1-1-2006

An application of a genetic algorithm to retail staff scheduling

Maryam Khashayardoust
Ryerson University

AN APPLICATION OF A GENETIC ALGORITHM TO RETAIL STAFF SCHEDULING

by

Maryam Khashayardoust

B.Sc. in Industrial Engineering
University of Science and Technology, Tehran, 1999

A thesis presented to
Ryerson University
in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in
Mechanical Engineering

Toronto, Ontario, Canada

DECLARATION

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend my thesis to other institutions or individuals for the purpose of scholarly research.

Maryam Khashayardoust

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Maryam Khashayardoust

ABSTRACT

AN APPLICATION OF A GENETIC ALGORITHM TO RETAIL STAFF SCHEDULING

Maryam Khashayardoust

Master of Applied Science in Mechanical Engineering

Ryerson University, 2006

Staff scheduling has received increasing attention over the past few years because of its widespread use, economic significance and difficulty of solution. For most organizations, the ability to have the right staff on duty at the right time is a critically important factor when attempting to satisfy their customers' requirements. The purpose of this study is to develop a genetic algorithm (GA) for the retail staff scheduling problem, and investigate its effectiveness. The proposed GA is compared with the conventional, linear integer programming approach. The GA is tested on a set of six real-world problems. Three are tested using a range of population size and mutation rate parameters. Then all six are solved with the best of those parameters. The results are compared to those obtained with the branch-and-bound algorithm. It is shown that GA can produce near-optimal solutions for all of the problems, and for half of them, it is more successful than the branch-and-bound method.

ACKNOWLEDGMENT

The following thesis benefited from the guidance and assistance of several people. First, I would like to thank Dr. Saeed Zolfaghari, my supervisor. His direction, instructive comments and support were invaluable in allowing me to complete this project. I would also like to thank Dr. Ahmed El-bouri for his time and support.

A special note of thanks goes to Communications and Information Technology Ontario (CITO), Workbrain, Inc., and Ryerson University. This thesis would have been impossible without their funding and support.

Many of the staff at Workbrain, Inc. shared their knowledge and time. In particular, I would like to thank Dr. Vinh Quan, Eddie Hsu, Brett Gersekowski, Alan Owens and Igor Lopata.

Finally, I'd like to express my love and gratitude to my parents. They have always encouraged and supported me, especially in my education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

LP	Linear Programming
IP	Integer Programming
MIP	Mixed Integer Programming
IGP	Integer Goal Programming
GA	Genetic Algorithm
SA	Simulated Annealing
TS	Tabu Search
N	chromosome length (No. of employees)
P	population size
m	mutation rate
RV_1	response variable I: GA optimal objective value
RV_2	response variable II: GA search time
E	set of employees
SE	salaried employees' subset
FE	fixed-shift employees' subset
NF	non-fixed-shift employees' subset
NFS	non-fixed-shift and non-salaried employees' subset
CH	set of choices in the GA solution
ch_i	set of choices for employee i in the GA solution
D	set of days
I	set of intervals
J	set of jobs
S	set of shifts in the GA solution (chromosome)
d	day index
i	employee index
j	shift index
k	job index
n	constraint index

t	interval index
z	schedule index
e_i	employee i
L_j	length of shift j expressed in interval
A_{ij}	1, if employee i is available for shift j; 0, otherwise
X_{ij}	1, if employee i is assigned to shift j; 0, otherwise
T_{dj}	1, if shift j happens on day d; 0, otherwise
U_{ij}	1, if employee i has the required skill for shift j; 0, otherwise
$T_i^{\min}$	minimum intervals per week for fixed-shift employee i (expressed in interval)
P_n	penalty for violating soft constraint n
a	penalty for total hours worked by all employees
P_7	unit penalty for violating the C_7 constraint
P_9	unit penalty for violating the C_9 constraint
P_{10}	penalty per interval for giving salaried employees less than their required intervals per schedule (unit penalty for violating the C_{10} constraint)
$P_{10'}$	penalty per interval for giving salaried employees more than their required intervals per schedule (unit penalty for violating the C_{10} constraint)
h_i	total intervals per week for employee i which is calculated for the GA solution (expressed in interval)
R_{ktd}	preferred number of required employees for job k on interval t of day d
S_{ktd}	number of assigned employees to job k on interval t of day d (calculated from the GA solution)
P_{ktd}	unit penalty for understaffing of job k on interval t of day d
$R_{ktd}^{\min}$	minimum number of required employees for job k on interval t of day d
RI_i	required intervals per schedule for salaried employee i (expressed in interval)
IP_i	paid time-off intervals for salaried employee i (expressed in interval)
UBI_j	unpaid break intervals for shift j (expressed in interval)
PS_i	paid Intervals per schedule for employee i which is calculated for the GA solution (expressed in interval)
G_i	group-weight for employee i

O_{ij}	option-weight for shift j and employee i
S_i	seniority-weight for employee i
W_{ij}	skill-level-weight for employee i for the associated skill of shift j
ID_i^{max}	maximum intervals per day for employee i (expressed n interval)
IS_i^{max}	maximum intervals per schedule for employee i (expressed in interval)
ND	number of days in the scheduling period
is_i	maximum intervals per schedule for employee i for the GA schedule (expressed in interval)
CD_i	number of consecutive days that employee i has worked in the previous schedule
CD^{max}	number of maximum consecutive working days
L_{di}	length of shift on day d for employee i
F_{nz}	fitness value of rule n for schedule z
TF_z	total fitness value for schedule z
Gap_{mb}	percentage gap between Mosel's best solution and best bound
Gap_{gm}	percentage gap between GA's and Mosel's best solution
Gap_{gb}	percentage gap between GA's best solution and Mosel's best bound
Q	quality
T	time

Chapter 1

INTRODUCTION AND LITERATURE REVIEW

1.1 Personnel Scheduling and Rostering

Efficient employment of manpower has always been a major concern in any organization and is one of the most important approaches to acquire productivity. Because of its high importance, considerable endeavours have been dedicated to tackling the problem.

Personnel scheduling for an organization is described as the process of producing optimized timetables for its employees. This process starts with the determination of staffing requirements and terminates with the allocation of each individual to a specific task over a period of time. The work regulations associated with the relevant workplace agreements must be observed and the individual work preferences should be valued during this process. Personnel scheduling problems are typically highly constrained and complex optimization problems. Finding good solutions for these highly constrained and complex problems is extremely difficult and even it is more difficult to determine optimal solutions that minimize costs, meet staff preferences, fulfill shift equity among staff and satisfies all the workplace constraints.

1.2 Personnel Scheduling Steps

The labour scheduling process starts with the determination of staffing requirements which is referred to demand modeling in literature. Ernst et al. (2004a) describe two main components in demand modelling: (1), a method for translating incident data to a demand for staff, and (2), a method for forecasting incidents, unless the incidents are derived from a known timetable. These methods are varied in different applications of staff scheduling. In some applications of rostering, the demand may be generated from incidents in a reasonably straightforward, though possibly complicated, manner. For example, in crew scheduling in airlines, railways, buses, and mass transit, the demand for crew is very straightforward because crew rosters are determined by known timetables. In this case, the demand for staff is attained from lists of individual tasks to be

performed. The tasks are defined in terms of an earliest starting time, a latest finishing time, a duration, a location and possibly the skills required to perform the task. The other examples are nurse scheduling and ambulance services, in which the demand is obtained directly from a specification of the number of staff that are required to be on duty during different shifts. For nurse scheduling, a shift is often defined as a day's work for a worker. In other applications of rostering, such as call centres and retailers, the nature of demand is fluctuating. In these cases, there are no known timetables or list of tasks, the likelihood of future incidents, such as the fluctuating arrivals of customer queries to a call centre, must be modeled by forecasting techniques. The conversion from an incident forecast to staffing requirements is accomplished using techniques such as queuing theory or simulation. The result of such a conversion is the number of staff required at each skill level for each time period.

After performing demand modeling, the next steps of rostering are (1) shift scheduling, (2) days-off scheduling, (3) tour scheduling and (4) task assignment. Bechtold et al. (1991) and Ernst et al. (2004a, b) describe these terms as follows:

1. *Shift scheduling*: Shift scheduling involves selecting a set of the best shifts from a (large) pool of candidate shifts on a single day for each employee. Each shift is a set of employee work schedules defined by start, finish and break times across a daily planning horizon. Shift scheduling for transportation systems is recognized as crew scheduling.

2. *Days-off Scheduling*: Days-off scheduling specifies work and non-work days for employees when the employee work week is shorter than the operating week (rostering horizon).

3. *Tour Scheduling*: Tour scheduling addresses both days-off and shift scheduling over a weekly planning horizon. The process involves both choosing the off days for the workers and allocating shifts for each of their working days over the rostering horizon. Tour scheduling reduces to shift scheduling when the rostering horizon is one day.

4. *Task Assignment*: The last step of labour scheduling is task assignment. Task assignment is the process of allocating a set of tasks, with specified start and end times and skill requirements, between a group of workers who have already been assigned to a set of working shifts (Ernst et al, 2004a, b). If shift definition for an organization includes the start time, end time, task and skill requirement, this step is the same as shift scheduling.

1.3 Personnel Scheduling Applications

Ernst et al. (2004a, b) categorize labour scheduling problems into several applications including airlines, buses, railways, mass transit, nurse scheduling, call centers, manufacturing, health care systems, civic services and utilities, venue management, protection and emergency services, hospitality and tourism, financial services and sales. Ernst et al. (2004a) provides the number of related papers that have been worked on each application (Table 1.1).

As it is observed from Table 1.1, personnel scheduling has received little attention for retail business in the literature.

1.4 Personnel Scheduling Solution Methods

Ernst et al. (2004a) categorize the personnel scheduling solution techniques as follows:

1.4.1 Artificial intelligence approaches

Artificial Intelligence (AI) can be described as the simulation of certain human intelligence processes using machines, especially computer systems. These processes include learning, reasoning and self-correction. The learning is used to acquire information and rules; the entire knowledge of the system designer about the process to be controlled is stored as rules (Zimmermann, 1996). The reasoning process involves using the rules to reach approximate or definite conclusions. Burke et al. (2004) states: "Reasoning process imitates the human style of reasoning in which problems are solved using past experience, on the premise that similar problems require similar solutions". Artificial intelligence techniques including fuzzy set theory,

search and expert systems have been applied to rostering problems. Fuzzy set theory has been used to solve air crew scheduling problems.

Application	No. of Papers	Application	No. of Papers
Buses	129	Civic Services and Utilities	22
Nurse Scheduling	103	Venue Management	19
Airlines	99	Protection and Emergency Services	16
Railways	37	Other Applications	14
Call Centers	37	Transportation Systems	12
General	33	Hospitality and Tourism	7
Manufacturing	29	Financial Services	6
Mass Transit	28	Sales	3
Health Care Systems	23		

Table 1.1 - Categorization of papers by application (Ernst et al., 2004a, p.34)

When there are many human factors that cannot easily be modeled in software, and usually are left up to the person in charge of rostering, the expert system approach can work well. An expert system is a computer program that simulates the judgement and behaviour of a human, or organisation, with expert knowledge and experience in a particular field. Typically, a complex set of rules for applying the knowledge base to a set of situations is setup. For scheduling and rostering systems, these rules would be used to construct duty pairings and rosters.

1.4.2 Constraint Programming (CP)

Domain variables are used to describe the data representing a problem. Each variable has a set of potentially feasible values, known as an associated domain. The different relationships that must be met by a set of variables is known as a constraint. Many scheduling and rostering problems contain complex rules that are very hard to model as mathematical equations. Constraint programming (CP) provides a powerful tool for finding feasible solutions to these types of problems. When a problem is highly constrained or when non-optimal feasible solutions are sufficient, CP becomes particularly valuable. Nurse rostering, where the rules are often rigid and highly constrained, is one area where pure CP approaches have been used and have performed well. Some people have attempted to combine CP with traditional Operation Research (OR) techniques because optimization with CP is generally quite inefficient. One example of that is

using CP as a pre-processing technique to reduce the problem size before using linear programming (LP) based branch-and-bound method to perform the optimization.

1.4.3 Meta-heuristics

Meta-heuristic such as genetic algorithm (GA), tabu search (TS) and simulated annealing (SA) are most generally used to solve problems classified as NP. Herbert (1994) classifies the decision problems into two groups: (1), class p (polynomial) and (2), class NP (nondeterministic polynomial). P is the class of decision problems for which it is easy to find a solution. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer. A decision problem is NP-complete if it belongs to class NP and every problem in NP is quickly reducible to it; i.e., if an algorithm exists that can solve an NP-complete problem, then it can solve all NP problems. A problem is hard if it is not possible to guarantee, at most, a polynomial running time for the worst case scenario of the problem. The complexity of a problem is the cost of solving the problem and usually is measured in running time. Running time is usually expressed as a function of input size, if the running time is at most a polynomial function of the input size, then the problem is easy, otherwise is hard. The fact that a problem is hard does not mean that every instance of it has to be hard, but it means it is not possible to devise an algorithm for which fast performance for all instances of the problem can be guaranteed. So if it is desired to see whether the problem is hard or not, or if an algorithm is fast, the worst case scenario of it must be considered; sometimes the average case input can be considered too.

Meta-heuristics form an important class of methods that solve hard, and usually discrete, optimisation problems. Problems which appear to be particularly appropriate for meta-heuristics include timetabling and real-life scheduling problems.

The popularity of these types of methods for solving rostering problems is due to a number of factors including:

1. They tend to be relatively robust. They cannot be guaranteed to produce an optimal solution, but they can usually produce a reasonably good feasible solution for a wide range

of input data in a limited amount of running time. In contrast to this, integer programming approaches often run the risk of not returning any feasible solutions for a long time.

2. Most meta-heuristics are relatively simple to implement and allow problem specific information to be incorporated.

3. Heuristics make it easy to deal with complex objectives, whether these are real staffing costs or penalties for violating constraints that are desirable but not mandatory. Airline and bus driver crew scheduling, nurse rostering and cyclic staff scheduling problems are some examples of rostering applications to which meta-heuristics have been implemented.

1.4.4 Mathematical Programming Approaches

In mathematical programming approaches, scheduling and rostering problems are formulated as linear programs or linear integer programs, or general mathematical programs. The lowest cost solutions are usually achieved using algorithms based on mathematical programming approaches. However there are a number of shortcomings with these approaches that prevent them from being universally applied:

1. Mathematical programming formulations are more limiting in what constraints and objectives can be expressed easily. Hence, these approaches are more commonly applied to unsophisticated versions of the real world rostering problem or where there are few complications in the original problem (Ernst et al., 2004a).

2. Implementing a good integer programming method for a particular crew scheduling and rostering problem is a quite difficult and lengthy process. This is only warranted when the advantage of the reduced cost of solutions obtained is significant, and when the rostering rules and regulations are relatively static over time (Ernst et al., 2004a).

3. Cai and Li (2000) state that "The staff scheduling problem is in general very difficult to solve even if it is only involved with a single objective and one skill. In fact the problem has been known to be NP-complete (section 1.4.3). When multiple criteria and skills are included, the problem becomes much more difficult to be solved by traditional mathematical programming techniques".

Ernst et al. (2004b) categorize the solution techniques employed for rostering problems more comprehensively than above in Table 1.2, and also represent the number of existent papers in each category. Some methods are special cases of more general methods. For example, linear programming and integer programming are special cases of mathematical programming. It is appropriate to identify these individual methods in order to indicate their importance as rostering solution techniques.

1.5 Genetic Algorithm (GA)

In this thesis, a genetic algorithm technique is developed and applied to a real-life retail rostering problem. A genetic algorithm is a meta-heuristic search technique, invented by Holland (1975), which can find the global optimal solution in complex search spaces. A genetic algorithm is based on the natural genetic processes implemented to the optimization of data structures. Genetic mutation, crossover and selection operators are used in GA technique. GA works with a population of candidate solutions, each candidate solution is represented as a string of bits which is called a chromosome, where the interpretation of the bit string is problem specific. Each bit string in the population is assigned a value according to a problem-specific fitness function. These strings are recombined by using the crossover and mutation operators to create the new population and then, via the fitness function, the best strings (or chromosomes) are selected from the old and new population. A more detailed description of the GA is provided in chapter 3 of this thesis.

1.6 Branch-and-bound

In this thesis, the proposed genetic algorithm is compared with the existent branch-and-bound method. *Branch-and-bound* is an approach developed for solving discrete and combinatorial

optimization problems. The discrete optimization problems are problems in which the decision variables assume discrete values from a specified set; when this set is set of integers, the problem is an integer programming problem. The combinatorial optimization problems, on the other hand, are problems of choosing the best combination out of all possible combinations. Most combinatorial problems can be formulated as integer programs (Murty, 1995).

Method	No. of Papers	Method	No. of Papers
Branch-and-Bound	14	Lagrangean Relaxation	32
Branch-and-Cut	9	Linear Programming	35
Branch-and-Price	30	Matching	36
Column Generation	48	Mathematical Programming	27
Constraint Logic Programming	46	Network Flow	38
Constructive Heuristic	133	Other Meta-Heuristic	11
Dynamic Programming	17	Other Methods	35
Enumeration	13	Queuing Theory	32
Evolution	4	Set Covering	58
Expert Systems	15	Set Partitioning	72
Genetic Algorithms	28	Simple Local Search	39
Goal Programming	19	Simulated Annealing	20
Integer Programming	139	Simulation	31
Iterated Randomised Construction	5	Tabu Search	16

Table 1.2 - Categorization of papers by solution method (Ernst et al., 2004a, p.41)

Branch-and-bound approach includes branching, bounding and fathoming steps (Hillier and Liberman, 2001):

Branching is done by partitioning the entire set of feasible solutions into smaller and smaller subsets. In bounding step, for each new sub-problem a bound on how good it's best feasible solution can be, obtained. The standard way of doing this is to solve a simpler relaxation of the sub-problem. In most cases, a relaxation of a problem is obtained by deleting ("relaxing") one set of constraints that had made the problem difficult to solve. For IP problems, the most troublesome constraints are those requiring the respective variables to be integer. Therefore, the most widely used relaxation is *LP relaxation* which deletes this set of constraints. For each sub-problem, by applying simplex method to its LP relaxation, its LP optimal solution (Z) is obtained. The bound

for the sub-problem is obtained by rounding down (for IP-maximum problem) the value of Z. Then the best bound of the sub-problems (Z^*) is used in the fathoming step. Fathoming (conquering) is done partially by bounding how good the best solution in the subset can be and then discarding the subset if its bound indicates that it cannot possibly contain an optimal solution for the original problem.

1.7 Effectiveness of the Genetic Algorithm in Rostering

In this section, the effectiveness of GA for personnel scheduling is investigated by reviewing relevant literature on the subject.

Levine (1996) describes how a genetic algorithm can be effective in solving scheduling optimization problems as follows:

1. There is no need to solve the LP relaxation (section 1.6), because GA works directly with integer solutions.

2. Genetic algorithm has the advantage of being flexible in handling the variations in the model such as adding a constraint or evaluation function to the problem, or modifying them. For more traditional methods, it is hard to add or modify a new constraint or objective function.

3. At any iteration, genetic algorithms contain a population of feasible solutions. Arabeyre et al. (1969) states: "The knowledge of a family of good solutions is far more important than obtaining an isolated optimum".

Bailey et al. (1997) state: "most real-life scheduling problems are NP-hard, meaning that for even moderate size problems, algorithms will find them difficult to solve, due to the exponential size of the solution space. With increasing problem size and complexity in terms of objectives and constraints, integer programming can require too much computing power to be practical. The

technique of the genetic algorithm attempts to find a solution close to (but not guaranteed to be) the optimal solution in a reasonable amount of time".

Abboud et al. (1998) performed a survey in the retail field and applied a genetic annealing meta-heuristic (genetic algorithm and simulated annealing) to a salesman force problem. The problem was to distribute the salesmen force over the branches of a company satisfying several goals simultaneously and considering the salesmen's abilities, satisfactions and preferences. The goals were defined as:

1. Maximize the total gross sales of all branches.
2. Maximize the gross sales of each branch.
3. Maximize the satisfaction of each salesman.

The constraints were defined as:

1. Each salesman should be assigned to only one branch.
2. At least two salesmen should be assigned per branch.

In this problem, the required number of salesmen for each branch was not specified. Instead, for each branch a fuzzy target, gross sales, was given. A total target gross sales of all branches was also fuzzily specified. This problem was formulated as a fuzzy combinatorial programming problem and by applying the Bellman and Zadeh's fuzzy decision method (Bellman and Zadeh, 1970) the problem was transformed into a mixed integer programming problem (MIP). The developed genetic annealing delivered a nearly optimal solution in a practical amount of time for this problem. The solution of this problem does not specify which salesman, for which task and what time is assigned to a specific branch; it only specifies how many employees to be assigned to each branch. In this thesis the solution does specify employee assignments.

Bailey et al. (1997) applied the genetic algorithm (GA), simulated annealing (SA) and integer programming (IP) techniques to a nurse scheduling problem with different skill levels and then compared the results of them. The problem is to schedule 27 nurses for a six week period considering three shifts per day, four staff groups and three skill levels. One major requirement for

this problem is that at least one member of each team must be scheduled for the day time shifts because of patient needs. The results of these three methods have been compared in terms of solution quality, required time for generating a (near) optimal solution and scalability. Scalability explains the capability of the above optimization techniques under increasing new constraints (in this study the staff preference). The reported results are as follows:

- Quality: IP > (SA, GA)
- Time: SA < IP < GA
- Scalability: (SA, GA) > IP

Bailey et al. (1997) explained that by adding the staff preference constraint to the problem the solution time for IP increases exponentially meaning that the required time for IP increases exponentially with the population size. But adding the staff preferences to a simulated annealing or genetic algorithm would only linearly increase the evaluation time of the objective function. This is because for IP, constraints and goals are often interpreted as a large number of additional variables or constraints in the mathematical program, but for GA and SA constraints and goals do not increase the formulated size of the problem. It is necessary to mention that the quality of the solution after adding the employee preference constraint to the problem has not been examined in this research but has been referred for the future research. The results of the above study show that a genetic algorithm can generate optimal or near-optimal solutions in a relatively short time for the nurse scheduling problem without considering staff preferences.

Burke et al. (2001) applied GA, tabu search (TS) and a memetic algorithm (the combination of GA and TS) to a nurse scheduling problem including 20 nurses with different qualifications (such as head nurse, regular nurse, nurse aid, student, etc...) and responsibilities. Some of the nurses can replace people from another category (depending on their qualifications). This replacement is sometimes necessary to cover for staff shortages, but it is not desirable and will be penalized in the schedule evaluation function. This model has one hard constraint and can have up to 30 soft constraints. The hard constraint is the number of required nurses for each category and for each duty of the planning period which is often one month. Some of the soft constraints applied in this model are as follows:

1. Minimum time between two assignments, depending on the type of duties involved.
2. Maximum and minimum work hours during the planning period for different type of staff (full time, part time, night nurse and etc.)
3. Maximum number of assignments during the planning period (e.g., for full time nurses this is usually restricted to 20 assignments per 4 weeks).

The evaluation function for this problem is implemented as a series of modules, each corresponding to a soft constraint. The user fixes the parameters and sets the penalty weight per unit violation of the constraint. This study was examined over 4 test cases. These test cases were different in terms of the number of soft constraints. The results for these test cases show that memetic algorithms are robust enough to produce excellent solutions; but unacceptable solutions usually arise when the constraints on the problem are contradictory. It is then very hard to find the very narrow valleys in the solution space, which contain good schedules. The issue with the basic GA is repairing the solution when it is destroyed by the crossover operator. A common issue with rostering problems is that the quality of a solution can be destroyed by the crossover operator. The problem with tabu search is that it is not robust enough to handle difficult problems.

In the previous two studies, the size of the problems that were considered was small (27 employees and 20 employees). In this thesis, problems with various sizes (17 to 600 employees) have been applied; a large problem size is more applicable to most retail companies. Also, a shift includes only start and end times while in this thesis, a shift definition is more comprehensive, it includes day, location, start and end times, skill and activity .

Cai and Li (2000) developed a GA for the problem of scheduling staff with mixed skills. The problem was formulated as a multi criteria optimization model with three objectives as follows:

1. Primary objective: To minimize the total cost for assigning personnel to meet the manpower demands over time.
2. Secondary objective: To maximize the staffing surplus when assigning cost is almost the same.

3. Tertiary objective: To minimize the variation of surplus staff over different period of times.

The primary criterion is similar to traditional staff scheduling model which is a single criterion integer optimization problem. The second criterion has been included to reduce the risk of underestimation of actual demands that may occur due to an inaccurate forecast. The third criterion is included to achieve a more balanced staff distribution. In this model, jobs with different skill requirements and employees with multiple skills have been considered. This model considers two types of jobs which consecutively needs D_{1t} and D_{2t} staff at time t, where $t = 1, 2, ..., T$ and T is scheduling horizon. And also there are three types of staff. The first staff group with skill 1 can be assigned to type-1 job, the second staff group with skill 2 can be assigned to type-2 job and the third staff group can be assigned to either type-1 or type-2 job. In the applied GA to the above problem, each individual or chromosome, in the population has been encoded as a four element vector, each element of the vector corresponds to each staff group that is assigned to each job; element 1 shows the feasible schedules for staff group 1, element 2 shows the feasible schedules for staff group 2, element 3 shows the feasible schedules for staff group 3 who are assigned to job 1 and element 4 shows the feasible schedules for staff group 3 who are assigned to job 2. The position of each gene in the chromosome represents one feasible schedule and the value of the gene represents the number of workers assigned to the corresponding schedules. Infeasible offspring may be created by the crossover operation even though the parents are feasible. Each solution is checked by every constraint in the scheduling problem and if it violates the constraint, it is repaired by some heuristics. Computational studies have shown the effectiveness of the model and the proposed approach. In all cases the proposed GA worked reliably, and successfully delivered good, feasible solutions, in about 8 to 10 minutes.

In the above study, the number of jobs, and the demands for staff for each job, is limited (only two types of jobs and two demands). The solution of the problem does not specify which employee is assigned to which job; it only specifies the number of workers assigned to each schedule. This thesis differs from the above study in that no limitations are placed on the number of jobs or the number of demands, and the solution does specify employee assignments. In addition, this thesis

considers employee type (salary, part time, full time and etc), employee availability and employee seniority; none of which were included in the study by Cai and Li.

1.8 Research Scope and Objectives

In this thesis, the problem of scheduling staff with different skills and employee types for large retailers is reviewed. The objective of this study is to determine a schedule for each employee, for a specific location of the retailer's store that minimizes the amount of total payroll cost and penalties for violating the various rules and employee preferences. In this study, a GA is applied as the solution method to this problem and the effectiveness of GA for this problem is investigated. As it was indicated in section 1.3, personnel scheduling/rostering has received little attention for retail business in the literature; however, plenty of works have been done in the other areas of personnel scheduling (such as crew scheduling, nurse scheduling, call centers and rostering with multi-criteria) in the literature. Shift definition, large range in the problem size, different employee types, various skills, various skill levels, various jobs and locations, different staff demands, considering employee's availability and seniority are the features that differentiate this study from the other studies in the literature. In this study, a shift includes day, location, start and end times, skill and activity, while in the other cited studies in the literature shift includes only start and end times. A large range in problem size is considered in this study (from 17 employees to 600 employees), while in the other studies, only problems with small size (such as 27 employees) have been considered. In this study, there is no limitation on the number of employee types, skills, jobs and required employees, but these have been limited in the other studies; for instance, scheduling staff with two different skills or scheduling staff for two specific jobs with two specific staffing demands have been discussed.

For many of the personnel scheduling applications, GA has been used as a solution method and good encouraging results have been achieved. To the best of the author's knowledge, GA has been implemented for the retail personnel scheduling only in one case by Abboud et al. (1998); please see section 1.7. What differentiates the mentioned survey (Abboud et al., 1998) from this thesis are: The required number of salesmen for each branch was not specified; instead, for each

branch, a fuzzy target gross sales was given. But in this thesis, the required number of employees is considered. The solution of this problem does not specify which salesman, for which task and what time is assigned to a specific branch; it only specifies how many employees to be assigned to each branch. But the solution of problem in this thesis specifies which employee is assigned to which shift (day, location, start and end times, skill and activity). In contrast with this thesis, employee type (salary, part time, full time and etc), employee availability, employee seniority have not been considered in Abboud's work. The applied constraints and objectives in this thesis are more comprehensive than Abboud's survey.

1.9 Thesis Structure

The structure of this thesis is as follows: In chapter 2, a detailed description of the problem is provided. In chapter 3, a general concept of the genetic algorithm is described in more detail and a GA is proposed for solving the problem under study. In chapter 4, the results of design of experiment and analysis of variance (ANOVA) on three test problems are explained. In chapter 5, the performance of proposed GA is verified with the existent integer goal programming model (IGP). In chapter 6, the conclusion and future research are addressed.

Chapter 2

PROBLEM DESCRIPTION

2.1 Overview

In this study, the problem of scheduling staff for large retailers is reviewed. The objective of this study is to determine a schedule for each employee for a specific location of the retailer's store that minimizes the amount of total payroll cost and penalties for violating the various rules and employee preferences.

Large retailers continuously encounter fluctuation in customer demand and workforce availability, and as a result, face the issue of understaffing or overstaffing. The key for a retailer success is scheduling an adequate number of employees considering employee preferences. Understaffing leads to poor customer service, causing reduced customer conversion rates and a potential loss of revenues. Understaffing can also lead to employee dissatisfaction. Overstaffing results in payroll expenditures. Appropriate labour scheduling enhances customer service, operational efficiency and employee satisfaction. It is very important for a retailer to ensure the right employees, with the right skills, be in the right place at the right time. Retailers are involved with plenty of rules or constraints related to either their own policy or labour union rules. These rules make the labour scheduling problem in a retail environment very complex. Quan (2004) divides the rules for a retailer into four categories as follows:

1- Basic workload assignment
2- Government or union regulations
3- Cost & budget considerations
4- Employee quality of life

2.2 Retailers Rules

Each retailer has its own policies and rules. The most common rules for retailers are listed and described below. It is necessary to mention that not all of these rules have been implemented in this project; the list of implemented rules in this study is provided in section 2.3.

1. *Honour skill requirements*: This rule ensures that each employee is only assigned to those jobs for which he/she has the required skills. The skills are divided into job related skills such as ability to stacking shelves or loading trucks and side skills such as first aid.
2. *Honour employee availability*: According to this rule the availability of all the employees must be examined to make sure they are not scheduled for shifts for which they are not available.
3. *Enforce maximum shifts per day*: Union rules may stipulate that each employee can only work one shift per day. This will prevent employees from working a few hours, going home and then coming back to work for another few hours. Such shift patterns may be viewed as distressing to the employees. Each employee can individually opt out of this rule.
4. *Enforce minimum hours per shift*: This rule ensures that an employee is given at least the specified minimum hours per shift (for example, union regulations may stipulate that staff must be given at least four hours per shift).
5. *Enforce maximum hours per day and per schedule*: Each individual staff member can have different maximum allowable hours or union rules may stipulate that employees may not work more than for instance 60 hours per schedule. This rule ensures that the schedule does not exceed the specified maximum hours for each employee.
6. *Enforce maximum consecutive work days*: This rule ensures that staff members are not scheduled for more than maximum consecutive days, for instance three days.
7. *Satisfy staff requirements*: This constraint ensures that the schedule satisfies workload requirements (the preferred and minimum number of required staff for a specific job, skill and activity) for each department in the store and for each day and time interval.
8. *Enforce minimum hours per week for fixed shift employees*: This constraint ensures that fixed shift employees are scheduled for their total fixed hours. A fixed shift is a shift for

which the start and end times do not change. Staff may have a special arrangement where they are given fixed weekly shifts such as, Monday to Friday, 9am to 5pm. An employee who is assigned to this type of shift is referred to as a fixed shift employee.

9. *Enforce required hours for salaried employees*: This rule ensures that the salaried employees are scheduled for their required number of hours.
10. *Assign staff considering seniority*: Given it is possible to schedule two or more employees for the same shift, this rule ensures that the most senior employee is scheduled for the job.
11. *Skill level better than required level*: This constraint ensures that the employee with the highest skill level is selected first.
12. *Preferred employee type (full time, part time, seasonal)*: Union rules may specify that one employee type should be preferred over another. For example, full time staff should be given preference over part time staff. Therefore, when two or more personnel can be assigned to a shift, the preferred one (full time employee) is selected.
13. *Maximum weekends per calendar month*: Some constraints are in place for employee wellbeing and morale, for example, not scheduling an employee to work more than two Saturdays in a month, in order to allow them more time at home with their families.
14. *Enforce maximum shifts per schedule*: This rule prevents staff from being scheduled for more than the considered maximum shifts per schedule, for example six shifts. Unlike the maximum shifts per day rule, each employee can not opt in or out of this rule. Each employee can not choose their own maximum number of shifts per schedule. Fixed shift employees are excluded from this rule.
15. *Enforce maximum nights per schedule*: Union rules or local labour laws may stipulate that if an employee is a minor (e.g. under 15 years old), they cannot be scheduled for more than a certain number of nights after a set time (e.g. two nights after 8pm) as it is dangerous for them to get home late at night.
16. *Budget limit & schedule to budget*: The budget may be defined in terms of payroll cost or total hours worked. The policy of "budget limit" ensures that the cost of a schedule for all the employees does not exceed the budget limit. While the "schedule to budget" policy indicates that the schedule must use the entire budget assigned to a given store; for example

some people may be scheduled who will effectively have no real work to do, but they are there just to use the assigned budget for that store.

17. *Limit overtime per day, per schedule*: Company policy and/or union labour agreements may dictate limits on the maximum number of overtime hours for the day or schedule. A general policy to minimize overall payroll costs would force individual employee wages for regular and overtime hours to be taken into account for any given schedule.
18. *Enforce not scheduling Saturday and Sunday on the same weekend*: This rule ensures that an employee gets at least one day off for the weekend.
19. *Enforce minimum hours per schedule*: This rule ensures that an employee is scheduled at least for a minimum number of hours. Employees can have different minimum hours.
20. *Enforce minimum consecutive days off*: Many retail chains are open seven days a week now and labour laws often require that employees be allowed to have two days off in any seven day period (to simulate a weekend). It does not necessarily have to be Saturday and Sunday, just any two consecutive days.

2.3 Implemented Staff Constraints in this Study

Some of the constraints from the list provided in section 2.2 are implemented in this study and are listed below. Each rule or constraint depending on the company policy or union regulation is classified as soft, hard or scheduling preferences. In this thesis, classifying constraints as hard or soft has been done according to the policy of the industry partner. Hard constraints are those that must be satisfied. Soft constraints are those that can be broken but there is a penalty for breaking them. These rules and scheduling preferences are described in detail in chapter 3.

2.3.1 Hard Constraints

1. Honour employee availability
2. Honour employee skills
3. Enforce maximum one shift per day
4. Enforce maximum intervals per day
5. Enforce maximum intervals per schedule
6. Enforce maximum consecutive days

2.3.2 Soft Constraints

1. Enforce minimum intervals per week for fixed shift employees
2. Satisfy minimum staff requirements
3. Satisfy staff requirements
4. Enforce required intervals per week for salaried employees

2.3.3 Scheduling Preferences

1. Seniority
2. Skill level
3. Option-weight
4. Group-weight

2.4 Employee Classification

In this study, an employee is classified into the following categories:

- Part time
- Full time
- Fixed shift
- Salary
- Hourly

2.5 Data Generation

The data that is used in this study is generated by existing forecasting and scheduling systems (the industry partner). The forecasting system performs sales forecasts for the associate retailer and generates workload requirements. The scheduling system keeps track of employee availability, employee preferences and retailer regulations (minimum/maximum hours that each employee can work, maximum shifts per day/schedule, limit overtime per day, per schedule and so on), employee category (full time, part time, fixed shift, salaried, hourly), employee specifications (seniority, wage rate, skills, ...), employee schedule history (previous weekends worked, previous nights

worked) and shift rules. The existent pre-processor (written in Java) uses the information from these two systems and generates several data files, some of which are used in this study. A list of applied data files for this study is given in chapter 3.

2.6 Optimization Methods

In this study for solving the explained labour scheduling problem, two optimization techniques are implemented. They are shown in Figure 2.1 and described as follows:

2.6.1 Branch-and-bound (Mosel™)

One approach is integer goal programming. The inputs (data files) from the pre-processor are fed into another existent system with an integer goal programming model (written in Mosel™ – see Appendix II) which performs optimization through the branch-and-bound method and produces the optimal or near optimal schedules.

2.6.2 Genetic Algorithm (GA)

The other approach is implementing a genetic algorithm. In this study the proposed genetic algorithm has been written in Java. This algorithm includes the same hard constraints and objective functions as Mosel and uses the same data files generated by the pre-processor. This algorithm is described in detail in chapter 3.

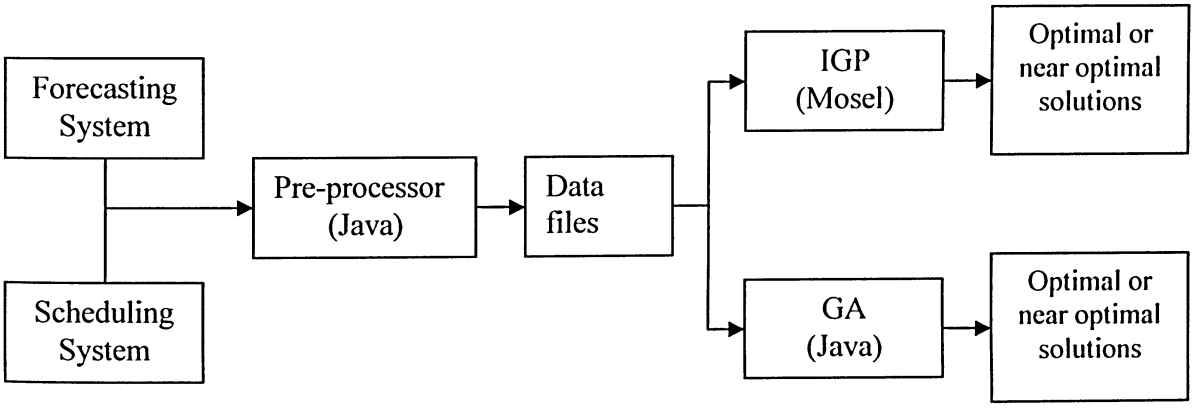

Figure 2.1 – Optimization methods

Chapter 3

METHODOLOGY

3.1 Overview

A genetic algorithm (GA) is developed in this thesis to solve the labour scheduling problem for retailers. In this chapter, first the general concept and steps of the genetic algorithm is described, and then in section 3.3, the developed GA is explained in detail.

3.2 General Genetic Algorithm

3.2.1 Background

A genetic algorithm (GA) is a meta-heuristic search technique, invented by Holland (1975) which can find the global optimal solution in complex search spaces. Meta-heuristics are based on searching the solution space and attempt to avoid getting stuck in local optima and move towards global optimum. Solution x is called a local optimum when there is no solution with a better value within a small neighbourhood of x.

The GA procedure involves the creation, with a suitable data representation, of a population of individuals representing feasible solutions. The population is constantly updated by generating new members of the population from existing members, and removing the weakest members using fitness functions. After many iterations, the best member in the population will potentially be the optimal, or close to an optimal solution to the problem.

An important feature of GA is searching several paths simultaneously starting with an initial population, i.e., a set of random initial solutions. Each individual entry or member in the population is called a chromosome. Each chromosome is equivalent to a feasible solution containing a sequence of binary, real numbers or strings known as genes.

During an evolution process, a GA performs a process of selection and recombination to produce a successor population, the next generation. The new population may consist of both parent chromosomes and newly generated chromosomes called offspring. The recombination process for generating the offspring chromosomes is called crossover.

The offspring chromosome may then pass through an operation known as mutation. Mutation is a process of modifying the structure of a selected chromosome by arbitrarily changing one or more genes. A fitness function, normally the decision objective function, is used to evaluate the offspring and parent chromosomes. The best chromosomes in terms of fitness among the parents and offspring will be selected for the next generation. As this process is iterated, a sequence of successive generations evolves and the average fitness of the chromosomes tends to increase until some stopping criterion is reached. In this way, a GA evolves the best solution to a given problem.

3.2.2 Genetic Algorithm Components

As it is shown in Figure 3.1 a genetic algorithm has eight basic components.

3.2.2.1 Representation

The first step is to apply a suitable genetic representation to the solution of the problem. How to encode a solution of the problem into a chromosome is a key issue when using genetic algorithms. The method of representation has a major influence on the GA performance in terms of accuracy and computational time. Each candidate solution or individual is likened to a chromosome which is represented by a sequence of genes from a string of bits. The interpretations of the bit string is problem specific and could consist of binary digits (0,1), floating point numbers, integers, symbols, matrices and so on. Each bit string in the population is assigned a value according to a problem-specific fitness function. Various representations can be used for a specific problem with some advantages and drawbacks. For example, consider the bin packing problem in which n objects are placed into a number of bins (at most n bins). Each object has a weight and each bin has a limited capacity. The objective is to minimize the number of used bins such that the total weight of the objects in each bin does not exceed its capacity. Figure 3.2 indicates one way to encode the solution of the bin packing problem into a chromosome.

Chromosome

5	1	5	6	4	3

Bin

Figure 3.2 - Bin Based Representation

The position of the gene represents an object and the value of the gene represents a bin to which the object belongs. The chromosome 5 1 5 6 4 3 in Figure 3.1 encodes a solution where the first object is in bin 5, the second in bin 1, the third in bin 5, the fourth in bin 6, the fifth in bin 4 and the sixth is in bin 3. The advantage of this representation is the constant length of the chromosomes, which allows for the application of standard genetic operators. However, one of the drawbacks is redundancy, i.e., a noticeable number of chromosomes encode the same solution of the problem. The degree of redundancy grows exponentially with the number of bins, i.e., with the size of the problem. So the size of the space the genetic algorithm has to search is much larger than the original space of solutions. For example, 4 5 5 3 2 2 and 4 1 1 5 3 3 both represent the same solution to the problem, the 2^{nd} and 3^{rd} objects are assigned to one bin, the fifth and sixth to one bin, and the 1^{st} and 4^{th} are assigned to another two bins. Another drawback of the above representation is producing infeasible solutions in that a bin may be assigned too many objects, passing its capacity.

Figure 3.2 depicts better representation for the bin packing problem that indicates which objects belong to which bins. The chromosome in figure 3.2 can be rendered as BACBED, if the previous representation is used.

1,4	5	3	6	2	Object
B	E	C	D	A	Bin

Figure 3.3 - Group Based Representation

The encoding above is adapted to the objective function of the bin-packing problem, as the objective function depends on grouping objects (Gen and Cheng, 2000). Another example is labour scheduling problems. In many papers, solutions for labour scheduling problems have been expressed as vectors of positive integers $X = (X_1, X_2, ..., X_N)$. Each chromosome in the population is

represented as an N-element solution vector. Each gene (element) in a chromosome indicates one of the labour tours (schedules). The value assigned to each gene (X_i) is a positive integer which represents the number of employees assigned to the corresponding tour. The length of a chromosome depends on the number of feasible employee schedules (Easton and Mansour, 1999).

3.2.2.2 GA Parameters

There are three important control parameters of a simple GA. They are the population size, the crossover rate and the mutation rate. These three parameters and their effects on GA performance have been studied and reported on by many researchers which have been led to the following conclusions (Rothlauf, 2002). Population size refers to the number of individuals, solutions or chromosomes in the population. A large population size uses many samples from the search space and as a result the probability of convergence to a global optimal solution increases. However, because of the concurrent handling of many solutions, the computation time per iteration increases. A small population size reduces the probability of convergence to a global optimal solution. The frequency of the crossover operation is controlled by the crossover rate. A high frequency will increase the speed of convergence to a favourable region which is advantageous at the start of optimization, but if too high can result in saturation around one solution. A low frequency will decrease the speed of convergence and can result in a diffuse region of solutions. A large diversity in the population is beneficial and can be achieved with a high mutation rate, but the higher the mutation rate the greater the risk of instability. If the mutation rate is too low, it makes it more difficult for a GA to find a global optimal solution.

3.2.2.3 Initialization

For any GA, a number of feasible solutions have to be created as the initial population. The initial population can be generated randomly by a random number generator. In the cases that a prior knowledge about the given optimization problem exists, this knowledge can be used to form an initial population. In this way, GA starts searching in a set of approximately known solutions and as a result converges to an optimal solution in less time.

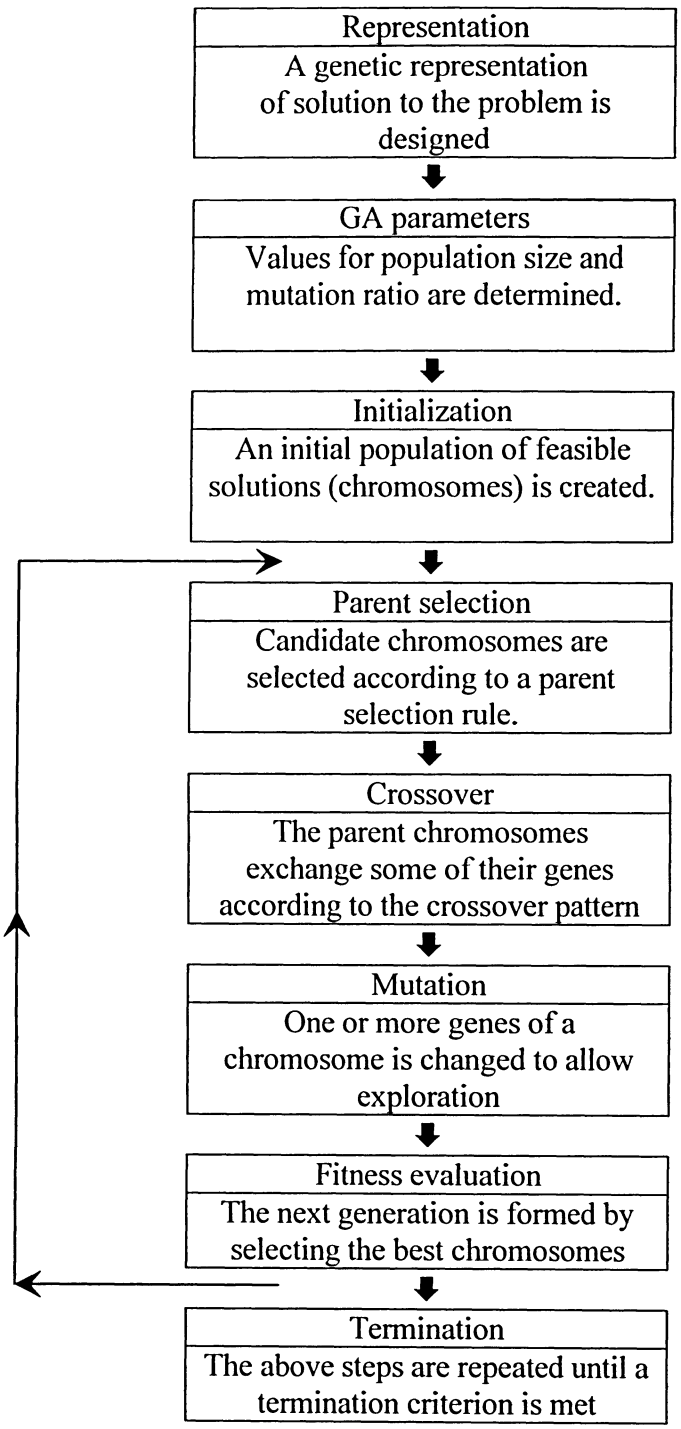

Figure 3.1 - Basic Components of a Genetic Algorithm

3.2.2.4 Parent Selection

Candidate chromosomes are selected according to a parent selection rule. In each generation, a set of offspring chromosomes are produced through a recombination process of two parent chromosomes which will be explained later. Zolfaghari and Liang (2003) describe three types of parent selection rule as follows.

Random Selection

This rule is considered as the simplest selection rule. As per this rule, two parents are selected randomly from the current population without taking account of the quality of the parents. There is no restriction for selecting a parent. A parent can be selected any number of times to take part in the process of generating offspring chromosomes.

Roulette Wheel Selection

The quality of parents is considered in this rule. A selection probability p_k is assigned to each individual or chromosome C_k in the population, this probability is proportional to the fitness value f_k of the chromosome. First, all chromosomes are ranked based on their fitness value and then p_k for each chromosome is calculated by dividing its fitness value to the total cumulative fitness value of all chromosomes in the population. A uniform random number in the range of [0, 1] is generated and according to Figure 3.4 by comparing this random number with the p_k, a parent is selected. This process is repeated until the desired number of parents is gained. A model roulette wheel can be made displaying these probabilities. The selection process is based on spinning the wheel the number of times equal to population size, each time selecting a single chromosome for the new population.

$$p_k = \frac{f_k}{\sum_i f_i}$$ k = Position index of a chromosome

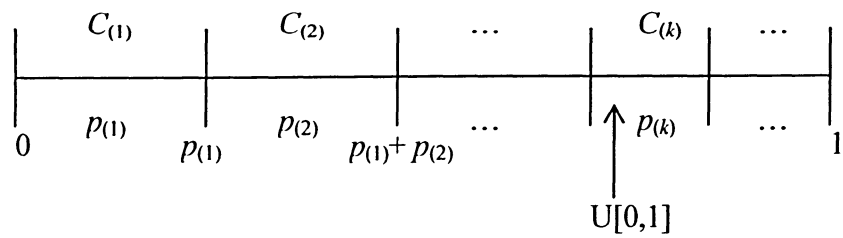

Figure 3.4 - Roulette Wheel Selection

Stochastic Universal Sampling

In this rule, similar to the roulette wheel selection rule, chromosomes are mapped to a line. Each segment of the line is equal in size to the chromosome fitness value. Then n pointers equal to population size are placed with equal space over the line. The position of the first pointer is randomly generated over the range [0, $1/n$] and the consecutive pointers are placed at $1/n$ units apart on the line according to Figure 3.5.

$$p_k = \frac{f_k}{\sum_i f_i}$$ k = Position index of a chromosome

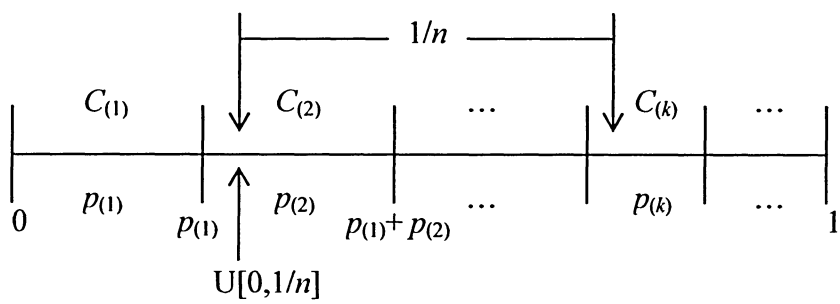

Figure 3.5 - Stochastic Universal Sampling

3.2.2.5 Crossover or Recombination

Two existing individuals (parents) are picked from the current population by the selection operation and two new individuals (offspring) are created by the crossover operation. The

crossover is used to perform a local search to try to find an improved solution. Many crossover techniques exist; the most common of them are as follows:

One-Point Crossover

A crossover point on the parent chromosome is selected. All data beyond that point is swapped between the two parent chromosomes. The two resulting individuals are the children or offspring. Figure 3.6 depicts a one-point crossover.

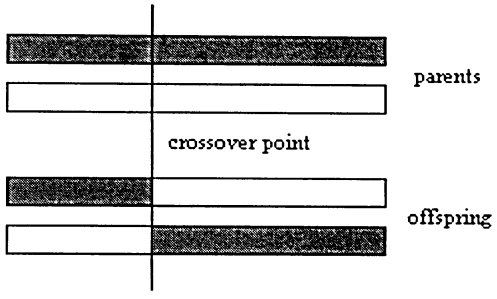

Figure 3.6 - One-Point Crossover

Two-Point Crossover

Two crossover points are selected for the parent chromosomes and all the genes between the two points are swapped between the two parents and two offspring are generated. Figure 3.7 depicts a two-point crossover.

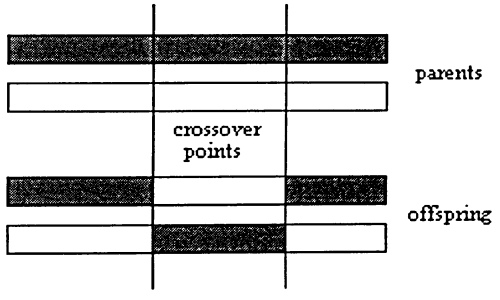

Figure 3.7 - Two-Point Crossover

Uniform Crossover

This type of crossover is accomplished by selecting two parent solutions and randomly taking a component from one parent to form the corresponding component of the child.

A template chromosome of binary genes with the same length as parents is randomly generated. Each gene of the child is formed by the accordant gene of the template chromosome, in that for offspring 1 if the value of the accordant gene is 1, the gene is taken from parent 1 and if 0 it is taken from parent 2; and for offspring 2 if the value of the accordant gene is 1 it is taken from parent 2 and if 0 it is taken from parent 1. Figure 3.8 illustrates a uniform crossover.

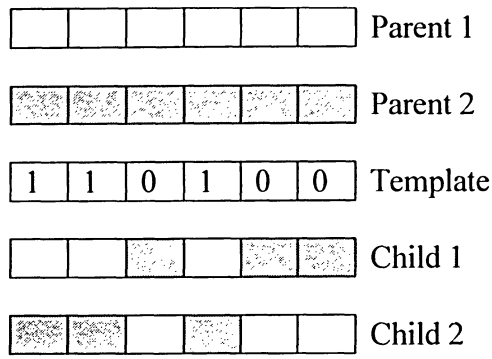

Figure 3.8 - Uniform Crossover

3.2.2.6 Mutation

The mutation operator tends to make small random changes in one parent to form one child. The mutation operator selects one or more gene randomly and changes their value. The purpose of mutation is an attempt to explore the entire solution space and escaping from the local optimum by preventing the population of chromosomes from becoming too similar to each other. Figure 3.9 depicts a mutation operation.

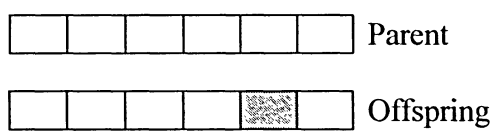

Figure 3.9 - Mutation

3.2.2.7 Fitness Evaluation

The fitness evaluation function acts as an interface between the GA and the optimization problem. The GA fitness function evaluates the quality of solutions (chromosomes) according to the objective of the optimization problem. Fitness function might be complex or simple depending on the optimization problem at hand. Consider the population size for a GA is n, i.e. the number of

feasible solutions in the current population equals to n. After conducting crossover on the population, the population size including parent and offspring solutions increases to $2n$ if the crossover produces two children from a pair of parents. By the fitness function the $2n$ chromosomes are ranked and the best n chromosomes out of the $2n$ available ones are selected as the population of the next generation. Therefore, the new population consists of the best individuals from the parents and offspring.

3.2.2.8 Termination

This process (steps 4 to 7 in Figure 3.1 - parent selection, crossover, mutation and fitness evaluation) is repeated until a stopping criterion is met. The most frequently stopping criteria are as follows:

1. Maximum number of generations
2. Population convergence criterion when the sum of the deviations among individuals becomes smaller than a specified threshold.
3. Lack of improvement in the best solution over a specified number of generations.

3.3 The GA Design for Retail Labour Scheduling

In this section, a genetic algorithm is designed for the retail labour scheduling problem. This GA has been written in Java. A portion of the Java code is included in Appendix III. The steps that were described in section 3.2.2 are represented by the box labelled "Genetic Algorithm" in Figure 3.10. The GA uses the "potential solutions" and the other "data files" (explained in section 3.3.2) as input and then finds the global optimal or near optimal solution through the GA steps.

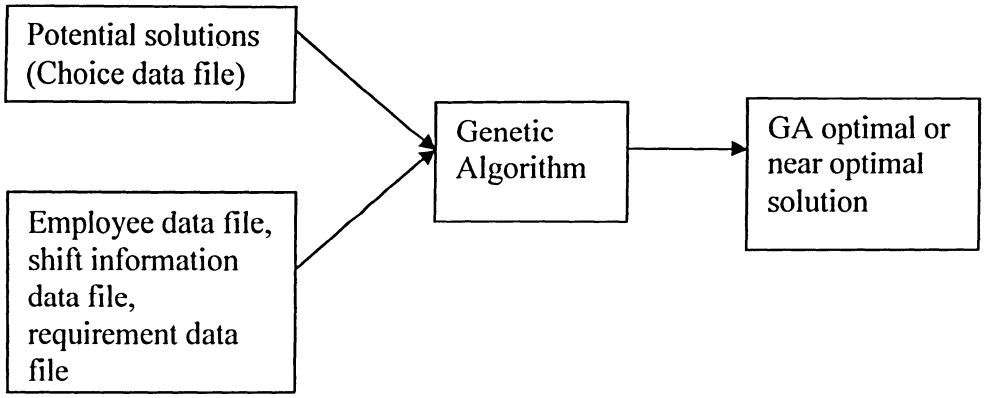

Figure 3.10 - Genetic Algorithm

To describe the developed GA in this study, first some terms and data files are described as follows:

3.3.1 Terms

Interval: An interval is the smallest segment of time that is defined for scheduling purpose. In this study an interval can be 15 min, 30 min or 60 min. Counting intervals starts from 12:00am. For instance, if interval is defined as 15 min, a day includes 96 intervals. Interval 1 is 12:00am to 12:15am, and interval 96 is 11:45pm to 12:00am. The start time and length of a shift are expressed in intervals.

Shift: A shift specifies day, location, position, skill, activity, start time and length. An example of a shift ID is as follows: (please note the following examples are provided in the same format as the data files)

4, 46304*20002*12481*10001.37.10 which indicates

day = 4 (Thursday)

location = 46304 (sports equipment store)

position = 20002 (sales associate)

skill = 12481(knowledge of sports equipments)

activity = 10001 (providing information to customers for what they are looking for)

shift start interval = 37 (i.e. if interval is 15 min, shift starts at 9:00am)

shift length = 10 intervals (shift ends at interval 46, i.e., if interval is 15 min, shift ends at 11:30am)

Job: A job specifies location, position, skill and activity.

Here is an example of a job: 46304*20002*12481*10001 (i.e. providing information to the customers at the sports equipment store as a sales associate and having the knowledge of sports equipments as the required skill)

Choice: A choice is formed by assigning a possible shift to an employee.

'4, 46304*20002*12481*10001.37.10', 252574, 1.9744, 1.9 is an example of one choice which indicates employee 252574 is assigned to shift '46304*20002*12481*10001.37.10'

shift = 46304*20002*12481*10001.37.10

employee = 252574

Associated with each choice there are "option-weight" and "skill-level-weight" values which are used for developing the GA fitness function in section 3.3.4.5. For the above example

option-weight = 1.9744

skill-level-weight = 1.9

Option-Weight: Associated with each choice is an option-weight. A smaller option-weight means the particular choice is more likely to be chosen. The existent system considers some factors and constraints for calculating option-weight as follows:

1. Preferred employee type: Any number of different employee types can be defined such as full time, part time, seasonal and regular. Suppose the preferred employee type has been specified as full time. If the employee associated with a choice is full time, pre-processor does not do anything, but if he is not full time, it makes the associated option-weight for that choice higher, so that the choice is less likely to be chosen.
2. Max Saturdays (or Sundays) per calendar month: Suppose the Max Saturdays (or Sundays) per calendar month is equal to 2. If, for example, one employee has already worked two Saturdays in the calendar month the pre-processor makes the associated option-weight for all of employee's Saturday choices in the current schedule higher. This will prevent the optimization model from giving the employee a Saturday shift.

3. Longest shifts are preferred: If the shift length of a choice is long, the pre-processor makes the associated option-weight for that choice smaller so that the choice is more likely to be chosen.
4. Earliest shifts are preferred: If the related shift of a choice starts earlier, the pre-processor makes the associated option-weight for that choice smaller so that the choice is more likely to be chosen.

Staff group: Organize staff members governed by the same shift rules, break rules and staff rules into logical "teams" for scheduling. Staff groups are defined for each location.

Shift rules: Shift rules are applied to a location and a staff group which specifies the "Min shift length", "Max shift length", "shift start time", "shift end time" and "interval length".

3.3.2 Data Files Used in This Study

For each data file, only the data that are used for this thesis are listed.

3.3.2.1 Requirement Data File

The following data are extracted from the data file:

- Preferred number of required employees for job k on interval t of day d: R_{ktd}
- Minimum number of required employees for job k on interval t of day d: $R_{ktd}^{\min}$
- Unit penalty of understaffing of job k on interval t of day d: P_{ktd}

3.3.2.2 Employee Data File

The following data are extracted from this file for each employee:

- Employee Id: i
- Maximum intervals per schedule for employee i: $IS_i^{\max}$
- Maximum intervals per day for employee i: $ID_i^{\max}$
- Fixed-shift employees: FE
- Salaried employees: SE

- Minimum intervals per week for fixed-shift employee i: $T_i^{\min}$
- Required intervals per schedule for salaried employee i: RI_i
- The number of paid time-off intervals for salaried employee i: IP_i
- The number of consecutive days from the previous schedule: CD_i
- Seniority-weight for employee i: S_i
- Group-weight for employee i: G_i

3.3.2.3 Shift Information File

- Unpaid break intervals for shift j: UBI_j

3.3.2.4 Choice Data File

- Choices (The combination of employees and shifts)
- Option-weight for shift j and employee i: O_{ij}
- Skill-level-weight for employee i for the associated skill of shift j: W_{ij}
- If choice c is mandatory or not

3.3.3 GA Terms

Potential solutions: Potential solutions include all the potential valid combination choices of shifts and employees and are derived from the choice data file. The employee skill and employee availability are the factors that determine the choice validity.

Search space: Genetic algorithm search space includes all the potential solutions. GA explores the search space and finds the optimal solution.

Feasible solution: The feasibility of a GA solution is verified by the hard constraints which are explained in section 3.3.4.2. A solution is feasible only if all the hard constraints of the problem are satisfied.

Optimal solution: Optimal solution is the employee/shift combination which minimizes the value of fitness function (also called the best schedule).

3.3.4 The Steps of the Proposed GA

In this section, the steps of the proposed GA are described as follows:

3.3.4.1 Genetic Representation of Solution to the Problem

In this study, the solution of GA is the schedule for all of the employees which can indicate how to encode a solution of the problem into a chromosome. Each individual or chromosome is represented by a one-week schedule for all the employees. Figure 3.11 shows the schema of one chromosome. In each chromosome or individual, each weekly schedule for each employee is considered as one gene. The chromosome length is the number of employees (N).

3.3.4.2 Create an Initial Population of Solutions

GA explores the search space in order to find the optimal solution starting with an initial population. GA's search space is included all the potential solutions. Potential solutions in this study include all the potential valid combination choices of shifts and employees and are derived from the choice data file. The employee skill and employee availability are the factors that determine the choice validity. In this study, initial population with size p is generated randomly. Figure 3.12 shows the schema of a population with size p. Initial population must be feasible and during the search process the feasibility of the solutions always must be maintained. The feasibility of a GA solution is verified by the hard constraints. A constraint is considered as hard if it must be satisfied. A solution is feasible only if all the hard constraints of the problem are satisfied. As it was indicated in chapter 2, the hard constraints in this project are as follows:

1. *Honour employee availability*

The GA initial population of solutions is always feasible in terms of satisfying employee availability. Because, as it was mentioned earlier, the GA search space is the choice data file which is created by the pre-processor. When this rule is checked, the pre-processor will not create a choice for any employee who is not available to work on a particular day and interval. Thus, the initial population of GA always satisfies this constraint. However, this feasibility must be maintained during the search process by crossover and mutation operation. As it will be described

in sections 3.3.4.3 and 3.3.4.4, the implemented crossover and mutation operator in this study do not disturb this feasibility.

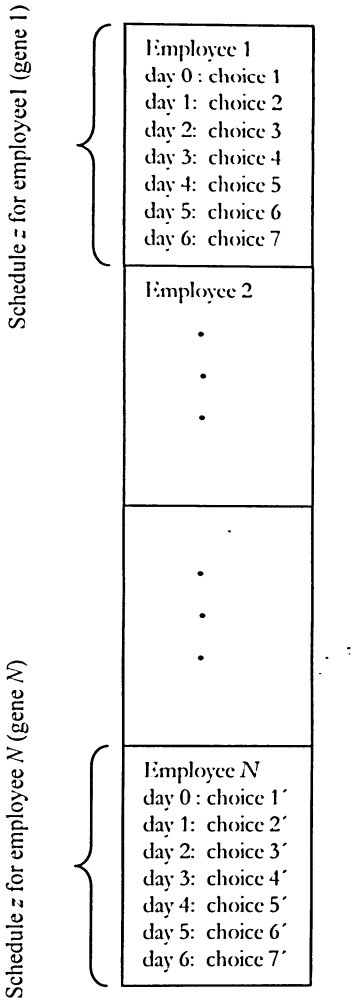

Figure 3.11 - Chromosome or Individual Representation

2. *Honour employee skills*

Similar to the above, the pre-processor does not create a choice for any employee who does not have the required skill for a specific job. As a result, the GA initial population of solutions which is derived from the choice data file is always feasible. The same discussion as "honour employee availability" holds for maintaining the feasibility during the search process.

3. *Enforce maximum one shift per day*

The feasibility of a GA solution for this constraint is simply maintained by the gene structure. Each gene of each solution is a weekly schedule for each employee containing one shift per day. The implemented crossover and mutation operator do not disturb the feasibility of the GA solution for this constraint, because they do not change the gene structure.

4. *Enforce maximum intervals per day*

Because of "Enforce maximum one shift per day" constraint and the fact that pre-processor does not generate any choice in choice data file unless maximum intervals per shift for an employee is satisfied, the maximum intervals per day for employee *i*, is always satisfied by the gene structure and does not to be checked. The implemented crossover and mutation operator do not disturb the feasibility of the GA solution for this constraint, because they do not change the gene structure.

5. *Enforce maximum intervals per schedule*

This rule must be checked for any GA solution during the search process including the initial population of solutions. It is shown in Figure 3.13 that as a schedule is generated, the algorithm compares the total intervals of the GA solution for employee *i* (is_i) with the allowable maximum intervals for that employee (IS_i^{max}). IS_i^{max} is obtained from the employee data file; if $is_i > IS_i^{max}$, a repair method is applied to make it feasible. Figure 3.14 shows the repair process for this constraint. Fixed-shift and salaried employees are not considered for this constraint.

Following notations are used in Figure 3.13 and 3.14.

L_{di} = Length of shift on day *d* for employee *i*

ND = Number of days in the scheduling period

ch_i = Set of choices for employee *i* in the GA solution

e_i = Employee *i*

6. *Enforce maximum consecutive days*

This rule must be checked for any GA solution during the search process including the initial population of solutions. The consecutive days are "carried forward" from last week. For example, if a person worked Friday, Saturday, Sunday last week and then Monday and Tuesday this week,

the person is considered to be working for five consecutive days. Figure 3.15, depicts how this constraint is checked for any GA solution by the algorithm. If a GA solution violates this constraint, a repair method is implemented to that solution in order to make that solution feasible. Fixed-shift employees are not considered by this constraint. Following notations are used in Figure 3.15.

CD_i = number of consecutive days that employee i has worked in the previous schedule (obtained from the employee data file)

CD^{max} = number of maximum consecutive days that is defined as a parameter for the problem

3.3.4.3 Parent Selection and Genetic Operator Crossover

The one-point crossover is used for this algorithm. At each iteration of GA, parent selection and crossover operation are performed as per following steps:

1. Two chromosomes (parents) are selected from the initial population randomly.
2. One employee is selected randomly from the first chromosome (parent 1) then one day is selected (start with day 0), and the shift belonging to that day and that employee is selected from this chromosome.
3. The same employee and day as the ones in step 2 is selected from the second chromosome (parent 2); then the shift belonging to this employee and day is selected from this chromosome.
4. Steps 2 and 3 are repeated for all the days in the scheduling period (7 days).
5. The start interval and shift length of all the selected shifts from parent 1 are swapped with the ones from parent 2 and two offspring (child 1 and child 2) are generated.

The crossover operation is repeated as many times as the population size.

By performing crossover as above, the feasibility of the GA solutions is maintained for "honour employee availability", "honour employee skill", "enforce maximum one shift per day" and "enforce maximum intervals per day" hard constraints of the problem.

Another way of doing crossover operation was also examined. In this crossover, instead of being limited to the same employee (step 3 above), any operator could be selected. The problem was, too many infeasible solutions were generated in terms of employees' unavailability, and GA had to spend plenty of time for repairing the generated infeasible solutions. For this study, only the first crossover method (crossover over the same employee) has been implemented.

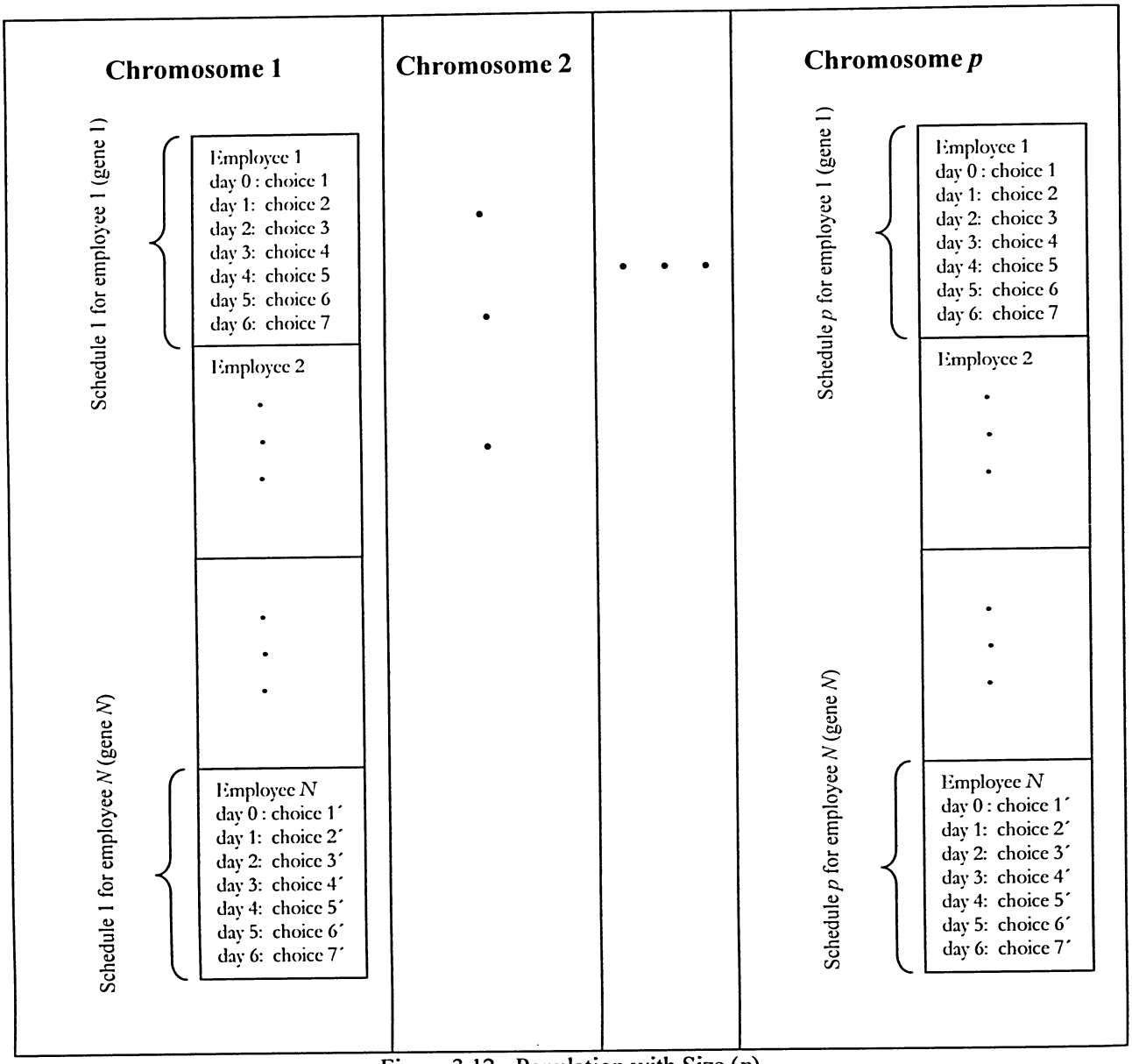

Figure 3.12 - Population with Size (p)

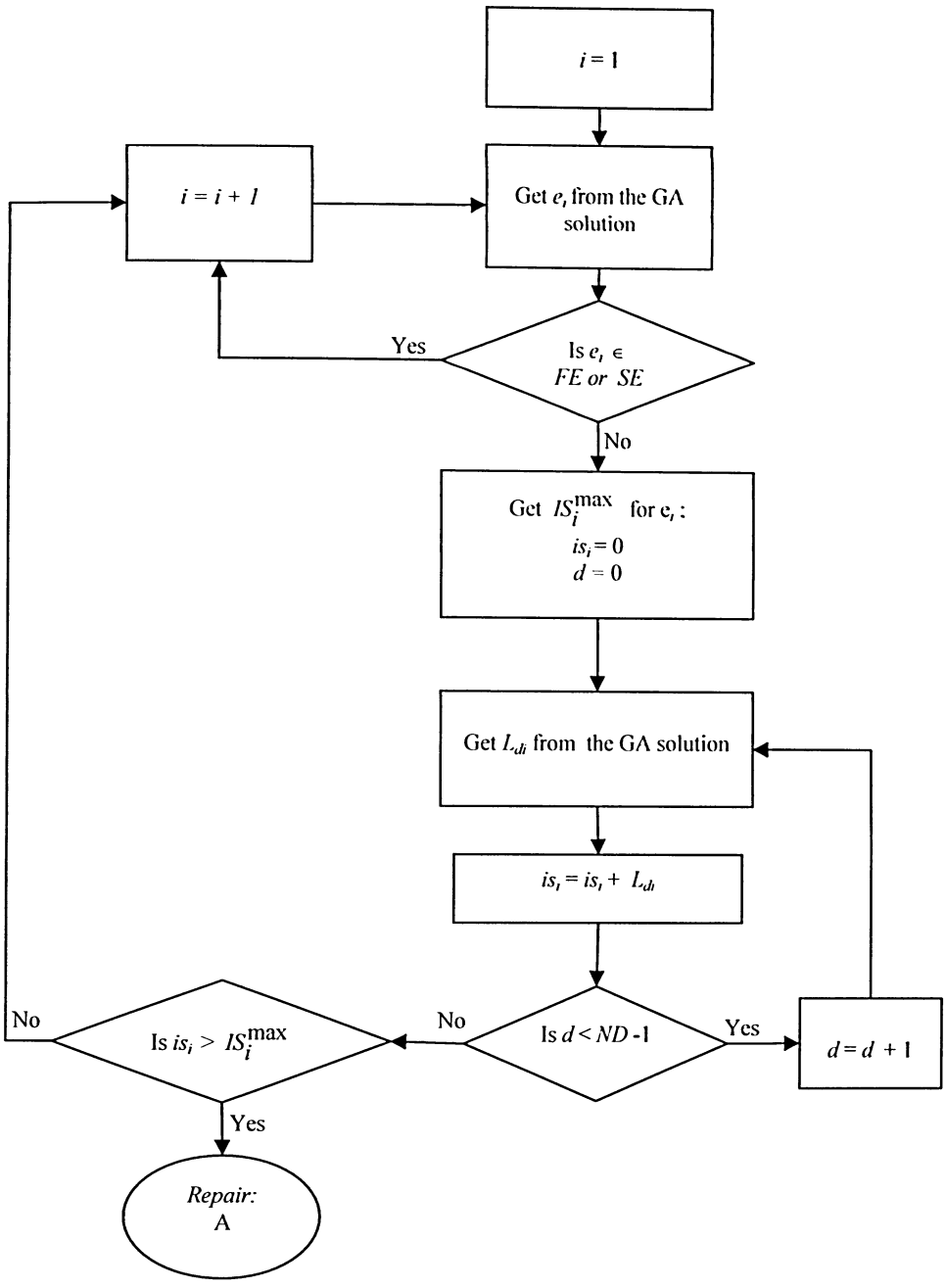

Figure 3.13 - Flowchart for "Enforce maximum intervals per schedule"

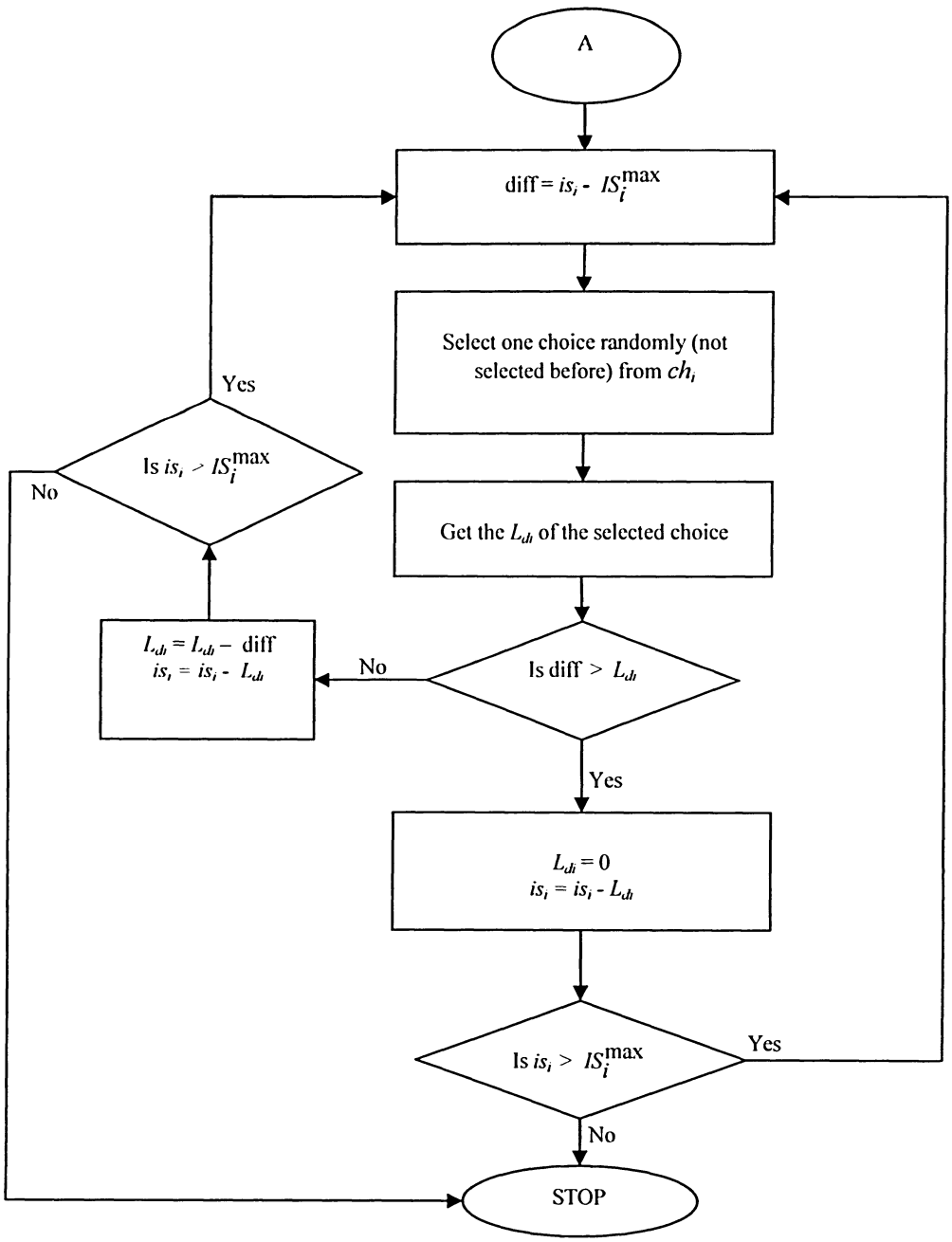

Figure 3.14 - Flowchart for "Enforce maximum intervals per schedule"

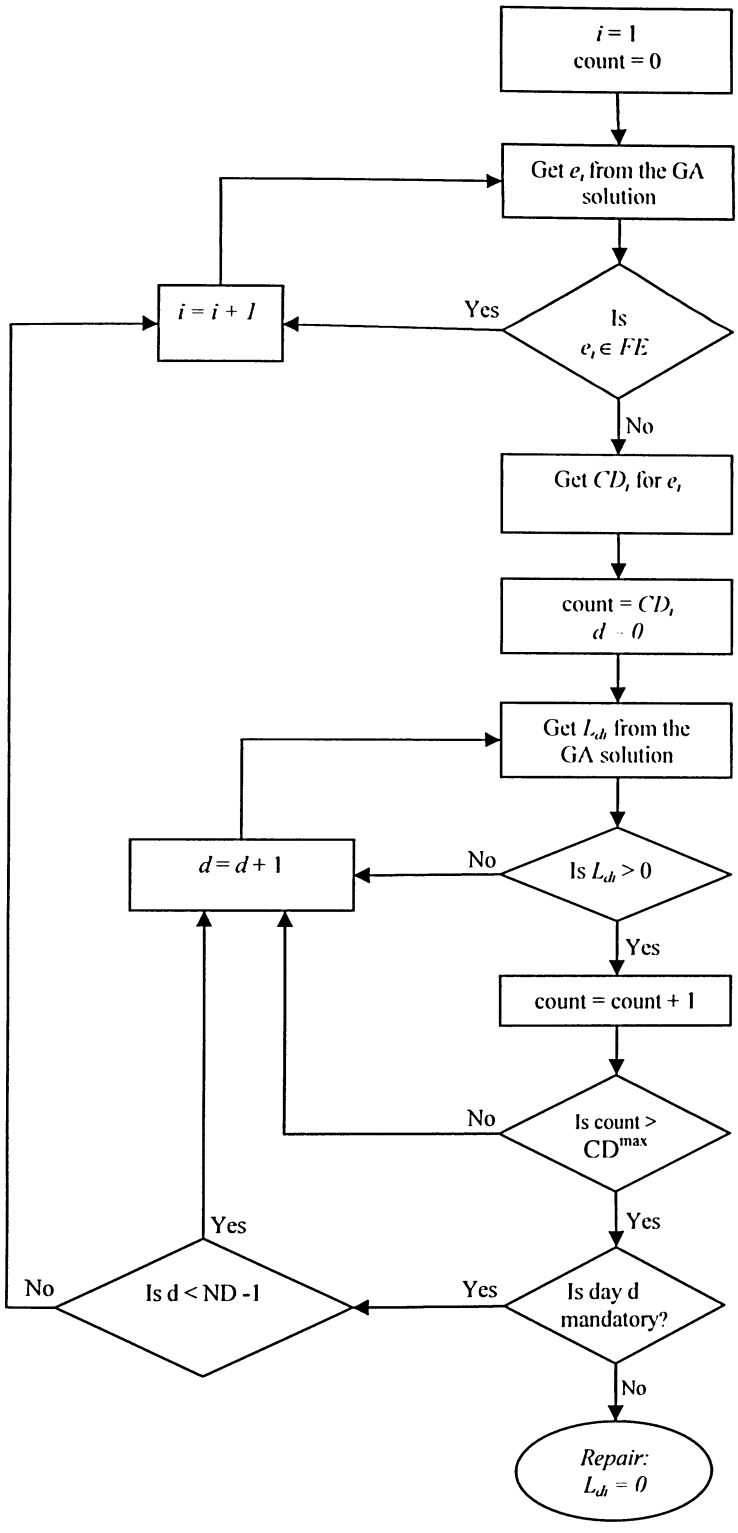

Figure 3.15 - Flowchart for "Enforce maximum consecutive days"

3.3.4.4 Genetic Operator Mutation

In this study, mutation is applied to the offspring. After the creation of child 1, an integer random number y in the range [0, b] is generated. If y is smaller than mutation threshold, mutation is performed on child 1 in the following steps.

1. One gene for one random employee is selected from child 1
2. One choice for one random day is selected from that gene
3. One choice is selected from the potential choices for that employee and that day
4. The selected choice on step 2 is replaced with the selected choice on step 3.

The same procedure is repeated for child 2.

Example:

mutation base = b = 100000

mutation threshold = mutation rate * mutation base

mutation rate = 0.3

mutation threshold = 0.3 * 100000 = 30000

y = 456

456 < 30000 mutation operation is performed on child 1.

In this algorithm, if mutation rate is 0.3, 30% of the time mutation is performed. If mutation rate is selected greater than or equal to 1, the mutation is always performed on the offspring.

3.3.4.5 GA Evaluation (Fitness) Function

After crossover and mutation operation, the fitness value of the parent population and offspring population is calculated by the fitness function which is developed in this section and the best individuals i.e., the individuals with the best fitness value either from the parent or offspring population are selected and the new population is evolved. The number of best selected individuals is as many as the population size.

The fitness of the GA solutions is based on the objective of the retail labour scheduling problem. As it was explained in chapter 2, the objective of the labour scheduling problem for retailers is to determine a schedule that minimizes the amount of total hours worked as well as the penalties from violating the specified staffing rules (soft constraints) and other scheduling preferences.

Soft constraints are those that can be broken, but a penalty is considered for breaking them. The penalty value of breaking a rule can vary from small to large considering the importance level of that rule. Each soft constraint and also scheduling preferences like seniority, option-weight, group-weight and skill-level-weight are considered as a component of the GA fitness function.

In the section below, in order to derive the fitness function for the proposed GA, first a mathematical integer programming (IP) model representing objective function, soft and hard constraints and then an IGP model are provided. In order to develop the IGP model, a penalty is defined for the violation of each soft constraint of the IP model. After assigning a penalty, each soft constraint of the IP model becomes a component of the objective function for the IGP model. The GA fitness function is represented in section 3.3.4.5.4.

3.3.4.5.1 Notations

In this section, the notations which are used for the *IP* and *IGP* model are provided as follows:

E	set of employees
SE	salaried employees' subset
FE	fixed-shift employees' subset
NF	non-fixed-shift employees' subset
NFS	non-fixed-shift and non-salaried employees' subset
D	set of days
C	set of constraints
I	set of intervals
J	set of jobs
S	set of shifts in the GA solution (chromosome)
d	day index
i	employee index
j	shift index
k	job index

n	constraint index
t	interval index
z	schedule index
L_j	length of shift j (expressed in interval)
A_{ij}	1, if employee i is available for shift j; 0, otherwise
X_{ij}	1, if employee i is assigned to shift j; 0, otherwise
T_{dj}	1, if shift j happens on day d; 0, otherwise
U_{ij}	1, if employee i has the required skill for shift j; 0, otherwise
$T_i^{\min}$	minimum intervals per week for fixed-shift employee i (expressed in interval)
P_n	penalty for violating soft constraint n
a	penalty for total hours worked by all employees
P_7	unit penalty for violating the C_7 constraint
P_9	unit penalty for violating the C_9 constraint
P_{10}	penalty per interval for giving salaried employees less than their required intervals per schedule (unit penalty for violating the C_{10} constraint)
$P_{10'}$	penalty per interval for giving salaried employees more than their required intervals per schedule (unit penalty for violating the C_{10} constraint)
h_i	total intervals per week for employee i which is calculated for the GA solution (expressed in interval)
R_{ktd}	preferred number of required employees for job k on interval t of day d
S_{ktd}	number of assigned employees to job k on interval t of day d (calculated from the GA solution)
P_{ktd}	unit penalty for understaffing of job k on interval t of day d
$R_{ktd}^{\min}$	minimum number of required employees for job k on interval t of day d
RI_i	required intervals per schedule for salaried employee i (expressed in interval)
IP_i	paid time-off intervals for salaried employee i (expressed in interval)
UBI_j	unpaid break intervals for shift j (expressed in interval)
PS_i	paid Intervals per schedule for employee i which is calculated for the GA solution (expressed in interval)
G_i	group-weight for employee i

O_{ij}	option-weight for shift j and employee i
S_i	seniority-weight for employee i
W_{ij}	skill-level-weight for employee i for the associated skill of shift j
ID_i^{max}	maximum intervals per day for employee i (expressed in interval)
IS_i^{max}	maximum intervals per schedule for employee i (expressed in interval)
is_i	maximum intervals per schedule for employee i for the GA schedule (expressed in interval)
CD_i	number of consecutive days that employee i has worked in the previous schedule
CD^{max}	number of maximum consecutive working days

3.3.4.5.2 IP Model

$$\text{minimize } Z = \sum_{j\in S}\sum_{i\in E} aL_jX_{ij} + \sum_{j\in S}\sum_{i\in NF} O_{ij}X_{ij} + \sum_{j\in S}\sum_{i\in NF} G_iX_{ij} + \sum_{j\in S}\sum_{i\in E} W_{ij}L_jX_{ij} + \sum_{j\in S}\sum_{i\in NF} S_iL_jX_{ij}$$

subject to

$X_{ij} \in \{0, 1\}, A_{ij} \in \{0, 1\}, U_{ij} \in \{0, 1\}$ $\forall i \in E, \forall j \in S$ (C_0)

$T_{dj} \in \{0, 1\}$ $\forall d \in D, \forall j \in S$

hard constraints:

- $X_{ij} \le A_{ij}$ $\forall i \in E, \forall j \in S$ (C_1)
- $X_{ij} \le U_{ij}$ $\forall i \in E, \forall j \in S$ (C_2)
- $\sum_j T_{dj}X_{ij} \le 1$ $\forall i \in E, \forall d \in D$ (C_3)
- $\sum_{j\in S} T_{dj}L_jX_{ij} \le ID_i^{max}$ $\forall i \in NFS, \forall d \in D$ (C_4)
- $\sum_{j\in S} L_jX_{ij} \le IS_i^{max}$ $\forall i \in NFS$ (C_5)

- $\sum_{j \in S} X_{ij} + CD_i \leq CD^{\max}$ $\quad \forall i \in NF$ $\quad (C_6)$

soft constraints:

- $\sum_{j \in S} L_j X_{ij} \geq T_i^{\min}$ $\quad \forall i \in FE$ $\quad (C_7)$
- $S_{ktd} \geq R_{ktd}$ $\quad \forall k \in J,\ \forall t \in I,\ \forall d \in D$ $\quad (C_8)$
- $S_{ktd} \geq R_{ktd}^{\min}$ $\quad \forall k \in J,\ \forall t \in I,\ \forall d \in D$ $\quad (C_9)$
- $\sum_{j \in S} (L_j - UBI_j) + IP_i = RI_i$ $\quad \forall i \in SE$ $\quad (C_{10})$

3.3.4.5.3 Description of the IP Model Components

I. Objective function components of the IP model

1. *Total hours worked objective function component:* The purpose of this objective is to determine a schedule that minimizes the amount of total hours worked by all the staff.

   minimize $Z_1 = \sum_{j \in S} \sum_{i \in E} a L_j X_{ij}$

   a = penalty for total hours worked by all employees, this will help reduce the total hours of the schedule

   L_j = length of shift j (expressed in interval)

   X_{ij} = 1, if employee i is assigned to shift j; 0 otherwise

2. *Option-weight objective function component:* As it was explained in section 3.3.1, associated with each choice is an option-weight. A smaller option- weight means the particular choice is more likely to be chosen. Fixed-shift employees are not considered by this objective component. This objective component for a schedule is simply defined as follows:

   minimize $Z_2 = \sum_{j \in S} \sum_{i \in NF} O_{ij} X_{ij}$

   O_{ij} = option-weight for shift j and employee i (derived from the choice data file)

3. *Group-weight objective function component:* Each employee belongs to a staff group. Each staff group has a group-weight assigned to it. A higher group-weight means employees belonging to that group are less likely to get scheduled. Fixed-shift employees are not considered by this objective component.

minimize $Z_3 = \sum_{j \in S} \sum_{i \in NF} G_i X_{ij}$

G_i = group-weight for employee i (obtained from the employee data file)

4. *Skill-level-weight objective function component:* A retailer may divide each skill into different levels. It is desired that a choice with the highest skill level for the associated employee is selected. Each choice includes a skill-level-weight in the choice data file; the smaller skill-level- weight represents the higher skill level. The purpose of this objective is to find a schedule with the highest skill level or the lowest skill level weight. The length of the shift (L_j) in (Z_4) indicates that the highest skill level is desired over entire time of the schedule.

minimize $Z_4 = \sum_{j \in S} \sum_{i \in E} W_{ij} L_j X_{ij}$

W_{ij} = skill-level-weight that employee i has for the associated skill of shift j (derived from the choice data file)

5. *Seniority-weight objective function component:* For all the employees who are not fixed-shift, given two or more employees can be scheduled for the same day/shift, this rule ensures that the staff with the highest seniority (lowest seniority-weight) is assigned to the schedule. The purpose of this objective is to find a schedule with the highest seniority level. Fixed-shift employees are not considered for this objective component. The length of the shift (L_j) in (Z_5) indicates that the highest seniority (lowest seniority-weight) is desired over entire time of the schedule.

minimize $Z_5 = \sum_{j \in S} \sum_{i \in NF} S_i L_j X_{ij}$

S_i = seniority-weight for employee i (obtained from the employee data file)

II. Hard constraints of the IP *model*

In section 3.3.4.2 hard constraints were explained in detail. Here they are explained in terms of the formulation.

1. *Honour employee availability* (C_1): The purpose of this constraint is to prevent the assigning of shifts to employees who are not available for those shifts.

   $X_{ij} \leq A_{ij}$ $\quad \forall i \in E,\ \forall j \in S$

   $A_{ij} \in \{0, 1\}$ 1, if employee i is available for shift j; 0 otherwise

2. *Honour employee skills* (C_2): The purpose of this constraint is to prevent the assigning of shifts to employees who do not have the required skill for those shifts.

   $X_{ij} \leq U_{ij}$ $\quad \forall i \in E,\ \forall j \in S$

   $U_{ij} \in \{0, 1\}$ 1, if employee i has the required skill for shift j; 0 otherwise

3. *Enforce maximum one shift per day* (C_3): This constraint prevents the assigning of more than one shift per day to employees.

   $\sum_j T_{dj} X_{ij} \leq 1$ $\quad \forall i \in E,\ \forall d \in D$

   $T_{dj} \in \{0, 1\}$ 1, if shift j happens on day d; 0, otherwise

4. *Enforce maximum intervals per day* (C_4): This constraint ensures that the schedule assigned to each employee does not exceed the maximum intervals per day for that employee. This constraint excludes the fixed-shift and salaried employees.

   $\sum_{j \in S} T_{dj} L_j X_{ij} \leq ID_i^{\max}$ $\quad \forall i \in NFS,\ \forall d \in D$

   $ID_i^{\max}$ = maximum intervals per day for employee i expressed in terms of interval (obtained from the employee data file)

5. *Enforce maximum intervals per schedule* (C_5): This constraint ensures that the assigned schedule to each employee does not exceed the maximum intervals per schedule for that employee. This constraint excludes the fixed-shift and salaried employees.

$$\sum_{j \in S} L_j X_{ij} \leq IS_i^{\max} \qquad \forall i \in NFS$$

$IS_i^{\max}$ = maximum intervals per schedule for employee i expressed in terms of interval (obtained from the employee data file)

6. *Enforce maximum consecutive days* (C_6): This constraint prevents staff from being scheduled for more then the maximum number of consecutive days. This constraint excludes the fixed-shift employees.

$$\sum_{j \in S} X_{ij} + CD_i \leq CD^{\max} \qquad \forall i \in NF$$

CD_i = number of consecutive days that employee i has worked in the previous schedule (obtained from the employee data file)

$CD^{\max}$ = number of maximum consecutive working days that is defined as a parameter for the problem

III. Soft constraint of the IP model:

After assigning penalties for violating these constraints, each soft constraint of the IP model will be an objective component for the IGP model. The purpose of these objective components is to discover a schedule which minimizes the violation of these constraints.

1. *Enforce minimum intervals per week for fixed-shift employees* (C_7): This constraint ensures that fixed-shift employees are scheduled at least for their minimum intervals per week. In any GA solution if the minimum intervals per week are not assigned to a fixed-shift employee, a penalty is considered for that solution. After defining the penalty for C_7, this constraint will be one objective component (Z_6) for the IGP model as follows. The violation from this constraint for the GA solution is obtained by Z_6.

$$\sum_{j \in S} L_j X_{ij} \geq T_i^{\min} \qquad \forall i \in FE$$

P_7 = penalty for violating the "Enforce minimum intervals per week for fixed shift employees" constraint

T_i^{min} = minimum intervals per week required for fixed-shift employee i expressed in terms of interval (obtained from the employee data file)

$h_i = \sum_{j \in S} L_j X_{ij}$ = total intervals per week assigned to employee i expressed in terms of interval (calculated for the GA solution)

$$\text{minimize } Z_6 = \sum_{i \in FE} (T_i^{\min} - \sum_{j \in S} L_j X_{ij}) \times P_7 \times Y_7$$

$Y_7 \in \{0,1\}$ $Y_7 = 1$, if $h_i < T_i^{min}$; 0, otherwise

2. *Satisfy staff requirements* (C_8): As it was implied earlier, this constraint ensures that the workload requirements for each job (i.e. location, position, skill and activity) on each interval of a day are satisfied. Associated with each job and each interval of a day, there is a preferred number of required employees (R_{ktd}) in the requirement data file. For each GA solution the number of assigned employees to job k on interval t of day d is calculated (S_{ktd}). If $S_{ktd} < R_{ktd}$, that GA solution is penalized with the pre-defined understaffing penalty (P_{ktd}). After defining penalty for C_8, this constraint will be one objective component (Z_7) for the IGP model as follows. The violation from this constraint for the GA solution is obtained by Z_7.

R_{ktd} = preferred number of required employees for job k on interval t of day d (obtained from the requirement data file)

S_{ktd} = number of assigned employees to job k on interval t of day d (calculated from the GA solution).

$$S_{ktd} \geq R_{ktd} \qquad \forall k \in J,\ \forall t \in I,\ \forall d \in D$$

P_{ktd} = Unit penalty for understaffing of job k on interval t of day d (obtained from the requirement data file)

minimize $Z_7 = \sum_{k\in J}\sum_{t\in I}\sum_{d\in D}(R_{ktd} - S_{ktd}) \times P_{ktd} \times Y_{ktd}$

$Y_{ktd} \in \{0,1\}$ $Y_{ktd} = 1$, if $S_{ktd} < R_{ktd}$; 0, otherwise

3. Satisfy minimum staff requirements (C_9): This constraint ensures that the minimum number of required staff for each job on each interval of a day is satisfied. Associated with each job and each interval of a day, there is a minimum number of required employees ($R_{ktd}^{\min}$) in the requirement data file. For each GA solution, the number of assigned employees to job k on interval t of day d is calculated (S_{ktd}). In any GA solution, if the minimum required number of staffs for a job on an interval of a day is not satisfied ($S_{ktd} < R_{ktd}^{\min}$), a penalty is considered for that solution. This constraint will be one objective component for the IGP model as follows. The violation from this constraint for the GA solution is obtained by Z_8.

$R_{ktd}^{\min}$ = minimum number of required employees for job k on interval t of day d (obtained from the requirement data file).

$S_{ktd} \geq R_{ktd}^{\min}$ $\forall k \in J,\ \forall t \in I,\ \forall d \in D$

P_9 = unit penalty for violating the C_9 constraint.

minimize $Z_8 = \sum_{k\in J}\sum_{t\in I}\sum_{d\in D}(R_{ktd}^{\min} - S_{ktd}) \times P_9 \times Y_9$

$Y_9 \in \{0,1\}$ $Y_9 = 1$, if $S_{ktd} < R_{ktd}^{\min}$; 0, otherwise

4. Enforce required intervals for salaried employees per schedule (C_{10}): This constraint ensures that the salaried employees are scheduled for their required intervals. After assigning penalty, this constraint will be two objective components (Z_9 and Z_{10}) of the IGP model. The GA solution violation from this constraint is obtained by Z_9 and Z_{10}.

$\sum_{j\in S}(L_j - UBI_j) + IP_i = RI_i$ $\forall i \in SE$

RI_i = required intervals per schedule for salaried employee i (obtained from the employee data file, expressed in terms of interval)

IP_i = number of paid time-off intervals per schedule for the salaried employee i (obtained from the employee data file, expressed in terms of interval)

UBI_j = number of unpaid break intervals for shift j (obtained from the shift-information data file, expressed in terms of interval)

PS_i = paid intervals per schedule for employee i, expressed in terms of interval, which is calculated as follows:

$$PS_i = \sum_j (L_j - UBI_j) + IP_i$$

P_{10} = penalty per interval for giving salaried employees less than their required intervals per schedule

$P_{10'}$ = penalty per interval for giving salaried employees more than their required intervals per schedule

minimize:

$$Z_9 = \sum_{i \in SE} (\sum_{j \in S} (L_j - UBI_j) + IP_i - RI_i) \times Y_{10} \times P_{10} = \sum_{i \in SE} (PS_i - RI_i) \times Y_{10} \times P_{10}$$

$$Z_{10} = \sum_{i \in SE} (\sum_{j \in S} RI_i - (L_j - UBI_j) - IP_i) \times Y_{10'} \times P_{10'} = \sum_{i \in SE} (RI_i - PS_i) \times Y_{10'} \times P_{10'}$$

$Y_{10} \in \{0,1\}$ 1, if $PS_i > RI_i$; 0, otherwise

$Y_{10'} \in \{0,1\}$ 1, if $PS_i < RI_i$; 0, otherwise

3.3.4.5.4 IGP Model

The objective function of the following IGP model is the fitness function of the GA. And the constraints of the IGP (hard constraints), as was explained in section 3.3.4.2, are used to determine the feasibility of each GA solution (chromosome).

$$\text{minimize } Z = \sum_{j\in S}\sum_{i\in E} aL_j X_{ij} + \sum_{i\in S}\sum_{i\in NF} O_{ij} X_{ij} + \sum_{j\in S}\sum_{i\in NF} G_i X_{ij}$$

$$+ \sum_{j\in S}\sum_{i\in E} W_{ij} L_j X_{ij} + \sum_{j\in S}\sum_{i\in NF} S_i L_j X_{ij} + \sum_{i\in FE} (T_i^{\min} - \sum_{j\in S} L_j X_{ij}) \times P_7 \times Y_7$$

$$+ \sum_{k\in J}\sum_{t\in I}\sum_{d\in D} (R_{ktd} - S_{ktd}) \times P_{ktd} \times Y_{ktd} + \sum_{k\in J}\sum_{t\in I}\sum_{d\in D} (R_{ktd}^{min} - S_{ktd}) \times P_9 \times Y_9$$

$$+ \sum_{i\in SE} (\sum_{j\in S} (L_j - UBI_j) + IP_i - RI_i) \times Y_{10} \times P_{10}$$

$$+ \sum_{i\in SE} (\sum_{j\in S} RI_i - (L_j - UBI_j) - IP_i) \times Y_{10'} \times P_{10'}$$

subject to

$X_{ij} \in \{0, 1\}, A_{ij} \in \{0, 1\}, U_{ij} \in \{0, 1\}$ $\forall i \in E, \forall j \in S$ (C_0)

$T_{dj} \in \{0, 1\}$ $\forall d \in D, \forall j \in S$

$Y_7 \in \{0,1\}$ 1, if $h_i < T_i^{\min}$; 0, otherwise

$Y_{ktd} \in \{0,1\}$ 1, if $S_{ktd} < R_{ktd}$; 0, otherwise

$Y_9 \in \{0,1\}$ 1, if $S_{ktd} < R_{ktd}^{\min}$; 0, otherwise

$Y_{10} \in \{0,1\}$ 1, if $PS_i > RI_i$; 0, otherwise

$Y_{10'} \in \{0,1\}$ 1, if $PS_i < RI_i$; 0, otherwise

hard constraints:

- $X_{ij} \le A_{ij}$ $\forall i \in E, \forall j \in S$ (C_1)
- $X_{ij} \le U_{ij}$ $\forall i \in E, \forall j \in S$ (C_2)
- $\sum_j T_{dj} X_{ij} \le 1$ $\forall i \in E, \forall d \in D$ (C_3)
- $\sum_{j\in S} T_{dj} L_j X_{ij} \le ID_i^{\max}$ $\forall i \in NFS, \forall d \in D$ (C_4)
- $\sum_{j\in S} L_j X_{ij} \le IS_i^{\max}$ $\forall i \in NFS$ (C_5)
- $\sum_{j\in S} X_{ij} + CD_i \le CD^{\max}$ $\forall i \in NF$ (C_6)

3.3.4.5.5 Implemented Penalty Values and Weights

The user is free to define weights and the penalties associated with a constraint violation in the IGP model (section 3.3.4.5.4). The following values have been used by the industry partner in Mosel. For numerical experiments in this thesis, in order to compare GA and Mosel, the penalty values and weights in the GA have been set the same as those used in Mosel. These values are as follows:

$a = 5$

$P_7 = 400{,}000$

$P_9 = 399{,}900$

$P_{10} = 4000$

$P_{10'} = 400$

$S_i = (S_i/1000) \times 0.12$

$W_{ij} = 0.12 \times W_{ij}$

3.3.4.6 Evaluation and Repair Mechanism of the GA

A key feature of the proposed GA is that all the solutions (schedules) in the population are feasible at all times. A schedule is evaluated as it is generated. This occurs regardless of how it is created; from initial population, crossover, mutation or any change in the schedule such as repair. Every solution must be evaluated for fitness, and if necessary repaired, as it is created or modified in order to maintain its feasibility. The evaluation mechanism for the proposed GA therefore includes the ability to repair the schedules while determining the cost of the schedule. This ensures that there are no infeasible solutions in the population.

Figure 3.16 depicts the evaluation and repair mechanism of the GA. In this flowchart each hard constraint and objective function component is referred to as a rule. Suppose the total number of rules is equal to R and schedule z is evaluated by the sequence of the R rules and an evaluation value (F_{nz}) is returned for each rule n. If rule n is an objective function component, the objective value of that (any value greater than or equal to zero) is returned for F_{nz}.If rule n is a hard

constraint and is satisfied by schedule z (if the schedule z is feasible), the value of zero is returned for F_{nz}. If rule n is a hard constraint and is violated by the schedule z, a pre-defined very large number (an error code) is returned. If an error is returned, the repair method specific to that hard constraint is called (two repair methods specific to two hard constraints were explained in section 3.3.4.2). The repair method modifies the schedule in a way that ensures that the hard constraint will pass its own evaluation test and the value of zero is returned for F_{nz}. Modifying the schedule during the repair process in effect creates a new schedule, and so the sequence of individual rule evaluations has to start at the beginning again. Once all hard constraints and objective function components have been checked and have passed, the fitness values from each individual rule evaluation are totalled to determine the overall fitness value (TF_z) for the schedule z.

3.3.4.7 The Termination Criteria

In the proposed GA, the lack of improvement in the best solution over a specified number of generations is implemented as a termination criterion. For the test problems of this study, if the GA fitness value is the same over 50,000 iterations, the GA terminates the search process. The value of 50000 iterations was selected after some preliminary tests. The preliminary tests showed that if a small value is selected as the number of iterations, there is a high chance that the entire GA search space is not investigated, and there is a possibility of further improvement in the GA solution. Choosing a large value such as "50,000" iterations, ensures that there is a very low probability of further improvement in the GA solution. Please note that GA search time that will be used in chapter 4 is not the time taken to complete 50,000 iterations, but it is the time of iteration for when the GA best objective value is first observed.

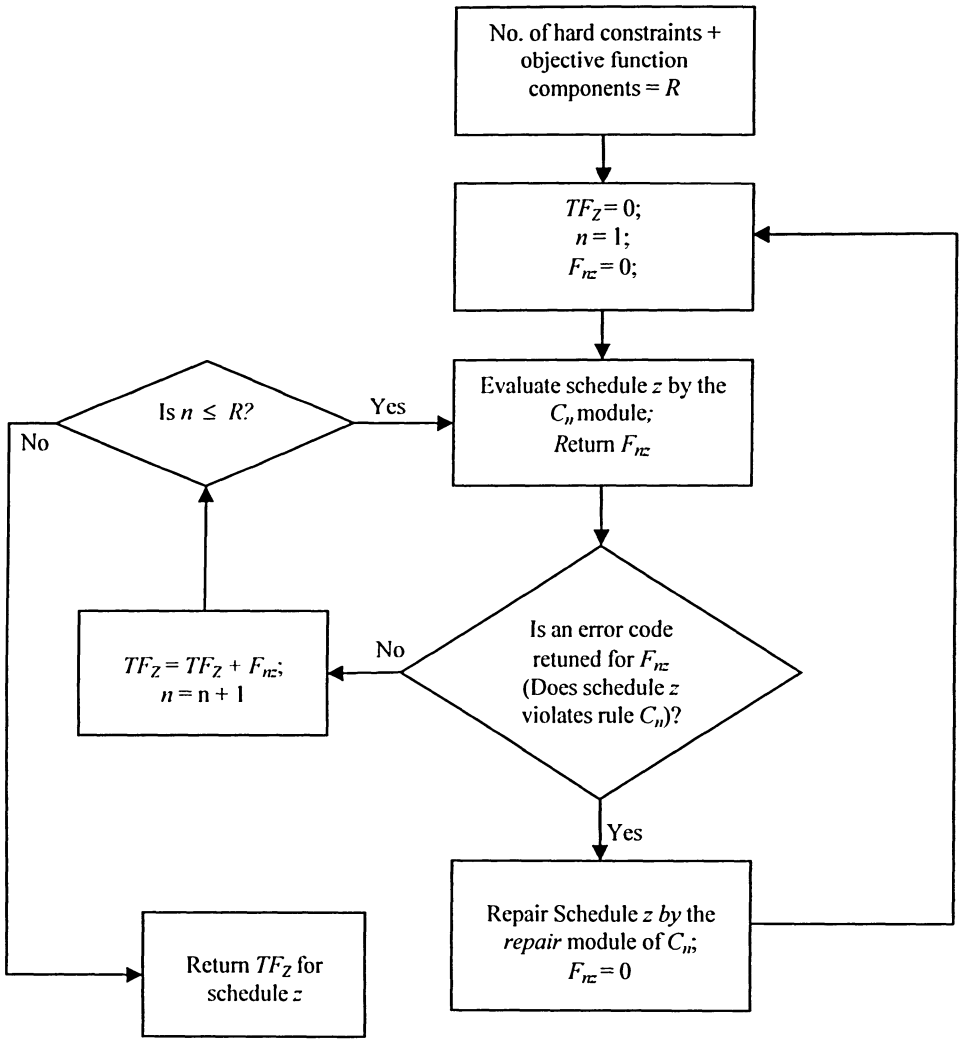

Figure 3.16 - Flowchart for the Evaluation and Repair Mechanism of the GA

Chapter 4

EXPERIMENTAL DESIGN AND STATISTICAL ANALYSIS

4.1 Experimental Design

In an experiment, the values (levels) of one or more input, or independent variables, are chosen and the values of the output, or dependent variables are observed. The purpose of an experiment is to investigate the relationship between input and output variables.

Experimental design is performed using the following steps (Berger and Maurer, 2002):

1. Planning the experiment
2. Designing the experiment
3. Analyzing data from the experiment

4.1.1 Planning the Experiment

The planning process is vital to the success of the experiment, and it consists of the following steps:

1. Establish the dependent, or response, variable(s) or performance measure.
2. Identify the input, or independent, variable(s) or factor(s) that potentially affect the response variable.
3. Determine the number of levels and values for each factor.

4.1.2 Designing the Experiment

In this step, the type of design of experiment is selected. Factorial design and one-at-a-time design are two types of experimental design. In the one-at-a-time design, the effect of each factor on the response variable(s) is investigated separately, while in the factorial design, the effect of factors on the response variable(s) are studied simultaneously. Fewer observations are needed for factorial design compared to one-factor-at-a-time, in which separate experiments for each factor are conducted independently. Also, studying the interaction between factors on the response variable is possible with factorial design and not with the one-at-a-time. The simplest factorial experiment

considers two factors, with two levels for each, and is called a 2^2 factorial experiment. Replication allows a researcher to investigate interaction. The experimental runs in a factorial experiment should also be randomized. Randomization attempts to reduce the impact that bias could have on the experimental results. When there are many factors, many experimental runs are necessary. For example, experimenting with 13 factors at three levels each produces 3^{13} = 1,594,323 different combinations. The full factorial design may not be feasible due to high cost or insufficient resources. In this case, a fractional factorial design may be used.

4.1.3 Analyzing Data from the Experiment

Sometimes the conclusions from an experiment seem deceptively clear. Often the results are not obvious, even when appear that way. It is important to tell whether an observed difference is indicating a real difference, or is caused by fluctuating levels of background noise. To make this judgment, a statistical method called analysis of variance (ANOVA), which was developed by Sir Ronald Fisher (year 1925), is employed. The objective of ANOVA is to investigate whether the level of a factor (or interaction of factors) influences the value of the response variable (performance measure). In order to investigate this influence, these two alternatives are defined into two hypotheses, a null hypothesis H_0 and an alternate hypothesis H_1, and then the hypothesis test is performed.

If the response variable is called *Y*, and the factor is called *X*, then the two alternative hypotheses would be as follows:

H_0: Level of *X* does not affect *Y*

H_1: Level of *X* does affect *Y*

In order to carry out the hypothesis test, F_{calc} statistic is calculated according to the ANOVA Table (Table 4.1) and from F_{calc}, the *P*-value is calculated.

The *P*-value is described as the weight of evidence against the null hypothesis. For a one-sided upper-tailed test, the *P*-value refers to the area under the curve to the right of the test statistic (on an *F* curve, to the right of F_{calc}; on a $\overline{X}$ curve, the area to the right of $\overline{X}$ and so on). Here the test

statistic is F_{calc}. A smaller P-value or a larger confidence level represents stronger evidence against H_0.

In ANOVA, besides studying the influence of each factor on the response variable, the effect of interaction of factors on the response variable is also investigated. If the effect of one factor varies depending on the level of another factor; there is *interaction* between the two factors. A useful way to depict interaction between two factors is with an interaction plot. In an interaction plot, only when the lines are parallel there is no interaction.

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}
Rows	$SSB_r = nc\sum_{i=1}^{R}(\bar{Y}_{i..} - \bar{Y}_{...})^2$	$R-1$	$\frac{SSB_r}{R-1}$	$\frac{\frac{SSB_r}{(R-1)}}{\frac{SSW}{RC(n-1)}}$
Columns	$SSB_c = nR\sum_{j=1}^{C}(\bar{Y}_{.j.} - \bar{Y}_{...})^2$	$C-1$	$\frac{SSB_c}{C-1}$	$\frac{\frac{SSBc}{(C-1)}}{\frac{SSW}{RC(n-1)}}$
Interaction	$SSI_{r,c} = TSS - SSB_r - SSB_c - SSW$	$(R-1)(C-1)$	$\frac{SSI}{(R-1)(C-1)}$	$\frac{\frac{SSI}{(R-1)(C-1)}}{\frac{SSW}{RC(n-1)}}$
Error	$SSW = \sum_{i=1}^{R}\sum_{j=1}^{C}\sum_{k=1}^{n}(Y_{ijk} - \bar{Y}_{ij.})^2$	$RC(n-1)$	$\frac{SSW}{RC(n-1)}$	
Total	$TSS = \sum_{i=1}^{R}\sum_{j=1}^{C}\sum_{k=1}^{n}(Y_{ijk} - \bar{Y}_{...})^2$	$nRC-1$		

Table 4.1 - ANOVA Table

SSB_r = Sum of square between rows

SSB_c = Sum of square between columns

SSW = Sum of square within columns

TSS = Total sum of squares

df = Degree of freedoms

MS = Mean square

R = Number of rows

C = Number of columns

n = Number of replications

$\bar{Y}_{...}$ = Grand mean (mean of all the data)

$\bar{Y}_{i..}$ = Mean of row i

$\bar{Y}_{.j.}$ = Mean of column j

$\bar{Y}_{ij.}$ = Mean of cell [i, j]

4.2 Experimental Design for this Study

In this section, the implementation of the above steps to this study are discussed.

4.2.1 Planning the Experiment

For this study response variables and factors are established as follows:

Response variable I: GA optimal objective value (RV_1)

Response variable II: GA search time (RV_2)

Factor I: *population-size* (p)

Factor II: *mutation-rate* (m)

These two factors are crossed or cross-classified, meaning that each level of one factor is in combination with each level of the other factor.

After doing some preliminary tests on different values for *population-size* and *mutation-rate,* as depicted in Table 4.2, eight levels for population size and six levels for mutation rate were selected. It is necessary to explain that levels 1 to 8 of the *population-size* are only applied to one test problem; for the other two test problems only the first five levels are employed. In Table 4.2,"N" refers to the size of the problem, which is defined as the number of employees for that problem.

FACTORS	
population-size	*mutation-rate*
Level 1 = *N/12*	Level 1 = 0
Level 2 = *N/6*	Level 2 = 0.3
Level 3 = *N/3*	Level 3 = 0.4
Level 4 = *N/2*	Level 4 = 0.5
Level 5 = *N*	Level 5 = 0.7
Level 6 = *2N*	Level 6 = 1
Level 7 = *7N*	
Level 8 = *12N*	

Table 4.2 - Level of Factors

4.2.2 Designing the Experiment

In this study, a full factorial cross design is implemented to three test problems of different sizes. The number of employees (N) in each problem specifies the size of the problem. The test problem (1), with $N = 290$ employees, is referred to as the large size problem; the test problem (2), with $N = 131$ employees, is referred to as the medium size problem; and the test problem (3), with $N = 17$ employees, is referred to as the small size problem. Each problem is solved by the proposed GA considering the constraints and scheduling preferences that were explained in chapter 3.

Test problems (1) and (2) have been run by the proposed GA with five levels of factor I and six levels of factor II; Test problem (3) has been run by the proposed GA with eight levels of factor I and six levels of factor II. To account for the randomness effect, each scenario has been replicated three times with three different random seeds. For test problems (1) and (2), a total of 90 GA runs and for test problem (3), a total of 144 GA runs have been carried out.

4.2.3 Analyzing Data from the Experiment

In this study, for analyzing the experiment, a two-factor analysis of variance (ANOVA) with the following hypotheses is performed on the test problems described in *4.2.2.* As it was pointed out

earlier, analysis of variance determines if row or column factors, or interaction effects are statistically significant.

For response variable (I), which is defined as GA optimal objective value, the following three hypotheses are tested:

$H_0^{(1)}$: All levels of the *population-size* (row factor) have the same effect on the GA optimal objective value.

$H_0^{(2)}$: All levels of the *mutation-rate* (column factor) have the same effect on the GA optimal objective value.

$H_0^{(3)}$: There is no interaction between row and column factors.

For response variable (II), which is defined as GA search time, the following three hypotheses are tested:

$H_0^{(4)}$: All levels of the *population-size* (row factor) have the same effect on the GA search time.

$H_0^{(5)}$: All levels of the *mutation-rate* (column factor) have the same effect on the GA search time.

$H_0^{(6)}$: There is no interaction between row and column factors.

4.3 Discussion of ANOVA and Interaction Plots for Test Problems

The results of ANOVA for the mentioned test problems are provided in this section. The Minitab™ software has been used for performing ANOVA and plotting the interaction graphs. For all three problems, interaction plots are named as plot $(A)_S$ and $(B)_S$. These plots illustrate the interaction between two factors, *population*-size (p) and *mutation-rate* (m), while the response variable for plot $(A)_S$ is the GA optimal objective value (response variable I ; RV_1) and for plot $(B)_S$ is the GA search time in minutes (response variable II ; RV_2). The subscripts L, M and S refer to the large, medium and small problems respectively (for example, plot A_L is the large problem interaction plot for RV_1). In addition to the plots in this chapter, more detailed plots are provided in

Appendix I. The *P*-value from the analysis and variance tables is compared with the commonly used alpha level of 0.05 and 0.1.

4.3.1 Large Size Problem

The optimal objective values and search times obtained from 90 GA runs for the large problem are represented in tables 4.3 and 4.4 respectively. In this problem, five levels for factor I (p) and six levels for factor II (m), are implemented with three replications, i.e. $R = 5$, $C = 6$ and $n = 3$. The F_{calc} statistic is calculated according to section 4.1.3 and the results are depicted in tables 4.5 and 4.6.

	Replication #	$m = 0$	$m = 0.3$	$m = 0.4$	$m = 0.5$	$m = 0.7$	$m = 1$
$p = N/12 = 24$	1	2,902,454,979	116,884,363	116,883,998	116,883,295	120,484,863	120,484,590
	2	2,886,854,689	116,883,345	116,884,956	116,883,690	116,883,758	116,885,253
	3	2,779,255,395	116,883,626	116,884,778	116,884,370	116,883,655	116,883,486
cell mean		2,856,188,354	116,883,778	116,884,577	116,883,785	118,084,092	118,084,443
$p = N/6 = 48$	1	2,777,255,738	116,883,870	116,883,161	116,882,886	116,884,346	116,882,633
	2	2,833,656,469	116,883,118	116,884,580	116,883,740	116,884,043	116,885,205
	3	2,754,454,906	116,883,604	116,883,584	116,883,791	116,884,932	116,883,784
cell mean		2,788,455,704	116,883,531	116,883,775	116,883,472	116,884,440	116,883,874
$p = N/3 = 97$	1	2,686,855,678	116,882,730	116,882,581	116,883,499	116,885,087	116,882,654
	2	2,590,057,768	116,882,799	116,881,885	116,884,563	116,882,553	116,884,146
	3	1,926,864,482	116,884,348	116,881,771	116,884,863	116,883,257	116,884,762
cell mean		2,401,259,309	116,883,292	116,882,079	116,884,308	116,883,632	116,883,854
$p = N/2 = 145$	1	2,692,055,748	116,884,459	116,882,922	116,885,064	116,882,528	116,885,496
	2	2,558,859,893	116,884,485	116,885,106	116,882,758	116,882,525	116,884,071
	3	1,420,069,823	116,885,081	116,882,656	116,885,327	116,884,940	116,886,042
cell mean		2,223,661,821	116,884,675	116,883,561	116,884,383	116,883,331	116,885,203
$p = N = 290$	1	968,077,822	117,284,441	116,882,761	117,282,674	116,883,367	116,883,301
	2	2,526,858,558	116,883,434	116,882,841	116,884,455	116,883,145	116,881,915
	3	695,680,055	116,883,388	116,882,923	117,684,433	117,682,934	117,684,161
cell mean		1,396,872,145	117,017,088	116,882,842	117,283,854	117,149,815	117,149,792

Table 4.3 - GA optimal objective values for large problem

The following outcomes are achieved by reviewing the ANOVA tables and interaction plots for this problem:

a. Examining Table 4.5, it can be seen that the *P*-value for rows (0.021), for columns (0.000) and for interaction (0.000) is small. Hence, there is strong evidence for rejecting the $H_0^{(1)}$,

$H_0^{(2)}$ and $H_0^{(3)}$ hypotheses, meaning that the p and m factors, and also the interaction of the factors, have significant effect on RV_1.

	Replication #	m = 0	m = 0.3	m = 0.4	m = 0.5	m = 0.7	m = 1
$p = N/12 = 24$	1	0.02200	10.60943	44.24967	8.88333	1.87233	1.69883
	2	0.00804	12.52762	3.96889	3.29107	12.47779	1.69284
	3	0.01129	8.60390	4.27744	2.97658	6.76321	9.99802
cell mean		0.01378	10.58032	17.49866	5.05033	7.03778	4.46323
$p = N/6 = 48$	1	0.12433	11.03117	60.23300	52.29683	8.02250	27.33333
	2	0.05344	8.46844	44.37932	6.55730	12.36967	12.16191
	3	0.03332	13.45424	27.10334	8.32738	40.61499	10.03778
cell mean		0.07036	10.98462	43.90522	22.39384	20.33572	16.51101
$p = N/3 = 97$	1	0.20550	28.26983	128.18200	13.47100	22.88817	16.60000
	2	0.30827	204.26966	46.17130	35.94241	31.08172	11.06414
	3	0.69268	99.31092	106.01044	7.01314	36.08354	13.19003
cell mean		0.40215	110.61681	93.45458	18.80885	30.01781	13.61806
$p = N/2 = 145$	1	0.17483	175.19617	137.97267	18.53767	50.80217	15.66667
	2	0.42068	10.43663	30.66895	72.38160	17.75927	105.50592
	3	1.85883	23.32153	113.18798	51.93301	17.23234	58.75988
cell mean		0.81811	69.65144	93.94320	47.61743	28.59793	59.97749
$p = N = 290$	1	5.56150	71.79367	60.19883	83.11100	29.09900	26.32717
	2	1.59743	80.43997	130.91663	43.09979	17.28932	50.12954
	3	5.57359	111.87253	75.82214	274.16564	58.22210	17.55282
cell mean		4.24417	88.03539	88.97920	133.45881	34.87014	31.33651

Table 4.4 - GA search time (min) for large problem

b. From Table 4.6, the small P-value for rows and columns (0.000) indicates that there is strong evidence for rejecting the $H_0^{(4)}$ and $H_0^{(5)}$ hypotheses, but the P-value for interaction (0.180) is large (a value larger than 0.05) which does not provide strong evidence for rejecting $H_0^{(6)}$. So it is concluded that the p and m factors have significant effect on RV_2, but the interaction between them does not.

c. From Figure 4.1 (A_L), it is clear that GA has the poorest performance in terms of quality when the mutation operator is not implemented (m = 0) or when the *mutation-rate* is very small (m<0.3). It is also evident that there is a large difference in the GA optimal objective value between m < 0.3 and the other levels of m. In order to find out more precisely which levels of m produce the best GA solution, the worst level of *mutation-rate* (m = 0) is omitted and ANOVA is repeated for the remaining data. The results of ANOVA following this change are depicted in tables 4.7 and 4.8 and the interaction graphs are represented by

Figure 4.3 (A'_L) and Figure 4.4 (B'_L). By omitting the first level of m (m =0), the conclusion that was reached in part (a) is no longer valid. From Table 4.7, it can be seen that the P-value for rows (0.14), for columns (0.506) and for interaction (0.771) is large, which implies that there is no strong evidence for rejecting $H_0^{(1)}$, $H_0^{(2)}$ and $H_0^{(3)}$. Therefore, the main effect, and the interaction of the m and p factors on RV_1, is not significant. However, the results obtained in part (b) remain valid.

d. The existence of parallelism between most of the lines in Figure 4.3 (A'_L), supports the conclusion obtained in part (c) that the effect of the interaction between factors on RV_1 is not significant. Figure 4.3 (A'_L) demonstrates that for a small *population-size* (*N/12*), choosing a large *mutation-rate* (0.7 and 0.1) impairs the quality of the GA solution, but for other levels of the *population-size,* changing the *mutation-rate* has minimal effect on the GA optimal objective value. It also indicates that choosing m = 0.4, with any level of *population-size,* generates the best GA solution in terms of quality. Another effect is that for a constant m, when p increases up to level 4, quality does not change and after level 4, quality degrades. From these observations, it can be concluded that the best combination for RV_1 is a small *population-size* (*N/6*, *N/3*, *N/2*) with any *mutation-rate* of 0.3 or higher ($m \geq 0.3$).

e. A solid pattern can not be observed in Figure 4.4 (B'_L). In general, it can be concluded that by increasing the level of m while holding the level of p constant, after exceeding the second level of m (0.4), the GA search time decreases. Also it can be deduced that by increasing the level of p, while holding the level of m constant, in general the GA search time increases. Hence, the best combination for the RV_2 is choosing a small *population-size* (*N/12*, *N/6*, *N/3*) and a large *mutation-rate* ($m \geq 0.5$).

f. From part (d), it was concluded the best combination for RV_1 is choosing a small *population-size* (*N/6*, *N/3*, *N/2*) with any *mutation-rate* ($m \geq 0.3$) and from part (e) it was deduced the best combination for RV_2 is choosing a small *population-size* (*N/12*, *N/6*, *N/3*) and a large *mutation-rate* ($m \geq 0.5$). Therefore, any p with the value of (*N/6* and *N/3*) and

any m with the value of (0.5, 0.7 and 1) generate the best overall solution in terms of both quality and time.

g. The conclusions that have been derived above, are also observed in the main effect plots: Figure 4.5 (C'_L) and Figure 4.6 (D'_L). Figure 4.5 (C'_L) supports the conclusion that *N/6*, *N/3* and *N/2* are the best levels of p for RV_1, and Figure 4.6 (D'_L) supports the outcome that the larger the population size, the larger the search time.

h. As a result of performing ANOVA for the large size problem, running GA with a small *population-size* (*N/6* or *N/3*) and a large *mutation-rate* (0.5 or 0.7 or 1) is recommended for any other large size problem in order to get the most optimal solution in the shortest possible time. However, for the large size problem, choosing a *mutation-rate* of 0.4 with any *population-size* generates the best GA solution in terms of quality.

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	6.87319E+17	4	1.71830E+17	3.14	0.021	Yes
Columns (*mutation-rate*)	6.13979E+19	5	1.22796E+19	224.32	0.000	Yes
Interaction	3.43503E+18	20	1.71752E+17	3.14	0.000	Yes
Error	3.28452E+18	60	5.47420E+16			
Total	6.88048E+19	89				

Table 4.5 - ANOVA for Large Problem- RV_1 (with $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	38265	4	9566.14	6.47	0.000	Yes
Columns (*mutation-rate*)	45564	5	9112.70	6.16	0.000	Yes
Interaction	40204	20	2010.22	1.36	0.180	No
Error	88735	60	1478.92			
Total	212768	89				

Table 4.6 - ANOVA for Large Problem- RV_2 (with $m = 0$)

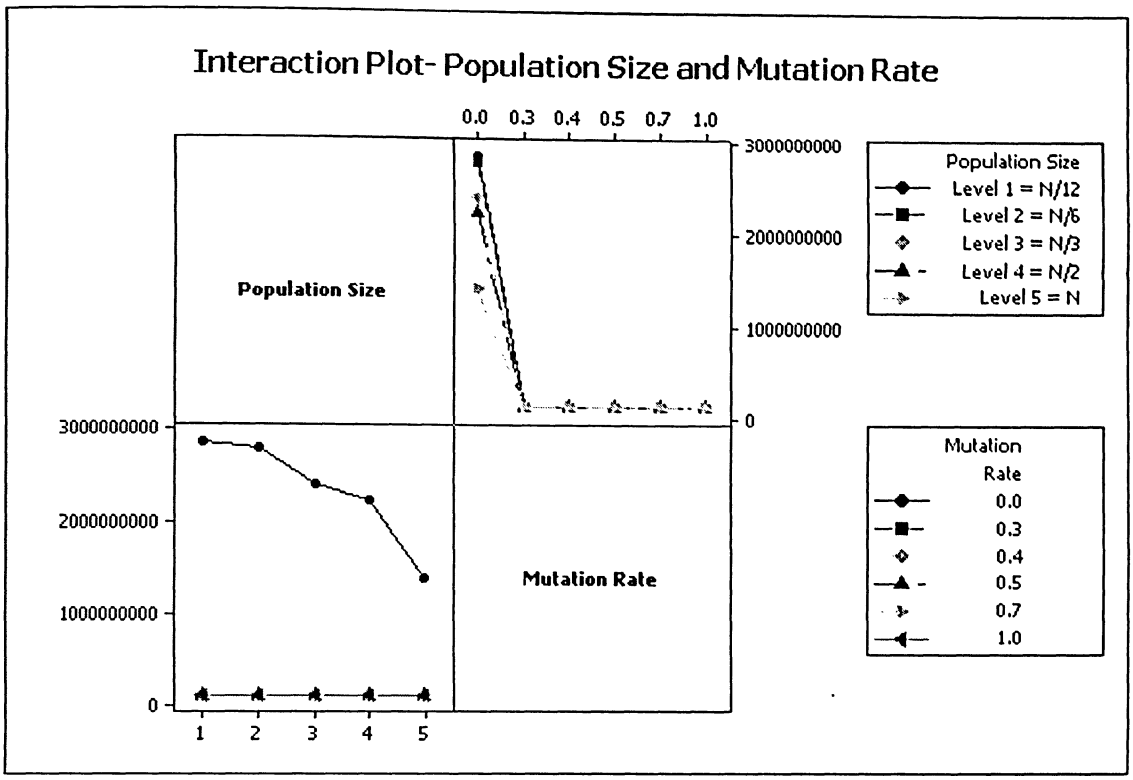

Figure 4.1 - (A_L) - RV_1 (with $m = 0$)

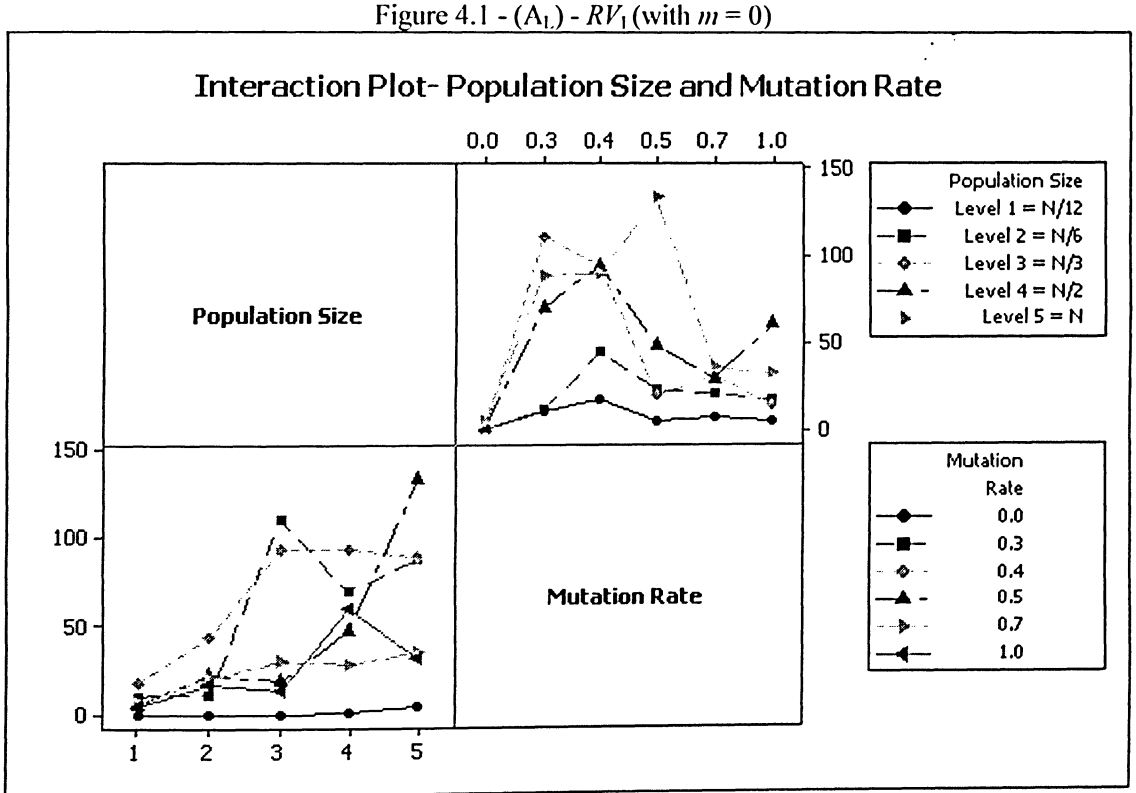

Figure 4.2 - (B_L) - RV_2 (with $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	2.69861E+12	4	6.74653E+11	1.82	0.140	No
Columns (*mutation-rate*)	1.24843E+12	4	3.12108E+11	0.84	0.506	No
Interaction	4.21571E+12	16	2.63482E+11	0.71	0.771	No
Error	1.85685E+13	50	3.71370E+11			
Total	2.67313E+13	74				

Table 4.7 - ANOVA for Large Problem - RV_1 (without $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	45025	4	11256.3	6.34	0.000	Yes
Columns (*mutation-rate*)	22493	4	5623.4	3.17	0.021	Yes
Interaction	33406	16	2087.9	1.18	0.318	No
Error	88723	50	1774.5			
Total	189647	74				

Table 4.8 - ANOVA for Large Problem - RV_2 (without $m = 0$)

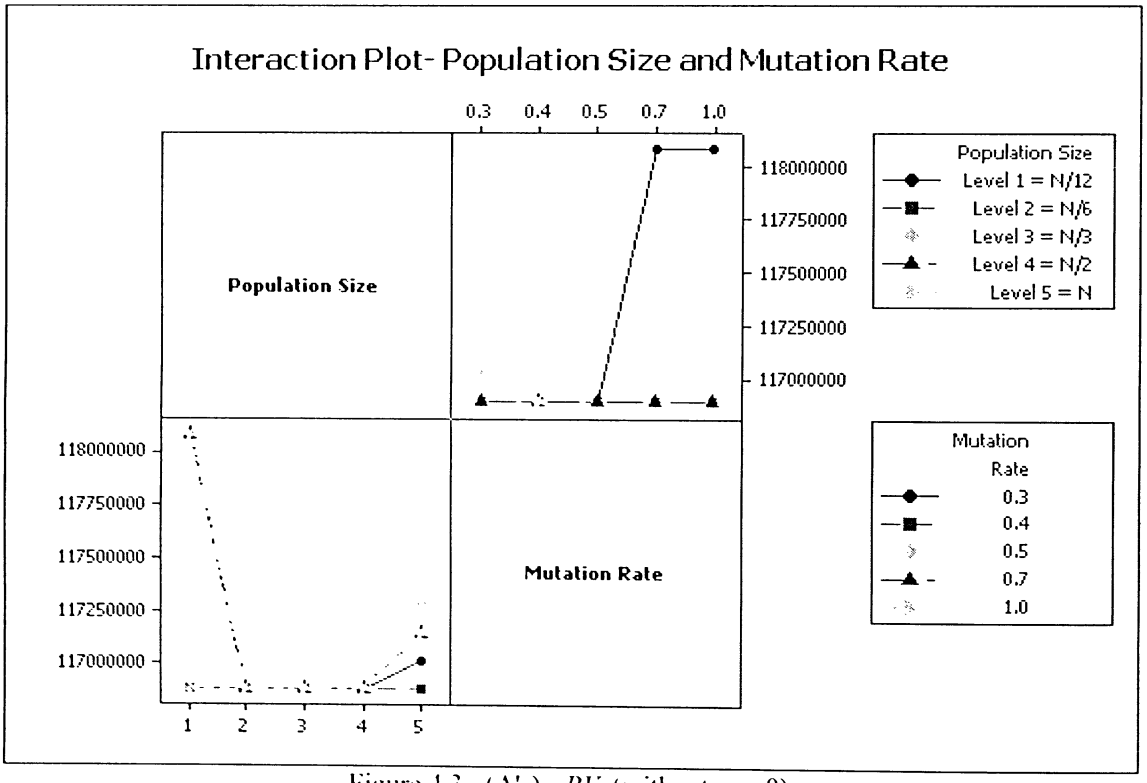

Figure 4.3 - (A'_1) - RV_1 (without $m = 0$)

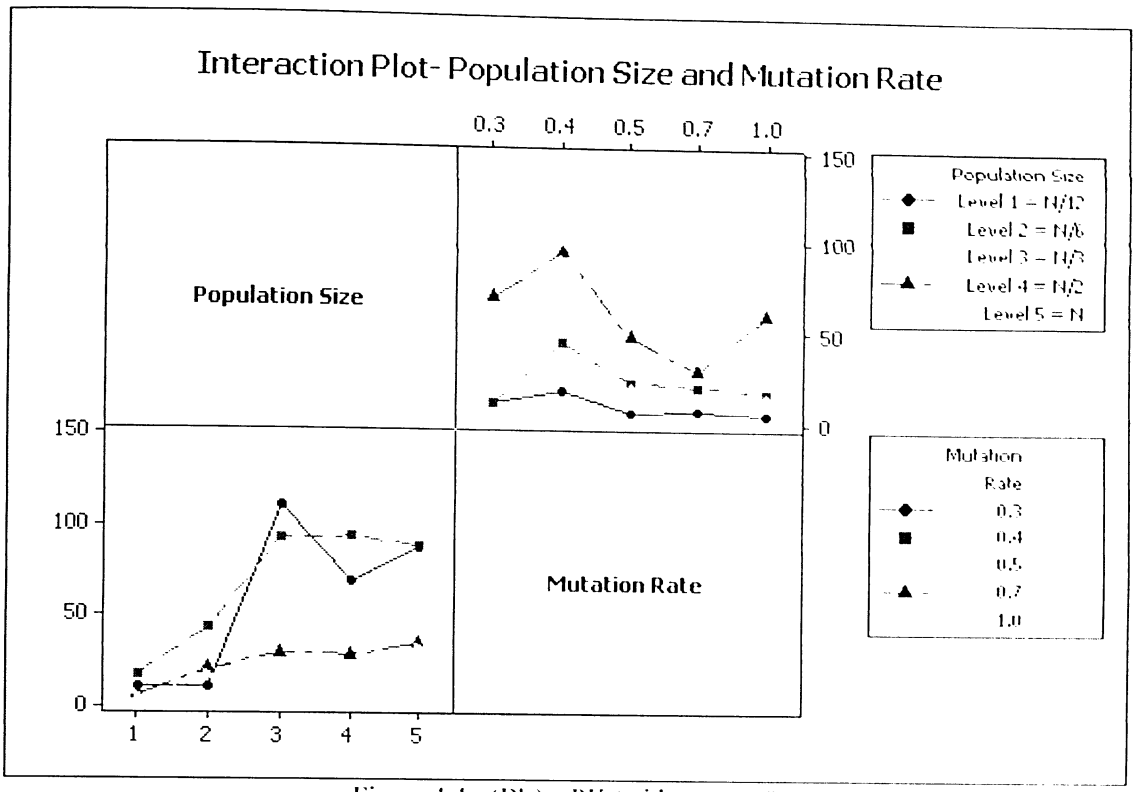

Figure 4.4 - (B'_1) - RV_3 (without m 0)

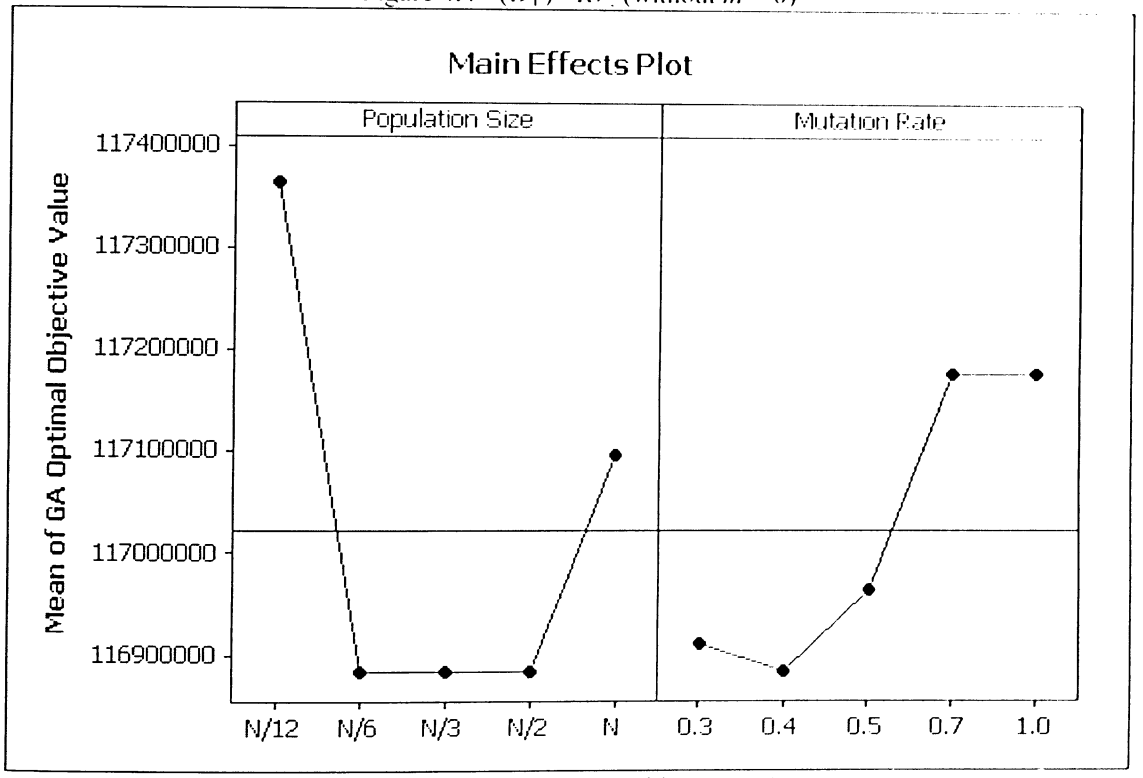

Figure 4.5 - (C''_1) - RV_1 (without m 0)

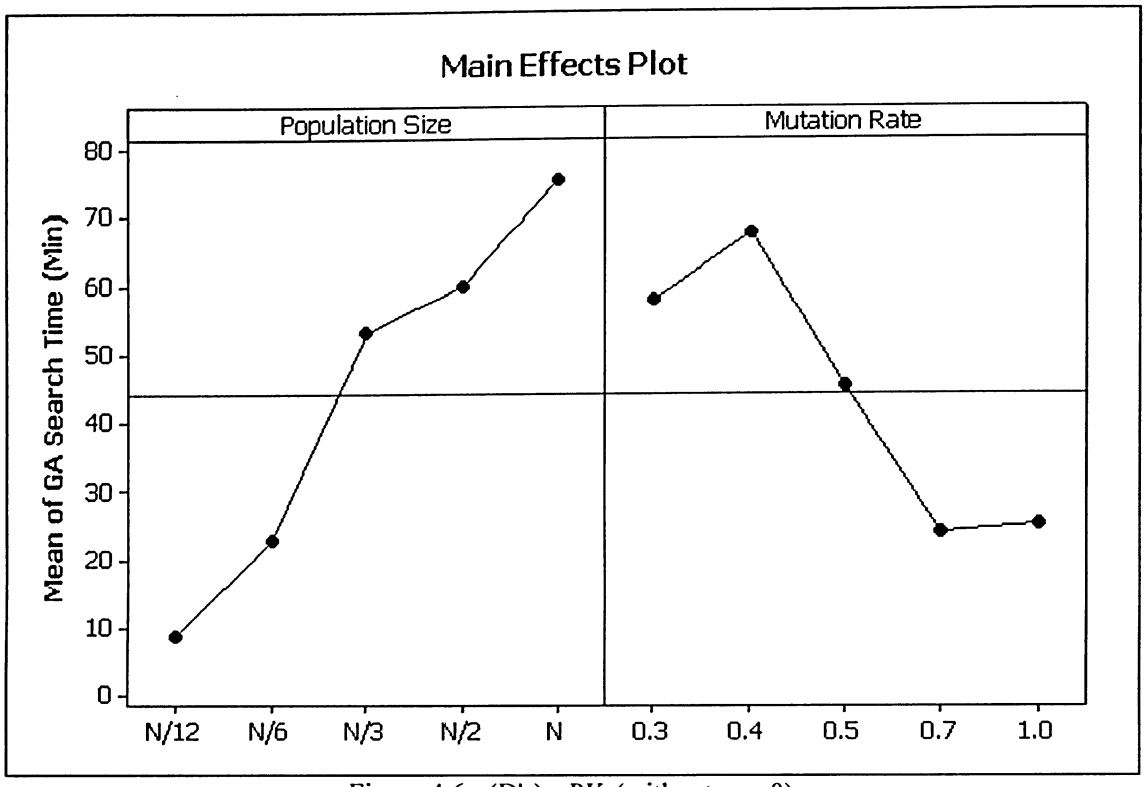

Figure 4.6 - (D'$_L$) - RV_2 (without $m = 0$)

4.3.2 Medium Size Problem

The optimal objective values and search times, obtained from 90 GA runs, for the medium problem are represented in tables 4.8 and 4.9 respectively. In this problem, five levels for factor I (p) and six levels for factor II (m), are implemented with three replications, i.e. $R = 5$, $C = 6$ and $n = 3$. The F_{calc} statistic is calculated according to section 4.1.3 and the results are depicted in tables 4.9 and 4.10. The following outcomes are attained by reviewing the ANOVA tables and interaction plots for this problem:

a. From Table 4.11, the small P-value for rows and columns (0.000) indicates that there is strong evidence for rejecting $H_0^{(1)}$ and $H_0^{(2)}$. But the P-value for interaction (0.907) is large (a value larger than 0.05) which does not provide strong evidence for rejecting $H_0^{(3)}$. So it is concluded that p and m factors have significant effect on RV_1, but the interaction between them does not.

	Replication #	m = 0	m = 0.3	m = 0.4	m = 0.5	m = 0.7	m = 1
$p = N/12 = 11$	1	802,017,274	641,174,120	779,624,177	728,429,038	657,970,526	606,777,870
	2	794,412,189	769,216,570	769,616,735	757,998,670	581,967,569	580,772,590
	3	786,000,038	604,368,070	608,369,472	573,166,299	730,000,866	730,808,979
cell mean		794,143,167	671,586,253	719,203,461	686,531,336	656,646,320	639,453,146
$p = N/6 = 22$	1	789,188,613	622,373,455	741,558,225	608,776,773	608,777,920	614,376,071
	2	774,800,699	573,165,993	583,973,754	573,965,784	573,966,738	595,595,075
	3	784,020,814	723,179,685	573,167,718	782,768,824	782,768,824	573,566,361
cell mean		782,670,042	639,573,044	632,899,899	655,170,460	655,171,161	594,512,502
$p = N/3 = 43$	1	784,790,921	574,767,957	576,367,052	576,367,540	577,569,555	573,967,595
	2	780,800,728	574,365,635	573,567,156	573,566,668	573,565,527	573,166,457
	3	775,574,729	577,568,179	573,965,034	573,967,268	573,565,479	573,566,344
cell mean		780,388,793	575,567,257	574,633,081	574,633,825	574,900,187	573,566,799
$p = N/2 = 65$	1	747,583,167	573,566,495	574,366,360	574,765,802	576,373,271	574,368,063
	2	772,386,757	573,165,589	573,565,779	573,565,449	573,966,140	573,964,744
	3	753,572,536	573,165,880	574,365,272	573,565,906	573,566,819	575,165,953
cell mean		757,847,487	573,299,321	574,099,137	573,965,719	574,635,410	574,499,587
$p = N = 131$	1	666,007,046	576,765,796	575,166,406	574,767,262	574,765,462	573,966,007
	2	711,194,940	573,965,705	574,367,116	573,565,440	573,166,283	573,164,990
	3	661,567,907	573,967,005	573,165,487	573,165,653	573,565,860	574,366,849
cell mean		679,589,964	574,899,502	574,233,003	573,832,785	573,832,535	573,832,615

Table 4.9 - GA optimal objective values for medium problem

b. Observing Table 4.12, it can be seen that the P-value for rows (0.000) and columns is small (0.053), so there is strong evidence for rejecting $H_0^{(4)}$ and $H_0^{(5)}$. But the P-value for interaction (0.322) is large (a value larger than 0.1) which indicates that there is no strong evidence for rejecting the $H_0^{(6)}$ hypothesis. These observations mean that the p and m factors have significant effect on the GA search time.

	Replication #	m = 0	m = 0.3	m = 0.4	m = 0.5	m = 0.7	m = 1
$p = N/12 = 11$	1	0.00252	1.81978	0.03046	0.55817	0.22238	1.62033
	2	0.00239	0.03675	0.03413	0.04194	1.49443	1.46703
	3	0.00862	0.94453	1.53164	7.13960	0.06186	0.06164
cell mean		0.00451	0.93369	0.53208	2.57990	0.59289	1.04967
$p = N/6 = 22$	1	0.00865	2.61417	0.08650	2.48333	16.96350	0.72450
	2	0.01390	14.06818	3.26094	7.95488	9.11068	1.30294
	3	0.00643	0.14814	33.06205	0.02567	0.02349	9.24816
cell mean		0.00966	5.61016	12.13650	3.48796	8.69923	3.75853
$p = N/3 = 43$	1	0.03983	6.73750	5.43317	4.69583	1.92583	4.10200
	2	0.02926	23.71185	6.74698	29.03971	33.36266	48.88990
	3	0.05737	4.33147	9.41175	3.13898	33.82589	29.23093
cell mean		0.04215	11.59361	7.19730	12.29151	23.03813	27.40761
$p = N/2 = 65$	1	0.12060	39.65433	11.09050	11.28967	2.26667	3.99100
	2	0.04492	32.25750	12.95563	9.54173	4.07699	8.43543
	3	0.13064	11.40193	7.37604	7.75174	6.16391	17.66372
cell mean		0.09872	27.77125	10.47406	9.52771	4.16919	10.03005
$p = N = 131$	1	0.82350	22.03700	16.23717	7.93917	11.07767	9.93817
	2	0.37063	45.28748	100.05815	21.50243	115.47392	38.94908
	3	0.73289	18.28580	38.96001	15.41704	17.97956	7.70458
cell mean		0.64234	28.53676	51.75178	14.95288	48.17705	18.86394

Table 4.10 - GA search time (min) for medium problem

c. From Figure 4.7 (A_M), the same discussion regarding ($m < 0.3$) that was addressed in part (c) of the large problem also holds true here. The worst level of m ($m = 0$) is omitted and ANOVA is repeated for the remaining data. The results of ANOVA following this modification are represented in tables 4.13 and 4.14, and the interaction graphs are represented by Figure 4.9 (A'_M) and Figure 4.10 (B'_M). By omitting the first level of m (m =0), only the conclusion for columns that was reached in part (a) is no longer valid, which indicates that p has significant effect on RV_1 but m and interaction do not. From Table 4.12, the P-value for columns (0.797) does not provide strong evidence for rejecting $H_0^{(2)}$. The conclusion from this discussion is that only the effect of p is significant on RV_1. However, by examining Table 4.14 it is clear that the results obtained in part (b): "only the p factor has the significant effect on RV_2" remains valid.

d. The presence of parallelism between most of the lines in Figure 4.9 (A'_M), supports the conclusion obtained in part (c) that the effect of interaction between factors on RV_1 is not significant. Figure 4.9 (A'_M) demonstrates that for all the levels of m ($m \geq 0.3$), by increasing the level of p while holding the level of m constant, the quality of GA solution improves up to level 3 of m ($m = 0.5$), and after passing this level, no further improvement is observed. It also shows for levels 3, 4 and 5 of p, increasing the level of m while holding the level of p constant, does not change the quality of RV_1. Hence the best combination for RV_1 is choosing a large *population-size* (*N/3*, *N/2* and *N*) with any *mutation-rate* ($m \geq 0.3$). This conclusion supports the outcome of part (c): "only the effect of p is significant on RV_1".

e. A solid pattern can not be observed in Figure 4.10 (B'_M). But in general it can be concluded that choosing a small p can reduce the GA search time. Also it can be noticed that for level 1 to level 4 of p, with all of the levels of m the GA search time does not vary significantly and remains in the range of 0 to 30 minutes.

f. The conclusions that have been obtained above are also observed in Figure 4.11 (C'_M) and Figure 4.12 (D'_M) main effect plots. Figure 4.11 (C'_M) supports the conclusion from part (c) that *N/3*, *N/2* and *N* are the best levels of *p* for the GA search time.

g. From part (c), it was concluded that choosing a large *population-size* (*N/3*, *N/2* and *N*), with any *mutation-rate* ($m \geq 0.3$), generates the best GA optimal objective value. By combining this result and the results from part (e), and observing main effect plots in Figures 4.11 and 4.12, it is deduced that choosing a small *p* (*N/3*) and a large *m* (1) can generate the best overall solution in terms of both quality and time. A good solution can also be produced with mutation rates m=0.5 and m=0.7.

h. As a result of performing ANOVA for the medium size problem, running GA with a small *population-size* (*N/3*) and a large mutation rate (0.5 or 0.7 or 1) is recommended for any other medium size problem in order to get the most optimal solution in the shortest possible time.

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	1.36820E+17	4	3.42051E+16	12.87	0.000	Yes
Columns (*mutation-rate*)	2.95239E+17	5	5.90477E+16	22.21	0.000	Yes
Interaction	3.11750E+16	20	1.55875E+15	0.59	0.907	No
Error	1.59501E+17	60	2.65835E+15			
Total	6.22735E+17	89				

Table 4.11- ANOVA for Medium Problem - RV_1 (with $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	7140.7	4	1785.18	6.86	0.000	Yes
Columns (*mutation-rate*)	3044.1	5	608.81	2.34	0.053	Yes
Interaction	6023.6	20	301.18	1.16	0.322	No
Error	15621.5	60	260.36			
Total	31829.8	89				

Table 4.12 - ANOVA for Medium Problem - RV_2 (with $m = 0$)

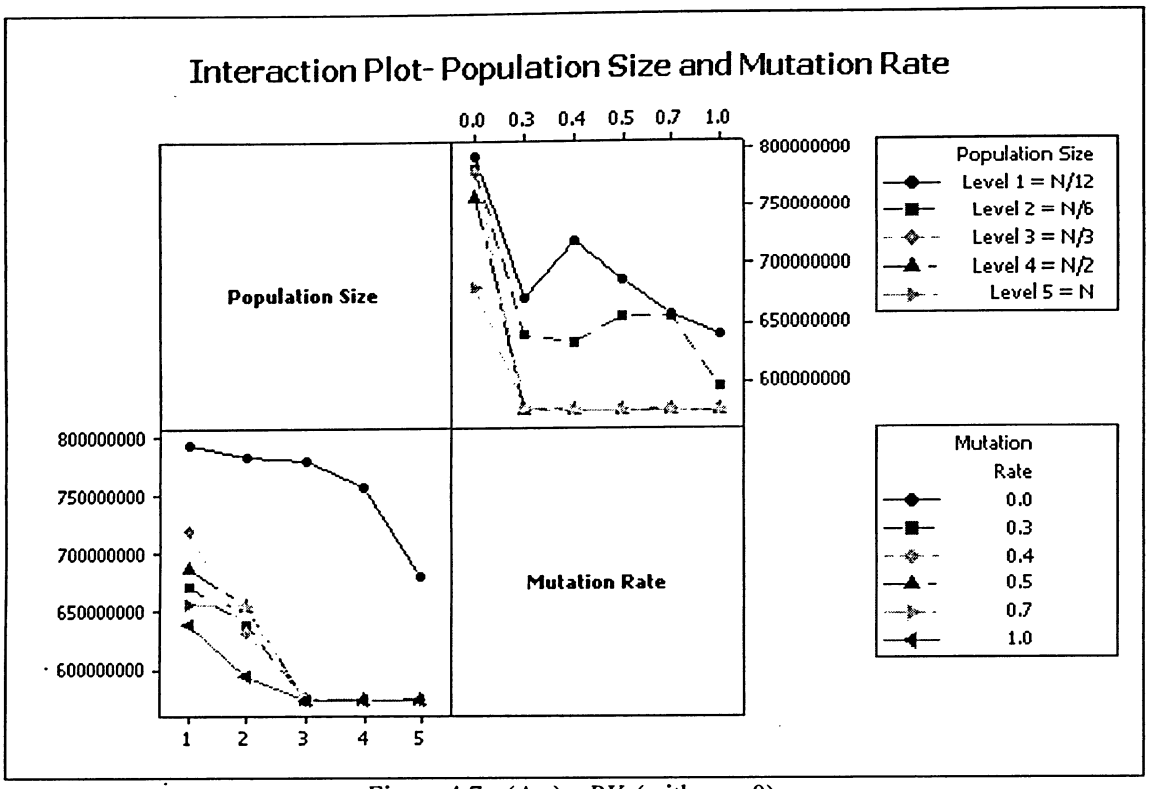

Figure 4.7 - (A_M) - RV_1 (with $m = 0$)

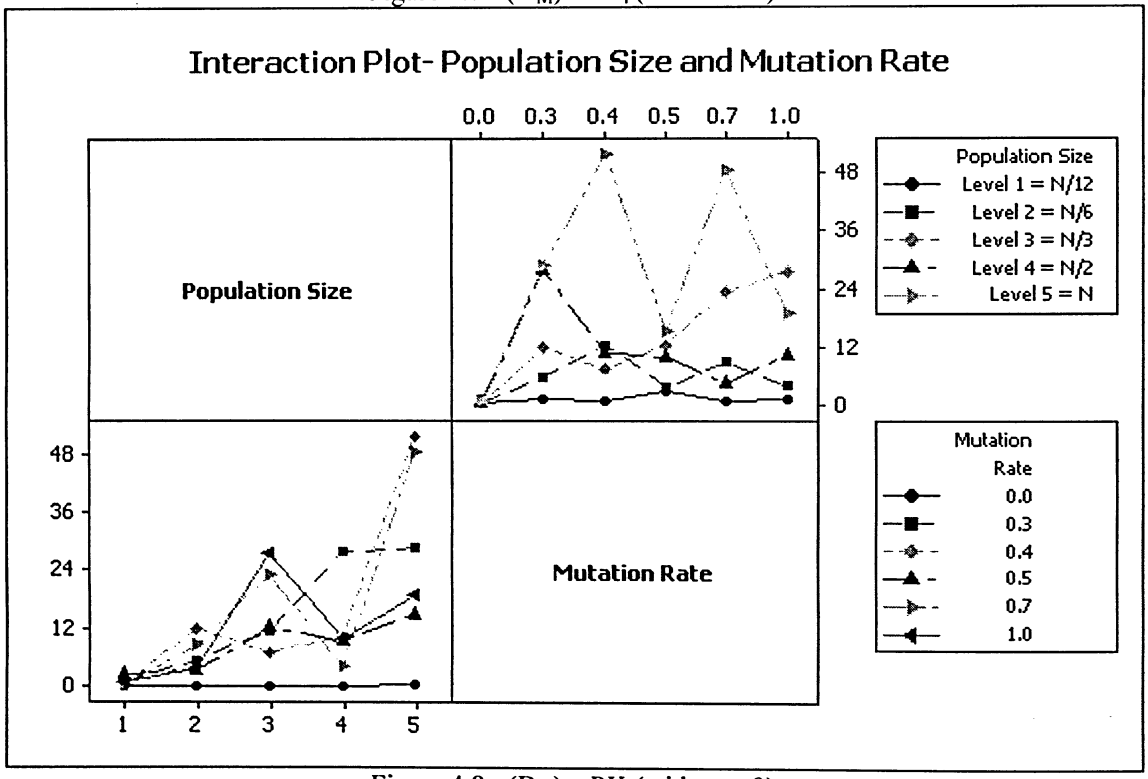

Figure 4.8 - (B_M) - RV_2 (with $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	1.28994E+17	4	3.22486E+16	10.25	0.000	Yes
Columns (*mutation-rate*)	5.21846E+15	4	1.30462E+15	0.41	0.797	No
Interaction	1.33208E+16	16	8.32551E+14	0.26	0.997	No
Error	1.57380E+17	50	3.14761E+15			
Total	3.04914E+17	74				

Table 4.13 - ANOVA for Medium Problem - RV_1 (without $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	8497.8	4	2124.44	6.80	0.000	Yes
Columns (*mutation-rate*)	716.0	4	179.01	0.57	0.683	No
Interaction	4665.6	16	291.60	0.93	0.539	No
Error	15621.3	50	312.43			
Total	29500.8	74				

Table 4.14 - ANOVA for Medium Problem – RV_2 (without $m = 0$)

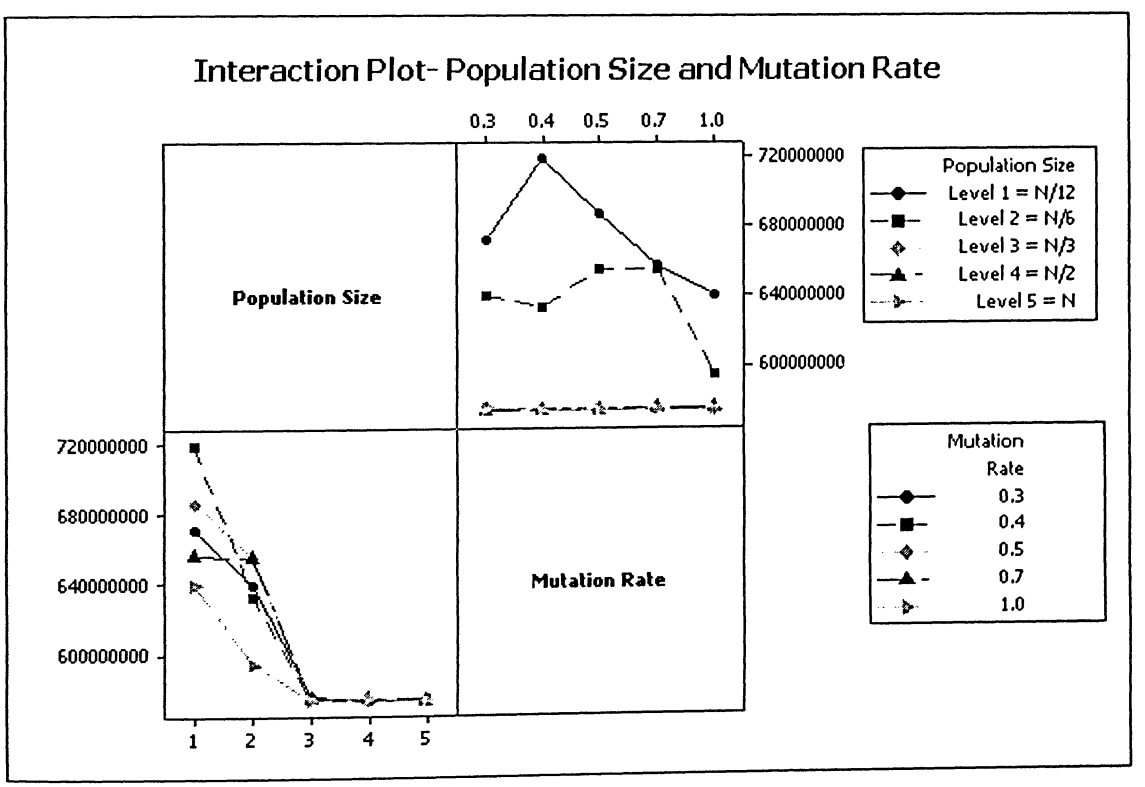

Figure 4.9 - (A'_M) - RV_1 (without $m = 0$)

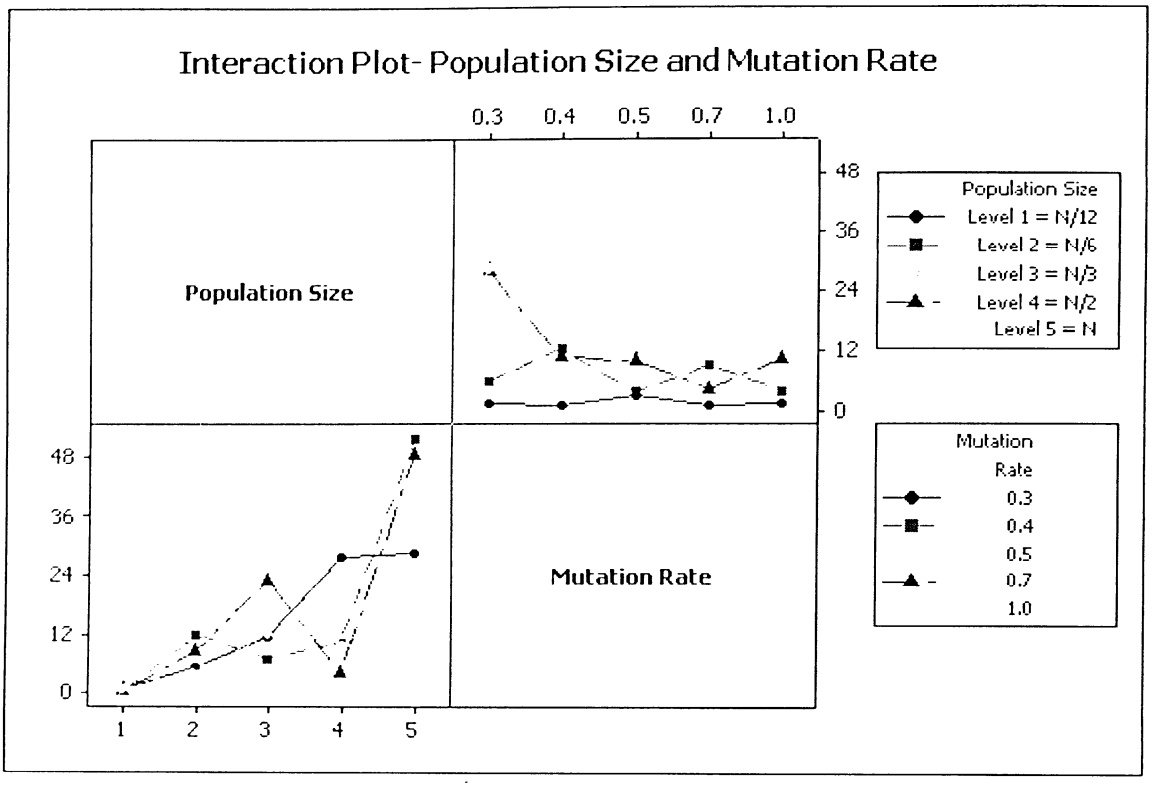

Figure 4.10 - (B'_M) - RV_2 (without $m = 0$)

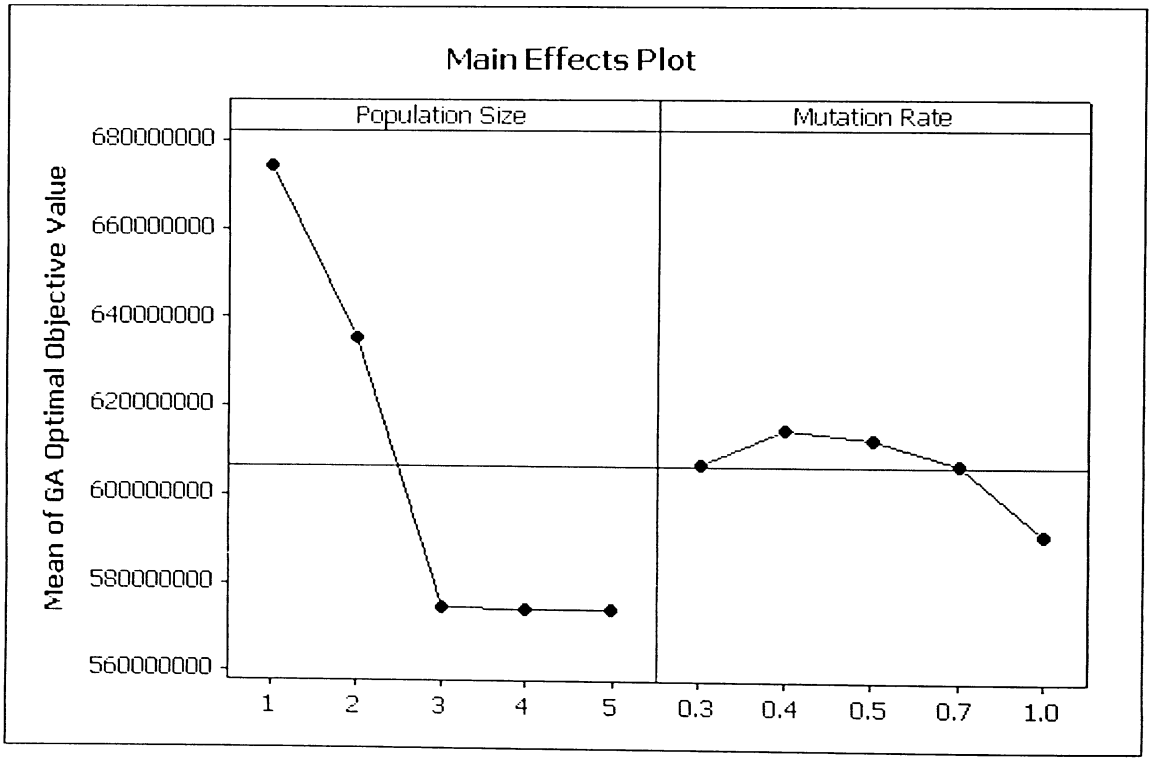

Figure 4.11 - (C'_M) - RV_1 (without $m = 0$)

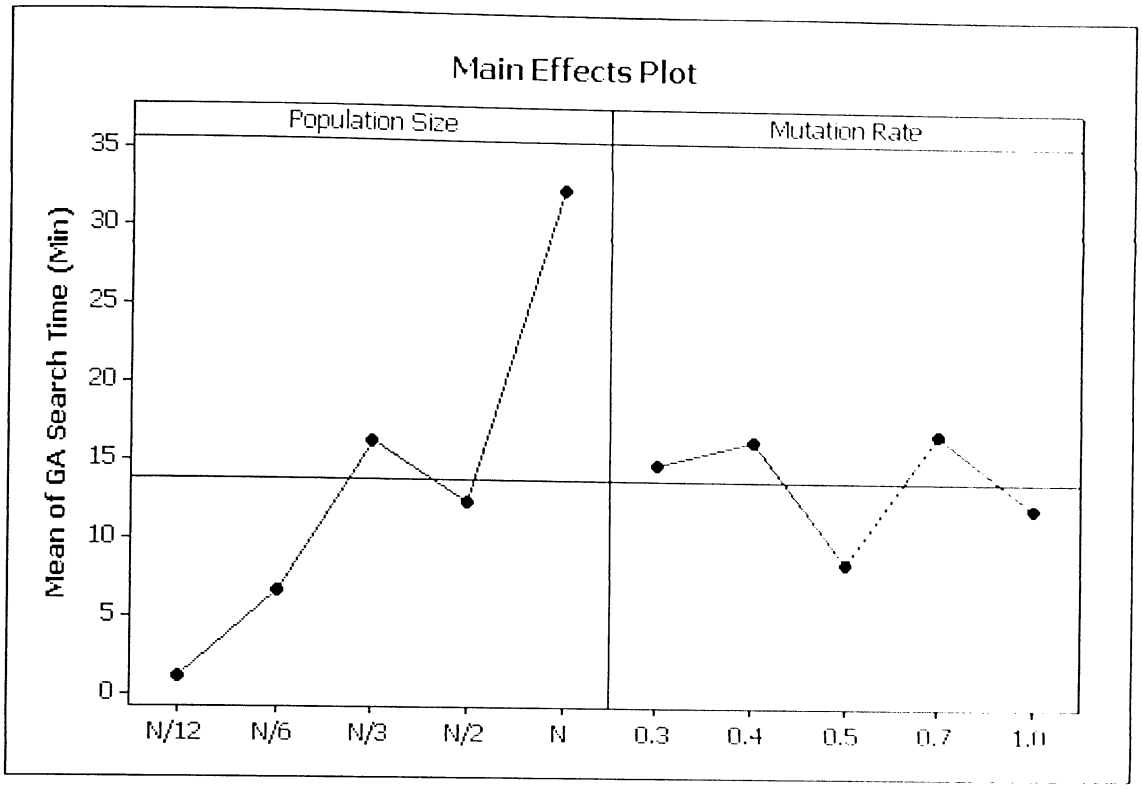

Figure 4.12 - (D'_M) - RV_3 (without m = 0)

4.3.3 Small Size Problem

The optimal objective values and search times obtained from 144 GA runs for the small problem are represented in tables 4.15 and 4.16. For this problem, eight levels of factor I (*population-size*) and six levels of factor II (*mutation-rate*) are implemented with three replications, i.e. R = 8, C = 6 and n =3. For the large and medium problems, only five levels for population size were used, but eight levels were required for the small problem before a pattern of consistency was exhibited. The F_{calc} statistic is calculated according to section 4.1.3 and the results are depicted in tables 4.17 and 4.18.

The following outcomes are achieved by reviewing the ANOVA tables and interaction plots for this problem:

a. Examining Table 4.17, it can be seen that the P-value for columns (0.000), for rows (0.104) and interaction (0.000) is small which indicates that there is strong evidence for

rejecting $H_0^{(1)}$, $H_0^{(2)}$ and $H_0^{(3)}$. Hence, the m and p factors and the interaction of them have a significant effect on GA optimal objective value.

	Replication #	m = 0	m = 0.3	m = 0.4	m = 0.5	m = 0.7	m = 1
$p = N/12 = 2$	1	110,327,548	97,131,443	96,731,420	96,331,202	97,531,027	96,331,523
	2	121,127,039	96,731,499	96,331,810	97,531,284	95,931,347	99,131,364
	3	117,127,013	95,931,379	95,931,787	102,729,662	99,130,643	95,931,383
cell mean		116,193,867	96,598,107	96,331,672	98,864,049	97,531,006	97,131,423
$p = N/6 = 3$	1	104,329,016	95,931,663	97,131,373	96,731,427	97,531,107	97,931,450
	2	121,126,419	96,330,759	97,131,661	95,931,876	95,931,789	96,731,368
	3	110,727,740	95,931,551	95,931,572	97,131,195	96,731,473	95,931,610
cell mean		112,061,058	96,064,658	96,731,535	96,598,166	96,731,456	96,864,809
$p = N/3 = 6$	1	101,529,776	96,331,547	96,331,501	96,731,506	96,331,493	96,331,215
	2	107,128,850	95,931,293	95,931,361	95,931,300	95,931,611	95,931,283
	3	111,929,249	96,731,511	98,731,057	100,330,130	96,331,318	95,931,271
cell mean		106,862,625	96,331,450	96,997,973	97,664,312	96,198,141	96,064,590
$p = N/2 = 9$	1	100,730,128	101,129,964	96,331,566	97,131,334	97,931,089	96,731,142
	2	106,328,627	95,931,224	96,331,395	99,130,987	98,730,839	99,130,335
	3	107,128,531	96,331,185	96,331,572	96,331,540	96,331,262	96,331,809
cell mean		104,729,095	97,797,458	96,331,511	97,531,287	97,664,397	97,397,762
$p = N = 17$	1	96,331,683	96,730,815	96,731,080	97,531,314	97,131,302	96,331,343
	2	98,731,140	99,530,664	99,530,216	98,730,362	98,730,993	98,730,357
	3	103,530,257	99,130,547	99,130,926	98,730,680	98,730,988	98,731,045
cell mean		99,531,027	98,464,009	98,464,074	98,330,785	98,197,761	97,930,915
$p = 2N = 34$	1	100,329,978	99,130,592	96,731,115	99,130,663	99,130,351	99,530,563
	2	98,730,916	99,529,971	98,730,675	98,730,357	99,130,658	99,130,335
	3	98,730,908	99,130,300	99,129,987	99,130,623	99,130,300	98,730,357
cell mean		99,263,934	99,263,621	98,197,259	98,997,214	99,130,436	99,130,418
$p = 7N = 119$	1	99,130,312	99,130,620	99,130,592	99,130,297	99,130,581	99,130,592
	2	98,730,473	98,730,357	98,730,670	98,730,357	98,730,993	98,730,357
	3	99,130,101	99,129,987	99,130,263	99,130,503	99,129,987	99,129,987
cell mean		98,996,962	98,996,988	98,997,175	98,997,052	98,997,187	98,996,979
$p = 12N = 204$	1	99,130,307	99,130,592	99,130,269	99,130,269	99,130,269	99,130,269
	2	98,730,680	98,730,357	98,730,357	98,730,357	98,730,680	98,730,357
	3	99,130,351	99,129,987	99,129,987	99,129,987	99,129,987	99,129,987
cell mean		98,997,113	98,996,979	98,996,871	98,996,871	98,996,979	98,996,871

Table 4.15 - GA optimal objective values for small problem

b. From Table 4.18, the small P-value for rows and columns (0.000) indicates that there is strong evidence for rejecting $H_0^{(4)}$ and $H_0^{(5)}$. But the P-value for interaction (0.818) is large which does not provide strong evidence for rejecting $H_0^{(6)}$. So it is concluded that both the p and the m factors have significant effect on RV_2, but the interaction of them does not.

c. From Figure 4.13 (A_S), the same discussion regarding ($m < 0.3$) that was addressed in part (c) of the large problem also holds true here. The worst level of m ($m = 0$) is omitted, and ANOVA is repeated for the remaining data. The results of ANOVA following this

modification are represented in tables 4.19 and 4.20 and the interaction graphs are represented by Figure 4.15 (A'_S) and Figure 4.16 (B'_S). By omitting the first level of m (m =0), the conclusions that were reached in part (a) and part (b) are no longer valid. From Table 4.19, the small P-value for rows (0.000) and large P-values for columns (0.442) and for interaction (0.771) imply that there is strong evidence for rejecting $H_0^{(1)}$, but there is no strong evidence for rejecting $H_0^{(2)}$ and $H_0^{(3)}$. Therefore only the effect of p factor on RV_1 is significant. From Table 4.20, the small P-value for rows (0.000), large P-value for columns (0.570) and interaction (0.805) imply that there is only strong evidence for rejecting $H_0^{(4)}$, meaning that only the effect of p factor on RV_2 is significant.

	Replication #	m = 0	m = 0.3	m = 0.4	m = 0.5	m = 0.7	m = 1
$p = N/12 = 2$	1	0.00014	0.02993	0.03334	0.03296	0.01369	0.01111
	2	0.00012	0.01691	0.02557	0.01253	0.01743	0.00573
	3	0.00042	0.06015	0.02104	0.00641	0.01007	0.04019
cell mean		0.00022	0.03566	0.02665	0.01730	0.01373	0.01901
$p = N/6 = 3$	1	0.00060	0.02203	0.00998	0.01367	0.00733	0.01055
	2	0.00021	0.03821	0.01432	0.01633	0.03542	0.00468
	3	0.00053	0.02218	0.01200	0.01390	0.01868	0.01991
cell mean		0.00044	0.02747	0.01210	0.01463	0.02048	0.01171
$p = N/3 = 6$	1	0.00020	0.02375	0.01552	0.01065	0.00845	0.02622
	2	0.00162	0.01118	0.02274	0.02270	0.01454	0.00995
	3	0.00292	0.01469	0.00545	0.00516	0.02586	0.01890
cell mean		0.00158	0.01654	0.01457	0.01284	0.01628	0.01836
$p = N/2 = 9$	1	0.00188	0.01725	0.01288	0.01341	0.00721	0.01277
	2	0.00117	0.00977	0.01353	0.01421	0.01097	0.01359
	3	0.00112	0.01198	0.04598	0.03249	0.02716	0.00832
cell mean		0.00139	0.01300	0.02413	0.02004	0.01511	0.01156
$p = N = 17$	1	0.00781	0.01567	0.00740	0.00740	0.01938	0.01915
	2	0.00418	0.00816	0.06205	0.02521	0.02155	0.02162
	3	0.00550	0.03016	0.01592	0.03948	0.01946	0.01723
cell mean		0.00583	0.01799	0.02846	0.02403	0.02013	0.01933
$p = 2N = 34$	1	0.01306	0.01614	0.01103	0.03632	0.02998	0.02837
	2	0.00970	0.02506	0.01772	0.01768	0.02306	0.02500
	3	0.01136	0.04914	0.03199	0.04067	0.02022	0.03539
cell mean		0.01137	0.03011	0.02025	0.03156	0.02442	0.02959
$p = 7N = 119$	1	0.01750	0.05707	0.04033	0.03321	0.03447	0.03086
	2	0.02880	0.03505	0.03553	0.03645	0.04893	0.03320
	3	0.03684	0.04983	0.04257	0.04997	0.03185	0.03410
cell mean		0.02771	0.04732	0.03948	0.03988	0.03842	0.03272
$p = 12N = 204$	1	0.04192	0.05770	0.04795	0.05670	0.06987	0.06377
	2	0.05108	0.05262	0.05359	0.05081	0.04687	0.05912
	3	0.04137	0.04737	0.04302	0.06742	0.05310	0.05433
cell mean		0.04479	0.05256	0.04818	0.05831	0.05661	0.05907

Table 4.16 - GA search time (min) for small problem

d. The lack of parallelism between the lines in Figure 4.15 (A'_S) represents the existence of interaction between the *p* and *m* factors. For a constant level of *p*, when *m* increases a solid pattern is not observed, but it can be concluded that a large *p* (such as *N*, *2N*, *7N* and 12*N*) generates the worst GA solution in terms of quality. The straight line in Figure 4.15 (A'_S) for a large *population-size* (such as *2N*, *7N* and 12N), indicates that changing the *mutation-rate* does not affect the quality of the GA solution significantly, i.e. the same GA optimal objective value is produced by any *mutation-rate*. This only became obvious at the higher levels of *p*. It was unnecessary for the medium and large problems to use this many levels of population size, as a consistent result became evident at lower levels of *p*. For a constant level of mutation rate, when population size increases a solid pattern is not observed, but in general after passing level 4 of *p* (*N/2*) the quality of the GA solution deteriorates. After passing level 7 of *p* (*7N*), the quality of GA does not change, indicating that for large *population-sizes* ($p \geq 7N$) the value of *mutation-rate* does not affect RV_1. It is also observed that any level of *mutation-rate* with a small *population-size* (*N/6*, *N*/3, *N/2*) generates a good GA solution in terms of quality, this conclusion supports the result obtained from part (c) that only *p* has the significant effect on RV_1. A very small *p* (*N/12*) is not recommended, because as it is clear from this plot that for any level of *m*, after passing this level of *p* (*N/12*), the RV_1 improves. As a conclusion from Figure 4.15 (A'_S), a small *p* (*N/6* and *N/3*) with a small *m* (0.3, 0.4) is the best combination for GA optimal objective value.

e. The lack of parallelism between the lines in Figure 4.16 (B'_S) represents the existence of interaction between the *p* and *m* factors. For a constant level of *p*, when *m* increases a solid pattern is not observed, but it can be concluded that a large *p* (such as *N*, *2N*, 7N and 12*N*) generates the worst GA solution in terms of search time. For a constant level of mutation rate, when population size increases a solid pattern is not observed, but in general for any level of *m*, after passing level 4 of *p* (*N/2*) the GA search time increases. It is also seen that a small population size (*N/12*, *N/6*, *N*/3, *N/2*) with any level of *mutation-rate* generates a faster solution, this conclusion supports the result obtained from part (c) that only *p* has the

significant effect on RV_2. As a conclusion from Figure 4.16 (B'_S), a small p (*N/12, N/6, N/3, N/2*) with any m is the best combination for GA search time.

f. Some of the conclusions reached in the above parts are also verified by the main effects plots in Figure 4.17 (C'_S) and Figure 4.18 (D'_S). These two plots support the outcome of part (c) that the effect of p on RV_1 and RV_2 is significant while the effect of m is not. Also, it is concluded that the effect of m on RV_1 is more than the effect of m on RV_2. From Figure 4.17 (C'_S), it is seen that when *population-size* increases the quality of solution gets worse, and after passing level 7 of p (*2N*) it does not change. It also shows a small p (*N/6, N/3*) with any *mutation-rate* produces a good GA solution in terms of quality and time; however, a small mutation rate like (0.3 and 0.4) slightly produces a better one.

g. From part (d), it was concluded the best combination for RV_1 is choosing a small p (*N/6, N/3*) with a small m (0.3, 0.4), and from part (e) it was deduced the best combination for the RV_2 is choosing a small p (*N/12, N/6, N/3, N/2*) with any level of m. Therefore, a small p (*N/6, N/3*) with a small m (0.3, 0.4) will generate the best overall solution in terms of both quality and time.

h. As a result of performing ANOVA for the small size problem, running GA with a small *population-size* (*N/6* or *N/3*) and a small mutation rate (0.3, 0.4) is recommended for any other small size problem in order to get the most optimal solution in the shortest possible time.

Source of Variability	***SSQ***	***df***	***MS***	F_{calc}	***P*- value**	**Significant?**
Rows (*population-size*)	5.34904E+13	7	7.64148E+12	1.76	0.104	Yes
Columns (*mutation-rate*)	9.00624E+14	5	1.80125E+14	41.54	0.000	Yes
Interaction	1.01695E+15	35	2.90557E+13	6.70	0.000	Yes
Error	4.16295E+14	96	4.33640E+12			
Total	2.38736E+15	143				

Table 4.17 - ANOVA for Small Problem - RV_1 (with $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	0.0249998	7	0.0035714	34.43	0.000	Yes
Columns (*mutation-rate*)	0.0050513	5	0.0010103	9.74	0.000	Yes
Interaction	0.0027642	35	0.0000790	0.76	0.818	No
Error	0.0099568	96	0.0001037			
Total	0.0427721	143				

Table 4.18 - ANOVA for Small Problem - RV_2 (with $m = 0$)

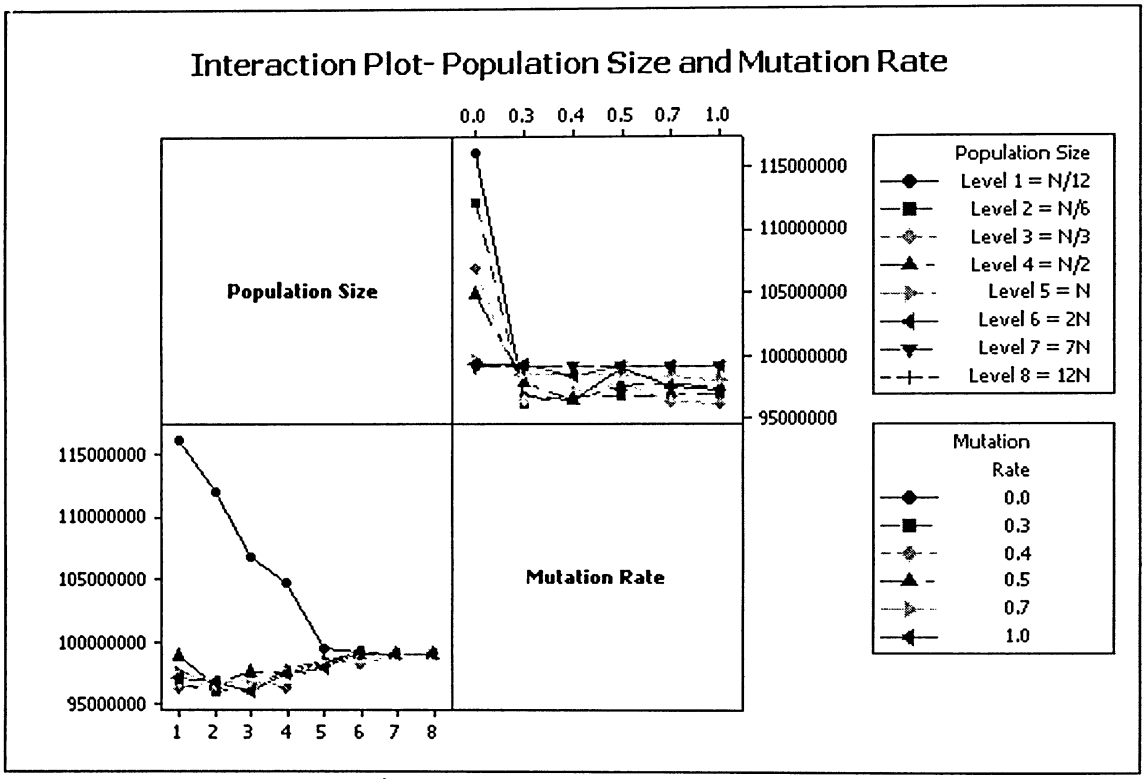

Figure 4.13 - (A_S) - RV_1 (with $m = 0$)

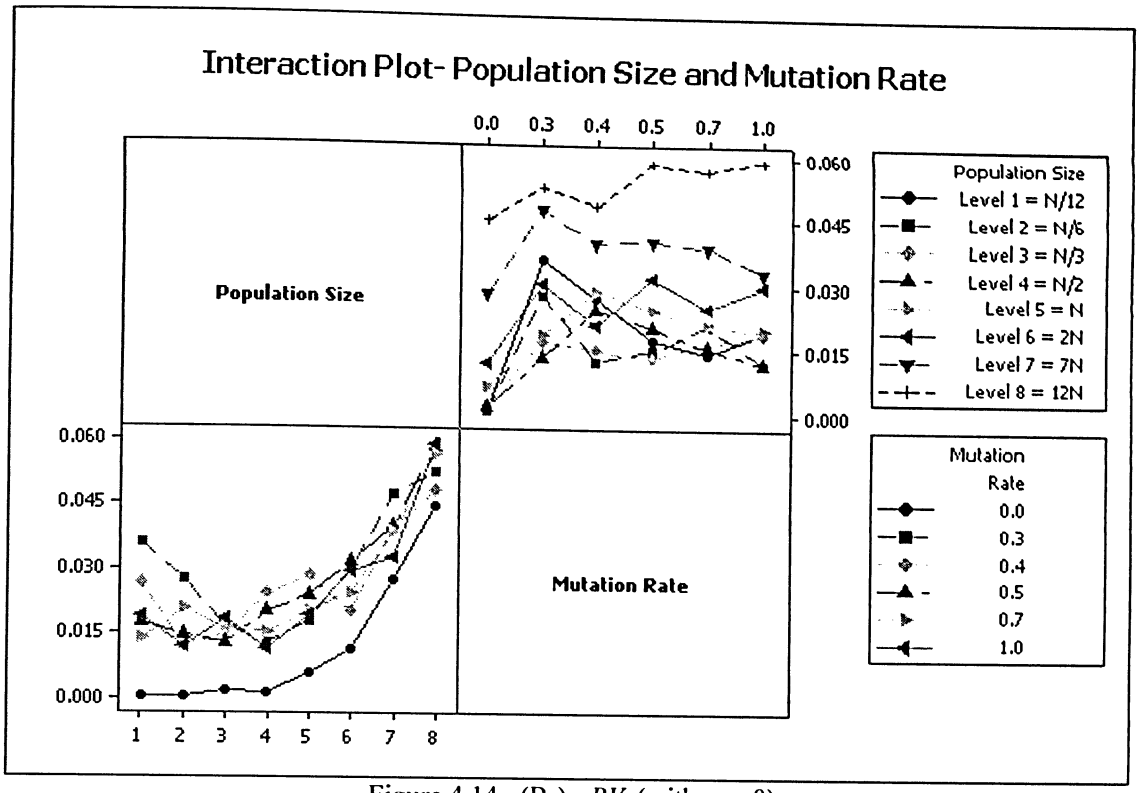

Figure 4.14 - (B_S) - RV_2 (with $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	1.13563E+14	7	1.62233E+13	12.28	0.000	Yes
Columns (*mutation-rate*)	4.99196E+12	4	1.24799E+12	0.94	0.442	No
Interaction	2.03578E+13	28	7.27064E+11	0.55	0.961	No
Error	1.05656E+14	80	1.32070E+12			
Total	2.44569E+14	119				

Table 4.19 - ANOVA for Small Problem - RV_1 (without $m = 0$)

Source of Variability	*SSQ*	*df*	*MS*	F_{calc}	*P*- value	Significant?
Rows (*population-size*)	0.0196700	7	0.0028100	23.20	0.000	Yes
Columns (*mutation-rate*)	0.0003566	4	0.0000891	0.74	0.570	No
Interaction	0.0025360	28	0.0000906	0.75	0.805	No
Error	0.0096917	80	0.0001211			
Total	0.0322543	119				

Table 4.20 - ANOVA for Small Problem – RV_2 (without $m = 0$)

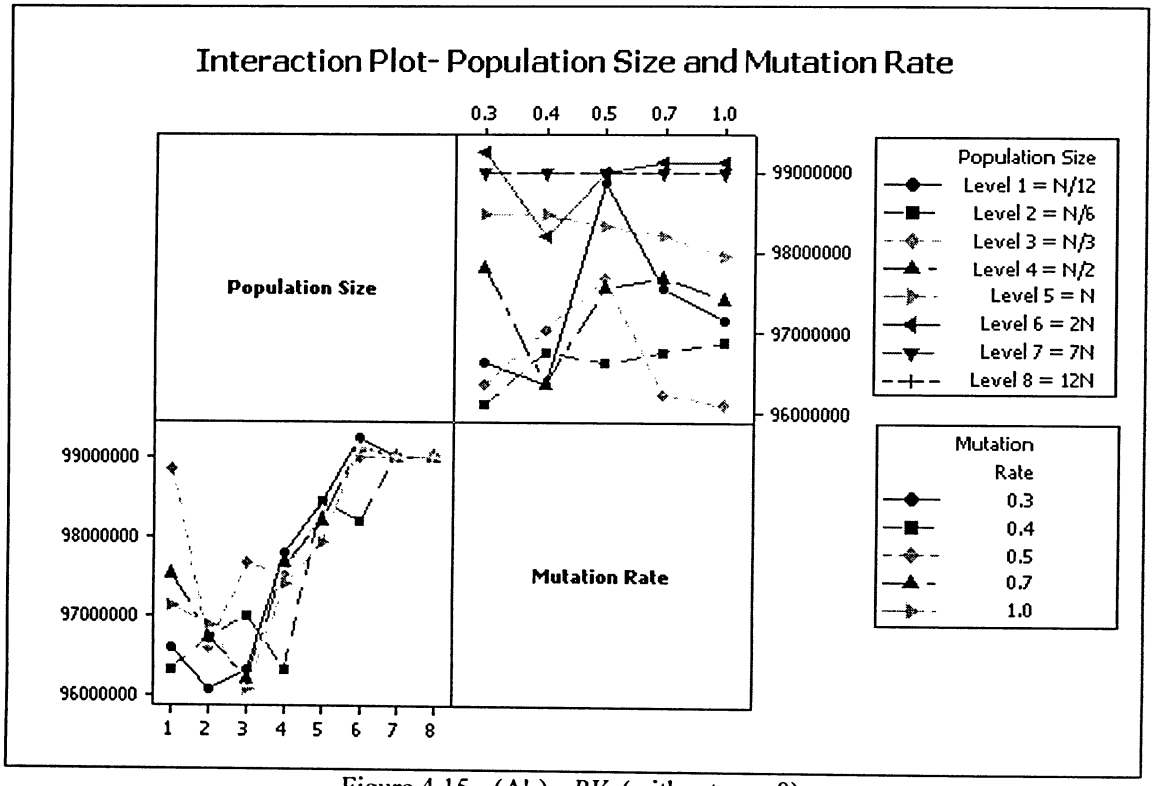

Figure 4.15 - (A's) - RV_1 (without $m = 0$)

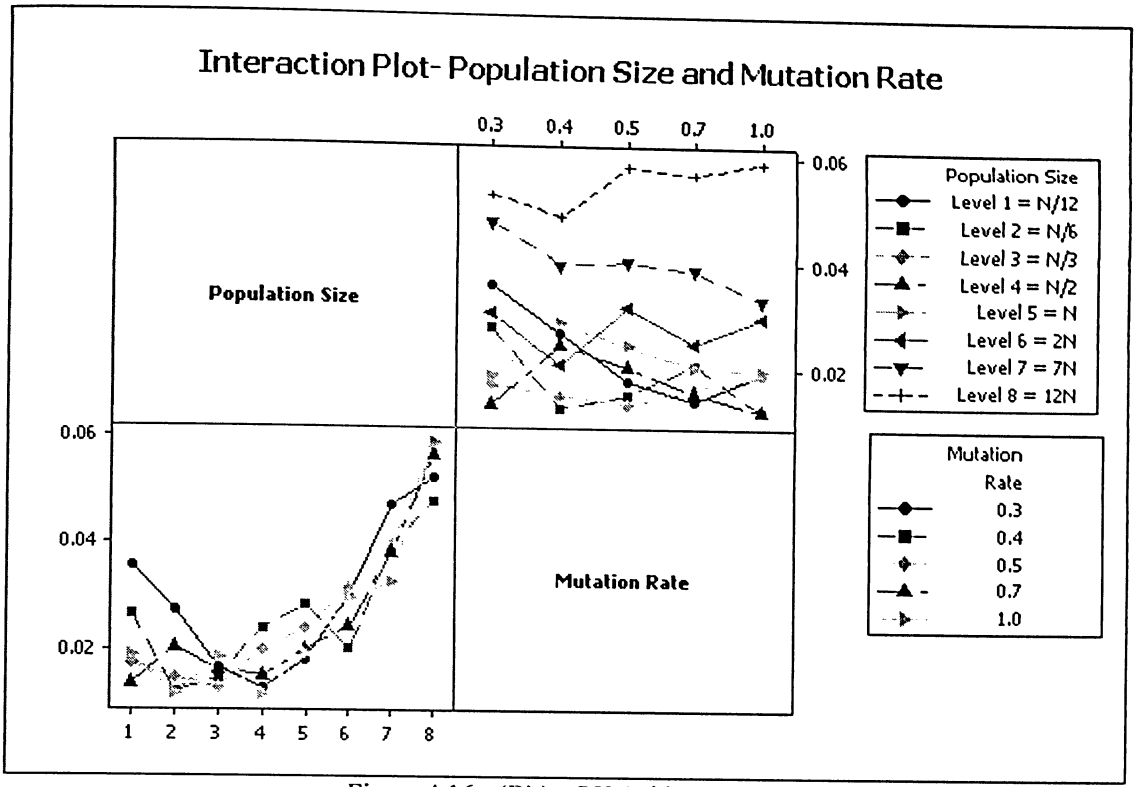

Figure 4.16 - (B's) - RV_2 (without $m = 0$)

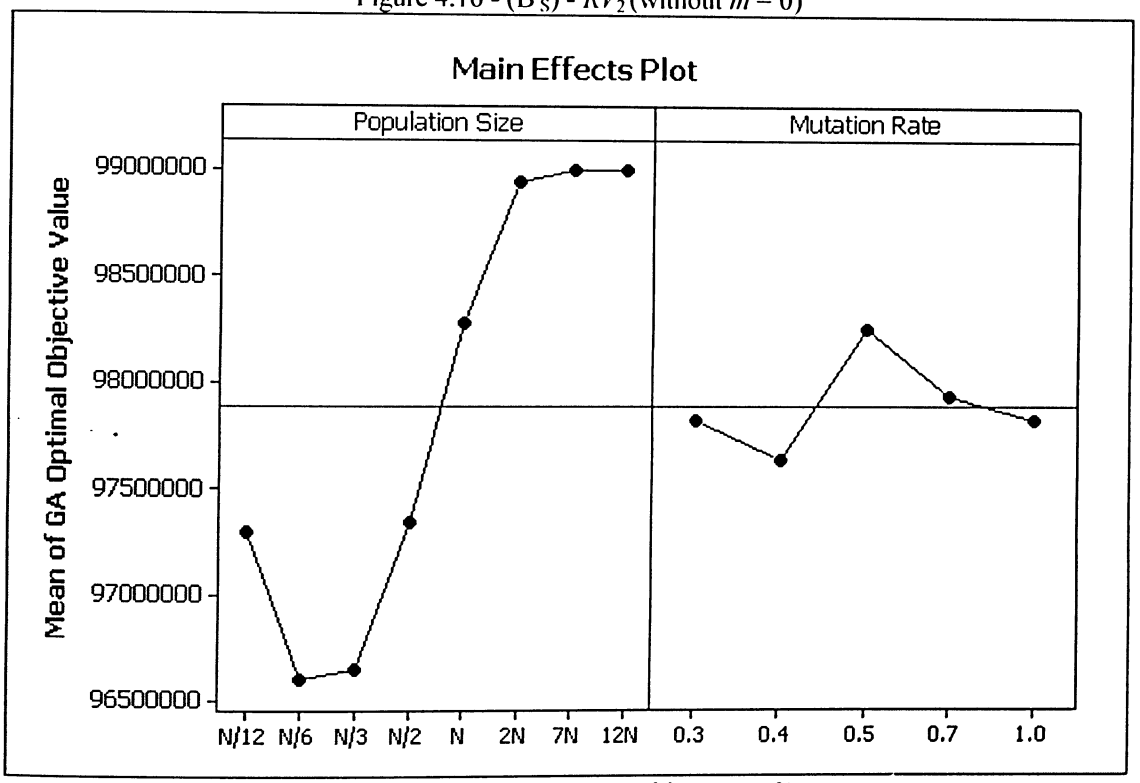

Figure 4.17 - (C's) - RV_1 (without $m = 0$)

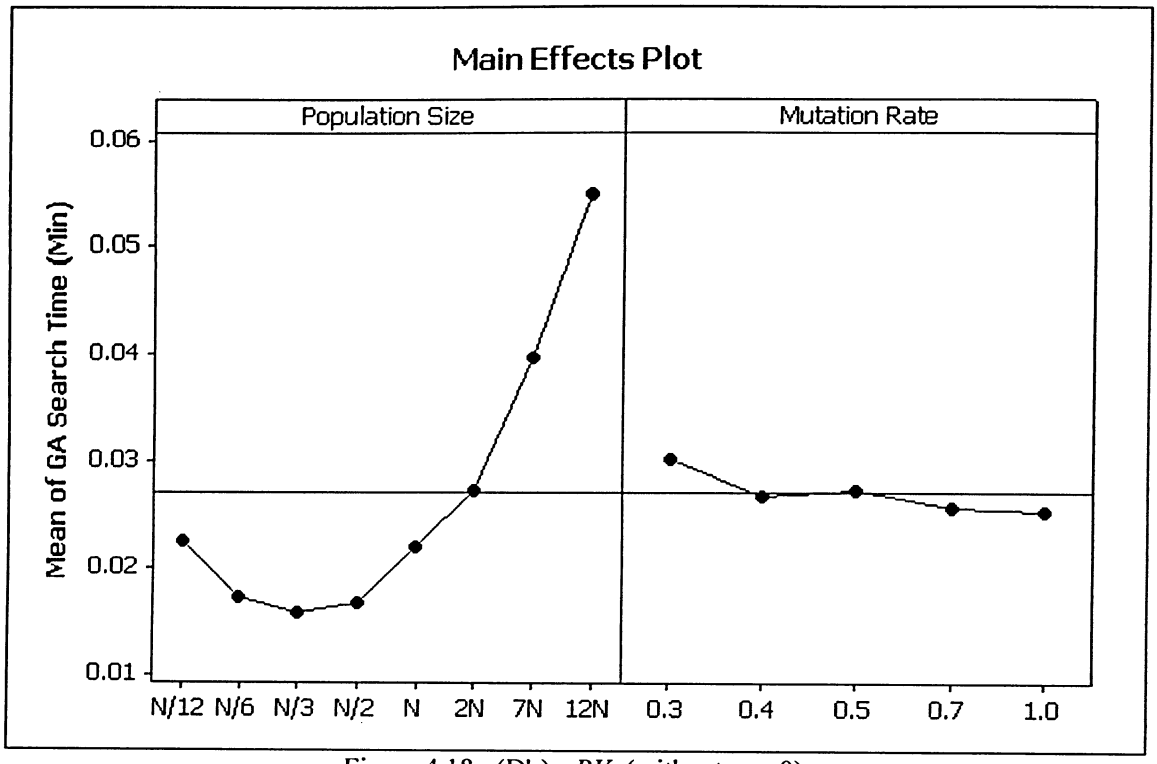

Figure 4.18 - (D's) - RV_2 (without $m = 0$)

4.4 Overall Conclusion

After reviewing the results of ANOVA for the three problems above, the following overall conclusions are reached:

1. Regardless of size of the problem, implementing any *mutation-rate* less than 0.3 ($m < 0.3$), with any *population-size,* generates a poor GA solution in terms of quality.

2. For the problems of size medium and large, defining the size of the problem as N, running GA with a small *population-size* (*N/6* or *N/3*) and a large *mutation-rate* (0.5 or 0.7 or 1) is recommended in order to get the most optimal solution in the shortest possible time. However, for the large size problem choosing a *mutation-rate* of 0.4, with any *population-size,* generates the best GA solution in terms of quality.

3. For the problem of small size, defining the size of the problem as N, running GA with a small *population-size* ($N/6$ or $N/3$) and a small *mutation-rate* (0.3, 0.4) is recommended in order to get the most optimal solution in the shortest possible time. However, applying a small *population-size* ($N/6$ or $N/3$) with a large *mutation-rate* (0.5 or 0.7 or 1) to a small size problem does not generate a bad solution.

Chapter 5

EFFECTIVENESS OF THE PROPOSED GA

5.1 The Proposed GA Verification and Effectiveness

In this section, the GA is compared with the existent branch-and-bound method (Mosel) for the purpose of checking the proximity of the GA solution to the optimal solution. In order to do so, six problems, including the three problems from chapter 4, have been solved with both methods, and then the results have been compared.

The hardware and software specifications used for running Mosel and GA are provided in Appendix II.

Before solving each problem by GA, a test was performed in order to verify the accuracy of GA. First each problem was solved by Mosel, and then the generated schedule (solution) from Mosel was fed as input to GA. The GA produced the same objective values for each objective component (described in chapter 3) as Mosel. In this way, the accuracy of modeling each constraint and objective function (in Java code) was verified. After this verification test the problem was solved by GA. What follows is a comparison of GA and Mosel for each of the six problems. First, some terms that are used in this section are defined.

5.2 Terms

Best Solution: The value of the best integer feasible solution found by Mosel or GA; also known as the Best Objective Value.
Best Bound: A bound on the value of the best integer feasible solution found thus far by Mosel.
Gap_{mb}: The percentage gap between Mosel's best solution and best bound.
Gap_{gm}: The percentage gap between GA's and Mosel's best solution.
Gap_{gb}: The percentage gap between GA's best solution and Mosel's best bound.
Q: Quality

T: Time

GA Time: GA Search Time

GA Objective: GA's Best Objective Value for a Specific GA's Run

5.3 Compared Problems

5.3.1 Problem 1 (Large problem from chapter 4)

This problem includes 289 employees (N = 289). The best integer solution found by Mosel is "116,881,006.40"; the percentage gap between Mosel's best solution and best bound is "0.00003%". Mosel found this solution after "0.03" min. The same problem was solved with GA. The GA best solutions with different p and m parameters were provided in chapter 4. The percentage gap between these GA solutions and the above Mosel solution are represented in Figure 5.1 and Figure 5.2. Figure 5.1 depicts the gaps (Gap_{gm}) for all the GA parameters (five levels of p and six levels of m) and Figure 5.2 depicts the gaps (Gap_{gm}) only for the best parameters that were concluded from ANOVA ($p = N/3$ and $N/6$, m = 0.4, 0.5.0.7, 1). Figure 5.2 indicates that solving GA with the above parameters ensures that the proximity of GA to Mosel's best solution is less than 0.004 %. The best and worst case scenarios in terms of quality and time, for the GA solutions that are depicted in Figure 5.2, are given in Table 5.1.

	GA Objective	GA Time (Min)	Gap_{gb}%	Gap_{gm}%	m	p
best case scenario (Q)	116,881,771.00	106.01	0.00068	0.00065	0.4	$N/3$= 97
worst case scenario (Q)	116,885,205.00	12.16	0.00362	0.00359	1	$N/6$= 48
best case scenario (T)	116,883,740.00	6.56	0.00237	0.00234	0.5	$N/3$= 97
worst case scenario (T)	116,882,581.00	128.18	0.00137	0.00135	0.4	$N/3$= 97

Table 5.1 - The best and worst case scenarios of GA solutions (Problem 1)

As observed in Table 5.1, Mosel functions between 0.00065% and 0.00359% better than GA in terms of quality for this problem.

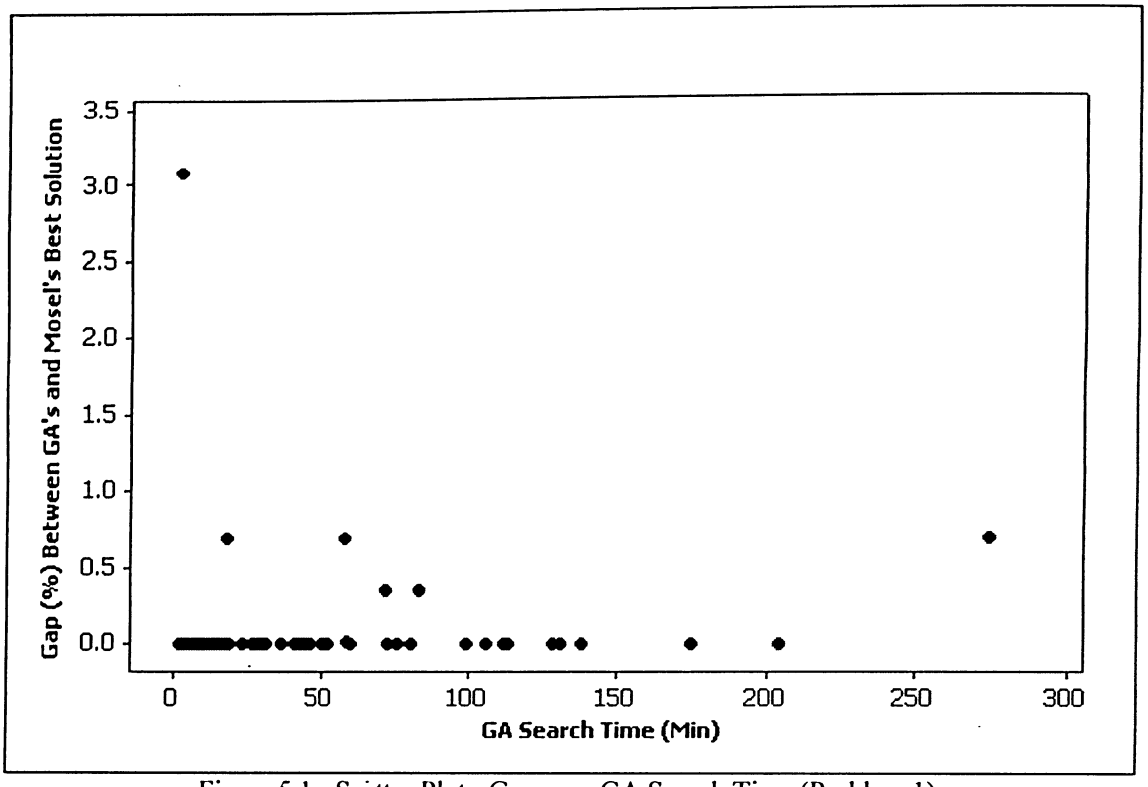

Figure 5.1 - Scatter Plot - Gap_{gm} vs. GA Search Time (Problem 1)

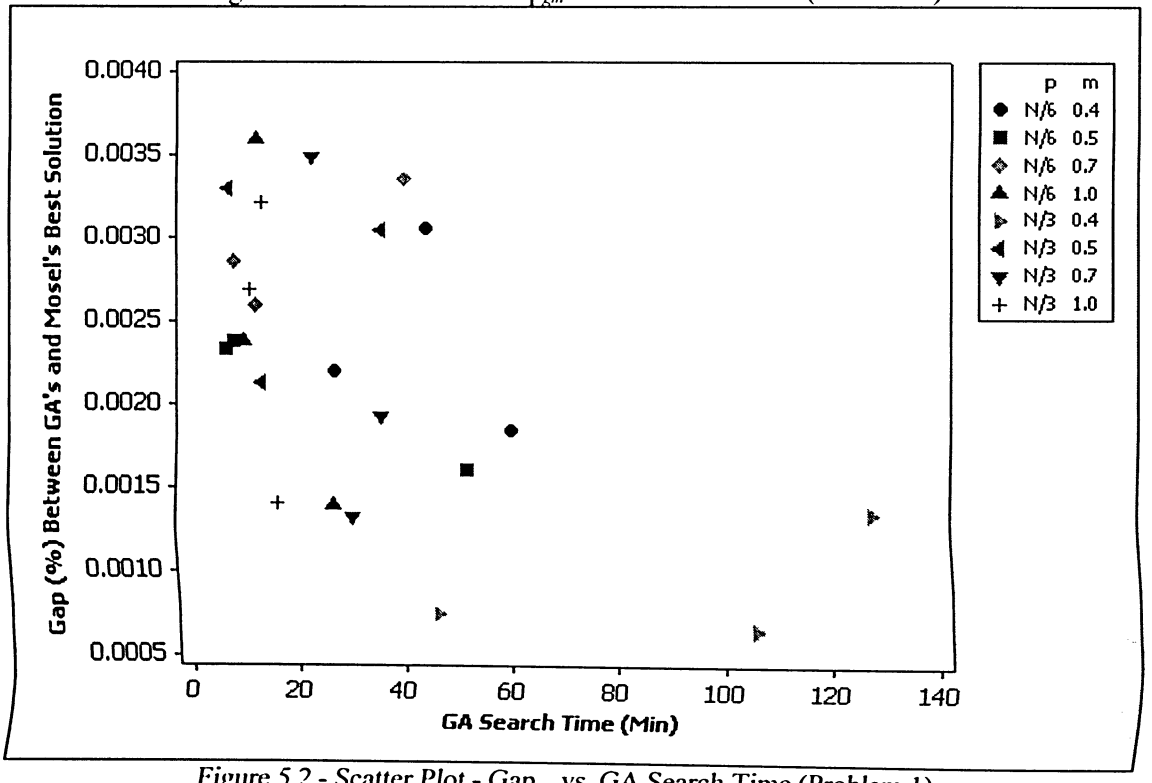

Figure 5.2 - Scatter Plot - Gap_{gm} vs. GA Search Time (Problem 1)

5.3.2 Problem 2 (Medium problem from chapter 4)

This problem includes 131 employees (N = 131). The best integer solution found by Mosel is "556,770,618.50"; the percentage gap between Mosel's solution and best bound is "0.12%". Mosel found this solution after "10.7" min. Mosel finds a better integer solution, "556,770,070.90", after 10 hours.

The same problem was solved with GA. GA's best solutions with different p and m parameters were provided in chapter 4. The percentage gap between these GA solutions and the above Mosel solution are represented in Figure 5.3 and Figure 5.4. Figure 5.3 depicts the gaps (Gap_{gm}) for all the GA parameters (five levels of p and six levels of m), and Figure 5.4 depicts the gaps (Gap_{gm}) only for the best parameters that were concluded from ANOVA (p = N/2 and N/3, m = 0.5.0.7, 1). The best and worst case scenarios, in terms of quality and time, for GA solutions that are depicted in Figure 5.4 are given in Table 5.2.

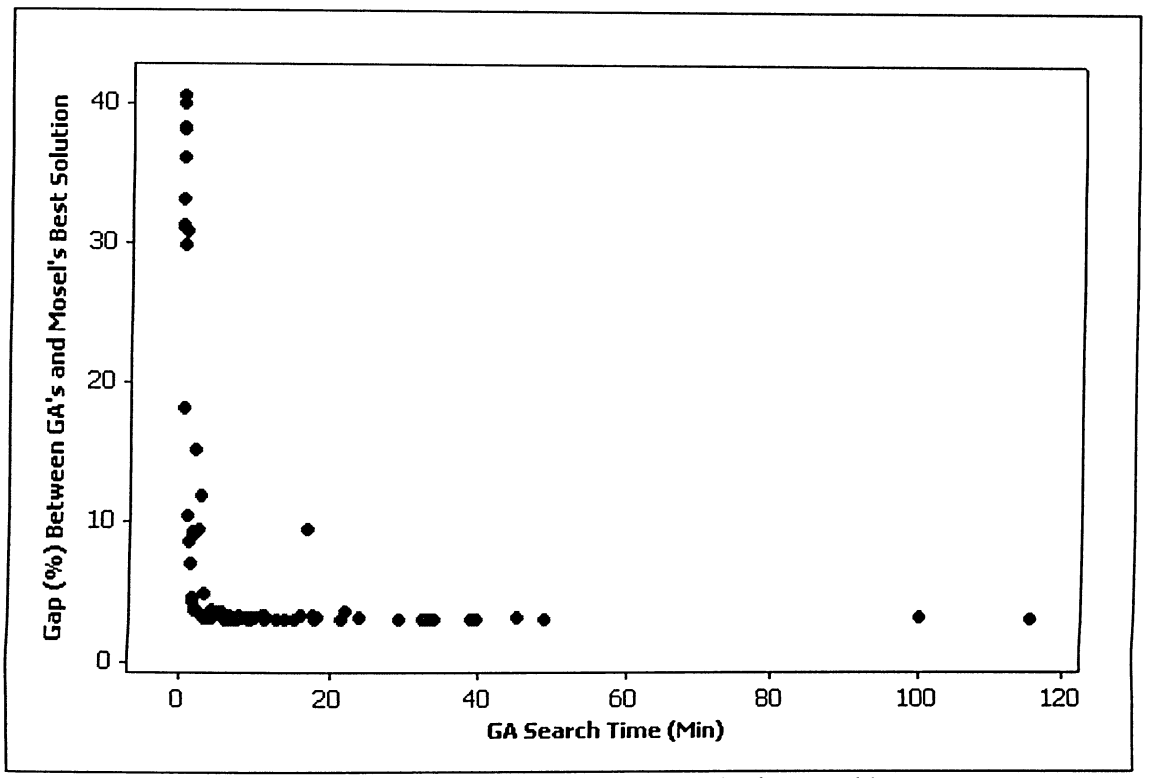

Figure 5.3 - Scatter Plot - Gap_{gm} vs. GA Search Time (Problem 2)

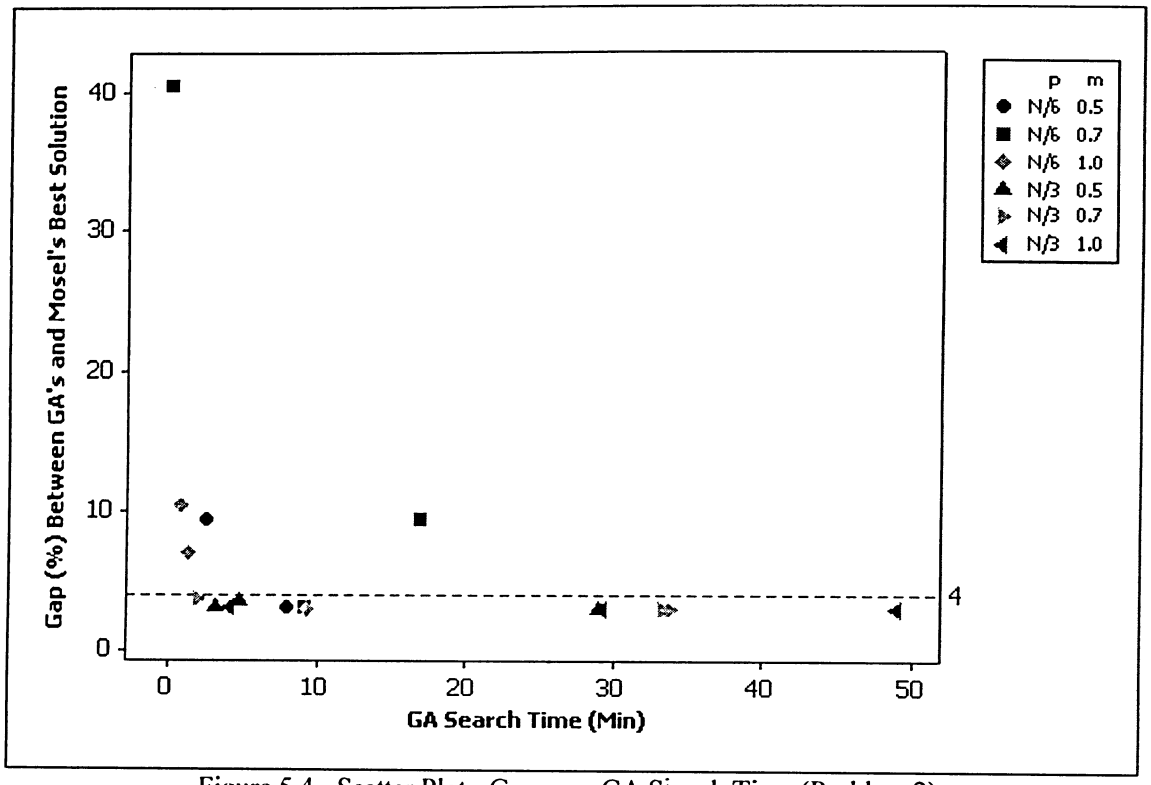

Figure 5.4 - Scatter Plot - Gap_{gm} vs. GA Search Time (Problem 2)

	GA Objective	GA Time (Min)	Gap_{gb}%	Gap_{gm}%	m	p
best case scenario (Q)	573,166,457.00	48.88990	3.07024	2.94481	1	N/3 = 43
worst case scenario (Q)	782,768,824.00	0.02567	40.76220	40.59090	0.5	N/6 = 22
best case scenario (T)	782,768,824.00	0.02349	40.76220	40.59090	0.7	N/6 = 22
worst case scenario (T)	573,166,457.00	48.88990	3.07024	2.94481	1	N/3 = 43

Table 5.2 - The best and worst case scenarios of GA solutions (Problem 2)

As observed in Table 5.2, Mosel functions between 2.95% and 40.59% better than GA in terms of quality for this problem. However, as it is observed from Figure 5.4, the Gap_{gm} for most of GA's objectives is less than 4%.

5.3.3 Problem 3 (Small problem from chapter 4)

This problem includes 17 employees (N = 17). The best integer solution found by Mosel is "95,931,457.24" and best bound is "95,931,457.24"; the percentage gap between the best Mosel solution and best bound is "0.00%". Mosel found this solution after "0.001" min. Mosel denotes that the problem is unfinished which indicates it is still possible to find a better integer solution than best bound (95,931,457.24).

The same problem was solved with GA. The GA best solutions with different p and m parameters were provided in chapter 4. The percentage gap between these GA solutions and the above Mosel solution are represented in Figure 5.5 and Figure 5.6. Figure 5.5 depicts the gaps (Gap_{gm}) for all the GA parameters (eight levels of p and six levels of m), and Figure 5.6 depicts the gaps (Gap_{gm}) only for the best parameters that were concluded from ANOVA (p = $N/2$ and $N/3$, m = 0.3 and 0.4). The value of "-0.00017" for Gap_{gm} from Table 5.3, indicates that GA found a better solution than Mosel for this problem.

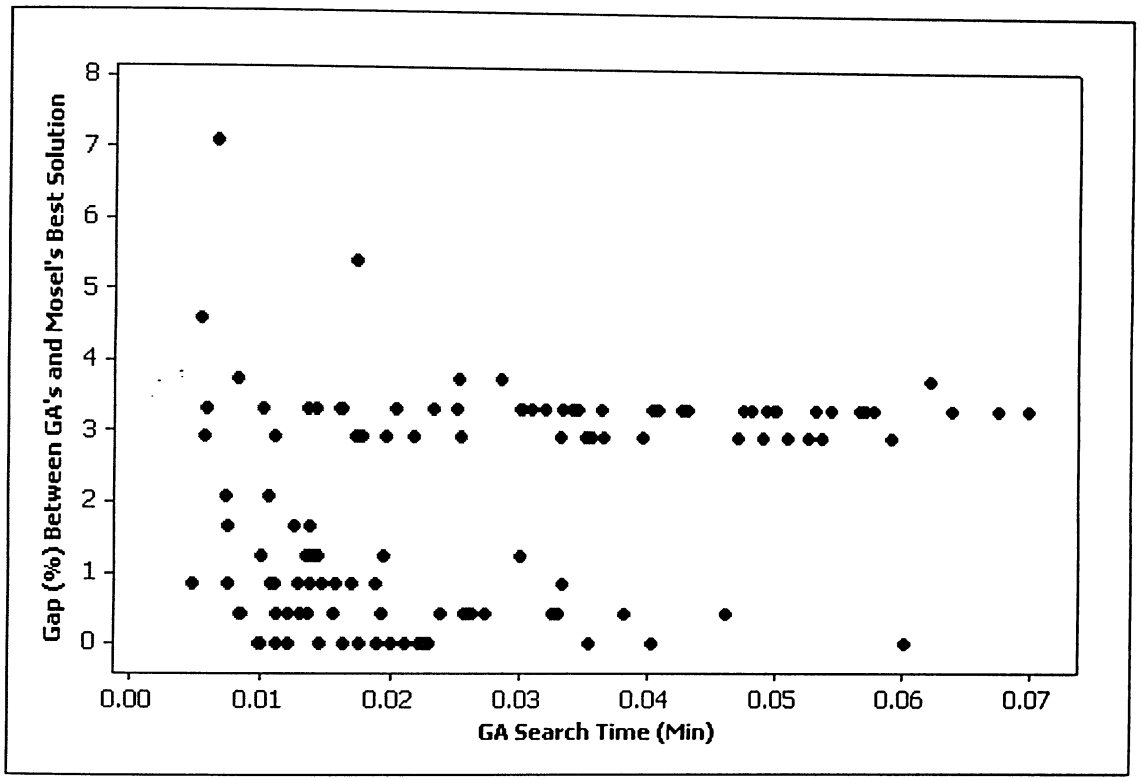

Figure 5.5 - Scatter Plot - Gap_{gm} vs. GA Search Time (Problem 3)

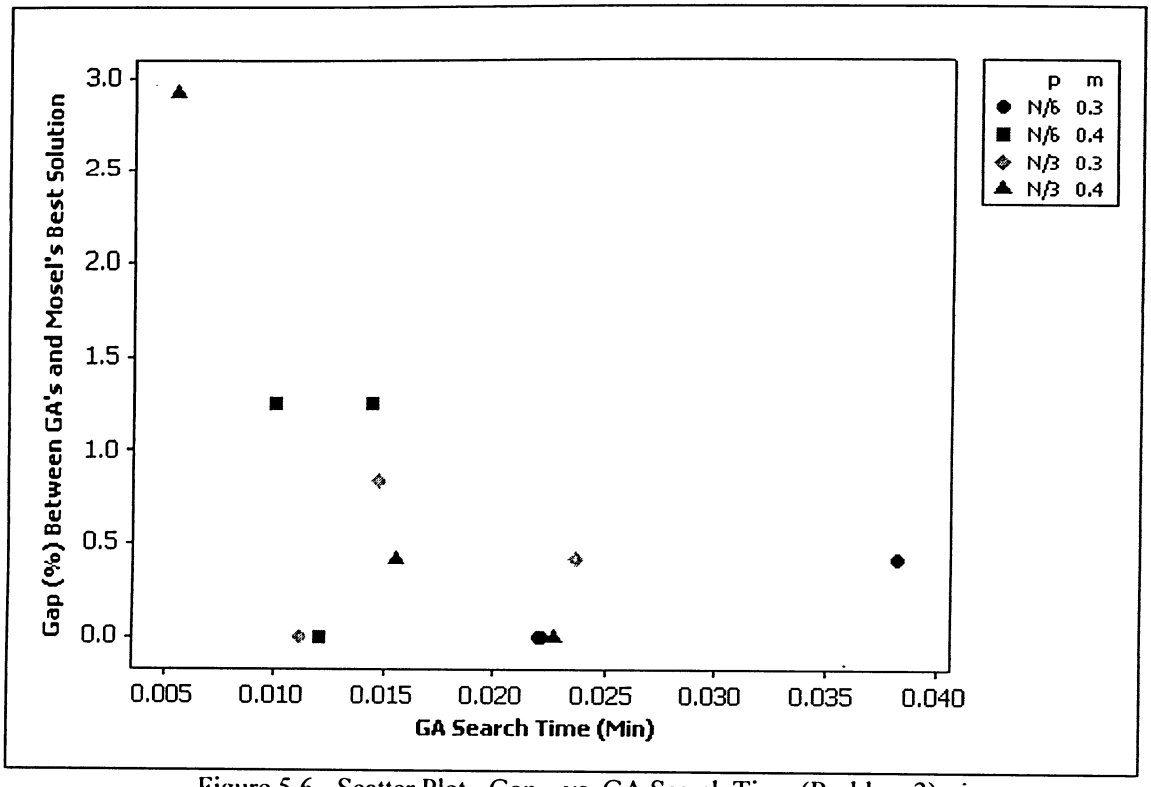

Figure 5.6 - Scatter Plot - Gap_{gm} vs. GA Search Time (Problem 3)

	GA Obj	GA Time(Min)	Gap_{gb}%	Gap_{gm}%	*m*	*p*
best case scenario (*Q*)	95,931,293.00	0.01118	-0.00017	-0.00017	0.3	*N*/3 = 6
worst case scenario (*Q*)	98,731,057.00	0.00545	2.91833	2.91833	0.4	*N*/3 = 6
best case scenario (*T*)	98,731,057.00	0.00545	2.91833	2.91833	0.4	*N*/3 = 6
worst case scenario (*T*)	96,330,759.00	0.03821	0.41624	0.41624	0.3	*N*/6 = 3

Table 5.3 - The best and worst case scenarios of GA solutions (Problem 3)

5.3.4 Problem 4

This problem includes 45 employees (N = 45). The best integer solution found by Mosel is "2,177,299,943.00" and best bound is "1,958,935,552.00"; the percentage gap between Mosel's best solution and best bound is "11.15%". Mosel found this solution after "8.60" min. Mosel could not find a better solution after increasing the search time to 10 hours. The same problem was solved with GA using the following parameters: $p = N/2$ and $N/3$ and m = 0.3, 0.4, 0.5, 0.7 and 1. With each of the above parameters, GA found a better solution than Mosel. The worst case scenario for GA was found with m = 0.3 and $p = N/3$ parameters. The associated values are as follows: objective value of "2,154,503,746.00", search time of "13.13" min, the Gap_{gb} of "9.98339%" and the

Gap_{gm} of "-1.04700%". All other parameters produced the same best case scenario result; the solution with the objective value of "2,147,704,852.00", the Gap_{gb} of "9.63632%" and the Gap_{gm} of "-1.35926%". The search times ranged from "5.58" min for $m = 1$ and $p = N/6$, to "14.15" min for $m = 0.4$ and $p = N/6$. As a result GA behaves between 1.36% and 1.05% better than Mosel in terms of quality for this problem.

5.3.5 Problem 5

This problem includes 331 employees ($N = 331$). The best integer solution found by Mosel is "787,926,158.70" and best bound is "787,923,291.50 "; the percentage gap between Mosel's best solution and best bound is "0.04%". Mosel found this solution after "2.85" min. The same problem was solved with GA with the parameters in Table 5.4; the parameters were chosen according to results of ANOVA. The results of GA runs and the calculated gaps are also displayed in Table 5.4. As observed in Table 5.4, Mosel's best solution is 3.42% better than GA's worst solution; but GA can find a better solution than Mosel in terms of both quality and time with $m = 0.4$ and $p = N/12$ parameters .

GA Objective	GA Time (Min)	Gap_{gb}%	Gap_{gm}%	m	p
806,057,710.00	0.01	2.30155	2.30117	0.4	$N/6 = 55$
806,059,510.00	0.01	2.30177	2.30140	0.5	$N/6 = 55$
806,057,710.00	0.01	2.30155	2.30117	0.7	$N/6 = 55$
806,059,090.00	0.01	2.30172	2.30135	1	$N/6 = 55$
806,057,710.00	0.02	2.30155	2.30117	0.4	$N/3 = 110$
806,057,710.00	0.02	2.30155	2.30117	0.5	$N/3 = 110$
806,057,710.00	0.02	2.30155	2.30117	0.7	$N/3 = 110$
806,911,060.00	0.02	2.40985	2.40948	1	$N/3 = 110$
780,458,052.00	0.74	-0.94746	-0.94782	0.4	$N/12 = 28$
814,864,421.00	0.02	3.41926	3.41888	0.4	$N/2 = 165$
814,863,761.00	0.05	3.41917	3.41880	0.4	$N = 331$

Table 5.4 - GA best solutions and associated Gaps for different parameters (Problem 5)

5.3.6 Problem 6

This problem includes 599 employees (N = 599). The best integer solution found by Mosel is "926,575,489.30" and best bound is "926,239,040.00 "; the percentage gap between Mosel's solution and best bound is "0.04%". Mosel found this solution after "73.30" min. Mosel could not find a better solution than above after increasing the search time to 10 hours. The same problem was solved with GA with the parameters in Table 5.5; these parameters were chosen according to results of ANOVA. The gaps are depicted in Table 5.5 and Figure 5.7. As observed in Table 5.5, and from Figure 5.7, Mosel functions between 7.34% (worst case scenario of GA) and 1.6% (best case scenario of GA) better than GA in terms of quality for this problem. However, GA can produce a better solution in terms of time with most of the GA parameters. The best solution in terms of time and quality is the solution with a Gap_{gm} of "2.59268%", search time of "46.93" min and is generated by m = 0.4 and p = N/6 parameters.

GA Objective	GA Time (Min)	Gap_{gb}%	Gap_{gm}%	m	p
945,402,341.00	81.12	2.0689	2.03187	0.4	N/3 =100
968,201,469.00	99.94	4.53041	4.49245	0.5	N/3 =100
980,198,676.00	47.71	5.82567	5.78724	0.7	N/3 =100
941,393,574.00	164.33	1.63614	1.59923	1	N/3 =100
950,598,649.00	46.93	2.6299	2.59268	0.4	N/6 =200
975,799,150.00	41.65	5.35068	5.31243	0.5	N/6 =200
976,600,044.00	50.27	5.43715	5.39886	0.7	N/6 =200
980,200,030.00	41.97	5.82582	5.78739	1	N/6 =200
994,596,361.00	25.54	7.38009	7.34110	0.4	N/12=50

Table 5.5 - GA best solutions and associated Gaps for different parameters (Problem 6)

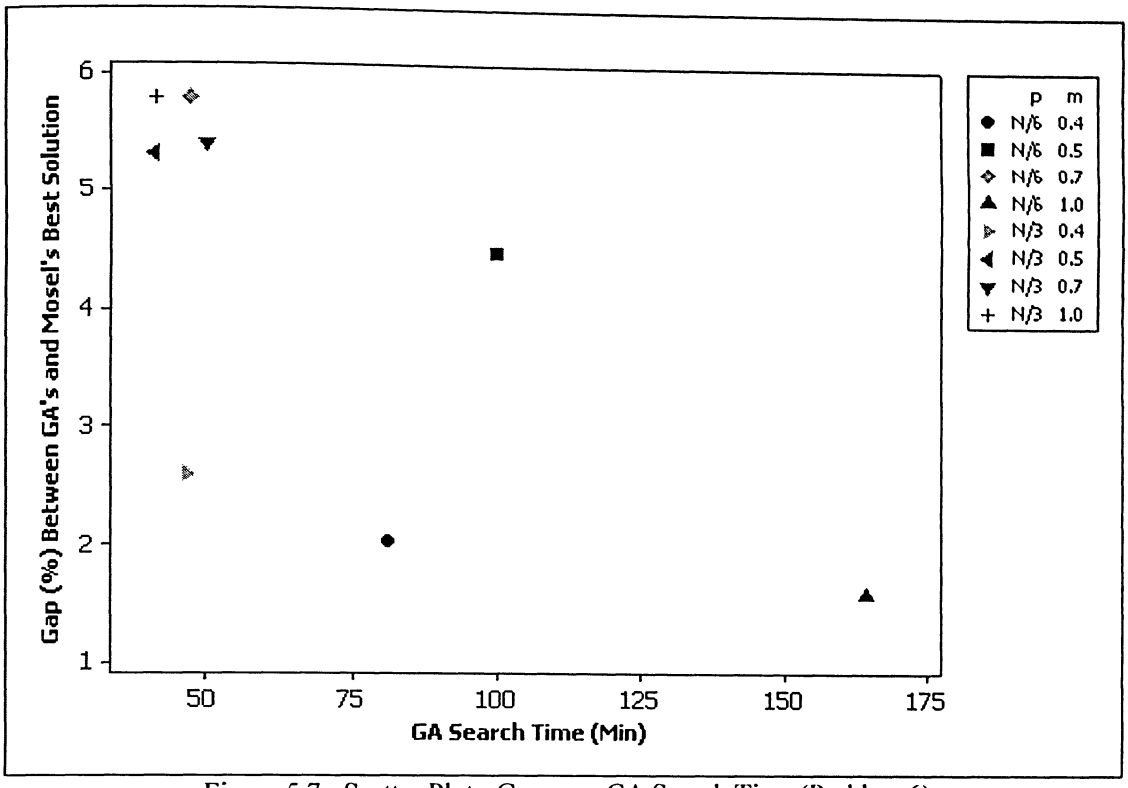

Figure 5.7 - Scatter Plot - Gap_{gm} vs. GA Search Time (Problem 6)

5.4 Conclusion of the Comparison

GA found good solutions for all of the above problems in a reasonable amount of time, and found better solutions than branch-and-bound in three of the above problems. The range of problems tested was large (N=17 to N=599) which shows that the proposed GA is capable of producing optimal, or near optimal, feasible solutions for different sizes of problems.

Chapter 6

CONCLUDING REMARKS AND FUTURE RESEARCH

6.1 Concluding Remarks

In this thesis, a real-world retail labour scheduling problem with different skills, employee types, employee preferences, scheduling preferences, objectives and constraints, that has more difficulties than the problems solved in the literature, has been examined. A GA meta-heuristic has been implemented as the solution approach to the problem. To the best of the author's knowledge, GA has not been used for this type of problem before.

The proposed GA was compared with the conventional, linear integer programming approach. The GA was tested on a set of six real-world problems. In order to tune the GA parameters, three test problems were tested using a range of population size and mutation rate parameters. Then all six were solved with the best of those parameters. The results were compared to those obtained with the branch-and-bound algorithm. It was shown that GA can produce near-optimal solutions for all of the problems, and for half of them, it is more successful than the branch-and-bound method.

Branch-and-bound method (Mosel) is an exact method, however under some circumstances, such as when the size of the problem is large or when the problem is complicated caused by adding some constraints or objective function components to the problem, it is possible that branch-and-bound can not find an optimal solution. In the cases where branch-and-bound method does not behave well, GA can be suggested as an alternative method.

6.2 Key Aspects

Below, some of the key aspects of this thesis are provided.

1. The problem under study is a real-life retail scheduling problem. The size of the problem tackled in this thesis is larger than what the author came across in literature; for instance, one of the test cases is to schedule 599 employees.

2. The objective functions and constraints are more comprehensive compared to those of other studies in the literature.

3. The definition of shift is more comprehensive than the cited studies in the literature. A shift includes day, location, start and end times, skill and activity.

4. Employee type such as fixed-shift, full time, part time, salaried and scheduling preferences such as seniority and the level of skills are considered.

5. There is no limitation on the number of jobs, skills, and skill levels.

6. Shift scheduling, days-off scheduling and task assignment steps (or, in other words, tour scheduling and task assignment steps) are performed simultaneously. Days-off scheduling is performed by "maximum consecutive days" hard constraint.

7. The proposed GA is very flexible to adding more objective functions and constraints.

8. The proposed GA approach can be applied to any non-linear fitness functions, although the fitness function considered in this thesis is a linear function.

9. The solution of this GA specifies which employee is assigned to which shift while the solution of the other reviewed GAs in the literature, only specify the optimal number of assigned employees.

10. The proposed GA is capable of producing optimal, or near optimal solutions for different sizes of problems. A large range of problems (N=17 to N=599) were tested in this thesis.

11. At each iteration of the proposed GA, the feasibility of each solution is checked by every hard constraint in the scheduling problem and if it violates the constraint it is repaired by some repair methods, therefore at each iteration the genetic algorithm contains a population of feasible solutions.

6.3 Future Research

This thesis covered only a GA meta-heuristic. Further study to tackle the problem of retail labour scheduling could be performed using other meta-heuristic approaches, such as tabu search or simulated annealing. When more solution methods become available, it would be beneficial to compare their performance. Developing a memetic algorithm that incorporates other solution methods into the GA is another area of future research.

One of the strengths of the proposed GA is that additional constraints and objectives can be incorporated easily. Further work could be done by adding more constraints and comparing the effectiveness of the GA with different constraints enabled.

More complex crossover methods and dynamic mutation rates could affect the quality and speed of solutions, and additional testing in these areas could enhance the abilities of GA for solving real-world problems.

APPENDIX I: ANOVA PLOTS

This appendix contains all the individual plots used in chapter 4 – Experimental Design and Statistical Analysis. For example, Figure (A_{L1}) and Figure (A_{L2}) are the individual plots of (A_L) in chapter 4.

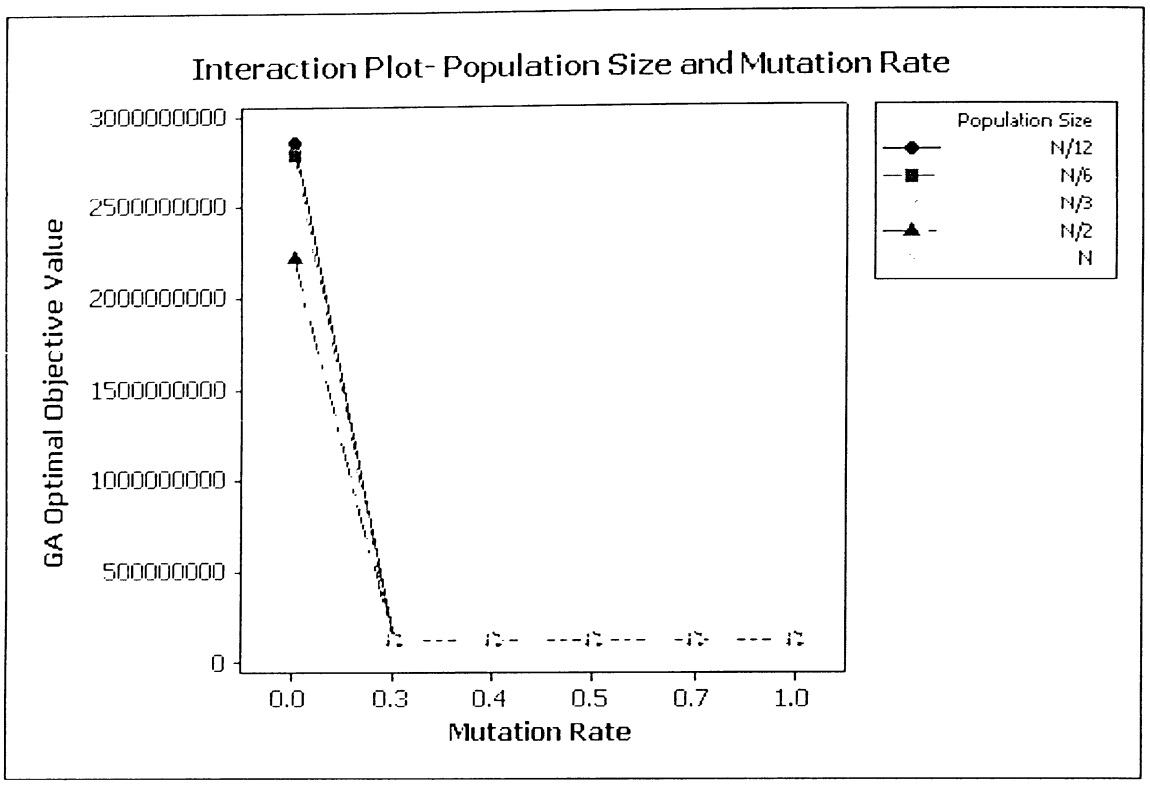

Figure (Λ_{11}) - RV_1 (with $m = 0$)

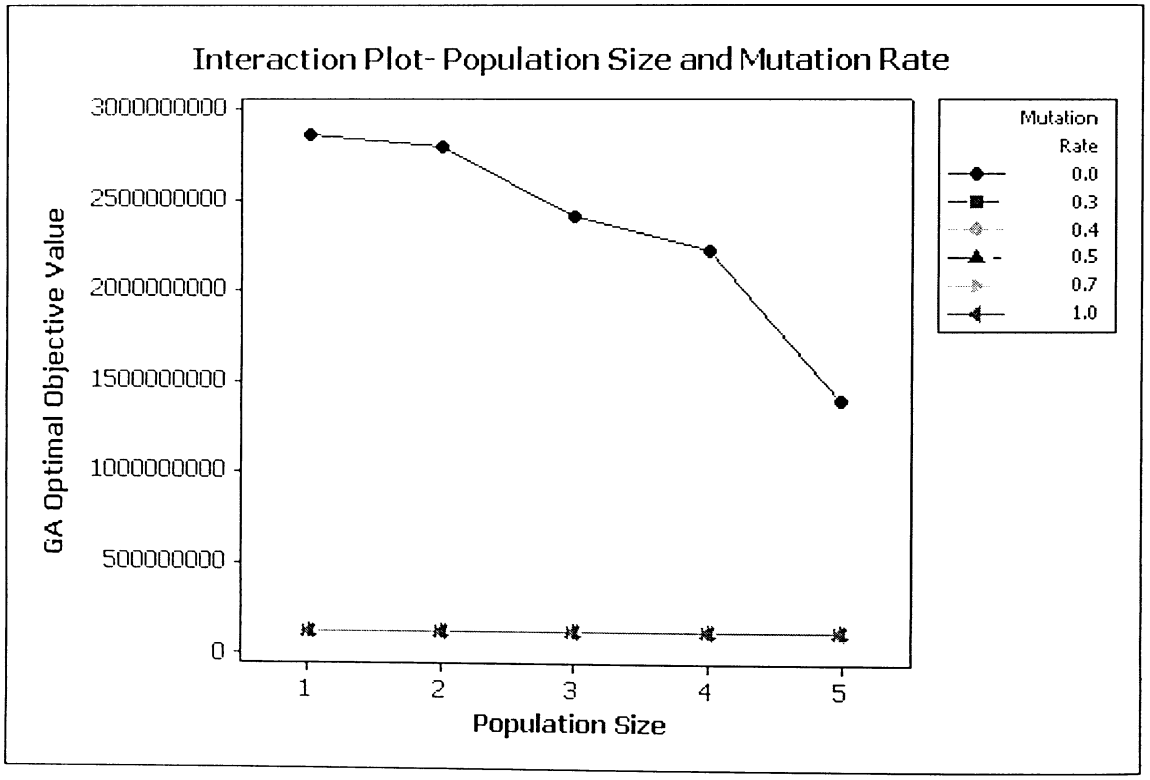

Figure (Λ_{12}) - RV_1 (with $m = 0$)

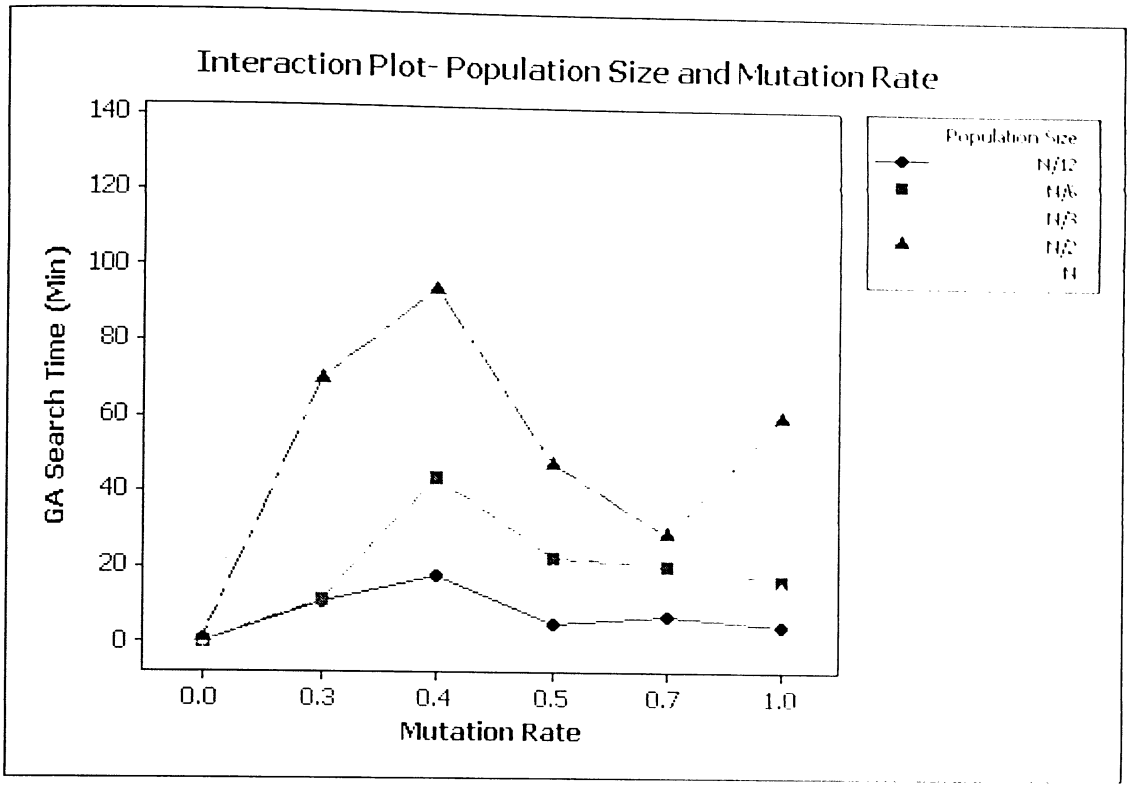

Figure (B_{11}) - RV_1 (with m 0)

Figure (B_{12}) - RV_2 (with m 0)

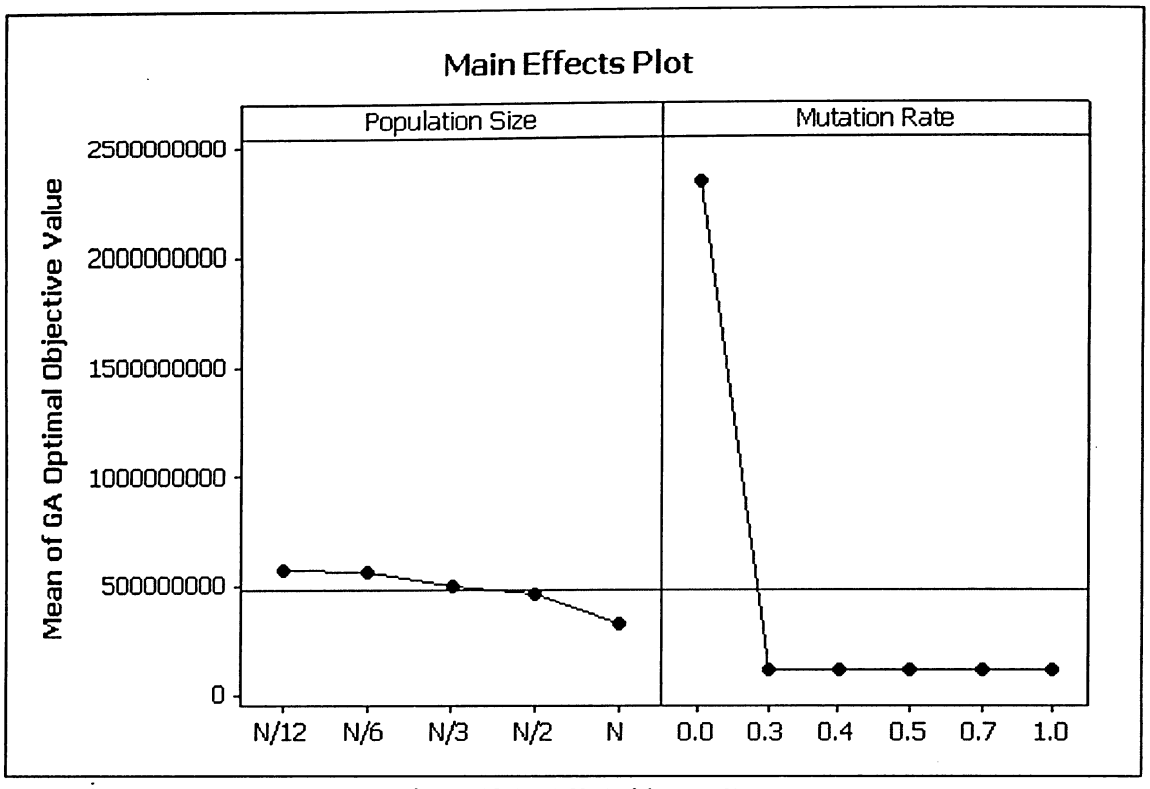

Figure (C_L) - RV_1 (with $m = 0$)

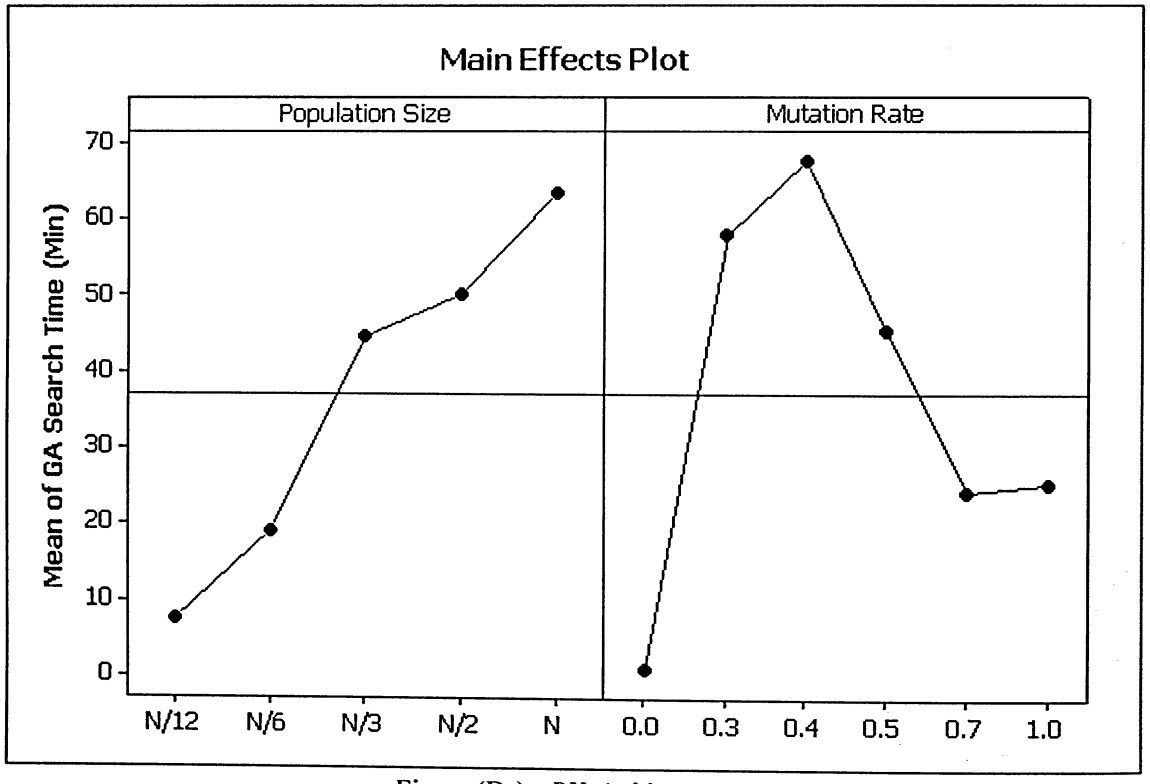

Figure (D_L) - RV_2 (with $m = 0$)

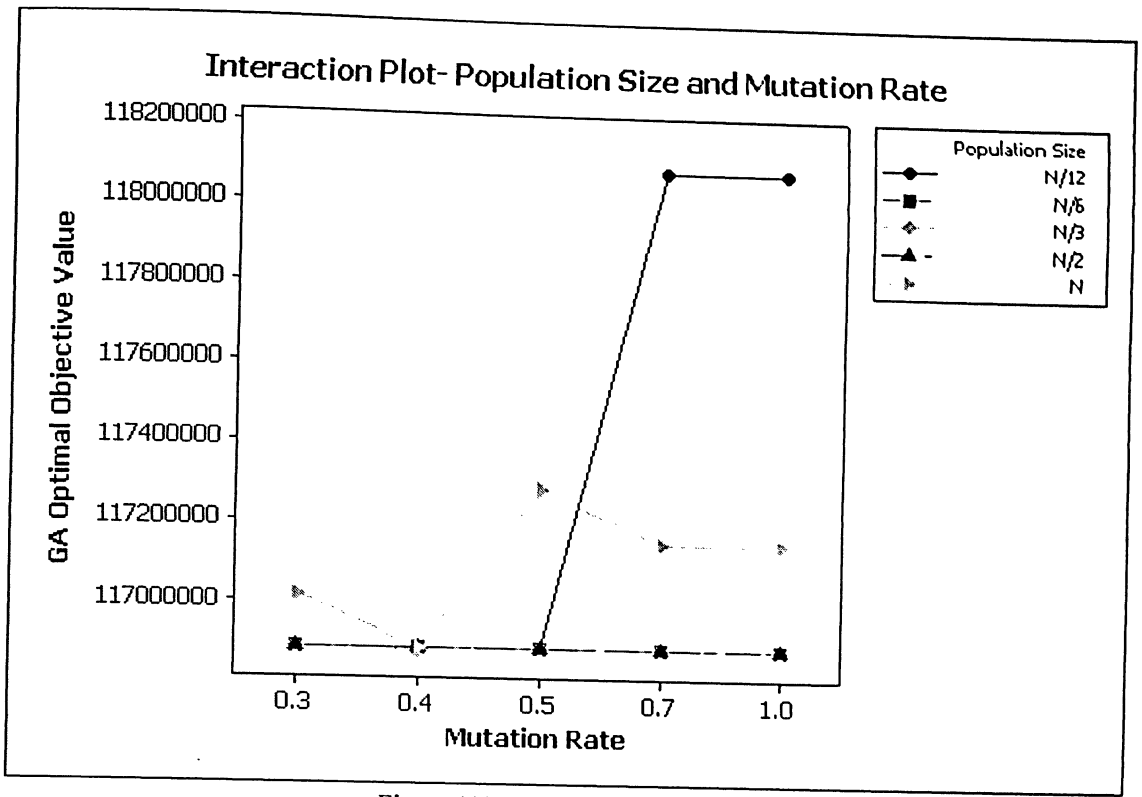

Figure (A'_{L1}) - RV_1 (without $m = 0$)

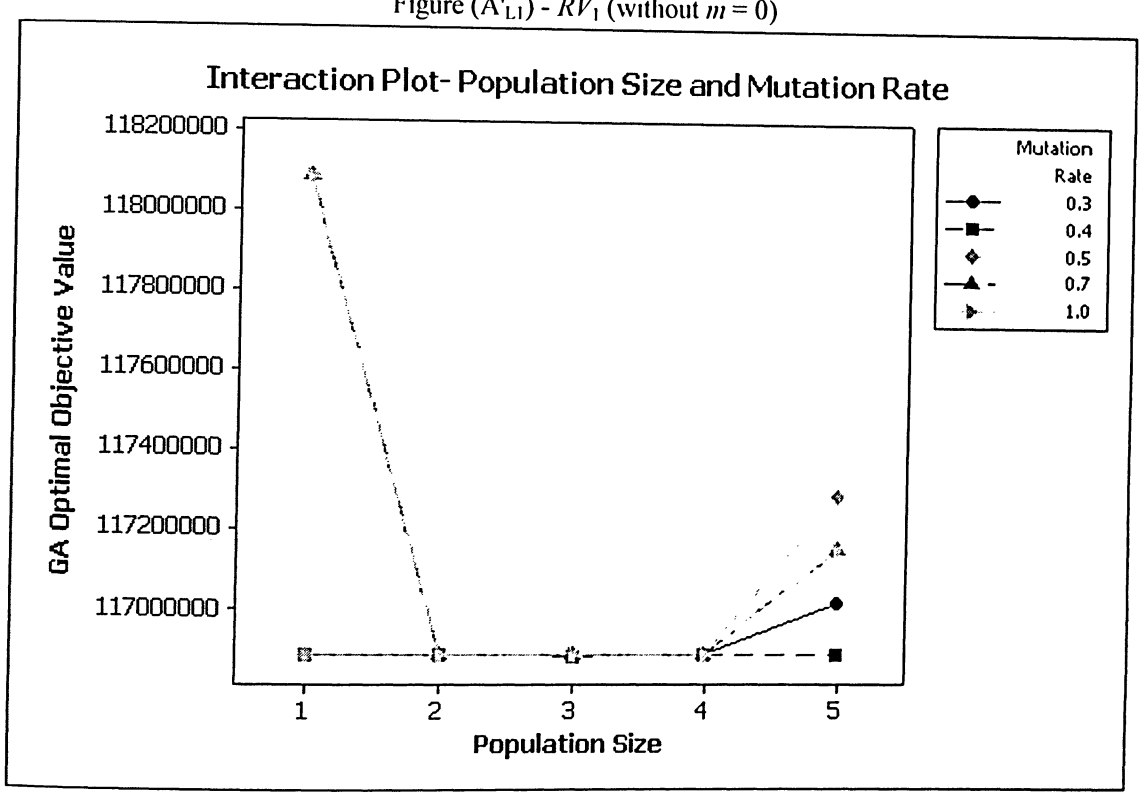

Figure (A'_{L2}) - RV_1 (without $m = 0$)

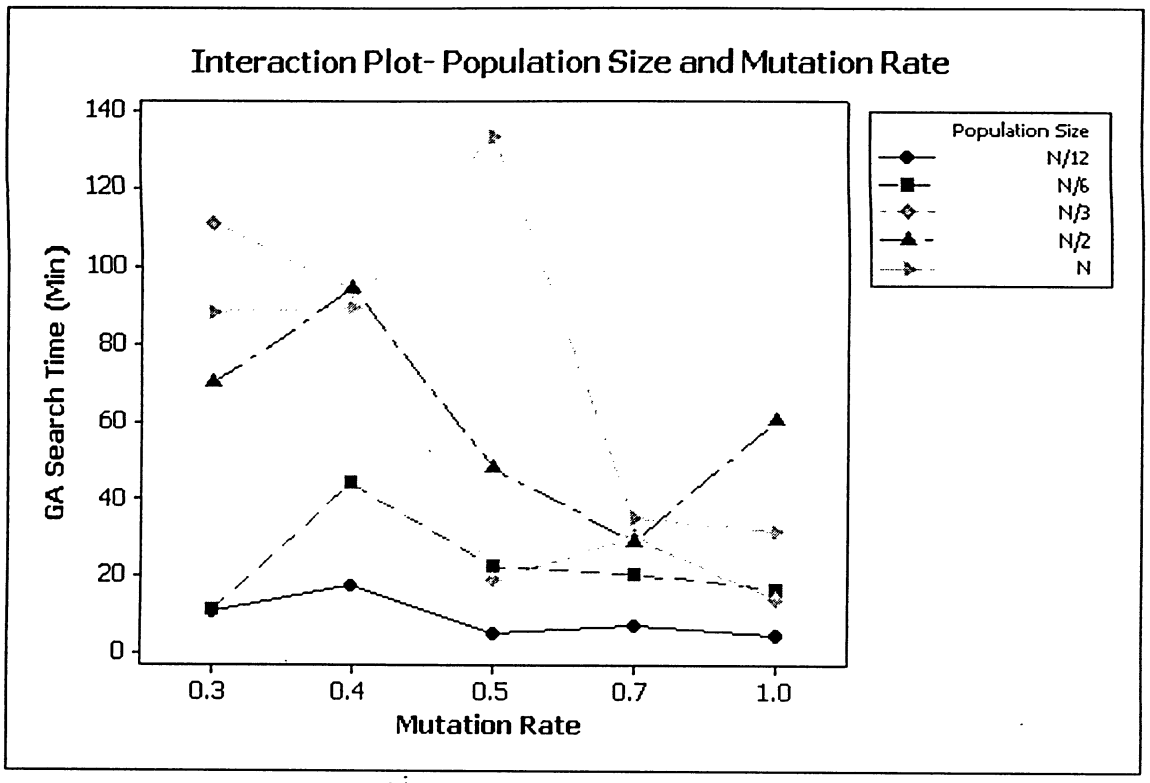

Figure (B'_{L1}) - RV_2 (without $m = 0$)

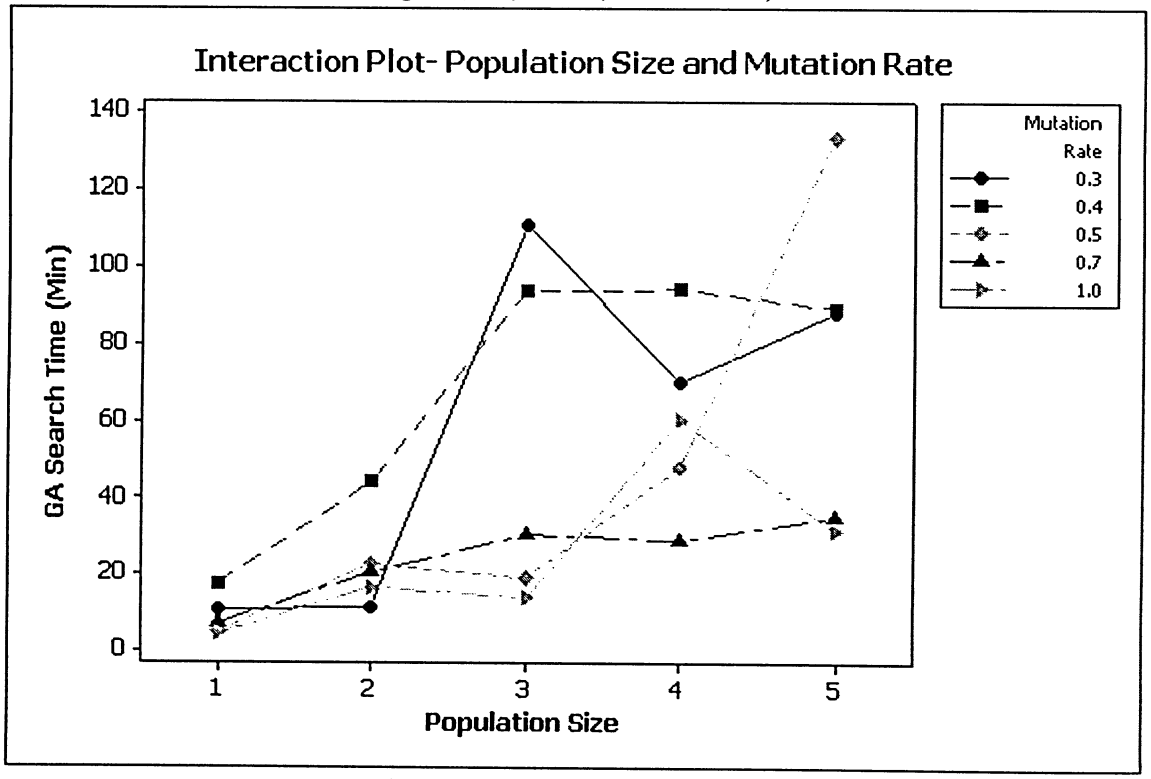

Figure (B'_{L2}) - RV_2 (without $m = 0$)

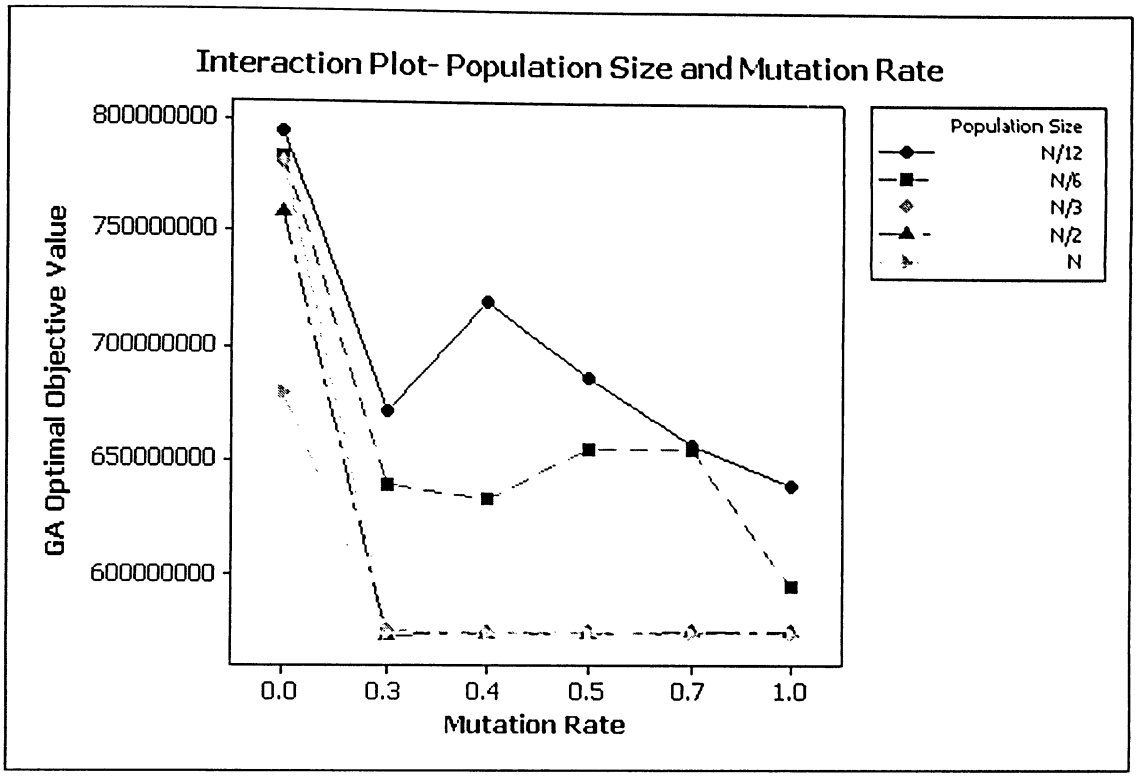

Figure (A_{M1}) - RV_1 (with $m = 0$)

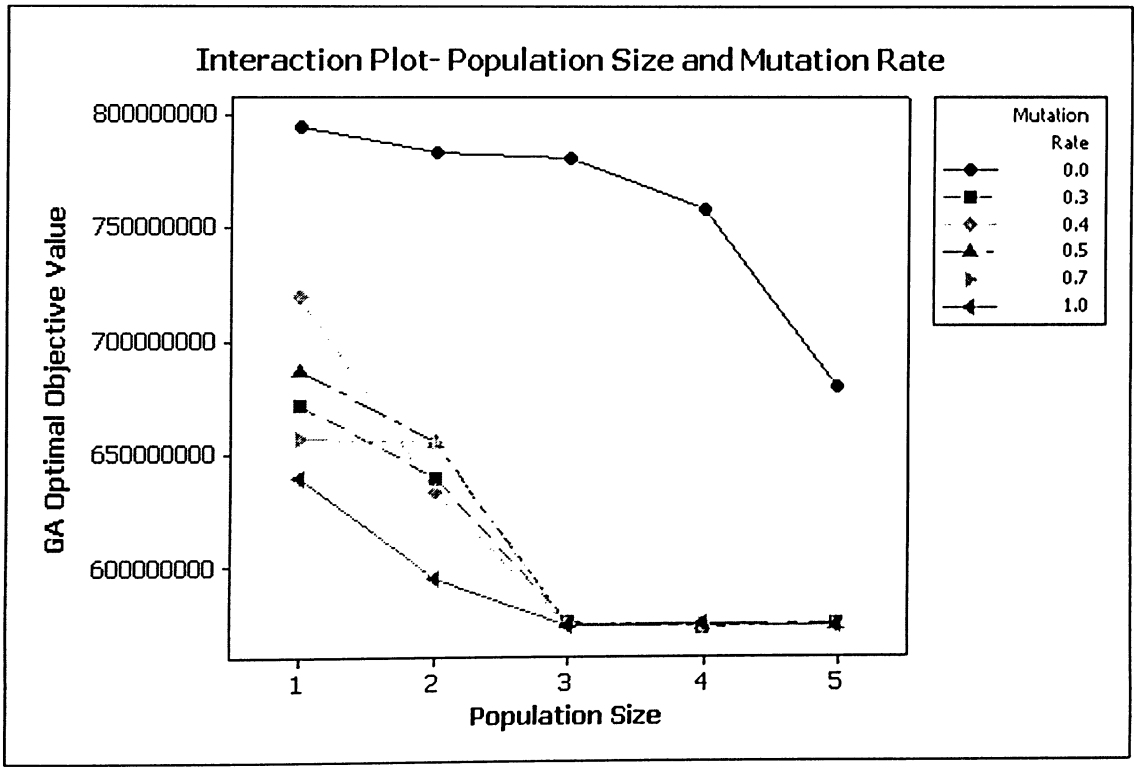

Figure (A_{M2}) - RV_1 (with $m = 0$)

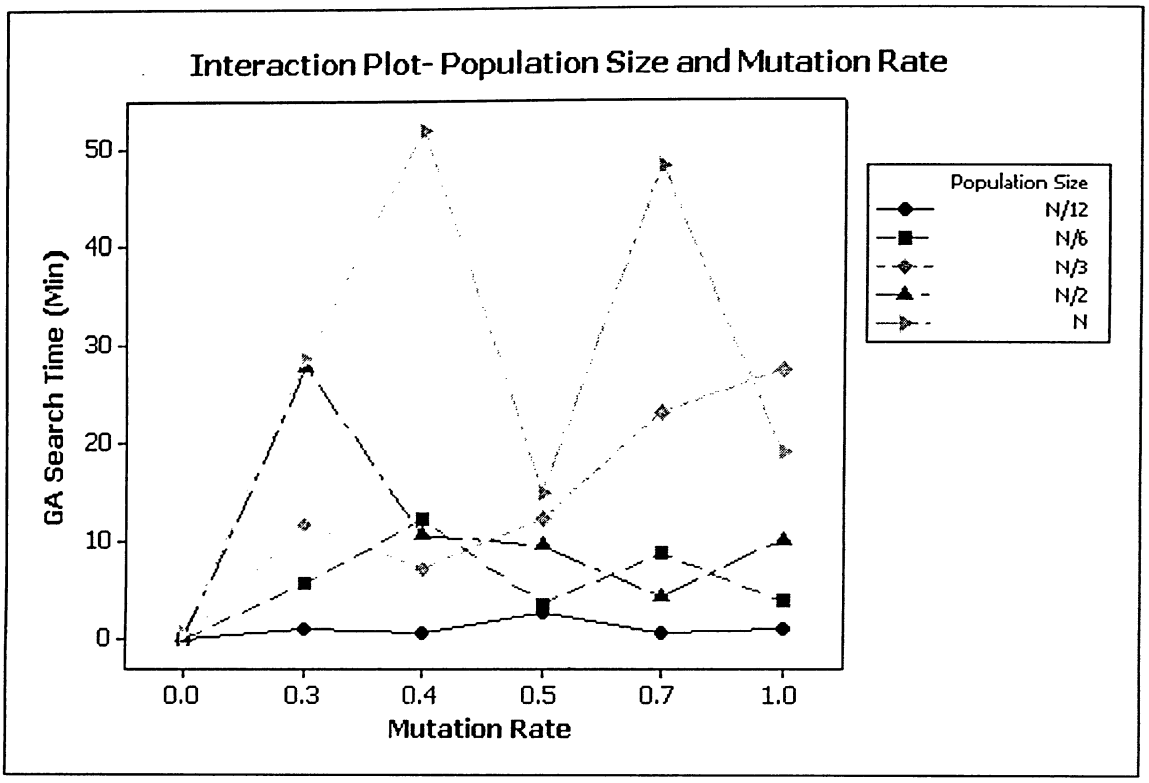

Figure (B_{M1}) - RV_2 (with $m = 0$)

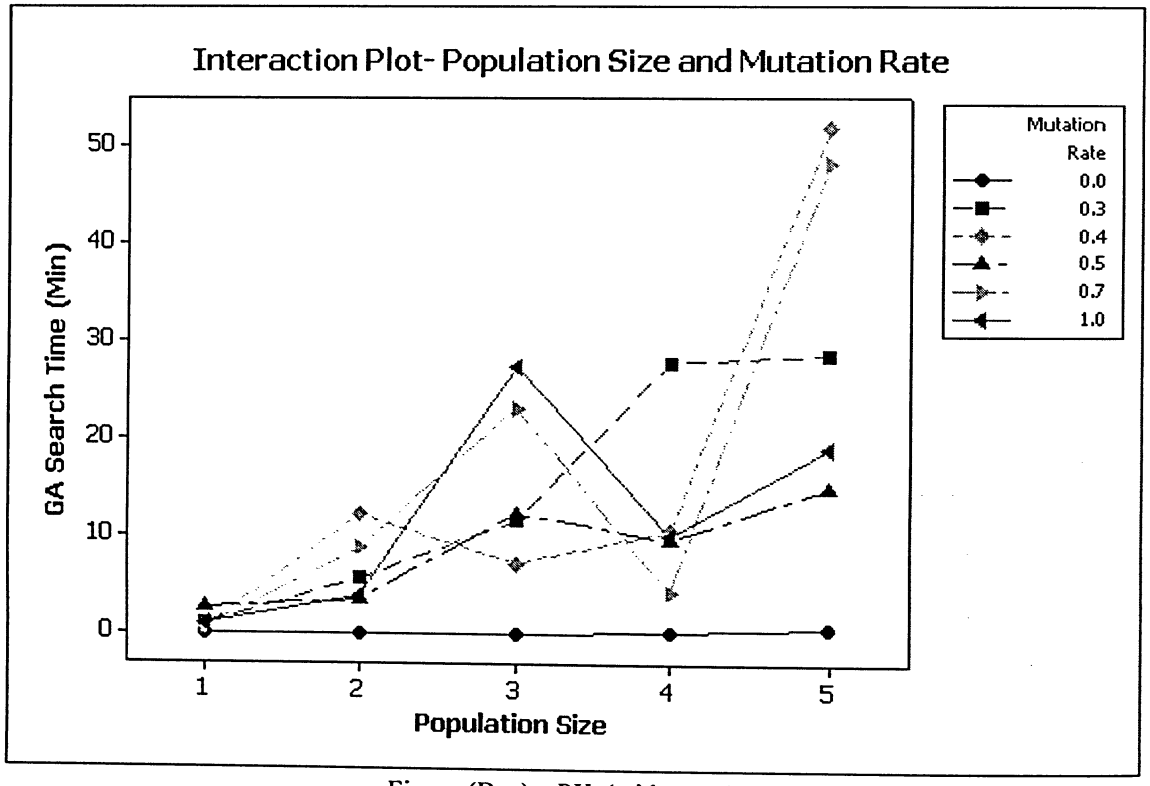

Figure (B_{M2}) - RV_2 (with $m = 0$)

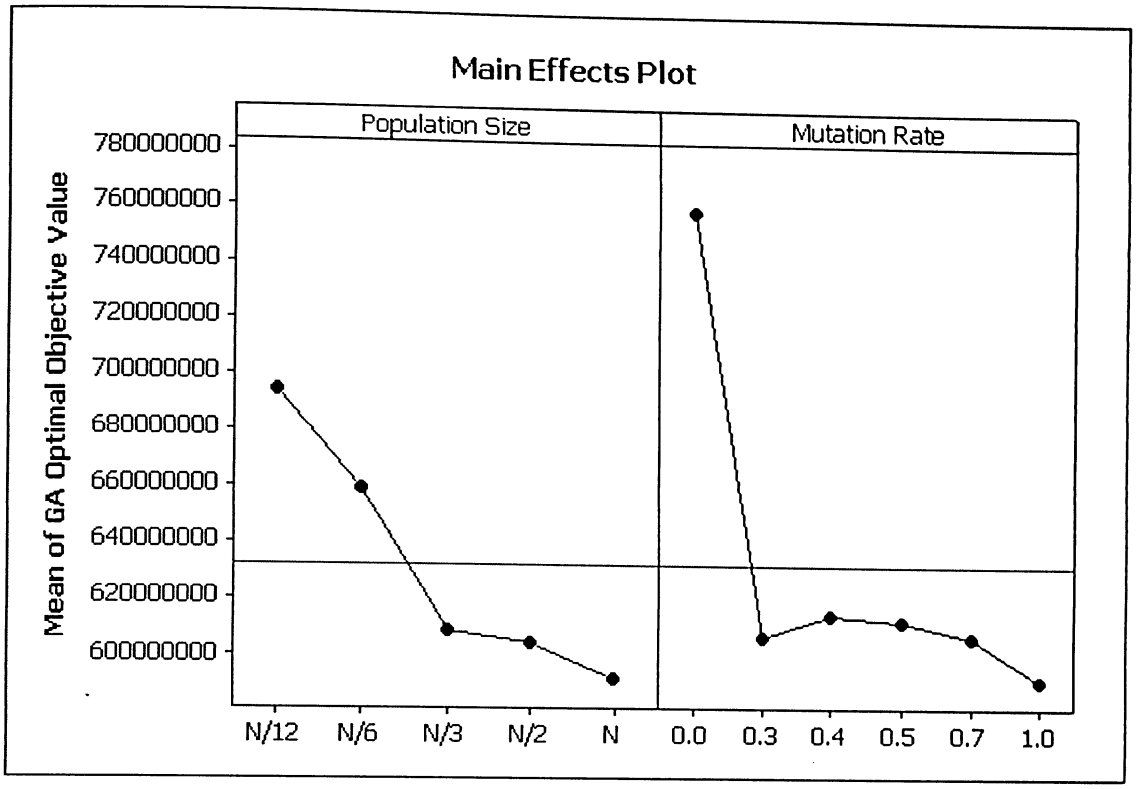

Figure (C_M) - RV_1 (with $m = 0$)

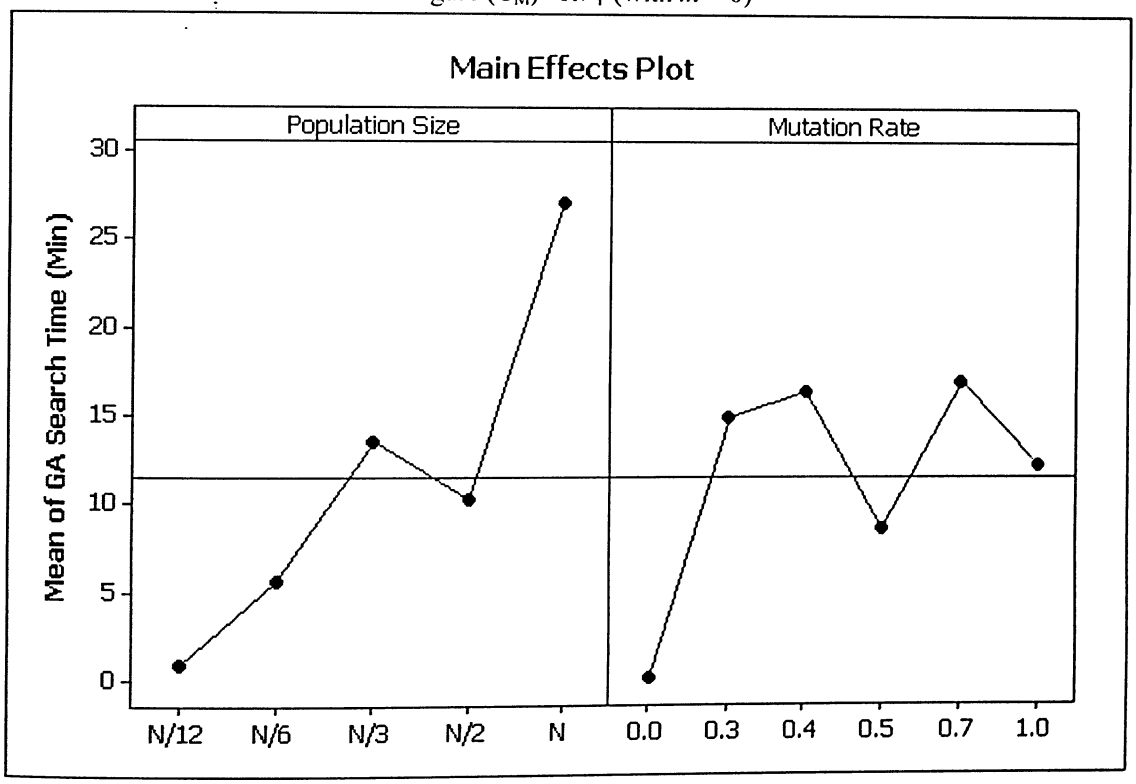

Figure (D_M) - RV_2 (with $m = 0$)

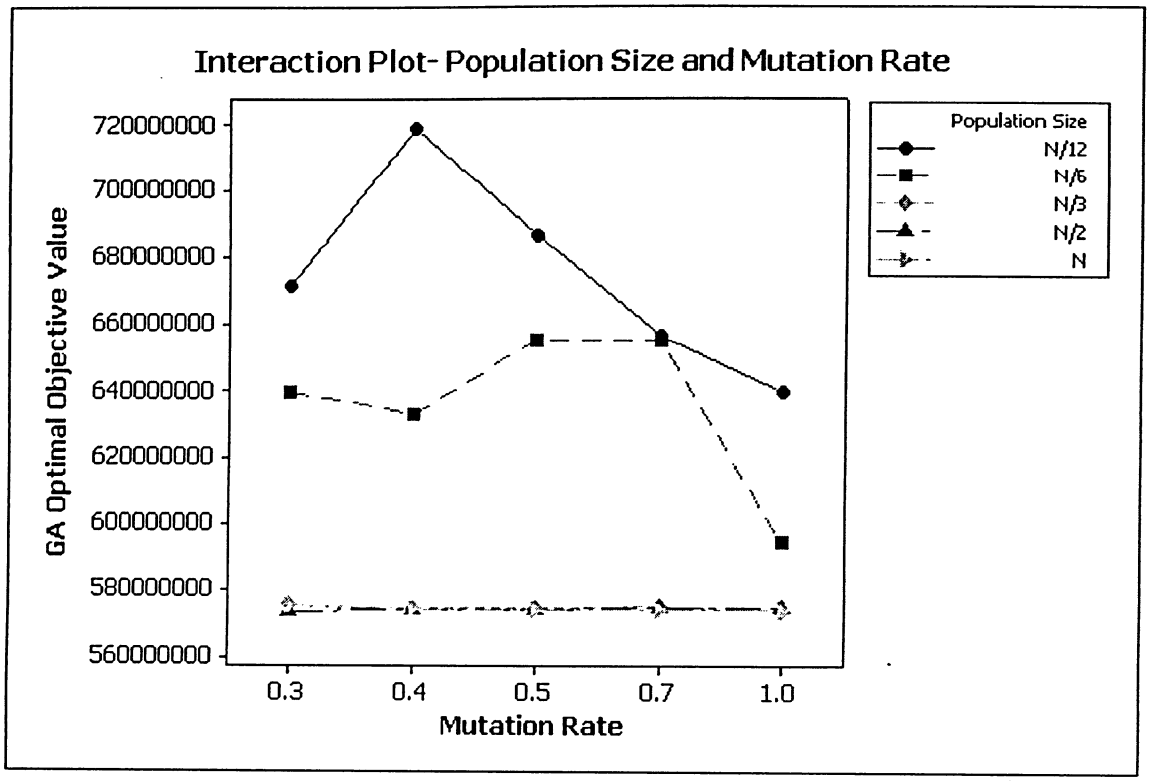

Figure (A'_{M1}) - RV_1 (without $m = 0$)

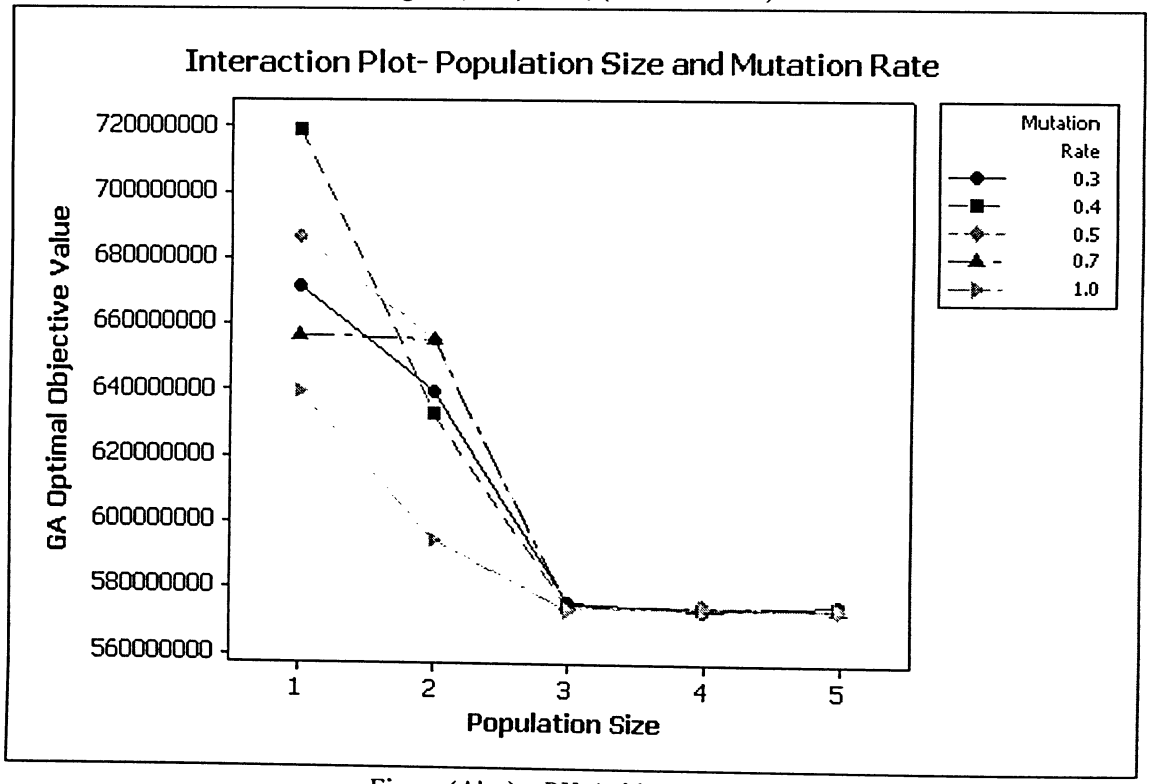

Figure (A'_{M2}) - RV_1 (without $m = 0$)

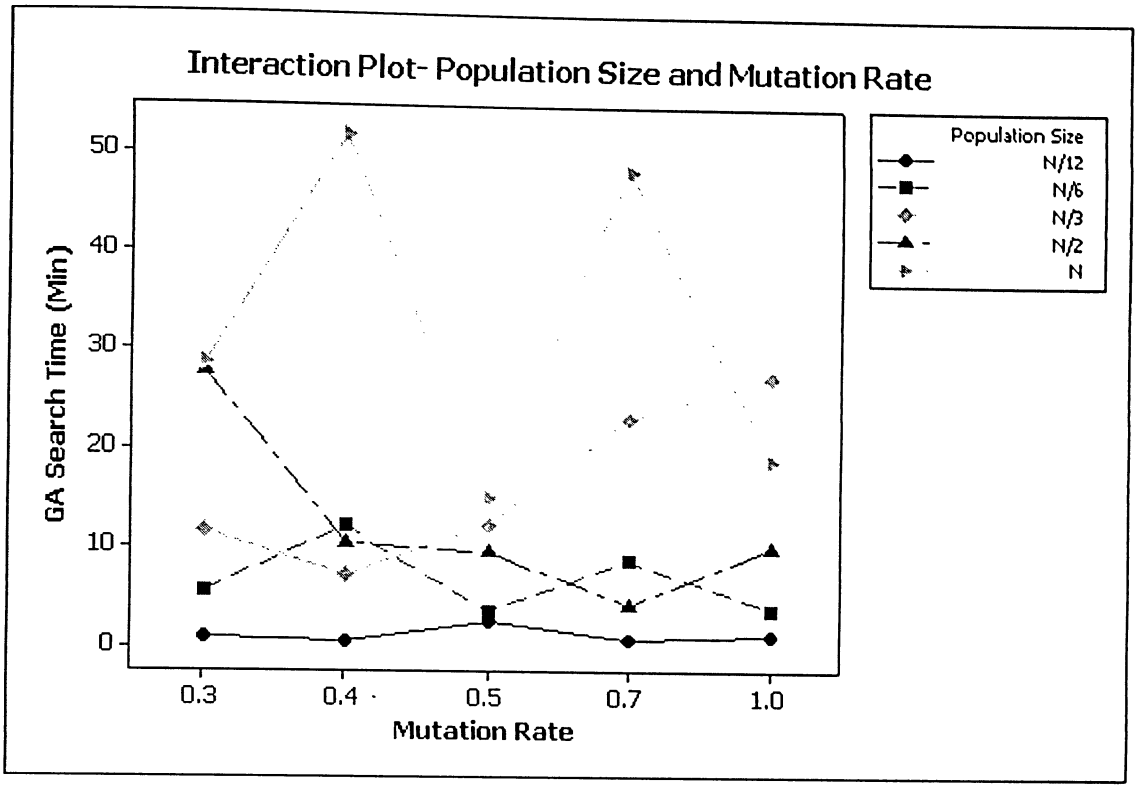

Figure (B'_{M1}) - RV_2 (without $m = 0$)

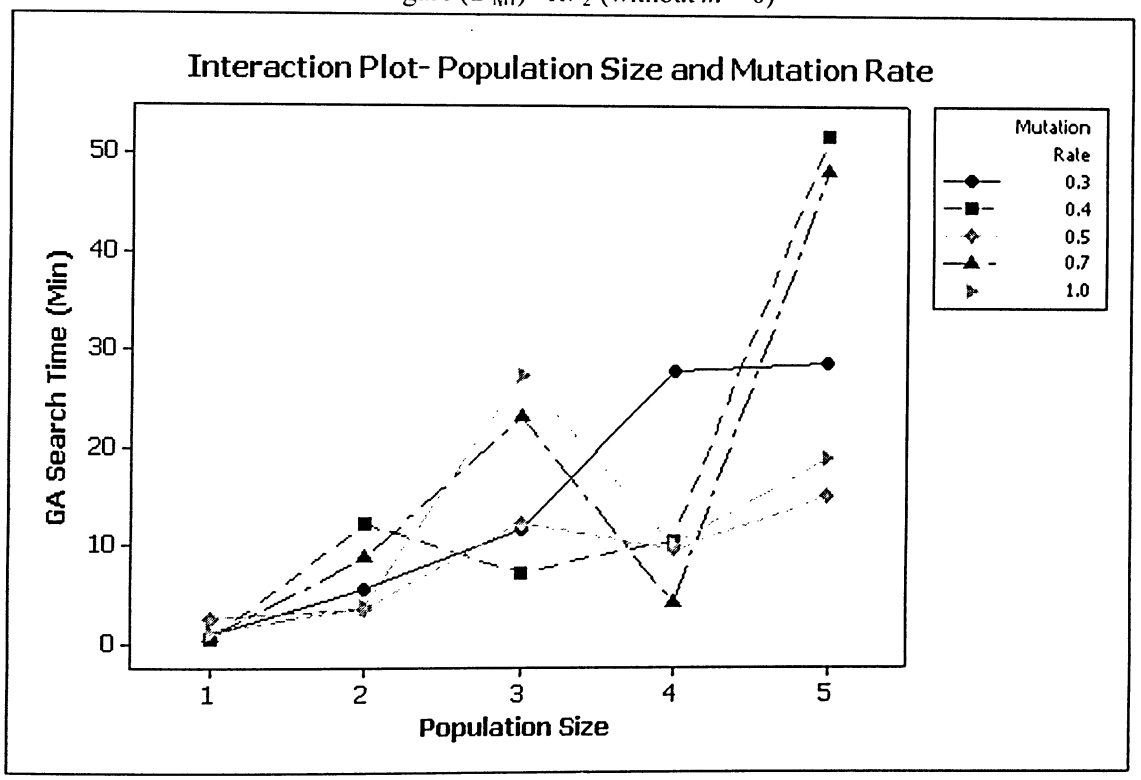

Figure (B'_{M2}) - RV_2 (without $m = 0$)

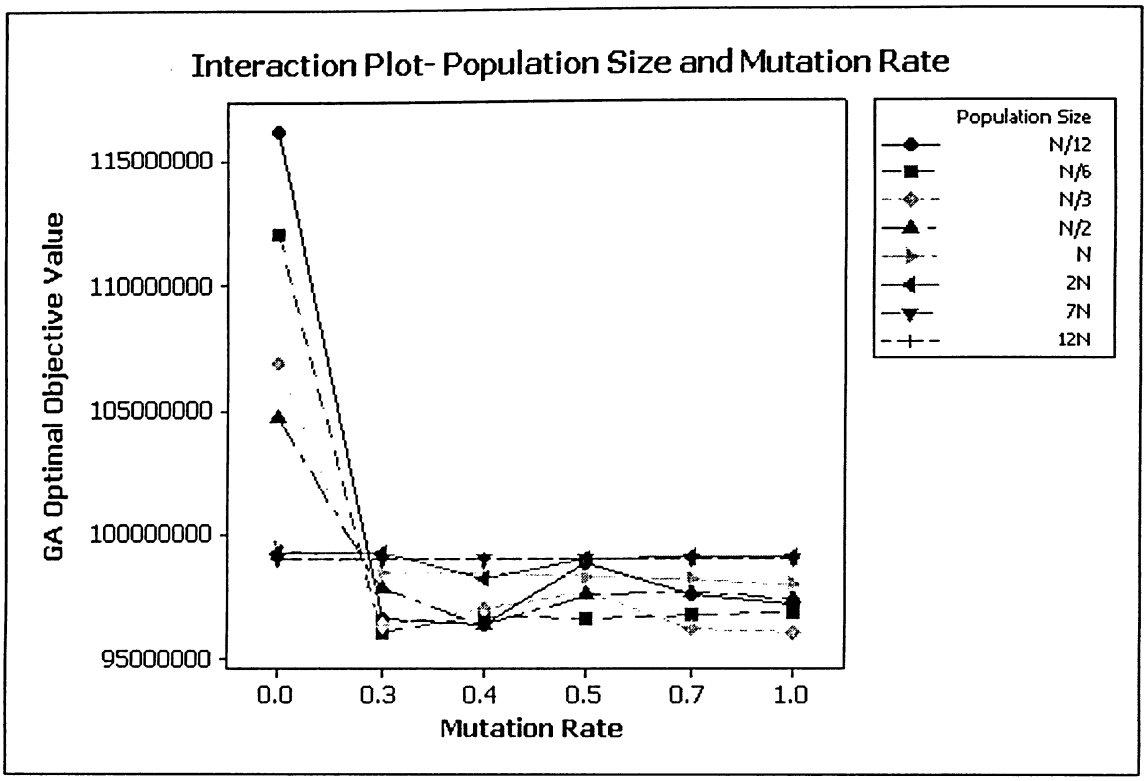

Figure (A_{S1}) - RV_1 (with $m = 0$)

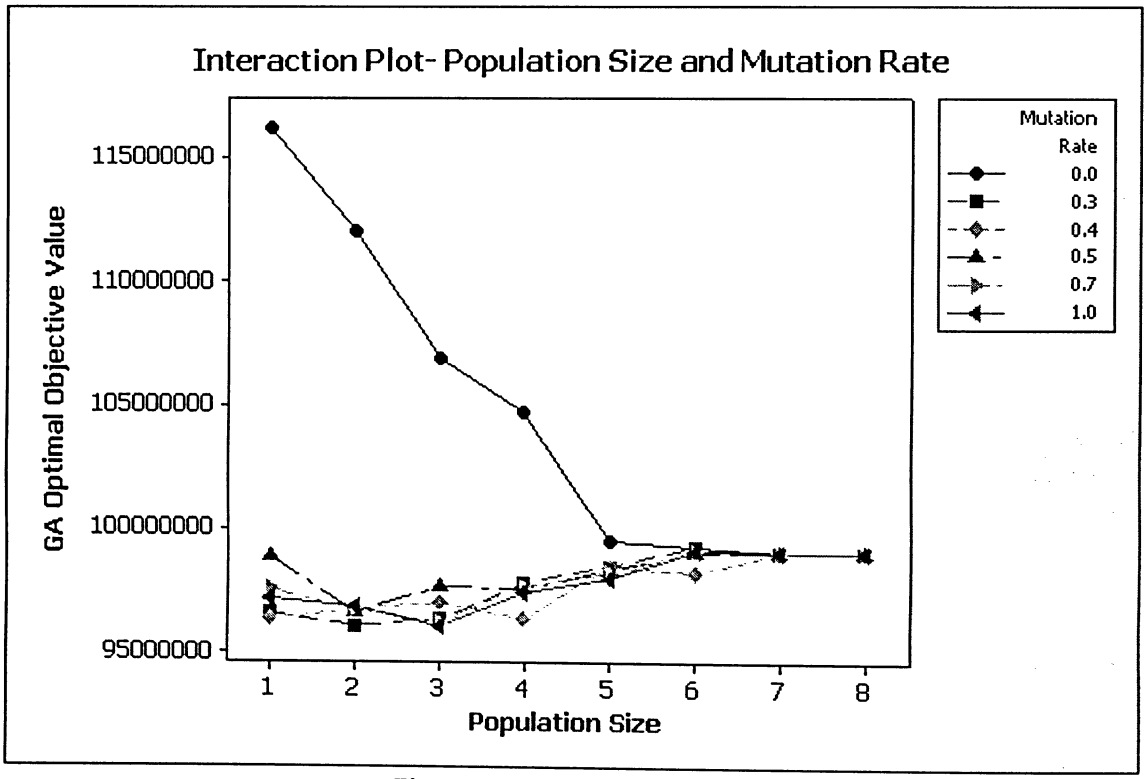

Figure (A_{S2}) - RV_1 (with $m = 0$)

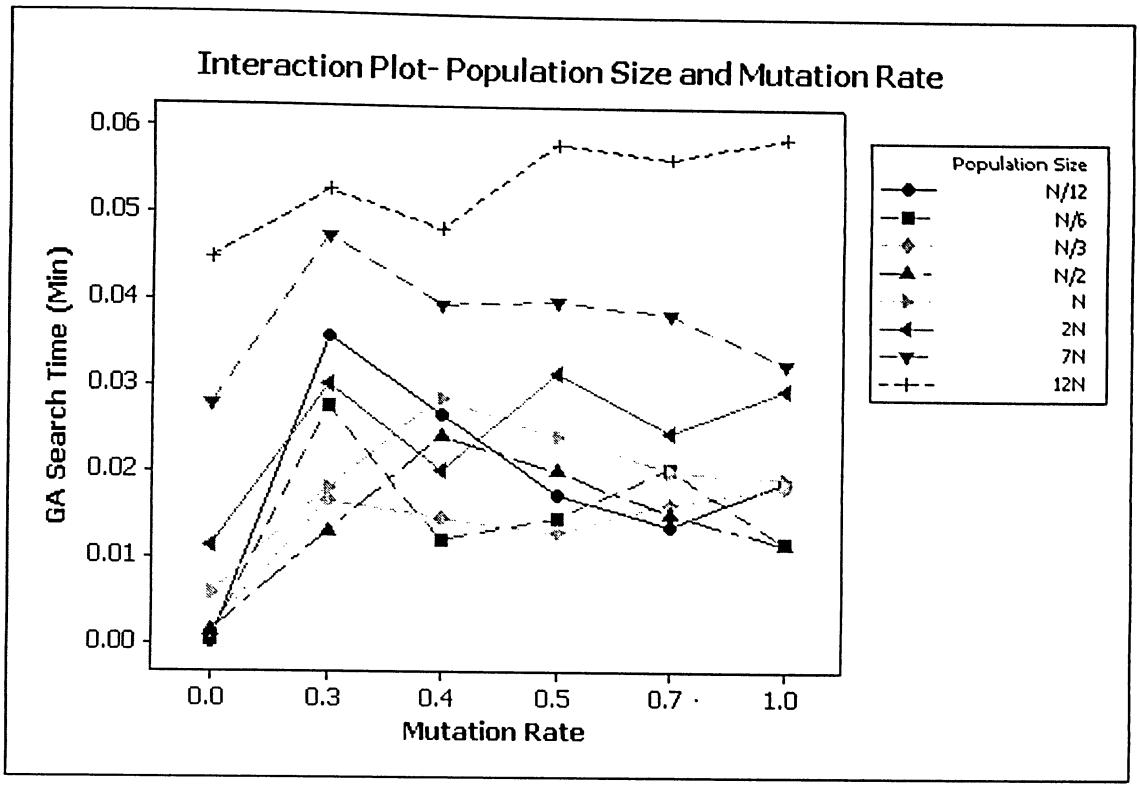

Figure (B_{S1}) - RV_2 (with $m = 0$)

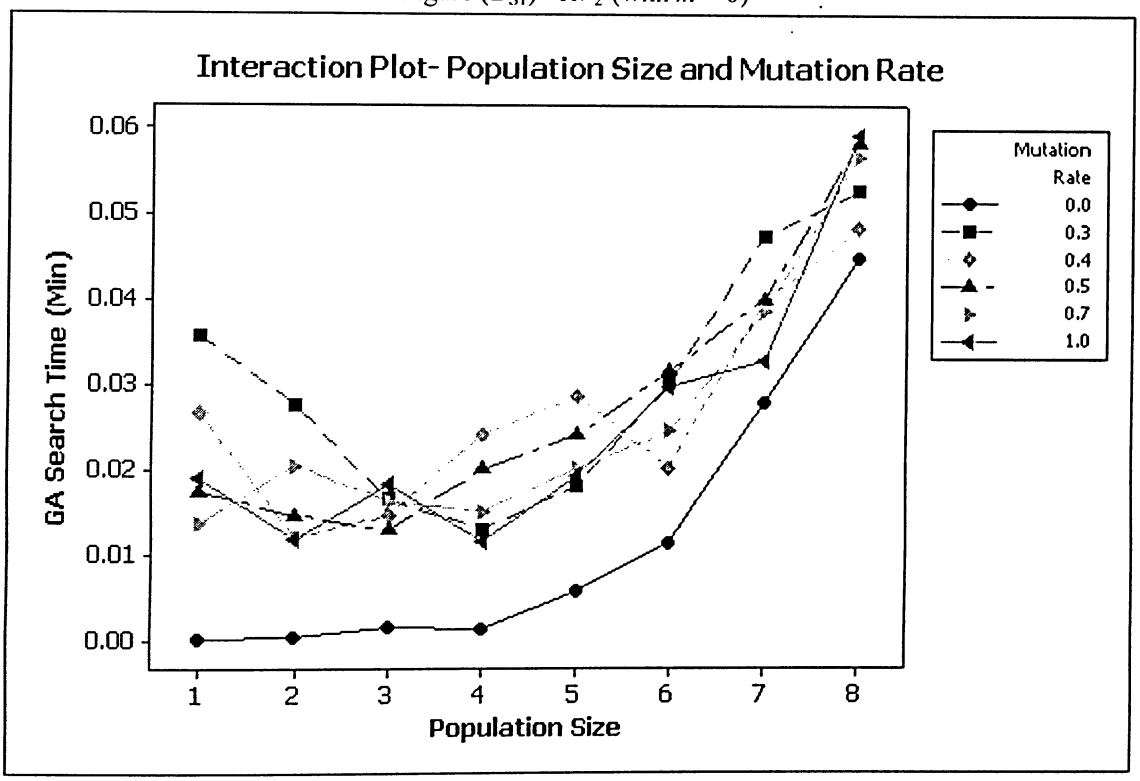

Figure (B_{S2}) - RV_2 (with $m = 0$)

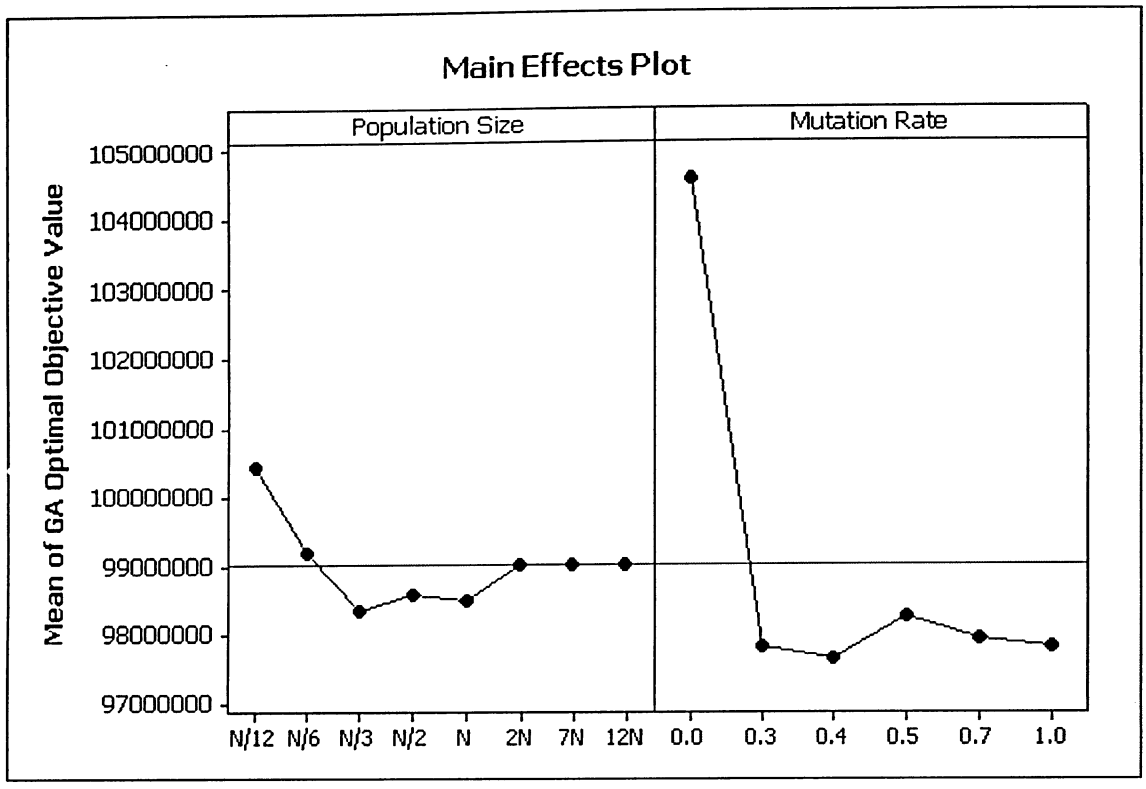

Figure (C_S) - RV_1 (with $m = 0$)

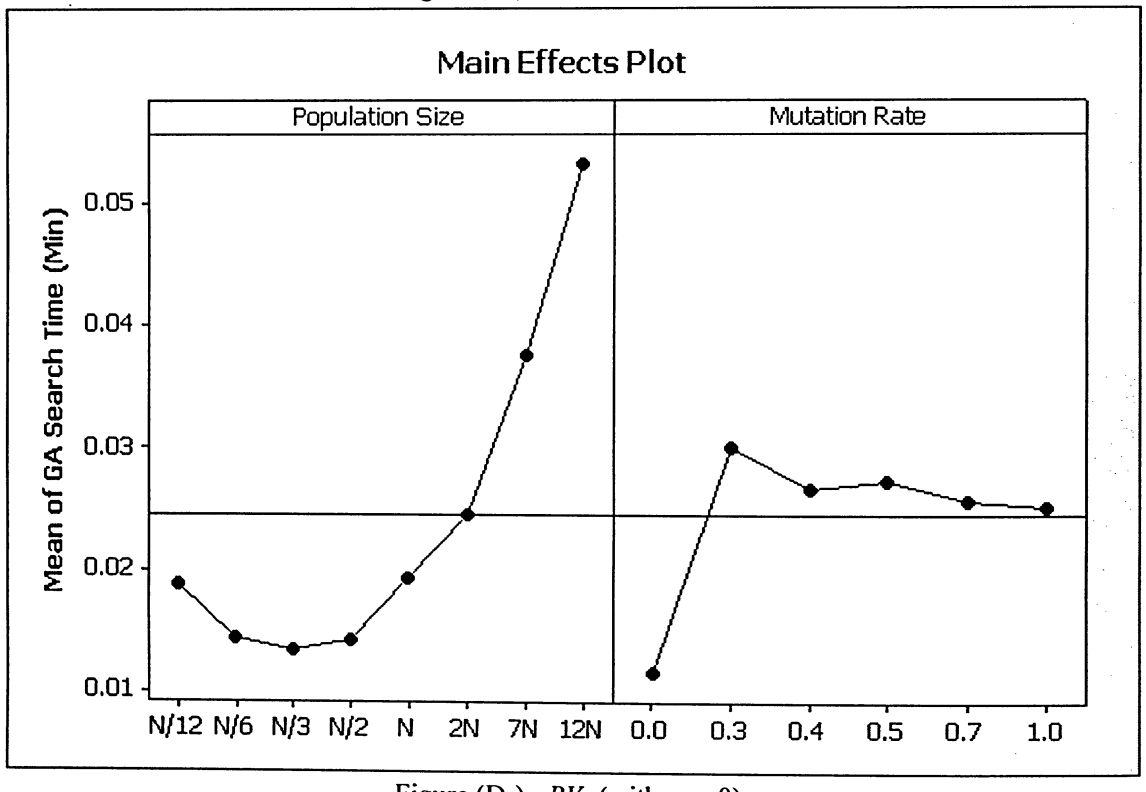

Figure (D_S) - RV_2 (with $m = 0$)

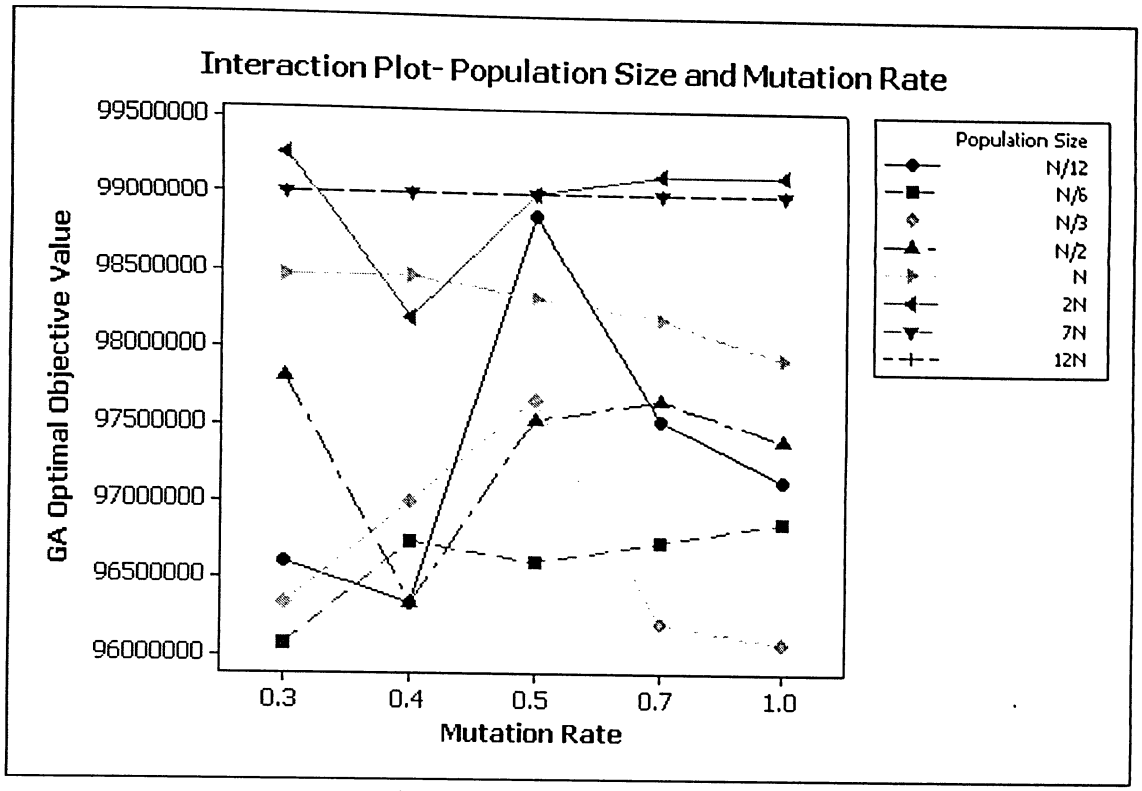

Figure (A'_{S1}) - RV_1 (without $m = 0$)

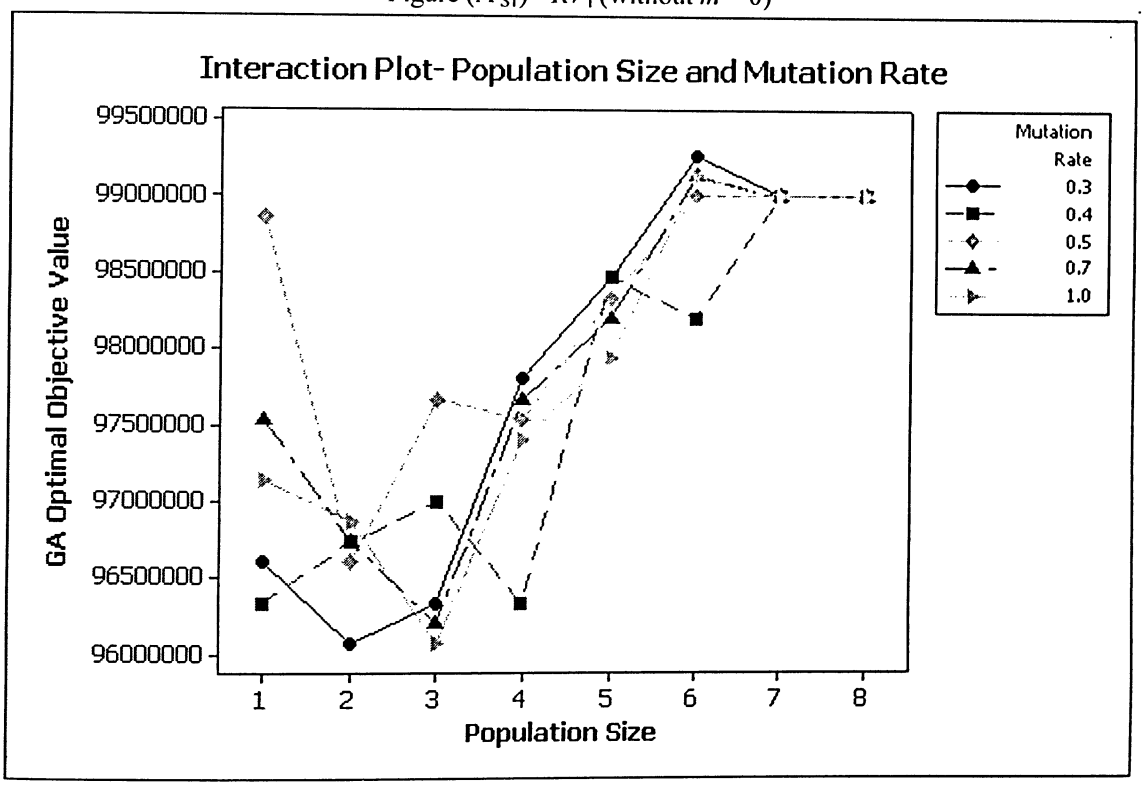

Figure (A'_{S2}) - RV_1 (without $m = 0$)

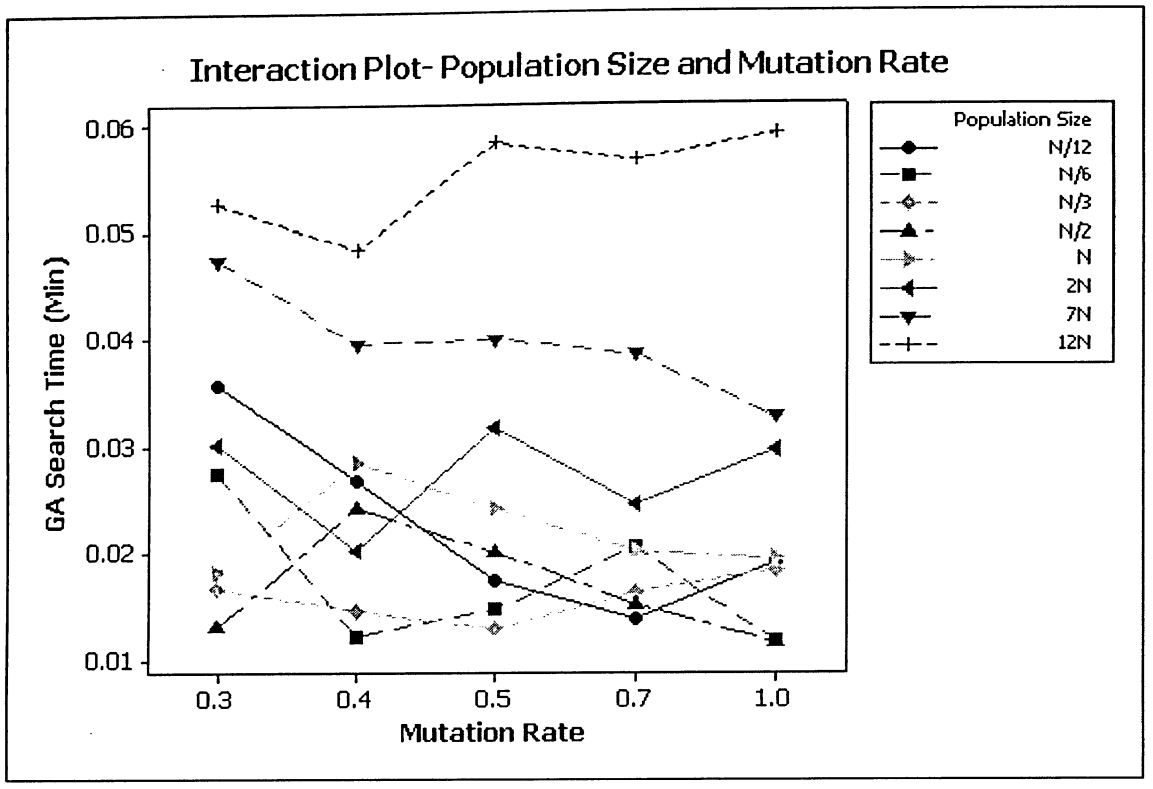

Figure (B'_{S1}) - RV_2 (without $m = 0$)

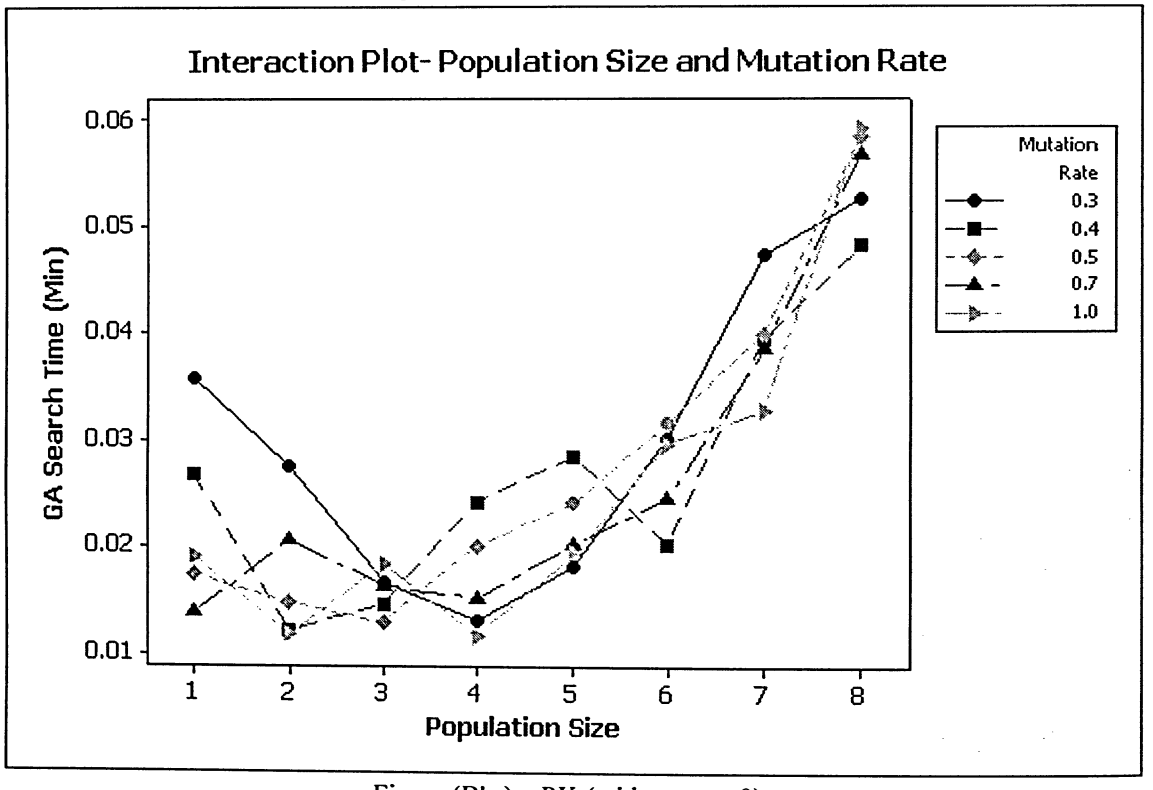

Figure (B'_{S2}) - RV_2 (without $m = 0$)

APPENDIX II: GA AND MOSEL ENVIRONMENT

For this study two computer systems were used for running Mosel™ and GA. Both systems were carefully chosen in order to ensure that the performance characteristics were identical.

Mosel is a modeling and programming language, and is part of the Xpress-MP™ software produced by Dash Optimization (www.dashoptimization.com). Xpress-MP is a suite of mathematical modeling and optimization tools used to solve linear, integer, quadratic, non-linear, and stochastic programming problems.

Mosel hardware and software system specifications:

- Dual Intel Xeon 2.80GHz CPU with 512KB cache
- Linux 2.4 Kernel
- DASH Xpress-MP version 2004e

GA hardware and software system specifications:

- Dual Intel Xeon 2.80GHz CPU with 512KB cache
- Linux 2.4 Kernel
- Sun Java version 1.5

APPENDIX III: JAVA CODE

Included below is a small portion of the java code used to model the proposed GA in this thesis.

Main.java

```
/*
 * Main.java
 *
 * Created on March 14, 2005, 3:31 AM
 */

package geneticalgorithm;

import geneticalgorithm.algorithm.GeneticBase;
import geneticalgorithm.algorithm.GeneticChoice;
import geneticalgorithm.algorithm.rule.EnforceFixedShiftMinHours;
import geneticalgorithm.algorithm.rule.MaxConsecutiveDays;
import geneticalgorithm.algorithm.rule.MaxHoursPerSchedule;
import geneticalgorithm.algorithm.rule.MinRequirementsRule;
import geneticalgorithm.algorithm.rule.OBJSalary;
import geneticalgorithm.algorithm.rule.ObjOptionWeight;
import geneticalgorithm.algorithm.rule.ObjSeniority;
import geneticalgorithm.algorithm.rule.ObjSkillLevelWeight;
import geneticalgorithm.algorithm.rule.Rule;
import geneticalgorithm.algorithm.rule.StrictRequirementsRule;
import geneticalgorithm.algorithm.rule.TotalHoursRule;
import geneticalgorithm.model.Context;
import geneticalgorithm.model.Population;
import geneticalgorithm.model.Schedule;
import geneticalgorithm.util.Random;
import java.util.List;
import java.util.Arrays;
import java.util.Iterator;

/**
 *
 * @author mkhashayardoust
 */
public class Main {

    private static final long LIMIT = 1;

     /** The set of rules to be applied by this algorithm. */
    protected static List rules;

      /** The context for this algorithm. */
     protected static Context context;

    /** Creates a new instance of Main */
    public Main() {
    }

     /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception {
        if ( args.length > 1 ) {
            Random.setSeed(Long.parseLong(args[1]));
        }
        int POPUATION_SIZE = 55;
        double MUTATION_RATE = 0.7;
        GeneticBase genetic = new GeneticChoice(POPUATION_SIZE, MUTATION_RATE);
        genetic.addRule(MaxHoursPerSchedule.RULE);
        genetic.addRule(new MaxConsecutiveDays(5));
        genetic.addRule(TotalHoursRule.RULE);
        genetic.addRule(MinRequirementsRule.RULE);
        genetic.addRule(StrictRequirementsRule.RULE);
        genetic.addRule(EnforceFixedShiftMinHours.RULE);
        genetic.addRule(ObjOptionWeight.RULE);
        genetic.addRule(ObjSkillLevelWeight.RULE);
        genetic.addRule(ObjSeniority.RULE);
```

```
        genetic.addRule(OBJSalary.RULE);
        genetic.readAll(args[0]);
        long start = System.currentTimeMillis();

        Population pop = genetic.getPopulation();
        long[] diffs = new long[16];
        Arrays.fill(diffs, 0);
        int index = 0;
        long movingSum = 0;
        long lastBest = 0L;
        int ITERATION =50000;
        int jloop = 50;
        int j = 0;
        int countZeroMovingSum = 0;
        Schedule best = null;
        for (int i = 0; i <= ITERATION ; i++) {
            pop = genetic.getPopulation(pop);
            best = (Schedule) pop.getSchedules().get(0);
            movingSum -= diffs[index];
            diffs[index] = lastBest - best.getCost();
            lastBest = best.getCost();
            movingSum += diffs[index];
            if (movingSum == 0){
                countZeroMovingSum++;
                if(countZeroMovingSum >49500){
                    System.out.println("countZeroMovingSum is:" + countZeroMovingSum);
                    break;
                }
            } else{
                countZeroMovingSum = 0;
            }

            index = (index + 1) & 0x0f;

            System.out.println("Iteration: " + i
                    + ", Best Schedule Cost: "
                    + best.getCost() + ", Moving sum: " + movingSum
                    + ", index: " + index);

           j++ ;
           if (j == jloop){
            double time = ((double) (System.currentTimeMillis() - start)) / 1000.0;
            System.out.println("Processing time for iteration: "+ i +" is: "+ time + " seconds");
            j = 0;
           }
        }

        genetic.evaluate(best);

        double time = ((double) (System.currentTimeMillis() - start)) / 1000.0;

        System.out.println("Processing time: " + time + " seconds");
        System.out.println("Schedule " + best.getName()
                           + " = " + best.getCost());
        System.out.println("optimal solution is:" + best);
        System.out.println("Iteration: " + ITERATION);
        System.out.println("Population Size :" + POPUATION_SIZE);
        System.out.println("Mutation Rate :" + MUTATION_RATE);

        System.out.println("Rules");
        for (Iterator n = genetic.getRules().iterator(); n.hasNext(); ) {
            Rule rule = (Rule) n.next();
            System.out.println(rule + ": " + rule.getLastResult());
        }
    }
}
```

model/Context.java

```
/*
 * Context.java
 *
 * Created on March 25, 2005, 12:39 PM
 */

package geneticalgorithm.model;

import geneticalgorithm.model.ShiftRegistry;
import java.util.HashMap;
```

```
import java.util.Map;

/**
 *
 * @author mkhashayardoust
 */
public class Context {

    /** The label used for the set of employee details. */
    /** variable LBL_EMPLOYEES store a reference to an object of type label. */
    private static final Label LBL_EMPLOYEES = new Label("employees");

    /** The label used for the set of choices in this context. */
    private static final Label LBL_CHOICES = new Label("choices");

    /** The label used for the set of staff requirements. */
    private static final Label LBL_REQUIREMENTS = new Label("requirements");

    /** The label used for the interval length in this context. */
    private static final Label LBL_INTERVAL_LENGTH = new Label("intLength");

    /** The label used for the shift information in this context. */
    private static final Label LBL_SHIFT_INFORMATION = new Label("shiftinformation");

    /** The label used for the number of days in this context. */
    private static final Label LBL_NUMBER_OF_DAYS = new Label("numDays");

    /** The internal map of all context values. */
    private Map values;

    /**
     *default constructor
     */
    public Context(){
    }

    /**
     * Creates a new context.
     *
     * @param days the number of days in the schedule associated with this
     * context
     * @param intervalLength the interval length in this context
     */
    public Context(int days, int intervalLength) {
        this(new Integer(days), new Integer(intervalLength));
    }

    /**
     * Creates a new context.
     *
     * @param days the number of days in the schedule associated with this
     * context
     * @param intervalLength the interval length in this context
     */
    public Context(Integer days, Integer intervalLength) {
        values = new HashMap();
        values.put(LBL_INTERVAL_LENGTH, intervalLength);
        values.put(LBL_NUMBER_OF_DAYS, days);
        values.put(LBL_EMPLOYEES, new EmployeeRegistry(getIntervalLength()));
        values.put(LBL_CHOICES, new ChoiceRegistry());
        values.put(LBL_SHIFT_INFORMATION, new ShiftRegistry());
        values.put(LBL_REQUIREMENTS,
                new RequirementRegistry(days.intValue(),
                                intervalLength.intValue()));
    }

    /**
     * Get a specified value from this context.
     *
     * @param name the name of the value to be retrieved
     *
     * @return the specified value, or <code>null</code> if there is no value
     * associated with the specified name in this context
     */
    public Object get(String name) {
        return values.get(name);
    }

    /**
```

```
     * Set a specified value in this context.
     *
     * @param name the name of the value to be stored
     * @param value the value to be associated with the specified name in this
     * context
     *
     * @return the old value associated with the specified name, or
     * <code>null</code> if no value was previously defined
     */
    public Object set(String name, Object value) {
        return this.values.put(name, value);
    }

    /**
     * Get the interval length for this context.
     *
     * @return the interval length in minutes
     */
    public int getIntervalLength() {
        return ((Number) this.values.get(LBL_INTERVAL_LENGTH)).intValue();
    }

    /**
     * Get the number of days for the schedule associated with this context.
     *
     * @return the number of days
     */
    public int getNumberOfDays() {
        return ((Number) this.values.get(LBL_NUMBER_OF_DAYS)).intValue();
    }

    /**
     * Get the set of all employees known in this context.
     *
     * @return the set of employees
     */
    public EmployeeRegistry getEmployees() {
        return (EmployeeRegistry) this.values.get(LBL_EMPLOYEES);
    }

    /**
     * Get the set of all choices known in this context.
     *
     * @return the set of choices
     */
    public ChoiceRegistry getChoices() {
        ChoiceRegistry chr = (ChoiceRegistry) this.values.get(LBL_CHOICES);
        return  chr;
    }

    /**
     * Get the set of all shift information known in this context.
     *
     * @return the set of shift information
     */
    public ShiftRegistry getShiftInformation() {
        return (ShiftRegistry) this.values.get(LBL_SHIFT_INFORMATION);
    }

    /**
     * Get the set of all requirements known in this context.
     *
     * @return the set of requirements
     */
    public RequirementRegistry getRequirements() {
        return (RequirementRegistry) this.values.get(LBL_REQUIREMENTS);
    }

    public String toString() {
        return "Requirements:" + this.values.get(LBL_REQUIREMENTS);
    }
    /**
     * Private internal class used to keep internally defined context attributes
     * in a separate namespace from user-defined attributes.
     */
    private static class Label {
        private String name;

        public Label(String name) {
            this.name = name;
```

```
        }

        public String toString() {
            return this.name;
        }

        public int hashCode() {
            return this.name.hashCode() + 1;
        }

        public boolean equals(Object obj) {
            return this == obj;
        }
    }
}
```

model/Population.java

```
/*
 * Population.java
 *
 * Created on March 19, 2005, 7:25 AM
 */

package geneticalgorithm.model;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;


/**
 *
 * @author mkhashayardoust
 */
public class Population {

    /** The list of schedules in this population. */
    private List schedules;

    /** An unmodifiable list for external use. */
    private List external;

    /** The number of days represented in this population. */
    private int days;

    /**
     * Creates a new population
     */
    public Population() {
        schedules = new ArrayList();
        external = Collections.unmodifiableList(schedules);
        days = 0;
    }

    /**
     * Create a new population that is a copy of an existing population.
     *
     * @param population the population to be copied
     */
    public Population(Population population) {
        this();
        for (Iterator n = population.getSchedules().iterator(); n.hasNext(); ) {
            addSchedule((Schedule) n.next());
        }
    }

    /**
     * Get the list of schedules in this population.
     *
     * @return the list of schedules
     */
    public List getSchedules() {
        return external;
    }

    /**
     * Get the size of this population.
```

```
     *
     * @return the number of schedules in this population
     */
    public int getSize() {
        return schedules.size();
    }

    /**
     * Get the number of days represented in this population. This will be one
     * greater than the highest day number found.
     *
     * @return the number of days represented in this population
     */
    public int getNumberOfDays() {
        return days;
    }

    /**
     * Add a schedule to this population.
     *
     * @param schedule the schedule to be added
     */
    public void addSchedule(Schedule schedule) {
        schedules.add(schedule);

        if (schedule.getNumberOfDays() > days) {
            days = schedule.getNumberOfDays();
        }
    }

    /**
     * Get a schedule from this population.
     *
     * @param index the schedule in this population
     *
     * @return the requested schedule
     */
    public Schedule getSchedule(int index) {
        return (Schedule) schedules.get(index);
    }

    /**
     * Truncate this population to the specified size.
     *
     * <p>The population will first be sorted in order of cost, and only the
     * lowest cost schedules will be retained after truncation. If the
     * population size is already equal to or less than the target size, this
     * method will have no effect.</p>
     *
     * @param size the target size of the population
     */
    public void truncate(int size) {
        if ( schedules.size() > size ) {
            Collections.sort(schedules);
            while (schedules.size() > size) {
                schedules.remove(schedules.size() - 1);
            }
        }
    }

    /**
     * Return a string representation of this population.
     */
    public String toString() {
        String lineSep = System.getProperty("line.separator", "/n");
        StringBuffer buf = new StringBuffer();
        int len = schedules.size();
        for (int i = 0; i < len; i++) {
            buf.append("Schedule: ").append(i).append(lineSep)
               .append(schedules.get(i)).append(lineSep);
        }
        return buf.toString();
    }
}
```

algorithm/GeneticBase.java

```
/*
 * GeneticBase.java
```

```
 *
 * Created on March 14, 2005, 9:22 AM
 */

package geneticalgorithm.algorithm;

import geneticalgorithm.algorithm.rule.Rule;
import geneticalgorithm.bean.Choice;
import geneticalgorithm.bean.Employee;
import geneticalgorithm.bean.ShiftInformation;
import geneticalgorithm.model.ChoiceRegistry;
import geneticalgorithm.model.ShiftRegistry;
import geneticalgorithm.model.Context;
import geneticalgorithm.model.EmployeeRegistry;
import geneticalgorithm.model.Population;
import geneticalgorithm.model.Schedule;
import geneticalgorithm.util.Random;
import java.io.IOException;
import java.io.Reader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;



/**
 *
 * @author mkhashayardoust
 */
public abstract class GeneticBase {

    /**
     * Add zero-length shifts to a registry.
     *
     * @param the registry to have zero-length shifts added
     */
    public static void addZeroLengthShifts(ChoiceRegistry choices,
                                           Context context) {
        ShiftRegistry shifts = context.getShiftInformation();
        int maxDays = choices.getNumberOfDays();
        EmployeeRegistry employees = context.getEmployees();
        for (Iterator n = choices.getEmployees().iterator(); n.hasNext(); ) {
            Integer empId = (Integer) n.next();
            Employee employee = employees.getEmployee(empId);
           for (int day = 0; day < maxDays; day++) {
                Choice choice = new Choice();
                choice.setEmployee(empId.intValue());
                choice.getShift().setLength(0);
                choice.getShift().setStartInterval(1);
                choice.getShift().setDay(day);
                choices.addChoice(choice);
                ShiftInformation shift = new ShiftInformation(choice.getShift(), day, 1, 0, 0, 0);
                shifts.addShiftInformation(shift);
            }
        }
    }

    /** The default interval length to be used. */
    private static final int DEFAULT_INTERVAL_LENGTH = 15;

    /** The default schedule length to be used. */
    private static final int DEFAULT_SCHEDULE_LENGTH = 7;

    /** The context for this algorithm. */
    protected Context context;

    /** The size of the population to be used by this algorithm. */
    protected int size;

    /** The number of days per schedule. */
    protected int scheduleLength;

    /** The set of rules to be applied by this algorithm. */
    protected List rules;
    protected List publicRules;

    /**
     * Creates a new instance of <code>GeneticBase</code>.
     *
```

```
 * @param populationSize the size of the populations to be generated by this
 * algorithm
 */

public GeneticBase(){
}

public GeneticBase(int populationSize) {
    this(populationSize, DEFAULT_SCHEDULE_LENGTH, DEFAULT_INTERVAL_LENGTH);
}

/**
 * Creates a new instance of <code>GeneticBase</code>.
 *
 * @param populationSize the size of the populations to be generated by this
 * algorithm
 * @param intervalLength the interval length in minutes to be used by this
 * algorithm
 */
public GeneticBase(int populationSize,
                   int days,
                   int intervalLength) {
    context = new Context(days, intervalLength);
    size = populationSize;
    scheduleLength = DEFAULT_SCHEDULE_LENGTH;
    rules = new ArrayList();
    publicRules = Collections.unmodifiableList(rules);
}

/**
 * Add a rule to this algorithm.
 *
 * @param rule the rule to be added
 */
public void addRule(Rule rule) {
    rules.add(rule);
}

/**
 * Get the current list of rules.
 *
 * @return the list of rules
 */
public List getRules() {
    return publicRules;
}

/**
 * Return the context for this genetic algorithm instance.
 */
public Context getContext() {
    return this.context;
}

/**
 * Read all files into this genetic algorithm for processing.
 *
 * @param prefix the filename prefix for the set of files to be loaded
 *
 * @throws IOException if there is an error reading from the specified
 * stream
 */
public void readAll(String prefix) throws IOException {
    readEmployees(prefix + "TmpSchd_Employees.txt");
    readChoices(prefix + "TmpSchd_Choices.txt");
    readRequirements(prefix + "TmpSchd_StaffRequirements.txt");
    readShiftInformation(prefix + "TmpSchd_ShiftInfo.txt");
    System.out.println("file is:" + prefix);
}

/**
 * Read the choices from the specified stream.
 *
 * @param rdr the stream from which to read the choices for this algorithm
 *
 * @throws IOException if there is an error reading from the specified
 * stream
 */
public void readChoices(Reader rdr) throws IOException {
    context.getChoices().readChoices(rdr);
```

```
}

/**
 * Read the choices from the specified file.
 *
 * @param filename the name of the file from which the choices will be
 * read
 *
 * @throws IOException if there is an error reading from the specified
 * file
 */
public void readChoices(String filename) throws IOException {
    context.getChoices().readChoices(filename);

}

/**
 * Read the shift information from the specified file.
 *
 * @param filename the name of the file from which the choices will be
 * read
 *
 * @throws IOException if there is an error reading from the specified
 * file
 */
public void readShiftInformation(String filename) throws IOException {
    context.getShiftInformation().readShifts(filename);
}

/**
 * Read the employees from the specified stream.
 *
 * @param rdr the stream from which to read the employees for this algorithm
 *
 * @throws IOException if there is an error reading from the specified
 * stream
 */
public void readEmployees(Reader rdr) throws IOException {
    context.getEmployees().readEmployees(rdr);
}

/**
 * Read the employees from the specified file.
 *
 * @param filename the name of the file from which the employees will be
 * read
 *
 * @throws IOException if there is an error reading from the specified
 * file
 */
public void readEmployees(String filename) throws IOException {
    context.getEmployees().readEmployees(filename);
}

/**
 * Read the requirements from the specified stream.
 *
 * @param rdr the stream from which to read the requirements for this
 * algorithm
 *
 * @throws IOException if there is an error reading from the specified
 * stream
 */
public void readRequirements(Reader rdr) throws IOException {
    context.getRequirements().readRequirements(rdr);
}

/**
 * Read the requirements from the specified file.
 *
 * @param filename the name of the file from which the requirements will be
 * read
 *
 * @throws IOException if there is an error reading from the specified
 * file
 */
public void readRequirements(String filename) throws IOException {
    context.getRequirements().readRequirements(filename);
}
```

```
/**
 * Add the specified schedule to the specified population.
 *
 * <p>This method will also evaluate the cost of the schedule. Schedules
 * with a cost of <code>Long.MAX_VALUE</code> indicate a will be discarded
 * and not added. Other schedules will have their cost set according to the
 * calculation done by this algorithm.
 *
 * @param population the population to which the schedule is to be added
 * @param schedule the schedule to be added
 */
protected void addSchedule(Population population, Schedule schedule) {
    long cost = evaluate(schedule);
    if (cost != Long.MAX_VALUE) {
        schedule.setCost(cost);
        population.addSchedule(schedule);
    } else {
        System.out.println("schedule rejected: count    "
                           + population.getSize());
    }
}

/**
 * Get the initial population for this algorithm.
 *
 * @return the initial population
 */
public Population getPopulation() {
    Population population = new Population();
    ChoiceRegistry base = context.getChoices();
    addZeroLengthShifts(base,context);
    ChoiceRegistry temp = new ChoiceRegistry(base);

    while (population.getSize() < size) {
        Schedule schedule = new Schedule(
                Integer.toString(population.getSize()));
        for (Iterator n = context.getEmployees().getEmployees().iterator();
                n.hasNext(); ) {
            Employee emp = (Employee) n.next();
            int empId = emp.getEmployeeId();
            for (int day = 0; day < scheduleLength; day++) {
                List choiceList = temp.getChoices(empId, day);
                if (choiceList.isEmpty()) {
                    choiceList = base.getChoices(empId, day);
                    if (choiceList.isEmpty()){
                        continue;
                    }
                    temp.addChoices(choiceList);
                }
            Choice choice = (Choice) choiceList.get(Random.nextInt(
                    choiceList.size()));
            schedule.addChoice(choice);
            temp.removeChoice(choice);
            }
        }
        addSchedule(population, schedule);
    }
    return population;
}

/**
 * Evaluate the cost of the specified schedule according to the rules used
 * by this algorithm.
 *
 * @param schedule the schedule to be evaluated
 */
  public long evaluate(Schedule schedule) {
    int len = rules.size();
    long result = 0L;
    for (int i=0; i < len; i++) {
        Rule rule = (Rule) rules.get(i);
        long tmp = rule.evaluate(schedule, context);
        if (tmp == Long.MAX_VALUE) {
            if (rule.repair(schedule, context)) {
                result = 0L;
                tmp = 0;
                i = -1;
            } else {
                return tmp;
            }
```

```
            }
            result += tmp;
        }
        return result;
    }

    /**
     * Get the next generation population for this algorithm.
     *
     * @param population the current generation
     *
     * @return the next generation population
     */
    public abstract Population getPopulation(Population population);
}
```

algorithm/GeneticV1.java

```
/*
 * GeneticV1.java
 *
 * Created on March 19, 2005, 8:01 AM
 */

package geneticalgorithm.algorithm;

import geneticalgorithm.bean.Choice;
import geneticalgorithm.model.ChoiceRegistry;
import geneticalgorithm.model.Context;
import geneticalgorithm.model.CrossoverDetail;
import geneticalgorithm.model.Population;
import geneticalgorithm.model.Schedule;
import geneticalgorithm.util.Random;
import java.util.List;

/**
 *
 * @author mkhashayardoust
 */
public class GeneticV1 extends GeneticBase {

    /** The attribute prefix used to store the crossover data for each day. */
    public static final String ATTRIBUTE_CROSSOVER = "crossover";

    /**
     * The attribute prefix used to store the primary parent schedule of a new
     * schedule.
     */
    public static final String ATTRIBUTE_MOTHER = "mother";

    /**
     * The attribute prefix used to store the secondary parent schedule of a new
     * schedule.
     */
    public static final String ATTRIBUTE_FATHER = "father";

    /**
     * The list of property names used in the crossovers performed by this
     * algorithm.
     */
    private String properties = "startInterval,length";

    private static final int MUTATION_BASE = 100000;

    /** A flag indicating whether or not to store meta-data information in the
     * schedules created by this algorithm.
     */
    private boolean debug = true;

    /** A serial counter used to generate schedule names. */
    int serialName = 0;

    /** The threshold level for causing mutation in this genetic alorithm. */
    private int mutationThreshold;

    /**
     * Creates a new instance of <code>GeneticBase</code>.
     *
     * @param populationSize the size of the populations to be generated by this
```

```
 * algorithm
 */
public GeneticV1(int populationSize) {
    this(populationSize, 0.0);
}

/**
 * Creates a new instance of <code>GeneticBase</code>.
 *
 * @param populationSize the size of the populations to be generated by this
 * algorithm
 * @param mutationRate the rate at which mutations will occur on each
 * generation
 */
public GeneticV1(int populationSize, double mutationRate) {
    super(populationSize);
    this.mutationThreshold = (int) (mutationRate * MUTATION_BASE);
}

/**
 * Creates a new instance of <code>GeneticBase</code>.
 *
 * @param populationSize the size of the populations to be generated by this
 * algorithm
 * @param days the number of days per schedule for this algorithm
 * @param intervalLength the interval length in minutes to be used by this
 * algorithm
 */
public GeneticV1(int populationSize, int days, int intervalLength) {
    super(populationSize, days, intervalLength);
}

/**
 * Double the size of the existing population by doing a crossover
 * operation.
 *
 * @param population the starting population
 *
 * @return a list of the original population's schedules plus the results
 * of the crossover operation
 */
protected void crossover(Population population) {
    List schedules = population.getSchedules();
    int size = population.getSize();
    int max = size << 1;

    while (population.getSize() < max) {
        crossover(population,
                  (Schedule) schedules.get(Random.nextInt(size)),
                  (Schedule) schedules.get(Random.nextInt(size)) );
    }
}

protected Schedule createSchedule(Schedule mum, Schedule dad) {
    Schedule result = new Schedule(Integer.toString(serialName++), mum);
    return result;
}

/**
 * Produce a new schedule by performing a crossover between two existing
 * schedules.
 *
 * @param population the population that is being grown by the crossover
 * operation
 * @param mum one of the parent schedules
 * @param dad the other parent schedule
 */
protected void crossover(Population population, Schedule mum, Schedule dad) {
    assert(mum.getNumberOfDays() == dad.getNumberOfDays());
    Schedule kid1 = createSchedule(mum, dad);
    Schedule kid2 = createSchedule(dad, mum);
    Integer[] mumEmps = mum.getEmployeeIds();
    Integer[] dadEmps = dad.getEmployeeIds();

    for (int day = mum.getNumberOfDays() - 1; day >= 0; day--) {
        Integer emp1 = mumEmps[Random.nextInt(mumEmps.length)];
        Integer emp2 = dadEmps[Random.nextInt(dadEmps.length)];

        crossover(kid1, emp1, day, kid2, emp2, day);
```

```
        if (debug) {
            String key = ATTRIBUTE_CROSSOVER + '.' + day;
            kid1.setAttribute(key, new CrossoverDetail(mum.getName(),
                                                       dad.getName(),
                                                       day, emp2.intValue(),
                                                       day, emp1.intValue(),
                                                       properties));
            kid2.setAttribute(key, new CrossoverDetail(dad.getName(),
                                                       mum.getName(),
                                                       day, emp1.intValue(),
                                                       day, emp2.intValue(),
                                                       properties));
        }
    }

    if (Random.nextInt(MUTATION_BASE) < mutationThreshold) {
        mutate(kid1, context);
    }

    if (Random.nextInt(MUTATION_BASE) < mutationThreshold) {
        mutate(kid2, context);
    }

    addSchedule(population, kid1);
    addSchedule(population, kid2);
}

/**
 * Cause a mutation in the specified schedule.
 *
 * @param schedule the schedule to mutate
 * @param context the context of the current algorithm
 */
protected void mutate(Schedule schedule, Context context) {
  ChoiceRegistry choices = context.getChoices();
   int maxDays = choices.getNumberOfDays();
   Integer[] empIds = schedule.getEmployeeIds();
   Integer emp = empIds[Random.nextInt(empIds.length)];
   int day = Random.nextInt(maxDays);
   Choice choice= (Choice) schedule.getChoices(emp, day).get(0);
   schedule.removeChoice(choice);
   List listChoice = context.getChoices().getChoices(emp, day);
   choice = (Choice) listChoice.get(Random.nextInt(listChoice.size()));
   schedule.addChoice(choice);
}

/**
 * Swap the required properties of the two choices, as required by the
 * crossover operation.
 *
 * @param kid1 the first schedule involved in the crossover
 * @param empId1 the employee for which details are to be swapped in the
 * first schedule
 * @param day1 the day on which details are taken from the first schedule
 * @param kid2 the second schedule involved in the crossover
 * @param empId2 the employee for which details are to be swapped in the
 * second schedule
 * @param day2 the day on which details are taken from the second schedule
 */
protected void crossover(Schedule kid1, Integer empId1, int day1,
                         Schedule kid2, Integer empId2, int day2) {
    Choice src1 = (Choice) kid1.getChoices(empId1, day1).get(0);
    Choice src2 = (Choice) kid2.getChoices(empId2, day2).get(0);

    Choice dest1 = new Choice(src1);
    Choice dest2 = new Choice(src2);

    dest1.getShift().setLength(src2.getLength());
    dest1.getShift().setStartInterval(src2.getStartInterval());
    dest2.getShift().setLength(src1.getLength());
    dest2.getShift().setStartInterval(src1.getStartInterval());

    kid1.removeChoice(src1);
    kid1.addChoice(dest1);
    kid2.removeChoice(src2);
    kid2.addChoice(dest2);
}

public Population getPopulation(Population population) {
    Population result = new Population(population);
```

```
        crossover(result);
        result.truncate(population.getSize());
        return result;
    }
}
```

model/CrossoverDetail.java

```
/*
 * CrossoverDetail.java
 *
 * Created on March 24, 2005, 2:26 PM
 */

package geneticalgorithm.model;

/**
 *
 * @author mkhashayardoust
 */
public class CrossoverDetail {

    /** The name of the first parent schedule used in the crossover. */
    private String mum;

    /** The name of the second parent schedule used in the crossover. */
    private String dad;

    /** The day of the schedule from which the crossover data was obtained. */
    private int fromDay;

    /** The day of the schedule to which the crossover data was applied. */
    private int toDay;

    /** The numeric id of the employee to which the crossover was applied. */
    private int toEmpId;

    /**
     * The numeric id of the employee from which the crossover data was
     * obtained.
     */
    private int fromEmpId;

    /**
     * A comma-separated list of property names that were involved in the
     * exchange described by this instance.
     */
    private String properties;

    /**
     * Creates new crossover details.
     * @param mother the name of the primary parent schedule
     * @param father the name of the secondary parent schedule
     * @param fromDay the day of the schedule from which the crossover data
     * was obtained
     * @param fromEmpId the numeric id of the employee from which the crossover
     * data was obtained
     * @param toDay the day of the schedule to which the crossover data was
     * applied
     * @param toEmpId the numeric id of the employee to which the crossover
     * data was applied
     * @param properties a comma-separated list of property names that were
     * exchanged in the crossover
     */
    public CrossoverDetail(String mother, String father,
                           int fromDay, int fromEmpId,
                           int toDay, int toEmpId,
                           String properties) {
        this.mum = mother;
        this.dad = father;
        this.fromDay = fromDay;
        this.fromEmpId = fromEmpId;
        this.toDay = toDay;
        this.toEmpId = toEmpId;
        this.properties = properties;
    }

    /**
     * Get the name of the primary parent schedule involved in this crossover.
```

```
     *
     * @return the parent schedule name
     */
    public String getMother() {
        return mum;
    }

    /**
     * Get the name of the secondary parent schedule involved in this crossover.
     *
     * @return the parent schedule name
     */
    public String getFather() {
        return dad;
    }

    /**
     * Get the day of the schedule from which the crossover data was obtained.
     */
    public int getFromDay() {
        return fromDay;
    }

    /**
     * Get the numeric id of the employee from which the crossover data was
     * obtained.
     */
    public int getFromEmployee() {
        return fromEmpId;
    }

    /**
     * Get the day of the schedule to which the crossover data was applied.
     */
    public int getToDay() {
        return toDay;
    }

    /**
     * Get the numeric id of the employee to which the crossover data was
     * applied.
     */
    public int getToEmployee() {
        return toEmpId;
    }

    /**
     * Get the list of properties involved in the exchange.
     *
     * @param a comma-separated list of property names that were exchanged in
     * the crossover operation described by this object
     */
    public String getProperties() {
        return properties;
    }

    /**
     * Return a string representation of this instance.
     */
    public String toString() {
        return mum + '/' + dad + ':'
             + fromDay + ',' + fromEmpId
             + '-' + toDay + ',' + toEmpId;
    }
}
```

algorithm/GeneticChoice.java

```
/*
 * GeneticChoice.java
 *
 * Created on June 7, 2005, 6:13 PM
 */

package geneticalgorithm.algorithm;

import geneticalgorithm.bean.Choice;
import geneticalgorithm.model.Schedule;
```

```
/**
 *
 * @author mkhashayardoust
 */
public class GeneticChoice extends GeneticV1{

    /**
     * Creates a new instance of <code>GeneticChoice</code>.
     *
     * @param populationSize the size of the populations to be generated by this
     * algorithm
     */
    public GeneticChoice(int populationSize) {
        super(populationSize);
    }


    /**
     * Creates a new instance of <code>GeneticChoice</code>.
     *
     * @param populationSize the size of the populations to be generated by this
     * algorithm
     * @param mutationRate the rate at which mutations will occur on each
     * generation
     */
    public GeneticChoice(int populationSize, double mutationRate) {
        super(populationSize, mutationRate);
    }

    /**
     * Swap the required properties of the two choices, as required by the
     * crossover operation.
     *
     * @param kid1 the first schedule involved in the crossover
     * @param empId1 the employee for which details are to be swapped in the
     * first schedule
     * @param day1 the day on which details are taken from the first schedule
     * @param kid2 the second schedule involved in the crossover
     * @param empId2 the employee for which details are to be swapped in the
     * second schedule
     * @param day2 the day on which details are taken from the second schedule
     */
    protected void crossover(Schedule kid1, Integer empId1, int day1,
                             Schedule kid2, Integer empId2, int day2) {
        Choice src1 = (Choice) kid1.getChoices(empId1, day1).get(0);
        Choice src2 = (Choice) kid2.getChoices(empId1, day1).get(0);

        kid1.removeChoice(src1);
        kid1.addChoice(src2);
        kid2.removeChoice(src2);
        kid2.addChoice(src1);
    }
}
```

algorithm/rule/MaxConsecutiveDays.java

```
/*
 * MaxConsecutiveDays.java
 *
 * Created on April 10, 2005, 10:12 AM
 */

package geneticalgorithm.algorithm.rule;

import geneticalgorithm.bean.Choice;
import geneticalgorithm.bean.Employee;
import geneticalgorithm.model.Context;
import geneticalgorithm.model.EmployeeRegistry;
import geneticalgorithm.model.Schedule;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

/**
 *
 * @author mkhashayardoust
 */
public class MaxConsecutiveDays extends RuleBase {
    /**
```

```
     * The maximum number of consecutive days that employees
     * are allowed to work.
     */
    private int maxConsecutiveDays;

    /**
     * Creates a new instance of MaxConsecutiveDays.
     */
    public MaxConsecutiveDays(int maxConsecutiveDays) {
        this.maxConsecutiveDays = maxConsecutiveDays;
    }

    public long evaluateImpl(Schedule schedule, Context context) {
        int count = 0;
        boolean isMandatory = true;
        EmployeeRegistry registry = context.getEmployees();
        Collection employees = registry.getEmployees();
        for (Iterator n = employees.iterator(); n.hasNext(); ) {
            Employee employee = (Employee) n.next();
            if (!employee.isFixedShift()) {
                count = employee.geConsecutiveDays();
                for (int day=0; day <= context.getNumberOfDays(); day++) {
                    List list = (List)schedule.getChoices(employee.getEmployeeId(), day);
                    if(isWorkDay(list)) {
                        count++;
                        isMandatory = isMandatory && hasMandatoryChoice(list);
                        if (count > this.maxConsecutiveDays && !isMandatory) {
                            return Long.MAX_VALUE;
                        }
                    } else {
                        count = 0;
                        isMandatory = true;
                    }
                }
            }
        }
        return 0;
    }

    /**
     * Test if any choice in the choice list is mandatory.
     *
     * @param list the list of choices to be tested
     *
     * @return true if any of the choices are mandatory
     */
    private boolean hasMandatoryChoice(List list) {
        for (Iterator n = list.iterator(); n.hasNext(); ) {
            if (((Choice) n.next()).isMandatory()) {
                return true;
            }
        }
        return false;
    }

    /**
     * Test if any choice in the choice list has the interval length
     * greater than zero.
     *
     * @param list the list of choices to be tested
     *
     * @return true if any of the choices have the interval length greater than
     * zero
     */
    private boolean isWorkDay(List list) {
       for (Iterator n = list.iterator(); n.hasNext(); ) {
            if (((Choice) n.next()).getLength() > 0) {
                return true;
            }
        }
        return false;
    }

    /**
     * Set the shift length of all the choices in a list to zero.
     *
     * @param list the list of choices to be modified
     */
    private void zeroShiftLengths(List list) {
        for (Iterator n = list.iterator(); n.hasNext(); ) {
```

```
            ((Choice) n.next()).getShift().setLength(0);
        }
    }


    /**
     * Attempt to repair this schedule so that it does not break a hard
     * constraint.
     *
     * @param schedule the schedule to be repaired
     * @param context the context in which the current rule is to be repaired
     *
     * @return <code>false</code>
     */
    public boolean repair(Schedule schedule, Context context) {
        int count = 0;
        boolean isMandatory = true;
        EmployeeRegistry registry = context.getEmployees();
        Collection employees = registry.getEmployees();
        for (Iterator n = employees.iterator(); n.hasNext(); ) {
            Employee employee = (Employee) n.next();
            if (!employee.isFixedShift()) {
                count = employee.geConsecutiveDays();
                for (int day=0; day <= context.getNumberOfDays(); day++) {
                    List list = (List)schedule.getChoices(employee.getEmployeeId(), day);
                    if(isWorkDay(list)) {
                        count++;
                        isMandatory = isMandatory && hasMandatoryChoice(list);
                        if (count > this.maxConsecutiveDays && !isMandatory) {
                            zeroShiftLengths(list);
                            count = 0;
                            isMandatory = true;
                        }
                    } else {
                        count = 0;
                        isMandatory = true;
                    }
                }
            }
        }
        return true;
    }

    /**
     * Return a string representation of this rule.
     */
    public String toString() {
        return "Maximum Consecutive Days = " + maxConsecutiveDays;
    }
}
```

algorithm/rule/ObjSeniority.java

```
/*
 * Seniority.java
 *
 * Created on May 3, 2005, 8:07 AM
 */

package geneticalgorithm.algorithm.rule;

import geneticalgorithm.bean.Choice;
import geneticalgorithm.bean.Employee;
import geneticalgorithm.model.Context;
import geneticalgorithm.model.EmployeeRegistry;
import geneticalgorithm.model.Schedule;
import java.util.List;
import java.util.Collection;
import java.util.Iterator;


/**
 *
 * @author mkhashayardoust
 */
public class ObjSeniority extends RuleBase  {

    /** The singleton instance of this rule. */
```

```
    public static final ObjSeniority RULE = new ObjSeniority();

    /** Creates a new instance of Seniority */
    private ObjSeniority() {
    }

    /**
     * Evaluate this rule for a specified schedule in the current context.
     *
     * @param schedule the schedule to be evaluated
     * @param context the context in which the current rule is to be evaluated
     *
     * @return <code>0</code>
     */
    public long evaluateImpl(Schedule schedule, Context context) {
        int seniority = 0;
        double objSeniority = 0;
        double senior = 0;
        EmployeeRegistry registry = context.getEmployees();
        Collection employees = registry.getEmployees();
        for (Iterator n = employees.iterator(); n.hasNext(); ) {
            Employee employee = (Employee) n.next();
            if (!employee.isFixedShift()){
                seniority = employee.getSeniority();
                for (int day = 0; day < context.getNumberOfDays(); day++) {
                    List list= (List)schedule.getChoices(employee.getEmployeeId(), day);
                    for (int i=0; i< list.size(); i++){
                        senior=((double)seniority/1000)*((Choice) list.get(i)).getLength()* 0.12;
                        objSeniority += senior;

                }
                }
            }
        }
        return (long)objSeniority;
    }

}
```

REFERENCES

Abboud, N., Inuiguchi, M., Sakawa, M. and Uemura, Y., 1998. Manpower Allocation Using Genetic Annealing. *European Journal of Operational Research*, **111** (2), pp.405-420.

Arabeyre, J.P., Fearnley, J., Steiger, F.C. and Teather, W., 1969. The Airline Crew Scheduling Problem: A Survey. *Transportation Science*, **3** (2), pp.140-163.

Bailey, R.N., Garner, K.M. and Hobbs, M.F., 1997. Using Simulated Annealing and Genetic Algorithms to Solve Staff Scheduling Problems. *Asia - Pacific Journal of Operational Research*, **14** (2), pp.27-43.

Bechtold, S.E., Brusco, M.J. and Showalter, M.J., 1991. A Comparative Evaluation of Labor Tour Scheduling Methods. *Decision Sciences*, **22** (4), pp. 683-699.

Bellman, R.E. and Zadeh, L.A., 1970. Decision Making in a Fuzzy Environment. *Management Science* **17** (4), pp.141 - 164.

Berger, P.D. and Maurer, R.E., 2002. *Experimental Design with Applications in Management, Engineering, and the Sciences*. Belmont, CA: Thomson Group.

Burke, E., Cowling, P., Causmaecker, P.D and Berghe, G.V., 2001. A Memetic Approach to the Nurse Rostering Problem. *Applied Intelligence*, **15** (3), pp.199-214.

Burke, E.K, Causmaecker, P.K., Berghe, G.V and Landeghem, H.V., 2004. The State of The Art of Nurse Rostering. *Journal of Scheduling*, **7** (6), pp.441-499.

Cai, X. and Li, K.N., 2000. A Genetic Algorithm for Scheduling Staff of Mixed Skills Under Multi- Criteria. *European Journal of Operational Research*, **125** (2), pp.359-369.

Easton, F.F. and Mansour, N., 1999. A Distributed Genetic Algorithm for Deterministic and Stochastic Labour Scheduling Problems. *European Journal of Operational Research*, **118** (3), pp. 505–523.

Ernst, A.T., Jiang, H., Krishnamoorthy, M. and Sier, D., 2004a. Staff Scheduling and Rostering: A Review of Applications, Methods and Models. *European Journal of Operational Research*, **153** (1), pp.3-27.

Ernst, A.T., Jiang, H., Krishnamoorthy, M., Owens, B. and Sier, D., 2004b. An Annotated Bibliography of Personnel Scheduling and Rostering. *Annals of Operations Research*, **127** (1-4), pp.21-144.

Gen, M. and Cheng, R., 2000. *Genetic Algorithms and Engineering Optimization.* New York: Wiley.

Herbert, S.W., 1994. *Algorithms and Complexity.* [e-book]. University of Pennsylvania. Available from: www.cis.upenn.edu/wilf / E-books[cited 2005]

Hillier, F.S and Liberman, G.J., 2001. *Introduction to Operations Research.* 7th ed. Boston, Toronto: McGraw-Hill.

Holland, J.H., 1975. *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press.

Levine, D., 1996. Application of a Hybrid Genetic Algorithm to Airline Crew Scheduling. *Computers & Operations Research*, **23** (6), pp.547-558.

Rothlauf, F., 2002. *Representations for Genetic and Evolutionary Algorithms.* Heidelberg: Physica-Verlag.

Murty, K.G., 1995. *Operations Research: Deterministic Optimization Models*. Englewood Cliffs, NJ.: Prentice Hall.

Quan, V., 2004. Retail Labour Scheduling, *ORMS Today*, **31** (6), pp.32-36.

Zimmermann, H.J., 1996. *Fuzzy Set Theory and Its Applications*. 3rd ed. Boston, Dordrecht, London: Kluwer Academic Publishers.

Zolfaghari, S. and Liang, M., 2003. A New Genetic Algorithm for the Machine/Part Grouping Problem Involving Processing Time and Lot Sizes. *Computers & Industrial Engineering*, **45** (4), pp.713- 731.