1999

Implementation of genetic algorithms in optical wavelength ring routed network design.

Ziad. Kobti
University of Windsor

IMPLEMENTATION OF GENETIC ALGORITHMS IN OPTICAL WAVELENGTH RING ROUTED NETWORK DESIGN

by

ZIAD KOBTI

A Thesis
Submitted to the College of Graduate Studies and Research
Through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

1999

0-612-62231-2

ABSTRACT

The design process of the ring routed wavelength optical network encompasses the search for a layout order of nodes such that the total overall traffic is minimized in terms of distant communication and traffic frequencies. The optimal solution is a member of a massive domain set reaching sizes in the order of n!, where n is the number of nodes in the network; typically n = 100. A brute force, linear search algorithm can be implemented, but when executed in search for an optimal solution, the algorithm is a computational challenge as it becomes time consuming and renders itself unfeasible in realistic design time criteria. In researching a suitable search algorithm with the flexibility to adapt to changing network traffic parameters and render a near optimal solution in reasonable design time constraints, the genetic algorithm presents a candidate solution to be tested. The goal of this thesis is to implement a custom genetic algorithm and examine its potential in reaching near optimal solutions in the optical design application framework. The course of work was dividided into three major development stages. In the first stage, a simple object oriented GA model was developed (SGA). Then the model was customized to the ring routed network design application, known as the Simple Optical Genetic Algorithm (SOGA) and finally the revised algorithm is implemented and termed the Optical Genetic Algorithm (OGA). In smaller networks (up to 12 nodes) the GA is compared to a brute force linear algorithm to test its performance. For larger networks, the GA was compared to a random search algorithm to test its effectiveness. In both cases, the GA has shown to surpass the other algorithms in generating a pool of near optimal solutions in reasonable time constraints.

ACKNOWLEDGEMENTS

Over the past three years of persistence, I wish to dedicate my work to all the people who exceeded the limit in assisting and guiding me in this journey. A special thanks to my supervisor Dr. Subir Bandyopadhyay for going above and beyond to accommodate me in his already tight schedule. My committee members: Dr. Christie Ezeife and Dr. Yash Aneja whose much input and comments helped me shape a more solid and proven research. Special thanks to my undergraduate research supervisor Dr. Hugh B. Fackrell whose insight played a major role in shaping my attitude and philosophy toward research appreciation and excellence. My appreciation goes out to Dr. Peter Tsin who took the time to discuss complexity measures related to the algorithm.The handpicked few: Dr. Robert D. Kent, and Dr. Richard Frost who set the example of research excellence and dedication. Last but not least I dedicate all my work to my great mother and supporting family without whom none of this would have been possible.

The Java Developer Kit 1.x and Java are trademarks or properties of Sun Microsystems, Inc. who offer this technology free for education and research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 GENERAL OVERVIEW

With the advancement of optical network technology, the optical fiber has become a dominant medium for telecommunication. The distinguishing features include the enormous bandwidth capacity, which is in the order of hundreds of megabits, the immunity to noise, and the minimal signal loss provided by the single mode fiber. A network system that relies on fiber optic connections provides its users the flexibility of what the technology has to offer from speed, security, and accuracy but at a higher cost than other conventional network media like the copper wire.

Coupled with the choice of the medium, a design topology, or layout, is required in order to form the network connectivity. The network industry defines three major topology standards: linear, ring, and star. Each topology distinguishes itself by a unique physical layout and its corresponding advantages and disadvantages. The application that the thesis work is investigating centers on the ring topology.

An optical fiber can transmit a signal across long distances at nearly the speed of light. However, during a transmission process, only one signal can travel along the fiber at a particular wavelength. The fiber can handle numerous simultaneous connections, each travelling in one direction at a unique frequency. Based on the number of simultaneous connections that are required at any given time, or average traffic, the amount of fiber optic cables between nodes is determined. A node can be described as a point on the network where fiber cables are joined via sophisticated switchboards, and some may allow a point of access providing connectivity of a local system to the network loop.

Since the costs associated with this relatively infant technology are high, it is essential to design a network with minimum wiring without compromising its efficiency. One effective method to minimize wiring between two busy endpoints, or nodes, is to simply locate them to a proximity to each other. For example, two endpoints that experience high volume traffic between them should ideally be connected directly to each other with as many fiber bundles required to handle communication and avoid network congestion.

Network layouts that do not take actual traffic scenarios fail to live up to their promise of speed as bottlenecks and congestion overcome the system. A careful network design that accounts for the dynamic traffic patterns of communication is hence an ideal investment. Before physically installing the network, designing its layout and connectivity while taking into account its traffic patterns and node locations is essential in network engineering. The application adopted by the thesis involves a hypothetical network that employs the ring topology accompanied with the communication or cost matrix that describes its anticipated traffic. The goal is to provide a mean to design an efficient network layout that accommodates the anticipated traffic scenario.

1.2 PROBLEM FORMULATION

Physical settings and traffic patterns are distinct for every network. In some cases, even within the same network structure, the traffic dynamics may change overtime corresponding to user needs. For instance, network settings that do not undergo frequent physical changes, and that require quicker packet delivery would typically use the ring topology. A lower cost yet slower network would deploy a linear topology. However, the ring wavelength layout is a common topology and is examined in further detail in the course of this investigation in terms of its design optimization.

1.2.1 BRIEF REVIEW OF THE RINGED WAVELENGTH NETWORK ARCHITECTURE

In the realm of optical networks, a wavelength-routed network based on the ring topology is one that transmits signals at a specified frequency bounded in a cyclic loop; hence the term ring.

In a simple layout, the ring network is composed of a set of nodes, or switching centers, where the signal is either generated or re-generated for transmission. The fiber cables form the connectivity, or link, between any two nodes. It carries a signal from one node to the next at a unique wavelength (λ). Overall, the nodes can be mapped around a circular shape forming the ring network as shown in figure 1.1. The figure also shows the nodes and the connecting fiber optic cable.

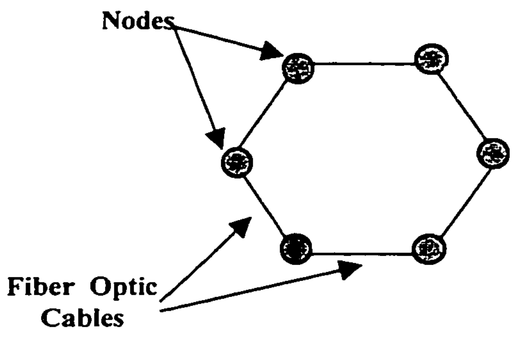


FIGURE 1.1: SIMPLE RING TOPOLOGY NETWORK COMPOSITION MODEL

In order to transport data from one station to another within the ring network, data packets are first converted to light signals at the starting node. Each signal travels from the initiating node to a target destination via a free channel within a fiber optic cable.

Each fiber transmits a light wave at a unique frequency in order not to interfere with other simultaneous connections across the same fiber. No two waves may share the same frequency by definition. After travelling a certain distance, light waves become weak signals and need to be reproduced; at this point, signal regeneration takes place at the node costing time and using resources. In addition to regeneration, a node serves as a linkage point and signal conversion from one wavelength to another. The reason for such possible conversion is that if a signal is travelling at a given wavelength, and after going to the node, at the next fiber another signal may be using the same wavelength. For the original signal to continue its journey through the next fiber without collision, it must be converted and transmitted at a different readily available wavelength. Figure 1.2 illustrates a simple model describing the functionality of the nodes during a signal transmission and the two possible directions a signal can travel across the ring to reach its destination.

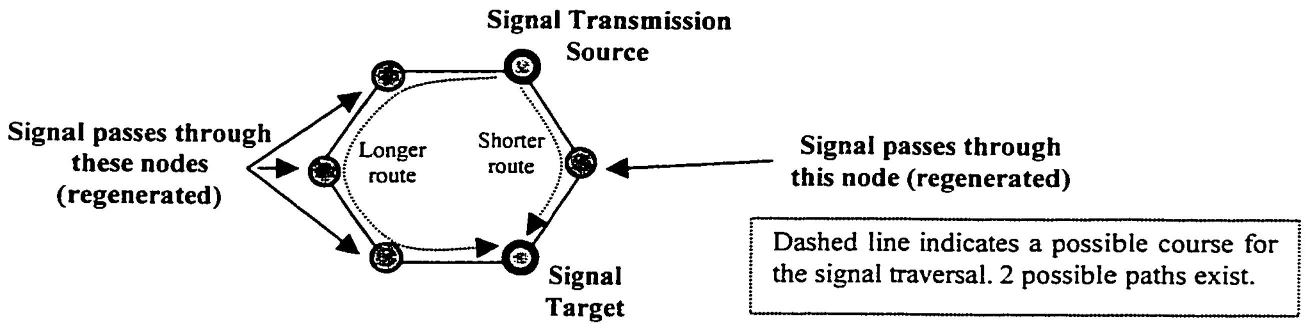

Signal Transmission Source
Signal passes through these nodes (regenerated)
Longer route
Shorter route
Signal passes through this node (regenerated)
Dashed line indicates a possible course for the signal traversal. 2 possible paths exist.
Signal Target

FIGURE 1.2: SIGNAL TRAVERSAL AND ROUTING MODEL

1.2.2 PRESENTING THE PROBLEM

Every conversion, regeneration, and distance traversal has an associated cost that reduces the overall performance. Consequently, many constraints arise that reflect the physical and conceptual designs. If a signal is to travel from a source to a destination node, then it is favourable to find the shortest path such signal should travel. Each node has its own probability of communication. Relative frequencies are given as they reflect the expected traffic flow (or costs associated) per traversal. By minimizing signal traversal distances by means of allowing the signal to reach its destination with minimal crossing of alternate nodes, a faster network flow can be implemented by design.

It is therefore extremely essential to simulate such network and predict its main traffic bottlenecks. Then attempt to minimize the total amount of fibers by optimizing relative positioning of nodes. The ultimate goal is to find the optimal combination order of nodes to keep the network costs at a minimum.

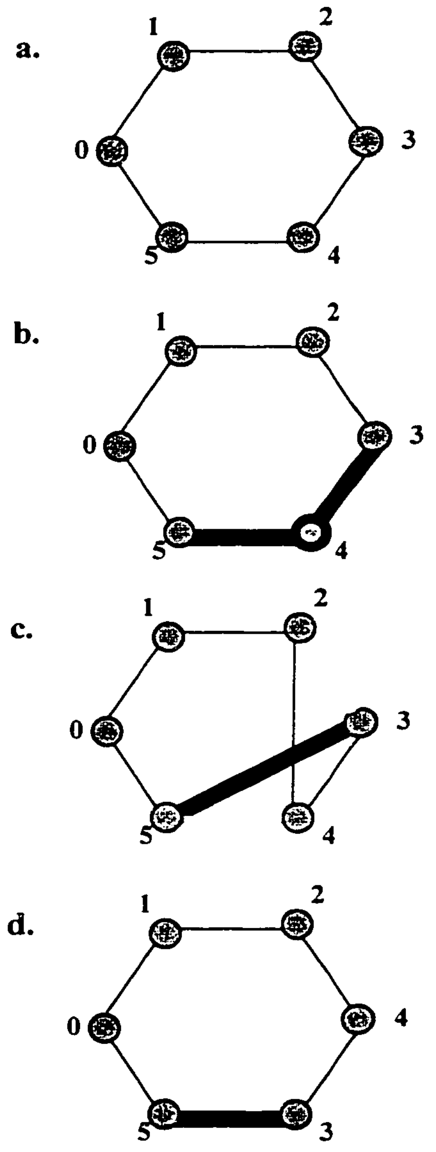
The optimization of the ring layout is based on two criteria:

1. The node distance factor, in terms of the number of nodes that need to be traversed in order to reach a designated destination. Bi-directional traversal is accounted for, and the minimal distance is chosen as preferred route.
2. The cost/traffic pattern described in a 2D matrix. A value of 0 indicates no communication, and the higher the value the higher the costs of communication.

An example with 6 nodes could be solved using a linear or brute force algorithm that cycles through all the possible combinations and retrieve the best performing layouts. Unfortunately, the same algorithm when having to deal with a network of a magnitude of 100 nodes would prove unfeasible as the number of combinations to be tested become in the magnitude of 100 factorial. The need for a better search algorithm is required, and the Genetic Algorithm will be put to the test as a possible candidate solution.

The optimization scenario involved in this application is illustrated in figure 1.3. The first figure (a) describes the initial six node ring network. Each node is assigned a unique number to identify it. The sequence of the nodes can be described in terms of listing the nodes by their assigned numbers. For instance, the network is described by the sequence 012345. Figure (b) stages a traffic scenario where heavy communication occurs between nodes 5 and 3. In order for them to communicate, they use the shorter route around the ring which involves traversing across node 4. To eliminate this slower performance of having to cross an intermediate node, 3 and 5 could be redesigned to connect directly to each other (c), hence avoiding the middle step at crossing node 4. The new resulting layout now described in figure (d) shows that the sequence 012435 is a better model that accounts for the traffic scenario staged between nodes 3 and 5.

A typical network however that contains much more nodes and traffic criteria could be as easily detected. A specialized algorithm will be required to produce a better performing node sequence by means of detecting the staged traffic patterns.



A simple ring network where each node is designated by a corresponding number.

Patterns: 53#### or 35####
Where:
1. "#" indicates any node
2. The pattern is a permutation
3. Mask applies for all cyclic combinations i.e.
   53#### ⇔ #53### ⇔ ##53## ⇔ ###53# ⇔ ####53

Based on the detected pattern, connecting nodes 5 and 3 directly would be a better design in terms of minimizing the cost of having to pass through node 4 unnecessarily.

A redraw of the ring shows that the sequence of nodes now: 0124**35** forms a better design that accounts to the heavy traffic between nodes 5 and 3.

FIGURE 1.3: SIMULATION OF A 6 NODE NETWORK OPTIMIZATION EXAMPLE

1.3 PROBLEM STATEMENT

The six node network and its simple traffic scenario staged in figure 1.3 is a mere elementary example. Networks in the order of hundreds of nodes with complex multi-node traffic scenarios are nearly impossible to be designed by hand. Even a heuristic search mechanism that tests every combination is rendered by its massive computational time unfeasible. The need for a new approach that can achieve near optimal solutions within acceptable time limitations is imminent.

In order to visualize a realistic network, which is typically in the order of hundreds of nodes, a diagrammatic representation is produced. Figure 1.4 illustrates such a network by numbering each node starting from 0 with the last node being N-1 where N is the number of nodes in the ring.

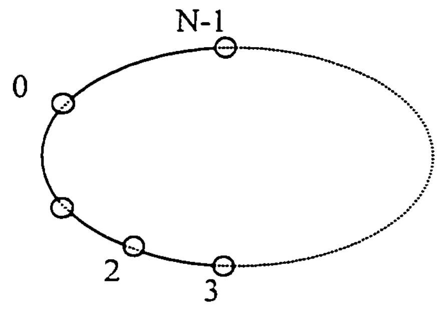


FIGURE 1.4: RING NETWORK LAYOUT IN THE ORDER OF N NODES

1.3.1 DOMAIN SET DEFINITION

In order to formalize the problems' constraints, a domain set description is in order. The solution space, or domain, is defined as the set of permutations of a size corresponding to that of the nodes in the network (N). For instance, given a ringed network of size N, a possible solution to the layout of the nodes would be the permutation

[0;1;2;3;...;N-1] where each number uniquely identifies a node. The search space is the set of all permutations of size N. For large networks of order 100 would accumulate a search space of nearly 100 factorial, rendering the search space extremely vast to be systematically searched within feasible time constraints.

1.3.2 OBJECTIVE FUNCTION

For the search algorithm to evaluate each permutation for its performance relative to others, an objective function is needed.

In order to describe the traffic and cost pattern scenario related to the network, positive integer values are provided in an **N** x **N** matrix **A** describing the traffic frequencies and costs between nodes. The row and column indices start from 0. The contents of the matrix give the expected traffic between pairs of nodes on the network. Ultimately, the ideal solution corresponds to a permutation evaluating to the minimum relative cost. That is it requires the lowest cost and minimizes unnecessary traffic flows. Restated in mathematical terms, the goal of the search algorithm is to find a permutation Π of the sequence of numbers 0, 1,..., N-1, such that the following expression is minimum (figure 1.5). The function is formulated by Dr. Bandyopadhyay's current work:

$$\sum\sum(f(\pi(i),\pi(j)))\times A(i,j)$$

FIGURE 1.5: FITNESS EVALUATION FUNCTION

Traffic/Cost Matrix

i →

j ↓

	0	1	2	3	4	5
0	0	4	2	7	9	3
1	4	0	6	8	3	4
2	2	6	0	3	2	9
3	7	8	3	0	6	3
4	9	3	2	6	0	2
5	3	4	9	3	2	0

FIGURE 1.6: SAMPLE TRAFFIC/COST MATRIX

Where:

- Subscripts $i \neq j$ since a node cannot communicate with itself.
- $\pi(x)$ denotes the value in position x after the permutation. Here the summation is over all *i* and over all *j* and the distance function *f* is defined as follows:

$$f(x, y) = min((x - y)_N, (y - x)_N)$$

Function f(x, y) calculates the shorter distance between the source and target nodes, so as to define the traversal route over the shorter path.

The product of the distance, in terms of nodes traversed, and the communication frequencies between nodes, described in the matrix, for every node on the network, produces a resulting total cost value. An optimal solution is a permutation with the minimal cost value.

In terms compatible to the genetic algorithm, the objective evaluation function is also termed as the fitness function. Every solution, or permeation, corresponds to an *individual*, and the set of solutions is a *population*.

1.4 EARLY CONTRIBUTIONS AND PREVIOUS WORK

The background research that eventually led to the Thesis formulation is twofold. The problem formulation itself, where the ringed network cost minimization scenario is manifested, formed the first requirement for the work. The second set of possible contributions was researched on the grounds of finding related work that used Genetic Algorithms.

At the debut of the thesis work, at the School of Computer Science, University of Windsor, genetic algorithms constituted a new field not explicitly developed. A separate research project conducted by Dr. Subir Bandyopadhyay involves the investigation of design methodologies for optical networks. Of particular importance, the search for an optimal design layout for the ring routed optical network scenario was refined to become a candidate application to be attempted to be solved by means of a genetic algorithm. Based on Dr. Bandyopadhyay's formulation of the cost function, which is presented in the problem statement, a fitness function will be adapted for the Genetic Algorithm.

In researching previous work on Genetic Algorithms, a myriad of publications and active research on the subject were found. Unfortunately, the subset of studies that actually involved Genetic Algorithms and Optical Network design was very dim. These studies did not directly involve design optimization using cost minimization measures, rather they centered around reliability constraints and survivability of the network. Collected papers include studies by Gavish *et. al.*(1989), Cardwell *et. al.* (1989), Davis *et. al.* (1993), and Amiri and Pirkul (1997). Many of the studies actually release pointers on how to handle some aspects of the problem that relates to routing, constraints and parameter handling.

In developing the Genetic Algorithm, another subset of research studies suddenly claimed its importance. Since the nodes in the network are unique, it can be deduced that the solution is a permutation. Therefore, in dealing with Genetic Algorithms that handle permutations, studies on the Travelling Salesman Problem (TSP) emerged as key to assist in the implementation process. References on TSP related studies are referenced throughout the report as necessary.

1.5 PROPOSED SOLUTION / DEFINITION

The nature of the problem stated above indicates domain constraints; not to mention that the domain is vast for the large values of N. Hence, the need of an algorithm that can exploit a large domain space without sacrificing time constraints. The genetic algorithm will be implemented and tested for its feasibility in solving this problem. The genetic algorithm is a search method used to process and locate better solutions to a given problem among a finite number of plausible solutions. A solution is said to solve the problem if it satisfies a given objective function. The object of a search procedure is to minimize the number of objective function evaluations necessary to locate a satisfactory solution. A search procedure is said to be efficient if the number of solutions evaluated is small in comparison to the size of the search space. The smaller this ratio, the more efficient the search procedure is. Robustness is a measure of how efficiency of a given search procedure changes drastically when the problem parameters are changed slightly.

1.6 THE THESIS STATEMENT

The network design problem that is presented has not been previously attempted. Experimental testing is initiated to test the feasibility limits of a brute force approach and establish a benchmark for small networks, up to 12 nodes. The genetic algorithm that is independently developed to attempt a solution is compared with other conventional heuristics. Larger networks will be tested and their comparative results are extrapolated. The working GA will be fine tuned for a particular problem constraint and parameter set.

1.7 OBJECTIVES AND SCOPE OF THE THESIS WORK

Due to the nature of the work with genetic algorithms, the reader may be easily swayed to believe that the thesis is about studying an aspect of optical networks. Contrary to this misleading thought, the object of the thesis is to implement and study genetic algorithms. To better understand this purpose, consider a typical linear search algorithm. In order to study the algorithm itself, it is imperative to test it against some appropriate data, hence the need for a search space and direction. This analogy implies the way genetic algorithms are being studied in this work. The optimization function to determine the optimal solution for the design of the ring routed network supplies the genetic algorithm with convincingly large domain space and sufficiently complex fitness function. Building on this optical network application model the focus on the genetic algorithm solution to the problem can be investigated. Furthermore, the choice of the ring routed optical network design problem opens new possibilities for other investigators in the field that may be proven beneficial.

The report will detail all the aspects of genetic algorithms that are used in the study. It, however, will not go in detail elaboration about optical networks beyond the that would be considered essential for a reader to understand the problem and follow along with the GA study.

1.8 ORGANIZATION OF THE THESIS REPORT

The Thesis report is composed of six major chapters.

The first chapter introduces the reader to some necessary background on the problem. It proposes the problem to be worked on and defining its merits and importance to the computing community. The thesis statement is formulated and the course of work is setup.

In the second chapter, the reader is exposed to the essential workings of Genetic Algorithms that will play a major role in the implementation in subsequent chapters. The reader is assumed to have little or no background experience in Genetic Algorithms, and hence the reason behind this fast paced introductory chapter.

The third chapter introduces the preliminary work of the thesis. As a starting point in the implementation process, an initial object oriented model that implements a simple fitness function $f(x) = x^2$, in Java is produced; it is known as the Simple Genetic Algorithm (SGA). Founded on this basic structure, the Simple Optical Genetic Algorithm (SOGA) rises as the most basic solution to the ring routed optical network design problem. The SOGA implements the algorithm for the problem at hand with minimal optimization features.

Following the simple solution, in chapter four, the algorithm is optimized and enhanced to achieve better performance relative to its predecessor. The new algorithm is known as the OGA for Optical Genetic Algorithm.

In chapter five, comparative studies emerge to test the suitability of the OGA and measure its performance against other streamline methods. In particular, the random and brute force searches are compared.

Finally, in the sixth chapter, a summary and discussion of the work is presented along with recommendations for future studies.

CHAPTER 2: LITERATURE REVIEW

The scope of this chapter intends to familiarize the reader with the basic workings of a Genetic Algorithm (GA). The reader will obtain a solid foundation of understanding in the field. In addition, the literature review is critical in order to locate other relevant studies by other researchers pioneering in the field, and expand on the existing models if any.

2.0 A BRIEF BACKGROUND TO EVOLUTIONARY COMPUTING

The theory of evolution is firmly rooted in the Darwinian teachings. Adaptation by means of natural selection poses an essential ingredient for the living continuum in spite of changing and harsh environment factors. Given a sufficient evolutionary time, an organism optimizes its survival qualities by either adapting itself, or passing on the superior genes to its offspring. With the advent of computer modeling, Evolutionary Programming (EP) is in its basic form mimicking the mechanisms by which nature and living organisms employed for millions of years in the fight for survival. (Mitchel, 96)

2.1 A CONDENSED HISTORY OF GENETIC ALGORITHMS

When John Holland first coined the Genetic Algorithm Theorem in his book Adaptation in Natural and Artificial Systems (1975), he was faced with lack of enthusiasm and much skepticism. Hence, in the early years, the work on Genetic Algorithms (GA's) was restricted to the University of Michigan pioneers. Once multiprocessor and parallel systems introduced a new complexity, the need for new methods was required. GA's were suddenly the main course of experimentation in serious research objectives. In no time GA's spread internationally. There are thousands of GA researchers today worldwide. Applications vary from different disciplines including business, engineering and social science. Some major contributors include David E. Goldberg's textbook Genetic Algorithms, in search, optimization, and machine learning, 1989 which remains an important reading for all GA newcomers.

2.2 ON GENETICS, POPULATIONS, AND EVOLUTION: THE ESSENTIALS

In order to understand a Genetic Algorithm, it is essential to be familiar with some basic terminology and genetic concepts.

Every individual possesses distinct phenotypic (or physical) features coded by an underlying genotype (or set of genes). A gene is hence the basic unit for coding an individual. The gene itself is coded within a chromosome. The latter is composed of alleles. The position of an allele within a chromosome strand is referred to as locus. Each allele represents a single attribute. The coding of natural alleles involves specific sequences of amino acids (later form proteins).

During the course of cell growth many factors would arise. Mutation is a way to alter the contents of one or more alleles. Causes of mutations are not well known, but mainly include point radiation. For GA purposes, mutation is known to occur at low frequencies affecting random locations. Another known factor is crossover of chromosomes. In the process, two chromosomes would swap parts of their strands (Figure 2.1). The point at which crossing over occurs is known as crossing point, or crossover point. Single or multiple points for crossing over may occur. This is dictated by a low frequency, randomly generated position(s).

Reproduction is strictly the production of new chromosomes from existing ones. The focus will be on two parents and two children. The parents are chosen by a selection process based on fitness values. In other words, within a population, individuals with best fitness values are more likely to be selected for reproduction. This is called the Selection process; many variations to this process have been studied.

It follows that the artificial gene used in a GA codes a single chromosome, which in turn encapsulates a set of alleles whose overall effect produce a meaningful value. Unlike its ribonucleic acid counterpart, the allele is restricted in its composition to a single binary digit. Hence, at least in a computer scientist's point of view, a strand of alleles, or a chromosome is literally a stored binary representation of a given value.

A set of individuals would make up a population. In a large-scale population study, observers can examine population size, fitness, along with other factors such as immigration, death rate, and the like. Population dynamics is such a field. To simulate such irregularities in an arbitrary population is irrelevant for our purposes, but essential to our method; a population must replicate and prosper, to go extinct is equivalent to a "no solution".

Parent 1: 1 1 1 1 | **1 1**   ⇒   Child 1: 1 1 1 1 **0 0**

Parent 2: 0 0 0 0 | **0 0**   ⇒   Child 2: 0 0 0 0 **1 1**

FIGURE 2.1: AN EXAMPLE OF ONE-POINT CROSSOVER

2.3 ARTIFICIAL GENETICS

The term "Artificial Genetics" refers to the notion of adapting natural observed phenomena into computer generated simulations; specifically, the term refers to the simulation of genetics and population evolution.

2.3.1 BIOLOGICAL AND NATURAL MODELING

Since the invention of the first computer, scientists strive to mimic nature in its success over the battle of survival. Simulating biological and natural phenomena went beyond mere curiosity reaching grounds levels of fictitious imagination and folk stories. With the failure of many highly funded experiments over the century criticism and scientific acceptance to such techniques reached stiffer levels. However, a handful of scientists in the 50's and 60's era have accomplished some eye opening experiments. Holland's work although was not formally manifested until 1975's publication of his text, he was influenced by earlier work. The important difference is that he armed his work with an irrefutable proof: the Schema Theorem.

2.3.2 THE SCHEMA THEOREM

The Schema Theorem and the Building Block Hypothesis form the mathematical foundation of Genetic Algorithms. The theorem was developed by J. Holland and first published in 1975. (Holland, 1975)

Schema Theorem:

Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.

Building Block Hypothesis:

A genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called the building blocks.

2.3 INGREDIENTS OF GENETIC ALGORITHM

Genetic algorithms are significantly different from alternative search strategies. Goldberg (1989) suggests that genetic algorithms differ from existing search strategies in four ways.

First, rather than using the parameters themselves, GA's use a coding of the parameters. From an Artificial Intelligence perspective, a GA is subsymbolic, representing and applying knowledge without symbol manipulation as would be the case in many AI techniques.

Second, GA's search from a population of points rather than from a single point (such as the typical hill-climbing algorithm, simulated annealing, or tabu search).

Third, GA's use payoff (objective function) information, not derivatives or other auxiliary knowledge.

Fourth, GA's use probabilistic transition rules, not deterministic rules.

Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet (Goldberg, 89). In order to make up a genetic algorithm, certain components are required. They are described in the following sections.

2.3.1 DOMAIN SET AND PARAMETERS

The set of all possible answers forms the domain set. As the problem grows more complex, its solution space could become very large. Conventional search algorithm, such as a linear algorithm, tends to comb the massive domain space sequentially and testing every value using an objective function. Such algorithm, although feasible and yield an absolute answer eventually. Its main drawback is that in large domains the time constraints that normally placed to solve a given problem overthrow the slow algorithm. Furthermore, these conventional algorithms act on varying the value of a single parameter and retest its objective value. This is in discord with the mechanics of the GA as previously stated by Goldberg (89).

Instead of acting on a single parameter and single solution, the GA differs in at least two ways. First, it searches for a solution pattern that account to all parameters, and second, it acts on a manageable small subset of the domain set, known as a population.

In the Thesis' proposed problem the domain set is numeric. However, there are multiple constraints and rules attached to these numbers. A valid solution is a permutation containing the numbers 0 through N-1 where N is the number of nodes in the network. For example, a network with six nodes would have as domain, and possible solutions, all the permutations of size 6. A quick calculation leads that there are 720 possible permutations that make up the solution space.

Possible Permutations:

0, 1, 2, 3, 4, 5
0, 1, 2, 3, 5, 4
0, 1, 2, 4, 3, 5
0, 1, 2, 4, 5, 3
...
5, 4, 3, 2, 1, 0
(720 Permutations total)

$P^{d}_{r} = n! / (n-r)!$

(n distinct objects, taking them r at the time)

For a typical network in the order of 100 nodes, one would only imagine the massive solution space (100 factorial). In that latter case, a linear search, or a brute force approach, would be of use only if time and resources are of no object. In critical problems whose constraints and parameters change overtime such traditional algorithms are rendered unfeasible. The GA emerges as an adaptive algorithm that can yield progressively improved solution subsets in the form of populations over multiple generations.

Constraints:

Mutation: 1,4,2,3,0 1,0,2,3,0 X

Cross Over:

1 4 2 3 0

2 3 1 0 4 can't cross

1 4 2 3 0

4 1 3 2 0   OK

Altering the permutation randomly could result in high lethality rate, hence should impose constraints (eg. [1 4] * * *) where "*" indicates any allele.

2.3.2 FITNESS EVALUATION FUNCTION

Every individual, and its underlying gene, is evaluated for its performance level. The fitness value is the calculated performance measure that provide the genetic algorithm in deciding whether to carry the individual's genes into future gene pools or not. Evaluating a Chromosome string performance is based on a mathematical function which itself is based on the variety of parameters and logical testing that simulates realistic criteria for a problem. The fitness function, also known as the objective function, generates an objective value where, in typical GA problems, the higher the value the better fit it indicates. Higher fit values are used in the selection process to pick the better fit individuals whose genes will be used to generate the following generation and maintain a progressively better fit genetic continuity overtime.

NUMERIC EXAMPLE:

Given:

Number of Nodes N = 4

and Matrix A [NxN] as follows:

	0	1	2	3
0	0	–	–	–
1	5	0	–	–
2	2	4	0	–
3	1	7	3	0

(Matrix representation of the traffic frequencies in Gb's)

For the permutation: **3120**

The cost value is evaluated as follows:

Cost (3120) = $[A(3,1) \times D_{min}(3,1)]$		$(7 * 1) +$
$+ [A(3,2) \times D_{min}(3,2)]$		$(3 * 2) +$
$+ [A(1,2) \times D_{min}(1,2)]$		$(4 * 1) +$
$+ [A(3,0) \times D_{min}(3,0)]$		$(1 * 1) +$
$+ [A(1,0) \times D_{min}(1,0)]$		$(5 * 2) +$
$+ [A(2,0) \times D_{min}(2,0)]$		$(2 * 1) =$ **30**

Where:

- A(i,j) is the associated communication cost (expected traffic in Gb's) between pairs of nodes i and j in a ringed routed network (circular).

- $D_{min}(i,j)$ is the minimum distance between the two nodes (in either direction since the network is closed and circular).

2.3.3 CODING / DECODING VALUES

The solution is coded as to what is referred to in the lingo as a gene, or chromosome. It is common to code a given value in binary format. For instance, the number 5 is coded as 0000 1001 where the chromosome length is 8. The length of the chromosome is fixed in the algorithm, and is originally given the size to accommodate the largest coded value from the solution space to be explored. The binary bit representation is not the only way to code a gene, in this thesis the integer representation is coded instead.

2.3.4 RANDOM NUMBER GENERATION

In order to exploit larger solution space, the algorithm depends on a careful gene shuffling in order to reconstruct input variables. For example, in single point mutation, an allele position is randomly chosen to be toggled. The end result is a mutated gene. There are other uses of random number generation, especially in the roulette wheel selection method. Other selection methods may also depend on randomly choosing parents.

In many cases, the use of a biased coin toss selection method also relies on random selection. In this scenario, an example would be mutation rate. For a mutation rate of 1%, we use the random equation to decide whether to mutate or not. The equation will return true - ie. mutate - 1% of the time, and hence conforming to the rate.

For simplicity, a language dependent generation function was reused. In Java, the function used is Math.random( ).

2.3.5 OPERATORS

To form a simple working genetic algorithm Goldberg (89) recommends three critical operators: Crossover, mutation, and selection.

2.3.5.1 Crossover

Crossover is critical to the search for better solutions, but it also has its disadvantages. It is disruptive by definition; by means of shuffling a chromosome string good chromosomes can be severely distorted. The more frequent the crossover is, the more chaotic the search would become; that is because the new distorted string does not capitalize on previous successes. [BAU94]. Infrequent crossing keeps the search confined to narrow regions of the search space.

There is a number of proposed crossover techniques:

One-point crossover: Only one cut is made in each chromosome.

Two-point crossover: Involves two cuts, and the middle sub-string is swapped.

Multiple-point crossovers have also been developed and tested.

Uniform crossover: developed by G. Syswerda in 1989, potentially allows any pattern to be swapped. It allows two parent strings to produce two children. First, we decide on a bit-by-bit basis how the child strings will relate to the parent strings.

2.3.5.2 Mutation

Mutation is an important operator whose main objective is to keep the population uniformity in check. Given a certain mutation frequency, usually 0.01%, random chromosome strings are selected and mutated. The scenario of breaking uniformity and introduce diversity is to allow the search algorithm to perform searches for other solution patterns, or graphically put it would be the search for other possible peeks in the function range.

Basic binary mutation used in the Simple Genetic Algorithm simply toggles the value of a randomly selected bit. A 00011001 string would be possible mutated to 00111001, where the third bit is toggled.

As constraints are imposed more heavily on the structure of the chromosome string, mutation operators become restricted and less flexible. For instance, in permutation sequences 012345 to simply toggle a random number to something else would create an invalid string, hence lethality follows. More carefully planned mutation algorithm have been heavily studied in the literature especially with TSP related problems.

2.3.6 SELECTION METHODS, REGENERATION

The initial population, or initial set of chromosome strands, is randomly generated as outlined in the Genetic Algorithm definition. The population corresponds to a set of candidate solutions to the problem at hand. With every subsequent generation, the population pool progresses closer and closer to the optimal solution. It is worth noting that the progression does not guarantee that the optimal solution will actually be reached, although it is possible.

In true adherence to Darwin's survival of the fittest scenario, in each population, which is assigned a generation number, if individuals were to sorted by their fitness values, then there would ideally be a range from best fit to worst fit individuals. A best fit individual implies that the solution conveyed by the underlying string is closer to the optimal than the less fit sibling. Furthermore, in creating a trend toward better overall population fitness overtime, it is essential to drop the worst fit individuals and replace them with new ones.

An introductory selection process to accomplish the elimination of bad fit individuals overtime is detailed by Goldberg (89) and known as the roulette wheel selection. (see figure 2.8)

2.3.7 PENALTY FUNCTION AND CONSTRAINT PARAMETERS

The constraints imposed on the structure of the chromosome strand play a major role in the fitness of the individual. The solution set criteria is very strict for a given problem. For example, in the travelling salesman problem, any possible solution must be a permutation. Otherwise, the sequence is not a feasible solution as pre-defined by the problem requirements. In a similar criteria, the problem at hand defined in this thesis requires a permutation as a solution. Nevertheless, if a crossover or any other operator is to be carelessly applied on a strand then there yields the possibility of generating a strand that is not a permutation. A simple example would be the single point mutation where a toggle on a single allele always generates an invalid solution, better known as a lethality. (eg. 012345 becomes 012445, where the 3 was randomly mutated to another number).

One way to deal with this problem is to allow lethalities at the reproduction level, but when it comes down to fitness evaluation the fitness value is assigned a negated (or zero) number that would guarantee that this strand, which represents an invalid solution, would not be chosen as a parent to pass its genes to the next generation.

Another obscure problem with lethality which only emerged itself later after the implementation is when in large string possibility spaces, and especially where the constraints are very strict, and thereby extremely narrowing the actual domain, the search for surviving solutions becomes unlikely. In the permutation example the possible search space for a strand of size 30 spans values from 0,0,0....0,0,0 to 29,29,29....29,29,29. When permutation constraints are imposed to the solution space, then the resulting possible solution space is restricted to only permutation spanning 0,1,2...28,29 to 29,28...2,1,0. Furthermore, given two valid strands (permutations), and using a crossover technique that simply swaps a randomly chosen sub-strand of two individuals will generate an invalid permutation. As it turns out from the results that, aside from minor statistical exceptions, the trend is that all generated genes fail to adhere to the permutation constraint. When the fitness function applies the penalty, no chromosomes may be chosen for subsequent generations. Therefore, it is advisable not to use the penalty function in this exercise and opt for more carefully planned operators.

2.4 THE SIMPLE GENETIC ALGORITHM

According to Goldberg (89), a simple genetic algorithm that yields good results in many practical problems is composed of three operators:

1. Reproduction
2. Crossover
3. Mutation

2.4.1 BACKGROUND AND FRAMEWORK

In order to achieve a basic understanding of the mechanics involved to produce a genetic algorithm, Goldberg (89) introduces a problem known as the Simple Genetic Algorithm (SGA). The problem employs a simple population and objective function for the sake of developing a skeletal GA and at the same time be able to test its functionality and observe its behaviour.

Ultimately, the problem describes a black box function that is fed candidate solutions as input and outputs a performance measure.

Ideally, the performance function is known, but in large space its output involves the search

2.4.2 EXAMPLE BY HAND

The following table illustrates a sample initial population. It is composed of 4 individuals (strings), each with a corresponding value, binary-coded and integer-decoded, fitness value, and the calculated probability to be selected for mating to form the following generation (PSelect). In this case, the selection method used is the roulette wheel as described by Goldberg (89).

Table 2.1 - Initial sample population (randomly generated) and its corresponding values

String No.	Initial Population	x Value (unsign int)	f(x) x^2	Pselect $f_i / \Sigma f$	Actual Count (Roulette)
1	01101	13	169	0.14	1
2	11000	24	576	0.49	2
3	01000	8	64	0.06	0 (not selected)
4	10011	19	361	0.31	1
		Sum	1170	1.00	4.0
		Avg.	293	0.25	1.0
		Max	576	0.49	2.0

Table 2.2 - Subsequent Generation based on the initial population and roulette wheel selection.

Mating Pool after Reproduction	Mate (randomly selected)	New Population	x value	f(x) x^2
0110|1	2	01100	12	144
1100|1	1	11001	25	625
11|000	4	11011	27	729
10|011	3	10000	16	256
			Sum	1754
			Avg.	439
			Max	729

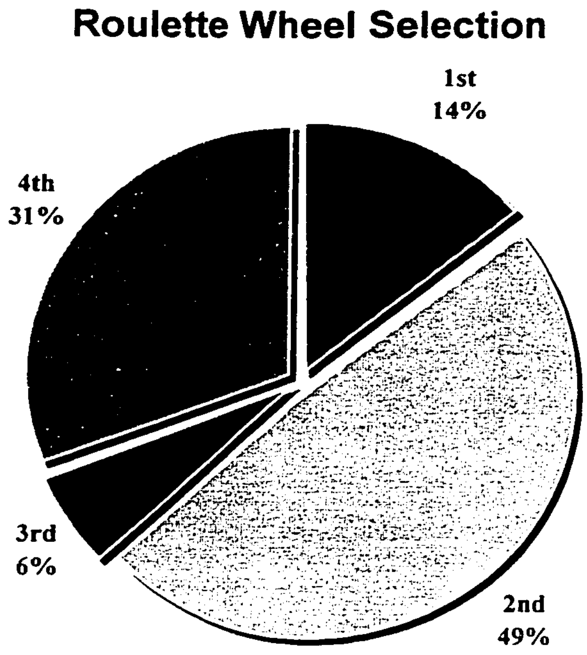


Figure 2.2: Roulette wheel selection

Roulette wheel selection for the 4 strings from the initial population.

2.5 IMPLEMENTATION OF THE SGA

As an entry point to the eventual development of a customized genetic algorithm that effectively solves the problem at hand, a need to establish a founding, or simplest case, algorithm. Goldberg's SGA, although intended for a simple problem dealing with an f(x) = x^2 function only, it encapsulates within it a number of essential concepts critical in creating a genetic algorithm. Therefore, as a start, the SGA is developed in the language chosen for the thesis work, namely Java, and re-designed to conform with the object oriented paradigms of the language.

Implementation code for the algorithm is supplemented in the Appendix for user's reference.

The workings of the SGA are the same as that of the one developed by Goldberg in Pascal. Once the SGA is constructed in Java, it is thoroughly tested with binary chromosomes, the same ones used in Goldberg's algorithm, for the sole purpose of replicating his scientific model. Indeed, upon executing the program the results have shown improving overall performance in populations overtime as expected.

Results of a sample run of a Java SGA:

The next step of the implementation is to revise the algorithm and make minimal alterations needs in order to solve the new fitness function required to solve the problem at hand. The next chapter introduces the Simple Optical Genetic Algorithm (SOGA) which is the outcome of the modified SGA.

The source code for this algorithm (SGA) is provided in the appendix for reference. In effect, after coding and executing this algorithm results have shown improvement in the search. The program was tested with a fitness function of $f(x) = x^2$, and with a population size of 10 and domain range from 0 to 100.

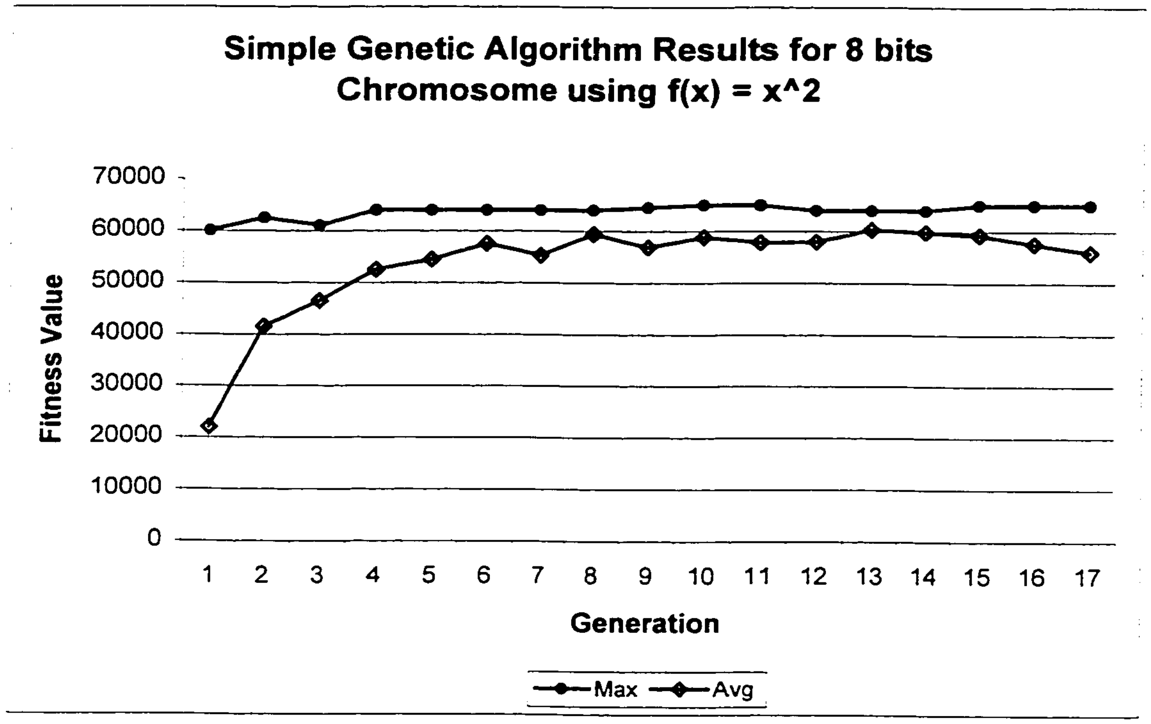


FIGURE 2.3: SIMPLE GENETIC ALGORITHM EXECUTION SHOWING INITIAL IMPROVEMENT AND CONVERGENCE

CHAPTER 3: SOLUTION MODEL CONSIDERATIONS

3.1 DESIGN FRAMEWORK

In the course of researching existing literature on Genetic Algorithms, there was no shortage of publications in the field. However, when narrowing down the search to focus on studies that cover optical network design, no matching publications were found. A related topic that is well investigated deals with permutation problems and especially the Travelling Salesman Problem (TSP) scenarios. Although the problem at hand is unique, it is possible to first approach it as a modified version of the TSP to certain extents.

Prior to starting the thesis work, as well as throughout the early stages of development, multiple exhaustive searches were conducted using the Internet and available library catalogues in the quest for a Java based source code of a Genetic Algorithm. At the time of search, only a handful of studies are documented to be using Java in GA implementation. However, no publicly available Java source code could then be located. Overtime, the search was repeated, only the number of researchers who are turning to Java as a language of choice is starting to grow drastically. At the time of writing this research paper, publications dealing with Java based Genetic Algorithms were identified (Papers are listed in the References section).

The first step in tackling the problem is to develop a simple working model in order to confirm the feasibility of using Genetic Algorithms in the quest of a proper solution. The implementation of the Simple Optical Genetic Algorithm (SOGA) was initiated and based primarily on the previously implemented and more general Simple Genetic Algorithm (SGA) predecessor.

3.2 MODELING THE SOGA

As a corner stone to the object oriented programming path that is adopted for the implementation of the solution, a simplified model is designed. Figure 3.1 reveals the hierarchical representation of the algorithm in detail. First, an underlying population defines the solution space. Each individual within the population represents a single candidate solution from a vast solution space. In turn, an elementary genetic representation, or gene depicts the individual. An individual may be composed of one or more gene. Finally, the gene itself is composed of as many alleles needed in order to properly represent a solution. Genes and individuals differ at an abstraction level.

An individual is measured based on its fitness relative to others in a population, while a gene represents a solution value or, in the case of a permutation, a sequence of nodes that form the solution. The algorithm includes regeneration and population selection methods that set the course of evolving solutions.

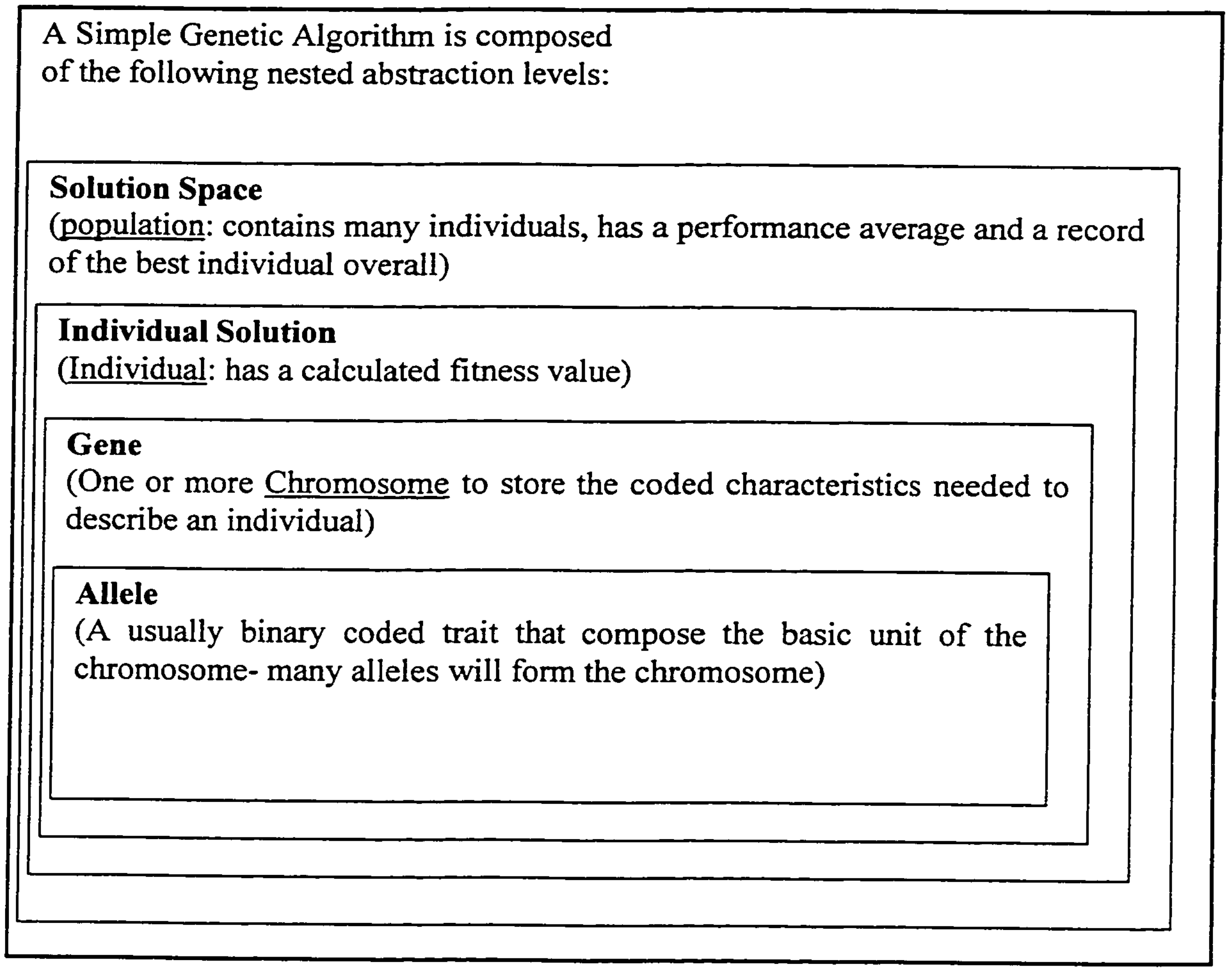

A Simple Genetic Algorithm is composed of the following nested abstraction levels:
Solution Space
(population: contains many individuals, has a performance average and a record of the best individual overall)
Individual Solution
(Individual: has a calculated fitness value)
Gene
(One or more Chromosome to store the coded characteristics needed to describe an individual)
Allele
(A usually binary coded trait that compose the basic unit of the chromosome- many alleles will form the chromosome)

FIGURE 3.1: GENETIC ALGORITHM OBJECT ABSTRACTION (ENCAPSULATION) MODEL

The Simple Genetic Algorithm (SGA) described by Goldberg (1989), is modeled as a first stepping stone in developing a custom genetic algorithm in Java.

The complex algorithm is broken into the following smaller entities in hierarchical order: The allele, the chromosome, the individual, the population, and the SGA.

1. The allele is the basic unit of a genetic algorithm. The object exports the basic entity of the gene model. In the SGA, it is a bit value represented by either 0 or 1. In the SOGA the allele is a positive integer identifying a node on the network.

2. The set of alleles aligned together of a given size n would form an array corresponding to the chromosome. The chromosome encapsulates and maintains the sequence of the alleles. It maintains their calculated value; for instance, it would maintain the decimal equivalent value of the binary array.

Encoded Value = 100101

Allele Representation	**1**	**0**	**0**	**1**	**0**	**1**
	2^0	2^1	2^2	2^3	2^4	2^5
	1	**2**	**4**	**8**	**16**	**32**

Decoded Value = 1 + 8 + 32 = 41

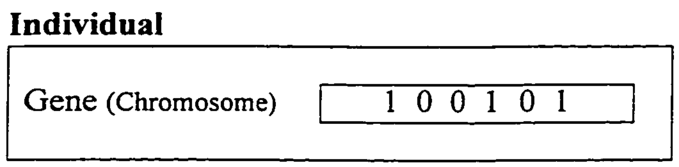
FIGURE 3.2: SIMPLE GENE BINARY REPRESENTATION

Figure 3.2 illustrates a sample chromosome composition. Alleles could be encoded in binary representation and the actual decoded value may need to be recalculated.

3. At the next level of abstraction, the Individual emerges as the active unit. It may include one or more chromosomes depending on the problem. In this case, it would be one chromosome per individual. Furthermore, at a conceptual level, an individual is examined by its fitness value as opposed to the actual decimal value equivalent in the chromosome. Within the individual, the fitness function takes place in the background calculating the fitness of the individual upon its creation.

Figure 3.3 shows the individual makeup in a simplest model that contains a single gene. Individuals may contain more than one gene, depending on the problem at requirements.

In the SOGA, an individual is composed of a single gene. The individual is described by its fitness, that is how well it measures as a solution.



Fitness: f(x)
where f(x) is the fitness evaluation function,
and x is the decoded value of the gene.

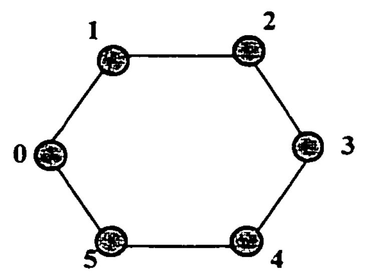
FIGURE 3.3: INDIVIDUAL REPRESENTATION

4. In the broader scale, the population is the major player. It is simply the set of individual as defined by the population size. Along follows the selection and re-generation mechanism deployed within the population. Depending on the criteria of the selection model, the population size is dictated as either a constant throughout or variable. A constant population size is maintained in the case of the SGA.

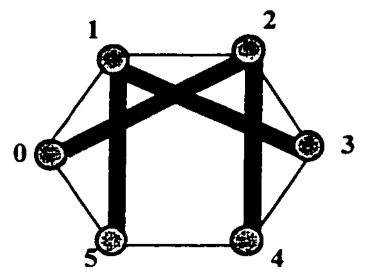
In addition, the population maintains a report generation scheme to produce detailed snapshots of the listed individuals and their sub-properties, as well as overall population performance measures. The algorithm will keep track of the minimum, maximum, and average fitness of the population.

5. Finally, as a test class, the SGA instantiates the population, which in turn cascades the creation down the hierarchy to the allele level, and guides the re-generation of subsequent populations.

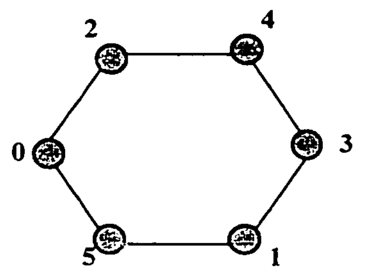
3.3 FROM SGA TO SOGA



Given a sequence of nodes located on a ring network, we need to take into account the communication patterns between them.



The thick line indicates the heavy traffic patterns associated between the nodes communicating with each other. In this example, heavy traffic between nodes 0, 2 and 4, and between 5, 1 and 3.



In order for nodes 0 and 2, for instance, to communicate with each other in the ring structure, packets travelling from 0 must path though node 1, then node 2. Or alternatively, can travel in the opposite direction and relay through nodes 5, 4, 3, the reach 2.
It would be however more efficient, based on this traffic pattern, to link 0, 2 and 4 together, and 3, 1, and 5 together as shown.

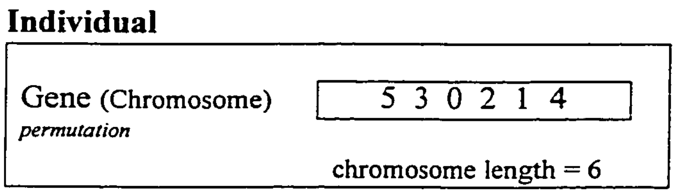
FIGURE 3.4: APPLICATION MODEL SCENARIO BY HAND

The simplicity of the SGA serves mainly as means to confirm the proper functionality of the object based model in Java. Moving on, the implementation of the first GA prototype that satisfies the application at hand proceeds. In order to obtain a conceptual understanding of the application, figure 3.4 explains how, based on a traffic pattern, the ring network design is adjusted. Initially, the algorithm has undergone intense work by hand in order to produce suitable candidate ingredients for the algorithm to meet the constraints put forth by the application criteria.

The following is a list of initial constraints and problems:

1. Permutation constraints on the parameters.
2. Mutation causing lethality.
3. Crossover causing lethality.

In the first criteria, the domain set of the problem is limited only to permutations. For example, a size 6 chromosome would have to be composed of a permutation such as 012345 or 530214 etc. The enforcement of this permutation is critical, since a non-permutation chromosome string is not a solution. One approach toward non-permutation strings is labeled under lethality. A lethal chromosome is one whose fitness evaluation function leads to a negative number (hence forcing it to be a minimum and discarded after selection). It is feasible to implement this criterion in the fitness algorithm and test it. However, it did not take long after executing the algorithm to realize that lethality is dominant in populations and the algorithm never succeeded in regenerating better solutions. The reason for this effect has to do with the fact that the domain subset that actually form the solution space, that is permutations, is a relatively smaller subset than the larger combination where the values are being selected from. The combination set is much greater than the permutation set, especially as the length of the permutation digits increases. Therefore, a random initial generation of the population is often composed of mostly lethal genes to begin with. The algorithm never recovered and took a turn to the worse. The use of penalty function had to be abandoned and constraints had to be introduced instead.



Fitness: f(x) = a positive numeric value based on the defined fitness function. Coding and decoding is not required since direct mapping representation is used..

FIGURE 3.5: INDIVIDUAL AND GENE REPRESENTATION IN THE SOGA

The first and most important constraint is to guarantee that, at any given time, a chromosome strand is valid and yield a solution; its performance at this point is not in question. The first step in the genetic algorithm is to randomly generate an initial population set to form the first generation. The individual genes, or chromosomes, that form the population are randomly generated. In the model at hand, we are using numeric representations of chromosomes. For instance, the string 042358176 represents a possible solution for the nodes sequence. It is a permutation indeed. The initial random string generation function simply generates random numbers between 0 and n-1 inclusive, where n is the size of the string. Appending the numbers together would form the final unchecked random string. In order to guarantee that the string is a permutation, the redesigned procedure works in a slightly different manner. It first starts with an ordered string from 0 to n-1, such as 012345678, and then it carries on random shuffling of individual alleles. First it randomly generate two different numbers indicating the position of the allele to be shuffled, and then performs a swap of the two alleles. In a manner of speaking it is similar to two point mutations repeated many times. The shuffling procedure is repeated n times to guarantee the possibility of reaching out any possible permutation. As a result, all the individuals in the initial population are valid chromosomes strings and hence candidate solutions.

In the regeneration phase, where crossover and mutations occur, the possible production of invalid strings arises again and need to be dealt with in order to maintain permutations at all time. The mutation procedure is dealt with first due to its striking simplicity. The two point swap is implemented, and this procedure is well documented in almost every literature that deals with TSP problems. It 's implementation involves the random choice of two loci positions and interchanging the alleles involved. Example, one mutation of gene 042315876 would be 0**8**2315**4**76. The bold digits show the two alleles that were swapped.

The crossover algorithm however is more demanding and requires further investigation.

3.4 CROSS OVER PROBABILITY AND CORRECTION FACTORS, DISCUSSION

Given an initial population G_0 (created randomly), and prior to re-generating we are addressing the issue of dealing with a correction factor to achieve the proper indicated probability of crossing over.

In a simple GA, the selection process is random, but toward a bias. (ie. those with higher fitness values will get a better chance to be picked.

However, in this model, when we pick random strings we risk the problem of survability. If the two selected strings do not achieve some compatiblity in the permutation sequence then crossing them over will ultimately lead to a lethality.

For example: String 1: 5624310 and String 2: 6235401 cannot be crossed over successfully. (The compatibility pattern is in the form [abcd]||[ef] meaning, for a string (chromosome) of length 6 and cross-over site of 4 we need to have the first 4 alleles of String 1 to be the same as those in string 2 (but may of course be in any order - or permutation). It follows that the remaining two alleles are also the same (different order maybe).

Once the two strings are selected, crossover and mutate are needed as dictated by a probability factor already in effect.

The issue becomes as follows:

In order to achieve a given probability of crossing over (say 0.1%) then the selection process must be carefully executed to ensure non-lethal combinations. In this case the constraint satisfaction presents an overhead, plus the risk of not finding compatible pairs in a limited size population.

As an alternative, the probability for the latter event to happen is calculated.

For a string of length 10 (i=10):

Step 1:

$10! = 3{,}628{,}800$
$9! = 362{,}880$
$8! = 40{,}320$
$7! = 5{,}040$
$6! = 720$
$5! = 120$
$4! = 24$
$3! = 6$
$2! = 2$
$1! = 1$

Step 2:

$1! \times 9! = 362{,}880$
$2! \times 8! = 80{,}640$
$3! \times 7! = 30{,}240$
$4! \times 6! = 17{,}472$
$5! \times 5! = 14{,}400$
$6! \times 4! = 17{,}472$
$7! \times 3! = 30{,}240$
$8! \times 2! = 80{,}640$
$9! \times 1! = 362{,}880$

SUM = 996,864

Correction Factor = SUM / 10!

= 996,864 / 3,628,800

= 0.2747

Corrected p-CrossOver = p-CrossOver / Correction Factor

= 0.1 / 0.2747

= 0.364 call it Cp-crossover

We have to ensure that the $0 < \text{Cp-crossover} < 1$

(since it may get larger than 1)

This method restricts us from choosing the crossover site randomly, rather it does a search for compatible ones, and if none is found, it will use itself.

To elaborate, the search goes as follows:

Start from first allele, and check if there is another string that starts with same allele. (a linear search of the population for now) If one match is found, then this will be used as a partner for crossover. If not, we proceed to the next allele and check if there exists another string whose first 2 alleles are a permutation of our original 2 alleles. Proceed the search until the population is exhausted. If none is found, then crossover the original string with itself.

Cross Over Algorithm- Corrective Probability

1	Randomly select two individual **I_1** and **I_2** from the population (biased selection as outlined in Roulette Wheel Selection Algorithm)	e.g. I_1 : 2 3 1 5 0 4 I_2 : 3 1 2 0 4 5
2	Find the first crossover point starting from the left most allele: 1. Set Crossover point at 0 (i.e. no cross over) 2. Initialize mask bits to 0. 3. Loop across the gene alleles and toggle the corresponding mask bits. Increment the location of the crossover point *xpos* with every iteration. 4. Keep looping until the two mask values are identical. 5. Perform cross over at position *xpos*. if *xpos* = the size of the gene (ie. 6 in this example) then no crossover is done because the no valid crossover point can be found for the two incompatible chromosomes.	I_1 : | 2 3 1 5 0 4 xpos = 0 I_2 : | 3 1 2 0 4 5 0 1 2 3 4 5 I_1' 0 0 0 0 0 0 I_2' 0 0 0 0 0 0 I_1 : 2 | 3 1 5 0 4 xpos = 1 I_2 : 3 | 1 2 0 4 5 0 1 2 3 4 5 I_1' 0 0 1 0 0 0 I_2' 0 0 0 1 0 0 I_1 : 2 3 | 1 5 0 4 xpos = 2 I_2 : 3 1 | 2 0 4 5 0 1 2 3 4 5 I_1' 0 0 1 1 0 0 I_2' 0 1 0 1 0 0 I_1 : 2 3 1 | 5 0 4 xpos = 3 I_2 : 3 1 2 | 0 4 5 0 1 2 3 4 5 I_1' 0 1 1 1 0 0 I_2' 0 1 1 1 0 0 crossover possible at position 3
3	Crossover occurs at position 3, resulting in two offsprings with valid permutations. Overall, this process guarantees to yields non lethal offsprings, regardless of whether a crossover can be performed or not.	I_1 : **2 3 1** | **5 0 4** xpos = 3 I_2 : 3 1 2 | 0 4 5 Crossover yields: I_1' : **2 3 1** | 0 4 5 I_2' : 3 1 2 | **5 0 4**

FIGURE 3.6: MODIFIED CROSSOVER WITH COMPATIBILITY TESTING

3.5 JAVA CLASSES

The implementation of the Java classes is supplemented in the Appendix of the thesis. The structure and make up of the classes reflect very closely the encapsulation and hierarchy described earlier in this chapter. The only prevalent disadvantage in using Java as a language for implementation is its speed. Typically, Java programs execute slower on virtual machines (Java VM) due to built in issues in the engine running on a host machine. However, this problem could be eliminated by using the recently release JIT and native language compilers that convert java classes to actual native machine executable code. The cost of course is that the resulting executable is machine dependent, and the Java program will have to be recompiled for every platform it needs to execute on.

3.6 SOGA

After establishing the initial foundation for the custom genetic algorithm and the application solution model, the implementation of the SOGA takes off. The implementation is supplemented in the Appendix. In the course of development, some additional concepts arose and helped shape the final algorithm.

3.6.1 SELECTION

Regeneration at this time is based on the SGA's approach - the roulette wheel selection As described by Goldberg (Goldberg, 89). Individual parents are chosen based on their fitness values. Ideally, the higher fit individuals are more likely to be chosen that the lower fit counterparts.

Roulette Selection Algorithm:

1. Calculate the fitness value for every individual in the population.
2. Compute the sum total fitness of all the individuals
3. Generate a random number from 1 to the sum calculated from 2.
4. Loop in sequence starting from the first individual in the population list while accumulating their fitness total. Continue until the new total is smaller than the full total (from 2). The individual at which the loop stop is selected.

3.6.2 MUTATION

The mutation operator is carefully modified to use the swapping technique. This is where two alleles are randomly chosen and swapped positions. This technique is well documented in variety of literature and it is suitable for its simplicity, speed, and controlled frequency. A default rate of 0.01 is used.

3.6.3 CROSSOVER

The corrected crossover algorithm that prohibits lethality is deployed (discussed earlier). A crossover rate of 0.5 is used.

3.6.4 FITNESS FUNCTION

A minor setback was developed by the fitness function at development time. The fact that the accumulated value computed by the function indicates the cost associated with an individual reversed the testing logic. The higher the fitness value, the more costs are associated with the subjected individual. The genetic algorithm however, searches for a maximum, indicating that the higher the value the better the performance. In order to correct this factor, the accumulated value from the fitness function is subtracted from a large sentinel value. That is the final fitness value f(x) is : $f(x) = 10000 - f'(x)$; where 10000 is some large sentinel value, which may be adjusted accordingly to ensure a non-negative value for fitness.

3.7 MODEL EXECUTION AND RESULTS

The initial model was extensively tested with many random variations in order to ensure its performance and quality in finding suitable solutions. The mindset at this point is to test whether node sequence patterns can be detected, and overall whether the performance can match that of a heuristic search in small samples.

3.7.1 TEST SETUP

In order to check for patterns a biased matrix was constructed. The idea is to simulate patterns by forming communication clusters. For instance, to simulate that two nodes 1 and 2 communicate frequently with each other, the communication matrix will reflect a high number between them. Consequently, the resulting search query should establish the pattern where nodes 1 and 2 are next to each other. Templates would include something in the form of 12**** or 21****.

Biased Clustered Matrix

	0	1	2	3	4	5
0	0	1	9	1	1	1
1	1	0	1	9	1	9
2	9	1	0	1	9	1
3	1	9	1	0	1	1
4	1	1	9	1	0	1
5	1	9	1	1	1	0

Random Matrix

	0	1	2	3	4	5
0	0	4	2	7	9	3
1	4	0	6	8	3	4
2	2	6	0	3	2	9
3	7	8	3	0	6	3
4	9	3	2	6	0	2
5	3	4	9	3	2	0

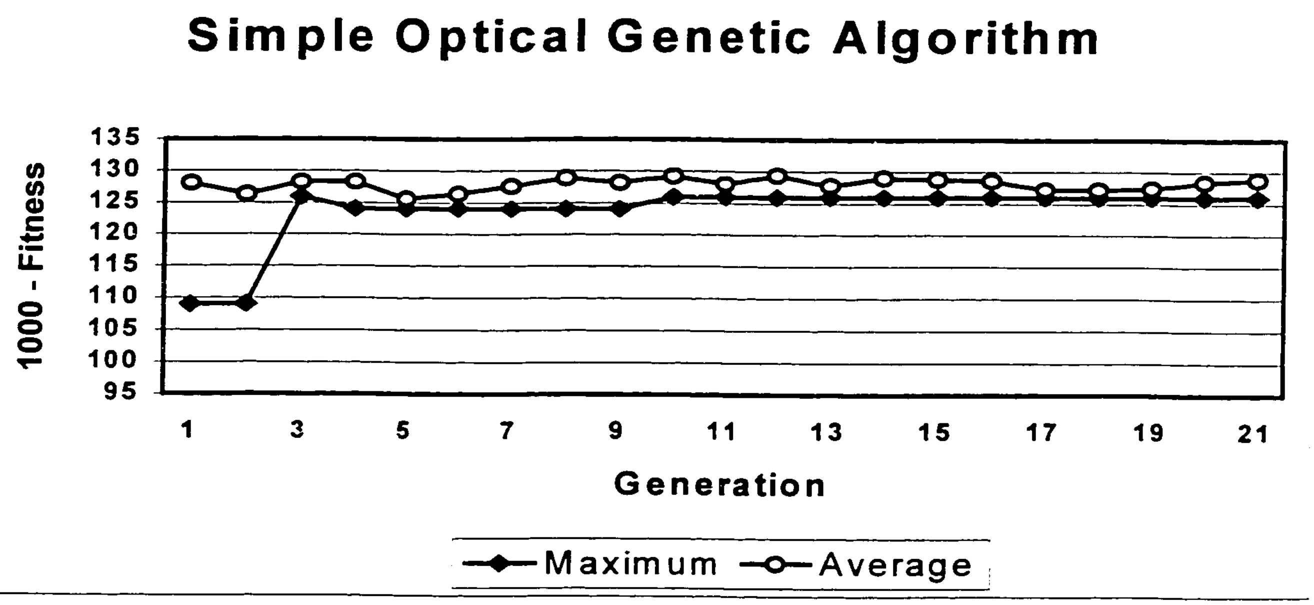
Simple Traffic Matrix:

	0	1	2	3	4	5
0	0	1	9	1	1	1
1	1	0	1	9	1	9
2	9	1	0	1	9	1
3	1	9	1	0	1	1
4	1	1	9	1	0	1
5	1	9	1	1	1	0

Patterns:
024###
420### = 0###43 and
513### or
315### = 5###31
(all cyclic combinations are accepted)

Best Combinations:
024513, 024315, 420513 (051342), 420315 (031542)

FIGURE 3.7: BIASED AND RANDOM MATRIX SIMULATION



```
Simple Genetic Algorithm - (c)1997-1998 by Ziad Kobti - All Rights Reserved
============================================================================
pcross = 0.7741935483870968

Generation:0  Max: 891.0 Min: 855.0 Avg: 871.9 Total Fitness: 17438.0
Generation:1  Max: 891.0 Min: 863.0 Avg: 873.7 Total Fitness: 17474.0
Generation:2  Max: 874.0 Min: 863.0 Avg: 871.7 Total Fitness: 17434.0
Generation:3  Max: 876.0 Min: 870.0 Avg: 871.7 Total Fitness: 17434.0
Generation:4  Max: 876.0 Min: 870.0 Avg: 874.4 Total Fitness: 17488.0
Generation:5  Max: 876.0 Min: 870.0 Avg: 873.6 Total Fitness: 17472.0
Generation:6  Max: 876.0 Min: 870.0 Avg: 872.4 Total Fitness: 17448.0
Generation:7  Max: 876.0 Min: 870.0 Avg: 871.1 Total Fitness: 17422.0
Generation:8  Max: 876.0 Min: 870.0 Avg: 871.8 Total Fitness: 17436.0
Generation:9  Max: 874.0 Min: 870.0 Avg: 870.8 Total Fitness: 17416.0
Generation:10 Max: 874.0 Min: 870.0 Avg: 872.0 Total Fitness: 17440.0
Generation:11 Max: 874.0 Min: 870.0 Avg: 870.6 Total Fitness: 17412.0
Generation:12 Max: 874.0 Min: 870.0 Avg: 872.2 Total Fitness: 17444.0
Generation:13 Max: 874.0 Min: 870.0 Avg: 871.0 Total Fitness: 17420.0
Generation:14 Max: 874.0 Min: 870.0 Avg: 871.2 Total Fitness: 17424.0
Generation:15 Max: 874.0 Min: 870.0 Avg: 871.4 Total Fitness: 17428.0
Generation:16 Max: 874.0 Min: 870.0 Avg: 872.8 Total Fitness: 17456.0
Generation:17 Max: 874.0 Min: 870.0 Avg: 872.8 Total Fitness: 17456.0
Generation:18 Max: 874.0 Min: 870.0 Avg: 872.6 Total Fitness: 17452.0
Generation:19 Max: 874.0 Min: 870.0 Avg: 871.6 Total Fitness: 17432.0
Generation:20 Max: 874.0 Min: 870.0 Avg: 871.2 Total Fitness: 17424.0
Number of Cross-Overs: 319
Number of Mutations  : 246
```

FIGURE 3.8: SAMPLE EXECUTION AND SIMPLIFIED RESULTS REVEALING THE CONVERSION OF THE SOLUTION

A large number of samples were tested with varied chromosome sizes ranging from 6 to 12. It was observed that in some cases the SOGA achieved optimal solutions. Also, in the cases involving the biased matrix, the algorithm have shown to favour solutions with the predicted pattern. However, when tested against larger samples, that is larger chromosomes, the algorithm showed early conversion at times, rendering the solutions vulnerable to chance failure, thereby denying the genetic algorithm its robust reputation.

In order to properly run the algorithm against larger chromosome sizes, it should be revised to encompass a more aggressive search mechanism in order to avoid solution fixation and early convergence. Some suggestions include higher mutation factors, and others recommend more aggressive crossover procedures. These enhancement factors, along with other advanced techniques will be introduced in the next phase of development that introduces the Optical Genetic Algorithm (OGA).

CHAPTER 4: THE OPTICAL GENETIC ALGORITHM (OGA)

4.1 IMPLEMENTATION DEVELOPMENT DIRECTION

Following the initial development of a minimal working model, the investigation is now geared toward an attempt to improve performance yield and attempt a full scale 100 node wide network problem. Since optimization involves a complex study on its own, the prime choice is to seek other researchers lead in optimization studies and adapt existing techniques while closely monitoring improvements.

Due to the lack of directly related studies to the problem at hand, the closest studied problem scenario involves the Travelling Salesman Problem (TSP). Much research has been conducted by scientists that show various techniques for optimizing the Genetic Algorithm used in the TSP solution model. Unfortunately, such attempts are tested based on unrelated problem applications. In order to realistically enhance the model at hand, one should ultimately exhaust all the available literature on the optimization criteria and adapt them to the problem in question and select the best based on the overall test results. In practice, optimization deserves its own research paper. For the purpose of the algorithm developed in this thesis, the optimization is limited to re-introducing an alternate crossover technique that is recommended by other studies as a suitable alternative. Recommendations from other studies are also noted in terms of selection and mutation techniques.

Staging a worst case scenario can assist in creating the framework of the study by enhancing our prediction of what a reasonable solution is. In the case of the Genetic

Algorithm being developed, failure to detect better fit patterns and hence divergence is the ultimate disaster. Showing convergence and improved near optimal values are on the other hand a sign of a successful algorithm.

4.2 TSP REVISITED

The basic SOGA, described in the previous chapter, achieved its goal in terms of presenting the GA as a feasible technique to solve the problem in a smaller scale. More importantly, it showed the extents and weaknesses of the algorithm. For instance, the selection routine of best fit individuals to be re-used into the next generation was simple based on a game of chance, that is the roulette wheel selection. Best-fit individual are likely, but not guaranteed, to survive over the next generation.

While keeping the problem of optimization in mind, many TSP related literature was revisited. The process led to a modest list, from various researchers, that provide criteria for different crossover, mutation, and selection techniques. The References section lists the major papers. The drawback remains the fact that in order to test and compare every selection technique would be time consuming and in practice deserves an independent study, which is beyond the scope of the thesis.

The choice of which algorithms are to be adopted into the new Optical Genetic Algorithm (OGA) are based on deductive reasoning and supported results and recommendation from the studies of origin.

4.3 INTRODUCING THE OPTICAL GENETIC ALGORITHM (OGA)

Implementation of the SOGA at this point produced a working solution model for the thesis problem. It provided both a feasible solution and shed more insight to learn more about Genetic Algorithms. The main course of the thesis now shifts to explore Genetic Algorithms in terms of attempting to combine different known techniques and push the limit to solve the 100 node application problem.

In the course of development, many choices that were used in the implementation of the SOGA have failed or led to weak results. Problems ranged from lethality to failure to converge. Such encountered pitfalls include different aspect of the GA:

- The use of digit (integer) representation instead of binary modeling in order to simplify the coding/decoding schemes of the stored values.
- The development of the single point swap mutation as opposed to random toggling of genes that often rendered the value unusable.
- Evolvement of the crossover solution from single point crossover to more complex look-ahead checking and probability correction analysis
- Goldberg's roulette wheel selection that relied on frequencies and statistical chance in choosing mating pools.
- Choosing to establish constraints over penalty as the odds of generating an invalid gene (not a permutation) are frequent.
- Many runtime tweaks and alterations on populations sizes, mutation, and crossover rates. A thorough investigation of the rate values deserves its own study and is beyond the scope of this thesis.

4.4 IMPROVED OPERATORS AND TECHNIQUES

4.4.1 CROSSOVER

Single point crossover caused major pitfalls and in fact prompted the redesign of the whole genetic algorithm in resorting to the corrective crossover method described in chapter 3. The imminent observation of this algorithm dictates that nearly every crossover leads to a lethal gene. Hence, the population would likely go instinct at the first regeneration, rendering the genetic algorithm useless. But it was also observed that a very small ratio of these genes can in fact be successfully crossed over. The first attempt to this problem was to perform this compatibility testing and couple it with an increased calculated rate of crossover in order to achieve the original rate. For example, a calculated crossover rate of 4.5 percent would be used in order to achieve the originally desired 0.1 percent; since a lot of those would be crossovers will be cancelled after checking the compatibility of the genes. This approach indeed led to a successful regeneration, however, the overall performance of the genetic algorithm was sacrificed due to the added complexity embedded with this specific crossover technique. Nevertheless, the original SOGA model proved successful for small solution spaces, tested for up to the magnitude of 12!.

After resorting to new literature, new research work showed an unusually aggressive and active crossover algorithm. The heuristic crossover, designed for permutation and TSP problem sets, works with any gene and literally builds the new gene based on a decisive comparison in reclaiming alleles.

In the TSP problem, the heuristic crossover builds an offspring by choosing a random city as the starting point for the offspring's tour. Then it compares the two edges (from both parents) leaving this city and selects the better (shorter) edge. The city on the other end of the selected edge serves as a starting point in selecting the shorter of the two edges leaving this city.

This algorithm was adapted to the OGA with relative ease since the basic standard structure of the algorithm, and the Object Oriented Design allowed the 'plug-and-play' capability. Note that with this technique combined with the new selection method introduced in this chapter obsolete the use of direct crossover probabilities. There remains a probability, but it is now shifted to the selection routine. There it is decided on how many top performer genes to keep, how many to duplicate, and all the rest enter the pool of being used in crossover. For the preliminary OGA, all the middle genes are used in the crossover.

Crossover Algorithm by means of an example:

Given 2 parent strings:

P_1 = 3 0 2 4 1 5
P_2 = 2 1 5 0 3 4

Sample traffic/Cost Matrix A:

	0	**1**	**2**	**3**	**4**	**5**
0	0	4	2	7	9	3
1	4	0	6	8	3	4
2	2	6	0	3	2	9
3	7	8	3	0	6	3
4	9	3	2	6	0	2
5	3	4	9	3	2	0

Generate Offspring as follows:

1	Obtain the first allele (left most) of parent P_2 (P2 also may be used as tie breaker later encountered in step 4)	Offspring value: **2 - - - - -**
2	Locate the allele value that succeeds allele '2' in each parent. For $P_1 \rightarrow 4$ and $P_2 \rightarrow 1$ Since the two values are different, resort to the traffic matrix and retrieve their cost values in relation to '2'. A(2, 4) = 2, and A(2, 1) = 6 The next allele would be '1' since its cost is higher, i.e. nodes '2' and '1' communicate more frequently with each other and are better located together.	**2 1 - - - -**
3	Repeat step 2 with allele '1': '5' succeeds '1' in both parents, so it is chosen	**2 1 5 - - -**
4	Repeat step 2 with allele '5': Obtains: '3' from P1 and '0' from P2 A(5, 3) = 3 and A(5, 0) = 3 Special tied case, may break the tie by using the one form parent 2 (since it is the tie breaker by choice)	**2 1 5 0 - -**
5	Repeat step 2 with allele '0': Obtains: '2' from P1 and '3' from P2 A(0, 2) = 2 and A(0, 3) = 7 Use '3' as the next allele	**2 1 5 0 3 -**
6	Repeat step 2 with allele '3': Obtains: '0' and '4' Since 0 is already used, choose '4'	**Resulting Offspring:** **2 1 5 0 3 4**

FIGURE 4.1: GREEDY CROSSOVER EXAMPLE

4.4.2 MUTATION

The two point swap mutation was maintained for its simplicity. It also was a dominant approach used in TSP problems in the GA literature.

4.4.3 SELECTION PROCESS

The roulette wheel selection, used in Goldberg's Simple Genetic Algorithm as well as many prominent studies[ref], has backfired in terms of maintaining an already established maximum. In other words, each generation has at least one 'best solution'. Although that solution would have a high probability to be selected for the next mating pool, there is the dim chance that it does not get selected and over the generations we observed that the 'best solution' from a previous generation is lost as it does not get selected. As a result of this observed phenomena over multiple executions, it was imperative to find an approach that both aggressively promoted the good genes to the mating pool as well as maintains the persistence of the best local solution over the generations.

A combing of the literature landed me a study that implemented a fairly aggressive routine to guarantee the selection of the best gene set from a population. Furthermore, the technique duplicates those good genes to override the weaker ones. It not only enhances the gene pool involved in selection, but also assists in faster instinction and discards the weaker genes.

The sub algorithm involves a preliminary sort to the genes according to their fitness. The top 10% performers are copied over to the next generation, hence guaranteeing a persistent maximum. Next, reproduce a number of the top performers (a small number such as 2 is suggested) to override the worst performers; hence eliminating the worst values. The remaining genes are used in the typical selection process, and crossover, to populate the remainder of the new population.

In general the procedure involves sorting the individuals based on their fitness values; keep the top 10% performers and carry them over to the next generation; reproduce the top 2% to override the worst 2%; reselect the rest and use them in crossovers.

4.5 RESULTS

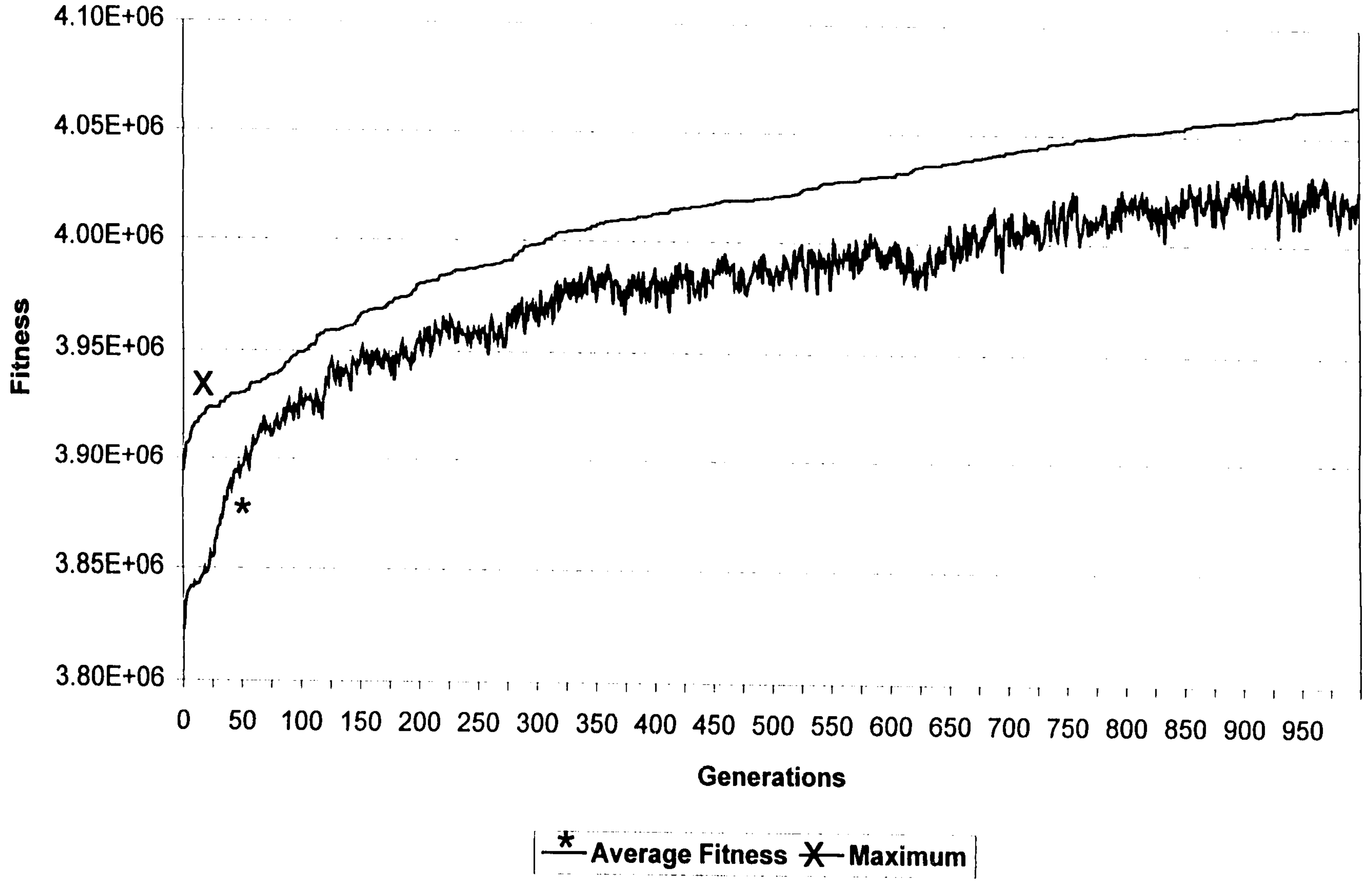
The primary objective of the experimental OGA is to demonstrate improvement in the quality of the generated solutions. With every regeneration the population contains individual solutions that are closer to the optimal solution. The OGA is tested for this purposed and contrasted to its predecessor, the SOGA. Further optimization models are beyond the scope of this thesis and will be studied in future research.

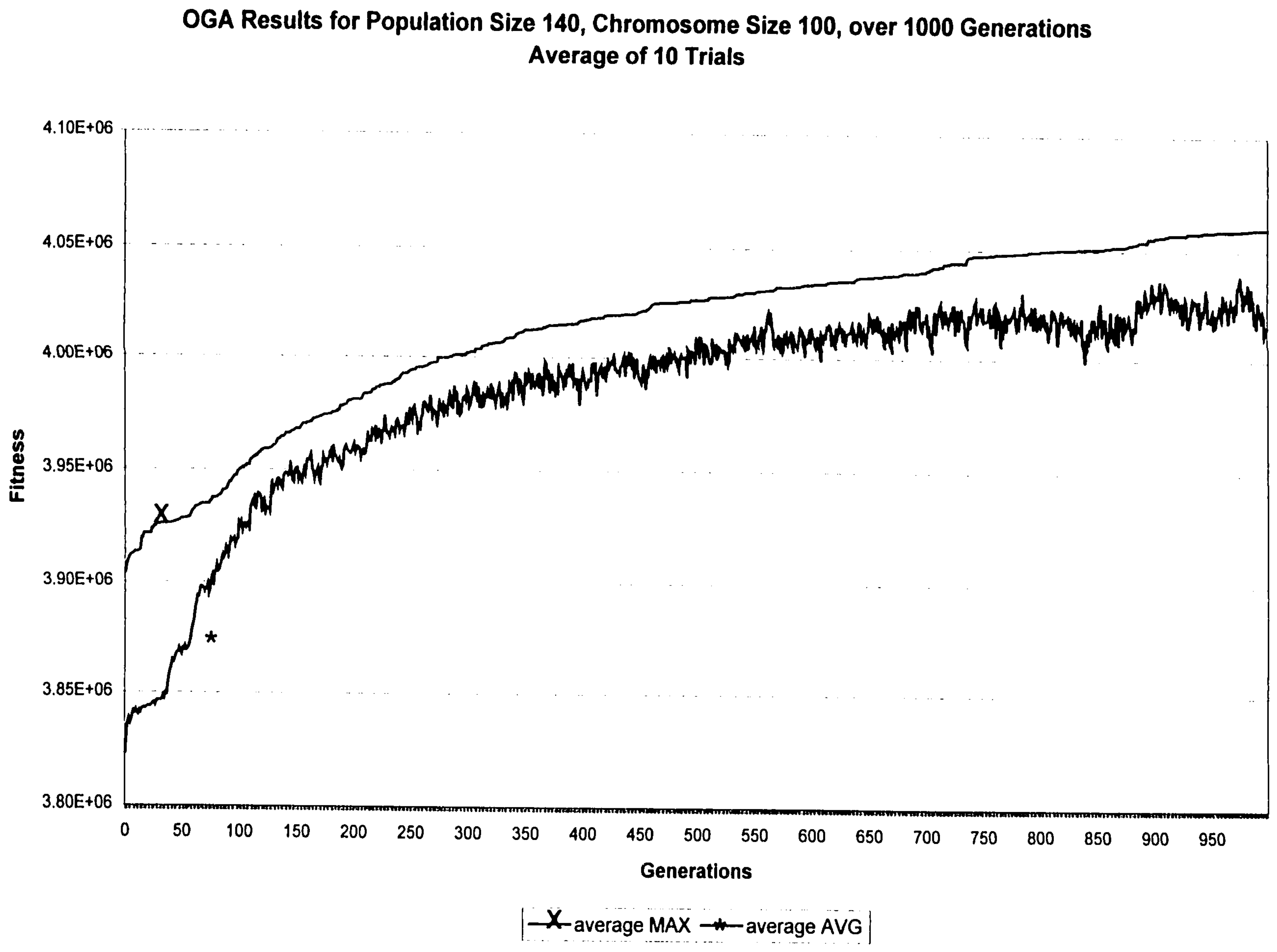
The sample execution run consisted of a fixed communication matrix size of 30x30, hence requiring a chromosome strand of length 30. Crossover rates, which are now dictated by the selection ratios, are not altered. Mutation rate also maintained at the previously recommended constant size. The only variable consisted of the population size.

Sample runs were conducted and the results are shown in the figures in this chapter.

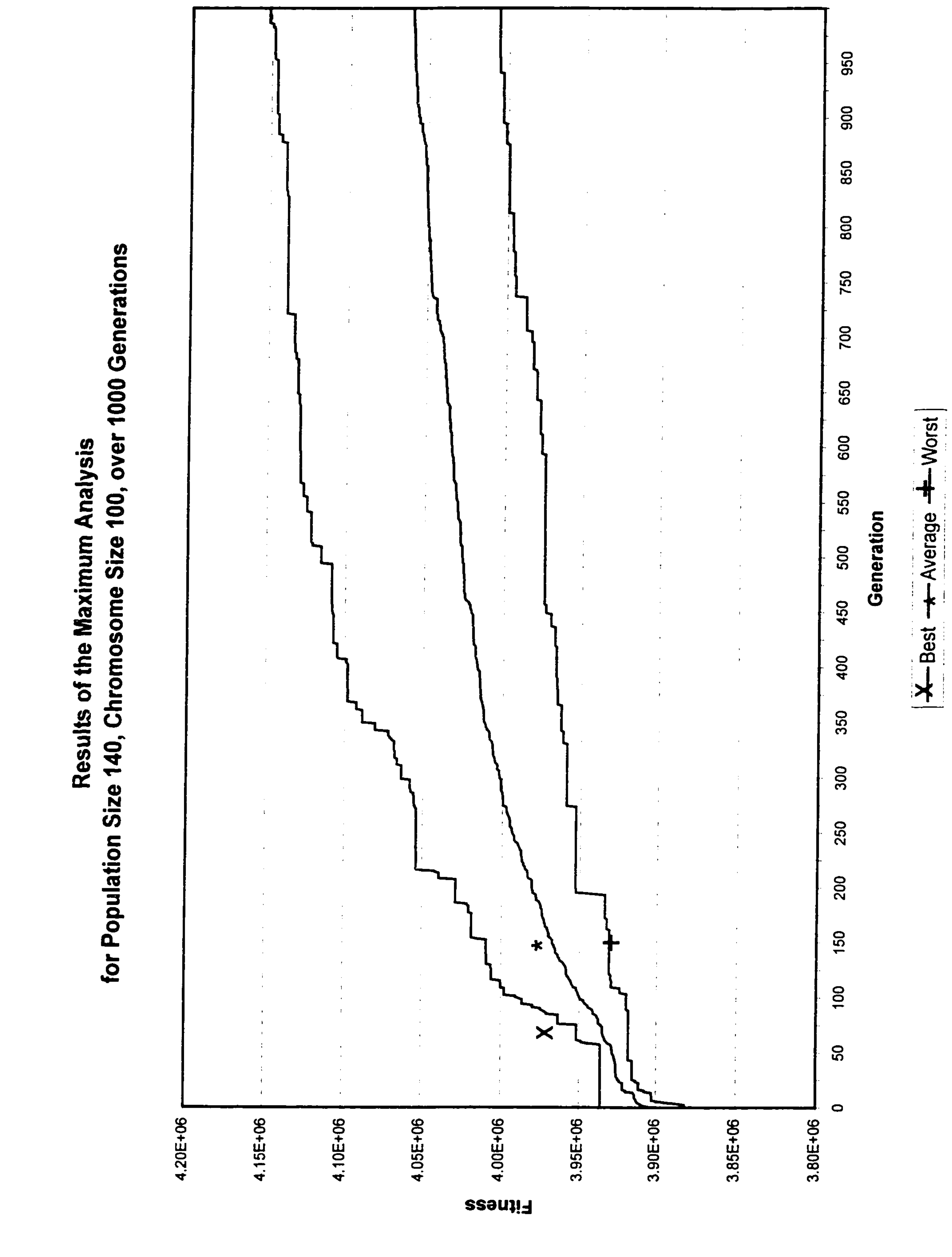
The dominant observation based on the results concludes that larger populations (300+) tend to yield more optimal answers within 10 generations. On the other hand, the smaller the population size the longer it takes to yield favourable solutions.

OGA Results for Population Size 100, Chromosome Size 100, over 1000 Generations, Average of 10 Trials




OGA Results for Population Size 140, Chromosome Size 100, over 1000 Generations
Average of 10 Trials
Fitness
4.10E+06
4.05E+06
4.00E+06
3.95E+06
3.90E+06
3.85E+06
3.80E+06
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
Generations
average MAX
average AVG

Results of the Maximum Analysis
for Population Size 140, Chromosome Size 100, over 1000 Generations



CHAPTER 5: EVALUATION AND COMPARATIVE STUDIES

5.1 COMPARATIVE TECHNIQUES

Given a particular traffic matrix, crossover and mutation probabilities, the execution of the genetic algorithm on its own would produce results of unknown performance as to how well they actually solve the problem. By means of tracking the average fitness of each generation over time, it is possible to see the trend of the performance. However, without knowing beforehand the absolute boundaries of the solution space it would be impossible to predict the improvement ratio.

The major barrier to compare the performance of the GA is ironically the very reason why the GA is being used in the first place. To reiterate, due to the massive solution space that needs to be searched conventional search methods and systematic searches are not feasible due to time constraints. However, if such systematic and heuristic methods can be tested against smaller domains then their resulting trends can be extrapolated for larger solution spaces.

Two methods were developed for the purpose of comparing their outcomes to that of the GA. First, the brute force heuristic approach is used to search the complete domain for solutions. As a result, for a given problem scenario, that has its own cost matrix and network size, it is possible to establish the absolute boundaries of the fitness values. The second approach, loosely termed as random search, is based on a random search of the solution space. Its primary purpose is to establish that the genetic algorithm is not merely a random search. This is critical since the GA itself makes use of random selections in many of its stages.

Other proposed search algorithms included tabu search, a hill climbing based model. However, due to the time limitations allotted for this thesis work these studies will be observed in future work.

5.2 BRUTE-FORCE HEURISTIC

The brute force heuristic search method plays an important role in initial development. Its primary purpose is to perform a complete search of the solution space and reproduce the best as well as the worst overall values, or individuals. As a result, by knowing the absolute limits of the solution ahead of time, it becomes possible to study the GA development in terms of its performance and improvement.

The brute force is basically a linear search modified to account for permutations and cyclic redundancies. For instance, the algorithm will start with 0123 and proceed to 0132, 0213, ... , until 3210. Due to the nature of the application problem at hand, that is with the ring network topology, strings such as 0123 and 2301 are equivalent since they represent the same sequence of nodes. Since the linear search has its inherit reduced performance, such optimization was implemented to speed up the process.

General Procedure:

1. Starting with the permutation 0,1,2,3,4,5,6,7,...,n-1, calculate the fitness value based on the same fitness evaluation used in the OGA. (where n is the length of chromosome)

2. Iterate until the ending permutation n-1,...,7,6,5,4,3,2,1,0 is reached, while maintaining a record of the worst and best fit permutations and their corresponding fitness values.

5.2.1 ESTABLISHING EXPERIMENTAL CONTROLS

Like any other experiment, controls form an essential foundation in order to monitor progress and avoid follies. In the case of OGA, the algorithm produces results of unknown performance measures. For example, if permutations of ranging values from 800 to 900 are suggested as a good solution set after a few generation runs, how well does this measure against the optimal solution?

In order to answer this answer, it would be critical to know what the optimal solution is, as well as the worst case. Unfortunately, this is only possible for small chromosomes, particularly those less than 14. During the course of testing, supercomputers with multiple processors at speeds in excess of 300MHz were used to compute absolute boundaries. An execution with chromosome size 15 would fail to return within the 3 hours limit of the process life before it is terminated by the Kernel. Even if it did succeed in few hours, size 100 remains a computational challenge to actually attain its absolute boundaries.

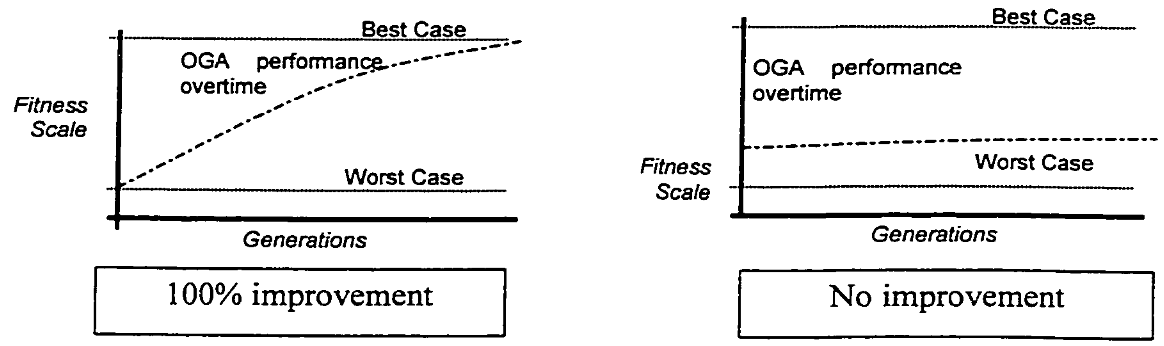


FIGURE 5.1: HYPOTHETICAL SCENARIOS DEPICTING INTERPRETATIONS OF PERFORMANCE IMPROVEMENTS

Problem Analysis / The brute force model:

Simple pseudocode : Linear search that cycles through all combinations and determine the best solution.

```
1.  Start
2.  Initialize at 012345...n-1
3.  Calculate fitness value
4.  Set max, min to fitness value in 3
5.  Begin Loop
    6.  GetNextPermutation
    7.  Calculate Fitness
    8.  Update max, min if any
    9.  Goto 6
10. Stop
```

Search Domain optimization:

Rationale: since the ring network structure wraps around, it does not matter at what point we start.

Hence, values such as 012345, 501234, 450123, 345012, 234501, and 123450 all have the same fitness value.

Therefore, it suffice to perform the search at the first level of permutation, ie. 0***** or for size n, we will have (n-1)! Instead.

Example: N=6, so the domain size is (N-1)!

Whatever maxima, or minima found in the form 0*****, other solution can be interpreted by ordered shuffeling as shown in the previous example.

This will assist in the development of the OGA as such: The domain search space would be restricted to the template 0***...n-1. The final domain size is reduced to :

Domain size = N!/N == (N-1)!

Improvement in domain/search space reduction by:

(N-1)! X 100 / N!

or

N!/Nx100x(1/N!)

= 100/N (This is the new percentage size of the search space. Eg. N=5, then 100/5 = 20% - only have to search 20% of the space.)

For large values. Eg. For N=200, 100/200=0.5 -> only have to search 0.5% of the original 200! (NOTE: 0.5% of 200! Remains a HUGE space! That's 199!)

5.2 RANDOM SEARCH

The randomness that is inheritly embedded in the Genetic Algorithm raises the question of its integrity: Is it just an other random search, and good solutions are mere chance encounters? Furthermore, since the linear search is not feasible for large solutions spaces, the random search can be tested as a possiblity to compare against the GA.

General Procedure:

1. Collect a random sample of size equal to the population size of the GA from the solution space.
2. Calculate the best, worst, average, and total fitness measures (Same as OGA)
3. Repeat and collect data for as many different cases as needed.

Another alternative is as follows:

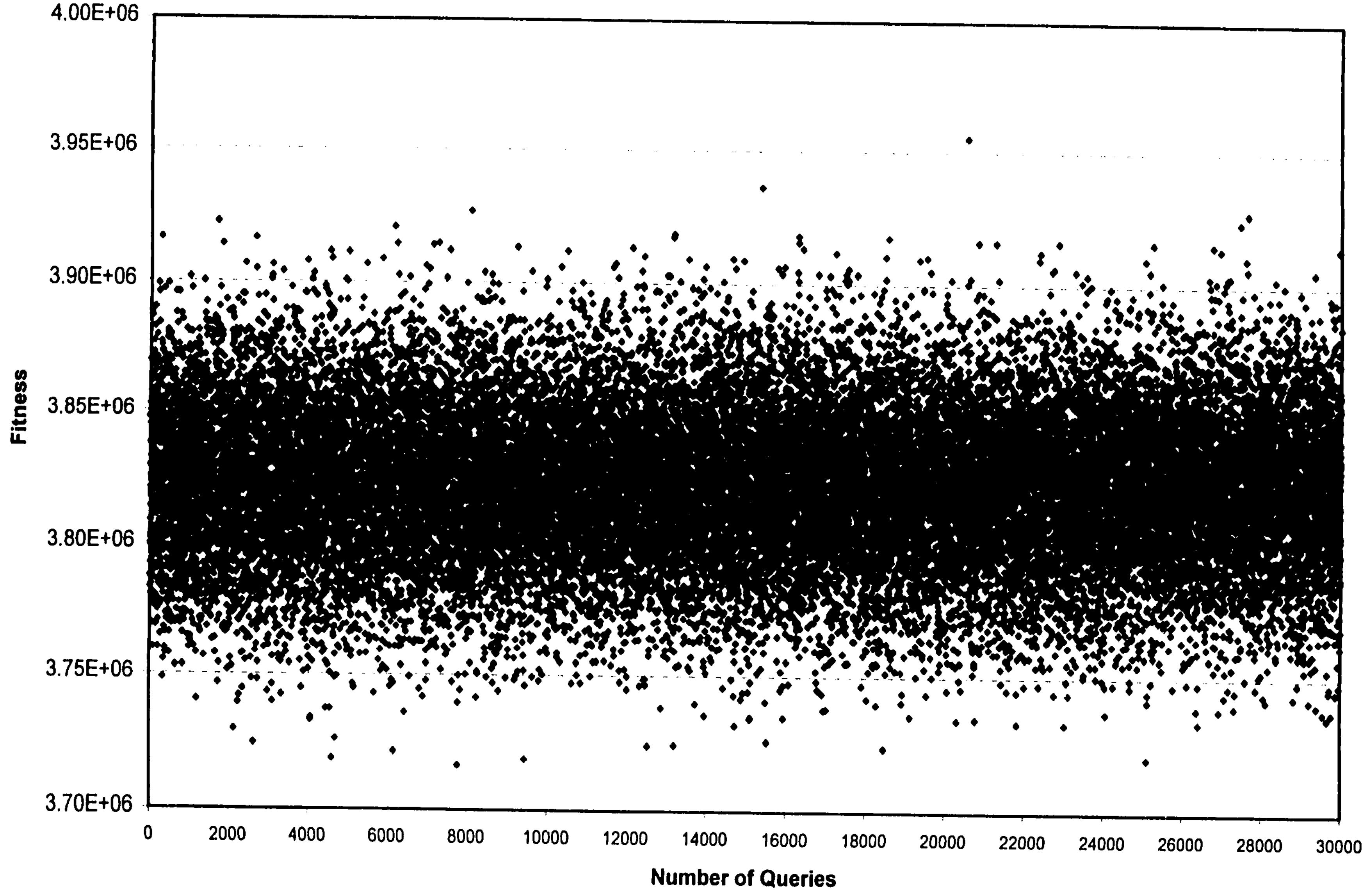
1. Collect a random sample of size equal to the population size.
2. Calculate the best, worst, average, and total fitness measures (Same as OGA)
3. Repeat steps one and two for n number of generations used by the GA.
4. Present the best set (based on best total fitness) for comparison with the OGA's last (or best) generation.

5.3 OBSERVATIONS

The linear search, when executed for specific communication matrix, and permutation size, ultimately yields the absolute minimum and maximum solutions. The results are used in the context of the GA that is been tested in order to monitor its improvement.

The Random search was especially tested for larger spaces, in particular for chromosome size 100, with the same matrix used in the OGA. Figure 5.2 shows the plot of all the generated solutions in the order they were generated. 30,000 points were generated for this test. Indeed, upon examination of the best, worst and average solutions, the random search underperforms compared to its OGA counterpart.

Random Search Results - 100 Nodes, raw data

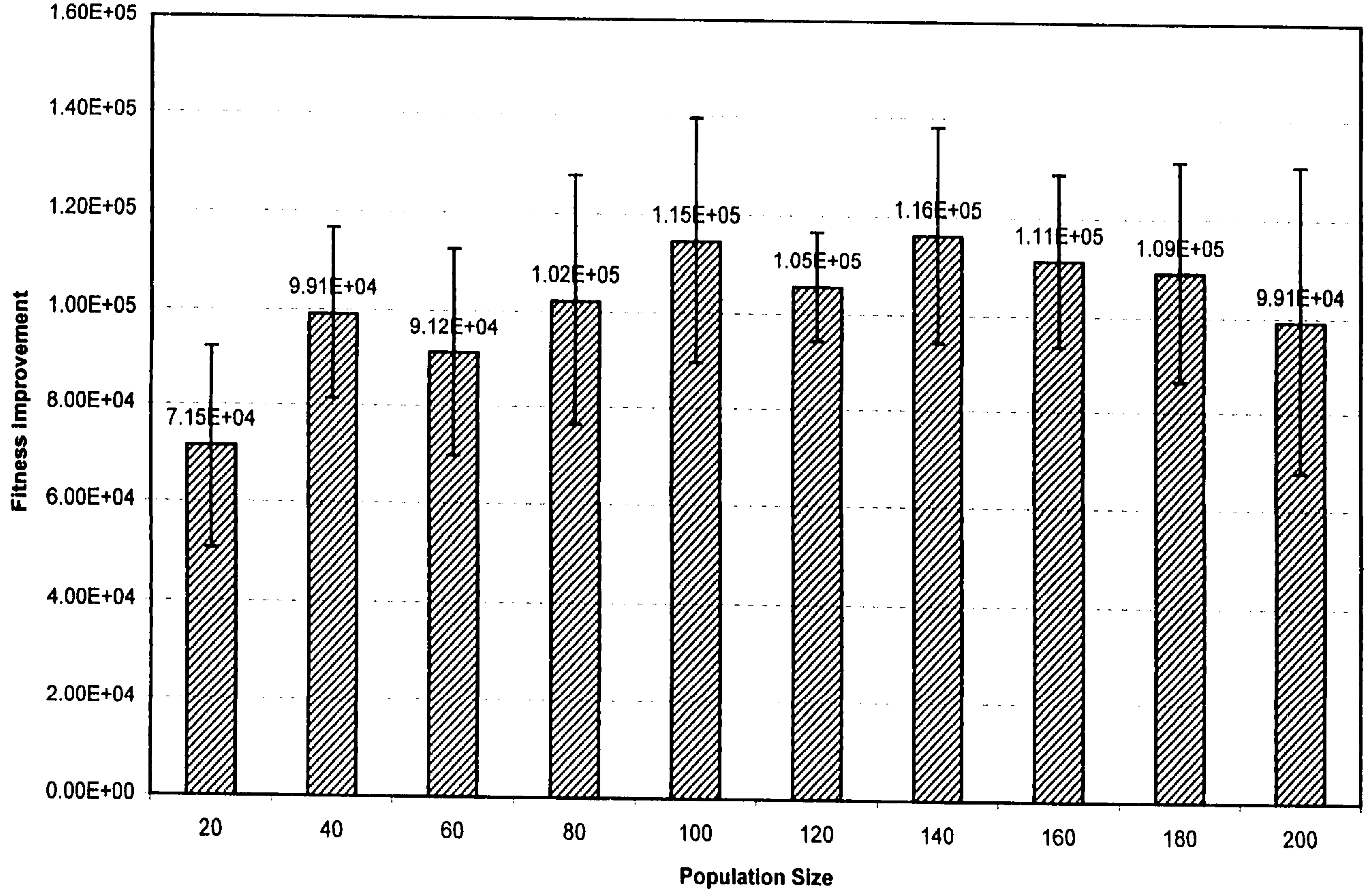


5.4 SENSITIVITY STUDY OF THE POPULATION SIZE

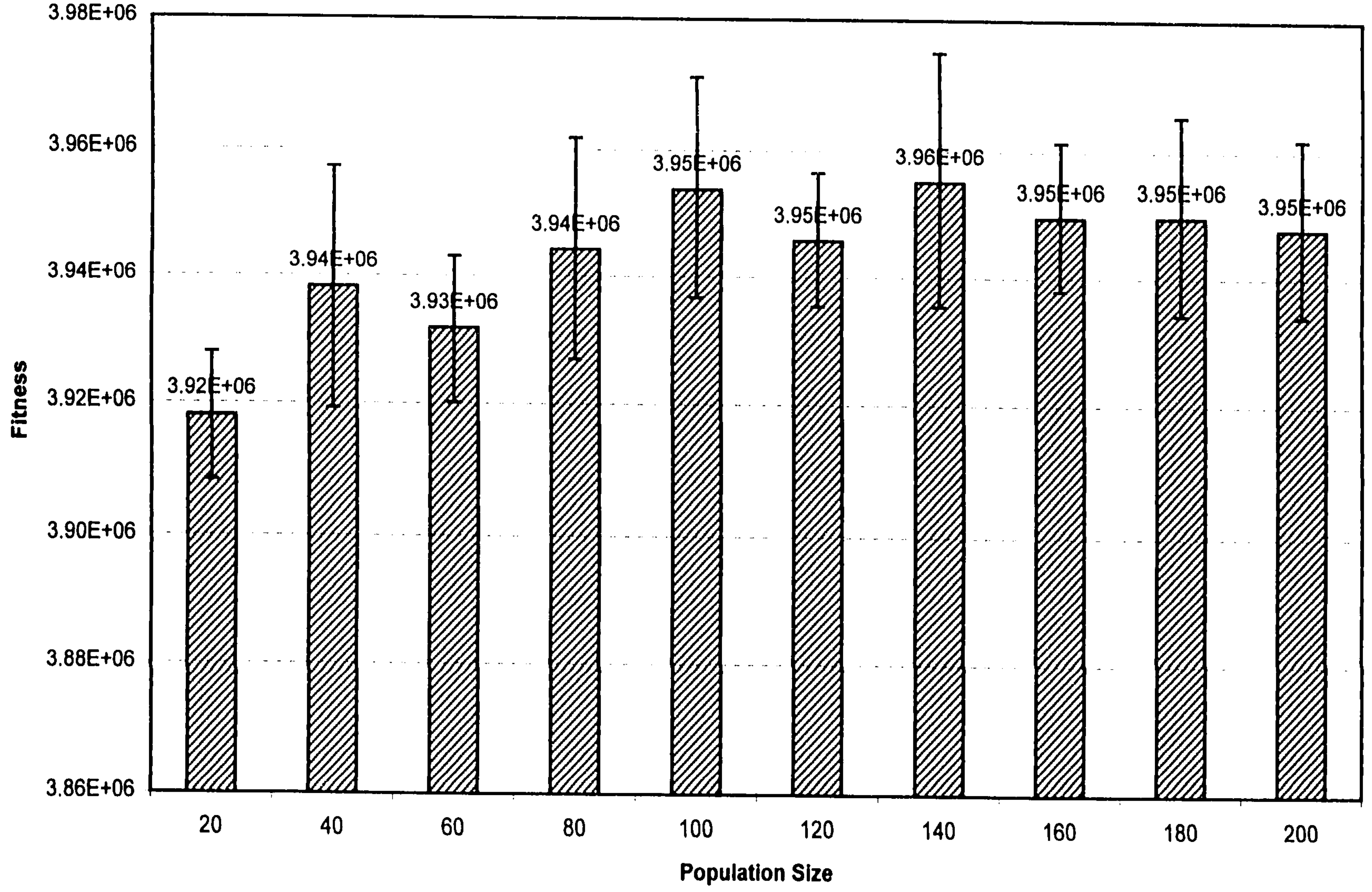
Optimization of the genetic algorithm is a vast and time consuming study that was decided to be put off for future studies. However, population size sensitization was performed in this research. The goal is to define the best population size that could be used for a fixed set of conditions. Ultimately, this will lead to the development of the best performing OGA based on its population size, all else remaining constant.

The study was conducted for population sizes ranging from 20, 40, 60, ..., 180, 200. All other variables from communication matrix, mutation rate, crossover criteria were kept constant. Each run was repeated 10 times and averaged with computed standard statistical deviations. The following plots reveal the results, showing that populations of size 140 were the best performers in terms of yielding the best average and maximum solutions, closely followed by size 100. It should be noted at this point that size 100 remains better than 140 in terms of the time it takes to generate the results.

Average Population Improvement over 100 Generations



Maximum Fitness After 100 Generations



5.5 COMPLEXITY ISSUES

Upon a close examination of the fitness function, it showed to have a quadratic complexity **O**(n^2). The fitness function forms the core computation of the algorithm. Its complexity can produce significant effects in the overall performance of the algorithm. In addition to the fitness evaluation function, the algorithm moves into a linear complexity resulting from the repetition involved to generate new individuals.

CHAPTER 6: SUMMARY AND CONCLUSIONS

6.1 CONTRIBUTIONS OF THE THESIS WORK AND COMPLETED OBJECTIVES

The genetic algorithm was originally chosen for the course of this work for the following purposes:

1. To investigate the proclaimed functionality and robust success of the genetic algorithm in the ring optical network design area,
2. To devise a suitable solution for the ring routed optical network design problem, used as an application model, and
3. To implement and test a genetic algorithm that tackles the previous two objectives.

With the final development of the Optical Genetic Algorithm (OGA) these set objectives became a reality.

Overall, the thesis work has established the founding principles behind the development of a custom Genetic Algorithm Search specific to the optical network design application at hand. Its merits are directly transparent to other researchers who investigate wavelength routed optical networks in further details. With the advent of this search method, the OGA, researchers now have a feasible technique to search a massive solution space for near optimal results.

6.2 FINDINGS DIRECTLY RELATED TO THE THESIS

As conventional heuristics that systematically search the solution space were proven to be of no feasible value when it comes to massive spaces, in the order of 100 factorial. The genetic algorithm is better suited to tackle such massive solution spaces.

With multiple executions of the optical genetic algorithm, it was noted that the algorithm consistently converges to progressively better solutions, and thereby supporting its reputation as a robust algorithm.

As the chromosome size increases, the number of generations required to produce high quality results increases. Increase in population size does not always guarantee better results, as shown by the sensitivity study. In the case of 200 individuals in a population as opposed to 100, and 140, the choice would fall on size 100. Simply, because in addition to its good performance relative to other sizes, it takes less computational time to reproduce generations as opposed to its larger size population counterparts.

6.3 OTHER FINDINGS

One setback in the implementation process was the relatively slow execution speed of the Java program. This is purely an inherent property of the Java machine and not the algorithm itself. If the portability of the algorithm is not an issue, future attempts to rewrite the algorithm into faster execution languages like C++ would be a worthy asset. This is especially true in real time applications where the implications of speed are of utmost importance. However, for the given application that the thesis work is tackling, the speed was not a bottleneck factor. At the time of writing this thesis, native Java compilers are becoming more abundant. Their promising advent to convert slow machine independent java classes to faster platform specific executables eliminates the need to rewrite the algorithm in other languages.

The choice of the Java language for the implementation in this case is more advantageous, despite the sacrifice of runtime speed. Most importantly, the generated class is portable by nature and hence executable on a multitude of platform. Furthermore, the purity of the object oriented design in Java has revealed a better structuring of the algorithm, rendering it more flexible and open for quick adjustments or changes. This is especially useful since future researchers who may choose this algorithm in their development have the choice of running this algorithm on their platform of choice with no alterations or conversions in this respect.

6.4 RECOMMENDATIONS

Based on the observation of the results, the genetic algorithm has shown improved overall performance over the brute force heuristic and the random search methods. Although the OGA would at times find the optimal solution in smaller genes, it however does not guarantee it every time.

It is the recommendation of this thesis to adopt the OGA as a candidate search algorithm in studies involving the ring routed wavelength network. Furthermore, with slight modifications, the algorithm can be readily adapted in other general search. The fitness function for instance would be easily substituted due to the model's object oriented approach in design.

6.5 FUTURE WORK

A number of ideas came across to expand this study and expand the research horizon. Unfortunately, due to time limitations imposed on the thesis work, these propositions are only listed for future research reference:

- Adapt the genetic algorithm to investigate other topologies (other than ring, such as linear and star).
- Perform a sensitivity study to optimize crossover and mutation probabilities.
- Develop further complexity and optimization measures for the algorithm so as to control and monitor its performance under different criteria.
- Develop comparative studies with other evolutionary and intelligent searches, including the Tabu search.

6.6 A VIEW TOWARD THE FUTURE

The optical genetic algorithm manifests itself as a working search solution for large spaces. Optimization in terms of crossover, mutation, and selection techniques remain one aspect dealing with the internals of the algorithm. On the outer, and more applicable view, more realistic problem scenarios can be adopted to put the algorithm to its ultimate test. Enhanced applications include scenarios that account for different fiber types, different bandwidth, as well as the survivability of the network when an unexpected break in the wire or switch occurs. The algorithm may redesigned in the future to account for single node failure. The GA now has to assist in designing a network with the best performance not only under static conditions, but also showing good performance upon random single or multiple point failure. The robustness of the genetic algorithm along with its flexibility opens the door to an endless number of new areas of research applications.

REFERENCES

[AIZA93] Aizawa, Akiko N. and Benjamin W.Wah. “Dynamic control of genetic algorithms in a noisy environment.” *ICGA93*, pp. 48 – 55.

[ALT94] Lee Altenberg. “Emergent Phenomena in Genetic Programming,” *Proceedings of the Third Annual Conference on Evolutionary Programming*, 24-26 February 1994, pp. 233-241.

[ANG94] Peter J. Angeline. “Genetic Programming: A Current Snapshot,” *Proceedings of the Third Annual Conference on Evolutionary Programming*, 24-26 February 1994, pp. 224-232.

[BACK93] Bäck Thomas. “Optimal Mutation Rates in Genetic Search.” *ICGA93*, pp.2–8.

[BACK96] Bäck Thomas. *Evolutionary Algorithms in Theory and Practice*. Oxford University press: New York, 1996.

[BAU94] Richard J. Bauer Jr. *Genetic Algorithms and Investment Strategies*. Johen Wiley & Sons Incorporated. Toronto: 1994.

[BLAN93] Joe L. Blanton and Roger L. Wainwright. “Multiple vehicle routing with time and capacity constraints using genetic algorithms.” *ICGA93*, pp. 452-457.

[BRUN93] Ralf Bruns. “Direct chromosome representation and advanced genetic operators for production scheduling.” *ICGA93*, pp.352-359.

[CARD89] Richard H.Cardwell, Clyde L.Monma and Tsong-Ho Wu. “Computer-Aided Design Procedures for Survivable Fiber Optic Networks.” *IEEE Journal on Selected Areas in Communications*, Vol.7, No.8, October 1989, pp. 1188-1197.

[CHAM95] Lance Chambers, editor. *Practical Handbook of Genetic Algorithms, Applications Volume 1*. New York: CRC Press, 1995.

[DAVI91] Lawrence Davis, editor. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991.

[DAVI93] Davis Lawrance, et. al. *A Genetic Algorithm for Survivable Network Design*. ICGA93, pp. 408 - 415.

[EAST93] Fred F.Easton and Nashat Mansour. “A distributed genetic algorithm for employee staffing and scheduling problems.” *ICGA93*, pp.360-367.

[ESHE93] Eshelman J. Larry and J. David Schaffer. "Crossover's Niche." *ICGA93*, pp. 9 – 14.

[FONS93] Carlos M.Fonseca and Peter J.Fleming. "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization." *ICGA93*, pp. 416-423.

[GABB91] Paula S.Gabbert, *et.al.* "A System for Learning Routes and Schedules with Genetic Algorithms." *ICGA91*, pp. 430-436.

[GAVI89] Bezalel Gavish, Pierre Trudeau, Moshe Dror, Michel Gendreau, and Lorne Mason. "Fiber Circuit Network Design Under Reliability Constraints." *IEEE Journal on Selected Areas in Communications*, Vol.7, No.8, October 1989, pp.1181-1187.

[GOL89] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Incorporated: Don Mills, Ontario, 1989

[GOLD91] Goldberg, D. E., Deb, K. & Korb, B. (1991). Don't worry, be messy. *Proceedings of the Fourth International Conference in Genetic Algorithms and their Applications,* 24-30.

[GOLD93] Goldberg, David E., Kalyanmoy Dev, Hillol Kargupta and Georges Harik. "Rapid, accurate optimization of difficult problems using fast messy genetic algorithms." *ICGA93*, pp. 56 – 64.

[GOO95] Suran Goonatilake and Philip Treleaven, editors. *Intelligent Systems for Finance and Business*. John Wiley & Sons. Toronto: 1995.

[GOOS95] Goos Gerhard, Jursi Hartmanis, and Jan van Leeuwen, editors. *Evolutionary Computing*. Springer-Verlag. New York: 1995.

[GRE93] John J. Grefenstette, "Genetic Algorithms," *IEEE Expert*, October 1993, pp. 5-8.

[GRUA93] Gruau, Frédéric. "Genetic Synthesis of Modular Neural Networks." *ICGA93*, pp. 318 – 325.

[HAM95] K.Hamada, T.Baba, K.Sato, and M.Yufu. "Hybridizing a Genetic Algorithm with Rule-Based Reasoning for Production Planning," *IEEE Expert,* October 1995, pp. 60-67.

[HARV93] Harvey Inman. "The Puzzle of the Persistent Question Marks: A Case Study of Genetic Drift." *ICGA93*, pp. 15 – 21.

[HIB93] Hibbert, D. Brynn. "Display of Chemical Structures in Two Dimensions and the Evolution of Molecular Recognition." Panel, *ICGA93*, p.637.

[HOL92] Holland, H. John. "Genetic Algorithms," *Scientific American*, July 1992, pp. 66 – 72.

[HOW95] Les M. Howard and Donna J. D'Angelo. "The GA-P: A Genetic Algorithm and Genetic Programming Hybrid," *IEEE Expert*, June 1995, pp. 11-15.

[JON93] Jones, Gareth, Robert D.Brown, David E.Clarck, Peter Willett and Robert C.Glen. "Searching databases of two-dimensional and three-dimensional chemical structures using genetic algorithms." *ICGA93*, pp. 597 – 602.

[KARG91] Hillol Kargupta and R.E.Smith. "System Identification with Evolving Polynomial Networks." *ICGA91*, pp. 370-376.

[KIDW93] Michelle D. Kidwell. "Using genetic algorithms to schedule distributed tasks on a bus-based system." *ICGA93*, pp.368-374.

[KOSA91] Corey Kosak, Joe Marks and Stuart Shieber. "A Parallel Genetic Algorithm for Network-Diagram Layout." *ICGA91*, pp. 458-465.

[KUO93] Kuo, Ting and Shu-Yuen Hwang. "A Genetic Algorithm with Disruptive Selection." *ICGA93*, pp. 56 – 64.

[LEU95] Man Leung Wong and Kwong Sak Leung. "Inducing Logic Programs with Genetic Algorithms: The Genetic Logic Programming System," *IEEE Expert*, October 1995, pp. 68-76.

[MIC92] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. 3rd edition. Springer: New York: 1996.

[MIT96] Melanie Mitchell, *An Introduction to Genetic Algorithms*, MIT Press: Cambridge 1996.

[NAGA96] Tomoharu Nagao. "Homogeneous Coding for Genetic Algorithm Based Parameter Optimization." *Proceedings of 1996 IEEE International Conference on Evolutionary Computation*, 1996, pp. 426-430.

[PAL96] Sankar K. Pal and Paul P. Wang, editors. *Genetic Algorithms for Pattern Recognition*. CRC Press Incorporated: New York: 1996.

[POWE93] Powell, David and Michael M.Skolnick. "Using Genetic Algorithms in Engineering design optimizzation with non-linear constraints." *ICGA93*, p. 424 – 431.

[PUN95] W.F.Punch, R.C.Averill, E.D.Goodman, Shyh-Chang Lin, and Y.Ding. "Using Genetic Algorithms to Design Laminated Composite Structures," *IEEE Expert*, February 1995, pp.42-49.

[RAMS93] Ramsey, Connie Loggia and John J. Grefenstette. "Case-Based Initialization of Genetic Algorithms." *ICGA93*, pp. 84 – 91.

[SANN96] Nobuo Sannomiya and Hitoshi Iima. "Application of Genetic Algorithms to Scheduling Problems in Manufacturing Processes." *Proceedings of 1996 IEEE International Conference on Evolutionary Computation*, 1996, pp.523-528.

[SCH93] Alan C.Schultz, John J. Grefenstette, and Kenneth A.De Jong, "Test and Evaluation by Genetic Algorithms," *IEEE Expert*, October 1993, pp.9-14.

[SCHA89] Schaffer J. David, Richard A.Caruana, Larry J.Eshelman and Rajarshi Das. "A study of control parameters affecting online performance of genetic algorithms for function optimization." *ICGA89*, pp. 51-60.

[SCHO93] Schoenauer, Marc and Spyros Xanthakis. "Constrained GA optimization." *ICGA93*, pp. 573 – 579.

[SHIN93] Shing, Man-Tak and Gary B.Parker. "Genetic algorithms for the development of real-time multi-heuristic search strategies." *ICGA93*, pp. 565 – 572.

[SHON93] Shonkwiler R. "Parallel Genetic Algorithms." *ICGA93*, pp. 199 – 205.

[TAC95] Walter Alden Tackett. "Mining the Genetic Program," *IEEE Expert*, June 1995, pp. 28-38.

[TAM96] S.M.Tam and K.C.Cheung. "Design Optimization of An Internal Sensor by Genetic Algorithms." *Proceedings of 1996 IEEE International Conference on Evolutionary Computation*, 1996, pp.500-505.

[THAN93] Sam R.Thangiah, Rajini Vinayagamoorthy and Ananda V.Gubbi. "Vehicle routing with time deadlines using genetic and local algorithms." *ICGA93*, pp. 506-513.

[THIE93] Thierens, Dirk and David E. Goldberg. "Mixing in Genetic Algorithms," *ICGA93*, pp. 38 – 45.

[TORR91] Jan Torreele. "Temporal Processing with Recurrent Networks: An Evolutionary Approach." *Proceedins of the Fourth International Conference on Genetic Algorithms*, University of California, 1991, pp. 555-560.

[TUR84] Raymond Turner. *Logics for Artificial Intelligence*. Ellis Horwood Limited: Toronto, 1984.

[UCK93] Serdar Uckun, Sugato Bagchi, and Kazuhiko Kawamura, "Managing Genetic Search in Job Shop Scheduling," *IEEE Expert*, October 1993, pp.15-24.

[UNGE93] Unger, Ron and John Moult. "A Genetic Algorithm for 3D Protein Folding Simulations." *ICGA93*, pp. 581 – 589.

[VIGN91] G.A.Vignaux and Z.Michalewicz. "A Genetic Algorithm for the Linear Transportation Problem." *IEEE Transactions on Systems, Man, and Cybernetics*, Vol.21, No.2, March/April 1991, pp. 445-452.

[WHI93] L. Darrell Whitley, editor. *Foundations of Genetic Algorithms 2*. Morgan Kaufmann Publishers, Incorporated: San Mateo, California: 1993.

[WIEN93] Wienholt, Willfried. "A refined genetic algorithm for parameter optimization problems." *ICGA93*, pp. 589 – 596.

[WINT95] G.Winter, J.Périaux, M.Galán, and P.Cuesta. *Genetic Algorithms in Engineering and Computer Science*. Toronto: John Wiley & Sons, 1995.

APPENDIX A: GLOSSARY OF GA RELATED TERMS AND ABBREVIATIONS

The following table summarizes the key terms used in the field of Genetic Algorithms that the reader need to be familiar with.

TERM:	MEANING:
EP	Evolutionary Programming
GA	Genetic Algorithm
GP	Genetic Programming
OGA	Optical Genetic Algorithm, an improved version of the SOGA.
SGA	Simple Genetic Algorithm, described by David E. Goldberg (89)
SOGA	Simple Optical Genetic Algorithm, a modification of the SGA to attempt a simple solution for the Optical wavelength design network described in the thesis
Chromosome	or gene, is a linear array of alleles. Contains the coded value of a single possible solution. It is made up of a number of alleles.
Gene	see chromosome
Phenotype	The physical characteristics of an individual. eg. the actual value.
Genotype	The genetic makeup of an individual.
Allele	A single element that, when put together, form a chromosome.
Locus	The location of an allele within a chromosome strand.
Individual	A higher representation of the solution. The actual solution value.
Fitness	A measure of an individual's performance within a population based on a fitness evaluation function. Based on which, an individual is either to be selected for mating or eliminated from the following generation.
Fitness function	A mathematical function used to calculate the fitness value of an individual.
Population	A set of individual at any given generation time. A population form the solution space at any given time.

Generation	The timeframe keeper of a population. Initial generation starts with a random population, and overtime, following generations hold populations that contain progressively higher performance solutions (or higher fit individuals).
Crossover	A genetic operator where two parent chromosome strands exchange genetic information to produce two offsprings to be added in the following generation.
Mutation	A genetic operator used to ensure diversity in genes and allow the solution model to explore new values. It has many possible forms of implementation. It involves the alteration of one or more allele in a chromosome.
Inversion	The toggling effect where typically an allele with binary value 0 is reversed to 1, and vice versa.
Selection	Upon regeneration, higher fit individuals within a population are selected for mating. Their genes, in whole or in part, are carried on to the following generation.
Adaptation	Based on Darwin's survival of the fittest approach to evolution, higher individuals form better solutions and alter themselves to changing parameters to adapt and survive.
Extinction	Low fitness individuals that do not get selected for mating are disowned and discontinued in future generations.

APPENDIX B: SGA SOURCE CODE

The following listing is the complete source code for the Simple Genetic Algorithm (SGA) written in Java and developed using OOP modeling.

This is a Java version adapted from Goldberg's original implementation of the SGA that used Pascal and modular coding.

see \results_sga, sga.txt

and out3.xls

FILE: SGA.java

```
/** ***********************************************************************
    SIMPLE GENETIC ALGORITHM (SGA)
    Written by Ziad Kobti

    Demonstrates the simple genetic algorithm as modeled
    using OOP technique in Java.
    Uses fitness function f(x) = x^2 for preliminary testing

    Original concept is based on the SGA model
    developed by David E. Goldberg (1989) in Pascal

    (c) 1997-99 All Rights Reserved
*/

import java.lang.Math;

// Main Object class: sga
class sga
{
        public int number_of_mutations;

        boolean Flip(double p)  //0<p<1
        {
            double j;
            j=Math.random();
            if (j<=p)
              return true;
            else
              return false;
        }

        Individual CrossOver(Individual I1, Individual I2, int x,
                             int lchrom, double pmutation)
        {
            int j;
            Individual Child= new Individual(lchrom);
            for (j=0; j<=x; j++)
            {
             Child.gene.A_allele[j].b_bit = I1.gene.A_allele[j].b_bit;
             if (Flip(pmutation))
              {
                number_of_mutations++;
                Child.gene.Mutate(j);
              }
            }

            for (j=x+1; j<lchrom; j++)
            {
             Child.gene.A_allele[j].b_bit = I2.gene.A_allele[j].b_bit;
             if (Flip(pmutation))
              {
                number_of_mutations++;
                Child.gene.Mutate(j);
              }
            }
            return Child;
        }

      // Main routine
      public static void main(String args[])
      {
            int maxpop = 100;
            int maxstring = 30;

            int popsize, lchrom, gen, maxgen;
            double pcross, pmutation, sumfitness;
            int nmutation, ncross;
            double avg, max, min;

            sga SGA=new sga();

            Population oldpop, newpop;

            System.out.println("Simple Genetic Algorithm - (c)1997-1999
by Ziad Kobti - All Rights Reserved");

System.out.println("==================================================
======================\n");

            gen = 0;
            // initdata
            popsize   = 50;        // Population Size
            lchrom    = 8;         // Length of Chromosome
            maxgen    = 20;         // Number of Re-Generations
            pcross    = 0.1;       // Probability of Crossing Over
            pmutation = 0.01;      // Probability of Mutation
            nmutation = 0;         // Actual Number of Mutations
            ncross    = 0;         // Actual Number of Crossing Over

             // initpop
            int mate1,mate2;
            int jcross;

            oldpop = new Population(popsize, lchrom);
            oldpop.InitReport();

            newpop = new Population(popsize, lchrom);

            do{

               for (int i=0; i<popsize; i=i+2)
               {
                mate1 = oldpop.Select();
                mate2 = oldpop.Select();

                // Chrossover routine debug line
                // System.out.println("i is "+i+" ;mate1 is "+mate1+"
                //;mate2 is "+mate2);

                if (SGA.Flip(pcross))
                {
                   jcross = (int) (Math.random() * (lchrom - 1));
                   ncross++;
                }
                else
                {
                   jcross = lchrom-1;
                }

                newpop.Indiv[i]   = SGA.CrossOver(oldpop.Indiv[mate1],
oldpop.Indiv[mate2], jcross, lchrom, pmutation);
                newpop.Indiv[i+1] = SGA.CrossOver(oldpop.Indiv[mate2],
oldpop.Indiv[mate1], jcross, lchrom, pmutation);

                nmutation = SGA.number_of_mutations;

                newpop.Indiv[i].parent1 = mate1;
                newpop.Indiv[i].parent2 = mate2;
                newpop.Indiv[i].xsite = jcross;
                newpop.Indiv[i+1].parent1 = mate1;
                newpop.Indiv[i+1].parent2 = mate2;
                newpop.Indiv[i+1].xsite = jcross;

          }

          newpop.Report(gen+1);
```

```
            oldpop=newpop;
            gen++;
          } while (gen<maxgen);

          System.out.println("\nTotal Number of Cross-Overs: " + ncross);
          System.out.println("Total Number of Mutations  : " +
                                  nmutation);
   }
}
```

FILE: Population.java

```
/**
   Class:
          Population

  Proterties:
          int size : stores the number of individuals
          double fitness: average fitness of the population
          double sum? : f(x)

  Methods:
          Regenerate : create next population
 */

import java.lang.Math;

class Population
{
  public Individual[] Indiv; // = new Individual[10];
  public int size;
  public double max, min, avg, sumfitness;

  public Population()
  {
        // Constructor: Initialize Population with 10 Individuals

        size = 10;
          sumfitness = 0;
        Indiv = new Individual[size];
        for (int i = 0; i < size; i++)
        {
              Indiv[i] = new Individual(5);
                  sumfitness += Indiv[i].Fitness();
                  //System.out.println(Indiv[i].gene + " " +
Indiv[i].gene.Value() + " " + Indiv[i].Fitness());
          }

        for (int j = 0; j < size; j++)
        {
                  Indiv[j].pselect = Indiv[j].Fitness() / sumfitness;
        }

    }
```

```
public Population(int s, int l)
{
      // Constructor: Initialize size
      double j;
      size = s;
      Indiv = new Individual[size];

      Indiv[0] = new Individual(l);
      sumfitness = Indiv[0].Fitness();
      min = sumfitness;
      max = sumfitness;

      for (int i = 1; i < size; i++)
      {
            Indiv[i] = new Individual(l);
            j = Indiv[i].Fitness();
            sumfitness += j;
            if (j>max) max = j;
            if (j<min) min = j;
      }
      avg = sumfitness / size;
}

public Population(Population PopSource)
      // This is a copy constructor:
      // you can create a new pop from an existing one
{
    size = PopSource.size;
    // Indiv = new Individual(PopSource.Indiv);
}

public int Select()
{
      double rand, partsum;
      int j;

      partsum = 0.0;
      j = 0;
      rand = Math.random() * sumfitness;
      //   System.out.println("rand is : "+ rand);
      do
      {
          partsum+=Indiv[j].Fitness();
          j++;
      } while ((partsum < rand) && (j != size));

      return j-1;
}


public void InitReport()
{
        // Prints the population contents of the first generation
        System.out.println("\nGeneration 0:");
        System.out.println("\n---------------------------------------
-----");
```

```
          System.out.println("\nNo String     (x)    f(x)    Parent1
Parent2 XSite");

          for (int i = 0; i < size; i++)
          {
              System.out.println(i+" "+Indiv[i].gene + "    " +
Indiv[i].gene.Value() + "   " + Indiv[i].Fitness() + "
"+Indiv[i].parent1 +" " +Indiv[i].parent2+" "+Indiv[i].xsite);
          }
          System.out.println("---------------------------------------
---");
          System.out.println("Max: "+max+" Min: "+min+" Avg: "+avg+"
Total Fitness: "+sumfitness);
  }

  public void Report(int g)  // g is the number of the generation
  {
          // Prints the population contents of the first generation
          System.out.println("\nGeneration:"+g);
          System.out.println("\n-------------------------------------
-----");
          System.out.println("\nNo String     (x)    f(x)    Parent1
Parent2 XSite");

        double j;

        sumfitness = Indiv[0].Fitness();
        min = sumfitness;
        max = sumfitness;

        for (int i = 1; i < size; i++)
        {
              j = Indiv[i].Fitness();
              sumfitness += j;
              if (j>max) max = j;
              if (j<min) min = j;
        }
        avg = sumfitness / size;

          for (int i = 0; i < size; i++)
          {
              System.out.println(i+" "+Indiv[i].gene + "    " +
Indiv[i].gene.Value() + "   " + Indiv[i].Fitness() + "
"+Indiv[i].parent1 +" " +Indiv[i].parent2+" "+Indiv[i].xsite);
          }
          System.out.println("---------------------------------------
---");
          System.out.println("Max: "+max+" Min: "+min+" Avg: "+avg+"
Total Fitness: "+sumfitness);
  }

}
```

FILE: Individual.java

```
/*
 *
 * Class:
 *          Individual
 *
 * Proterties:
 *              Chromosome gene : Genetic make-up/alleles
 *              double value : decoded value of gene; x
 *              double fitness : f(x)
 *
 * Methods:
 *          Individual() : constructor, creates default gene
 *              Individual(int l) : overloaded constructor with l alleles
 *              in constructor, calculate value and fitness
 */
class Individual
{
      public Chromosome gene;
      public double pselect; // probability of selection f/Sum(f)
    protected int parent1, parent2, xsite;

      public Individual()
      {
            gene = new Chromosome(4);
        parent1 = -1;
        parent2 = -1;
      }

      public Individual(int l) // this one
      {
        gene = new Chromosome(l);
        parent1 = -1;
        parent2 = -1;
        xsite = -1;
      }

    public Individual(int l, int p1, int p2)
      {
            gene = new Chromosome(l);
        parent1 = p1;
        parent2 = p2;
    }

    public double Fitness()
      {
            // Calculate the fitness of the individual
            // given x, find f(x)

            // f(x) := x^2
            return Math.pow(gene.Value(), 2);
      }

      boolean Flip(double p)
      {
            // Biased coin, True p of the time
            // 0 <= p <= 1
            if (p == 0)
                  return false;
            else
                return (Math.random() <= p);
      }

}
```

FILE: Chromosome.java

```
/*
 *
 * Class:
 *          Chromosome
 *
 * Proterties:
 *        int length : number of alleles
 *        boolean Alleles[length] : binary string to make up chromosome
 *
 * Methods:
 *        Chromosome()   : constructor, generates 10 alleles randomly
 *        Chromosome(int a) : overloaded constructor, generates 'a'
alleles randomly
 *        Mutate()   : executes a mutation
 *        Get(int pos) : get allele value at position pos from left,
start at 0
 *        Put(int pos) : overwrites allele value at position pos from
left, start at 0
 *        int Length() : returns length, number of alleles/or size of
chromosome
 *
 */

import java.lang.Math;

public class Chromosome
{
    public int i_length;
    //protected int locus;
    public Allele[] A_allele;

    Chromosome()
      {
            i_length = 10;
                GenerateRandomChromosome();
      }

      Chromosome(int l)
      {
            i_length = l;
                GenerateRandomChromosome();
      }

      public void GenerateRandomChromosome()
      {
            int which;

                A_allele = new Allele[i_length];

            for (int c = 0; c < i_length; c++)
            {
                    A_allele[c] = new Allele();

                    which = 1 + (int) (Math.random() * 2);
                    if (which == 1)
                          A_allele[c].b_bit = true;
                    else
                          A_allele[c].b_bit = false;
              }
        }

     public void Mutate(int i_locus)
        {
              // Mutate one or more allele in the chromosome string
              // implements single point mutation at random position

              // missing mutation rate, this is placed at the calling
function level
                  //int i_locus;
                  //i_locus = 0 + (int) (Math.random() * (i_length - 1));
              A_allele[i_locus].Toggle();
        }


     public double Value()
        {
              // Calculate the raw/decoded value of the binary string


              // DECODE RULE:
              // Given Binary string of Length "0 -> (length-1)"
              // where 2^0 is the right most at locus (length-1)
              //   and 2^length-1 is left most at locus 0

                  int    j, i = 0;
                  double accum = 0;

              for (j = i_length-1; j >= 0; j--)
              {
                      if (A_allele[j].b_bit)
                         i = 1;
                      else
                         i = 0;
                         accum = accum + (Math.pow(2, (i_length - 1 - j))
* i);
              }
              return accum;
        }


      int Length()
      {
              return i_length;
      }

      int length()
      {
                  return i_length;
        }

        public String toString()
        {
                String s_contents = "";
                    for (int i=0; i<i_length; i++)
                {
                        if (A_allele[i].b_bit)
                            s_contents += "1";
                        else
                            s_contents += "0";
                    }

                    return s_contents;
        }

}
```

FILE: Allele.java

```
/**
   (c)1999 by Ziad Kobti. All Rights Reserved

   Class: Allele                 -- Implements a single bit allele

   Variables:
     boolean b_bit               -- Bit value

   Methods:
     boolean Allele()            -- Constructor: Initializes bit to 0
     boolean Allele(boolean)  -- Overloaded Constructor:
                                    Initializes to the given bit value
     void Toggle()               -- Toggles bit value
*/


public class Allele
{
    public boolean b_bit; // bit value

    // Constructor
    public Allele()
      {
               b_bit = false;
      }

    // Overloaded Constructor with parameter
    public Allele(boolean b)
      {
            b_bit = b;
      }

    // Toggles bit value
    public void Toggle()
      {
            b_bit = !b_bit;
      }

}
```

APPENDIX C: SOGA SOURCE CODE LISTING

Source code listing for the Simple Optical Genetic Algorithm (SOGA). Developed by Ziad Kobti. Based on the SGA and modified to solve the Optical Network design problem at hand.

FILE: SOGA.java

```
import java.lang.Math;




/**

   Final revisions:

1. need to add a column for corrected fitness where:
   take maxfitness of population(+1?),
   and recalculate the new fitness values based on max-fitness.
   use the new fitness value in the selection process
2. for cross over, you can select randomly 2 mates, but not for
   the cross-over point. Do a search for an appropriate x-over
   point if any, otherwise none. (by swapping complete strings,
   and do not increment counter)
  (use qsort on chromosome substrings to check if compatible
permutations)
   No! use bitmapping instead to check compatibility
*/

class soga
{
      // Keep track of actual number of mutations
       public int number_of_mutations;

      // Do a biased coin flip based on a given probability
       boolean Flip(double p)  //0<p<1
        {
          double j;
          j=Math.random();
          if (j<=p)
           return true;
          else
           return false;
        }

       // Calculate factorial, used in the fitness function
       public long factorial(long n)
        {
         // do some more base cases to speed things up
         // if(n=2) return 2;
         // if (n=3) return 6;
         // if (n=4) return 24;
         // if (n=5) return 95; etc..
         if (n <= 1) //base case
           return 1;
         else
           return n * factorial(n-1);
        }

       // Implements correction factor for the probability of
       // crossing over- assures no lethality while maintaining
       // accurate probability for cross-over
       public double CorrectPCross(double p, int l)
        {
           double a=0;
           double b=0;
           long i,j;

           // correction factor implementation
           for (i=1; i<l; i++)
            {
             j = l-i;
             a = a+ (factorial(i)*factorial(j));
            }
           b = a / factorial(l);
           b = p / b;
           if (b > 1) {b=1;}
           return b;
        }

       // Performs actual cross over routine. If Cross over is lethal,
then
       // it returns the same pair by using 0 position for cross over
       Individual CrossOver(Individual I1, Individual I2, int x, int
lchrom, double pmutation)
        {
           int j;
           Individual Child= new Individual(lchrom);
           for (j=0; j<=x; j++)
           {
            Child.gene.A_allele[j].i_bit = I1.gene.A_allele[j].i_bit;
            if (Flip(pmutation))
             {
               number_of_mutations++;
               Child.gene.Mutate(j,j);
             }
           }
```

```
            for (j=x+1; j<lchrom; j++)
            {
             Child.gene.A_allele[j].i_bit = I2.gene.A_allele[j].i_bit;
             if (Flip(pmutation))
              {
                number_of_mutations++;
                Child.gene.Mutate(j,j);
              }
            }
            return Child;
        }

  // Checks if two strings are compatible for a living offspring
  // IsCompatible(oldpop.Indiv[mate1], oldpop.Indiv[mate2], k, lchrom)
  public boolean IsCompatible(Individual I1, Individual I2, int x, int
l)
    {
      int i,j;
      boolean c =false;
      boolean A[] = new boolean [l];

      for (i=0; i<l; i++) { A[i] = false; }
      // System.out.println("check the chromosomes" + I1.gene +", " +
I2.gene);
      for (i=0; i<=x; i++)
       {
         // System.out.println ("Comparing " +
I1.gene.A_allele[i].i_bit + ", and "
         //  + I2.gene.A_allele[i].i_bit);
         A[I1.gene.A_allele[i].i_bit]  = !A[I1.gene.A_allele[i].i_bit];
         A[I2.gene.A_allele[i].i_bit]  = !A[I2.gene.A_allele[i].i_bit];
       }
         c=false;
         for (j=0; j<l; j++) { if (A[j] == true) {c=true;} }

         if (c == false)
          {
            // System.out.println("Found compatible between " + I1.gene
+
            //         " and " + I2.gene + " at position " + x);
            return true;
          }
         return false;
    }

    // Main routine
     public static void main(String args[])
      {
            int maxpop = 100;
            int maxstring = 30;

            int popsize, lchrom, gen, maxgen;
            double pcross, pmutation, sumfitness;
            double pcrossNew;
            int nmutation, ncross;
            double avg, max, min;
            soga SOGA=new soga();

            Population oldpop, newpop;

            System.out.println("Simple Genetic Algorithm - (c)1997-1998
by Ziad Kobti - All Rights Reserved");

System.out.println("==================================================
======================\n");

            gen = 0;

            // initdata
            popsize   = 20;        // Population Size
            lchrom    = 6;         // Length of Chromosome
            maxgen    = 10;         // Number of Re-Generations
            pcross    = .4;        // Probability of Crossing Over
            pmutation = .1;       // Probability of Mutation
            nmutation = 0;         // Actual Number of Mutations 0
            ncross    = 0;         // Actual Number of Crossing Over 0

            //do correction for probability of cross-over
            pcross = SOGA.CorrectPCross(pcross, lchrom);

            // pcross=pcrossNew;
            System.out.println("pcross = " + pcross);

             // initpop
            int mate1,mate2;
            int jcross=0;

            int k, i;


            oldpop = new Population(popsize, lchrom);
            oldpop.InitReport();

            newpop = new Population(popsize, lchrom);

            do{

               for (i=0; i<popsize; i=i+2)
               {
                mate1 = oldpop.Select();
                mate2 = oldpop.Select();

                // Chrossover routine
                // System.out.println("i is "+i+" ;mate1 is "+mate1+"
;mate2 is "+mate2);

                // Iwashere
                if (SOGA.Flip(pcross))
                {
                 jcross=lchrom-1;
                 //  jcross = (int) (Math.random() * (lchrom - 1));
                 // Do not find crossing point (jcross) randomly. do a
linear search/compatibility check routine
```

```
                    boolean foundcompatible=false;
                    for (k=lchrom-2; (k>0) || (foundcompatible ==
false); k--)
                     {
                       if (SOGA.IsCompatible(oldpop.Indiv[mate1],
oldpop.Indiv[mate2], k, lchrom))
                         {
                           foundcompatible=true;
                           jcross = k;
                           ncross++;
                         }
                       else
                         { jcross = lchrom-1; }

                     }

                }
               // else
               // {
               //    jcross = lchrom-1;
               // }

                newpop.Indiv[i]   = SOGA.CrossOver(oldpop.Indiv[mate1],
oldpop.Indiv[mate2], jcross, lchrom, pmutation);
                newpop.Indiv[i+1] = SOGA.CrossOver(oldpop.Indiv[mate2],
oldpop.Indiv[mate1], jcross, lchrom, pmutation);

                nmutation = SOGA.number_of_mutations;

                newpop.Indiv[i].parent1 = mate1;
                newpop.Indiv[i].parent2 = mate2;
                newpop.Indiv[i].xsite = jcross;
                newpop.Indiv[i+1].parent1 = mate1;
                newpop.Indiv[i+1].parent2 = mate2;
                newpop.Indiv[i+1].xsite = jcross;

          }

          newpop.Report(gen+1);
          oldpop=newpop;
          gen++;
        } while (gen<maxgen);

        System.out.println("\nNumber of Cross-Overs: " + ncross);
        System.out.println("Number of Mutations  : " + nmutation);
  }
}
```

FILE: Population.java

```
/**
```

```
  Class:
    Population

  Proterties:
    int size          -- stores the number of individuals
    double fitness    -- average fitness of the population
    double sum

  Methods:
    Regenerate

 */

import java.lang.Math;

class Population
{
  public Individual[] Indiv; // = new Individual[10];
  public int size;
  public double max, min, avg, sumfitness;

  public Population()
  {
        // Constructor: Initialize Population with 10 Individuals

        size = 10;
          sumfitness = 0;
        Indiv = new Individual[size];
        for (int i = 0; i < size; i++)
        {
              Indiv[i] = new Individual(5);
          sumfitness += Indiv[i].Fitness();
          //System.out.println(Indiv[i].gene + " " +
Indiv[i].gene.Value() + " " + Indiv[i].Fitness());
      }
        for (int j = 0; j < size; j++)
        {
          Indiv[j].pselect = Indiv[j].Fitness() / sumfitness;
        }

   }

  public Population(int s, int l)
  {
        // Constructor: Initialize size
        double j;
        size = s;
        Indiv = new Individual[size];

        Indiv[0] = new Individual(l);
        sumfitness = Indiv[0].Fitness();
        min = sumfitness;
        max = sumfitness;

        for (int i = 1; i < size; i++)
        {
              Indiv[i] = new Individual(l);
```

```
                j = Indiv[i].Fitness();
                sumfitness += j;
                if (j>max) max = j;
                if (j<min) min = j;
         }
         avg = sumfitness / size;
  }

  public Population(Population PopSource)
         // This is a copy constructor:
         // you can create a new pop from an existing one
  {
       size = PopSource.size;
       // Indiv = new Individual(PopSource.Indiv);
  }

  public int Select()
  {
   double rand, partsum;
   int j;

   partsum = 0.0;
   j = 0;
   rand = Math.random() * sumfitness;
   //   System.out.println("rand is : "+ rand);
   do
   {
     partsum+=Indiv[j].Fitness();
     j++;
   } while ((partsum < rand) && (j != size));

   return j-1;
  }


  public void InitReport()
  {
          // Prints the population contents of the first generation
          System.out.println("\nGeneration 0:");
          System.out.println("\n----------------------------------------
-----");
          System.out.println("\nNo String    (x)    f(x)    Parent1
Parent2 XSite");

          for (int i = 0; i < size; i++)
          {
              System.out.println(i+" "+Indiv[i].gene + "    " +
Indiv[i].gene.Value() + "   " + Indiv[i].Fitness() + "
"+Indiv[i].parent1 +" " +Indiv[i].parent2+" "+Indiv[i].xsite);
          }
          System.out.println("----------------------------------------
---");
          System.out.println("Max: "+max+" Min: "+min+" Avg: "+avg+"
Total Fitness: "+sumfitness);
  }

  public void Report(int g)  // g is the number of the generation
  {
          // Prints the population contents of the first generation
          System.out.println("\nGeneration:"+g);
          System.out.println("\n----------------------------------------
-----");
          System.out.println("\nNo String     (x)    f(x)    Parent1
Parent2 XSite");

        double j;

        sumfitness = Indiv[0].Fitness();
        min = sumfitness;
        max = sumfitness;

        for (int i = 1; i < size; i++)
        {
             j = Indiv[i].Fitness();
             sumfitness += j;
             if (j>max) max = j;
             if (j<min) min = j;
        }
        avg = sumfitness / size;

          for (int i = 0; i < size; i++)
          {
               System.out.println(i+" "+Indiv[i].gene + "    " +
Indiv[i].gene.Value() + "   " + Indiv[i].Fitness() + "
"+Indiv[i].parent1 +" " +Indiv[i].parent2+" "+Indiv[i].xsite);
          }
          System.out.println("----------------------------------------
---");
          System.out.println("Max: "+max+" Min: "+min+" Avg: "+avg+"
Total Fitness: "+sumfitness);
  }

}
```

FILE: Individual.java

```
/**

  Class:
    Individual

  Proterties:
    Chromosome gene   -- Genetic make-up/alleles
    double value      -- decoded value of gene; x
    double fitness    -- f(x) value

  Methods:
    Individual()
```

```
     Individual(int l)


  */
 class Individual
 {
       public Chromosome gene;
       public double pselect; // probability of selection f/Sum(f)
         protected int parent1, parent2, xsite;
         private int chrom_length;

         public int A[][] = { {0,4,2,7,9,3},
                              {4,0,6,8,3,4},
                              {2,6,0,3,2,9},
                              {7,8,3,0,6,3},
                              {9,3,2,6,0,2},
                              {3,4,9,3,2,0} };

       public Individual(int l) // this one
       {
         chrom_length = l;
         gene = new Chromosome(l);
         parent1 = -1;
         parent2 = -1;
         xsite = -1;
       }

     public Individual(int l, int p1, int p2)
       {
           gene = new Chromosome(l);
           parent1 = p1;
           parent2 = p2;
         }

     public double Fitness()
       {
             // Calculate the fitness of the individual
             // given x, find f(x)
                 int dist, dist_a, dist_b;
                 double accum =0;
                 for (int i=0; i<chrom_length;i++)
                 {
                   for (int j=i+1; j<chrom_length;j++)
                    {
                     dist_a = gene.A_allele[i].i_bit -
gene.A_allele[j].i_bit;
                     if (dist_a < 0) { dist_a += chrom_length; }
                     dist_b = gene.A_allele[j].i_bit -
gene.A_allele[i].i_bit;
                     if (dist_b < 0) { dist_b += chrom_length; }

                     dist = Math.min(dist_a, dist_b);

                     accum+= dist * A[i][j];
                    }
                 }
                 return (1000 - accum); //Math.pow(gene.Value(), 2);
```

```
        }

        boolean Flip(double p)
        {
               // Biased coin, True p of the time
               // 0 <= p <= 1
               if (p == 0)
                      return false;
               else
                          return (Math.random() <= p);
        }


}
```

FILE: Chromosome.java

```
/**
   (c)1999 by Ziad Kobti. All Rights Reserved

   Class: Chromosome          -- Implements a chromosome strand (gene)

   Variables:
     int i_length             -- Length of the chromosome strand
     Allele[] A_allele        -- Array of alleles

   Methods:
     Chromosome(int l)
     public void GenerateRandomChromosome()
     public void Mutate(int i_locus1, int i_locus2)
     public double Value()
     public String toString()


*/



import java.lang.Math;

public class Chromosome
{
    public int i_length;
    public Allele[] A_allele;

    Chromosome(int l)
      {
             i_length = l;
```

```
            GenerateRandomChromosome();
      }

    public void GenerateRandomChromosome()
      {
            int which;
            int p1, p2;
            A_allele = new Allele[i_length];

            for (int c = 0; c < i_length; c++)
            {
                        A_allele[c] = new Allele();
                        // which = 1 + (int) (Math.random() * i_length);
                        A_allele[c].i_bit = c;
            }

            // random shuffle
            for (int c=0; c < i_length; c++)
                {
                  p1 = 0 + (int) (Math.random() * (i_length));
                  p2 = 0 + (int) (Math.random() * (i_length));
                  Mutate(p1, p2);
                }

        }

    public void Mutate(int i_locus1, int i_locus2)
      {
            // Mutate one or more allele in the chromosome string
            // implements single point mutation at random position
            // missing mutation rate, this is placed at the calling
function level
            // int i_locus;
            // i_locus = 0 + (int) (Math.random() * (i_length - 1));

            int temp;
            temp = A_allele[i_locus2].i_bit;
            A_allele[i_locus2].i_bit = A_allele[i_locus1].i_bit;
            A_allele[i_locus1].i_bit = temp;
      }


    public double Value()
      {
               return 0;
      }

    public String toString()
      {
         String s_contents = "";
         for (int i=0; i<i_length; i++)
           {
              s_contents += A_allele[i].i_bit;
           }
         return s_contents;
      }
}
```

FILE: Allele.java

```
/**
  (c)1999 by Ziad Kobti. All Rights Reserved

  Class: Allele             -- Implements a single integer digit allele

  Variables:
    int i_bit               -- Bit value

  Methods:
    int Allele()            -- Constructor: Initializes bit to 0
    int Allele(int)         -- Overloaded Constructor:
                               Initializes to the given digit

*/

public class Allele {

    public int i_bit; // bit value

    // Constructor: default digit is 0
    public Allele()
      {
                i_bit = 0;
      }

    // Overloaded Constructor with parameter
    public Allele(int b)
      {
                i_bit = b;
        }
}
```

APPENDIX D: OGA SOURCE CODE LISTING

Source code listing for the Optical Genetic Algorithm (OGA). Developed by Ziad Kobti to present an improved version over its predecessor the SOGA.

FILE: OGA.java

```
import java.io.*;
import java.util.*;

public class OGA
{
  public static void main(String argv[])
  {
   int x = 10;
   x = Integer.parseInt(argv[0]);
//    Population P = new Population(300, 30);
    Population P = new Population(x, 30);
    for (int I=0; i < 30; i++)
    {
     P.Sort();
     P.Report(i);
     P.CrossOver();
    }
  }
}
```

FILE: Population.java

```
/*
 *
 * Class:
 *          Population
 *
 * Proterties:
 *            int size : stores the number of individuals
 *            double fitness: average fitness of the population
 *            double sum? : f(x)
 *
 * Methods:
 *          Regenerate : create
 */
```

```
import java.lang.Math;
class Population
{


  public Individual[] Indiv; // = new Individual[10];
  public int rank[];
//  public tempIndiv Indiv;
  public int size;
  public double max, min, avg, sumfitness;
  public int chromlength;

  public Population(int s, int l)
  {
        // Constructor: Initialize size
        double j;
        size = s;
          chromlength = l;

        Indiv = new Individual[size];

          this.rank  = new int[s];
          for (int r=0; r<s; r++)
           { this.rank[r] = r; }

        Indiv[0] = new Individual(l);
//        tempIndiv = new Individual(l);
        sumfitness = Indiv[0].Fitness();
        min = sumfitness;
        max = sumfitness;

        for (int i = 1; i < size; i++)
        {
              Indiv[i] = new Individual(l);
              j = Indiv[i].Fitness();
              sumfitness += j;
              if (j>max) max = j;
              if (j<min) min = j;
        }
        avg = sumfitness / size;
  }

  public void Sort()
  {       Individual TempIndiv = new Individual(chromlength);
          for (int c = 0; c < size; c++)
        {
            for(int d=(size-2); d>=c; d--)
            {
                  if (Indiv[d].fitvalue < Indiv[d+1].fitvalue)
                  {
                    TempIndiv = Indiv[d];
                    Indiv[d] = Indiv[d+1];
                    Indiv[d+1] = TempIndiv;
                  }
            }
        }
```

```
    }

   public int Select()
   {
    double rand, partsum;
    int j;

    partsum = 0.0;
    j = 0;
    rand = Math.random() * sumfitness;
    //   System.out.println("rand is : "+ rand);
    do
    {
      partsum+=Indiv[j].Fitness();
      j++;
    } while ((partsum < rand) && (j != size));

    return j-1;
   }



   public int SelectFromRank()
   {
    double rand, partsum;
    int j;

    partsum = 0.0;
    j = 0;
    double subsumfitness = 0;
    for (int c=0; c<=39; c++)
     {
      subsumfitness += Indiv[rank[c]].fitvalue;
     }
    rand = Math.random() * subsumfitness;
    //   System.out.println("rand is : "+ rand);
    do
    {
      partsum+=Indiv[rank[j]].Fitness();
      j++;
    } while ((partsum < rand) && (j != (40)));

//   System.out.println("Selected: ");
    return rank[j-1];
   }

/*  public void SortPopulation(int lobound, int hibound)
  {
   int pivot, loSwap, hiSwap;

   if (hiBound - loBound == 1)
    {
     if (Indiv[loBound].fitvalue > Indiv[hiBound].fitvalue)
      {
       temp = Indiv[loBound].fitvalue;
  }
*/
/*
  public void GetTopIndividuals()
  {
   for (int i = 0
  }
*/
/*
  public void InitReport()
  {
          // Prints the population contents of the first generation
          System.out.println("\nGeneration 0:");
          System.out.println("\n---------------------------------------
-----");
          System.out.println("\nNo String    (x)   f(x)   Parent1
Parent2 XSite");

          for (int i = 0; i < size; i++)
          {
              System.out.println(i+" "+Indiv[i].gene + "    " +
Indiv[i].gene.Value() + "  " + Indiv[i].Fitness() + "
"+Indiv[i].parent1 +" " +Indiv[i].parent2+" "+Indiv[i].xsite);
          }
          System.out.println("-----------------------------------------
---");
          System.out.println("Max: "+max+" Min: "+min+" Avg: "+avg+"
Total Fitness: "+sumfitness);
  }
  */

  public void Report(int g)  // g is the number of the generation
  {
          // Prints the population contents of the first generation
          System.out.println("\nGeneration:"+g);
          System.out.println("\n---------------------------------------
-----");
          System.out.println("\nNo String    (x)   f(x)   Parent1
Parent2 XSite");

        double j;

          sumfitness = Indiv[0].fitvalue;
        min = sumfitness;
        max = sumfitness;

        for (int i = 1; i < size; i++)
        {
                j = Indiv[i].fitvalue;
              sumfitness += j;
              if (j>max) max = j;
              if (j<min) min = j;
        }
        avg = sumfitness / size;

          for (int i = 0; i < size; i++)
          {
                System.out.println(i+" "+Indiv[i].gene + "   " +
Indiv[i].fitvalue);
           }
           System.out.println("-----------------------------------------
---");
           System.out.println("Max: "+max+" Min: "+min+" Avg: "+avg+"
Total Fitness: "+sumfitness);
   }

/*
public void SortRanks()
 {
  int i=0;
  int p=0;
  int t=0;
  for (i=0; i < size; i++)
   { this.rank[i] = i; }

  for (i=0; i<size; i++)
   {
     t=rank[i];
     p = FindMax(i, size-1);
     this.rank[i] = this.rank[p];
     this.rank[p] = t;
   }

  for (i=0; i < size; i++)
   { System.out.println
("Rank:"+this.rank[i]+";F:"+Indiv[this.rank[i]].fitvalue); }

 }
*/
public int FindMax(int a, int b)
 {
   double max = Indiv[this.rank[a]].fitvalue;
   int p = a;
   for (int x=a+1; x <= b; x++)
    {
     if (Indiv[this.rank[x]].fitvalue > max)
      { p = x; max = Indiv[this.rank[x]].fitvalue; }
    }
   return p;
 }


public void CrossOver()
{
  int p1,p2;
  for (int m=0; m<size/2;m++)
  {
   Indiv[m+size/2].fitvalue = Indiv[m].fitvalue;
   Indiv[m+size/2].gene = Indiv[m].gene;
  }
   Indiv[size-1].fitvalue = Indiv[0].fitvalue;
   Indiv[size-1].gene = Indiv[0].gene;

   Indiv[size-2].fitvalue = Indiv[1].fitvalue;
```

```
    Indiv[size-2].gene = Indiv[1].gene;

    for (int p=size/4; p <size; p++)
    {
    do{
     p1 = 2 + (int) (Math.random() * (size-2));
     p2 = 2 + (int) (Math.random() * (size-2));
//     System.out.println(p1 + " " + p2 + " " + p);
     } while (p1 == p || p2 == p || p1 == p2);
  // System.out.println("before: " + Indiv[p].gene);
     Indiv[p].gene = Cross(Indiv[p1].gene, Indiv[p2].gene);
     Indiv[p].Fitness();

     if ((int)(Math.random() * 100) > 79) // 20% mutation rate
     {
      Indiv[p].gene.Mutate();
     }

  //  System.out.println("after: " + Indiv[p].gene);
    }

}

public Chromosome Cross(Chromosome Parent1, Chromosome Parent2)
{
 int a, b, c;
 Chromosome child = new Chromosome(chromlength);
// System.out.println("P1:" + Parent1 + " P2:" + Parent2);
 child.SetAllele(0, Parent2.GetAllele(0));
  boolean found = false;

 for (int i=1; i<chromlength; i++)
 {
  a = FindNextValue(Parent1, child.GetAllele(i-1));
  b = FindNextValue(Parent2, child.GetAllele(i-1));
  if (Indiv[0].A[child.GetAllele(i-1)][a] >
Indiv[0].A[child.GetAllele(i-1)][b])
    {
     c = a;
     for (int j=0; j< i && c != b; j++)
     {
       if (child.GetAllele(j) == c)
       {
         c=b;

       }
     }

    }
  else
    {
     c = b;
     for (int j=0; j< i && c != a; j++)
     {
       if (child.GetAllele(j) == c)
       {
         c=a;
         }
       }
     }


      for (int j=0; j< i && found==false; j++)
      {
        if (child.GetAllele(j) == c)
        {
          found = true; // get any valid value
        }
      }





   if (found == true)
   {
    c = -1;
    do {
    found = false;
    c++;
     for (int j=0; j < i; j++)
     {
      if (child.GetAllele(j) == c)
        found = true;
     }

    }while(found == true);
   }
   child.SetAllele(i,c);

  }
  return child;
 }

public int FindNextValue (Chromosome g, int key)
  {
    for (int i=0; i < chromlength-1; i++)
    {
     if (g.GetAllele(i) == key)
       return g.GetAllele(i+1);
    }
    return g.GetAllele(0);
  }

}
```

FILE: Individual.java

```
/**
 *
 * Class:
 *                 Individual
 *
 * Proterties:
 *             Chromosome gene : Genetic make-up/alleles
 *             double value : decoded value of gene; x
 *             double fitness : f(x)
 *
 * Methods:
 *                 Individual() : constructor, creates default gene
 *             Individual(int l) : overloaded constructor with l alleles
 *             in constructor, calculate value and fitness
 */
class Individual
{
    public Chromosome gene; // single chromosome
    public double pselect, fitvalue; // probability of selection
f/Sum(f)
    private int parent1, parent2, xsite;
    private int chrom_length;

 /** The large clustered matrix:
     This matrix is designed to put the genetic algorithm to the test
in
     1. Being able to handle large matrix size
     2. Discover patterns generated by means of clustered communication
        e.g. By examining this matrix onr can see the most fit pattern
to
        be in the following circular arrangement:
        [0 1 2 3 4 5][6 7 8 9 10][11 12 13 14 15]
        ...[16 17 18 19 20][21 22 23 24 25][26 27 28 29]
        where [x y z] represents any combination of xyz, xzy, yxz, ...
        that is they are grouped together, but of any order
 */

 public int A[][] =
{ { 0,50,50,50,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  {50, 0,50,50,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  {50,50, 0,50,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  {50,50,50, 0,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  {50,50,50,50, 0,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  {50,50,50,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0,50,50,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0,50, 0,50,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0,50,50, 0,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0,50,50,50, 0,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0,50,50,50,50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50,50,50, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50, 0,50,50,50, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50, 0,50,50, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50,50, 0,50, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50,50,50, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50,50,50, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50, 0,50,50,50, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50, 0,50,50, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50,50, 0,50, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,50,50,50, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,50,50,50,50, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,50,
0,50,50,50, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,50,50, 0,50,50, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,50,50,50, 0,50, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,50,50,50,50, 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,50,50,50},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0,50, 0,50,50},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0,50,50, 0,50},
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0,50,50,50, 0} };


/** a typical random small matrix
    Note the diagonal symmetry
*/
/*          public int A[][] = { {0,4,2,7,9,3},
                                 {4,0,6,8,3,4},
                                 {2,6,0,3,2,9},
                                 {7,8,3,0,6,3},
                                 {9,3,2,6,0,2},
                                 {3,4,9,3,2,0} };
  */

    /** The constructor requires only the size of the individual
        -a random chromosome is generated
        -assume one gene/chromosome per individual
        -default parents and cross-site are -1
    */
    public Individual(int l) // this one
    {
         chrom_length = l;
         gene = new Chromosome(l);
         parent1 = -1;
         parent2 = -1;
         xsite = -1;
    }

    /** overloaded constructor - Not used */
    public Individual(int l, int p1, int p2)
      {
         gene = new Chromosome(l);
         parent1 = p1;
         parent2 = p2;
        }

    public double Fitness()
      {
           // Calculate the fitness of the individual
           // given x, find f(x)
               int dist, dist_a, dist_b;
               double accum =0;
               for (int i=0; i<chrom_length;i++)
               {
                 for (int j=i+1; j<chrom_length;j++)
                  {
                   dist_a = gene.GetAllele(i) - gene.GetAllele(j);
                   if (dist_a < 0) { dist_a += chrom_length; }
                   dist_b = gene.GetAllele(j) - gene.GetAllele(i);
                   if (dist_b < 0) { dist_b += chrom_length; }

                   dist = Math.min(dist_a, dist_b);

                   accum+= dist * A[i][j];
                  }
               }
               fitvalue = (100000 - accum);
               return fitvalue; //Math.pow(gene.Value(), 2);
     }

     public boolean Flip(double p)
     {
           // Biased coin, True p of the time
           // 0 <= p <= 1
           if (p == 0)
                 return false;
           else
               return (Math.random() <= p);
     }


}
```

FILE: Chromosome.java

```
/**

 Class Chromosome:
 Describes a Chromosome structure
 A chromosome is an array of alleles
 it is also known as the gene itself
 @author Ziad Kobti
 @version 2.0

*/

import java.lang.Math;

public class Chromosome
{
    /** i_length : length of the chromosome */
    private int i_length;

    /** A_allele : the array of alleles definition */
    private Allele[] A_allele;

    /** initializes the chromosome to the given size
        and produces a random set of alleles of values
        ranging from 0 to l-1
    */
    public Chromosome(int l)
    {
        i_length = l;
        GenerateRandomChromosome();
    }

   public int GetAllele(int pos)
   {
     return A_allele[pos].GetAllele();
   }

   public void SetAllele(int pos, int newallele)
   {
    A_allele[pos].SetAllele(newallele);
   }


    /** Generates random values of alleles ranging from
        0 to l-1 and fills the array A_allele[]
    */
    private void GenerateRandomChromosome()
    {
        int which;
        int p1, p2;
        A_allele = new Allele[i_length];

        for (int c = 0; c < i_length; c++)
            {
                        A_allele[c] = new Allele();
                        // CONSTRAINT:
                        // When creating random values the permutation
                        // constraint is not satisfied. Instead we'll
                        // generate values from 0 to l-1 to fill the
array
                        // and then do random shuffling to guarantee the
                        // final array to be random, but a permutation.
                        // NO: which = 1 + (int) (Math.random() *
i_length);
                        A_allele[c].SetAllele(c);
            }

            // random shuffling of an otherwise uniform array
            // occurs using a repeated random two point swapping
            // the Mutate method accomlishes this
            for (int c=0; c < i_length; c++)
                {
                  p1 = 0 + (int) (Math.random() * (i_length)); //
0<=p1<i_length
                  p2 = 0 + (int) (Math.random() * (i_length)); //
0<=p2<i_length
                  Mutate(p1, p2);
                }

        }

    /** CONSTRAINT:
        In order to preserve the permutation status of the gene
        array, we cannot randomly pick a point and toggle it to
        something else (single point mutation). Instead the two
        point mutation technique is deployed which will guarantee
        the permutation status of the gene.
        The algorithm will randomly pick to positions in the array
        and swap them. No new values are introduced within the gene
        rather only two values re-arranged.
        MUTATION RATE: not readable at this point. This method does
        not decide whether mutation is to take place or not, it
        simply does it. The decision process based on the given
        rate of mutation is done at the calling level.
    */
    public void Mutate(int i_locus1, int i_locus2)
    {
       // Single point mutation code in comments
       // Mutate one or more allele in the chromosome string
       // implements single point mutation at random position
       // missing mutation rate, this is placed at the calling function
level
       // int i_locus;
       // i_locus = 0 + (int) (Math.random() * (i_length - 1));

       int temp;
       temp = A_allele[i_locus2].GetAllele();
```

```
        A_allele[i_locus2].SetAllele(A_allele[i_locus1].GetAllele());
        A_allele[i_locus1].SetAllele(temp);
    }


    public void Mutate()
    {
        //int size; // defines n point swap (ie how many alleles to
swap)
        int i_locus1, i_locus2;
        int temp;
        //size = 1; ALWAYS SINGLE POINT MUTATION
        // + (int) (Math.random() * (( int) (i_length / 2));
        i_locus1   = 0 + (int) (Math.random() * i_length);
        i_locus2 = i_locus1;
        while (i_locus2 == i_locus1)
        {
            i_locus2   = 0 + (int) (Math.random() * i_length);
        }

        temp = A_allele[i_locus2].GetAllele();
        A_allele[i_locus2].SetAllele(A_allele[i_locus1].GetAllele());
        A_allele[i_locus1].SetAllele(temp);
    }



    /** The value of the chromosome is the decoded equivalent
        of the encoded array. In a typical binary array, the
        value would be the decimal equivalent of the binary
        array. However, in this particular case, the array is
        made up of integers together forming a permutation.
        Hence, there is no lower level decoding to occur.
        (ie. the value is just the array itself, which is implemented
        in the next method toString())
     */
    public double Value()
    {
        return 0;
    }


    /** This method will produce a string representation of the
        contents of the gene array for examination purposes.
    */
    public String toString()
    {
        String s_contents = "";
        for (int i=0; i<i_length; i++)
        {
            s_contents += A_allele[i].GetAllele();
        }
        return s_contents;
    }
}
```

FILE: Allele.java

```
/**

 Class Allele:
 Describes the basic unit of the gene
 An allele is a 0 or positive integer
 @author Ziad Kobti
 @version 2.0
 (c) 1997-99 by Ziad Kobti - All rights reserved

*/


public class Allele {

    /** Private integer bit value */
    private int i_bit;

    /** Default value of an allele is 0 CAUTION: May cause lethality */
    public Allele()
    {
        i_bit = 0;
    }

    /** Overloaded Constructor with parameter */
    public Allele(int b)
    {
        SetAllele(b);
    }

    /** SetAllele value to a new value */
    public void SetAllele(int b)
     {
         if (b < 0)
         {
             System.out.println("Warning: Illegal allele
representation: " + b);
         }
         i_bit = b;
     }

     /** Retrieves an allele value */
     public int GetAllele()
     {
         return i_bit;
     }
}
```

APPENDIX E: RANDOM SEARCH SOURCE CODE

The following listing is the complete source code for the Random Search Algorithm heuristic written in Java.

FILE: Random.java

```
class random
{
 public int A[];
 public int size;

public int F[][] = {
          { 0, 13, 7, 79, 84, 57, 34, 2, 14, 66, 41, 94, 74, 94, 58, 42,
48, 14, 21, 23, 66, 19, 0, 98, 53, 14, 86, 91, 12, 4, 81, 10, 39, 67,
44, 73, 49, 84, 8, 64, 7, 70, 85, 14, 63, 30, 71, 15, 67, 35, 51, 20,
94, 63, 80, 20, 70, 14, 79, 19, 55, 81, 11, 12, 18, 44, 75, 41, 74, 90,
8, 78, 93, 2, 75, 83, 88, 63, 38, 16, 41, 4, 33, 55, 59, 82, 95, 58,
76, 29, 1, 3, 72, 22, 22, 21, 7, 20, 24, 80, },
          { 13, 0, 38, 88, 73, 3, 0, 2, 54, 7, 31, 14, 48, 33, 25, 33,
47, 20, 8, 83, 66, 55, 48, 57, 8, 34, 4, 35, 63, 87, 77, 42, 95, 40,
99, 55, 29, 16, 65, 1, 2, 25, 0, 78, 25, 80, 25, 59, 67, 85, 78, 39,
29, 75, 20, 92, 89, 70, 62, 58, 23, 87, 27, 46, 53, 78, 76, 44, 5, 23,
93, 40, 77, 80, 73, 50, 94, 1, 20, 80, 48, 63, 97, 20, 64, 25, 64, 57,
14, 9, 80, 3, 12, 46, 23, 2, 28, 36, 58, 29, },
          { 7, 38, 0, 67, 91, 44, 16, 79, 0, 59, 73, 77, 16, 1, 3, 55,
7, 1, 49, 40, 99, 87, 69, 11, 87, 93, 17, 67, 94, 49, 78, 52, 54, 40,
80, 81, 11, 52, 61, 41, 13, 94, 18, 5, 86, 40, 27, 2, 22, 84, 88, 90,
52, 53, 20, 70, 8, 39, 69, 77, 43, 91, 29, 16, 94, 84, 51, 97, 77, 53,
33, 42, 83, 51, 83, 65, 0, 90, 46, 26, 15, 68, 40, 10, 96, 40, 92, 68,
86, 90, 67, 41, 90, 45, 17, 52, 97, 98, 4, 61, },
          { 79, 88, 67, 0, 96, 57, 41, 45, 71, 88, 19, 64, 91, 96, 76,
83, 59, 4, 87, 79, 17, 10, 85, 21, 25, 2, 93, 48, 40, 43, 54, 42, 65,
52, 53, 51, 23, 28, 96, 62, 21, 6, 81, 24, 78, 43, 13, 64, 87, 88, 42,
15, 27, 8, 85, 60, 15, 96, 68, 81, 66, 15, 63, 51, 66, 9, 31, 39, 18,
2, 78, 43, 75, 44, 2, 97, 30, 25, 7, 45, 9, 13, 33, 38, 32, 94, 44, 33,
57, 59, 39, 46, 79, 68, 72, 82, 66, 83, 89, 73, },
          { 84, 73, 91, 96, 0, 31, 21, 38, 93, 7, 95, 99, 7, 52, 37, 69,
55, 57, 39, 11, 81, 28, 85, 92, 33, 14, 73, 97, 30, 35, 90, 34, 64, 80,
27, 11, 95, 11, 99, 21, 5, 87, 45, 92, 20, 7, 96, 2, 73, 53, 88, 80, 1,
23, 35, 74, 48, 84, 70, 7, 75, 61, 86, 64, 54, 57, 42, 61, 70, 38, 5,
9, 30, 65, 40, 61, 35, 50, 82, 51, 44, 0, 70, 59, 36, 85, 35, 7, 37,
98, 8, 79, 89, 7, 31, 6, 76, 84, 8, 55, },
          { 57, 3, 44, 57, 31, 0, 51, 81, 80, 1, 63, 85, 75, 17, 99, 95,
11, 83, 34, 0, 29, 57, 77, 7, 34, 66, 36, 47, 73, 54, 73, 25, 53, 27,
75, 17, 47, 50, 64, 86, 73, 4, 58, 22, 11, 7, 54, 7, 25, 92, 88, 2, 10,
85, 64, 13, 95, 97, 78, 29, 12, 15, 73, 70, 38, 31, 1, 36, 7, 68, 21,
82, 2, 59, 69, 42, 17, 38, 0, 13, 11, 28, 35, 63, 65, 87, 11, 69, 66,
49, 51, 26, 82, 29, 66, 28, 61, 2, 15, 26, },
          { 34, 0, 16, 41, 21, 51, 0, 69, 39, 51, 11, 44, 88, 10, 90,
34, 26, 88, 46, 49, 51, 5, 45, 65, 97, 55, 36, 81, 36, 69, 0, 29, 8,
58, 67, 47, 71, 25, 81, 85, 73, 66, 94, 48, 70, 48, 3, 64, 76, 40, 5,
75, 9, 75, 35, 55, 22, 30, 40, 84, 23, 80, 19, 92, 98, 22, 17, 45, 4,
44, 39, 34, 49, 46, 11, 68, 31, 16, 43, 90, 79, 88, 51, 79, 44, 36, 83,
23, 46, 62, 64, 84, 32, 85, 8, 32, 93, 42, 72, 40, },
          { 2, 2, 79, 45, 38, 81, 69, 0, 23, 68, 32, 10, 40, 47, 6, 32,
75, 49, 64, 14, 41, 43, 16, 24, 17, 94, 74, 94, 67, 99, 76, 24, 84, 98,
19, 83, 73, 3, 15, 81, 46, 92, 69, 30, 57, 42, 29, 25, 28, 55, 94, 22,
32, 89, 49, 87, 67, 87, 19, 58, 50, 53, 70, 83, 52, 16, 36, 33, 66, 91,
74, 41, 50, 34, 42, 64, 16, 78, 88, 2, 10, 98, 51, 78, 60, 95, 74, 29,
86, 32, 74, 12, 11, 77, 39, 3, 53, 24, 93, 23, },
          { 14, 54, 0, 71, 93, 80, 39, 23, 0, 38, 21, 94, 95, 70, 0, 58,
5, 34, 63, 33, 28, 33, 14, 75, 36, 78, 63, 87, 89, 27, 92, 83, 33, 24,
63, 70, 37, 53, 19, 30, 81, 7, 96, 47, 26, 8, 63, 91, 42, 51, 77, 71,
40, 28, 52, 95, 82, 4, 91, 65, 6, 28, 77, 45, 79, 53, 64, 68, 27, 4,
67, 59, 70, 80, 86, 54, 60, 77, 72, 65, 76, 68, 11, 90, 33, 85, 37, 14,
2, 73, 70, 9, 56, 70, 59, 11, 3, 8, 46, 99, },
          { 66, 7, 59, 88, 7, 1, 51, 68, 38, 0, 99, 8, 74, 74, 69, 15,
28, 37, 20, 71, 85, 57, 54, 25, 24, 27, 71, 34, 55, 16, 70, 16, 94, 10,
27, 55, 21, 96, 9, 30, 58, 64, 92, 94, 88, 66, 76, 15, 63, 1, 6, 50,
27, 33, 27, 91, 29, 45, 32, 36, 77, 12, 63, 57, 12, 37, 9, 9, 27, 25,
17, 94, 33, 92, 9, 36, 32, 75, 34, 33, 85, 32, 3, 64, 72, 98, 35, 82,
8, 46, 61, 10, 86, 93, 5, 42, 85, 49, 17, 12, },
          { 41, 31, 73, 19, 95, 63, 11, 32, 21, 99, 0, 14, 4, 63, 47, 8,
64, 89, 86, 46, 68, 88, 18, 24, 28, 15, 86, 55, 77, 16, 11, 45, 2, 64,
79, 52, 24, 32, 86, 87, 44, 30, 52, 18, 8, 22, 57, 79, 58, 7, 31, 73,
33, 61, 74, 50, 46, 16, 88, 96, 26, 41, 51, 87, 14, 20, 44, 14, 50, 78,
55, 16, 13, 63, 2, 14, 2, 52, 4, 67, 21, 3, 83, 55, 8, 57, 50, 2, 88,
33, 24, 69, 55, 74, 37, 30, 97, 15, 78, 98, },
          { 94, 14, 77, 64, 99, 85, 44, 10, 94, 8, 14, 0, 35, 98, 60,
41, 38, 59, 29, 91, 83, 67, 80, 22, 64, 27, 86, 59, 26, 88, 53, 44, 94,
57, 50, 76, 90, 56, 78, 33, 90, 10, 55, 53, 27, 4, 87, 66, 4, 10, 16,
42, 49, 37, 25, 44, 83, 20, 53, 34, 92, 32, 87, 43, 76, 66, 69, 59, 50,
13, 8, 51, 84, 23, 93, 20, 40, 31, 48, 24, 37, 0, 22, 92, 92, 5, 84,
32, 7, 8, 25, 82, 83, 6, 13, 76, 69, 90, 90, 32, },
          { 74, 48, 16, 91, 7, 75, 88, 40, 95, 74, 4, 35, 0, 54, 93, 53,
94, 86, 44, 27, 14, 67, 30, 55, 91, 74, 71, 58, 36, 14, 37, 85, 57, 67,
30, 58, 63, 69, 72, 44, 58, 8, 87, 27, 42, 48, 25, 24, 20, 3, 76, 10,
46, 18, 81, 21, 36, 47, 23, 93, 12, 72, 95, 53, 64, 96, 89, 30, 22, 66,
65, 52, 99, 27, 48, 9, 44, 68, 0, 39, 39, 45, 91, 33, 57, 83, 35, 89,
79, 3, 25, 79, 0, 49, 31, 36, 89, 43, 27, 81, },
          { 94, 33, 1, 96, 52, 17, 10, 47, 70, 74, 63, 98, 54, 0, 37, 8,
45, 10, 80, 93, 35, 27, 18, 13, 86, 68, 54, 67, 22, 71, 82, 16, 82, 77,
89, 41, 68, 9, 37, 38, 9, 67, 14, 28, 22, 69, 0, 46, 75, 54, 61, 55,
32, 97, 97, 56, 59, 0, 17, 18, 76, 61, 44, 47, 38, 60, 1, 77, 25, 97,
64, 72, 57, 42, 37, 6, 16, 57, 15, 76, 29, 72, 12, 20, 26, 53, 24, 70,
63, 40, 22, 35, 68, 97, 62, 21, 60, 67, 55, 36, },
          { 58, 25, 3, 76, 37, 99, 90, 6, 0, 69, 47, 60, 93, 37, 0, 49,
76, 0, 56, 31, 29, 71, 68, 64, 23, 28, 82, 12, 16, 62, 58, 17, 62, 40,
87, 94, 28, 7, 42, 87, 50, 95, 60, 93, 10, 67, 65, 94, 34, 31, 42, 20,
99, 69, 60, 12, 96, 76, 17, 20, 43, 86, 28, 46, 69, 84, 77, 10, 82, 1,
92, 84, 21, 4, 45, 30, 30, 18, 74, 55, 61, 37, 21, 62, 87, 70, 27, 91,
16, 65, 97, 15, 75, 2, 32, 55, 59, 35, 55, 68, },
          { 42, 33, 55, 83, 69, 95, 34, 32, 58, 15, 8, 41, 53, 8, 49, 0,
72, 73, 20, 78, 9, 39, 33, 2, 78, 77, 0, 38, 36, 33, 38, 93, 69, 74,
90, 48, 18, 72, 29, 31, 91, 10, 89, 64, 51, 64, 10, 78, 7, 28, 28, 5,
14, 59, 95, 0, 13, 33, 96, 53, 24, 40, 40, 22, 72, 42, 38, 93, 43, 69,
59, 99, 38, 37, 43, 80, 14, 16, 60, 93, 65, 14, 96, 69, 52, 66, 69, 9,
30, 74, 51, 38, 9, 82, 38, 22, 5, 17, 48, 74, },
          { 48, 47, 7, 59, 55, 11, 26, 75, 5, 28, 64, 38, 94, 45, 76,
72, 0, 74, 80, 46, 0, 68, 25, 85, 27, 10, 24, 36, 15, 27, 57, 13, 38,
13, 71, 58, 78, 7, 72, 23, 95, 44, 7, 7, 56, 18, 73, 35, 80, 95, 50,
60, 47, 6, 32, 7, 4, 57, 35, 82, 29, 55, 22, 37, 51, 23, 83, 34, 9, 85,
1, 10, 63, 38, 28, 96, 24, 81, 91, 21, 62, 79, 55, 9, 85, 35, 60, 83,
67, 26, 73, 92, 67, 81, 4, 78, 59, 50, 24, 78, },
          { 14, 20, 1, 4, 57, 83, 88, 49, 34, 37, 89, 59, 86, 10, 0, 73,
74, 0, 73, 67, 82, 34, 14, 50, 98, 79, 34, 95, 54, 33, 86, 93, 23, 88,
21, 10, 13, 67, 31, 49, 19, 61, 87, 30, 31, 8, 88, 22, 65, 40, 50, 44,
48, 62, 67, 24, 5, 18, 97, 19, 72, 20, 8, 39, 87, 51, 76, 95, 77, 21,
5, 71, 21, 66, 86, 63, 7, 92, 23, 79, 95, 29, 96, 47, 37, 42, 36, 44,
89, 59, 54, 33, 32, 4, 72, 10, 62, 13, 64, 30, },
          { 21, 8, 49, 87, 39, 34, 46, 64, 63, 20, 86, 29, 44, 80, 56,
20, 80, 73, 0, 22, 69, 50, 7, 24, 75, 33, 48, 71, 51, 24, 40, 8, 21,
33, 20, 9, 69, 95, 7, 91, 46, 19, 16, 70, 49, 37, 78, 47, 71, 94, 16,
34, 76, 53, 40, 32, 72, 30, 52, 79, 41, 61, 6, 14, 0, 66, 25, 89, 74,
76, 35, 21, 70, 50, 54, 68, 56, 27, 3, 52, 58, 99, 30, 68, 45, 43, 34,
86, 18, 93, 64, 23, 65, 81, 80, 84, 13, 20, 64, 34, },
          { 23, 83, 40, 79, 11, 0, 49, 14, 33, 71, 46, 91, 27, 93, 31,
78, 46, 67, 22, 0, 43, 83, 96, 55, 32, 50, 75, 82, 88, 5, 68, 47, 14,
91, 62, 31, 50, 14, 79, 76, 29, 65, 36, 57, 5, 88, 15, 9, 67, 12, 31,
25, 51, 43, 63, 56, 36, 87, 0, 86, 12, 74, 40, 43, 84, 36, 82, 83, 86,
89, 51, 73, 80, 51, 58, 69, 81, 64, 89, 92, 31, 34, 91, 83, 30, 67, 94,
16, 14, 40, 31, 48, 71, 1, 54, 89, 47, 60, 23, 68, },
          { 66, 66, 99, 17, 81, 29, 51, 41, 28, 85, 68, 83, 14, 35, 29,
9, 0, 82, 69, 43, 0, 45, 92, 11, 86, 39, 73, 45, 58, 36, 1, 98, 63, 77,
76, 77, 16, 60, 47, 31, 30, 58, 72, 56, 27, 84, 70, 27, 3, 62, 75, 98,
71, 22, 64, 18, 44, 20, 58, 65, 50, 42, 67, 84, 14, 45, 62, 15, 80, 81,
77, 32, 91, 99, 27, 80, 0, 62, 53, 86, 20, 42, 34, 38, 94, 40, 88, 68,
6, 66, 49, 76, 31, 37, 88, 79, 67, 22, 17, 81, },
          { 19, 55, 87, 10, 28, 57, 5, 43, 33, 57, 88, 67, 67, 27, 71,
39, 68, 34, 50, 83, 45, 0, 74, 38, 95, 26, 8, 93, 52, 85, 5, 58, 13,
38, 71, 54, 7, 53, 64, 3, 47, 67, 94, 51, 33, 1, 12, 81, 70, 82, 63,
62, 59, 70, 61, 38, 51, 91, 52, 41, 44, 8, 20, 35, 27, 45, 84, 47, 87,
26, 61, 73, 7, 15, 48, 14, 8, 20, 11, 41, 27, 50, 74, 18, 13, 0, 3, 49,
2, 57, 30, 32, 20, 52, 56, 66, 40, 43, 10, 93, },
          { 0, 48, 69, 85, 85, 77, 45, 16, 14, 54, 18, 80, 30, 18, 68,
33, 25, 14, 7, 96, 92, 74, 0, 18, 19, 75, 73, 21, 28, 13, 12, 64, 92,
67, 98, 50, 28, 43, 4, 52, 41, 34, 6, 30, 36, 77, 13, 18, 16, 88, 51,
35, 70, 76, 70, 44, 16, 82, 15, 62, 79, 44, 4, 89, 96, 73, 85, 90, 92,
49, 83, 65, 57, 50, 59, 86, 74, 75, 78, 16, 53, 94, 67, 36, 34, 60, 56,
94, 86, 51, 50, 78, 61, 39, 51, 12, 3, 39, 41, 75, },
          { 98, 57, 11, 21, 92, 7, 65, 24, 75, 25, 24, 22, 55, 13, 64,
2, 85, 50, 24, 55, 11, 38, 18, 0, 22, 95, 81, 42, 93, 28, 33, 62, 36,
3, 92, 79, 51, 39, 37, 34, 90, 95, 55, 77, 71, 75, 49, 74, 81, 98, 52,
54, 71, 81, 95, 46, 4, 25, 18, 98, 66, 64, 19, 80, 3, 3, 26, 32, 42,
63, 25, 33, 55, 92, 47, 20, 98, 25, 3, 19, 77, 6, 98, 18, 64, 24, 79,
17, 47, 57, 75, 7, 99, 46, 89, 11, 3, 80, 33, 57, },
        { 53, 8, 87, 25, 33, 34, 97, 17, 36, 24, 28, 64, 91, 86, 23,
78, 27, 98, 75, 32, 86, 95, 19, 22, 0, 33, 13, 62, 31, 46, 96, 60, 12,
20, 27, 51, 77, 58, 40, 51, 95, 96, 46, 57, 66, 35, 55, 79, 69, 64, 43,
37, 25, 68, 16, 37, 14, 62, 89, 87, 14, 32, 59, 72, 88, 76, 70, 32, 61,
72, 3, 49, 0, 0, 35, 86, 46, 99, 11, 14, 4, 65, 21, 43, 53, 10, 24, 51,
88, 83, 13, 20, 86, 26, 44, 35, 31, 83, 36, 64, },
        { 14, 34, 93, 2, 14, 66, 55, 94, 78, 27, 15, 27, 74, 68, 28,
77, 10, 79, 33, 50, 39, 26, 75, 95, 33, 0, 20, 97, 80, 42, 34, 27, 77,
0, 23, 10, 65, 61, 65, 35, 90, 63, 84, 25, 9, 20, 64, 71, 36, 59, 16,
94, 51, 21, 97, 47, 53, 45, 38, 55, 24, 39, 62, 59, 64, 16, 39, 11, 40,
55, 77, 63, 66, 43, 83, 25, 71, 93, 19, 54, 52, 1, 16, 89, 69, 0, 75,
2, 74, 81, 73, 58, 60, 98, 29, 59, 34, 71, 99, 56, },
        { 86, 4, 17, 93, 73, 36, 36, 74, 63, 71, 86, 86, 71, 54, 82,
0, 24, 34, 48, 75, 73, 8, 73, 81, 13, 20, 0, 9, 75, 7, 44, 67, 62, 36,
87, 20, 9, 47, 71, 56, 87, 36, 10, 50, 39, 57, 73, 85, 37, 5, 71, 22,
24, 91, 37, 29, 80, 82, 88, 81, 56, 53, 55, 29, 92, 17, 37, 27, 23, 7,
53, 11, 76, 96, 93, 15, 25, 49, 66, 33, 17, 26, 6, 7, 2, 75, 50, 81,
17, 83, 33, 18, 24, 58, 58, 47, 93, 84, 51, 88, },
        { 91, 35, 67, 48, 97, 47, 81, 94, 87, 34, 55, 59, 58, 67, 12,
38, 36, 95, 71, 82, 45, 93, 21, 42, 62, 97, 9, 0, 41, 90, 33, 46, 36,
29, 13, 78, 13, 92, 79, 79, 65, 39, 36, 62, 55, 66, 13, 27, 27, 72, 28,
74, 58, 51, 23, 45, 26, 95, 85, 25, 94, 24, 75, 87, 71, 63, 25, 97, 5,
21, 42, 81, 92, 94, 38, 88, 21, 10, 52, 40, 58, 80, 62, 58, 80, 18, 81,
66, 91, 57, 29, 3, 31, 87, 32, 97, 64, 16, 25, 53, },
        { 12, 63, 94, 40, 30, 73, 36, 67, 89, 55, 77, 26, 36, 22, 16,
36, 15, 54, 51, 88, 58, 52, 28, 93, 31, 80, 75, 41, 0, 44, 41, 80, 98,
30, 44, 59, 74, 61, 57, 69, 93, 21, 18, 55, 22, 62, 83, 97, 90, 94, 87,
86, 77, 32, 62, 28, 54, 47, 21, 29, 51, 41, 31, 1, 17, 8, 23, 9, 59,
87, 14, 37, 43, 70, 9, 51, 79, 68, 34, 76, 77, 38, 13, 75, 60, 57, 2,
68, 48, 2, 82, 91, 45, 25, 68, 10, 12, 55, 56, 75, },
        { 4, 87, 49, 43, 35, 54, 69, 99, 27, 16, 16, 88, 14, 71, 62,
33, 27, 33, 24, 5, 36, 85, 13, 28, 46, 42, 7, 90, 44, 0, 7, 23, 31, 23,
46, 16, 52, 44, 38, 9, 55, 84, 34, 35, 89, 62, 23, 10, 33, 64, 8, 76,
41, 32, 90, 16, 87, 47, 20, 26, 88, 64, 22, 76, 14, 28, 87, 59, 10, 24,
71, 38, 0, 0, 6, 48, 42, 12, 12, 33, 72, 97, 43, 34, 85, 4, 22, 24, 56,
51, 49, 59, 42, 69, 79, 54, 53, 58, 46, 42, },
        { 81, 77, 78, 54, 90, 73, 0, 76, 92, 70, 11, 53, 37, 82, 58,
38, 57, 86, 40, 68, 1, 5, 12, 33, 96, 34, 44, 33, 41, 7, 0, 22, 52, 0,
55, 75, 93, 95, 56, 36, 70, 92, 26, 75, 1, 24, 22, 66, 74, 56, 75, 76,
47, 6, 45, 99, 62, 78, 84, 88, 21, 84, 42, 43, 52, 15, 47, 61, 23, 0,
42, 70, 62, 22, 87, 10, 62, 77, 72, 54, 34, 12, 58, 16, 60, 50, 39, 5,
7, 35, 77, 90, 46, 48, 73, 3, 42, 8, 50, 51, },
        { 10, 42, 52, 42, 34, 25, 29, 24, 83, 16, 45, 44, 85, 16, 17,
93, 13, 93, 8, 47, 98, 58, 64, 62, 60, 27, 67, 46, 80, 23, 22, 0, 67,
88, 10, 95, 28, 47, 83, 53, 59, 10, 17, 13, 36, 71, 2, 60, 54, 76, 80,
3, 7, 79, 5, 70, 23, 16, 24, 96, 76, 40, 34, 16, 66, 83, 26, 60, 72,
20, 41, 9, 15, 58, 39, 97, 54, 47, 72, 89, 96, 18, 49, 88, 64, 96, 4,
14, 79, 52, 14, 94, 18, 30, 18, 98, 8, 84, 77, 77, },
        { 39, 95, 54, 65, 64, 53, 8, 84, 33, 94, 2, 94, 57, 82, 62,
69, 38, 23, 21, 14, 63, 13, 92, 36, 12, 77, 62, 36, 98, 31, 52, 67, 0,
45, 45, 68, 61, 83, 59, 40, 79, 66, 93, 97, 83, 42, 77, 40, 89, 76, 79,
11, 45, 69, 91, 78, 84, 39, 69, 2, 21, 22, 8, 69, 18, 44, 52, 91, 43,
50, 51, 40, 0, 70, 24, 16, 49, 51, 77, 72, 76, 51, 15, 12, 5, 96, 57,
73, 29, 44, 60, 34, 38, 9, 58, 36, 34, 48, 85, 27, },
        { 67, 40, 40, 52, 80, 27, 58, 98, 24, 10, 64, 57, 67, 77, 40,
74, 13, 88, 33, 91, 77, 38, 67, 3, 20, 0, 36, 29, 30, 23, 0, 88, 45, 0,
31, 72, 99, 94, 41, 12, 70, 56, 74, 27, 47, 6, 89, 47, 91, 29, 64, 58,
90, 18, 56, 14, 80, 93, 91, 87, 14, 83, 95, 63, 33, 79, 72, 29, 39, 45,
29, 52, 52, 28, 5, 77, 21, 6, 99, 41, 28, 22, 55, 14, 16, 44, 71, 81,
71, 1, 6, 75, 48, 50, 64, 10, 64, 46, 28, 9, },
          { 44, 99, 80, 53, 27, 75, 67, 19, 63, 27, 79, 50, 30, 89, 87,
90, 71, 21, 20, 62, 76, 71, 98, 92, 27, 23, 87, 13, 44, 46, 55, 10, 45,
31, 0, 53, 87, 59, 45, 87, 51, 49, 2, 74, 9, 84, 23, 38, 20, 36, 80,
25, 84, 93, 28, 54, 65, 5, 88, 72, 99, 87, 24, 4, 77, 46, 12, 70, 67,
41, 55, 11, 5, 6, 12, 69, 36, 38, 4, 8, 78, 69, 50, 89, 79, 21, 8, 66,
31, 51, 61, 1, 37, 15, 81, 29, 27, 59, 86, 86, },
          { 73, 55, 81, 51, 11, 17, 47, 83, 70, 55, 52, 76, 58, 41, 94,
48, 58, 10, 9, 31, 77, 54, 50, 79, 51, 10, 20, 78, 59, 16, 75, 95, 68,
72, 53, 0, 6, 81, 89, 29, 2, 16, 60, 45, 9, 2, 67, 6, 56, 38, 40, 90,
14, 16, 96, 29, 76, 9, 9, 46, 47, 44, 60, 83, 62, 5, 82, 14, 40, 77,
54, 98, 86, 42, 10, 15, 15, 22, 27, 18, 25, 72, 56, 92, 73, 20, 56, 34,
19, 41, 92, 31, 57, 88, 11, 29, 67, 75, 36, 59, },
          { 49, 29, 11, 23, 95, 47, 71, 73, 37, 21, 24, 90, 63, 68, 28,
18, 78, 13, 69, 50, 16, 7, 28, 51, 77, 65, 9, 13, 74, 52, 93, 28, 61,
99, 87, 6, 0, 98, 46, 49, 23, 60, 73, 22, 2, 73, 26, 25, 0, 98, 16, 90,
15, 27, 87, 44, 56, 56, 18, 24, 38, 31, 69, 71, 15, 96, 84, 77, 15, 94,
73, 69, 87, 31, 49, 69, 87, 43, 37, 47, 65, 85, 6, 42, 56, 38, 51, 30,
97, 24, 64, 94, 71, 81, 39, 82, 72, 37, 71, 57, },
          { 84, 16, 52, 28, 11, 50, 25, 3, 53, 96, 32, 56, 69, 9, 7, 72,
7, 67, 95, 14, 60, 53, 43, 39, 58, 61, 47, 92, 61, 44, 95, 47, 83, 94,
59, 81, 98, 0, 88, 48, 86, 41, 0, 86, 18, 72, 94, 52, 97, 58, 91, 74,
88, 58, 18, 97, 11, 43, 47, 87, 19, 19, 49, 10, 22, 54, 32, 4, 18, 13,
44, 36, 59, 91, 79, 93, 12, 62, 93, 43, 27, 78, 52, 38, 41, 6, 75, 81,
44, 45, 73, 48, 32, 22, 26, 76, 84, 35, 75, 58, },
          { 8, 65, 61, 96, 99, 64, 81, 15, 19, 9, 86, 78, 72, 37, 42,
29, 72, 31, 7, 79, 47, 64, 4, 37, 40, 65, 71, 79, 57, 38, 56, 83, 59,
41, 45, 89, 46, 88, 0, 96, 44, 87, 57, 30, 18, 45, 24, 49, 75, 31, 59,
11, 66, 76, 61, 47, 70, 59, 35, 42, 30, 34, 83, 75, 26, 27, 78, 24, 26,
99, 48, 40, 19, 52, 30, 62, 15, 54, 3, 66, 58, 82, 38, 97, 41, 31, 98,
71, 65, 65, 84, 76, 45, 21, 10, 85, 89, 25, 77, 98, },
          { 64, 1, 41, 62, 21, 86, 85, 81, 30, 30, 87, 33, 44, 38, 87,
31, 23, 49, 91, 76, 31, 3, 52, 34, 51, 35, 56, 79, 69, 9, 36, 53, 40,
12, 87, 29, 49, 48, 96, 0, 98, 16, 81, 46, 12, 31, 17, 7, 97, 47, 83,
66, 13, 34, 49, 32, 19, 27, 56, 67, 86, 93, 64, 84, 40, 18, 67, 28, 47,
22, 35, 58, 57, 71, 12, 90, 58, 22, 70, 13, 66, 26, 45, 1, 56, 88, 34,
16, 86, 56, 94, 20, 76, 8, 82, 79, 38, 95, 30, 93, },
          { 7, 2, 13, 21, 5, 73, 73, 46, 81, 58, 44, 90, 58, 9, 50, 91,
95, 19, 46, 29, 30, 47, 41, 90, 95, 90, 87, 65, 93, 55, 70, 59, 79, 70,
51, 2, 23, 86, 44, 98, 0, 94, 76, 74, 23, 14, 90, 65, 77, 32, 2, 96,
46, 91, 57, 17, 74, 10, 93, 96, 86, 83, 45, 60, 99, 23, 24, 98, 94, 36,
22, 47, 4, 82, 38, 73, 74, 96, 47, 75, 14, 60, 25, 61, 44, 25, 96, 32,
66, 13, 93, 93, 19, 28, 39, 44, 46, 73, 16, 38, },
          { 70, 25, 94, 6, 87, 4, 66, 92, 7, 64, 30, 10, 8, 67, 95, 10,
44, 61, 19, 65, 58, 67, 34, 95, 96, 63, 36, 39, 21, 84, 92, 10, 66, 56,
49, 16, 60, 41, 87, 16, 94, 0, 37, 23, 3, 1, 61, 47, 19, 79, 70, 70,
62, 55, 3, 50, 47, 7, 87, 34, 40, 96, 91, 60, 16, 71, 69, 52, 13, 55,
83, 84, 57, 22, 86, 33, 46, 96, 10, 61, 87, 48, 93, 0, 20, 91, 68, 49,
72, 99, 99, 60, 5, 90, 71, 67, 71, 14, 15, 70, },
          { 85, 0, 18, 81, 45, 58, 94, 69, 96, 92, 52, 55, 87, 14, 60,
89, 7, 87, 16, 36, 72, 94, 6, 55, 46, 84, 10, 36, 18, 34, 26, 17, 93,
74, 2, 60, 73, 0, 57, 81, 76, 37, 0, 99, 35, 45, 87, 76, 26, 95, 14,
52, 71, 28, 3, 78, 12, 19, 24, 18, 58, 99, 24, 28, 96, 75, 8, 26, 92,
43, 57, 84, 21, 49, 71, 50, 80, 44, 87, 50, 66, 36, 29, 22, 33, 44, 67,
43, 65, 8, 59, 33, 66, 24, 70, 64, 46, 29, 53, 85, },
        { 14, 78, 5, 24, 92, 22, 48, 30, 47, 94, 18, 53, 27, 28, 93,
64, 7, 30, 70, 57, 56, 51, 30, 77, 57, 25, 50, 62, 55, 35, 75, 13, 97,
27, 74, 45, 22, 86, 30, 46, 74, 23, 99, 0, 30, 87, 73, 48, 10, 82, 33,
17, 3, 49, 80, 7, 5, 58, 74, 7, 21, 75, 84, 13, 79, 26, 33, 19, 24, 7,
43, 76, 11, 69, 77, 64, 52, 76, 20, 5, 45, 71, 68, 63, 11, 0, 95, 69,
87, 80, 31, 62, 94, 95, 70, 93, 47, 88, 31, 29, },
        { 63, 25, 86, 78, 20, 11, 70, 57, 26, 88, 8, 27, 42, 22, 10,
51, 56, 31, 49, 5, 27, 33, 36, 71, 66, 9, 39, 55, 22, 89, 1, 36, 83,
47, 9, 9, 2, 18, 18, 12, 23, 3, 35, 30, 0, 38, 68, 17, 51, 8, 0, 28,
17, 29, 56, 23, 51, 87, 79, 84, 90, 10, 36, 68, 10, 99, 96, 37, 42, 42,
25, 10, 76, 44, 25, 83, 56, 24, 34, 63, 45, 86, 44, 8, 76, 78, 51, 23,
64, 74, 55, 80, 1, 37, 61, 37, 42, 48, 21, 85, },
        { 30, 80, 40, 43, 7, 7, 48, 42, 8, 66, 22, 4, 48, 69, 67, 64,
18, 8, 37, 88, 84, 1, 77, 75, 35, 20, 57, 66, 62, 62, 24, 71, 42, 6,
84, 2, 73, 72, 45, 31, 14, 1, 45, 87, 38, 0, 14, 19, 47, 50, 91, 21,
80, 62, 7, 55, 8, 63, 80, 17, 95, 80, 78, 10, 88, 81, 63, 21, 41, 62,
82, 18, 26, 57, 26, 77, 47, 44, 94, 11, 47, 29, 64, 15, 27, 15, 99, 26,
63, 33, 65, 19, 91, 4, 11, 6, 52, 25, 80, 49, },
        { 71, 25, 27, 13, 96, 54, 3, 29, 63, 76, 57, 87, 25, 0, 65,
10, 73, 88, 78, 15, 70, 12, 13, 49, 55, 64, 73, 13, 83, 23, 22, 2, 77,
89, 23, 67, 26, 94, 24, 17, 90, 61, 87, 73, 68, 14, 0, 4, 83, 9, 43,
36, 46, 9, 91, 33, 5, 15, 21, 80, 27, 14, 93, 14, 19, 75, 49, 72, 87,
49, 3, 17, 1, 98, 15, 66, 6, 75, 37, 90, 10, 80, 75, 90, 60, 84, 25,
19, 61, 14, 76, 60, 42, 69, 15, 32, 55, 54, 73, 48, },
        { 15, 59, 2, 64, 2, 7, 64, 25, 91, 15, 79, 66, 24, 46, 94, 78,
35, 22, 47, 9, 27, 81, 18, 74, 79, 71, 85, 27, 97, 10, 66, 60, 40, 47,
38, 6, 25, 52, 49, 7, 65, 47, 76, 48, 17, 19, 4, 0, 18, 92, 45, 26, 31,
94, 81, 90, 94, 56, 0, 6, 17, 77, 81, 34, 80, 78, 19, 60, 97, 46, 71,
76, 89, 9, 25, 7, 18, 0, 21, 19, 2, 64, 79, 92, 10, 14, 88, 52, 83, 41,
88, 30, 1, 44, 87, 51, 36, 59, 32, 94, },
        { 67, 67, 22, 87, 73, 25, 76, 28, 42, 63, 58, 4, 20, 75, 34,
7, 80, 65, 71, 67, 3, 70, 16, 81, 69, 36, 37, 27, 90, 33, 74, 54, 89,
91, 20, 56, 0, 97, 75, 97, 77, 19, 26, 10, 51, 47, 83, 18, 0, 17, 24,
25, 34, 31, 34, 66, 22, 40, 15, 65, 23, 35, 67, 20, 87, 43, 84, 38, 1,
38, 33, 20, 37, 19, 98, 55, 67, 52, 76, 73, 24, 91, 81, 73, 21, 22, 52,
17, 61, 50, 65, 14, 33, 84, 87, 13, 74, 83, 20, 50, },
        { 35, 85, 84, 88, 53, 92, 40, 55, 51, 1, 7, 10, 3, 54, 31, 28,
95, 40, 94, 12, 62, 82, 88, 98, 64, 59, 5, 72, 94, 64, 56, 76, 76, 29,
36, 38, 98, 58, 31, 47, 32, 79, 95, 82, 8, 50, 9, 92, 17, 0, 85, 50,
29, 30, 58, 64, 62, 96, 87, 12, 20, 62, 34, 26, 64, 72, 96, 90, 4, 62,
73, 47, 17, 33, 0, 76, 63, 98, 41, 16, 25, 82, 54, 32, 46, 48, 61, 46,
20, 95, 54, 58, 5, 11, 24, 66, 43, 43, 6, 34, },
        { 51, 78, 88, 42, 88, 88, 5, 94, 77, 6, 31, 16, 76, 61, 42,
28, 50, 50, 16, 31, 75, 63, 51, 52, 43, 16, 71, 28, 87, 8, 75, 80, 79,
64, 80, 40, 16, 91, 59, 83, 2, 70, 14, 33, 0, 91, 43, 45, 24, 85, 0,
98, 79, 72, 11, 20, 91, 31, 99, 92, 11, 44, 75, 21, 81, 89, 27, 76, 3,
60, 17, 3, 65, 33, 10, 86, 35, 77, 16, 52, 31, 29, 51, 81, 53, 52, 24,
89, 20, 78, 37, 44, 4, 86, 71, 97, 59, 52, 49, 70, },
        { 20, 39, 90, 15, 80, 2, 75, 22, 71, 50, 73, 42, 10, 55, 20,
5, 60, 44, 34, 25, 98, 62, 35, 54, 37, 94, 22, 74, 86, 76, 76, 3, 11,
58, 25, 90, 90, 74, 11, 66, 96, 70, 52, 17, 28, 21, 36, 26, 25, 50, 98,
0, 46, 40, 7, 96, 34, 23, 7, 90, 84, 41, 84, 56, 88, 35, 87, 33, 16,
31, 30, 35, 16, 68, 29, 18, 92, 17, 55, 89, 91, 71, 43, 38, 9, 19, 92,
70, 46, 87, 81, 80, 4, 46, 59, 34, 93, 81, 12, 79, },
        { 94, 29, 52, 27, 1, 10, 9, 32, 40, 27, 33, 49, 46, 32, 99,
14, 47, 48, 76, 51, 71, 59, 70, 71, 25, 51, 24, 58, 77, 41, 47, 7, 45,
90, 84, 14, 15, 88, 66, 13, 46, 62, 71, 3, 17, 80, 46, 31, 34, 29, 79,
46, 0, 16, 22, 11, 5, 55, 42, 48, 11, 20, 55, 4, 75, 41, 36, 81, 82,
75, 29, 87, 41, 93, 16, 48, 38, 18, 41, 9, 99, 81, 24, 24, 33, 1, 25,
94, 55, 33, 34, 8, 20, 66, 42, 30, 11, 7, 30, 66, },
        { 63, 75, 53, 8, 23, 85, 75, 89, 28, 33, 61, 37, 18, 97, 69,
59, 6, 62, 53, 43, 22, 70, 76, 81, 68, 21, 91, 51, 32, 32, 6, 79, 69,
18, 93, 16, 27, 58, 76, 34, 91, 55, 28, 49, 29, 62, 9, 94, 31, 30, 72,
40, 16, 0, 48, 88, 18, 61, 52, 67, 10, 29, 80, 35, 52, 99, 35, 10, 63,
68, 69, 39, 46, 77, 42, 37, 66, 98, 63, 57, 16, 15, 75, 1, 85, 76, 9,
74, 86, 80, 83, 49, 90, 95, 88, 87, 78, 44, 60, 43, },
        { 80, 20, 20, 85, 35, 64, 35, 49, 52, 27, 74, 25, 81, 97, 60,
95, 32, 67, 40, 63, 64, 61, 70, 95, 16, 97, 37, 23, 62, 90, 45, 5, 91,
56, 28, 96, 87, 18, 61, 49, 57, 3, 3, 80, 56, 7, 91, 81, 34, 58, 11, 7,
22, 48, 0, 4, 95, 10, 55, 43, 83, 50, 38, 65, 39, 27, 16, 10, 74, 47,
44, 56, 27, 65, 98, 40, 71, 80, 6, 31, 37, 74, 83, 94, 78, 16, 66, 53,
33, 44, 98, 90, 41, 30, 8, 18, 66, 44, 35, 80, },
        { 20, 92, 70, 60, 74, 13, 55, 87, 95, 91, 50, 44, 21, 56, 12,
0, 7, 24, 32, 56, 18, 38, 44, 46, 37, 47, 29, 45, 28, 16, 99, 70, 78,
14, 54, 29, 44, 97, 47, 32, 17, 50, 78, 7, 23, 55, 33, 90, 66, 64, 20,
96, 11, 88, 4, 0, 35, 11, 93, 26, 60, 79, 39, 72, 15, 17, 24, 21, 60,
64, 96, 97, 53, 85, 18, 28, 89, 69, 27, 77, 89, 72, 64, 38, 34, 89, 71,
52, 55, 81, 87, 34, 54, 28, 93, 72, 64, 76, 67, 49, },
        { 70, 89, 8, 15, 48, 95, 22, 67, 82, 29, 46, 83, 36, 59, 96,
13, 4, 5, 72, 36, 44, 51, 16, 4, 14, 53, 80, 26, 54, 87, 62, 23, 84,
80, 65, 76, 56, 11, 70, 19, 74, 47, 12, 5, 51, 8, 5, 94, 22, 62, 91,
34, 5, 18, 95, 35, 0, 2, 11, 70, 64, 64, 24, 59, 2, 75, 60, 31, 25, 4,
3, 34, 64, 49, 0, 64, 99, 53, 50, 27, 29, 67, 82, 1, 52, 82, 56, 53, 9,
80, 29, 55, 39, 94, 35, 53, 69, 59, 60, 67, },
        { 14, 70, 39, 96, 84, 97, 30, 87, 4, 45, 16, 20, 47, 0, 76,
33, 57, 18, 30, 87, 20, 91, 82, 25, 62, 45, 82, 95, 47, 47, 78, 16, 39,
93, 5, 9, 56, 43, 59, 27, 10, 7, 19, 58, 87, 63, 15, 56, 40, 96, 31,
23, 55, 61, 10, 11, 2, 0, 29, 53, 5, 54, 81, 30, 33, 87, 43, 68, 72,
68, 48, 95, 10, 1, 10, 11, 9, 63, 38, 57, 56, 64, 50, 84, 48, 80, 34,
41, 99, 22, 30, 75, 59, 83, 67, 86, 29, 16, 74, 48, },
        { 79, 62, 69, 68, 70, 78, 40, 19, 91, 32, 88, 53, 23, 17, 17,
96, 35, 97, 52, 0, 58, 52, 15, 18, 89, 38, 88, 85, 21, 20, 84, 24, 69,
91, 88, 9, 18, 47, 35, 56, 93, 87, 24, 74, 79, 80, 21, 0, 15, 87, 99,
7, 42, 52, 55, 93, 11, 29, 0, 60, 50, 16, 67, 13, 4, 4, 63, 90, 69, 76,
41, 38, 60, 40, 40, 39, 76, 28, 52, 33, 9, 7, 86, 18, 44, 69, 27, 18,
25, 8, 57, 13, 43, 82, 50, 72, 0, 63, 37, 48, },
        { 19, 58, 77, 81, 7, 29, 84, 58, 65, 36, 96, 34, 93, 18, 20,
53, 82, 19, 79, 86, 65, 41, 62, 98, 87, 55, 81, 25, 29, 26, 88, 96, 2,
87, 72, 46, 24, 87, 42, 67, 96, 34, 18, 7, 84, 17, 80, 6, 65, 12, 92,
90, 48, 67, 43, 26, 70, 53, 60, 0, 92, 49, 1, 42, 60, 38, 46, 69, 89,
51, 83, 16, 76, 97, 3, 4, 0, 95, 92, 37, 8, 54, 12, 68, 24, 62, 95, 34,
83, 16, 59, 30, 80, 77, 12, 91, 71, 38, 17, 47, },
        { 55, 23, 43, 66, 75, 12, 23, 50, 6, 77, 26, 92, 12, 76, 43,
24, 29, 72, 41, 12, 50, 44, 79, 66, 14, 24, 56, 94, 51, 88, 21, 76, 21,
14, 99, 47, 38, 19, 30, 86, 86, 40, 58, 21, 90, 95, 27, 17, 23, 20, 11,
84, 11, 10, 83, 60, 64, 5, 50, 92, 0, 73, 34, 34, 89, 49, 37, 2, 22,
37, 37, 11, 85, 28, 22, 45, 77, 73, 26, 96, 13, 1, 50, 51, 15, 15, 11,
28, 24, 52, 18, 70, 17, 93, 35, 91, 96, 97, 34, 97, },
        { 81, 87, 91, 15, 61, 15, 80, 53, 28, 12, 41, 32, 72, 61, 86,
40, 55, 20, 61, 74, 42, 8, 44, 64, 32, 39, 53, 24, 41, 64, 84, 40, 22,
83, 87, 44, 31, 19, 34, 93, 83, 96, 99, 75, 10, 80, 14, 77, 35, 62, 44,
41, 20, 29, 50, 79, 64, 54, 16, 49, 73, 0, 12, 90, 20, 83, 90, 81, 42,
83, 9, 2, 97, 79, 23, 40, 92, 20, 71, 44, 94, 70, 85, 89, 74, 23, 78,
67, 48, 24, 31, 19, 58, 43, 18, 82, 73, 85, 11, 1, },
        { 11, 27, 29, 63, 86, 73, 19, 70, 77, 63, 51, 87, 95, 44, 28,
40, 22, 8, 6, 40, 67, 20, 4, 19, 59, 62, 55, 75, 31, 22, 42, 34, 8, 95,
24, 60, 69, 49, 83, 64, 45, 91, 24, 84, 36, 78, 93, 81, 67, 34, 75, 84,
55, 80, 38, 39, 24, 81, 67, 1, 34, 12, 0, 16, 65, 75, 99, 79, 16, 27,
20, 12, 74, 37, 33, 39, 54, 31, 15, 84, 16, 19, 24, 50, 16, 3, 65, 64,
65, 78, 92, 24, 4, 64, 73, 96, 70, 83, 8, 11, },
        { 12, 46, 16, 51, 64, 70, 92, 83, 45, 57, 87, 43, 53, 47, 46,
22, 37, 39, 14, 43, 84, 35, 89, 80, 72, 59, 29, 87, 1, 76, 43, 16, 69,
63, 4, 83, 71, 10, 75, 84, 60, 60, 28, 13, 68, 10, 14, 34, 20, 26, 21,
56, 4, 35, 65, 72, 59, 30, 13, 42, 34, 90, 16, 0, 65, 25, 27, 57, 68,
35, 49, 47, 28, 70, 33, 96, 61, 82, 96, 65, 15, 25, 70, 31, 11, 53, 53,
27, 93, 65, 87, 78, 84, 9, 22, 47, 30, 52, 62, 7, },
        { 18, 53, 94, 66, 54, 38, 98, 52, 79, 12, 14, 76, 64, 38, 69,
72, 51, 87, 0, 84, 14, 27, 96, 3, 88, 64, 92, 71, 17, 14, 52, 66, 18,
33, 77, 62, 15, 22, 26, 40, 99, 16, 96, 79, 10, 88, 19, 80, 87, 64, 81,
88, 75, 52, 39, 15, 2, 33, 4, 60, 89, 20, 65, 65, 0, 64, 0, 66, 33, 46,
21, 83, 39, 13, 57, 22, 51, 4, 37, 20, 34, 33, 66, 61, 1, 57, 6, 1, 34,
82, 4, 14, 57, 50, 12, 20, 56, 30, 35, 94, },
        { 44, 78, 84, 9, 57, 31, 22, 16, 53, 37, 20, 66, 96, 60, 84,
42, 23, 51, 66, 36, 45, 45, 73, 3, 76, 16, 17, 63, 8, 28, 15, 83, 44,
79, 46, 5, 96, 54, 27, 18, 23, 71, 75, 26, 99, 81, 75, 78, 43, 72, 89,
35, 41, 99, 27, 17, 75, 87, 4, 38, 49, 83, 75, 25, 64, 0, 92, 63, 0,
21, 62, 68, 87, 91, 20, 8, 44, 45, 38, 90, 74, 27, 12, 53, 55, 46, 56,
36, 21, 80, 55, 17, 28, 94, 18, 50, 1, 34, 80, 50, },
        { 75, 76, 51, 31, 42, 1, 17, 36, 64, 9, 44, 69, 89, 1, 77, 38,
83, 76, 25, 82, 62, 84, 85, 26, 70, 39, 37, 25, 23, 87, 47, 26, 52, 72,
12, 82, 84, 32, 78, 67, 24, 69, 8, 33, 96, 63, 49, 19, 84, 96, 27, 87,
36, 35, 16, 24, 60, 43, 63, 46, 37, 90, 99, 27, 0, 92, 0, 32, 28, 97,
85, 18, 4, 78, 17, 85, 31, 15, 77, 66, 41, 60, 30, 77, 39, 35, 55, 59,
19, 53, 8, 27, 99, 59, 67, 18, 61, 27, 45, 85, },
        { 41, 44, 97, 39, 61, 36, 45, 33, 68, 9, 14, 59, 30, 77, 10,
93, 34, 95, 89, 83, 15, 47, 90, 32, 32, 11, 27, 97, 9, 59, 61, 60, 91,
29, 70, 14, 77, 4, 24, 28, 98, 52, 26, 19, 37, 21, 72, 60, 38, 90, 76,
33, 81, 10, 10, 21, 31, 68, 90, 69, 2, 81, 79, 57, 66, 63, 32, 0, 46,
43, 37, 53, 56, 89, 37, 26, 62, 88, 80, 80, 63, 9, 81, 87, 34, 62, 43,
50, 28, 41, 51, 43, 27, 54, 53, 4, 29, 64, 18, 35, },
        { 74, 5, 77, 18, 70, 7, 4, 66, 27, 27, 50, 50, 22, 25, 82, 43,
9, 77, 74, 86, 80, 87, 92, 42, 61, 40, 23, 5, 59, 10, 23, 72, 43, 39,
67, 40, 15, 18, 26, 47, 94, 13, 92, 24, 42, 41, 87, 97, 1, 4, 3, 16,
82, 63, 74, 60, 25, 72, 69, 89, 22, 42, 16, 68, 33, 0, 28, 46, 0, 34,
86, 80, 49, 51, 67, 66, 75, 83, 3, 52, 48, 47, 28, 27, 15, 93, 46, 83,
54, 27, 86, 14, 46, 24, 67, 96, 26, 99, 93, 67, },
        { 90, 23, 53, 2, 38, 68, 44, 91, 4, 25, 78, 13, 66, 97, 1, 69,
85, 21, 76, 89, 81, 26, 49, 63, 72, 55, 7, 21, 87, 24, 0, 20, 50, 45,
41, 77, 94, 13, 99, 22, 36, 55, 43, 7, 42, 62, 49, 46, 38, 62, 60, 31,
75, 68, 47, 64, 4, 68, 76, 51, 37, 83, 27, 35, 46, 21, 97, 43, 34, 0,
47, 55, 27, 53, 27, 40, 91, 98, 27, 90, 19, 22, 72, 79, 68, 22, 15, 81,
75, 50, 87, 13, 65, 21, 96, 92, 56, 21, 89, 99, },
        { 8, 93, 33, 78, 5, 21, 39, 74, 67, 17, 55, 8, 65, 64, 92, 59,
1, 5, 35, 51, 77, 61, 83, 25, 3, 77, 53, 42, 14, 71, 42, 41, 51, 29,
55, 54, 73, 44, 48, 35, 22, 83, 57, 43, 25, 82, 3, 71, 33, 73, 17, 30,
29, 69, 44, 96, 3, 48, 41, 83, 37, 9, 20, 49, 21, 62, 85, 37, 86, 47,
0, 62, 37, 90, 80, 53, 99, 10, 81, 48, 41, 28, 49, 29, 18, 12, 44, 54,
90, 81, 85, 80, 44, 73, 26, 14, 82, 25, 16, 8, },
        { 78, 40, 42, 43, 9, 82, 34, 41, 59, 94, 16, 51, 52, 72, 84,
99, 10, 71, 21, 73, 32, 73, 65, 33, 49, 63, 11, 81, 37, 38, 70, 9, 40,
52, 11, 98, 69, 36, 40, 58, 47, 84, 84, 76, 10, 18, 17, 76, 20, 47, 3,
35, 87, 39, 56, 97, 34, 95, 38, 16, 11, 2, 12, 47, 83, 68, 18, 53, 80,
55, 62, 0, 25, 42, 0, 80, 68, 71, 27, 19, 16, 93, 24, 95, 76, 72, 28,
2, 27, 12, 27, 11, 77, 97, 65, 45, 51, 21, 56, 11, },
          { 93, 77, 83, 75, 30, 2, 49, 50, 70, 33, 13, 84, 99, 57, 21,
38, 63, 21, 70, 80, 91, 7, 57, 55, 0, 66, 76, 92, 43, 0, 62, 15, 0, 52,
5, 86, 87, 59, 19, 57, 4, 57, 21, 11, 76, 26, 1, 89, 37, 17, 65, 16,
41, 46, 27, 53, 64, 10, 60, 76, 85, 97, 74, 28, 39, 87, 4, 56, 49, 27,
37, 25, 0, 66, 41, 18, 94, 51, 10, 24, 50, 59, 97, 15, 65, 24, 25, 28,
52, 31, 14, 70, 53, 69, 42, 22, 10, 36, 13, 24, },
          { 2, 80, 51, 44, 65, 59, 46, 34, 80, 92, 63, 23, 27, 42, 4,
37, 38, 66, 50, 51, 99, 15, 50, 92, 0, 43, 96, 94, 70, 0, 22, 58, 70,
28, 6, 42, 31, 91, 52, 71, 82, 22, 49, 69, 44, 57, 98, 9, 19, 33, 33,
68, 93, 77, 65, 85, 49, 1, 40, 97, 28, 79, 37, 70, 13, 91, 78, 89, 51,
53, 90, 42, 66, 0, 23, 28, 26, 62, 72, 82, 57, 34, 49, 76, 52, 74, 27,
66, 60, 51, 49, 70, 28, 92, 19, 77, 12, 79, 40, 5, },
          { 75, 73, 83, 2, 40, 69, 11, 42, 86, 9, 2, 93, 48, 37, 45, 43,
28, 86, 54, 58, 27, 48, 59, 47, 35, 83, 93, 38, 9, 6, 87, 39, 24, 5,
12, 10, 49, 79, 30, 12, 38, 86, 71, 77, 25, 26, 15, 25, 98, 0, 10, 29,
16, 42, 98, 18, 0, 10, 40, 3, 22, 23, 33, 33, 57, 20, 17, 37, 67, 27,
80, 0, 41, 23, 0, 72, 79, 39, 31, 43, 99, 94, 15, 98, 26, 87, 99, 45,
13, 78, 24, 44, 91, 6, 24, 49, 38, 13, 44, 37, },
          { 83, 50, 65, 97, 61, 42, 68, 64, 54, 36, 14, 20, 9, 6, 30,
80, 96, 63, 68, 69, 80, 14, 86, 20, 86, 25, 15, 88, 51, 48, 10, 97, 16,
77, 69, 15, 69, 93, 62, 90, 73, 33, 50, 64, 83, 77, 66, 7, 55, 76, 86,
18, 48, 37, 40, 28, 64, 11, 39, 4, 45, 40, 39, 96, 22, 8, 85, 26, 66,
40, 53, 80, 18, 28, 72, 0, 22, 12, 61, 0, 35, 76, 62, 62, 73, 99, 75,
85, 89, 34, 66, 8, 23, 70, 77, 19, 52, 91, 15, 88, },
          { 88, 94, 0, 30, 35, 17, 31, 16, 60, 32, 2, 40, 44, 16, 30,
14, 24, 7, 56, 81, 0, 8, 74, 98, 46, 71, 25, 21, 79, 42, 62, 54, 49,
21, 36, 15, 87, 12, 15, 58, 74, 46, 80, 52, 56, 47, 6, 18, 67, 63, 35,
92, 38, 66, 71, 89, 99, 9, 76, 0, 77, 92, 54, 61, 51, 44, 31, 62, 75,
91, 99, 68, 94, 26, 79, 22, 0, 32, 26, 85, 86, 5, 70, 33, 57, 13, 35,
51, 72, 34, 79, 90, 88, 97, 29, 82, 73, 98, 21, 16, },
          { 63, 1, 90, 25, 50, 38, 16, 78, 77, 75, 52, 31, 68, 57, 18,
16, 81, 92, 27, 64, 62, 20, 75, 25, 99, 93, 49, 10, 68, 12, 77, 47, 51,
6, 38, 22, 43, 62, 54, 22, 96, 96, 44, 76, 24, 44, 75, 0, 52, 98, 77,
17, 18, 98, 80, 69, 53, 63, 28, 95, 73, 20, 31, 82, 4, 45, 15, 88, 83,
98, 10, 71, 51, 62, 39, 12, 32, 0, 73, 23, 35, 36, 39, 39, 56, 25, 44,
78, 13, 93, 85, 73, 16, 87, 60, 64, 65, 15, 8, 56, },
          { 38, 20, 46, 7, 82, 0, 43, 88, 72, 34, 4, 48, 0, 15, 74, 60,
91, 23, 3, 89, 53, 11, 78, 3, 11, 19, 66, 52, 34, 12, 72, 72, 77, 99,
4, 27, 37, 93, 3, 70, 47, 10, 87, 20, 34, 94, 37, 21, 76, 41, 16, 55,
41, 63, 6, 27, 50, 38, 52, 92, 26, 71, 15, 96, 37, 38, 77, 80, 3, 27,
81, 27, 10, 72, 31, 61, 26, 73, 0, 36, 42, 85, 76, 52, 48, 51, 7, 53,
58, 48, 13, 2, 44, 36, 38, 79, 82, 89, 63, 28, },
          { 16, 80, 26, 45, 51, 13, 90, 2, 65, 33, 67, 24, 39, 76, 55,
93, 21, 79, 52, 92, 86, 41, 16, 19, 14, 54, 33, 40, 76, 33, 54, 89, 72,
41, 8, 18, 47, 43, 66, 13, 75, 61, 50, 5, 63, 11, 90, 19, 73, 16, 52,
89, 9, 57, 31, 77, 27, 57, 33, 37, 96, 44, 84, 65, 20, 90, 66, 80, 52,
90, 48, 19, 24, 82, 43, 0, 85, 23, 36, 0, 79, 43, 32, 67, 22, 98, 16,
72, 22, 9, 49, 3, 74, 15, 16, 43, 66, 69, 68, 4, },
          { 41, 48, 15, 9, 44, 11, 79, 10, 76, 85, 21, 37, 39, 29, 61,
65, 62, 95, 58, 31, 20, 27, 53, 77, 4, 52, 17, 58, 77, 72, 34, 96, 76,
28, 78, 25, 65, 27, 58, 66, 14, 87, 66, 45, 45, 47, 10, 2, 24, 25, 31,
91, 99, 16, 37, 89, 29, 56, 9, 8, 13, 94, 16, 15, 34, 74, 41, 63, 48,
19, 41, 16, 50, 57, 99, 35, 86, 35, 42, 79, 0, 60, 62, 50, 25, 26, 72,
89, 16, 79, 97, 29, 62, 76, 45, 53, 23, 43, 36, 8, },
          { 4, 63, 68, 13, 0, 28, 88, 98, 68, 32, 3, 0, 45, 72, 37, 14,
79, 29, 99, 34, 42, 50, 94, 6, 65, 1, 26, 80, 38, 97, 12, 18, 51, 22,
69, 72, 85, 78, 82, 26, 60, 48, 36, 71, 86, 29, 80, 64, 91, 82, 29, 71,
81, 15, 74, 72, 67, 64, 7, 54, 1, 70, 19, 25, 33, 27, 60, 9, 47, 22,
28, 93, 59, 34, 94, 76, 5, 36, 85, 43, 60, 0, 58, 65, 67, 49, 16, 75,
68, 67, 34, 48, 7, 28, 97, 99, 62, 32, 45, 7, },
          { 33, 97, 40, 33, 70, 35, 51, 51, 11, 3, 83, 22, 91, 12, 21,
96, 55, 96, 30, 91, 34, 74, 67, 98, 21, 16, 6, 62, 13, 43, 58, 49, 15,
55, 50, 56, 6, 52, 38, 45, 25, 93, 29, 68, 44, 64, 75, 79, 81, 54, 51,
43, 24, 75, 83, 64, 82, 50, 86, 12, 50, 85, 24, 70, 66, 12, 30, 81, 28,
72, 49, 24, 97, 49, 15, 62, 70, 39, 76, 32, 62, 58, 0, 21, 34, 99, 17,
51, 8, 41, 68, 4, 20, 54, 86, 24, 71, 21, 18, 60, },
          { 55, 20, 10, 38, 59, 63, 79, 78, 90, 64, 55, 92, 33, 20, 62,
69, 9, 47, 68, 83, 38, 18, 36, 18, 43, 89, 7, 58, 75, 34, 16, 88, 12,
14, 89, 92, 42, 38, 97, 1, 61, 0, 22, 63, 8, 15, 90, 92, 73, 32, 81,
38, 24, 1, 94, 38, 1, 84, 18, 68, 51, 89, 50, 31, 61, 53, 77, 87, 27,
79, 29, 95, 15, 76, 98, 62, 33, 39, 52, 67, 50, 65, 21, 0, 18, 42, 26,
77, 20, 27, 90, 26, 75, 4, 17, 96, 98, 66, 75, 67, },
          { 59, 64, 96, 32, 36, 65, 44, 60, 33, 72, 8, 92, 57, 26, 87,
52, 85, 37, 45, 30, 94, 13, 34, 64, 53, 69, 2, 80, 60, 85, 60, 64, 5,
16, 79, 73, 56, 41, 41, 56, 44, 20, 33, 11, 76, 27, 60, 10, 21, 46, 53,
9, 33, 85, 78, 34, 52, 48, 44, 24, 15, 74, 16, 11, 1, 55, 39, 34, 15,
68, 18, 76, 65, 52, 26, 73, 57, 56, 48, 22, 25, 67, 34, 18, 0, 71, 95,
72, 72, 29, 58, 56, 21, 91, 15, 97, 81, 36, 49, 2, },
          { 82, 25, 40, 94, 85, 87, 36, 95, 85, 98, 57, 5, 83, 53, 70,
66, 35, 42, 43, 67, 40, 0, 60, 24, 10, 0, 75, 18, 57, 4, 50, 96, 96,
44, 21, 20, 38, 6, 31, 88, 25, 91, 44, 0, 78, 15, 84, 14, 22, 48, 52,
19, 1, 76, 16, 89, 82, 80, 69, 62, 15, 23, 3, 53, 57, 46, 35, 62, 93,
22, 12, 72, 24, 74, 87, 99, 13, 25, 51, 98, 26, 49, 99, 42, 71, 0, 64,
10, 44, 73, 67, 23, 49, 51, 39, 6, 71, 12, 8, 53, },
          { 95, 64, 92, 44, 35, 11, 83, 74, 37, 35, 50, 84, 35, 24, 27,
69, 60, 36, 34, 94, 88, 3, 56, 79, 24, 75, 50, 81, 2, 22, 39, 4, 57,
71, 8, 56, 51, 75, 98, 34, 96, 68, 67, 95, 51, 99, 25, 88, 52, 61, 24,
92, 25, 9, 66, 71, 56, 34, 27, 95, 11, 78, 65, 53, 6, 56, 55, 43, 46,
15, 44, 28, 25, 27, 99, 75, 35, 44, 7, 16, 72, 16, 17, 26, 95, 64, 0,
63, 66, 40, 30, 86, 87, 80, 45, 37, 58, 69, 80, 80, },
          { 58, 57, 68, 33, 7, 69, 23, 29, 14, 82, 2, 32, 89, 70, 91, 9,
83, 44, 86, 16, 68, 49, 94, 17, 51, 2, 81, 66, 68, 24, 5, 14, 73, 81,
66, 34, 30, 81, 71, 16, 32, 49, 43, 69, 23, 26, 19, 52, 17, 46, 89, 70,
94, 74, 53, 52, 53, 41, 18, 34, 28, 67, 64, 27, 1, 36, 59, 50, 83, 81,
54, 2, 28, 66, 45, 85, 51, 78, 53, 72, 89, 75, 51, 77, 72, 10, 63, 0,
27, 12, 68, 33, 97, 0, 58, 75, 31, 22, 23, 16, },
          { 76, 14, 86, 57, 37, 66, 46, 86, 2, 8, 88, 7, 79, 63, 16, 30,
67, 89, 18, 14, 6, 2, 86, 47, 88, 74, 17, 91, 48, 56, 7, 79, 29, 71,
31, 19, 97, 44, 65, 86, 66, 72, 65, 87, 64, 63, 61, 83, 61, 20, 20, 46,
55, 86, 33, 55, 9, 99, 25, 83, 24, 48, 65, 93, 34, 21, 19, 28, 54, 75,
90, 27, 52, 60, 13, 89, 72, 13, 58, 22, 16, 68, 8, 20, 72, 44, 66, 27,
0, 58, 8, 29, 68, 43, 73, 18, 6, 79, 68, 79, },
          { 29, 9, 90, 59, 98, 49, 62, 32, 73, 46, 33, 8, 3, 40, 65, 74,
26, 59, 93, 40, 66, 57, 51, 57, 83, 81, 83, 57, 2, 51, 35, 52, 44, 1,
51, 41, 24, 45, 65, 56, 13, 99, 8, 80, 74, 33, 14, 41, 50, 95, 78, 87,
33, 80, 44, 81, 80, 22, 8, 16, 52, 24, 78, 65, 82, 80, 53, 41, 27, 50,
81, 12, 31, 51, 78, 34, 34, 93, 48, 9, 79, 67, 41, 27, 29, 73, 40, 12,
58, 0, 77, 85, 74, 90, 15, 21, 10, 64, 10, 55, },
          { 1, 80, 67, 39, 8, 51, 64, 74, 70, 61, 24, 25, 25, 22, 97,
51, 73, 54, 64, 31, 49, 30, 50, 75, 13, 73, 33, 29, 82, 49, 77, 14, 60,
6, 61, 92, 64, 73, 84, 94, 93, 99, 59, 31, 55, 65, 76, 88, 65, 54, 37,
81, 34, 83, 98, 87, 29, 30, 57, 59, 18, 31, 92, 87, 4, 55, 8, 51, 86,
87, 85, 27, 14, 49, 24, 66, 79, 85, 13, 49, 97, 34, 68, 90, 58, 67, 30,
68, 8, 77, 0, 45, 36, 10, 59, 99, 94, 98, 20, 86, },
          { 3, 3, 41, 46, 79, 26, 84, 12, 9, 10, 69, 82, 79, 35, 15, 38,
92, 33, 23, 48, 76, 32, 78, 7, 20, 58, 18, 3, 91, 59, 90, 94, 34, 75,
1, 31, 94, 48, 76, 20, 93, 60, 33, 62, 80, 19, 60, 30, 14, 58, 44, 80,
8, 49, 90, 34, 55, 75, 13, 30, 70, 19, 24, 78, 14, 17, 27, 43, 14, 13,
80, 11, 70, 70, 44, 8, 90, 73, 2, 3, 29, 48, 4, 26, 56, 23, 86, 33, 29,
85, 45, 0, 37, 39, 43, 74, 76, 65, 71, 39, },
          { 72, 12, 90, 79, 89, 82, 32, 11, 56, 86, 55, 83, 0, 68, 75,
9, 67, 32, 65, 71, 31, 20, 61, 99, 86, 60, 24, 31, 45, 42, 46, 18, 38,
48, 37, 57, 71, 32, 45, 76, 19, 5, 66, 94, 1, 91, 42, 1, 33, 5, 4, 4,
20, 90, 41, 54, 39, 59, 43, 80, 17, 58, 4, 84, 57, 28, 99, 27, 46, 65,
44, 77, 53, 28, 91, 23, 88, 16, 44, 74, 62, 7, 20, 75, 21, 49, 87, 97,
68, 74, 36, 37, 0, 1, 13, 18, 56, 46, 25, 17, },
          { 22, 46, 45, 68, 7, 29, 85, 77, 70, 93, 74, 6, 49, 97, 2, 82,
81, 4, 81, 1, 37, 52, 39, 46, 26, 98, 58, 87, 25, 69, 48, 30, 9, 50,
15, 88, 81, 22, 21, 8, 28, 90, 24, 95, 37, 4, 69, 44, 84, 11, 86, 46,
66, 95, 30, 28, 94, 83, 82, 77, 93, 43, 64, 9, 50, 94, 59, 54, 24, 21,
73, 97, 69, 92, 6, 70, 97, 87, 36, 15, 76, 28, 54, 4, 91, 51, 80, 0,
43, 90, 10, 39, 1, 0, 45, 60, 53, 74, 22, 22, },
          { 22, 23, 17, 72, 31, 66, 8, 39, 59, 5, 37, 13, 31, 62, 32,
38, 4, 72, 80, 54, 88, 56, 51, 89, 44, 29, 58, 32, 68, 79, 73, 18, 58,
64, 81, 11, 39, 26, 10, 82, 39, 71, 70, 70, 61, 11, 15, 87, 87, 24, 71,
59, 42, 88, 8, 93, 35, 67, 50, 12, 35, 18, 73, 22, 12, 18, 67, 53, 67,
96, 26, 65, 42, 19, 24, 77, 29, 60, 38, 16, 45, 97, 86, 17, 15, 39, 45,
58, 73, 15, 59, 43, 13, 45, 0, 35, 35, 97, 91, 28, },
          { 21, 2, 52, 82, 6, 28, 32, 3, 11, 42, 30, 76, 36, 21, 55, 22,
78, 10, 84, 89, 79, 66, 12, 11, 35, 59, 47, 97, 10, 54, 3, 98, 36, 10,
29, 29, 82, 76, 85, 79, 44, 67, 64, 93, 37, 6, 32, 51, 13, 66, 97, 34,
30, 87, 18, 72, 53, 86, 72, 91, 91, 82, 96, 47, 20, 50, 18, 4, 96, 92,
14, 45, 22, 77, 49, 19, 82, 64, 79, 43, 53, 99, 24, 96, 97, 6, 37, 75,
18, 21, 99, 74, 18, 60, 35, 0, 76, 94, 82, 30, },
          { 7, 28, 97, 66, 76, 61, 93, 53, 3, 85, 97, 69, 89, 60, 59, 5,
59, 62, 13, 47, 67, 40, 3, 3, 31, 34, 93, 64, 12, 53, 42, 8, 34, 64,
27, 67, 72, 84, 89, 38, 46, 71, 46, 47, 42, 52, 55, 36, 74, 43, 59, 93,
11, 78, 66, 64, 69, 29, 0, 71, 96, 73, 70, 30, 56, 1, 61, 29, 26, 56,
82, 51, 10, 12, 38, 52, 73, 65, 82, 66, 23, 62, 71, 98, 81, 71, 58, 31,
6, 10, 94, 76, 56, 53, 35, 76, 0, 10, 76, 42, },
          { 20, 36, 98, 83, 84, 2, 42, 24, 8, 49, 15, 90, 43, 67, 35,
17, 50, 13, 20, 60, 22, 43, 39, 80, 83, 71, 84, 16, 55, 58, 8, 84, 48,
46, 59, 75, 37, 35, 25, 95, 73, 14, 29, 88, 48, 25, 54, 59, 83, 43, 52,
81, 7, 44, 44, 76, 59, 16, 63, 38, 97, 85, 83, 52, 30, 34, 27, 64, 99,
21, 25, 21, 36, 79, 13, 91, 98, 15, 89, 69, 43, 32, 21, 66, 36, 12, 69,
22, 79, 64, 98, 65, 46, 74, 97, 94, 10, 0, 37, 97, },
          { 24, 58, 4, 89, 8, 15, 72, 93, 46, 17, 78, 90, 27, 55, 55,
48, 24, 64, 64, 23, 17, 10, 41, 33, 36, 99, 51, 25, 56, 46, 50, 77, 85,
28, 86, 36, 71, 75, 77, 30, 16, 15, 53, 31, 21, 80, 73, 32, 20, 6, 49,
12, 30, 60, 35, 67, 60, 74, 37, 17, 34, 11, 8, 62, 35, 80, 45, 18, 93,
89, 16, 56, 13, 40, 44, 15, 21, 8, 63, 68, 36, 45, 18, 75, 49, 8, 80,
23, 68, 10, 20, 71, 25, 22, 91, 82, 76, 37, 0, 99, },
          { 80, 29, 61, 73, 55, 26, 40, 23, 99, 12, 98, 32, 81, 36, 68,
74, 78, 30, 34, 68, 81, 93, 75, 57, 64, 56, 88, 53, 75, 42, 51, 77, 27,
9, 86, 59, 57, 58, 98, 93, 38, 70, 85, 29, 85, 49, 48, 94, 50, 34, 70,
79, 66, 43, 80, 49, 67, 48, 48, 47, 97, 1, 11, 7, 94, 50, 85, 35, 67,
99, 8, 11, 24, 5, 37, 88, 16, 56, 28, 4, 8, 7, 60, 67, 2, 53, 80, 16,
79, 55, 86, 39, 17, 22, 28, 30, 42, 97, 99, 0, }
```

```
      };



 public random(int s)
  {
   size = s;
   A = new int [s];
   for (int i=0; i<s; i++) { A[i] = i; }
  }

/* public random(random P)
  {
   this.size = P.size;
   for(int i=0; i < this.size; i++) { this.A[i] = P.A[i];}
  }
*/

 public double fitness()
  {
            // Calculate the fitness of the individual
            // given x, find f(x)
                int dist, dist_a, dist_b;
                double accum =0;
                for (int i=0; i<this.size;i++)
                {
                  for (int j=i+1; j<this.size;j++)
                   {
                    dist_a = A[i] - A[j];
                    if (dist_a < 0) { dist_a += this.size; }
                    dist_b = A[j] - A[i];
                    if (dist_b < 0) { dist_b += this.size; }

                    dist = Math.min(dist_a, dist_b);

                    accum+= dist * F[i][j];
                   }
                }
                return (10000000 - accum); //Math.pow(gene.Value(), 2);


  }

public void GetRandomString()
 {
  int x,y, t;
  // random shuffle
  for (int c=0; c<(size*2); c++)
   {
     x = 0 + (int) (Math.random() * (size));
     y = 0 + (int) (Math.random() * (size));
     t = A[x];
     A[x] = A[y];
     A[y] = t;
   }
 }
```

```
public int LargestElementPosition(int from, int to)
 {
  int max, i, p;
  max = A[from]; p = from;
  for (i=from; i<=to; i++)
   { if (A[i] > max) { max = A[i]; p = i; } }
  return p;
 }

public int NextLargestElementPosition(int which, int from, int to)
 {
  int max, i, p;
  max = -1;
  // max = A[from];
  p = from;
  for (i=from; i<=to; i++)
   { if (A[i] > A[which]) {max = A[i]; p = i; } }

  if (max == -1) return -1;

  for (i=from; i<=to; i++)
  { if ((A[i] > A[which]) && (A[i] < max)) { max = A[i]; p = i; } }
  return p;
 }


public void SortElements(int from, int to)
 {
  int t,p;
  for (int i=to; i >= from; i--)
   {
     t = A[i];
     p = LargestElementPosition(from, i);
     A[i] = A[p];
     A[p] = t;
   }
 }

public String toString()
 {
  String temp = new String();
  temp = "";
  for (int i=0; i<size; i++)
   { temp = temp + A[i]; }
  return temp;
 }

public void GetNextPermutation() //throws exception
 {
  int temp, p;
  int i = size -1;
  if (i <=0) return;
  while ((A[i] < A[i-1]) && (i > 0)) { i--; }
  if (i > 0)
   {
    temp = A[i-1];
      p = NextLargestElementPosition(i-1, i, this.size - 1);
```

```
        if (p > -1)
        {
          A[i-1] = A[p];
          A[p] = temp;
        SortElements(i, this.size - 1);
        }
     }
   }

 public static void main(String args[])
   {

    int c;
    double min, m;
    c = 0;
    random R = new random(100);
    random T = new random(100);
    min = 0;  //max

    for (c=0; c<30000; c++)
     {
       R.GetRandomString();
       m = R.fitness();
       if (m > min)
        {
         min = m;
         for (int cc=0; cc < R.size; cc++)
          {
           T.A[cc] = R.A[cc];
          }
        }
//       System.out.println(c+": "+R+", "+m);
       System.out.println(c+" "+m);
     }
    System.out.println("+++Minimum found: " +T+ ", "+min);

/*   permute P = new permute(15);
    int i=1;
    while (i==1)
//   for (int i=0; i<9999721; i++)
     {
      System.out.println( P +", " + P.fitness());
      P.GetNextPermutation();
     }
 */


  }
}
```

APPENDIX F: BRUTE FORCE SEARCH SOURCE CODE

The following listing is the complete source code for the Brute Force (Linear) Search Algorithm heuristic written in Java.

FILE: Permute.java

```
class permute
{
 public int A[];
 public int size;

 public int F[][] = { { 0, 1, 1, 1, 1, 1, 1, 2, 4, 5, 6, 9,10,12,20},
                      { 1, 0, 1, 1, 1, 1, 1, 1, 2, 3, 4, 7,10,11,19},
                      { 1, 1, 0, 1, 1, 1, 1, 1, 1, 2, 4, 6, 8,10,17},
                      { 1, 1, 1, 0, 1, 1, 1, 1, 1, 3, 5, 6, 9,12,14},
                      { 1, 1, 1, 1, 0, 1, 1, 1, 1, 2, 7, 8, 9,10,13},
                      { 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 2, 9,11,11,12},
                      { 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 2, 6, 7, 9,10},
                      { 2, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 2, 5, 7, 9},
                      { 4, 2, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 8},
                      { 5, 3, 2, 3, 2, 1, 1, 1, 1, 0, 1, 1, 2, 6, 6},
                      { 6, 4, 4, 5, 7, 2, 2, 1, 1, 1, 0, 1, 2, 3, 4},
                      { 9, 7, 6, 6, 8, 9, 6, 2, 1, 1, 1, 0, 1, 2, 3},
                      {10,10, 8, 9, 9,11, 7, 5, 1, 2, 2, 1, 0, 1, 2},
                      {12,11,10,12,10,11, 9, 7, 1, 6, 3, 2, 1, 0, 1},
                      {20,19,17,14,13,12,10, 9, 8, 6, 4, 3, 2, 1, 0},
};


 public permute(int s)
  {
   size = s;
   A = new int [s];
   for (int i=0; i<s; i++) { A[i] = i; }
  }

 public permute(permute P)
  {
   this.size = P.size;
   for(int i=0; i < this.size; i++) { this.A[i] = P.A[i];}
  }

 public double fitness()
  {
          // Calculate the fitness of the individual
          // given x, find f(x)
               int dist, dist_a, dist_b;
               double accum =0;
               for (int i=0; i<this.size;i++)
               {
                 for (int j=i+1; j<this.size;j++)
                  {
                   dist_a = A[i] - A[j];
                   if (dist_a < 0) { dist_a += this.size; }
                   dist_b = A[j] - A[i];
                   if (dist_b < 0) { dist_b += this.size; }

                   dist = Math.min(dist_a, dist_b);

                   accum+= dist * F[i][j];
                  }
               }
               return (accum); //Math.pow(gene.Value(), 2);


 }

public int LargestElementPosition(int from, int to)
 {
  int max, i, p;
  max = A[from]; p = from;
  for (i=from; i<=to; i++)
   { if (A[i] > max) { max = A[i]; p = i; } }
  return p;
 }

public int NextLargestElementPosition(int which, int from, int to)
 {
  int max, i, p;
  max = -1;
  // max = A[from];
  p = from;
  for (i=from; i<=to; i++)
   { if (A[i] > A[which]) {max = A[i]; p = i; } }

  if (max == -1) return -1;

  for (i=from; i<=to; i++)
  { if ((A[i] > A[which]) && (A[i] < max)) { max = A[i]; p = i; } }
  return p;
 }


public void SortElements(int from, int to)
 {
  int t,p;
  for (int i=to; i >= from; i--)
   {
     t = A[i];
     p = LargestElementPosition(from, i);
     A[i] = A[p];
     A[p] = t;
```

```
    }
  }

 public String toString()
  {
   String temp = new String();
   temp = "";
   for (int i=0; i<size; i++)
    { temp = temp + A[i]; }
   return temp;
  }

 public void GetNextPermutation() //throws exception
  {
   int temp, p;
   int i = size -1;
   if (i <=0) return;
   while ((A[i] < A[i-1]) && (i > 0)) { i--; }
   if (i > 0)
    {
     temp = A[i-1];
       p = NextLargestElementPosition(i-1, i, this.size - 1);
       if (p > -1)
       {
         A[i-1] = A[p];
         A[p] = temp;
       SortElements(i, this.size - 1);
       }
    }
  }

 public static void main(String args[])
  {
   permute P = new permute(15);
   int i=1;
   while (i==1)
//   for (int i=0; i<9999721; i++)
    {
     System.out.println( P +", " + P.fitness());
     P.GetNextPermutation();
    }
  }
}
```

APPENDIX G: OGA RESULTS

Optical Genetic Algorithm: Sample execution results of the OGA tested with chromosome size 100, over 1000 generations:

```
Optical Genetic Algorithm OGA:
-----------------------------
(c)1999 Ziad Kobti. All Rights Reserved.

Population Size  : 100
Chromosome Length: 100
Greedy Crossover
Two point swap mutation

G MAX        MIN        AVG        TotFit
0 3892744.0 3736707.0 3827604.62 3.82760462E8
1 3892744.0 3755539.0 3837232.84 3.83723284E8
2 3916770.0 3761262.0 3842268.25 3.84226825E8
3 3916770.0 3780648.0 3840666.61 3.84066661E8
4 3916770.0 3735781.0 3843094.99 3.84309499E8
5 3916770.0 3761402.0 3841983.65 3.84198365E8
6 3916770.0 3759923.0 3843832.49 3.84383249E8
7 3916770.0 3773178.0 3843456.92 3.84345692E8
8 3916770.0 3765939.0 3841112.47 3.84111247E8
9 3916770.0 3768249.0 3841718.68 3.84171868E8
10 3916770.0 3768821.0 3839530.68 3.83953068E8
11 3916770.0 3748449.0 3843674.86 3.84367486E8
12 3919060.0 3752680.0 3844300.48 3.84430048E8
13 3919060.0 3768753.0 3846020.84 3.84602084E8
14 3919060.0 3748886.0 3840130.14 3.84013014E8
15 3919060.0 3760415.0 3848640.55 3.84864055E8
16 3919060.0 3727860.0 3847366.84 3.84736684E8
17 3919060.0 3769751.0 3848649.39 3.84864939E8
18 3919060.0 3765046.0 3841040.16 3.84104016E8
19 3919060.0 3763180.0 3844441.39 3.84444139E8
20 3919060.0 3760119.0 3842845.71 3.84284571E8
21 3919060.0 3768924.0 3847500.33 3.84750033E8
22 3919060.0 3754772.0 3845696.43 3.84569643E8
23 3919060.0 3763950.0 3858961.55 3.85896155E8
24 3919060.0 3766438.0 3850889.48 3.85088948E8
25 3919060.0 3756570.0 3851985.35 3.85198535E8
26 3919060.0 3746895.0 3851622.17 3.85162217E8
27 3919060.0 3770402.0 3852038.75 3.85203875E8
28 3919060.0 3748319.0 3846911.41 3.84691141E8
29 3919060.0 3746895.0 3843881.81 3.84388181E8
30 3919060.0 3764254.0 3855611.36 3.85561136E8
31 3919060.0 3752162.0 3851194.69 3.85119469E8
32 3919060.0 3750773.0 3854883.04 3.85488304E8
33 3919060.0 3781908.0 3860795.78 3.86079578E8
```

```
34 3919060.0 3787392.0 3865475.84 3.86547584E8
35 3919060.0 3776111.0 3866643.23 3.86664323E8
36 3919060.0 3794491.0 3862046.46 3.86204646E8
37 3919060.0 3768074.0 3856700.28 3.85670028E8
38 3919060.0 3772592.0 3865362.98 3.86536298E8
39 3919060.0 3771109.0 3862981.24 3.86298124E8
40 3919060.0 3777535.0 3865151.03 3.86515103E8
41 3919060.0 3782497.0 3859272.71 3.85927271E8
42 3919060.0 3753749.0 3857094.85 3.85709485E8
43 3919060.0 3777225.0 3863013.49 3.86301349E8
44 3919060.0 3781581.0 3869143.49 3.86914349E8
45 3919060.0 3775371.0 3865149.6 3.8651496E8
46 3919060.0 3767828.0 3864346.49 3.86434649E8
47 3919060.0 3794090.0 3873937.55 3.87393755E8
48 3919060.0 3786075.0 3859539.83 3.85953983E8
49 3919060.0 3753111.0 3860371.65 3.86037165E8
50 3919060.0 3791207.0 3864842.63 3.86484263E8
51 3919060.0 3764022.0 3868383.08 3.86838308E8
52 3919060.0 3774927.0 3865703.35 3.86570335E8
53 3919060.0 3756613.0 3859170.68 3.85917068E8
54 3919060.0 3752955.0 3849874.67 3.84987467E8
55 3919060.0 3755818.0 3847414.6 3.8474146E8
56 3919060.0 3748292.0 3851339.25 3.85133925E8
57 3919060.0 3750366.0 3855076.3 3.8550763E8
58 3919060.0 3776616.0 3880322.69 3.88032269E8
59 3919060.0 3765466.0 3888019.11 3.88801911E8
60 3919060.0 3768345.0 3886103.41 3.88610341E8
61 3919060.0 3781162.0 3878614.38 3.87861438E8
62 3919060.0 3786323.0 3876602.55 3.87660255E8
63 3919060.0 3789067.0 3880230.14 3.88023014E8
64 3919060.0 3777187.0 3888440.94 3.88844094E8
65 3919060.0 3773318.0 3888267.63 3.88826763E8
66 3919060.0 3780232.0 3899188.83 3.89918883E8
67 3920838.0 3790015.0 3912312.58 3.91231258E8
68 3920838.0 3787786.0 3916320.72 3.91632072E8
69 3920838.0 3779241.0 3896192.26 3.89619226E8
70 3926132.0 3779241.0 3902922.56 3.90292256E8
71 3926132.0 3779241.0 3895802.51 3.89580251E8
72 3927055.0 3777626.0 3907094.71 3.90709471E8
73 3927055.0 3760885.0 3913354.95 3.91335495E8
74 3927055.0 3812635.0 3912682.3 3.9126823E8
75 3927055.0 3778430.0 3903966.3 3.9039663E8
76 3927055.0 3800573.0 3920730.9 3.9207309E8
77 3930707.0 3927055.0 3927201.08 3.92720108E8
78 3930707.0 3803070.0 3916230.77 3.91623077E8
79 3930707.0 3810674.0 3925294.63 3.92529463E8
80 3932195.0 3811668.0 3919353.97 3.91935397E8
81 3937963.0 3908508.0 3931225.49 3.93122549E8
82 3937963.0 3812704.0 3909606.91 3.90960691E8
83 3939119.0 3798842.0 3926661.26 3.92666126E8
84 3943627.0 3798166.0 3927832.35 3.92783235E8
85 3945581.0 3835329.0 3937156.47 3.93715647E8
86 3945581.0 3800890.0 3925735.55 3.92573555E8
87 3945581.0 3800890.0 3924987.79 3.92498779E8
88 3945581.0 3822675.0 3938607.52 3.93860752E8
89 3945581.0 3816999.0 3938920.55 3.93892055E8
90 3945581.0 3812907.0 3922775.37 3.92277537E8
```

```
91 3945581.0 3819882.0 3936275.86 3.93627586E8
92 3945581.0 3835329.0 3943928.95 3.94392895E8
93 3945581.0 3767646.0 3923744.53 3.92374453E8
94 3945581.0 3787235.0 3921033.09 3.92103309E8
95 3945581.0 3793994.0 3927039.12 3.92703912E8
96 3945581.0 3784024.0 3917882.2 3.9178822E8
97 3945581.0 3814691.0 3910286.8 3.9102868E8
98 3945581.0 3835329.0 3935003.51 3.93500351E8
99 3945581.0 3835329.0 3933928.62 3.93392862E8
100 3945581.0 3791246.0 3920636.24 3.92063624E8
101 3945581.0 3795295.0 3923591.19 3.92359119E8
102 3945581.0 3798682.0 3937426.55 3.93742655E8
103 3947653.0 3826128.0 3936963.54 3.93696354E8
104 3947653.0 3823043.0 3933485.68 3.93348568E8
105 3947653.0 3802471.0 3937098.96 3.93709896E8
106 3947653.0 3802471.0 3940005.07 3.94000507E8
107 3947653.0 3807901.0 3942166.66 3.94216666E8
108 3953801.0 3822934.0 3938659.26 3.93865926E8
109 3953801.0 3822934.0 3936071.89 3.93607189E8
110 3953801.0 3821796.0 3937893.4 3.9378934E8
111 3953801.0 3861796.0 3941431.68 3.94143168E8
112 3965777.0 3773660.0 3939524.68 3.93952468E8
113 3965777.0 3773660.0 3931054.82 3.93105482E8
114 3965777.0 3823943.0 3934165.61 3.93416561E8
115 3965777.0 3788481.0 3933178.55 3.93317855E8
116 3974133.0 3773711.0 3936763.75 3.93676375E8
117 3974133.0 3792077.0 3943848.41 3.94384841E8
118 3974133.0 3829025.0 3949487.9 3.9494879E8
119 3974133.0 3796442.0 3941895.82 3.94189582E8
120 3981821.0 3796442.0 3957207.46 3.95720746E8
121 3981821.0 3796442.0 3949499.29 3.94949929E8
122 3981821.0 3748596.0 3944893.47 3.94489347E8
123 3981821.0 3782365.0 3959230.25 3.95923025E8
124 3981947.0 3803445.0 3975438.02 3.97543802E8
125 3981947.0 3826133.0 3975603.56 3.97560356E8
126 3981947.0 3771540.0 3978821.11 3.97882111E8
127 3981947.0 3782850.0 3955648.03 3.95564803E8
128 3981947.0 3788076.0 3928259.23 3.92825923E8
129 3981947.0 3833311.0 3962788.62 3.96278862E8
130 3981947.0 3842983.0 3969857.16 3.96985716E8
131 3981947.0 3793253.0 3941057.99 3.94105799E8
132 3981947.0 3796726.0 3929292.41 3.92929241E8
133 3981947.0 3792759.0 3967905.66 3.96790566E8
134 3981947.0 3865341.0 3965393.38 3.96539338E8
135 3981947.0 3797534.0 3971359.13 3.97135913E8
136 3981947.0 3819151.0 3971606.34 3.97160634E8
137 3984607.0 3814181.0 3972689.99 3.97268999E8
138 3984607.0 3829566.0 3967291.44 3.96729144E8
139 3984607.0 3803070.0 3957972.99 3.95797299E8
140 3984607.0 3815141.0 3964158.29 3.96415829E8
141 3984607.0 3805008.0 3971919.46 3.97191946E8
142 3984607.0 3822412.0 3968557.3 3.9685573E8
143 3984607.0 3781524.0 3950831.38 3.95083138E8
144 3984607.0 3806118.0 3960390.39 3.96039039E8
145 3984607.0 3781895.0 3973734.43 3.97373443E8
146 3984607.0 3781262.0 3937287.02 3.93728702E8
147 3984607.0 3795131.0 3951318.07 3.95131807E8
```

```
148 3984607.0 3794236.0 3945584.67 3.94558467E8
149 3984607.0 3750543.0 3957537.51 3.95753751E8
150 3984607.0 3809490.0 3959133.71 3.95913371E8
151 3984607.0 3803882.0 3954421.13 3.95442113E8
152 3984607.0 3825027.0 3961554.23 3.96155423E8
153 3984607.0 3816760.0 3952531.8 3.9525318E8
154 3984607.0 3826680.0 3956682.17 3.95668217E8
155 3984607.0 3766132.0 3954868.35 3.95486835E8
156 3984607.0 3846597.0 3963065.22 3.96306522E8
157 3984607.0 3778095.0 3950203.27 3.95020327E8
158 3984607.0 3766574.0 3940181.79 3.94018179E8
159 3984607.0 3787708.0 3941896.92 3.94189692E8
160 3984607.0 3766132.0 3930978.18 3.93097818E8
161 3984607.0 3795333.0 3949617.1 3.9496171E8
162 3984607.0 3795333.0 3958100.72 3.95810072E8
163 3984607.0 3816491.0 3965840.07 3.96584007E8
164 3984607.0 3786619.0 3954214.78 3.95421478E8
165 3984607.0 3786619.0 3946065.57 3.94606557E8
166 3984607.0 3807510.0 3942199.11 3.94219911E8
167 3984607.0 3816491.0 3961826.08 3.96182608E8
168 3984607.0 3816491.0 3952296.63 3.95229663E8
169 3984607.0 3773788.0 3938090.78 3.93809078E8
170 3984607.0 3815832.0 3981426.34 3.98142634E8
171 3984607.0 3811626.0 3971367.35 3.97136735E8
172 3984607.0 3800903.0 3971752.7 3.9717527E8
173 3984607.0 3816491.0 3964455.16 3.96445516E8
174 3990493.0 3797488.0 3946334.8 3.9463348E8
175 3990493.0 3804532.0 3924520.69 3.92452069E8
176 3990493.0 3789613.0 3958673.55 3.95867355E8
177 3990493.0 3810095.0 3964134.71 3.96413471E8
178 3990493.0 3802334.0 3938010.59 3.93801059E8
179 3990493.0 3800979.0 3954154.78 3.95415478E8
180 3990493.0 3816491.0 3950511.83 3.95051183E8
181 3990493.0 3794902.0 3951017.91 3.95101791E8
182 3990493.0 3816491.0 3969133.9 3.9691339E8
183 3990493.0 3816491.0 3967855.96 3.96785596E8
184 3990493.0 3777456.0 3969303.91 3.96930391E8
185 3990493.0 3816491.0 3959643.42 3.95964342E8
186 3990493.0 3775612.0 3941855.83 3.94185583E8
187 3990493.0 3794737.0 3931664.22 3.93166422E8
188 3990493.0 3792327.0 3954665.58 3.95466558E8
189 3990493.0 3805949.0 3949088.23 3.94908823E8
190 3990493.0 3814730.0 3956879.64 3.95687964E8
191 3990493.0 3795042.0 3954924.07 3.95492407E8
192 3990493.0 3816491.0 3963425.13 3.96342513E8
193 3990493.0 3792641.0 3934934.81 3.93493481E8
194 3990493.0 3807312.0 3926463.95 3.92646395E8
195 3990493.0 3774238.0 3933039.33 3.93303933E8
196 3990493.0 3782113.0 3924094.68 3.92409468E8
197 3990493.0 3766899.0 3931231.6 3.9312316E8
198 3990493.0 3789681.0 3918287.33 3.91828733E8
199 3990493.0 3796860.0 3961008.43 3.96100843E8
200 3990493.0 3820718.0 3978656.04 3.97865604E8
201 3990493.0 3791392.0 3971209.21 3.97120921E8
202 3990493.0 3811074.0 3959589.25 3.95958925E8
203 3990493.0 3789038.0 3938451.77 3.93845177E8
204 3990493.0 3785315.0 3976153.51 3.97615351E8
```

```
205 3990493.0 3816517.0 3968838.34 3.96883834E8
206 3990493.0 3799435.0 3946943.7 3.9469437E8
207 3990493.0 3800552.0 3953135.12 3.95313512E8
208 3990493.0 3802775.0 3967657.38 3.96765738E8
209 3990493.0 3802775.0 3935131.94 3.93513194E8
210 3990493.0 3823593.0 3968876.92 3.96887692E8
211 3990493.0 3793505.0 3957009.01 3.95700901E8
212 3990493.0 3823593.0 3966412.38 3.96641238E8
213 3990493.0 3799718.0 3970969.09 3.97096909E8
214 3990493.0 3823593.0 3967657.25 3.96765725E8
215 3990493.0 3818736.0 3951204.96 3.95120496E8
216 3990493.0 3823593.0 3965392.89 3.96539289E8
217 3990493.0 3774367.0 3938200.22 3.93820022E8
218 3990493.0 3807135.0 3971437.59 3.97143759E8
219 3990493.0 3781551.0 3946299.73 3.94629973E8
220 3990493.0 3767309.0 3970103.08 3.97010308E8
221 3990493.0 3812522.0 3968908.95 3.96890895E8
222 3990493.0 3800873.0 3965402.28 3.96540228E8
223 3990493.0 3800873.0 3963210.41 3.96321041E8
224 3990493.0 3802500.0 3963310.11 3.96331011E8
225 3990493.0 3802500.0 3946982.32 3.94698232E8
226 3990493.0 3796129.0 3957712.5 3.9577125E8
227 3990493.0 3823593.0 3954674.14 3.95467414E8
228 3990493.0 3785710.0 3952424.02 3.95242402E8
229 3990493.0 3800275.0 3971840.63 3.97184063E8
230 3990493.0 3823593.0 3943695.31 3.94369531E8
231 3993796.0 3790107.0 3941135.09 3.94113509E8
232 3993796.0 3793961.0 3939253.08 3.93925308E8
233 3993796.0 3830004.0 3967134.96 3.96713496E8
234 3993796.0 3824178.0 3952856.5 3.9528565E8
235 3993796.0 3794629.0 3976945.34 3.97694534E8
236 3993796.0 3796392.0 3957195.45 3.95719545E8
237 3993796.0 3796601.0 3955585.49 3.95558549E8
238 3993796.0 3799846.0 3951236.51 3.95123651E8
239 3993796.0 3793961.0 3948436.74 3.94843674E8
240 3993796.0 3793961.0 3954744.54 3.95474454E8
241 3993796.0 3827717.0 3961836.89 3.96183689E8
242 3993796.0 3759140.0 3960869.16 3.96086916E8
243 3993796.0 3782106.0 3965666.11 3.96566611E8
244 3993796.0 3801448.0 3948270.7 3.9482707E8
245 3993796.0 3826076.0 3967955.14 3.96795514E8
246 3993796.0 3807134.0 3962480.76 3.96248076E8
247 3993796.0 3807134.0 3955441.12 3.95544112E8
248 3993796.0 3772022.0 3953718.97 3.95371897E8
249 3993796.0 3813848.0 3962982.9 3.9629829E8
250 3993796.0 3807790.0 3939830.7 3.9398307E8
251 3993796.0 3817177.0 3951749.6 3.9517496E8
252 3993796.0 3760523.0 3966497.5 3.9664975E8
253 3993796.0 3830004.0 3977334.85 3.97733485E8
254 3993796.0 3825619.0 3964695.84 3.96469584E8
255 3993796.0 3825619.0 3978427.6 3.9784276E8
256 3993796.0 3824953.0 3960911.85 3.96091185E8
257 3993796.0 3807049.0 3961925.1 3.9619251E8
258 3993796.0 3804359.0 3938979.7 3.9389797E8
259 3993796.0 3801448.0 3951762.0 3.951762E8
260 3993796.0 3782834.0 3947445.92 3.94744592E8
261 3993796.0 3827717.0 3979583.14 3.97958314E8
```

```
262 3993796.0 3799356.0 3970000.71 3.97000071E8
263 3993796.0 3799356.0 3959737.81 3.95973781E8
264 3993796.0 3818884.0 3953219.72 3.95321972E8
265 3993796.0 3792960.0 3935286.1 3.9352861E8
266 3993796.0 3827717.0 3968477.02 3.96847702E8
267 3993796.0 3830004.0 3971469.28 3.97146928E8
268 3993796.0 3827717.0 3961492.94 3.96149294E8
269 3993796.0 3827717.0 3967804.72 3.96780472E8
270 3993796.0 3818862.0 3972597.99 3.97259799E8
271 3993796.0 3782106.0 3949495.01 3.94949501E8
272 3993796.0 3788286.0 3949977.93 3.94997793E8
273 3993796.0 3827717.0 3964547.44 3.96454744E8
274 3993796.0 3827717.0 3977126.61 3.97712661E8
275 3993796.0 3782024.0 3963523.37 3.96352337E8
276 3993796.0 3827717.0 3976797.87 3.97679787E8
277 3993796.0 3788886.0 3962167.02 3.96216702E8
278 3993796.0 3813581.0 3959219.41 3.95921941E8
279 3993796.0 3830004.0 3980314.39 3.98031439E8
280 3993796.0 3830004.0 3966001.91 3.96600191E8
281 3997501.0 3834263.0 3972142.41 3.97214241E8
282 3997501.0 3774334.0 3943196.45 3.94319645E8
283 3997501.0 3788809.0 3955956.13 3.95595613E8
284 3997501.0 3818410.0 3980304.68 3.98030468E8
285 3997501.0 3798818.0 3960750.36 3.96075036E8
286 3997501.0 3818980.0 3966411.59 3.96641159E8
287 3997501.0 3816811.0 3983107.2 3.9831072E8
288 3997501.0 3821614.0 3972739.58 3.97273958E8
289 4005573.0 3805635.0 3969540.83 3.96954083E8
290 4005573.0 3821614.0 3970730.18 3.97073018E8
291 4005573.0 3766007.0 3963607.98 3.96360798E8
292 4005573.0 3791789.0 3977986.16 3.97798616E8
293 4005573.0 3763999.0 3960461.75 3.96046175E8
294 4005573.0 3763999.0 3982885.63 3.98288563E8
295 4005573.0 3787673.0 3968635.86 3.96863586E8
296 4005573.0 3808501.0 3959400.19 3.95940019E8
297 4005573.0 3763999.0 3961966.58 3.96196658E8
298 4005573.0 3763999.0 3982590.28 3.98259028E8
299 4005573.0 3811304.0 3966893.1 3.9668931E8
300 4005573.0 3828127.0 3999777.97 3.99977797E8
301 4005573.0 3820848.0 3971525.44 3.97152544E8
302 4005573.0 3790909.0 3951018.53 3.95101853E8
303 4005573.0 3763999.0 3980852.42 3.98085242E8
304 4005573.0 3763999.0 3971948.55 3.97194855E8
305 4005573.0 3763999.0 3969621.54 3.96962154E8
306 4005573.0 3763999.0 3965233.17 3.96523317E8
307 4005573.0 3763999.0 3979759.87 3.97975987E8
308 4005573.0 3763999.0 3992393.64 3.99239364E8
309 4005573.0 3800228.0 3973966.88 3.97396688E8
310 4005573.0 3777873.0 3970343.15 3.97034315E8
311 4005573.0 3763999.0 3969848.12 3.96984812E8
312 4005573.0 3763999.0 3983037.66 3.98303766E8
313 4006118.0 3763999.0 3953754.22 3.95375422E8
314 4006118.0 3763999.0 3980644.01 3.98064401E8
315 4006118.0 3828127.0 3997388.16 3.99738816E8
316 4006118.0 3835355.0 3999881.21 3.99988121E8
317 4006118.0 3851262.0 4004559.64 4.00455964E8
318 4006118.0 3766509.0 3976243.23 3.97624323E8
```

```
319 4006118.0 3770618.0 3985035.1 3.9850351E8
320 4006118.0 3792538.0 3980866.58 3.98086658E8
321 4006118.0 3801519.0 3974674.3 3.9746743E8
322 4006118.0 3800569.0 3983191.35 3.98319135E8
323 4006118.0 3860601.0 3997434.31 3.99743431E8
324 4006118.0 3813482.0 3985962.69 3.98596269E8
325 4006118.0 3761939.0 3979208.49 3.97920849E8
326 4006118.0 3761939.0 3991244.54 3.99124454E8
327 4006118.0 3859159.0 4002200.71 4.00220071E8
328 4006118.0 3847818.0 3996303.24 3.99630324E8
329 4006118.0 3813914.0 3977008.1 3.9770081E8
330 4006118.0 3816950.0 4001756.92 4.00175692E8
331 4006118.0 3798300.0 4000068.98 4.00006898E8
332 4006118.0 3814667.0 3997078.56 3.99707856E8
333 4006118.0 3792538.0 3982757.71 3.98275771E8
334 4006118.0 3826331.0 3996973.46 3.99697346E8
335 4006118.0 3785529.0 3980507.22 3.98050722E8
336 4006118.0 3857375.0 3999049.59 3.99904959E8
337 4006118.0 3775677.0 3989330.96 3.98933096E8
338 4006118.0 3856349.0 3989062.56 3.98906256E8
339 4006118.0 3825307.0 3995524.79 3.99552479E8
340 4006118.0 3768383.0 3980855.25 3.98085525E8
341 4006118.0 3821120.0 3998116.7 3.9981167E8
342 4012255.0 3821120.0 3987001.02 3.98700102E8
343 4012255.0 3826331.0 4005730.66 4.00573066E8
344 4012255.0 3769463.0 3998376.88 3.99837688E8
345 4012255.0 3846649.0 3999669.07 3.99966907E8
346 4012255.0 3882577.0 3997358.55 3.99735855E8
347 4012255.0 3826162.0 3986696.13 3.98669613E8
348 4012255.0 3830601.0 3993171.39 3.99317139E8
349 4012255.0 3872224.0 4002316.33 4.00231633E8
350 4012255.0 3808928.0 3979316.25 3.97931625E8
351 4012255.0 3823472.0 3991816.2 3.9918162E8
352 4012255.0 3889095.0 4003207.68 4.00320768E8
353 4012255.0 3821870.0 3976498.36 3.97649836E8
354 4012255.0 3821870.0 3996453.39 3.99645339E8
355 4012255.0 3784762.0 3994795.3 3.9947953E8
356 4012255.0 3770141.0 3941072.5 3.9410725E8
357 4012255.0 3773678.0 3968834.84 3.96883484E8
358 4012255.0 3815898.0 4001002.23 4.00100223E8
359 4012255.0 3832761.0 3999523.28 3.99952328E8
360 4012255.0 3841599.0 4000117.14 4.00011714E8
361 4012255.0 3825154.0 3994133.9 3.9941339E8
362 4012255.0 3836670.0 4004360.81 4.00436081E8
363 4012255.0 3845261.0 3994310.52 3.99431052E8
364 4012255.0 3787486.0 3991450.08 3.99145008E8
365 4012255.0 3787486.0 3988187.08 3.98818708E8
366 4012255.0 3808837.0 4000079.49 4.00007949E8
367 4012255.0 3849302.0 4004025.7 4.0040257E8
368 4012255.0 3804781.0 3961442.99 3.96144299E8
369 4012255.0 3786349.0 3961570.17 3.96157017E8
370 4012255.0 3887364.0 3997052.29 3.99705229E8
371 4012255.0 3817062.0 3971247.83 3.97124783E8
372 4012255.0 3828767.0 3988411.08 3.98841108E8
373 4012255.0 3821227.0 3986067.54 3.98606754E8
374 4012255.0 3765397.0 3954191.74 3.95419174E8
375 4012255.0 3827019.0 3977819.56 3.97781956E8
```

376	4012255.0	3829698.0	4004654.48	4.00465448E8
377	4012255.0	3833563.0	3991411.15	3.99141115E8
378	4012255.0	3822545.0	3982993.24	3.98299324E8
379	4012255.0	3896756.0	3995459.55	3.99545955E8
380	4012255.0	3815198.0	3957865.08	3.95786508E8
381	4014806.0	3821870.0	3989596.02	3.98959602E8
382	4014806.0	3817300.0	3965897.34	3.96589734E8
383	4014806.0	3822861.0	3971328.31	3.97132831E8
384	4014806.0	3795313.0	3968310.46	3.96831046E8
385	4015208.0	3819037.0	3990634.45	3.99063445E8
386	4015208.0	3795832.0	3974376.23	3.97437623E8
387	4015208.0	3792450.0	3951246.4	3.9512464E8
388	4015208.0	3810016.0	3966798.75	3.96679875E8
389	4015208.0	3822861.0	3994767.67	3.99476767E8
390	4015208.0	3792510.0	3979765.02	3.97976502E8
391	4015208.0	3761459.0	3962245.11	3.96224511E8
392	4015208.0	3792510.0	3961578.79	3.96157879E8
393	4015208.0	3810016.0	3950642.5	3.9506425E8
394	4015208.0	3793212.0	3964414.37	3.96441437E8
395	4015208.0	3792510.0	3960891.06	3.96089106E8
396	4015208.0	3802884.0	3973020.63	3.97302063E8
397	4015208.0	3792510.0	3961427.42	3.96142742E8
398	4018744.0	3807751.0	3973942.7	3.9739427E8
399	4018744.0	3800762.0	4002244.88	4.00224488E8
400	4018744.0	3807834.0	3987211.8	3.9872118E8
401	4019682.0	3807834.0	3970623.5	3.9706235E8
402	4019682.0	3803745.0	3993954.81	3.99395481E8
403	4019682.0	3792753.0	4001472.11	4.00147211E8
404	4019682.0	3761429.0	3981111.43	3.98111143E8
405	4019682.0	3801865.0	3949052.6	3.9490526E8
406	4019682.0	3801865.0	3980244.41	3.98024441E8
407	4019682.0	3799481.0	3982496.62	3.98249662E8
408	4019682.0	3816570.0	4001351.72	4.00135172E8
409	4019682.0	3801865.0	3979953.75	3.97995375E8
410	4019682.0	3799350.0	3974919.46	3.97491946E8
411	4019682.0	3784307.0	3948448.43	3.94844843E8
412	4019682.0	3770703.0	3967329.24	3.96732924E8
413	4019682.0	3801865.0	3998401.53	3.99840153E8
414	4023640.0	3801865.0	4003640.33	4.00364033E8
415	4025568.0	3796851.0	3976679.2	3.9766792E8
416	4025568.0	3797978.0	4002269.86	4.00226986E8
417	4025568.0	3816750.0	3994025.21	3.99402521E8
418	4025568.0	3810011.0	3993633.5	3.9936335E8
419	4025568.0	3773356.0	3975856.43	3.97585643E8
420	4025568.0	3801038.0	4014737.56	4.01473756E8
421	4025568.0	3827990.0	4023590.88	4.02359088E8
422	4025568.0	3790050.0	4012124.92	4.01212492E8
423	4025568.0	3793574.0	4003622.08	4.00362208E8
424	4025568.0	3814491.0	4023457.23	4.02345723E8
425	4025858.0	3812255.0	4006585.48	4.00658548E8
426	4025858.0	3831041.0	3999939.96	3.99993996E8
427	4025858.0	3782279.0	4014377.77	4.01437777E8
428	4025858.0	3782279.0	3975366.06	3.97536606E8
429	4025858.0	3762494.0	3972338.02	3.97233802E8
430	4025858.0	3792465.0	3985867.49	3.98586749E8
431	4025858.0	3779895.0	3972905.37	3.97290537E8
432	4025858.0	3782279.0	3975499.98	3.97549998E8

```
433 4025858.0 3792465.0 3966293.13 3.96629313E8
434 4025858.0 3792465.0 3955545.14 3.95554514E8
435 4025858.0 3779895.0 3970772.06 3.97077206E8
436 4025858.0 3773092.0 3948615.64 3.94861564E8
437 4025858.0 3762181.0 3957363.08 3.95736308E8
438 4025858.0 3779895.0 3952753.05 3.95275305E8
439 4025858.0 3770151.0 3969418.41 3.96941841E8
440 4025858.0 3753921.0 3982034.3 3.9820343E8
441 4025858.0 3792465.0 3975734.71 3.97573471E8
442 4025858.0 3792465.0 3979697.26 3.97969726E8
443 4025858.0 3790558.0 3956013.11 3.95601311E8
444 4025858.0 3787836.0 3969208.33 3.96920833E8
445 4025858.0 3774409.0 3970090.8 3.9700908E8
446 4025858.0 3792465.0 3971406.07 3.97140607E8
447 4025858.0 3760783.0 3966078.2 3.9660782E8
448 4025858.0 3773857.0 3923574.94 3.92357494E8
449 4025858.0 3762181.0 3953405.61 3.95340561E8
450 4025858.0 3782279.0 3980248.56 3.98024856E8
451 4029291.0 3792465.0 3976487.47 3.97648747E8
452 4032853.0 3788702.0 3975662.84 3.97566284E8
453 4032853.0 3757595.0 3961018.1 3.9610181E8
454 4032853.0 3771827.0 3999547.66 3.99954766E8
455 4032853.0 3773300.0 4007585.61 4.00758561E8
456 4032853.0 3796287.0 4019754.92 4.01975492E8
457 4032853.0 3770487.0 4004316.86 4.00431686E8
458 4032853.0 3787629.0 4007834.61 4.00783461E8
459 4032853.0 3839614.0 4013300.19 4.01330019E8
460 4032853.0 3871574.0 4015637.02 4.01563702E8
461 4032853.0 3871574.0 4012785.48 4.01278548E8
462 4032853.0 3870481.0 4010977.03 4.01097703E8
463 4032853.0 3837087.0 3994444.52 3.99444452E8
464 4032853.0 3783815.0 4000640.63 4.00064063E8
465 4032853.0 3890882.0 4025159.01 4.02515901E8
466 4032853.0 3862759.0 4025685.21 4.02568521E8
467 4032853.0 3784733.0 3985429.43 3.98542943E8
468 4032853.0 3774624.0 3992705.66 3.99270566E8
469 4033795.0 3827748.0 4011887.74 4.01188774E8
470 4033795.0 3831167.0 4007829.97 4.00782997E8
471 4033795.0 3820710.0 3991177.7 3.9911777E8
472 4033795.0 3838253.0 4023521.34 4.02352134E8
473 4033795.0 3783901.0 4026291.08 4.02629108E8
474 4033795.0 3804088.0 4031413.79 4.03141379E8
475 4033795.0 3855277.0 4031988.44 4.03198844E8
476 4033795.0 3833275.0 4010338.03 4.01033803E8
477 4033795.0 3795578.0 4002893.95 4.00289395E8
478 4033795.0 3747352.0 4000408.71 4.00040871E8
479 4033795.0 3849162.0 4004520.99 4.00452099E8
480 4033795.0 3791136.0 4008758.49 4.00875849E8
481 4033795.0 3809288.0 4018645.67 4.01864567E8
482 4033795.0 3809288.0 4006898.65 4.00689865E8
483 4033795.0 3816410.0 3988750.81 3.98875081E8
484 4033795.0 3890489.0 4013818.87 4.01381887E8
485 4033795.0 3849229.0 4023574.87 4.02357487E8
486 4033795.0 3849229.0 4017829.83 4.01782983E8
487 4033795.0 4018806.0 4033645.11 4.03364511E8
488 4033795.0 3786050.0 4020995.77 4.02099577E8
489 4033795.0 3810768.0 4009432.85 4.00943285E8
```

```
490 4035775.0 3794579.0 4010960.67 4.01096067E8
491 4035775.0 3768320.0 4021244.94 4.02124494E8
492 4036295.0 3768320.0 4007134.67 4.00713467E8
493 4036295.0 3842420.0 4025079.25 4.02507925E8
494 4036295.0 3773132.0 3981574.45 3.98157445E8
495 4036295.0 3807016.0 4007295.58 4.00729558E8
496 4036295.0 3826382.0 4015709.22 4.01570922E8
497 4036295.0 3795525.0 3988330.71 3.98833071E8
498 4036295.0 3798744.0 4006928.94 4.00692894E8
499 4036295.0 3830756.0 4001163.03 4.00116303E8
500 4036295.0 3826841.0 4020367.39 4.02036739E8
501 4036295.0 3819125.0 4002522.92 4.00252292E8
502 4036295.0 3800831.0 4018362.58 4.01836258E8
503 4036295.0 3832271.0 3998978.3 3.9989783E8
504 4036295.0 3829232.0 4005181.99 4.00518199E8
505 4036295.0 3788173.0 3997985.12 3.99798512E8
506 4036295.0 3818415.0 3989842.92 3.98984292E8
507 4036295.0 3779152.0 3998151.59 3.99815159E8
508 4036295.0 3789973.0 4007335.71 4.00733571E8
509 4036295.0 3792183.0 4001589.93 4.00158993E8
510 4036295.0 3793445.0 4011567.11 4.01156711E8
511 4036295.0 3783795.0 3989751.19 3.98975119E8
512 4036295.0 3783795.0 4007242.52 4.00724252E8
513 4036295.0 3804495.0 3998718.48 3.99871848E8
514 4036295.0 3826306.0 4003582.82 4.00358282E8
515 4036295.0 3832271.0 4016615.84 4.01661584E8
516 4036295.0 3822901.0 4003013.06 4.00301306E8
517 4036295.0 3800831.0 3985387.42 3.98538742E8
518 4036295.0 3799820.0 4007167.01 4.00716701E8
519 4036295.0 3799820.0 4024786.24 4.02478624E8
520 4036295.0 3832271.0 4009909.77 4.00990977E8
521 4036295.0 3822021.0 4013800.8 4.0138008E8
522 4036295.0 3793377.0 4013517.36 4.01351736E8
523 4036295.0 3783579.0 3971385.17 3.97138517E8
524 4036295.0 3808775.0 4007162.28 4.00716228E8
525 4036295.0 3808401.0 4008941.64 4.00894164E8
526 4038607.0 3788756.0 4005820.82 4.00582082E8
527 4038607.0 3776501.0 4012711.87 4.01271187E8
528 4038607.0 3777387.0 4008279.34 4.00827934E8
529 4038607.0 3776730.0 3995414.13 3.99541413E8
530 4038607.0 3812673.0 3968372.06 3.96837206E8
531 4038607.0 3832271.0 4024396.31 4.02439631E8
532 4038607.0 3779152.0 4009766.89 4.00976689E8
533 4038607.0 3816117.0 4012968.54 4.01296854E8
534 4038607.0 3788526.0 3993694.85 3.99369485E8
535 4038607.0 3787184.0 4016407.78 4.01640778E8
536 4038607.0 3841619.0 4028441.15 4.02844115E8
537 4038607.0 3793087.0 4016476.08 4.01647608E8
538 4038607.0 3841110.0 4007323.61 4.00732361E8
539 4038607.0 3797642.0 4019143.0 4.019143E8
540 4038607.0 3795872.0 4020153.81 4.02015381E8
541 4038607.0 3787484.0 4024404.9 4.0244049E8
542 4038607.0 3814880.0 3992369.96 3.99236996E8
543 4038607.0 3799749.0 4018500.84 4.01850084E8
544 4038607.0 3795646.0 4015127.48 4.01512748E8
545 4038607.0 3771994.0 4020022.24 4.02002224E8
546 4038607.0 3870963.0 4028711.37 4.02871137E8
```

```
547 4038607.0 3833468.0 4029091.59 4.02909159E8
548 4039023.0 3855587.0 4021506.06 4.02150606E8
549 4039023.0 3806711.0 3972862.83 3.97286283E8
550 4039023.0 3770325.0 4009844.74 4.00984474E8
551 4039023.0 3782221.0 3999943.52 3.99994352E8
552 4039023.0 3806583.0 4035012.64 4.03501264E8
553 4042645.0 3818379.0 4011744.3 4.0117443E8
554 4042645.0 3798850.0 3990051.73 3.99005173E8
555 4042645.0 3821369.0 4022725.01 4.02272501E8
556 4042645.0 3832201.0 4032409.68 4.03240968E8
557 4042645.0 3809128.0 4016838.55 4.01683855E8
558 4042645.0 3773789.0 4016546.27 4.01654627E8
559 4042645.0 3797277.0 4009079.95 4.00907995E8
560 4042645.0 3832201.0 3995070.36 3.99507036E8
561 4042645.0 3768093.0 4028943.55 4.02894355E8
562 4042645.0 3768093.0 4008656.18 4.00865618E8
563 4042645.0 3821805.0 4000295.33 4.00029533E8
564 4042645.0 3821392.0 4010764.4 4.0107644E8
565 4042645.0 3853647.0 4029723.22 4.02972322E8
566 4043798.0 3797767.0 4023810.54 4.02381054E8
567 4043798.0 3757116.0 4017997.83 4.01799783E8
568 4043798.0 3847983.0 4032322.85 4.03232285E8
569 4043798.0 3879440.0 4031310.21 4.03131021E8
570 4043798.0 3824205.0 4008454.89 4.00845489E8
571 4043798.0 3825751.0 4039560.7 4.0395607E8
572 4043798.0 3857053.0 4041930.55 4.04193055E8
573 4043798.0 3856354.0 4019559.42 4.01955942E8
574 4043798.0 3764447.0 4031711.69 4.03171169E8
575 4043798.0 3788225.0 3992213.35 3.99221335E8
576 4043798.0 3803741.0 3980623.55 3.98062355E8
577 4043798.0 3798650.0 4006512.39 4.00651239E8
578 4043798.0 3784971.0 3992095.37 3.99209537E8
579 4043798.0 3799278.0 4035124.36 4.03512436E8
580 4043798.0 3799278.0 3994199.38 3.99419938E8
581 4043798.0 3797855.0 3999266.02 3.99926602E8
582 4043798.0 3781161.0 3995835.67 3.99583567E8
583 4043798.0 3925737.0 4042162.83 4.04216283E8
584 4043798.0 3915988.0 4042519.9 4.0425199E8
585 4043798.0 3794798.0 4007498.88 4.00749888E8
586 4043798.0 3780908.0 4033687.51 4.03368751E8
587 4043798.0 3840113.0 4023590.44 4.02359044E8
588 4043798.0 3829053.0 4025085.47 4.02508547E8
589 4043798.0 3786981.0 4019124.62 4.01912462E8
590 4043798.0 3781283.0 4008018.86 4.00801886E8
591 4043798.0 4043798.0 4043798.0 4.043798E8
592 4044000.0 3859993.0 4036447.82 4.03644782E8
593 4044000.0 3850941.0 4031472.44 4.03147244E8
594 4044000.0 3802324.0 4015635.25 4.01563525E8
595 4044000.0 3796560.0 4017599.4 4.0175994E8
596 4044000.0 3911792.0 4037465.12 4.03746512E8
597 4044000.0 3778019.0 4021966.21 4.02196621E8
598 4044000.0 3860805.0 4020254.1 4.0202541E8
599 4044266.0 3821193.0 4035774.55 4.03577455E8
600 4044266.0 3838804.0 4016755.4 4.0167554E8
601 4044266.0 3814685.0 4017333.59 4.01733359E8
602 4044266.0 3796879.0 4030777.91 4.03077791E8
603 4044266.0 3821505.0 4025138.08 4.02513808E8
```

```
604 4044266.0 3842497.0 4029527.35 4.02952735E8
605 4044266.0 3812804.0 4009275.7 4.0092757E8
606 4044266.0 3814135.0 4017667.78 4.01766778E8
607 4044266.0 3825693.0 4022653.17 4.02265317E8
608 4044266.0 3802911.0 4005665.11 4.00566511E8
609 4044266.0 3818502.0 4002203.56 4.00220356E8
610 4044266.0 3816844.0 4016912.28 4.01691228E8
611 4044266.0 3756272.0 3988667.86 3.98866786E8
612 4044266.0 3773691.0 4008730.21 4.00873021E8
613 4044266.0 3823642.0 4008584.88 4.00858488E8
614 4044266.0 3823040.0 3996648.39 3.99664839E8
615 4044266.0 3857979.0 4017873.04 4.01787304E8
616 4044266.0 3836935.0 3978312.14 3.97831214E8
617 4044266.0 3786084.0 4018609.24 4.01860924E8
618 4044266.0 3847435.0 4009711.73 4.00971173E8
619 4052276.0 3807180.0 4011906.09 4.01190609E8
620 4052276.0 3784153.0 3988627.12 3.98862712E8
621 4052276.0 3785964.0 3970362.63 3.97036263E8
622 4052276.0 3828689.0 4032636.74 4.03263674E8
623 4052276.0 3794841.0 4002523.45 4.00252345E8
624 4052276.0 3798619.0 4012657.63 4.01265763E8
625 4052276.0 3823861.0 4013448.16 4.01344816E8
626 4052276.0 3810634.0 4034240.39 4.03424039E8
627 4052276.0 3794841.0 4040882.3 4.0408823E8
628 4052276.0 3778612.0 3988428.06 3.98842806E8
629 4052597.0 3794841.0 4014675.6 4.0146756E8
630 4052597.0 3785588.0 4020428.26 4.02042826E8
631 4052597.0 3785588.0 4004500.74 4.00450074E8
632 4052597.0 3789988.0 4008684.74 4.00868474E8
633 4052597.0 3789988.0 4003165.61 4.00316561E8
634 4052597.0 3818411.0 4018413.03 4.01841303E8
635 4052665.0 3790278.0 4004317.47 4.00431747E8
636 4052665.0 3790102.0 4011527.72 4.01152772E8
637 4052665.0 3803869.0 4027233.95 4.02723395E8
638 4052665.0 3811159.0 4002684.66 4.00268466E8
639 4052665.0 3793221.0 4028190.32 4.02819032E8
640 4052665.0 3811159.0 4015207.05 4.01520705E8
641 4052665.0 3811159.0 4033705.53 4.03370553E8
642 4052665.0 3793801.0 4033950.02 4.03395002E8
643 4052665.0 3801993.0 3993544.67 3.99354467E8
644 4052665.0 3811159.0 4014710.54 4.01471054E8
645 4052665.0 3799619.0 4030768.92 4.03076892E8
646 4052665.0 3778004.0 4019978.55 4.01997855E8
647 4052665.0 3811159.0 4021122.07 4.02112207E8
648 4053279.0 3780273.0 4029978.72 4.02997872E8
649 4053279.0 3780273.0 4009821.75 4.00982175E8
650 4053279.0 3790078.0 3983935.22 3.98393522E8
651 4053279.0 3790078.0 4015694.02 4.01569402E8
652 4053279.0 3785175.0 3999535.55 3.99953555E8
653 4053279.0 3790078.0 4011968.39 4.01196839E8
654 4053279.0 3772632.0 3975128.51 3.97512851E8
655 4053279.0 3788560.0 3992427.27 3.99242727E8
656 4053279.0 3790078.0 4019289.25 4.01928925E8
657 4053279.0 3790078.0 4006826.48 4.00682648E8
658 4053279.0 3753740.0 4016122.74 4.01612274E8
659 4053279.0 3775526.0 3965299.62 3.96529962E8
660 4053279.0 3821357.0 4009514.04 4.00951404E8
```

```
661 4053279.0 3779677.0 4024894.95 4.02489495E8
662 4053279.0 3790078.0 4008790.99 4.00879099E8
663 4053279.0 3764655.0 4009511.33 4.00951133E8
664 4053279.0 3790078.0 4021145.57 4.02114557E8
665 4053279.0 3790078.0 4001821.18 4.00182118E8
666 4053279.0 3773311.0 3996111.42 3.99611142E8
667 4053279.0 3802155.0 4017844.09 4.01784409E8
668 4053279.0 3790078.0 4003189.27 4.00318927E8
669 4053279.0 3790078.0 4000608.75 4.00060875E8
670 4053279.0 3790078.0 3999601.86 3.99960186E8
671 4053279.0 3790078.0 3986174.77 3.98617477E8
672 4053279.0 3768674.0 4001673.7 4.0016737E8
673 4053279.0 3790078.0 3985922.05 3.98592205E8
674 4053279.0 3790078.0 3985741.65 3.98574165E8
675 4053724.0 3752412.0 3996000.64 3.99600064E8
676 4053724.0 3786036.0 3981312.54 3.98131254E8
677 4053724.0 3790078.0 4007089.7 4.0070897E8
678 4053724.0 3765055.0 3989062.54 3.98906254E8
679 4053724.0 3779163.0 4004650.08 4.00465008E8
680 4053724.0 3839482.0 4036469.41 4.03646941E8
681 4053724.0 3790439.0 4020693.98 4.02069398E8
682 4053724.0 3811474.0 4030240.51 4.03024051E8
683 4054257.0 3839482.0 4027063.44 4.02706344E8
684 4054257.0 3814285.0 4020683.2 4.0206832E8
685 4054257.0 3768085.0 4042431.14 4.04243114E8
686 4054257.0 3785030.0 4032608.03 4.03260803E8
687 4055259.0 3785030.0 4047232.49 4.04723249E8
688 4055259.0 3839636.0 4044974.23 4.04497423E8
689 4055259.0 3768729.0 4009757.99 4.00975799E8
690 4055259.0 3790758.0 3969473.38 3.96947338E8
691 4055259.0 3801885.0 4003532.04 4.00353204E8
692 4055259.0 3788250.0 3991623.73 3.99162373E8
693 4055259.0 3790812.0 4008040.87 4.00804087E8
694 4055259.0 3786989.0 4009674.12 4.00967412E8
695 4055259.0 3801885.0 4015913.68 4.01591368E8
696 4055259.0 3801885.0 4034834.21 4.03483421E8
697 4055259.0 3774604.0 4022442.34 4.02244234E8
698 4055259.0 3801885.0 4015440.97 4.01544097E8
699 4055259.0 3801885.0 3992313.67 3.99231367E8
700 4055259.0 3801885.0 3984869.7 3.9848697E8
701 4055259.0 3794590.0 3949281.14 3.94928114E8
702 4055259.0 3801885.0 4007194.88 4.00719488E8
703 4055259.0 3801885.0 4022149.51 4.02214951E8
704 4055259.0 3790853.0 3997317.13 3.99731713E8
705 4055259.0 3790693.0 3990598.66 3.99059866E8
706 4055259.0 3801885.0 3990875.25 3.99087525E8
707 4055259.0 3801885.0 3969997.94 3.96999794E8
708 4055259.0 3789574.0 3995525.23 3.99552523E8
709 4055259.0 3801885.0 4008851.55 4.00885155E8
710 4055259.0 3799787.0 3996879.39 3.99687939E8
711 4055259.0 3801885.0 4006175.79 4.00617579E8
712 4055259.0 3776850.0 3981857.38 3.98185738E8
713 4055259.0 3801885.0 3985589.56 3.98558956E8
714 4055551.0 3801885.0 3990467.08 3.99046708E8
715 4055551.0 3791007.0 3991103.28 3.99110328E8
716 4055551.0 3801885.0 3998720.97 3.99872097E8
717 4055551.0 3778489.0 4045457.48 4.04545748E8
```

718	4055551.0	3745094.0	4012318.42	4.01231842E8
719	4055551.0	3796256.0	4024674.71	4.02467471E8
720	4055551.0	3815777.0	4042021.38	4.04202138E8
721	4055551.0	3771063.0	4033109.62	4.03310962E8
722	4055551.0	3771063.0	4041571.44	4.04157144E8
723	4055801.0	4055551.0	4055553.5	4.0555535E8
724	4055801.0	3797286.0	4039355.03	4.03935503E8
725	4055801.0	3833068.0	4036242.06	4.03624206E8
726	4055801.0	3794281.0	4023561.83	4.02356183E8
727	4055801.0	3768026.0	4018579.17	4.01857917E8
728	4055801.0	3794281.0	4023816.84	4.02381684E8
729	4055801.0	3794281.0	4005877.76	4.00587776E8
730	4055801.0	3782033.0	4014627.02	4.01462702E8
731	4055801.0	3794281.0	4032826.24	4.03282624E8
732	4055801.0	3794281.0	4037368.99	4.03736899E8
733	4055801.0	3794281.0	4028041.79	4.02804179E8
734	4055801.0	3759211.0	4011810.29	4.01181029E8
735	4055801.0	3789582.0	3998466.81	3.99846681E8
736	4055801.0	3794281.0	4025743.72	4.02574372E8
737	4055801.0	3830310.0	4044613.75	4.04461375E8
738	4055801.0	3793285.0	3990402.42	3.99040242E8
739	4055801.0	3794281.0	4037729.3	4.0377293E8
740	4055801.0	3794281.0	4021274.69	4.02127469E8
741	4055801.0	3816833.0	4049548.85	4.04954885E8
742	4055801.0	3788504.0	4044350.34	4.04435034E8
743	4055801.0	3787508.0	4006627.89	4.00662789E8
744	4055801.0	3786357.0	4001956.02	4.00195602E8
745	4055801.0	3781902.0	4020903.52	4.02090352E8
746	4055801.0	3794281.0	4029825.4	4.0298254E8
747	4055801.0	3794281.0	4041284.86	4.04128486E8
748	4055801.0	3794281.0	4010086.26	4.01008626E8
749	4055801.0	3766402.0	3988410.22	3.98841022E8
750	4055801.0	3785097.0	4015852.58	4.01585258E8
751	4055801.0	3794281.0	4043940.06	4.04394006E8
752	4055801.0	3820749.0	4038088.25	4.03808825E8
753	4055801.0	3794281.0	4023394.17	4.02339417E8
754	4055801.0	3781357.0	3997563.12	3.99756312E8
755	4055801.0	3771243.0	4017100.66	4.01710066E8
756	4057277.0	3772474.0	4007069.51	4.00706951E8
757	4057277.0	3792524.0	3989718.82	3.98971882E8
758	4057277.0	3794281.0	4026386.56	4.02638656E8
759	4057277.0	3781332.0	3959531.39	3.95953139E8
760	4057277.0	3769163.0	3999899.78	3.99989978E8
761	4057277.0	3776735.0	3989354.28	3.98935428E8
762	4057277.0	3808243.0	4038947.08	4.03894708E8
763	4057277.0	3796557.0	3996168.37	3.99616837E8
764	4057277.0	3755596.0	4035371.99	4.03537199E8
765	4057277.0	3811665.0	4014736.46	4.01473646E8
766	4057277.0	3786721.0	4032275.5	4.0322755E8
767	4057277.0	3798901.0	4014368.62	4.01436862E8
768	4057277.0	3774959.0	4002269.5	4.0022695E8
769	4057277.0	3811665.0	4000451.19	4.00045119E8
770	4057277.0	3796643.0	4024591.26	4.02459126E8
771	4057570.0	3793980.0	4041649.25	4.04164925E8
772	4057570.0	3811665.0	4036960.46	4.03696046E8
773	4058088.0	3760584.0	4013184.1	4.0131841E8
774	4058088.0	3796820.0	4039795.42	4.03979542E8

```
775 4058088.0 3800127.0 4029756.76 4.02975676E8
776 4058088.0 3790902.0 3998822.56 3.99882256E8
777 4058088.0 3774729.0 4021844.43 4.02184443E8
778 4058088.0 3814341.0 4024196.13 4.02419613E8
779 4058088.0 3894198.0 4047982.28 4.04798228E8
780 4058088.0 3859256.0 4037440.42 4.03744042E8
781 4058088.0 3830524.0 4034350.81 4.03435081E8
782 4058088.0 3830524.0 4042807.69 4.04280769E8
783 4058088.0 3830524.0 4034075.82 4.03407582E8
784 4058088.0 3754295.0 4000153.89 4.00015389E8
785 4058088.0 3805646.0 4005182.13 4.00518213E8
786 4058088.0 3800127.0 4038585.87 4.03858587E8
787 4058088.0 3764865.0 4009648.23 4.00964823E8
788 4058088.0 3827363.0 4044197.84 4.04419784E8
789 4058088.0 3784657.0 4034092.33 4.03409233E8
790 4058088.0 3789763.0 4013376.2 4.0133762E8
791 4058088.0 3789763.0 4014811.55 4.01481155E8
792 4058088.0 3825273.0 4015129.03 4.01512903E8
793 4058088.0 3830524.0 4039933.8 4.0399338E8
794 4058088.0 3809972.0 4038542.79 4.03854279E8
795 4058088.0 3784586.0 4008208.02 4.00820802E8
796 4058088.0 3779131.0 4036665.32 4.03666532E8
797 4058088.0 3830524.0 4031898.09 4.03189809E8
798 4058088.0 3830524.0 4031147.27 4.03114727E8
799 4058088.0 3816844.0 4036563.12 4.03656312E8
800 4058088.0 3800185.0 4026446.59 4.02644659E8
801 4058088.0 3830524.0 4046395.4 4.0463954E8
802 4058088.0 3830524.0 4048031.28 4.04803128E8
803 4058088.0 3830524.0 4025046.25 4.02504625E8
804 4058088.0 3802780.0 4018301.52 4.01830152E8
805 4058088.0 3795695.0 4019853.59 4.01985359E8
806 4058088.0 3792100.0 4032590.36 4.03259036E8
807 4058088.0 3783872.0 4002750.28 4.00275028E8
808 4058088.0 3830524.0 4046668.9 4.0466689E8
809 4058088.0 3798272.0 4021324.65 4.02132465E8
810 4058088.0 3819476.0 4019048.09 4.01904809E8
811 4058088.0 3821905.0 4007119.34 4.00711934E8
812 4058088.0 3789232.0 4004351.47 4.00435147E8
813 4058088.0 3797863.0 4034258.3 4.0342583E8
814 4058088.0 3797863.0 4030141.57 4.03014157E8
815 4058088.0 3830524.0 4040686.04 4.04068604E8
816 4058088.0 3830524.0 4032586.14 4.03258614E8
817 4058088.0 3788480.0 4013366.75 4.01336675E8
818 4058088.0 3801765.0 4028536.14 4.02853614E8
819 4058088.0 3747244.0 4029277.82 4.02927782E8
820 4058088.0 3747244.0 4027907.72 4.02790772E8
821 4058088.0 3792602.0 4023094.11 4.02309411E8
822 4058088.0 3787688.0 4028607.96 4.02860796E8
823 4058088.0 3830524.0 4045502.9 4.0455029E8
824 4058680.0 3762628.0 4038363.52 4.03836352E8
825 4058680.0 3776702.0 4037768.42 4.03776842E8
826 4058680.0 3797599.0 4022041.7 4.0220417E8
827 4058680.0 3778367.0 4003781.61 4.00378161E8
828 4058680.0 3773759.0 4017631.64 4.01763164E8
829 4058680.0 3805450.0 4010731.25 4.01073125E8
830 4058680.0 3794751.0 3999547.72 3.99954772E8
831 4058680.0 3813350.0 4018337.44 4.01833744E8
```

```
832 4058680.0 3777085.0 4013498.21 4.01349821E8
833 4058680.0 3806982.0 4040466.49 4.04046649E8
834 4058680.0 3802910.0 4029418.58 4.02941858E8
835 4058680.0 3786355.0 4029562.89 4.02956289E8
836 4058680.0 3894345.0 4044049.12 4.04404912E8
837 4058680.0 3855936.0 4050465.74 4.05046574E8
838 4058680.0 3846762.0 4056012.02 4.05601202E8
839 4058680.0 3829849.0 4056391.69 4.05639169E8
840 4058680.0 3825607.0 4050805.57 4.05080557E8
841 4058680.0 3825607.0 4022156.2 4.0221562E8
842 4058680.0 3905342.0 4052491.12 4.05249112E8
843 4058680.0 3838023.0 4040925.56 4.04092556E8
844 4058680.0 3825763.0 4049499.87 4.04949987E8
845 4058680.0 3788915.0 4042609.41 4.04260941E8
846 4058680.0 3793986.0 4022327.96 4.02232796E8
847 4058680.0 3786355.0 4014578.84 4.01457884E8
848 4058680.0 3786355.0 4045179.26 4.04517926E8
849 4058680.0 3790990.0 4022805.07 4.02280507E8
850 4058680.0 3821596.0 4056309.16 4.05630916E8
851 4058680.0 3785259.0 4029199.15 4.02919915E8
852 4058680.0 3785259.0 4051163.87 4.05116387E8
853 4058680.0 3865528.0 4054158.7 4.0541587E8
854 4058680.0 3892542.0 4041845.79 4.04184579E8
855 4058680.0 3815267.0 4037383.08 4.03738308E8
856 4058680.0 3818894.0 4046965.86 4.04696586E8
857 4063502.0 3818894.0 4020551.78 4.02055178E8
858 4063502.0 3761163.0 4044350.93 4.04435093E8
859 4063502.0 3768201.0 4033218.87 4.03321887E8
860 4063502.0 3818894.0 3997916.61 3.99791661E8
861 4063502.0 3793742.0 4014659.02 4.01465902E8
862 4063502.0 3796592.0 4021019.21 4.02101921E8
863 4063502.0 3796592.0 4036616.75 4.03661675E8
864 4063502.0 3778286.0 4032926.06 4.03292606E8
865 4063502.0 3818894.0 4053953.51 4.05395351E8
866 4063502.0 3788518.0 4026700.92 4.02670092E8
867 4063502.0 3818894.0 4047384.65 4.04738465E8
868 4063502.0 3780967.0 4023237.66 4.02323766E8
869 4063502.0 3782073.0 4018928.9 4.0189289E8
870 4063502.0 3767700.0 4033852.0 4.033852E8
871 4063502.0 3818894.0 4049108.98 4.04910898E8
872 4063502.0 3818894.0 4044807.27 4.04480727E8
873 4063502.0 3787122.0 4021214.44 4.02121444E8
874 4063502.0 3805742.0 4016511.45 4.01651145E8
875 4063502.0 3818894.0 4032622.74 4.03262274E8
876 4063502.0 3806495.0 4005095.7 4.0050957E8
877 4063502.0 3818894.0 4025625.39 4.02562539E8
878 4063502.0 3818894.0 4042621.36 4.04262136E8
879 4063502.0 3818894.0 4017080.44 4.01708044E8
880 4063502.0 3803467.0 4042827.05 4.04282705E8
881 4063502.0 3780805.0 4016531.6 4.0165316E8
882 4063502.0 3776624.0 4010013.91 4.01001391E8
883 4063502.0 3813165.0 4042761.59 4.04276159E8
884 4063502.0 3798383.0 4013212.41 4.01321241E8
885 4063502.0 3791910.0 4042632.67 4.04263267E8
886 4063502.0 3818894.0 4014653.46 4.01465346E8
887 4063502.0 3818894.0 4019456.88 4.01945688E8
888 4063502.0 3818894.0 4047063.3 4.0470633E8
```

```
889 4063502.0 3815101.0 4012029.37 4.01202937E8
890 4063502.0 3818894.0 4041381.22 4.04138122E8
891 4063502.0 3818709.0 4045676.46 4.04567646E8
892 4063502.0 3799258.0 4038693.76 4.03869376E8
893 4063502.0 3798991.0 4025520.44 4.02552044E8
894 4063502.0 3815830.0 4014489.87 4.01448987E8
895 4063502.0 3818894.0 4048079.2 4.0480792E8
896 4063502.0 3818894.0 4039557.66 4.03955766E8
897 4063502.0 3882930.0 4047067.9 4.0470679E8
898 4063502.0 3818894.0 4027173.91 4.02717391E8
899 4063502.0 3809289.0 4013038.01 4.01303801E8
900 4063502.0 3816345.0 4027215.65 4.02721565E8
901 4063502.0 3818894.0 4015274.81 4.01527481E8
902 4063502.0 3912179.0 4055796.7 4.0557967E8
903 4063502.0 3818894.0 4051887.34 4.05188734E8
904 4063502.0 3798906.0 4016489.82 4.01648982E8
905 4063502.0 3794926.0 3989961.26 3.98996126E8
906 4063502.0 3799015.0 4018634.36 4.01863436E8
907 4063502.0 3786227.0 3999026.01 3.99902601E8
908 4063502.0 3796334.0 3996564.12 3.99656412E8
909 4063502.0 3800737.0 4017008.32 4.01700832E8
910 4063502.0 3764646.0 4005661.69 4.00566169E8
911 4063502.0 3791879.0 4021811.07 4.02181107E8
912 4063502.0 3785354.0 4010856.07 4.01085607E8
913 4063502.0 3808857.0 4009121.74 4.00912174E8
914 4063502.0 3769637.0 4021028.3 4.0210283E8
915 4063502.0 3795531.0 3991792.97 3.99179297E8
916 4063502.0 3793592.0 4030470.25 4.03047025E8
917 4063502.0 3821652.0 4033041.12 4.03304112E8
918 4063502.0 3821652.0 4012831.4 4.0128314E8
919 4063502.0 3769637.0 3991925.98 3.99192598E8
920 4063502.0 3807731.0 3996887.73 3.99688773E8
921 4063502.0 3807517.0 4006668.45 4.00666845E8
922 4063502.0 3769637.0 3990246.99 3.99024699E8
923 4063502.0 3750793.0 3990057.35 3.99005735E8
924 4063502.0 3750793.0 3973519.07 3.97351907E8
925 4063502.0 3797499.0 3985761.49 3.98576149E8
926 4063502.0 3794419.0 4013309.08 4.01330908E8
927 4063502.0 3777525.0 4004060.15 4.00406015E8
928 4063502.0 3783929.0 4015955.49 4.01595549E8
929 4063502.0 3789284.0 3990774.78 3.99077478E8
930 4063502.0 3795340.0 3991968.92 3.99196892E8
931 4063502.0 3769637.0 4003431.72 4.00343172E8
932 4063502.0 3821652.0 4025699.31 4.02569931E8
933 4063502.0 3788933.0 4017610.31 4.01761031E8
934 4063502.0 3762622.0 4004866.83 4.00486683E8
935 4063502.0 3775207.0 4008383.24 4.00838324E8
936 4063502.0 3769637.0 4004528.3 4.0045283E8
937 4063502.0 3754495.0 3994976.23 3.99497623E8
938 4063502.0 3770942.0 3986133.76 3.98613376E8
939 4063502.0 3804556.0 3998132.49 3.99813249E8
940 4063502.0 3788954.0 4000108.76 4.00010876E8
941 4063502.0 3823568.0 4003051.36 4.00305136E8
942 4063502.0 3823568.0 4039128.93 4.03912893E8
943 4063502.0 3765345.0 3975537.85 3.97553785E8
944 4063502.0 3799002.0 4007487.88 4.00748788E8
945 4063502.0 3776166.0 3988584.28 3.98858428E8
```

```
946 4063502.0 3754336.0 4035729.02 4.03572902E8
947 4063502.0 3821652.0 4011972.25 4.01197225E8
948 4063502.0 3821652.0 4017073.05 4.01707305E8
949 4063502.0 3823568.0 4022828.64 4.02282864E8
950 4063502.0 3769637.0 4009400.64 4.00940064E8
951 4063502.0 3754460.0 4004303.39 4.00430339E8
952 4063502.0 3799566.0 3996678.3 3.9966783E8
953 4063502.0 3769637.0 3974632.64 3.97463264E8
954 4063502.0 3808682.0 3991936.11 3.99193611E8
955 4063502.0 3817455.0 4022857.13 4.02285713E8
956 4063502.0 3821652.0 3997790.48 3.99779048E8
957 4063502.0 3777234.0 3989180.51 3.98918051E8
958 4063502.0 3796923.0 3986707.42 3.98670742E8
959 4063502.0 3808430.0 4007572.0 4.007572E8
960 4063502.0 3802411.0 4010382.92 4.01038292E8
961 4063502.0 3821652.0 3994400.06 3.99440006E8
962 4063502.0 3821652.0 4031161.49 4.03116149E8
963 4063502.0 3763211.0 4018662.59 4.01866259E8
964 4063502.0 3770221.0 3968416.54 3.96841654E8
965 4063502.0 3787874.0 4000145.79 4.00014579E8
966 4063502.0 3787874.0 4021384.55 4.02138455E8
967 4063502.0 3799189.0 3993620.31 3.99362031E8
968 4063502.0 3788121.0 4019925.26 4.01992526E8
969 4063502.0 3821652.0 4007889.09 4.00788909E8
970 4063502.0 3769637.0 3975779.41 3.97577941E8
971 4063502.0 3799109.0 4014809.98 4.01480998E8
972 4063502.0 3802711.0 4026670.1 4.0266701E8
973 4063502.0 3769496.0 4017841.68 4.01784168E8
974 4063502.0 3783204.0 4016646.91 4.01664691E8
975 4063502.0 3782454.0 4002545.82 4.00254582E8
976 4063502.0 3772735.0 4035863.43 4.03586343E8
977 4063502.0 3769312.0 4013843.54 4.01384354E8
978 4063502.0 3796116.0 4015059.76 4.01505976E8
979 4063502.0 3807154.0 4033514.96 4.03351496E8
980 4063502.0 3774326.0 4002301.06 4.00230106E8
981 4063502.0 3799566.0 4010083.52 4.01008352E8
982 4063502.0 3769637.0 3981159.99 3.98115999E8
983 4063502.0 3813727.0 4012642.95 4.01264295E8
984 4063502.0 3793156.0 4013993.98 4.01399398E8
985 4063502.0 3807330.0 4021903.11 4.02190311E8
986 4063502.0 3813282.0 4006153.77 4.00615377E8
987 4063502.0 3821652.0 4025919.77 4.02591977E8
988 4063502.0 3813439.0 3992024.1 3.9920241E8
989 4063502.0 3821652.0 4017792.22 4.01779222E8
990 4063502.0 3821652.0 4028635.98 4.02863598E8
991 4063502.0 3821652.0 4007784.89 4.00778489E8
992 4063502.0 3766812.0 3971636.27 3.97163627E8
993 4063502.0 3807305.0 4026502.88 4.02650288E8
994 4063502.0 3799566.0 4009535.72 4.00953572E8
995 4063502.0 3775172.0 4007197.78 4.00719778E8
996 4063502.0 3775172.0 3986965.09 3.98696509E8
997 4063502.0 3812078.0 3993980.13 3.99398013E8
998 4063502.0 3823568.0 4003056.2 4.0030562E8
999 4063502.0 3856515.0 4045438.33 4.04543833E8
```

APPENDIX H: POPULATION SIZE SENSITIVITY STUDY RESULTS

The purpose of this study is to sensitise the population size to be used with the OGA. Samples were collected ranging from 20, 40, 60, ..., 180, 200. Each repeated 10 times. Statistical variance was computed to yield the standard deviation.

Results are plotted, and graphs are shown in chapter 5.

Table Convention:

AVG20 = Population Size = 20; Taking the average fitness from a generation G0 (initial) and G99 (Last generation).

AVG 20

G0	G99	G99-G0	max99
3830792.3	3933602.15	102809.85	3936247
3825972.2	3860012.65	34040.45	3910448
3821434.75	3870921.6	49486.85	3929143
3833481.25	3915963	82481.75	3915963
3823315.6	3903976.75	80661.15	3913612
3821029.45	3910450.7	89421.25	3920144
3828208.15	3883027.4	54819.25	3915267
3829318.5	3892540.55	63222.05	3901028
3818548.8	3894941.7	76392.9	3922023
3813187.8	3894680	81492.2	3916858
3824528.88	**3896011.65**	**71482.77**	**3918073.3**
	STD DEVIATION	20721.30817	9747.39765

AVG40

G0	G99	G99-G0	max99
3822497	3923421	100923.9	3939917
3820793	3933412	112619.1	3950490
3819948	3913835	93887.08	3913835
3823902	3936881	112978.7	3950833
3825644	3900098	74453.92	3922565
3823472	3903842	80370	3915289

3826764	3957115	130351	3969705
3832395	3911092	78696.67	3958881
3819423	3919599	100176.2	3927473
3821900	3928113	106212.7	3933116
3823674	**3922741**	**99066.91**	**3938210**
	STD DEVIATION	17665.98	18967.19

AVG60

G0	G99	G99-G0	max99
3825221	3939330	114109.3	3946700
3817889	3940996	123107.5	3954094
3817486	3921570	104084.4	3928818
3819234	3901806	82571.43	3930690
3829879	3901471	71591.95	3934468
3827485	3885649	58163.85	3917371
3816501	3924023	107521.3	3931652
3820623	3920718	100095.1	3928882
3824537	3898241	73704.52	3917003
3818169	3894906	76736.85	3926970
3821702	**3912871**	**91168.62**	**3931665**
	STD DEVIATION	21395.8	11523.43

AVG80

G0	G99	G99-G0	max99
3822366	3915926	93559.56	3942020
3821069	3907355	86285.56	3919846
3822633	3943698	121065.1	3945724
3819473	3914845	95371.5	3926666
3823398	3922418	99020.45	3938449
3822716	3899673	76956.68	3943291
3820429	3959626	139197.4	3968936
3824538	3966016	141478.1	3978027
3822510	3884631	62121.05	3938947
3819058	3924598	105540.2	3941538
3821819	**3923879**	**102059.6**	**3944344**
	STD DEVIATION	25618.23	17423.3

AVG100

G0	G99	G99-G0	max99
3825670	3899373	73702.8	3936756
3825841	3939112	113270.7	3951377
3823031	3936373	113341.8	3947872
3823836	3980783	156946.9	3984460
3825321	3926209	100888.5	3937383
3828976	3934934	105958.5	3959251

3814923	3956792	141868.9	3972587
3821582	3938929	117347.4	3957831
3823177	3957543	134366	3962389
3823639	3911035	87396.78	3928390
3823600	**3938108**	**114508.8**	**3953830**
	STD DEVIATION	24974.74	17189.81

AVG120

G0	**G99**	**G99-G0**	**max99**
3826057	3933453	107395.2	3954884
3820866	3923559	102693	3936350
3824684	3930949	106264.9	3934452
3827391	3925090	97698.61	3944846
3821615	3938249	116634.4	3959748
3824998	3911366	86368.45	3936578
3823420	3934542	111122.6	3943754
3821137	3945557	124419.8	3964930
3828061	3936789	108728.4	3945608
3827023	3919591	92568.68	3938096
3824525	**3929915**	**105389.4**	**3945925**
	STD DEVIATION	11205.16	10596

AVG140

G0	**G99**	**G99-G0**	**max99**
3824551	3913038	88487.15	3956850
3818457	3941313	122855.3	3962341
3823391	3945219	121827.8	3951445
3821009	3965063	144053.9	3986401
3823355	3930168	106812.2	3941963
3821455	3980393	158938.5	3990088
3823284	3920200	96916.22	3942536
3824840	3927711	102871	3931200
3824010	3942314	118304.1	3951102
3824387	3924139	99751.98	3937280
3822874	**3938956**	**116081.8**	**3955121**
	STD DEVIATION	22007.25	19742.63

AVG160

G0	**G99**	**G99-G0**	**max99**
3823594	3929629	106034.7	3945559
3823164	3937608	114443.3	3943241
3826822	3917103	90280.68	3950249
3821135	3931637	110502.8	3945430
3822872	3911464	88591.71	3938385
3821240	3927763	106523	3956812

3822250	3931887	109637.9	3936612
3817569	3953902	136332.6	3956595
3822146	3926754	104608.1	3946767
3822975	3967286	144311.4	3977192
3822377	**3933503**	**111126.6**	**3949684**
	STD DEVIATION	17582.07	11734.36

AVG180

G0	**G99**	**G99-G0**	**max99**
3826049	3913752	87703.23	3932887
3825003	3922598	97595.41	3947478
3818298	3961478	143180	3977931
3822220	3924445	102224.4	3950854
3824393	3958317	133924	3975683
3820643	3943659	123015.9	3945433
3825688	3892683	66995.35	3940871
3822322	3941979	119656.9	3948312
3824929	3937218	112288.4	3946700
3822351	3924433	102082.6	3931628
3823190	**3932056**	**108866.6**	**3949778**
	STD DEVIATION	22489.25	15615.78

AVG200

G0	**G99**	**G99-G0**	**max99**
3820020	3950002	129981.6	3952868
3826445	3884194	57748.78	3942741
3826167	3959774	133607.5	3964071
3824075	3934044	109968.6	3950913
3823123	3936179	113056.5	3968072
3820533	3909199	88666.35	3929742
3819941	3937659	117717.9	3956933
3818450	3942479	124029	3951740
3822659	3868521	45862.15	3934340
3824578	3894631	70052.87	3927368
3822599	**3921668**	**99069.13**	**3947879**
	STD DEVIATION	31485.83	13985.19

Summary:

POP SIZE	G0	G99	G99-G0	max99
20	3824528.88	3896011.65	71482.77	3918073
40	3823673.94	3922740.848	99066.9075	3938210
60	3821702.302	3912870.922	91168.62	3931665
80	3821819.069	3923878.634	102059.565	3944344
100	3823599.556	3938108.383	114508.827	3953830
120	3824525.17	3929914.573	105389.403	3945925
140	3822873.859	3938955.681	116081.822	3955121
160	3822376.733	3933503.344	111126.612	3949684
180	3823189.53	3932056.157	108866.627	3949778
200	3822599.102	3921668.234	99069.132	3947879

Sample execution of results:

```
Optical Genetic Algorithm OGA:
----------------------------
(c)1999 Ziad Kobti. All Rights Reserved.

Population Size  : 100
Chromosome Length: 100
Greedy Crossover
Two point swap mutation

G MAX       MIN       AVG       TotFit
0 3885126.0 3737163.0 3825669.93 3.82566993E8
1 3885126.0 3766535.0 3834268.4 3.8342684E8
2 3885126.0 3767179.0 3837363.24 3.83736324E8
3 3888830.0 3759185.0 3833651.86 3.83365186E8
4 3911339.0 3756472.0 3842758.03 3.84275803E8
5 3911339.0 3752945.0 3834955.05 3.83495505E8
6 3911339.0 3732666.0 3839868.11 3.83986811E8
7 3911339.0 3750188.0 3838528.68 3.83852868E8
8 3911339.0 3753097.0 3841161.0 3.841161E8
9 3911339.0 3764751.0 3845344.78 3.84534478E8
10 3911339.0 3751126.0 3842566.98 3.84256698E8
11 3911339.0 3729296.0 3840027.29 3.84002729E8
12 3911339.0 3742388.0 3838416.38 3.83841638E8
13 3911339.0 3746224.0 3844257.96 3.84425796E8
14 3911339.0 3750510.0 3843060.48 3.84306048E8
15 3911339.0 3764938.0 3844752.79 3.84475279E8
16 3911339.0 3754280.0 3847068.79 3.84706879E8
17 3911339.0 3756851.0 3843201.06 3.84320106E8
18 3911339.0 3753324.0 3847454.95 3.84745495E8
19 3911339.0 3750282.0 3846744.19 3.84674419E8
20 3911339.0 3746425.0 3845289.16 3.84528916E8
21 3911339.0 3752013.0 3843603.8 3.8436038E8
22 3914270.0 3734256.0 3841464.85 3.84146485E8
23 3914270.0 3754353.0 3842380.43 3.84238043E8
24 3914270.0 3754047.0 3843290.93 3.84329093E8
25 3914270.0 3767811.0 3845231.72 3.84523172E8
```

```
26 3914270.0 3766055.0 3845195.82 3.84519582E8
27 3914270.0 3744840.0 3847116.72 3.84711672E8
28 3914270.0 3753027.0 3845924.7 3.8459247E8
29 3914270.0 3739779.0 3845043.68 3.84504368E8
30 3914270.0 3778394.0 3845717.78 3.84571778E8
31 3914270.0 3764189.0 3848180.71 3.84818071E8
32 3926368.0 3740277.0 3852995.54 3.85299554E8
33 3926368.0 3765800.0 3850479.77 3.85047977E8
34 3926368.0 3748991.0 3850794.34 3.85079434E8
35 3926368.0 3761408.0 3858952.18 3.85895218E8
36 3926368.0 3759868.0 3861694.65 3.86169465E8
37 3926368.0 3769461.0 3873167.73 3.87316773E8
38 3926368.0 3792707.0 3878697.07 3.87869707E8
39 3926368.0 3771427.0 3864842.26 3.86484226E8
40 3926368.0 3773748.0 3864179.94 3.86417994E8
41 3926368.0 3771442.0 3870595.73 3.87059573E8
42 3926368.0 3777362.0 3872697.85 3.87269785E8
43 3926368.0 3784321.0 3867280.25 3.86728025E8
44 3926368.0 3777151.0 3872581.21 3.87258121E8
45 3926368.0 3773125.0 3873163.97 3.87316397E8
46 3926368.0 3765912.0 3872530.34 3.87253034E8
47 3926368.0 3780784.0 3867673.54 3.86767354E8
48 3926368.0 3777151.0 3875366.26 3.87536626E8
49 3926368.0 3779089.0 3868404.88 3.86840488E8
50 3926368.0 3763038.0 3864622.8 3.8646228E8
51 3926368.0 3776827.0 3862260.09 3.86226009E8
52 3926368.0 3773907.0 3863155.53 3.86315553E8
53 3926368.0 3774116.0 3859613.87 3.85961387E8
54 3926368.0 3765392.0 3864135.1 3.8641351E8
55 3926368.0 3779924.0 3862031.53 3.86203153E8
56 3926368.0 3768281.0 3860254.45 3.86025445E8
57 3926368.0 3776057.0 3858608.86 3.85860886E8
58 3926368.0 3766957.0 3861819.09 3.86181909E8
59 3926368.0 3751630.0 3854490.02 3.85449002E8
60 3926368.0 3794238.0 3863039.99 3.86303999E8
61 3926368.0 3776039.0 3862202.47 3.86220247E8
62 3926368.0 3779249.0 3871886.06 3.87188606E8
63 3926368.0 3770496.0 3861854.26 3.86185426E8
64 3936756.0 3788181.0 3863034.37 3.86303437E8
65 3936756.0 3775720.0 3860565.19 3.86056519E8
66 3936756.0 3772058.0 3869447.62 3.86944762E8
67 3936756.0 3793898.0 3897031.51 3.89703151E8
68 3936756.0 3774047.0 3898962.14 3.89896214E8
69 3936756.0 3797967.0 3896090.84 3.89609084E8
70 3936756.0 3833777.0 3903620.85 3.90362085E8
71 3936756.0 3771415.0 3900823.38 3.90082338E8
72 3936756.0 3820259.0 3904424.8 3.9044248E8
73 3936756.0 3795319.0 3890442.88 3.89044288E8
74 3936756.0 3817681.0 3906435.49 3.90643549E8
75 3936756.0 3795382.0 3896926.26 3.89692626E8
76 3936756.0 3804034.0 3907516.28 3.90751628E8
77 3936756.0 3802901.0 3894193.13 3.89419313E8
78 3936756.0 3797420.0 3897313.09 3.89731309E8
79 3936756.0 3758684.0 3894995.33 3.89499533E8
80 3936756.0 3785316.0 3901029.67 3.90102967E8
81 3936756.0 3775737.0 3889274.93 3.88927493E8
82 3936756.0 3800737.0 3899871.86 3.89987186E8
```

```
83 3936756.0 3779603.0 3904793.82 3.90479382E8
84 3936756.0 3800700.0 3900542.97 3.90054297E8
85 3936756.0 3771368.0 3902132.41 3.90213241E8
86 3936756.0 3800052.0 3914238.39 3.91423839E8
87 3936756.0 3800052.0 3910101.81 3.91010181E8
88 3936756.0 3781560.0 3902567.98 3.90256798E8
89 3936756.0 3781560.0 3902970.91 3.90297091E8
90 3936756.0 3821293.0 3911617.09 3.91161709E8
91 3936756.0 3815837.0 3904569.88 3.90456988E8
92 3936756.0 3823543.0 3912294.22 3.91229422E8
93 3936756.0 3763447.0 3904687.49 3.90468749E8
94 3936756.0 3763447.0 3910084.68 3.91008468E8
95 3936756.0 3797943.0 3918585.93 3.91858593E8
96 3936756.0 3820186.0 3915944.05 3.91594405E8
97 3936756.0 3790157.0 3898334.75 3.89833475E8
98 3936756.0 3780329.0 3900750.77 3.90075077E8
99 3936756.0 3780329.0 3899372.73 3.89937273E8
```

VITA AUCTORIS

- Born in Lebanon (1975)
- F.J. Brennan High School (1989-1992)
- Bachelor of Biological Sciences and Computer Science, Honours (1992-1996)
- Master of Computer Science (1996 -)
- Part Time Faculty and Corporate Trainer - St. Clair College of Applied Arts and Technology (1998 -)
- Programmer Analyst / Computer Programmer (1997 -)