

**GENETIC ALGORITHM WITH PUNCTUATED EQUILIBRIA:
ANALYSIS OF THE TRAVELLING SALESPERSON PROBLEM INSTANCE**

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

Daniel Bogdan Ignat

March 24, 1998

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 TCC Courses.

Approved _____ (Technical Advisor)

Professor Worthy N. Martin

Approved _____ (TCC Advisor)

Professor Patricia C. Click

Preface

Message to the Reader

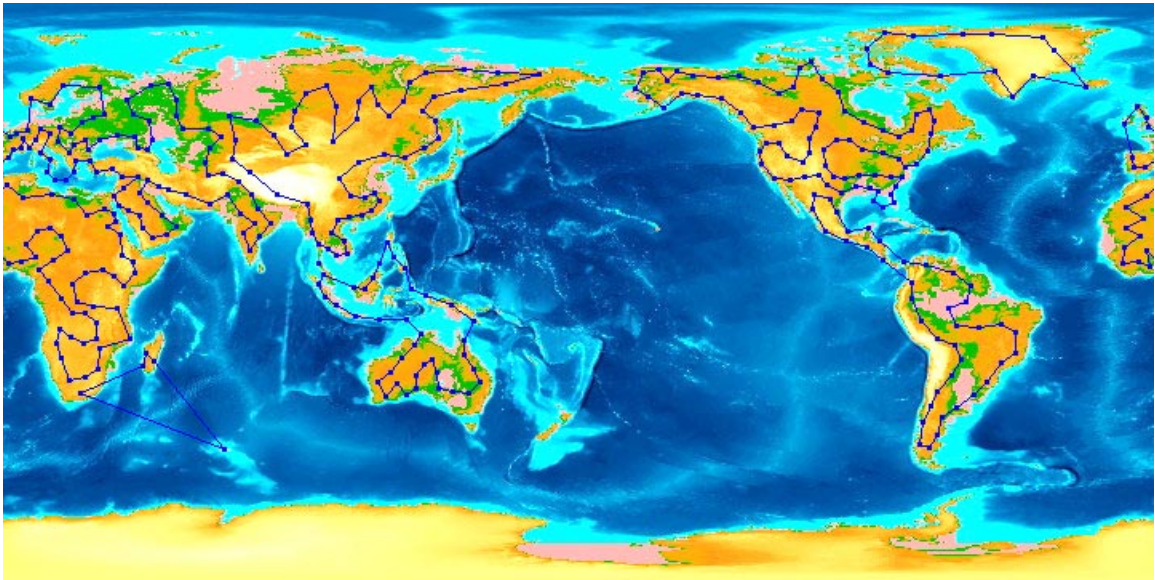
This technical report presents the results of a research endeavor using tools that are still in the developmental and experimental stages. This fact means that the experiments that were actually run do not represent as extensive a statistical base as the author would have liked. As a result, the results presented here are limited in their scope and should not be taken as the crux of the entire project. This project will continue beyond the deadline for the technical report and more results may be added as they become available. However, at the moment, the only results that can be presented are those which were compiled despite the limitations imposed on the project by the tools it used.

Acknowledgements

The author would like to thank his Technical Advisor, Professor Worthy N. Martin, of the Department of Computer Science, for his invaluable contributions to this research project. Professor Martin was a source of information regarding the state of the art of the field of genetic algorithms. He also helped out in many other respects, often providing the project with direction when it became unclear how to make the best use of the limited resources available.

The author would also like to thank his TCC Advisor, Professor Patricia C. Click, of the Department of Technology, Culture, and Communication, for her help with the preparation of this technical report, for which she provided valuable advice and guidelines.

The other people whom the author would like to thank are the members of the Legion Project in the Department of Computer Science who provided technical support for the Legion system whenever it was necessary. They include Mark R. Hyett, John F. Karpovich, Adam J. Ferrari, Greg B. Lindahl, and Norman F. Beekwilder.



Frontispiece. The Travelling Salesperson Problem, which simply involves finding a tour of minimum length for a given set of points in the Cartesian plane, is NP-Complete. The difficulty of finding solutions to such problems in real time, as well as their versatility of application, makes a heuristic such as the Genetic Algorithm with Punctuated Equilibria one of the most formidable tools available in the search for optimal solutions.

Table Of Contents

PREFACE	I
MESSAGE TO THE READER	I
ACKNOWLEDGEMENTS	I
TABLE OF FIGURES.....	IV
GLOSSARY OF TERMS.....	V
ABSTRACT	VIII
CHAPTER I: INTRODUCTION	1
A. PROBLEM DEFINITION.....	1
1. <i>Concepts</i>	1
a. NP-Complete Problems [8].....	1
b. Travelling Salesperson Problem	2
c. Genetic Algorithms	2
d. Genetic Algorithms with Punctuated Equilibria	3
2. <i>Context</i>	3
B. REVIEW OF RELEVANT LITERATURE	4
C. RATIONALE AND OBJECTIVES	6
D. OVERVIEW OF TECHNICAL REPORT	7
CHAPTER II: METHODOLOGY	8
A. THE LEGION PROJECT AND MPL	8
B. GAPE SOFTWARE AND ISSUES	9
1. <i>Evolution</i>	9
a. Crossover and Niebel’s Underlying Algorithms	9
2. <i>Migration</i>	12
C. EXPERIMENTATION TECHNIQUES	12
CHAPTER III: HYPOTHESES.....	14
A. VARIABLE NUMBER OF POPULATIONS	14
B. VARIABLE NUMBER OF EPOCHS.....	14
C. PARALLELIZATION AND EXECUTION SPEED	14
CHAPTER IV: RESULTS.....	16
A. VARIABLE NUMBER OF POPULATIONS	16
B. VARIABLE NUMBER OF EPOCHS.....	16
C. PARALLELIZATION AND EXECUTION SPEED	19
CHAPTER V: CONCLUSION.....	20
A. FACTUAL SUMMARY.....	20
B. INTERPRETATIONS OF RESULTS.....	21
C. RECOMMENDATIONS FOR FUTURE STUDY.....	22
REFERENCES	23
APPENDIX A: ANNOTATED REFERENCE MANUAL.....	25

Table of Figures

FRONTISPIECE.....	II
FIGURE 1: HYPERCUBES OF N DIMENSIONS.....	4
FIGURE 2: EILON-CHRISTOFIDES 51-CITY TSP.....	11
FIGURE 3: VARIABLE NUMBER OF POPULATIONS.....	17
FIGURE 4: VARIABLE NUMBER OF EPOCHS.....	17
FIGURE 5: VARIABLE NUMBER OF EPOCHS.....	18
FIGURE 6: VARIABLE NUMBER OF EPOCHS.....	18

Glossary of Terms

Deterministic Algorithm

- an algorithm for which every operation is uniquely defined

Formula

- an expression composed of literals, conjunctions (*and*), and disjunctions (*or*)

GA

- see **Genetic Algorithm**

GAPE

- see **Genetic Algorithm with Punctuated Equilibria**

Genetic Algorithm

- an algorithm which approaches a solution to a given NP-complete problem by repeatedly taking a population of existing individuals related to that problem, performing operations on them to produce offspring individuals, and then selecting only certain individuals from the newly expanded population to survive; it is hoped that the population gradually evolves to include individuals which are good approximations to the optimal solution

Genetic Algorithm with Punctuated Equilibria

- a GA that has multiple populations evolving independently of one another and periodically exchanging individuals in order to avoid settling for local optima in the search for the optimal solution to a given NP-Complete problem

Heuristic Search

- the search for approximately optimal solutions to computationally complex problems

Hypercube

- an n -dimensional cube with 2^n nodes such that each node has n bi-directional links to adjacent nodes; this topology is often used to connect populations in a GAPE environment for the purpose of inter-epoch migration

Legion

- an ongoing project at the Department of Computer Science which is attempting to construct a worldwide virtual computer which can make efficient use of the pooled resources inherent in a distributed network of workstations by managing the parallel computation of programs written in MPL

Map

- a set of nodes in the Cartesian plane

Mentat Programming Language

- an extension of the C++ programming language which is intended to provide programmers with the ability to create objects for parallel computation

MPL

- see **Mentat Programming Language**

Nondeterministic Algorithm

- an algorithm which contains operations whose outcomes are not uniquely defined, yet are limited to a specific set of possibilities; when executing such operations, a decision may be made for any of these outcomes, subject to some termination condition

NP

- the set of all decision problems solvable by a nondeterministic polynomial time algorithm

NP-Complete

- the problem is NP-hard and is also in the set NP

NP-Hard

- Satisfiability reduces to the given problem

P

- the set of all decision problems solvable by a deterministic polynomial time algorithm

Polynomial Time Algorithm

- an algorithm solving a problem of polynomial complexity in $O(p(n))$ time, $p(x)$ being a polynomial, n being the problem size

Polynomially Complex Problem

- there exists a polynomial $p(x)$ such that the computing time for the problem is $O(p(n))$, n being the problem size

Problem Deformation/Perturbation

- a technique involving the perturbation of a TSP map in order to exploit obscure tours which are not easily chosen otherwise; this technique hopes that obvious tours on the perturbed map will turn out to be obscure tours when evaluated on the original map

Reduction

- a problem L_1 *reduces* to a problem L_2 if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm which solves L_2

Satisfiability Problem

- determine whether there exists some assignment of truth values to its variables which makes a formula true

Travelling Salesperson Problem

- given a set of nodes in the Cartesian plane, find a path of minimum length starting from one node, passing once through each of the other nodes, and returning to the node of origin; the Travelling Salesperson Problem is NP-Complete

TSP

- see **Travelling Salesperson Problem**

Abstract

The relatively new field of Genetic Algorithms with Punctuated Equilibria (*GAPE*) has yet to completely prove itself to the computing community. Due to the length of the running times of the existing software implementations of *GAPE*, quite a bit of analysis remains to be done in order to assess the value of its contributions to the field of heuristic search. This project attempted to probe further into the analysis of *GAPE* on the Travelling Salesperson Problem (*TSP*) by taking an existing sequential *GAPE* implementation and converting it into a parallel version to be run on a Legion distributed network. This would allow extended experimentation which would not be possible with a sequential implementation due to the computing resources it would require of one machine.

The results of our experimentation revealed that a variable number of populations has little effect on *GAPE*'s performance, at least for the Eilon-Christofides 51-city *TSP*. However, experimenting with a variable number of epochs revealed that instances with many, small epochs tend to find better solutions more quickly at first, while instances with a few, large epochs tend to be slower at the onset but also tend to find better solutions overall. Furthermore, we experienced a speedup of about two as compared to the previous sequential version during the evolution phase, but we also experienced very significant communication costs during the migration phase, which had the effect of offsetting the aforementioned speedup.

This technical report presents the background, methodology, hypotheses, results, and conclusions of this research project, in the hope that future study along these lines will benefit from its existence.

Chapter I: Introduction

The Genetic Algorithm with Punctuated Equilibria (*GAPE*), which is based on evolution theory, is among the most promising new variants in the field of heuristic search, the search for approximately optimal solutions to computationally complex problems. Because the existing sequential software implementations of GAPE leave many questions unanswered due to the length of time required to carry out the relevant experiments, this project attempted to probe further into GAPE analysis by using a parallel software implementation.

This project was undertaken within a group structure, and thus the author would like to acknowledge the help of Justin L. Turner, an undergraduate Rodman Scholar in the Department of Computer Science, for his part in this endeavor. Mr. Turner shared in the implementation of the software used in this project, as well as in the development of certain common parts of this technical report (i.e., figures and appendices). Any future uses of the second person in this technical report should be understood as referring to this group structure.

A. Problem Definition

1. Concepts

a. NP-Complete Problems [8]

Let x_1, x_2, \dots, x_n , be Boolean variables. A *literal* is either a variable or its negation. A *formula* in the propositional calculus is an expression composed of literals, conjunctions (*and*), and disjunctions (*or*). The problem of *satisfiability* is to determine whether there exists some assignment of truth values to its variables which makes a formula true.

A *deterministic algorithm* is an algorithm for which every operation is uniquely defined. A *nondeterministic algorithm* is an algorithm which contains operations whose outcomes are not uniquely defined, yet are limited to a specific set of possibilities. When executing such operations, a decision may be made for any of these outcomes, subject to some termination condition.

A problem is of *polynomial complexity* if there exists a polynomial $p(x)$ such that the computing time for the problem is $O(p(n))$, n being the problem size. An algorithm solving such a problem in $O(p(n))$ time is called a *polynomial time algorithm*.

A problem L_1 *reduces* to a problem L_2 if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm which solves L_2 .

P is the set of all decision problems solvable by a deterministic polynomial time algorithm. NP is the set of all decision problems solvable by a nondeterministic polynomial time algorithm.

A problem is *NP-hard* if and only if satisfiability reduces to it. A problem is *NP-complete* if and only if it is NP-hard and is also in the set NP.

b. Travelling Salesperson Problem

The Travelling Salesperson Problem (*TSP*) is an NP-complete problem. Given a set of nodes (*map*) in the Cartesian plane, the TSP is to find a path of minimum length starting from one node, passing once through each of the other nodes, and returning to the node of origin (*tour*).

c. Genetic Algorithms

A Genetic Algorithm (*GA*) is an algorithm which approaches a solution to a given NP-complete problem by taking a set of existing individuals (*parents*) and performing operations (*crossovers*) on them to produce a new set of individuals (*offspring*). The population is thus temporarily increased, so *selection* takes place over the entire population (both parents and offspring), and only a certain number of individuals continue into the next *generation*, as evaluated by an objective *fitness function*. Although randomness plays a large role in order to avoid stagnation in the population's evolution, ideally the offspring should eventually become "better" (i.e., closer to the optimal individual, as evaluated by the aforementioned fitness function) than their predecessors were, in a way similar to the evolution of biological species. GAs may also emulate other aspects of biological evolution, such as *mutation*.

The individuals spoken of in the preceding paragraph may be represented by several different entities. The only requirement is that an individual should be an entity relevant to finding a solution to the problem that the GA is attempting to solve, preferably

an entity which easily lends itself to crossover. For example, in the case where the GA is attempting to solve the TSP, one might think that an individual should be an actual tour (i.e., a potential solution). However, research has shown that tours do not easily nor effectively lend themselves to crossover in order to produce offspring tours. As a result, other representations of individuals are used. For this project, a technique known as *problem deformation/perturbation* was used in order to solve the crossover dilemma, as will be explained later.

d. Genetic Algorithms with Punctuated Equilibria

GAPE uses the standard GA described above and adds a twist to it: *migration*. That is, multiple populations are allowed to develop independently of one another, emulating biological systems where natural barriers partition the global population and isolate each subpopulation from the others. Then, a certain number of individuals are exchanged between populations once every certain number of generations (*epoch*). Theoretically, this reduces the chance of settling for local optima in the search for the optimal solution to the given problem. Populations are “connected” to one another for the purpose of migration through some kind of graph (*interconnection topology*), often a *hypercube*. A hypercube is an n -dimensional cube with 2^n nodes such that each node has n bi-directional links to adjacent nodes. **Figure 1** shows some sample hypercubes of different dimensions. If each node of a hypercube represents a population, then each population has n populations adjacent to it with which it exchanges individuals once per epoch, during migration.

2. Context

NP-complete problems are extremely difficult to solve in real-time for any instances except the most trivial ones. Although it has yet to be proven that these problems are unsolvable in polynomial time, there is much evidence that seems to suggest this. In other words, it is very likely that the time it takes to find the optimal solution cannot be expressed by a polynomial function of the problem size, but may rather have to be expressed by an exponential or even faster-growing function of the problem size. Nonetheless, NP-complete problems are some of the most rewarding to solve. Among their ranks are VLSI (Very Large Scale Integration) design and the TSP,

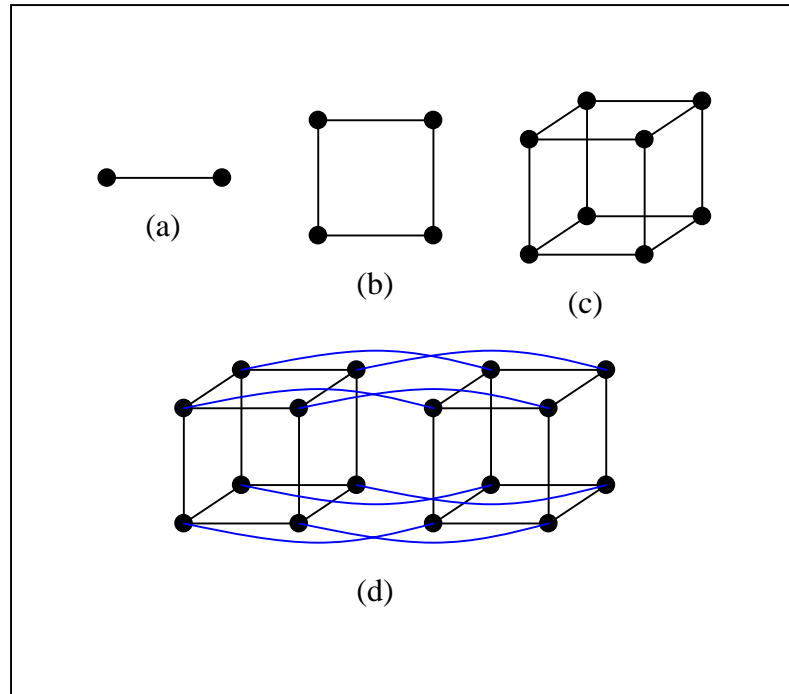


Figure 1: Hypercubes of n dimensions. (a) $n = 1$ (b) $n = 2$ (c) $n = 3$ (d) $n = 4$

which is a problem undertaken by many enterprises on a day-to-day basis, including postal services and delivery-based firms, in the endeavor of optimal path-planning.

The GAPE concept is relatively new, with its origins at the University of Virginia in 1987. It was originally applied to the floorplan design problem and, consequently, has many applications to the field of VLSI design. The specific NP-complete problem to which GAPE is applied is limited neither to VLSI design nor to the TSP, as long as parent individuals for the given problem can be crossed-over effectively to produce offspring individuals. GAPE is also certainly worthy of study even for the rather theoretical TSP, some of whose applications were mentioned above.

B. Review of Relevant Literature

The genetic algorithm with punctuated equilibria (GAPE) is a fairly new variant of the genetic algorithm (GA). The research attention it has received so far has not been extensive, and it is still regarded with skepticism by many because it remains unproven in many areas of application. The areas it has thus far touched upon, including more practical ones, such as VLSI design, as well as more theoretical ones, such as the TSP,

have supported some theories that GAPE may prove successful yet. Consequently, a study of further performance characteristics of GAPE is in order.

Srinivas and Patnaik provide a fairly comprehensive analysis of the basic characteristics and issues surrounding the development of GAs [14]. Some of these issues, such as how to deal with the dilemma of settling for local minima, were pivotal to the arrival of the GAPE concept.

Cohon, Hedge, Martin, and Richards introduced GAPE in 1987 [1], defining the algorithm and the characteristics which set it apart from the standard GA. The initial concept of punctuated equilibria, which set the stage for GAPE, arose out of a need to explain certain paleontological dilemmas in the geological record [4], [5]. The two main principles of punctuated equilibria are *allopatric speciation* and *stasis*. Allopatric speciation states that a new species will quickly develop if a few members of an existing species are isolated into a new environment. Stasis is simply a notion of lack of change.

One of the first applications came in 1991 with the aforementioned group's work on the Floorplan Design Problem [2]. Floorplan design is one of the most significant steps of VLSI design, involving the minimization of the weighted sum of area and wirelength measures for a given set of modules to be arranged on a chip. This was the first use of distributed genetic algorithms, and it was proven successful when it outperformed the simulated annealing approach in terms of both the optimal solution found and the average cost. However, this was only one possible application of GAPE, and others were still waiting to be found and experimented with.

The next logical step soon afterwards was the application of GAPE to VLSI routing [3]. Again, this work was another example of a parallel genetic algorithm outperforming 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 [10]. Some research has also been done to understand why some evolutionary algorithms succeed when applied to the VLSI field and why others fail [11].

Niebel has already analyzed the performance of GAPE on the TSP [13]. In his experiments, Niebel found GAPE to come within 1% of the known best solution while outperforming the standard GA, which only came within 2%. However, Niebel's GAPE

was designed for sequential processing and there remain questions which it cannot answer due to the lack of processing power inherent in a sequential implementation.

In order to move beyond the limits of sequential programming, the GAPE implementation to be developed during this project will be programmed using the Mentat Programming Language (MPL) [6]. This language is an extension of the C++ programming language which can enable objects to be processed on several machines in parallel as opposed to on one machine sequentially. An MPL program will usually be processed over a Legion distributed network of workstations, which is a system attempting to provide one virtual computer to all its users, linking workstations to supercomputers and even to parallel supercomputers [7], [9].

As far as the field's perception of GAPE, it has been received with much anticipation, despite the fact that it has not yet taken the field by storm. Although it has met with success in many of its applications, many still question its performance in the areas which have yet to be researched. Thus several heuristics, such as simulated annealing, continue to be used for many problems whose solution requires more of an open approach when searching for the best type of algorithm to use [12].

C. Rationale and Objectives

At a time when the future of computing lies in parallel processing, those algorithms with a higher degree of parallelism will eventually outperform the others, even if the others are inherently superior for a given problem. The sheer computing power required to realize some algorithms sequentially makes them obsolete for practical implementations. At the same time, algorithms that may not be as effective at finding the optimal solution but which have a higher degree of parallelism will become algorithms of choice, especially when the demands of real-time systems start being met only by the power of parallelized algorithm implementations. It is these observations that make GAPE an attractive alternative for heuristic search.

GAPE has already proven itself in some cases when compared to the standard GA. However, many analyses remain to be performed, including both additional comparisons to the standard GA as well as studies on GAPE's performance for various parameters. Along these lines, the objectives for this project included

- comparing GAPE instances with a constant total number of individuals but varying numbers of populations and individuals per population
- comparing GAPE instances with a constant total number of generations but varying numbers of epochs and generations per epoch
- parallelizing Niebel's software implementation of GAPE in order to make it suitable for MPL and for runs on a Legion distributed network

D. Overview of Technical Report

Chapter 2 presents our methodology in carrying this project to completion. This includes discussions of the Legion Project and MPL, the issues and underlying algorithms brought up by Niebel's sequential version of the GAPE software, and discussions of the experimentation techniques employed in order to obtain the results. Chapter 3 discusses the hypotheses we formulated before engaging in the experimentation, as well as the reasoning we followed in arriving at those hypotheses. Chapter 4 merely presents our results without interpretation or bias. Chapter 5 reveals our conclusions concerning the results we obtained. Interpretations of those results are presented here and recommendations for future study are also made. Finally, the appendix provides an annotated reference manual for the Legion version of the software in order to aid its future maintainers.

Chapter II: Methodology

This chapter presents a discussion of the methodology we employed in order to carry out the experimentation for this project and obtain the results. The first section presents some background on MPL and the Legion Project in the Department of Computer Science. The second section discusses the modifications we made to Niebel's sequential GAPE software in order to make it compatible with MPL and the Legion distributed network. Finally, the third section discusses the experimentation techniques we employed in order to obtain our results.

A. The Legion Project and MPL

The Legion Project was recently started within the Department of Computer Science with an aim to eventually provide a worldwide virtual computer that makes effective and efficient use of the pool of computing resources that becomes available when multiple workstations are connected together by a network. The Legion operating system works within a UNIX environment to manage the distributed network and the allocation of its resources. However, the core of the Legion system is MPL, as the only way a program can make use of Legion's parallel environment is if the program is written in this language.

MPL is an extension of the C++ programming language and is intended to provide C++ programmers with the ability to create objects for parallel computation. These objects are able to make use of the computing resources of multiple workstations connected to a Legion distributed network. Such an object is created by using the **mentat** keyword before a C++ class, thus declaring that all instances of the class should be created for a parallel distributed computing environment, with the possibility of computation occurring on many different machines. MPL introduces several programming issues that the creators of the C++ language did not have to deal with, such as what to do with pointers, whose values point to addresses on one machine that are meaningless on another machine. However, while MPL is still in the developmental and experimental stage, the Legion group is finding some interesting and effective solutions to these dilemmas, and thus MPL's power and ease of use should grow with time.

B. GAPE Software and Issues

The GAPE software discussed in this section was originally developed by William D. Niebel to run on a single machine. As one of our objectives, the parallelization of this software involved substantial modifications to the way the software manages the different GAPE entities, if not to Niebel's evolution algorithms themselves. This section will discuss the current state of the parallel version of this software, as we have modified it from the original.

The software begins by instantiating entire populations and sending them off to the various machines available, ready to begin evolution. The scheduling of which population is sent to which machine is handled internally by the Legion system and is therefore hidden from the MPL programmer. While the scheduling algorithms will eventually be intelligent, they do not perform any kind of analysis at the moment upon the machines belonging to the network, so the scheduling is performed without knowledge of each machine's resource availability, workload, etc. Legion does provide the programmer with the power to schedule the populations himself, but we realized this would have been a very time-consuming task, and so left it up to the Legion system.

1. Evolution

After the populations have been instantiated on their respective host machines, a signal is sent to each of population to begin evolving. An epoch then begins, and each population cycles its individuals through crossover, selection, and mutation once for each generation in the epoch.

a. Crossover and Niebel's Underlying Algorithms

A discussion of crossover naturally begets an analysis of several implementation issues which must be resolved at one point or another. The most important of these issues is how best to represent individuals for the problem at hand. This was alluded to earlier but purposefully left ambiguous so that the reader may have time to see some of the complexities involved in this decision.

That is, what exactly shall an individual be in our attempt to solve the given TSP instance? Niebel made this decision when he carried out his work, and we chose to abide by that decision. The decision was that an individual would not be an actual tour of the

given TSP map, since there is no obvious way to effectively cross two tours to produce an offspring. Instead, Niebel decided to use the technique of problem deformation, as mentioned earlier. This means that individuals would actually be perturbed instances of the original TSP map.

One of the algorithms Niebel mentioned provides a computationally efficient and effective way to construct a tour for any given TSP map that will be at most 1.5 times the length of the optimal tour. Therefore, if an individual is taken to be a perturbed version of the original TSP map, this algorithm can act as a function and generate a tour for each individual. The generated tour is then examined on the original TSP map and its fitness is calculated. The individual's survival depends on this fitness.

In GAPE terms, the perturbed map is actually an internal representation (*genotype*) of an individual, while the tour is its external representation (*phenotype*). The conversion from genotype to phenotype is achieved via an intermediate data structure called a *minimum spanning tree (mst)*. A spanning tree of a graph G is a subgraph of G which contains all the vertices of G and is also a tree. A spanning tree of a graph G is a minimum spanning tree if there does not exist another spanning tree of G which has shorter total length. As the reader may have already inferred, a minimum spanning tree of a graph G is not unique.

This decision to represent individuals by perturbed maps of the original TSP map was made for two principal reasons. The first was based on the reasoning that this representation provides an efficient and obvious way to perform crossover on two individuals in order to obtain an offspring individual. The second was based on the reasoning that, while the most obvious tours on the original TSP map are usually not the optimal ones, it is easy to construct these tours, and therefore the optimal tour on the original TSP map may well turn out to be an obvious tour on a perturbed version of that map. **Figure 2** shows (a) the original map, (b) one of its minimum spanning trees, (c) its generated tour, (d) a perturbed map, (e) one of its minimum spanning trees, and (f) its generated tour for the Eilon-Christofides 51-city TSP instance, one of the more popular instances for experimentation. Notice that, while the perturbed map is not too different from the original map, the two minimum spanning trees are quite different in certain places, and the one from the perturbed map results in at least one optimization in the tour

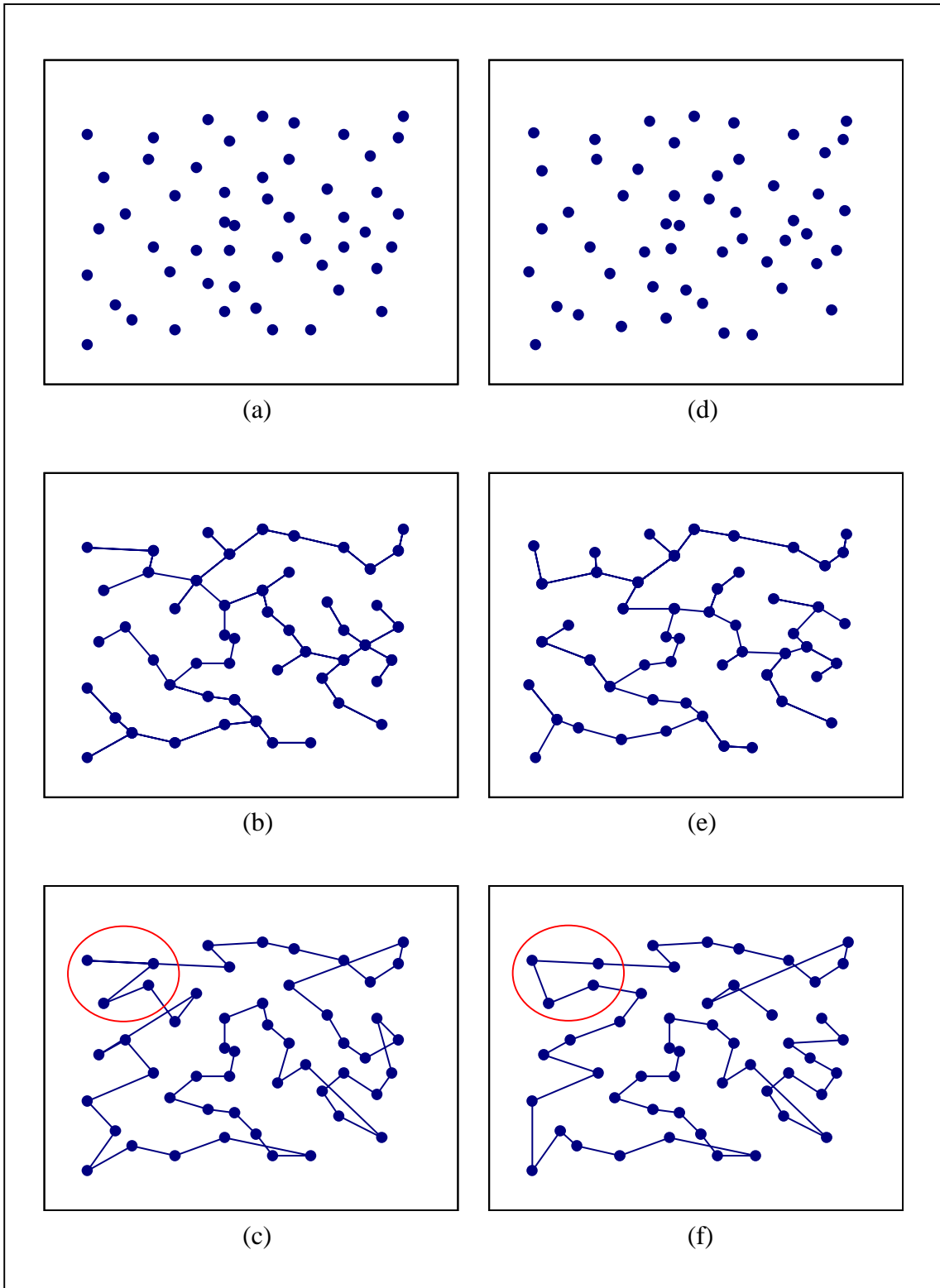


Figure 2: Eilon-Christofides 51-city TSP. (a) original map (b) minimum spanning tree (c) generated tour (d) perturbed map (e) minimum spanning tree (f) generated tour generated from the perturbed map (compare the circled upper left corners).

2. Migration

After all the generations of the epoch have been processed, migration must take place. A random selection of migrants is made using fitness as a weight, with the actual number of migrants selected being based on the connectivity matrix. The migrants are then sent out from each population to each of the adjacent populations, as dictated by the connectivity matrix. The temporary population swelling due to this immigration is remedied by a period of selection in which the population is reduced back to its normal size.

C. Experimentation Techniques

The first series of experiments we ran involved keeping constant the percentage of the population that the total number of incoming immigrants represented and varying the number of populations while keeping the total number of individuals constant. The actual experiments which were run for this series are summarized in **Table 1**. Five runs were performed for each experiment listed.

Pops	Inds/Pop	Epochs	Gens/Epoch	Migrants	Adj Pops
2	1280	16	50	576	1
4	640	16	50	288	2
8	320	16	50	144	3
16	160	16	50	72	4
32	80	16	50	35	5
64	40	16	50	18	6
128	20	16	50	7	7

Table 1: First Series of Experiments. *CONSTANT: [(Inds/Pop)/Migrants], [Pops * Inds/Pop];*
VARIABLE: Pops, Inds/Pop, Migrants

The second series of experiments made use of the third experiment in **Table 1** as a starting point and varied the number of epochs while keeping the total number of generations constant. The actual parameters for this series are listed in **Table 2**. Five runs again were performed for each experiment listed.

Pops	Inds/Pop	Epochs	Gens/Epoch	Migrants	Adj Pops
8	320	1	512	144	1
8	320	2	256	144	2
8	320	4	128	144	3
8	320	8	64	144	4
8	320	16	32	144	5
8	320	32	16	144	6
8	320	64	8	144	7
8	320	128	4	144	8
8	320	256	2	144	9
8	320	512	1	144	10

Table 2: Second Series of Experiments. *CONST: [(Gens/Epoch) / Epochs]; VARIABLE: Gens/Epoch, Epochs*

The third experiment was more informal than the previous two in that we merely wanted to formulate an idea about how the Legion version of the software would run as compared to the sequential version of the software with regard to speed. We therefore performed some runs under both versions, timing the evolution phase separately from the migration phase.

Chapter III: Hypotheses

At the onset of this project, we formulated several hypotheses regarding the experiments we would eventually run. These hypotheses were mostly based on our own understanding of GAPE and its merits, as well as a bit of intuition. This chapter presents these hypotheses in three sections, each one corresponding to one of our objectives.

A. Variable Number of Populations

Our first hypothesis concerned the first series of experiments regarding the variable number of populations and the total percentage of immigrants. We hypothesized that the best results would occur for intermediate population sizes and numbers of populations, while both the extreme of a few large populations or that of a large number of small populations would yield worse results. This hypothesis is based on the reasoning that a small number of large populations would act in pretty much the same way as a series of concurrent GA runs, which we believe to be inferior to GAPE, while a large number of small populations would approach the results of a single large GA instance. We then figured that an intermediate population size and an intermediate number of populations would best highlight the merits of GAPE, if indeed they were a reality.

B. Variable Number of Epochs

For the second series, we hypothesized, following similar logic, that a GAPE run with an intermediate epoch size and an intermediate number of epochs would yield the best results because a run with a few large epochs would lessen the impact that migration has upon the algorithm's workings and because a run with a large number of small epochs would approach a single population communicating within itself very frequently and very freely, forming crossovers between one individual and almost any other.

C. Parallelization and Execution Speed

Our third experiment, albeit less formal than the others, attempted to arrive at some relative execution speeds for the evolution phase of GAPE and the migration or communication phase. Knowing fully well that it is a lot less costly for a machine to

communicate within its infrastructure rather than without, we expected that the Legion version of the GAPE software would perform the evolution phase a lot faster than the sequential version because it has its populations spread out over multiple machines, so that each population is allocated a larger amount of computing resources. Consequently, we also expected the Legion version to perform a lot more poorly during the migration or communication phase of GAPE because it has to perform its interpopulation communication across a network of machines rather than within one specific machine, as the sequential version does. As previously mentioned, the latter is accomplished at a higher latency cost.

Chapter IV: Results

This chapter briefly presents the results of our research in three sections, each one once again corresponding to one of our three objectives. The first section presents the results of the first series of experiments on variable numbers of population. The second section presents the results of the second series of experiments on variable numbers of epochs. Finally, the third section presents the results of the third informal experiment on the execution times of the Legion version of the GAPE software and the sequential version.

A. Variable Number of Populations

The first series of experiments gave us less than promising results. In fact, there was little, if anything, to distinguish the behavior of any one experiment from any other. The results of this series are presented in **Figure 3**, in which the x -axis represents the generation number (i.e., time) while the y -axis represents the length of the best tour found so far within a given population.

B. Variable Number of Epochs

The second series of experiments was a bit more enlightening. In fact, these results show that the experiment with 512 epochs was the fastest to find better solutions, with the experiment with 256 epochs following in at a close second. The other experiments with a relative large number of epochs also seemed to outperform the ones with a relatively lower number of epochs. However, when it came to finding the best solutions, the experiments with relatively fewer epochs tended to fare better in the end, implying that the two sets of curves must at some point cross over one another. The results of this series are presented in **Figure 4**, in which the x -axis represents the generation number (i.e., time) while the y -axis represents the length of the best tour found so far within a given population. **Figure 5** and **Figure 6** are zoomed-in versions of **Figure 4**, with the former showing the details near the point that the two sets of experiments cross over each other and the latter showing the details near the end of the experiments.

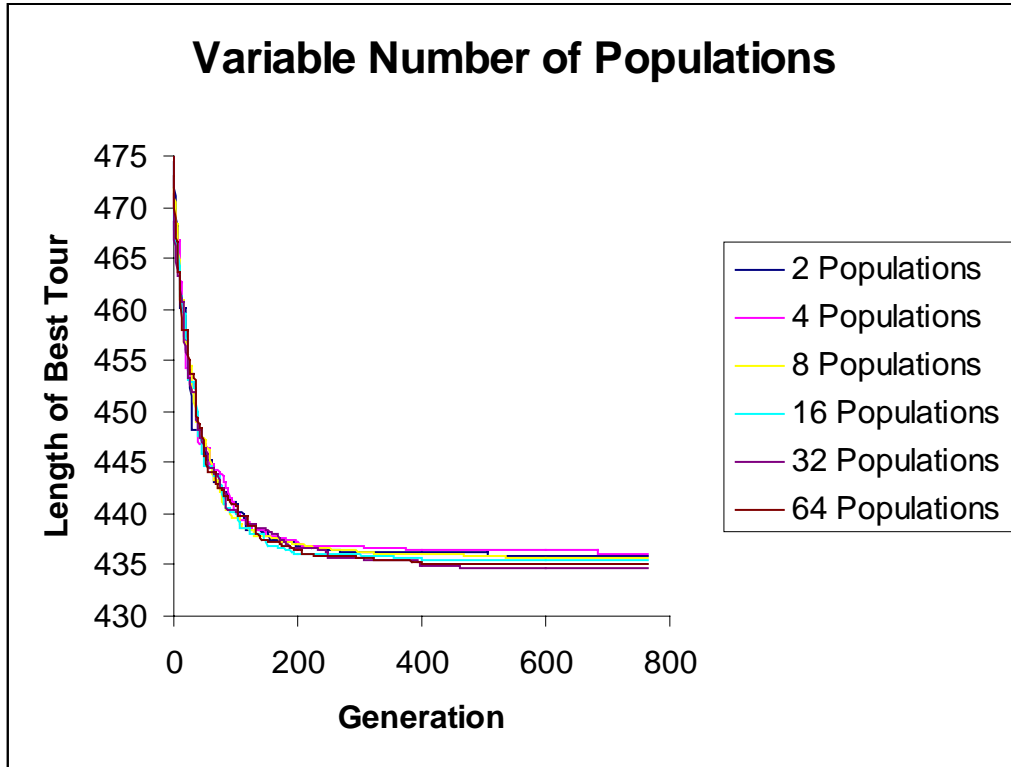


Figure 3: Variable Number of Populations.

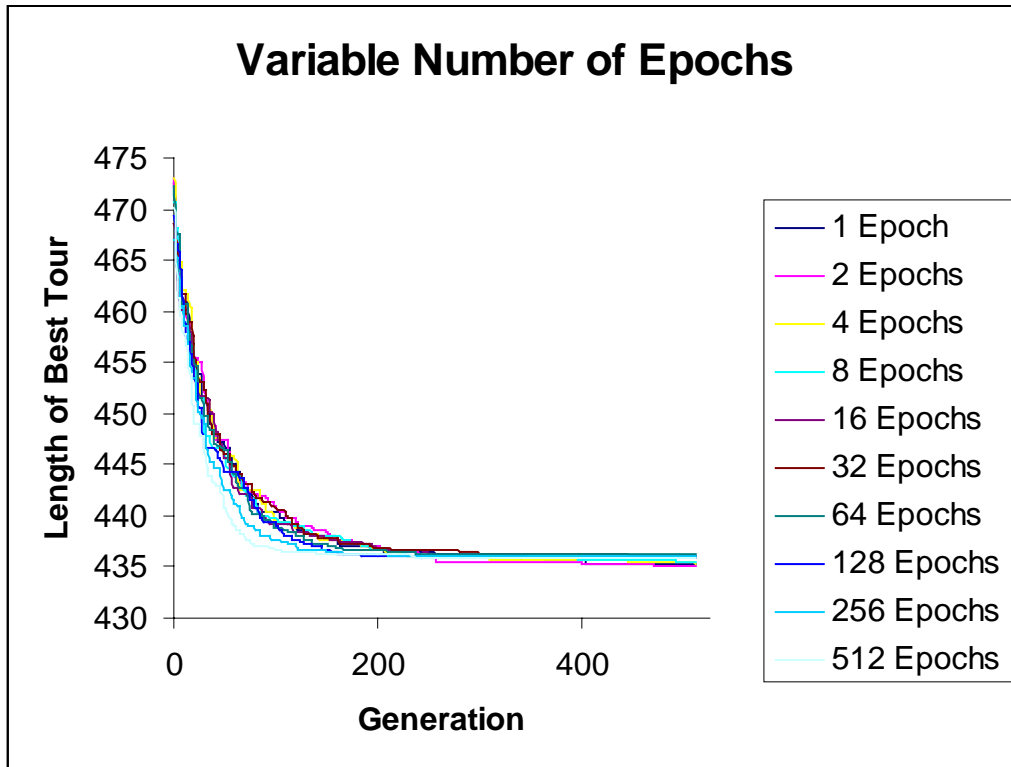


Figure 4: Variable Number of Epochs.

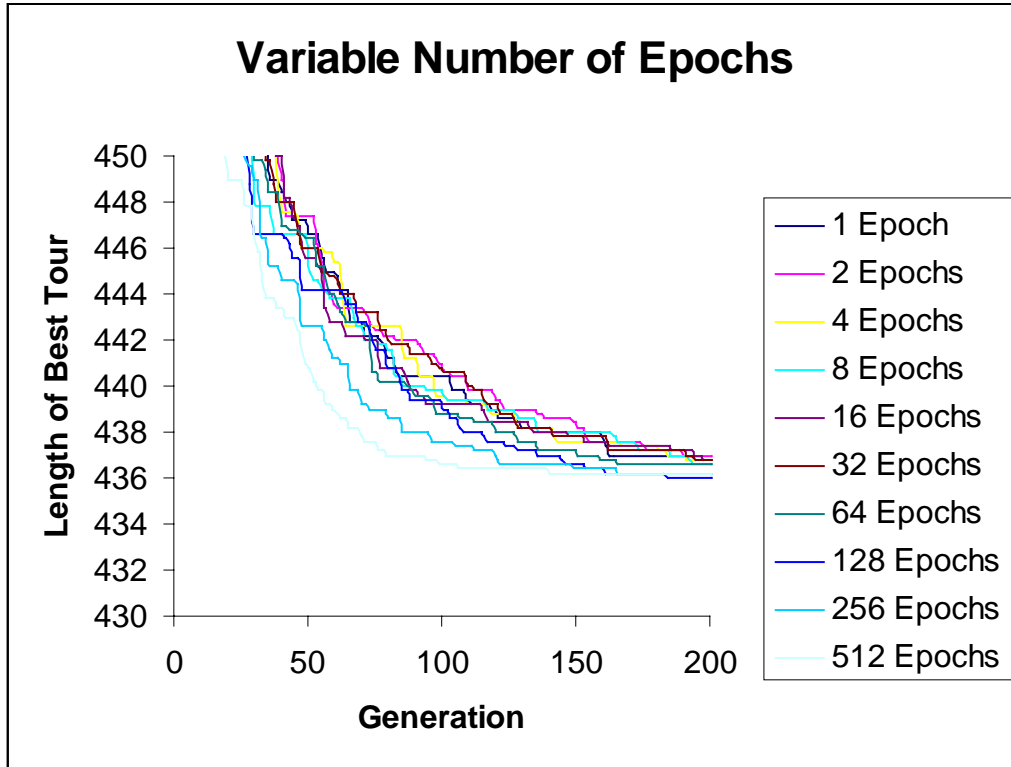


Figure 5: Variable Number of Epochs. *Zoomed to show details around middle.*

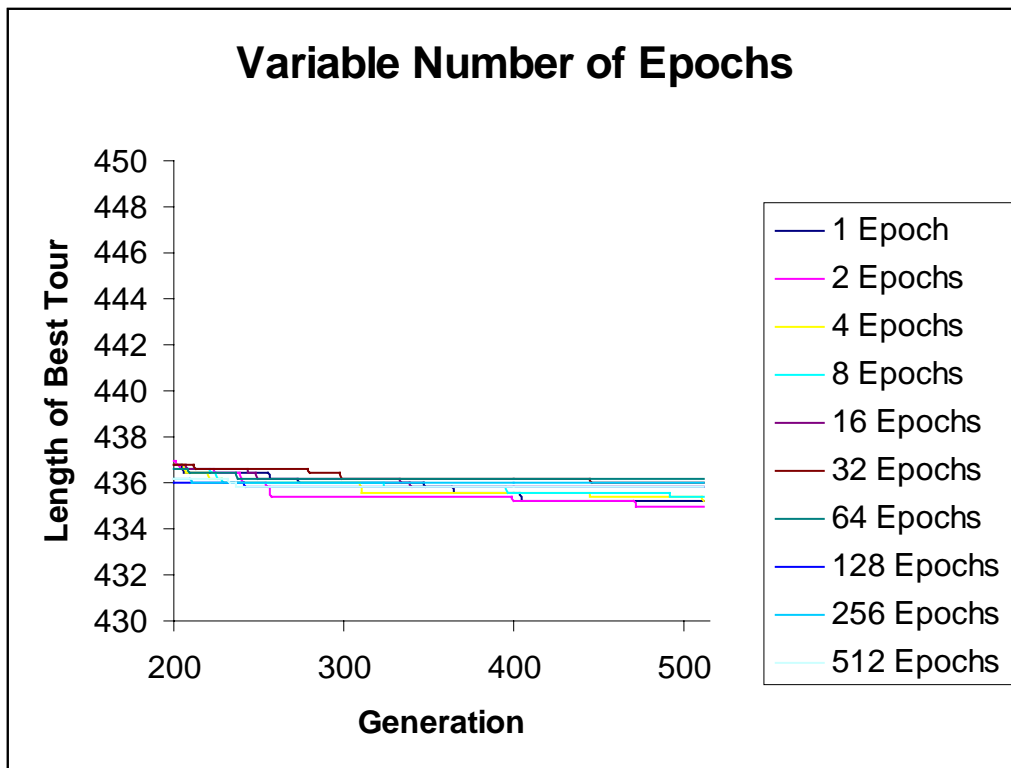


Figure 6: Variable Number of Epochs. *Zoomed to show details at end.*

C. Parallelization and Execution Speed

We determined that there is a speedup of roughly two during the evolution phase in the Legion version of the software over the sequential version undergoing the same phase. However, we also noticed that the communication required by the Legion version is extremely slow compared to that required by the sequential version of the software, and this fact seemed to nullify the previous speedup during evolution.

Chapter V: Conclusion

This chapter presents the conclusions of this research project. The first section presents a factual summary of the data and findings presented in earlier chapters. The second section provides a discussion of the author's interpretation of the results obtained as part of this project. Finally, the third section presents some recommendations for future work and study in this field.

A. Factual Summary

To summarize the data and findings of the earlier chapters, it is necessary to restate the three types of experiments carried out as part of this project.

First, we tested a variable number of populations while keeping the total number of individuals and total percentage of the population represented by immigrants constant. We found no significant difference between the behavior of the different experiments. They all seemed to follow the same general curve and they all intermingled with one another erratically within this small degree of freedom.

The second experiment was similar to the first, with the exception of testing a variable number of epochs instead of populations. This time, the total number of generations was kept constant, as were the other variables which varied in the first experiment. The results in this case did present us with some interesting issues. That is, in the experiments with many, small epochs GAPE started off finding better solutions a lot faster than in those with fewer, larger epochs. However, somewhere in the middle of the runs, these two sets switched, and so in the experiments with fewer, larger epochs GAPE ended up finding slightly better solutions in the end than in those with many, smaller epochs.

Finally, the third experiment, the most informal of the three, was an attempt at determining the relative execution speeds of the Legion version of the GAPE software and of the sequential version. We found that, during evolution, the Legion version experienced a speedup of about two over the sequential version. However, during the migration or communication phase of the algorithm, the Legion version was very slow relative to the sequential version.

We are now ready to state some interpretations of the above findings and data.

B. Interpretations of Results

The first series of experiments provided us with practically inconclusive data because there was no differentiation between the experiments. We believe this may be attributed to the fact that we used the Eilon-Christofides 51-city TSP as the TSP for all our testing thus far, simply because it is the smallest one we have that is interesting and thus the fastest one to experiment with. However, the lack of differentiation in the results of the first series of experiments suggests that a larger TSP instance is necessary in order for the experiments to show some differentiation in their results. Therefore, an instance such as the Smith-Thompson 70-city TSP or the Eilon-Christofides 101-city TSP, both of which are larger than the aforementioned 51-city problem, may be better suited for this type of experiment.

The second series of experiments seems to suggest that there is eventually a tradeoff between the time to find a solution and its length. That is, while in the experiments with many, small epochs GAPE generally ran faster at first and found better solutions in less time, the curves ended up crossing those resulting from the experiments with fewer, larger epochs, in which GAPE found solutions in the end that were closer to optimal. This proved to be contrary to our belief that the extreme experiments would not yield results as good as those of the more intermediate experiments. Quite the contrary, the intermediate experiments in this case achieved neither speed nor proximity to the optimal solution. Rather, as their nature implies, they achieved a more balanced result.

Finally, for our third experiment, we concluded that the Legion distributed network which we were using for these experiments is quite slow and it must be managing a substantial external workload. The speedup of two during the evolution phase is to be expected since Legion has access to networked resources. Nonetheless, even in this area our hopes were dashed since we expected a little more than merely a speedup of two. As for the migration phase, we concluded that this unfortunate fact will become less influential as both the Legion technology and the distributed networks which Legion runs on improve drastically in the near future. Also, when the Legion version of

GAPE is run on a network that can intelligently schedule one population to a processor, the speedup should be tremendous for both phases.

C. Recommendations for Future Study

For future study, the author would like to point out that there are almost too many possibilities to mention. These experiments with GAPE have so many variables that there will not be a lack of possibilities anytime soon.

First, all the experiments run for this project were run for the Eilon-Christofides 51-city TSP, so, needless to say, they could all be run again using the other two TSP instances mentioned above.

Second, all the experiments could be performed over again for larger populations, more total individuals, larger epochs, and more total generations.

Third, the interconnection topology can be altered. Instead of a hypercube, other topologies may be used and their effects on GAPE determined, as well as which particular characteristics caused them to be better or worse than the traditional hypercube.

Fourth, the fitness function could be experimented with. A novel idea suggested by Justin Turner would be to emulate niche environments by varying the fitness function across populations. This might push evolution to have geographic characteristics.

These are just some examples of the possibilities for future research and study, and it should be remembered that any of the variables which we touched upon can be grounds for extended analysis.

References

- [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] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. United States of America: Computer Science Press, Inc., 1978, pp. 501-558.
- [9] M. Lewis and A. S. Grimshaw, *The Core Legion Object Model*. University of Virginia Computer Science Technical Report No. CS-95-35: August, 1995.
- [10] J. Lienig, "A Parallel Genetic Algorithm for Performance-Driven VLSI Routing," in *IEEE Transactions on Evolutionary Computation*. New York: IEEE, 1997.
- [11] 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.
- [12] W. N. Martin, personal communication.

[13] 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.

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

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**
-