An empirical study of parallel genetic algorithms for the traveling salesman and job-shop scheduling problems

by David Michael Helzer

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
A genetic algorithm (GA) is a problem-solving technique that is based on the concepts of natural selection and genetics. GAs have become powerful methods for solving difficult optimization problems.

A GA begins with a varied population of individuals, each of which is a candidate solution to the problem being solved. The fitness of each individual is determined by how well it solves the particular problem. The GA evolves this population using crossover and mutation operators. Those individuals with high fitness levels are more likely to be selected for crossover than those with low fitness levels. The goal of the GA is to evolve a population of highly fit individuals, one of which can serve as the solution to the problem.

A GA can be parallelized to reduce its execution time by dividing the population of individuals among a number of different processors, each of which represents an island. The islands of subpopulations evolve independently from each other, promoting diversity among the indidividuals. Occasionally, individuals may be allowed to migrate between islands, depending on the physical topology of the islands.

This thesis describes how GAs have been implemented to solve the traveling salesman and job-shop scheduling problems. Four variations of PGAs have been designed, each of which uses a different island topology.

The four PGAs and the serial GA are compared for solution qualities obtained and execution times required. The PGA with high island connectivity and migration is shown to obtain higher quality solutions than other models examined in most cases. PGA models have shown to obtain solutions with error percentage improvements over serial GAs of 100.00%, 6.49%, and 21.21% for three selected traveling salesman problem data sets. These PGA models completed their executions in 12.1%, 7.2%, and 7.7% of the elapsed times required by the serial models. For the job-shop scheduling problem, solutions with error percentages that were 28.42%, 28.60%, and 64.50% better than the serial models' best solutions were achieved by the PGAs for three data sets. The elapsed times required by these PGAs were 15.1%, 13.4%, and 13.1% of the times required by the serial GAs.

AN EMPIRICAL STUDY OF PARALLEL GENETIC ALGORITHMS FOR THE TRAVELING SALESMAN AND JOB-SHOP SCHEDULING PROBLEMS

by

David Michael Helzer

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2001

APPROVAL

of a thesis submitted by

David Michael Helzer

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Year-Back Yoo — Y. B. Yoo (Signature) — 4/19/01 Date

Approved for the Department of Computer Science

Dr. J. Denbigh Starkey — J Denbigh Starkey (Signature) — 4/19/01 Date

Approved for the College of Graduate Studies

Dr. Bruce McLeod — Bruce R. McLeod (Signature) — 4-19-01 Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature David Heker

Date April 19, 2001

ACKNOWLEDGEMENTS

I would like to thank Dr. Year-Back Yoo for introducing me to the field of parallel computing and for being a source of much guidance and expertise throughout the research process of this thesis. Much gratitude is also given to the other members of my graduate committee, Drs. Gary Harkin, John Paxton, and Denbigh Starkey, for providing me with a computer science education that has prepared me for a future in the software field.

I would also like to thank my parents, Richard and Renae Helzer, for instilling in me a strong desire to succeed. Thanks are given to my sister, Melissa Helzer, for her support and friendship.

This work was sponsored by the National Science Foundation's Partnership for Advanced Computational Infrastructure with computing time from the Pittsburgh Supercomputing Center through grant number ASC010015P.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF TABLES – CONTINUED

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

LIST OF FIGURES – CONTINUED

ABSTRACT

A genetic algorithm (GA) is a problem-solving technique that is based on the concepts of natural selection and genetics. GAs have become powerful methods for solving difficult optimization problems.

A GA begins with a varied population of individuals, each of which is a candidate solution to the problem being solved. The fitness of each individual is determined by how well it solves the particular problem. The GA evolves this population using crossover and mutation operators. Those individuals with high fitness levels are more likely to be selected for crossover than those with low fitness levels. The goal of the GA is to evolve a population of highly fit individuals, one of which can serve as the solution to the problem.

A GA can be parallelized to reduce its execution time by dividing the population of individuals among a number of different processors, each of which represents an island. The islands of subpopulations evolve independently from each other, promoting diversity among the indidividuals. Occasionally, individuals may be allowed to migrate between islands, depending on the physical topology of the islands.

This thesis describes how GAs have been implemented to solve the traveling salesman and job-shop scheduling problems. Four variations of PGAs have been designed, each of which uses a different island topology.

The four PGAs and the serial GA are compared for solution qualities obtained and execution times required. The PGA with high island connectivity and migration is shown to obtain higher quality solutions than other models examined in most cases. PGA models have shown to obtain solutions with error percentage improvements over serial GAs of 100.00%, 6.49%, and 21.21% for three selected traveling salesman problem data sets. These PGA models completed their executions in 12.1%, 7.2%, and 7.7% of the elapsed times required by the serial models. For the job-shop scheduling problem, solutions with error percentages that were 28.42%, 28.60%, and 64.50% better than the serial models' best solutions were achieved by the PGAs for three data sets. The elapsed times required by these PGAs were 15.1%, 13.4%, and 13.1% of the times required by the serial GAs.

CHAPTER 1

INTRODUCTION

Biological Foundations

The field of computer science is devoted to using computers to find solutions to various problems. A genetic algorithm (GA) is one such method used to solve problems. The ideas behind GAs are based on Charles Darwin's theory of natural selection and basic genetic principles. Darwin reasoned that some living organisms within a population are more adapted to their environment than other organisms. Those that are more adapted are said to have high fitness levels. Having high fitness levels provides individuals with a greater chance of surviving to reproduce.

Gregor Mendel was the first biologist to explain how traits pass from parents to offspring in the form of genes. The offspring of sexual reproduction typically do not identically resemble their parents. Instead, the genetic code of a child organism is obtained by combining the genetic codes of both parent organisms. Fit organisms, then, are likely to pass good fragments of genetic code to their young, thereby creating fit offspring. Over time, a population of organisms is likely to evolve into a more fit population and become more adapted to its environment.

Genetic Algorithm Overview

John Holland, at the University of Michigan, developed genetic algorithms in the 1960's. It was in 1975 that his ideas were presented to the world in his book *Adaptation in Natural and Artificial Systems* (Holland, 1975). GAs are relatively easy to implement with a computer program and have been used to successfully find solutions to a variety of problems including scheduling, optimization, and machine learning.

GAs begin with a population of individuals, each representing a candidate solution, or *chromosome*, to the problem being solved. The chromosomes themselves are made up of a set of values, or *genes*. The possible values that each gene can take on are known as the gene *alleles*. Each of these solutions is evaluated with a fitness function used to determine how well the solution solves the problem. The candidate solutions are selected based on their fitness levels to be *crossed* with other solutions to form new chromosomes (simulating reproduction in the natural world). A GA may also employ *mutation*, or random alteration of the genetic codes of individuals. During each evolutionary step in a GA, a set of children chromosomes is generated to replace the parent chromosomes. With each generation, better solutions are strived for until, after creating the final generation, the best candidate solution in the population is selected to serve as the solution to the problem being solved.

Parallel Genetic Algorithm Overview

Since the creation of the numerous generations in a GA can require a substantial amount of time to run on a computer, GAs are good candidates for parallel processing. Parallel computing seeks to use multiple central processing units (CPUs) within a single computer system to work together to solve problems. Typically, parallel computer systems can be expected to solve problems faster than serial systems (those with only one CPU).

Various methods for parallelization of GAs exist. One of the most popular of these methods is the island parallel genetic algorithm (PGA), during which the initial generation's chromosomes are divided among a set of processors. Each processor represents an island that is separated from the other islands. Each island's subpopulation of candidate solutions is evolved independently from the other subpopulations. This helps to promote diversity among the individuals and can help to generate higher quality solutions faster than would be possible with a serial GA. Occasionally, candidate solutions are allowed to migrate between islands, to ensure that the good solutions are allowed to spread to other subpopulations and so the genetic algorithm, as a whole, is working toward a common solution. The physical layout, or *topology*, of the islands determines which islands are allowed to exchange individuals via migration.

Experimental Goals

This thesis seeks to explain how genetic algorithms can be implemented to solve problems and how they can be parallelized. Little documentation exists comparing the quality of PGAs with that of serial GAs. Two optimization problems, the traveling salesman and job-shop scheduling, are presented, along with methods used to find solutions to those problems with a GA. Additionally, new experiments have been performed to determine how the serial genetic algorithm compares to four variations of the island model PGA. Answers to the following three questions were reached.

1) What type of island topology can be expected to obtain the best solution qualities when using a PGA?
2) How much improvement in solution quality can be expected when using an island model PGA compared to a serial GA?
3) How much speedup can be expected when using an island model PGA compared to a serial GA?

It is hoped that the results obtained from the experiments with the traveling salesman and job-shop scheduling problems can be generalized to other problems that can be solved with a GA as well. The goal of this thesis has not been to develop the best possible genetic algorithms to solve these problems. Instead, the focus has been on determining how the serial and parallel variations of the programs compare with each other.

Hardware Resources

The National Science Foundation's Parternship for Advanced Computational Infrastructure supported this research by providing computing time on a Cray T3E system at the Pittsburgh Supercomputing Center. The Cray T3E contains 512 Digital Alpha 450 MHz processors arranged in a three-dimensional torus topology. The memory of the system is physically distributed with 128 MB for each processor.

Message Passing Interface

To implement the parallelism and perform the communication between the processors, the Message Passing Interface (MPI) was used. MPI is a popular standard that includes a set of portable library functions that allow inter-process communication to take place. When using MPI, the number of processors desired is specified when running a program. At the start of the program, a process is launched on each of the processors desired. Each process is assigned a unique identification number. It is with this identification number that processes can indicate the other processes with which they are to communicate. All of the programs were implemented in C.

CHAPTER 2

GENETIC ALGORITHMS

Combinatorial Optimization Problems

A genetic algorithm is composed of a set of operations modeled after those found in the natural world. GAs are most often used to find good solutions to combinatorial optimization problems that have large search spaces. A combinatorial optimization problem is one for which a discrete set of possible solutions exists. Examples of such problems include ordering, scheduling, and packing problems. For all of these problem types, there exists an optimal solution in a search space of all candidate solutions. Many of these types of problems belong to time complexity classes for which polynomial solutions remain unknown. To perform an exhaustive search through a large search space to find the optimal solution to one of these problems could require a great deal of time.

Heuristics and the Search Space

A GA is one of many heuristics classified under the field of artificial intelligence. A heuristic is a set of rules used to guide a search through a search space. Rather than performing an exhaustive search through all candidate solutions in a search space, heuristics seek to examine only some of the candidate solutions and still obtain a good one. Other heuristic techniques include simulated annealing and tabu search.

An example of a search space is shown in Figure 1. This space is that of the function $f(x, y) = x + y$, where $(0 \leq x, y \leq 10)$. One problem to be solved may be to find the (x, y) values that maximize this function. The solution, then, would be the pair $(x = 10, y = 10)$. One can see that even when only integer values are considered, there are eleven possible values that the parameters x and y can take. This gives a total of 121 different combinations of (x, y) pairs that would have to be examined to perform an exhaustive search and guarantee that the function be maximized. One could design a genetic algorithm to maximize the value for $f(x,y)$ and avoid having to perform such a search.

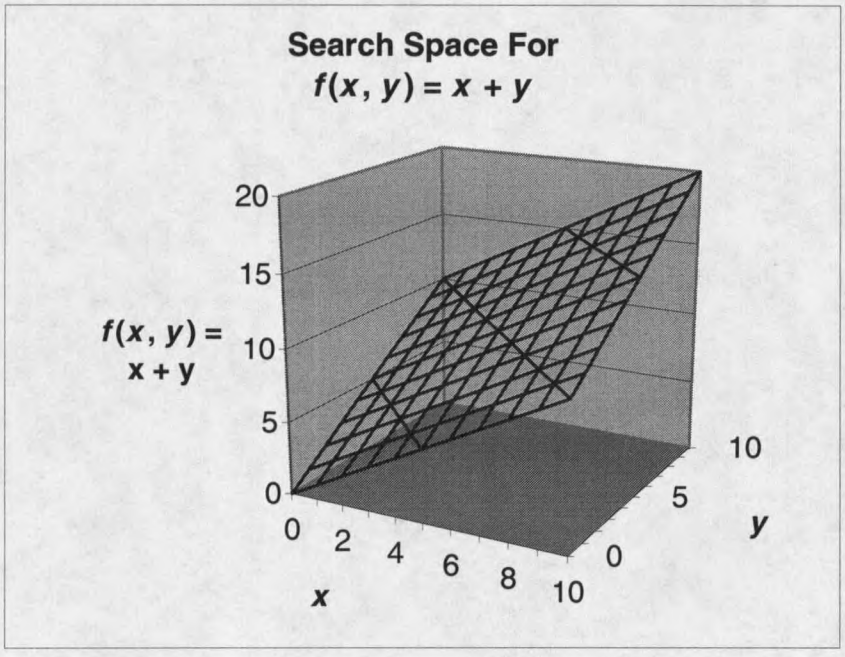


Figure 1: Search Space for the Function $F(x, y) = x + y$

One benefit to using a GA is that it can overcome the challenges posed by search spaces that are uneven. Such search spaces may have groups of good solutions separated by bad ones. Rather than converging upon a local optimum as many other heuristics tend to do with uneven search spaces, a genetic algorithm is able to search through many locations in a search space simultaneously, thereby reducing the chances of a poor, premature convergence (Mitchell, 1996).

Genetic Algorithm Definition

A typical GA follows the steps shown in Figure 2, each of which will be described in further detail. Much variation can exist in GAs. However, the steps shown here are common in most.

```
 1) Randomly generate n individuals.
 2) Loop once for each generation desired:
 3)    Calculate the fitness f(x) of each individual x.
 4)    Place the best number_elite individuals in new population.
 5)    Loop to generate (n - number_elite) new individuals:
 6)       Select two parent chromosomes.
 7)       Cross the parents to generate two new individuals.
 8)       With some probability, mutate the children.
 9)       Place the two children into a new population.
10)    End loop.
10)    Replace the current population with the new population.
11) End loop.
11) The most fit individual is selected as the solution.
```

Figure 2: Genetic Algorithm

Initial Population

A GA must begin with some initial population of individuals, each of which represents a candidate solution to the problem being solved. Typically, this population of chromosomes is generated using some method involving randomness. It has been shown that starting with an initial population of chromosomes with high fitness levels can be beneficial. They should be generated using some type of randomized heuristic that is known to build high quality solutions to the problem (Ahuja & Orlin, 1997). As in the field of genetics, each chromosome is made up of a set of genes. In nature, these genes encode some type of individual trait; in a genetic algorithm, the genes encode a candidate solution.

Selection of Individuals

To make use of the idea of natural selection, a GA must provide a method by which fit individuals are more likely to be selected for crossover than weak individuals. This is performed by developing a problem-specific fitness function that determines how well a candidate solution solves the problem. Once the fitness levels of the chromosomes in a population are known, a probability of selection can be assigned to each chromosome. Selection of only the best individuals can cause a GA's population to quickly become very similar, thereby converging upon an answer without examining enough of the search space. Selection of too many weak individuals may prevent a GA

from locating the best solutions due to concentrating too heavily on the poor ones (Mitchell, 1996).

Fitness-Proportionate Selection

Under fitness-proportionate selection, an individual's probability of selection is directly proportionate to its fitness level (Mitchell, 1996). The probability of an individual being selected using fitness-proportionate selection is given by the formula

$$P(x) = \frac{Fitness(x)}{\sum_{i=1}^{n} Fitness(i)}.$$

An example population of individuals that has been assigned selection probabilities in this manner is listed in Table 1.

x	*Fitness (high is good)*	*Selection Probability*
1	2	2 / 16 = 0.13
2	4	4 / 16 = 0.25
3	3	3 / 16 = 0.19
4	7	7 / 16 = 0.44
	Sum = 16	*Sum = 1.0*

Table 1: Example Population with Fitness-Proportionate Selection Probabilities

Rank Selection

Rank selection can serve as an alternative to fitness-proportionate selection. Under rank selection, very weak individuals have a greater chance of being selected than

under fitness-proportionate selection, possibly preventing premature convergence (Mitchell, 1996). When using this method, a population's individuals are first ranked according to their fitness levels from n to 1 (n being the chromosome having the highest fitness level). The probability of selecting an individual x is given by the formula

$$P(x) = \frac{Rank(x)}{\sum_{i=1}^{n} Rank(i)}.$$

An example of a population with selection probabilities assigned using rank selection assignment is shown in Table 2.

x	*Fitness Level (high is good)*	*Rank*	*Selection Probability*
1	2	1	1/10 = 0.10
2	4	3	3/10 = 0.30
3	3	2	2/10 = 0.20
4	7	4	4/10 = 0.40
		Sum = 10	*Sum = 1.0*

Table 2: Example Population with Ranked Selection Probabilities

Solution Encoding

Whenever designing any type of computer program, potential solutions to the problem being solved must be represented in some way. The representation chosen for a GA is especially important since it can determine how well it performs. It must be flexible enough to provide crossover and mutation operators. It is important that all possible solutions be able to be encoded and that only valid solutions be represented. It is

also best if a small change in a chromosome produces only a small change in the solution it represents (Kershenbaum, 1997).

Binary Encodings

The simplest method for encoding solutions is using only strings containing the binary numbers zero and one. With only binary numbers to represent solutions, chromosomes are typically very long strings. Holland argued that solutions encoded with long strings containing a small number of alleles perform better than those encoded with short strings containing a large number of alleles. He reasoned that long strings promote more implicit parallelism, which will be described shortly (Mitchell, 1996).

An example of a binary encoding system is shown in Figure 3. The problem is to find the square root of the number 16. While this is not a combinatorial optimization problem, it could be solved using a genetic algorithm. The binary equivalents of the numbers being represented could be used for the actual chromosomes.

Problem: Find the square root of 16.

Chromosome	01101	00101	00100	10001
Solution	13	5	4	17

Figure 3: Binary Encoding Example

Many-Character Encodings

Rather than using only binary numbers to represent candidate solutions, a GA can use a wider range of characters. Such an encoding is referred to as a many-character encoding. When using a wider variety of characters to represent solutions, chromosomes tend to be much shorter than when represented with only binary numbers. They are natural to use but can be difficult to provide the flexibility required for the crossover and mutation operators. The theory that they do not perform as well as binary encodings has been questioned. The performance seems to depend on the problem being solved and the biological operators used (Mitchell, 1996).

An example of a many-character encoding system is shown in Figure 4. The problem being solved is the traveling salesman. Each of the cities is assigned an index number. The chromosomes, then, contain an ordered list of indices representing the order in which the cities are visited.

Problem: Traveling Salesman – Given a set of city indices and distances between those cities, find the shortest path that visits all cities exactly once and ends on the same city it starts with.

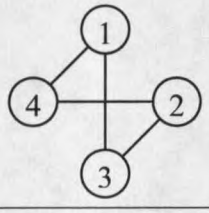
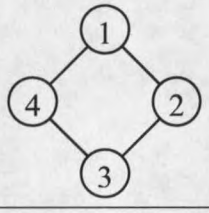
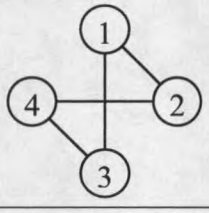
Chromosome	1-3-2-4-1	1-2-3-4-1	1-2-4-3-1
Solution			

Figure 4: Many-Character Encoding Example

Tree Encodings

When using either a binary or a many-character encoding, the chromosomes are typically of a fixed length. Tree encodings, which are represented within a program as tree data structures, have the advantage of allowing for solutions from an open-ended search space. However, they can grow very large, reducing the structure of the chromosomes and making them difficult to understand (Mitchell, 1996).

Figure 5 contains an example of a problem for which a tree encoding would be appropriate. The problem being solved is that of forming logical sentences from a number of words. In each case, the sentence formed can be obtained by using an in-order traversal of the tree.

Problem: Generate a logical sentence from any combination of the words "cow," "the," "apples," and "eats."

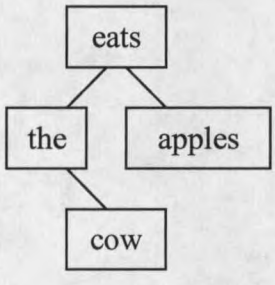
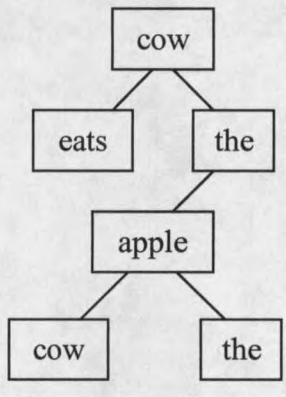
Chromosome		
Solution	"the cow eats apples"	"eats cow cow apple the the"

Figure 5: Tree Encoding Example

Biological Operators

Once the initial candidate solutions have been generated, they must be taken through the evolutionary process. This is done using a set of genetic operators. The three most common operators are crossover, mutation, and elitism. These operators resemble similar operations that take place in the natural world.

Crossover

Crossover simulates sexual reproduction between individuals in a population. Two individuals are selected from the population to generate one or more new individuals. Typically, two parents generate two new children during a single crossover. This, however, can vary. Ideally, the new individuals that are developed will have some similarities to both of the parent individuals. An example of a binary crossover is shown in Figure 6. In this crossover, a point is selected among the bits of the chromosomes. New individuals are generated by taking subsequences from both of the parents and combining them into new individuals.

Parents	*Children*
1 0 1 | 0 0 1 0	1 0 1 | 0 1 1 1
1 1 0 | 0 1 1 1	1 1 0 | 0 0 1 0

Figure 6: Crossover Example

Mutation

Mutation, just as in nature, should have a low probability of occurrence as it can lead to chromosomes that have low fitness levels. The mutation operator is typically used without regard to the final consequences. One example of a mutation is shown in Figure 7. In this example, a random number of the bits have been selected in the chromosome and those binary values have been flipped. Similar mutation methods can be designed for the other types of encodings.

Original	*Mutant*
1 0 0 1 0 1	1 1 0 0 0 1

Figure 7: Mutation Example

Elitism

Elitism ensures that a genetic algorithm's best individual(s) will never be lost between generation steps. Before selection of parents to create new chromosomes, a specified number of the best chromosomes in the current generation are copied to the new generation. Elitism has been shown to improve the performance of GAs (Mitchell, 1996).

Program Termination

At some point, a GA must halt and provide a solution to the problem being solved. The number of evolutionary steps to be taken can be determined by a couple of

different methods. Perhaps the simplest method is to have the GA always go through some specified number of generations. However, this may be wasteful of time if the GA converges upon a good answer early in the evolutionary process. Rather than always evolving a specified number of generations, the GA could terminate when the candidate solutions in the population become very similar and advancement toward better solutions is no longer being made.

Theoretical Foundations

While genetic algorithms have been very successful at solving many problems, questions exist about the theoretical foundations regarding the types of problems they work well for and why they work. John Holland believed that they work by "discovering, emphasizing, and recombining good 'building blocks' of solutions in a highly parallel fashion," according to Mitchell (1996, 27). Good solutions are made up of good building blocks, known as *schemas*. Over time, these schemas are likely to remain intact during crossover and mutation. The fitness of an individual, then, contains information about the fitness levels of all of its individual schemas (Forrest & Mitchell, 1993). While a GA may only contain n individuals, it actually contains many more schemas. This allows for a large portion of the search space to be implicitly evaluated simultaneously (Holland, 1975). The ability of GAs to simultaneously search many areas of a search space is known as *implicit parallelism*.

Advantages

When programming a GA, one significant advantage is that they are relatively simple to code, even when the details to the problem being solved are unclear. They only require that a fitness function, an encoding method, and biological operators be defined (Levine, 1997). When a computer scientist is confronted with a problem for which a direct algorithm cannot be found, a genetic algorithm can be a viable alternative. For some problems, there may not be a polynomial-time solution because they fall into a time complexity class such as nondeterministic-polynomial (NP) or exponential. GAs are able to efficiently find good solutions in large search spaces without having to perform an exhaustive search of the search space.

Disadvantages

Perhaps the greatest disadvantage to using a GA to solve a problem is that finding a proper encoding method for candidate solutions is necessary but can be difficult. The crossover and mutation operators must be designed to promote new solutions while maintaining characteristics of both parent individuals. Also, genetic algorithms can take a substantial amount of time to run, compared to more direct domain-specific algorithms. It has also been shown that domain-specific approaches tend to offer better performance than GAs (Levine, 1997). Another drawback of GAs is that when they are in their pure form and do not contain any problem-specific algorithms, they tend to converge upon solutions of poor quality. For that reason, some type of domain-specific hill-climbing

algorithm can benefit (Ahuja & Orlin, 1997). The crossover and mutation operators may also be designed to benefit the problem being solved by incorporating problem-specific algorithms. Such a GA is known as a *hybrid*. The theoretical foundations behind GAs are also weak. There is a lack of proof regarding the rate of GA convergence upon good solutions (Levine, 1997).

CHAPTER 3

PARALLEL GENETIC ALGORITHMS

Parallel Computer Systems

Parallel computer systems contain multiple processors. These processors can work together to solve a problem. This can provide the ability to solve problems faster than with serial computer systems. This is especially important when solving a problem that is computationally expensive and can require a great deal of time to process on a serial computer system. A parallel computer can also be an effective way to build a cheap computer with the power of a supercomputer. In recent years, relatively small increases in processor speed have come at a great cost. Therefore, it can be cheaper to harness many weaker CPUs than one very powerful CPU. An additional motivation for using parallel computer systems is that the speed of serial computers will eventually begin to stabilize (Kumar, Grama, Gupta, & Karypis, 1994). This is partly due to the limitation of the speed of light, which states that nothing, including computer electronics, can travel faster than light.

Algorithm Parallelization

Parallel computer systems require parallel algorithms to achieve performance improvements. Most traditional algorithms have been developed using a serial approach that only makes use of one processor. These algorithms must be redesigned to make use

of multiple CPUs when they are available. The process of dividing an algorithm into smaller units is known as *task partitioning*. The ability of an algorithm to make use of additional processors effectively is known as the *parallelizability* of the algorithm. Some algorithms are naturally more parallelizable and can make use of more processors than others. The amount of task partitioning that can take place is the *degree* of parallelization (Kumar et al., 1994).

Processor Granularity

Most parallel computer systems available range along a spectrum of those having many weak processors to those with only a few very powerful processors. Systems with many (in the thousands of) weak processors are known as fine-grain systems. Coarse-grain systems have a small number (fewer than twenty) of very powerful processors available. Medium-grain systems fall between these two extremes with a medium number of processors with the power of standard PCs. The *granularity* of a particular system is the ratio of the time required for communication to the time required for computation (Kumar et al., 1994). When selecting a particular system for an algorithm, it is important to take into consideration both the computation power and the amount of communication that is required. When parallelizing a genetic algorithm, the granularity of the system is important, as will be seen with the different models of the PGA.

Parallel Genetic Algorithms

While one goal of a PGA is that of obtaining a final solution faster than that which is available using a serial system, an added benefit is the possibility of obtaining a *better* solution than is attainable using a serial GA. In nature, better organisms can be the result of maintaining diversity among a population. This can also be the case with GAs. Using parallel computation can help to promote diversity in a population. A number of methods of parallelization of genetic algorithms exist, a few of which will be discussed here.

Farmed Parallel Genetic Algorithm

The farmed PGA was the first of the parallel models to be developed. With this method, processors are divided among a master-slave architecture. One of the processors acts as the farmer and the other processors serve the farmer as workers. The farmer is responsible for maintaining the population and selecting which individuals are to be crossed. The worker processors perform the crossover and mutation operations. This design is illustrated in Figure 8 (Alba & Cotta, 1998).

The farmed PGA provides a very straightforward method of parallelization. It is easily scalable to many processors since numerous workers could be made use of. One great disadvantage, however, is the amount of costly communication that must take place between the farmer and worker processors. Since only a single population is maintained, additional diversity beyond that of the serial GA is not obtained.

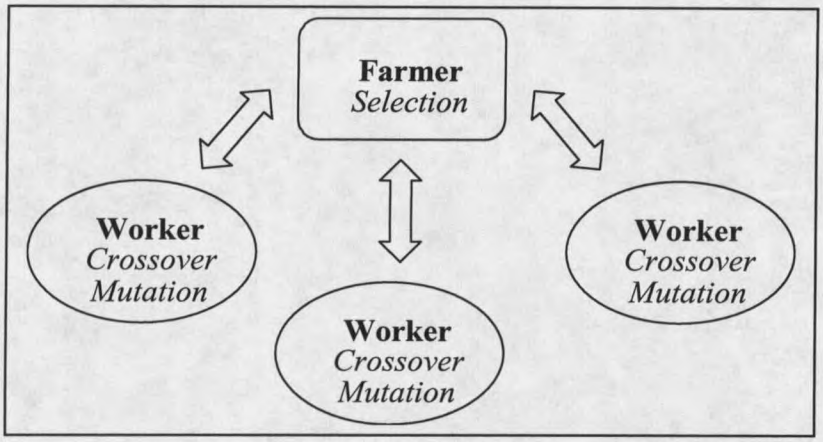


Figure 8: Farmed Parallel Genetic Algorithm Design

Cellular Parallel Genetic Algorithm

One method of parallelization of GAs that makes use of fine-grain parallelism is the cellular PGA. In this model, each individual is assigned to a different processor in the system. The individuals are laid out in some type of topology, which, hopefully, mimics the underlying processor topology of the computer system. The individuals are then restricted to crossover with their neighbors (Mühlenbein, 1989). After crossover, each child must replace one of the parent individuals in the population. Figure 9 shows a set of individuals in a two-dimensional mesh topology. One can see that if the individuals could only be crossed with their neighbors, those on opposite ends of the mesh would have little effect on each other.

The biggest advantage of the cellular PGA is the isolation of the subpopulations. Maintaining the various neighborhoods will help promote diversity (Alba & Cotta, 1998). Also, it is easy for this design to make use of numerous processors in a system. The

greatest disadvantage is that it requires much communication between processors. It also requires as many processors as there are individuals in the population, so it is not a very scalable design.

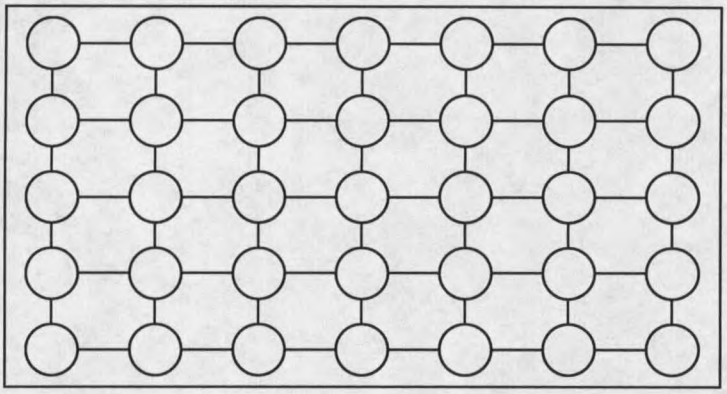


Figure 9: Cellular Parallel Genetic Algorithm Design

Island Parallel Genetic Algorithm

The island PGA is the most popular model among the GA parallelization methods. With an island PGA, a number of subpopulations are allowed to evolve in parallel. At the beginning of the program, the individuals are divided among a number of processors, each of which represents an island. Just as in nature, each island evolves its own individuals separate from the other islands. Occasionally, individuals are allowed to migrate to other islands to allow the good genes to spread (Lin, Punch, & Goodman, 1994). This algorithm is given in full in Figure 10 and an illustration is shown in Figure 11. In the layout of the islands in Figure 11, a ring topology is shown. Islands can only exchange individuals through migration with adjacent islands. The topology of the

islands determines this adjacency. It is best if the topology of the islands mimics that of the processors but it is not essential, since communication is rare.

```
 1) Randomly generate n individuals.
 2) Give each of the i islands n/i individuals.
 3) In parallel, loop once for each generation desired:
 4)    Calculate the fitness f(x) of each individual x.
 5)    Place the best number_elite individuals in new population.
 6)    Loop to generate (n - number_elite) new individuals:
 7)       Select two parent chromosomes.
 8)       Cross the parents to generate two new individuals.
 9)       With some probability, mutate the children.
10)       Place the two children into a new population.
11)    End loop.
12)    Replace the current population with the new population.
13)    If migration time, select individual(s) to send.
14)    Send individuals to and receive individuals from neighbors.
15) End loop.
16) The most fit individual among all islands is selected as the
    solution.
```

Figure 10: Island Parallel Genetic Algorithm

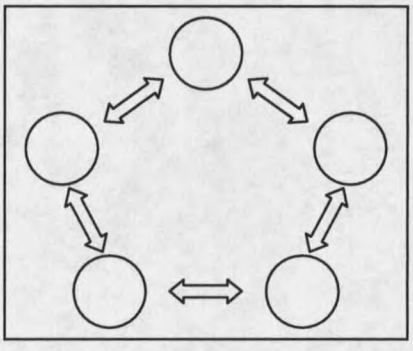


Figure 11: Island Parallel Genetic Algorithm Design

The island PGA model allows diversity to exist since different populations evolve independently from each other. It is also easily parallelizable to many processors since many islands could be maintained. Communication costs are low since the only communication required occurs at migration time. Since relatively few processors are

needed, each of which requires high computation and little communication with other processors, a medium or coarse-grain computer system should be selected. A drawback to this model is that it involves many additional parameters including the number of islands, the period between migrations, and the number of migrants allowed. Determining the optimal values for these parameters can be difficult (Alba & Cotta, 1998).

Speedup from Parallelism

One measure used to compare the execution times of parallel and serial algorithms is known as speedup. The speedup of a particular parallel algorithm is given by the formula

$$Speedup = \frac{Serial\_Time}{Parallel\_Time}.$$

When an algorithm is designed for a parallel computer with n processors, speedups of less than n are typical. This is due to the multiple processors having to be initialized and to communicate with each other. Speedup of more than n is known as *superlinear* speedup (Kumar et al., 1994). In subsequent chapters, the speedup of parallelizing GAs will be used to determine how much improvement in speed has been achieved.

CHAPTER 4

PARALLEL IMPLEMENTATION

Island Topologies

Five different variations of a genetic algorithm were implemented to solve both the traveling salesman and job-shop scheduling problems. The first was the serial GA, which makes use of only one processor and maintains only a single population of individuals. Four variations of the island model PGA were developed, each using eight islands of subpopulations. Each island is evolved on its own CPU. The island connectivity of the PGAs determines which islands are allowed to communicate via migration. In all of the GAs, a total of 128 individuals are used. This number of individuals falls within the range of 50 to 200, which is common in most GAs. With the PGAs, the 128 individuals are evenly distributed among the eight islands such that each island evolves 16 individuals. A single elite individual is saved by each subpopulation between generations to ensure that the islands never lose their best individuals. The serial model also saves a single elite individual between generations.

Migration

Depending on the topology of the islands used by the PGAs, individuals may be allowed to migrate between the islands. Whenever an individual is to migrate, it is selected from all of the island's individuals with ranked probabilities. Before migrating,

a copy of the individual is made. The original individual remains on the island while the copy emigrates. When an island receives a single immigrant, it replaces the weakest individual on that island. For the general case, the *n* immigrants received replace the *n* weakest individuals on the island. In all of the PGAs for this thesis, migration occurs after every ten generations. A single individual migrates along every line of connectivity.

Serial Model

The first GA variation implemented was the serial model shown in Figure 12, in which all 128 individuals are placed in a single population. Only one processor is used by the program and this implementation does not require the use of parallel processing. The single circle represents the single population of 128 individuals used by this model.

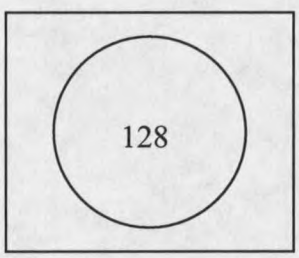


Figure 12: Single Population Topology

Parallel Models

The simplest island topology implemented was that with eight islands having no connections with any other processors. In this case, each island evolves independently from the others without any migration. This design is illustrated in Figure 13.

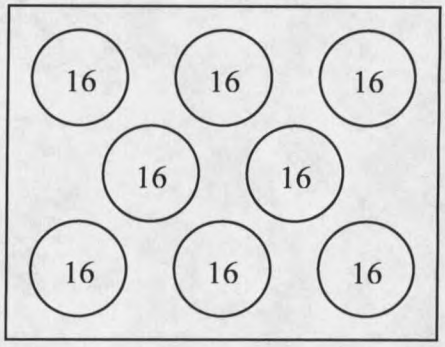


Figure 13: No Connectivity Island Topology

The second island topology examined represents a model with limited migration. To represent this layout, a unidirectional ring connects the islands. At migration time, a single individual migrates from each island to the next island in the ring where it replaces that island's weakest individual. This topology is illustrated in Figure 14.

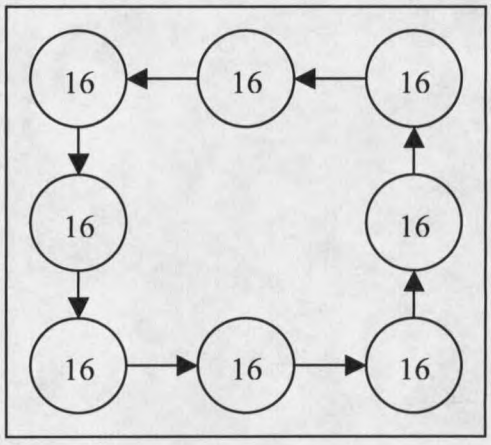


Figure 14: Unidirectional Ring Island Topology

The three-dimensional hypercube island topology, shown in Figure 15, represents a model having moderate connectivity. Each island is connected to three other islands. At migration time, each island sends out three individuals and receives three individuals.

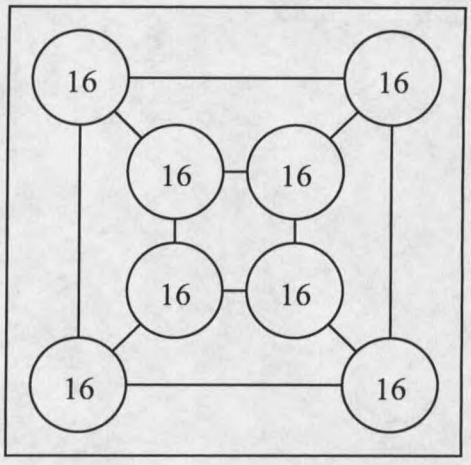


Figure 15: Three-Dimensional Hypercube Island Topology

The fully connected island model in Figure 16 represents high connectivity. Each island is connected to every other island. At migration time, each island sends a copy of one of its individuals to every other island. Each island receives a total of eight individuals at migration time.

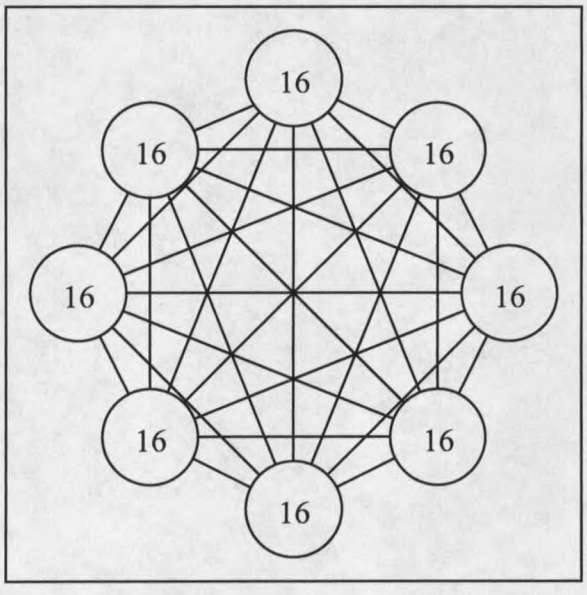


Figure 16: Full Connectivity Island Topology

Statistics

Throughout the GA process, two statistical values about the population are measured. The first is the best individual's fitness level. This is the individual that would be selected to serve as the solution to the problem if the genetic algorithm was terminated at that time. The second level recorded is the average fitness level of the population. The execution of each of the GAs is terminated after one hundred generations. The two statistical values are recorded every five generations. In the PGA models, an additional process is launched at the start of the program to record the statistics. After five generations, each of the islands communicates with this statistics process. Having this additional process allows statistics about the population (made of eight subpopulations) as a whole to be recorded. The PGA models, then, use a total of nine CPUs.

CHAPTER 5

TRAVELING SALESMAN PROBLEM

Problem Description

The *symmetric traveling salesman problem* (TSP) is defined as follows. Given a set $C = \{c_1, c_2, \ldots c_n\}$ of n cities and a set D of distances between each pair of cities (where the distance from city a to city b is the same as that from city b to city a), find the minimal length path that visits each city exactly once and ends with the same city it starts with.

The TSP has become a standard in the field of computer science because it is easy to describe and yet it falls under the nondeterministic polynomial (NP)-complete time complexity class, making it have high computational complexity. It has become one of the most popular problems used to test sequence optimization algorithms. One can see that the problem is a combinatorial optimization one since there is a finite set of possible answers that will solve it and one of those possibilities is the optimal solution. Since the TSP is combinatorial in nature, a GA is a good heuristic candidate that can be used to find a near-optimal solution. To do so involves developing a representation method for candidate solutions along with crossover and mutation operators to evolve the population.

Candidate Solution Representation

The most natural encoding method that can be used for the TSP is a many-character encoding. Each chromosome consists of an ordered list of city indices indicating the order in which the cities are to be visited. Figure 17 shows two examples of TSP paths and the many-character encoding chromosomes that can be used to represent those paths. It should be noted that the description of the TSP requires each path to end with the same city it starts with. Since this is always true, it is not necessary to indicate in the paths themselves that the final city is the same as the starting city.

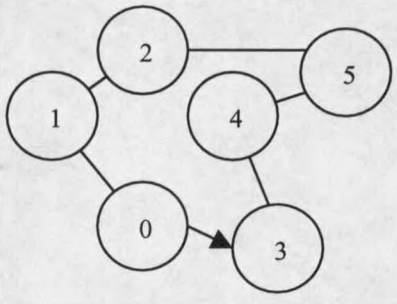
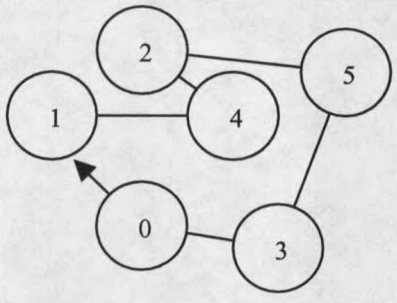
Solution		
Chromosome	0 3 4 5 2 1	0 1 4 2 5 3

Figure 17: TSP Chromosome Examples

Initial Population

As described in Chapter 2, starting with a population of fit individuals has proven effective compared to starting with a randomly generated population (Ahuja & Orlin, 1997). One method that can be used to generate fit individuals for the TSP is by using the farthest insertion algorithm to generate each of the initial chromosomes. This is a

traditional domain-specific algorithm that finds approximate TSP solutions. Figure 18 details the steps of this algorithm (Rosenkrantz, Stearns, & Lewis, 1977).

```
1) Create a subgraph a-a from a single city a.
2) Loop until all cities have been added to the subgraph:
3)    Find the city c that is not in the subgraph and that is
      farthest from any city in the subgraph.
4)    Find the path (a, b) in the subgraph that minimizes dist_ac +
      dist_cb - dist_ab.
5)    Insert city c between cities a and b in the subgraph.
6) End loop.
```

Figure 18: Farthest Insertion Algorithm

The algorithm begins by adding some initial city to the subgraph. If the algorithm was always initialized with the same city, all of the individuals generated would be exactly the same. To prevent this, the starting city is selected randomly. The worst case of the quality of the solutions generated using farthest insertion has been proven to be

$$\frac{farthest\_insertion}{optimal} \leq 2\ln(number\_cities) + 0.16,$$

making the farthest insertion algorithm a good algorithm for quickly obtaining high quality TSP paths (Rosenkrantz et al., 1977).

Selection of Parent Individuals

The fitness of an individual is equal to the total distance of the path it represents. When selecting the two parents to be crossed, the first is selected using rank selection (as described in Chapter 2). To select the second parent, all of the population's individuals are assigned equal probabilities of selection. This ensures that the weak individuals have

a good chance of being selected as at least one of the parents. Since the initial individuals are generated using farthest insertion, it can be assumed that even those that are ranked as weak are still quite fit.

Crossover Operator

A number of different crossover methods exist for the TSP. The crossover method selected for this thesis was the partially mapped crossover. It ensures that the children receive some of the ordering of the cities from both of the parents. Figure 19 describes the algorithm in full (Goldberg & Lingle, 1985).

```
1) Select two cut points within the list of cities (the regions
   between the points are the mapping sections).
2) Copy the mapping section of the first parent to the second
   child.
3) Copy the mapping section of the second parent to the first
   child.
4) Copy into child 1 the elements from parent 1.
5) Copy into child 2 the elements from parent 2.
6) During steps 4 and 5, if the city being added from the parent
   is already present in the child, follow the mapping section
   until a city that has not been added is located.
```

Figure 19: Partially Mapped Crossover Algorithm

It should be noted that the absolute positions of some elements of both parents are preserved (Larrañaga, Kuijpers, Murga, Inza, & Dizdarevich, 1999). An example of the partially mapped crossover is shown in Figure 20. The mapping sections have been marked in both the parents and the children. When building the first child (in the left column), the mapping section |3 4 1| is first copied to it from the second parent (in the right column). The cities from the first parent are then to be copied to the child. If a city

is already in the child, the mapping section is followed to find the next unvisited city. Since city 3 has already been added, the mapping 3 ↔ 1 can be followed. Again, since city 1 has already been added, the mapping 1 ↔ 0 is followed. City 0 has not yet been added so it is. This pattern is followed until both of the children have been generated.

Parents	3 4 | 1 2 0 | 5 6	0 5 | 3 4 1 | 2 6
Children	0 2 | 3 4 1 | 5 6	3 5 | 1 2 0 | 4 6

Figure 20: Partially Mapped Crossover Example

Mutation Operator

Like crossover, a number of methods for mutation of chromosomes exist for the TSP. The simplest is the exchange mutation and it has been selected as the mutation operator for this thesis. It works as shown in Figure 21 (Banzhaf, 1990). Figure 22 shows an example of the exchange mutation occurring with a path of six cities.

```
Randomly select two cities in the path.
Swap the positions of the two cities.
```

Figure 21: Exchange Mutation Algorithm

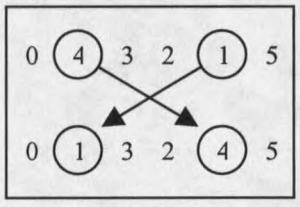


Figure 22: Exchange Mutation Example

To improve the quality of the GAs, a dynamic mutation rate has been implemented. This allows the mutation rate to change dynamically as the population of individuals becomes more or less similar. At the beginning of the program, the mutation rate is set at 0.1 so that one in ten individuals created from crossover will be mutated. When the percentage of difference between the population's best and worst individuals' fitness levels becomes very small, the mutation rate is increased by a factor of 0.05. The highest level of mutation allowed is 0.4. The mutation rate is decreased by 0.05 when the percentage of difference between the best and worst individuals' fitness levels in a population becomes large. A mutation rate of 0.4 is quite high considering that prior research has shown that mutation can cause the average fitness of a population to worsen. Since the goal of the GAs for this thesis is only to find a single good solution, the average fitness level of the population is not of concern. The dynamic mutation rate has been implemented in order to allow new solutions to be discovered that would not likely be obtained with standard crossover or low mutation rates.

Local Optimization

After generating new individuals from crossover or mutation, local optimization, or hill-climbing, can be used to improve the overall quality of the GA (Ahuja & Orlin, 1997). One method for TSP local optimization is the 2-Optimal (or 2-Opt) algorithm. It improves a tour by repeatedly exchanging two edges at a time when such an exchange reduces the path length. This exchange process continues until no better solution can be found by exchanging edges (Lin, 1965). The algorithm is described in Figure 23 (Syslo,

Deo, & Kowalik, 1983). An example of swapping two edges to improve a path length is shown in Figure 24. The 2-Opt algorithm is used to possibly improve every individual after it has been generated from crossover or mutated.

```
1) Let E = (e1, e2, …, en) be the undirected edges in the current
   path where each edge ei is made up of both ei-start and ei-
   finish city index values.
2) Loop:
3)    λ = 0.
4)    Loop for i=1 to n-2 do:
5)       Loop for j=i+2 to n (or n-1 when i=1) do:
6)          Let new_edge_1 = edge from ei-start to ej-start.
7)          Let new_edge_2 = edge from ei-finish to ej-finish.
8)          If ((ei + ej) - (new_edge_1 + new_edge_2) > λ):
9)             λ= ((ei + ej) - (new_edge_1 + new_edge_2).
10)            Save i and j for later.
11)         End if.
12)      End loop.
13)   End loop.
14)   If (λ > 0)
15)      Remove edges ei and ej from E.
16)      Add edges new_edge_1 and new_edge_2 to E.
17)   End if.
18) Until λ = 0.
```

Figure 23: 2-Opt Local Optimization Algorithm

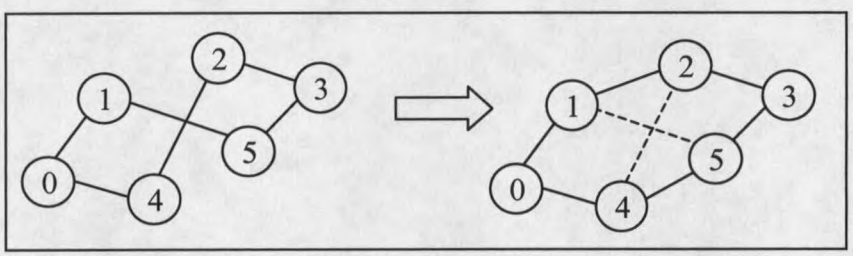


Figure 24: Exchange of Edges by the 2-Opt Algorithm Example

Data Sets

To test the effectiveness of the four variations of the PGA compared to the serial GA at solving the TSP, three data sets were selected from the TSPLIB 95 (Reinelt, n.d.). The library contains TSP data files that are popular throughout the literature. The files selected were *a280*, *pcb442*, and *att532*. The numbers of cities within the data sets are specified in the names of the files. The files were selected solely for the numbers of cities they contain so that three different sizes of files would be represented in the experiments and results.

Data Collection

As described in Chapter 4, two statistical values are recorded throughout the execution of the GAs. The first is the fitness level of the best individual in the population. The second is the average fitness level of the entire population. To determine which of the parallelization models can be expected to obtain the highest quality TSP solutions, the five models were all executed on the same data sets. For every test variation performed, the program was executed a total of ten times. Each time, the program's random number generator was seeded with a different value. Each element of data shown in the following tables represents the average of these ten runs.

One should note that the data shown in this chapter is of a concise form. The fitness levels of only four of the one hundred generations have been listed. Graphs of all

of the generations' fitness levels can be viewed in Appendix A. These graphs can be useful in making comparisons of the five models.

Since the data sets used in these experiments are popular in the TSP literature, the optimum solution qualities have already been proven. Therefore, the error percentages from the optimum values can be calculated.

Best Individual Fitness Levels

Tables 3, 4, and 5 show the fitness levels of the best candidate solutions found throughout the execution of the GAs for the three data sets. The levels obtained at four generations for each of the selected data sets are shown. Since the object of the TSP is to find a minimal length path, those individuals with fitness levels that are low are superior to those with high levels. The fitness level of the model that performed the best in each of the generations has been marked with an asterisk. The amount of improvement in error percentage made from the serial GA to the best performing PGA model during the one-hundredth generation for each of the data sets is recorded in the data tables.

When solving both the *a280* and *att532* data sets, the fully connected PGA model exhibited the best performance in the one-hundredth generation. The three-dimensional hypercube model obtained the highest quality fitness levels for the *pcb442* data set. Compared to the serial models, the solution quality improvements for the best individual fitness levels were 100.00% (for *a280*), 6.49% (for *pcb442*), and 21.21% (for *att532*) during the one-hundredth generation. Overall the fully connected PGA model can be recommended for obtaining high quality individuals when solving the TSP.

	Data Set: a280 Optimum: 2,579									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	2590.1	0.43	2599.8	0.81	2597.9	0.73	2592.7	0.53	2586.4*	0.29
50	2582.3	0.13	2597.4	0.71	2584.0	0.19	2580.1	0.04	2579.0*	0.00
75	2582.3	0.13	2597.4	0.71	2579.6	0.02	2579.2	0.01	2579.0*	0.00
100	2582.3	0.13	2594.6	0.60	2579.6	0.02	2579.2	0.01	2579.0*	0.00
Best Error Percentage in Generation 100: Parallel Full Connectivity Improvement Over Serial: 100.00%										

Table 3: Best Individual Fitness Per Generation for TSP *a280*

	Data Set: pcb442 Optimum: 50,778									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	52049.7*	2.50	52574.1	3.54	52147.8	2.70	52143.4	2.69	52122.3	2.65
50	51616.1	1.65	52499.9	3.39	51747.0	1.91	51590.3	1.60	51520.2*	1.46
75	51335.0	1.10	52499.9	3.39	51516.0	1.45	51237.6*	0.91	51319.2	1.07
100	51169.3	0.77	52499.9	3.39	51369.6	1.17	51143.5*	0.72	51274.5	0.98
Best Error Percentage in Generation 100: Parallel 3-D Hypercube Error Percentage Improvement Over Serial: 6.49%										

Table 4: Best Individual Fitness Per Generation for TSP *pcb442*

	Data Set: att532 Optimum: 27,686									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	28268.2*	2.10	28445.4	2.74	28374.7	2.49	28285.5	2.17	28280.0	2.15
50	28103.8	1.51	28248.7	2.03	28134.9	1.62	28035.7	1.26	27989.1*	1.09
75	27950.7	0.96	28239.7	2.00	27991.0	1.10	27908.9	0.81	27875.5*	0.68
100	27868.4	0.66	28239.7	2.00	27926.4	0.87	27855.8	0.61	27830.3*	0.52
Best Error Percentage in Generation 100: Parallel Full Connectivity Error Percentage Improvement Over Serial: 21.21%										

Table 5: Best Individual Fitness Per Generation for TSP *att532*

Average Population Fitness Levels

Along with the best individual fitness levels, the average population fitness levels obtained at the four selected generations during the GAs have also been recorded. When executing a GA, the average population fitness level can be expected to improve as the population evolves. Tables 6, 7, and 8 show the average population fitness levels obtained by the GAs for the three TSP data sets.

In these experiments, one can see that the fully connected PGA model obtained the highest average population fitness levels by the one-hundredth generation when solving both the *a280* and *att532* data sets while the three-dimensional hypercube model performed the best when solving the *pcb442* data set. These models had error percentage improvements over the serial models of 87.50% (for *a280*), 50.57% (for *pcb442*), and 52.42% (for *att532*). Again, the fully connected PGA model can be recommended for obtaining high average population fitness levels when solving the TSP.

	Data Set: a280 Optimum: 2,579									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	2607.8	1.12	2619.3	1.56	2613.1	1.32	2610.1	1.21	2601.9*	0.89
50	2586.6	0.29	2615.1	1.40	2592.9	0.54	2584.7	0.22	2580.5*	0.06
75	2582.9	0.15	2614.3	1.37	2583.6	0.18	2579.7	0.03	2579.6*	0.02
100	2583.0	0.16	2613.6	1.34	2580.7	0.07	2579.6	0.02	2579.5*	0.02
Best Error Percentage in Generation 100: Parallel Full Connectivity Error Percentage Improvement Over Serial: 87.50%										

Table 6: Average Population Fitness Per Generation for TSP *a280*

	Data Set: pcb442 Optimum: 50,778									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	53284.3	4.94	52918.8	4.22	52761.3	3.91	52791.4	3.97	52744.1*	3.87
50	52894.4	4.17	52751.4	3.89	52158.3	2.72	51971.2*	2.35	51797.9	2.01
75	52298.5	2.99	52736.9	3.86	51893.1	2.20	51454.6	1.33	51410.9*	1.25
100	51672.7	1.76	52732.3	3.85	51643.1	1.70	51219.6*	0.87	51292.1	1.01
Best Error Percentage in Generation 100: Parallel 3-D Hypercube Error Percentage Improvement Over Serial: 50.57%										

Table 7: Average Population Fitness Per Generation for TSP *pcb442*

	Data Set: att532 Optimum: 27,686									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	28832.7	4.14	28680.3	3.59	28654.9	3.50	28641.1	3.45	28632.7*	3.42
50	28596.9	3.29	28391.3	2.55	28339.3	2.36	28192.2	1.83	28146.9*	1.66
75	28268.7	2.10	28381.5	2.51	28146.8	1.66	28009.9	1.17	27931.8*	0.89
100	28029.2	1.24	28379.3	2.50	28028.1	1.24	27888.4	0.73	27849.6*	0.59
Best Error Percentage in Generation 100: Parallel Full Connectivity										
Error Percentage Improvement Over Serial: 52.42%										

Table 8: Average Population Fitness Per Generation for TSP *att532*

Execution Times

Along with the statistics about the populations, the times required to run all of the programs have been recorded. Two different variations of time are important for examination. First is the elapsed time, which measures the total amount of time that passes between the start and the finish of a program. In contrast, the CPU time of a program measures the total amount of computing time that occurs on all processors.

Table 9 shows the elapsed time averages for the serial and parallel models. One can expect the elapsed times of the serial models to be greater than any of the parallel models since they only make use of a single processor. Table 10 shows the total CPU time averages for each of the models. The numbers in this table are much more similar to each other than those in Table 9 since the CPU time reflects the total computation time of all processors. A percentage value of the serial version has been supplied for each entry.

The execution times required by the PGA models that exhibited the best individual fitness levels by the one-hundredth generation should be noted. They required 12.1% (by the fully connected model for *a280*), 7.2% (by the three-dimensional hypercube model for *pcb442*), and 7.7% (by the fully connected model for *att532*) of the elapsed times required by the serial model. Using nine processors, the fully connected model achieved superlinear speedups when obtaining solutions for *pcb442* and *att532*. These speedups could be due to each of the islands having to maintain fewer individuals than the serial model. Those models used 109.5% (by the fully connected model for *a280*), 64.9% (by the three-dimensional hypercube model for *pcb442*), and 70.9% (by the fully connected model for *att532*) of the CPU time used by the serial model.

	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Data Set	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*
a280	434.8	100.0	38.3	8.8	44.7	10.3	49.1	11.3	52.7	12.1
pcb442	5833.0	100.0	175.6	3.0	390.4	6.7	418.4	7.2	374.4	6.4
att532	6694.1	100.0	317.8	4.7	502.3	7.5	525.5	7.9	514.0	7.7

Table 9: Elapsed Times Per Data Set for TSP

	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Data Set	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*
a280	429.2	100.0	340.9	79.4	398.5	92.8	438.2	102.1	469.8	109.5
pcb442	5774.5	100.0	1572.0	27.2	3499.1	60.6	3749.1	64.9	3354.3	58.1
att532	6571.8	100.0	2846.6	43.3	4501.5	68.5	4709.4	71.7	4605.9	70.9

Table 10: CPU Times Per Data Set for TSP

Conclusion

Having executed the different island topology models on three TSP data sets, it can be concluded that the model that exhibited the best overall performance was the fully connected PGA model. The three-dimensional hypercube model closely follows this. The serial model usually performed with the fourth highest solution qualities, performing only better than the PGA model with no island connectivity. Parallelizing the GA for solving the TSP has proven to be an effective method for obtaining higher quality solutions faster than is possible with a serial GA.

In order to determine if the parallel model with full connectivity obtained error percentages that were statistically better than the serial model, a Wilcoxon signed rank test has been performed. A Wilcoxon test can be used to compare the solution qualities obtained by two different algorithms. The quality of each algorithm is determined by the solutions it obtained at all four examined generations for each of the three data sets. The Wilcoxon test revealed that the fully connected PGA's ability to obtain higher quality best individuals than the serial model approached significance. The fully connected PGA was able to obtain average population fitness levels that were statistically better than the serial model. The complete calculations of these Wilcoxon tests can be viewed in Appendix B.

CHAPTER 6

JOB-SHOP SCHEDULING PROBLEM

Problem Description

The job-shop scheduling problem is defined as follows. Given a set $J = \{j_1, j_2, \ldots j_n\}$ of n jobs, where each job j_i consists of a set $O = \{o_1, o_2, \ldots o_m\}$ of m ordered operations specifying one of m machines from set $M = \{1, 2, \ldots m\}$ and the amount of uninterrupted time required for processing on that machine, find the ordering of the operations on the machines that requires the minimum makespan. The *makespan* of a schedule is the amount of time required for all of the scheduled machines to finish processing their assigned operations. At any time, a machine can only be processing a single operation. Only one operation of a job may be processed at a time.

The JSSP is, perhaps, the most general of the scheduling problems. Like the TSP, the JSSP is combinatorial in nature, having an optimal answer among a discrete set of possibilities. The JSSP is also an NP-complete time complexity problem.

Candidate Solution Representation

For the JSSP, a many-character encoding is the most natural. Each chromosome can be stored as a list of scheduled machines. The schedule for a particular machine is the list of operations that are to be performed on it and the times at which they take place. Table 11 shows an example of the type of data used to describe a JSSP, consisting of a

Job	*ID*	*Machine*	*Time*	*ID*	*Machine*	*Time*	*ID*	*Machine*	*Time*
1	1	1	15	2	2	17	3	3	12
2	4	3	13	5	1	20	6	2	15
3	7	2	20	8	3	21	9	1	19

Table 11: JSSP Input Example

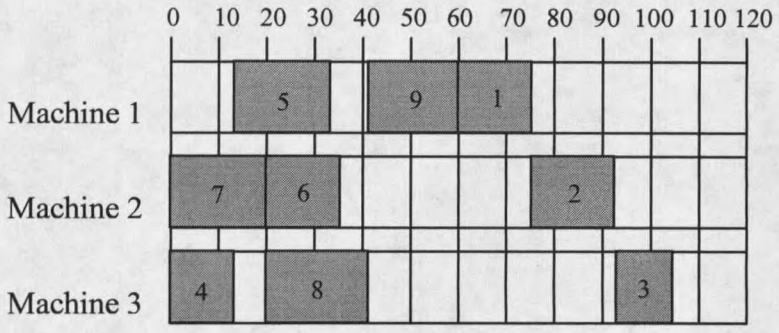
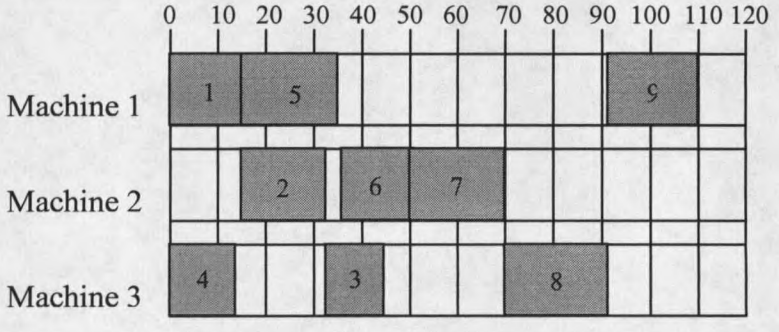
Individual 1	
Chromosome	1: 5 (13-33), 9 (41-60), 1 (60-75) 2: 7 (0-20), 6 (20-35), 3 (75-92) 3: 4 (0-13), 8 (20-41), 2 (92-104)
Solution Represented	
Makespan	104
Individual 2	
Chromosome	1: 1 (0-15), 5 (15-35), 9 (91-110) 2: 2 (15-32), 6 (35-50), 7 (50-70) 3: 4 (0-13), 3 (32-44), 8 (70-91)
Solution Represented	
Makespan	110

Figure 25: JSSP Chromosome Examples

set of jobs and various operations to be performed (each operation has been assigned a unique identification number). Figure 25 shows two example chromosomes that could be built from the supplied input.

Initial Population

One of the difficulties of the JSSP is that of ensuring that the individuals are valid. An algorithm proposed by Giffler and Thompson can be used to generate a valid schedule given a list of jobs to be performed. The algorithm works as shown in Figure 26 (Giffler & Thompson, 1960):

```
1) Loop:
2)     Build READY as the set of all of the earliest unscheduled
       operations among the jobs.
3)     Find operation early which is the operation with the
       earliest completion time among READY.
4)     Let mach be the machine required by operation early.
5)     Build CONFLICT as the set of all of the operations in READY
       that require machine mach and have a start time before the
       earliest completion time of early.
6)     Select an operation assign from CONFLICT.
7)     Schedule assign as the next operation on machine m with a
       completion time set to the earliest completion time for
       assign.
8) Until all operations have been scheduled.
```

Figure 26: Giffler-Thompson Algorithm

Figure 27 shows an example of one step in the Giffler-Thompson algorithm. Each of the rectangles represents an operation in the READY set. One of the operations on machine 1 has the earliest completion time. That operation is named *early*. Therefore, all of the other operations that require machine 1 and have a start time that is before the completion time of *early* are part of the CONFLICT set.

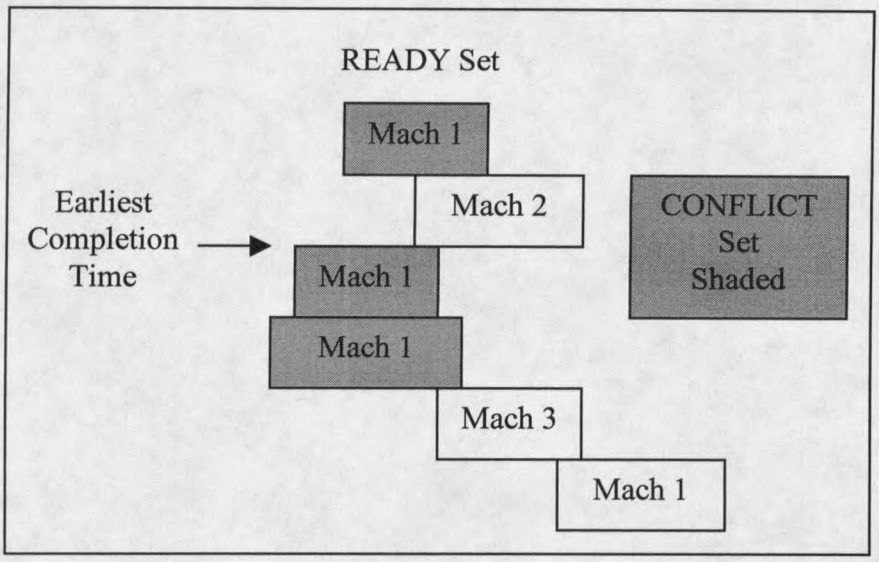


Figure 27: Giffler-Thompson Algorithm Example

Selection of Parent Individuals

As with the TSP, the first parent supplied to the crossover operator is selected using rank selection while the second is selected from all individuals having an equal probability of selection. The fitness of an individual is equal to the makespan of the schedule it represents.

Crossover Operator

For a crossover operator, the children individuals should receive some characteristics from each of the parent individuals. A variation of the Giffler-Thompson algorithm can be used to ensure that the children individuals generated are valid. This algorithm works as shown in Figure 28. The matrix CHOOSE should be generated in a

random fashion and it is used to determine which scheduled operations are obtained from which parents. If the two schedules are identical to each other, the children generated will be identical. As the algorithm is, only a single child can be generated from the parents. However, by creating a second matrix that contains the opposite values found in the CHOOSE matrix, a second child can be created to complement the first. Based on the values found in the CHOOSE matrix, a child could resemble one parent more than the other (Yamada & Nakano, 1992).

```
1) Let CHOOSE be a matrix of 1's and 2's generated randomly.
   CHOOSE should have dimensions of number_operations rows and
   Number_machines columns.
2) Loop:
3)    Build READY as the set of all of the earliest unscheduled
      operations among the jobs.
4)    Find operation early which is the operation with the
      earliest completion time among READY.
5)    Let mach be the machine required by operation early.
6)    Build CONFLICT as the set of all of the operations in
      READY that require machine mach and have a start time
      before the earliest completion time of early.
7)    Let next_op equal the number of current operations
      assigned to
      mach + 1.
8)    Examine matrix CHOOSE.
         If CHOOSE[next_op][mach] is 1, select the operation
            assign from CONFLICT that is the earliest in
            parent 1.
         If CHOOSE[next_op][mach] is 2, select the operation
            assign from CONFLICT that is the earliest in
            parent 2.
9)    Schedule assign as the next operation on machine m with a
      completion time set to the earliest completion time for
      assign.
10) Until all operations have been scheduled.
```

Figure 28: Giffler-Thompson Crossover Algorithm

Mutation Operator

Mutation has not been implemented in this genetic algorithm. This is consistent with the JSSP literature.

Data Sets

For the job-shop scheduling problem, three data sets of different sizes were selected from the Operations Research (OR) Library (Beasley, n.d.). Those files selected were *la20* (10 jobs, 10 operations), *la30* (20 jobs, 10 operations), and *la35* (30 jobs, 10 operations) (Lawrence, 1984). Since these data sets are popular throughout the JSSP literature, their optimal solution makespans are known.

Data Collection

For the JSSP, the same statistical values are recorded as with the TSP. The first is the fitness level of the population's best individual. The second is the population's average fitness level. Every test variation was tested a total of ten times. Each time, the program's random number generator was seeded with a different value. Each table element here represents the average of ten runs. Graphs of both the best and average fitness levels can be found in Appendix A.

Best Individual Fitness Levels

Tables 12, 13, and 14 show the fitness levels of the best individuals found by the five different models at four generations throughout the GAs. The fitness level of the model that performed the best in each generation has been marked with an asterisk. The percentage of error from the optimal value has also been indicated beside each fitness level.

One can see that the unidirectional ring PGA model obtained the best individual fitness level when solving the *la20* data set while the fully connected model exhibited the best performance when solving the *la30* and *la35*. These PGA models had error improvements over the serial models of 28.42% (for *la20*), 28.60% (for *la30*), and 64.50% (for *la35*). As the data set size increased, the effects of parallelizing the GA were more pronounced.

	Data Set: la20 Optimum: 902									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	934.2*	3.57	957.0	6.10	945.3	4.80	940.6	4.28	935.8	3.75
50	927.7	2.85	957.0	6.10	927.7	2.85	929.7	3.07	924.3*	2.47
75	927.7	2.85	957.0	6.10	922.1*	2.23	923.5	2.38	924.3	2.47
100	927.7	2.85	957.0	6.10	920.4*	2.04	922.0	2.22	924.3	2.47
Best Error Percentage in Generation 100: Parallel Unidirectional Ring Error Percentage Improvement Over Serial: 28.42%										

Table 12: Best Individual Fitness Per Generation for JSSP *la20*

	Data Set: la30 Optimum: 1,355									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	1436.2	5.99	1433.5	5.79	1437.7	6.10	1432.2*	5.70	1433.8	5.82
50	1418.7	4.70	1432.9	5.75	1420.0	4.80	1422.2	4.96	1409.2*	4.00
75	1416.0	4.50	1432.9	5.75	1414.9	4.42	1412.5	4.24	1399.3*	3.27
100	1415.1	4.44	1432.9	5.75	1412.5	4.24	1403.3	3.56	1397.9*	3.17
Best Error Percentage in Generation 100: Parallel Full Connectivity Error Percentage Improvement Over Serial: 28.60%										

Table 13: Best Individual Fitness Per Generation for JSSP *la30*

	Data Set: la35 Optimum: 1,888									
	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Gen	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*	*Fitness*	*Error (%)*
25	1941.0	2.81	1923.6*	1.89	1929.7	2.21	1939.3	2.72	1934.4	2.46
50	1925.8	2.00	1921.6	1.78	1921.7	1.78	1921.8	1.79	1904.9*	0.90
75	1920.0	1.69	1921.6	1.78	1918.5	1.62	1909.3	1.13	1902.3*	0.76
100	1920.0	1.69	1921.6	1.78	1913.6	1.36	1903.4	0.82	1899.4*	0.60
Best Error Percentage in Generation 100: Parallel Full Connectivity Error Percentage Improvement Over Serial: 64.50%										

Table 14: Best Individual Fitness Per Generation for JSSP *la35*

Average Population Fitness Levels

Tables 15, 16, and 17 contain the average population fitness levels obtained by the five models at four generations of the GAs. The fully connected PGA exhibited the best performance in the one-hundredth generation with error percentage improvements of 27.25% (for *la20*), 34.63% (for *la30*), and 49.21% (for *la35*) over the serial model.

Data Set: la20 Optimum: 902										
	Serial		Parallel No Connectivity		Parallel Unidirectional Ring		Parallel 3-D Hypercube		Parallel Full Connectivity	
Gen	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)
25	998.3	10.68	1008.0	11.75	995.4*	10.36	1005.8	11.51	1001.0	10.98
50	950.0	5.32	994.0	10.20	970.6	7.61	961.7	6.62	933.6*	3.50
75	940.5	4.27	991.0	9.87	944.4	4.70	937.7	3.96	927.0*	2.77
100	934.1	3.56	991.0	9.87	933.3	3.47	927.5	2.83	925.4*	2.59
Best Error Percentage in Generation 100: Parallel Full Connectivity Error Percentage Improvement Over Serial: 27.25%										

Table 15: Average Population Fitness Per Generation for JSSP *la20*

Data Set: la30 Optimum: 1,355										
	Serial		Parallel No Connectivity		Parallel Unidirectional Ring		Parallel 3-D Hypercube		Parallel Full Connectivity	
Gen	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)
25	1518.0	12.03	1514.3*	11.76	1531.3	13.01	1524.1	12.48	1520.7	12.23
50	1469.3	8.44	1485.1	9.60	1478.2	9.09	1477.6	9.05	1462.4*	7.93
75	1444.0	6.57	1479.5	9.19	1458.6	7.65	1463.1	7.98	1425.0*	5.17
100	1428.2	5.40	1479.5	9.19	1449.4	6.97	1425.4	5.20	1402.8*	3.53
Best Error Percentage in Generation 100: Parallel Full Connectivity Error Percentage Improvement Over Serial: 34.63%										

Table 16: Average Population Fitness Per Generation for JSSP *la30*

Data Set: la35 Optimum: 1,888										
	Serial		Parallel No Connectivity		Parallel Unidirectional Ring		Parallel 3-D Hypercube		Parallel Full Connectivity	
Gen	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)	Fitness	Error (%)
25	2019.2*	6.95	2038.3	7.96	2035.8	7.83	2037.9	7.94	2041.0	8.10
50	1971.8	4.44	2000.8	5.97	1989.1	5.35	2004.0	6.14	1965.2*	4.09
75	1948.9	3.23	1997.5	5.80	1979.1	4.82	1987.3	5.26	1928.1*	2.12
100	1935.9	2.54	1997.0	5.77	1964.1	4.03	1962.8	3.96	1912.4*	1.29
Best Error Percentage in Generation 100: Parallel Full Connectivity Error Percentage Improvement Over Serial: 49.21%										

Table 17: Average Population Fitness Per Generation for JSSP *la35*

Execution Times

Along with the fitness level statistics, the times to run all of the models used to solve the JSSP have also been recorded. Table 18 shows the elapsed times and Table 19 shows the CPU times for the five models. One interesting note is that when solving the TSP, the PGAs required less CPU time than the serial GAs. When solving the JSSP, the PGAs require more than the serial GAs.

The execution times required by the models that obtained the best individual fitness levels by the one-hundredth generation should be noted. They required 15.1% (by the unidirectional ring model for *la20*), 13.4% (by the fully connected model for *la30*), and 13.1% (by the fully connected model for *la35*) of the total elapsed times required by the serial model. The top-performing PGA models used 122.2% (by the unidirectional ring model for *la20*), 117.2% (by the fully connected model for *la30*), and 117.2% (by the fully connected model for *la35*) of the CPU time used by the serial model.

	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hypercube*		*Parallel Full Connectivity*	
Data Set	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*	*Elaps. Time (sec)*	*% of Ser.*
la20	11.9	100.0	1.8	15.1	1.8	15.1	1.8	15.1	1.8	15.1
la30	47.9	100.0	6.3	13.2	6.3	13.2	6.4	13.4	6.4	13.4
la35	102.7	100.0	13.5	13.1	13.5	13.1	13.5	13.1	13.5	13.1

Table 18: Elapsed Times Per Data Set for JSSP

	Serial		*Parallel No Connectivity*		*Parallel Unidirectional Ring*		*Parallel 3-D Hyperdube*		*Parallel Full Connectivity*	
Data Set	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*	*CPU Time (sec)*	*% of Ser.*
la20	11.7	100.0	14.4	123.1	14.3	122.2	14.3	122.2	14.3	122.2
la30	47.2	100.0	54.8	116.1	54.8	116.1	54.9	116.3	55.3	117.2
la35	101.7	100.0	119.1	117.1	119.0	117.0	119.1	117.0	119.2	117.2

Table 19: CPU Times Per Data Set for JSSP

Conclusion

As with the TSP, the fully connected PGA exhibited the best ability at obtaining both best individual and average population fitness levels by the one-hundredth generation. It also was the best performer the most number of times in all four of the generations examined. Parallelizing the GA used to solve the JSSP has been shown to be an effective method for quickly obtaining high quality solutions. Wilcoxon tests in Appendix B have shown that the parallel models with full connectivity were able to obtain statistically better best individual and average population solutions than the serial models.

CHAPTER 7

CONCLUSION

Genetic algorithms have proven to be effective methods for finding near-optimal solutions to combinatorial optimization problems. By providing a flexible encoding system for solutions and using the ideas of natural selection to choose fit individuals for crossover, the GA can quickly find a good solution to a problem within even a very large search space.

Genetic algorithms to solve the traveling salesman and job-shop scheduling problems have been implemented. One serial and four island model parallel GAs have been designed to solve each of the problems. Each of the parallel models maintains a different topology of its islands, allowing individuals to migrate along the island connections.

When solving the TSP, the best parallel models obtained best individual error percentages for the three data sets that were 100.00%, 6.49%, and 21.21% better than the corresponding serial models in the one-hundredth generation. These PGA models used only 12.1%, 7.2%, and 7.7% of the elapsed times used by the serial GA models. The top performing parallel models' best individual error percentages exhibited improvements of 28.42%, 28.60%, and 64.50% over the serial models for the three examined JSSP data sets. These PGA models' percentages of elapsed times required by the serial models were 15.1%, 13.4%, and 13.1%. The fully connected PGA exhibited the best

performance for obtaining both best individual fitness levels and average population fitness levels.

The TSP and JSSP problems have been selected as representatives of combinatorial optimization problems. The results shown here can hopefully be extended to other types of problems that are combinatorial in nature. Additional work with other types of problems could help solidify the ideas presented.

GAs themselves will most likely continue to be a viable method for solving problems for which more direct algorithms do not exist. Additional ideas from nature, such as sexual differentiation of individuals and predator-prey relationships, could be beneficial in GAs.

Parallel computing will likely continue in its popularity for solving problems that require a great deal of time. Since a GA can require much computational time, it is well suited for implementation on parallel computing systems. PGAs are capable of obtaining both faster execution times and higher quality solutions by fully exploiting the powers of parallel computer systems, making them a powerful technique for solving problems.

REFERENCES CITED

Ahuja, R. and Orlin, J. (1997). Developing fitter genetic algorithms. *INFORM Journal on Computing, 9*(3), 251-253.

Alba, E. and Cotta, C. (1998). *The on-line tutorial on evolutionary computation.* Department of Languages and Sciences of the Computation, University of Málaga, Spain. Retrieved September 21, 2000, from the World Wide Web: http://polaris.lcc.uma.es/~ccottap/semEC/.

Banzhaf, W. (1990). The "molecular" traveling salesman. *Biological Cybernetics, 64*, 7-14.

Beasley, J. (n.d.). *OR-Library.* Imperial College Management School, United Kingdom. Retrieved January 15, 2001, from the World Wide Web: http://mscmga.ms.ic.ac.uk/info.html.

Forrest, S. and Mitchell, M. (1993). What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Machine Learning, 13*, 285-319.

Giffler, B., and Thompson, G. (1960). Algorithms for solving production scheduling problems. *Operations Research, 8*, 487-503.

Goldberg, D., and Lingle, R. (1985). Alleles, loci and the TSP. In J. Grefenstette (Ed.) *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 154-159). Hillsdale, NJ: Lawrence Erlbaum.

Holland, J. (1975). *Adaptation in natural and artificial systems.* Ann Arbor, MI: The University of Michigan Press.

Kershbaum, A. (1997). When genetic algorithms work best. *INFORM Journal on Computing 9*(3), 254-255.

Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). *Introduction to Parallel Computing.* Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc.

Lawrence, S. (1984). *Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (Supplement).* Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

Larrañaga, P., Kuijpers, R., Murga, R., Inza, I., and Dizdarevich, S. (1999). Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review 13*, 129-170.

Levine, D. (1997). Genetic algorithms: A practitioner's view. *INFORM Journal on Computing, 9*(3), 256-259.

Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal, 44*, 2245-2269.

Lin, S., Punch, W., & Goodman, E. (1994). Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Sixth IEEE Symposium on Parallel and Distributed Processing* (pp. 28-37). Los Alamitos, CA: IEEE Computer Society Press.

Mitchell, M. (1996). *An introduction to genetic algorithms*. Cambridge, MA: The MIT Press.

Mühlenbein, H. (1989). Parallel genetic algorithms, population genetics and combinatorial optimization. In H. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 416-421). San Mateo, CA: Morgan Kauffman.

Reinelt, G. (n.d.). *TSPLIB 95*. Interdisciplinary Center for Scientific Computing, University of Heidelberg, Germany. Retrieved January 10, 2001, from World Wide Web: http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95.

Rosenkrantz, D., Stearns, R., and Lewis, P. (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing, 6*(3), 563-581.

Syslo, M., Deo, N., and Kowalik, J. (1983). *Discrete optimization problems*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Yamada, T., and Nakano, R. (1992). A genetic algorithm applicable to large-scale job-shop problems. In Manner, R., and Manderick, B. (Eds.), *Proceedings of the 2nd Parallel Problem Solving from Nature* (pp. 281-290). Amsterdam, Netherlands: Elsevier Science Publishers.

APPENDICES

APPENDIX A

FITNESS LEVEL GRAPHS

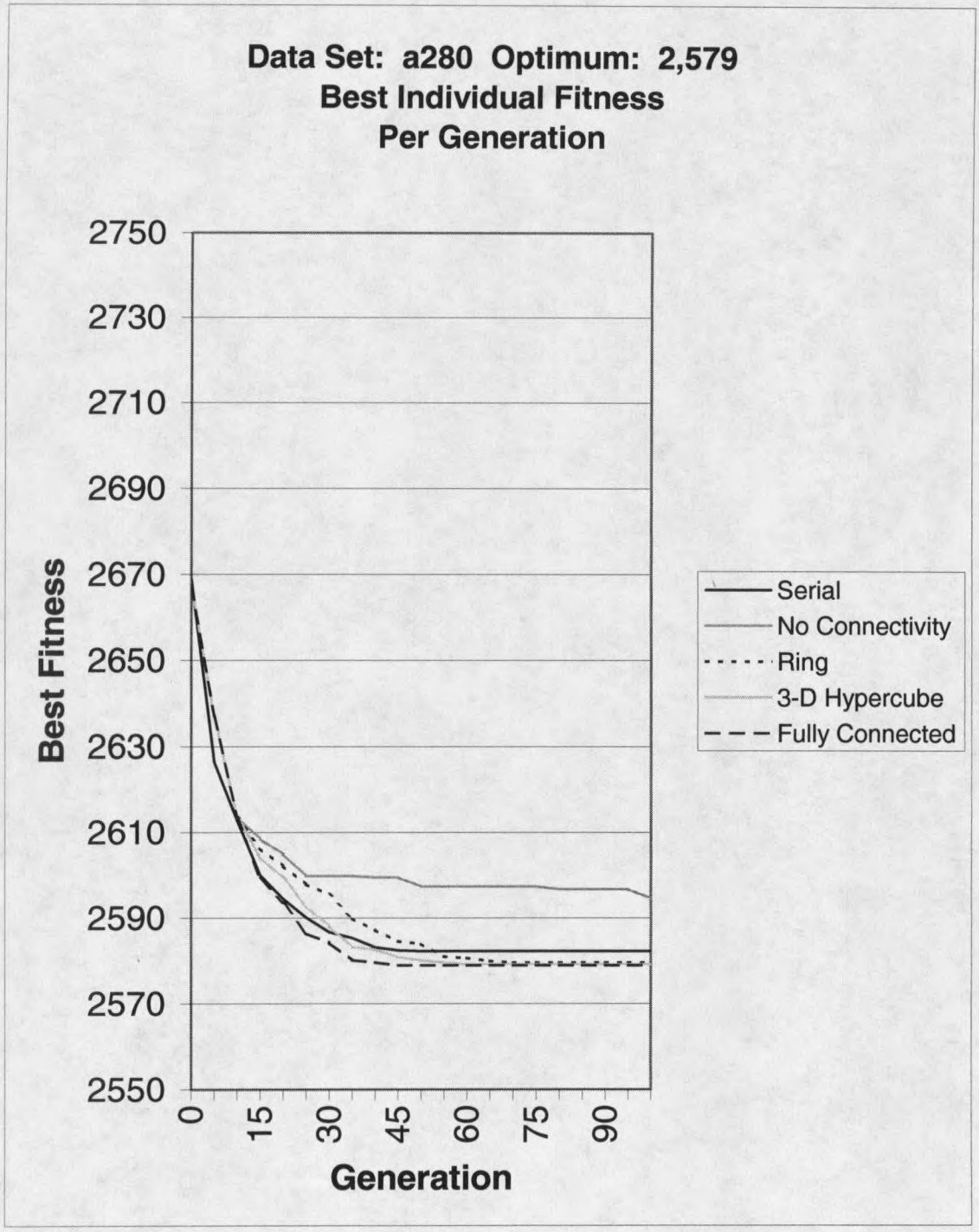


Figure 29: Best Individual Fitness Per Generation for *a280*

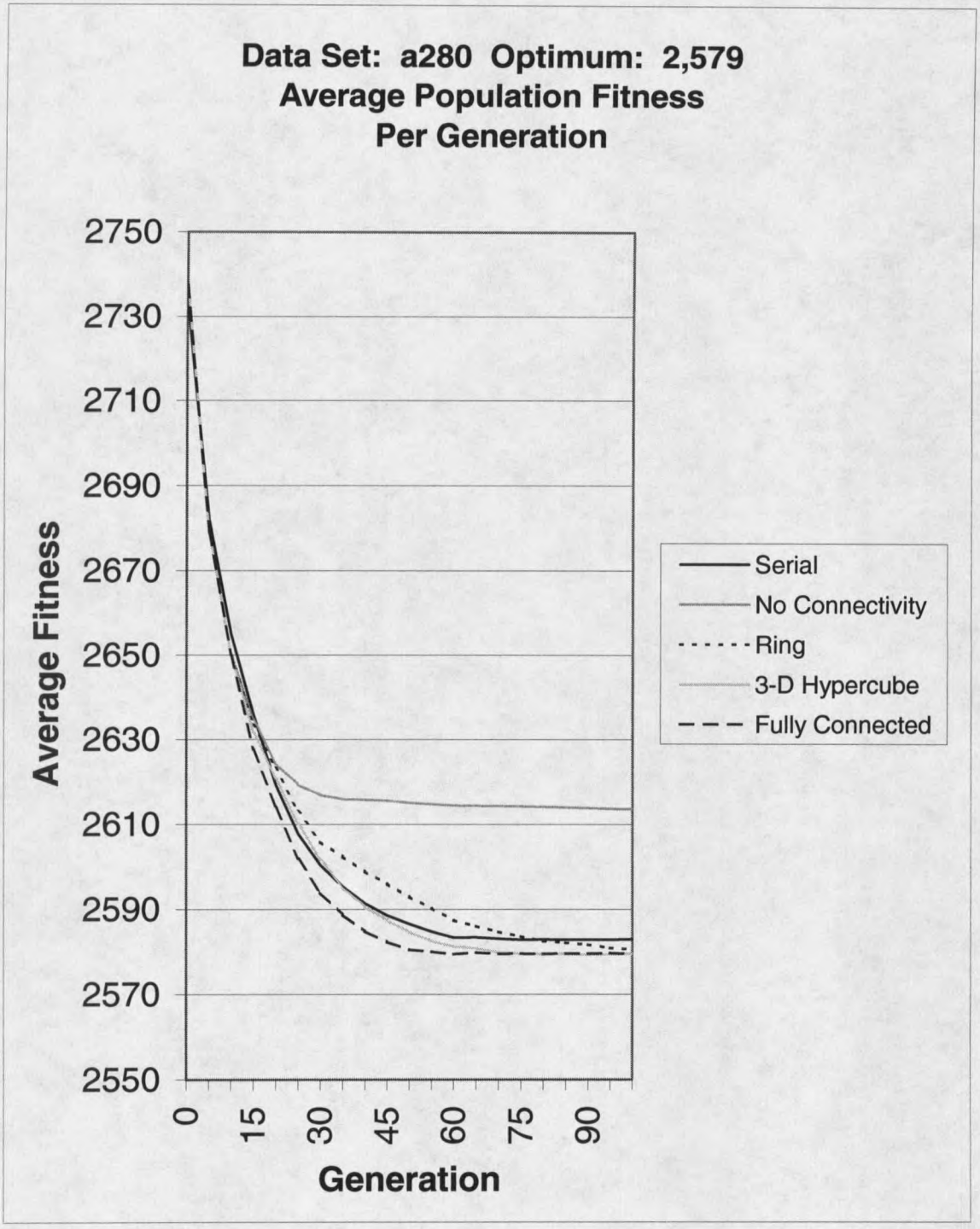


Figure 30: Average Population Fitness Per Generation for *a280*

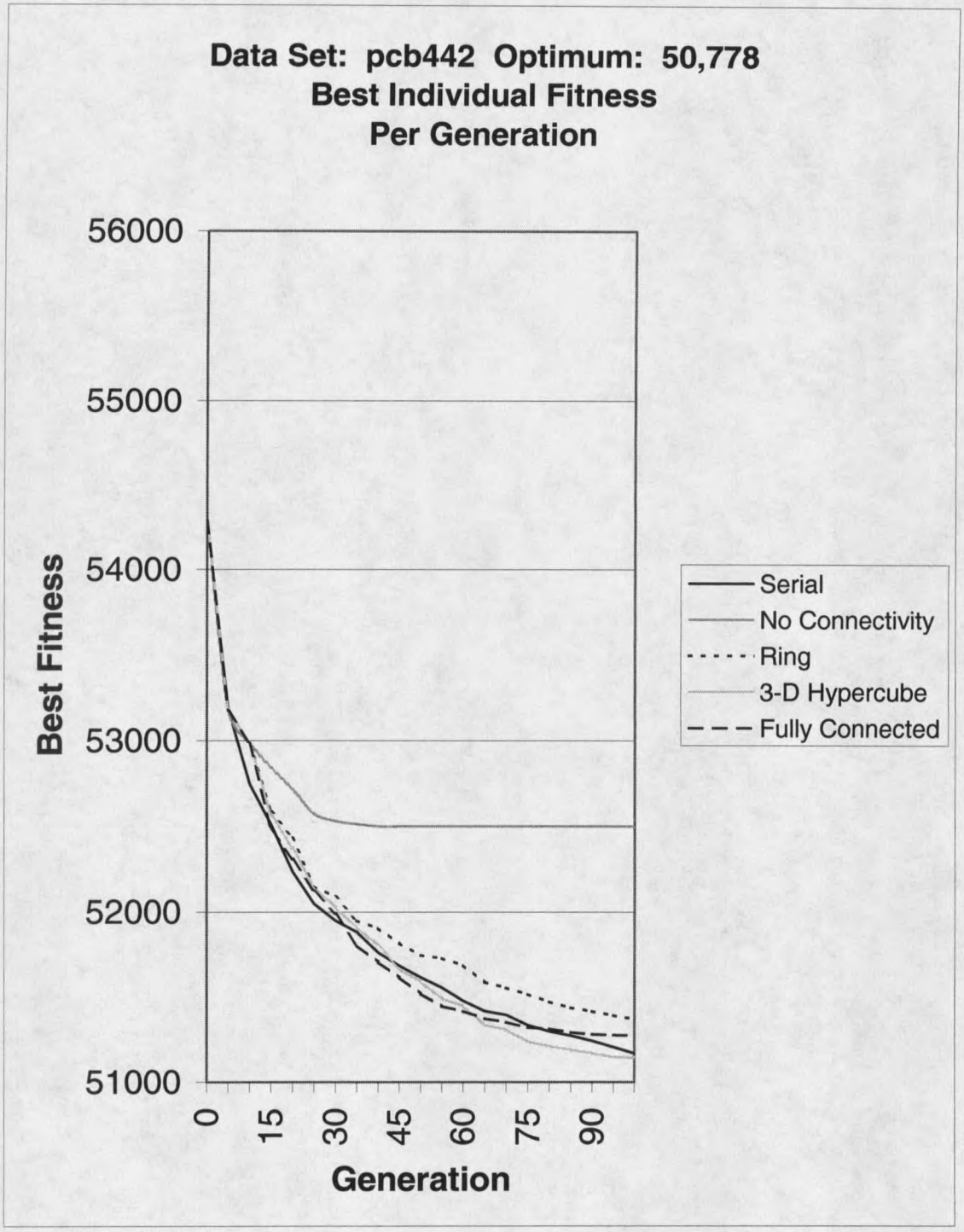


Figure 31: Best Individual Fitness Per Generation for *pcb442*

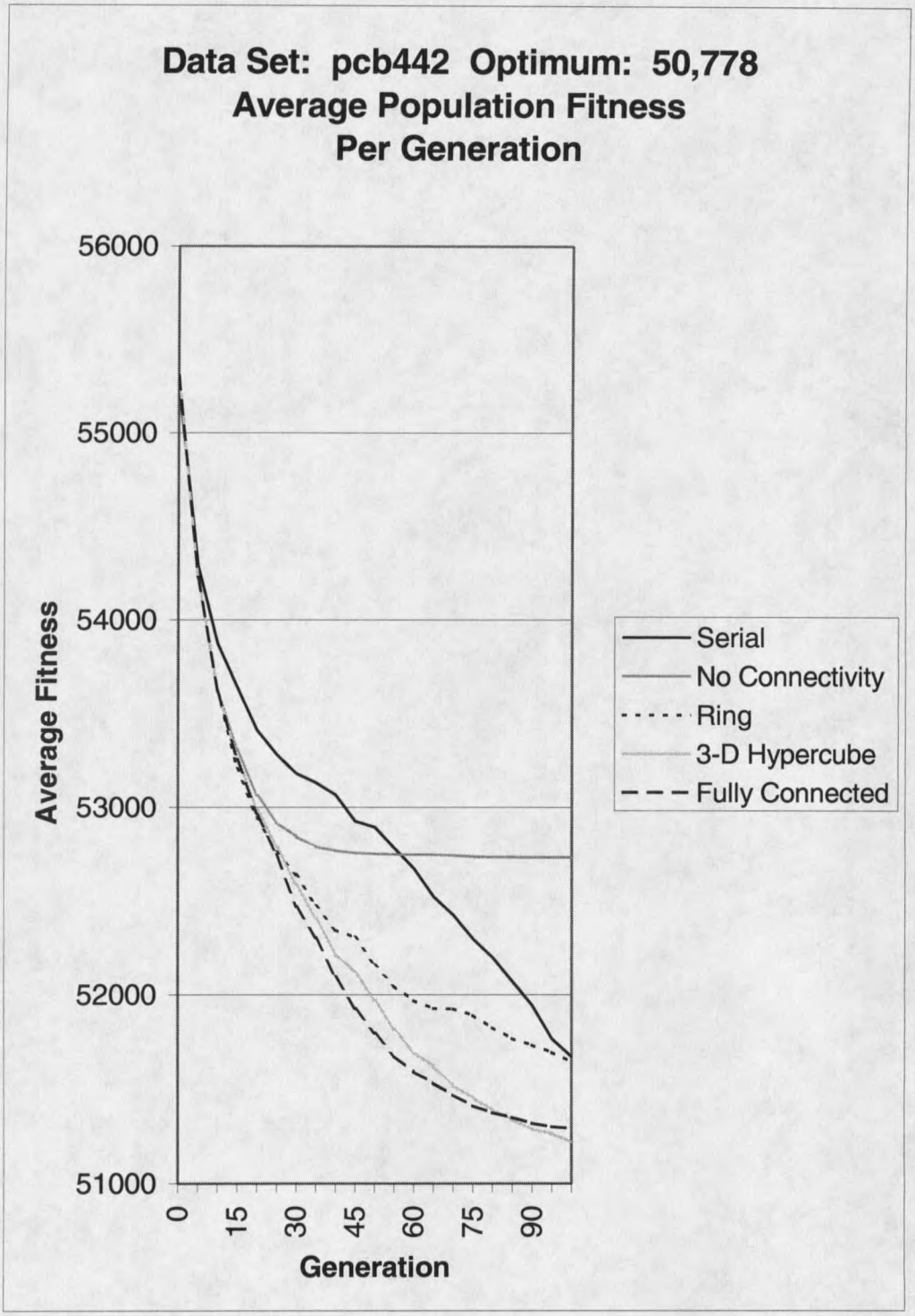


Figure 32: Average Population Fitness Per Generation for *pcb442*

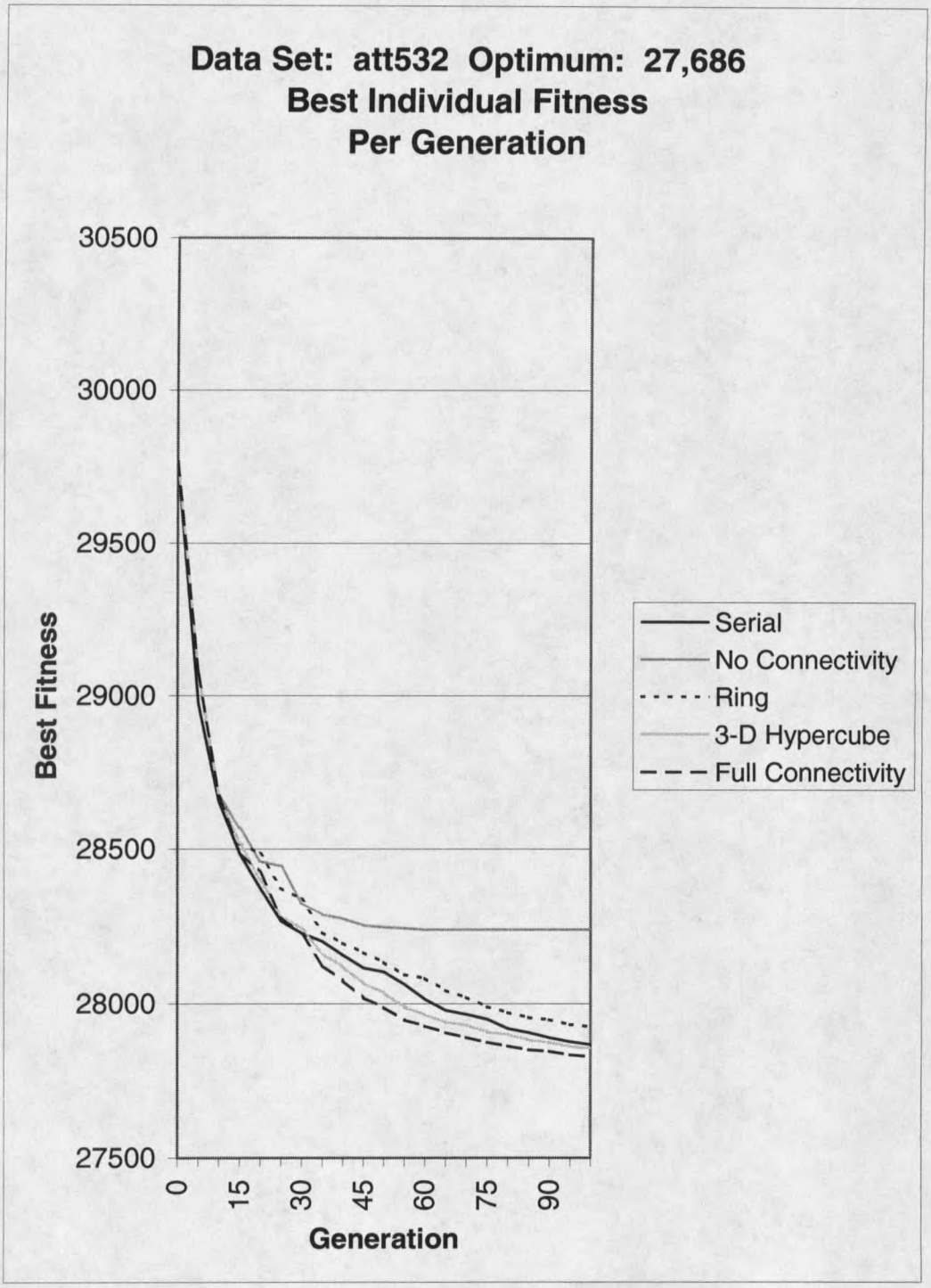


Figure 33: Best Individual Fitness Per Generation for *att532*

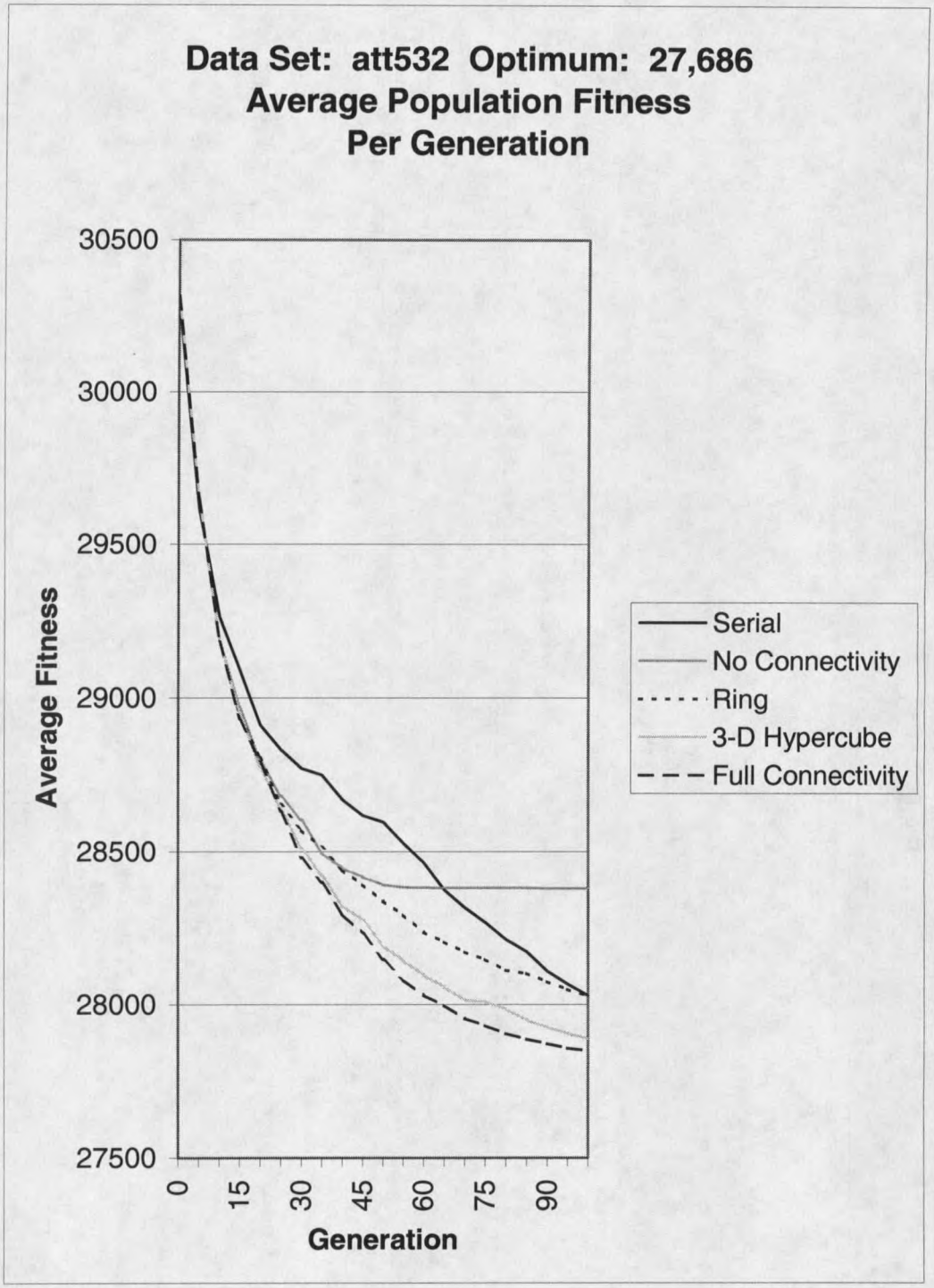


Figure 34: Average Population Fitness Per Generation for *att532*

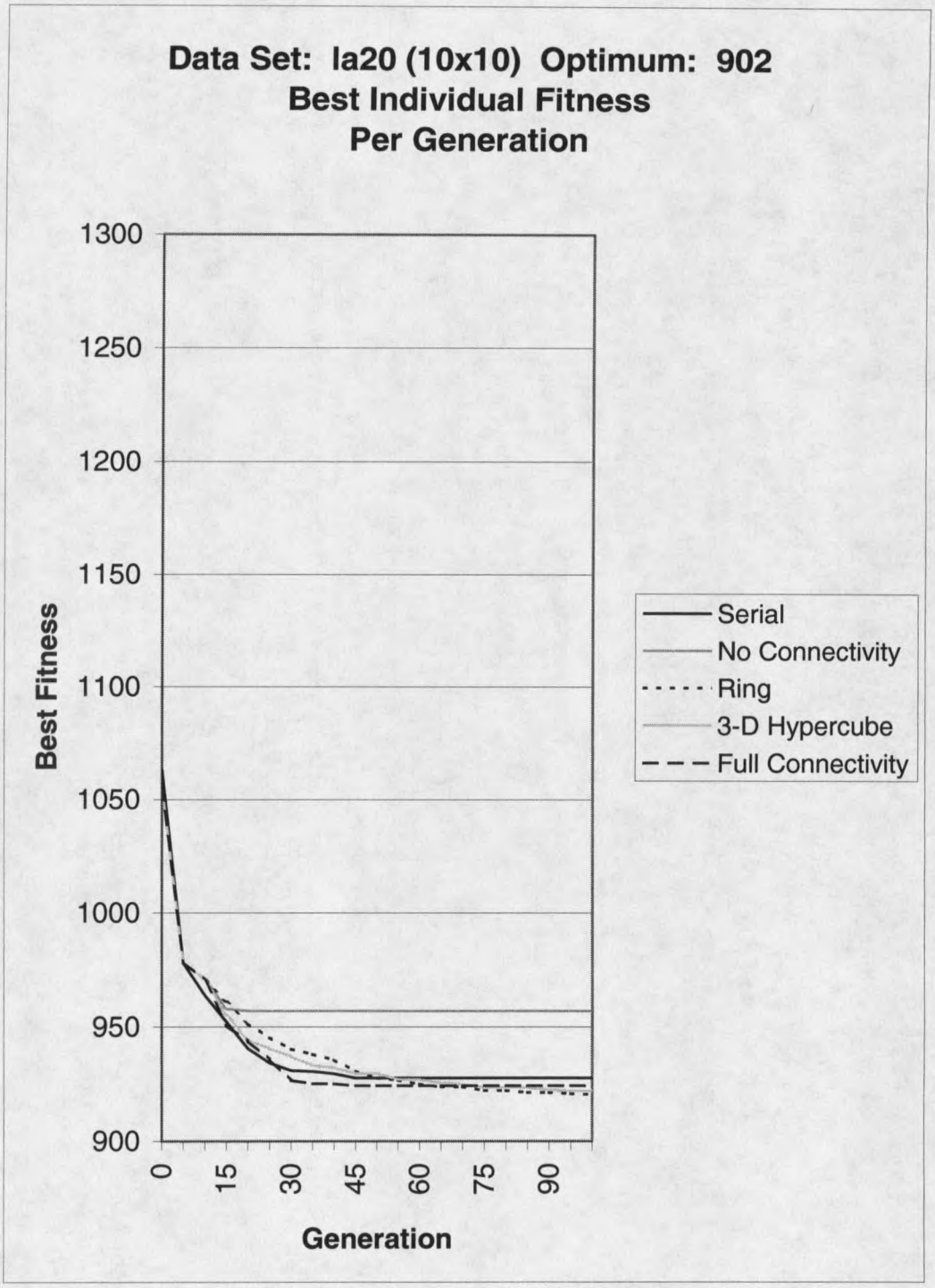


Figure 35: Best Individual Fitness Per Generation for *la20*

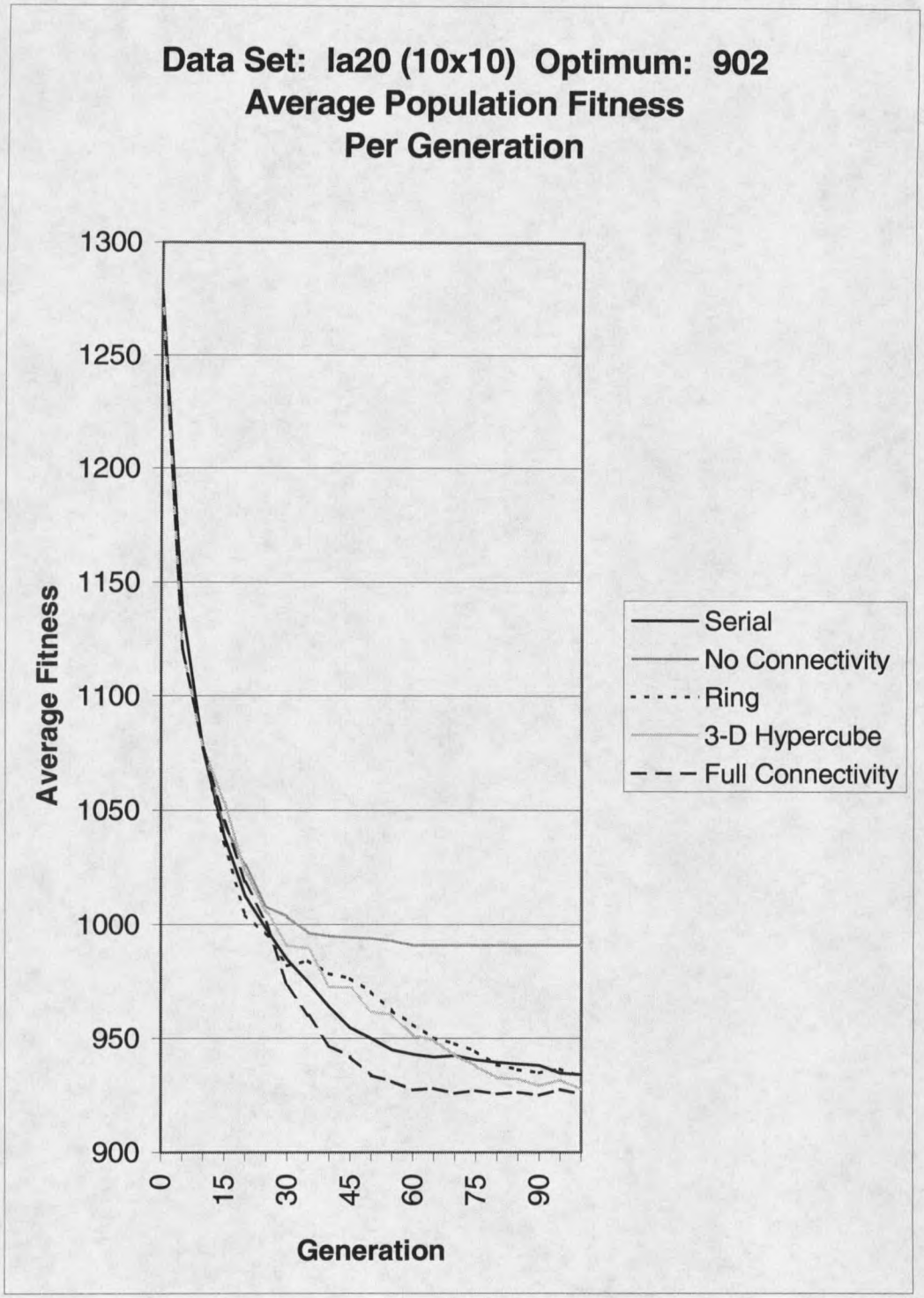


Figure 36: Average Population Fitness Per Generation for *la20*

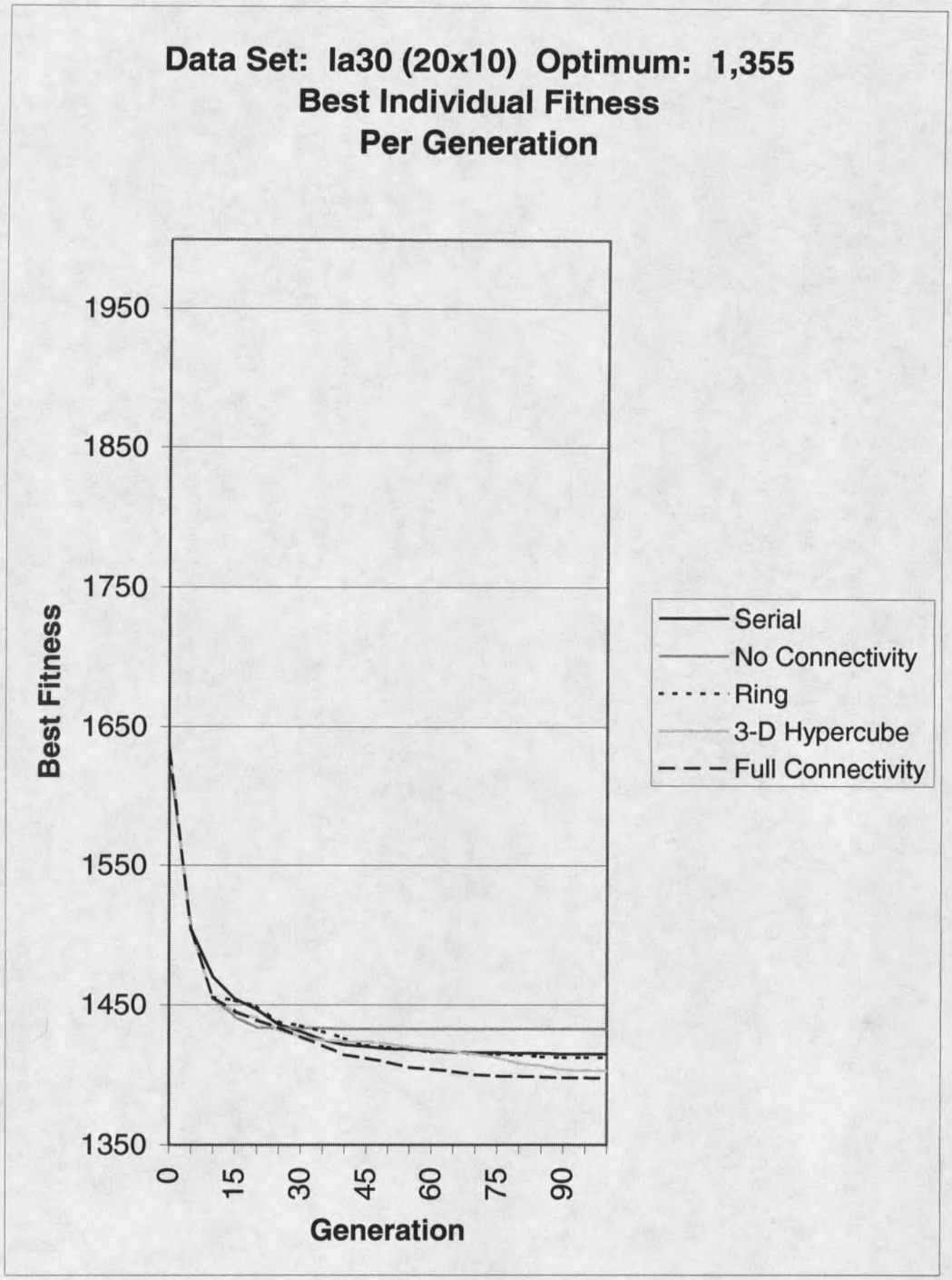


Figure 37: Best Individual Fitness Per Generation for *la30*

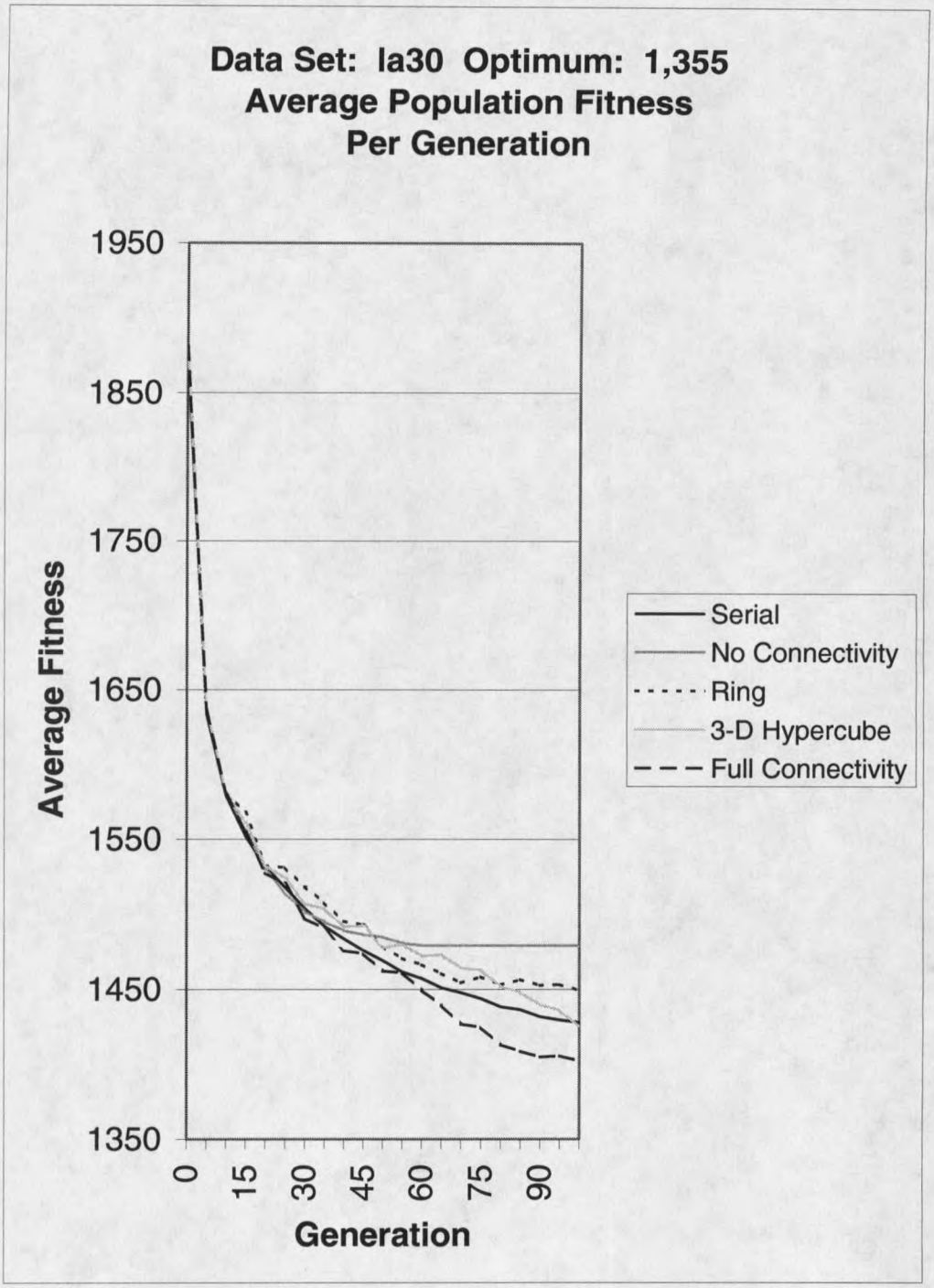


Figure 38: Average Population Fitness Per Generation for *la30*

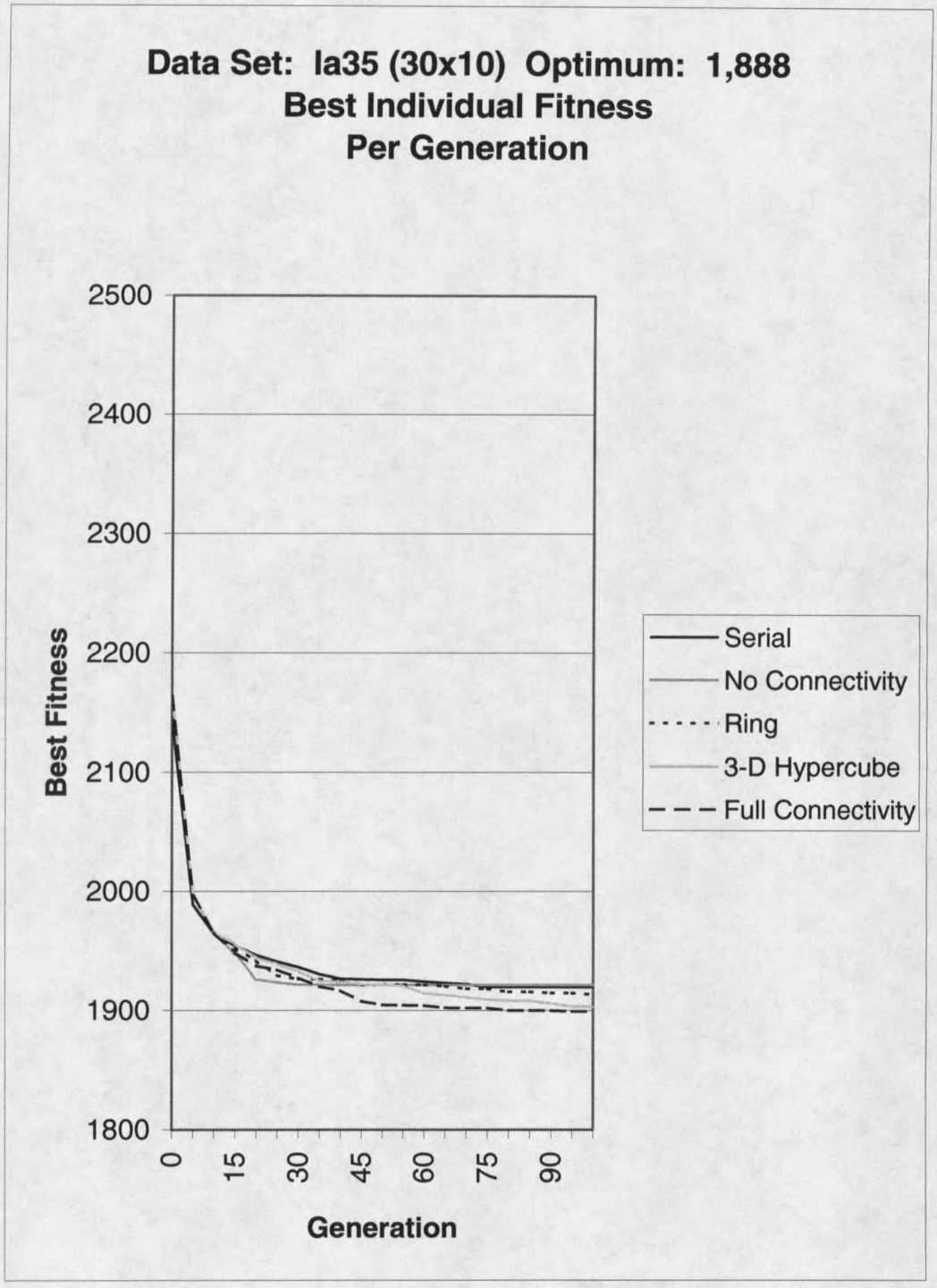


Figure 39: Best Individual Fitness Per Generation for *la35*

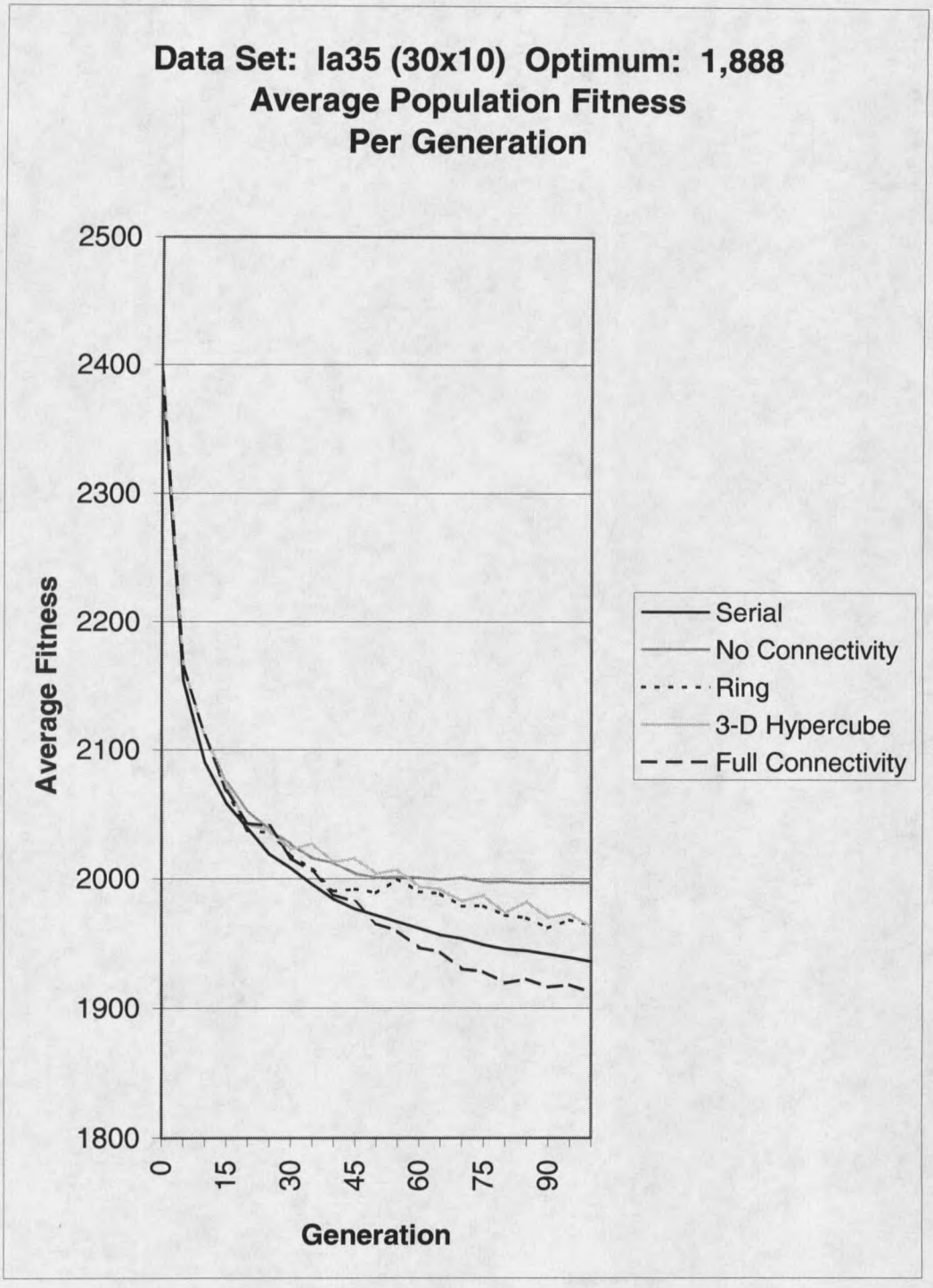


Figure 40: Average Population Fitness Per Generation for *la35*

APPENDIX B

STATISTICAL CALCULATIONS

Goal: Prove that the parallel with full connectivity model GA is statistically better than the serial model at obtaining best individuals at specific generations when solving the TSP.

Method: Wilcoxon Test

Let $E[x]$ = expected quality of serial model
and $E[y]$ = expected quality of parallel with full connectivity model.

Let $H_0 = E[x] = E[y]$
and $H_a = E[x] > E[y]$.

	Gen	*x*	*y*	*x-y*	*|x-y|*	*Rank |x-y|*	*Signed Rank |x-y|*
a280	25	0.43	0.29	0.14	0.14	6.5	6.5
	50	0.13	0.00	0.13	0.13	4	4
	75	0.13	0.00	0.13	0.13	4	4
	100	0.13	0.00	0.13	0.13	4	4
pcb442	25	2.50	2.65	-0.15	0.15	8	-8
	50	1.65	1.46	0.19	0.19	9	9
	75	1.10	1.07	0.03	0.03	1	1
	100	0.77	0.98	-0.21	0.21	10	-10
att532	25	2.10	2.15	-0.05	0.05	2	-2
	50	1.51	1.09	0.42	0.42	12	12
	75	0.96	0.68	0.28	0.28	11	11
	100	0.66	0.52	0.14	0.14	6.5	6.5
							Sum W = 38

Table 20: Wilcoxon Test for TSP Best Individual Fitness Levels

We can reject the null hypothesis if $W > W_{1-\alpha}$.

The critical value $W(\alpha)$ for $n \geq 10$ is

$$W(\alpha) = Z(\alpha)\sqrt{n(n+1)(2n+1)/6}$$

where $Z(\alpha)$ is the standard normal fractile such that a region of size α is to the left of $Z(\alpha)$.

Then, for n=12 and α=0.05, $W_{1-0.05} = 41.94$.
Since $W = 38 < W_{1-0.05} = 41.94$ the null hypothesis is not rejected implying that the quality of the serial model is equal to the quality of the parallel with full connectivity model at finding best individual fitness levels throughout the GA at α=0.05. It approaches significance at α=0.10.

Goal: Prove that the parallel with full connectivity model GA is statistically better than the serial model at obtaining average population fitness levels at specific generations when solving the TSP.

Method: Wilcoxon Test

Let $E[x]$ = expected quality of serial model
and $E[y]$ = expected quality of parallel with full connectivity model.

Let $H_0 = E[x] = E[y]$
and $H_a = E[x] > E[y]$.

	Gen	x	y	x-y	|x-y|	Rank |x-y|	Signed Rank |x-y|
a280	25	1.12	0.89	0.23	0.23	3.5	3.5
	50	0.29	0.06	0.23	0.23	3.5	3.5
	75	0.15	0.02	0.13	0.13	1	1
	100	0.16	0.02	0.14	0.14	2	2
pcb442	25	4.94	3.87	1.07	1.06	8	8
	50	4.17	2.01	2.16	2.16	12	12
	75	2.99	1.25	1.74	1.75	11	11
	100	1.76	1.01	0.75	0.75	7	7
att532	25	4.14	3.42	0.72	0.72	6	6
	50	3.29	1.66	1.63	1.63	10	10
	75	2.10	0.89	1.21	1.22	9	9
	100	1.24	0.59	0.65	0.65	5	5
							Sum $W = 78$

Table 21: Wilcoxon Test for TSP Average Population Fitness Levels

We can reject the null hypothesis if $W > W_{1-\alpha}$.

The critical value $W(\alpha)$ for $n \geq 10$ is

$$W(\alpha) = Z(\alpha)\sqrt{n(n+1)(2n+1)/6}$$

where $Z(\alpha)$ is the standard normal fractile such that a region of size α is to the left of $Z(\alpha)$.

Then, for n=12 and α=0.05, $W_{1-0.05} = 41.94$.
Since $W = 78 > W_{1-0.05} = 41.94$ the null hypothesis is rejected, inferring that the quality of the parallel with full connectivity model is better than the serial model at finding average individual fitness levels throughout the GA at α=0.05.

Goal: Prove that the parallel with full connectivity model GA is statistically better than the serial model at obtaining best individuals at specific generations when solving the JSSP.

Method: Wilcoxon Test

Let $E[x]$ = expected quality of serial model
and $E[y]$ = expected quality of parallel with full connectivity model.

Let $H_0 = E[x] = E[y]$
and $H_a = E[x] > E[y]$.

	Gen	*x*	*y*	*x-y*	*|x-y|*	*Rank |x-y|*	*Signed Rank |x-y|*
la20	25	3.57	3.75	-0.18	0.18	2	-2
	50	2.85	2.47	0.38	0.38	5	5
	75	2.85	2.47	0.38	0.38	5	5
	100	2.85	2.47	0.38	0.38	5	5
la30	25	5.99	5.82	0.17	0.17	1	1
	50	4.70	4.00	0.70	0.70	7	7
	75	4.50	3.27	1.23	1.23	11	11
	100	4.44	3.17	1.27	1.27	12	12
la35	25	2.81	2.46	0.35	0.35	3	3
	50	2.00	0.90	1.11	1.11	10	10
	75	1.69	0.76	0.93	0.93	8	8
	100	1.69	0.60	1.09	1.09	9	9
							Sum W = 74

Table 22: Wilcoxon Test for JSSP Best Individual Fitness Levels

We can reject the null hypothesis if $W > W_{1-\alpha}$.

The critical value $W(\alpha)$ for $n \geq 10$ is

$$W(\alpha) = Z(\alpha)\sqrt{n(n+1)(2n+1)/6}$$

where $Z(\alpha)$ is the standard normal fractile such that a region of size α is to the left of $Z(\alpha)$.

Then, for n=12 and α=0.05, $W_{1-0.05} = 41.94$.
Since $W = 74 > W_{1-0.05} = 41.94$ the null hypothesis is rejected, inferring that the quality of the parallel with full connectivity model is better than the serial model at finding average individual fitness levels throughout the GA at α=0.05.

Goal: Prove that the parallel with full connectivity model GA is statistically better than the serial model at obtaining average population fitness levels at specific generations when solving the JSSP.

Method: Wilcoxon Test

Let $E[x]$ = expected quality of serial model
and $E[y]$ = expected quality of parallel with full connectivity model.

Let $H_0 = E[x] = E[y]$
and $H_a = E[x] > E[y]$.

	Gen	*x*	*y*	*x-y*	*|x-y|*	*Rank |x-y|*	*Signed Rank |x-y|*
la20	25	10.68	10.98	-0.30	0.30	2	-2
	50	5.32	3.50	1.82	1.82	11	11
	75	4.27	2.77	1.50	1.50	10	10
	100	3.56	2.59	0.97	0.97	5	5
la30	25	12.03	12.23	-0.20	0.20	1	-1
	50	8.44	7.93	0.51	0.51	4	4
	75	6.57	5.17	1.40	1.40	9	9
	100	5.40	3.53	1.88	1.88	12	12
la35	25	6.95	8.10	-1.16	1.16	7	-7
	50	4.44	4.09	0.35	0.35	3	3
	75	3.23	2.12	1.10	1.10	6	6
	100	2.54	1.29	1.24	1.24	8	8
							Sum W = 58

Table 23: Wilcoxon Test for JSSP Average Population Fitness Levels

We can reject the null hypothesis if $W > W_{1-\alpha}$.

The critical value $W(\alpha)$ for $n \geq 10$ is

$$W(\alpha) = Z(\alpha)\sqrt{n(n+1)(2n+1)/6}$$

where $Z(\alpha)$ is the standard normal fractile such that a region of size α is to the left of $Z(\alpha)$.

Then, for n=12 and α=0.05, $W_{1-0.05} = 41.94$.
Since $W = 58 > W_{1-0.05} = 41.94$ the null hypothesis is rejected, inferring that the quality of the parallel with full connectivity model is better than the serial model at finding average individual fitness levels throughout the GA at α=0.05.