Genetic Algorithms with Punctuated Equilibria: Parallelization and Analysis

A Thesis
In TCC 402

Presented to:

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
Of the Requirements for the Degree
Bachelor of Science in Computer Science

by
Justin Turner

March 7, 1998

Technical Advisor: Dr. Worthy Martin
TCC Advisor: Dr. Ingrid Soudek

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in Humanities Courses.

Signed ______________________________

Approved ____
Dr. Worthy Martin

Approved ____
Dr. Ingrid Soudek

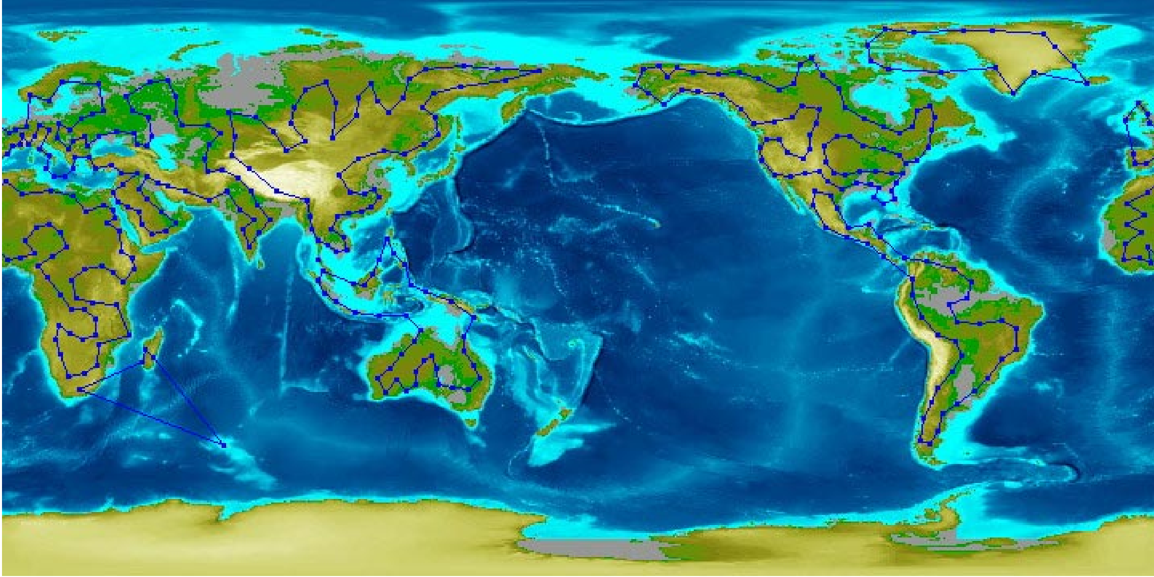


Figure 1: Sample Application of the Travelling Salesman Problem. The travelling salesman problem could be used to find the shortest roundtrip path through a several cities across the globe. *Created by Dan Ignat and Justin Turner, March 1998.*

Preface

Acknowledgements

I wish to thank my technical advisor, Dr. Worthy Martin, for his expertise and support throughout the research for this document. Professor Martin provided the inspiration and direction for this project and aided my partner and me throughout the endeavor. I would also like to thank Dr. Ingrid Soudek for her valuable advice and assistance in the creation of the technical report. Finally, I would like to thank Mark Hyatt, John Karpovich, Adam Ferrari, and Greg Lindahl for their assistance with the Legion system.

The project described herein is a joint venture with Dan Ignat, an undergraduate student in the math department with a computer science concentration. Mr. Ignat collaborated with me on all aspects of the research and software development, as well as the figures and appendices contained within this document.

Note to the reader

The results contained within this document represent a preliminary set of those that will be collected throughout the course of our research. Due to the experimental nature of the system that the software was developed under, we were unable to generate a large base of results or examine all of the aspects of GAPE that were planned in the proposal for this document by the deadline for submission. This project will continue past the deadline for the technical report, and additional results will be added as they are produced.

PREFACE **3**
Acknowledgements 3
GLOSSARY OF TERMS **6**
1.0 INTRODUCTION **8**
1.1 Travelling Salesman Problem 8
1.2 NP-Completeness 9
1.3 Genetic Algorithms 10
1.4 GAPE 11
1.5 Literature Review 13
1.6 Rationale and Scope 14
1.7 Overview of Remainder of Report 15
2.0 METHODOLOGY **17**
2.1 Parallel Objects and Legion 17
2.2 Software Implementation 18
2.2.1 Instantiation 18
2.2.2 Evolution and Niebel's Algorithms 19
2.2.3 Migration 23
2.3 Testing 24
2.4 Equipment 26
3.0 HYPOTHESES **27**
3.1 Varying the Number of Populations 27
3.2 Varying the Generations per Epoch 28
3.3 Parallelizing the Code 29
4.0 RESULTS **30**
4.1 Variable Number of Populations 30
4.2 Variable Number of Generations per Epoch 30
4.3 Timing of Parallel and Sequential Code 31
5.0 CONCLUSIONS **34**
5.1 Summary 34
5.2 Interpretation 35
5.3 Recommendations 36
WORKS CITED **37**
APPENDIX A: ANNOTATED REFERENCE MANUAL **38**

Glossary of Terms

Crossover - The combining of two individuals to form a third with combined traits from both parents
Epoch - A period of time during which a population evolves without outside influence
Fitness - A measure of the quality of an individual
Genetic Algorithm - A problem solving approach that mimics evolution by treating solutions as individuals and combining them to form new solutions
Genetic Algorithms with Punctuated Equilibra - An extension of genetic algorithms where individuals are grouped into populations, which evolve separately and occasionally exchange individuals.
Genotype - The hidden, inner representation of an individual's traits
Hypercube - An **n**-dimensional cube consisting of 2^n nodes, where each node is connected to **n** other nodes.
Individual - A solution to the problem being analyzed, which can be combined with other individuals to generate new ones
Interconnection Matrix - An array that describes the interconnections between populations in a GAPE run.
Legion - A system that can link several computer of various architectures into a distributed network, and can execute parallel programs on them
Mentat Programming Language - A language for writing parallel programs for the Legion system
Minimum Spanning Tree - The graph with the smallest total length that traverses all points and has no cycles
Mutation - A random change in one element of an individual
NP-Complete - A set of problems that have been shown to be very difficult to solve exactly for large instances
Phenotype - The physical manifestation of a genotype
Population - An isolated set of individuals that evolve over time
Reduction - Choosing the individuals that will survive to the next generation based on fitness
Selection - Choosing the individuals that will mate based on fitness
Traveling Salesman Problem - The problem of finding the shortest roundtrip distance through a set of points

Abstract

This paper describes the creation of a parallel implementation of a genetic algorithms with punctuated equilibria (GAPE) application for the travelling salesman problem (TSP). It also investigates two aspects of GAPE by using the results from the software. These aspects are the relationship between the number of populations used and the solution quality over time, and the relationship between the number of generations per epoch and the solution quality over time. The analysis of these variables is meant to shed light on their role in GAPE algorithms in general.

We believe that a parallel implementation of GAPE run over a large network will provide a significant speedup over sequential implementations. Further, we feel that the results described within indicate that the number of populations have relatively little effect on the solution quality of GAPE, especially for small problem sizes, and that isolating populations for extended periods of time produces better long-term results than those obtained with frequent communication.

1.0 Introduction

Genetic Algorithms with Punctuated Equilibria (GAPE) is a nontraditional way of solving problems that are too complex to be solved exactly in a reasonable amount of time. Due to the distinct populations that are the defining characteristic of GAPE algorithms, GAPE software lends itself very well to parallelization. This paper describes the implementation of a parallel GAPE algorithm, and investigates two aspects of GAPE using the results of the software. The results should be extendable to GAPE algorithms in general.

1.1 Travelling Salesman Problem

In its simplest form, the Travelling Salesman Problem (TSP) is the following: *Given a set of **n** cities, what is the shortest path that a salesman can take in which he visits each city once and returns to his original starting point?* This problem has a wide array of applications. These include any problems where one wishes to find the route through a set of points that has the minimum possible distance. Some problems that the TSP applies to include oil exploration, computer chip design, telephone line routing, and protein modeling [17]. The frontispiece illustrates a possible TSP application. It shows a set of paths traversing various cities across the globe. Minimizing the distance of these paths might be important for delivery routes or telephone line connections.

This seemingly simple problem has been studied for over two centuries by mathematicians, but the efficiency of the solutions developed is still not satisfactory for many modern applications. The reason behind this relies on the concept of NP-completeness.

1.2 NP-Completeness

NP-complete is the name that scientists have given to a set of problems that is very difficult to solve precisely. The concept divides problems into two classes: those whose solution is a polynomial-time algorithm, and those whose best-known algorithms are nonpolynomial. Polynomial-time problems are those for which the time to arrive at a solution is related to the size of the problem by a polynomial. For instance, the minimum time required to multiply two random square matrices of numbers together is directly proportional to the square of the size of the matrix. This means that if a computer took four seconds to multiply two matrices of size 10x10 together, it might take 16 seconds to multiply two matrices of size 20x20 together and about a minute to multiply a 40x40 matrix.

On the other hand, nonpolynomial-time algorithms are those for which the solution cannot be related to the size of the problem by a polynomial. For instance, the relationship may be on the order of 2^n or n^n where n is the size of the problem. As a result, the time necessary to solve a problem tends to increase dramatically with the problem size. If a problem of size 100 takes a minute to solve, a problem of size 200 might take a day, and a problem of size 300 might take a lifetime. Problems that fall into this latter class are termed "NP-complete" by computer scientists. They include the travelling salesman problem, to which this project is being applied.

It is often infeasible to solve large instances of NP-complete problems exactly. Since it is potentially impractical to find the best solution to the travelling salesman problem when large numbers of cities are involved, researchers have looked to other methods. These include techniques that do not attempt to find the best solution to a given

problem, but instead find an approximate solution. Ideally, this solution will be very close to the optimal solution in quality, but can be found in a small fraction of the time that it would take to find the best solution. One method that is commonly used for NP-complete problems is genetic algorithms, which is the topic of this paper.

1.3 Genetic Algorithms

Genetic algorithms are a method of solving problems based on sexual reproduction and genetics. In sexual reproduction, two individuals contribute traits that combine to form a new individual, who carries those traits from both parents onto the next generation. Sometimes the offspring is better suited to its environment than its parents, and sometimes it is worse suited. The better the individual, the greater chance it has of surviving to mate with other individuals and thus contribute its genetic material to the next generation [19].

Similarly, with genetic algorithms, a population of individuals evolves over time. However, the individuals in this case are solutions to the problem at hand. The algorithm begins with a pool of random solutions. Each solution is treated like an individual, and combines with other solutions to produce new ones. Some of these solutions will be superior to their parents, and others will be inferior.

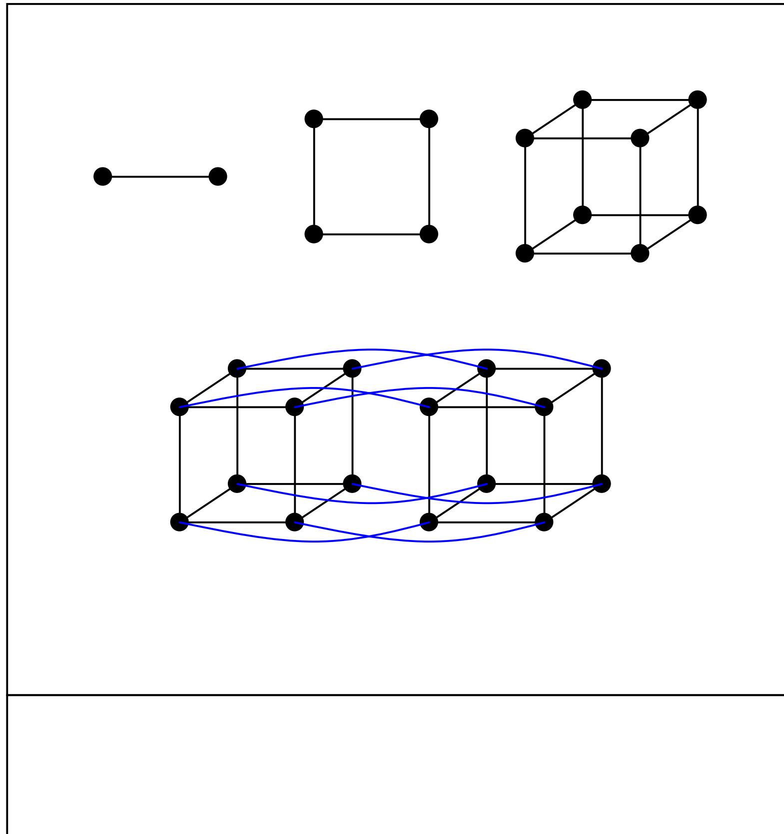
The chance a solution has to reproduce is determined by its fitness, which is a function of how good the solution is. Over time, good solutions combine to produce better and better solutions and, ideally, an answer that is close to the optimal one is produced in a relatively short period of time.

1.4 GAPE

In nature, mating is also influenced by the location of individuals. For instance, a cockroach in England has a much higher probability of mating with another English cockroach than with one in China. Biologists often discuss this concept in terms of populations, which are isolated groups of individuals evolving separately and adapting to their particular environments. Over time, if there are not major environmental changes, evolution within a population reaches equilibrium, where relatively little overall change occurs. Occasionally, some geological incident will occur, allowing individuals from populations to intermingle. When this intermingling happens, new genetic material is introduced, resulting in a surge in evolution. Paleontologists refer to this as the theory of punctuated equilibria [8].

A variation on genetic algorithms called genetic algorithms with punctuated equilibria (GAPE) has been developed based on this theory. The algorithm's solutions are partitioned into several populations that are isolated from each other. These populations are allowed to evolve separately, gradually moving towards a stasis. After a set amount of time referred to as an epoch has elapsed, the populations exchange some solutions, resulting in a burst of evolution.

An interconnection matrix governs communication between populations. The matrix defines the populations to which each population sends individuals after the end of each epoch. At one extreme, an interconnection matrix with no connections would be equivalent to all populations evolving separately with no interaction. At the other extreme, a matrix with all connections filled would be similar to a traditional genetic algorithm with one population.



There is a variety of popular interconnection matrices, including grids and toruses. For all experiments contained within this document, we used a structure called a hypercube. A hypercube is, in effect, a cube of **n** dimensions. Its most notable property is that each of the 2^n nodes, which are populations in this case, has exactly **n** nodes connected to it. As illustrated in figure 2, a hypercube of one dimension would be a line. Each of the two nodes thus has one connection. A hypercube of two dimensions would be a square, with each of the four nodes having two connections. A hypercube of three dimensions would be a cube. A hypercube of four dimensions can be visualized as two

cubes, with each node on the first cube connected to the corresponding node on the second cube.

The software described within this report applies GAPE to the travelling salesman problem. In this case, the individuals will be actual tours, since they are solutions to the problem. The method by which individuals reproduce is somewhat complicated and will be explained in detail within.

1.5 Literature Review

The concepts of modern genetic algorithms were first developed by John Holland in the 1960s. His goal was to develop the theory and procedures necessary for the creation of general progams and machines with unlimited capability to adapt to arbitrary environments [15]. Before him, a handful of biologists had introduced concepts related to those involved in modern genetic algorithms. Among them were N.A. Barricelli and A.S. Fraser, who were attempting to understand natural phenomena [8].

Srinivas and Patnaik provided a fairly comprehensive analysis of the basic characteristics and issues surrounding the development of GAs [16]. Some of the issues that they discussed were influential in the formulation of the genetic algorithms with punctuated equilibria (GAPE) method.

GAPE is a somewhat new concept, developed by Cohoon, Hedge, Martin, and Richards in 1987 [1]. They defined the algorithm and the characteristics which differentiated it from the sequential GA as an alternative to traditional genetic algorithms at the University of Virginia in April of 1991.

It was originally applied to the Floorplan Design Problem [2], which is an important step in the VLSI design cycle. It involves arranging a given set of modules in

the plane to minimize the weighted sum of area and wirelength measures [5]. The GAPE results exceeded those of the simulated annealing method, which is a more widely accepted form of search, in both average cost and optimal solution. GAPE has the potential for success in many other aspects and is well thought of in the field, although it has not enjoyed the popularity that other methods have [6].

The next application of GAPE was VLSI routing [3]. On this example, GAPE once again outperformed the traditional sequential implementations. Similar results were obtained for other problem instances, including the VLSI channel and switchbox routing problems, which used a parallel implementation over a distributed network of workstations [11]. Some research has also been done to understand why some evolutionary algorithms succeed when applied to the VLSI field and why others fail [12].

In 1997, Riccardo Poli introduced the concept of Parallel Distributed Genetic Programming, whereby he produced a new form of genetic programming with a high degree of parallelism. His analysis indicated that parallelism provided significantly increased performance, which suggests that GAPE might also perform quite well over multiple processors, as it is highly parallelizable [7].

In February of 1996, William Niebel applied the GAPE method to the travelling salesman problem on a single processor. During his testing, he found that the method performed well, on average coming within 1% of the known best solution while the standard GA solution only came within 2% [9].

1.6 Rationale and Scope

My partner Daniel Ignat and I have expanded on the work done by William Niebel in 1996. We have rewritten the genetic algorithm code to run over a Legion

system. This allows the software to run in parallel over several computers. If a large network of computers is used, this should result in a significant speedup over the single processor algorithm, especially for runs with large numbers of populations.

In addition, we have analyzed the GAPE algorithm as applied to the Travelling Salesman Problem in two respects. The first objective was to examine the effects of varying the total number of populations while keeping the number of individuals and migration rates constant. This involves looking at the effect of spreading the individuals among a number of populations on the quality of the final solution and the speed at which a good solution is obtained. The second aspect we looked at was the effect of varying the number of generations per epoch, while holding the total number of generations constant. This highlights the effect of letting populations evolve on their own to reach local minima, as opposed to having frequent communication to bring in higher quality individuals from outside.

Two objectives that were proposed were not completed in time to be included in this report. These were the effects of varying interconnection matrix of populations, thus changing which populations are allowed to intermingle with which others, and the effects of varying the fitness function among different populations, thus putting different selective pressures on different populations. These will be included in a later version of this document.

1.7 Overview of Remainder of Report

The body of this report will first explain the implementation of the parallel GAPE software. It will then relate our hypotheses pertaining to the effectiveness of parallelizing the GAPE algorithm and to the results from testing. Next, the data collected from the

experimental software runs will be presented, along with an analysis of this data, and an interpretation of the data relative to the project's objectives. Finally, some recommendations for future research will be presented.

2.0 Methodology

This section contains a description of the methods we used to carry out the software development and to obtain the results. The work performed on this thesis consisted of writing the software to distribute GAPE across a Legion system, and analyzing two major variables of the GAPE method: the number of populations and the number of generations per epoch.

2.1 Parallel Objects and Legion

The analyses to be performed in this research project would require a cumbersome amount of time using existing GAPE software. Therefore, the first major research task was the development of a parallel equivalent of the sequential GAPE implementation written by William Niebel. To do so, we chose to use the Legion system, which is an object-based software package designed to allow distributed computing.

The Legion system allows the programmer to define objects as *Mentat* classes, which indicates that they are to be distributed across the network. As a result, member functions of several instances of that class can do their work in parallel, with interruptions occurring only when data needs to be given to another process, or received from another process. Therefore, a significant speedup can result if many objects are able to run independently for extended periods of time.

There are also major drawbacks to running objects in parallel. There is a large overhead when parallel objects are created, due to the fact that an entire new process needs to be created on another machine, and all the information that allows it to communicate with other processes must be established. Additionally, memory references

outside a given class are generally costly, since they require the process to communicate over the network. If large packets of data are transferred from one parallel object to another, the slowdown can be very costly. This is because data from all arrays and pointers must be extracted, packaged, and sent over the network due to the disjoint memory spaces of the computers on the network.

As a result of these factors, we decided to use the population class as our only distributed class. This choice was made because each population spends the vast majority of its time evolving, which requires no communication between populations. Communication is only necessary at the end of each epoch, when individuals are exchanged.

2.2 Software Implementation

William Niebel wrote a sequential version of GAPE for the travelling salesman problem in 1996. In writing a parallel version of the algorithm, we were able to reuse the majority of Niebel's low-level algorithms. However, major modifications were made to the way in which objects are handled and to the high level execution of the software. These aspects, along with the basic theory behind the software, will be described below. A more detailed description of the software is included as Appendix A.

2.2.1 Instantiation

After completing some preliminary initialization, the software begins by creating the various population instances. Each population receives its own process and is moved to one of the computers on the Legion network. We allow the Legion system to manage which computer each process is sent to. Legion is in a developmental stage and at this

point randomly chooses a computer to send each process to, so the code would run faster if we were to force an equal number of populations to go to each machine. However, when Legion is completed, it will decide where to send processes more intelligently; and therefore, we have allowed the system to control process distribution with the hopes that this will be the best decision in the long term.

Next, a signal is sent to each of the populations from the main process telling them to begin the evolution process. After this point, the only communication between processes will occur between epochs.

2.2.2 Evolution and Niebel's Algorithms

During an epoch, each population evolves without influence from the other populations. Each generation consists of selection for mating, reproduction, reduction, and mutation. In order to understand these processes, one must first understand the underlying representation of individuals as implemented by Mr. Niebel.

2.2.2.1 Niebel's Algorithms

As stated earlier, an individual in the case of the travelling salesman problem is a tour of the city. Its fitness is therefore related to the length of the tour, with a lower tour length resulting in better fitness. The immediate problem with this implementation of an individual is that there is not an obvious way to cross two tours, since a tour is simply a list of the cities in a particular order. One could try to place the cities in a location that is near to the average of its location in the two parents, but this would result in many cities wishing to be placed near to the middle, and thus would probably not retain the characteristics of its parents.

In order to remedy this, the software represents the genotype of individuals by problem maps. A genotype is an underlying representation, as opposed to a phenotype which is the manifestation of the genotype. For example, in genetics, a genotype might be a string of a certain four amino acids in an allele, and the phenotype might be brown eyes. Different genotypes are generated by actually "perturbing" the cities on the map, or shifting them slightly based on the standard deviation of the population. This allows for an easy implementation of crossover.

Another benefit of the perturbed map representation is that a perturbed map may translate into a tour that would be very difficult to find using a normal tour representation because it is not similar to other good tours. This could be very beneficial because the optimal tour for a map is often not one that is easily generated through crossover.

When two perturbed maps reproduce, each city in their offspring will be the average of the locations of the same city in the two parents. Thus, the offspring will be a blend of its parents, which is what is desired. The offspring is then perturbed so that the cities do not all converge on the center. Figure 3 part (a) shows a plot of a 51 city map. Part (d) shows a perturbed version of the map.

Now that we have a genotype that is easy to mate, we must have a way to translate it into a phenotype, or a tour. This is done by using a minimum spanning tree (MST) as an intermediate structure. A spanning tree is a connected graph without cycles that touches all vertices of the map. In other words, it is a set of edges connected all cities in the map without creating any loops. When the sum of the edges of a spanning

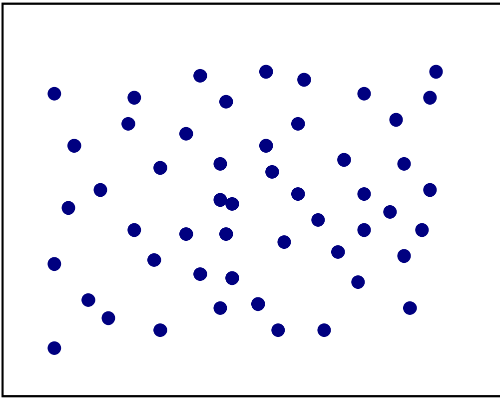
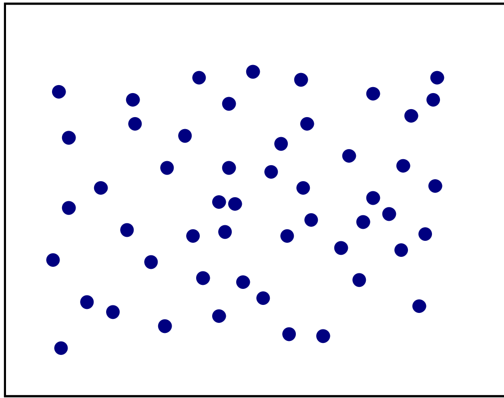
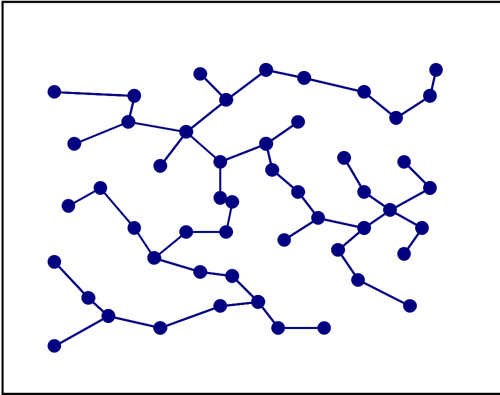
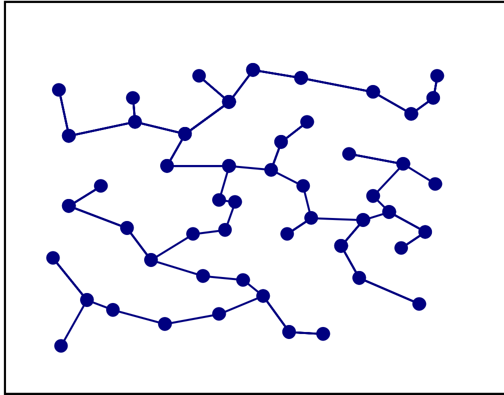
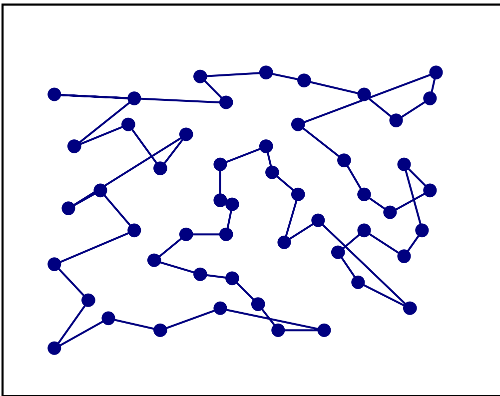
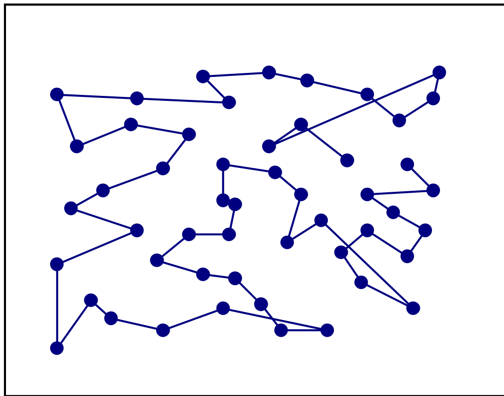







Figure 3: Changing from Genotype to Phenotype

Created by Dan Ignat and Justin Turner, March 1998.

tree has the smallest possible distance, it is a minimum spanning tree. There is an efficient algorithm that finds an MST for a set of points, which is explained in the function Individual::mstify() in appendix A. Figure 3 part (b) shows the minimum spanning tree generated from the 51 city map. Part (e) shows the MST generated from the perturbed version.

Finally, a phenotype (tour) must be developed from the intermediate MST structure. The algorithm Niebel employs to do this is also described in detail in Appendix A. Figure 3 part (c) shows the tour generated from the original problem map. Part (f) shows the tour generated from the perturbed map.

2.2.2.2 Selection

During any given generation, a population goes through the processes of selection for mating, reproduction, reduction, and mutation. Selection involves picking individuals that will be used for mating. The selection is a random process weighted by the fitness of the individual. Thus, individuals whose tours have a short length will have a greater probability of being selected to mate.

2.2.2.3 Reproduction

Once two individuals are selected, they mate to form a third individual which is added to the population. The new individual's city coordinates are generated by averaging the coordinates of each of the cities in both parents and then perturbing them. This results in an offspring that shares traits from both parents.

2.2.2.4 Reduction

After reproduction there is a surplus of individuals due to the new individuals that were generated. Thus, some must be eliminated in order to return the population to its original size. Individuals are selected for survival based on their fitness in a manner similar to the selection for mating.

2.2.2.5 Mutation

Finally, random individuals are mutated. Mutation involves changing the location of a random city in the selected individual, guaranteeing that there are no individuals that cannot be seen.

These five stages are performed during each generation of every population. Since none of the steps involve communication with other populations, the entire evolution step is run in parallel. The main process waits for all of the population processes to signal that they are complete before moving on to the migration stage.

2.2.3 Migration

The migration stage involves transferring individuals between populations as governed by the interconnection matrix. A signal to each population that needs to send individuals by the main process, and the populations respond by packaging an array of the individuals and sending them to the main process. The main process then reroutes the package to the population that is to receive the individuals, without unpackaging them.

No reduction is done until all individuals have been transferred. This allows all of the transfers to be done in parallel, since it does not matter in which order the individuals are received at a given population.

The main process waits until all exchanges have completed, and then sends a signal to each population to recalculate the total fitness, which is necessary to do the reduction. It then sends a signal to each population to reduce to the original number of individuals as explained in the previous section. It then sends a signal for each population to print its current best individual to a file. Since all of these are done independently of other populations, they can all be executed in parallel. All of the processes append to the same file. As a result, since the results will come in out of order due to the parallelization, the populations must write an integer identifying themselves each time they write to a file. The data can be sorted into a more usable order after the program has finished executing.

These steps of evolution, migration, and reduction are carried out for each epoch until the program has completed. The results are then sorted out and compiled by helper programs. It should be easy to modify this software to test several aspects of genetic algorithms.

2.3 Testing

Two major aspects of GAPE were analyzed using the software. First, we analyzed the effect of varying the number of populations on the quality of results produced. We ran a series of tests on the 51 city map displayed in figure 3. The number of epochs was 16 and the number of generations per epoch was 50 for all runs. We varied the number of populations from 2 to 64 by powers of two, keeping the total number of individuals constant. In other words, the number of populations multiplied by the individuals per population remained the same. We also kept the percentage of a population migrating as constant as possible while still maintaining the hypercube

structure. The following table shows all six runs. Each run was executed five times to ensure consistency.

Number of Populations	*Number of Migrants*	*Individuals per Population*
2	576	1280
4	144	640
8	48	320
16	18	160
32	7	80
64	3	40

Table 1: Varying the Number of Populations

A second aspect of gape that was tested was the effect of varying the number of generations per epoch on the solution quality. The number of populations was kept constant at 8, the base population was held at 320, and the number of migrants was held at 48 for all runs. The number of epochs was varied from 1 to 512 by powers of 2, and the total number of generations was held constant at 512. In other words, the product of the number of epochs and number of generations was held constant. The following table shows all 10 runs. Each run was executed five times to ensure consistency.

Number of Epochs	*Number of Generations*
1	512
2	256
4	128
8	64
16	32
32	16
64	8
128	4
256	2
512	1

Table 2: Varying the Generations per Epoch

Finally, we looked at the execution time of the parallel version of GAPE in comparison to the sequential version. We timed the two most time consuming portions of the program, the evolution with an epoch and the communication between populations, and compared the results.

2.4 Equipment

The equipment used in this project was the stonesoup UNIX cluster. It consists of four networked PCs running the Redhat Linux operating system. Future computation and testing will be performed on a larger network allowing further distribution of objects.

3.0 Hypotheses

We hypothesize that small and large numbers of populations may generate better solutions at first, but that the best quality solutions will be generated by more moderate numbers of populations. Further, we feel that runs with few generations per epoch will perform well at first, but that those with moderate numbers of generations per epoch will produce the best solutions in the end. Finally, we feel that the parallelization of the code will produce a moderate speedup running over a four computer network.

3.1 Varying the Number of Populations

The first set of tests involved varying the number of populations while keeping the total number of individuals constant, and keeping the influx of individuals at a set percentage of the population size. At the one extreme, if there were only one population, we would have a traditional genetic algorithm. In this case, we would expect the best individuals to take over the population relatively quickly, and thus for the best solution to be better at than GAPE runs with more populations at the beginning of the run. However, we would anticipate that the GAPE runs, due to their superior diversity, would win out in the end and find the better solution.

At the other extreme, if there were a large number of populations with very few individuals in them, there would be very little diversity within the populations. We would expect that the populations would have more difficulty reaching local minima and that good solutions from other populations could easily take over a population and severely decrease diversity. Although small populations might have the effect of

generating better solutions in the short term, we feel that it will hurt the final solution quality.

Therefore, in the case of varying the number of populations, we expect that the runs with very small numbers of populations will perform best at the beginning, and that the runs with moderate amounts of populations will run the poorest at the start. However, by the end of the run, we feel that the runs with moderate amounts of populations will have generated the best solutions.

3.2 Varying the Generations per Epoch

The second set of tests involved varying the number of generations per epoch while keeping population size, migrants, and total number of generations constant. When there was only one epoch, the run was the equivalent of several one-population GAs running concurrently. We would expect that, even though the results will show the best individual out of all the populations, the separate small GAs will not generate as good results as the GAPE algorithms with moderate number of generations per epoch.

On the other hand, when there is one generation per epoch, there would be communication after every generation. We would expect that the constant communication would allow good individuals to take over many populations, and thus limit diversity. This would cause good tours to be generated at first, but would end up being harmful in the long run. The constant communication would also inhibit populations from moving towards local minima.

Therefore, in the case of varying the number of generations per epoch, we expect that the runs with very few generations per epoch will generate good results at the beginning of the run, but will not end up with the best solution. We expect the runs with

very few epochs to perform poorly throughout. Finally, we expect the runs with a moderate number of generations per epoch to produce the best solutions.

3.3 Parallelizing the Code

We expect that paralellizing the code will produce a substantial speedup for the evolution portion of the code. This is because there is no communication necessary between epochs, and therefore the code should be able to run completely in parallel. We expect that the time to complete the rest of the program will increase slightly, and that all function calls will take somewhat longer due to the added overhead of the Legion system. Finally, we feel that the communication portion of the code will take significantly longer because it now has to send data across a network. Overall, we feel that there will be a moderate speedup even though we are only running on a network of four computers, and that there will be a significant speedup once the new Legion network is installed.

4.0 Results

The experiment with various numbers of populations resulted in very similar results for all numbers of populations. The experiment involving various numbers of generations per epoch showed that those with a large number of epochs did very well at the beginning, but ended up finding worse solutions than those with small numbers of epochs. The parallel code was shown to run in approximately the same time on a four computer network, but with a significant speedup on the evolution.

4.1 Variable Number of Populations

Figure 4 shows the results from the experiments involving a variable number of populations. In these experiments, the number of total individuals was held constant. Thus, as the number of populations increases, the number of individuals per population decreases. All other variables were also held constant, including the percentage of the population that is immigrated at any given epoch.

The results show that the larger number of populations find the best solutions, but the difference between them is very slight. Please note that the y axis represents the tour length, and thus lower values are preferable. The optimal length for this map is known to be 426.

4.2 Variable Number of Generations per Epoch

Figure 5 shows the results from the experiments involving a variable number of generations per epoch. In these experiments, the total number of generations was held

constant, so that as the number of generations per epoch increases, the number of epochs decreases. All other variables were held constant.

Figure 5b shows that there is a clear difference at the beginning of the experiments. The runs with the greatest numbers of epochs, 128 and 256, perform significantly better than the rest of the runs. Figure 5c, however, illustrates that there is a swap, and that the epochs with the least number of epochs end up finding the best solutions in the end.

4.3 Timing of Parallel and Sequential Code

We timed the two most significant portions of the parallel and sequential code against each other on the Appnet cluster of the Department of Computer Science, namely the evolution and the migration. In the original sequential version, almost all of the time was taken up in the evolution stage. We found that the parallel code, when running on a network of four computers, produced a speedup of 2.1 on the evolution portion. However, the migration ended up taking significantly longer than the original, and the code ended up running in approximately the same time.

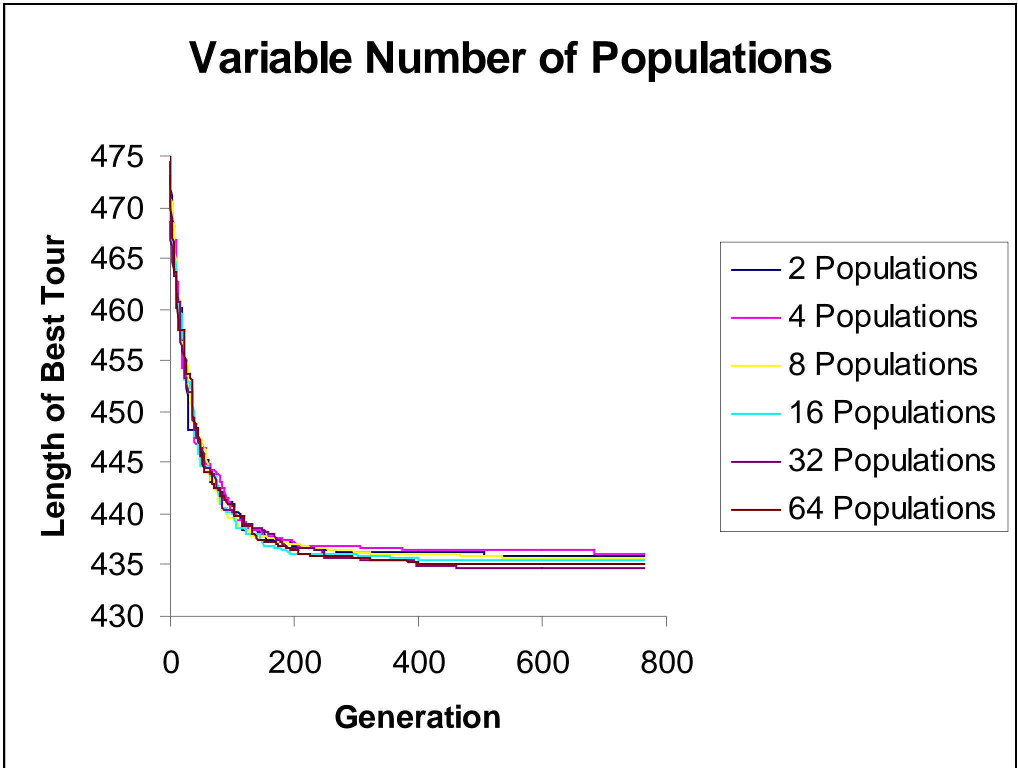


Figure 4: Runs of Various Numbers of Populations
Created by Dan Ignat and Justin Turner, March 1998.

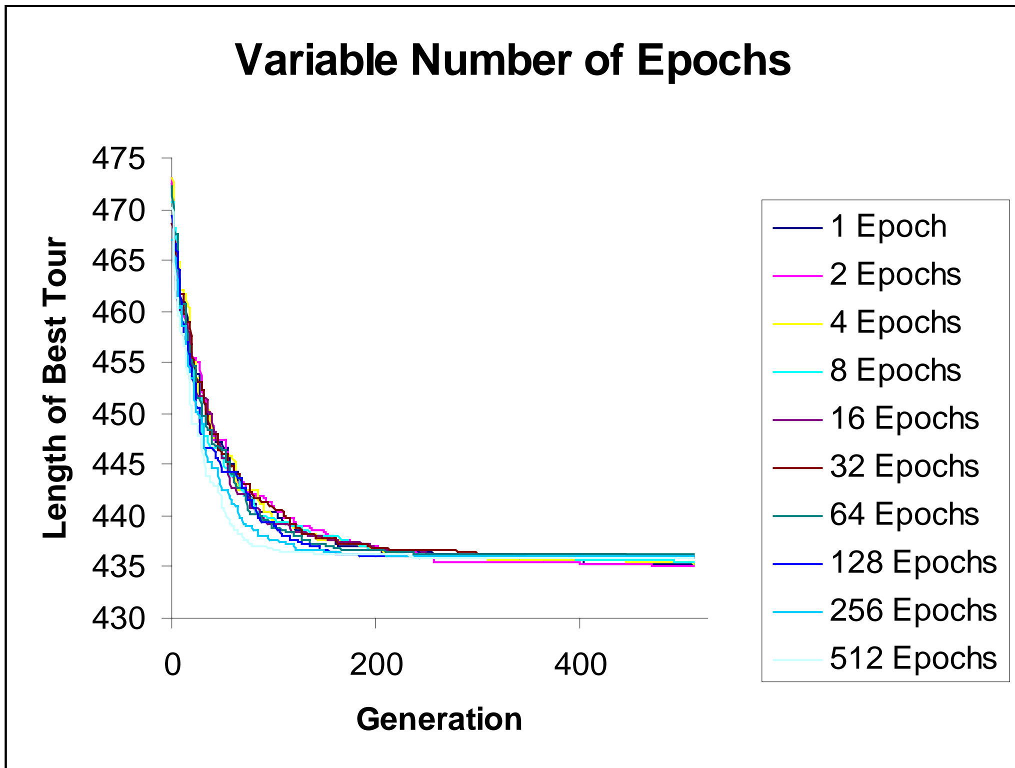


Figure 5a: Runs of Various Numbers of Epochs
Created by Dan Ignat and Justin Turner, March 1998.

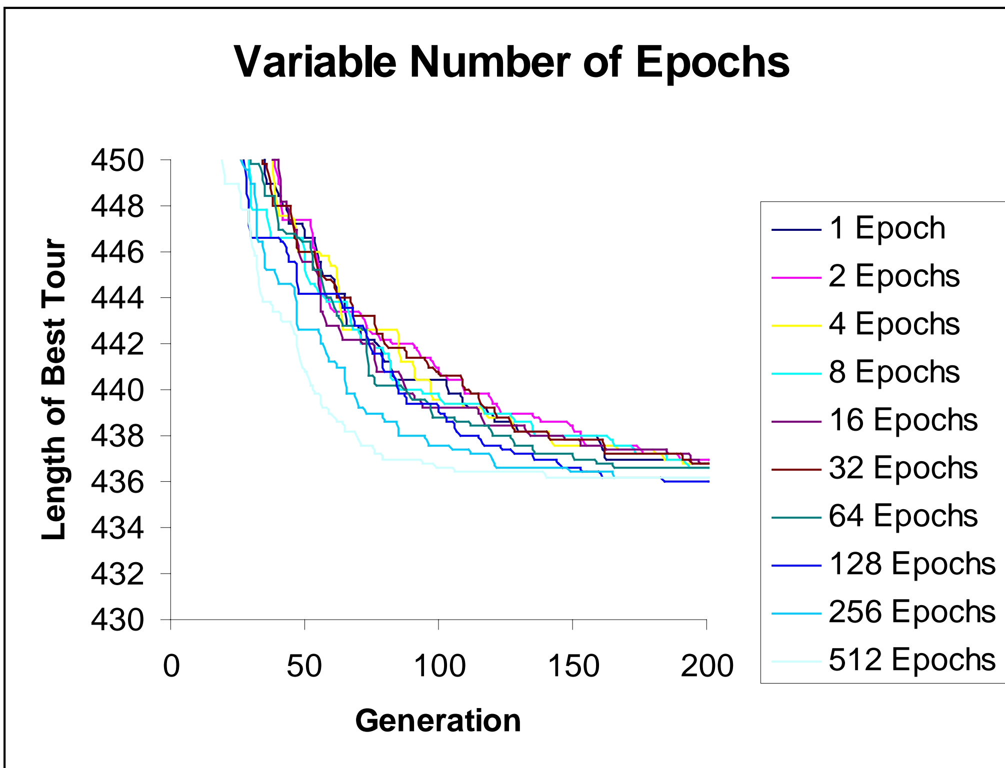


Figure 5b: Closeup of First Half
Created by Dan Ignat and Justin Turner, March 1998.

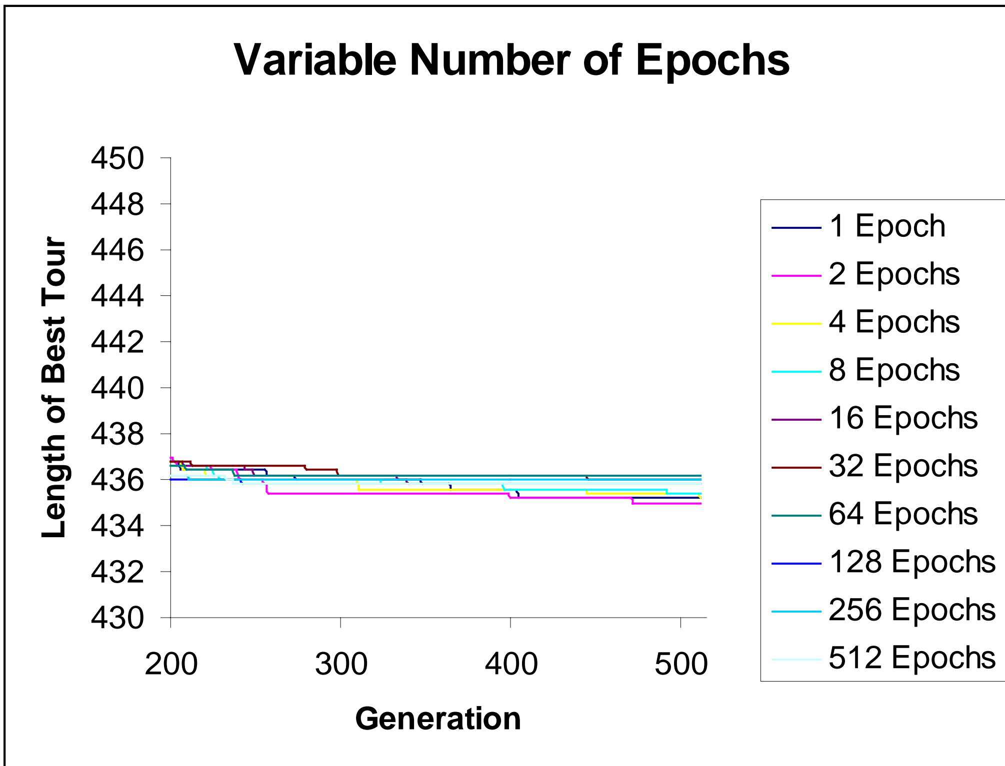


Figure 5c: Closeup of Second Half
Created by Dan Ignat and Justin Turner, March 1998.

5.0 Conclusions

The results explained within this paper indicate that a parallel implementation of GAPE is only somewhat helpful with the current version of Legion, but that it has the potential to produce a very high speedup as Legion improves and better networks are installed. Varying the number of populations had little effect, but this may be because of the small number of cities in the problem instance. The results obtained from varying the number of generations per epoch seem to indicate that intermingling should be done infrequently in order to preserve diversity and allow populations to seek out local minima.

5.1 Summary

Genetic Algorithms with Punctuated Equilibria provide a means of generating approximate solutions to problems. This can be valuable because there are many problems for which a perfect solution cannot be found in a reasonable amount of time once the problem instance grows to a large size.

One aspect of GAPE that is highly appealing is that it is parallelizable, due to the fact that its populations run independently of each other for extended periods of time. Using a network of four computers, a speedup of 2.1 was obtained on the evolution, which is the most time consuming aspect of the sequential version. The slow down in communication speed canceled out the benefit of this, but some reasons why this happened will be explained below.

The results from the test in which the number of populations varied showed no noticeable correlation between the number of populations and solution quality. The results from the experiment where the generations per epoch varied, however, showed a

clear pattern of the runs with the largest number of epochs starting off best, and the runs with the smallest number of populations producing the best final results.

5.2 Interpretation

Using a network of four computers, we obtained a speedup of 2.1 on the evolution portion of the software. The maximum we could have theoretically obtained was 4, if the processes were distributed perfectly and there were no overhead. The major reason that the speedup is low is that Legion has not yet implemented an algorithm for distributing processes among computers. At present, it randomly selects the computer to which to send each process. As a result, we have some processes finishing evolution much faster than others.

The other major problem with the parallel code was the communication speed, which is partially due to a bug in the Legion system. During communication, when immigrants are transferred from one population to another, the system does not recognize the direct transfer from population to population and instead sends the individuals through the main process, doubling the time it takes. As Legion improves, and as larger networks are set up, we predict that the performance of this parallel implementation will improve dramatically, especially for runs with large numbers of populations.

The results from varying the number of populations showed no correlation between the number of populations and solution quality. We believe, however, that this may change for a map of a larger size. The problem we see is that the runs are all getting very close to the optimal solution, and thus there is not much room to distinguish between them. Further tests will be done to determine whether this assessment is accurate.

The results from varying the number of generations per epoch show that the populations with large numbers of epochs, and thus small numbers of generations, do very well at the beginning of the run. These results are what we expected, because the massive amount of communication causes the best individuals to get spread out over many of the populations, which causes the algorithm to find their local minima very quickly. However, the communication also lessens the diversity of the populations significantly. This results in them performing poorly in the long run. The results show that the populations with 512 and 256 epochs clearly start off best, end up among the worst, while the runs with long epochs start out mediocre but end up finding the best solutions. These findings indicate that it is important to let populations evolve and reach their local minima.

5.3 Recommendations

The work that will be done in the immediate future on this project includes analyzing the effects of modifying the interconnection matrix and rerunning the tests contained within this document using problem maps of a larger size to try to obtain better defined results.

Several other aspects of GAPE might be interesting to test. These include experimenting with interconnection matrices that are well-suited to a parallel distribution. For instance, having populations communicate mainly with other populations which are on the same computer as them, or are close by, might produce better results. Other possible tests include modifying the mutation rate and varying the fitness function across populations.

Works Cited

[1] J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. S. Richards, “Punctuated Equilibria: A Parallel Genetic Algorithm,” in *Proceedings of the Second International Conference on Genetic Algorithms*, 1987, pp. 148-154.

[2] J. P. Cohoon, S. U. Hedge, W. N. Martin and D. S. Richards, “Distributed Genetic Algorithms for the Floorplan Design Problem,” in *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 4, pp. 483-492, April 1991.

[3] J. P. Cohoon, W. N. Martin, and D. S. Richards, “Genetic Algorithms and Punctuated Equilibria in VLSI,” in *Parallel Problem Solving from Nature*, H. P. Schwefel and R. Männer, eds., Lecture Notes in Computer Science, vol. 496, Berlin: Springer Verlag, 1991, pp. 134-144.

[4] N. Eldredge and S. J. Gould, “Punctuated Equilibria: An Alternative to Phyletic Gradualism,” in *Models of Paleobiology*, T. J. M. Schopf, Ed. San Francisco: CA, Freeman, Cooper and Co., 1972, pp. 82-115.

[5] N. Eldredge, *Time Frames*. New York: Simon and Schuster, 1985.

[6] A. S. Grimshaw et al., *Mentat Homepage*: “http://www.cs.virginia.edu/~mentat/”, University of Virginia Computer Science Department.

[7] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver and P. F. Reynolds, Jr., *A Synopsis of the Legion Project*. University of Virginia Computer Science Technical Report No. CS-94-20: June, 1994.

[8] D. Hartl, E. Jones, *Genetics: Principles and Analysis.* Sudbury, MA: Jones and Bartlett Publishers, 1998.

[9] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. United States of America: Computer Science Press, Inc., 1978, pp. 501-558.

[10] M. Lewis and A. S. Grimshaw, *The Core Legion Object Model*. University of Virginia Computer Science Technical Report No. CS-95-35: August, 1995.

[11] J. Lienig, “A Parallel Genetic Algorithm for Performance-Driven VLSI Routing,” in *IEEE Transactions on Evolutionary Computation*. New York: IEEE, 1997.

[12] J. Lienig and J. P. Cohoon, “Evolutionary Algorithms Applied to the Physical Design of VLSI Circuits,” in *Handbook of Evolutionary Computation*. Oxford: Oxford University Press, 1997, ch. G3.10, pp. 1-8.

[13] W. N. Martin, personal communication.

[14] W. Niebel, “Genetic Algorithm / Punctuated Equilibria Applied to the Traveling Salesperson Problem with Problem Deformation / Perturbation,” University of Virginia Computer Science Technical Report: February, 1996.

[15] L. M. Rocha, *Evolutionary Systems and Artificial Life.* Los Alamos, NM, 1995.

[16] M. Srinivas and Lalit M. Patnaik, “Genetic Algorithms: A Survey,” in *Proceedings of the IEEE*. New York: IEEE, 1994, pp. 17-26.

[17] T. Wong, "Applications of Genetic Algorithms," in *SURPRISE 96 Journal.* London: Imperial College Department of Computing, 1996.

[18] H. Wilf, *Algorithms and Complexity*. New Jersey: Prentice-Hall, Inc., 1986.

[19] G.Winter, J.Periaux & M.Galan, *Genetic Algorithms in Engineering and Computer Science.* New York: John Wiley and Son Ltd., 1995.

Appendix A: Annotated Reference Manual

`class Runparms`

`void Runparms::read()`

- Reads and processes the parameter file specified as the second argument in the executable's command line
- Stores constants specified in parameter file into the class' member variables

`char *Runparms::optTourFileName()`

- Returns a string containing the name of the file describing the "best" published tour for the current TSP instance

`char *Runparms::mapFileName()`

- Returns a string containing the name of the file describing the map of the current TSP instance

`char *Runparms::connectionFileName()`

- Returns a string containing the name of the file describing the interconnection topology matrix for the current GAPE run

---

`class Net`

`Net::Net()`

- Allocates and initializes an array of populations

`Net::~Net()`

- Deallocates the array of populations

`void Net::read()`

- Reads interconnection topology matrix file into an array and verifies its integrity

`void Net::run()`

- For each epoch
  - If in epoch 0
    - calls **`Population::show()`** on each population
  - Else
    - calls **`Population::selfSource()`** on each population
    - exchanges individuals between populations based on interconnection array
    - calls **`Population::recalcFitness()`** on each population (required by migration)
    - calls **`Population::reduce()`** on each population
    - calls **`Population::recalcFitness()`** on each population once again

    - calls **`Population::show()`** on each population
  - Calls **`Population::newEpoch()`** on each population

`stateful mentat class Population`

`Population::Population()`

- Opens various output files for parallel writing across all Population instances
- Creates a local copy of the TSP instance
- Allocates and initializes an array of individuals
- Initializes member variables with values from parameters
- Calls **`Population::recalcFitness()`** on itself

`int Population::selectToMate()`

- Randomly selects a first parent using prowess as a weight

`int Population::selectToMate2()`

- Randomly selects a second parent using prowess as a weight and making sure not to duplicate first parent

`int Population::selectToSurvive()`

- Randomly selects an individual to survive using longevity as a weight

`int Population::reproduce()`

- Reproduces individuals
  - Selects parents
  - Generates offspring
  - Keeps count on offspring improving over neither parent, one parent, or both parents

`int Population::mutate()`

- Mutates individuals
  - Selects mutant
  - Calls **`Individual::mutate()`** on mutant individual
  - Calls **`Individual::develop()`** on mutant individual
  - Keeps count on mutants improving, worsening, or remaining the same

`int Population::newGen()`

- Processes a new generation
  - Calls **`Population::selfSource()`** on itself
  - If **`Population::totalProwess`** > 0
    - calls **`Population::reproduce()`** on itself
    - calls **`Population::recalcFitness()`** on itself
  - If the population has swelled beyond its maximum size
    - calls **`Population::reduce()`** on itself
    - calls **`Population::recalcFitness()`** on itself

int Population::newEpoch()

- Processes a new epoch
  - For each generation
    - calls **Population::newGen()** on itself
    - calls **Population::show()** on itself

int Population::reduce()

- Uses **Population::selectToSurvive()** to select individuals for survival into the next generation
- Rearranges array of individuals so that the scattered survivors are all at the beginning

int Population::selfSource()

- Marks all individuals in the base population as being carried over from the previous generation, as opposed to being the result of crossover, mutation, or migration

int Population::recalcFitness()

- Calculates various population statistics
- Calculates fitness and longevity for each individual

IndividualArray Population::sendEmmigrants()

- Returns an array of individuals selected from the population to be emmigrants

int Population::receiveImmigrants()

- Takes an array of individuals (immigrants) as a parameter
- Adds the immigrants to the populations array of individuals
- Calls **Individual::develop()** on each immigrant

int Population::show()

- Prints out various population statistics

---

class Individual

Individual::Individual()

- Initializes various member variables
- Calls **Individual::reset()** on itself

void Individual::reset()

- Resets the individual's **map**
- Calls **Individual::develop()** on itself

void Individual::scatter()

- Calls **City::scatter()** on each of the cities on the individual's **map**
- Sets its **source** to **INITIAL** (i.e., individual is not the result of crossover, mutation, or migration)

`void Individual::mutate()`

- Calls **`City::mutate()`** on a random city on the individual's **`map`**
- Sets its **`source`** to **`MUTATION`**
- Calls **`Individual::develop()`** on itself
- Creates graph of and prints the mutation vector

`void Individual::mstify()`

- Creates an **`mst`** (the intermediate structure used in the conversion from genotype to phenotype) from the individual's **`map`**

`void Individual::opt2()`

- Creates a **`tour`** from the individual's **`mst`** (the intermediate structure used in the conversion from genotype to phenotype)
  - For each vertex in the individual's **`mst`**, create a list of vertices connected to it
  - Start with **`origin`** city and traverse first edge of mst
  - From each new node visited, traverse the untraversed edge at greatest clockwise angle relative to the previously traversed edge
  - If current node is a leaf or no untraversed edges remain, backtrack to previous node
  - The **`tour`** is constructed by connecting each newly visited node and then connecting the last node back to the **`origin`**

`void Individual::develop()`

- Calls **`Individual::mstify()`** on itself
- Sets the **`origin`** of the individual's **`map`** to the first leaf
- Calls **`Individual::opt2()`** on itself
- Sets its **`objective`** to the length of the newly-generated **`tour`**

`void Individual::xover()`

- Calls **`City::xover()`** on each city in the individual's **`map`**
- Sets its **`source`** to **`XOVER`**
- Calls **`Individual::develop()`** on itself
- Creates graph of and prints the crossover vectors

`class CitySet`

`double CitySet::floatDistance()`

- Returns the distance between two cities passed as parameters

`int CitySet::distance()`

- Returns the distance between two cities passed as parameters rounded off to the nearest integer

`void CitySet::add()`
- Adds to itself the city whose characteristics are passed as parameters

`int CitySet::label()`
- Returns the label of the city passed as a parameter

`void CitySet::read()`
- Reads in the original TSP map from a file
  - Reads in preamble, which contains information about the rest of the map file
  - Reads in the city coordinates
- Calculates standard deviation of inter-city distances

`class City`

`void City::perturb()`
- Perturbs its **`x`** and **`y`** coordinates using the **`StdDev`** passed as a parameter

`void City::scatter()`
- Calls **`City::perturb()`** with **`city_scatterStdDev`**

`void City::mutate()`
- Calls **`City::perturb()`** with **`city_mutationStdDev`**

`void City::xover()`
- Sets its **`x`** and **`y`** coordinates based on the corresponding means of the **`x`** and **`y`** coordinates of the two cities passed in as parameters, and then offset using **`city_xoverStdDev`**

`class Tour`

`void Tour::reset()`
- Resets member variables

`int Tour::visited()`
- Returns whether or not the city passed in as a parameter has been visited

`void Tour::visit()`
- Visits the city passed in as a parameter and adds it to the tour

`int Tour::findObjective()`
- Returns the length of the tour

`void Tour::read()`
- Reads in a tour from a file

- Reads in preamble, which contains information about the rest of the tour file
- Reads in the cities and visits each one as it is read in

`class EdgeSet`

`EdgeSet::EdgeSet()`
- Calls **`EdgeSet::reset()`** on the **`EdgeSet`**

`void EdgeSet::reset()`
- Reinitializes member variables

`void EdgeSet::add()`
- Adds the edge passed in as a parameter to the **`EdgeSet`**

`Edge EdgeSet::remove()`
- Returns the edge specified by the parameter
  - Swaps the edge specified by the parameter with the last edge in the **`EdgeSet`** in order to keep it around for later display

`int EdgeSet::firstLeaf()`
- Returns the first vertex of degree 1 (i.e., the first leaf)

`void EdgeSet::iterator()`
- Reinitializes the iterator of the **`EdgeSet`**
  - Fills **`EdgeSet::sequence[]`** with randomly ordered vertices from the **`EdgeSet`**
  - Reinitializes **`EdgeSet::sequencer`**

`int EdgeSet::nextEdge()`
- Returns the next edge from the iterator

`class EdgeNode`

`void EdgeNode::refresh()`
- Sets the vertices of **`EdgeNode::edge`** to the two cities passed in as parameters
- Sets **`EdgeNode::cost`** to the distance between the two cities passed in as parameters

---

`class EdgeQueue`

`EdgeQueue::EdgeQueue()`
- Creates an **`EdgeNode`** for each edge from one city to any other city and adds it to **`EdgeQueue::edgenodes`**

- Heapifies **`EdgeQueue::edgenodes`**

`void EdgeQueue::heapify()`
- Recursively converts **`EdgeQueue::edgenodes`** into a heap sorted by cost

`Edge EdgeQueue::remove()`
- Returns an **`Edge`** removed from **`EdgeQueue::edgenodes`** and reheapifies the rest of the heap

---

`class Edge`

`class IncidentNode`

`IncidentNode *IncidentNode::checkout()`
- Returns the first removed **`IncidentNode`** from the linked list pointed to by **`IncidentNode::first`**

`void IncidentNode::checkin()`
- Adds the **`IncidentNode`** passed in as a parameter to the top of the linked list pointed to by **`IncidentNode::first`**

`class Range`

`class StdDev`

`void StdDev::load()`
- Sets **`StdDev::x`** and **`StdDev::y`** to the two parameters passed in, respectively

---

`class VertexPartition`

`void VertexPartition::VertexPartition()`
- Sets **`VertexPartition::nVertices`** to the parameter passed in
- Fills **`VertexPartition::sets[]`** with numbers from **`0`** to **`VertexPartition::nVertices - 1`** (i.e., each vertex belongs only to its own set in the partition at first)
- Sets **`VertexPartition::kard`** to **`VertexPartition::nVertices`**

`int VertexPartition::card()`
- Returns **`VertexPartition::kard`**

`int VertexPartition::setContaining()`

- Returns the number of the set which contains the vertex passed in as a parameter

`void VertexPartition::combine()`
- Combines the first vertex set passed in as a parameter and the second vertex set passed in as a parameter
  - Replaces all occurrences of the first vertex set passed in as a parameter in **`VertexPartition::set[]`** with the second vertex set passed in as a parameter
- Decrements **`VertexPartition::kard`**

`class IndividualArray`

`IndividualArray::IndividualArray()`
- Sets **`IndividualArray::length`** to **`0`**

`int IndividualArray::add()`
- Adds the **`Individual`** passed in as a parameter to the **`IndividualArray`**

`class GapeString`

`GapeString::GapeString()`
- Copies the string passed in as a parameter to **`GapeString::string`**

`char *GapeString::getString()`
- Returns **`GapeString::string`**