /* it's an effective mutation */
{
    Bitstring[i] = k;
    New[i].Needs_evaluation = 1;
}
if (New[i].Needs_evaluation)
    Pack (Bitstring, New[i].Gene, Length);
/* update next mutation location */
if (M_rate < 1.0)
    Mu_next += ceil (log(Rand()) / log(1.0 - M_rate));
else
    Mu_next += 1;
/* adjust Mu_next for next Generation */
Mu_next -= bits;
}
Trace("Mutate completed");
Appendix B:

New Programs For DGENESIS

All programs listed in Appendix B are those programs which have either been edited, rewritten, or created by Mike Criswell. Those parts that have not been altered from the original program are in italics, while the original parts are in bold letters.

CxCROSS.C

/*
 * file: cxcross.c
 *
 * purpose: perform CX crossover on entire population
 *
 * modified: April 29, 1996
 *
 */
#include "extern.h"
char premask[CHARSIZE] = { '000', '200', '300', '340',
    '360', '370', '374', '376' };
char postmask[CHARSIZE] = { '377', '377', '077', '037',
    '017', '007', '003', '001' };
void Crossover()
{
    register int mom, dad; /* participants in the crossover */
    register int xpoint1; /* first crossover point w.r.t. structure */
    register int xpoint2; /* second crossover point w.r.t. structure */
    register int xbyte1; /* first crossed byte */
    register int xcity1; /* first crossed city in int arrays */
    register int xbyte2; /* last crossed byte */
    register int xcity2; /* last crossed city in int arrays */
    register int i,j,k; /* loop control variables */
    register char temp; /* used for swapping alleles */
    static int last; /* last element to undergo Crossover */
    int diff; /* set if parents differ from offspring */
    char *kid1; /* pointers to the offspring */
    char *kid2;
    static int firstflag = 1;
    int numsize;
    int citynum;
    int bytenum;
    double k1city,k2city;
    double *ka1;
    double *ka2;
    double *cpy1;
    double kid1_array[65536];
    double kid1_array_cpy[65536];
    double kid2_array[65536];
    ka1 = kid1_array;
    ka2 = kid2_array;

    // Crossover logic...
    /* Perform the crossover here */
    /* Update arrays and pointers */
Appendix B

cpy1 = kid1_array_cpy;
Trace("Crossover entered");
Dtrace("crossover");
if (firstflag)
{
    last = (C_rate*Popsize*Gapsize) - 0.5;
    firstflag = 0;
}
for (mom=0; mom < last; mom += 2)
{
    dad = mom + 1;
    /* kids start as identical copies of parents */
    kid1 = New[mom].Gene;
    kid2 = New[dad].Gene;
    numsize = Length/Genes;
    citynum = Genes;
    /* translate binary to float vector */
    Unpack (New[mom].Gene, Bitstring, Length);
    FloatRep(Bitstring,ka1,Genes);
    Unpack (New[dad].Gene, Bitstring, Length);
    FloatRep(Bitstring,ka2,Genes);
    for (j=0;j<citynum;j++) cpy1[j] = 0;
    /* choose two Crossover points */
    xpoint1 = Randint(0,Length);
    xpoint2 = Randint(0,Length-1);
    /* guarantee that xpoint1 < xpoint2 */
    if (xpoint2 >= xpoint1)
        xpoint2++;
    else
    {
        i = xpoint1;
        xpoint1 = xpoint2;
        xpoint2 = i;
    }
    xbyte1 = xpoint1 / CHARSIZE;
    xcity1 = xpoint1 / Genes;
    xbyte2 = xpoint2 / CHARSIZE;
    xcity2 = xpoint2 / Genes;
    /* do parents differ outside cross segment? */
    diff = 0;
    for (i=0; i < xbyte1; i++) diff += (kid1[i] != kid2[i]);
    for (i=xbyte2+1; i < Bytes; i++) diff += (kid1[i] != kid2[i]);
    if (diff) /* they do */
    {
        /* set up "keeper" array */
        i=0;
        while (cpy1[i]!=1)
        {
            cpy1[i]=1;
            k2city = ka2[i];
            for (j=0;j<citynum;j++)
            {
                if (ka1[j]==k2city)
                { }
Appendix B

i=j;
break;
}
}

/* crossover genes outside of cycle */
for (j=0;j<citynum;j++){
  if (cpy1[j]==0){
    k1city = ka1[j];
    ka1[j] = ka2[j];
    ka2[j] = k1city;
  }
}

/* Integers back to Strings */
StringRep(ka1.Bitstring,Genes);
Pack (Bitstring,New[mom].Gene, Length);
StringRep(ka2.Bitstring,Genes);
Packed (Bitstring,New[dad].Gene, Length);
/* set evaluation flags */
New[mom].Needs_evaluate = 1;
New[dad].Needs_evaluate = 1;
}

Trace("Crossover completed");

------------EVALUATE.C----------------------

/*
* GENESIS Copyright (c) 1986, 1990 by John J. Grefenstette
* This program may be freely copied for educational
* and research purposes. All other rights reserved.
* *
* file: evaluate.c
* *
* purpose: evaluate the current population by
* calling the user-defined function "eval"
* *
* modified: 13 feb 86
* 12 nov 86: pass Length to Unpack()
* 15 sep 90: handle floating pt representation,
* change args to eval
* 18 oct 93: consider newly arrived migrants (ECP)
* 13 mar 96: evaluates new migrants
*/
#include "extern.h"
extern double eval();
void Evaluate(nNewMigrants)
int nNewMigrants;
/
Appendix B

register double performance;
register int i;
Trace("Evaluate entered");
Dtrace("evaluate");
for (i=0; i<(Popsize + nNewMigrants); i++)
{
    if (New[i].Needs_evaluation)
    {
        Unpack(New[i].Gene, Bitstring, Length);
        if (Floatflag)
            FloatRep(Bitstring, Vector, Genes);
        New[i].Perf = eval(Bitstring, Length, Vector, Genes);
        performance = New[i].Perf;
        New[i].Needs_evaluation = 0;
        if (i<Popsize)
        
            Trials++;
            Spin = 0; /* we're making progress */
        if (Trials == 1)
            Best = performance;
        if (Savesize) Savebest(i);
        if (BETTER(performance, Best))
            Best = performance;
        Onsum += performance;
        Offsum += Best;
        if (Dumpflag) Checkpoint(Ckptfile);
    } else{

        performance = New[i].Perf;
        if (BETTER(performance, Best))
            Best = performance;
        if (Savesize) Savebest(i);
    }
}

/*for (i = Popsize; i < Popsize + nNewMigrants; i++)
    performance = New[i].Perf;
    if (BETTER(performance, Best))
        Best = performance;
    if (Savesize) Savebest(i);
*/
Trace("Evaluate completed");

-----------------------------INPOP.C-------------------------------

/*
* ================================================================
* *
* Distributed GENESIS
* *
* Erick Cantu-Paz
* */

43
Appendix B

* ecantu@lamport.rhon.itam.mx
* Instituto Tecnologico Autonomo de Mexico
* 1993
*
* file: inipop.c
*
* purpose: Communication of initial structures
*
*/
#include "extern.h"
define ENDPOP "end_of_initial_pop"
extern int Sockets[]; /* endpoint for communications */
extern PROCESS *ProcTable; /* process table, managed by main process */
/
* read next structure from 'fp'
*/
char *ReadStructure(fp, bitstring)
FILE *fp;
char *bitstring;
{
    char *status="a";
    char line[200];
    int j;
    if ( Floatflag )
        for (j = 0; (j < Genes) && (status != 0); j++) {
            status = ReadLine(fp, line, 200);
            sscanf(line, "%lf", &Vector[j]);
        }
    StringRep(Vector, bitstring, Genes);
} else {
    status = ReadLine(fp, line, 200);
    sscanf(line, "%s", bitstring);
}

    return status;
} /*
* Send initial structures (read from Initfile)
* returns number of main's initial structs
*/
int SendPop()
{
    register int i, j, k; /* loop control */

    int nStructures; /* # of structs in Initfile */
    int pop; /* # of structs for each process*/
    int returnvalue;
    FILE *fp; /* pointer to Initfile */
Appendix B

char msg[40];  /* error message */
Trace("SendPop entered");
nStructures = CountLines(Initfile);
if (Floatflag)
    nStructures /= Genes;
if (nStructures > Totalpop) nStructures = Totalpop;
}

/* open Initfile */
if ((fp = fopen(Initfile, "r") == NULL) {
    printf(msg, "SendPop: can't open %s", Initfile);
    Error(msg);
}

/* main process */
if (Propiniflag)
    pop = nStructures / ProcTable[0].pop;
else
    pop = nStructures / Processes;
for (i = 0; i < pop; i++){
    ReadStructure(fp, Bitstring);
    Pack(Bitstring, New[i].Gene, Length);
    New[i].Needs_evaluation = 1;
}
returnvalue = pop;

/* send initial structures to other processes */
for (i = 1; i < Processes; i++){
    k = i;
    if (Propiniflag)
        pop = nStructures / ProcTable[i].pop;
    else
        pop = nStructures / Processes;
    for (j = 0; j < pop; j++){
        k=j;
        ReadStructure(fp, Bitstring);
        if (Send(Sockets[0], ENDPOP, strlen(ENDPOP), ProcTable[j].addr) < 0)
            IOSend("SendPop can't send");
    }
}

/* send end of population message */
for (j = 1; j < Processes; j++)
    if (Send(Sockets[0], ENDPOP, strlen(ENDPOP), ProcTable[j].addr) < 0)
        IOSend("SendPop can't send end-of-pop");
fclose(fp);
Trace("SendPop completed");
return returnvalue;
}

/* Receives initial structures
* returns number of structs received
*/
int ReceivePop()
Appendix B

```c
register int i;
char buffer[MAXMSG]; /* send/receive buffer */
int status; /* indicates last structure */
Trace("ReceivePop entered");
bzero(buffer, MAXMSG);
recvfrom(Sockets[0], buffer, MAXMSG, 0, (struct sockaddr *) NULL, 0);
i = 0;
status = strcmp(buffer, ENDPOP);
while (status != 0) {
    if (strlen(buffer) > 0) {
        Pack(buffer, New[i].Gene, Length);
        New[i].Needs_evaluation = 1;
i++;
    } else {
        IOError("ReceivePop Can't receive");
        status = strcmp(buffer, ENDPOP);
    }
Trace("ReceivePop completed");
    return i;
}
```

---

INITMAKER.C

```c
/* Initmaker.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#define Randint(low, high) ((int)((1+(rand()%(high))))
#define INPUTSTR(Prompt, Var)
    char str[30];
    printf (Prompt); printf("[\%s]", Var);
    fflush (stdout);
    sscanf (gets (str), "\%s”, Var);
}
#define INPUT(Prompt, Format, Var)
    char str[50];
    printf (Prompt); printf("[\"]");
    printf (Format, Var);
    printf ("."); fflush (stdout);
    sscanf (gets (str), Format, &Var);
}

main()
{
    int x, temp;
```
Appendix B

```c
int *individual;
int st_array [65536];
register int i,j,k;
int genes,population,stop = 0;
FILE *fp;
char exp[80]; /* name of experiment */
char linkfile[30]; /* file to write the links */

st_array[0] = 0;
individual = st_array;
printf ("\n\tDGENSEIS tsp initfile Generator Program\n\n");
INPUT ("Enter number of Genes", "%d", genes);
INPUT ("Enter number of Structures", "%d", population);
exp[0] = 0;
INPUTSTR ("Experiment name", exp);
sprintf (linkfile, "init.%s", exp);

fp = fopen (linkfile, "w");
if (fp == NULL)
    printf ("links: can't open linkfile\n");
else {
    for (i = 0; i < population; i++) {
        for (j = 0; j < genes; j++) {
            individual[j] = j + 1;
        }
        for (j = 0; j < (genes * 5); j++) {
            x = (Randint(1, genes) - 1);
            temp = individual[x];
            individual[x] = individual[0];
            individual[0] = temp;
        }
        for (j = 0; j < genes; j++) {
            fprintf (fp, "%.0f\n", individual[j]);
        }
    }
    fclose (fp);
    printf ("\nfile \"%s\" created successfully\n\n", linkfile);
}
```

---MUTATE.C-----------------------------

/*
 * GENESIS Copyright (c) 1986, 1990 by John J. Grefenstette
 * This program may be freely copied for educational
 * and research purposes. All other rights reserved.
 *
 * file: mutate.c
 *
 * purpose: Perform mutation on the current population.
 */
Appendix B

The global variable Mu_next indicates the position of
next mutation, treating the entire population as a linear
array of positions.

* modified: 7 feb 98
* 12 nov 86: pass length to Pack() and Unpack()
*/
#include "extern.h"
void Mutate()
{
    static int bits;    /* number of bits per pop */
    register int i;         /* index of the Mutated structure */
    register int j;         /* position within the structure */
    register int x;         /* second point within the structure */
    register char k;        /* a random allele */
    register int open;      /* currently Unpacked Gene */
    static int firstflag = 1;
    double tempdbl;
    double *flt_ary;
    double flt_array[65536];
    flt_ary = flt_array;
    Trace("Mutate entered");
    Dtrace("mutation");
    if (firstflag)
        
    bits = Gapsize*Popsize*Length + 0.5;
    firstflag = 0;
    
    if (M_rate > 0.0)
    
    open = -1;
    while (Mu_next<bits)
    
        i = Mu_next/Length;         /* Mutated structure */
        j = Mu_next % Genes;        /* Mutated position */
        x = Randint(0,(Genes-1));   /* Second Mutated position */
    if (open != i) /* need to Unpack structure i */
        
        Unpack (New[i].Gene, Bitstring, Length);
        FloatRep(Bitstring,flt_ary,Genes);
        open = i;
    
    if (x != j) /* it's an effective mutation */
    
        tempdbl = flt_ary[j];
        flt_ary[j] = flt_ary[x];
        flt_ary[x] = tempdbl;
    
    New[i].Needs_evaluation = 1;


48
Appendix B

if (New[i].Needs_evaluation)
    StringRep(bit_ary, Bstring, Genes);
    Pack ( Bitstring, New[i].Gene, Length);
}
/* update next mutation location */
if (M_rate < 1.0)
    Mu_next += ceil (log(Rand()) / log(1.0 - M_rate));
else
    Mu_next += 1;
/* adjust Mu_next for next Generation */
Mu_next -= bits;
} Trace("Mutate completed");

--------------------OXCROSS.C-------------------------

/*
 * file: oxcross.c
 * *
 * purpose: perform two-point OX crossover on entire population
 * *
 */
#include "extern.h"
char premask[CHARSIZE] = { '000', '200', '300', '340',
    '360', '370', '374', '376'};
char postmask[CHARSIZE] = { '377', '177', '077', '037',
    '017', '007', '003', '001'};
void Crossover()
{
    register int mom, dad; /* participants in the crossover */
    register int xpoint1; /* first crossover point w.r.t. structure */
    register int xpoint2; /* second crossover point w.r.t. structure */
    register int xbyte1; /* first crossed byte */
    register int xbyte2; /* last crossed byte */
    register int xcity1; /* first crossed city in int arrays */
    register int xcity2; /* last crossed city in int arrays */
    register int i,j,k,n; /* loop control variables */
    register int last; /* used for swapping alleles */
    static int flag = 1;
    int numsize;
    int citynum;
    int bytenum;
    int x;

49
Appendix B

```c
int add_it, go_ahead = 1;
double k1city, k2city;
double *ka1;
double *ka2;
double *cpy1;
double kid1_array [65536];
double kid1_array_cpy [65536];
double kid2_array [65536];
ka1 = kid1_array;
ka2 = kid2_array;
cpy1 = kid1_array_cpy;
Trace("Crossover entered");
Dtrace("crossover");
if (firstflag)
{
llast = (C_rate*Psize*Gapsize) * 0.5 ;
firstflag = 0;
}
for (mom=0; mom < last ; mom += 2)
{

dad = mom + 1;
/* kids start as identical copies of parents */
kid1 = New[mom].Gene;
kid2 = New[dad].Gene;
numsize = Length/Genes;
citynum = Genes;
/* translate binary to float vector */
Unpack (New[mom].Gene, Bitstring, Length);
FloatRep(Bitstring, ka1, Genes);
FloatRep(Bitstring, cpy1, Genes);
Unpack (New[dad].Gene, Bitstring, Length);
FloatRep(Bitstring, ka2, Genes);
/* choose two Crossover points */
xpoint1 = Randint(0,Length);
xpoint2 = Randint(0,Length-1);
/* guarantee that xpoint1 < xpoint2 */
if (xpoint2 >= xpoint1)
    xpoint2++;
else
{
i = xpoint1;
xpoint1 = xpoint2;
xpoint2 = i;
}
xbyte1 = xpoint1 / CHARSIZE;
xcity1 = xpoint1 / Genes;
xbyte2 = xpoint2 / CHARSIZE;
xcity2 = xpoint2 / Genes;
/* do parents differ outside cross segment? */
diff = 0;
for (i=0; i < xbyte1; i++) diff += (kid1[i] != kid2[i]);
```
Appendix B

for (i=xbyte2+1; i < Bytes; i++) diff += (kid1[i] != kid2[i]);
if (diff) /* they do */
{

/* chromosome 1 */
    i=xcity2+1;
    j=0;
    x=citynum+2;
    for (z=0;z<x;z++){
        if (i==citynum){
            i=0;
        }
        else if (j==xcity1){
            j=xcity2+1;
        }
        else {
            add_it = 1;
            for (k=xcity1;k<(xcity2+1);k++){
                if (cpy1[i]==ka2[k]) add_it=0;
            }
            if (add_it){
                ka1[j] = cpy1[i];
                j++;
            }
            i++;
        }
    }
/*copy ka2 into cpy1 */
for (i=0;i<citynum;i++){
    cpy1[i]=ka2[i];
}
/* chromosome 2 */
i=xcity2+1;
    j=0;
    for (z=0;z<x;z++){
        if (i==citynum) i=0;
        else if (j==xcity1) j=xcity2+1;
        else {
            add_it = 1;
            for (k=xcity1;k<(xcity2+1);k++){
                if (cpy1[i]==ka1[k]) add_it=0;
            }
            if (add_it){
                ka2[j] = cpy1[i];
                j++;
            }
            i++;
        }
    }
/* crossover middle part */
for (k=xcity1;k<(xcity2+1);k++){
Appendix B

k1city = ka1[k];
ka1[k] = ka2[k];
ka2[k] = k1city;
}
/* Integers back to Strings */
StringRep(ka1,Bitstring,Genes);
Pack(Bitstring,New[mom],Gene,Length);
StringRep(ka2,Bitstring,Genes);
Pack(Bitstring,New[dad],Gene,Length);
/* set evaluation flags */
New[mom].Needs_evaluation = 1;
New[dad].Needs_evaluation = 1;

/
/ Trace("Crossover completed");
/

-------------------------------------------PMXCROSS.C-------------------------------------------

/*
* GENESIS Copyright (c) 1986, 1990 by John J. Grefenstette
* This program may be freely copied for educational
* and research purposes. All other rights reserved.
* *
* file: pmxcross.c
*
* purpose: perform two-point PMX crossover on entire population
* *
* #include "extern.h"
char premask[CHARSIZE] = { '000', '200', '300', '340',
   '360', '370', '374', '376' };
char postmask[CHARSIZE] = { '377', '777', '077', '037',
   '017', '007', '003', '001'};
void Crossover()
{/n
   register int mom, dad; /* participants in the crossover */
   register int xpoint1; /* first crossover point w.r.t. structure */
   register int xpoint2; /* second crossover point w.r.t. structure */
   register int xbyte1; /* first crossed byte */
   register int xcity1; /* first crossed city in int arrays */
   register int xbyte2; /* last crossed byte */
   register int xcity2; /* last crossed city in int arrays */
   register int i,j,k; /* loop control variables */
   register char temp; /* used for swapping alleles */
   static int last; /* last element to undergo Crossover */
   int diff; /* set if parents differ from offspring */
   char *kid1; /* pointers to the offspring */
   char *kid2;
static int firstflag = 1;
int numsize;
int citynum;
int bytenum;
double k1city,k2city;
double *ka1;
double *ka2;
double *cpy1;
double kid1_array[65536];
double kid1_array_cpy[65536];
double kid2_array[65536];
ka1 = kid1_array;
ka2 = kid2_array;
cpy1 = kid1_array_cpy;
Trace("Crossover entered");
Dtrace("crossover");
if (firstflag)
  /
  last = (C_rate*Popsize*Gapsize) * 0.5 ;
  firstflag = 0;
/
for (mom=0; mom < last ; mom += 2)
/
  dad = mom + 1;
  /* kids start as identical copies of parents */
  kid1 = New[mom].Gene;
  kid2 = New[dad].Gene;
  numsize = Length/Genes;
  citynum = Genes;
  /* translate binary to float vector */
  Unpack (New[mom].Gene , Bitstring, Length);
  FloatRep(Bitstring,ka1,Genes);
  FloatRep(Bitstring,cpy1,Genes);
  Unpack (New[dad].Gene , Bitstring, Length);
  FloatRep(Bitstring,ka2,Genes);
  /* choose two Crossover points */
  xpoint1 = Randint(0,Length);
  xpoint2 = Randint(0,Length-1);
  /* guarantee that xpoint1 < xpoint2 */
  if (xpoint2 >= xpoint1)
    xpoint2++;
else
/
  i = xpoint1;
  xpoint1 = xpoint2;
  xpoint2 = i;
/
  xbyte1 = xpoint1 / CHARSIZE;
  xcity1 = xpoint1 / Genes;
  xbyte2 = xpoint2 / CHARSIZE;
  xcity2 = xpoint2 / Genes;
Appendix B

/* do parents differ outside cross segment? */
diff = 0;
for (i=0; i < xbyte1; i++) diff += (kid1[i] != kid2[i]);
for (i=xbyte2+1; i < Bytes; i++) diff += (kid1[i] != kid2[i]);
if (diff) /* they do */
{
    /* perform crossover on chromosome1 */
    for (i=city1; i<(city2+1); i++)
    {
        k1city = ka1[i];
        k2city = ka2[i];
        for (j=0; j<citynum; j++)
        {
            if (ka1[j]==k2city)
            {
                ka1[j]=k1city;
                ka1[i]=k2city;
            }
        }
    }

    /* crossover on chromosome2 */
    for (i=city1; i<(city2+1); i++)
    {
        k1city = cpyI[i];
        k2city = ka2[i];
        for (j=0; j<citynum; j++)
        {
            if (ka2[j]==k1city)
            {
                ka2[j]=k2city;
                ka2[i]=k1city;
            }
        }
    }

    /* Integers back to Strings */
    StringRep(ka1, Bitstring, Genes);
    Pack (Bitstring, New[mom].Gene, Length);
    StringRep(ka2, Bitstring, Genes);
    Pack (Bitstring, New[dad].Gene, Length);
    /* set evaluation flags */
    New[mom].Needs_evaluation = 1;
    New[dad].Needs_evaluation = 1;
}

Trace("Crossover completed");

#include <math.h>
double eval(str, length, vect, genes)
char str[];
    /* string representation */
int length;
    /* length of bit string */
double vect[];
    /* floating point representation */
int genes;
    /* number of elements in vect */
{
register int i,j,lastcity;
double a,b= 0.0;
double edgestart,totdist;
struct city_coords {
    double x;
    double y;
};
struct city_coords travelmap[] = {
    0.0,0.0,
    0.0,1.0,
    0.0,2.0,
    0.0,3.0,
    0.0,4.0,
    1.0,4.0,
    2.0,4.0,
    3.0,4.0,
    4.0,4.0,
    4.0,3.0,
    4.0,2.0,
    4.0,1.0,
    4.0,0.0,
    3.0,0.0,
    2.0,0.0,
    1.0,0.0
};
totdist = 0;
edgestart = 0;
for (i=1; i< genes; i++) /* calculate distances from starting pos */
{
    j = vect[i];
    lastcity = vect[i-1];
    a = (travelmap[j].x)-(travelmap[lastcity].x);
    b = (travelmap[j].y)-(travelmap[lastcity].y);
    edgestart = sqrt((a*a) + (b*b));
    totdist = edgestart + totdist;
}
/* and back to starting pos */
lastcity = vect[0];
a = (travelmap[j].x)-(travelmap[lastcity].x);
b = (travelmap[j].y)-(travelmap[lastcity].y);
edgestart = sqrt((a*a) + (b*b));
totdist = edgestart + totdist;
return (totdist);
Bibliography


