THE APPLICATION OF A GENETIC ALGORITHM TO A SCHEDULING PROBLEM

KENT JOSEPH HOSTUK

1995

THE APPLICATION OF A GENETIC ALGORITHM TO A SCHEDULING PROBLEM

A Thesis Submitted to the College of
Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Civil Engineering
University of Saskatchewan
Saskatoon

By

Kent Joseph Kostuk

August, 1995

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Civil Engineering
University of Saskatchewan
Saskatoon, Saskatchewan S7N 0W0

ABSTRACT

Over the past three decades a significant amount of time and effort has been expended in an attempt to optimize complex scheduling problems as a way to reduce costs. These scheduling problems are often difficult to solve because of their combinatorial nature.

Many Civil Engineering problems deal with the logistics of coordinating the movement of goods or people between various modes of transportation. Problems of this type, which can be classified as Doubly Constrained Traveling Salesman Problems (DCTSP), are particularly difficult because the deliveries must be made within prespecified windows of opportunity.

The Genetic Algorithm (GA) has been identified as a method to solve combinatorial problems. Its capability to solve complex scheduling problems has been explored herein by applying the GA technique to a complex real world scheduling problem which can be modeled as a DCTSP. The problem selected was the design of the National Hockey League's (NHL) 1992-1993 playing schedule. The NHL playing schedule was selected because: it is easily modeled as a DCTSP, the data necessary to formulate the problem was readily available, and the existing solution could act as a benchmark to measure the GA's performance.

To illustrate the robustness of the GA it was applied to a variety of classical Operations Research (OR) problems. Solving the OR problems provided an excellent opportunity to compare the performance of various GA modeling techniques. When applying the GA to the NHL scheduling problem, the

problem size and complexity was increased incrementally. Initially the GA was used to optimize the schedule only on distance. Next, optimization was based on meeting a subset of the NHL's constraints as well as minimizing distance traveled. Finally, the GA was used to optimize the schedule by meeting the complete set of constraints and minimizing the distance traveled.

The GA illustrated its flexibility by solving the OR test suite with minimal modifications. As the problems became more difficult new chromosome structures and reproduction schemes were introduced to improve the GA's performance. In the NHL scheduling application the GA was able to create scheduling solutions but it was unable to improve upon the schedule recently developed using the Decision Support System now in use by the NHL.

The GA's failure was attributed to one, or a combination of, three factors: limited computer resources, chromosomes were not adequate representations for the problem, and the problem space was either too large or deceptive for the GA to find a local optima.

The GA is a robust tool. It can solve a variety of linear and non-linear problems. This in turn suggested, and was shown to be true, that it can solve problems with hybrid characteristics such as the doubly constrained traveling salesman problem.

ACKNOWLEDGEMENTS

I would like to thank Dr. Gordon Sparks for giving me the freedom to do my research in an area which I would find interesting, but to still keep me 'reined in' allowing me to get the job done in an acceptable fashion. I would also like to thank my wife, Delilah, for her never ending support and faith in my abilities. Without being maudlin, I would also like to thank my parents for all that they have done to support and encourage my academic achievements.

I would also like to acknowledge the contributions of various members of the Computer Science department who have been very patient with my neophyte inquiries and provided me nearly carte blanche access to their computer equipment.

Financial support for this work was obtained from the University of Saskatchewan.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Background - The Scheduling Problem

Scheduling is an everyday occurrence. Something as simple as two people getting together for a meeting, or even to just have coffee, requires a certain amount of scheduling. Scheduling two people to meet is a relatively simple task. But, small increases in the number of people involved will significantly increase the difficulty of the task. This is because scheduling is a combinatorial problem. The complexity of a problem will expand non-linearly with each additional event to be scheduled. Thus problems which were once simple to solve, become incomprehensible with the addition of a few new parameters.

Scheduling problems are not solely confined to the coordination of two people's daily timetables. The domain of Civil Engineering is full of processes which are scheduling problems or could be easily modeled as scheduling problems: project management, transportation of goods, and refuse collection. Any situation which requires a person, good, or task to be coordinated with another person, good or task at a prespecified time can be modeled as a scheduling problem. As stated previously the scheduling problem is a combinatorial problem. Consequently the complexity of many everyday problems exceeds the capabilities of most available analytical tools. One of two approaches must be taken. A common, low risk (but not necessarily the best) approach is to simplify the problem, and solve the new, simplified problem. The more efficient, but also more difficult approach is to develop new analytical tools to handle the problem 'as is.'

Scheduling is basically an allocation problem where scarce resources

must be allocated objectively (usually on a minimum cost basis). Problems of this type fall within the domain of Operations Research (OR). In the literature there are references to the scheduling of truck fleets (Powell et al. 1988), airline crews (Jones 1989), canal cleaning (Singer and Moritz 1987), and professional baseball leagues (Cain 1972). There are almost as many solutions as there are problems. This condition has arisen because scheduling is often used as a tool to gain a competitive advantage. Efficient scheduling allows a company to reduce their costs, or move more goods at the same cost. These savings are directly reflected in their profitability. To maintain their competitive edge companies will protect their scheduling techniques as proprietary information. Another reason for this proliferation of solutions is because many problems are conceptually quite similar, but are unique enough that a solution from one situation is not immediately transportable to another.

Solutions may be problem specific, but the general technique used to solve them is not. The first step is to mathematically model the environment in question so that a combination of independent elements or conditions can be compared quantitatively. Most solutions attempt to use a balance between experience based, rule-of-thumb solutions (heuristics) and mathematically tractable solutions (such as linear programming - a best-fit solution to a set of simultaneous equations). In practice, the ratio of heuristics and linear programming required to solve the problem varies with the complexity of the problem. Usually as the solution space grows, so does the reliance on heuristics to solve the problem. This reliance on heuristics generates problem specific or non-robust solutions.

1.2 The Doubly Constrained Traveling Salesman Problem

One of the most complicated types of scheduling problem is the doubly constrained traveling salesman problem (DCTSP). The DCTSP is an enhancement of the standard traveling salesman problem (TSP). Since the TSP is a very simple problem it will be described first. The elements which make the DCTSP more difficult to solve will then be introduced.

The objective of the standard TSP is to have the salesman travel the minimum distance and still visit all of his clients. The complexity of the TSP is (n-1)!/2. This means that if the salesman has four clients, there are (4-1)!/2=3 different routes which he can take. But, if there are eight clients (twice as many) there are (8-1)!/2=2520 different routes. As one can see the complexity of the TSP can expand rapidly.

The DCTSP is different from the basic or standard TSP as follows. In the classic TSP the salesman is visiting a variety of clients, attempting to minimize the distance traveled in making his tour. In this problem the salesman has the luxury of assuming that the clients will always be in their office and available to meet with him. To increase the level of difficulty of the problem, the clients can establish windows of opportunity for when they will see the salesman. Now, the salesman must consider client availability in addition to distance. To increase the difficulty of the problem to an additional level. Now imagine that each of the clients are themselves salesmen. The time that they are available to meet with a salesman is the time that they are in their own office. Scheduling the one salesman would be a difficult enough task, but the DCTSP attempts to schedule all of the salesman to minimize the distance traveled by all of the salesmen.

From an engineering prospective there are a variety of problems which can be classified as a TSP. Any problem where one must find the shortest route passing through every point in a set of points is a TSP. Example problems range from designing delivery truck routes, to routing cable through a building, to designing circuit board layouts. The TSP with time windows problem is a generic description of the school bus scheduling problem or, the transportation of perishable goods. In the basic TSP distance is the only factor. The TSP with time windows forces one to consider time and distance. The salesman must now arrive, or complete the entire journey, within a prespecified time window. The DCTSP further enhances the problem in that one is now trying to schedule a group of salesman. Each of these salesman are trying to meet with each of the other salesman. The DCTSP, which will be shown to describe a sports league schedule, can also be used to describe problems such as the design of courier hub and spoke networks or bus schedules. In these problems the vehicles must coordinate their arrivals at specified locations to transfer their loads (be it goods or people).

1.3 Applying Computers to the Scheduling Problem

Computers have traditionally been used to solve mathematically intensive scheduling problems such as the DCTSP. What computers are not well suited for is the heuristic knowledge that has been traditionally relied upon to solve many of these problems. Heuristics are a double-edged sword. Under conditions where people must make quick but not necessarily optimal choices heuristics are a cost-effective tool. But, when designing an automated Decision Support System (DSS), those same heuristics can reduce the cost

effectiveness of a system. The assumptions that heuristics are based upon can limit the long term usefulness of a system. Any change in the system may negate the heuristic. Also, to encode the heuristic, employees' must explain their methodology (this insight may not be readily brought forth under conditions where the DSS is perceived as a threat to job security) (Irgon 1990).

Schedules are dynamic, and unless an automated system can adapt to the various changes to the environment that the schedule must serve, then automation is not economically viable. Ideally software needs to be developed which can be applied to a basic problem type. A robust model would adapt to an evolving environment and could be transported between applications. Only basic expert knowledge (an understanding of the problem's constraints) of the problem would be required to modify the model. Ideally no significant reprogramming would be required, only a change in basic system parameters.

1.4 The Genetic Algorithm

A technique which can deal with the non-linear problems which occur in real-world applications is the Genetic Algorithm (GA). The GA was developed by John Holland at the University of Michigan in the late 60s and early 70s (Holland 1992). The algorithm is modeled after the process of natural selection found in successful biological systems (survival of those best adapted to the surrounding environment).

An explicit explanation of the GA and an example problem can be found in Appendix A. In general the GA evolves a variety of solutions until

an optimal solution is found. This is achieved by encoding the problem, or model of the problem, as a chromosome. This chromosome, which is often a binary string representation, is then mated with other chromosomes. Through reproduction and mutation new, and hopefully better, chromosomes are generated. Over time, the chromosomes converge towards the optimal solution.

Most algorithms which solve non-linear problems are limited in their scope, but the adaptive nature of the GA allows it to efficiently solve many types of non-linear (and linear) problems. The robust nature of the GA makes it an excellent candidate for solving the DCTSP.

The GA's greatest attribute is its flexibility. It is applicable to a variety of problems and is easily modified. These factors make it a good alternative to solve the DCTSP.

This is not to say that the GA is the perfect algorithm. It does have its share of limitations, but in general its flexibility overcomes most potential problems. Like other analytical methods the GA requires a mathematical model of the problem. But, unlike other algorithms and Expert Systems, maintenance is not a problem because it can be implemented in a variety of software languages, and since the basic algorithm is easily understood, in-house Information Systems staff should be easily able to maintain the code. Verification that the solution is optimal is not possible for most real-world solutions because the GA is a probabilistic system. There are no guarantees that a global optimum is reached. But, since the algorithm is mathematically founded (Holland 1992a), there is assurance the solution is at least a local

optimum, and that it is the best solution found to date.

The GA's ability to search several regions of the solution space simultaneously makes it unique among other non-linear algorithms. The existence of a combinatorial explosion can be exploited, instead of acting as a barrier which is usually the case (Michalewicz 1992). This characteristic makes GAs a candidate solution method for many classical problems found in Operations Research which have traditionally been solved in the field using heuristic methods: the doubly constrained traveling salesman problem, the traveling salesman problem, facility location or (minimax and maximin problems where one attempts to minimize the maximum distance any one person must travel, or maximize the minimum distance a person must travel), vehicle routing, and scheduling. These problems all share the common trait that as they increase in size, they become increasingly difficult (if not impossible) to solve analytically.

The scheduling of a professional sports league can easily be modeled as a DCTSP. In any league there is a collection of teams (salesmen) which must visit a specific set of teams (clients) a predefined number of times. To further complicate the scheduling process teams are only available for short windows of opportunity. The prime objective of this problem is to minimize the league's traveling costs. Creating a mathematical model to solve this problem is a difficult task. But, since most professional sports leagues have a continuous ebb and flow of new teams and new locations, these changes further complicate the modeling process. The robust nature of the GA makes it well suited to solve this very complex and continuously evolving problem.

The ease of modeling the league schedule as a DCTSP was not the only reason the NHL schedule was selected as the testing ground for the GA. Designing a test problem which is truly representative of a real-world problem is a difficult task. Since the base data used to generate the NHL schedule was readily available it was felt that a real problem would be a better test of the GA's ability than a model based on conjecture and the researcher's implicit biases. Also, because a known solution exists, the GA's performance could be measured against a reliable benchmark.

1.5 Objectives

The objective of this thesis is to explore the GA's ability to solve large scale scheduling problems. The model selected is the National Hockey League's (NHL) 1992-1993 regular season playing schedule.

1.6 Scope

The scope of this thesis will be restricted to the design and development of a Genetic Algorithm which can be applied by the NHL to create their playing schedule. The software will be designed using a relatively available software language (ANSI-C) so that it will work on a personal computer. Demonstrating that the GA can solve the NHL scheduling problem will imply that it will be useful relative to a number of similarly complex scheduling problems for which analytical solution methods have not as yet been developed.

1.7 Methodology

The methodology that will be followed to achieve the objectives of this

thesis is as follows:

1) Demonstrate the robustness of the GA by solving a variety of increasingly difficult OR problems.

2) Introduce various GA modeling techniques to evaluate their potential application in solving the NHL schedule.

3) Apply the Genetic Algorithm to a simplified model of the NHL schedule (six teams and a shortened schedule) using distance as the only factor in the objective function.

4) Expand the model to full scale (twenty-four teams and an eighty-two game schedule) using distance as the only factor in the objective function.

5) Expand the objective function to include additional league constraints such as uniform distribution of games, restricting the length of time a team is on the road, and making the last weeks of play consist primarily of inter-division play.

6) Evaluate the schedule the GA generates in comparison to the schedule completed manually by the NHL.

1.8 Layout Of The Thesis

To meet the objectives this thesis has been divided into five sections:

1. Introduction
2. Investigating the GA's Robustness
3. Applying the GA to a Real World Problem
4. The GA and Large Scale Problems
5. Discussion of Results, Conclusions and Recommendations.

The first chapter serves as an overview of the importance and complexity of the scheduling problem. The general concepts of the GA are also introduced, as is the reasoning for attacking the scheduling problem with the GA.

The second chapter contains applications of the GA to various classes of problems in Operations Research: linear programming, Integer-linear programming, job scheduling problem, the traveling salesman problem, and finally, the transportation problem.

The third chapter examines a real world (and also a large scale) scheduling application; the NHL's 1992-1993 playing schedule. The first topic covered in this chapter is a discussion of the complexity issues involved with creating the NHL schedule. This is followed by a complete description of the modeling process. The third part of this chapter is a discussion of the GA's results.

The fourth chapter expands upon the discussion of the GA's ability to solve other large scale, real-world problems. This chapter includes a review of the feasibility and limitations of developing a large scale GA.

The fifth chapter summarizes the work done in this thesis. Conclusions are made, and some possible extensions for further research are put forth.

2. INVESTIGATING THE GENETIC ALGORITHM'S ROBUSTNESS

The primary goal of this chapter is to illustrate the GA's ability to solve a variety of problem types. Why is this important? There exist a variety of algorithms and heuristics which solve specific problem types extremely well. But, once they are applied to problems outside of their scope they are basically useless. Under real-world conditions few problems can be packaged neatly into the domains of these problem solving techniques. Showing that the GA can solve a variety of specific problems effectively suggests that it is a technique which can deal with a broad range of real world problems.

This is not to say that the GA is a panacea to all Management Science/Operations Research (MS/OR) problems. It will be very evident that the GA is not well suited to solve certain problems. It will get the job done, but not with the elegance of a specialized algorithm. The results from these examples will define the boundaries of usefulness of the GA. Thus, when faced with a problem similar to one in this chapter, one will know beforehand the preferred approach to successfully applying the GA to that problem type.

For the reader not already familiar with the GA, Appendix A gives a detailed description of the method by way of an example problem.

A secondary goal of this chapter is to introduce a variety of strategies or techniques which can be used to improve the GA's performance. The most important strategy when designing the GA is to follow a standardized methodology. Following a standardized methodology allows the designer to 'see through the noise' in a problem and focus on how best to apply the GA.

This methodology will be introduced and ***strictly*** followed in this chapter. Some of the techniques which will be introduced to improve the GA's performance include the use of multi-character alphabets, matrix chromosome structure, and self correcting reproduction and mutation operators.

Finally, the last goal of this chapter is to act as a reference. Most GA texts refer in passing to the application of GAs to classic MS/OR problems. Michalewicz's text (1992a) is one of the few which actually goes into any detail of the GA's application, but his list is incomplete. The eight problems (from five different problem types) covered in this chapter is still not exhaustive, but extends the basis that Michalewicz established.

2.1 Linear Programming

There are two example problems in this category: the Giapetto problem, and the Dakota problem. Both problems are from Wilson (Wilson 1989). Linear Programming problems have been studied extensively, and several exact solution techniques are known. Problems that could be solved using LP include determining the optimal blend for cattle feed or concrete mix, finding a lowest cost distribution strategy for a trucking firm, or designing an efficient production schedule. The form of an LP problem is:

$$\max \text{ (or min)} \quad z = c^t x \tag{2.1}$$

$$\text{subject to} \quad Ax = b \tag{2.2}$$

Note the objective function (equation 2.1) is linear, hence the name Linear Programming. A series of constraints (equation 2.2) define the combination

of parameters which constitute a valid solution. From a mathematical perspective, the constraints define (or restrict) the solution space.

Traditional techniques to solve LP problems are well known, well understood, and very efficient. Any GA research which has been completed in this area has focused on techniques to reduce the GA search space (Michalewicz 1992a). The techniques used in the LP examples to follow do not follow this research direction. The methods which will be introduced are ones which can be built upon to solve the more difficult non-linear problems to follow.

2.1.1 The Giapetto Problem

The Giapetto problem is a simple LP problem. It is included as an introduction to the LP problem and to introduce the first variation on the classic binary representation scheme of the GA.

This problem is a production problem. The company, Giapetto's Woodcarving Inc., manufactures toy soldiers and trains. The production of each toy requires different amounts of finishing labour and carpentry labour. The profit margin on each toy is also different. The demand of toy soldiers is known to be limited to forty in one week. Giapetto wishes to maximize his profits. Below, is a matrix of the pertinent values.

Table 2.1 Production Parameters for the Giapetto Problem

	Soldiers	Trains	Available Resources
Profits/unit	3	2	
Finishing Labour Costs	2	1	100
Carpentry Labour Costs	1	1	80

The values in the matrix can be used to develop the following model.

$$\text{Maximize} \quad z = x_1 + x_2$$

$$\begin{aligned} \text{Subject to} \quad & 2x_1 + x_2 \leq 100 \quad \text{(Finishing Constraint)} && (2.3) \\ & x_1 + x_2 \leq 80 \quad \text{(Carpentry Constraint)} && (2.4) \\ & x_1 \leq 40 \quad \text{(Soldier Demand Constraint)} && (2.5) \end{aligned}$$

Where x_1 is the number of soldiers produced, and x_2 is the number of trains produced.

The optimal solution is x_1=20, x_2=60, for a profit of 180 (Giapetto should produce 20 soldiers and 60 trains). Figure 2.1 contains the graphical interpretation of the solution space (the shaded area) defined by the constraints. The point representing the optimal value is circled.

Now that the problem is defined, the next step is to design the GA. The first stage in designing a GA is to determine a representation scheme. In the example in Appendix A the chromosome represented a single variable. This problem requires the chromosome to represent two variables. This is achieved by subdividing the chromosome into two parts, each corresponding to one variable.

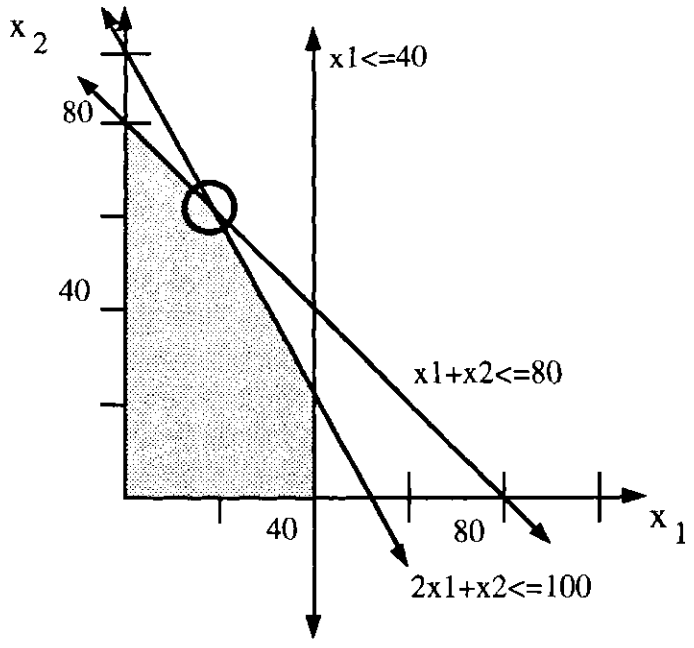


Figure 2.1 Description of the Solution Space for the Giapetto Problem

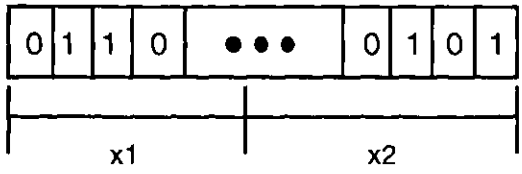


Figure 2.2 A Schematic of the Allocation of the Chromosome Structure Between Two Variables

Since the chromosome is divided into two regions, the total length is a function of the length of the binary number required to represent each variable.

In reality this problem should be solved as an integer problem (Giapetto could make 0.75 of a toy soldier, but he would be hard pressed to sell it for 0.75 of its normal price). To increase this problem's difficulty it shall be approached with two decimal place precision. By setting x_1 to 0, the largest

feasible value for x_2 is 80. Similarly, the largest feasible value of x_2 is 40. To achieve the desired precision the feasible values become 8000 (80 x 100) and 4000. The calculation used to find the length of the binary number necessary to represent these values are shown below.

$$\frac{\ln 4000}{\ln 2} = 11.97 \rightarrow 12 \quad \text{eg. } 2^{12} - 1 = 4095 \tag{2.7}$$

$$\frac{\ln 8000}{\ln 2} = 12.97 \rightarrow 13 \quad \text{eg. } 2^{13} - 1 = 8191 \tag{2.8}$$

To represent x_1 and x_2 in one chromosome, the total chromosome length must be 25. The search space contains 2^{25}, or 33,554,432 points. There are no restrictions on exploring the search space. Since not all of these points are valid solutions a penalty function was required to reduce the viability of a non-valid solution propagating from generation to generation.

Once the representation scheme is designed a fitness measure must be selected. The obvious metric is the objective function. The traditional method of measuring a chromosome's fitness is over the range of 0 and 1. Because the Giapetto problem has a known solution, we can normalize the objective function by dividing it by the known optimum[1] . A chromosome with optimal fitness will have a fitness of 1.

Penalty functions[2] can be subdivided into three basic groupings:

- total loss, or zero contribution,

---

[1] The value of the objective function is also called the raw fitness, and what we are calling the fitness is also known as the normalized fitness.

[2] Although penalty functions are referred to in published GA research, the 'black-box' or proprietary nature of most work precludes detailed explanation of their implementations. For the lack of any existing naming convention, the author has implemented his own.

- semi-uniform contribution,
- and, semi-proportional contribution.

Total loss functions result in the loss of all genetic information (Figure 2.3). Chromosomes violating a constraint are given a fitness of zero. This is the simplest implementation of a penalty function. Since these chromosomes have no fitness, the likelihood of being selected to reproduce is negligible.

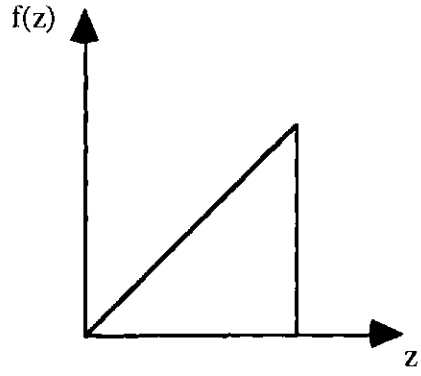


Figure 2.3 A Graphical Depiction of the Total Loss Penalty Function

Semi-uniform contribution penalty functions (Figure 2.4) maintain a chromosome's genetic information. Illegal chromosomes (those violating a constraint) are set equal to some fractional value. Setting the fitness to zero would result in the 'genetic information' contained in the chromosome to be lost to subsequent generations. This is fine if the chromosome exceeded all of the constraints by a large amount, but not if the constraints are only marginally exceeded. Keeping these chromosomes is especially important when the population converges near the optimum. If the search space is being heavily sampled near the optimum, and a significant number of chromosomes created violate the constraints, then a significant number of

chromosomes will not contribute to the next generation. Those chromosomes that survive will quickly dominate the population, resulting in a significant loss of genetic diversity. Under this scenario, it may take a long time for the GA to reach the optimum value because the population begins to slowly creep towards the optimum. Significant leaps in fitness are penalized if they overshoot their target. This function has been labeled 'semi-uniform' because the penalty is not applied to all of the chromosomes violating the constraints. In some cases, some of the chromosomes may have a normalized fitness that is less than the arbitrary penalty. In these cases, the chromosome is assigned the smaller of the two values. This prevents very weak chromosomes from increasing their fitness due to a violation. Implementing this semi-uniform penalty function is slightly more difficult than implementing the total loss function. The difficulty is in assessing the size of the penalty. If it is too small, undesirable chromosomes are selected to reproduce. If the penalty is too large the chromosome is never selected.

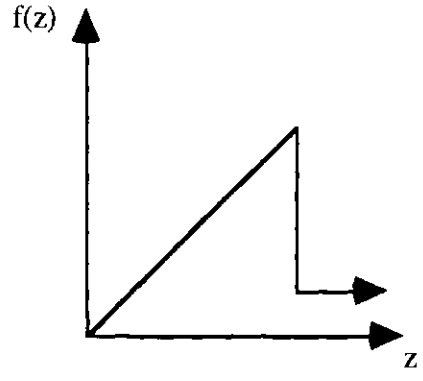


Figure 2.4 A Graphical Depiction of the Semi-uniform Penalty Function

A semi-proportional penalty function (illustrated in Figure 2.5) can be implemented one of three ways:

- as a function of the number of penalties violated

- as a function of which constraints were violated
- as a combination of the number of penalties and which constraints were violated

This penalty function is the most complicated to model, but it is often the most efficient. The first implementation is based on the assumption that the number of constraints violated are inversely proportional to the chromosome's fitness. The more constraints that are violated, the smaller the fitness value. Ideally, the fitness should be scaled to the optimum, but in most real problems the optimum is not known, preventing any kind of accurate (or proportional) scaling from occurring. A more involved implementation is applicable when the constraints can be prioritized. For example, if constraint A is twice as important as constraint B, then any violation of constraint A should be penalized twice as much as a violation of constraint B. Again, the prefix 'semi' is used to denote that the penalty is assigned only if the normalized fitness is greater than the penalty fitness.

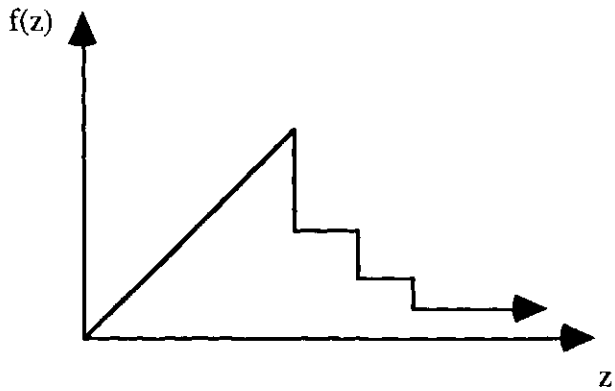


Figure 2.5 A Graphical Depiction of the Semi-Proportional Penalty Function

For the Giapetto Problem a semi-uniform penalty function was implemented. The presence of the constraints (Ax=b) prevent the chromosomes from having a fitness larger than 1. Any chromosome which

exceeds a constraint is given a fitness of 0.001.

With the fitness measure determined, the critical parameters must be set. The critical parameters in the GA are population size and the number of generations that the GA evolves. In general the population size should be as large as possible. It was found that this problem was simple enough that populations of 100 or more rapidly solved the problem. Reducing the population size to twenty provided enough of a challenge to make the problem interesting.

Initially, there is no way of determining the number of generations required to solve the problem. There are three methods used to determine the termination criteria

- set the number of generations equal to some arbitrary high number (1000, say)
- to terminate when the solution comes within a specified percentage of the solution
- if some desired performance level is reached or if a specified time limit was reached.

For the Giapetto problem, the GA was set to run until it had evolved 120 generations.

Standard genetic operators (crossover and mutation) were implemented because a binary representation was chosen (as described in Appendix A). The crossover rate is usually arbitrarily set above 0.5. After performing some initial tests the crossover rate was set to 0.7. The mutation rate is usually selected to produce at least one mutation in each chromosome (for this chromosome with a length of 25, $1/25 = 0.04$) (Koza 1993, Goldberg 1989). That rate was found to be too disruptive as the GA approached the

optimum, so it was reduced to 0.033.

High mutation rates are often beneficial in the early stages of the GA. This is because it allows the GA to sample a wide variety of regions simultaneously, but at the same time converge towards the optimum solution through reproduction. But, as the chromosomes become more uniform in structure (and thus are all generally near the optimum solution) the mutation operator which aided in bringing these chromosomes towards a unifying goal becomes a disruptive force. In other words, if a chromosome is weak relative to the optimal solution, mutation is usually quite beneficial. But, if the chromosome is near optimal then it can weaken the chromosome.

Table 2.2 Genetic Algorithm Parameters for the Giapetto Problem

Crossover Rate	0.7
Mutation Rate	0.033
Population size	20
Chromosome Length	25
Maximum Number of Generations	120
Normalizing Value	180

The initial population was generated probabilistically (similar to flipping a coin 20 x 25 = 500 times).

All problems in this chapter, unless otherwise noted, were run on an Intel based 33 MHz-386 processor. The software was written following ANSI-C based conventions and was compiled using DOS and Windows versions of the Borland C compiler (Version 3.1). Although the DOS version of the software ran up to twice as fast as the Windows version, access to the DOS

compiler was severely limited so speed was compromised for the sake of convenience. It should also be noted that the random number generator used for all problems in this research was the standard random number generator supplied with the compiler on that platform. Random number generator seeds ranged from 1 - 500. Statistical analysis was not performed on the GA results, but there did not appear to be any correlation to how fast (the number of generations the GA took to solve the problem) the GA solved the problem relative to the seed's value.

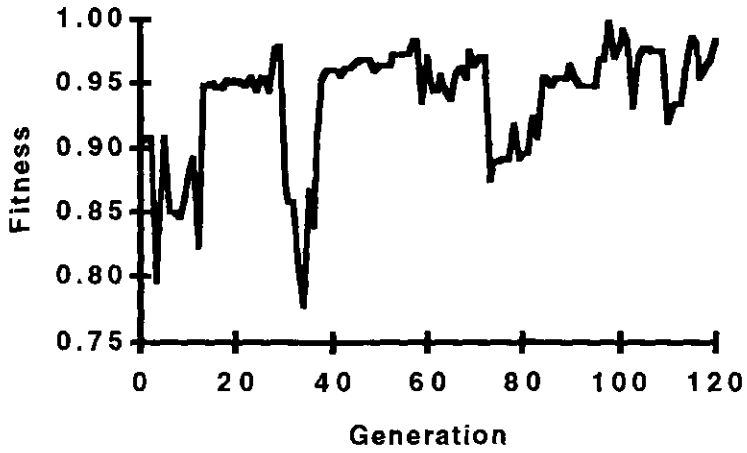


Figure 2.6 A Graph of the Best Chromosome Fitness in Each Generation of a GA Run

The optimal solution of 1.0 was not found, but the best individual in generation 98 had a fitness of 0.9992 (x_1=19.88,x_2=60.11). It is not surprising that the exact solution was not found considering that of the approximately $34(10^6)$ points in the search space, only one corresponds to the exact answer. But, for all intents and purposes, the solution was found.

2.1.2 The Dakota Problem.

The Dakota Problem is also an LP problem. It is introduced for the following reasons:

- the problem contains three variables (which increases the complexity of the problem)
- the form of the objective function challenges the GA more than the Giapetto problem
- since the GA again failed to find the *exact* solution, this appears to be a good time to introduce a reliability calculation developed for the GA

Before the problem can be dissected, it must be introduced. The Dakota Furniture Company manufactures three product lines: desks, tables, and chairs. The resources required for these products are lumber and labour (finishing and carpentry). The resources required for each product are shown in Table 2.3, and the available resources and the revenue from each product are in Tables 2.4 and 2.5 respectively.

Table 2.3 Resources Required to Produce Product lines

Resource	Desk	Table	Chair
Lumber (in board-feet)	8	6	1
Finishing hours	4	2	1.5
Carpentry hours	2	1.5	0.5

Table 2.4 Revenues From Each Product Line

	Desk	Table	Chair
Revenue/Unit	$60	$30	$20

Table 2.5 Resources Available to the Dakota Furniture Company

Resources	Available
Lumber (in board-feet)	48
Finishing hours	20
Carpentry hours	8

The owner's objective is to maximize the company's revenues. The LP model is formulated as below:

$$\text{Maximize} \quad z = 60x_1 + 30x_2 + 20x_3$$

$$\begin{aligned}
\text{Subject to} \quad & 8x_1 + 6x_2 + x_3 \leq 48 \quad \text{(Lumber Constraint)} \quad (2.3)\\
& 4x_1 + 2x_2 + 1.5x_3 \leq 20 \quad \text{(Finishing Constraint)} \quad (2.4)\\
& 2x_1 + 1.5x_2 + 0.5x_3 \leq 8 \quad \text{(Carpentry Constraint)} \quad (2.5)
\end{aligned}$$

Where x_1 is the number of desks produced, x_2 is the number of tables produced, and x_3 the number of chairs produced.

The optimal solution is $x_1=2$, $x_2=0$, $x_3=8$, for a total revenue of 280.

The problem is defined, so now the GA can be designed. A binary representation scheme will meet the needs of this problem. The chromosome is divided into three groupings, each corresponding to one of the three variables. As in the Giapetto problem, to calculate the length of the binary string representing each variable, the upper bound of each variable is required. Based on the constraints the upper bounds for x_1, x_2, and x_3 are 4, 5.33, and 13.33. To be able to calculate to two decimal place precision the chromosome length required is 9+10+11=30. Since these are upperbound values, and knowing the optimal values using traditional methods, the lengths were reduced to 9+9+10=28 genes long. The resulting search space

was 2^{28}=268,435,456 points, (as compared to the original length of 2^{30} which is approximately 1 billion points). Since all of the points are accessible by the GA, and not all of them are valid solutions, a penalty function is required.

As in the previous example, the semi-uniform penalty function was implemented for this problem. The objective function divided by the known optimal value was used as the measure of merit. Any chromosome violating a constraint was given a fitness of 0.001.

A population size of 100 was selected. As in the Giapetto Problem, the population was sized for illustrative purposes and not to generate a rapid solution. The number of generations that the GA should evolve was set at 300. It was found that after 100 generations the GA did not appear to show any improvement (consequently Figure 2.7 only depicts generations one to 100). The crossover rate was 0.7, and the mutation rate was 0.033. The initial population was selected probabilistically. Table 2.6 contains a summary of the parameters.

Table 2.6 Genetic Algorithm Parameters for the Dakota Problem

Crossover Rate	0.7
Mutation Rate	0.033
Population size	100
Chromosome Length	28
Maximum Number of Generations	300
Normalizing Value	280

The DOS version took approximately 150 seconds to solve this problem, the Windows version took 280 seconds.

The chromosome which evolved the highest fitness had a fitness of 0.9961 (x_1=1.89, x_2=0.09, x_3=8.14, z=278.9) and appeared in generation 58.

The explanation for the lack of convergence is based on two properties of the problem. One is the design of the chromosome structure, particularly the use of precision to the hundredths. This problem is only practical with an integer solution. Solving the problem for an integer solution would have required a length of eleven. This is a search space of only 2048 points, insignificantly easy for the GA. The second, and more intrinsic problem is that the shape of the hyperplane representing the search space is relatively flat (there is a minimal amount of differentiation amongst the points that are near optimal and the optimal value). This lack of clear differentiation results in the GA taking a random walk in the neighbourhood of the optimal value. The best way to deal with a search space of this nature is to change the fitness function to a polynomial. This amplifies the fitness of the optimal value relative to other solutions.

Since the GA is a probabilistic method there is no guarantee of finding the best solution. The GA is best suited to find an approximate solution quickly. The time required to find a solution is not fixed, but through minimal experience, one can estimate how long it will take a GA to converge given the population size and chromosome length. In situations where a GA must be relied upon to repeatedly solve the same problem (but with ever-changing parameters), an estimation of the optimal number of generations required to converge can be calculated (Koza 1992).

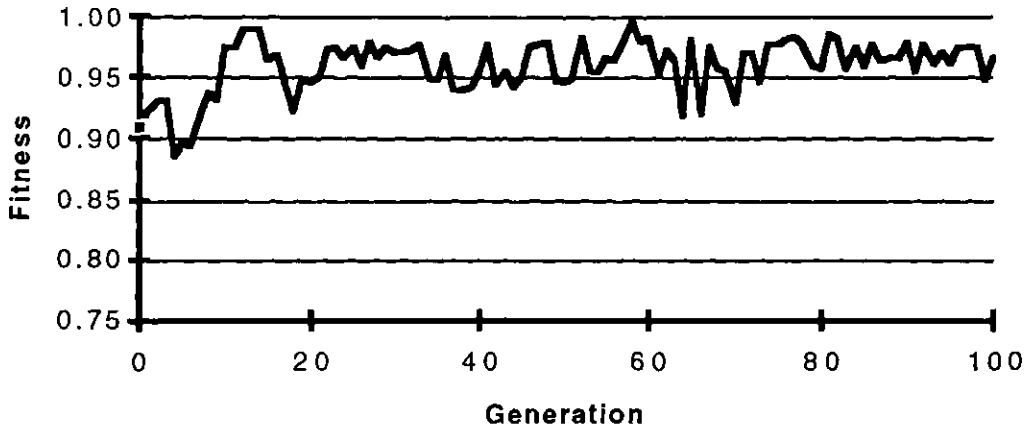


Figure 2.7 A Graph of the Best Chromosome Fitness in Each Generation of a GA Run

To make this calculation a large number of GA trials must be completed. For the Dakota problem, the GA was instructed to run until the solution came within one percent of optimum, or one thousand generations had evolved. Based on three hundred runs, a frequency distribution of the number of generations required to solve the problem was created. This in turn was converted to a cumulative probability distribution. The calculation below derives a predictive model for the number of required trials for a GA to find a solution within a given probability.

z	=	the certainty level that a solution is found (if z=0.99 then there is 1 chance in 100 no solution is found)
P(M,i)	=	probability of finding a solution by i generations with a population size of M
1-P(M,i)	=	probability of not finding a solution
$[1-P(M,i)]^n$	=	probability of not finding a solution after running the GA for i generations, n times
$1-[1-P(M,i)]^n$	=	probability of finding at least one solution after running the GA for i generations, n times

Now, setting the probability of finding at least one solution equal to the desired certainty level and solving for n (n being the number of times the GA would have to be run to find a solution given a population size of M, and only evolving i generations).

$$z = 1 - [1 - P(M,i)]^n \tag{2.13}$$

$$1 - z = [1 - P(M,i)]^n \tag{2.14}$$

$$n = \left\lceil \frac{\ln(1-z)}{\ln[1-P(M,i)]} \right\rceil \tag{2.15}$$

From the number of runs required, one can find the number of individuals processed (the population size multiplied by the number of generations multiplied by the number of runs) to find the solution within the required level of certainty. Graphing this value against the number of generations clearly points out the most efficient number of generations that the GA should evolve. Table 2.7 and Figure 2.8 illustrate this point. Note that since P(M, i) is defined as the probability of finding a solution by i generations with a population size M, this is the same as the cumulative probability.

Figure 2.8 superimposes two graphs. The white squares show the cumulative probability of successfully solving the problem given the GA evolves a specified number of generations each run. For example, if the GA runs for 200 generations, then there is a 77% chance it will find the solution in one run). This is, of course, based on the 300 sample runs, where 77% of all runs solved the Dakota Problem within 200 generations. Based on equation 2.15 the necessary number of iterations for 99% certainty is 3.13 or 4 repetitions. Thus, after running this problem four times, for 200 generations

you would have processed 201 generations x 100 chromosomes/generations x 4 runs or 80,400 chromosomes. Figure 2.8 illustrates that for the Dakota Problem it was most efficient to run it six times for only 80 generations (48,600 chromosomes processed).

Table 2.7 Summary of Results for 300 Runs of the Dakota Problem

Generation	Count	Probability	Cumulative Probability	Required # of Runs	Individuals Processed
20	50	0.167	17%	25.26	54,600
40	50	0.167	33%	11.36	49,200
60	34	0.113	45%	7.78	48,800
80	28	0.093	54%	5.93	48,600
100	11	0.037	58%	5.36	60,600
120	16	0.053	63%	4.63	60,500
140	12	0.040	67%	4.15	70,500
160	11	0.037	71%	3.75	64,400
180	12	0.040	75%	3.35	72,400
200	7	0.023	77%	3.13	80,400
250	18	0.060	83%	2.60	75,300
300	10	0.033	86%	2.31	90,300
350	10	0.033	90%	2.03	105,300
400	12	0.040	94%	1.67	80,200
450	9	0.030	97%	1.35	90,200
500	1	0.003	97%	1.31	100,200
600	1	0.003	97%	1.27	120,200
700	2	0.007	98%	1.18	140,200
800	2	0.007	99%	1.07	160,200
900	4	0.013	100%	1.00	90,100

The key point being illustrated by Figure 2.8 is that it is not always better to allow the GA to run endlessly until some solution is found There is a cut-off point for all problems. If the GA does not find a solution by that generation, then it should be restarted with a new initial population. Equation 2.15 is the tool which can be used to estimate the computational effort required to solve a problem. From a practical perspective this calculation is only effective if one has the opportunity to solve a sample

problem (under repeated trials) which is quite similar to the actual problem to be solved.

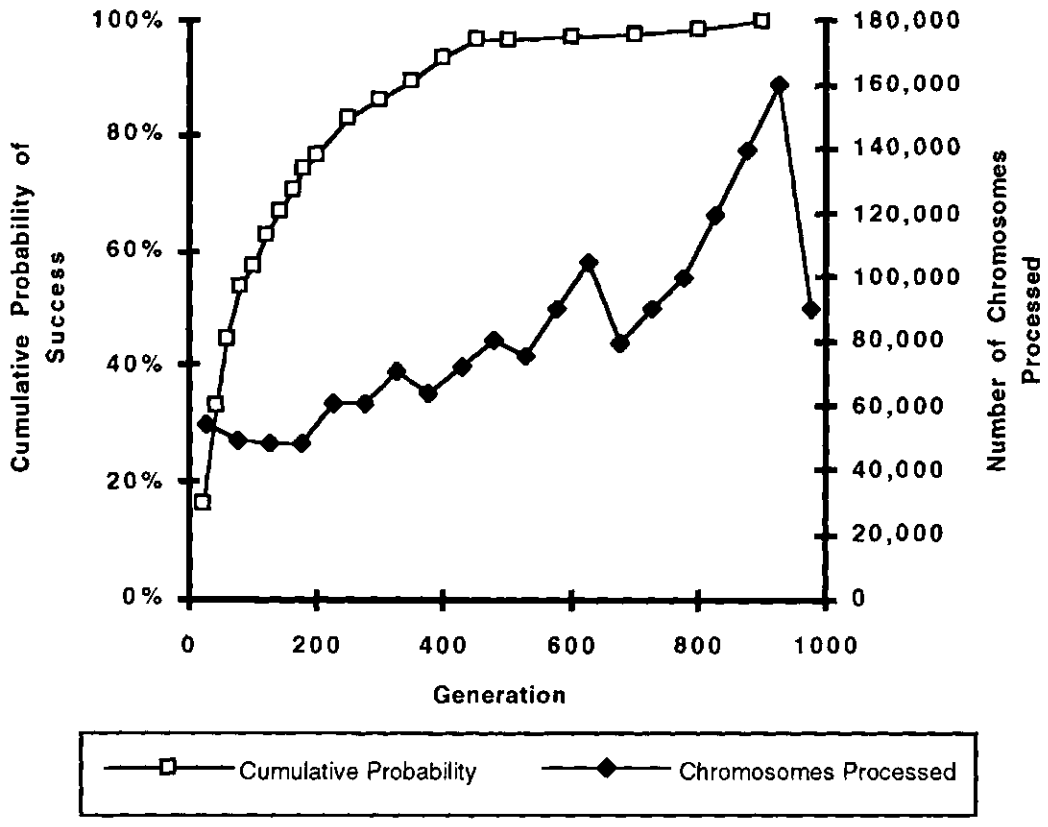


Figure 2.8 Graph of Cumulative Probabilities and Number of Individuals Processed vs. The Number of Generations a GA Evolves (Dakota Problem)

2.2 Integer Programming

Integer Programming (IP) problems are best described as LP problems where the solution must be integer. Most IP problems are solved as a variation of LP problems using a branch and bound technique. Integer Programming is used in situations where an exact integer solution is required. All of the examples discussed for LP also fall within this domain. .

Other IP problems include selecting a crew roster for an airline schedule, vehicle routing, and facility location. A special subset of IP that can be easily modeled with the binary chromosome of the GA is 0-1 programming.

The GA has been applied to the IP problems generally classified as set covering (Beasley and Chu, 1994) and set partitioning. (Chu and Beasley, 1995). The methods used in these two papers are significantly more complicated than will be presented in this section, because the researchers were attempting to solve a complex problem and not illustrate basic concepts as is the case here.

The IP section of this chapter will introduce several more modeling techniques for the GA. The first topic to be introduced will be how to deal with a minimization problem. The previous two examples dealt with maximizing profits and revenues. The problem in this section (from Wilson 1989) deals with minimizing cost. Also, we will examine how well the GA handles the discontinuous search space of an integer problem. With the previous problems, the search space was continuous because it described a 'floating-point' world. This is not the case in IP. And finally, it will be shown that the GA's performance will be relatively poor using the methods introduced so far. A new methodology, chromosome repair, will be introduced and its effectiveness will be measured.

2.2.1 The Nickles Problem

This IP problem can be further classified as a fixed-charge problem. Fixed-charge problems are characterized by the fact that there is a cost associated with the production of some good, regardless of the quantity of that

good being produced. In this problem, the J. C. Nickles company receives credit card payments from four regions of the U.S. (West, Midwest, East, and South). The company can place processing centres in four different cities (Los Angeles, Chicago, New York, and Atlanta). The company loses daily interest for each day cheques are in the mail, so it wants to reduce the amount of time the cheques spend in the mail. But, there is a fixed cost of operating each processing centre. Thus, J. C. Nickles must determine how many processing centres it must open, and where it should locate them.

The objective function is shown below. Each x_{ij} will equal either a 0 or a 1 (hence the name 0-1 programming). If region i sends its payments to city j then x_{ij} will equal 1, (0 otherwise). Each y_j will equal 1 if a processing centre is opened in city j, (0 otherwise).

The coefficients represent the associated costs. The payment coefficient (e.g. 28 x_{11}) is the cost associated with the delay when region i sends its payment to city j. Similarly, the coefficient to the y-variable is the cost associated with opening a processing centre in city j.

$$\begin{aligned} \text{Minimize} \quad z = {} & 28x_{11} + 84x_{12} + 112x_{13} + 112x_{14} \\ & + 60x_{21} + 20x_{22} + 50x_{23} + 50x_{24} \\ & + 96x_{31} + 60x_{32} + 24x_{33} + 60x_{34} \\ & + 64x_{41} + 40x_{42} + 40x_{43} + 16x_{44} \\ & + 50y_1 + 50y_2 + 50y_3 + 50y_4 \end{aligned} \tag{2.16}$$

There are two types of constraints that must be modeled. The first constraint type (type 1) makes sure that each region sends its payments to a single city. The second constraint type (type 2) makes sure that if a region sends its payments to a city, then that city must have a processing centre.

Below are the type 1 constraints.

$$x_{11} + x_{12} + x_{13} + x_{14} = 1 \quad \text{(West region constraint)} \quad (2.17)$$
$$x_{21} + x_{22} + x_{23} + x_{24} = 1 \quad \text{(Midwest region constraint)} \quad (2.18)$$
$$x_{31} + x_{32} + x_{33} + x_{34} = 1 \quad \text{(East region constraint)} \quad (2.19)$$
$$x_{41} + x_{42} + x_{43} + x_{44} = 1 \quad \text{(South region constraint)} \quad (2.20)$$

The type 2 constraints are shown below.

$$x_{11} + x_{12} + x_{13} + x_{14} \leq 4y_1 \quad \text{(Los Angeles constraint)} \quad (2.21)$$
$$x_{21} + x_{22} + x_{23} + x_{24} \leq 4y_2 \quad \text{(Chicago constraint)} \quad (2.22)$$
$$x_{31} + x_{32} + x_{33} + x_{34} \leq 4y_3 \quad \text{(New York constraint)} \quad (2.23)$$
$$x_{41} + x_{42} + x_{43} + x_{44} \leq 4y_4 \quad \text{(Atlanta constraint)} \quad (2.24)$$

The optimal solution to this problem is z=242, where y_1=1, y_3 =1, x_{11}=1, x_{23}=1, x_{33}=1, x_{43}=1 (processing centres would be in Los Angeles and New York, with the residents in the West sending their payments to Los Angeles, and the rest of the country to New York).

As previously stated, the first step in designing the GA is to establish the chromosome structure. Since each of the variable types (x_{ij} and y_j) must equal 0 or 1, it is only natural to represent each variable as a binary gene in the chromosome. The resulting structure is a binary chromosome with a length of twenty. The resulting search space is 2^{20} or approximately one million points. Since not all of the points are valid solutions, a penalty function (which will be discussed later) was necessary.

The fitness measure for this problem differs from the previous problems because they were maximization problems, and this is a minimization problem. The GA will normally only work on a maximization

problem. To convert a minimization problem to maximization, subtract the objective function from some constant. As the objective function gets smaller, the difference between it and the constant gets larger. The trick is to find a constant which is large enough that valid solutions do not create a negative fitness. This constant will also be used as the normalizing value. Unless the optimum (minimum) value is 0, the most fit individual will not have a normalized fitness of 1.

The upperbound, or highest cost scenario would have each region send their payments to the city which takes the longest to receive their mail (highest cost in terms of delay) and there would be a processing centre open in each city. Under this scenario we have the following:

Table 2.8 Highest Costs Associated with Sending Mail Between Each Region and City

Region	City	Cost
1	3 or 4	112
2	1	60
3	1	96
4	1	64
		Total Cost = 332

Adding this cost to the cost of having four processing centres (200) gives a total cost of 532. This is not a feasible arrangement since the payments are only being sent to two cities, and a cost has been allocated as if four processing centres exist. But, it is best to select a large value when selecting an upperbound for a minimization problem because the standard GA algorithm will not work with a mix of positive and negative fitnesses (see Appendix B).

For this problem the GA was designed to terminate on the condition that it had evolved 200 generations or found a solution less than or equal to 250 (which is within three percent of the optimal solution).

The standard genetic operators were used for this problem. Illegal chromosome structures could evolve, so a penalty function was implemented. Each chromosome that represented an illegal condition (a region sent their cheques to more than one city, for example) was given a penalty fitness of 0.01. The fitness of the optimum chromosome (242) was 0.5451, and the fitness of a chromosome which would just terminate a run (250) would be 0.5301. Thus, the penalty fitness was approximately two percent of an optimal chromosome. At first glance this may appear high, but this is because the valid search space was quite sparse and illegal chromosomes were easily created. Also, if the penalized chromosome fitness was too small relative to the legal chromosomes, then the information in these chromosomes would be lost.

The probability of mutation was selected to produce one mutation in each chromosome (1/20=0.05). The probability of crossover (0.8) was selected after several initial trials. The trials indicated that a higher than normal crossover rate performed better. This was due to the high likelihood of generating illegal chromosomes. Initially it is difficult for the GA to evolve legal chromosomes. The high crossover rate 'stirs up' the chromosome population until a few legal chromosomes are created. The fitness of the initial legal chromosomes was approximately ten to twenty times that of the illegal ones. The legal chromosomes quickly dominated the next generation, and the GA had a starting point from which to evolve. The population size

selected (400 chromosomes) was quite large relative to the previous problems (the Dakota problem had a population size to search space ratio of 100:27(10^6), for this problem the ratio is 400:1(10^6)). This was necessary because the smaller population sizes were not conducive to the GA quickly evolving legal chromosomes. Consequently, the GA spent crucial early time steps mating weak chromosomes in an attempt to create legal ones. A large population size reduced the 'floundering' to less than five generations. The maximum number of generations each GA was to evolve for this problem was selected after observing that if the GA had not already found a solution by 150 generations, it was unlikely to find a solution.

Table 2.9 Genetic Algorithm Parameters for the J. C. Nickles Problem

Crossover Rate	0.8
Mutation Rate	0.05
Population size	400
Chromosome Length	20
Maximum Number of Generations	200
Normalizing Value	532

Running the J. C. Nickles problem under a Windows compiled version of the software on three computers (each 33 MHz-386 Intel based machines), for 100 repetitions on each machine took 6.25, 5.5, and 6 hours respectively (an average of 5.92 hours per 100 repetitions, or 17.75 hours for 300).

The difference in the time each PC to complete 100 cycles can be attributed the fact that although the three PCs were identically configured, they did not have the same effective throughput.

This section (and all subsequent sections of this chapter) will only use the graph illustrating the most effective number of generations that the GA should evolve. This gives a better perspective of the efficacy and efficiency of the GA. Another benefit of the efficiency diagram is that the large number of trials required to generate the figure helps alleviate any concern that the pseudo-random nature of the random number generator is responsible for the success of the GA.

This graph has the same general shape as the corresponding graph for the Dakota problem (Figure 2.8). The major difference is that Figure 2.9 is much more linear (particularly from generation 80 onward). The linearity is a function of the success (or relative lack of success) the GA had solving the problem in 80 generations or less. Once the GA had reached 80 generations, if it had not solved the problem, it was unlikely to do so. The graph depicting the GA's probability of success only reaches 58%. Based on three hundred runs, 42% of all attempts to solve this problem with a GA (given the parameters used to create this figure) will not find a solution in less than 250 generations, no matter how long it searches. If approximately one million chromosomes were methodically constructed, at least one solution would be found (400 chromosomes per generations x 2621 generations = 2^{10} chromosomes). But, because of the nature of the GA and the shape of this solution space, once it converges in a sub-optimal region of the search space, it is unlikely to leave.

The most useful information (in terms of efficiency) from this figure is that for the GA to find a solution with 99% certainty it would be most efficient to evolve 20 generations per cycle, and run the GA fifteen times.

Based on the time it took to evolve 300 generations, this would take approximately 17.75 hours (20 x 15 = 300). Each of the 15 runs could be run independently (reducing the time required to approximately one hour). It is this amenability to parallelization that makes the GA such a potentially powerful tool.

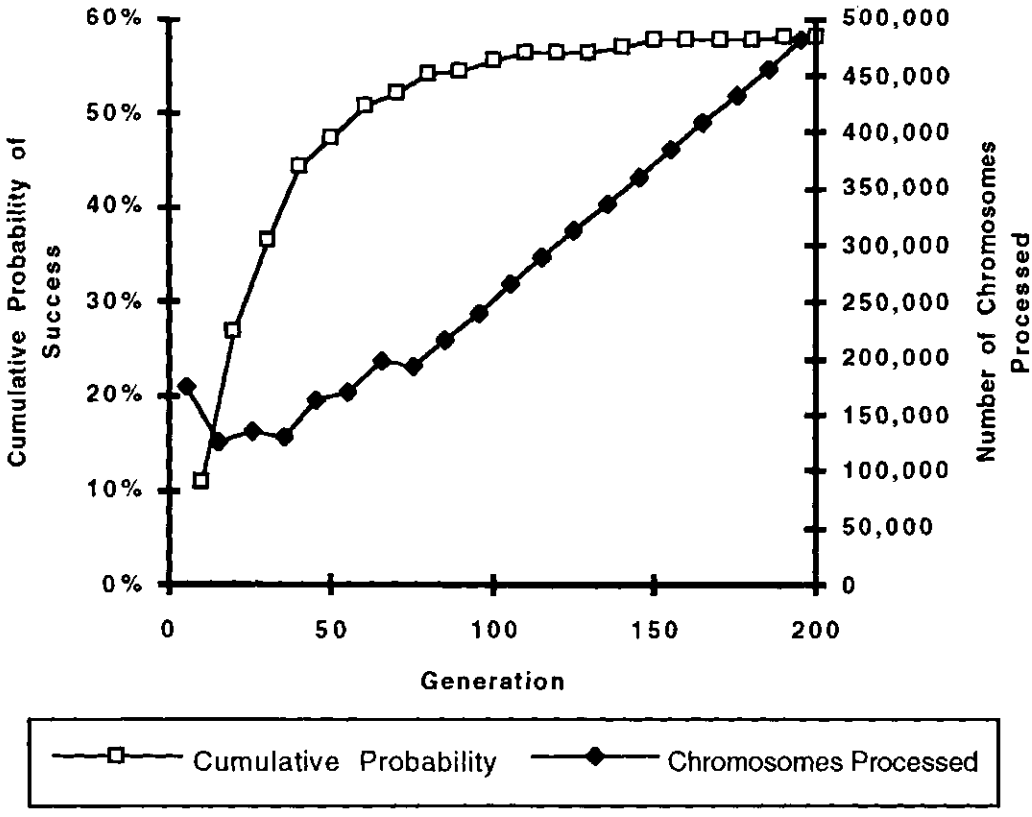


Figure 2.9 A Graph of the Cumulative Probabilities and Number of Individuals Processed vs. The Number of Generations a GA Evolves (J. C. Nickles problem)

2.2.2 The J. C. Nickles Problem with Chromosomes Repair

This is not a new IP problem, but a different approach to the previous one. The previous implementation was quite inefficient, which ultimately

required a large number of repetitions before a solution could be found. The primary source of inefficiency was the lack of restrictions placed on generating illegal chromosomes. With the chromosome structure implemented for this problem, there was little that could be done to prevent 'illegal' chromosomes from being created. The approach taken was to fix or repair each chromosome violating a constraint. This removes all illegal chromosomes from the population.

There are three ways that a chromosome could be illegal. A type 1 constraint (a region can only send its cheques to one city) can be violated two ways: a region does not send its cheques to any city, or the cheques are sent to more than one city. Figure 2.10 shows a chromosome segment where cheques from the West are sent to three cities. Any violation of a type 1 constraint is handled by sending the payments to the lowest cost city.

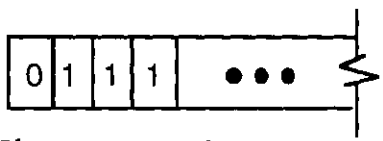


Figure 2.10 Segment of Chromosome Representing the Destination for the West Region's cheques (Chicago, New York and Atlanta)

The repair method could have been designed to randomly select one of the destinations, but a simple heuristic was included (send payment to the lowest cost city). The heuristic does not necessarily force the solution to the optimal combination because having four processing centres open with each region mailing their cheques to the nearest is not the optimum. But, it is the heuristic with which most people would start.

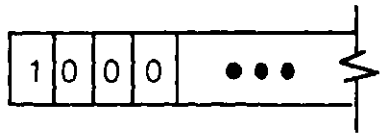


Figure 2.11 Chromosome Segment After Being Repaired

The type 2 constraint (if a city is being sent cheques, the city must have a processing centre) violation is corrected by forcing the opening of a centre any time a region mails a cheque to that city. Again, knowledge about the problem is exploited as part of the repair mechanism.

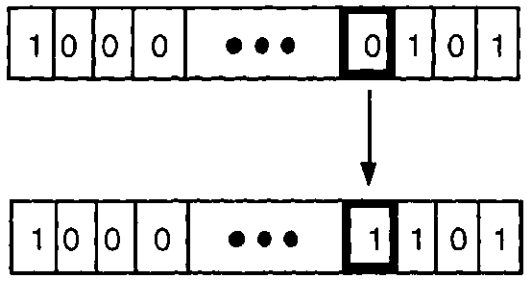


Figure 2.12 Correcting a Type 2 Constraint Violation. The West Region is sending its cheques to Los Angeles so a processing centre is opened.

Using these repair mechanisms, the J. C. Nickles problem was re-run using the parameters shown in Table 2.10.

Table 2.10 Genetic Algorithm Parameters for the J. C. Nickles Problem

Crossover Rate	0.8
Mutation Rate	0.05
Population size	200
Chromosome Length	20
Maximum Number of Generations	200
Normalizing Value	532

Note that the only parameter change was the population size. The population size was cut in half (from 400 to 200). It was postulated that if the new method is effective, a reduction in the population size should have almost no effect on the success rate. An additional benefit of reducing the population size was that it also reduced the required processing time.

The problem was run on the same equipment as in the previous example. The run was distributed between the same three PCs. The time required by each was 120 minutes (141 runs), 105 minutes (150 runs), and 80 minutes (100 runs). The total time required was approximately five hours, or 100 minutes per 100 runs (compared to 355 minutes per 100 runs without chromosome repair).

As can be seen in Figure 2.13, the success rate was 90%. Without repair, the success rate was only 60%. Even with a smaller population size, the solution rate was increased by fifty percent. The higher success rate (fewer runs went the full 200 generations) coupled with the smaller population size were primary factors in reducing the time required to solve this problem. A significant number of solutions were generated in the initial population (Generation 0). Subsequently, the graph shows that when using the repair method for this problem, it is better to repeatedly generate a random initial population and not create any offspring. Basically what has happened is that the heuristic is effective enough (for a problem of this size) for a random walk to be more efficient than the GA.

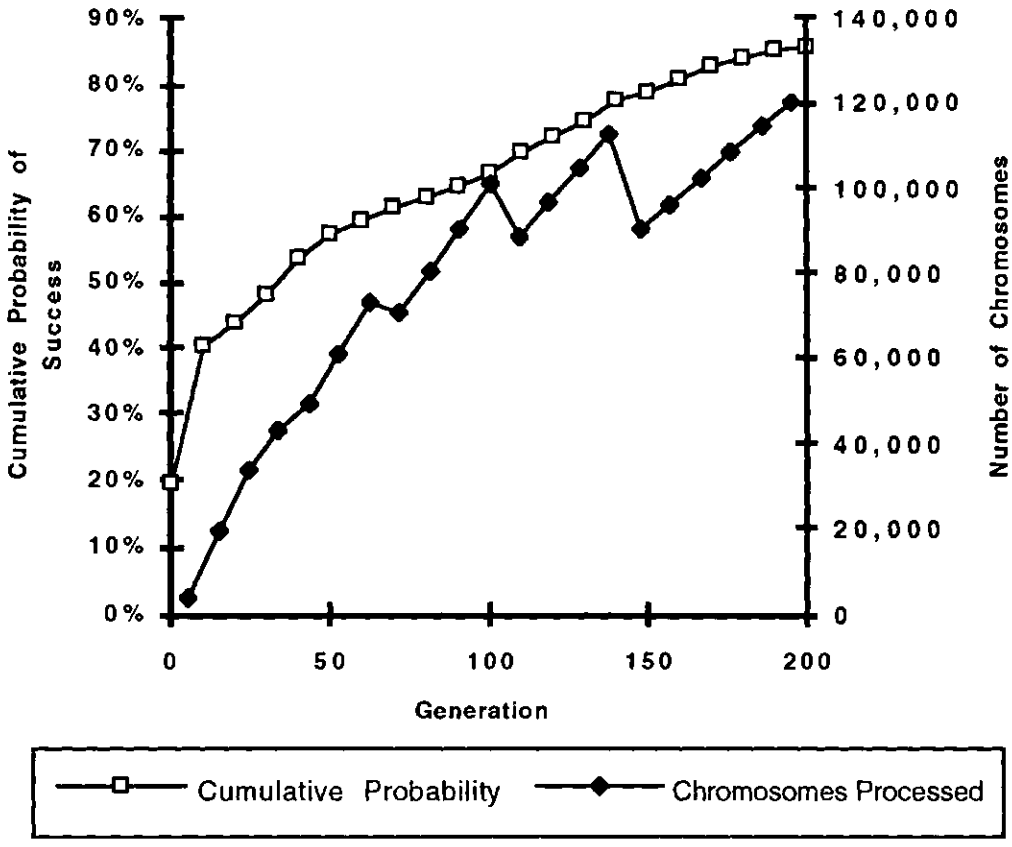


Figure 2.13 Graph of Cumulative Probabilities and Number of Individuals Processed vs. The Number of Generations a GA Evolves (J. C. Nickles Problem with Chromosome Repair)

2.3 Job Scheduling

The job scheduling problem is another classic problem. The basic premise is how can a sequence of jobs to be performed on one machine (each with a specified starting and finishing time) and be ordered in such a way that the total delay is minimized. The job scheduling problem will introduce the use of a semi-proportional penalty function.

The problem used in this section is from a suite of problems used to test a heuristic developed in the 1970s (Chowdhury 1977). There are 8 jobs with the following completion times: 1, 2, 8, 9, 10, 12, 13, and 16. Each is required to start at the same time, but only one piece of equipment is available to complete the tasks. The objective is to schedule the jobs so that the standard deviation of each job's delay is minimized (a slight modification of the classical description of the problem). For example, if there were three jobs (taking 1, 2, and 8 time steps to complete) and they were performed in sequence, the total completion time would be 1+2+8=11, but the delay in starting the first job is 0, the second job is 1, and 3 for the last job. The standard deviation of the delay (0.817) would be found, and then the jobs would be shuffled in search of a smaller standard deviation. A three job problem is simple since there are only 3!=6 permutations of job sequences. But, because of the nature of combinatorial problems, a seemingly simple problem of eight jobs and one machine is surprisingly complex (8!=40,320 different sequences of jobs).

The optimal solution to this problem has a standard deviation of 15.94.

The binary structure used previously can be applied to this problem as well. The largest job number is eight, which can be represented by a binary string three long. The chromosome would be subdivided into eight regions, each representing a job number. The total chromosome length would be 8x3=24. The search space contains 2^{24}=16,772,216 points. Not all of the points are valid, so a penalty function was required. This penalty function will be described later in this section.

This problem, like the previous, is a minimization problem. In the previous problem the constant used to convert the minimization problem to a maximization problem was determined by constraints on the model. This problem has no constraints, so this is not an option. An upperbound value could have been used. But, if one could predict an upperbound or worst case job sequence, then a best case could also be found. This would essentially eliminate the need for a GA. To alleviate this problem a 'quick and dirty' approach was used. A constant was arbitrarily selected, and after a few sample runs, it was found that 100 would work satisfactorily. Thus, the constant was redefined as 100.

Once the constant was determined, the issue of selecting a penalty function was addressed. The only illegal structures that can be created in this problem are chromosomes where job numbers are repeated. This would mean that not all of the jobs are completed since the chromosome length prevents any more than eight jobs from being scheduled. In an effort to steer the GA away from producing incomplete production schedules, a penalty function implemented. The function penalizes chromosomes proportional to the number duplicate jobs in the schedule. The penalty fitness function is

$$\text{Penalty Fitness} = \frac{0.2}{(\#\text{ of duplicate jobs})}$$

An example of job duplication would be a schedule with one job being repeated six times, or two jobs repeated four times each. Either way, there would be six duplicates. Table 2.11 contains the various penalty functions for this problem.

Table 2.11 Fitness Penalties Corresponding to the Number of Duplicate Jobs

Number of Duplicates	Penalized Fitness
1	0.2000
2	0.1000
3	0.0667
4	0.0500
5	0.0400
6	0.0333
7	0.0286

A chromosome with optimal fitness would have a normalized fitness of (100-15.94)/100=0.86. A chromosome with only one duplicate has a fitness approximately one quarter of optimal. The constant is sufficiently large that few (if any) chromosomes improve their fitness when the penalty function is applied. Thus, this penalty function could be better described as a proportional fitness function.

The crossover rate for this problem was reduced to the more traditional level of 0.6. The mutation rate was left at the same value as the previous problem (0.05). The population size was set at 200, and the maximum

number of generations was set to 400. Initial experiments found these values to be adequate, so they were not changed.

Table 2.12 Genetic Algorithm Parameters for the Job Scheduling Problem

Crossover Rate	0.6
Mutation Rate	0.05
Population size	200
Chromosome Length	24
Maximum Number of Generations	400
Normalizing Value	100

The GA was designed to terminate under two conditions: once the GA had evolved 400 generations, or, once the raw fitness of a chromosome fell within the range of 254 and 254.4. It should be noted that for increased precision the variance, instead of the standard deviation was stored as the raw fitness. These two values (254 and 254.4) correspond to standard deviations of 15.937 and 15.950 respectively.

The problem was run on PCs configured similarly to those in the previous example. The run was distributed between four PCs, each performing 100 runs. The completion times were approximately 9.25, 9.5, 9.5 and 9.4 hours. The total time required was approximately 37.65 hours, or 9.4125 hours per 100 runs. Figure 2.14 depicts the performance curves for this problem.

The first point that Figure 2.14 illustrates is that the GA was not particularly efficient. The cumulative probability of success was only sixteen percent. The low success rate resulted in a significantly high number of individuals being processed for this problem as compared the previous

examples. Considering that a search space of 2^{24} is approximately 17 million points, processing three million chromosomes is quite inefficient. Also, a combinatorial search of 8!=40,320 is not computationally complex. In fact, a methodical, combinatorial search would have found the solution with approximately one eightieth the effort. The GA appears to have not handled this problem type well at all. The underlying problem was not the GA, but the modeling technique. Although the job got done, the binary chromosome structure did not lend itself to reproducing legal chromosomes from legal chromosomes using the traditional genetic operators (crossover and mutation).

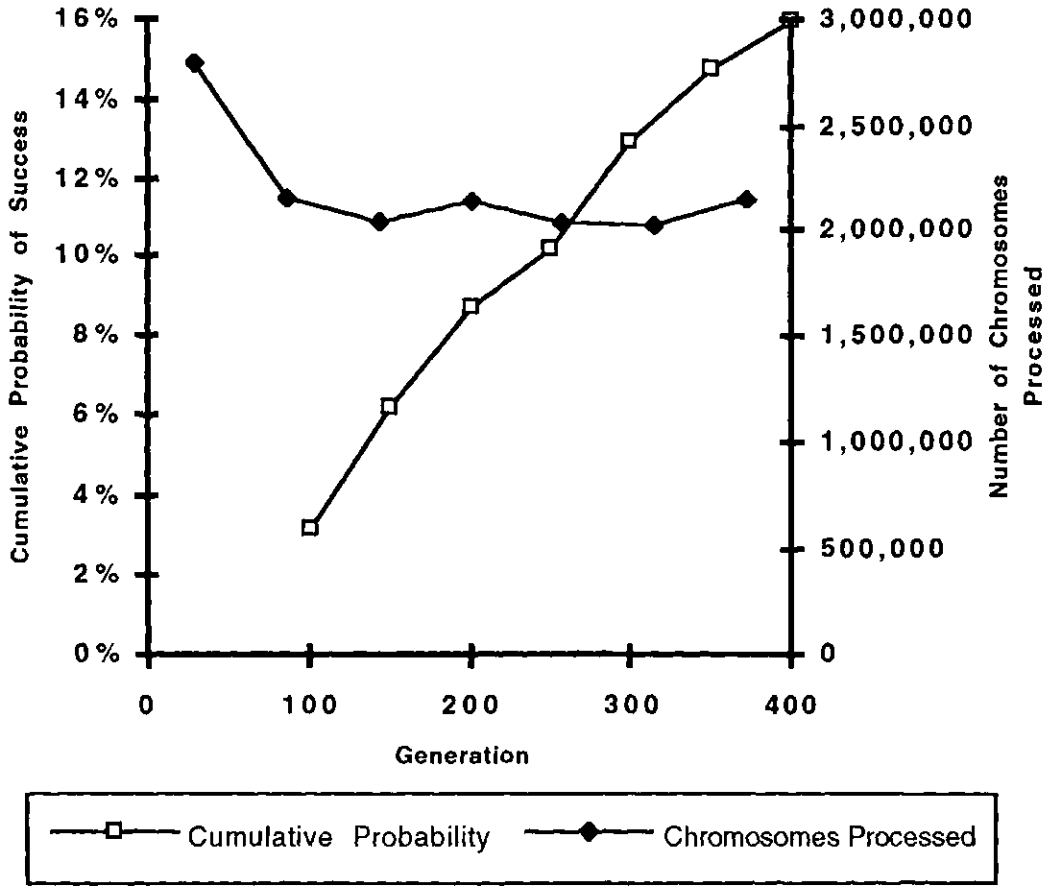


Figure 2.14 Graph of Cumulative Probabilities and Number of Individuals Processed vs. The Number of Generations a GA Evolves (Job Scheduling Problem)

The next problem, the Traveling Salesman Problem, has a similar form to this problem. It will be used to introduce a new genetic structure and genetic operators that will significantly improve the performance of the GA when applied to problems of this form.

2.4 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is one of the oldest recorded optimization problems. The problem is quite simple: given a collection of destinations, in what sequence should they be visited to minimize the distance traveled? The TSP is a problem that is faced in fields such as: marketing (designing sales routes), vehicle routing (school buses, milk trucks, delivery vehicles), communications (cable routing), and computer engineering (circuit design).

The TSP is included to introduce a new genetic structure and operators. The source of this structure and the operators is Michalewicz (1992a). The structure is still an array, but instead of a binary representation, integers are used (an increase in the alphabet size from {0,1} to {1,2,...,9,10}). The example problem is from Larson and Odoni (1981). The salesman must visit ten cities and can select any one of the ten as the origin. This problem is small relative to those used to challenge leading edge TSP algorithms (which until the 1970s could not handle any more than 100 cities), but large enough that it cannot be easily solved by enumerating all possibilities.

The only constraint of the TSP is that each city must be visited. To solve a TSP, one only needs to know how many cities are to be visited, and

the distances between them. Table 2.13 gives the distances between each city. The matrix is symmetric, so only half is shown.

Table 2.13 Half of the Symmetric Trip Matrix Showing the Distances Between Each City

From/To	1	2	3	4	5	6	7	8	9	10
1	0	25	43	57	43	61	29	41	48	71
2		0	29	34	43	68	49	66	72	91
3			0	52	72	96	72	81	89	114
4				0	45	71	71	95	99	108
5					0	27	36	65	65	65
6						0	40	66	62	46
7							0	31	31	43
8								0	11	46
9									0	36
10										0

The optimal tour has a length of 331, and consists of the following sequence:

{1,3,2,4,5,6,10,9,8,7,1}

The structure is quite simple; an array of 10 numbers, with each position in the array having the potential to take on any value from 1 to 10. The search space's size would be 10^{10} or 10 billion if all of the array positions could be any value from 1 to 10. But, once a point is visited it can not be revisited. This reduces the search space to 10!=3,628,800. The search space is further reduced by the fact that any sequence can be cycled forward or backward (A B C D = D C B A). This reduces the space by half. The space is further reduced by the alphabet size (10) since the salesman can start at any location (A B C D = C D B A). Thus, the actual search space is 10!, but the number of potential unique solutions is only 10!/(2x10)=181,440.

Since this is a minimization problem a normalizing, upper bound constant is required. Similar to the Job Scheduling Problem, it is difficult to find an upperbound feasible solution. But what can be found quite easily is an upperbound infeasible value. This value was found by scanning across each row in the trip matrix and finding the furthest distance between cities. The sum of these distances gives the total length for an infeasible tour.

Table 2.14 Summary of the Upperbound Estimate of the Tour Length of a TSP

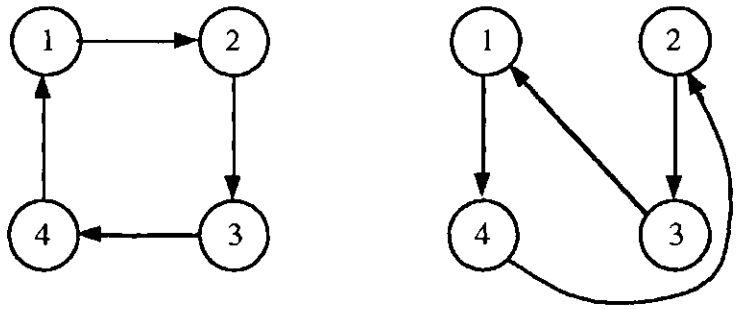
From	To	Length
1	10	71
2	10	91
3	10	114
4	10	108
5	3	72
6	3	96
7	3	72
8	4	95
9	4	99
10	3	114
		Total = 932

The GA was designed to terminate once the optimal solution was found, or after evolving 300 generations.

Crossover is the primary genetic operator. But its form is quite different than the traditional crossover operator. Mutation was also used, but it was also implemented quite differently than in previous problems, and thus its influences are difficult to compare to the traditional model. The initial population was generated by randomly sorting numbers from 1 to 10 for each chromosome.

To explain the crossover method implemented, a 4 city TSP will be used. The two tours are represented by the two arrays in Figure 2.15.

Parent Tour 1	1	2	3	4
Parent Tour 2	2	3	1	4



Parent Tour 1 Parent Tour 2

Figure 2.15 Chromosome and Graphical Representation of Two Parent Tours

The method implemented here does not randomly mix the chromosomes, but implicitly considers the inherent strength of specific sequences. This methodology attempts to exploit the similarities between strong chromosomes, but at the same time attempts to guarantee that tours are complete. This is done by promoting strong edges (the links between the cities). The first step is to select a starting city. Working from left to right, either city 1 or 2 would be chosen. The city with the fewest subsequent connections is selected as the first city. For example, if City A is only connected to one other city, and City B is connected to three others, it is better to choose City A as the starting point. This is because to get to and from City A, one would have to pass through the only city it is connected to twice. In other words, it is more difficult to get to City A than City B from the other cities. Selecting City A first will decrease the likelihood of not being able to construct a complete tour.

Table 2.15 Summary of Available Connections (Linkages)

City	Connected To
1	2,4,3
2	1,3,4
3	2,4,1
4	3,1,2

For this example City 1 and City 2 are connected to 3 others. When this occurs a random selection is made.

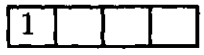


Figure 2.16 First City in New Tour

The next city will be selected from the list of cities that are connected to City 1. City 1 is linked to cities 2, 4 and 3. Again the selection of the second city in the tour is based upon which city has the fewest number of available connections.

Table 2.16 Connections of Cities Connected to City 1

City	Connected To
2	4,3
3	2,4
4	3,2

Since each city is connected to an equal number of cities, a random selection can be made.

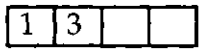


Figure 2.17 Two Cities in New Tour

City 3 is connected to cities 2, 4 and 1. But since 1 has been sequenced, only 2 and 4 are available.

Table 2.17 Connections of Cities Connected to City 3

City	Connected To
2	4
4	2

Both have the same number of connections so a random pick is made. Once the third city is selected, the last city is also known. (Thus, for n cities, there are only n-1 degrees of freedom.)

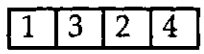


Figure 2.18 A Complete Tour

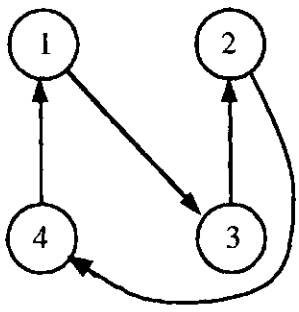


Figure 2.19 Graphical Representation of Offspring Tour

Note that distance was never a factor when constructing the new tour, only sequence and availability. This nearly guarantees that tours are always

completed (any incomplete tours can be either repaired or the algorithm is reapplied to the parents until a legal offspring is created). This method is easily extended to any number of cities. Also note that this version of crossover has only one offspring. Thus, two crossovers must be performed to create two offspring. This additional overhead is more than compensated by the high success rate for creating complete chromosomes (which means no costly repairs are required).

Mutation is no longer performed as the genes are transferred to the offspring. If it was, infeasible tours would be created. Instead of performing mutation at the gene level, it occurs after the offspring are produced. Mutation is performed by randomly selecting two cities and swapping their positions.

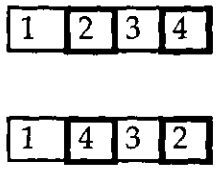
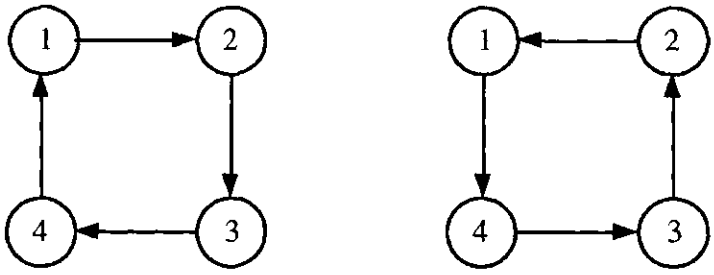


Figure 2.20 Chromosome Representation of Tour Before and After Mutation

Originally, if the probability of mutation was set to 0.1 and the chromosome was 10 long, the expected number of mutations was one per chromosome. Now, the length of the chromosome has no bearing on the expected number of mutations. Mutation will affect a tour one of two ways. The first effect is that it will change the direction of travel. The other possible effect is that a linkage between two cities may be created. This new linkage will change the relative linkage between at least two and at most four cities.

In summary, mutation happens less frequently, but when it does occur, it can make significant changes in the routing.



Before Mutation After Mutation

Figure 2.21 Graphical Representation of Tour Before and After Mutation

A population size of 300 was selected, and the number of generations to be evolved was set at 300 (or terminate if the optimum of 331 was reached). To get a large mixing of chromosomes a crossover rate of 0.8 was chosen. The mutation rate was set to 0.1. Because of the difference between how this mutation operator functions and how the traditional mutation operator functions, no accurate comparisons of the magnitude of this quantity can be made.

Table 2.18 Genetic Algorithm Parameters for the TSP

Crossover Rate	0.8
Mutation Rate	0.1
Population size	300
Chromosome Length	10
Maximum Number of Generations	300
Normalizing Value	932

The TSP problem was run on the same Intel based PC's described in earlier problems. Running the GA under a Windows compiled version of the software on various PCs took a collective 835 minutes for 644 runs, or approximately 130 minutes per 100 runs.

The shape of the curves in Figure 2.22 follow the expected shape. The cumulative curve reached the 100% level indicating that all of the runs found the exact solution. The efficiency curve shows that the optimal number of generations for a population of 300 is 210 (63,300 individuals processed).

The job scheduling problem (many jobs, one machine) could also have used a chromosome structure like the one used here for the TSP (Fox and McMahon 1991). Consider that both are concerned only with the sequence of events. The only difference is in how distance is measured (time versus length) and the measure of merit (standard deviation of delay versus distance traversed).

2.5 Transportation Problem

The transportation problem is the name given to any problem where the distribution of some good must be allocated between a set of source points and destination points. The objective of these problems is usually to minimize distribution costs. Problems of this type include production (factory to market), power generation (power plant supplying a city/factory), and human resources (employees and jobs).

The primary objective of this section is to explain and demonstrate the last and most complex chromosome structure, the matrix. Along with a new

chromosome structure, new reproduction and mutation operators will also be introduced (Michalewicz 1992a).

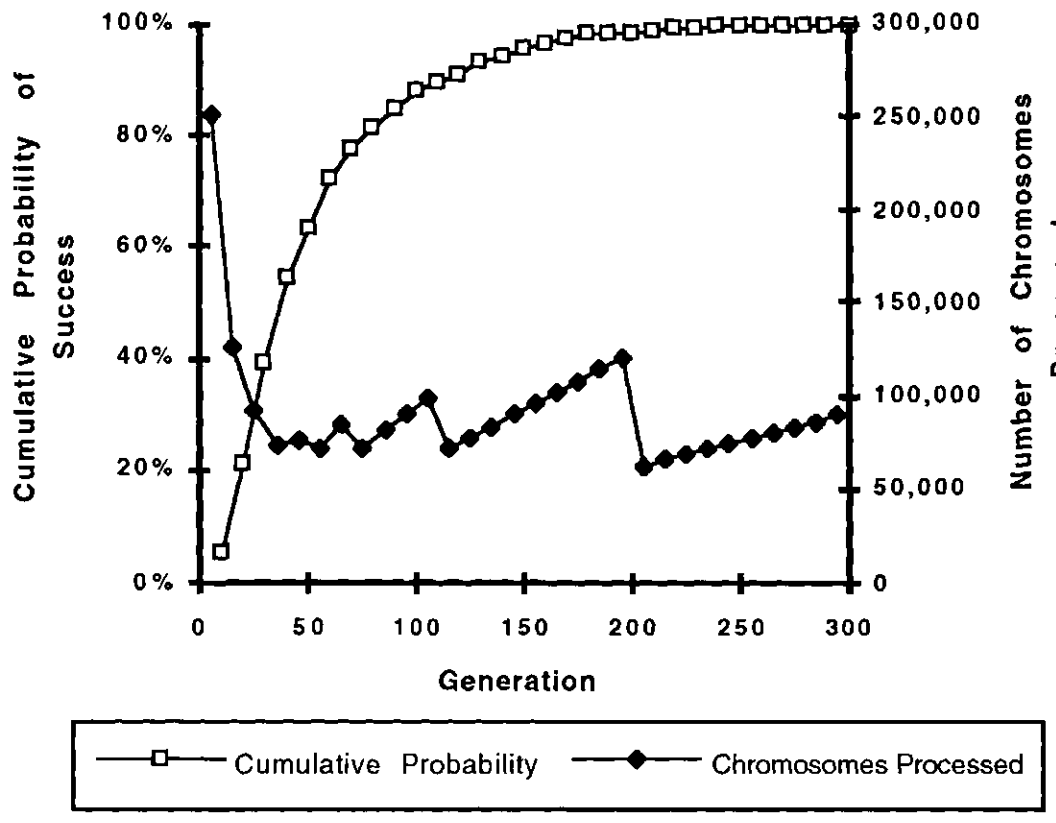


Figure 2.22 Graph of Cumulative Probabilities and Number of Individuals Processed vs. The Number of Generations a GA Evolves (TSP)

The problem that will be used in this section is from Wilson (1989). A good is supplied from three locations {35 50 40}, and is equal to the total demand from four locations {45 20 30 30}. The costs of moving the goods from each supply point to the destination point can be found in Table 2.19.

Table 2.19 Cost Matrix Between Source and Destination

Source\Destination	1	2	3	4
1	8	6	10	9
2	9	12	13	7
3	4	9	16	5

The optimal solution has a minimum transportation cost of z=1020. The allocation of goods is shown in Table 2.20

Table 2.20 Optimal Allocation of Goods Between Source and Destination

Source\Destination	1	2	3	4
1		10	25	
2	45		5	
3		10		30

As previously stated, the chromosome is represented as a matrix, with each element of the matrix having an integer value. The search space is bounded by the supply and demand constraints. Since no row can have a sum larger than the corresponding supply level and no column can have a total larger than the demand level, the search space is less than infinity, but it is difficult to estimate its exact size. An approximate upperbound would be the permutation of each supply and/or demand constraint taken 3 times.

$$\text{Upper Bound} = \max[45P3 \times 20P3 \times 30P3 \times 30P3,\ 35P4 \times 50P4 \times 40P4] = 1.523 \times 10^{19} \qquad (2.26)$$

Defining an upperbound constant to find the optimal minimum is relatively straight forward. The first step was to find the highest supply cost (multiply the supply value from each node with the highest associated transportation cost) and the highest destination cost (multiply the total

demand by the highest associated transportation cost). These calculations are shown below.

$$\begin{bmatrix}35 & 50 & 40\end{bmatrix}\begin{bmatrix}0 & 0 & 10 & 0\\ 0 & 12 & 0 & 0\\ 0 & 0 & 16 & 0\end{bmatrix} = 1640 \quad \text{(Supply constraints)} \tag{2.27}$$

$$\begin{bmatrix}0 & 0 & 0 & 9\\ 0 & 12 & 0 & 0\\ 14 & 0 & 16 & 0\end{bmatrix}\begin{bmatrix}45\\ 20\\ 30\\ 30\end{bmatrix} = 1620 \quad \text{(Destination constraints)} \tag{2.28}$$

The larger value (1640) represents an infeasible supply allocation (the demand at destination nodes 1 and 4 are not satisfied). The smaller value (1620) represents an infeasible demand allocation (Sources 1 and 2 are not exhausted, and Source 3 is overdrawn). Since 1620 will still be larger than the feasible highest cost solution the constant was set at 1620. The optimal chromosome will have a normalized fitness of 0.3704.

A penalty function is not required for this problem because illegal chromosomes are not produced.

The genetic operators used with this structure are quite interesting. The operators are designed in such a way that all chromosomes created are legal. Before the operators are described, the method used to generate an initial 'legal' population should be discussed. Each matrix is generated from the upper left corner, from left to right, down to the lower right corner. The value of the upper left corner of the matrix is a randomly generated number between 0 and the min(Source 1, Destination 1)=min(35,45). This prevents too many goods (more than can be supplied or received) from being shipped.

	45	20	30	30
35	20			
50				
40				

Figure 2.23 First Random Value Placed in Allocation Matrix

The next value in that row is the min(amount not yet supplied from Source 1, the demand at Destination 2)=min(35-20,45).

	45	20	30	30
15	**20**	15		
50				
40				

Figure 2.24 Allocation of Supply To Next Demand Point

	45	5	30	30
0	**20**	**15**	0	0
50				
40				

Figure 2.25 Allocation of Supply To Remaining Demand Points

This is repeated throughout the row. All of the supply from the source in that row is distributed. Each row is allocated in this fashion The only difference is that now each matrix element must equal the min(amount left to be supplied, amount left to be received).

	25	5	30	30
0	**20**	**15**	**0**	**0**
25	25			
40				

Figure 2.26 Random Allocation of Supply To First Demand Point

	0	5	30	30
0	**20**	**15**	**0**	**0**
25	**25**	5	20	0
40				

Figure 2.27 Allocation of Supply To Remaining Demand Points

The exception is in the last row. The leftmost element is not randomly generated, but calculated so that the supply and demand constraints are all met. If at the end of generating the matrix all of the supply and demand constraints are not met, the allocation matrix is discarded and a new one is generated.

	20	0	10	30
0	**20**	**15**	**0**	**0**
0	**25**	**5**	**20**	**0**
40	0	0	10	30

Figure 2.28 Allocation of Supply To Remaining Demand Points

The actual genetic operators are quite elegant in their effectiveness. The crossover operator is actually an averaging or interpolating operator. The procedure is best described by example. The initial parent matrices are shown in Figure 2.29.

20	5	5	5
10	10	25	5
15	5	0	20

Parent Allocation Matrix 1

5	10	15	5
30	10	0	10
10	0	15	15

Parent Allocation Matrix 2

Figure 2.29 Two Parent Matrices Selected for Crossover

The first step is to create a new matrix which consists of the sum of each element divided by two and rounded down.

12	7	10	5
20	10	12	7
12	2	7	17

Rounded Average Matrix

Figure 2.30 The Rounded Average Matrix

The next step is to create a matrix that contains the remainder from the previous operation.

1	1	0	0
0	0	1	1
1	1	1	1

Remainder Matrix

Figure 2.31 Matrix Containing Remainders From the Rounded Average Matrix

Each parent allocation matrix is feasible, the corresponding rows and columns add up to the same values. Consequently, the Remainder Matrix will always have an even number of 1s in each row and each column. The 1s in the Remainder Matrix are split evenly to create two new matrices, each with a row sum and column sum which is half of the row and column summations of the remainder matrix. (This is done by finding polygons defined by the 1s and allocating odd and even corners of the polygons to the two new matrices.)

1	0	0	0
0	0	1	0
0	1	0	1

Half Remainder Matrix 1

0	1	0	0
0	0	0	1
1	0	1	0

Half Remainder Matrix 2

Figure 2.32 The Half Remainder Matrices

The new matrices (offspring) are created by adding the Rounded Average matrix and the Half Remainder matrix.

13	7	10	5
20	10	13	7
12	3	7	18

Child Allocation Matrix 1

12	8	10	5
20	10	12	8
13	2	8	17

Child Allocation Matrix 2

Figure 2.33 The Offspring Matrices

Note that the two offspring matrices meet all of the supply and demand constraints.

The mutation operator is also quite simple. With the transportation matrix, changing one element affects three others. This is because changing an element means that the supply and demand for that element of the allocation matrix will not be met unless another element in that row (and one from that column) is increased. The fourth element changed is the element which shares the intersection of the row and column of the two elements that were also changed. The mutation operator requires four matrix elements to be selected. This is done by randomly selecting two rows and two

columns. The intersection of these two sets (rows and columns) are the four necessary elements (the elements not shaded in Figure 2.34).

18	7	10	5
20	10	18	7
12	3	7	18

Child Allocation Matrix 1

Figure 2.34 An Offspring Matrix with Elements Selected for Mutation

These four points create their own 2x2 matrix. The upper left element is a random number between 0 and the existing value.

4	

Figure 2.35 A Randomly Generated Element of the Mutation Matrix

Since the sum of the rows and columns of this sub-matrix can not change, (or the original Child Allocation matrix will become invalid) the elements sharing the row and column change.

4	8
13	

Figure 2.36 Updating the Row and Column Elements

This in turn requires the fourth element to change. These updated values are then placed in the elements of the original matrix, resulting in a mutated, but legal matrix. Thus, although there are four elements, there is only one degree of freedom for this matrix.

4	8
13	4

Figure 2.37 Completed Mutation Matrix

18	4	10	8
20	13	18	4
12	3	7	18

Child Allocation Matrix 1

Figure 2.38 Offspring Matrix After Mutation is Completed

The control parameters of crossover and mutation were determined experimentally. The data structure and the basic algorithm for this problem type were described by Michalewicz (1992a) but he did not mention any parametric values. After reviewing notes from the documentation of his source code Genetic2 (Michalewicz 1992b) it was found that the recommended rates were one crossover per forty chromosomes per generation, and a relatively high mutation rate of ten percent. Consequently the standard values of 0.8 (crossover rate) and 0.05 (mutation rate) gave very poor results. After experimenting with Michalewicz's values, the rates were set at 0.02 (crossover) and 0.2 (mutation). The population size was varied (50, 100, 150 and 200) and termination conditions were set at 500 generations, a fitness less than or equal to 1030 (1020 was optimal, and 1030 is approximately one percent more than optimal). The crossover rate must be set low because a large amount of change occurs during reproduction. Usually reproduction explores some intermediate location between the two parents. But, under this operator an intermediate chromosome may occupy a new region of the solution space. The mutation operator is still quite subtle in its effects, but,

similar to the TSP problem, it is so different in comparison to the traditional implementation that a comparison of its magnitude is fruitless.

Table 2.21 Genetic Algorithm Parameters for the Transportation Problem

Crossover Rate	0.02
Mutation Rate	0.2
Population size	50,100,150,200
Chromosome Length	(3x4)=12
Maximum Number of Generations	500
Normalizing Value	1620

This problem was run on a Silicon Graphics (SGI) parallel computer, using the SGI supplied C compiler (in ANSI-C mode). The speed of the equipment reduced the time requirements of the GA to the point that running the GA for four population sizes was feasible. The most efficient population size was 100 chromosomes. The results from this run are illustrated below.

The success rate for this population size was approximately 40%, but the smaller population size compensated for the efficiency of larger populations. The optimal number of generations was 60. Thus, the optimal combination of population size of 100 chromosomes, running 60 generations per run, for 15 runs should find the solution to the problem with 99% certainty. The total number of individuals processed would be 91,500.

The four runs were plotted together to determine whether there was an indication if extremely large or small population sizes are better for this problem. The curves for probability and the number of individuals processed are shown below.

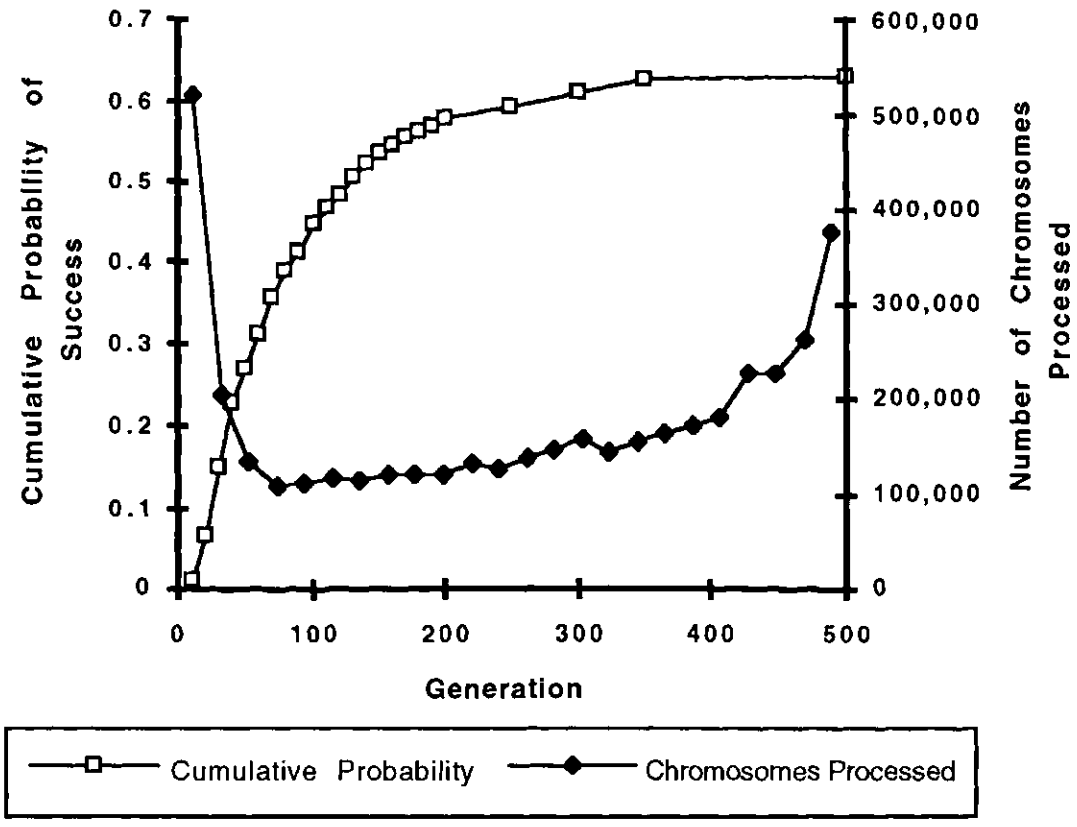


Figure 2.39 Graph of Cumulative Probabilities and Number of Individuals Processed vs. The Number of Generations a GA Evolves (Transportation Problem)

It should be noted that all of the runs used the same initial population. An initial population of 200 chromosomes was generated and saved. Each run, would use this population as its initial generation. For the runs which required fewer than 200 chromosomes, only the first 50, 100, or 150 chromosomes were included in the initial generation. Using a common initial population was done for two reasons. The primary reason was for speed. Generating the initial population had the same computational overhead as running the GA for 500 generations. Thus, if this was done for the 300 or more runs used to generate the data for Figures 2.40 and 2.41, then

the time required to complete the calculations would have doubled. The second reason was so that each run would be comparable. If each initial population was the same, then no one population size would have an advantage because of a significantly stronger initial generation.

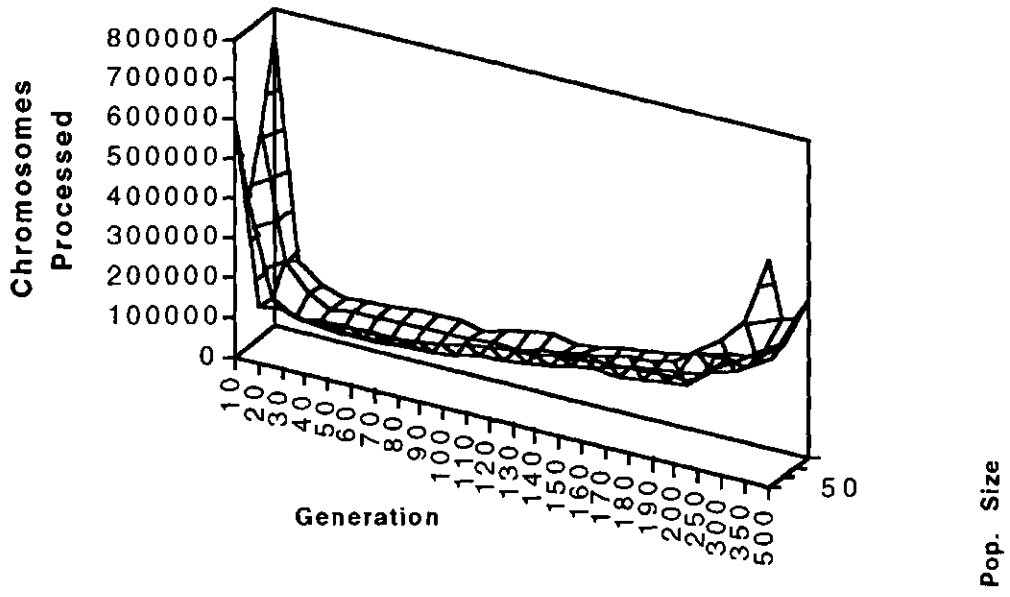


Figure 2.40 Graph of The Number of Individuals Processed vs. The Number of Generations a GA Evolves Given Various Population Sizes (Transportation Problem)

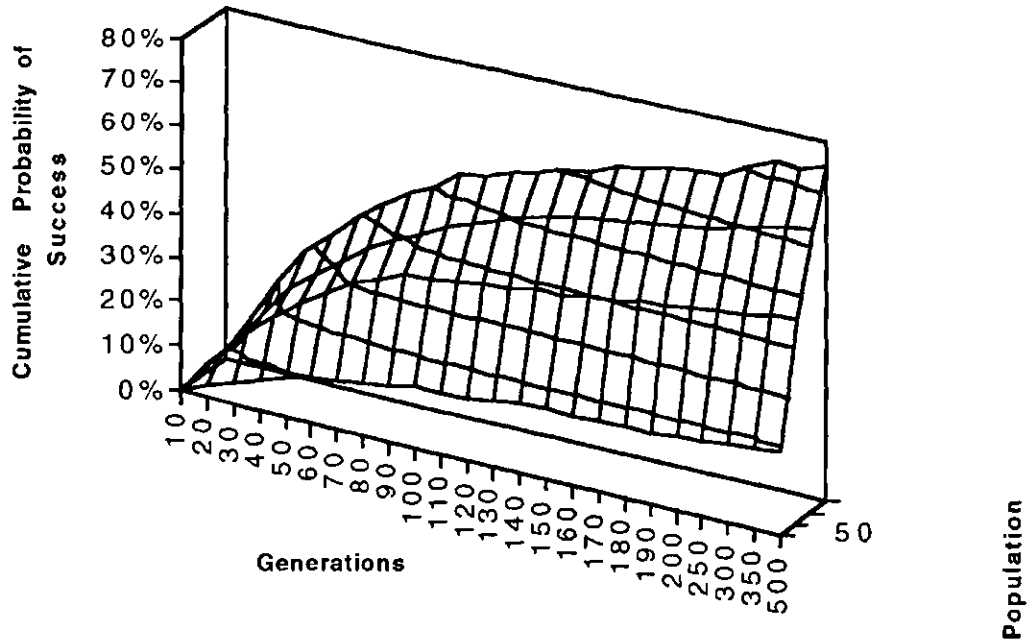


Figure 2.41 Graph of The Cumulative Probability of Success vs. The Number of Generations a GA Evolves Given Various Population Sizes (Transportation Problem)

The surface illustrating the number of individuals processed (Figure 2.40) indicates a valley at the population size of 100 chromosomes. This is followed by a relatively flat plane for population sizes of 150 and up. This indicates that there may be a more efficient population size above the 200 chromosome level. The most likely scenario is that the known local minimum is global. Larger populations will approach the efficiency of the 100 chromosome population but not improve upon this threshold. The probability surface (Figure 2.41) illustrates that as the population size increases the percentage of runs that find an answer increases. This increase in success

does not compensate for the additional overhead associated with the larger population sizes. It should be noted that if the definition of a successful run was relaxed to within five percent of the optimal answer, the success rates of the various population sizes would be those found in Table 2.22.

Table 2.22 Approximate Success Rate For Various Population Sizes Given a Relaxed Solution Threshold (Within 5% of Known Optimum)

	Population Size			
	50	100	150	200
Success Rate	99.2%	99.7%	99.4%	99.0%

2.6 Summary

A large amount of information has been covered in this chapter. The GA was presented solving five problem types through the use of seven sample problems. Table 2.23 below has been compiled to summarize the topics covered in each problem.

At this point, the flexibility of the GA should be evident. There is no question that it is not the most efficient method to solve some of these sample problems, but in general the method will find an acceptable solution. The point that was most likely made was that the GA can not be blindly applied to a problem. Knowledge about a problem, and its inherent difficulties must be considered when designing the GA. The parameters chosen to drive the GA are secondary to the way the problem is engineered. One can take the largest population size that their computer can handle, and use rule of thumb parameters and the GA will converge upon the solution. But without knowing the best chromosome structure, or a good

representation scheme for all of the constraints, and the GA could flounder about aimlessly. This floundering reduced performance to not much better than a random walk. In fact, to paraphrase John Koza (Koza 1993) premature convergence only occurs when the GA doesn't find the solution you are looking for. Good GA engineering can minimize 'bad' premature convergence.

Table 2.23 Summary of the Topics Covered in Chapter 2

Section and Problem Type	Topics Introduced and Covered
2.1 Linear Programming (Giapetto Problem)	• Multi-constraint problem. • Multi-variable representation. • Introduction to penalty functions.
2.1 Linear Programming (Dakota Problem)	• Reliability calculation. • Problems with using the GA for Linear Programming.
2.2 Integer Programming (J. C. Nickles)	• Solving a minimization problem with a GA. • Using the GA in a sparse, non-continuous solution space. • Problems where premature convergence occur. • Use of chromosome repair to improve the solution rate. • Use of problem specific knowledge to improve performance.
2.3 Job Scheduling	• The weakness of the binary representation in some problems. • Implementation of a proportional penalty function.
2.4 Traveling Salesman Problem	• Introduction to an integer alphabet. • Example of the strength of the blind search method used by the GA. • Portability of concepts between like problem types. • Benefits and efficiency of only producing legal chromosomes.
2.5 Transportation Problem	• Introduction to an integer matrix chromosome. • Example of how larger populations are not always best.

3. APPLYING THE GENETIC ALGORITHM TO A REAL WORLD PROBLEM

The Genetic Algorithm (GA) is well suited to solving complex, non-linear problems. Scheduling of any type: transportation schedules, manpower assignment, and equipment usage, fall within this domain. Unfortunately many scheduling problems such as assigning couriers to pick-up and delivery routes must be solved as the need arises, not in a well planned, deterministic environment. This precludes the use of the GA and most other quantitative approaches because of the time they require to find a solution (exact or approximate).

Fortunately there are a significant number of scheduling problems which have a deterministic nature. The doubly constrained traveling salesman problem (DCTSP), which was described in Chapter 1, is a scheduling problem classification that is deterministic in nature. There are a variety of problems that can be modeled as DCTSP: coordination of buses transferring passengers at a terminal; the design of hub and spoke networks for a national courier; and the design of a professional sport league schedule. The problem which will be studied in this chapter is the creation of the National Hockey League's (NHL) 1992-1993 schedule. The reason for choosing the NHL playing schedule was because of the availability of the data. Designing a test problem is a significant task. This is especially true, when the problem must be designed to reflect real-world conditions, and also must adequately test the algorithm or solution in question. Since, the NHL playing schedule problem shares characteristics of other DCTSP and the data necessary to formulate and solve the problem was available, the NHL playing schedule was used as the model problem.

This chapter will follow the basic approach used in Chapter Two: problem background, GA design methodology, a review of the results and interpretation of their meaning; discussion of problems encountered when designing the GA, and finally conclusions and insights about the application.

3.1 The NHL Schedule

Creating a schedule for the NHL is a time consuming and tedious task. The scheduling process is begun each October, eleven months before the start of the next season. There have been several past attempts to create a software package automating the scheduling process. Only recently has any moderate success been achieved. The most recent approach implements a Decision Support System (DSS). The DSS has two modes of operation. One mode allows the scheduler to manually generate a schedule which is then evaluated on preset criteria. The second mode has the DSS automatically generate the schedule, with the scheduler intervening intermittently to fine tune the process (Fleurent and Ferland 1991). The drawback of this method is that it does not allow the scheduler to simultaneously experiment with a wide variety of approaches. Under this approach, a potential solution is continuously refined until it meets a variety of constraints. Alternatively, the GA explores a wide variety of potential schedules before refining a select few to meet the league's requirements.

All of the major North American professional sports leagues, except for hockey, have had some sort of computerized scheduling since the 1980s. Hockey is unique because of the combination of the league's size (24 teams), the large number of games each team plays (41 home, and 41 away), and most teams play in a facility where they are one of many tenants.

Football, specifically the National Football League (NFL), has 28 teams, and their season consists of only 16 games. Most NFL teams are the primary tenants at their stadiums, and rarely have to worry about scheduling around the needs of other tenants during the course of their season because teams play only once a week, and almost exclusively on Sundays.

Similarly the Canadian Football League has 12 teams. Each team plays once a week in a season that is 18 weeks long. Again, the home facilities for most of the teams in the league are one tenant facilities.

Major League Baseball has approximately the same number of teams as hockey (26 overall), but they are split into two leagues (the 12 team American League and the 14 team National League). The teams only play teams in their own league during the regular season. Since scheduling is a combinatorial problem, a league half the size of the NHL is significantly easier to schedule than the NHL's. Of all the professional leagues, baseball teams play the most games per season (over 160 games), but the teams will play a series of three to four games over three to four days against a specific opponent. This significantly reduces the amount of travel required of each team. In hockey, if Team A hosts Team B, the earliest that Team A will host Team B again is seven days.

Basketball is the sport most comparable to hockey in terms of league size (28 teams) and number of games played (each team plays 81 games in the regular season). In most cities where basketball shares facilities with hockey, hockey has traditionally been the 'poor cousins' in terms of box office

attendance and revenues, giving basketball a preferred status in terms of selecting booking dates.

The comparison between the major sports gives one a general feel for why hockey is more difficult to schedule. The complexity of the NHL schedule is better understood after looking at the numerous constraints that must be met. In the 1992-93 season the league had 24 teams divided into two conferences, which in turn was further divided into two divisions of 6 teams.[1] Table 3.1 shows the league alignment.

Table 3.1 NHL League Alignment for the 1992-1993 Season

Prince Of Whales Conference		Clarence Campbell Conference	
Adams Division	**Patrick Division**	**Norris Division**	**Smythe Division**
Boston Bruins	New Jersey Devils	Chicago Blackhawks	Calgary Flames
Buffalo Sabres	New York Islanders	Detroit Red Wings	Edmonton Oilers
Hartford Whalers	New York Rangers	Minnesota Stars	Los Angeles Kings
Montreal Canadiens	Philadelphia Flyers	St. Louis Blues	San Jose Sharks
Ottawa Senators	Pittsburgh Penguins	Tampa Bay Lightning	Vancouver Canucks
Quebec Nordiques	Washington Capitals	Toronto Maple Leafs	Winnipeg Jets

The NHL plays a fully interlocking schedule, which means that every team plays every other team at least once at their home facility. Table 3.2 illustrates the allocation of games between all of the teams. This game allocation table is determined by the NHL teams' GMs, as coordinated by the league office. The fully interlocking schedule significantly increases the complexity of the league's scheduling process. Each column in the table is headed by the home team's number. Reading down the column gives the number of times a team hosts the team in that row. For example, Boston (Team 1) hosts Buffalo (Team 2) three times. Buffalo hosts Boston four times.

[1] The league abandoned this alignment at the completion of the 1992-1993 season when the league added teams in Los Angeles, Miami and the Stars moved to Dallas.

The 41s at the bottom of each column are the total number of home and away games each team plays.

Note that although every team plays 41 games at home and 41 away, the schedule is not symmetric. Teams will host teams within their division 3, 4, or 5 times. The difference between hosting and visiting the same team is never more than one game. In other words if Team A hosts Team B for three games, it will not visit Team B any more than four times. The only symmetry is that each team hosts a team from outside its conference once. Teams will host teams that are in its conference, but not in its division, either once or twice.

Table 3.2 Allocation of Games Between Teams for the 1992-1993 NHL Season

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
BOS	0	4	3	5	3	4	1	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1
BUF	3	0	5	4	4	3	1	1	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1
HFD	4	4	0	3	4	4	2	2	2	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1
MTL	4	3	4	0	3	4	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1
OTT	4	3	3	4	0	4	2	2	1	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1
QUE	3	4	3	3	5	0	2	1	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1
NJ	2	2	2	1	2	2	0	3	3	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1
NYI	2	2	2	2	1	2	4	0	3	4	4	3	1	1	1	1	1	1	1	1	1	1	1	1
NYR	1	2	1	2	2	2	4	4	0	3	3	5	1	1	1	1	1	1	1	1	1	1	1	1
PHI	2	1	2	2	1	2	3	5	4	0	4	3	1	1	1	1	1	1	1	1	1	1	1	1
PIT	2	2	2	1	2	1	5	3	4	3	0	4	1	1	1	1	1	1	1	1	1	1	1	1
WSH	2	2	2	2	2	1	3	4	4	4	3	0	1	1	1	1	1	1	1	1	1	1	1	1
CHI	1	1	1	1	1	1	1	1	1	1	1	1	0	5	4	3	3	3	2	2	2	2	2	1
DET	1	1	1	1	1	1	1	1	1	1	1	1	4	0	4	4	4	3	2	2	2	2	1	1
MIN	1	1	1	1	1	1	1	1	1	1	1	1	3	3	0	5	4	4	2	2	1	1	2	2
STL	1	1	1	1	1	1	1	1	1	1	1	1	4	3	4	0	4	4	2	1	2	1	2	2
TB	1	1	1	1	1	1	1	1	1	1	1	1	4	3	3	3	0	5	1	2	2	2	2	2
TOR	1	1	1	1	1	1	1	1	1	1	1	1	4	4	3	3	4	0	2	2	2	2	1	2
CGY	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	2	2	2	0	4	4	5	3	3
EDM	1	1	1	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	3	0	3	4	4	5
LA	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	1	1	2	3	4	0	4	4	4
SJ	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	4	3	3	0	4	4
VAN	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	1	2	4	3	5	3	0	3
WPG	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	1	4	4	3	3	4	0
	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41

Besides the league's game allocation, the schedule must meet some basic constraints. Table 3.3 contains the guidelines or constraints that the NHL follows when generating a schedule. Note that the right hand column

qualifies the constraint's type as either hard (H) or soft (S). A hard constraint is a constraint which must be met. Most hard constraints are easily met. If this was not the case valid solutions would be few and far between. For instance, the first constraint gives the number of games which must be played, and when they are to be played. Since the league defines the allocation table of who plays who, as long as any schedule meets that requirement there will be 984 games in the schedule. The second part of the constraint basically states that a game can not be scheduled during the summer. For most problems hard constraints define the basic domain of the problem. The soft constraints (those that don't necessarily have to be met for a solution to be valid) qualifies a solution within a set of valid solutions as being more preferred. The NHL does not specifically qualify their constraints as hard or soft, this has been done by the author. Any constraint which could not be easily renegotiated by the General Managers (GMs), or affected the length or size of the schedule, was classified as a hard constraint.

After the schedule is created, it is submitted to the GM of each team. Each GM can negotiate a change to the schedule. The GMs will attempt to change the schedule to improve their own team's schedule (the dual of improving a team's schedule is making an opponent's schedule more difficult, which may occur by design or by chance). The negotiation process can result in the final schedule violating some of the soft constraints.

Table 3.3 Scheduling Constraints Followed by the NHL

	Constraint	Constraint Type H=Hard S=Soft
1)	Total of 984 games to be played between Oct. 6/92 and Apr. 15/93.	H
2)	The games should be distributed evenly through the 27 weeks.	S
3)	No games may be scheduled on Dec. 24 or Dec. 25 or during the All-Star Game break (day prior to/day of/day following).	H
4)	Games on Boxing Day Dec. 26 must not require more than 2 hours and 30 minutes travel.	S
5)	The number of occurrences of 4 games in 5 dates should be minimized.	S
6)	Certain clubs require a minimum number of home games on specific days of the week.	S
7)	Preferred dates should be scheduled before using Alternate dates.	S
8)	A club which is out of its home building for 7 days should have at least 3 games scheduled during that time (6 games in 14 days).	S
9)	A team should not play more than 7 consecutive games away from it home building.	S
10)	After being away from its home building for 7 days or more, a club should be scheduled on its first available preferred date.	S
11)	Consecutive visits from a team within the home team's own division should be at least 14 days apart (30 days for teams from other divisions).	S
12)	Routing on long road trips should minimize the total distance traveled.	H
13)	The number of weekend games should be maximized. Also, for a club which has 2 potential home dates on a weekend, it is preferable to schedule only one of these, if by so doing, it allows another club (which has only one potential weekend date) to play a weekend home game.	S
14)	Teams in the same division are concerned about playing their opponents at equal strength, which means both teams played the night before, or both teams did not play the night before.	S
15)	Teams from the Smythe Division coming to play a road trip in the east would like at least one game in their division before arriving in the east.	S

3.2 Designing The GA

The GA requires a certain amount of front end design (alphabet size, chromosome structure, adaptation methods, fitness function, control parameters, and termination conditions). This section will outline the methods and parameters used to solve this problem as well as discuss the design behind these parameters.

There are four major preparatory steps in designing a GA: selecting a chromosome structure, identifying the measure of merit, determining an efficient population size, and selecting the termination conditions.

3.2.1 Alphabet Selection and Chromosome Design

The chromosome structure cannot be designed without first selecting an alphabet. A binary representation is the theoretically ideal representation. Unfortunately the chromosome length necessary for a binary representation was prohibitive. Table 3.4 indicates the number of characters necessary had a binary representation been implemented. A 24 character alphabet was selected because of the reduction in chromosome length and the ease of interpreting the chromosomes as a schedule by the casual observer. In a 24 character alphabet, each team is mapped to a letter of the alphabet (Team 1 is mapped to A, Team 2 to B and so forth). This reduces the storage requirements for a schedule to 1x24x180=4320.

After selecting an alphabet the chromosome structure can be designed. Since this scheduling problem is very similar to the Transportation Problem introduced in Chapter 2 its matrix structure was utilized. Each row in the

matrix was assigned to a day in the schedule. Each column was assigned to a home team. Thus, each element of the matrix represented a potential home game date for a team. A letter (over the range of A through X) in one of these

Table 3.4 Memory Required for the Binary Representation of an NHL Schedule

Length of binary string needed to represent 24 teams:	5
Maximum Number of Teams:	24
Length of Schedule:	180
Total Length: 5 x 24 x 180	21,600

elements would represent the visiting team for that game. For example, if there was a 'C' in the element represented by (Day 2, Team 1) it would be interpreted as Team 1 (Boston) was the host for Team C (Team 3, which is Montreal) on Day 2 of the schedule. A blank means that either the team is on the road, or it is home and not playing. This can be quickly ascertained by scanning the row and checking for the team's presence at another facility.

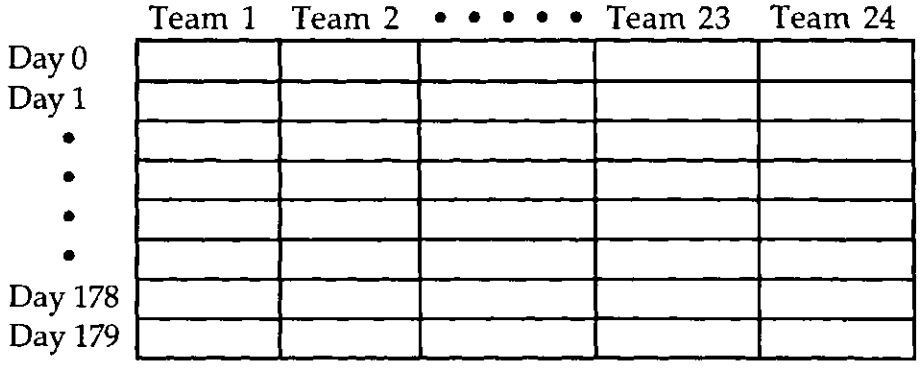


Figure 3.1 Schematic of the Matrix Chromosome Structure

3.2.2 Generating the Initial Population

For this problem, designing the chromosome is only half of the challenge. Usually, generating the initial population is very straightforward

once the chromosome structure has been determined. For the NHL problem, randomly assigning visiting teams to the matrix (shown in Figure 3.1) would not work. Each chromosome must represent a complete schedule. For a schedule to be complete each team must play all of the games they have been allocated (as shown in Table 3.2). Thus, when assigning visiting teams, with home teams this allocation must be taken into consideration.

The availability of the home arena must be considered in addition to the games allocation. For each hockey season, all of the teams submit a list of requested home game dates (at least 50 preferred dates, as well as some alternate home dates). This list of dates acts as an outline of facility availability. Consequently, the final schedule, and ideally each chromosome, should reflect the home date requests of all the teams in the league. In practice this list of requests acts as a starting point for generating a schedule. For the GA, these requests were used as a starting point for generating initial chromosomes, and also acted as a corrective guide, or template, when repairing infeasible schedules.

Generating the home game request template (which will be referred to as the template from here on) was straightforward. The basic objective was to generate a list of 41 home dates for each team. Scheduling games on these dates would guarantee that the home team's facility was available (otherwise they would not have requested it) and reduce the number of teams available to travel on certain dates.

The first step in generating the template was to randomly select a day in the schedule. If the selection was not random, then the template would

generate a bias towards the days selected first (the beginning of the schedule).. This bias would skew the distribution of games.

A selection algorithm was necessary to give preferred requests a higher priority than alternate requests. A key element of the selection algorithm was that no more than twelve games can be played on any one date. The number of requests for home games on a selected day are counted. If the total requests (preferred and alternate) are less than or equal to 12 then the process is simple. Each of the twelve teams are granted their home game request. If there are more than twelve preferred requests, the twelve teams which have the highest ratio of unscheduled games to unfilled requests are scheduled. This ranking system helps minimize the likelihood that the schedule will be short games at the completion of the algorithm.

The reasoning behind this ratio is clearly illustrated in Figure 3.2. If the ratio of unscheduled games to unfilled requests was subdivided into four regions, it becomes fairly obvious that a team with a large number of available home game dates (unfilled requests) and few games left to be scheduled (unscheduled games) should not have any difficulty scheduling the remainder of their schedule. But, if the converse is true (few available home dates or requests left to be filled and several games to be scheduled) then this is a difficult if not impossible task to complete. Consequently those teams which find themselves in the upper left quadrant of Figure 3.2 are given a higher priority when determining which teams' are placed in the template.

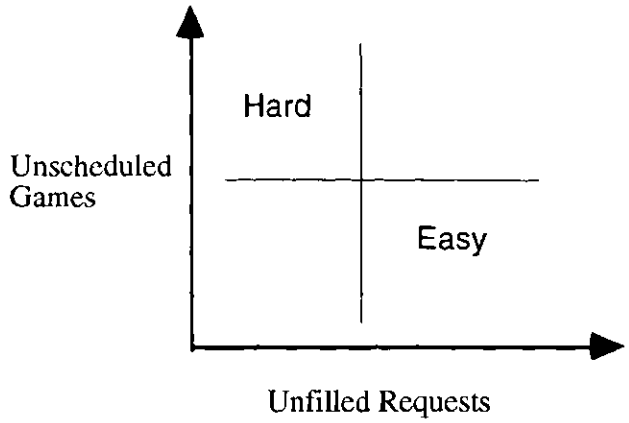


Figure 3.2 Relative level of difficulty when fulfilling game requests.

Finally, if the number of requests exceeds twelve, but the number of preferred requests is less than twelve, all of the preferred requests are granted. The balance of alternate home game requests are assigned on the basis of the teams with the highest ratio of unscheduled games to unfilled requests. This is repeated for each day of the schedule.

The initial population is generated based on the home game template. The schedule is generated on a day-by-day basis. A day is randomly selected, and from the template a list of all the home teams is created. A home team is randomly selected from this list. Next, a list of potential visiting teams is generated. This list consists of all of the teams, not on the home team list, who have not played their allocated games against the selected home team, and who are not already playing on that day. The visiting team is selected randomly from this list. If there are no teams that fit this criteria, the home team is not assigned an opponent. Once the list of home teams is exhausted for that day, a new day is selected.

Loop through the list of dates in the schedule

Randomly select a day in the schedule

Count all of the preferred and alternate requests for this date

If the total number of requests is ≤ 12
    make each team a home team on this date
Else
    If the total number of preferred requests > 12
        Rank the preferred teams based on (unscheduled games)/(unfilled requests), and then select the 12 highest.
    Else
        Save all of the preferred requests
        If the preferred requests < 12
            Rank the teams with alternate requests on (unscheduled games)/(unfilled requests), and then select highest until (preferred requests + alternate requests) = 12.

Figure 3.3 Algorithm for generating the home game request template.

3.2.3 Crossover and Mutation

The biggest advantage of crossover is that it is simple to implement. There are three steps to crossover: select the parents, determine if 'conception' occurs, and locate the crossover point. Each of these steps has a probabilistic element. The parents are selected based on their relative fitness. The more fit a chromosome is, the more likely it is selected to reproduce. After two chromosomes are selected to reproduce, it must be determined probabilistically whether or not they will reproduce. A random number generator is used to simulate a biased coin toss. The bias is equal to the crossover rate selected by the GA's designer. If the chromosomes do not mate, they are passed into the next generation as clones (exact duplicates). If reproduction does occur the crossover point must be determined. The crossover point is randomly selected from 179 potential sites (the first

crossover point is between day 0 and day 1, the last between day 178 and day 179). As is normal for crossover, the tail of one parent is attached to the head of the other. The only drawback to this method is that it can create incomplete schedules. For instance, if two schedules have their games poorly distributed (one has the majority of its games in the beginning of the schedule, the other at the end of the schedule), their offspring will include one schedule with too few games, and one with too many.

Randomly select a day in the schedule

Generate a list of home teams (Home_List) based on the home team request template

Generate a list of visiting teams, which is just the list of teams not in Home_List (Vis_List)

Randomly select a home team from Home_List

Create a list of potential visiting teams based on the following criteria (Pot_Vis_List)

- Already in (Vis_List)
- Has not already played the required allocation of games with home team.

Randomly select a team from Pot_Vis_List
Remove team from Vis_List

Loop until Home_List is exhausted or remaining teams have no match

Loop until list of days in the schedule have been exhausted or remaining teams have no match.

Figure 3.4 Algorithm for generating the initial population of chromosomes.

The mutation operator is quite simple. As the chromosomes are passed to the next generation, mutation is performed on each chromosome

on a daily basis. After reproduction is complete the chromosome is passed to the next generation one day at a time. A simulation of a biased coin flip determines if the day is to be mutated. The bias is equal to the mutation rate, and is selected by the GA designer. Mutation is performed by swapping visiting teams between the home teams. This method does not add any teams to a day on which they are not already playing. An unfortunate side-effect is that it can add games (or delete games) from the schedule. Thus, two teams may not play their allocated number of games.

3.2.4 Chromosome Repair

Not all of the schedules created by the GA are complete. Some schedules have too many games, some not enough. These illegal chromosomes can be created when generating the initial population (although it does not happen very often) as well as during crossover and reproduction. Schedules with too many games are easily fixed. The key is to find all of the pairs of teams scheduled for more than their required allocation and randomly delete excess games. For schedules with not enough games, the repair algorithm is slightly more complex, but still rather straight forward.

- Find each team pair with a deficit allocation.
- Find all of the available open dates where the two teams have the proper (visitor, home) configuration (ideally the home game should fit the template, but it is not necessary).
- Randomly add the appropriate number of games to the schedule.

3.2.5 Fitness Function

The fitness function had to be designed to minimize distance traveled (which implies travel costs are minimized) and yet still maximize the number of constraints the schedule is able to meet. To simplify the objective

function, the minimization component of the problem was converted to a maximization problem. This allows all components of the objective function to 'march to the beat of the same drummer.' To convert the minimization problem to a maximization problem an upper bound constant (UB) was necessary. Subtracting the chromosome fitness from the UB gave a new value to be maximized. As the chromosome fitness decreases (the distance traveled drops) the difference between the UB and chromosome fitness becomes larger. Maximizing this difference is the same as minimizing the distance traveled. The constant also serves to normalize the 'distance fitness' of each chromosome so that it would fall in the range of [0,1].

The most straightforward way to calculate an UB value was to find the absolute furthest each team could travel in the course of a season. This distance is the distance traveled if each team traveled directly home after every game. The calculation is very simple (it is illustrated in Figure 3.2). Multiply the distance or cost matrix (the cost to travel between cities is expected to be proportional to the distance traveled, although it does not have to be since the matrix acts as a lookup table) and the allocation matrix times a scalar of two (this is because the allocation matrix times the distance matrix only gives the distance for a one way trip between a pair of cities).

[A] = Allocation Matrix
[D] = Distance - Cost Matrix
[A][D]•2 = Upper Bound for the Distance Traveled

Figure 3.5 Example Calculation for Upper Bound Constant

The UB was found to be 1,786,184 miles. It was rounded up to 2,000,000 miles to make certain calculations more convenient. The fitness measure for the distance component was

$$\text{Normalized Fitness} = \frac{UB - ChromosomeFitness}{UB}$$

Figure 3.6 Example Calculation of a Chromosome's Normalized Fitness

The above formula only measures the distance traveled. It does not quantify how well the other constraints are met. Each constraint (11 in total) was measured over the range of [0,1]. For most of the constraints the measurement was based on the fraction of teams meeting the constraints.

Two fitness functions were developed.

$$f(x) = \frac{1 + \sum_{i}^{11} constraints_i}{12}(\text{Normalized Fitness}) \qquad (3.1)$$

$$f(x) = \frac{(\text{Normalized Fitness})k + \sum_{i}^{11} constraints_i}{k + 11} \qquad (3.2)$$

The fitness function in equation 3.1 was designed so that if none of the eleven constraints were met, the chromosome would still have some fitness (one twelfth the normalized fitness). This was done to prevent the possibility of a chromosome being assigned a fitness of zero. If a chromosome is given a fitness of zero, there is no chance that it will be able to contribute to the next generation. Because this chromosome has been designed such that all

chromosomes will be complete schedules, it was felt that it was better to give a chromosome some potential to propagate its genetic information, than to have it automatically terminate as if it had been an incomplete and thus useless schedule.

Equation 3.2 was designed to put the normalized fitness and the constraints on more of an equal footing. By varying 'k' the influence of the normalized fitness could be adjusted. As k approaches infinity the function would behave as if distance was the only factor in determining fitness. A value of k=1 made distance no more important than any other constraint. There was no need to modify this function as was done in equation 3.1 since each chromosome is guaranteed to have a non-zero fitness.

After some initial trials the results using the two fitness functions were compared and Equation 3.1 was found to produce better results, Thus, it was adopted for the remainder of the research.

3.2.6 Population Size

Next to be considered was population size. Population size is usually set proportional to the chromosome length. Because of the size of the chromosomes in this model (180x24=4320 characters) the population needed to be as large as possible. Consequently, the population size (2000 chromosomes) was set as large as the computing environment allowed.

The results from trials with a population size of 2000, run for hundreds of generations, were not significantly better than those which had a population size of 100 and were run for several thousands of generations.

Since this problem appeared to benefit from longer evolutionary cycles, the smaller population size was selected since it required less computational effort.

3.2.7 The Search Space

The two most important points about the search space are its size and whether or not all points in the space can be reached by the algorithm.

It is important to study the search space because the GA is a blind search method (it exploits the shape of the space, with no knowledge of the problem), estimating the size of the search space helps estimate the difficulty of the problem. Also, examination of the space may give insight to the shape of the space.

The exact size of the search space is difficult to estimate for this problem because of the inconsistency in the number of games played during each day. But, some approximate estimates can be established. The absolute largest size the search space could be is (180!)(24!). The 24! is the number of ways that you could arrange 24 teams, and 180! is the number of ways you could order the 180 days. This is a gross over estimate since all of the days do not have 24 teams playing at one time. Through the course of a season, there are an average of 6 games played per night. Taking six teams from the 24, they can be arranged in 24!/18! different ways. A more reasonable estimate for the search space is then (180!)x24x23x22x21x20x19. This is a reduction of 18! in the estimated size of the solution space, or approximately a factor of 10^{15}. Considering that 10^9 is a billion, 10^{15} is a significant decrease in the estimate of the solution space's size.

Because of the existence of allocation criteria (a hard constraint) and scheduling guidelines (soft constraints) not all of the points in the solution space described above are members of the valid solution space. All of the points in the above solution space can be reached, but their likelihood of contributing to the next generation depends upon their fitness (a combination of the distance traveled and how well the other constraints are met).

3.2.8 Summary Of Parameters

The software was written in ANSI C, and run on a Silicon Graphics parallel computer. It should be noted that the parallel architecture of this system was not exploited; the equipment was implemented strictly for its processing speed. A population of 2000 chromosomes running for 953 generations took 336,420 seconds (3.89 days). A population of 100 chromosomes running for 5000 generations took 79,562 seconds (22.1 hours).

Table 3.5 GA Parameters for the NHL Problem

Crossover Rate	0.005, 0.02, 0.2, 0.7
Mutation Rate	0.01--0.000 001
Population size	100, 500, 1000, 2000
Chromosome Length	24 x 180 = 4320
Maximum Number of Generations	300--10,000

3.3 Results

A variety of permutations of the parameters in Table 3.5 were implemented while investigating this problem. During the initial testing phase distance was the only constraint the GA was required to optimize. This relatively simple objective function permitted extremely small population

sizes. The population sizes implemented were 10, 50, 100 and sometimes 500. The focus at this stage of the research was to evaluate the GA's ability to converge using traditional crossover and mutation rates of 0.7 and 0.01.

Convergence did occur, but it was quite small. After reviewing the schedules that the GA was building it was determined that the lack of generation to generation improvement was because the schedules generated were of such a radically different structure. Two schedules of relatively equal strength would often have different numbers of games scheduled on the same dates. It was at this point that it was realized that some underlying similarity should be forced onto the schedules. This is where the template concept was introduced to the GA.

The template concept improved the GA performance. From this point permutations of the crossover rate (P_X), mutation rate (P_m), and population size was explored.

In general, five P_X and P_m rates were selected and tested with each of the previously mentioned population sizes. The general process was to check the GA's behaviour with

- low P_X and low P_m
- high P_X and regular Pm
- regular P_X and regular P_m

The general consensus of these tests were that the GA performed best at traditional P_X of 0.7 to 0.9 and P_m of 0.01 regardless of population size.

After this set of testing was complete, the other constraints were

introduced. The parameters used in the previous testing acted as a starting point for this testing. The additional complexity introduced by the additional constraints caused the GA to founder. Based on the author's experience in working with the GA, some basic rules of thumb, consistent with other researcher's experience (Koza, 1992) are:

- if the GA is showing slow steady progress it is a good idea to increase the population size.
- if the GA is behaving as if it is taking a random walk through the data then the crossover rate should be changed.

The GA was foundering so in an effort to increase the generation to generation improvement the P_X was dropped substantially and the P_m was dropped by an order of magnitude. The population size of some of the runs were also increased from the standard 100 to 2000. The low P_X, coupled with the low P_m appeared to improve the performance of the GA. With the lower P_X, it was noted that the larger population size did not appear to outperform the smaller population size. The larger number of generations with the smaller population was able to overcompensate for the larger number of chromosomes but fewer repetitions.

Comparing the results from the best GA run and the actual schedule used by the NHL for the 1992-93 season (Table 3.6) shows that the GA was not able to generate an improved schedule. The GA generated schedule required the league to travel approximately 30% further than the actual schedule. The generated schedule was nearly as successful as the actual schedule in satisfying the league's constraints. But, since the fitness function is strongly influenced by the distance traveled, the overall fitness of the NHL schedule exceeded the GA schedule by approximately 70%

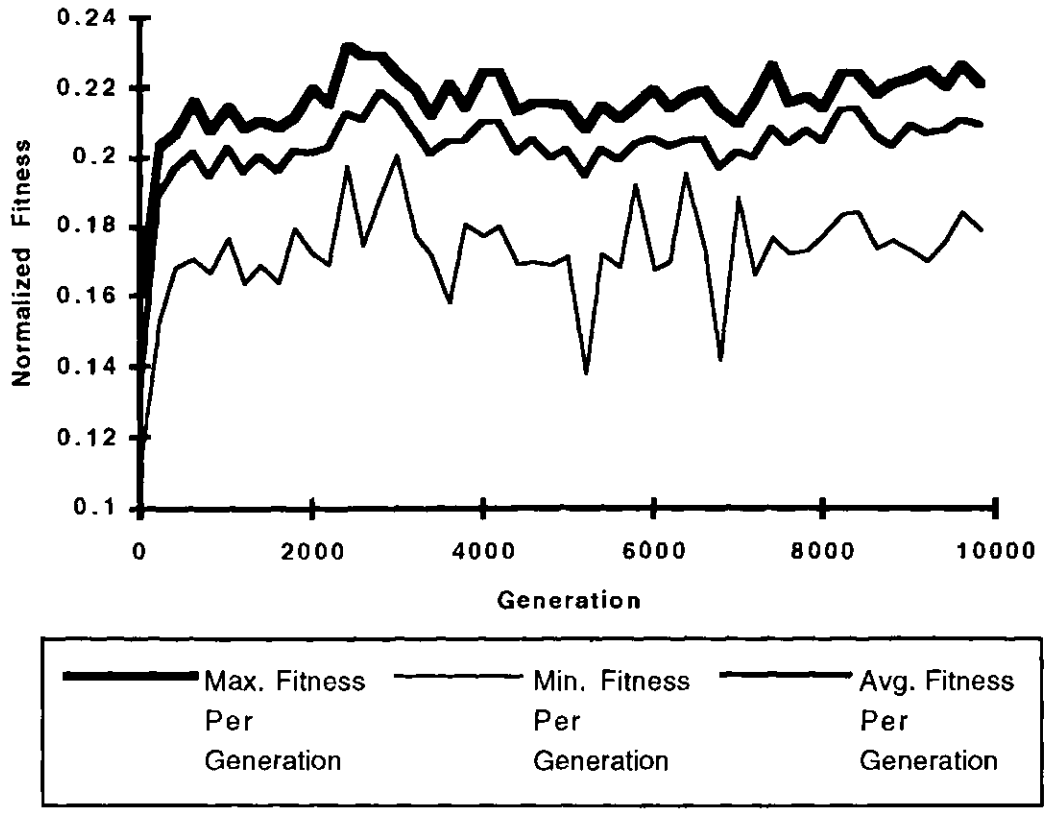


Figure 3.7 Graph of Best, Worst and Average Chromosome Fitness in Each Generation Population Size=100. Parameters of Pop=100, Gen=10,000, P_X=0.2, P_m=0.0001, Time=50,306 seconds.

Before the GA's results are further analyzed, the basis for the DSS's success should be analyzed. The basis for this analysis comes from a description of the DSS in Ferland and Fleurent (1991). The DSS success can be largely attributed to its ability to enhance the expert scheduler's ability to generate a schedule. In a strictly book keeping role, the DSS allows the expert to build a schedule as he always has, but in the process of doing so, a variety of statistics are tracked allowing the expert to easily manage the overwhelming amount of data available. The DSS also includes some of the heuristic knowledge of the scheduler, allowing the software to generate parts of the

schedule under the expert's supervision. The expert can then take over control under conditions where experience or sublime heuristics can outperform the computational brute force of the DSS.

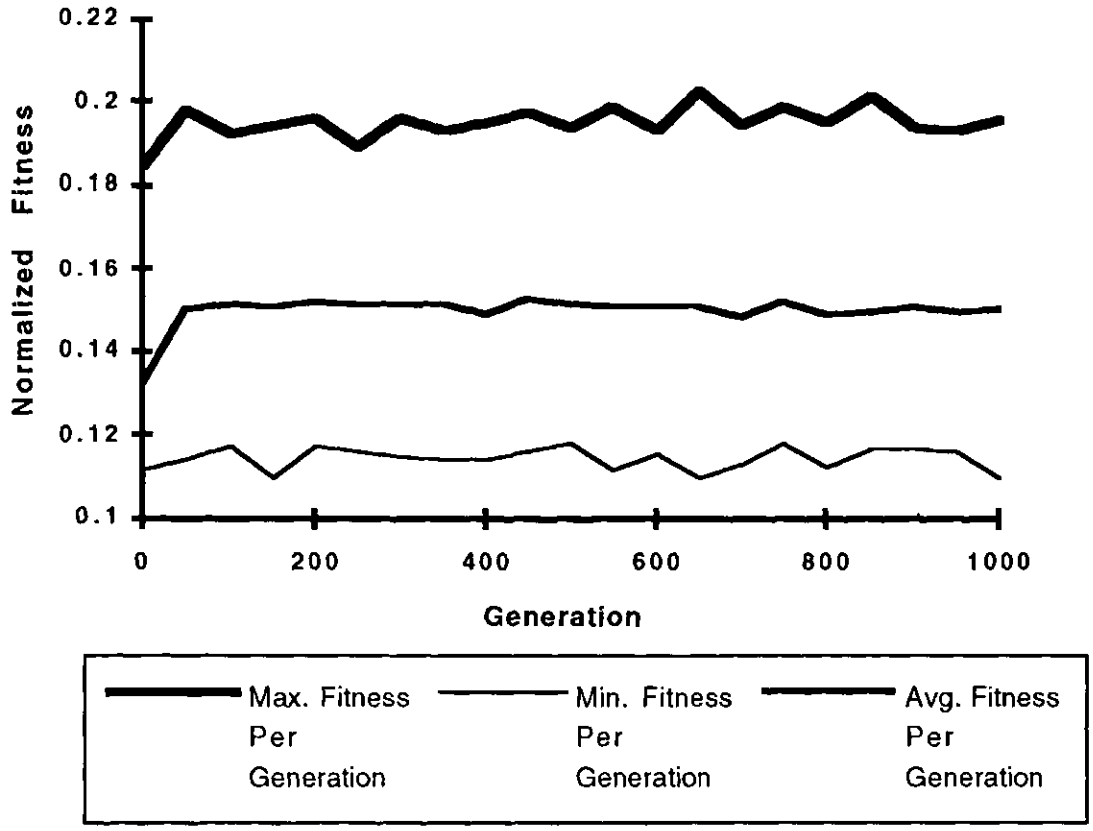


Figure 3.8 Graph of Best, Worst and Average Chromosome Fitness in Each Generation Population Size = 500. Parameters of Pop=500, Gen=1000, P_X=0.8, P_m=0.01, Time=124,074 seconds.

Why does this work better? Using optimization and GA terminology, the NHL schedule understands the general topology of the search space (or part of the previous schedule). The initial starting point is usually the previous schedule. The scheduling expert, because of years of experience, is able to identify and exploit certain scheduling patterns. With the DSS, the expert can experiment with possible schedules, and with the variety of

information and statistics available he can immediately generate the information necessary to decide if this is a direction that he wishes to pursue in terms of generating a schedule. In essence, the scheduler is behaving in some respects like a GA, plus he has the additional ability of being able to exploit local knowledge about the problem.

Table 3.6 Comparison of GA Schedule Results to the Actual NHL Schedule

	NHL	GA
Distance (miles)	982,831	1,308,933
Constraint	8.720	7.331
Total Fitness	0.4120	0.2400

The GA's was successful in evolving continuously improved solutions. Unfortunately it was unable to find a solution which improved upon the NHL's actual schedule. The primary factors for the GA not finding an improved solution were the population size, chromosome structure/reproduction method, and problem difficulty.

Based on the author's previous research (Kostuk 1993) if the population size is increased by an order of magnitude and the results do not significantly improve, then it is very likely that even the new population size is still too small. Hardware limitations prevented the population from being set any higher than 2000 chromosomes. Any population size greater than 2000 would take a significant amount of computational effort. If this approach was taken, then it would be recommended to subdivide the population so that it can to be analyzed in parallel by several processors.

The work done in Chapter 2, and in the application section of GA texts

(Goldberg 1989, Michalewicz 1992a), indicate that in cases where the GA underperforms, such as the case here, a change in the chromosome structure or the reproductive strategy may improve the GA's performance. A perfect example is the application of the GA to the Traveling Salesman Problem (TSP). The first researchers who tried to solve the TSP using the GA had only moderate success because the reproductive strategies were no more than variations of traditional crossover. The edge recombination crossover method used in Chapter 2 for the TSP example, was successful for a variety of reasons. The most important reason being that all of the offspring created with this method are legal and complete tours. If a similar method (in terms of reproducing chromosomes which do not require repair) could be developed for the NHL problem, significant inroads could be made.

There is always the possibility that the NHL problem is too difficult to be solved efficiently with a GA. The problem space may be such that it is deceptive to the GA. The GA 'thinks' a certain region will pay-off, but in fact it is one of the worst solutions. Solution spaces for real world problems which display this characteristic have not been documented. Quite possibly, because they would represent failed research. Solution spaces which do display this characteristic have been developed specifically to test the prowess of the GA (Goldberg, 1989).

The GA has been known to quickly reach the neighbourhood of a solution, but yet still fail to find the solution. Maybe this is such a problem and it requires a hybrid solution. An example of such a problem is the TSP with time windows. The traveling salesman must visit all of his clients, but in doing so must still travel the shortest possible distance. The difference

between this model, and the classic TSP is that the clients are only available for certain periods of time. An application of the time windows model is in designing routes for school buses (Thangiah, et al, 1991). The GA's role in this application was to rapidly find an approximate solution and then another algorithm was applied to find the optimal routing. This method has also been suggested by Costa (1992) specifically for the NHL problem.

3.4 Conclusions

The NHL problem is solvable by the GA, but to improve upon the NHL and its DSS created schedule one of the following must occur:

- computer architecture must improve so that increased population sizes can be handled,
- a new reproductive strategy or chromosome structure must be developed so illegal chromosomes are not reproduced,
- a hybrid solution method should be investigated.

4. THE GENETIC ALGORITHM AND LARGE SCALE PROBLEMS

This chapter's function is to serve as a forum to discuss the application of the GA to large scale problems. Large scale may be a misnomer, a better term may be real-world problems. Many real-world problems are so significant in size or are so complex that they cannot be adequately modeled or represented by OR models. The source of the complexity is often that these problems are a combination of several models. Unfortunately you can not solve a problem which is a combination of OR problems by combining OR solution techniques.

A discussion on the application of the GA to large scale problems, largely based on the research involved with solving the NHL problem, will act as the basis for this chapter. The chapter is divided into two sections. The first section deals with factors that constrain the success of the GA application. The second section covers suggested strategies for applying the GA.

4.1 Limiting Factors

Factors which could limit the success of the GA include the size of the solution space, the complexity of the solution space, hardware/software limitations, inadequate chromosome representation and reproduction methods, and time constraints.

Solution space size and solution space complexity are closely related. Difficulties arising from the solution space's size will be dealt with first. The more exact a model becomes, the larger the solution space grows. A good example of this is the first problem solved in Chapter 2. By increasing the resolution of the solution space to the hundredths, the solution space was expanded to 400x800=320,000 points from the necessary 4x8=32. For most real

world problems it is difficult to identify the extraneous information. The key is to find the level of granularity that will still allow the model to generate a valid solution, but to do so in a reasonable amount of time. When reducing the model's resolution, there is a possibility that the optimal solution will be missed. If the optimal solution is a spike the GA may explore the surrounding region, but miss the optimum because of the lack of resolution. Solution spaces which can not be reduced usually require long chromosomes to adequately search the space. Large chromosomes generate problems in terms of hardware limitations (usually storage issues).

Solution space complexity is a significant problem. Complexity is a problem if the space is noisy (containing numerous spikes of information), or if it is deceptive. If the space is noisy then it is difficult for the GA to evolve in the direction of the optimum. The GA will spend much of its time exploring various regions of the space but it will not be able to select specific regions as being significantly better. This in turn means that the GA will not be able to focus its resources on any specific region. The GA will flounder because there will be no, or little, improvement from generation to generation. In a deceptive solution space the GA is fooled into exploring sub-optimal regions. It puts more and more resources into exploring regions where an optimum will not be found.

The standard limitation for computer applications is processor speed and available memory. Aside from the blatant limiting effect of processor speed, the most restrictive hardware element is memory. If the memory is not physically available, or the compiler can not take advantage of the available memory, the GA is restricted in its ability to search the solution space. Large populations of long chromosomes require significant amounts of memory. The only option

available is to select the chromosome structure and the alphabet size to reduce the required memory. Unfortunately, chromosome length reduction does not necessarily translate to efficient problem solution. Furthermore, the memory efficient structure could be anything from a string to an n-dimensional entity. The more complex the structure, the more thought required to devise an appropriate reproductive strategy. An excellent example of the effects a reproductive strategy can make can be found in Chapter 2. The Job Scheduling Problem (JSP) and the Traveling Salesman Problem (TSP) were shown to be similar. Because they were similar each had the same chromosome structure and alphabet. But, their reproduction strategies were significantly different and so were their success rates.

No matter how well the GA is able to represent and solve a problem, if it can not be done within a reasonable time frame then it is not a practical solution. The only documented applications of the GA are in deterministic environments. When a GA is used in repeated situations where the input parameters are varied, but the standard model remains the same, the time window the GA requires to solve the problem is well known. This eliminates most of the uncertainty associated with the time the GA requires to solve the new problem, and allows the GA to become an effective business tool.

The GA research community has speculated on the application of a GA in a real-time system, but documented cases are not available. Imagine the GA as a dynamic system with a non-continuous forcing function (impulses). For a real-time system, the GA would run continuously. A person would update the input parameters (these would act as an impulse to the system) and the GA would evolve from its present steady state optimum, to a new optimal steady state

solution. This model assumes that the old optimum is already near the new optimum, expediting the evolution to the new solution. This concept is only valid if the new solution is always expected to be in the neighbourhood of the previous solution. In reality, starting from an old optimum may trap the GA, by not allowing it to search the solution space. If the solution is not in the neighbouring region there is no advantage for the GA to start from the previous solution since any starting point is as valid as any other.

4.2 Suggested Approaches

There are three basic approaches which can be followed to improve the probability of success when applying the GA to a real-world problem: small-scale prototypes, constraint relaxation, and the implementation of parallel processing. Each of these approaches has their own benefits and drawbacks. But, depending on the problem type, one may be more suitable than the others.

Small scale prototypes are the traditional engineering approach to solving large projects. Tackling a smaller project makes the task more manageable, yet it will still provide the necessary understanding of the final, full size, task. The only problem with this approach is that scaling up the solution from the small-scale problem may not work. The scaling up process may change the problem so radically from what was originally modeled that the GA's performance may not keep up. This will be particularly true for combinatorial problems as shown by the author's previous research (Kostuk 1993). This is a common problem in the Expert System field (Fox 1990).

Another methodology is to develop a model which is full scale, but with relaxed constraints. This prototype may take slightly longer to develop than the

previous method, but once completed it is only slightly different than the final product. More time is required but if successful, the likelihood of succeeding with the final project is significantly higher than if a small-scale prototype had been built. Again, the objective of this model is to simplify the model so the designer has the opportunity to become more familiar with the basic process. Once the familiarization period is complete the final constraints can be added.

The last approach is not so much a development strategy but an implementation approach. Parallelization is the subdivision of a process into a series of tasks which can be performed simultaneously, reducing the time required to complete the task. Parallelization makes the most use out of existing computational processor power. The GA is very amenable to parallelization, so the process should be given serious consideration whenever possible.

Parallelization of the GA can be implemented a variety of ways. One method which was used extensively when researching Chapters 2 and 3, and is the simplest implementation of parallelization, is to run the GA on several computers, each with different parameter sets. A complete suite of results are generated without any special programming requirements. The second method of implementing parallelization is to split the population amongst a variety of processors. For each generation, the processors calculate the fitness of a sub-population. The population and the appropriate statistics are centralized, reproduction occurs, and the next generation is reassigned to the processors. The third method of parallelization is applicable if more than one function is used to assess fitness. For example, in the NHL problem, each chromosome was ranked on distance and eleven other constraints. If these twelve constraints were each assigned a processor, the total fitness of a chromosome could be calculated

simultaneously. The last method of implementing parallelization is to subdivide the population and assign each subdivision to a processor (as mentioned in method two). Each processor evolves a population a predetermined number of generations, and then a prespecified amount of emigration and immigration occurs between the populations. The primary benefit of parallelization is a decrease in solution time, as well as a reduction in memory requirements. In fact, recent research has documented better than linear speed-up due to the implementation of genetic programming parallelization (Koza and Andre 1995). The primary drawback of the last two methods of parallelization is that it may be difficult to implement on anything other than a parallel computer.

5. SUMMARY AND CONCLUSIONS

5.1 Summary

In general, it can be stated that the GA is a flexible solution technique. The GA was shown to solve a variety of problem types, with little or no modification. Although clearly a robust technique, it was also identified that the GA's performance could be significantly improved through the careful selection of various GA parameters: alphabet size, chromosome structure and reproductive strategy.

The application of the GA to a variety of OR problems provided significant insight on how to approach the NHL scheduling problem. In particular, the experiments suggested a suitable chromosome structure (a matrix), a reproductive strategy (crossover), as well as how to use the fitness function to restrict the propagation of poor schedules. The GA's success in solving the NHL scheduling problem was however limited. The GA was able to successfully evolve better and better schedules over time, but none of the schedules were found to improve upon the existing schedule presently in use by the NHL.

5.2 Conclusions

The objective of this thesis was to explore the GA's ability to solve a complex, real-world scheduling problem. The GA was unable to improve upon the existing NHL schedule. There are three potential reasons for this lack of success.

The first reason brought forth was a lack of computing power. The equipment used to explore the GA's ability was unable to store population sizes over 2000 chromosomes in memory. Because of the long length of the chromosomes, significantly sized populations were required. Given the chromosome length used for this problem, chromosome populations of at least one order of magnitude larger than was possible to store in memory was necessary to solve the NHL problem as it is presently formulated.

The second possible reason for failure was identified as a modeling problem. From the experiments with the OR test suite in Chapter 2, it was shown that chromosome structure and reproductive methods have a significant effect on the success of the GA. In particular, the effort spent by the GA repairing, or replacing poor chromosomes are quite computationally expensive. If a method could have been developed which eliminated the production of illegal chromosomes, more effort could have been focused on exploring the valid solution space, instead of replacing chromosomes which had left the valid solution space.

The third suggested source of failure was that the solution space was too noisy, or possibly it was deceptive. A noisy solution space would not allow the GA to focus on a particular region of the solution space. All regions would appear equally viable. In the case that the solution space was deceptive, regions which would appear to be conducive to an optimal solution would actually contain average or even poor solutions. In either case, the GA would hone in on a region, and then aimlessly flounder. If this condition exists, it would suggest that a hybrid technique should be implemented. The GA would be used to do an initial search, and the regions

identified as possible solutions would then be evaluated using another optimization technique.

5.3 Further Investigation

It was stated in Chapter 3 that the GA solution to the NHL scheduling problem may be improved if a different chromosome structure or reproduction method could be developed. The following are two suggestions for potential research for the NHL and similar large-scale scheduling problems.

One of the reasons that a binary model was not considered was because of the extreme memory requirements to store such a model. A possible compromise would be to use the present chromosome structure to store each chromosome, and then convert it to a binary representation for reproductive purposes. The most natural chromosome structure to use with a binary representation would be a three dimensional structure. The X and Y-axis would represent the visitor and home teams. The Z-axis would represent the date. Thus the prism would have dimensions of 24x24x180=103,680 (as compared to the 24x180=4,320 character representation, which was actually used). A 1 in the element of the prism would represent a game. Thus, if {3,2,20}=1, Team 3 visits Team 2 on Day 20 of the schedule. Of course new three dimensional reproductive operators would have to be developed. But, since this method would be very similar to work that has been done in graph theory (Grover, et al 1990), there may be a way to adapt the operators used in the GA solution of the transportation problem, which also adopted methods from this field.

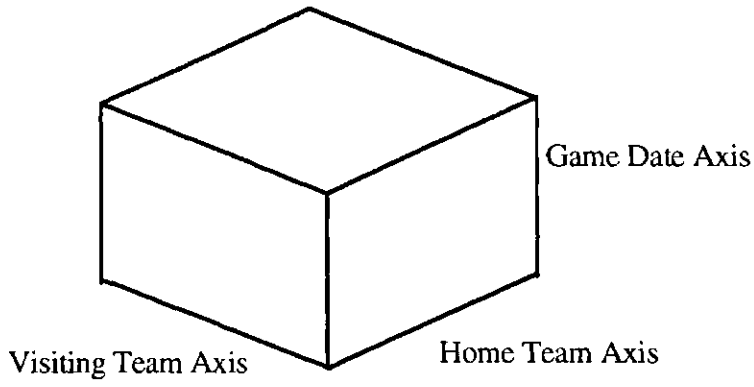


Figure 5.1 Schematic Representation of a Three Dimensional Chromosome Structure

In closing this thesis has demonstrated that the GA is a robust tool in the sense that it can be used to solve a variety of problems with minimal modifications. But, it was also shown that the GA's performance can be significantly improved when knowledge about the problem structure was exploited. Thus, the GA provides the Civil Engineer with another tool to attack scheduling problems, especially those that have proven unwieldy to existing or standard techniques.

REFERENCES

Baffes, P., and Wang, L. 1988. Mobile transporter path planning using a genetic algorithm approach. Space Station Automation IV, 1006, pp. 226-234.

Beasley, J.E., and Chu, P.C. 1994. A genetic algorithm for the set covering problem. Available from j.beasley@ic.ac.uk or p.chu@ic.ac.uk.

Cain, W.O., Jr. 1972. Bayesian discrete optimizing as an approach to a scheduling problem: major league baseball. Ph.D. dissertation, Harvard University.

Chu, P.C., and Beasley, J.E., 1995. A genetic algorithm for the set partitioning problem. Available from http://mscmga.ms.ic.ac.uk/pchu/pchu.html or http://mscmga.ms.ic.ac.uk/jeb/jeb.html

Colorni, A., Dorigo, M., and Maniezzo, V.M. 1990. Genetic algorithms and highly constrained problems: the time-table case. Proceedings of the first international conference on parallel problem solving from nature. *Edited by* Schewefel, H.-P., and Manner, R. Dortmund, Germany. pp. 55-59.

Costa, D. 1992 An evolutionary tabu search algorithm and the NHL scheduling problem. Internal working paper for the Department of Mathematics, National Polytechnical School of Lausanne.

Deb, K. 1991. Optimal design of a welded beam structure via genetic algorithms. AIAA Journal, 29(11), pp. 2013-2015.

Ferland, J.A., and Fleurent, C. 1991. Computer aided scheduling for a sport league. Infor. Vol. 29, No. 1:14-25.

Fox, B.R., and McMahon, M.B. 1991. Genetic operators for sequencing problems. Foundations of Genetic Algorithms, edited by Rawlins, G. Morgan Kaufmann Publishers, Los Altos, CA. pp. 284-300.

Fox, Mark S. 1990. AI and expert system myths, legends, and facts. IEEE Expert. Feb. 1990. pp. 8-19.

Gabbert, P.S., Brown, D.E., Huntley, C.L., Markowicz, B.P., and Sappington, D.E. 1991. A system for learning routes and schedules with genetic algorithms. Proceedings of the Fourth Internation Conference on Genetic Algorithms. pp. 430-436.

Goldberg, D.E. 1983. Computer-aided gas pipeline operation using genetic algorithms and rule learning. Ph.D. dissertation. University of Michigan.

Goldberg, D.E. 1989. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley.

Grover, L.J., Michalewicz, Z., Elia, P.V., and Janikow, C.Z. 1990. Genetic algorithms for drawing directed graphs. Methodologies for Intelligent Systems. Vol. 5, pp. 268-276.

Holland, J.H. 1992a. Adaptation in natural and artificial systems. Revised Second Edition. Cambridge, MA. MIT Press.

Holland, J.H. 1992b. Genetic algorithms. Scientific American. Volume 267(1). pp. 66-73.

Irgon, A., Zolnowski, J., Murray, K., and Gersho, M. 1990. Expert system development: a retrospective view of five systems. IEEE Expert. June 1990. pp. 25-39.

Jenkins, W.M. 1991. Structural optimisation with the genetic algorithm. The Structural Engineer, 69(24):418-422.

Johnston, P. 1991. Skating the red line. Canadian Business, May 1991, Toronto, ON. CB Media Limited. pp. 44-47.

Jones, R.D. 1989. Development of an automated airline crew bid generation system. Interfaces, Vol. 19, No. 4:45-51.

Kostuk, K. 1993. Solving the boolean 6-multiplexer problem with genetic algorithm co-evolution. Genetic Algorithms at Stanford. Stanford, CA. Stanford Bookstore. pp. 117-123.

Koza, J.R., and Andre, D. 1995. Parallel genetic programming on a network of transputers. Stanford University Internal Report Number STAN-CS-TR-95-1542. Available from elib.stanford.edu /pub/reports/cs/tr/95/1542.

Koza, J.R. 1993. Genetic algorithms class notes, Stanford University, unpublished.

Koza, J.R. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA. MIT Press.

Larson, R.C., and Odoni, A.R. 1981. Urban Operations Research. Englewood Cliffs, NJ. Prentice-Hall, Inc.

Michalewicz, Z. 1992. Genetic Algorithms + Data Structures = Evolution Programs. New York, NY. Springer-Verlag.

Michalewicz, Z. 1992. Genetic2 computer software code. Available from ftp.aic.nrl.nay.mil /pub/galist/sources-code/ga-source.

Powell, W.B., Sheffi, Y., Nickerson, K.S., Butterbaugh, K., and Atherton, S.A. 1988. Maximizing profits for north american van lines' truckload division: a new framework for pricing and operations. Interfaces, Vol. 18, No. 1 Jan.- Feb. 1988:21-41.

Singer, D., and Moritz., P. 1987. Expert system for scheduling cleaning operations for municipal canal systems. International Journal of Systems Science, Vol. 18, No. 7:1217-1226.

Thangiah, S.R., and Nygard, K. 1992. School bus routing using genetic algorithms. Applications of Artificial Intelligence X: Knowledge-Based Systems. Vol. 1707:387-398.

Thangiah, S.R., Nygard, K., and Juell, P.L. 1991. Gideon: a genetic algorithm system for vehicle routing with time windows. Proceedings of the Seventh IEEE Conference on Artificial Intelligence Application. Miami, FL. pp. 322-328.

Waters, C.D.J. 1990. Expert systems for vehicle scheduling. Journal of the Operational Research Society. Vol. 41, No. 6.:505-515.

APPENDIX A

Background on the Genetic Algorithm

With an Example

A.1 An Example Application

The best way to illustrate how the Genetic Algorithm (GA) works is to step through a simple example. A problem will be selected, formulated, and developed. Basic GA terminology and techniques will be explained as they are brought forth.

A simple, yet illustrative, problem is to find the maximum value of a quadratic function. The objective of this exercise is not to challenge the GA, but to illustrate how it works. The function selected is $-x^2 + 360x$. From basic calculus the maximum is easily found to be 180.

A.2 The Four Major Preparatory Steps

With the application selected, the GA can now be designed. There are four major preparatory steps that must be completed when designing a GA. The first step is to select a representation scheme. The representation scheme is also known as the **chromosome** structure. In the GA the chromosome represents either a potential solution, or the inputs to a model or simulation which is external to the GA. Traditionally chromosomes represent a solution to the problem in question. In the emerging area of Genetic Programming external simulations are applied more frequently (Koza 1992). A chromosome is made up of **genes**, the number of genes in the chromosome determines its length. For example the chromosome in Figure A.1 is made up of eight genes, thus its length is eight.

0 1 1 0 0 0 0 1

Figure A.1 An Example of a Chromosome Structure

The number of values which are possible in each gene position define the alphabet size. These values or **alleles** can come from an alphabet size as small as two characters to one of several hundred characters (an impractical size, but nonetheless still possible). The most fundamental alphabet size is two (a binary alphabet). Binary alphabets are the most efficient representation for searching a solution space, but, this representation is not always the most convenient for real-world problems. For this simplistic problem the binary scheme will be effective. The chromosome represents a binary number of length eight. Thus, the example chromosome can be interpreted as 97.[1]

The GA utilizes a **population** of chromosomes in its search for the best or optimal solution. Experience is the only way to estimate the optimal population size prior to running the algorithm. The only consistent rule of thumb is that the larger the population size the less time that will be necessary to find a solution to the problem. Post-processing methods exist to determine an optimal population size. But, the number of runs required to make a reasonable estimate make this prohibitive for anything but trivial problems, or problems which must be solved repeatedly.

The second preparatory step is to identify a fitness measure. Chromosome fitness determines which members of the population are better

[1] The day-to-day number system is the decimal or base 10 system. Each place holder can be any one of ten numbers from 0 to 9. In a binary system there are only two numbers, 0 and 1. Thus, 10 which is ten ($1x10^1 + 0x10^0$) in the decimal system, is two in the binary or base 2 system ($1x2^1 + 0x2^0$). For the example chromosome, its binary representation of 01100001 can be interpreted as ($0x2^7 + 1x2^6 + 1x2^5 + 0x2^4 + 0x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 64 + 31 + 1 = 97$).

solutions (and by what amount). For the example chromosome its fitness is $f(x) = -(97)^2 + (360 \times 97) = 25{,}511$.[2]

The third step is to select population parameters to control the GA. The parameters of note (which will be defined later) are: population size, maximum number of generations, probability of crossover (crossover rate), and probability of mutation (mutation rate).

The last step is to determine the method for terminating a run and how to designate the results. Genetic Algorithm termination can occur when an optimal solution is found, a prespecified performance level is attained, or after the maximum **generations** (time steps) have been completed. The method for designating the results could be either the best individual of the last generation or the best individual of the run (the best of the best). Since one rarely knows the solution in advance, most GA applications terminate after the completion of a prespecified number of generations. The solution is the best (most fit) individual of the run. This will be the method used for all examples in this thesis.

A.3 Applying the Genetic Algorithm

Now that an understanding of the basic elements of the GA exists, it is time to step back and take a look at the general methodology of the GA. The GA can be broken down into four steps:

- Generate an initial population of solutions to the problem on hand.
- Measure the fitness of each solution (and also determine if there is a solution in the population that meets the desired criteria).

[2] The fitness is calculated by substituting the decimal interpretation of the chromosome into the fitness function. Since the decimal value of the chromosome is 97, the fitness is 25,511.

- Reproduce the population in a manner such that the best chromosomes are present in the next generation proportional to their fitness relative to the population's average fitness. (The more fit the chromosome, the more offspring it will have.)

- Continue the process until a desired or satisfactory solution is obtained.

As noted above, the first step is to generate the initial population. The initial population is generated probabilisticly. The genes are randomly set at 0 or 1. Thus, generating an initial population of eight chromosomes of length eight is comparable to flipping a coin sixty-four times. Table A.1 contains information on the initial population of chromosomes.

Table A.1 An Initial Population of Chromosomes

#	Chromosome	x	fitness	Relative Fitness	Proportional Weighting
1	10011101	157	31871	1.161	0.145
2	01011100	92	24656	0.898	0.112
3	11110011	243	28431	1.036	0.129
4	01001011	75	21375	0.779	0.097
5	11101111	239	28919	1.053	0.132
6	11110001	241	28679	1.045	0.131
7	01100000	96	25344	0.923	0.115
8	10000111	135	30375	1.106	0.138
		Σ =	219650		
		Avg. =	27456		

The format used for Table A.1 follows a standard layout established by Goldberg's seminal text on GAs (Goldberg 1989). The headings appear slightly cryptic at first, so some explanations should be made now so the reader can focus on the table's content and not on its form.

Table A.2 A Key to the Standard GA Chromosome Fitness Table

Table Heading	Explanation of Heading
#	Each chromosome in the population is labeled to track its contribution to the next generation.
Chromosome	The chromosome itself.
x	The decimal equivalent of the binary representation of the chromosome.
Fitness	A measure of the chromosome's strength based on some predetermined fitness function. For this example the fitness function is $-x^2 + 360x$.
Relative Fitness	This value is calculated once the fitness of all the chromosomes is established. Relative fitness is the ratio of the chromosome's fitness to the population's average fitness.
Proportional Weighting	A ratio of the chromosome's fitness to the sum of the population's fitness. The sum of the proportional weighting is 1.0. If regions on a roulette wheel were set proportional to the relative fitness of the chromosomes (Figure A.2) then the proportion of the wheel allocated to each would be equal to the values in this column.

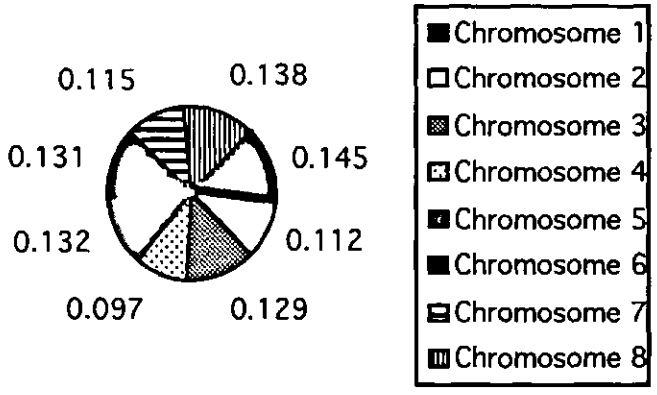


Figure A.2 Roulette Wheel with Proportional Weighting

Superimposing the location of the chromosomes from Table A.1 on a graph of the function gives the following:

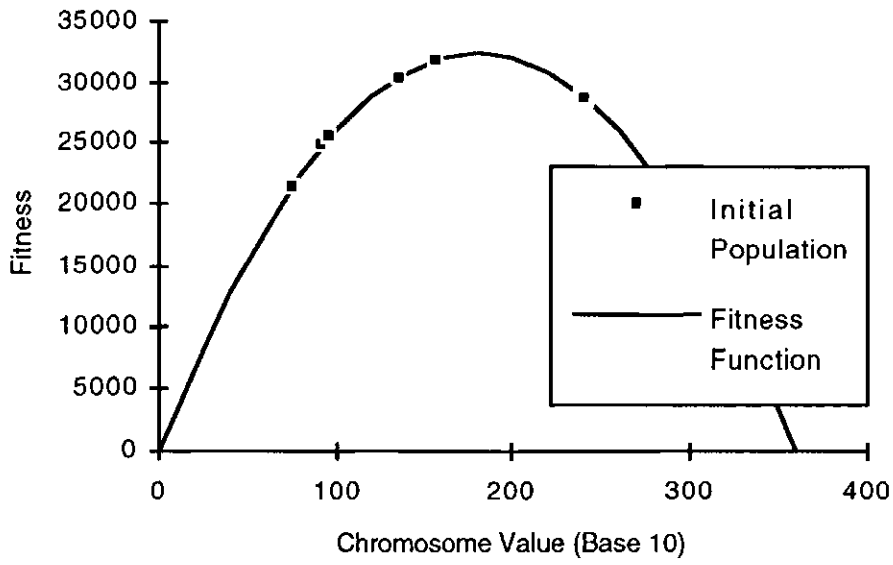


Figure A.3 Fitness of the Initial Population's Chromosomes

An interesting analogy is to imagine trying to find the highest elevation of a hill by dropping paratroopers onto the terrain. Once the original paratroopers are dropped (the initial population of eight chromosomes) they check their elevation above some datum. The elevation of each paratrooper is equivalent to each chromosome's fitness. Each paratrooper radios back his elevation to the pilot, who will drop more paratroopers in the areas where the paratroopers are higher than the average paratrooper elevation. This is continued until the peak is found (Koza 1993). In the GA, after the fitness of each chromosome is found, its relative fitness is calculated. Using a Monte-Carlo simulation, each chromosome is given a weighting on a roulette wheel proportional to its relative fitness. The wheel is spun enough times to create a new population (in this example it was eight times). Probabilisticly, the chromosomes which are above average fitness will be proportionately over-represented in the mating pool for the next generation, where those that are below average will proportionately lose their representation. In this example the initial population does not have any

individuals who are particularly stronger or weaker than its counterparts, consequently they all have relatively the same likelihood of being selected to reproduce.

Reproduction by proportional representation alone does not introduce an improved solution. If reproduction was the only operator used to improve the population, the only improvement would be an increase in the average fitness. Eventually the average fitness would equal the fitness of the best individual in the initial population. No more improvement would occur and no new regions of the search space would ever be explored. Two operators are used to introduce new solutions to the population: **crossover** and **mutation**.

Reproduction with crossover introduces changes to the population more quickly than mutation. Crossover rates are much higher than mutation rates but this is not the basis for crossover's greater influence. Crossover combines two solutions to explore new regions within the solution space (Holland 1992a). If the parent solutions are similar they will produce offspring which are located nearby. If the parents are significantly different, then the offspring will be located in a new and possibly unexplored region of the solution space. The survival of specific chromosomes is not as important as the survival of groups of chromosomes which belong to certain regions of the search space (specifically the region where the optimum is located). As long as the region containing the optimal solution is explored by chromosomes, then a solution will be found.

There are various forms of crossover. The example problem will examine simple crossover. In simple crossover, chromosomes are randomly selected two at a time from the mating pool (which was created with the roulette wheel

selection). Chromosomes four and five from the initial population will be used to illustrate simple crossover. These chromosomes will act as the parents for two members of the next generation.

Parent 1 (Chromosome 4)	1	1	1	0	1	1	1	1
Parent 2 (Chromosome 5)	1	1	1	1	0	0	0	1
Child 1	1	1	1	0	1	0	0	1
Child 2	1	1	1	1	0	1	1	1

Figure A.4 Reproduction of Two Chromosomes

Once two chromosomes are selected the crossover point needs to be determined. There are eight crossover sites for a chromosome of length eight. Most GAs will randomly select a point between one and the chromosome's length (inclusive). If the crossover point is less than the chromosome's length genetic information is mixed, creating new offspring. If the crossover point is equal to the chromosome's length, each parent is passed to the next generation as offspring (no mixing of genetic material occurs). Biologically this is equivalent to cloning.

Figure A.4 illustrates the basic process of simple crossover. Parent 1 and Parent 2 pass their genes up to the crossover point (denoted by the double bar) directly to Child 1 and Child 2 respectively. From the crossover point to the end of the chromosome, Parent 1 transfers its genes to Child 2 and Parent 2 to Child 1. Table A.3 contains the second generation of chromosomes.

Table A.3 The Second Generation of Chromosomes

#	Parents	Xsite	Chromosome	x	Fitness	Relative Fitness	Proportional Weighting
1	(4,5)	5	01101111	111	27639	0.923	0.115
2	(4,5)	5	11110001	241	28679	0.958	0.120
3	(7,7)	2	10000111	135	30375	1.014	0.127
4	(7,7)	2	10000111	135	30375	1.014	0.127
5	(4,7)	6	10101111	175	32375	1.081	0.135
6	(4,7)	6	11010101	213	31311	1.046	0.131
7	(2,7)	2	10000111	135	30375	1.014	0.127
8	(2,7)	2	11110011	243	28431	0.949	0.119
				Σ =	239560		
				Avg. =	**29945**		

Two things should be noted about Table A.3. First, two columns have been added to the standard table layout: Parents, and Xsite. The first column, Parents, shows which two chromosomes were mated to generate the present chromosome. The Xsite column shows the crossover site used in the reproduction process. The second point to note is that Chromosomes 1 and 2 have the same parents as the example in Figure A.4. But, Chromosomes 1 and 2 are not the same as the two offspring in the example.

Table A.4 Difference in Offspring Due to Mutation

	Child 1	Child 2
Example	11101001	11110111
Table 1.2	01101111	11110001

The earlier example did not include mutation, in contrast, the GA utilizes both reproduction and mutation. Mutation also introduces new genetic information to the population. The mutation rate or probability of mutation is usually set equal to the inverse of the chromosome length. For this example, the mutation rate would be one in eight (or 0.125). This means that for every eight genes transferred, approximately one mutation will occur. Mutation is easy to implement when a binary alphabet is in use. Once it is determined that mutation

should occur, it is simply a matter of changing a zero to a one, or a one to a zero (also known as a bit flip). As in nature mutation can be recessive or progressive. When searching for a specific value (as in the example) high mutation rates (high being defined as greater than the inverse of the population size) are usually recessive. What happens is that as the population begins to converge (fitness is generally high), the mutation operator is more likely to decrease the fitness than to increase it. Conversely, when a chromosome is weak, it is more likely that mutation will increase the fitness of a chromosome. So although often beneficial in the early stages of evolution, high mutation rates will eventually counteract the benefits of crossover. Some GA researchers have stated that mutation is not necessary at all (Koza 1993). The author's personal experience with Genetic Programming (a sub-field of GA) has shown this to be true, but not for GAs. In fact, personal observations have shown that without mutation, the GA tends to prematurely converge because of a lack of genetic diversity.

Figures A.5 and A.6 illustrate the progressive improvement of the GA. Figure A.5 is a graph of the data from Table A.3. Note how the chromosomes in Figure A.4 have become more clumped as compared to Figure A.3. Notice also how they have "moved up the hill." Figure A.6 shows an almost perfect solution (from the fifth generation), where the most fit chromosome (with a value of 181) has a fitness of 32,399. The exact solution (180) was found by the sixteenth generation. This behaviour is typical of a GA. Convergence to an approximate solution was relatively rapid and was then followed by slow incremental improvements to the exact solution.

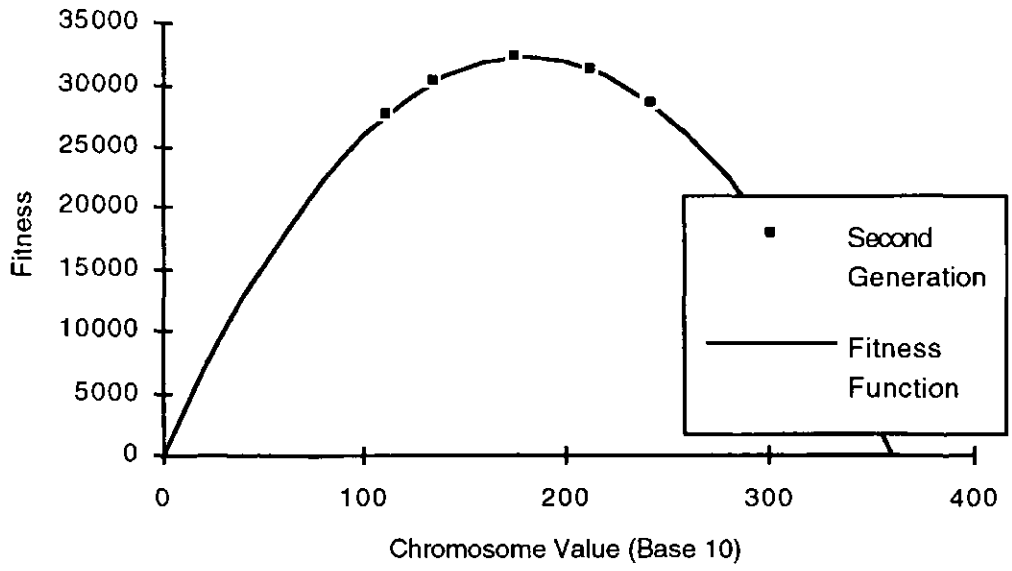


Figure A.5 Fitness of the Second Generation of Chromosomes

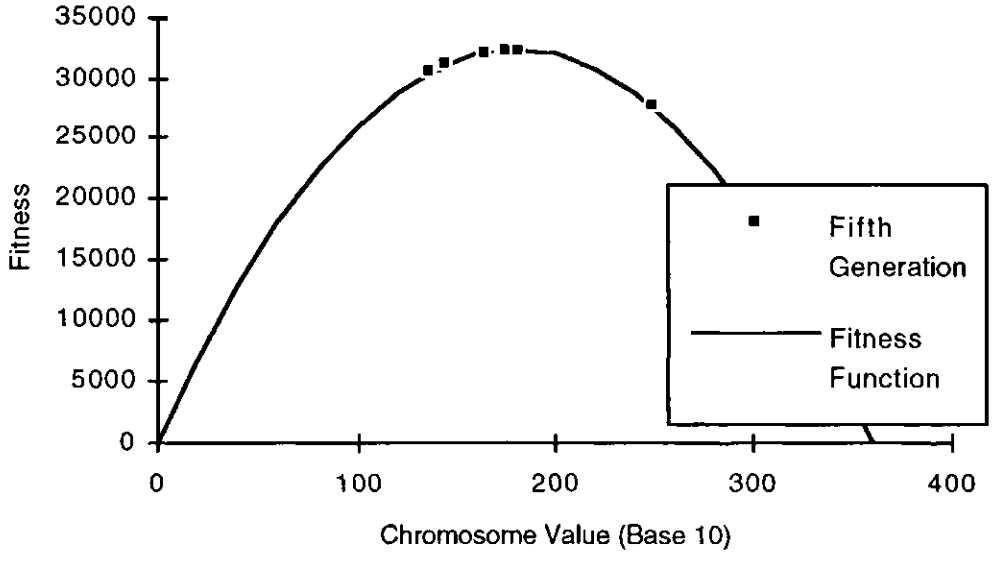


Figure A.6 A Near-Optimal Solution is Almost Found

APPENDIX B

Sample Program Code for the Genetic Algorithm

The following figures make up the basic code used in this reasearch. The code is based on Goldberg's Simple Genetic Algorithm (Goldberg 1989) wihch was originally translated to a dialect of C by Adam Conru while he was a graduate student at Stanford. That C code was further refined by the author so that it would compile using the Think C compiler on an Macintosh computer.

It should be noted that the code for this program was written to optimize a fitness function of x^{10} and the chromosome structure is a 30 bit binary string.

```
/* ************************************************  */
/*      Based on Goldberg's Simple Genetic Algorithm  */
/*      Originally converted to C by Adam Conru       */
/*      Modified for Think C by Kent J. Kostuk        */
/* ************************************************  */

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <errno.h>

/* ************************************************  */
/*      define the global variables and structure types */
/* ************************************************  */

int maxgen,gen;

#define maxstring 30
#define maxpop 100


typedef struct
    {int chrom[maxstring];      /*   Genotype = bit string          */
     float x;                   /*   Phenotype = unsigned integer   */
     float fitness;             /*   Objective function value       */
     int parent1, parent2;
     int xsite;                 /*   cross pt                       */
     }individual;

typedef struct
```

```
    {individual people[maxpop];
     float avg, max, min;
     int nmutation, ncross;
     int popsize, lchrom, gen;
     float pcross, pmutation,sumfitness;
     }population;

population oldpop, newpop;                    /* Two non-overlapping populations */

int  parent1, parent2, child1, child2;
```

Figure B.1 Definition of the global variables and data structures.

```
/* ************************************************** */
/*                        Utilities                   */
/*       MyRand: returns a float from 0 to n          */
/*       Flip: is a biased coing                      */
/*       1 is heads, all else is 0                    */
/*       rnd: returns an integer between low and high inclusive */
/* ************************************************** */

/*       Returns a random float from 0 to n    */
float MyRand(float n)
{return(n*rand()/RAND_MAX);}


/*       Flip a biased coin - 1 if heads else 0   */
int  flip(float probability)
{if (MyRand(1) < probability)
    {return(1);}
else
    {return(0);}
}


/*       Pick a random integer between low and high inclusively      */
int rnd(int low,int high)
{return (low + floor(0.5 + MyRand(high-low)));}
```

Figure B.2 Random number utility functions.

```
/* ************************************************** */
/*              Decoding and Objective Functions      */
/*              Change these for different problem    */
/* ************************************************** */

/*       Fitness function - f(x) = x**n                */
float objfunc(float x)
{return(pow(x, 10));}
```

```
/*      Decode string as unsigned binary integer                */
/*      Really only necessary if you need to intrepret the      */
/*      chromosome represesnetation.                            */
float decode(population *pop, int IndivID)
{int j;
 int lbits=pop->lchrom;
 float accum, powerof2;
 accum = 0.0; powerof2 = 1;
 for (j=0; j<lbits; j++)
   {if (pop->people[IndivID].chrom[j]==1) {accum = accum + powerof2;}
    powerof2 = powerof2 * 2;}
 return(accum/1073741823.0);}                    /*      return value from 0 to 1*/
```

Figure B.3 Fitness function calculation function and the function used to convert a 30 bit string to a base 10 float.

```
/* ************************************************ */
/*                    Collect Users Parameters                  */
/* ************************************************ */

void InitData(population *pop)
/*      Interactive data inquiry and setup      */
{int I;
 float F;
 printf("\n\n\n\n");
 printf("A Simple Genetic Algorithm\n");
 printf("Author: A.B.Conru        Modified for Think C by K.J.Kostuk\n\n");
 printf("******** SGA Data Entry and Initialization ***********\n\n");

pop->popsize=100;
pop->lchrom=30;
 maxgen=90;
pop->pcross=0.8;
pop->pmutation=.01;
 printf("Enter 1 to use defaults\n"); scanf("%d",&I);
 if (I!=1)
   {printf("Enter population size ------- > "); scanf("%d",&I);pop->popsize=I;
    printf("Enter chromosome length ----- > "); scanf("%d",&I);pop->lchrom=I;
    printf("Enter max. generations ------ > "); scanf("%d",&I); maxgen=I;
    printf("Enter crossover probability - > "); scanf("%f",&F);pop->pcross=F;
    printf("Enter mutation probability -- > "); scanf("%f",&F);pop->pmutation=F;}
pop->nmutation = 0;
pop->ncross = 0;}
```

Figure B.4 The procedure used to collect the user's GA parameters.

```
/* ********************************************************  */
/*                    Initialization of population                  */
/* ********************************************************  */

void InitPop(population *pop)
```

```
/*      Initialize a population at random       */
{int j, c;
 for (j=0; j<pop->popsize; j++)
   {for (c=0; c<pop->lchrom; c++) {pop->people[j].chrom[c] = flip(0.5);}
     pop->people[j].x = decode(pop,j);              /* Decode the string     */
     pop->people[j].fitness = objfunc(pop->people[j].x); /* Evaluate inital fitness */
     pop->people[j].parent1 = 0;
     pop->people[j].parent2 = 0;
     pop->people[j].xsite = 0;
   }
}
```

Figure B.5 Procedure generating the initial population.

```
/* ********************************************************  */
/*                    Population Statistics Calculations                 */
/* ********************************************************  */

void Statistics(population *pop)            /*      Calculate population        */
{int j;                                     /*      statistics                  */
 float f;
 /*      Initialize local variables         */
pop->sumfitness = 0; pop->min = 999999; pop->max = -999999;

 /*      Loop for max, min, sumfitness      */
 for (j=0; j<pop->popsize; j++)
   {f=pop->people[j].fitness;
    pop->sumfitness =pop->sumfitness + f;   /*      Accumulate fitness sum      */
    if (f>pop->max) pop->max = f;           /*      New max                     */
    if (f<pop->min) pop->min = f;           /*      New min                     */
   }
 /*      Calculate average      */
pop->avg =pop->sumfitness/pop->popsize;}
```

Figure B.6 Population statistics procedure. Calculates the maximum, minimum and average fitness of a generation.

```
/* ************************************************  */
/*                          Initial Report                          */
/* ************************************************  */

/*      Initial report    */
int InitReport(char *outfile, population *pop)
{FILE *file;
 if ((file = fopen(outfile,"w")) == NULL)
   {printf("%s: can't open %s\n",outfile,outfile);
    return(0);}

 fprintf(file,"------------------------------------------------------------\n");
 fprintf(file,"|    A Simple Genetic Algorithm - SGA - v1.0     |\n");
 fprintf(file,"------------------------------------------------------------\n\n");
```

```
fprintf(file,"  SGA Parameters\n");
fprintf(file,"  ------------------\n\n");
fprintf(file,"  Population size (popsize)       = %d\n",pop->popsize);
fprintf(file,"  Chromosome length (lchrom)      = %d\n",pop->lchrom);
fprintf(file,"  Maximum # of generation (maxgen)  = %d\n",maxgen);
fprintf(file,"  Crossover probability (pcross)    = %f\n",pop->pcross);
fprintf(file,"  Mutation  probability (pmutation) = %f\n\n",pop->pmutation);
fprintf(file,"  Initial Generation Statistics\n");
fprintf(file,"  ------------------------------\n\n");
fprintf(file,"  Initial population maximum fitness = %f\n",pop->max);
fprintf(file,"  Initial population average fitness = %f\n",pop->avg);
fprintf(file,"  Initial population minimum fitness = %f\n",pop->min);
fprintf(file,"  Initial population sum of fitness  = %f\n\n",pop->sumfitness);

fprintf(file,"==========================================================
==\n");

/*       Repeat some of the output to screen    */
 printf("  SGA Parameters\n");
 printf("  ------------------\n\n");
 printf("  Population size (popsize)                = %i\n",pop->popsize);
 printf("  Chromosome length (lchrom)               = %i\n",pop->lchrom);
 printf("  Maximum # of generation (maxgen)         = %i\n",maxgen);
 printf("  Crossover probability (pcross)           = %f\n",pop->pcross);
 printf("  Mutation  probability (pmutation)        = %f\n\n",pop->pmutation);
 printf("  Initial Generation Statistics\n");
 printf("  ------------------------------\n\n");
 printf("  Initial population maximum fitness       = %f\n",pop->max);
 printf("  Initial population average fitness       = %f\n",pop->avg);
 printf("  Initial population minimum fitness       = %f\n",pop->min);
 printf("  Initial population sum of fitness        = %f\n",pop->sumfitness);
 fclose(file);
 return(1);
}
```

Figure B.7 Procedure used to generate a report on the initial population.

```
/*********************************************  */
/*              Create Spaces - for aesthetics only          */
/* *********************************************  */

void Spaces(FILE *file, int n)
{int i;
 for (i=0; i<n; i++)
  fprintf(file," ");
}
```

Figure B.8 A utility procedure used to generate n spaces.

```
/* ****************************************** */
/*                Population Report           */
/* ****************************************** */

int Report(char *outfile)
{int c,j;
 FILE *file;
 if ((file = fopen(outfile, "a")) == NULL)
  {printf("%s: can't open %s\n",outfile,outfile);
   return(0);}

fprintf(file,"Population Report for Generation %i\n",gen);
fprintf(file," #       string                    x  fitness  parents xsite\n");

for (c=0; c<oldpop.popsize; c++)
  {fprintf(file," %d ",c);

   /*    New string (offspring)          */
   for (j=newpop.lchrom; j>0; j--) {fprintf(file,"%d",newpop.people[c].chrom[j-1]);}
   Spaces(file,40-newpop.lchrom);
   fprintf(file,"%5f %6.4f (%i %i)    %i",newpop.people[c].x,
           newpop.people[c].fitness,newpop.people[c].parent1,newpop.people[c].parent2,
           newpop.people[c].xsite);
   fprintf(file,"\n");
   }

/*      Generation statistics and accumulated values  */
fprintf(file," Gen=%2i max=%6.4f min=%6.4f avg=%6.4f sum=%6.4f nMut=%i
nCross=%i\n",
         gen,newpop.max,newpop.min,newpop.avg,newpop.sumfitness,
         newpop.nmutation,newpop.ncross);


fprintf(file,"==================================================
==\n");

 printf(" Gen=%2i max=%6.4f min=%6.4f avg=%6.4f sum=%6.4f nmutation=%i
ncross=%i\n",
        gen,newpop.max,newpop.min,newpop.avg,newpop.sumfitness,
        newpop.nmutation,newpop.ncross);

 fclose(file);
 return(1);}
```

Figure B.9 Reporting procedure used to store data on the chromosome in each generation

```
/* ************************************************** */
/*                    Initialization Coordinator      */
/* ************************************************** */

void initialize()
{InitData(&oldpop);          /* Collect user's parameter settings          */
 InitPop(&oldpop);           /* Generate initial population                */
 Statistics(&oldpop);        /* Collect statistics on population           */
 newpop=oldpop;              /* Save old population in new population      */
 InitReport("SGA-Report",&oldpop);        /* Generate initial report       */
 Report("SGA-Report");}      /* Save population data                       */
```

Figure B.10 Procedure used to generate the intial population.

```
/* ************************************************************ */
/*                    triops.sga                                */
/*3-operators:Reproduction (select), Crossover (crossover) & Mutation (mutation) */
/* ************************************************************ */

int select(population *pop)
/*      Select a single individual via roulette wheel selection      */
{float RandPt, partsum;              /* Random point on wheel, partial sum   */
 int j;                              /* population index                     */
 partsum = 0.0; j = 0;               /* Zero out counter and accumulator     */
 RandPt = MyRand(pop->sumfitness); /* Wheel point calc. uses random number [0,1] */

 while (j<pop->popsize-1 && partsum < RandPt)        /* Find wheel slot   */
  {j++; partsum += pop->people[j].fitness;}
 return(j);}

/*      Mutate an allele w/ pmutation, count number of mutations     */
int mutation(population *pop, int AlleleVal)
{if (flip(pop->pmutation)==1)                    /*    Flip the biased coin  */
   {pop->nmutation=pop->nmutation+1;
    if (AlleleVal==1) return(0); else return(1);  /*    Change bit value      */
    }
 else
   {return(AlleleVal);                           /*    No change             */}
}

/*      Cross 2 parent strings, place in 2 child strings             */
int crossover(int parent1, int parent2, population *LastPop,
              int child1, int child2, population *pop)
{int j,jcross;
 if (flip(pop->pcross)==1)                       /* Do crossover with pcross  */
   {jcross = rnd(1,pop->lchrom-1);               /* Cross between 1 and l-1   */
    pop->ncross=pop->ncross+1;}                  /* Increment crossover counter */
 else                                            /* Otherwise set cross site to */
   {jcross = pop->lchrom;}                       /* force mutation            */

/*      1st exchange, 1 to 1 and 2 to 2          */
for (j=0; j<jcross; j++)
```

```
 {pop->people[child1].chrom[j] = mutation(pop,LastPop->people[parent1].chrom[j]);
  pop->people[child2].chrom[j] = mutation(pop,LastPop->people[parent2].chrom[j]);}

/*      2nd exchange, 1 to 2 and 2 to 1        */
if (jcross!=pop->lchrom) /* Skip if cross site is lchrom--no crossover */
  {for (j=jcross; j<pop->lchrom; j++)
    {pop->people[child1].chrom[j] = mutation(pop,LastPop->people[parent2].chrom[j]);
     pop->people[child2].chrom[j] = mutation(pop,LastPop->people[parent1].chrom[j]);}}
return(jcross);}
```

Figure B.11 The base procedure to the algorithm. Chromosome selection, mutation and reproduction functions are included here. Note that mutation is called from the crossover function.

```
/* ************************************************** */
/*                      generate.sga                  */
/* ************************************************** */

void Generation()
/*      Create a new generation through select, crossover, and mutation        */
/*      Note: generation assumes an even-numbered popsize                      */
{int j, mate1, mate2, jcross;

 newpop.nmutation=0;
 newpop.ncross=0;
 /*     select, crossover, and mutation until newpop is filled                  */
 for (j=0; j<oldpop.popsize; j=j+2)
   {mate1 = select(&oldpop);                    /* pick pair of mates         */
    mate2 = select(&oldpop);
    /*   Crossover and mutation - mutation embedded within crossover           */
    jcross=crossover(mate1,mate2,&oldpop,j,j+1,&newpop);

    /*   Decode string, evaluate fitness, & record parentage date on both children */
    newpop.people[j].x = decode(&newpop,j);
    newpop.people[j].fitness = objfunc(newpop.people[j].x);
    newpop.people[j].parent1 = mate1;
    newpop.people[j].parent2 = mate2;
    newpop.people[j].xsite   = jcross;

    newpop.people[j+1].x = decode(&newpop,j+1);
    newpop.people[j+1].fitness = objfunc(newpop.people[j+1].x);
    newpop.people[j+1].parent1 = mate1;
    newpop.people[j+1].parent2 = mate2;
    newpop.people[j+1].xsite   = jcross;}}
```

Figure B.12 The procedure used to generate a new generation/population of chromosomes. Chromosomes are selected via roulette wheel selection; reproduction and mutation follow. Finally, the chromosomes are decoded and their fitness is calculated.

```
/* ********************************************************** */
/*                          Main program                      */
/* ********************************************************** */

main()
{gen = 0;
 srand(308);
 initialize();                  /*   Initial population stuff     */
 while (gen < maxgen)           /*   Main iterative loop          */
 {gen++;
  Generation();                 /*   Generate new population      */
  Statistics(&newpop);          /*   Calculate population stats   */
  Report("SGA-Report");         /*   Report on generation         */
  oldpop = newpop;              /*   advance the generation       */
 }
}
```

Figure B.13 Main procedure directs the flow of the algorithm. Note all of the work necessary to generate an initial population is performed in the initialize() procedure.