Sensitivity Analysis of Certain Time Dependent Matroid Base Models Solved by Genetic Algorithms

Joseph DeCicco

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

at

School of Computer Science and Information Systems

Pace University

May 11, 2002

Dissertation Approval

We hereby certify that this dissertation, submitted by Joseph DeCicco, satisfies the dissertation requirements for the degree of Doctor of Professional Studies and has been approved.

____-_______________

Michael Gargano 11 May 2002
Chairperson of Dissertation Committee

____-_______________

William Edelson 11 May 2002
Dissertation Committee Member

____-_______________

Allen Stix 11 May 2002
Dissertation Committee Member

School of Computer Science and Information Systems
Pace University 2002

Abstract

Since the advent of the computer, computer scientists have studied evolutionary systems with the idea that evolution could be used as an optimization tool for engineering problems. John Holland invented genetic algorithms (GA's) in the 1960s[2]. The genetic algorithm paradigm is an adaptive methodology based on Darwinian natural selection and genetic inheritance. It applies operations of selection (based on survival of the fittest), crossover (i.e., mating) and mutation to a given population of potential solutions to generate a new, more fit population of potential solutions. After a number of generations the process converges to an optimal or near optimal solution.

Although his goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature, Holland developed ways in which the mechanisms of natural adaptation might be implemented in computer systems. Holland's 1975 book "Adaptation in Natural and Artificial Systems" presented the genetic algorithm as a construct of biological evolution and gave a theoretical framework for using the genetic algorithm [18]. All of these genetic algorithms performed optimization on problems where the costs were fixed with no dependency on time.

Researchers Gargano and Edelson [1] have developed five theoretical models to study the use of genetic algorithms (GA) on optimal matroid base models whereby the matroid element costs are not fixed, but are time dependent. To date, only one has actually been coded and executed but without further analysis [6]. As stated previously, what is unique about these models is that matroid element costs are not fixed, but are time dependent. It is important to understand the impact of certain factors on the performance of genetic algorithms. Many factors such as population size, reproduction operators, fitness function, and encoding methods have a significant impact on the performance of the algorithm. Researchers have studied the impact on genetic algorithm's performance from various factors including problem encoding, crossover and mutation operators, population size, crossover and mutation rate, and stop criterion [DeJong, 1975; Grefenstette, 1986; Dengiz, Altiparmak, and Smith, 1997] [4].

This dissertation is designed to study the performance of genetic algorithms where the costs are not fixed but time dependent. Using two of the models developed by Gargano and Edelson, this dissertation set out to implement these models and study the factors that influence the performance of these types of genetic algorithms. Not only will the study focus on the classical factors such as population size, problem size, crossover and mutation rate but also new factors brought about by the fact that element costs are time dependent. These new factors which influence these types of genetic algorithms are the structure of the cost table and the introduction of slack time.

Acknowledgments

I would like to thank several individuals.

First, I would like to thank my advisor for his many contributions to my work here. He has served as an invaluable doorway for this research. He has become more than an advisor but a friend.

I would also like to thank my advisors, William Edelson and Allen Stix for their contributions. Their advice was quite valuable to the success of this research.

I would like to thank Fred Grossman. Again, this dissertation would not be possible without his strong leadership and caring personality.

I would like to thank my wife, Donna, who gave up our time together so that I may complete this. Also, without her encouragement and understanding this research would not be possible.

I would like to thank my parents, Vicenzo and Antoinette DeCicco for giving me the inspiration to have me complete something of this great importance.

I would like to thank my mother in-law, Nancy Bollella for accommodating the special necessities for me to complete this research.

Finally, I like to thank my special friend Arturo, who together with his sister both have provided many hours of support when it was needed most.

Table of Contents

Page

Dissertation Approval ii

Abstract iii

Acknowledgments iv

Table of Contents v

List of Tables vii

List of Figures viii

Chapter 1: Introduction 1

1.1: Description of Model 1: Minimal Node Base - A Directed Communications Network Application 3
1.2: Definition of Terms 11
1.3: Description of Model 2 - Minimum Bidding (with Slack Time) Application 16
1.4: Factors under Investigation 19

Chapter 2: The Importance of This Study 34

Chapter 3: The Cost Dependent Genetic Algorithm Revealed 39

3.1: Description of the Genetic Algorithm 39
3.2: Presentation of the Results. 50

Chapter 4: Results 57

4.1: Results from Model 1 57
4.2: Results of Model 2 72

Chapter 5: Conclusions 88

References..92

Appendix A. Calculation of Best Result using the Brute Force Method.................93

Appendix B. Java Code Used for this Study...105

List of Tables

Page

Table 1.1: Example Cost Table……………………………………………………... 5

Table 1.2: Table of Strong Components…………………………………………. 8

Table 1.3: Twelve Possible Combinations……………………………………….. 8

Table 1.4: Number of Bids for Links to Connect Four Cities………………………. 17

Table 1.5: Example of Cost Table……………………………………………….. 17

Table 1.6: Random Cost Table…………………………………………………... 20

Table 1.7: Non Decreasing Cost Table…………………………………………... 21

Table 1.8: Non Increasing Cost Table…………………………………………… 22

Table 1.9: Multi-Modal Cost Table……………………………………………… 23

Table 1.10: Random Multi-Modal Cost Table……………………………………… 24

Table 1.11: Number of Bids for Links Connecting Four Cities Including
One Slack Period…………………………………………………….. 25

Table 1.12: Cost Table with One Slack Bid……………………………………… 26

Table 1.13: Number Of Bids For Links Connecting Four Cities
Including Two Slack Periods…………………………………………... 27

Table 1.14: Cost Table with Two Slack Bids……………………………………. 28

Table 1.15: Conditions Examined With Slack Time……………………………….. 29

Table 1.16: Cost Table Showing Effect When Starting With Last Result………….. 29

Table 1.17: Conditions Used to Study the Effects of Population Size
Used in Model 1……………………………………………………... 31

Table 1.18: Conditions Used to Study the Effects of Population Size
Used in Model 2……………………………………………………... 31

Table 1.19: Different Problem Sizes Conditions Measured on Model 1…………… 32

Table 1.20: Different Problem Sizes Conditions Measured on Model 2…………… 32

Table 1.21: Mutation Rate Conditions Studied…………………………………... 33

Table 3.1: Mk for the Example Network………………………………………… 41

Table 3.2: Cost Table for Example Network…………………………………….. 41

Table 3.3: Example Collection of 30 Runs………………………………………. 53

Table 3.4: A Second Example Collection of 30 Runs……………………………... 54

List of Figures

Page

Figure 1.1: Example Network………………………………………………………. 4

Figure 1.2: Network's Four Main Sets of Nodes…………………………………… 6

Figure 1.3: Network Strong Components…………………………………………... 7

Figure 1.4: Encoding the Permutation……………………………………………… 10

Figure 1.5: Calculation of Fitness…………………………………………………... 10

Figure 1.6: Pictorial Description of a Grim……………………………………… 14

Figure 1.7: Description of One Generation………………………………………. 15

Figure 1.8: Transportation Network………………………………………………. 16

Figure 1.9: Transportation Network with One Slack Time……………………… 25

Figure 1.10: Transportation Network with Two Slack Times…………………… 27

Figure 3.1: Example Network Implemented……………………………………... 40

Figure 3.2: Example Encoding Following the Rules of Encoding…………………. 43

Figure 3.3: Example Encoding Not Following the Rules of Encoding…………….. 43

Figure 3.4: Comparison of Cost of Most Fit Result Per Generation for the Example Problem…………………………………………………... 51

Figure 3.5: Comparison of Cost of Most Fit Result Per Generation Where the Best Result Is Not Obtained……………………………... 52

Figure 3.6: Comparison of Cost of Most Fit Result Per Generation Where the Best Result Is Not Obtained Showing Where Leveling Off Began……………………………………………….. 55

Figure 4.1: Comparison of Different Costs Table Structures and Different Size of Problems to Number of Generations for Completion (No Adjustment)……………………………………………………….. 58

Figure 4.2: Comparison of Different Costs Table Structures and Different Size of Problems……………………………………………………….. 59

Figure 4.3: Comparison of Different Costs Table Structures and Different Size of Problems vs. Number of Generations for Completion………… 59

Figure 4.4: Number of Uncompleted Runs vs. Generation Number (s = 4)………... 60

Figure 4.5: Number of Uncompleted Runs vs. Generation Number (s = 6)………... 61

Figure 4.6: Number of Uncompleted Runs vs. Generation Number (s = 8)………... 61

Figure 4.7: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Random Cost Table and s = 4 (No Adjustment)………….. 62

Figure 4.8: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using the Random Cost Table and s = 4…………………. 63

Figure 4.9: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Random Cost Table and s = 4 (With Adjustment)………….. 63

Figure 4.10: Comparison of Different Population Sizes and Problem Size to Number of Generations for Completion Using the Random Cost Table (No Adjustment)……………………………….. 64

Figure 4.11: Comparison of Different Population Sizes and Problem Size to Number of Runs Stopped Before Completion Using the Random Cost Table………………………………………………. 65

Figure 4.12: Comparison of Different Population Sizes and Problem Size to Number of Generations for Completion Using the Random Cost Table (With Adjustment)………………………….. 65

Figure 4.13: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Non-Increasing Cost Table s = 4 (With Adjustment)……… 66

Figure 4.14: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using the Non- Increasing Cost Table and s = 4……….. 67

Figure 4.15: Comparison of Different Population Sizes and Size of the Problem to Number of Generations for Completion Using the Never Increasing Cost Table (With Adjustment)………………… 68

Figure 4.16: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Non-Decreasing Cost Table s = 4 (With Adjustment)…….. 68

Figure 4.17: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using the Non-Decreasing Cost Table and s = 4……….. 69

Figure 4.18: Comparison of Different Population Sizes and Size of the Problem to Number of Generations for Completion Using the Non-Decreasing Cost Table (With Adjustment)……………………… 70

Figure 4.19: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Multi-modal Cost Table s = 4 (With Adjustment)……….. 70

Figure 4.20: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using The Multi-modal Cost Table and s = 4…………… 71

Figure 4.21: Comparison of Different Population Sizes and Problem Size to Number of Generations for Completion Using the Multi-modal Cost Table (With Adjustment)………………………….. 72

Figure 4.22: Comparison of Different Costs Table Structures and Different Size of Problems to Number of Generations for Completion Using Model 2 (No Adjustment)………………………… 73

Figure 4.23: Comparison of Different Costs Table Structures and Different Size of Problems That Needed to Be Stopped in Model 2……………. 74

Figure 4:24: Number of Uncompleted Runs vs. Generation Number Using Model 2 (s = 4)………………………………………………… 75

Figure 4:25: Number of Uncompleted Runs vs. Generation Number Using Model 2 (s = 6)………………………………………………… 75

Figure 4:26: Number of Uncompleted Runs vs. Generation Number Using Model 2 (s = 8)………………………………………………… 76

Figure 4.27: Comparison of Problem Size, Generation Size, and Population Size vs. Mean Number of Generations for Completion Using the Random Cost Table in Model 2. (With Adjustment)……………... 77

Figure 4.28: Comparison Of Problem Size, Generation Size, and Population Size That Needed To Be Stopped Using the Random Cost Table in Model 2………………………………………. 78

Figure 4.29: Comparison of Problem Size, Generation Size, and Population Size vs. Mean Number of Generations for Completion Using the Non-Decreasing Cost Table in Model 2 (With Adjustment)…………. 79

Figure 4.30: Graph of Comparison of Problem Size, Generation Size, and Population Size That Needed to Be Stopped Using the Non-Decreasing Cost Table in Model 2…………………………………… 80

Figure 4.31: Comparison of Problem Size, and Mutation Rate for the Random and Non-Increasing Cost Tables vs. Mean Number of Generations for Completion in Model 2 (With Adjustment)………… 81

Figure 4.32: Comparison of Problem Size, and Mutation Rate That Needed to Be Stopped Using the Random and Non-Increasing Cost Table in Model 2……………………………………………………… 82

Figure 4.33: Comparison of a s = 6 Problem with 0, 1, 2 Slack Periods Using the Random and Non-decreasing Cost Table in Model 2 (With Adjustment)………………………………………….. 83

Figure 4.34: Comparison of an s = 6 Problem with 0, 1, 2 Slack Periods That Needed to Be Stopped Using the Random and Non-decreasing Cost Table in Model 2…………………………………………………… 83

Figure 4.35: Comparison of Problem size with 0, 1, 2 Slack Periods Using the Random and Non-decreasing Cost Table in Model 2 (With Adjustment)………………………………………………………. 84

Figure 4.36: Comparison of a s = 6 Problem with 0, 1, 2 Slack Periods Which Needed to Be Stopped Using the Random and Non-Decreasing Cost Table in Model 2…………………………………… 85

Figure 4.37: Comparison of Problem Size with 0, 1, 2 Slack Periods (Where s=6 Ccost Table Was Used for All) Using the Random and Non-decreasing Cost Table in Model 2 (With Adjustment)……... 86

Figure 4.38: Comparison of Problem Size with 0, 1, 2 Slack Periods (Where s=6 Cost Table WasUsed for All) Which Needed to Be Stopped Using the Random and Non-decreasing Cost Table in Model 2……………………………………………………… 87

Chapter 1: Introduction

Since the advent of the computer, computer scientists studied evolutionary systems with the idea that evolution could be used as an optimization tool for engineering problems. Box [1957), Friedman [1959), Bledsoe [1961), Bremermann [1962), and Reed, Toombs, and Baricelli [1967) developed evolution-inspired algorithms for optimization and also machine learning. John Holland invented genetic algorithms (GAs) in the 1960s [2]. The genetic algorithm paradigm is an adaptive methodology based on Darwinian natural selection and genetic inheritance. It applies operations of selection (based on survival of the fittest), crossover (i.e., mating) and mutation to a given population of potential solutions to generate a new, more fit population of potential solutions. After a number of generations the process converges to an optimal or near optimal solution. This methodology is a very useful and powerful tool for generating answers to difficult problems [6]. Although his goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature, Holland developed ways in which the mechanisms of natural adaptation might be imported into computer systems. Holland's 1975 book "Adaptation in Natural and Artificial Systems" presented the genetic algorithm as a construct of biological evolution and gave a theoretical framework for using the genetic algorithm [18]. All of these genetic algorithms performed optimization on problems where the costs were fixed with no dependence on time.

Researchers Gargano and Edelson [1] have developed five theoretical models to study the use of genetic algorithms (GA’s) on optimal matroid base models whereby the matroid

element costs are not fixed, but are time dependent. To date, only one has actually been coded and executed but without further analysis [6]. As stated previously, what is unique about these models is that matroid element costs are not fixed, but are time dependent. It is important to understand the impact of certain factors on the performance of genetic algorithms. Many factors such as population size, reproduction operators, fitness function, encoding methods etc., have a significant impact on the performance of the algorithm. Researchers have studied the impact on genetic algorithm's performance from various factors including problem encoding, crossover and mutation operators, population size, crossover and mutation rate, and stop criterion (DeJong, 1975; Grefenstette, 1986; Dengiz, Altiparmak, and Smith, 1997) [4].

This dissertation is designed to study the performance of genetic algorithms where the costs are not fixed but time dependent. Using two of the models developed by Gargano and Edelson, this dissertation set out to implement these models (which have not been previously implemented) so that a study of factors which influence the performance of these types of genetic algorithms can be understood. Not only will the study focus on the classical factors such as population size, problem size, crossover and mutation rate but also new factors brought about by the fact that element costs are time dependent. These new factors which influence these types of genetic algorithms are the structure of the cost table and the introduction of slack time.

In this chapter is a detailed description of both models is given. Each model will be discussed individually. Since a better understanding of the genetic algorithm can be achieved with reference to a specific model, one model will be described first. This will

be followed by a list of key definitions and a description of the second model. Finally, a detailed description of the factors under investigation for performance in this study will be given.

Chapter 2 will focus on the relevance of these models in solving real world problems. Chapter 3 will describe genetic algorithms in more detail by narrating each step of an example genetic algorithm whereby the optimal result is obtained. Specific details on how the results are reported will be provided. In this chapter we will also describe the projected outcomes based on observations of other types of research. Chapter 4 will present the findings of the experiments. Chapter 5 will focus on conclusions and any future work that can be done.

1.1: Description of Model 1: Minimal Node Base - A Directed Communications Network Application

This model involves the designing of a satellite communication network. An example of the type of network used is found in Figure 1.1 below. Each of the nodes is uniquely labeled 1 to 11 and the arrows show the direction of communication flow between each node.

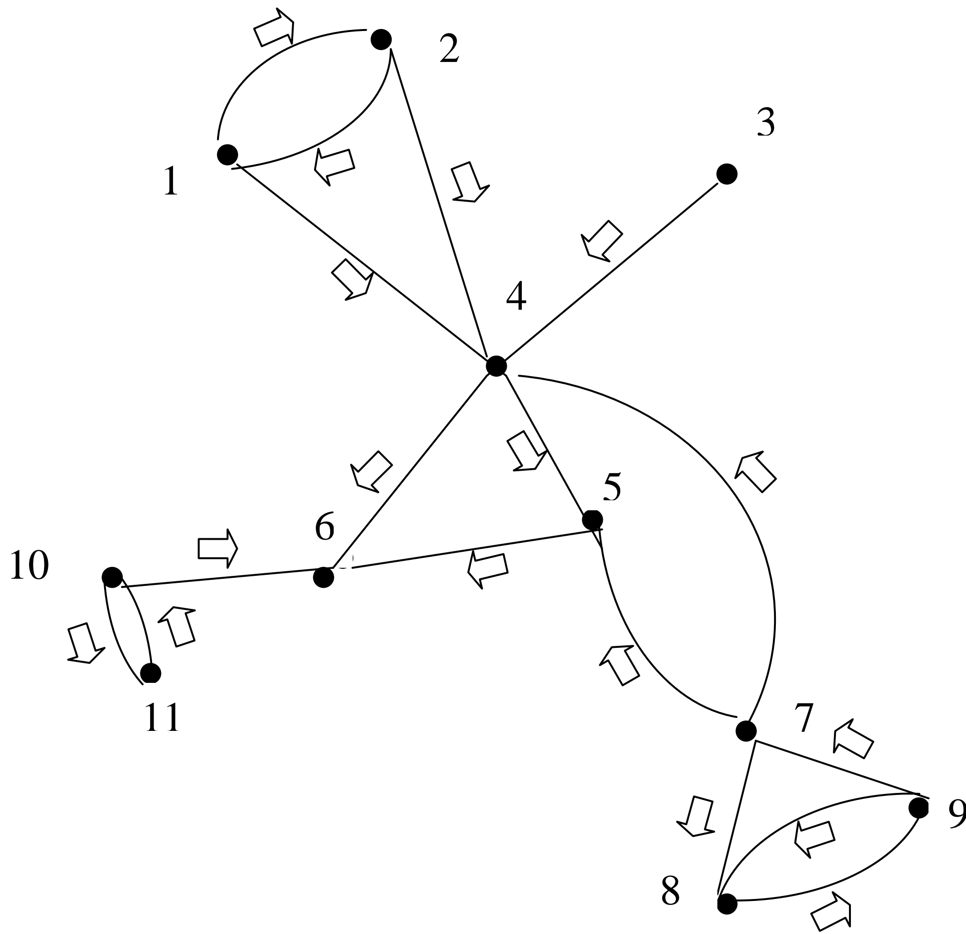

Figure 1.1: Example Network

The objective is to place satellite links (or receivers) for the least cost at specific nodes so that all of the nodes receive signals from the satellite links. The costs associated with building a satellite link at each node at different points in time is shown in Table 1.1.

Node	Cost (in millions of dollars)			
	T=1	**T=2**	**T=3**	**T=4**
1	23	33	34	43
2	11	14	13	56
3	22	22	22	24
4	93	25	44	52
5	86	36	22	45
6	35	12	25	22
7	14	14	13	25
8	13	22	32	42
9	64	76	85	95
10	92	83	73	63
11	23	45	56	36

Table 1.1: Example Cost Table

There are actually only four main sets of points in which a satellite link can be placed so that all of the nodes will receive a communication link. Figure 1.2 shows the four main sets of nodes where at least one communication link must be placed.

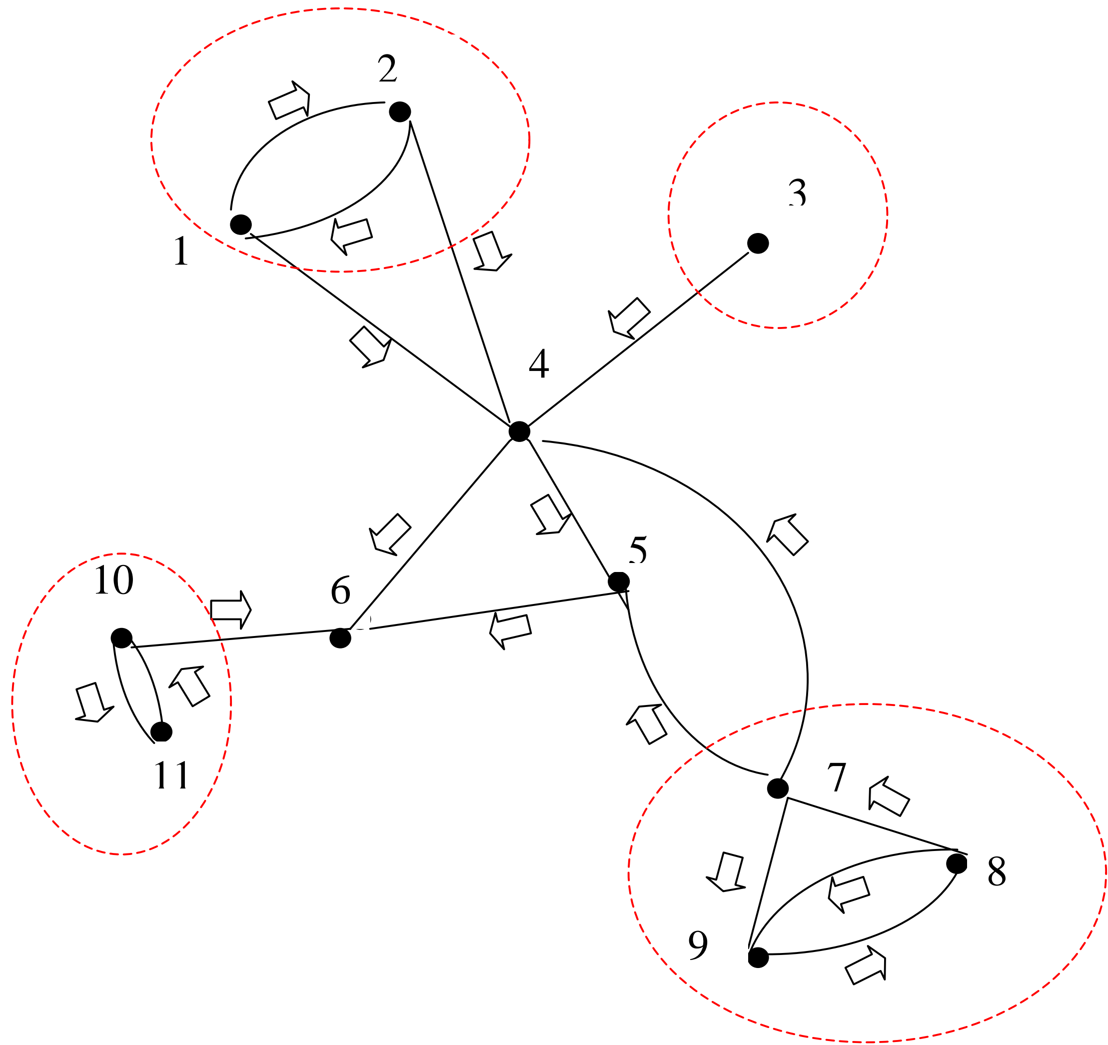

Figure 1.2: Network's Four Main Sets of Nodes

Each one of these areas where at least one communication link must be placed is called a strong component. In this case, there are four strong components. Figure 1.3 shows each of the strong components.

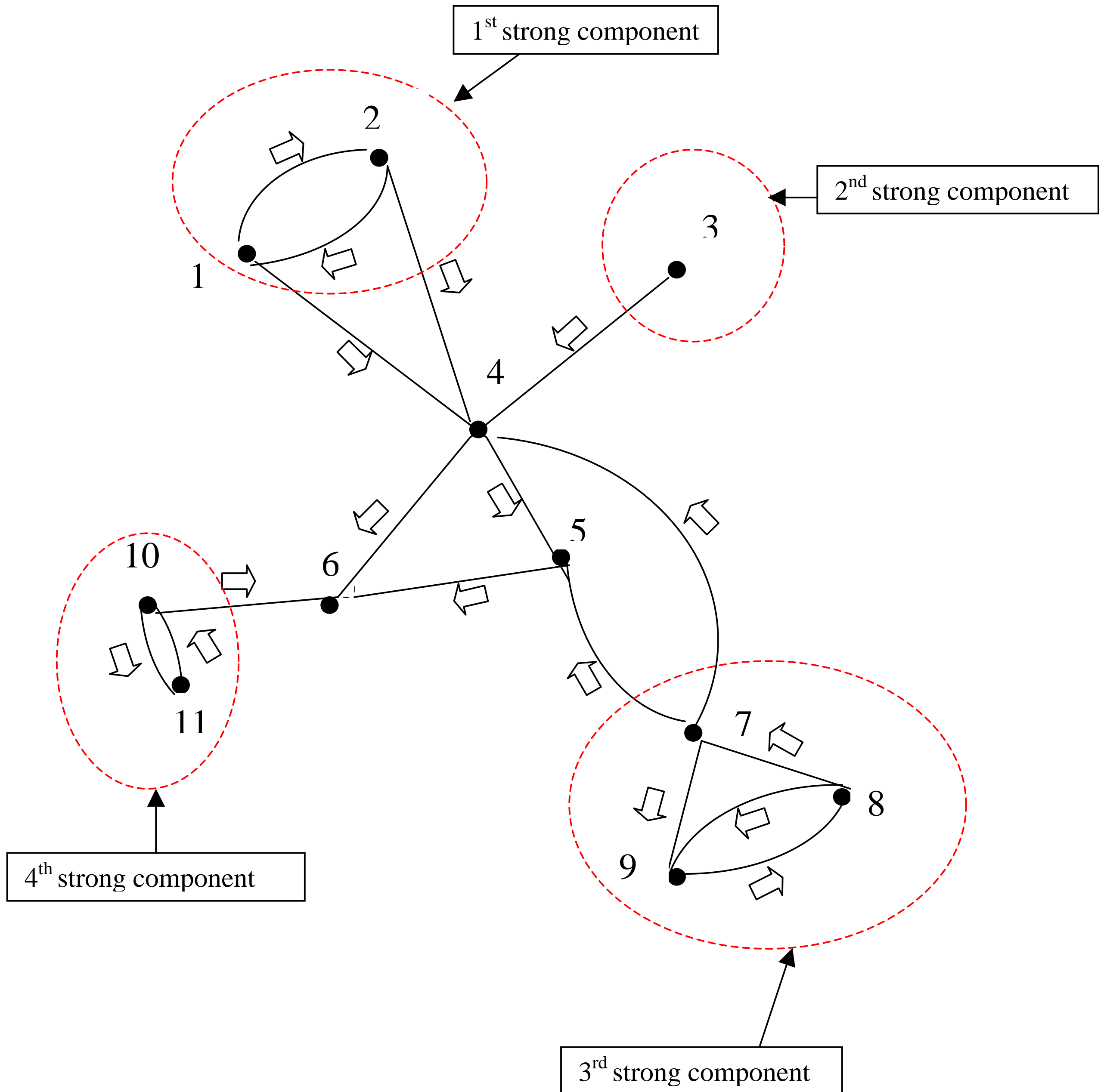

Figure 1.3: Network's Strong Components

The total number of strong components is defined as *s*(i.e., s = 4 above) The number of nodes where at least one communication link can be placed in each strong component is defined as M_k where *k* is the strong component. As can be seen in Table 1.2, the total

number of possible combinations is obtained by taking the product of all of the nodes. In this example the total number of possible combinations is 12 (i.e., 2 • 1 • 3 • 2)

Strong component (k)	M_k
1	2
2	1
3	3
4	2
All possible nodes in all strong components	12

Table 1.2: Table of Strong Components

In this example, the first communication link is built in the first strong component, the second communication link is built in the second strong component, the third communication link is built in the third strong component, and finally the fourth communication link is built on the fourth strong component. In this case there would be a total of twelve possible combinations. They are listed in Table 1.3 below.

1,3,7,10	1,3,7,11	2,3,7,10	2,3,7,11
1,3,8,10	1,3,8,11	2,3,8,10	2,3,8,11
1,3,9,10	1,3,9,11	2,3,9,10	2,3,9,11

Table 1.3: Twelve Possible Combinations

If on the other hand, the order is switched, the number of permutations would increase by $s!$. For example, instead of building the first communication link on the first strong component if the first communication link is built in the second strong component (3,1,7,10). The number of permutations would be 24 (= 4!). The total number of possible solutions is $M_1 \bullet M_2 \bullet M_3 \bullet \ldots \bullet s!$ which in this example is (12 • 24) or 288. For a small value of s, a brute force comparison can be used to determine the correct result. When s

is a large number, brute force is prohibitive and other algorithms such as GA's must be used since the problem is NP-hard [1].

Two entities need to be accounted for in a potential solution. Each component can contain multiple nodes therefore one node from each strong component must be identified as receiving the satellite link. Since there are s strong components, a total of s nodes must be identified. The second entity is the order in which each satellite link will be built. Since there are s links being built, we require s values showing the order in which they will be built. Thus the encoding length needed to determine one possible solution to the problem is equal to $2s$. The first s value's determine the nodes used and the second s values determine the order in which they are to be built.

In the example above, s is 4. An array of numbers that has $2s$ or 8 positions is needed to determine one possible solution. The first four determine the nodes to be used and the second four determine the order in which they are built. As an example, the array (2,3,9,10,2,2,1,1) will be used to encode one possible solution. This is called the genotype. The first step is to take the permutation of the set of numbers. This is done by taking each position from left to right, using the number in the first four positions to determine node numbers, and the numbers in the last four positions to determine the order. In the example (2,3,9,10,2,2,1,1), the first four values of the array (2,3,9,10) are used to determine the nodes numbers to be used and the second four values of the array (2,2,1,1) determine the order in which they are built. The permutation for the array (2,3,9,10,2,2,1,1) is (9,2,3,10). Figure 1.4 demonstrates how this is done. This is called the phenotype.

Position	Array	Permutation	
1	(2,x,x,x,2,x,x,x)	(_,2,_,_)	Put number 2 in the 2nd open position from the left.
2	(x,3,x,x,x,2,x,x)	(_,2,3,_)	Put number 3 in the 2nd open position from the left.
3	(x,x,9,x,x.x,1,x)	(9,2,3,_)	Put number 9 in the 1st open position from the left.
4	(x,x,x,10,x,x,x,1)	(9,2,3,10)	Put number 10 in the 1st open position from the left.

Figure 1.4: Encoding the Permutation

Thus the permutation (9,2,3,10) is one possible solution. The fitness of this member can be determined based on the total cost. The total cost is the sum of the costs of building each link at each time period. The position in the order determines the time period to use. Table 1.1 contains the cost of building each node at different positions in time. As can be seen in Figure 1.5 below, the total cost to build the four links (receiving stations) in order is 163. This is the fitness of one possible solution.

Fitness Value = Cost[Satellite Link in Position 1,Time =1] + … +Cost[Link in Position x, Time =x]… + Cost[Link in Position *s*, Time = *s*]

Fitness Value = Cost[9,1] + Cost[2,2] + Cost[3,3] + Cost[10,4]

Fitness Value = 64 + 14 + 22 + 63

Fitness Value = 163

Figure 1.5: Calculation of Fitness

This is only one member from the total population. The population is a subset from all 288 possibilities. It is important to remember that the goal is to build the network as inexpensively as possible. Therefore, the lower the total cost the more fit the solution. A more detailed description on how the genetic algorithm works will be provided in the subsequent chapters.

1.2: Definition of Terms

The following are definitions of key components.

1. Member (encoding)

   A member is simply one independent integer array. A member or encoding contains all of the components for one possible solution in addition to information about the position of each of the components. A member gets decoded as described in Figure 1.4 to become a permutation of nodes.

2. Permutation

   The permutation is an integer array that represents one possible solution. The integers are placed in the order that they are to be used. The creation of a permutation from an encoding is described in Figure 1.4. The permutation is used to calculate fitness as described in Figure 1.5.

3. Cost Table

   A cost table is simply a two dimensional array of cost values. The first dimension extends from left to right and contains an entry for each time point being considered. The second dimension from top to bottom contains an entry for each component of the problem being considered. Each value is the cost to build that component at that particular time. Table 1.1 is an example of a cost table.

4. Fitness

   Fitness is a calculation performed on one permutation. It is calculated by summing the sum of the cost of each component. The cost value for each component is obtained from the cost table. Calculation of fitness is described in Figure 1.5. Fitness

is used to determine how good the possible solution is compared to other possible solutions.

5. Population and Population Size

   The population is simply a group of members at the start of a new generation. Thus a population consists of the parents upon which operations of crossover and mutation will be performed. The population size is the number of members in this new generation.

6. Generation Size

   The generation size is the total number of members after crossover and mutation has been preformed on a population. It consists of both the original parents and the offspring (including the mutants) and is always greater than the population size.

7. Parent

   A parent is a member upon which the operation of crossover or mutation will be performed. For crossover two parents are necessary, but for mutation only one is needed.

8. Child (Offspring)

   A child is a member produced from the operation of crossover or mutation.

9. Crossover (or mating)

   A crossover (or mating) is simply taking two members that are randomly chosen and then randomly switching some of the values in certain positions of the arrays. An example is given below.

(2,3,9,10,2,2,1,1) Member #1 (or Parent #1)
(1,3,8,11,1,3,2,1) Member #2 (or Parent #2)
(3,3,9,11,1,3,1,1) New member #1 (or Child #1)

Positions 4, 5,and 6 were swapped.

10. Mutation

A mutation is an alteration of one or more values of a member to produce a new member (mutant). An example is given below.

(2,3,9,10,2,2,1,1) Member #1(or Parent #1)
(2,3,7,10,2,2,1,1) New Mutant member #1 (or Child #1)

Position 3 was changed.

11. Sort(or Sorting)

Sorting is the operation of ordering members based on fitness values. Here members are ordered from lowest to highest cost.

12. Grim (or Grim Reaper)

A grim is the operation of selecting more fit members. Less fit members are discarded. A grim always starts with a generation size number of members and takes it down to a population size number of members. The difference between the generation size and the population size is the number of members that will be discarded in a grim. The discarded members are less fit than the members that remain. Grim is always the last step in the cycle of one generation. A pictorial description is shown in Figure 1.6.

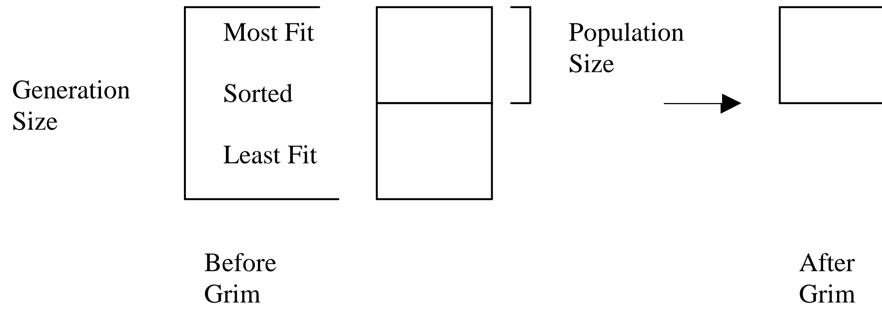

Figure 1.6: Pictorial Description of a Grim

13. <u>Generation</u>

One Generation consists of one complete cycle of a genetic algorithm performed on a population. It is achieved in several phases. Phase one involves performing crossover and mutation on a given population producing enough child members to attain a generation size. Phase two involves sorting. In the actual program, insertion sort can be done after each child member is created instead of waiting until all of the children are created. This was done in all programs created for this dissertation. The final phase involves a grim which removes the less fit members resulting in a new set of members of population size. A pictorial representation is shown in Figure 1.7.

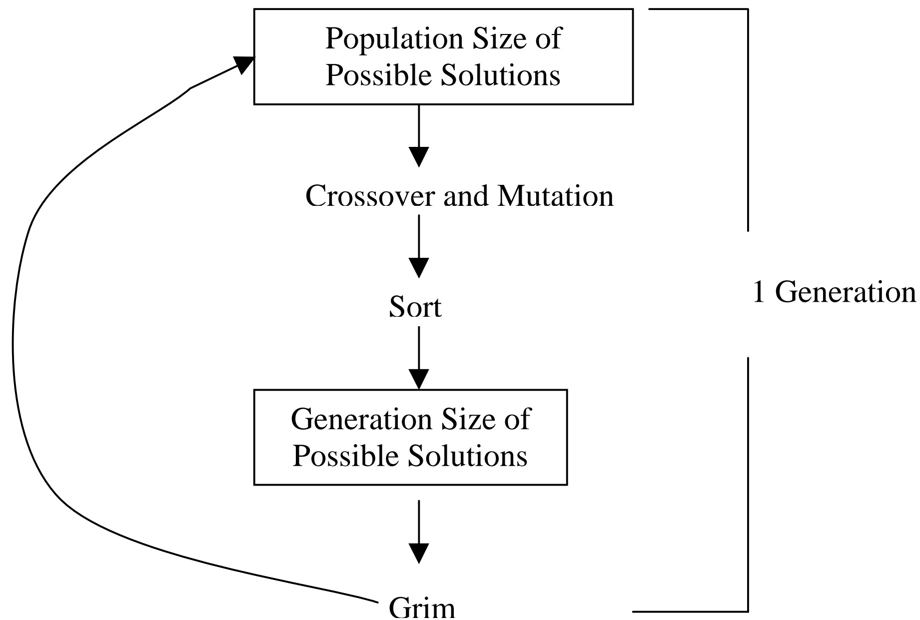

Figure 1.7: Description of One Generation

14. <u>Mutation rate</u>

The mutation rate is the percentage of all the new child members that will be produced by mutation. Obviously, the difference from 100 % will be the percentage of child members produced by crossover. For example, a 30 percent mutation rate would produce 30 out of 100 new members by mutation for each new generation and 70 out of 100 new members by crossover.

15. <u>Crossover rate</u>

The crossover rate is the percentage of all new child members that will be produced by crossover. The difference from 100 % will be the percentage of child members produced by mutation. The crossover rate is always 100 minus the mutation rate. For this reason, the mutation rate will normally be reported and the crossover rate can be derived from the mutation rate.

16. Brute force value (Best Value)

The brute force value is the fitness of the most fit member. It is calculated using a brute force technique where every possible permutation is calculated. This is always implemented in a separate program.

1.3: Description of Model 2 - Minimum Bidding (with Slack Time) Application

This model involves analyzing different bids from different contractors to construct a network connecting a group of cities. Due to budget limitations only one link will be contracted for construction each month. Different contractors having different price scales, which vary over time, bid on the construction of the different links of the potential network. Figure 1.8 below shows a transportation network consisting of four cities.

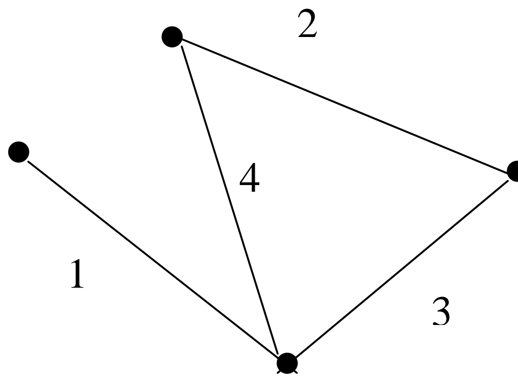

Figure 1.8: Transportation Network

Assuming there are several bids, Table 1.4 below shows the number of bids for each link to connect each of the four cities.

Link number	# of bids obtained to build the link	Bid numbers for all bids obtained to connect that link
1	1	1
2	4	2,3,4,5
3	2	6,7
4	1	8

Table 1.4: Number of Bids for Links to Connect Four Cities

Notice that there are four time periods because only one link will be built per time period. Also, there are eight total bids and several links have multiple bids. Table 1.5 below shows the cost associated with each of the eight bids. It is the cost table used for the genetic algorithm.

	Cost (in millions of dollars)			
Bid #	**T=1**	**T=2**	**T=3**	**T=4**
1	83	59	95	33
2	69	87	32	20
3	55	85	39	74
4	29	87	73	34
5	92	36	43	83
6	51	59	62	35
7	22	21	13	96
8	87	43	43	36

Table 1.5: Example of Cost Table

The number of links is designated s and the number of bids is designated n. In the example above, s is 4 and n is 8. One possible solution is encoded as an array of numbers that has $2s$ or 8 positions. The first four positions determine the bids to be used and the second four positions determine the order in which they are built. For example, the array

(1,3,7,8,3,1,2,1) will be used to encode one possible solution. The first step is to take the permutation of the set of numbers. This is done by taking each position from left to right and using the number in the first four positions to determine bid numbers, and the numbers in the second four positions to determine the order.

In the example (1,3,7,8,3,1,2,1), the first four values of the array (1,3,7,8) are used to determine which bid are used and the second four values of the array (3,1,2,1) are used to determine the order in which to build. As demonstrated below, the permutation for the array (1,3,7,8,3,1,2,1) is (3,8,1,7).

Position	Array	Encoding	
1	(1,x,x,x,3,x,x,x)	(_,_,1,_)	Put number 1 in the 3rd open position from the left.
2	(x,3,x,x,x,1,x,x)	(3,_,1,_)	Put number 3 in the 1st open position from the left.
3	(x,x,7,x,x.x,2,x)	(3,_,1,7)	Put number 7 in the 2nd open position from the left.
4	(x,x,x,8,x,x,x,1)	(3,8,1,7)	Put number 8 in the 1st open position from the left.

Using the example (3,8,1,7), the fitness of this member (one possible solution) can be determined based on the total cost. The total cost is the sum of the costs of each link at that time period. The position in the order determines the time period to use in Table 1.5. The total cost to build this one possible solution is 289.

Fitness Value = Cost[Link in Position 1, Time = 1] + … + Cost[Link in Position x, Time = x]… + Cost[Link in Position s, Time = s]

Fitness Value = Cost of Bid # 3 constructed in the first time period + Cost of Bid # 8 constructed in the second time period + Cost of Bid #1 constructed in the third time period + Cost of Bid # 7 constructed in the fourth time period.

Fitness Value = 55 + 43 + 95 + 96

Fitness Value = 289

In actuality, a thorough examination of the effect of slack time will be studied using this model. Slack time will be discussed in the next section.

1.4: Factors under Investigation

Before a discussion of the factors under investigation begins, an examination of the factors not under investigation is appropriate. Three factors that will not be studied in this dissertation are encoding methods, number of crossover points, and the number of changed values in a mutation. All the genetic algorithms implemented for both models use similar feasible encoding methods. Thus encoding methods will not have an influence on performance in this study. The number of crossover points used is a random number between 1 and the number of nodes studied. Every factor examined will have at least 30 runs. Every run will have anywhere from 40 to 200 crossovers per generation. Multiple generations will be performed for each run. Due to the high number of crossover operations that will occur, an equal representative number of crossover points should occur assuming a good randomness. This also holds true for the number of changed values in a mutation.

Chou and his colleagues used CPU time to measure the performance of the genetic algorithm [4]. The performance of genetic algorithms was measured here by using the number of generations to completion as the unit of measure. This standardizes performance criteria and eliminates any bias that may be introduced by comparing different processor capabilities. Solution quality is also measured by comparing the number of runs that find the best answer (determined by brute force) verses those that

converge before finding the best answer. Any genetic algorithm is either terminated when it finds the best answer (determined by brute force) or when it runs 50 generations without any improvement in the result. The numbers in each case were recorded.

Time dependent (costs not fixed) genetic algorithms introduce new factors that can effect performance. Never before a consideration, the structure of the cost table is a factor worth examining. To do this, a comparison was made of the differences between several different types of cost table structures. For each cost table structure studied, a mean of 30 runs was determined. The different types of cost table structures examined are listed below.

1. Completely Random(RA)
   A cost value for each time period is generated using a random number generator. Table 1.6 is an example random cost table. For any given node, as you traverse from left to right (from T=1 to T= 4), the values do not follow any pattern (they are completely random).

Node	**Cost (in millions of dollars)**			
	T=1	**T=2**	**T=3**	**T=4**
1	31	76	88	39
2	85	88	78	95
3	28	23	89	28
4	62	44	69	79
5	36	69	68	81
6	64	10	56	99
7	82	10	65	26
8	19	43	81	16
9	67	55	39	73
10	55	27	12	30

Table 1.6: Random Cost Table

2. Non-decreasing(ND)

   At each time period, the cost will increase a few percent from the cost of the previous time period. Table 1.7 is an example non-decreasing cost table. For any given node, as you traverse from left to right (from T=1 to T= 4), the values increase slightly (10%) but never decrease. In fact, a 10% increase was used for all non-decreasing cost tables studied.

Node	Cost (in millions of dollars)			
	T=1	T=2	T=3	T=4
1	62	68	74	80
2	16	18	20	22
3	56	62	68	74
4	52	57	62	67
5	21	23	25	27
6	34	37	40	43
7	35	39	43	47
8	39	43	47	51
9	56	62	68	74
10	64	70	76	82

Table 1.7: Non Decreasing Cost Table

3. Non-increasing(NI)

   As each time period progresses, the cost will decrease a few percent from the cost of the previous time period. Table 1.8 is an example non-increasing cost table. For any given node, as you traverse from left to right (from T=1 to T= 4), the values decrease slightly (10%) but never increase. In fact, 10% decrease was used for all non-increasing cost tables studied.

Node	Cost (in millions of dollars)			
	T=1	**T=2**	**T=3**	**T=4**
1	60	54	48	42
2	27	24	21	18
3	65	58	51	44
4	25	22	19	16
5	58	52	46	40
6	76	68	60	52
7	85	76	67	58
8	67	60	53	46
9	58	52	46	40
10	97	87	77	67

Table 1.8: Non Increasing Cost Table

4. Mult-modal (MM)

   The percent change varies from one time period to the next. Also from one time period to the next the value can increase as well as decrease. The percent of change from one time period to the next will be fixed.

   Table 1.9 shows an example multi-modal cost table. Note that for node 1, as you traverse from left to right, the values decrease slightly (10%) from T = 1 to T=2. Then from T=2 to T=3 they increase slightly (10%). Next for node 2, as you traverse from left to right, the values increase slightly (10%) from T =1 to T=2. Then from T=2 to T=3 they decrease slightly (10%). In fact, 10% change one way followed by a 10% change the other was used for all multi-modal cost tables studied.

Node	Cost (in millions of dollars)			
	T=1	T=2	T=3	T=4
1	91	82	91	82
2	35	39	35	39
3	40	36	40	36
4	22	24	22	24
5	25	22	25	22
6	32	35	32	35
7	72	65	72	65
8	71	78	71	78
9	14	13	14	13
10	49	54	49	54

Table 1.9: Multi-Modal Cost Table

5. Random Multi-modal (RM)

   The percent change will vary from one time period to the next. Also from one time period to the next the value can increase as well as decrease. The percent of change from one time period to the next will vary.

   Table 1.10 shows an example random multi-modal cost table. Note that for node 1, as you traverse from left to right, the values decrease slightly (randomly from 1 to 10%) from T = 1 to T=2. Then from T=2 to T=3 they increase slightly (randomly from 1 to 10%). Next for Node 2, as you traverse from left to right, the values increase slightly (randomly from 1 to 10%) from T =1 to T=2. Then from T=2 to T=3 they decrease slightly (randomly from 1 to 10%). In fact, a random change from 1 to 10% one way followed by a random change from 1 to 10% change the other was used for all random multi-modal cost tables studied.

Node	Cost (in millions of dollars)			
	T=1	**T=2**	**T=3**	**T=4**
1	75	68	75	68
2	63	66	60	64
3	48	44	47	46
4	89	96	91	95
5	26	24	26	25
6	79	84	83	89
7	20	18	20	19
8	15	16	14	15
9	30	28	31	28
10	62	65	63	67

Table 1.10: Random Multi-Modal Cost Table

Analysis for 30 different cost tables will be run for each variant described above. An analysis will then be performed to determine what effect, if any, varying the cost will have on the number of generations needed to obtain the most fit solution. The effect of the cost table will be investigated using both models.

Another factor that can effect performance that is unique to genetic algorithms where the costs are not fixed but time dependent is slack time. Slack time will be primarily investigated using model 2. In the original model, there was no time to rest at any period because we had to complete s tasks in s time periods. Here we have q + s time periods for the same tasks we can rest q periods and work s periods. The concept here is to measure whether adding dummy (slack) bids with zero costs for each period allow the genetic algorithm to obtain a more complete result in fewer generations. A better understanding of this can be obtained by examining model 2. In order to force the problem to add one period of slack time, one of the links must contain only one bid and that bid must have a value of zero. For example, adding one dummy node to Figure 1.8 results in Figure 1.9.

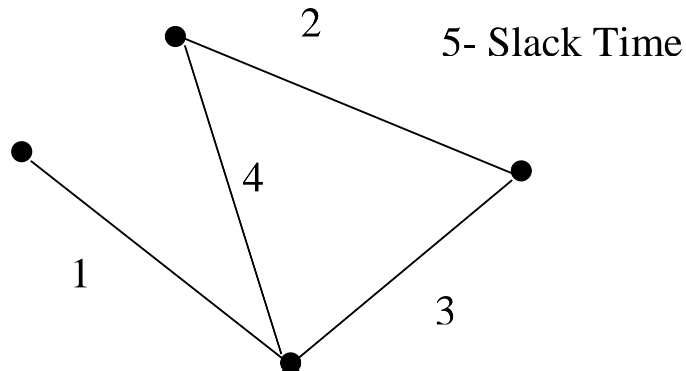

Figure 1.9: Transportation Network with One Slack Time

As seen in Table 1.11, there are several bids for each of the four links to the cities and one slack bid. There are eight total bids plus 1 slack bid for a total of nine bids. Note that several links have multiple bids.

Link number	# of Bids obtained to connect the 2 cities	Bid numbers for all bids obtained to connect that link
1	1	1
2	4	2,3,4,5
3	2	6,7
4	1	8
5	1	9

Table 1.11: Number of Bids for Links Connecting Four Cities Including One Slack Period

Table 1.12 below shows the cost associated with each of the nine bids. The number of links is s and the number of bids is n. The slack periods are referred to as q. There are five time periods because only one link will be built per time period. Thus, there is one (s + q) periods in which no link will be built. A zero is placed in each of the time points for Bid 9 since it is the slack bid.

	Cost (in millions of dollars)				
Bid #	**T=1**	**T=2**	**T=3**	**T=4**	**T=5**
1	83	59	95	33	15
2	69	87	32	20	36
3	55	85	39	74	59
4	29	87	73	34	69
5	92	36	43	83	34
6	51	59	62	35	32
7	22	21	13	96	73
8	87	43	43	36	21
9	0	0	0	0	0

Table 1.12: Cost Table with One Slack Bid

In the example above, s is 4 and q is 1. An array of numbers that has $2(s + q)$ or 10 positions is needed to determine one possible solution. The first five positions determine the bids to be used and the second five positions determine the order to build them. An additional time slot is needed when compared to the previous cost table in order to account for the extra slack time period. The array (2,1,9,8,6,2,3,2,1,1) will be used to encode one possible solution. The first step is to take the permutation of the set of numbers. This is done by taking each position from left to right and using the number in the first five positions to determine bid numbers and the second five positions to determine the order. The same procedure as described in Figure 1.4 would be performed except now for 5 positions instead of 4.

For completeness, let us show what happens with 2 slack periods as seen in Figure 1.10.

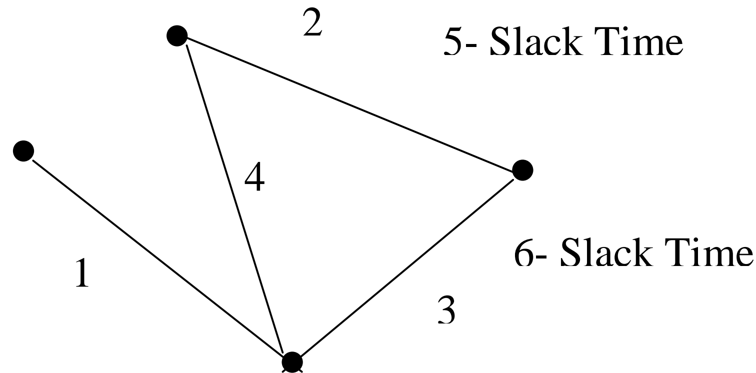

Figure 1.10: Transportation Network with Two Slack Times

There are several bids for each of the 4 links to the cities and two slack bids. Table 1.13 below shows the number of bids for links connecting each of the four cities.

Link number	# of Bids obtained to connect the 2 cities	Bid numbers for all bids obtained to connect that link
1	1	1
2	4	2,3,4,5
3	2	6,7
4	1	8
5	1	9
6	1	10

Table 1.13: Number Of Bids For Links Connecting Four Cities Including Two Slack Periods

Table 1.14 below shows the cost associated with each of the eight bids and two slack times. Notice that there are six time periods because only one link will be built per time period. Thus, there are two (q + s) periods in which no link will be built. Also, several links have multiple bids. Once again, zeros are placed in the time periods for the two slack bids, which are bid 9 and bid 10.

	Cost (in millions of dollars)					
Bid #	**T=1**	**T=2**	**T=3**	**T=4**	**T=5**	**T=6**
1	83	59	95	33	15	55
2	69	87	32	20	36	73
3	55	85	39	74	59	35
4	29	87	73	34	69	24
5	92	36	43	83	34	91
6	51	59	62	35	32	65
7	22	21	13	96	73	69
8	87	43	43	36	21	14
9	0	0	0	0	0	0
10	0	0	0	0	0	0

Table 1.14: Cost Table with Two Slack Bids

The number of links is s and the number of bids is n. The slack periods are referred to as q. In the example above, s is 4 and q is 2. An array of numbers that has $2(s + q)$ or 12 positions is needed to determine one possible solution. Notice that two time slots need to be added from the original cost table to account for the two extra slack time periods. The first six positions determine the bids to be used and the second six positions determine the order to build them. The array (2,10,1,9,8,6,4,2,3,2,1,1) will be used to encode one possible solution. The first step is to take the permutation of the set of numbers. This is done by taking each position from left to right and using the number in the first six positions to determine bid numbers and the second six positions to determine the order. The same procedure as described in Figure 1.4 would be performed except now for 6 positions instead of 4.

The conditions outlined in Table 1.15 were examined during this research. Further discussion will be given about conditions shown in line number 6, 7 and 8.

Line Number	Value of s	Value of q	Type of Run
1	4	0	Normal
2	5	0	Normal
3	6	0	Normal
4	5	1	Normal
5	6	2	Normal
6	4	0	Same Cost Table
7	5	1	Start with Previous Result
8	6	2	Start with Previous Result

Table 1.15: Conditions Examined with Slack Time

The first step to studing conditions in line number 6,7, and 8 shown in Table 1.15 was to create a cost table similar to that of Table 1.14. As shown in Table 1.16, only the portion of the cost table needed (depending on the number of slack periods, q) was used.

d = 0 d = 1 d = 2

	Cost (in millions of dollars)					
Bid #	T=1	T=2	T=3	T=4	T=5	T=6
1	83	59	95	33	15	55
2	69	87	32	20	30	73
3	55	85	39	74	59	35
4	29	87	73	34	69	24
5	92	36	43	83	34	91
6	51	59	62	35	32	65
7	22	21	13	96	73	69
8	87	43	43	36	21	14
9	0	0	0	0	0	0
10	0	0	0	0	0	0

Table 1.16: Cost Table Showing Effect When Starting With Last Result

An additional study was performed utilizing the encoding from the best result from condition in line number 6 (containing no slack periods) of Table 1.15 and applying it as

the first member in condition in line number 7 (containing one slack period). This was repeated on condition in line number 8 that contains two slack periods using the best result from condition in line number 7. This was done to see if the number of generations required to obtain the most fit result improved when the problem containing two slack periods was solved beginning with a result from the problem containing zero and one slack periods respectivily.

A detailed description of the factors never before considerd that can effect performance that is unique to genetic algorithms where the costs are not fixed, but time dependent have been given. Factors common to traditional genetic algorithms were also studied. These include population size, problem size, crossover and mutation rate.

In section 1.2, a definition is given for the population size as it pertains to this dissertation. As previously stated, the population is simply a group of members at the start of a new generation. The population consists of the parents to which operations of crossover and mutation will be performed. The population size is the number of members in that population. Another term, which is not seen in the literature, was also defined. The term generation size is the total number of members after crossover and mutations have been performed on a population. It consists of both the parents and the offspring and is always greater than the population size.

The first approach to studying the effect of population size was implemented using model 1. In this approach, the total number (size) of members after crossover and mutation has been preformed on a population, defined here as generation size, was always kept

constant at 100. The population size was varied between 10 and 90. The different conditions are outlined in Table 1.17.

#	Population Size	Generation Size	Number of Parents	Number of Offspring
1	10	100	10	90
2	25	100	25	75
3	50	100	50	50
4	75	100	75	25
5	90	100	90	10

Table 1.17: Conditions Used to Study the Effects of Population Size Used in Model 1

A second approach to studying the effect of population size was implemented using model 2. In this approach, both the population size and generation size were varied. The different conditions are outlined in Table 1.18.

#	Population Size	Generation Size	Number of Parents	Number of Offspring
1	50	100	50	50
2	50	200	50	150
3	100	200	100	100
4	150	200	150	50

Table 1.18: Conditions Used to Study the Effects of Population Size Used in Model 2

Thirty different runs were performed for each of the conditions outlined in Tables 1.16 and 1.17. Observations were made on what effect varying these conditions had on the number of generations needed to obtain the most fit solution.

In model 1, the problem size was measured by the number of strong components. In model 2, the problem size was measured by the number of links and slack time. Varying the number of nodes or links varies the length of the encoding. This will in turn vary the

length of the permutation code. The effect the length of the code has on the number of generations needed to obtain the most fit solution will be discussed in subsequent chapters. Table 1.19 shows the three different conditions measured on model 1.

Condition	Number of strong components
1	4
2	6
3	8

Table 1.19: Different Problem Sizes Conditions Measured on Model 1

Table 1.20 shows the different conditions measured on model 2. The same cost table was used for conditions 4, 5, and 6. In conditions 4, 5, and 6, the slack times studied were 0, 1, and 2, respectively. The experiment began by creating a cost table similar to Table 1.14. This same cost table was used for 4, 5, and 6 except that only the portion of the cost table needed was used. This is explained in detail in Table 1.15 and Table 1.16. Observations were made on what effect varying problem size had on the number of generations needed to obtain the most fit solution.

Condition	Number of Links	Number of slack periods	Comments
1	4	0	Normal
2	6	0	Normal
3	8	0	Normal
4	4	0	Same Cost Table
5	5	1	Start with Last Result
6	6	2	Start with Last Result

Table 1.20: Different Problem Sizes Conditions Measured on Model 2

The final factor that affects performance is the crossover and mutation rate. The mutation rate is the percentage of all the new child members that will be produced by mutation. The crossover rate is the percentage of all new child members that will be produced by

crossover. The crossover rate is always 100 minus the mutation rate. Table 1.21 shows the different conditions studied and which model was used to study each condition.

Condition	% Mutation Rate	% Crossover	Experimental Models
1	10	90	Model 1 and 2
2	30	70	Model 1 and 2
3	60	40	Model 2
4	90	10	Model 2

Table 1.21: Mutation Rate Conditions Studied

The relevance of this study will now be addressed.

Chapter 2: The Importance of This Study

As was stated in Chapter 1, Holland invented genetic algorithms in the early 1960's. Holland's 1975 book Adaptation in Natural and Artificial Systems presented the genetic algorithm as a construct of biological evolution and gave a theoretical framework for using the genetic algorithm [18]. The next paragraph is a quote taken directly from this book.

> Despite a wealth of data from many different fields and despite many insights, we are still a long way from a general understanding of adaptive mechanisms. The situation is much like that in the old tale of blind men examining an elephant- different aspects of adaptation acquire different emphases because of the points of contact. A specific feature will be prominent in one study, obscure in another. Useful and suggestive results remain in comparative isolation. Under such circumstances theory can be a powerful aid. Successful analysis separates incidental or "local" exaggerations from fundamental features. A broadly conceived analytic theory brings data and explanation into the coherent whole, providing opportunities for prediction and control. Indeed there is an important sense in which a good theory defines the objects with which it deals. It reveals their interactions, the methods of transforming and controlling them, and prediction of what will happen to them. Theory will have a central role in all that follows, but only insofar as it illuminates practice. For natural systems this means that theory must provide techniques for prediction and control; for artificial systems, it must provide practical algorithms and strategies.

Although written back in 1975, this quote was included because it describes in great detail the importance of what this dissertation is about. First we begin with theory. Gargano and Edelson (1) have developed five theoretical models to study the use of genetic algorithms (GA) on optimal matroid base models whereby the matroid element costs are not fixed, but are time dependent. Holland's quote "Theory will have a central role in all that follows, but only insofar as it illuminates practice." Two of these theoretical models were put into practice in this dissertation, and for the very reasons Holland talks about above. It is to reveal their interactions, and provide the techniques

for prediction and control. With the information provided from this dissertation, strategies for implementing genetic algorithms whereby the element costs are time depended will be achieved.

Peter Angeline[3] in 1994 provided an historical taxonomy of evolutionary computation. In this taxonomy, he starts his hierarchical first level split by separating methods those evolutionary computations that manipulate either a dynamic structure or a static structure. He placed genetic algorithms as typically evolving static representational structures. This is undoubtedly true for genetic algorithms that are not time dependent. Although not as dynamic as genetic programming, evaluating time dependent costs does add a dynamic component to the picture. Remember, there are still the fixed-length binary strings so genetic algorithms could never be placed in the dynamic split with genetic programming. He realizes the fact that some genetic algorithms have used dynamic representations, but now possibly with the emergence of genetic algorithms with time dependent costs, he may need to create a new first level split to account for these new types of genetic algorithms.

As the name implies, the idea of genetic algorithms derived from the field of Genetics. A dissertation such as this would not be complete without some mention of its relationship to Genetics. Although the entities are much simpler than the real biological ones, a relationship between the genetic algorithm and genetics will be given. Most of the following description was taken from Melanie Mitchell's highly-regarded genetic algorithm text [2].

All living organism cells consist of one or more chromosomes (strings of DNA) that serve as a "blueprint" to create the entire organism. Very roughly, one can think of a chromosome as an encoding. A chromosome can be divided conceptually into genes, each of which encodes a particular trait (such as hair color). There are different possible settings for hair color just like there are different possible solutions for a strong component in model 1. Each gene is located at a particular position on the chromosome.

In model 1, each term is a particular node in one of the strong components. Organisms usually have their chromosomes arrayed in pairs that are called a diploid. When the organism is ready to reproduce, the chromosomes become unpaired or haploid. During sexual reproduction, two haploid pairs from different parents recombine. In each parent, genes are exchanged (crossover) between each pair of chromosomes to form a gamete (a single chromosome), and then the gamates from the two parents pair up to create a full set of diploid chromosomes. In haplod sexual reproduction, genes are exchanged (crossover) between the two parent's single-strand chromosomes. Offspring are also subject to mutation, in which single nucleotides (elementary bits of DNA) are changed from parents to offspring. These changes are often associated with copying errors. The fitness of an organism is typically defined as the probability that an organism will live to reproduce (that is how viable is it). Fitness is a function of the number of offspring an organism can produce.

In a genetic algorithm, such as the one employed in model 1, the chromosome can refer to a candidate solution to the problem. Each gene on the chromosome can be thought of as one node in each strong component. The entire chromosome makes up one possible

solution which means that at least one node from each strong component is present on the chromosome along with information about the time slot each node will build the link. Crossover typically consists of exchanging genetic material between two single chromosomes or two haploid parents. Mutation consists of randomly changing one of the genes. Offspring are produced from the crossover (mating) and from the mutations. The organisms that are produced are evaluated based on how fit they are (done in model 1 by the sorting algorithm). The more fit a solution, meaning the cheaper the cost to build all the links (as in the case of model 1), the less chance of that solution being in the bottom of the sorted list of possible solutions. This implies that there is less of a likelihood for it to be removed by the grim reaper. This means the more fit the solution, the more generations it will survive. The more generations it will survive, the greater the chance it will be randomly chosen to produce more offspring from a mating or mutation. This is one of the underlying principles of Darwin’s survival of the fittest.

Model 1 involves designing a network. Chou [4] studied the performance of a genetic algorithm using a network design model. The difference here is that this model (and this study for that manner) involves not static cost but rather time dependent costs. Network design is studied because it has practical significance. There are many companies and institutions that provide network planning and system design tools. Netlinks [20], and Gilden [21] (this is developed by a university) are two examples. Gilden 2.0 uses many such algorithms as Kruskal’s algorithm, Prim’s algorithm, and Dijkstra’s algorithm. This genetic algorithm in this study was implemented with the software to account for costs that may change with time. This is not currently a feature in any of these software packages previously mentioned. Cox [19] is a Ph.D. who designs and constructs

programs in his consulting company (Cox Associates) to find solutions of difficult problems. He has implemented a genetic algorithm to design networks such as those in model 1. Now using a genetic algorithm where the costs are not fixed but time dependent, he would also be able to account for costs changing over time.

An optimization research problem [17] is being funded in England to find bids to match buyers to sellers of electricity. The problem is actually quite complicated but probably some of the principles from model 2 can be used in the daily bidding process described.

Chapter 3: The Cost Dependent Genetic Algorithm Revealed

This chapter will begin by actually implementing a simple example of a genetic algorithm. A description of the formats to be will also be discussed.

3.1: Description of the Genetic Algorithm

In chapter one, a definition of the genetic algorithm was given. In this chapter, a detailed example of the inner workings of genetic algorithms will be explored by implementing model 1 above with a simple example. Figure 3.1 contains the network that will be used in the example.

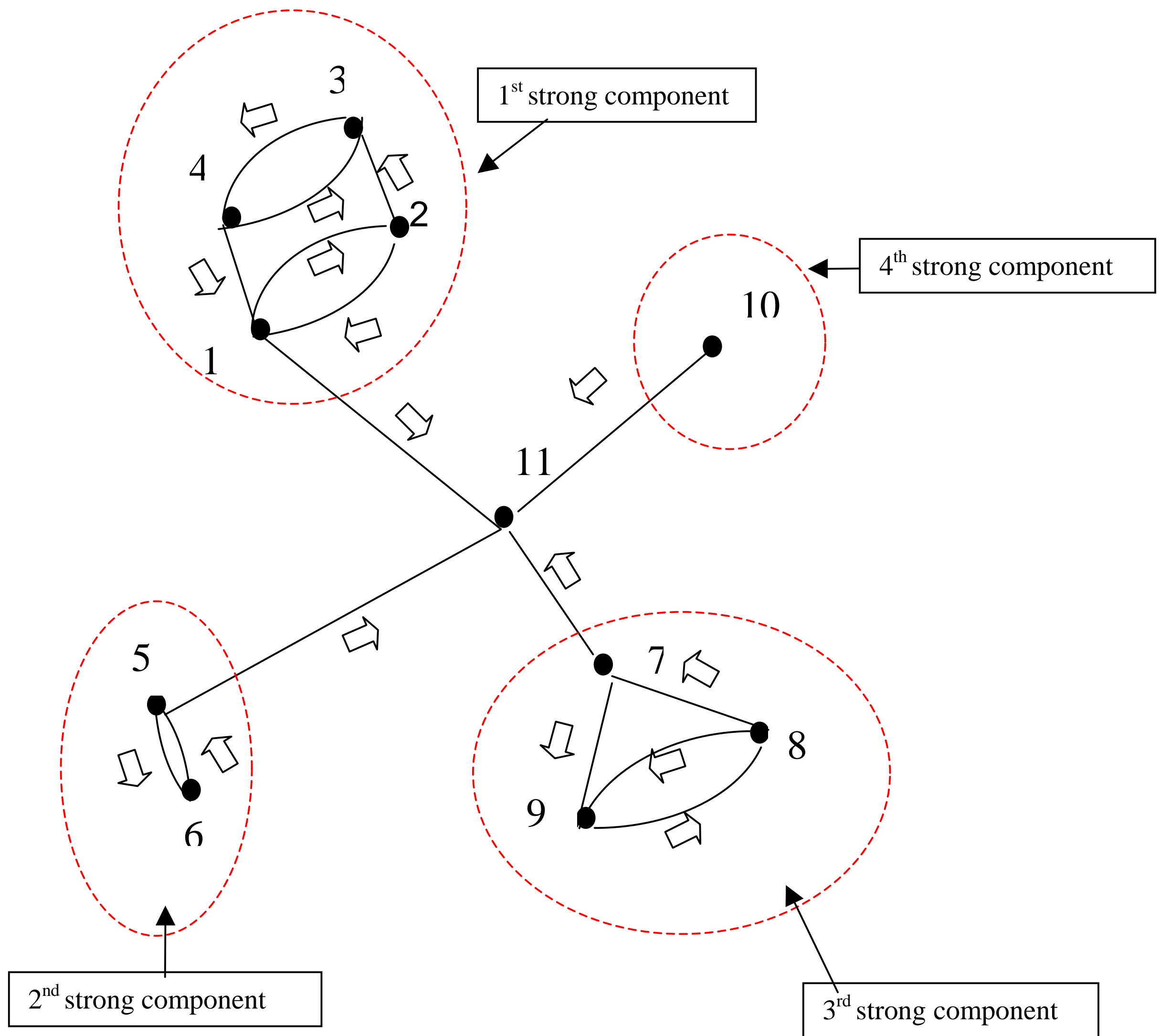

Figure 3.1: Example Network Implemented

The total number of strong components is *s,* which is equal to 4. M_k *is* the number of nodes where at least one communication link can be placed in each strong component. Table 3.1 summarizes these values for this network example.

Strong component (k)	M_k	Nodes in each strong component.
1	4	1, 2, 3, 4
2	2	5, 6
3	3	7, 8, 9
4	1	10
Product of the number of all possible nodes over all strong components	24	

Table 3.1: M_k for the Example Network

First note that a link at node 10 must be built. Also note that in this example node 11 is not included in any strong component because it can receive a signal from many nodes but does not supply any other nodes with signal. Thus, placing a satellite link at node 11 would not supply any other node and would add an additional cost with no benefit. The cost table for this problem is shown in Table 3.2. Note that for each node the costs are non-increasing over time.

Node	Cost (in millions of dollars)			
	T=1	**T=2**	**T=3**	**T=4**
1	97	87	77	67
2	93	84	75	66
3	46	41	36	31
4	27	24	21	18
5	12	11	10	9
6	42	38	34	30
7	44	40	36	32
8	21	19	17	15
9	66	59	52	45
10	96	86	76	66
11	86	77	72	68

Table 3.2: Cost Table for Example Network

The brute force value for this problem is obtained by checking every possible combination. The total number of possible combinations is the product of M_k and $s!$. For this example, $s!$ is 24($4! = 4 \bullet 3 \bullet 2 \bullet 1$) and the product of M_k is 24 ($4 \bullet 3 \bullet 2 \bullet 1$) so the total number of possible permutations is 576.

To truly appreciate how difficult this problem can be with just 4 time periods and 11 nodes, all possible combinations will be displayed in Appendix A. The best fit result (determined from brute force comparison of all the possible combinations) for this problem is the combination 5,8,4,10 with a total cost of 118. This result was obtained by a program that can find the best answer using brute force for small enough examples. See Appendix A for details.

Fitness Value = Cost [Satellite Link in Position 5 ,Time =1] + Cost [Satellite Link in Position 8 ,Time =2] + Cost [Satellite Link in Position 4 ,Time =3] + Cost [Satellite Link in Position 10 ,Time =4]

Fitness Value = Cost[5,1] + Cost[8,2] + Cost[4,3] + Cost[10,4]

Fitness Value = 12 + 19 + 21 + 66

Fitness Value = 118

For this problem, a population size of five and generation size of ten will be used. This means that five offspring will be produced from five parents. A mutation rate of 20 % will be used resulting in one offspring produced from mutation and four offspring produced from crossover. The process begins by selecting five different members of the population randomly using a random number generator according to the rules of encoding. The rules of encoding state that position 1 must contain a 1, 2, 3, or 4. Position 2 must contain a 5 or 6, and position 3 must contain a 7, 8, or 9. Finally position 4 must

contain a 10. The second four values must be able to determine the position. For example, encoding 3,5,7,10,1,3,1,1 following the rules of encoding would result in the permutation 3,7,10,5. Figure 3.2 explains how this is done.

Position	Array	Permutation	
1	(3,x,x,x,1,x,x,x)	(3,_,_,_)	Put number 3 in the 1st open position from the left.
2	(x,5,x,x,x,3,x,x)	(3,_,_,5)	Put number 5 in the 3rd open position from the left.
3	(x,x,7,x,x.x,1,x)	(3,7,_,5)	Put number 7 in the 1st open position from the left.
4	(x,x,x,10,x,x,x,1)	(3,7,10,5)	Put number 10 in the 1st open position from the left.

Figure 3.2: Example Encoding Following the Rules of Encoding

Encoding 3,5,7,10,1,3,1,3 would not follow the rules because you cannot place 10 in the 3rd open position from the left. This is shown in Figure 3.3.

Position	Array	Permutation	
1	(3,x,x,x,1,x,x,x)	(3,_,_,_)	Put number 3 in the 1st open position from the left.
2	(x,5,x,x,x,3,x,x)	(3,_,_,5)	Put number 5 in the 3rd open position from the left.
3	(x,x,7,x,x.x,1,x)	(3,7,_,5)	Put number 7 in the 1st open position from the left.
4	(x,x,x,10,x,x,x,3)	(3,7,_,5)	ERROR: Cannot put number 10 in the 3rd open position from the left because there is only one open position left.

Figure 3.3: Example Encoding Not Following the Rules of Encoding

Thus we see that the second four values in the array must follow the rules of encoding also. Let us begin by randomly electing five parents for our initial population. This is shown below sorted with the fitness value. The fitness is given as the cost in millions of dollars. This will be called Generation 0. The order refers to the order that member was created in that generation.

<u>Generation number = 0</u> (Sorted by Fitness)

1) Fitness = 171.0 Generation created = 0 Order = 4 Permutation = 3 7 10 5
   Encoding = 3 5 7 10 1 3 1 1

2) Fitness = 185.0 Generation created = 0 Order = 1 Permutation = 10 9 4 5
   Encoding = 4 5 9 10 3 3 2 1

3) Fitness = 191.0 Generation created = 0 Order = 3 Permutation = 8 10 2 5
   Encoding = 2 5 8 10 3 3 1 1

4) Fitness = 222.0 Generation created = 0 Order = 2 Permutation = 1 7 10 5
   Encoding = 1 5 7 10 1 3 1 1

5) Fitness = 244.0 Generation created = 0 Order = 5 Permutation = 6 9 1 10
   Encoding = 1 6 9 10 3 1 1 1

Now that five (the population size) parents are created , five offspring will randomly be created and added to the population as shown below.

1) Combined member 4 and member 3
   Crossover begins at position 3 and ends at position 4
   Encoding = 1 5 7 10 1 3 1 1 Fitness = 222.0 Permutation = 1 7 10 5
   Encoding = 2 5 8 10 3 3 1 1 Fitness = 191.0 Permutation = 8 10 2 5

   Encoding = 1 5 8 10 1 3 1 1 Fitness = 201.0 Permutation = 1 8 10 5 Offspring # 1

2) Combined member 2 and member 3
   Crossover begins at position 2 and ends at position 3
   Encoding = 4 5 9 10 3 3 2 1 Fitness = 185.0 Permutation = 10 9 4 5
   Encoding = 2 5 8 10 3 3 1 1 Fitness = 191.0 Permutation = 8 10 2 5

   Encoding = 4 5 8 10 3 3 2 1 Fitness = 145.0 Permutation = 10 8 4 5 Offspring # 2

3) Combined member 4 and member 1
   Crossover begins at position 1 and ends at position 3
   Encoding = 2 5 8 10 3 3 1 1 Fitness = 191.0 Permutation = 8 10 2 5
   Encoding = 4 5 8 10 3 3 2 1 Fitness = 145.0 Permutation = 10 8 4 5

   Encoding = 4 5 8 10 3 3 1 1 Fitness = 137.0 Permutation = 8 10 4 5 Offspring # 3

4) Combined member 3 and member 4

Crossover begins at position 3 and ends at position 4
Encoding = 3 5 7 10 1 3 1 1 Fitness = 171.0 Permutation = 3 7 10 5
Encoding = 4 5 9 10 3 3 2 1 Fitness = 185.0 Permutation = 10 9 4 5

Encoding = 3 5 9 10 1 3 1 1 Fitness = 190.0 Permutation = 3 9 10 5 Offspring # 4

5) Mutate member 1

2 Mutations will occur at positions 3 and 6
The new values for these new positions are 7 and 3 (and follow encoding rules)
Encoding = 4 5 8 10 3 3 1 1 Fitness = 137.0 Permutation = 8 10 4 5

Encoding = 4 5 7 10 3 3 1 1 Fitness = 160.0 Permutation = 7 10 4 5 Offspring # 5

Now these new five members are added to the original five parents to make a generation size of ten. They will be sorted (they are actual sorted using insertion sort in the code).

1) Fitness = 137.0 Generation created = 1 Order = 3 Permutation = 8 10 4 5
   Encoding = 4 5 8 10 3 3 1 1

2) Fitness = 145.0 Generation created = 1 Order = 2 Permutation = 10 8 4 5
   Encoding = 4 5 8 10 3 3 2 1

3) Fitness = 160.0 Generation created = 1 Order = 5 Permutation = 7 10 4 5
   Encoding = 4 5 7 10 3 3 1 1

4) Fitness = 171.0 Generation created = 0 Order = 4 Permutation = 3 7 10 5
   Encoding = 3 5 7 10 1 3 1 1

5) Fitness = 185.0 Generation created = 0 Order = 1 Permutation = 10 9 4 5
   Encoding = 4 5 9 10 3 3 2 1

______ Generation will be grimed from here to end ______

6) Fitness = 190.0 Generation created = 1 Order = 4 Permutation = 3 9 10 5
   Encoding = 3 5 9 10 1 3 1 1

7) Fitness = 191.0 Generation created = 0 Order = 3 Permutation = 8 10 2 5
   Encoding = 2 5 8 10 3 3 1 1

8) Fitness = 201.0 Generation created = 1 Order = 1 Permutation = 1 8 10 5
   Encoding = 1 5 8 10 1 3 1 1

9) Fitness = 222.0 Generation created = 0 Order = 2 Permutation = 1 7 10 5
   Encoding = 1 5 7 10 1 3 1 1

10) Fitness = 244.0 Generation created = 0 Order = 5 Permutation = 6 9 1 10
   Encoding = 1 6 9 10 3 1 1 1

Finally the five least fit members will be grimed leaving the population size of five. Remember the lower the cost the higher the fitness. This is the end of the first generation. They are listed again below as final five parents for generation one.

1) Fitness = 137.0 Generation created = 1 Order = 3 Permutation = 8 10 4 5
   Encoding = 4 5 8 10 3 3 1 1

2) Fitness = 145.0 Generation created = 1 Order = 2 Permutation = 10 8 4 5
   Encoding = 4 5 8 10 3 3 2 1

3) Fitness = 160.0 Generation created = 1 Order = 5 Permutation = 7 10 4 5
   Encoding = 4 5 7 10 3 3 1 1

4) Fitness = 171.0 Generation created = 0 Order = 4 Permutation = 3 7 10 5
   Encoding = 3 5 7 10 1 3 1 1

5) Fitness = 185.0 Generation created = 0 Order = 1 Permutation = 10 9 4 5
   Encoding = 4 5 9 10 3 3 2 1

Note that there were three offspring and two parents (labeled Generation created = 1) which were found to be more fit than the others and remained for the next generation. Now we reiterate the process again, five new members are added to the original five parents from the beginning of generation 1 to make a generation size of ten. They will be sorted (they are actual sorted using insertion sort in the code).

These five new offspring will be created as shown below.

1) Combined member 4 and member 2

Crossover begins at position 1 and ends at position 2
Encoding = 3 5 7 10 1 3 1 1 Fitness = 171.0 Permutation = 3 7 10 5
Encoding = 4 5 8 10 3 3 2 1 Fitness = 145.0 Permutation = 10 8 4 5

Encoding = 4 5 7 10 1 3 1 1 Fitness = 152.0 Permutation = 4 7 10 5

2) Combined member 4 and member 1

Crossover begins at position 1 and ends at position 2
Encoding = 3 5 7 10 1 3 1 1 Fitness = 171.0 Permutation = 3 7 10 5
Encoding = 4 5 8 10 3 3 1 1 Fitness = 137.0 Permutation = 8 10 4 5

Encoding = 4 5 7 10 1 3 1 1 Fitness = 152.0 Permutation = 4 7 10 5

3) Combined member 1 and member 2

Crossover begins at position 1 and ends at position 3
Encoding = 4 5 8 10 3 3 1 1 Fitness = 137.0 Permutation = 8 10 4 5
Encoding = 4 5 8 10 3 3 2 1 Fitness = 145.0 Permutation = 10 8 4 5

Encoding = 4 5 8 10 3 3 1 1 Fitness = 137.0 Permutation = 8 10 4 5

4) Combined member 1 and member 8

Crossover begins at position 2 and ends at position 3
Encoding = 4 5 8 10 3 3 1 1 Fitness = 137.0 Permutation = 8 10 4 5
Encoding = 4 5 7 10 3 3 1 1 Fitness = 160.0 Permutation = 7 10 4 5

Encoding = 4 5 7 10 3 3 1 1 Fitness = 160.0 Permutation = 7 10 4 5

5) Mutate member 3

2 Mutations will happen at positions 6 and 7
The new values for these new positions are 1 and 1
Encoding = 4 5 8 10 3 3 2 1 Fitness = 145.0 Permutation = 10 8 4 5

Encoding = 4 5 8 10 3 1 1 1 Fitness = 118.0 Permutation = 5 8 4 10

Note that this last mutation a best answer of 5,8,4,10 was found. The program would sort the ten members and grim the bottom five. It would then compare the fitness of the top value (in this case 118 previously found by the brute force program) and stop the genetic algorithm program. In this example it took the completion of two generations to find a

best (that which equals the brute force) value. The ten resulting members will be listed below.

1) Fitness = 118.0 Generation created = 2 Order = 5 Permutation = 5 8 4 10
   Encoding = 4 5 8 10 3 1 1 1

2) Fitness = 137.0 Generation created = 1 Order = 3 Permutation = 8 10 4 5
   Encoding = 4 5 8 10 3 3 1 1

3) Fitness = 137.0 Generation created = 2 Order = 3 Permutation = 8 10 4 5
   Encoding = 4 5 8 10 3 3 1 1

4) Fitness = 145.0 Generation created = 1 Order = 2 Permutation = 10 8 4 5
   Encoding = 4 5 8 10 3 3 2 1

5) Fitness = 152.0 Generation created = 2 Order = 1 Permutation = 4 7 10 5
   Encoding = 4 5 7 10 1 3 1 1

______Generation will be grimed from here to end______

6) Fitness = 152.0 Generation created = 2 Order = 2 Permutation = 4 7 10 5
   Encoding = 4 5 7 10 1 3 1 1

7) Fitness = 160.0 Generation created = 1 Order = 5 Permutation = 7 10 4 5
   Encoding = 4 5 7 10 3 3 1 1

8) Fitness = 160.0 Generation created = 2 Order = 4 Permutation = 7 10 4 5
   Encoding = 4 5 7 10 3 3 1 1

9) Fitness = 171.0 Generation created = 0 Order = 4 Permutation = 3 7 10 5
   Encoding = 3 5 7 10 1 3 1 1

10) Fitness = 185.0 Generation created = 0 Order = 1 Permutation = 10 9 4 5
   Encoding = 4 5 9 10 3 3 2 1

The final five parents for generation two are listed below.

1) Fitness = 118.0 Generation created = 2 Order = 5 Permutation = 5 8 4 10
   Encoding = 4 5 8 10 3 1 1 1

2) Fitness = 137.0 Generation created = 1 Order = 3 Permutation = 8 10 4 5
   Encoding = 4 5 8 10 3 3 1 1

3) Fitness = 137.0 Generation created = 2 Order = 3 Permutation = 8 10 4 5
   Encoding = 4 5 8 10 3 3 1 1

4) Fitness = 145.0 Generation created = 1 Order = 2 Permutation = 10 8 4 5
   Encoding = 4 5 8 10 3 3 2 1

5) Fitness = 152.0 Generation created = 2 Order = 1 Permutation = 4 7 10 5
   Encoding = 4 5 7 10 1 3 1 1

It is important to note a few things from the example above that will generally appear in the genetic algorithms implemented for this dissertation.

1. Whenever a crossover occurred it always involved two members. The two members are picked at random and can even be the same. The starting position is a random number between 1 and (S –1). The ending position is always a random number between (1 + starting position) and S. The rules for proper encoding discussed above always are followed.
2. Whenever a mutation occurs it always involves only one member. The member is picked at random. The number of positions that get picked for a mutation is a random number between 1 and S. The positions are picked at random and the new values are picked randomly but again the rules for proper encoding are always followed.
3. Only in generation 0 do we start with unique members. A check is made to ensure that no members are exactly alike. For all proceeding generations that follow, duplicate members are allowed.
4. The genetic algorithm is terminated when either of the following conditions occurs. First, after the completion of each generation, the fitness value of the most fit member is compared to the brute force value. If it equals the brute force value the genetic

algorithm is stopped. Second, after the completion of each generation, the fitness value of the most fit member is compared to the fitness value of the most fit member from the previous generation. If the are the equal, the previous generation number is recorded (this will be called the adjusted value) and the program also begins tracking the fitness value of the most fit member. If the fitness value remains constant for 50 generations, the genetic algorithm is stopped. The final generation number is also recorded (this will be called the unadjusted value).

5. Generation 0 was created by selecting random members did not have any mutations or crossovers occur in its creation. Generation 1 and succeeding generations have actually crossovers and mutations. For this reason when, for example (as in our example above), it is indicated that the best result was found in 2 generations, we have two generations involving crossover and mutations.

Now that an example has been given, it will hopefully become clearer how the results were calculated and tabulated.

3.2: Presentation of the Results.

The example above represents one actual run. For every factor measured in this dissertation, 30 different runs were preformed. The means of the 30 runs are presented so that a comparison can be made as to whether that factor has an effect when compared to other factors. This is done by comparing the mean of one factor to the mean of another. In most cases, an examination of one case from multiple factors will be presented at once.

This is done to reveal whether varying one factor a certain way will actually improve performance even when the second factor is varied.

Figure 3.4 is a plot of the total cost at each generation for the above example. Notice that a line is drawn representing the best (determined from brute force comparison of all the possible permutations) value. As seen in the figure, the total cost came closer to the best value after each generation. Finally, after generation two the best value was obtained. In this particular example the best value was attained. Solution quality for the result that was obtained was the best it could be. This means that the result from the genetic algorithm matched that of one determined by brute force comparison.

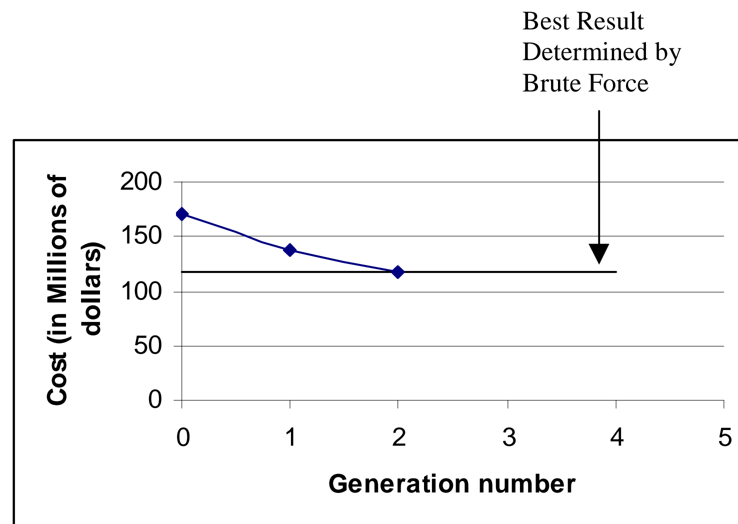

Figure 3.4: Comparison of Cost of Most Fit Result Per Generation for the Example Problem

With many of the runs however, solution quality is not the best it could be. A phenomenon that occurs in Figure 3.5 is observed. In the example shown in Figure 3.5, the solution comes close to obtaining the best solution but never actually attains it. After

every generation, the top member is evaluated and the fitness value does not change. In this example when the fitness value of the best member becomes 122 it remains constant for many generations. It must obtain a value of 118 for the genetic algorithm to stop using the brute force criteria. For all the studies implemented in this dissertation, if the of the fitness value of the best solution remains constant for 50 generations, the genetic algorithm is stopped.

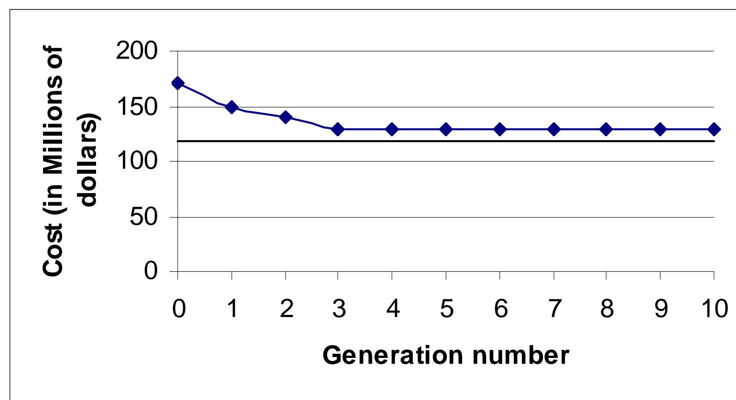

Figure 3.5: Comparison of Cost of Most Fit Result Per Generation Where the Best Result Is Not Obtained

In one case, the genetic algorithm was allowed to run a longer duration. This is one for a random cost table, population size of 50, generation size of 100 and 10 percent mutation rate. The genetic algorithm went on for over 4000 generations before it obtained the best possible result. One possible explanation is that the intial population did not have the proper building blocks to reach the exact solution quickly. It needed the prescence of the right mutation to supply the building blocks needed to obtain the best result.

As stated above, a comparison is done by taking the mean of 30 runs. If the genetic algorithms were allowed to continue until they stopped, the 4000 generation solution would be calculated in with, for example, 29 other results with values closer to the example above of 2 generations. Table 3.3 contains an example of 30 results from 30 runs.

Run #	# of Generations to complete	Run #	# of Generations to complete	Run #	# of Generations to complete
1	6	11	4	21	3
2	5	12	3	22	5
3	4	13	6	23	2
4	6	14	4	24	6
5	3	15	2	25	7
6	8	16	5	26	4
7	5	17	4	27	5
8	3	18	3	28	3
9	2	19	6	29	6
10	4	20	5	30	1

Table 3.3: Example Collection of 30 Runs

The mean of the 30 runs would be 4.33 generations. Now if the value the 30th run was changed to 4000 generations instead of 1 the mean would jump to 137.6 generations. As can be seen this would skew the results considerably and not present an accurate picture. In our study, the genetic algorithms were only allowed to run for 50 generations without any change before they were stopped. Even this could present some bias. Let us look at a second set of numbers in Table 3.4.

Run #	# of Generations to complete	Run #	# of Generations to complete	Run #	# of Generations to complete
1	6	11	4	21	3
2	5	12	3	22	5
3	4	13	6	23	2
4	6	14	4	24	6
5	3	15	2	25	7
6	8	16	5	26	4
7	5	17	4	27	55
8	3	18	3	28	53
9	2	19	6	29	56
10	4	20	5	30	51

Table 3.4: A Second Example Collection of 30 Runs

The results in Table 3.4 are simply the results taken from Table 3.3 with four values were changed to resemble a genetic algorithm that needed to be stopped after 50 runs of no improvement. Run number 27 was changed from 5 to55, run number 28 was changed from 3 to 53, run number 29 was changed from 6 to 56,and run number 30 was changed from 1 to 51, These simple four changes altered the mean from 4.33 to 11. So, there is a dilemma here. How can we report the mean of runs where the solution quality was not the same (because a run needed to be stopped) with that of perfect solution quality?

This was accounted for by making the following adjustment. The assumption was made that the lowest value that the genetic algorithm had obtained was good enough. This meant its value would be considered complete. Instead of using the generation number where the genetic algorithm was stopped as the result for that run, an adjustment was made. This adjustment uses a leveling-off paradigm . Figure 3.6 shows where leveling occurred on Figure 3.5. In that same example, the assumption would be made that 122 would be good enough because it had not changed for 50 generations. The adjustment

would be made by using the value of 3 generations, which is the first generation where the best member with a fitness value of 122 occurred, instead of 53 generations (the value when it stopped).

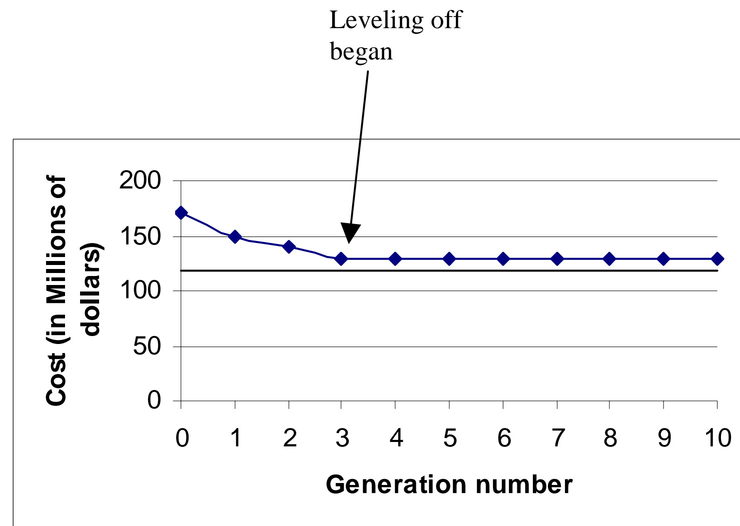

Figure 3.6: Comparison Of Cost Of Most Fit Result Per Generation Where The Best Result Is Not Obtained Showing Where Leveling Off Began

It is for this reason also that solution quality will always be measured as the number of runs out of 30 stopped. We know that if none of the 30 runs were stopped, all the runs have reached a best answer. If 10 of the runs were stopped this implies that only 20 runs have reached the best answer. A basis for comparison has now been established. A factor studied which has 20 runs stopped has a poorer solution quality than a different factor which has only 10 runs stopped. This is because in the second case 20 out of the 30 runs obtained the best answer while only 10 out of the 30 in the first case.

For all of the results, the adjusted means will be presented. Also, solution quality will be presented as the number of runs (out of 30) stopped. For many of the conditions, the

unadjusted results will be presented also. All of the results will be presented in a bar graph so that comparisons can be seen quite easily.

Also, in a few cases, a plot of the number of runs out of 30,which have not completed (Y axis) verses the generation number will be presented. This way of presenting the results gives a good comparison of how quickly (generation wise) one factor effecting genetic algorithms completes compared to another. This is especially useful in displaying changes.

In the next chapter, the results will be presented in the manner described above.

Chapter 4: Results

4.1: Results from Model 1

In this evaluation of results, a comparison of different factors will be considered concurrently. Most figures require no explanation. When a description of a figure is provided, the concentration will focus on noticeable distinctions. Solution quality will be assessed as the number of runs out of 30 that needed to be stopped. Runs are presented with and without adjustment as specified in the text. An adjustment is made when runs need to be stopped because the fitness of the most fit solution has not changed for at least 50 generations. An adjustment is made by recording the generation number when leveling begins to occur and not the generation number where the run was stopped. The adjustment is typically 50 generations. This allows results with lower solution quality to be compared to results with perfect solution quality (that is runs that did not need to be stopped). Means are based on the average of 30 runs. The following abbreviations are used in the subsequent figures. RA represents the random cost table. NI represents the non-increasing cost table. ND represents the non-decreasing cost table. MM represents the multi-modal cost table. RM represents the random multi-modal cost table.

Figure 4.1 compares the structure of the cost table and the size of the problem to the number of generations. All runs utilized a population size of 50, a generation size of 100 and a 10 percent mutation rate. As can be seen in the diagram, the data representing the structured cost tables (NI, ND, and MM) take fewer generations to complete compared to the data representing the random cost tables (RA and RM). As the complexity of the problem increases (larger s values), the number of generations also increases.

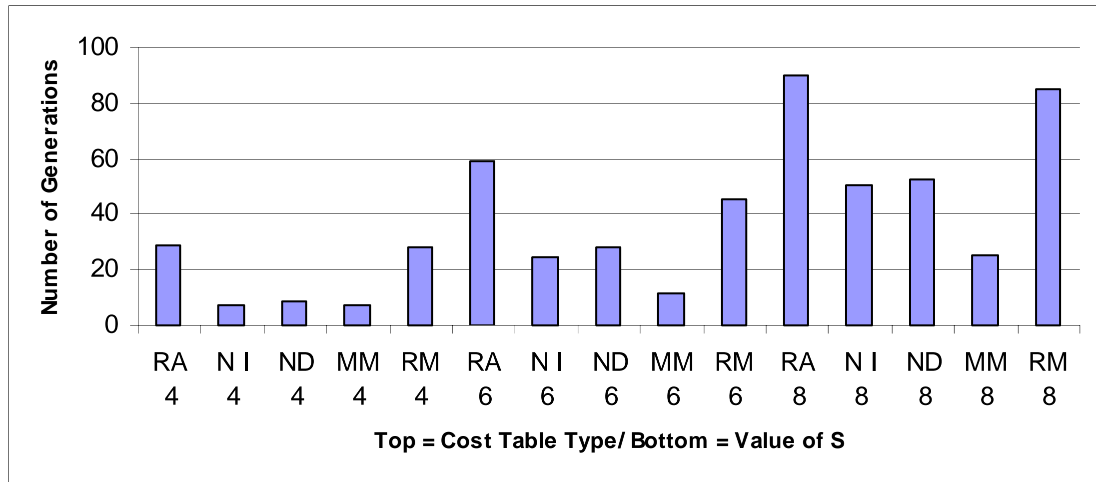

Figure 4.1: Comparison of Different Costs Table Structures and Different Size of Problems to Number of Generations for Completion (No Adjustment)

Figure 4.2 is a measure of the solution quality of the data presented in Figure 4.1. It compares the structure of the cost table and the size of the problem to the number of runs (out of 30) that needed to be stopped. The actual number of runs stopped is displayed on the top of each bar. The runs representing the structured cost tables (NI, ND, and MM) had fewer runs that needed to be stopped as compared to the random cost tables (RA and RM). Also, as the problem size increases the number of runs that need to be stopped also increases.

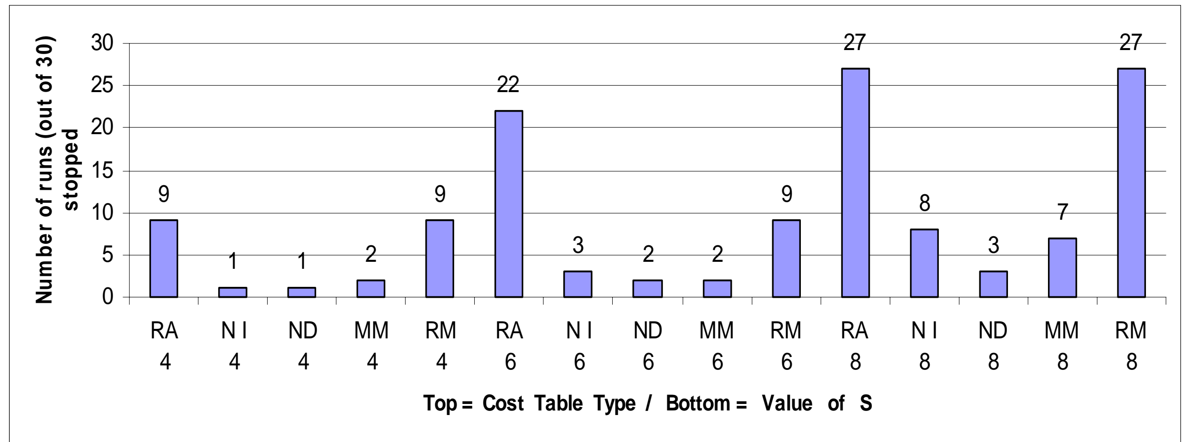

Figure 4.2: Comparison of Different Costs Table Structures and Different Size of Problems

The results for the random cost tables presented in Figure 4.1 may be skewed due to fact that the random cost tables appears to have a higher number of stopped runs. Figure 4.3 shows the same data after adjustments have been made. Notice first that the vertical axis has been reduced by about 50 generations. This is especially noticeable on the random cost tables. The same findings still hold true as the unadjusted results show in Figure 4.1, but for the larger problem size the difference in the NI and ND cost table is not as evident.

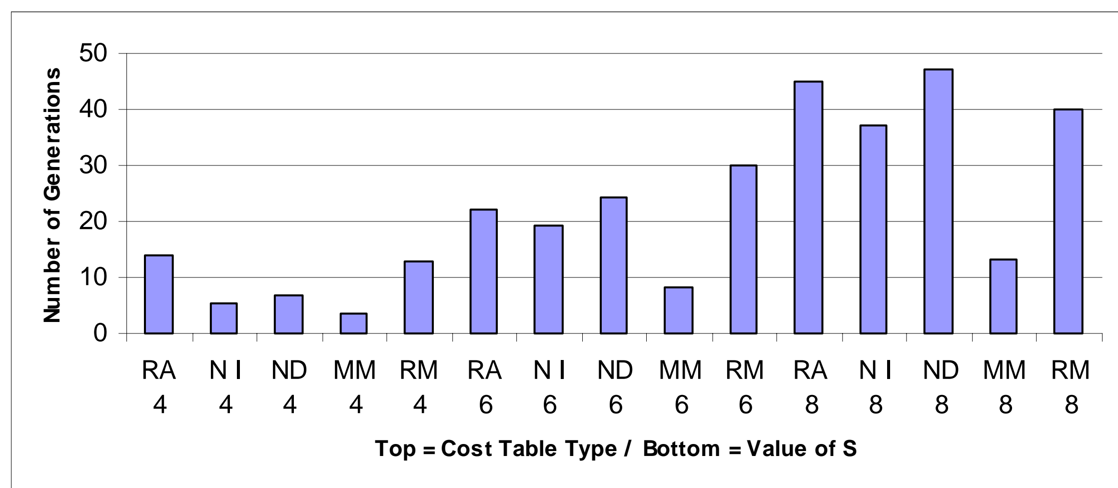

Figure 4.3: Comparison of Different Costs Table Structures and Different Size of Problems vs. Number of Generations for Completion

For the next experiment, solution quality is analyzed a bit differently. Figure 4.4 is a plot of data where S= 4 and represents the number of runs not completed per generation. Once again, the population size is 50, the generation size is 100 and the mutation rate is 10 percent. The runs representing the structured cost tables (NI, ND, and MM) come to completion more quickly than those of the random cost tables. As a result, there are a greater number of generations completed for structured cost tables when compared to random cost tables.

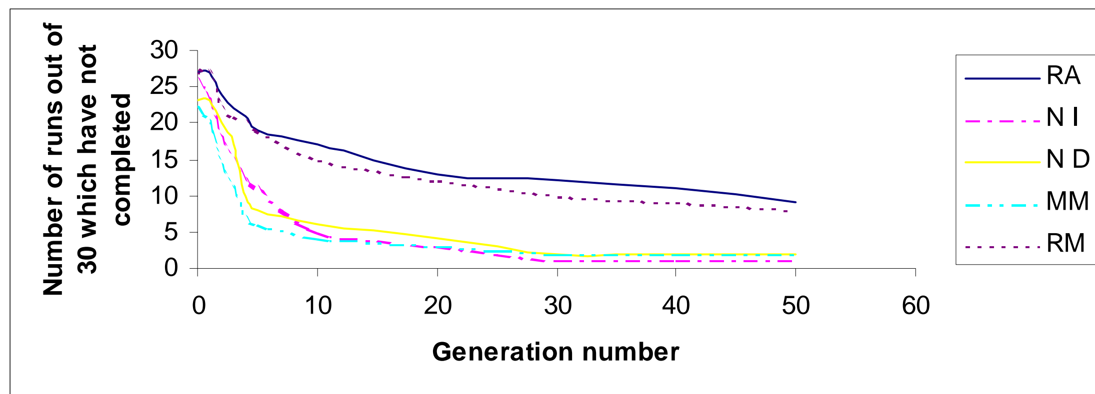

Figure 4.4: Number of Uncompleted Runs vs. Generation Number (s = 4)

The experimental analysis in Figure 4.5 is similar to that in Figure 4.4 except that a larger problem size (s = 6) is used. Notice that the runs representing the structured cost tables (NI, ND and MM) do not come to completion as quickly as they do when the problem size is smaller (s = 4). In fact, the number of runs that do not complete within 50 generations increases with increasing problem size. Increasing the problem size effects the genetic algorithm performance by requiring more generations to obtain an optimal answer.

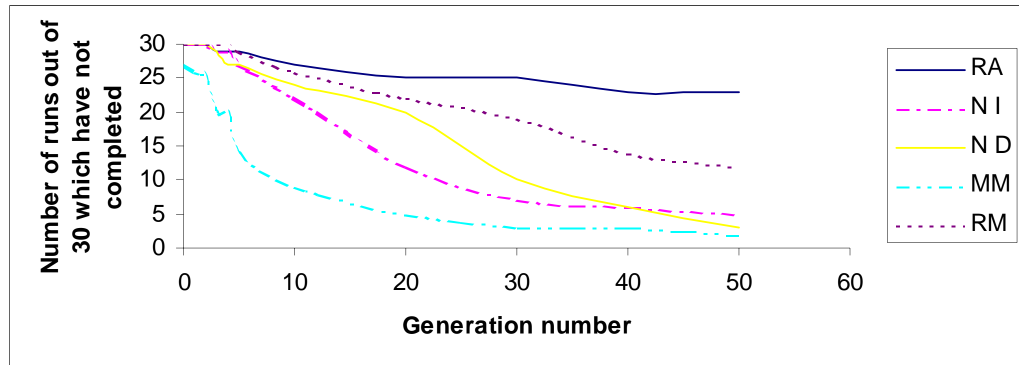

Figure 4.5: Number of Uncompleted Runs vs. Generation Number (*s* = 6)

This effect is even more pronounced as the problem size is increased further (*s* = 8). As can be seen in Figure 4.6, both the random and structured cost tables are affected.

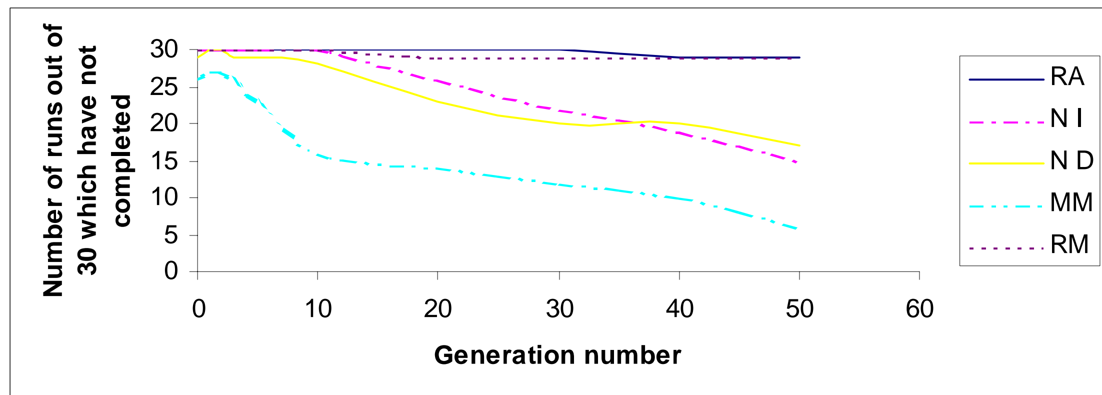

Figure 4.6: Number of Uncompleted Runs vs. Generation Number (*s* = 8)

For the preceding experiments, population size and percent mutation rate were been kept constant. Figure 4.7 shows the effect of varying the population size (at the beginning of each generation) and mutation rate on the number of generations to complete for a random cost table. In this experiment, the problem size is *s* = 4. When comparing the different population sizes (at the beginning of each generation), minor differences were

seen on the effect of the number of generations to complete. However, as the mutation rate increases from 10 to 30, there is a significant improvement in the number of generations needed to complete.

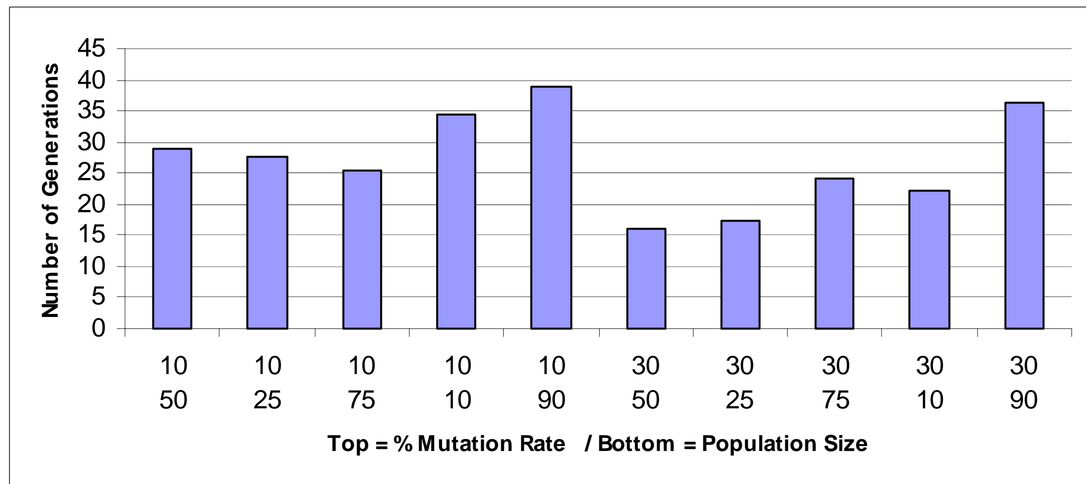

Figure 4.7: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Random Cost Table and *s* = 4 (No Adjustment)

An analysis of the solution quality for the data presented in Figure 4.7 are provided in Figure 4.8. It compares the mutation rate and population size to the number of runs (out of 30) that needed to be stopped. The actual number of runs stopped is displayed on the top of each bar. Once again, only minor differences were seen when comparing the different population sizes and increasing the mutation rate from 10 to 30 percent showed a small improvement.

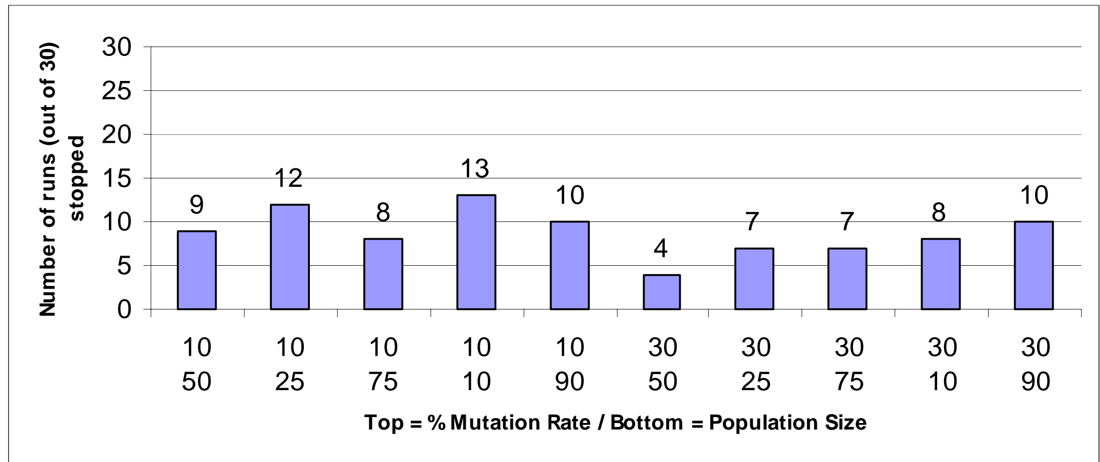

Figure 4.8: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using the Random Cost Table and s = 4

Adjustments were made (as before) to Figure 4.7 to account for runs that needed to be stopped. They are presented in Figure 4.9. Although the adjustment improved results, the overall findings in Figure 4.7 were for the most part duplicated here.

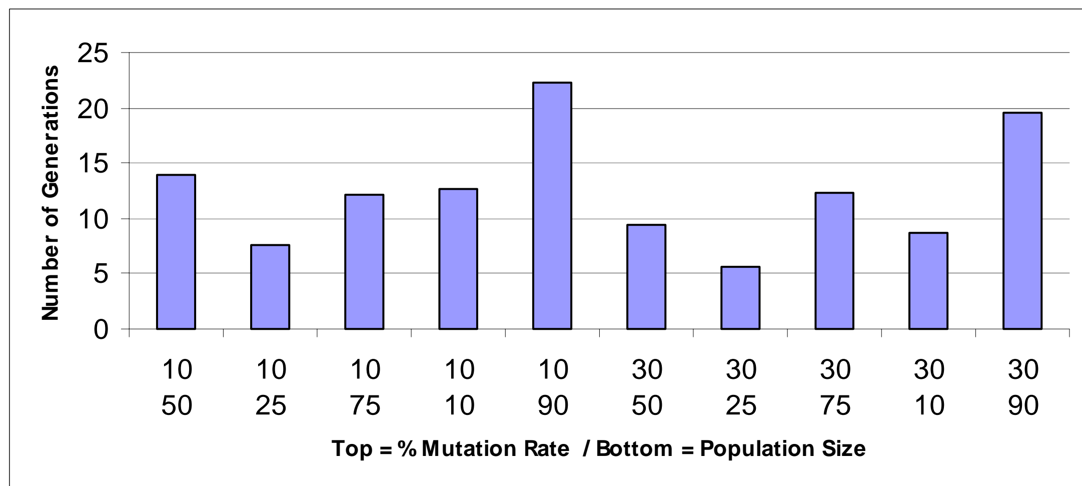

Figure 4.9: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Random Cost Table and s = 4 (With Adjustment)

Figure 4.10 demonstrates the effects of varying the size of the problem (*s*) and the population size on the number of generations needed to complete. Little or no difference is seen with varying the population size while more significant effect is seen when varying the size of the problem.

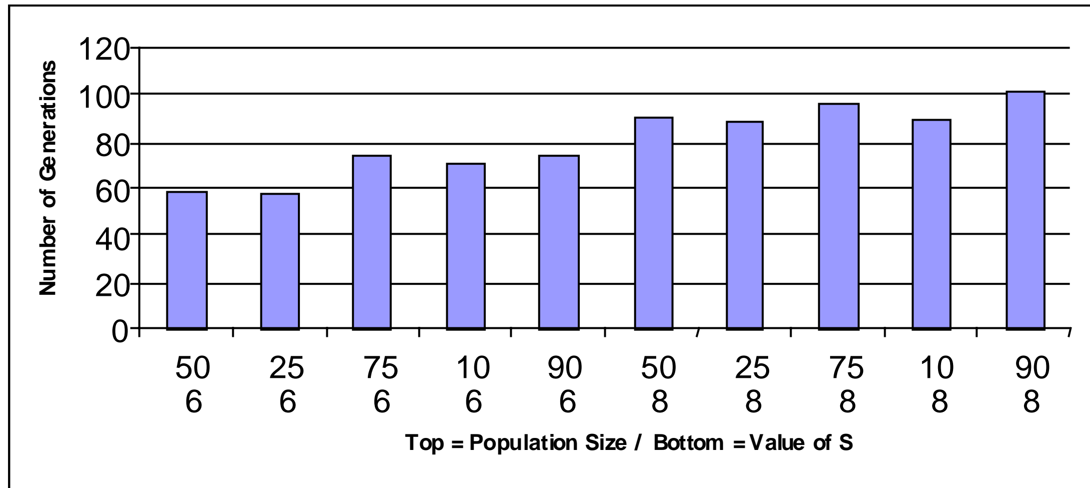

Figure 4.10: Comparison of Different Population Sizes and Problem Size to Number of Generations for Completion Using the Random Cost Table (No Adjustment)

An examination of the solution quality for the data displayed in Figure 4.10 is presented in Figure 4.11. The numbers above the bars represent the actual number of runs that needed to be stopped. As seen before, there is a greater effect when comparing the size of the problem but little effect when comparing the population size.

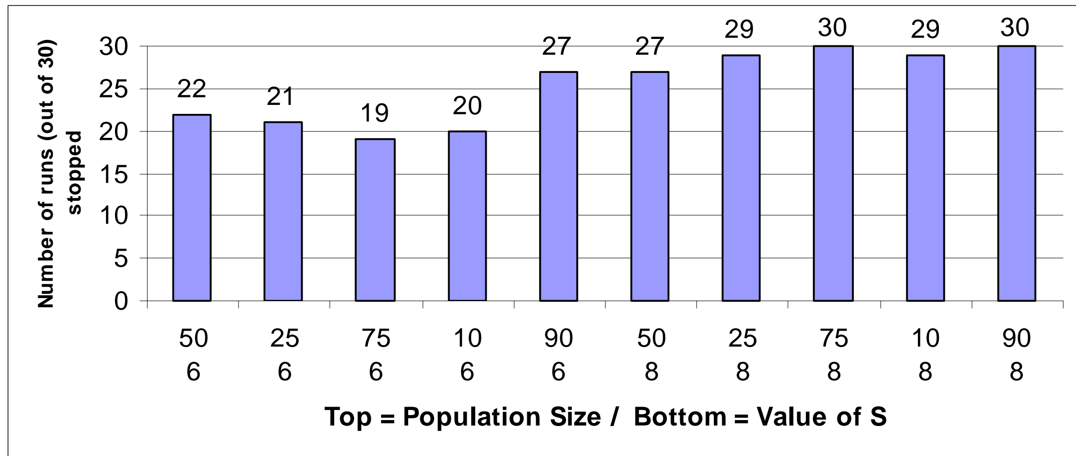

Figure 4.11: Comparison of Different Population Sizes and Problem Size to Number of Runs Stopped Before Completion Using the Random Cost Table

Adjustments were made (as before) to Figure 4.10 to account for runs that needed to be stopped. They are presented in Figure 4.12. Although the adjustment improved results overall, the findings in Figure 4.11 were for the most part duplicated here

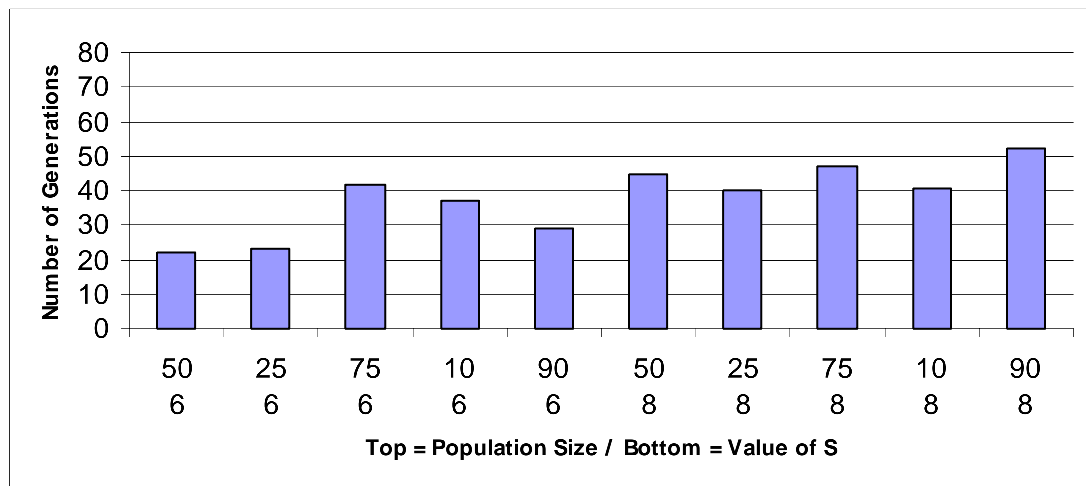

Figure 4.12: Comparison of Different Population Sizes and Problem Size to Number of Generations for Completion Using the Random Cost Table (With Adjustment)

The examination of varying the population size above was done using the random cost tables. The same examination will be done on the other types of cost tables. Since the adjusted tables show the same outcome as the non-adjusted tables, only the adjusted tables will be shown. Figure 4.13 shows the results of the number of generations of a non-increasing cost tables. This is the first example of a structured cost table. As can be seen the extreme case of a population size of 90 seems to have a negative effect on performance. Also not seen in the random cost table, population size 10 and 25 seem to have better performance than the other population sizes. As like the results of the random cost table, raising the mutation rate also increases the performance of the genetic algorithm.

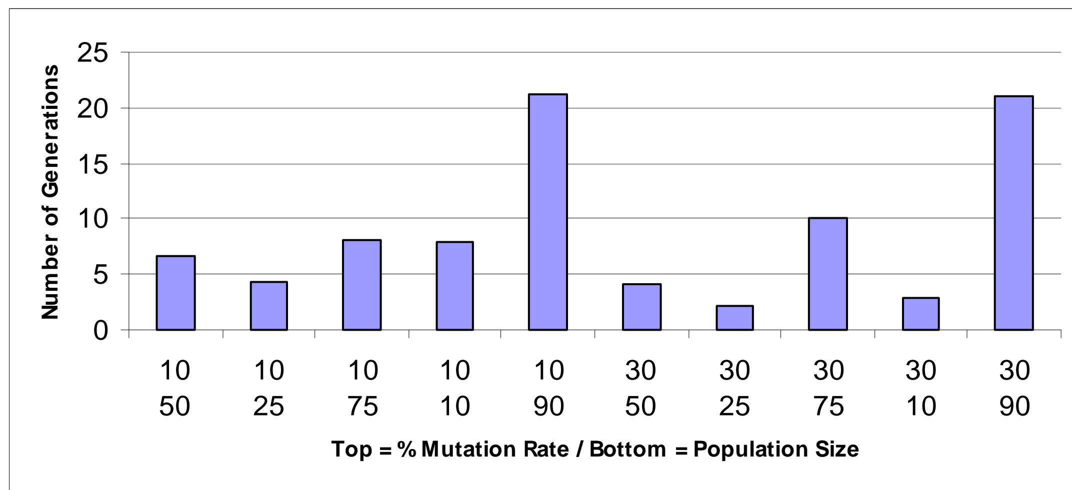

Figure 4.13: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Non-Increasing Cost Table s = 4 (With Adjustment)

Figure 4.14 displays the number of runs that needed to be stopped. Notice that the number of runs that needed to be stopped is less than that of the random cost table. Also, increasing the mutation rate eliminates the number of runs that needed to be stopped.

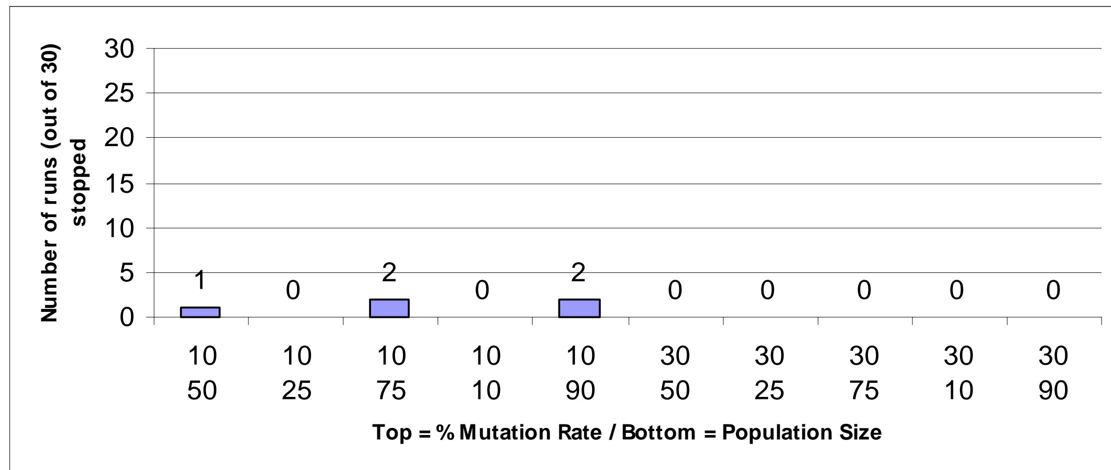

Figure 4.14: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using the Non- Increasing Cost Table and s = 4

Figure 4.15 displays the size of the problem and the population size. As with the random cost table, varying the size of the problem does have an effect on the performance of the problem. However in this case, varying the population size also has an effect on the performance. Population sizes 10 and 25 seem to improve performance.

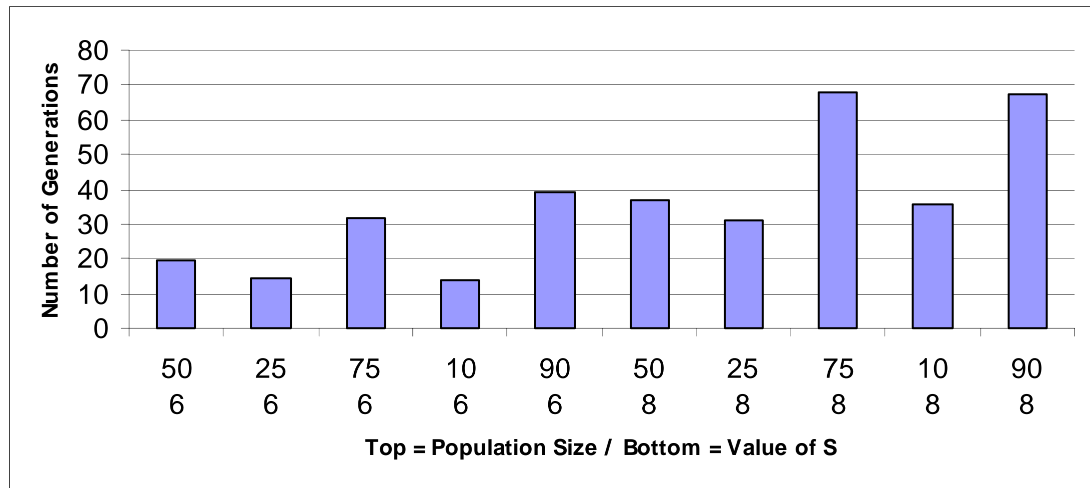

Figure 4.15: Comparison of Different Population Sizes and Size of the Problem to Number of Generations for Completion Using the Never Increasing Cost Table (With Adjustment)

An examination of the non-decreasing cost table in Figure 4.16 shows results similar to that of the non-increasing cost table. Population sizes 10 and 25 seem to improve performance and the extreme case of a population size of 90 seems to have a negative effect on performance.

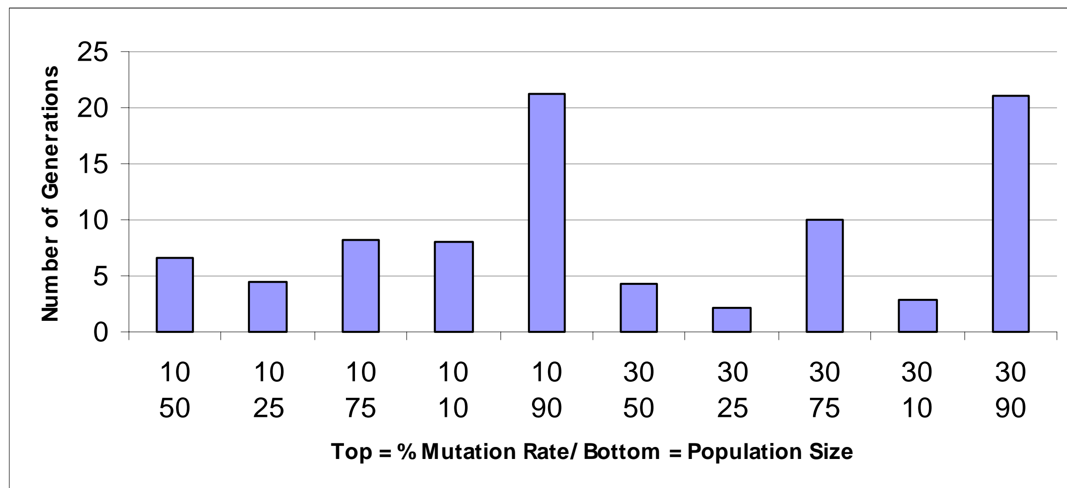

Figure 4.16: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Non-Decreasing Cost Table s = 4 (With Adjustment)

Figure 4.17 displays the number of runs that needed to be stopped for the non-decreasing cost table and is almost identical to Figure 4.14 that displays results for the non-increasing cost table.

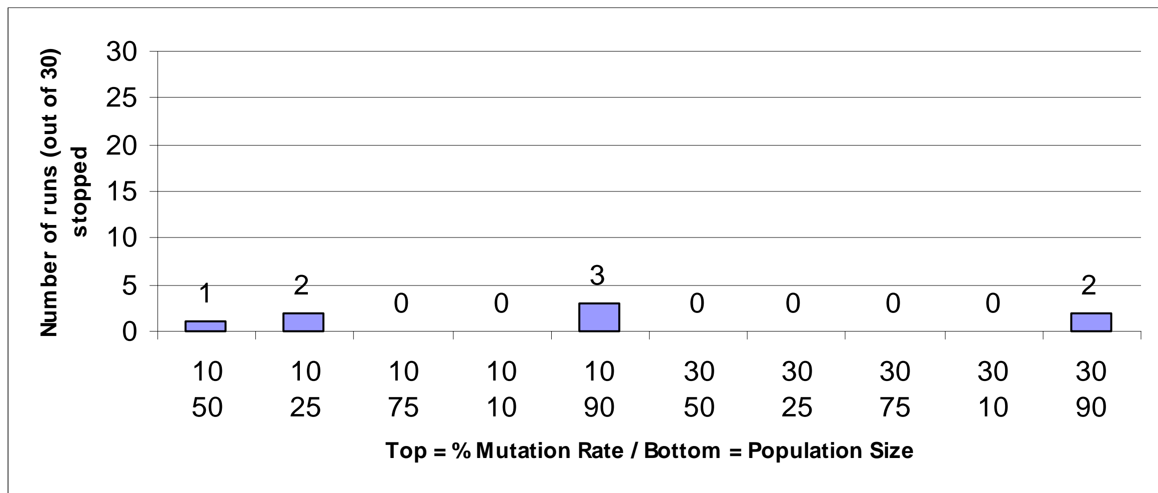

Figure 4.17: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using the Non-Decreasing Cost Table and *s* = 4

Figure 4.18 displays the size of the problem and the population size for the non-decreasing cost table vs. the number of generations to completion. A simular pattern is seen where as the problem size increases (s), the number of runs needed to complete also increases. Also, population sizes of 75 and 90 are clearly worse than the other population sizes.

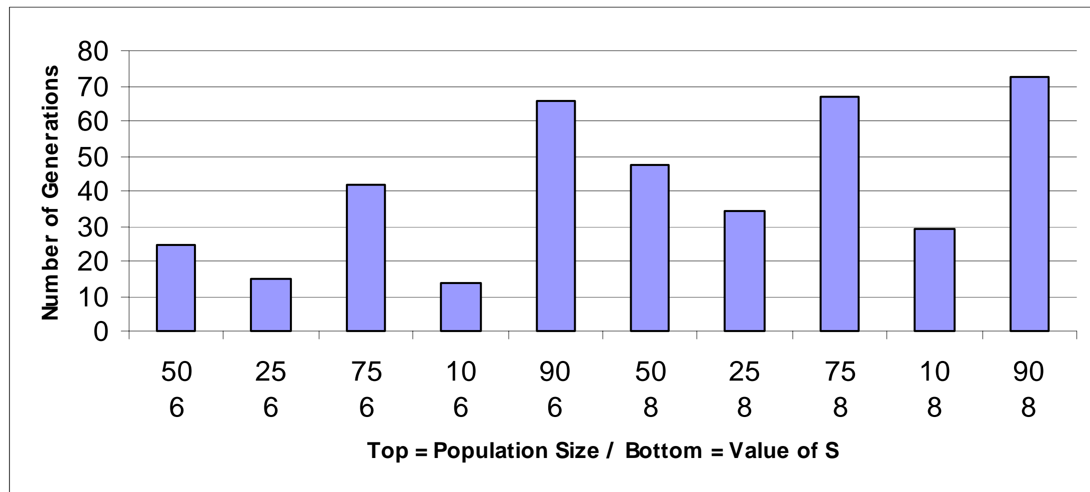

Figure 4.18: Comparison of Different Population Sizes and Size of the Problem to Number of Generations for Completion Using the Non-Decreasing Cost Table (With Adjustment)

Finally, the multi-modal cost table is examined. Figure 4.19 displays the results of varying population size and mutation rate on the multi-modal cost table. As can be seen, this cost table type performed the best out of all the other cost table structures.

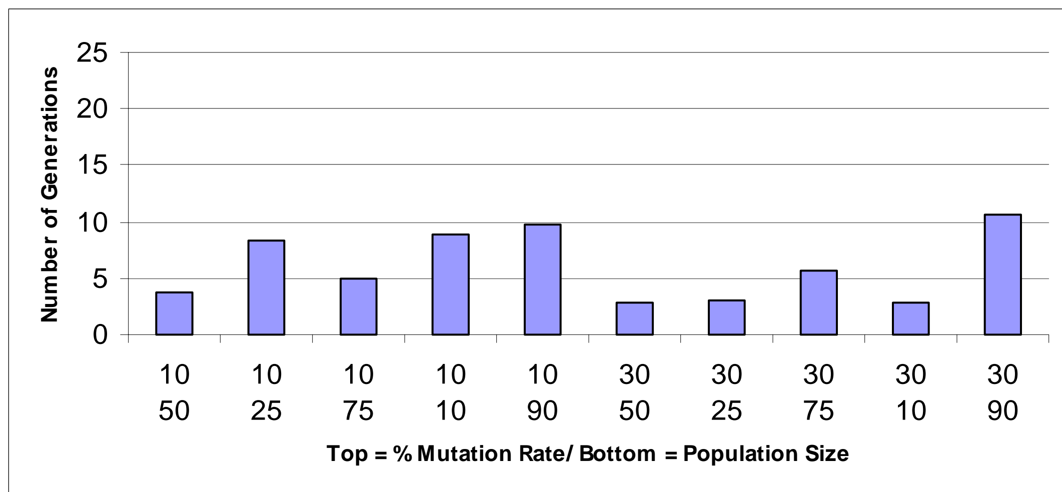

Figure 4.19: Comparison of Different Population Sizes and Different Mutation Rates to Number of Generations for Completion Using the Multi-modal Cost Table *s* = 4 (With Adjustment)

Figure 4.20 illustrates the number of runs that needed to be stopped with varying mutation rate and population size. The number of runs that needed to be stopped was considerably less than the other structured cost tables.

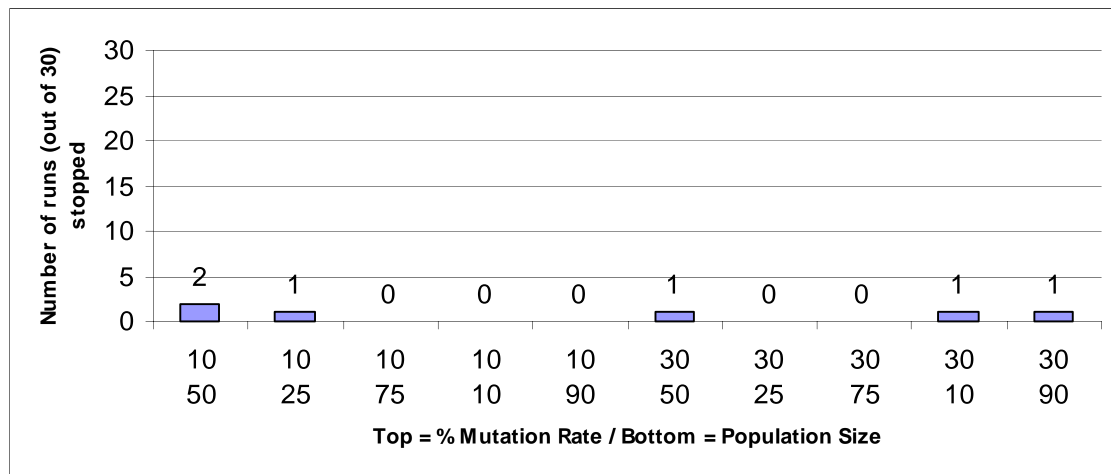

Figure 4.20: Comparison of Different Population Sizes and Different Mutation Rates to Number of Runs Stopped Before Completion Using The Multi-modal Cost Table and $s = 4$

Figure 4.21 displays the effect of varying different population sizes and problem sizes. For the multi-modal table, increasing the problem size from 6 to 8 did not seem to effect performance as it did in the other cost tables. Varying population size continues to have no effect on the number of generations.

Figure 4.21: Comparison of Different Population Sizes and Problem Size to Number of Generations for Completion Using the Multi-modal Cost Table (With Adjustment)

4.2: Results of Model 2

The presentation of results for the second model will begin with some of the same analyses performed on the first model that will allow for a comparison of the two models. An analysis of the different population sizes and mutation rates will be given as before. However, graphs will be summarized differently so that a more comprehensive look at slack time (which is unique to this model) can be studied.

Figure 4.22 compares the structure of the cost table along with the size of the problem. A population size of 50, generation size of 100 and a 10 percent mutation rate were used. The results for model 2 are similar to that of model 1. As can be seen in Figure 4.22, the data representing the structured cost tables (NI, ND, and MM) complete in fewer number of generations than does the data representing the random cost tables (RA and RM). As the complexity of the problem increases, the number of generations also increases.

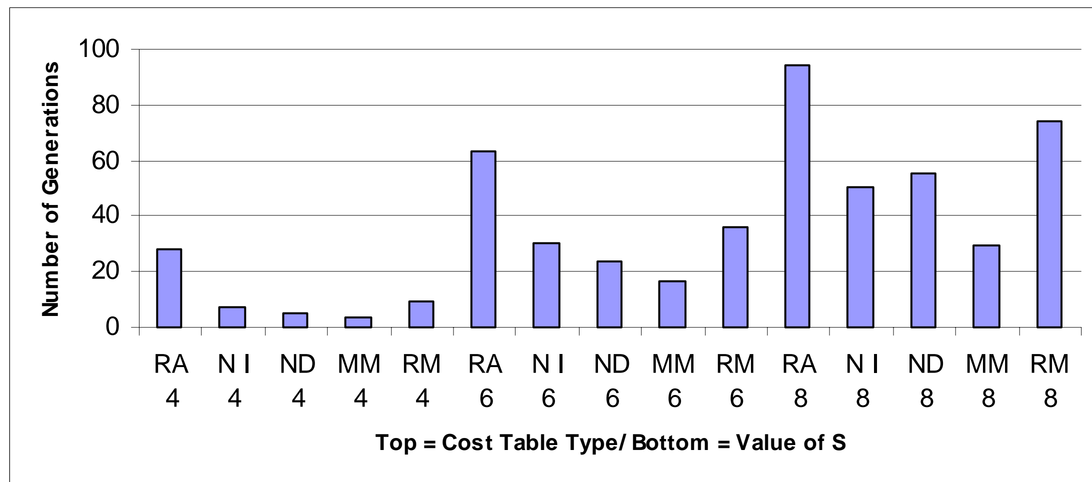

Figure 4.22: Comparison of Different Costs Table Structures and Different Size of Problems to Number of Generations for Completion Using Model 2 (No Adjustment)

Figure 4.23 measures the solution quality of the results presented in Figure 4.22. The numbers on top of the bars display the number of runs out of 30 that needed to be stopped. Once again, the runs representing the structured cost tables (NI, ND, and MM) had a fewer number of runs that needed to be stopped when compared to the random cost tables (RA and RM). Also, as the problem size increases the number of runs that need to be stopped also increases. Many similarities are seen between Figure 4.22 and Figure 4.23.

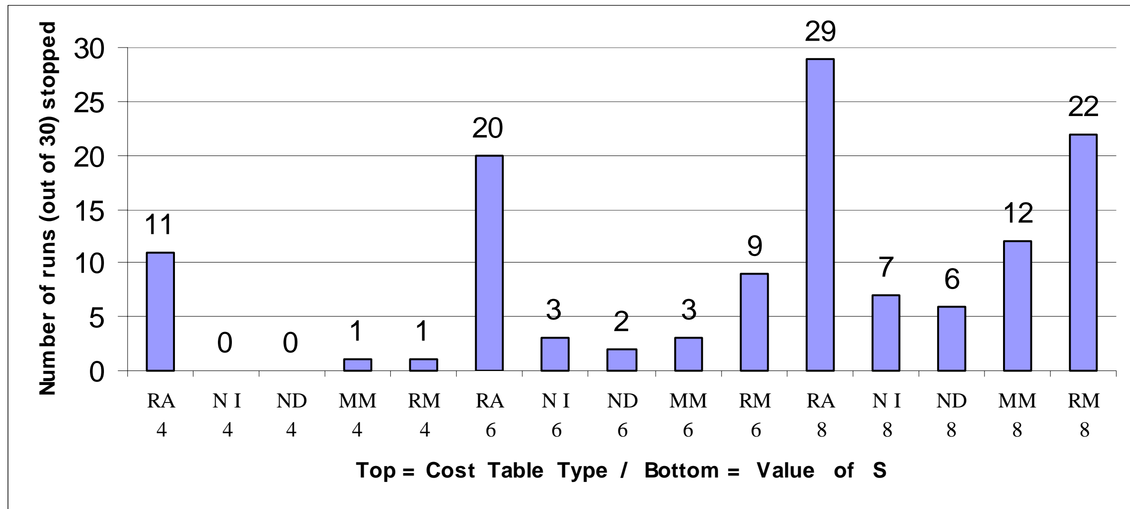

Figure 4.23: Comparison of Different Costs Table Structures and Different Size of Problems That Needed to Be Stopped in Model 2

As was done in Model 1, an inspection of quality was examined by plotting the number of runs that remained uncompleted as each generation progresses. Figure 4.24 displays these results. All data was collected using a population size at 50, a generation size of 100 and a mutation rate of 10 percent. Figure 4.24 plots data where *s* is 4. As evident by the way the curves on the graph seem to drop off more quickly, the runs representing the structured cost tables (NI, ND, and MM) seem to have a greater number of generations completed than the random cost tables (RA and RM).

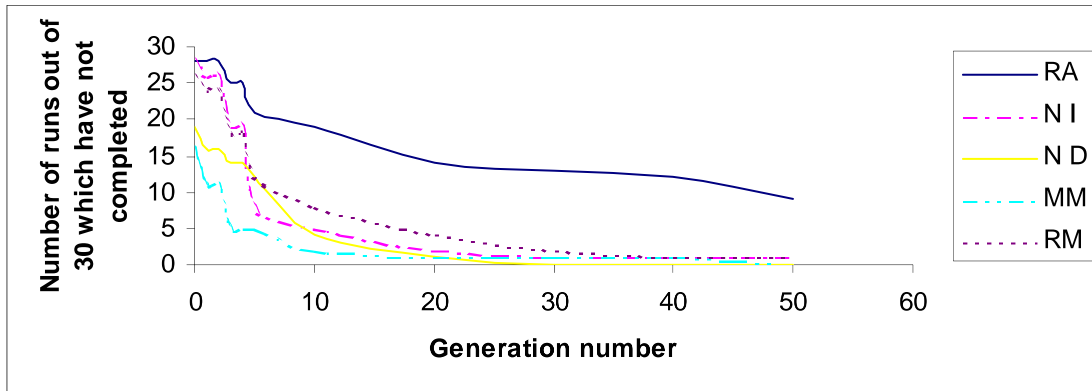

Figure 4:24: Number of Uncompleted Runs vs. Generation Number Using Model 2 (*s* = 4)

Figures 4.25 and 4.26 show the results where s is equal to 6 and 8, respectively.

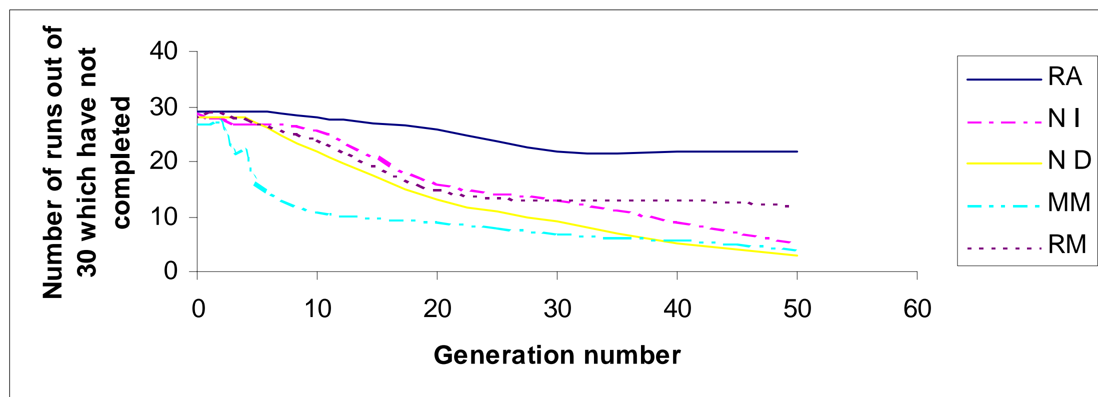

Figure 4:25: Number of Uncompleted Runs vs. Generation Number Using Model 2 (*s* = 6)

When comparing Figures 4.24, 4.25 and 4.26, it is evident that as problem size increases, fewer runs become completed per generation. This is evidenced by the fact that all the curves do not drop down as dramatically in Figure 4.25 as in Figure 4.24. Figure 4.26 shows this effect even more so as the number of generations increases from $s = 6$ to $s = 8$.

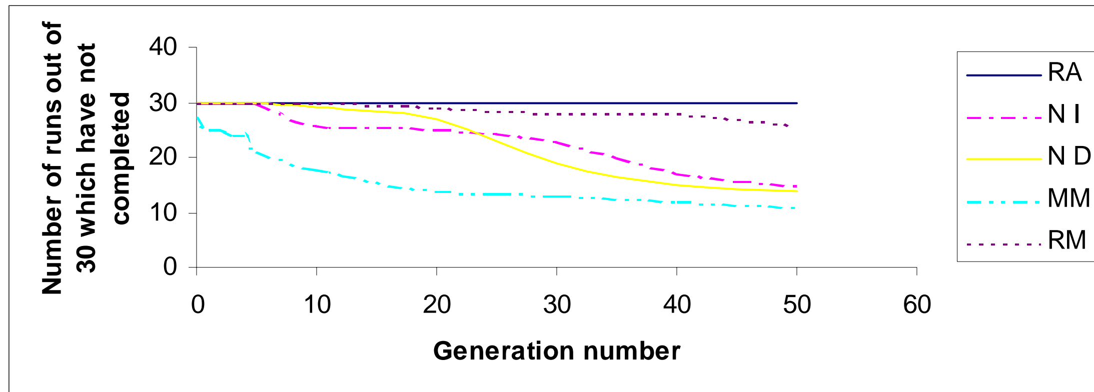

Figure 4:26: Number of Uncompleted Runs vs. Generation Number Using Model 2 (*s* = 8)

Figure 4.27 shows the comparison of different problem sizes, and different population and generation sizes. The data was collected using the random cost table. In this model an analysis of raising the generation size to 200 was studied. As can be seen, the common findings of increasing the problem size increases the number of generations needed to obtain the best result. Of interest here is that when the generation size was increased to 200, it generally reduced the number of generations needed to obtain the best result. Also, with regard to the generation size, when the population size was raised to 150 it always had a negative effect of increasing the number of generations needed to obtain a best result. One more interesting point worth mentioning. In some cases, it seems that changing the population size to 50 and the generation size to 200 seems to counteract the effect of problem size. The result for s equal to 8, population size equal to 50, and generation size equal to 200 requires fewer number of generations needed to obtain a best result then when *s* equal to 6, population size equal to 50, and generation size equal to 100. A discussion of increasing the generation size to 200 will continue in Chapter 5.

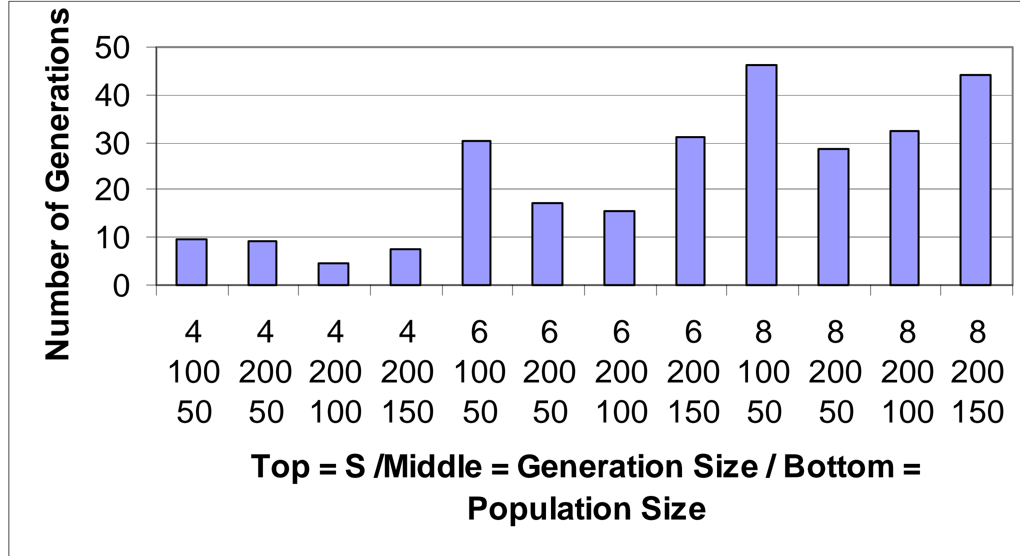

Figure 4.27: Comparison of Problem Size, Generation Size, and Population Size vs. Mean Number of Generations for Completion Using the Random Cost Table in Model 2. (With Adjustment)

Figure 2.27 examines the solution quality of the data displayed in Figure 4.26. The numbers on top of the bars display the actual value out of 30. Although the data may show improvement when inceasing population size with regards the number of generations needed to obtain the best result (when generation size is 200), it has little or no effect on solution quality. At the smaller problem size (*s*=4) some reduction in the number of runs which needed to be stopped is seen. As you increase the problem size (s = 6 and *s* = 8) the reduction is lessened or eliminated.

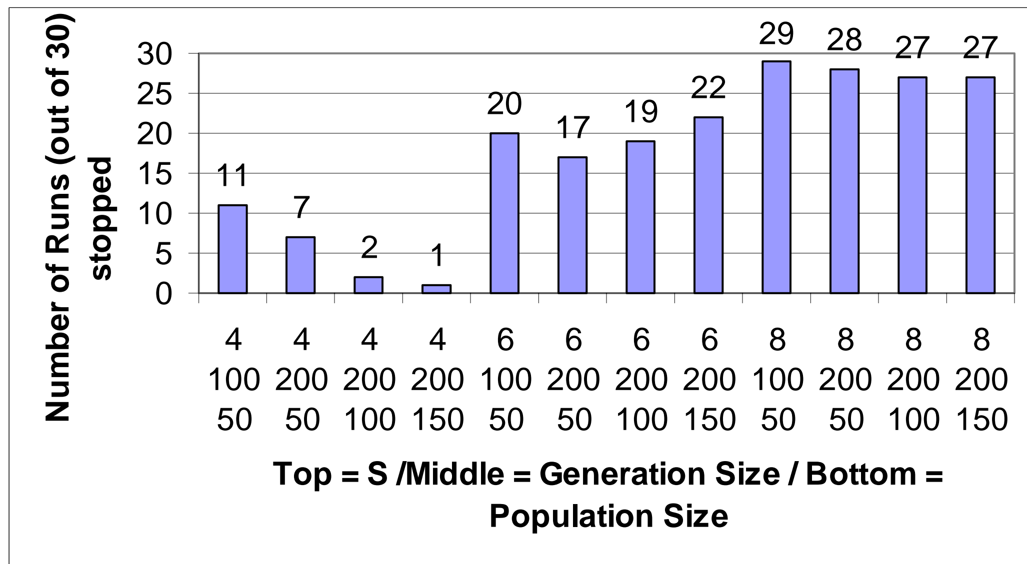

Figure 4.28: Comparison Of Problem Size, Generation Size, And Population Size That Needed To Be Stopped Using The Random Cost Table In Model 2

The results in Figure 4.27 and 4.28 were for a random cost table. Previous results indicate that a structured cost table performs better than a random cost table. For this reason Figure 4.29 and Figure 4.30 will perform the same analysis above on a non-decreasing cost table. When a structured cost table is used, the population size of 50 and generation size of 200 seems to have the least number of generations needed to obtain a best result. One more important finding, the population size of 150 and generation size of 200 performed the worst out of all the cases for all sizes of the problem.

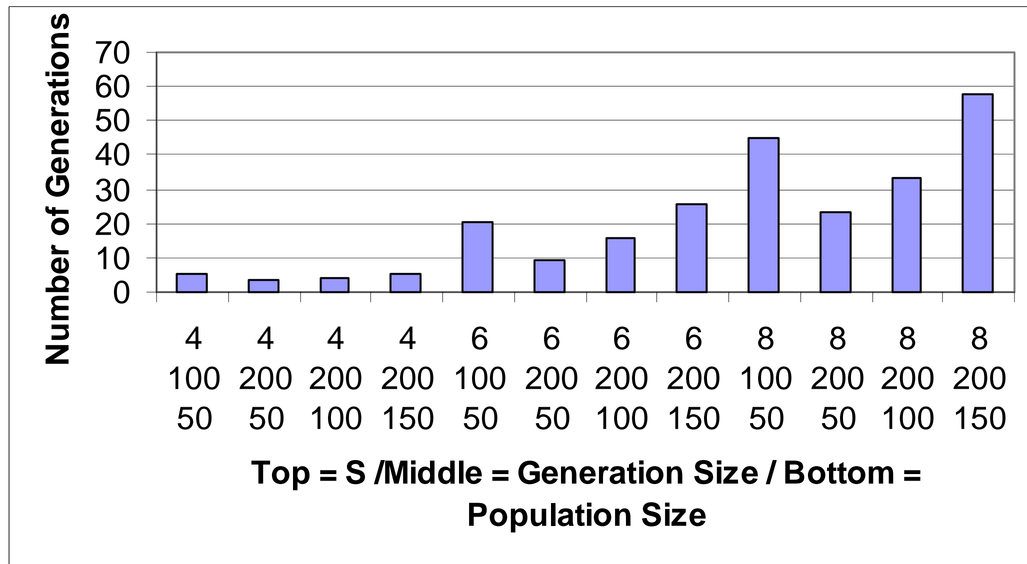

Figure 4.29: Comparison of Problem Size, Generation Size, and Population Size vs. Mean Number of Generations for Completion Using the Non-Decreasing Cost Table in Model 2 (With Adjustment)

Unfortunately, solution quality is quite good in a structured cost table (like the non-decreasing cost table). This makes it difficult to draw any strong conclusions from the data in Figure 4.30. The numbers on top of the bars in Figure 4.30 display the actual value out of 30.

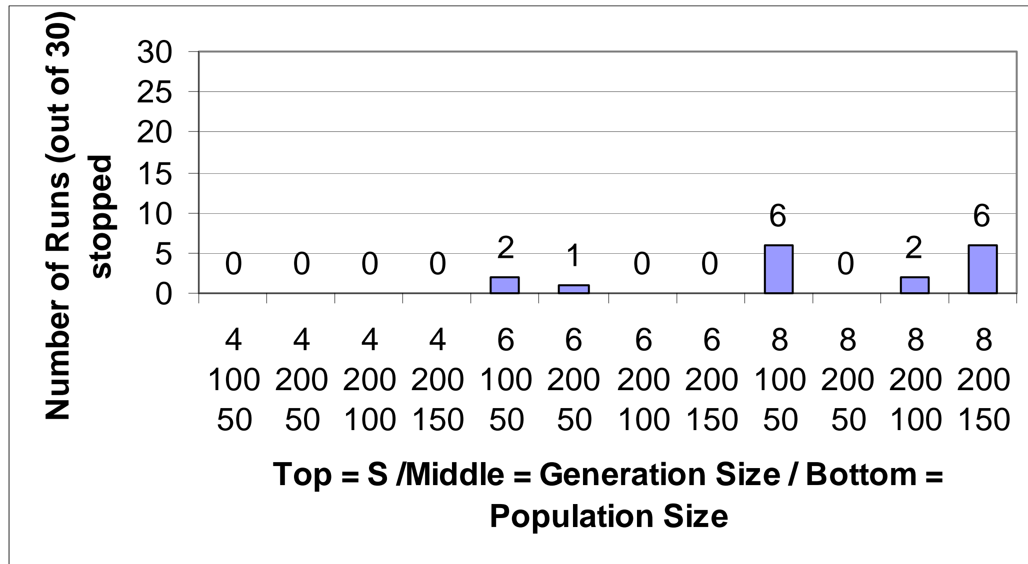

Figure 4.30: Graph of Comparison of Problem Size, Generation Size, and Population Size That Needed to Be Stopped Using the Non-Decreasing Cost Table in Model 2

An examination of percent mutation rate will now be performed on a random and structured cost table. Figure 4.31 displays these results for the random and non-increasing cost table. The data was collected using a population size of 50 and a generation size of 100. As can be seen, the 60 percent mutation rate in most cases tends to have the lowest number of generations needed to get a best result. Again, as problem size increases so does the number of generations needed to get a best result.

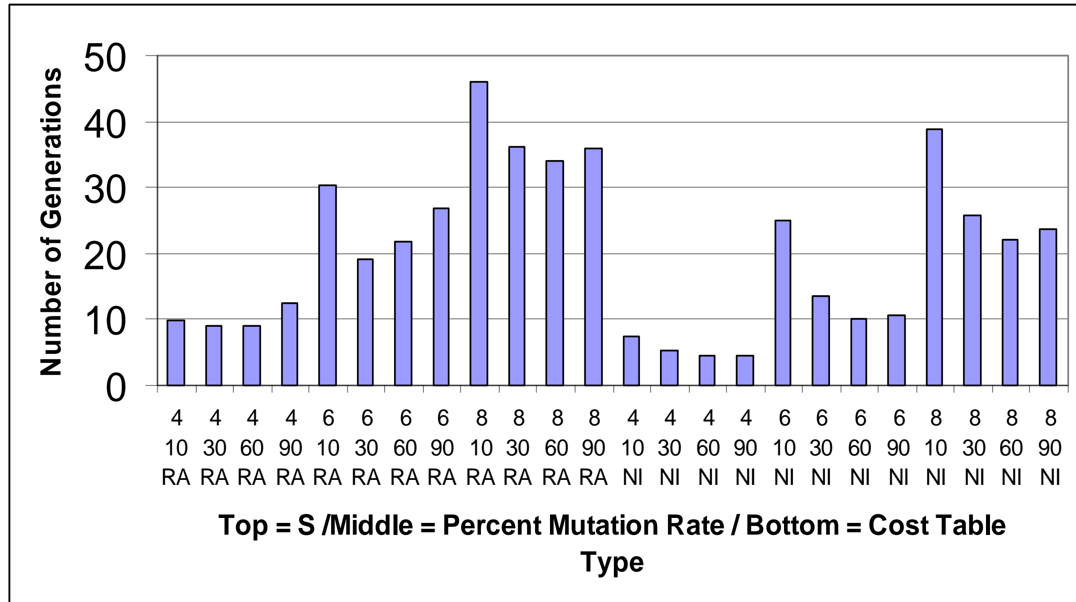

Figure 4.31: Comparison of Problem Size, and Mutation Rate for the Random and Non-Increasing Cost Tables vs. Mean Number of Generations for Completion in Model 2 (With Adjustment)

Figure 4.32 examines the same data as in Figure 4.31 for solution quality. The numbers on top of the bars display the actual value out of 30. The numbers on the graph represent the actual values. For the random cost table, as percent mutation rate increases the number of runs which need to be stopped decreases. No similar conclusion can really be drawn about the non-increasing cost table.

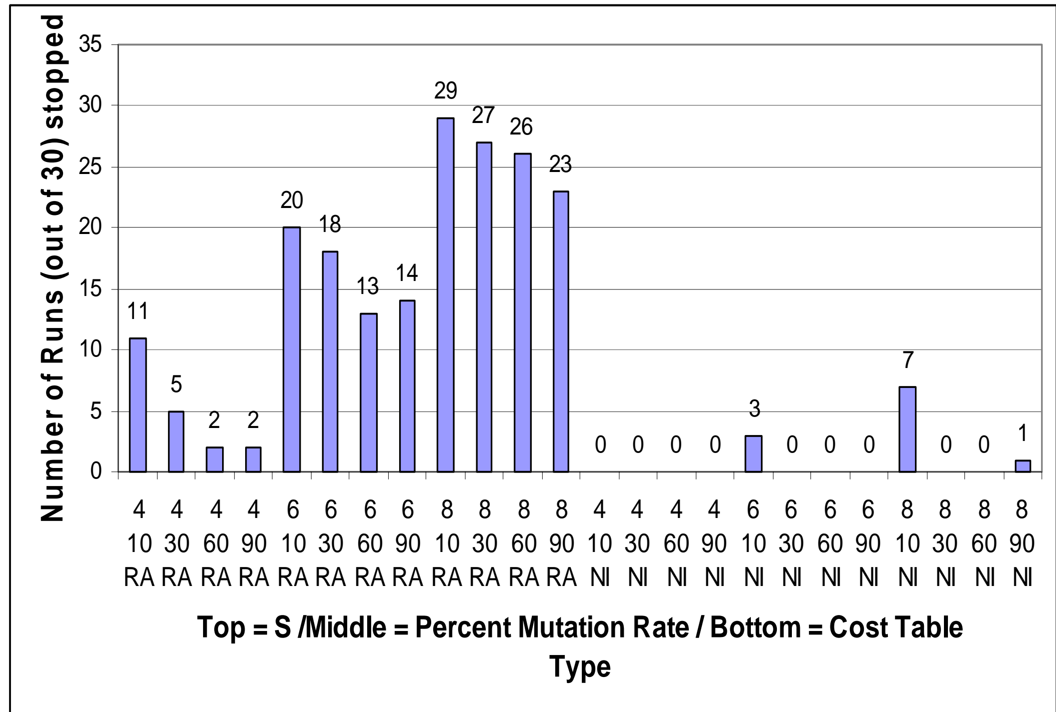

Figure 4.32: Comparison of Problem Size, and Mutation Rate That Needed to Be Stopped Using the Random and Non-Increasing Cost Table in Model 2

This model was also used to measure the effect of slack. In the first case slack was measured by taking an s=6 problem, and creating 1 period of slack by creating a row of cost 0 in the slack table. This was also done for 2 periods of slack. Population size was 50, generation size was 100 and mutation rate was 10 percent. Figure 4.33 shows these results. Slack had little or no effect on the number of generations needed to get a best result. In fact, looking at the data, adding slack time to the non-decreasing cost table seems to make it more random-like. The number of generations to produce a best result increased compared to that of the random table.

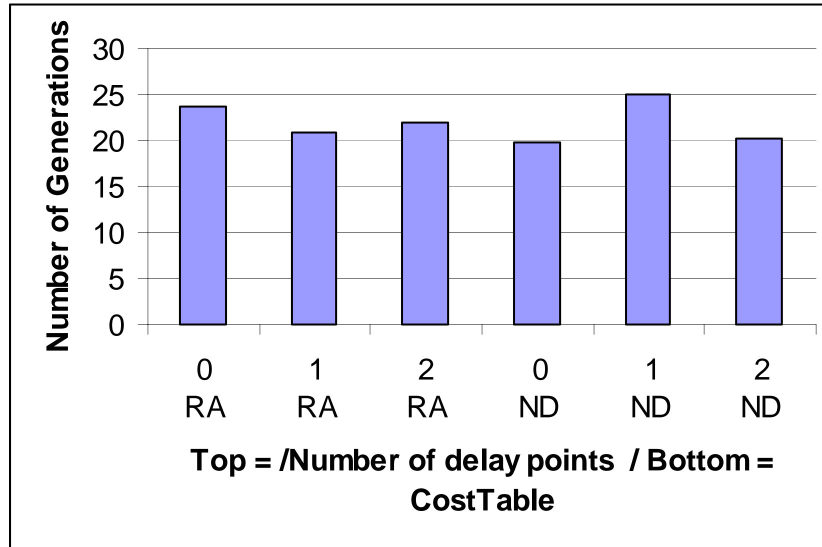

Figure 4.33: Comparison of a s = 6 Problem with 0, 1, 2 Slack Periods Using the Random and Non-decreasing Cost Table in Model 2 (With Adjustment)

Figure 4.34 displays the solution quality of the results in Figure 4.33. Slack in this case seemed to increase the number of runs which needed to be stopped.

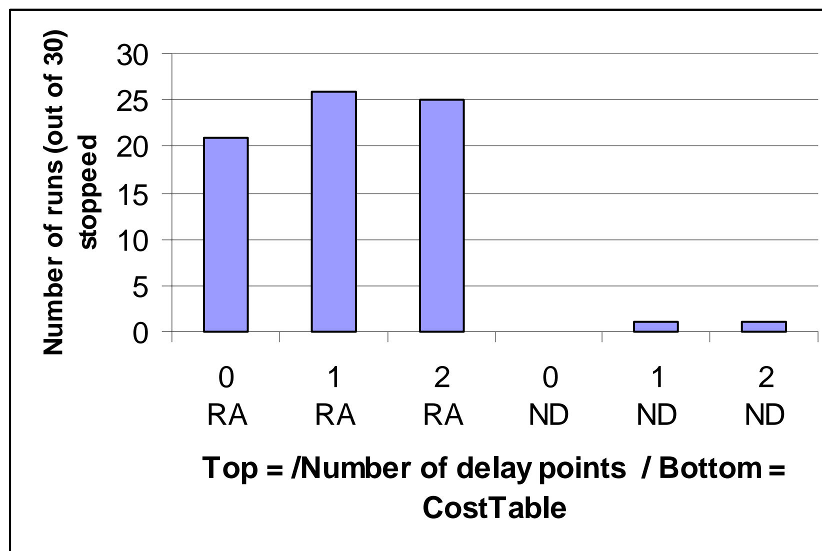

Figure 4.34: Comparison of an s = 6 Problem with 0, 1, 2 Slack Periods That Needed to Be Stopped Using the Random and Non-decreasing Cost Table in Model 2

In the next case an examination was done to see whether an s = 6 problem size with 2 slack periods would compare to a s = 5 problem size with 1 slack period and also a s = 4 with 0 slack periods. Figure 4.35 shows these results for both the random and non-decreasing cost table. The results were collected with a population size of 50, a generation size of 60 and a mutation rate of 10 percent. As seen in the figure, the number of generations increased as problem size increased similar to the other studies. Slack time did not change this effect.

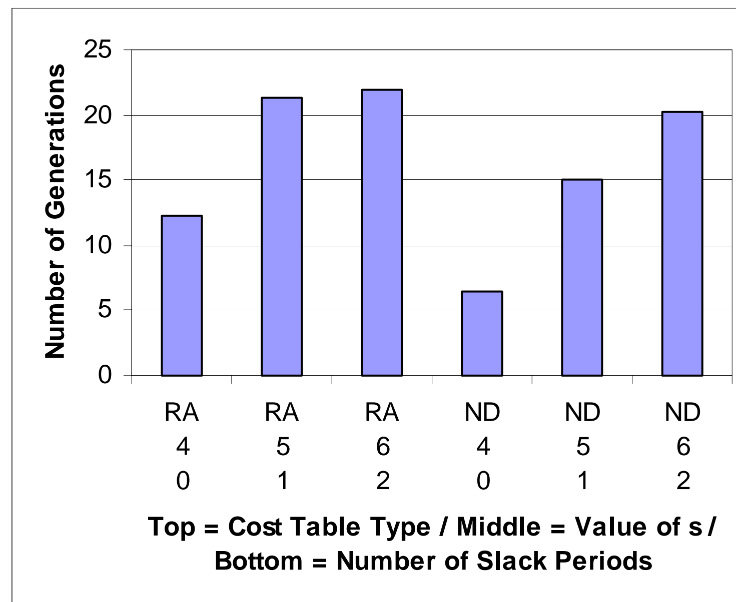

Figure 4.35: Comparison of Problem size with 0, 1, 2 slack periods using the Random and Non-decreasing Cost Table in Model 2 (With Adjustment)

Solution quality was also examined for the results in Figure 4.35 and is displayed in Figure 4.36. The numbers on top of the bars display the actual value out of 30. Notice that as the problem size increased the number of runs which did not finish also increased. The slack period did nothing to improve this result.

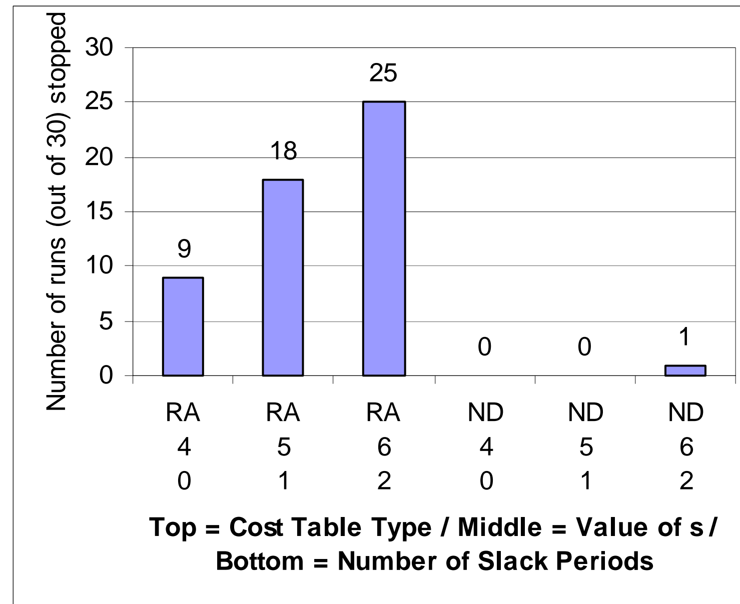

Figure 4.36: Comparison of a s = 6 Problem with 0, 1, 2 slack periods which needed to be stopped using the Random and Non-Decreasing Cost Table in Model 2

Finally one last examination of slack was done. The experiment in Figure 4.35 and Figure 4.36 was repeated but with some differences. First only an s = 6 with a 2 slack period cost table was produced. The s= 4 and s=5 would use only the portion of the cost table that it needed. Also, once the best answer was found for s = 4, it was used as the first encoding in the initial population for s = 5. Likewise, the best answer for s = 5 was used as the first encoding in the initial population for s = 6. See chapter 1, Table 1.16 for more details. Figure 4.37 shows these results. Again no improvement in the number of generations needed to obtain a best result is observed. However, the increase in the number of generations needed to obtain a best result due to an increase in problem size normally seen was not observed here. When going from s = 5 to s = 6 (that is from 1 slack periods to 2 slack periods) the results decreased instead of increased. This may

suggest a new possible way to increase the performance of genetic algorithms with larger problem sizes (a topic for future research). This will be discussed further in chapter 5.

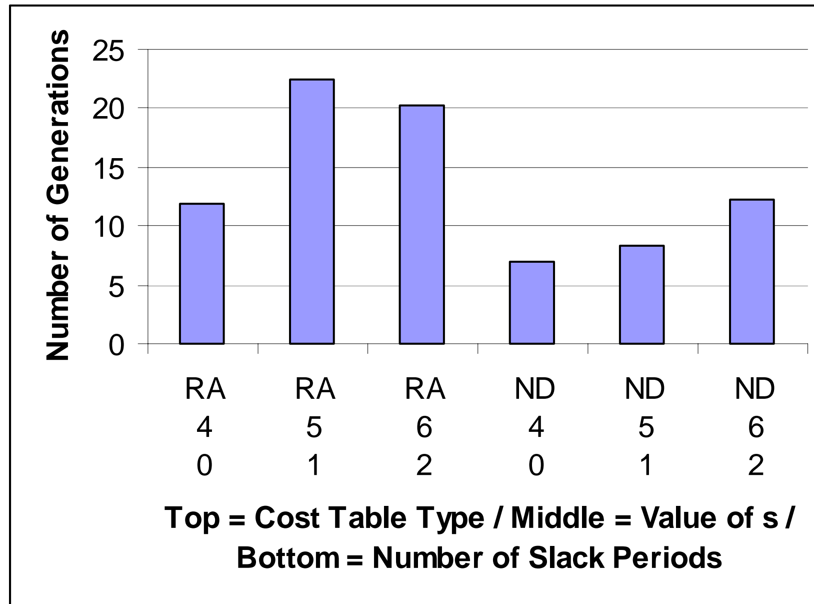

Figure 4.37: Comparison of Problem size with 0, 1, 2 slack periods(where an s=6 cost table was used for all) using the Random and Non-decreasing Cost Table in Model 2 (With Adjustment)

Figure 4.38 displays the solution quality of the data in Figure 4.37. The numbers on top of the bars display the actual value out of 30. Here, the number of runs that needed to be stopped increased as the problem size increased. The introduction of slack time and providing the best answer did not do anything to improve the number of runs that needed to be stopped. This leads us to the conclusions.

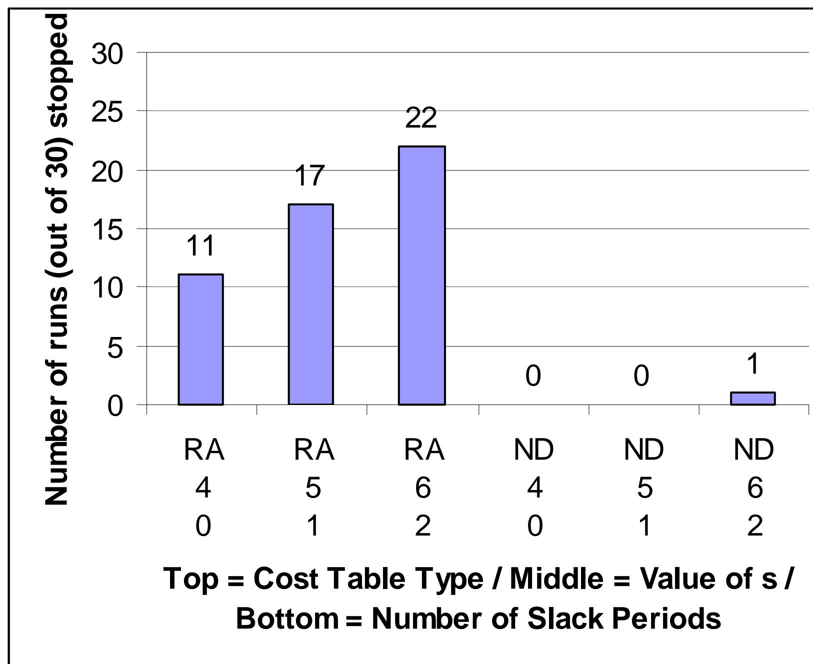

Figure 4.38: Comparison of Problem Size with 0, 1, 2 Slack Periods (Where an s=6 Cost Table was Used for All) Which Needed to be Stopped Using the Random and Non-decreasing Cost Table in Model 2

Chapter 5: Conclusions

Many graphs were presented in the preceding chapter and as a result, the data may seem a bit overwhelming. This final chapter will provide a summary of the work presented. Suggestions will also be given for future research.

A list of conclusions that can be drawn from the data in Chapter 4 is presented below.

1. The more random the cost table, the greater the number of generations needed to obtain a best result.

   As the "randomness" of the cost table increases, solution quality decreases. This means that the number of runs that needed to be stopped before optimal values were obtained increased. This was shown to be true in both models. The more structured cost tables, the non-decreasing, non- increasing, and multi-modal cost table all showed better performance. However, the random multi-modal also required a greater number of generations to obtain the best answer. Figures 4.1 through 4.5, 4.22, 4.24 through 4.26, 4.31, 4.33, 4.35 and 4.37 show the data that supports this claim.

   Note that Figure 4.4, the two random tables (RA and RM) stand out from the other cost tables in that a greater number of generations are required to obtain 15 runs completed (50 percent of the total runs per item studied completed).

2. As the problem size increased, the number of generations needed to obtain a best answer also increased.

As problem size increases, solution quality decreases all other things being equal. The number of runs that needed to be stopped before optimal values were obtained increased. This was shown to be true in both models. Figures 4.1 through 4.3, 4.10 through 4.12, 4.15, 4.18, 4.21, 4.22, 4.36, 4.27, and 4.29 through 4.32 show data that supports this.

3. <u>In most cases, as the mutation rate increased, the number of generations required to obtain a best answer decreased.</u>

   As the mutation rate increases solution quality increases. The number of runs that needed to be stopped before optimal values were obtained decreased. This is most likely due to the permutation (ordering) component of the models (that is the time dependency aspect of the model). This was shown to be 100 percent true for the examples in model 1 but only 60 percent true in model 2. A 90 percent mutation rate was the only point measured above 60 percent and there were times when 60 percent was better than 90 percent. More work needs to be done between 60 and 90 percent in order to determine the actual threshold. Figures 4.7 through 4.9, 4.14, 4.16, 4.17, 4.19, 4.31 and 4.32 show the data that supports this. In Figure 4.31, when *s* is 4, the 90% mutation turned out to be worse than 60% mutation. The same held true when *s* is 6 in Figure 4.32.

4. <u>In many cases, raising the generation size from 100 to 200 decreased the number of generations needed to obtain a best answer.</u>

The optimal conditions seem to be a population size of 50 or 100 with a generation size of 200. A population size of 150, however with a generation size of 200 seemed to hurt performance. The solution quality however did not exhibit the same degradation. Figures 4.26, 4.29, and 4.30 show these results.

Altering the population size and keeping the generation size constant, which was done in model 1, had very little effect on the number of generations needed to obtain a best answer. Some of the results showed the population size of 10 and 25 to perform better than the other conditions but that was not consistent in all studies. Figures 4.7, 4.9 through 4.13, 4.16, 4.18 through 4.20 and 4.29 show these results.

5. Slack time does not improve the number of generations needed to obtain a best answer.

   Solution quality is not improved with the introduction of slack time. It appears that the introduction of slack time increases the "randomness" of the cost table. The addition of slack time may actually worsen the situation because a cost of zero will affect the entire range of possible results.

In each case, an examination of favorable factors that lead to the least number of generations is always sought. The ultimate goal of this dissertation is that given certain characteristics of the parameters of the genetic algorithm to be implemented, a "prediction" can be made as to number of generations that will be required to obtain a best answer. If the structure of the cost table is particularly random, and the problem size

is quite large, a higher number of generations will usually be needed for a quality result. One way to counteract this may be to raise the mutation rate and/or increase the generation size.

There are several factors that are worth further study. The first is the number of crossover points in a mating. Further research to see whether varying the number of points leads to a certain effect would be of interest. Next, further work to find where the optimal rate for a mutation rate above 60 percent is suggested. Finer increments can be studied between 60 and 90 percent to find a true optimum.

Finally, the divide and conqueror scenario, which was somewhat studied using slack time, appears to have great potential for problem solving in fewer generations. In Figure 4.35, the result for a problem where s is 5 was used as the first member for a problem where s is 6 (This method is called seeding an initial population). A study could be preformed where a $s = 6$ problem could be broken up into smaller problems and the smaller problems could be solved. These results could then be used to seed the larger problem. Actually, to optimize this problem in this way may not be efficient. However, this method may perform better if the smaller problems were run only for a few generations and those results were used to seed rather than solving each of the smaller problems to completion.

This concludes this dissertation. I hope that these findings are useful and create the opportunity for future research.

References

[1] Gargano, M.L. and W. Edelson, Optimal Sequenced Matroid Bases Solved By Genetic Algorithms With Feasibility Including Applications, Congressus Numerantium 150, (2001) pp. 5-14.
[2] Mitchell M., An Introduction to Genetic Algorithms,Massachusetts Institute of Technology,1999
[3] Angeline, P. J. (1994) Genetic Programming: A Current Snapshot, In Proceedings of the Third Annual Conference on Evolutionary Programming, A. Sebald and L. Fogel (eds.), World Scientific, River Edge, NJ, pp. 224-232.
[4] Chou H., Premkumar G. and Chu C., Genetic Algorithms And Network Design: An Analysis Of Factors Influencing GA's Performance, eBusiness Research Center Working Paper 09-1999
[5] Chou H., Premkumar G. and Chu C., Telecommunications Network Design Decision: A Genetic Algorithm Approach, , eBusiness Research Center Working Paper 08-1999
[6] Edelson, W. and M. L. Gargano, Minimal Edge-Ordered Spanning Trees Solved By a Genetic Algorithm with Feasible Search Space, Congressus Numerantium 135, (1998) pp. 37-45.
[7] Hartsfield, N., G. Ringel, Pearls in Graph Theory, Academic Press, pp. 94 - 99.
[8] Goldberg, D.E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison Wesley, (1989).
[9] Davis, L., Handbook of Genetic Algorithms, Van Nostrand Reinhold, (1991).
[10] Gargano, M.L. and S. C. Friederich, On Constructing a Spanning Tree with Optimal Sequencing, Congressus Numerantium 71, (1990) pp. 67 – 72
[11] Gargano, M.L., L. V. Quintas and S. C. Friederich, Matroid Bases with Optimal Sequencing, Congressus Numerantium 82, (1991) pp. 65 - 77.
[12] Roberts, F.S. Discrete Mathematical Models, Prentice-Hall Inc.,(1970)
[13] Hillier, F.S. and G. J. Lieberman, Introduction to Operations Research, Holden-Day Inc. (1968).
[14] Rosen, K.H. Discrete Mathematics and Its Applications, Fourth Edition, Random House (1998).
[15] Michalewicz, Z., Heuristics for Evolutionary Computational Techniques, Journal of Heuristics, vol. 1, no. 2, (1996) pp. 596-597.
[16] Edelson, W. and M. L. Gargano, Feasible Encodings for GA Solutions of Constrained Minimal Spanning Tree Problems, late breaking paper at the 2000 Genetic and Evolutionary Computation Conference, 7/00, pp. 82 - 89.
[17] WebPages of England's SYS Research project for optimization of Electricity Pool http://www.sys.uea.ac.uk/~ajb/agents.html (April 2002)
[18] Holland, J.H., Adaption in Natural and Artificial Systems, The University of Michigan Press, (1975).
[19] WebPages of Cox Associates, http://www.cox-associates.com/index.html (May 2002)
[20] Webpages of HTL Netplan Software, http://www.htlt.com/products/NetPlan/Default.htm (May 2002)
[21] Webpages of Gilden Software, http://giden.nwu.edu/ (May 2002)

Appendix A. Calculation of Best Result using the Brute Force Method

This is the Brute Force calculation for the sample problem described in Chapter 3(Figure 3.1). The brute force value for this problem is obtained by checking every possible combination. The fitness was calculated as described in Figure 1.5. The cost table shown in Table 3.2 was used for fitness calculations. As the program that calculated the result ran it kept the current most fit result. When a combination with a better fitness was discovered the program would swap the result it currently contained with that of the more fit solution just discovered. The forth column in the table of all combinations below shows when this swap occurred. The total number of possible combinations is the product of Mk and s!. For this example, s! is 24($4! = 4 \bullet 3 \bullet 2 \bullet 1$) and the product of Mk (As described in Table 3.1) is 24 ($4 \bullet 3 \bullet 2 \bullet 1$) so the total number of possible permutations (combinations) is 576. The best fit result (which is combination number 308) for this problem is the combination 5,8,4,10 with a total cost of 118.

Combination Number	Fitness Value	Array Representing one possible result	Indicates that a more Fit value was found
1	225	10,2,7,5	Swap Occurred
2	182	10,3,7,5	Swap Occurred
3	165	10,4,7,5	Swap Occurred
4	209	10,1,8,5	
5	206	10,2,8,5	
6	163	10,3,8,5	Swap Occurred
7	146	10,4,8,5	Swap Occurred
8	244	10,1,9,5	
9	241	10,2,9,5	
10	198	10,3,9,5	
11	181	10,4,9,5	
12	249	10,1,7,6	
13	246	10,2,7,6	
14	203	10,3,7,6	
15	186	10,4,7,6	
16	230	10,1,8,6	
17	227	10,2,8,6	
18	184	10,3,8,6	
19	167	10,4,8,6	
20	265	10,1,9,6	
21	262	10,2,9,6	
22	219	10,3,9,6	
23	202	10,4,9,6	
24	228	10,1,7,5	
25	237	10,6,7,1	

26	191	10,5,8,1	
27	218	10,6,8,1	
28	226	10,5,9,1	
29	253	10,6,9,1	
30	209	10,5,7,2	
31	236	10,6,7,2	
32	190	10,5,8,2	
33	217	10,6,8,2	
34	225	10,5,9,2	
35	252	10,6,9,2	
36	174	10,5,7,3	
37	201	10,6,7,3	
38	155	10,5,8,3	
39	182	10,6,8,3	
40	190	10,5,9,3	
41	217	10,6,9,3	
42	161	10,5,7,4	
43	188	10,6,7,4	
44	142	10,5,8,4	Swap Occurred
45	169	10,6,8,4	
46	177	10,5,9,4	
47	204	10,6,9,4	
48	210	10,5,7,1	
49	201	10,8,1,5	
50	241	10,9,1,5	
51	220	10,7,2,5	
52	199	10,8,2,5	
53	239	10,9,2,5	
54	181	10,7,3,5	
55	160	10,8,3,5	
56	200	10,9,3,5	
57	166	10,7,4,5	
58	145	10,8,4,5	
59	185	10,9,4,5	
60	243	10,7,1,6	
61	222	10,8,1,6	
62	262	10,9,1,6	
63	241	10,7,2,6	
64	220	10,8,2,6	
65	260	10,9,2,6	
66	202	10,7,3,6	
67	181	10,8,3,6	
68	221	10,9,3,6	
69	187	10,7,4,6	
70	166	10,8,4,6	
71	206	10,9,4,6	
72	222	10,7,1,5	
73	192	10,8,5,1	
74	232	10,9,5,1	
75	237	10,7,6,1	
76	216	10,8,6,1	
77	256	10,9,6,1	

78	212	10,7,5,2	
79	191	10,8,5,2	
80	231	10,9,5,2	
81	236	10,7,6,2	
82	215	10,8,6,2	
83	255	10,9,6,2	
84	177	10,7,5,3	
85	156	10,8,5,3	
86	196	10,9,5,3	
87	201	10,7,6,3	
88	180	10,8,6,3	
89	220	10,9,6,3	
90	164	10,7,5,4	
91	143	10,8,5,4	
92	183	10,9,5,4	
93	188	10,7,6,4	
94	167	10,8,6,4	
95	207	10,9,6,4	
96	213	10,7,5,1	
97	243	10,6,1,7	
98	214	10,5,2,7	
99	241	10,6,2,7	
100	175	10,5,3,7	
101	202	10,6,3,7	
102	160	10,5,4,7	
103	187	10,6,4,7	
104	199	10,5,1,8	
105	226	10,6,1,8	
106	197	10,5,2,8	
107	224	10,6,2,8	
108	158	10,5,3,8	
109	185	10,6,3,8	
110	143	10,5,4,8	
111	170	10,6,4,8	
112	229	10,5,1,9	
113	256	10,6,1,9	
114	227	10,5,2,9	
115	254	10,6,2,9	
116	188	10,5,3,9	
117	215	10,6,3,9	
118	173	10,5,4,9	
119	200	10,6,4,9	
120	216	10,5,1,7	
121	222	10,2,5,7	
122	179	10,3,5,7	
123	162	10,4,5,7	
124	249	10,1,6,7	
125	246	10,2,6,7	
126	203	10,3,6,7	
127	186	10,4,6,7	
128	208	10,1,5,8	
129	205	10,2,5,8	

130	162	10,3,5,8	
131	145	10,4,5,8	
132	232	10,1,6,8	
133	229	10,2,6,8	
134	186	10,3,6,8	
135	169	10,4,6,8	
136	238	10,1,5,9	
137	235	10,2,5,9	
138	192	10,3,5,9	
139	175	10,4,5,9	
140	262	10,1,6,9	
141	259	10,2,6,9	
142	216	10,3,6,9	
143	199	10,4,6,9	
144	225	10,1,5,7	
145	184	8,1,5,10	
146	229	9,1,5,10	
147	204	7,2,5,10	
148	181	8,2,5,10	
149	226	9,2,5,10	
150	161	7,3,5,10	
151	138	8,3,5,10	Swap Occurred
152	183	9,3,5,10	
153	144	7,4,5,10	
154	121	8,4,5,10	Swap Occurred
155	166	9,4,5,10	
156	231	7,1,6,10	
157	208	8,1,6,10	
158	253	9,1,6,10	
159	228	7,2,6,10	
160	205	8,2,6,10	
161	250	9,2,6,10	
162	185	7,3,6,10	
163	162	8,3,6,10	
164	207	9,3,6,10	
165	168	7,4,6,10	
166	145	8,4,6,10	
167	190	9,4,6,10	
168	207	7,1,5,10	
169	175	8,5,1,10	
170	220	9,5,1,10	
171	225	7,6,1,10	
172	202	8,6,1,10	
173	247	9,6,1,10	
174	196	7,5,2,10	
175	173	8,5,2,10	
176	218	9,5,2,10	
177	223	7,6,2,10	
178	200	8,6,2,10	
179	245	9,6,2,10	
180	157	7,5,3,10	
181	134	8,5,3,10	

182	179	9,5,3,10	
183	184	7,6,3,10	
184	161	8,6,3,10	
185	206	9,6,3,10	
186	142	7,5,4,10	
187	119	8,5,4,10	Swap Occurred
188	164	9,5,4,10	
189	169	7,6,4,10	
190	146	8,6,4,10	
191	191	9,6,4,10	
192	198	7,5,1,10	
193	184	8,10,5,1	
194	229	9,10,5,1	
195	231	7,10,6,1	
196	208	8,10,6,1	
197	253	9,10,6,1	
198	206	7,10,5,2	
199	183	8,10,5,2	
200	228	9,10,5,2	
201	230	7,10,6,2	
202	207	8,10,6,2	
203	252	9,10,6,2	
204	171	7,10,5,3	
205	148	8,10,5,3	
206	193	9,10,5,3	
207	195	7,10,6,3	
208	172	8,10,6,3	
209	217	9,10,6,3	
210	158	7,10,5,4	
211	135	8,10,5,4	
212	180	9,10,5,4	
213	182	7,10,6,4	
214	159	8,10,6,4	
215	204	9,10,6,4	
216	207	7,10,5,1	
217	175	8,5,10,1	
218	220	9,5,10,1	
219	225	7,6,10,1	
220	202	8,6,10,1	
221	247	9,6,10,1	
222	197	7,5,10,2	
223	174	8,5,10,2	
224	219	9,5,10,2	
225	224	7,6,10,2	
226	201	8,6,10,2	
227	246	9,6,10,2	
228	162	7,5,10,3	
229	139	8,5,10,3	
230	184	9,5,10,3	
231	189	7,6,10,3	
232	166	8,6,10,3	
233	211	9,6,10,3	

234	149	7,5,10,4	
235	126	8,5,10,4	
236	171	9,5,10,4	
237	176	7,6,10,4	
238	153	8,6,10,4	
239	198	9,6,10,4	
240	198	7,5,10,1	
241	193	8,1,10,5	
242	238	9,1,10,5	
243	213	7,2,10,5	
244	190	8,2,10,5	
245	235	9,2,10,5	
246	170	7,3,10,5	
247	147	8,3,10,5	
248	192	9,3,10,5	
249	153	7,4,10,5	
250	130	8,4,10,5	
251	175	9,4,10,5	
252	237	7,1,10,6	
253	214	8,1,10,6	
254	259	9,1,10,6	
255	234	7,2,10,6	
256	211	8,2,10,6	
257	256	9,2,10,6	
258	191	7,3,10,6	
259	168	8,3,10,6	
260	213	9,3,10,6	
261	174	7,4,10,6	
262	151	8,4,10,6	
263	196	9,4,10,6	
264	216	7,1,10,5	
265	193	8,10,1,5	
266	238	9,10,1,5	
267	214	7,10,2,5	
268	191	8,10,2,5	
269	236	9,10,2,5	
270	175	7,10,3,5	
271	152	8,10,3,5	
272	197	9,10,3,5	
273	160	7,10,4,5	
274	137	8,10,4,5	
275	182	9,10,4,5	
276	237	7,10,1,6	
277	214	8,10,1,6	
278	259	9,10,1,6	
279	235	7,10,2,6	
280	212	8,10,2,6	
281	257	9,10,2,6	
282	196	7,10,3,6	
283	173	8,10,3,6	
284	218	9,10,3,6	
285	181	7,10,4,6	

286	158	8,10,4,6	
287	203	9,10,4,6	
288	216	7,10,1,5	
289	225	6,7,1,10	
290	174	5,8,1,10	
291	204	6,8,1,10	
292	214	5,9,1,10	
293	244	6,9,1,10	
294	193	5,7,2,10	
295	223	6,7,2,10	
296	172	5,8,2,10	
297	202	6,8,2,10	
298	212	5,9,2,10	
299	242	6,9,2,10	
300	154	5,7,3,10	
301	184	6,7,3,10	
302	133	5,8,3,10	
303	163	6,8,3,10	
304	173	5,9,3,10	
305	203	6,9,3,10	
306	139	5,7,4,10	
307	169	6,7,4,10	
308	118	5,8,4,10	Swap Occurred
309	148	6,8,4,10	
310	158	5,9,4,10	
311	188	6,9,4,10	
312	195	5,7,1,10	
313	237	6,10,1,7	
314	205	5,10,2,7	
315	235	6,10,2,7	
316	166	5,10,3,7	
317	196	6,10,3,7	
318	151	5,10,4,7	
319	181	6,10,4,7	
320	190	5,10,1,8	
321	220	6,10,1,8	
322	188	5,10,2,8	
323	218	6,10,2,8	
324	149	5,10,3,8	
325	179	6,10,3,8	
326	134	5,10,4,8	
327	164	6,10,4,8	
328	220	5,10,1,9	
329	250	6,10,1,9	
330	218	5,10,2,9	
331	248	6,10,2,9	
332	179	5,10,3,9	
333	209	6,10,3,9	
334	164	5,10,4,9	
335	194	6,10,4,9	
336	207	5,10,1,7	
337	225	6,7,10,1	

338	174	5,8,10,1	
339	204	6,8,10,1	
340	214	5,9,10,1	
341	244	6,9,10,1	
342	194	5,7,10,2	
343	224	6,7,10,2	
344	173	5,8,10,2	
345	203	6,8,10,2	
346	213	5,9,10,2	
347	243	6,9,10,2	
348	159	5,7,10,3	
349	189	6,7,10,3	
350	138	5,8,10,3	
351	168	6,8,10,3	
352	178	5,9,10,3	
353	208	6,9,10,3	
354	146	5,7,10,4	
355	176	6,7,10,4	
356	125	5,8,10,4	
357	155	6,8,10,4	
358	165	5,9,10,4	
359	195	6,9,10,4	
360	195	5,7,10,1	
361	231	6,10,7,1	
362	182	5,10,8,1	
363	212	6,10,8,1	
364	217	5,10,9,1	
365	247	6,10,9,1	
366	200	5,10,7,2	
367	230	6,10,7,2	
368	181	5,10,8,2	
369	211	6,10,8,2	
370	216	5,10,9,2	
371	246	6,10,9,2	
372	165	5,10,7,3	
373	195	6,10,7,3	
374	146	5,10,8,3	
375	176	6,10,8,3	
376	181	5,10,9,3	
377	211	6,10,9,3	
378	152	5,10,7,4	
379	182	6,10,7,4	
380	133	5,10,8,4	
381	163	6,10,8,4	
382	168	5,10,9,4	
383	198	6,10,9,4	
384	201	5,10,7,1	
385	231	6,1,7,10	
386	198	5,2,7,10	
387	228	6,2,7,10	
388	155	5,3,7,10	
389	185	6,3,7,10	

390	138	5,4,7,10	
391	168	6,4,7,10	
392	182	5,1,8,10	
393	212	6,1,8,10	
394	179	5,2,8,10	
395	209	6,2,8,10	
396	136	5,3,8,10	
397	166	6,3,8,10	
398	119	5,4,8,10	
399	149	6,4,8,10	
400	217	5,1,9,10	
401	247	6,1,9,10	
402	214	5,2,9,10	
403	244	6,2,9,10	
404	171	5,3,9,10	
405	201	6,3,9,10	
406	154	5,4,9,10	
407	184	6,4,9,10	
408	201	5,1,7,10	
409	237	6,1,10,7	
410	204	5,2,10,7	
411	234	6,2,10,7	
412	161	5,3,10,7	
413	191	6,3,10,7	
414	144	5,4,10,7	
415	174	6,4,10,7	
416	190	5,1,10,8	
417	220	6,1,10,8	
418	187	5,2,10,8	
419	217	6,2,10,8	
420	144	5,3,10,8	
421	174	6,3,10,8	
422	127	5,4,10,8	
423	157	6,4,10,8	
424	220	5,1,10,9	
425	250	6,1,10,9	
426	217	5,2,10,9	
427	247	6,2,10,9	
428	174	5,3,10,9	
429	204	6,3,10,9	
430	157	5,4,10,9	
431	187	6,4,10,9	
432	207	5,1,10,7	
433	224	2,10,7,5	
434	177	3,10,7,5	
435	158	4,10,7,5	
436	209	1,10,8,5	
437	205	2,10,8,5	
438	158	3,10,8,5	
439	139	4,10,8,5	
440	244	1,10,9,5	
441	240	2,10,9,5	

442	193	3,10,9,5	
443	174	4,10,9,5	
444	249	1,10,7,6	
445	245	2,10,7,6	
446	198	3,10,7,6	
447	179	4,10,7,6	
448	230	1,10,8,6	
449	226	2,10,8,6	
450	179	3,10,8,6	
451	160	4,10,8,6	
452	265	1,10,9,6	
453	261	2,10,9,6	
454	214	3,10,9,6	
455	195	4,10,9,6	
456	228	1,10,7,5	
457	206	2,5,7,10	
458	159	3,5,7,10	
459	140	4,5,7,10	
460	237	1,6,7,10	
461	233	2,6,7,10	
462	186	3,6,7,10	
463	167	4,6,7,10	
464	191	1,5,8,10	
465	187	2,5,8,10	
466	140	3,5,8,10	
467	121	4,5,8,10	
468	218	1,6,8,10	
469	214	2,6,8,10	
470	167	3,6,8,10	
471	148	4,6,8,10	
472	226	1,5,9,10	
473	222	2,5,9,10	
474	175	3,5,9,10	
475	156	4,5,9,10	
476	253	1,6,9,10	
477	249	2,6,9,10	
478	202	3,6,9,10	
479	183	4,6,9,10	
480	210	1,5,7,10	
481	209	2,7,5,10	
482	162	3,7,5,10	
483	143	4,7,5,10	
484	192	1,8,5,10	
485	188	2,8,5,10	
486	141	3,8,5,10	
487	122	4,8,5,10	
488	232	1,9,5,10	
489	228	2,9,5,10	
490	181	3,9,5,10	
491	162	4,9,5,10	
492	237	1,7,6,10	
493	233	2,7,6,10	

494	186	3,7,6,10	
495	167	4,7,6,10	
496	216	1,8,6,10	
497	212	2,8,6,10	
498	165	3,8,6,10	
499	146	4,8,6,10	
500	256	1,9,6,10	
501	252	2,9,6,10	
502	205	3,9,6,10	
503	186	4,9,6,10	
504	213	1,7,5,10	
505	218	2,7,10,5	
506	171	3,7,10,5	
507	152	4,7,10,5	
508	201	1,8,10,5	
509	197	2,8,10,5	
510	150	3,8,10,5	
511	131	4,8,10,5	
512	241	1,9,10,5	
513	237	2,9,10,5	
514	190	3,9,10,5	
515	171	4,9,10,5	
516	243	1,7,10,6	
517	239	2,7,10,6	
518	192	3,7,10,6	
519	173	4,7,10,6	
520	222	1,8,10,6	
521	218	2,8,10,6	
522	171	3,8,10,6	
523	152	4,8,10,6	
524	262	1,9,10,6	
525	258	2,9,10,6	
526	211	3,9,10,6	
527	192	4,9,10,6	
528	222	1,7,10,5	
529	221	2,10,5,7	
530	174	3,10,5,7	
531	155	4,10,5,7	
532	249	1,10,6,7	
533	245	2,10,6,7	
534	198	3,10,6,7	
535	179	4,10,6,7	
536	208	1,10,5,8	
537	204	2,10,5,8	
538	157	3,10,5,8	
539	138	4,10,5,8	
540	232	1,10,6,8	
541	228	2,10,6,8	
542	181	3,10,6,8	
543	162	4,10,6,8	
544	238	1,10,5,9	
545	234	2,10,5,9	

546	187	3,10,5,9	
547	168	4,10,5,9	
548	262	1,10,6,9	
549	258	2,10,6,9	
550	211	3,10,6,9	
551	192	4,10,6,9	
552	225	1,10,5,7	
553	212	2,5,10,7	
554	165	3,5,10,7	
555	146	4,5,10,7	
556	243	1,6,10,7	
557	239	2,6,10,7	
558	192	3,6,10,7	
559	173	4,6,10,7	
560	199	1,5,10,8	
561	195	2,5,10,8	
562	148	3,5,10,8	
563	129	4,5,10,8	
564	226	1,6,10,8	
565	222	2,6,10,8	
566	175	3,6,10,8	
567	156	4,6,10,8	
568	229	1,5,10,9	
569	225	2,5,10,9	
570	178	3,5,10,9	
571	159	4,5,10,9	
572	256	1,6,10,9	
573	252	2,6,10,9	
574	205	3,6,10,9	
575	186	4,6,10,9	
576	216	1,5,10,7	

Appendix B. Java Code Used for this Study

The java code used in this dissertation was broken up into 12 classes. The 12 classes were distributed into 12 java files. The 12 java files are listed below.

1. Brut.java -This code generates the Brute force result.
2. Cost.java -This code generates the Cost table.
3. Cross_Mut.java – This code performs Crossover and Mutation on one member.
4. Initial.java –This code reads in from files and sets the Intial input variables.
5. Main.java – This code contains the Main class needed in Java.
6. Member.java – This code generates new members and performs all member functions(an example is calculate fitness of a member).
7. Mult.java – This code manages multiple runs (usually 30). It calls Sing.java.
8. Pdiff.java – This code captures and calculates emperical differences from generation to generation.
9. Population.java - This code generates new populations of members and performs all population functions(an example is grimming a population).
10. Randm.java - This code generates all random numbers used in the entire program.
11. Sing.java - This code manages one complete single run.
12. Stat.java - This code calculates basic statistics on the run.

Each of these files is listed below.

Brut.java

```
// Takes care of running it Brut force
import java.util.StringTokenizer;
import java.text.NumberFormat;
import java.io.*;


public class Brut
{

double fitness;
int perm_length;
int [ ] perm;
int total_nub;
int [ ][ ] comb;
double factorial;
int [ ][ ] order;




//public void Brut(Population P,Initial I,double [ ][ ] Costtable)
        //{
        //int counter;
        //int position
        //Brut B = new Brut();
        //Brut Best = new Brut();
```

```
            //B.perm_length = I.S;
            //for (int i = 1; i <= M.perm_length ; i++)
            //Builds first half of encoding which is S long.
                        //{
                        //position =0;
                        //for (int j = I.M[i][2]; j <= I.M[i][3] ; j++)
                        //B.perm[i] = (
                        //temp = r.Randint(I.M[i][2],I.M[i][3]); // because i starts at 1
                         //M.encoding[i] = temp;
                        //}



            //}

public void Brut(Brut B,Initial I,double [ ][ ] Costtable)  throws IOException


{
                        String current;
                        Brut Best = new Brut();
                        Best.fitness = 999999999;
                        long counter = 0;
                        B.perm_length = I.S;
                        String Comb_file_name = "Comb" + I.S;
                        String outdir4 = I.Drive + "Comb";
                        File dir4 = new File(outdir4);
                        if (! dir4.exists()) dir4.mkdir();
                        File Comb_file = new File(dir4,Comb_file_name);
                        if (! Comb_file.exists()) B.Combinations(B,I);
                        B.factorial = B.factorial(I);
                        double cmk = B.CalcMK(I);
                        double totcomb = cmk * B.factorial;
                        //int totcomb = (cmk * (int)B.factorial);
                        I.Int_pw.println();
                        I.Int_pw.println("          S ! = " + B.factorial);
                        I.Int_pw.println("Product of Mk = " + cmk);
                        I.Int_pw.println("________________________");
                        I.Int_pw.println("              " + totcomb + " Combinations");
                        I.Int_pw.println();
                        if (I.write_total == true)
                        {
                                    I.Tot_pw.println();
                                    I.Tot_pw.println("          S ! = " + B.factorial);
                                    I.Tot_pw.println("Product of Mk = " + cmk);
                                    I.Tot_pw.println("________________________");
                                    I.Tot_pw.println("              " + totcomb + " Combinations");
                                    I.Tot_pw.println();
                        }
                        BufferedReader s = new BufferedReader(new FileReader(Comb_file));
                        while ((current = s.readLine()) != null )
                                    {
                                                B.filltable(current,B,I);
                                                while (B.order[(I.S + 1)][1] != I.S)
                                                {
                                                B.perm = new int [(I.S + 1)];
```

```
                                        B.tryone(B,I);
                                        B.CalcFitness(B,Costtable);
                                        counter = counter + 1;
                                        if (I.write_total == true) I.Tot_pw.print(counter + "," + B.fitness + ",");
                                        for (int i = 1; i <= I.S; i++)
                                                {
                                                if (I.write_total == true) I.Tot_pw.print(B.perm[i] + ",");
                                                }
                                        if (B.fitness < Best.fitness)
                                                {
                                                 Best.fitness = B.fitness;
                                                 Best.perm = B.perm;
                                                 if (I.write_total == true) I.Tot_pw.print("swap_occured,");
                                                }
                                        if (I.write_total == true) I.Tot_pw.println();
                                        }

                                }
                        I.Brute_pw.print(Best.fitness + ",");
                        I.Int_pw.println("Best fitness value = " + Best.fitness);
                        if (I.write_total == true) I.Tot_pw.println("Best fitness value = " + Best.fitness);
                                for (int i = 1; i <= I.S; i++)
                                                {
                                                if (I.write_total == true) I.Tot_pw.print(Best.perm[i] + ",");
                                                I.Int_pw.print(Best.perm[i] + ",");
                                                }
                                if (I.write_total == true) I.Tot_pw.println();
                                I.Int_pw.println();
                         B.fitness =          Best.fitness;
}

public void CalcFitness(Brut B, double [ ][ ] Costtable)
          {
                        //int time = 1;
                        double cost = 0;
           for (int i = 1; i <= B.perm_length  ; i++)
           {
                        cost = (cost + Costtable[(B.perm[i])][i]);
                        //System.out.println("Cost = " + cost);
                        //time = (time + 1);
           }
           B.fitness = cost;
          }

          public double CalcMK(Initial I)
          {

                        double Mk = I.M[1][1];
                        double temp = 0;
           for (int i = 2; i <= I.S  ; i++)
           {
                        temp = I.M[i][1];
                        Mk = (Mk * temp);
                        //System.out.println("Mk = " + Mk);
                        //System.out.println("Cost = " + cost);
                        //time = (time + 1);
```

```
            }
            return Mk;
            }

public double factorial(Initial I)
            {
            int value;
            double final_product;
            value = (I.S - 1);
            final_product = I.S;
            while (value > 1)
                        {
                        final_product = (final_product * value);
                        value = value - 1;
                        }
            return final_product;
            }
public void Combinations(Brut B,Initial I) throws IOException

            {
            boolean dup_exist;
            int counter1;
            Randm r = new Randm();
            int position;
            int counter2;
            int tempval;

            String Comb_file_name = ("Comb" + I.S);
            int [ ] check = new int[(I.S + 1)];
            int [ ] temp;
            //String Dir_name = I.Outdir + "Comb";
            String Dir_name = I.Drive + "Comb";
            File file = new File(Dir_name, Comb_file_name);
            FileWriter fw = new FileWriter(file);
            PrintWriter Comb = new PrintWriter(fw);
            B.factorial = B.factorial(I);
            System.out.println("B.factorial = " + B.factorial);
            int fact = (int)(B.factorial);
            //double fact = B.factorial;
            System.out.println("fact = " + fact);
            B.comb = new int[(fact + 1)][(I.S + 1)];

                        counter1 = 1;
                        while (counter1 <= fact)
                                    {
                                                            dup_exist = true;
                                                            while (dup_exist == true)
                                                            {
                                                                        for (int i = 1; i <= I.S; i++)
                                                                        //Loads in all possibilities 1 to S for position
                                                                                    {
                                                                                    check[i]=i;
                                                                                    }
                                                            temp = new int[(I.S + 1)];
                                                            for (int i = 1; i <= I.S; i++)
                                                            {
```

```
                                        position = r.Randint(1,((I.S + 1) - i));
                                        tempval = check[position];
                                        temp[i]= tempval;
                                        //System.out.print(temp[i] + ",");
                                        for (int j = position; j < (I.S - (i - 1)); j++)
                                //Note:  if position = last value of Array loop will not run but
last value
                                // gets reduced by 1 from using the expresion (S - (i - 1)).
                {
                                        check[j] = check[(j + 1)];
                                        }
                                }
                                //System.out.println();
                                dup_exist = B.checkdup(I,B,temp,counter1);
                                if (dup_exist == false)
                                {
                                        for (int i = 1; i <= I.S; i++)
                                                {
                                                B.comb[counter1][i]=temp[i];
                                                }
                                        counter1 = counter1 + 1;
                                        //System.out.println(counter1 + " out of " + fact);
                                 }
                                }
                        }

                B.bubbleSort(I,B.comb,fact);
         //counter1 = 0;
        for (int i = 1; i <= fact; i++)
        {
                //counter1 = counter1 + 1;
                //Comb.print(i + ",");
                for (int j = 1; j <= I.S; j++)
                {
                        Comb.print(B.comb[i][j] + ",");
                }
                Comb.println();
        }

        Comb.close();
    }

public void insert_one(int [ ] one_comb,int inpoint,int value,int numb_in_q)
    {
    int position;
    int tempcount;
    int [ ] tempq = new int[(numb_in_q + 1)];
    //System.out.print("inpoint = " + inpoint + ",");
    //System.out.print( "value = " + value  + ",");
    //System.out.print("numb_in_q = "+ numb_in_q + ",");
    //System.out.println();
    position = 0;
    tempcount = 0;
    if (numb_in_q == 0)
                {
                //System.out.println("in 1");
```

```
                    position = position + 1;
                    //System.out.print("numb_in_q = "+ numb_in_q + ",");
                    //System.out.println("position = " + position);
                    one_comb[position] = value;
                    //System.out.println("in 2");
                    }
        else
        {
        for (int i = 1; i <= numb_in_q; i++)
                {
                tempq[i] = one_comb[i];
                //System.out.print(one_comb[i] + " " + tempq[i]);
                //System.out.println();
                }
        for (int i = 1; i <= (numb_in_q + 1) ; i++)
                {
        //System.out.print("i = " + i + ",");
        //System.out.print("inpoint = " + inpoint + ",");
        //System.out.print( "value = " + value  + ",");
        //System.out.print("numb_in_q = "+ numb_in_q + ",");
        //System.out.println("position = " + position);
        //System.out.println();

                //System.out.print(i + " " + inpoint + " " + value  + " " + numb_in_q + ",");
                if (i == inpoint)
                    {
                        position = position + 1;
                        one_comb[position] = value;
                        //if (position < numb_in_q)
                        //{
                            //position = position + 1;
                            //tempcount = tempcount + 1;
                            //one_comb[position] = tempq[tempcount];
                        //}
                    }
                else
                    {
                    position = position + 1;
                    tempcount = tempcount + 1;
                    one_comb[position] = tempq[tempcount];
                    }
                }
        }
        for (int j = 1; j <= (numb_in_q + 1); j++)
                    {
                        System.out.print(one_comb[j] + ",");
                    }
                    System.out.println();

        } //End of insert_one


public static void bubbleSort(Initial I,int [ ][ ] comb,int n)
{
        //System.out.println("ok 1");
        int numSorted = 0;
```

```
            int index;
            int temp [ ];
            temp = new int[(I.S + 1)];
            while (numSorted < n)
            {
            //          System.out.println("ok 2");
                        for (index = 2; index <= (n - numSorted); index ++)
                        {
                                    //System.out.println("ok 3");
                                    if (comb[index][1] > comb[(index - 1)][1])
                                    {
                                    //System.out.println("ok 4");
                                    temp = comb[index];
                                    comb[index] = comb[(index - 1)];
                                    comb[(index - 1)] = temp;
                                    //System.out.println("ok 5");
                                    }
                        }
                        numSorted ++;
            }
}

public void filltable(String current,Brut B,Initial I)
            throws IOException
            {
            //System.out.println("in filltable:1");
            String s_numb;
            int numb;
            //System.out.println(current);
            StringTokenizer tok = new StringTokenizer(current, ",");
            B.order = new int [(I.S + 2)][6];
            //System.out.println("in filltable :2");
            B.order[(I.S + 1)][5] = 0;
            //System.out.println("in filltable :3");
            for (int i = 1; i <= I.S; i++)
                        {
                        s_numb = tok.nextToken();
                        numb = Integer.parseInt(s_numb);
                        //System.out.println("numb = " + numb);
                        B.order[i][1] = I.M[numb][1];
                        B.order[i][2] = I.M[numb][2];
                        B.order[i][3] = I.M[numb][3];
                        B.order[i][4] = B.order[i][2];
                        B.order[i][5] = 0;
                        //System.out.println("B.order[" + i + "][1] = " + B.order[i][1]);
                        //System.out.println("B.order[" + i + "][2] = " + B.order[i][2]);
                        //System.out.println("B.order[" + i + "][3] = " + B.order[i][3]);
                        //System.out.println("B.order[" + i + "][4] = " + B.order[i][4]);
                        //System.out.println("B.order[" + i + "][5] = " + B.order[i][5]);

                        }
            }
            public void RepeatArray(Brut B,int times,int numb_in_q)
            {
                        int orig_numb = B.total_nub;
                        for (int i = 1; i <= times; i++)
```

```
                {
                        for (int j = 1; j <= orig_numb; j++)
                        {
                        B.total_nub = B.total_nub + 1;
                        System.out.println("B.total_nub = " + B.total_nub);
                        //B.comb[B.total_nub] = B.comb[j];
                        for (int k = 1; k <= numb_in_q; k++)
                        {
                        B.comb[B.total_nub][k] = B.comb[j][k];
                        }

                        }
                }
        }

        public void tryone(Brut B,Initial I)
        {
                B.order[1][5] = 1;
                B.order[(I.S + 1)][1] = 0;
                for (int i = 1; i <= I.S; i++)
                {
                if (B.order[i][5] >= 1)
                        {
                                //System.out.println("tryone: 1");
                                B.order[i][4] = B.order[i][4] + 1;
                                B.order[i][5] = 0;
                                if (B.order[i][4] > B.order[i][3])
                                {
                                        //System.out.println(B.order[i][4] + " >= " + B.order[i][3]);
                                        //System.out.println("tryone: 2");
                                        B.order[(I.S + 1)][1] = B.order[(I.S + 1)][1] + 1;
                                        B.order[i][4] = B.order[i][2];
                                        B.order[(i + 1)][5] = 1;
                                }
                        }
                //System.out.println("B.order[" + i + "][4] = " + B.order[i][4]);
                //System.out.println("B.order[(I.S + 1)][1] = " + B.order[(I.S + 1)][1]);
                B.perm[i] = B.order[i][4];
                }
        }
        public boolean checkdup(Initial I,Brut B,int [ ] temp,int itemsInArray)
        {
        boolean result = false;
        int i = 1;
                while (i <= itemsInArray && result == false)
                                {
                                        int j = 1;
                                                        result = true;
        while (j <= I.S)
        {
        //System.out.println(B.comb[i][j] + " vs." + temp[j]);
         if (B.comb[i][j] != temp[j])
         {
                //System.out.println(" in ");
                result = false;
                j = (I.S + 1);
```

```
            }
            j = j + 1;
            }
            //System.out.print(i +",");
            i = i + 1;
                              }
            //true means it is a duplicate.
            return result;
 }


} //End of class
```

Cost.java

```
//Makes the cost table.
import java.io.*;
import java.util.StringTokenizer;

//Please note that SD refers to Non- Increasing (NI) and SI refers to Non-Decreasing (ND).

public class Cost
{

  double [ ][ ] Costtable;
//int[ ][ ] A = new int[2][2];

   public double [ ][ ] MakeCost(int rows, int cols)
            {
            Randm r = new Randm();
            double [ ][ ] A =  new double[rows][cols];
            for (int i = 0; i <= (rows -1); i++)
              {
            for (int j = 0; j <= (cols -1); j++)
                        {
            A[i][j] = r.Randdouble(2);
            //System.out.print(A[i][j]  + " ");
                        }
            //          System.out.println();
              }
            return A;
            }

public double [ ][ ] RAMakeCost(Initial I)
            {
            int rows = I.N;
            int cols = I.S;
            double lowerLimit = I.lowerLimit;
            double upperLimit = I.upperLimit;
            int skip = ((I.N - I.skip) + 1);
            Randm r = new Randm();
            double [ ][ ] A =  new double[(rows + 1)][(cols + 1)];
            for (int i = 1; i <= rows; i++)
              {
            for (int j = 1; j <= cols; j++)
                        {
```

```
                    if (i >= skip)
                    A[i][j] = I.skip_value;
                    else
                    A[i][j] = r.Randdouble(lowerLimit,upperLimit,0);
        //System.out.print(A[i][j]  + " ");
                }
        //          System.out.println();
          }
        return A;
        }

        public double [ ][ ] SI(Initial I)
        {
        int rows = I.N;
        int cols = I.S;
        double lowerLimit = I.lowerLimit;
        double upperLimit = I.upperLimit;
        double costchange = I.costchange;
        int skip = ((I.N - I.skip) + 1);
        Randm r = new Randm();
        double [ ][ ] A =  new double[(rows + 1)][(cols + 1)];
        for (int i = 1; i <= rows; i++)
          {
                if (i >= skip)
                    A[i][1] = I.skip_value;
                    else
                    A[i][1] = r.Randdouble(lowerLimit,upperLimit,0);
                double dec_costchange = (costchange / 100);
                double change2 = (dec_costchange * A[i][1]);
                double change = r.Round(change2,0);
                //System.out.println(costchange + " " + dec_costchange + " " + change);

        for (int j = 2; j <= cols; j++)
                {
                    if (i >= skip)
                    A[i][j] = I.skip_value;
                    else
                    A[i][j] = (A[i][(j - 1)] + change);
         //A[i][j] = r.Randdouble(lowerLimit,upperLimit,0);
        //System.out.print(A[i][j]  + " ");
                }
        //          System.out.println();
          }
        return A;
        }

public double [ ][ ] SD(Initial I)
        {
        int rows = I.N;
        int cols = I.S;
        double lowerLimit = I.lowerLimit;
        double upperLimit = I.upperLimit;
        double costchange = I.costchange;
        int skip = ((I.N - I.skip) + 1);
        Randm r = new Randm();
        double [ ][ ] A =  new double[(rows + 1)][(cols + 1)];
```

```
for (int i = 1; i <= rows; i++)
  {
        if (i >= skip)
              A[i][1] = I.skip_value;
              else
              A[i][1] = r.Randdouble(lowerLimit,upperLimit,0);
double dec_costchange = (costchange / 100);
double change2 = (dec_costchange * A[i][1]);
double change = r.Round(change2,0);
//System.out.println(costchange + " " + dec_costchange + " " + change);
for (int j = 2; j <= cols; j++)
        {
              if (i >= skip)
              A[i][j] = I.skip_value;
              else
              A[i][j] = (A[i][(j - 1)] - change);
 //A[i][j] = r.Randdouble(lowerLimit,upperLimit,0);
//System.out.print(A[i][j]  + " ");
        }
//      System.out.println();
  }
return A;
}

public double [ ][ ] MM(Initial I)
{
int rows = I.N;
int cols = I.S;
double lowerLimit = I.lowerLimit;
double upperLimit = I.upperLimit;
double costchange = I.costchange;
int skip = ((I.N - I.skip) + 1);
int up_down = 1;
Randm r = new Randm();
double [ ][ ] A =  new double[(rows + 1)][(cols + 1)];
for (int i = 1; i <= rows; i++)
  {
        if (i >= skip)
              A[i][1] = I.skip_value;
              else
              A[i][1] = r.Randdouble(lowerLimit,upperLimit,0);
double dec_costchange = (costchange / 100);
double change2 = (dec_costchange * A[i][1]);
double change = r.Round(change2,0);
//System.out.println(costchange + " " + dec_costchange + " " + change);
for (int j = 2; j <= cols; j++)
        {;
        //System.out.print(up_down);
        if (up_down == 1)
        {
              if (i >= skip)
                    A[i][j] = I.skip_value;
                    else
                    {
                    //System.out.println(",down");
                    A[i][j] = (A[i][(j - 1)] - change);
```

```
                                        up_down = 2;
                                        }
                        }
                        else
                        {
                                    if (i >= skip)
                                        A[i][j] = I.skip_value;
                                        else
                                        {
                                        //System.out.println(",up");
                                        A[i][j] = (A[i][(j - 1)] + change);
                                        up_down = 1;
                                        }
                        }

             //A[i][j] = r.Randdouble(lowerLimit,upperLimit,0);
            //System.out.print(A[i][j]  + " ");
                        }
            //          System.out.println();
              }
            return A;
            }

            public double [ ][ ] RM(Initial I)
            {
            int rows = I.N;
            int cols = I.S;
            double lowerLimit = I.lowerLimit;
            double upperLimit = I.upperLimit;
            double costchange = I.costchange;
            int up_down = 1;
            Randm r = new Randm();
            double change;
            int skip = ((I.N - I.skip) + 1);
            double [ ][ ] A =  new double[(rows + 1)][(cols + 1)];
            for (int i = 1; i <= rows; i++)
              {
                                    if (i >= skip)
                                    A[i][1] = I.skip_value;
                                    else
                                    A[i][1] = r.Randdouble(lowerLimit,upperLimit,0);
            double dec_costchange = (costchange / 100);
            double change2 = (dec_costchange * A[i][1]);
            double change3 = r.Round(change2,0);
            //System.out.println(costchange + " " + dec_costchange + " " + change);
            for (int j = 2; j <= cols; j++)
                        {;
                        //System.out.print(up_down);
                        if (up_down == 1)
                        {
                                    if (i >= skip)
                                        A[i][j] = I.skip_value;
                                        else
                                        {
                                        //System.out.println(",down");
                                        change = r.Randdouble(1,change3,0);
```

```
                                        A[i][j] = (A[i][(j - 1)] - change);
                                        up_down = 2;
                                        }
                        }
                        else
                        {
                                if (i >= skip)
                                        A[i][j] = I.skip_value;
                                        else
                                        {
                                        //System.out.println(",up");
                                        change = r.Randdouble(1,change3,0);
                                        A[i][j] = (A[i][(j - 1)] + change);
                                        up_down = 1;
                                        }
                        }

                //A[i][j] = r.Randdouble(lowerLimit,upperLimit,0);
                //System.out.print(A[i][j]  + " ");
                        }
                //              System.out.println();
                  }
                return A;
                }



   public void ShowCost(double [ ][ ] Costtable)
                        throws IOException

                {
   //Inout s = new Inout();
                int rows = (Costtable.length - 1);
  for (int i = 1; i <= rows; i++)
                  {

                  int cols = (Costtable[i].length - 1);
                  for (int j = 1; j <= cols; j++)
                        {
                          System.out.print(Costtable[i][j] + " ");
                          //s.send(Costtable[i][j]);

                        }
                  System.out.println();
                  }
                  //s.close();
                }

 public void WriteCost(double [ ][ ] Costtable,Initial I)
                throws IOException

  {
  //String file_name = "C" + I.Method + "R" + run_number;
  String file_name = I.Costtab;
  File file = new File(I.Outdir + "Cost", file_name);
  FileWriter fw = new FileWriter(file);
```

```
PrintWriter pw = new PrintWriter(fw);

 //Inout s = new Inout();
 int rows = (Costtable.length - 1);
for (int i = 1; i <= rows ; i++)
              {

              int cols = (Costtable[i].length - 1);
              for (int j = 1; j <= cols ; j++)
                          {
                            //System.out.print(Costtable[i][j] + " ");
                            pw.print(Costtable[i][j]  + ",");

                          }
                pw.println();
                //pw.println();
              }

pw.close();
}
public double [ ][ ] ReadCost(Initial I)
            throws IOException
{
  String current;
            //double [ ][ ] Costtable;
            double [ ][ ] Costtable = new double [(I.N + 1)][(I.S + 1)];
  //String file_name = "C" + I.Method + "R" + I.run_number;
            String file_name = I.Costtab;
  File file = new File(I.Outdir + "Cost", file_name);
  //ReadStream s = new ReadStream(new DataInputStream(new FileInputStream(File)));
  BufferedReader s = new BufferedReader(new FileReader(file));
  int i = 1;
  while ((current = s.readLine()) != null )
  {
            StringTokenizer tok = new StringTokenizer(current, ",");
            int j = 1;
            //String dir_name = tok.nextToken();
              while( tok.hasMoreTokens()){
                  String temp_hold = tok.nextToken();
                  //float price = Float.valueOf(temp_hold).floatValue();
                   double temp_val = Float.valueOf(temp_hold).doubleValue();
                  Costtable[i][j] = temp_val;
                          j++;
                                        }
              i++;

                              }
    return Costtable;
}

public void AddInit(double [ ][ ] Costtable,PrintWriter pw)
                          throws IOException

            {
                          pw.println();
                          pw.println("Cost Table");
```

```
                pw.println("____________");
                pw.println();

        int rows = (Costtable.length - 1);
 for (int i = 1; i <= rows; i++)
         {

         int cols = (Costtable[i].length - 1);
         pw.print(i + " |");
         for (int j = 1; j <= cols; j++)
                {
                 pw.print(Costtable[i][j] + " ");



                }
         pw.println();
         }

        }


}
```

Cross_Mut.java

```
// Takes care of Crossover and Mutation


public class Cross_Mut
{


public Member Crossover(Population P,Initial I,double [ ][ ] Costtable)
        {
        Member TempMemb;
        int Memb1_indx;
        Member Memb1;
        int Memb2_indx;
        Member Memb2;
        int low = 1;
        int start;
        int end;

        Randm r = new Randm();
 TempMemb  = new Member();
        Memb1 = new Member();
        Memb2 = new Member();
        Memb1_indx = r.Randint(low,P.pop_size);
        Memb2_indx = r.Randint(low,P.pop_size);
        //Next 4 lines is so you do not get same member crossing over
        while(Memb1_indx == Memb2_indx)
                {
                Memb2_indx = r.Randint(low,P.pop_size);
                }
        if (P.gen_in_use == 1)
```

```
                    {
            //System.out.println(P.gen_in_use);
                    Memb1 = P.Gen[Memb1_indx];
                    Memb2 = P.Gen[Memb2_indx];
  }
  else
                    {
  Memb1 = P.Gen[Memb1_indx];
                    Memb2 = P.Gen[Memb2_indx];
                    }
            TempMemb.encoding = new int[( Memb1.Encoding_lenght + 1)];
            start = r.Randint(low,(I.S - 1));
            end = r.Randint((start + 1),I.S);
            for (int i = 1; i <= Memb1.Encoding_lenght ; i++)
                    {
                    if (i >= start && i <= end )
                            {
                    //System.out.println(P.gen_in_use);
                            TempMemb.encoding[i]=Memb2.encoding[i];
            }
            else
                            {
   TempMemb.encoding[i]=Memb1.encoding[i];
                            }
                    }
                    TempMemb.Fillmember(I,TempMemb,Costtable);
                    if (I.write_total_log == true)
                                            {
                                            P.Gen_pw.println("Combined member " + Memb1_indx + " and
member " + Memb2_indx );
                                            P.Gen_pw.println("Crossover begins at position " + start + " and ends
at position " + end);
                                            Memb1.WriteMember(Memb1,P.Gen_pw);
                                            Memb2.WriteMember(Memb2,P.Gen_pw);

            P.Gen_pw.println("_____________________________________________________________________
____");
                                            TempMemb.WriteMember(TempMemb,P.Gen_pw);
                                            P.Gen_pw.println();
                                            }

                    return TempMemb;

            }

 public Member Mutation(Population P,Initial I,double [ ][ ] Costtable)
            {
            int max_number_of_mutations = (I.S - 1);
            //
            Member TempMemb;
            int Memb_indx;
            Member Memb1;
            int low = 1;
            int temp_int;
            int countdown;
            int number_of_mutations;
```

```
            int boundry_pointer;
            int trigger;
            int mut_counter;
            int counter2;
            int start;
            int end;

            Randm r = new Randm();
            TempMemb  = new Member();
            Memb1 = new Member();
            Memb_indx = r.Randint(low,P.pop_size);
            if (P.gen_in_use == 1)
                        {
             //System.out.println(P.gen_in_use);
                        Memb1 = P.Gen[Memb_indx];
            }
  else
                        {
   Memb1 = P.Gen2[Memb_indx];
                        }
            TempMemb.encoding = new int[( Memb1.Encoding_lenght + 1)];
            number_of_mutations = r.Randint(low,max_number_of_mutations);
            //number_of_mutations = 3;
            int mutation_index [ ] = new int [(number_of_mutations + 1)];
            boundry_pointer = 1;
            countdown = number_of_mutations;
            if (I.write_total_log == true) P.Gen_pw.println("Mutate member " + Memb_indx);
            if (I.write_total_log == true) P.Gen_pw.print(number_of_mutations + " Mutation(s) will happen at
position(s) " );
            for (int i = 1; i <= number_of_mutations; i++)
                        {
                        temp_int=r.Randint(boundry_pointer,(Memb1.Encoding_lenght - countdown));
                        countdown = countdown - 1;
                        boundry_pointer = (temp_int + 1);
                        mutation_index[i] = temp_int;
                        if (I.write_total_log == true)
                                    {
                                    if (i != number_of_mutations && i != 1)P.Gen_pw.print("," + temp_int);

                                                            else

                                                            {

                                                            if (i == 1) P.Gen_pw.print(temp_int);

                                                            else P.Gen_pw.print(" and " + temp_int);

                                                            }

                                    }
                        }
            //start = r.Randint(low,(I.S - 1));
            //end = r.Randint((start + 1),I.S);
            if (I.write_total_log == true) P.Gen_pw.println();
            if (I.write_total_log == true) P.Gen_pw.print("The value(s) are ");
            mut_counter = 1;
```

```
            trigger = mutation_index[mut_counter];
            for (int i = 1; i <= Memb1.perm_length ; i++)
                        {
                        if (i != trigger)
                                    {
                        //System.out.println(P.gen_in_use);
                                    TempMemb.encoding[i]=Memb1.encoding[i];
      }
            else
                                    {
                                    temp_int=r.Randint(I.M[i][2],I.M[i][3]);
                        TempMemb.encoding[i]=temp_int;
                        if (I.write_total_log == true) P.Gen_pw.print(temp_int + " ");
                        mut_counter = mut_counter + 1;
                        if (mut_counter <= number_of_mutations) trigger = mutation_index[mut_counter];

                                    }
                        }
            // THIS BEGINS SECOND HALF OF ENCODING:
            counter2 = 0;
            for (int i = (Memb1.perm_length + 1); i <= Memb1.Encoding_lenght; i++)
                        //Builds second half of encoding which is S long( from (S+1) to 2s.
            {
                        if (i != trigger)
                                    {
                                    //M.encoding[i] = r.Randint(min,(M.perm_length - counter));
                                    //counter = counter + 1;
                                                if (Memb1.encoding[i] >= low &&  Memb1.encoding[i] <=
(Memb1.perm_length - counter2)  )
                                                {
                                                            TempMemb.encoding[i] = Memb1.encoding[i];
                                                            counter2 = counter2 + 1;
        }
            else
                                                {
                                                            temp_int = r.Randint(low,(Memb1.perm_length -
counter2));
                                                            TempMemb.encoding[i] = temp_int;
                                                            if (I.write_total_log == true)
P.Gen_pw.print(temp_int + " ");
                                                            counter2 = counter2 + 1;
                                                            mut_counter = mut_counter + 1;
                                                            if (mut_counter <= number_of_mutations) trigger =
mutation_index[mut_counter];

                                                }
                                    }
      else
                                    {
                                    temp_int= r.Randint(low,(Memb1.perm_length - counter2));
                                    TempMemb.encoding[i] = temp_int;
                                    if (I.write_total_log == true) P.Gen_pw.print(temp_int + " ");
                                    counter2 = counter2 + 1;
                                    mut_counter = mut_counter + 1;
                                    if (mut_counter <= number_of_mutations) trigger =
mutation_index[mut_counter];
```

```
                              }
            }
            TempMemb.Fillmember(I,TempMemb,Costtable);

            if (I.write_total_log == true)
            {
                        P.Gen_pw.println();
                        Memb1.WriteMember(Memb1,P.Gen_pw);

            P.Gen_pw.println("________________________________________________________________________
_____");
                        TempMemb.WriteMember(TempMemb,P.Gen_pw);
                        P.Gen_pw.println();
            }
            return TempMemb;

 }
}
```

Initial.java

```
// Get inital Values for Program

import java.util.StringTokenizer;
import java.text.NumberFormat;
import java.io.*;
import java.lang.*;

public class Initial
{

            int run_number;
            int r2_run_number;
            int temp_run_number;
  String Outdir;
  String Drive;
  String Costtab;
  String Costtype;
  String Brutvalue;
  String run_type;
  String current;
  double lowerLimit;
  double upperLimit;
  double percent_of_brut;
  int skip;
  double skip_value = 0;
  int costchange;
  static String Method = "M2";
             int N;
             int S;
             int M_length = (S + 1);
             //int M [ ][ ] = new int[101][4];
             int M  [ ][ ];
```

```
            int number_of_generations;
            //M = new int[ ][ ];
            int percent_mutations;
            int pop_size;
            int gen_size;
            int change;
            PrintWriter Int_pw;
            PrintWriter Tot_pw;
            PrintWriter Brute_pw;
            PrintWriter Sum_pw;
            PrintWriter Sum2_pw;
            int max_dup_count;
           boolean write_total;
           boolean write_total_log;
           boolean write_total_setting;
           boolean write_total_setting_log;
           int mult_run_type;
           int mult_run_number; //R value
           int mult_run_input_number;
           int current_input_numb;



            public void GetOutDirName(Initial I) throws IOException
{
String file_name = "OutDir";
String current;
//String dir_name = "A";
BufferedReader s = new BufferedReader(new FileReader(file_name));
//ReadStream s = new ReadStream(new DataInputStream(new FileInputStream(file_name)));
  //while(! s.eof()){
 //while ((current = s.readLine()) != null )
 //{
current = s.readLine();
           StringTokenizer tok = new StringTokenizer(current, ",");
           String s_mult_run_type = tok.nextToken();
           I.mult_run_type = Integer.parseInt(s_mult_run_type);
           I.Drive = tok.nextToken();
           String lower = tok.nextToken();
           I.lowerLimit = Double.parseDouble(lower);
           //System.out.println("out1");
           String upper = tok.nextToken();
           I.upperLimit = Double.parseDouble(upper);
           String s_max_dup_count = tok.nextToken();
I.max_dup_count = Integer.parseInt(s_max_dup_count);
           String s_write_total_setting = tok.nextToken();
           I.write_total_setting = Boolean.getBoolean(s_write_total_setting);
           //System.out.println("out2");
           String s_write_total_setting2 = tok.nextToken();
           I.write_total_setting_log = Boolean.getBoolean(s_write_total_setting2);
           //System.out.println("out3");
           String s_I_mult_run_number = tok.nextToken();
           I.mult_run_number = Integer.parseInt(s_I_mult_run_number);
           //String s_s = tok.nextToken();
           //I.S = Integer.parseInt(s_s);
           //String s_mult_run_input_number = tok.nextToken();
```

```
            //I.mult_run_input_number = Integer.parseInt(s_mult_run_input_number);
   //.out.println("out4 ");
   }

  public void GetInitialS(Initial I) throws IOException
  {
  String file_name = I.Drive + "1\\OutDir2";
  String current;
  BufferedReader s = new BufferedReader(new FileReader(file_name));
  current = s.readLine();
            StringTokenizer tok = new StringTokenizer(current, ",");
            String s_s = tok.nextToken();
            I.S = Integer.parseInt(s_s);
            String s_mult_run_input_number = tok.nextToken();
            I.mult_run_input_number = Integer.parseInt(s_mult_run_input_number);

  }

public void GetInitial(String current,Initial I)
            throws IOException

            {
            String temp;
            String s_temp_run_number;
            int temp2;
            int low;
            int high;

  //
            //I.GetOutDirName(I);
            StringTokenizer tok = new StringTokenizer(current, ",");
            s_temp_run_number = tok.nextToken();
            //I.temp_run_number = Integer.parseInt(s_temp_run_number);
            I.run_number = Integer.parseInt(s_temp_run_number);
            I.run_type = tok.nextToken();
            //System.out.println(I.run_type);
            if (I.run_type.equals("R1"))
            {
                        I.write_total = true;
                        I.Costtab = tok.nextToken();
                        //System.out.println(I.Outdir);

                        String Int_file_name = "In" + I.Method + "R" + I.run_number;
                        String outdir1 = I.Outdir + "Results";
                        if (I.mult_run_type == 3) outdir1 = I.Outdir + "Results" + I.current_input_numb;
                        File dir1 = new File(outdir1);
                        if (! dir1.exists()) dir1.mkdir();
            File file1 = new File(dir1, Int_file_name);
            FileWriter fw = new FileWriter(file1);
            I.Int_pw = new PrintWriter(fw);

            String Tot_file_name = "To" + I.Method + "R" + I.run_number;
            File file2 = new File(I.Outdir + "Results", Tot_file_name);
            if (I.mult_run_type == 3) file2 = new File(I.Outdir + "Results" + I.current_input_numb,
Tot_file_name);
            FileWriter fw2 = new FileWriter(file2);
```

```
I.Tot_pw = new PrintWriter(fw2);

PrintWriter pw = I.Int_pw;

String n_gen = tok.nextToken();
        I.number_of_generations = Integer.parseInt(n_gen);
        String n_pop_size = tok.nextToken();
        I.pop_size = Integer.parseInt(n_pop_size);
                String n_gen_size = tok.nextToken();
        I.gen_size = Integer.parseInt(n_gen_size);
                String n_percent_mutations = tok.nextToken();
        I.percent_mutations = Integer.parseInt(n_percent_mutations);

String n_strong = tok.nextToken();
        I.N = Integer.parseInt(n_strong);
        //System.out.println(I.N);
        pw.println("Run number = " + run_number);
        I.Tot_pw.println("Run number = " + run_number);
        pw.println("N= " + I.N);
        I.Tot_pw.println("N= " + I.N);
        String s_strong = tok.nextToken();
        I.S = Integer.parseInt(s_strong);
        pw.println("S= " + I.S);
        I.Tot_pw.println("S= " + I.S);
        I.M = new int[(I.S + 1)][4];
        //System.out.println("S= " + I.S);
        pw.println("Mk   Low   High");
        I.Tot_pw.println("Mk   Low   High");
        low = 1;
        for (int i = 1; i <= (I.S); i++)
                {
                temp = tok.nextToken();
                //System.out.println(temp);
                temp2 = Integer.parseInt(temp);
                //System.out.println(temp2);
                I.M[i][1] = temp2;
                I.M[i][2] = low;
                high = low + temp2;
                I.M[i][3] = (high - 1);
                low = high;


                pw.print((I.M[i][1] + " "));
                I.Tot_pw.print((I.M[i][1] + " "));
                pw.print(("    " + I.M[i][2] + " "));
                I.Tot_pw.print(("    " + I.M[i][2] + " "));
                pw.print(("    " + I.M[i][3]));
                I.Tot_pw.print(("    " + I.M[i][3]));
                pw.println();
                I.Tot_pw.println();
                //System.out.print((I.M[i][1] + " "));
                //System.out.print((I.M[i][2] + " "));
                //System.out.print((I.M[i][3] + " "));
                //System.out.println();
                }
```

```
                    }  //End of if R
                    if (I.run_type.equals("R2"))
          {
                    I.write_total = true;

                    if (I.write_total_setting_log ) I.write_total_log = true;
                    else
                    {
                              if (I.run_number < 4 ) I.write_total_log = true;
                              else I.write_total_log = false;
                    }
                    //System.out.println("I.write_total_log = " + I.write_total_log);
                    //System.out.println("Run number = " + I.run_number);
                    I.Costtab = tok.nextToken();
                    I.Brutvalue = tok.nextToken();
                    //System.out.println(I.Outdir);

                    String Int_file_name = "In" + I.Method + "R" + I.run_number;
                    String outdir1 = I.Outdir + "Results";
                    if (I.mult_run_type == 3) outdir1 = I.Outdir + "Results" + I.current_input_numb;
                    File dir1 = new File(outdir1);
                    if (! dir1.exists()) dir1.mkdir();
          File file1 = new File(dir1, Int_file_name);
          FileWriter fw = new FileWriter(file1);
          I.Int_pw = new PrintWriter(fw);

          if (I.write_total_log == true)
                    {
          String Tot_file_name = "To" + I.Method + "R" + I.run_number;
          File file2 = new File(I.Outdir + "Results", Tot_file_name);
          if (I.mult_run_type == 3) file2 = new File(I.Outdir + "Results" + I.current_input_numb,
Tot_file_name);
          FileWriter fw2 = new FileWriter(file2);
          I.Tot_pw = new PrintWriter(fw2);
                    }
          PrintWriter pw = I.Int_pw;

          String n_percent_of_brut = tok.nextToken();
                    I.percent_of_brut= Double.parseDouble(n_percent_of_brut);
                    String n_pop_size = tok.nextToken();
                    I.pop_size = Integer.parseInt(n_pop_size);
                              String n_gen_size = tok.nextToken();
                    I.gen_size = Integer.parseInt(n_gen_size);
                              String n_percent_mutations = tok.nextToken();
                    I.percent_mutations = Integer.parseInt(n_percent_mutations);

          String n_strong = tok.nextToken();
                    I.N = Integer.parseInt(n_strong);
                    //System.out.println(I.N);
                    pw.println("Run number = " + run_number);

          if (I.write_total_log == true) I.Tot_pw.println("Run number = " + run_number);
                    pw.println("N= " + I.N);
                    if (I.write_total_log == true) I.Tot_pw.println("N= " + I.N);
                    String s_strong = tok.nextToken();
                    I.S = Integer.parseInt(s_strong);
```

```
            pw.println("S= " + I.S);
            if (I.write_total_log == true) I.Tot_pw.println("S= " + I.S);
            I.M = new int[(I.S + 1)][4];
            //System.out.println("S= " + I.S);
            pw.println("Mk   Low   High");
            if (I.write_total_log == true) I.Tot_pw.println("Mk   Low   High");
            low = 1;
            for (int i = 1; i <= (I.S); i++)
                    {
                    temp = tok.nextToken();
                    //System.out.println(temp);
                    temp2 = Integer.parseInt(temp);
                    //System.out.println(temp2);
                    I.M[i][1] = temp2;
                    I.M[i][2] = low;
                    high = low + temp2;
                    I.M[i][3] = (high - 1);
                    low = high;


                    pw.print((I.M[i][1] + " "));
                    if (I.write_total_log == true) I.Tot_pw.print((I.M[i][1] + " "));
                    pw.print(("    " + I.M[i][2] + " "));
                    if (I.write_total_log == true) I.Tot_pw.print(("    " + I.M[i][2] + " "));
                    pw.print(("    " + I.M[i][3]));
                    if (I.write_total_log == true) I.Tot_pw.print(("    " + I.M[i][3]));
                    pw.println();
                    if (I.write_total_log == true) I.Tot_pw.println();
                    //System.out.print((I.M[i][1] + " "));
                    //System.out.print((I.M[i][2] + " "));
                    //System.out.print((I.M[i][3] + " "));
                    //System.out.println();
                    }

            }  //End of if R2

            if (I.run_type.equals("C"))
            {
            I.write_total = true;
            String outdir2 = I.Outdir + "Cost";
            File dir2 = new File(outdir2);
            if (! dir2.exists()) dir2.mkdir();
            I.Costtab = tok.nextToken();
            I.Costtype = tok.nextToken();
            String cost_c = tok.nextToken();
            I.costchange = Integer.parseInt(cost_c);
            String skip_c = tok.nextToken();
            I.skip = Integer.parseInt(skip_c);
            String n_strong = tok.nextToken();
            I.N = Integer.parseInt(n_strong);
            //System.out.println(I.N);
            String s_strong = tok.nextToken();
            I.S = Integer.parseInt(s_strong);
            } //End of if C
    if (I.run_type.equals("B"))
            {
```

```
		if (I.write_total_setting ) I.write_total = true;
		else
		{
			if (I.run_number < 4 ) I.write_total = true;
			else I.write_total = false;
		}
		I.Costtab = tok.nextToken();
		//System.out.println(I.Outdir);
		I.Brutvalue = tok.nextToken();
		String outdir5 = I.Outdir + "Brute";
		File dir5 = new File(outdir5);
		if (! dir5.exists()) dir5.mkdir();
	File file5 = new File(dir5, I.Brutvalue);
	FileWriter fw = new FileWriter(file5);
	I.Brute_pw = new PrintWriter(fw);


		String Int_file_name = "InB" + I.Method + "R" + I.run_number;
		String outdir1 = I.Outdir + "Results";
		File dir1 = new File(outdir1);
		if (! dir1.exists()) dir1.mkdir();
	File file1 = new File(dir1, Int_file_name);
	fw = new FileWriter(file1);
	I.Int_pw = new PrintWriter(fw);
		PrintWriter pw = I.Int_pw;

	if (I.write_total == true)
	{
		String Tot_file_name = "ToB" + I.Method + "R" + I.run_number;
		File file2 = new File(I.Outdir + "Results", Tot_file_name);
		FileWriter fw2 = new FileWriter(file2);
		I.Tot_pw = new PrintWriter(fw2);
	}

		String n_strong = tok.nextToken();
		I.N = Integer.parseInt(n_strong);
		//System.out.println(I.N);
		pw.println("Run number = " + I.run_number);
		if (I.write_total == true) I.Tot_pw.println("Run number = " + I.run_number);
		pw.println("N= " + I.N);
		if (I.write_total == true) I.Tot_pw.println("N= " + I.N);
		String s_strong = tok.nextToken();
		I.S = Integer.parseInt(s_strong);
		pw.println("S= " + I.S);
		if (I.write_total == true) I.Tot_pw.println("S= " + I.S);
		I.M = new int[(I.S + 1)][4];
		//System.out.println("S= " + I.S);
		pw.println("Mk   Low   High");
		if (I.write_total == true) I.Tot_pw.println("Mk   Low   High");
		low = 1;
		for (int i = 1; i <= (I.S); i++)
			{
			temp = tok.nextToken();
			//System.out.println(temp);
			temp2 = Integer.parseInt(temp);
			//System.out.println(temp2);
```

```
                    I.M[i][1] = temp2;
                    I.M[i][2] = low;
                    high = low + temp2;
                    I.M[i][3] = (high - 1);
                    low = high;


                    pw.print((I.M[i][1] + " "));
                    if (I.write_total == true) I.Tot_pw.print((I.M[i][1] + " "));
                    pw.print(("     " + I.M[i][2] + " "));
                    if (I.write_total == true) I.Tot_pw.print(("     " + I.M[i][2] + " "));
                    pw.print(("     " + I.M[i][3]));
                    if (I.write_total == true) I.Tot_pw.print(("     " + I.M[i][3]));
                    pw.println();
                    if (I.write_total == true) I.Tot_pw.println();


              } //End of if B



 }

}

}
```

Main.java

```
// Main orogram
//import structure.ReadStream;
import java.io.*;                          //For io
import java.util.StringTokenizer;    //To tokenize each line of input.
import java.text.NumberFormat;     //To display the result in the currency format.


public class Main{
  public static void main(String[ ] args)
          throws IOException
   {

   Mult m = new Mult();
   m.Mult(m);
   }

}
```

Member.java

```
// Takes care of each member

import java.io.*;


public class Member
{
```

```
double fitness;
int Gen_created;
int order_created;
int Encoding_lenght;
int [ ] encoding;
int perm_length;
int [ ] perm;
//We add 1 to Encoding_lenght an S because we want to start from 1 not zero


public Member Newmember(Initial I,Member M,double [ ][ ] Costtable)
{
        //public Member M = new Member();
        M.Encoding_lenght = (I.S * 2);
        M.perm_length = I.S;
        M.encoding = new int[( M.Encoding_lenght + 1)];
        M.perm = new int [(M.perm_length + 1)];
        M.RandomEncoding(M,I);
        M.Encode(M);
        M.CalcFitness(M,Costtable);
        //M.ShowMember(M);
        return M;

}
//Check sysntax of int[ ]
public void RandomEncoding(Member M,Initial I)
        {

        int min = 1;
        int temp;
        int counter;
        Randm r = new Randm();
        //Notice below we only go to (S ) not (Encoding_lenght ) because second
        // half of encoding will have different min and max
        for (int i = 1; i <= M.perm_length ; i++)
        //Builds first half of encoding which is S long.
        {
                temp = r.Randint(I.M[i][2],I.M[i][3]); // because i starts at 1
                 M.encoding[i] = temp;
        }
        // THIS BEGINS SECOND HALF OF ENCODING:
        counter = 0;
        for (int i = (M.perm_length + 1); i <= M.Encoding_lenght; i++)
                //Builds second half of encoding which is S long( from (S+1) to 2s.
        {
                M.encoding[i] = r.Randint(min,(M.perm_length - counter));
                counter = counter + 1;
        }

}


public void Encode(Member M)
        {
        //int N = 1.N;
        //int S = 1.S;
```

```
            int min = 1;
            //int [(S + 1)] Etemp = new int[ ];
            int [ ] check = new int[(M.perm_length + 1)];
            int value;
            int position;
            int temp;
            int counter;
            for (int i = 1; i <= M.perm_length; i++)
            //Loads in all possibilities 1 to S for position
                        {
                        check[i]=i;
                        }
            counter = (M.perm_length + 1);  // 1 Where the position starts
            for (int i = 1; i <= M.perm_length; i++)
            //Builds first half of encoding which is S long.
                        {
                        value = M.encoding[i];
                        position = M.encoding[counter];
                        counter = counter + 1;
                        temp = check[position];
                        M.perm[temp] = value;
                        for (int j = position; j < (M.perm_length - (i - 1)); j++)
                        //Note:  if position = last value of Array loop will not run but last value
                        // gets reduced by 1 from using the expresion (S - (i - 1)).
            {
                                    check[j] = check[(j +1)];
                                                }
                        }
            }

public void CalcFitness(Member M, double [ ][ ] Costtable)
            {
                        //int time = 1;
                        double cost = 0;
             for (int i = 1; i <= M.perm_length  ; i++)
             {
                        cost = (cost + Costtable[(M.perm[i])][i]);
                        //System.out.println("Cost = " + cost);
                        //time = (time + 1);
             }
             M.fitness = cost;
            }

public void ShowMember(Member M)
 {
            System.out.print("Encoding = ");
            for (int i = 1; i <= M.Encoding_lenght; i++)
            {
                        System.out.print(M.encoding[i] + " ");
            }
            System.out.print("Permutation = ");
            for (int i = 1; i <= M.perm_length; i++)
            {
                                    System.out.print(M.perm[i] + " ");
            }
             System.out.print("  ");
```

```
                    System.out.print("Fitness = " + M.fitness);
 }



public void WriteMember(Member M,PrintWriter pw)
 {
        //System.out.print("Encoding = ");
        pw.print("Fitness = " + M.fitness);
        pw.print(" Gen_created = " + M.Gen_created);
        pw.print(" Order = " + M.order_created);
        pw.print(" Permutation = ");

        for (int i = 1; i <= M.perm_length; i++)
        {
                //System.out.println(i + " out of " + M.perm_length);
                        //System.out.print(M.perm[i] + " ");
                        pw.print(M.perm[i] + " ");
        }
         //System.out.print("  ");

        pw.print(" Encoding = ");
        for (int i = 1; i <= M.Encoding_lenght; i++)
        {
                //System.out.println(i + " out of " + M.Encoding_lenght);
                //System.out.print(M.encoding[i] + " ");
                pw.print(M.encoding[i] + " ");

        }

         //pw.print("  ");
                //System.out.print("Fitness = " + M.fitness);
                pw.println();
 }

public void Fillmember(Initial I,Member M,double [ ][ ] Costtable)
        {
        //public Member M = new Member();
        M.Encoding_lenght = (I.S * 2);
        M.perm_length = I.S;
        M.perm = new int [(M.perm_length + 1)];
        M.Encode(M);
        M.CalcFitness(M,Costtable);
        //M.ShowMember(M);
        //return M;
        }
}
```

Mult. Java

```
// To make Multiple runs
import java.io.*;
```

```
public class Mult

{



        public Sing sg = new Sing();
         public Initial I = new Initial();




        public void Mult()
                throws IOException
        {
        String current;
        I.run_number = 0;
        sg.row_counter = 0;
        I.GetOutDirName(I);
        String input = I.Outdir + "input.dat";


        BufferedReader s = new BufferedReader(new FileReader(input));

        current = s.readLine();
        I.run_number = I.run_number + 1;
        System.out.println(current);
        I.current = current;
        I.GetInitial(current,I);
        sg.WriteTitle(I);
        sg.Sing(I,sg);


        while ((current = s.readLine()) != null )
                        {

        I.run_number = I.run_number + 1;
        System.out.println(current);
        I.current = current;
        I.GetInitial(current,I);
        sg.Sing(I,sg);

                        }
        I.Sum_pw.close();
        I.Sum2_pw.close();
        }

}
```

Pdiff.java

```
import java.text.NumberFormat;
import java.io.*;
```

```
import java.util.Random;


public class PDiff
{
int [ ][ ] raw;
int [ ] count;
double [ ] mean;
double [ ] SD;
double [ ] SD2;
boolean [ ] bl;
int number = 11;
int numb_runs = 100;

public void Intialise(PDiff pd)
        {
        pd.raw = new int[(pd.number + 1)][(pd.numb_runs + 1)];
        pd.count = new int[(pd.number + 1)];
        pd.mean = new double[(pd.number + 1)];
        pd.SD = new double[(pd.number + 1)];
        pd.SD2 = new double[(pd.number + 1)];
        pd.bl  = new boolean[(pd.number + 1)];
        for (int i = 0; i <= pd.number; i++)
          {
          //System.out.print(i);
                pd.count[i] = 0;
                pd.mean[i] = 0;
                pd.SD[i] = 0;
                pd.SD2[i] = 0;
                for (int j = 0; j <= pd.numb_runs ; j++)
                        {
                        pd.raw[i][j] = 0;
                        }
          }
        }

public void init_bool(PDiff pd)
{
        for (int j = 0; j <= pd.number ; j++)
                        {
                        pd.bl[j] = false;
                        }
}


public int Percent_diff(PDiff pd,Population P)
{
double diff1 = 0;
double diff2 = 0;
int diff = 0;
Randm rand = new Randm();
double fitness;
double Brut = P.Brut_value;
if (P.gen_in_use == 1)
      {
        fitness = P.Gen[1].fitness;
```

```
        }
        else
        {
                        fitness = P.Gen2[1].fitness;
        }
diff1 = (((fitness - Brut)/Brut)*100);
diff2 = rand.Round(diff1,0);
diff = (int)(diff2);
return diff;
}

public void Pd(Population P,Initial I,PDiff pd,double val_to_check)
{
            //System.out.println("I.run_number =" + I.run_number);
            //System.out.println(" P.gen_number =" + P.gen_number );
            if (val_to_check > P.stop_value && P.dup_counter < I.max_dup_count)
            {
             if (P.gen_number == 0)
                        {
                        //pd.raw[2][I.r2_run_number] = pd.Percent_diff(pd,P);
                        pd.count[0] = pd.count[0] + 1;
                        pd.raw[0][pd.count[0]] = pd.Percent_diff(pd,P);
                        pd.bl[0] = true;
                        if (pd.raw[0][pd.count[0]] == 0)      pd.count[0] = pd.count[0] - 1;
                        }

            if (P.gen_number == 1)
                        {
                        //pd.raw[1][I.r2_run_number] = pd.Percent_diff(pd,P);
                        //System.out.println(" pd.raw[1][" + I.r2_run_number + "] = " +
pd.raw[1][I.r2_run_number]);
                        pd.count[1] = pd.count[1] + 1;
                        pd.raw[1][pd.count[1]] = pd.Percent_diff(pd,P);
                        pd.bl[1] = true;

                        }
            if (P.gen_number == 2)
                        {
                        //pd.raw[2][I.r2_run_number] = pd.Percent_diff(pd,P);
                        pd.count[2] = pd.count[2] + 1;
                        pd.raw[2][pd.count[2]] = pd.Percent_diff(pd,P);
                        pd.bl[2] = true;

                        }
            if (P.gen_number == 3)
                        {
                        //pd.raw[3][I.r2_run_number] = pd.Percent_diff(pd,P);
                        pd.count[3] = pd.count[3] + 1;
                        pd.raw[3][pd.count[3]] = pd.Percent_diff(pd,P);
                        pd.bl[3] = true;

                        }
            if (P.gen_number == 4)
                        {
                        //pd.raw[4][I.r2_run_number] = pd.Percent_diff(pd,P);
                        pd.count[4] = pd.count[4] + 1;
```

```
                pd.raw[4][pd.count[4]] = pd.Percent_diff(pd,P);
                pd.bl[4] = true;

                }
        if (P.gen_number == 5)
                {
        //pd.raw[5][I.r2_run_number] = pd.Percent_diff(pd,P);
                pd.count[5] = pd.count[5] + 1;
                pd.raw[5][pd.count[5]] = pd.Percent_diff(pd,P);
                pd.bl[5] = true;

                }
        if (P.gen_number == 10)
                {
                //pd.raw[6][I.r2_run_number] = pd.Percent_diff(pd,P);
                pd.count[6] = pd.count[6] + 1;
                pd.raw[6][pd.count[6]] = pd.Percent_diff(pd,P);
                pd.bl[6] = true;

                }
        if (P.gen_number == 20)
                {
                //pd.raw[7][I.r2_run_number] = pd.Percent_diff(pd,P);
                pd.count[7] = pd.count[7] + 1;
                pd.raw[7][pd.count[7]] = pd.Percent_diff(pd,P);
                pd.bl[7] = true;

                }
        if (P.gen_number == 30)
                {
                //pd.raw[8][I.r2_run_number] = pd.Percent_diff(pd,P);
                pd.count[8] = pd.count[8] + 1;
                pd.raw[8][pd.count[8]] = pd.Percent_diff(pd,P);
                pd.bl[8] = true;

                }
        if (P.gen_number == 40)
                {
                //pd.raw[9][I.r2_run_number] = pd.Percent_diff(pd,P);
                pd.count[9] = pd.count[9] + 1;
                pd.raw[9][pd.count[9]] = pd.Percent_diff(pd,P);
                pd.bl[9] = true;

                }
        if (P.gen_number == 50)
                {
                //pd.raw[10][I.r2_run_number] = pd.Percent_diff(pd,P);
                pd.count[10] = pd.count[10] + 1;
                pd.raw[10][pd.count[10]] = pd.Percent_diff(pd,P);
                pd.bl[10] = true;

                }
        }
else
        {
                if (val_to_check > P.stop_value)
```

```
                {
                        //pd.raw[11][I.r2_run_number] = pd.Percent_diff(pd,P);
                                pd.count[11] = pd.count[11] + 1;
                                pd.raw[11][pd.count[11]] = pd.Percent_diff(pd,P);
                                pd.bl[11] = true;

                }
        }
}
public void Print_out_raw(PDiff pd,Initial I,PrintWriter pw)
{
        //I.r2_run_number = I.r2_run_number - 1;
        //System.out.println("I.r2_run_number =" + I.r2_run_number);
        //pw.print(pd.raw[0][I.r2_run_number] +",");
        //pw.print(pd.raw[1][I.r2_run_number] +",");
        //pw.print(pd.raw[2][I.r2_run_number] +",");
        //pw.print(pd.raw[3][I.r2_run_number] +",");
        //pw.print(pd.raw[4][I.r2_run_number] +",");
        //pw.print(pd.raw[5][I.r2_run_number] +",");
        //pw.print(pd.raw[6][I.r2_run_number] +",");
        //pw.print(pd.raw[7][I.r2_run_number] +",");
        //pw.print(pd.raw[8][I.r2_run_number] +",");
        //pw.print(pd.raw[9][I.r2_run_number] +",");
        //pw.print(pd.raw[10][I.r2_run_number] +",");
        //pw.print(pd.raw[11][I.r2_run_number] +",");
        //I.r2_run_number = I.r2_run_number + 1;


        if (pd.bl[0]) pw.print(pd.raw[0][pd.count[0]] +",");
        else pw.print("0,");
        if (pd.bl[1]) pw.print(pd.raw[1][pd.count[1]] +",");
        else pw.print("0,");
        if (pd.bl[2]) pw.print(pd.raw[2][pd.count[2]] +",");
        else pw.print("0,");
        if (pd.bl[3]) pw.print(pd.raw[3][pd.count[3]] +",");
        else pw.print("0,");
 if (pd.bl[4]) pw.print(pd.raw[4][pd.count[4]] +",");
 else pw.print("0,");
        if (pd.bl[5]) pw.print(pd.raw[5][pd.count[5]] +",");
        else pw.print("0,");
        if (pd.bl[6]) pw.print(pd.raw[6][pd.count[6]] +",");
        else pw.print("0,");
        if (pd.bl[7]) pw.print(pd.raw[7][pd.count[7]] +",");
        else pw.print("0,");
        if (pd.bl[8]) pw.print(pd.raw[8][pd.count[8]] +",");
        else pw.print("0,");
        if (pd.bl[9]) pw.print(pd.raw[9][pd.count[9]] +",");
        else pw.print("0,");
        if (pd.bl[10]) pw.print(pd.raw[10][pd.count[10]] +",");
        else pw.print("0,");
        if (pd.bl[11]) pw.print(pd.raw[11][pd.count[11]] +",");
        else pw.print("0,");


}
public void Stat(PDiff pd,Initial I)
```

```
{
        Randm rand = new Randm();
        int ind_value;
        int nPlaces =2;
        double sum = 0;
        double squareSum =0;
        double initial_sd =0;
        double initial_sd2 =0;
        double initial_mean = 0;

        for (int i= 0; i <= pd.number; i++)
        {
                        for (int j= 0; j <= pd.count[i]; j++)
                        {
                                //System.out.println("i= " + i + ",j= " + j);
                                //System.out.println(" pd.raw[" + i + "][" + j + "] = " +
pd.raw[i][j]);
                                //if (pd.count[i] == 1) System.out.println("i= " + i + ",
pd.raw[" + i + "][" + j + "] = " + pd.raw[i][j]);
                                ind_value = pd.raw[i][j];
                                //System.out.println("ind_value =" + ind_value);
                                sum = sum + ind_value;
                                squareSum = squareSum + Math.pow(ind_value, 2);
                        }
                        initial_mean = sum/pd.count[i];
                        //if (pd.count[i] == 1) initial_mean = pd.raw[i][1];
                        pd.mean[i] = rand.Round(initial_mean,nPlaces);
                        //if (pd.count[i] == 1) System.out.println("pd.mean[i] = " +
pd.mean[i]);
                        initial_sd = Math.sqrt((squareSum - sum*sum/pd.count[i])/(pd.count[i]
- 1));
                        pd.SD[i] = rand.Round(initial_sd,nPlaces);
                        initial_sd2 = Math.sqrt((squareSum -
sum*sum/pd.count[i])/pd.count[i]);
                        pd.SD2[i] = rand.Round(initial_sd,nPlaces);
                        sum = 0;
                        squareSum =0;
                        initial_mean = 0;
                        initial_sd = 0;
                        initial_sd2 = 0;
        }
}
 public void printfinal(PDiff pd,PrintWriter pw)
 {
        for (int i= 0; i <= pd.number ; i++)
        {
        pw.print(pd.count[i] +",");
        pw.print(pd.mean[i] +",");
        pw.print(pd.SD[i] +",");
        pw.print(pd.SD2[i] +",");
        }

 }

 public void print_mean(PDiff pd,PrintWriter pw)
 {
```

```
            for (int i= 0; i <= pd.number ; i++)
            {

            pw.print(pd.mean[i] +",");

            }

 }
public void print_count(PDiff pd,PrintWriter pw)
 {
            for (int i= 0; i <= pd.number ; i++)
            {
            pw.print(pd.count[i] +",");

            }

 }
public void print_SD(PDiff pd,PrintWriter pw)
 {
            for (int i= 0; i <= pd.number ; i++)
            {

            pw.print(pd.SD[i] +",");

            }

 }
public void print_SD2(PDiff pd,PrintWriter pw)
 {
            for (int i= 0; i <= pd.number ; i++)
            {
            pw.print(pd.SD2[i] +",");
            }

 }


}           //End of Class
```

Population.java

```
// Takes care of the population

import java.util.StringTokenizer;
import java.io.*;


public class Population
{
static int numb_of_children;
int ratio;
int numb_of_Crossovers;
int numb_of_Mutations;
int pop_size;
int gen_size;
int gen_in_use;
```

```
int gen_number;
Member [ ] Gen;
Member [ ] Gen2;
int itemsInArray;
Member Last_top;
PrintWriter Gen_pw;
int dup_counter;
int gen_dup_started;
double stop_value;
double Brut_value;



public void SwitchGen(Population P)
        {
        if (P.gen_in_use == 1)
                        {
         //System.out.println(P.gen_in_use);
                        P.gen_in_use = 2;
    }
  else
                  {
    P.gen_in_use = 1;
                //System.out.println(P.gen_in_use);
                  }
        }

public void InitPop(Population P,Initial I,double [ ][ ] Costtable)
        {
        //Population P = new Population();
        P.initialvar(I,P);
        P.Gen = new Member[(gen_size + 1)];
        P.gen_in_use = 1;
        P.gen_number = 0;
        boolean dup_exist;

  //Start: to put first one in//
                Member M = new Member();
         M = M.Newmember(I,M,Costtable);
         M.Gen_created = P.gen_number;
  M.order_created = 1;
  P.itemsInArray = 1;
                P.insert(P,M);
                //End: to put first one in//
        for (int i = 2; i <= pop_size; i++)
                {
                dup_exist = true;
                while (dup_exist == true)
                  {
                        M = new Member();
                   M = M.Newmember(I,M,Costtable);
                         dup_exist = P.checkdup(P,M);
                  }
                M.Gen_created = P.gen_number;
  M.order_created = i;
  P.itemsInArray = i;
```

```
                        P.insert(P,M);
                        }
            Last_top.fitness =            P.Gen[1].fitness;
            }

            static void insert(Population P,Member newItem)
            {


    if (P.gen_in_use == 1)
                        {
    int itemsInArray = P.itemsInArray;
    int loc = itemsInArray - 1;  // Start at the end of the array.
    //System.out.println("itemsInArray= " + itemsInArray);
                                //          System.out.println("loc = " + loc + "Gen = " + P.gen_number + " "+
P.gen_in_use + " in use");


     /* Move items bigger than newItem up one space;
        Stop when a smaller item is encountered or when the
        beginning of the array (loc == 0) is reached. */

     while (loc > 0 && P.Gen[loc].fitness > newItem.fitness)
          {
            P.Gen[loc + 1] = P.Gen[loc];  // Bump item from A[loc] up to loc+1.
            loc = loc - 1;        // Go on to next location.
          }

     P.Gen[loc + 1] = newItem;  // Put newItem in last vacated space.
    }
         else
                        {
   int itemsInArray = P.itemsInArray;
   int loc = itemsInArray - 1;  // Start at the end of the array.
                                //System.out.println("itemsInArray= " + itemsInArray);
                                            //System.out.println("loc = " + loc);
                                //System.out.println("loc = " + loc + "Gen = " + P.gen_number + P.gen_in_use +
" in use");
     /* Move items bigger than newItem up one space;
        Stop when a smaller item is encountered or when the
        beginning of the array (loc == 0) is reached. */

    while (loc > 0 && P.Gen2[loc].fitness > newItem.fitness)
          {
            P.Gen2[loc + 1] = P.Gen2[loc];  // Bump item from A[loc] up to loc+1.
            loc = loc - 1;        // Go on to next location.
          }

    P.Gen2[loc + 1] = newItem;  // Put newItem in last vacated space.
   }
 }
public void NewGen(Population P,Initial I,double [ ][ ] Costtable)
 {
            int order_created = 0;
            int cross_remaining = P.numb_of_Crossovers;
            int mut_remaining = P.numb_of_Mutations;
```

```
        int mutcount=0;
        int crosscount=0;
        Cross_Mut X = new Cross_Mut();
         while ( cross_remaining > 0 || mut_remaining > 0 )
         {
                //System.out.println(cross_remaining + " " + mut_remaining);
                //System.out.println("P.ratio =" + P.ratio);
                        for (int j = 1; j <= P.ratio; j++)
                        {
                                //System.out.println(cross_remaining + " " + mut_remaining);
                                crosscount = crosscount + 1;
                                //System.out.println("crosscount =" + crosscount );
                                if (I.write_total_log == true) P.Gen_pw.print( crosscount + ") ");
                                if ( cross_remaining > 0)
                                        {
                                        Member Temp = new Member();
                                        Temp = X.Crossover(P,I,Costtable);
                                        cross_remaining = cross_remaining - 1;
                                        Temp.Gen_created = P.gen_number;
                                        order_created = order_created + 1;
                        Temp.order_created = order_created;
                                        P.itemsInArray = P.itemsInArray + 1;
                                        P.insert(P,Temp);
                                        }
                        }
                mutcount = mutcount + 1;
                if (I.write_total_log == true) P.Gen_pw.print(mutcount + ") ");
                //System.out.println("mut_remaining =" + mut_remaining);
                if (mut_remaining > 0 )
                        {
                        Member Temp = new Member();
                        Temp = X.Mutation(P,I,Costtable);
                        Temp.Gen_created = P.gen_number;
                        order_created = order_created + 1;
        Temp.order_created = order_created;
                        P.itemsInArray = P.itemsInArray + 1;
                        P.insert(P,Temp);
                        mut_remaining = mut_remaining - 1;
                        }

         }
 }

public void Grim(Population P)
 {
        if (P.gen_in_use == 1)
                        {
   P.Gen2 = new Member[(gen_size + 1)];
                        for (int i = 1; i <= P.pop_size; i++)
                                {
                                P.Gen2[i] = P.Gen[i];
                                }
                        P.SwitchGen(P);
                        P.itemsInArray = P.pop_size;
   }
 else
```

```
                        {
     P.Gen = new Member[(gen_size + 1)];
                              for (int i = 1; i <= P.pop_size; i++)
                                        {
                                        P.Gen[i] = P.Gen2[i];
                                        }
                              P.SwitchGen(P);
                              P.itemsInArray = P.pop_size;
    }
          }

public void SavTotalGen(Population P,Initial I)
          throws IOException
{
          //String Int_file_name = ("X" + I.Method + "R" + I.run_number + "G" + P.gen_number);
 //File file = new File(I.Outdir, Int_file_name);
 //FileWriter fw = new FileWriter(file);
 //P.Gen_pw = new PrintWriter(fw);
 Member M = new Member();
 if (P.gen_in_use == 1)
                              {
                              //P.Gen_pw.println();
    //P.Gen_pw.println("Generation number = " + P.gen_number);
    //P.Gen_pw.println("______________________________");
    I.Tot_pw.println();
    I.Tot_pw.println("Generation number = " + P.gen_number);
    I.Tot_pw.println("______________________________");

                              for (int i = 1; i <= P.itemsInArray; i++)
                                        {
                                        //P.Gen_pw.print(i + ") ");
                                        I.Tot_pw.print(i + ") ");
                                        //M.WriteMember(P.Gen[i],P.Gen_pw);
                                        M.WriteMember(P.Gen[i],I.Tot_pw);
                                        if (i == P.pop_size)
                                        {

                                                  //P.Gen_pw.println(" ______Generation will be cut here
______");
                                                  I.Tot_pw.println(" ______Generation will be cut here
______");
                                        }
                                                  //System.out.println(i + " out of " + P.itemsInArray);
                                        }
    }
 else
                        {
                        //P.Gen_pw.println();
                        //P.Gen_pw.println("Generation number = " + P.gen_number);
                        //P.Gen_pw.println("______________________________");
                        I.Tot_pw.println();
                        I.Tot_pw.println("Generation number = " + P.gen_number);
                        I.Tot_pw.println("______________________________");
                              for (int i = 1; i <= P.itemsInArray; i++)
                                        {
                                        //P.Gen_pw.print(i + ") ");
```

```
                                        I.Tot_pw.print(i + ") ");
                                        //M.WriteMember(P.Gen2[i],P.Gen_pw);
                                        M.WriteMember(P.Gen2[i],I.Tot_pw);
                                        if (i == P.pop_size)
                                        {
                                              //P.Gen_pw.println(" ______Generation will be cut here
______");
                                              I.Tot_pw.println(" ______Generation will be cut here
______");
                                        }
                                        //System.out.println(i + " out of " + P.itemsInArray );
                                        }
   }

 //P.Gen_pw.close();
 }

public void SavGen(Population P,Initial I)
      throws IOException
{

 Member M = new Member();
 if (P.gen_in_use == 1)
                              {
                              P.Gen_pw.println();
   P.Gen_pw.println("Generation number = " + P.gen_number);
   P.Gen_pw.println("____________________________");
   //I.Tot_pw.println();
   //I.Tot_pw.println("Generation number = " + P.gen_number);
   //I.Tot_pw.println("____________________________");

                              for (int i = 1; i <= P.itemsInArray; i++)
                                        {
                                        P.Gen_pw.print(i + ") ");
                                        //I.Tot_pw.print(i + ") ");
                                        M.WriteMember(P.Gen[i],P.Gen_pw);
                                        //M.WriteMember(P.Gen[i],I.Tot_pw);
                                        if (i == P.pop_size)
                                        {

                                              P.Gen_pw.println(" ______Generation will be cut here
______");
                                              //I.Tot_pw.println(" ______Generation will be cut here
______");
                                        }
                                              //System.out.println(i + " out of " + P.itemsInArray);
                                        }
   }
 else
                        {
                        P.Gen_pw.println();
                        P.Gen_pw.println("Generation number = " + P.gen_number);
                        P.Gen_pw.println("____________________________");
                        //I.Tot_pw.println();
                        //I.Tot_pw.println("Generation number = " + P.gen_number);
                        //I.Tot_pw.println("____________________________");
```

```
                              for (int i = 1; i <= P.itemsInArray; i++)
                                        {
                                        P.Gen_pw.print(i + ") ");
                                        //I.Tot_pw.print(i + ") ");
                                        M.WriteMember(P.Gen2[i],P.Gen_pw);
                                        //M.WriteMember(P.Gen2[i],I.Tot_pw);
                                        if (i == P.pop_size)
                                        {
                                                  P.Gen_pw.println(" ______Generation will be cut here
______");
                                                  //I.Tot_pw.println(" ______Generation will be cut here
______");
                                        }
                                        //System.out.println(i + " out of " + P.itemsInArray );
                                        }
     }


}


public void SavFinal(Population P,Initial I)
          throws IOException
{
          int Gen_to_print =3;
          Member M = new Member();
          if (P.gen_in_use == 1)
                              {
    //P.Gen_pw.println("Generation number = " + P.gen_number);
    //I.Int_pw.println("Generation number = " + P.gen_number);
                                        I.Int_pw.println("Final Results ");
                              for (int i = 1; i <= Gen_to_print; i++)
                                        {
                                        //P.Gen_pw.print(i + ") ");
                                        I.Int_pw.print(i + ") ");
                                        //M.WriteMember(P.Gen[i],P.Gen_pw);
                                        M.WriteMember(P.Gen[i],I.Int_pw);
                                                  //System.out.println(i + " out of " + P.itemsInArray);
                                        }
     }
 else
                    {
                    //P.Gen_pw.println("Generation number = " + P.gen_number);
                    //I.Int_pw.println("Generation number = " + P.gen_number);
                    I.Int_pw.println("Final Results ");
                              for (int i = 1; i <= Gen_to_print; i++)
                                        {
                                        //P.Gen_pw.print(i + ") ");
                                        I.Int_pw.print(i + ") ");
                                        //M.WriteMember(P.Gen2[i],P.Gen_pw);
                                        M.WriteMember(P.Gen2[i],I.Int_pw);
                                        //System.out.println(i + " out of " + P.itemsInArray );
                                        }
     }
}
```

```
public void ShowGen(Population P)
            throws IOException
{

 if (P.gen_in_use == 1)
                                    {
    //System,out.print("Generation number = " + P.gen_number);
                                    for (int i = 1; i <= P.itemsInArray; i++)
                                                {
                        //Here                  System.out.println("Generation number = " + P.gen_number + " ");
                                                P.Gen[i].ShowMember(P.Gen[i]);
                                                System.out.println();
                                                }
    }
 else
                          {
                                    for (int i = 1; i <= P.itemsInArray; i++)
                                                {
                                    //Here2     System.out.println("Generation number = " + P.gen_number + " ");
                                                P.Gen2[i].ShowMember(P.Gen2[i]);
                                                //System.out.println();
                                                }
    }

}



            public boolean checkdup(Population P,Member M)
            {
            boolean result= true;
                        for (int i = 1; i <= P.itemsInArray; i++)
                                                {
                                                            if (P.gen_in_use == 1)
                                    {
           if (P.Gen[i].fitness == M.fitness)
                                      {
            for (int j = 1; j <= M.perm_length; j++)
            {
             if (P.Gen[i].perm[j] != M.perm[j])
             {
                        result = false;
                        j = (M.perm_length + 1);
             }
            }
                                      }
                                                              else result = false;
         }
         else
                         {
          if (P.Gen2[i].fitness == M.fitness)
                                      {
            for (int j = 1; j <= M.perm_length; j++)
             {
             if (P.Gen2[i].perm[j] != M.perm[j])
               {
```

```
                        result = false;
                        j = (M.perm_length + 1);
             }
           }
                                   }
                                                   else result = false;
                        }
                                        }  //End of first loop
        //true means it is a duplicate.
        return result;
 }

 public void initialvar(Initial I,Population P)
 {
P.pop_size = I.pop_size;
P.numb_of_children = (I.gen_size - I.pop_size);
P.gen_size = I.gen_size;
double percent_mutations = I.percent_mutations;
double dec_mutations = (percent_mutations / 100);
double numb_of_mutations = (dec_mutations * P.numb_of_children);
Randm rand = new Randm();
double dec_mutations2 = rand.Round(numb_of_mutations,0);
P.numb_of_Mutations = (int)dec_mutations2;
//System.out.println(percent_mutations + " " + dec_mutations + " " + numb_of_mutations + " " +
dec_mutations2 + " " + P.numb_of_Mutations);
P.numb_of_Crossovers = (P.numb_of_children - P.numb_of_Mutations);

if (P.numb_of_Crossovers >= P.numb_of_Mutations) P.ratio =
(int)(P.numb_of_Crossovers/P.numb_of_Mutations);
else  P.ratio = (int)(P.numb_of_Mutations/P.numb_of_Crossovers);
//System.out.println(P.numb_of_Crossovers + " " + P.ratio);


 }

 public void Total(Population P,Initial I,double [ ][ ] Costtable)
        throws IOException
 {
        P.gen_number = 0;
        InitPop(P,I,Costtable);
  //P.ShowGen(P);
  // Here3 System.out.println("Generation number = " + P.gen_number + " ");
        P.SavTotalGen(P,I);
        Member M = new Member();
        for (int i = 1; i <=I.number_of_generations ; i++)
                   {
                    P.gen_number = i;
                    // Here4 System.out.println("Generation number = " + P.gen_number + " ");
                    String outdir2 = I.Outdir + "Output";
                    File dir2 = new File(outdir2);
                    if (! dir2.exists()) dir2.mkdir();
                    String Int_file_name = ("X" + I.Method + "R" + I.run_number + "G" + P.gen_number);
                    String outdir = I.Outdir + "Output\\Run" + I.run_number;
                    if (I.mult_run_type == 3) outdir = I.Outdir + "Output" + I.current_input_numb + "\\Run"
+ I.run_number;
                    File dir = new File(outdir);
```

```
                        if (! dir.exists()) dir.mkdir();
   File file = new File(dir, Int_file_name);

   FileWriter fw = new FileWriter(file);
   P.Gen_pw = new PrintWriter(fw);
   P.SavGen(P,I);

                        P.NewGen(P,I,Costtable);
                        P.SavTotalGen(P,I);
                        P.Grim(P);
                        P.Gen_pw.close();
                        if (P.gen_in_use == 1)
                                {
          //System.out.println
                                 I.Int_pw.print("Generation number = " + P.gen_number + " ");
                         M.WriteMember(P.Gen[1],I.Int_pw);
    }
 else
                         {
                         I.Int_pw.print("Generation number = " + P.gen_number + " ");
    M.WriteMember(P.Gen2[1],I.Int_pw);
                       //System.out.println(P.gen_in_use);
                         }

                        //P.ShowGen(P);
                       }
                       P.SavFinal(P,I);
                       //M.WriteMember(P.Gen2[i],I.Int_pw);
           }

public void Total_percent_of_brut(Population P,Initial I,double [ ][ ] Costtable,PDiff pd)
           throws IOException
{
           P.gen_number = 0;

           double val_to_check;

           String file_name;
           String outdir5 = I.Outdir + "Brute";
           file_name = outdir5 + "\\" + I.Brutvalue;
           BufferedReader s = new BufferedReader(new FileReader(file_name));
 String current = s.readLine();
           StringTokenizer tok = new StringTokenizer(current, ",");
           String n_Brute_value= tok.nextToken();
           P.Brut_value = Double.parseDouble(n_Brute_value);
           //Brut_value = Integer.parseInt(n_Brute_value);
           P.stop_value =(((I.percent_of_brut/100) * P.Brut_value) + P.Brut_value);
           //System.out.println("Stop value = " + P.stop_value);
           I.Int_pw.println();
           I.Int_pw.println("Brut value = " + P.Brut_value);
           I.Int_pw.println("Stop value = " + P.stop_value);
           I.Int_pw.println();
           P.Last_top = new Member();
           P.InitPop(P,I,Costtable);
           P.dup_counter = 0;
 //P.ShowGen(P);
```

```
//System.out.println("Generation number = " + P.gen_number + " ");
        P.SavTotalGen(P,I);
        val_to_check = P.Gen[1].fitness;
        pd.Pd(P,I,pd,val_to_check);
        Member M = new Member();
        while (val_to_check > P.stop_value && P.dup_counter < I.max_dup_count)
                {
                P.gen_number = P.gen_number + 1;
                //System.out.println("Generation number = " + P.gen_number + " ");

                if (I.write_total_log == true)
        {
                // Here6 System.out.println("Generation number = " + P.gen_number + " ");
                String outdir2 = I.Outdir + "Output";
                if (I.mult_run_type == 3) outdir2 = I.Outdir + "Output" + I.current_input_numb;
                File dir2 = new File(outdir2);
                if (! dir2.exists()) dir2.mkdir();
                String Int_file_name = ("X" + I.Method + "R" + I.run_number + "G" + P.gen_number);
                String outdir = I.Outdir + "Output\\Run" + I.run_number;
                if (I.mult_run_type == 3) outdir = outdir2 + "\\Run" + I.temp_run_number;
                File dir = new File(outdir);
                if (! dir.exists()) dir.mkdir();
  File file = new File(dir, Int_file_name);

  FileWriter fw = new FileWriter(file);
  P.Gen_pw = new PrintWriter(fw);
  P.SavGen(P,I);
        }
                P.NewGen(P,I,Costtable);
                if (I.write_total_log == true) P.SavTotalGen(P,I);
                //pd.Pd(P,I,pd);
                P.Grim(P);
                if (I.write_total_log == true) P.Gen_pw.close();
                if (P.gen_in_use == 1)
                        {
         //System.out.println
                        if (I.write_total_log == true)
                                {
                                I.Int_pw.print("Generation number = " + P.gen_number + " ");
                        M.WriteMember(P.Gen[1],I.Int_pw);
                        val_to_check = P.Gen[1].fitness;
                                }
   }
else
                {
                        if (I.write_total_log == true)
                                {
                        I.Int_pw.print("Generation number = " + P.gen_number + " ");
        M.WriteMember(P.Gen2[1],I.Int_pw);
                                //System.out.println(P.gen_in_use);
                                }
   val_to_check = P.Gen2[1].fitness;
                }
                P.check_duplicate(P);
                pd.Pd(P,I,pd,val_to_check);
                //P.ShowGen(P);
```

```
                }
                P.SavFinal(P,I);
                if (P.dup_counter == I.max_dup_count && I.write_total_log == true)
I.Int_pw.println("Stopped at " + I.max_dup_count);
                if (P.dup_counter == I.max_dup_count) I.Tot_pw.println("Stopped at " +
I.max_dup_count);
                if (P.dup_counter == I.max_dup_count) System.out.println("Stopped at " +
I.max_dup_count);
                //M.WriteMember(P.Gen2[i],I.Int_pw);
                        //pd.Pd(P,I,pd);
        }

        public void check_duplicate(Population P)
        {
                if (P.gen_in_use == 1)
                        {
    if (P.Gen[1].fitness == Last_top.fitness)
    {
        P.dup_counter = P.dup_counter + 1;
    }
    else
    {
    P.Last_top.fitness =   P.Gen[1].fitness;
    P.dup_counter = 0;
    P.gen_dup_started = P.gen_number;
    }
   }
        else
                 {
   if (P.Gen2[1].fitness ==Last_top.fitness)
    {
        P.dup_counter = P.dup_counter + 1;
    }
    else
    {
    P.Last_top.fitness = P.Gen2[1].fitness;
    P.dup_counter = 0;
    P.gen_dup_started = P.gen_number;
    }

                 }
        }

} //End of class
```

Randm.java

```
//Random Generator
import java.util.Random;



public class Randm
{

        Random random = new Random ();
```

```
        //Randm rand = new Randm();

    public int Randint()
        {
      int number = random.nextInt ();
        return number;
        }

        public int Randint(int lowerLimit, int upperLimit)
        {
        double initial = Math.random();
        int range = upperLimit - lowerLimit + 1;
        int number = (int)( (initial * range) + lowerLimit );
        if (range == 0) return lowerLimit;
        return number;
        }

    public double Randdouble()
        {
     double initial = Math.random();
     return initial;
                        }

        public double Randdouble(double lowerLimit, double upperLimit)
        {
        double initial = Math.random();
        //System.out.println ("initial =  "+ initial);
        double range = upperLimit - lowerLimit;
        double number = ( (initial * range) + lowerLimit );
        if (range == 0) return lowerLimit;
        return number;
                        }

                        public double Randdouble(int nPlaces)
        {
     Randm rand = new Randm();
        double initial = Math.random();
     double initial2 = rand.Round(initial,nPlaces);
     return initial2;
          }

        public double Randdouble(double lowerLimit, double upperLimit, int nPlaces)
        {
        Randm rand = new Randm();
        double initial = Math.random();
          //System.out.println ("initial =  "+ initial);
        double range = upperLimit - lowerLimit;
        double number = ( (initial * range) + lowerLimit );
        if (range == 0) number = lowerLimit;
        double number2 = rand.Round(number,nPlaces);
        return number2;
          }

          public static double Round(double fNum, int nPlaces)
                {
        double fMult = Math.pow(10, (double) nPlaces);
```

```
                        return (Math.round(fNum * fMult) / fMult);
                          }

                public void Testint ()
               {
      int dplaces = 2;
               int a = 2;
      int b = 15;
      double c = 2.7;
      double d = 30.6;
                        Randm r = new Randm();
                        int n = r.Randint();
                        System.out.println (n);
                        n = r.Randint(a, b);
                        System.out.println (n);
                        int m = r.Randint(a, b);
                        System.out.println (m);
                        double j = r.Randdouble();
                        System.out.println (j);
                        double k = r.Randdouble(c, d);
                        System.out.println (k);
                        double o = r.Round(k,dplaces);
                        System.out.println (o);
                        dplaces = 3;
                        o = r.Round(k,dplaces);
                        System.out.println (o);
                        dplaces = 2;
                        double l = r.Randdouble(dplaces);
                          System.out.println (l);
                          double q = r.Randdouble(c, d, dplaces);
                          System.out.println (q);
                          dplaces = 3;
                          l = r.Randdouble(dplaces);
                          System.out.println (l);
                          q = r.Randdouble(c, d, dplaces);
                          System.out.println (q);

               }
}
```

Sing.java

```
// Put what is to be done for a single run
               import java.io.*;

public class Sing
{
int row_counter;
double Brut_fitness;

               public void Sing(Initial I,Sing sg,Stat st,Mult m,PDiff pd)
                        throws IOException
               {

               int rows = I.N;
               int cols = I.S;
```

```
        int Encoding_lenght = (I.S * 2);
        Cost c = new Cost();
        //Member M = new Member();

        //st.countpop[pos] = 0;

        if (I.run_type.equals("R1"))
        {
        Population P = new Population();
        //Inout i = new Inout();
        //Intial I = new Initial();

        //System.out.println (run_number);
        c.Costtable = c.ReadCost(I);
        c.AddInit(c.Costtable,I.Int_pw);
        c.AddInit(c.Costtable,I.Tot_pw);
        //M.Newmember(I,M,c.Costtable);
        //c.ShowCost(c.Costtable);
        P.Total(P,I,c.Costtable);

        I.Int_pw.close();
        I.Tot_pw.close();

        I.Sum_pw.print(I.current + ",");
        //I.Sum_pw.print(",");
        I.Sum_pw.print(I.run_number + ",");
        I.Sum_pw.print(I.run_type + ",");
        I.Sum_pw.print(I.S + ",");
        I.Sum_pw.print(I.N + ",");
        if (P.gen_in_use == 1) I.Sum_pw.print(P.Gen[1].fitness + ",");
        else      I.Sum_pw.print(P.Gen[1].fitness + ",");
        I.Sum_pw.println(P.gen_number);

}

if (I.run_type.equals("R2"))
        {
        I.r2_run_number = I.r2_run_number + 1;
        Population P = new Population();

        //System.out.println (run_number);
        c.Costtable = c.ReadCost(I);
        c.AddInit(c.Costtable,I.Int_pw);
        c.AddInit(c.Costtable,I.Tot_pw);

        pd.init_bool(pd);
        P.Total_percent_of_brut(P,I,c.Costtable,pd);

        I.Sum_pw.print(I.current);
        I.Sum_pw.print(P.stop_value + ",");
        //I.Sum_pw.print(",");
        I.Sum_pw.print(I.run_number + ",");
        I.Sum_pw.print(I.run_type + ",");
        I.Sum_pw.print(I.S + ",");
        I.Sum_pw.print(I.N + ",");
        if (P.gen_in_use == 1) I.Sum_pw.print(P.Gen[1].fitness + ",");
```

```
else        I.Sum_pw.print(P.Gen2[1].fitness + ",");
I.Sum_pw.print(P.gen_number + ",");
if (P.dup_counter == I.max_dup_count)
            {
                        I.Sum_pw.print(P.gen_dup_started + ",");
                        I.Sum_pw.print("Stopped at " + I.max_dup_count + ",");
                        st.countpop[3] = st.countpop[3]  + 1;
            }
 else I.Sum_pw.print(",,");
            pd.Print_out_raw(pd,I,I.Sum_pw);

            //Next Lines set title for Summary2
            sg.row_counter = sg.row_counter + 1;
            // Stat begin
            st.statpop[1][sg.row_counter] = P.gen_number;
            st.countpop[1] = sg.row_counter;
            //Stat end

            I.Sum2_pw.print(sg.row_counter + ",");
            I.Sum2_pw.print(I.run_number + ",");
            I.Sum2_pw.print(I.Costtype + ",");
            I.Sum2_pw.print(I.pop_size + ",");
            I.Sum2_pw.print(I.gen_size + ",");
            I.Sum2_pw.print(I.percent_mutations + ",");
            I.Sum2_pw.print(I.N + ",");
            I.Sum2_pw.print(I.S + ",");

            for (int i = 1; i <= (I.S); i++)
                        {

                        I.Sum2_pw.print(I.M[i][1] + ",");
                        }
I.Sum2_pw.print(P.Brut_value + ",");
if (P.gen_in_use == 1) I.Sum2_pw.print(P.Gen[1].fitness + ",");
else        I.Sum2_pw.print(P.Gen2[1].fitness + ",");
I.Sum2_pw.print(P.gen_number + ",");
if (P.dup_counter == I.max_dup_count)
            {
                        st.statpop[2][sg.row_counter] = P.gen_dup_started;
                        st.countpop[2] = st.countpop[2]  + 1;
                        //I.Sum2_pw.print(",");
                        I.Sum2_pw.print(P.gen_dup_started + ",");
                        I.Sum2_pw.print(P.gen_dup_started + ",");


                        //I.Sum2_pw.print(P.gen_dup_started + ",");
                        //I.Sum2_pw.print("Stopped at " + I.max_dup_count);
            }
else
{
            st.statpop[2][sg.row_counter] = P.gen_number;
            st.countpop[2] = st.countpop[2]  + 1;
            I.Sum2_pw.print(",");
            I.Sum2_pw.print(P.gen_number + ",");
}
```

```
            pd.Print_out_raw(pd,I,I.Sum2_pw);

            I.Sum2_pw.println();
            I.Sum_pw.println();
            I.Int_pw.close();
            I.Tot_pw.close();
            }  //End if R2

  if (I.run_type.equals("C"))
  {
  if (I.Costtype.equals("RA")) c.Costtable = c.RAMakeCost(I);
  if (I.Costtype.equals("SI")) c.Costtable = c.SI(I);
  if (I.Costtype.equals("SD")) c.Costtable = c.SD(I);
  if (I.Costtype.equals("MM")) c.Costtable = c.MM(I);
  if (I.Costtype.equals("RM")) c.Costtable = c.RM(I);
  c.WriteCost(c.Costtable,I);

  I.Sum_pw.print(I.run_number + ",");
  I.Sum_pw.print(I.run_type + ",");
  I.Sum_pw.print(I.Costtype + ",");
  //I.Sum_pw.print(",");
  }  //End if C

            if (I.run_type.equals("B"))
  {
  Brut B = new Brut();
  c.Costtable = c.ReadCost(I);
  c.AddInit(c.Costtable,I.Int_pw);
            if (I.write_total == true) c.AddInit(c.Costtable,I.Tot_pw);
  B.Brut(B,I,c.Costtable);

            I.Int_pw.close();
            if (I.write_total == true) I.Tot_pw.close();
            I.Brute_pw.close();
            I.Sum_pw.print(I.run_number + ",");
            I.Sum_pw.print(I.run_type + ",");
            I.Sum_pw.print(I.S + ",");
            I.Sum_pw.print(I.N + ",");
            sg.Brut_fitness = B.fitness;
            I.Sum_pw.print(B.fitness + ",");
            //I.Sum_pw.print(",");
            //if (P.gen_in_use == 1) I.Sum_pw.print(P.Gen[1].fitness + ",");
            //else      I.Sum_pw.print(P.Gen[1].fitness + ",");
  } //End if B


            }
public void WriteTitle(Initial I)
            throws IOException
                        {
                        String Summary_file_name = "S" + I.S + I.Costtype + "R1.su1";
                        if (I.mult_run_type == 3) Summary_file_name = "S" + I.S + I.Costtype + "R" +
I.current_input_numb + ".su1";
                        String outdir1a = I.Outdir;
                        File dir1a = new File(outdir1a);
                        if (! dir1a.exists()) dir1a.mkdir();
```

```
            File file1a = new File(dir1a, Summary_file_name);
            FileWriter fw1a = new FileWriter(file1a);
            I.Sum_pw = new PrintWriter(fw1a);

            String Summary_file2_name ="S" + I.S + I.Costtype + "R1.sum";
            if (I.mult_run_type == 3) Summary_file2_name ="S" + I.S + I.Costtype + "R"  +
I.current_input_numb + ".sum";

                        String outdir2a = I.Outdir;
                        File dir2a = new File(outdir2a);
                        if (! dir2a.exists()) dir2a.mkdir();
            File file2a = new File(dir2a, Summary_file2_name);
            FileWriter fw2a = new FileWriter(file2a);
            I.Sum2_pw = new PrintWriter(fw2a);



            // next lines are for top lines for Sumart
            I.Sum_pw.print(",S = " + I.S + " ");
            //System.out.println("I.Costtype=" + I.Costtype);
                        if (I.Costtype.equals("RA")) I.Sum_pw.print("Random Numbers Cost Table ");
                        if (I.Costtype.equals("SI")) I.Sum_pw.print("Slightly Increasing Cost Table ");
            if (I.Costtype.equals("SD")) I.Sum_pw.print("Slightly Decreasing Cost Table ");
            if (I.Costtype.equals("MM")) I.Sum_pw.print("Multi Modal Cost Table ");
            if (I.Costtype.equals("RM")) I.Sum_pw.print("Random Multi Modal Cost Table ");
                        //I.Sum_pw.print(I.percent_mutations + " percent Mutation Rate");
            I.Sum_pw.print("  percent Mutation Rate");
            I.Sum_pw.println();
            //System.out.println("out6");

                        // next lines are for top lines
                        I.Sum2_pw.print(",S = " + I.S + " ");
                        if (I.Costtype.equals("RA")) I.Sum2_pw.print("Random Numbers Cost Table ");

                        if (I.Costtype.equals("SI")) I.Sum2_pw.print("Slightly Increasing Cost Table ");
            if (I.Costtype.equals("SD")) I.Sum2_pw.print("Slightly Decreasing Cost Table ");
            if (I.Costtype.equals("MM")) I.Sum2_pw.print("Multi Modal Cost Table ");
            if (I.Costtype.equals("RM")) I.Sum2_pw.print("Random Multi Modal Cost Table ");
                        I.Sum2_pw.print(I.percent_mutations + " percent Mutation Rate");
                        I.Sum2_pw.println();

                                    //Next Lines set title for Summary

            I.Sum_pw.print("Run Number,");
            I.Sum_pw.print("Run type,");
            I.Sum_pw.print("Costtable Type,");
            I.Sum_pw.print("Run Number,");
                        I.Sum_pw.print("Run Type,");
                        I.Sum_pw.print("S,");
                        I.Sum_pw.print("N,");
                        I.Sum_pw.print("Best Brute Force Value,");
            //I.Sum_pw.print(I.current + ",");
                        I.Sum_pw.print("Run Number,");
                        I.Sum_pw.print("Run Type,");
                        I.Sum_pw.print("Cost Table Name,");
                        I.Sum_pw.print("Brut Force Table Name,");
```

```
		I.Sum_pw.print("Precent allowed to come within Brute,");
		I.Sum_pw.print("Population Size,");
		I.Sum_pw.print("Generation Size,");
		I.Sum_pw.print("Percent Mutation Rate,");
		I.Sum_pw.print("N,");
		I.Sum_pw.print("S,");
		for (int i = 1; i <= (I.S); i++)
			{

			I.Sum_pw.print("MK" + i + ",");
			}
		I.Sum_pw.print("Stop Value,");
	//I.Sum_pw.print(",");
		I.Sum_pw.print("Run Number,");
		I.Sum_pw.print("Run Type,");
		I.Sum_pw.print("S,");
		I.Sum_pw.print("N,");
		I.Sum_pw.print("Best Fitness,");
		I.Sum_pw.print("Number of Generations,");
		I.Sum_pw.print("Generations leveling Started,");
		I.Sum_pw.print("Whether run stopped,");
		I.Sum_pw.print("Percent Difference from Brut after 0 Generation,");
		I.Sum_pw.print("Percent Difference from Brut after 1 Generation,");
		I.Sum_pw.print("Percent Difference from Brut after 2 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 3 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 4 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 5 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 10 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 20 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 30 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 40 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after 50 Generations,");
		I.Sum_pw.print("Percent Difference from Brut after stopped,");
		I.Sum_pw.println();
		//Next Lines set title for Summary2
		I.Sum2_pw.print("#,");
		I.Sum2_pw.print("Run Number,");
		I.Sum2_pw.print("Cost Table Type,");
		I.Sum2_pw.print("Population Size,");
		I.Sum2_pw.print("Generation Size,");
		I.Sum2_pw.print("Precent Mutation Rate,");
		I.Sum2_pw.print("N,");
		I.Sum2_pw.print("S,");
		for (int i = 1; i <= (I.S); i++)
			{

			I.Sum2_pw.print("MK" + i + ",");
			}

		I.Sum2_pw.print("Best Brute Force Value,");
		I.Sum2_pw.print("Best Fitness,");
		I.Sum2_pw.print("Number of Generations,");
		I.Sum2_pw.print("Generation Leveling off Started,");
		I.Sum2_pw.print("Adjusted Number of Generations,");
		I.Sum2_pw.print("Percent Difference from Brut after 0 Generation,");
		I.Sum2_pw.print("Percent Difference from Brut after 1 Generation,");
```

```
                I.Sum2_pw.print("Percent Difference from Brut after 2 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 3 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 4 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 5 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 10 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 20 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 30 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 40 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after 50 Generations,");
                I.Sum2_pw.print("Percent Difference from Brut after stopped,");


                I.Sum2_pw.println();

                }
}
```

Stat.java

```
import java.text.NumberFormat;
import java.io.*;
import java.util.Random;



public class Stat
{

int [ ][ ] statpop;
int [ ] countpop;
double [ ] mean;
double [ ] sd;
double [ ] sd2;
int [ ] min;
int [ ] max;
int [ ] count;
int [ ] eq0;
int [ ] le5;
int [ ] f5to10;
int [ ] f10to15;
int [ ] ab15;
int number = 3;
int numb_runs = 100;


public void Intialise(Stat st)
                                        {
st.statpop = new int[st.number][st.numb_runs];
st.countpop = new int[(st.number + 1)];
for (int i=1; i < (st.number + 1); i++)
{
st.countpop[i] = 0;
}
                st.mean = new double[st.number];
                 st.sd = new double [st.number];
```

```
                        st.sd2 = new double [st.number];
                        st.min = new int[st.number];
                        st.max = new int[st.number];
                        //st.count = new int[number];
                        st.eq0 = new int[st.number];
                        st.le5 = new int[st.number];
                        st.f5to10 = new int[st.number];
                        st.f10to15 = new int[st.number];
                        st.ab15 = new int[st.number];
                        }

public void stats(Stat st,int pos)
                        {
                        Randm rand = new Randm();
                        int ind_value;
                        int nPlaces =2;
                        double sum = 0;
                        double squareSum =0;
                        double initial_sd =0;
                        double initial_mean = 0;
                        st.min[pos] = 99999;
                        st.max[pos] = 0;
                        st.eq0[pos] = 0;
                        st.le5[pos] = 0;
                        st.f5to10[pos] = 0;
                        st.f10to15[pos] = 0;
                        st.ab15[pos] = 0;
                        int count = st.countpop[pos];
                        if (st.countpop[pos] > 1)
                        {
                                    for (int i= 1; i <= st.countpop[pos]; i++)
                                                {

                         ind_value = st.statpop[pos][i];
                       //System.out.println("ind_value =" + ind_value);
sum = sum + ind_value;
squareSum = squareSum + Math.pow(ind_value, 2);
if (ind_value < st.min[pos]) st.min[pos] = ind_value;
                        if (ind_value > st.max[pos]) st.max[pos] = ind_value;
                        if (ind_value == 0) st.eq0[pos] = st.eq0[pos] + 1;
                        if (ind_value >= 1 && ind_value <= 5) st.le5[pos] = st.le5[pos] + 1;
                        if (ind_value > 5 && ind_value <= 10) st.f5to10[pos] = st.f5to10[pos] + 1;
                        if (ind_value > 10 && ind_value <= 15) st.f10to15[pos] = st.f10to15[pos] + 1;
                        if (ind_value > 15) st.ab15[pos] = st.ab15[pos] + 1;
                           }
                        initial_mean = sum/st.countpop[pos];
                        st.mean[pos] = rand.Round(initial_mean,nPlaces);
                        initial_sd = Math.sqrt((squareSum - sum*sum/count)/(count - 1));
                        st.sd[pos] = rand.Round(initial_sd,nPlaces);
                        initial_sd = Math.sqrt((squareSum - sum*sum/count)/count);
                        st.sd2[pos] = rand.Round(initial_sd,nPlaces);



                        //System.out.println(st.countpop[pos] + ",");
                        //System.out.println(st.mean[pos] + ",");
                        //System.out.println(st.sd[pos] + ",");
```

```
                    //System.out.println(st.min[pos] + ",");
                     //System.out.println(st.max[pos] + ",");
                    //System.out.println(st.le5[pos] + ",");
                    //System.out.println(st.f5to10[pos] + ",");
                    //System.out.println(st.f10to15[pos] + ",");
                    //System.out.println(st.ab15[pos] + ",");

                    }
}

public void save(Stat st,int pos,PrintWriter pw)
                    {
                    pw.print(st.countpop[pos] + ",");
                    pw.print(st.mean[pos] + ",");
                    pw.print(st.sd[pos] + ",");
                    pw.print(st.sd2[pos] + ",");
                    pw.print(st.min[pos] + ",");
  pw.print(st.max[pos] + ",");
  pw.print(st.eq0[pos] + ",");
                    pw.print(st.le5[pos] + ",");
                    pw.print(st.f5to10[pos] + ",");
                    pw.print(st.f10to15[pos] + ",");
                    pw.print(st.ab15[pos] + ",");
                    }

                    public void save_sum(Stat st,PrintWriter pw,PDiff pd)
                    {
                    pw.println();
                    st.fill(pw);
                    pw.print("Count,,");
                    pw.print(st.countpop[1] + ",");
                    pw.print(st.countpop[3] + ",");
                    pw.print(st.countpop[2] + ",");
                    pd.print_count(pd,pw);
                    pw.println();
                    st.fill(pw);
                    pw.print("Mean,,");
                    pw.print(st.mean[1] + ",,");
                    pw.print(st.mean[2] + ",");
                    pd.print_mean(pd,pw);
                    pw.println();
                    st.fill(pw);
                    pw.print("SD(N-1),,");
                    pw.print(st.sd[1] + ",,");
                    pw.print(st.sd[2] + ",");
                    pd.print_SD(pd,pw);
                    pw.println();
                    st.fill(pw);
                    pw.print("SD(N),,");
                    pw.print(st.sd2[1] + ",,");
                    pw.print(st.sd2[2] + ",");
                    pd.print_SD2(pd,pw);
                    pw.println();
                    st.fill(pw);
                    pw.print("Min,,");
                    pw.print(st.min[1] + ",,");
```

```
                    pw.print(st.min[2] + ",");
                    pw.println();
                    st.fill(pw);
                    pw.print("Max,,");
                    pw.print(st.max[1] + ",,");
 pw.print(st.max[2] + ",");
 pw.println();
                    st.fill(pw);
                    pw.print(" = 0,,");
                    pw.print(st.eq0[1] + ",,");
                    pw.print(st.eq0[2] + ",");
                    pw.println();
                    st.fill(pw);
                    pw.print("< 5,,");
                    pw.print(st.le5[1] + ",,");
                    pw.print(st.le5[2] + ",");
                    pw.println();
                    st.fill(pw);
                    pw.print("5 to 10,,");
                    pw.print(st.f5to10[1] + ",,");
                    pw.print(st.f5to10[2] + ",");
                    pw.println();
                    st.fill(pw);
                    pw.print("10 to 15,,");
                    pw.print(st.f10to15[1] + ",,");
                    pw.print(st.f10to15[2] + ",");
                    pw.println();
                    st.fill(pw);
                    pw.print("> 15,,");
                    pw.print(st.ab15[1] + ",,");
                    pw.print(st.ab15[2] + ",");
                    }

                    public void fill(PrintWriter pw)
                    {
                    int numb = 12;
                    for (int i= 1; i <= numb; i++)
                                    {
                               pw.print(",");
                                    }
                                    }


public void total(Stat st)
                                    {
                                 int pos;
                              //st.Intialise(st);
                                 pos = 1;
                              st.stats(st,pos);
                                 pos = 2;
                              st.stats(st,pos);
                                    }


}
```