A Genetic Algorithm for Corrective Retesting

By
Khaled A. Fakih

May 1995

A Genetic Algorithm for Corrective Retesting

By
Khaled A. Fakih

THESIS

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
at the Lebanese American University
May 1995

[redacted]

Dr. Nashat Mansour (Advisor)
Assistant Professor of Computer Science
Lebanese American University

[redacted]

Dr. George Nasr
Assistant Professor of Electrical and Computer Engineering
Lebanese American University

[redacted]

Dr. Haidar Harmanani
Assistant Professor of Computer Science
Lebanese American University

Abstract

The optimal retesting problem is that of determining the minimum number of test cases needed for revalidating modified software in the maintenance phase. We present a genetic algorithm (GA) for solving this problem, based on the program's flow graph and an integer programming problem formulation. The algorithm deviates from classical GAs in that it incorporates some design choices to guarantee a final feasible solution and to improve the efficiency of the genetic search. These choices include elitist ranking, random feasibilization, penalization, and a hybridized hill-climbing procedure. The main advantage of this algorithm is that it does not suffer from exponential explosion for large program sizes. Further, the experimental results show that it finds an optimal number of retests faster than other known methods.

Acknowledgments

I would like to express my gratitude to Professor Nashat Mansour, my advisor, for the encouragement, help, and support he provided me. I extend my gratitude to the second readers, Professor George Nasr and Professor Haidar Harmanani.

I am very grateful to the Harriri Foundation for their financial support, and for Dr. Leila Khoury and Director Sami Mansour, of LAU, for their personal support.

Special thanks go to my colleagues Jalal Kawash, Simon Laham, and Gabriel Aghazarian for the valuable comments and discussions, the A.C.C supervisor Mr. Tarek Dana, and the assistant Mr. Ali Aleywan for their help.

Last, but not least, I would like to express my love to my parents, my sisters, my teachers, my friends, and my colleagues at LAU for their love and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software maintenance is known to be an expensive phase in the software development life cycle [Benett 1991]. Estimates have shown that it consumes between 50 to 80% of the total software cost [Leung and White 1991]. Usually, software maintenance is divided into three activities. *(i)* corrective, in response to an error report, *(ii)* perfective when the existing system is improved or enhanced, or *(iii)* adaptive when the system is adapted in response to changes in its requirements [Hartmann and Robson 1990]. In each of the three cases, the software has to be revalidated by regression testing.

Regression testing is a testing process which is applied after a program is modified. It aims for verifying that the modifications have not caused unintended adverse side effects and that the modified system still meets the requirements [IEEE 1983]. The two types of regression testing which have been identified are namely progressive and corrective regression testing [Leung and White 1989]. The former involves testing major changes to the specification, so whenever enhancements, or new data requirements are incorporated

into a system, the specifications will be altered to reflect the modifications. In general, such alterations are expressed in terms of new modules or routines being added, and perhaps old ones being eliminated. On the other hand, corrective regression testing is performed on a specification that essentially remains unchanged, so that only minor alterations, that do not affect the overall program structure, require revalidation [Hartmann and Robson 1989].

Typically, progressive regression testing is performed after adaptive or perfective maintenance [Lientz and Swason 1980]. Therefore it is usually employed at regular intervals, whilst the corrective retesting strategy is undertaken after corrective maintenance activities, that can occur at any time and may be invoked at irregular intervals, as after every correction [Hartmann and Robson 1989].

Regression testing is a significant component of maintenance. Hence, efficient and effective regression testing can reduce the cost of maintenance [Leung and White 1991]. However, there are two major problems in regression testing, *(i)* the test plan update problem and *(ii)* the test selection problem. The test plan update problem deals with the management of the test plan as a program is undergoing successive modifications. This problem could be divided into two activities a) test case classification problem and b) test update problem. The former involves grouping the test cases into three mutually exclusive sets : reusable, retestable, and obsolete. The later involves the deletion of obsolete test cases, and the addition of new structural and new specification test cases.

The test selection problem (or selective revalidation) is concerned with the selection of test cases to fully test a modified program. For selective revalidation, it is costly to repeat the whole set of test cases used in the initial development of the program . Moreover, it is unreliable to choose a random subset of test cases [Fischer 1977]. Therefore, it is important to select a subset of test cases that has minimal cardinality and yet accomplish the goals of regression testing [Mansour and Goel 1994]. This problem is referred to as optimal regression testing, or simply optimal retesting. In general, optimal retesting aims at determining exactly which subset of the previous executed test cases is required to be rerun, in order to show that there had been no deterioration of software reliability [Leung and White 1991].

In testing or regression testing, it is important to identify various levels of abstraction at which testing should be applied. *Unit testing* is applied to each individual procedure. *Integration testing* involves sets of procedures, ideally in an incremental fashion, so that one or more procedures are added to those already integration tested; the tests are devised so as to detect errors in the interface between the various sets and the newly added procedures. *Function testing* is applied to the entire software system, using the software functional specifications and a black-box testing approach. *System testing* is applied to the entire project, including the hardware platform, again typically using black-box testing, and may well be concerned with overall system performances as well as correct functionality [Hartmann and Robson 1988; White et al. 1993].

In this thesis, we use an integer programming problem formulation and present a genetic algorithm (GA) adapted, for the first time, to solve the problem of corrective software retesting. This GA produces optimal or near-optimal solutions and does not suffer from intractability. This makes its application to various program sizes realistic. Further, since classical GAs do not guarantee feasible solutions for constrained optimization problems, the algorithm is hybridized to satisfy feasibility requirements. The experimental results show that it finds an optimal number of retests faster than other known methods.

This thesis is organized as follows. Chapter 2 describes the retesting problem and its integer programming formulation. In chapter 3 , a genetic algorithm for solving the optimal restesting problem is presented. Chapter 4 includes a comparative experimental evaluation of HGA, B&B, and SA at both the unit and program levels. Chapter 5 includes a survey of current approaches to regression testing. Chapter 6 contains conclusions and suggestions for further research.

Chapter 2

Retesting Modified Software

This chapter describes Fischer's Methodology for selective revalidation [Fischer 1977; Fischer 1981]. The objective of this methodology is to present a quantitative approach for solving the retesting problem. This method could lead to tools which will decrease the cost associated with current maintenance practices, as well as increase the reliability of modifications done to software systems [Hartmann & Robson 1990].

1. Retesting Modified Software

1.1. The Definition of Program Segments

Analysis of large computer programs at the source statement level requires a very large capacity for data storage. Segmentation has been used as a method for data reduction at the module level by several test tools to group blocks of one or more executable statements together without loss of detail. These blocks are called segments. A segment is defined to be a contiguous sequence of executable statements with one entry point at the

beginning and one exit point at the end [Fischer 1977]. By executing the first statement in a segment, all other statements in that segment are also executed.

1.2. The Zero-One (0-1) Integer Programming Model

The restest methodology employs knowledge of the reachability among segments. The reachability matrix identifies, for each segment, all the other segments that can reach to it or be reached from it (either directly or indirectly). Two other matrices of importance are the module level set/use matrix and the cross reference matrix. The former identifies the status of all local varables, global variables, and arguments within the remote module while the later identifies what segments are tested by each test case. Both of these matrices are further discussed in subsequent sections of this chapter.

The selection of the optimal subset of test cases is accomplished by using the 0-1 integer programming technique with data provided by the reachability , set/use, and test case cross reference matrices. The integer programming model assumes that the module under consideration is represented by a control flow graph with **m** program segments. Also, it is assumed that the set of **n** test cases used in the initial development of the module are saved. Given that the program segment **k** has been modified, the optimal retesting problem solution must give the minimum number of test cases necessary to measure that all segments reachable from/to the modified code are executed at least once. An

abstraction formulation of Fischer's 0-1 integer programming model for retesting modified software modules is presented in Equation 1.1 [Fischer 1977; Davis and Kendrik 1971].

Minimize $\mathbf{Z} = c_1X_1 + c_2X_2 + \ldots\ldots\ldots + c_nX_n$ **(1.1)**

Subject to the following constraints :

$$\begin{array}{llll} a_{11}X_1 + a_{12}X_2 + \ldots\ldots & + a_{1n}X_n & \geq & b_1 \\ a_{21}X_1 + a_{22}X_2 + \ldots\ldots & + a_{2n}X_n & \geq & b_2 \\ \cdot \quad \cdot & \cdot & \cdot & \cdot \\ \cdot \quad \cdot & \cdot & \cdot & \cdot \\ \cdot \quad \cdot & \cdot & \cdot & \cdot \\ a_{m1}X_1 + a_{m2}X_2 + \ldots\ldots & + a_{mn}X_n & \geq & b_m \end{array}$$

These equations can be represented in a matrix notation as illustrated in Equation 1.2.

Minimize $\mathbf{Z} = c_1X_1 + c_2X_2 + \ldots\ldots\ldots + c_nX_n$ **(1.2)**

Subject to the following constraints :

$$\sum_{j=1}^{N} a_{ij}X_j \geq b_i \; ; \; i = 1, \ldots, M$$

a) The objective function **Z**, represents the optimal number of test cases required to be rerun after a modification is made to a particular program segment.

b) The coefficient c_j represents a cost element of the objective function **Z**. In this model, the cost elements are set to 1, as it is assumed that all test cases have an equal cost to set up and run.

c) The variable a_{ij} is an element of the constraint coefficient matrix which is taken directly from the test case cross reference matrix, i.e a_{ij} = 1 (or 0) of segment **i** is covered (or not covered) by test case **j**.

d) All values indicated by the b_i elements represent the lower bound of the constraint **i**. It is taken directly from the logical **OR** of the applicable rows and columns of the reachability matrix corresponding to the modified segment **k**. b_i = 1 (or 0) indicates whether segment **i** needs to (or needs not) be covered by the subset of retests due to the modification of segment **k**.

e) the variable X_j = 1 (or 0) indicates the inclusion (or exclusion) of test case **j** in the selected subset of retests.

The exact procedure is explained via example in subsequent sections. From the solution of this model, the value of the objective function **Z** will give the minimum number of test cases necessary to assure full retest coverage, and the values of X_j that are equal to 1 will identify the specific test cases which form the optimal retest subset.

2. Example

Figure 1 shows the flow of routine used as an example in this section.

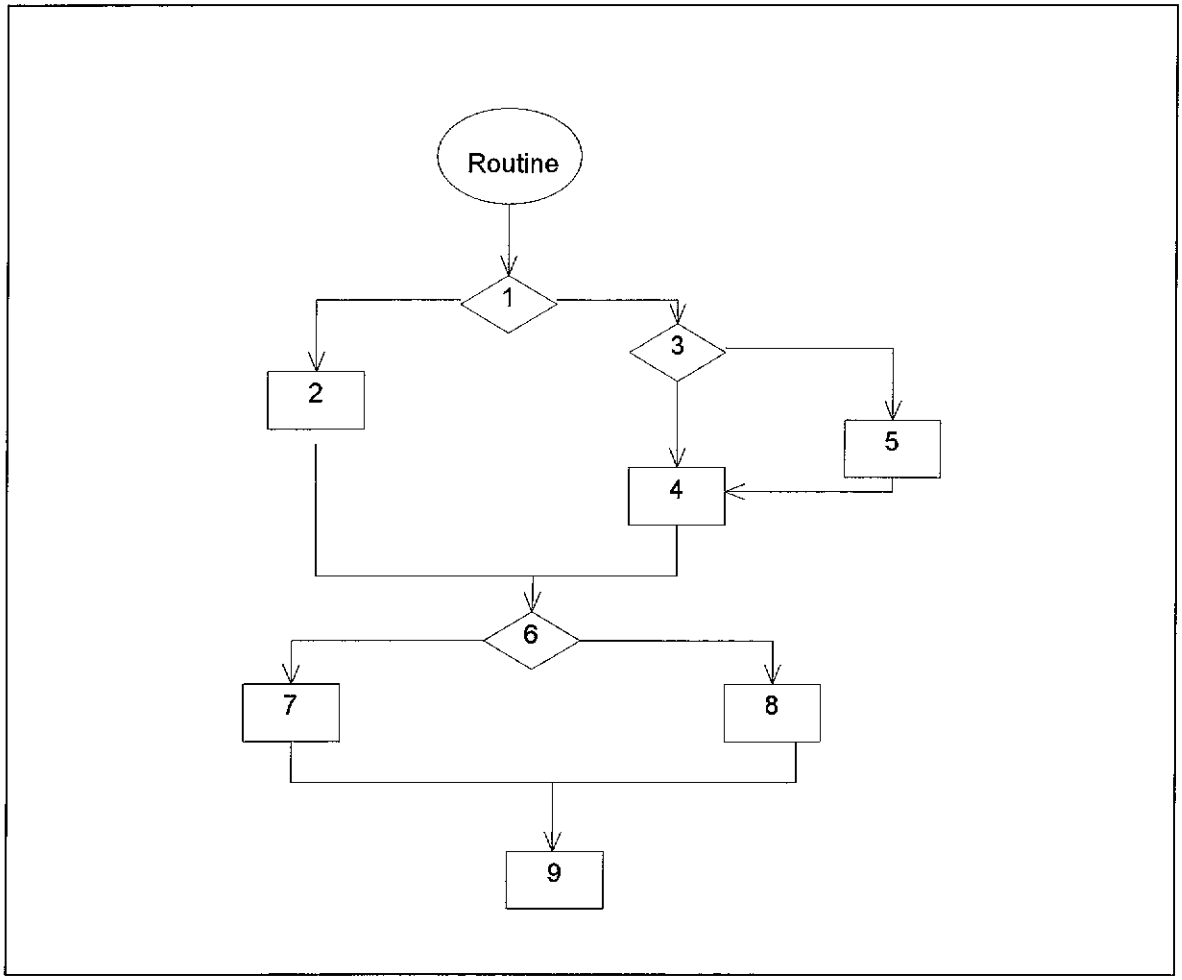


Figure 1. Control-flow graph presentation of a module.

Retest methodology views a software module as a directed graph where each node is a segment, and the arcs represent connectivity between segments. The directed graph for the module used as an example in Figure 1 is shown in Figure 2.

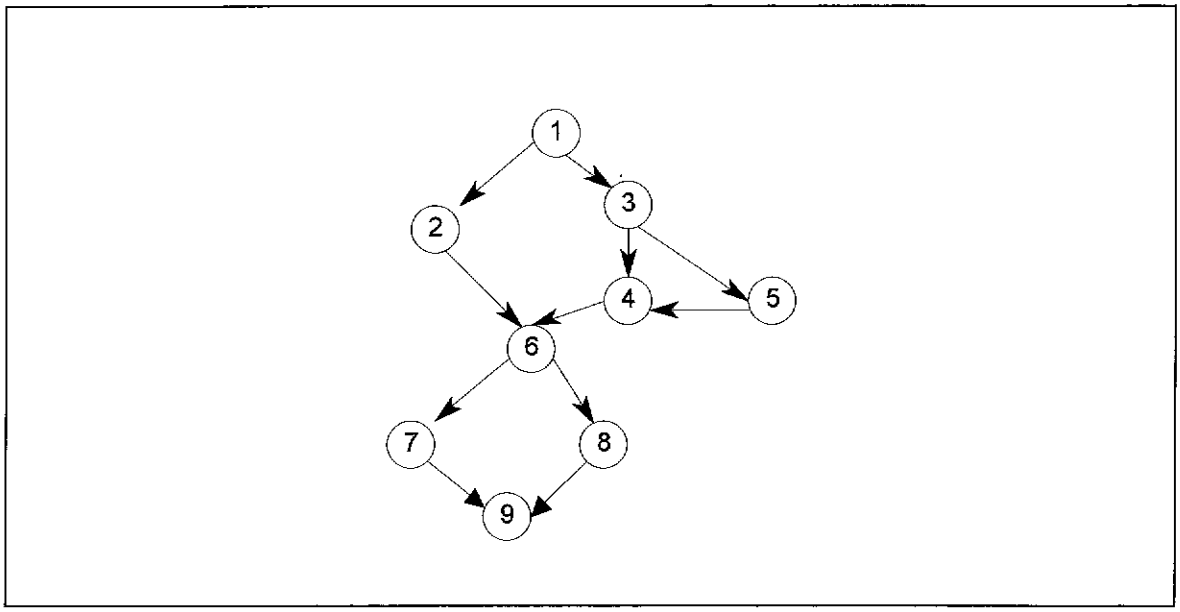


Figure 2. Directed graph presentation of a module.

Associated with each module is a connectivity matrix and a reachability matrix. The connectivity matrix reflects the program's control-flow by indicating any direct interconnections between the different segments. It is represented in the form of a square matrix, containing m^2 elements, in which a 1 entry shows that the two particular segments are directly connected, whereas a 0 entry suggests otherwise. Figure 3 illustrates the connectivity matrix based on the directed graph representation of the module shown in Figure 2.

The reachability matrix describes both the direct and indirect interconnections between the various program segments. It can be automatically generated from the connectivity matrix by applying Warshall's Algorithm [Warshall 1962]. The reachability matrix is of the same dimensions as of the connectivity matrix, with a 1 entry indicates whether a particular segment can be reached either directly or indirectly from another, whereas

a 0 suggests otherwise. Figure 4 illustrates the connectivity matrix based on the directed graph representation of the module shown in Figure 2.

		To Seg								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	1	0	1	1	0	0	0	0	0	0
From	**2**	0	0	0	0	0	1	0	0	0
Seg.	**3**	0	0	0	1	1	0	0	0	0
	4	0	0	0	0	1	0	0	0	0
	5	0	0	0	0	0	1	0	0	0
	6	0	0	0	0	0	0	1	1	0
	7	0	0	0	0	0	0	0	0	1
	8	0	0	0	0	0	0	0	0	1
	9	0	0	0	0	0	0	0	0	0

Figure 3. Connectivity matrix for module presented in Figure 2.

		To Seg								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	1	1	1	1	1	1	1	1	1	1
From	**2**	0	1	0	0	0	1	1	1	1
Seg.	**3**	0	0	1	1	1	1	1	1	1
	4	0	0	0	1	1	1	1	1	1
	5	0	0	0	0	1	1	1	1	1
	6	0	0	0	0	0	1	1	1	1
	7	0	0	0	0	0	0	1	0	1
	8	0	0	0	0	0	0	0	1	1
	9	0	0	0	0	0	0	0	0	1

Figure 4. Reachability matrix for module presented in Figure 2.

Additionally, a test case cross reference matrix is associated with each module. This matrix is constructed by identifying segments executed by test cases during the testing phase of software development. If any element (i, j) of the test case cross reference matrix is 1; segment **i** is executed by test case **j**; 0 otherwise. For example, in Figure 4, test case 1 exercises segments 1, 2, 6, 7 and 9 of the example module. Test case

dependency matrix is assumed to be generated and saved during the life cycle testing phase.

	Test Cases					
Segments	**1**	**2**	**3**	**4**	**5**	**6**
1	1	1	1	1	1	1
2	1	1	0	0	0	0
3	0	0	1	1	1	1
4	0	0	0	0	1	1
5	0	0	1	1	1	1
6	1	1	1	1	1	1
7	1	0	1	0	1	0
8	0	1	0	1	0	1
9	1	1	1	1	1	1

Figure 5. Test case cross reference matrix for module presented in Figure 1.

In this example, we assume a change to segment 2 is made. The following analysis should be performed to select the test cases for retest. The nine constraint expressions (i.e., one for each segment), corresponding to rows of the test case cross reference matrix, serve to assure that at least one test case executes every segment that is reached from or reaches to the modified segment [Fischer 1981]. The right hand values ($\mathbf{b_i}$**'s**) are the result of a logical OR operation performed between the row and column of the reachability matrix associated with modified segment (in this case row and column 2). The outcome is incorporated into the 0-1 integer programming model as the right hand side ($\mathbf{b_i}$**'s**) values.

Minimize $\mathbf{Z} = X_1 + X_2 + X_3 + X_4 + X_5 + X_6$

Subject to :

$$
\begin{array}{rcl}
X_1 + X_2 + X_3 + X_4 + X_5 + X_6 & \geq & 1 \\
X_1 + X_2 & \geq & 1 \\
+ X_3 + X_4 + X_5 + X_6 & \geq & 0 \\
X_5 + X_6 & \geq & 0 \\
X_3 + X_4 + X_5 + X_6 & \geq & 0 \\
X_1 + X_3 + X_5 & \geq & 1 \\
X_1 + X_2 + X_3 + X_4 + X_5 + X_6 & \geq & 1 \\
X_2 + X_4 + X_6 & \geq & 1 \\
X_1 + X_2 + X_3 + X_4 + X_5 + X_6 & \geq & 1
\end{array}
$$

Figure 6. 0-1 Integer programming model for module presented in Figure 1.

This model can be reduced by several reduction methods [Fischer 1977; Davis and Kendrik 1971] some of which were discussed above, and others of which are presented in Appendix **A**. After the reduction process, redundant constraints may appear and may be removed as shown in this example. The final model formulation reduces to :

Minimize $\mathbf{Z} = X_1 + X_2$

Subject to :

$$
\begin{array}{rcl}
X_1 & \geq & 1 \\
X_2 & \geq & 1
\end{array}
$$

Solution of this example shows that both $\mathbf{X_1}$ and $\mathbf{X_2}$ are 1 and the optimal value of the objective function is 2. This means that there are two test cases to be rerun and that they are test cases 1 and 2.

3. Concluding Remarks

We have presented Fisher's methodology for software retesting that could lead to tools which will decrease the high cost associated with current maintenece practices. This methedology is base on a 0-1 integer programming model formulation. Before establishing this model, information about the connectinity, reachability, and test coverage of program segments and their data dependencies must be accomodated. This information is obtained from the general flow analysis of the program, the results of which are stored in four matrices : connectivity, reachability, test case dependency (or cross reference), and variable set/use.

Chapter 3

A Hybrid Genetic Algorithm for Optimal Retesting

This chapter includes a genetic algorithm adapted for solving the optimal retesting problem. The algorithm incoporates some design choices to guarantee a feasible final solution and to improve the effeciency of the genetic search.

1. Background on Genetic Algorithms

Genetic Algorithms (GAs) are based on the mechanics of natural evolution [Holland 1975; Goldgerg 1989]. They mimic natural populations reproduction and selection operations to achieve efficient and robust optimization. Through their artificial evolution, successive generations search for beneficial adaptations in order to solve the problem. Each generation consists of a population of chromosomes, also called individuals. Each individual represents a possible solution. The initial generation consists of randomly created individuals. The Darwinian principle of reproduction and survival of the fittest and

the genetic operations of recombination (crossover) and mutation are used to create a new offspring population from the current population. The reproduction operation involves selecting, in proportion to fitness, an individual from the current population of individuals, and allowing it to survive by copying it into the new population. Then, two mates are randomly selected and crossover is done to create two new offspring individuals. After the operations of reproduction and crossover are performed in the current population, the population offspring (i.e the new generation) replaces the old population, and the process is repeated for many generations. Typically, the best individual that appeared in any generation of a run (i.e best-so-far individual) is designated as the result produced by the GA.

2. Solving the Optimal Retesting Problem

In this section, we describe how genetic algorithms can be adapted for solving the optimal restesting problem. We present the components of a hybrid genetic algorithm (HGA) for minimizing the cost function **Z** (Equation 1.1 in chapter 2). An outline of the algorithm is given in Figure 1 and its components are described in the following subsequent sections.

GAs often encounter the problem of premature convergence to local optima; otherwise, a long time may be required for the evolution to reach near-optimal solutions. Another problem is the likelihood of producing infeasible individuals as a result of crossover and

mutation. The HGA described in this section incorporates techniques and design choices for overcoming these two problems.

```
Random generation of initial population, Size POP;
Set rates for genetic operators;
Evaluate fitness of individuals;
repeat
        Rank individuals & allocate reproduction trials;
        for (i=1 to POP step 2) do
                Randomly select 2 parents from list of reproduction trials;
                Apply crossover & mutation;
        endfor
        Evaluate fitness of offspring's;
        Check feasibility of individuals;
        Do feasibilization, penalization & hill-climbing;
        Preserve the fittest-so-far (elitism);
until (termination criterion is satisfied)
solution = Fittest
```

Figure 1. Hybrid genetic algorithm for optimal retesting

2.1. Population and Chromosomal Representation

HGAs population is an array of POP individuals. An individual in the population is encoded as n-element vector $[X_1, X_2, X_3, \ldots X_n]$ that corresponds to a candidate solution for the optimal retesting problem. An element (gene) $X_j = 1$ (or 0) indicates the inclusion (or exclusion) of test case j from the selected subset of retests. The initial population of individuals is randomly generated.

2.2. Objective Function Evaluation

The fitness of an individual **Z** is evaluated by adding the genes of an individual. Henceforth, the optimal subset of restests corresponds to the minimum fitness **Z** of all feasible individuals.

2.3. Reproduction Scheme and Elitism

In HGA, the whole population is considered a single reproduction unit within which random selection is performed. Our reproduction scheme involves elitist ranking, followed by random selection of mates from the list of reproduction trials (or copies) assigned to the ranked individuals. In the ranking scheme [Baker 1985], the individuals are sorted by

fitness values. After sorting, each individual is assigned a rank based on a scale of equidistant values for the population. The ranks assigned to fittest and least-fit individuals are 1.2 and 0.8, respectively. Individuals with ranks greater than 1 are first assigned single copies. Then, the fractional part of their ranks and the ranks of the lower half of individuals are treated as probabilities for random assignment of copies. Figure 2 below illustrates ranking selection in a population of four individuals.

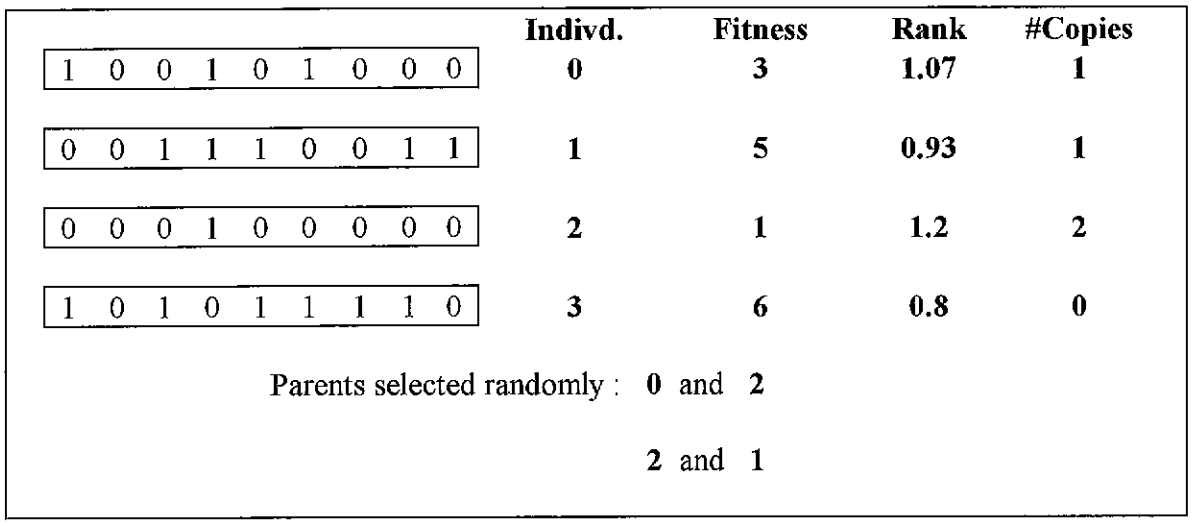


Figure 2. Illustration of ranking based selection scheme.

It has been found that ranking based selection with maximum rank of 1.2 produces individual survival percentage of 92 to 98% in different generations. This helps in maintaining population diversity and controlling premature convergence. Elitism is used to exploit good building blocks and to ensure that good candidate solutions are saved if the search is to be truncated at any point. Preservation of the fittest individual is done by replacing the fittest-so-far individual in place of the least-fit individual if it is better than the current-fittest.

2.4. Genetic Operators

The genetic operators employed in HGA are crossover and mutation, which are illsutrated in Figure 3. Pairs of individuals are randomly selected from the mating pool. Each pair of these strings undergoes crossover as follows : an integer position **k** along the string is selected at random between [1.. **n**], where n is equal to : string-length minus -1. The two new strings are created by swapping all characters between **k+1** and **n** inclusively. The standard mutation operator is employed. Strings and allele positions where the alteration of the value is going to occur are selected randomly. A mutation rate of 0.02 and crossover rate of 0.7 [Grefensttete 1986] are used in our implementation.

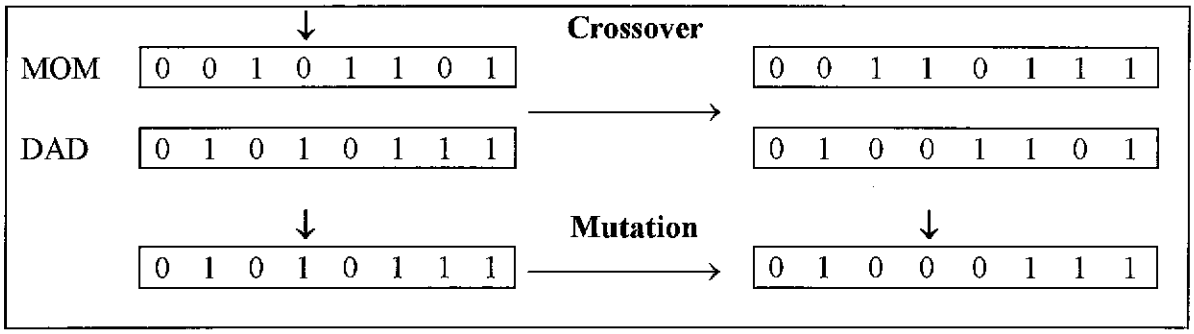


Figure 3. Genetic operators.

2.5. Feasibilization, Penalizing, and Hill-Climbing

Standard genetic operators like crossover and mutation frequently produce infeasible solutions for constrained optimization. Our HGA uses the technique described in [Easton and Mansour 1993] which is outlined in Figure 4 and described in detail.

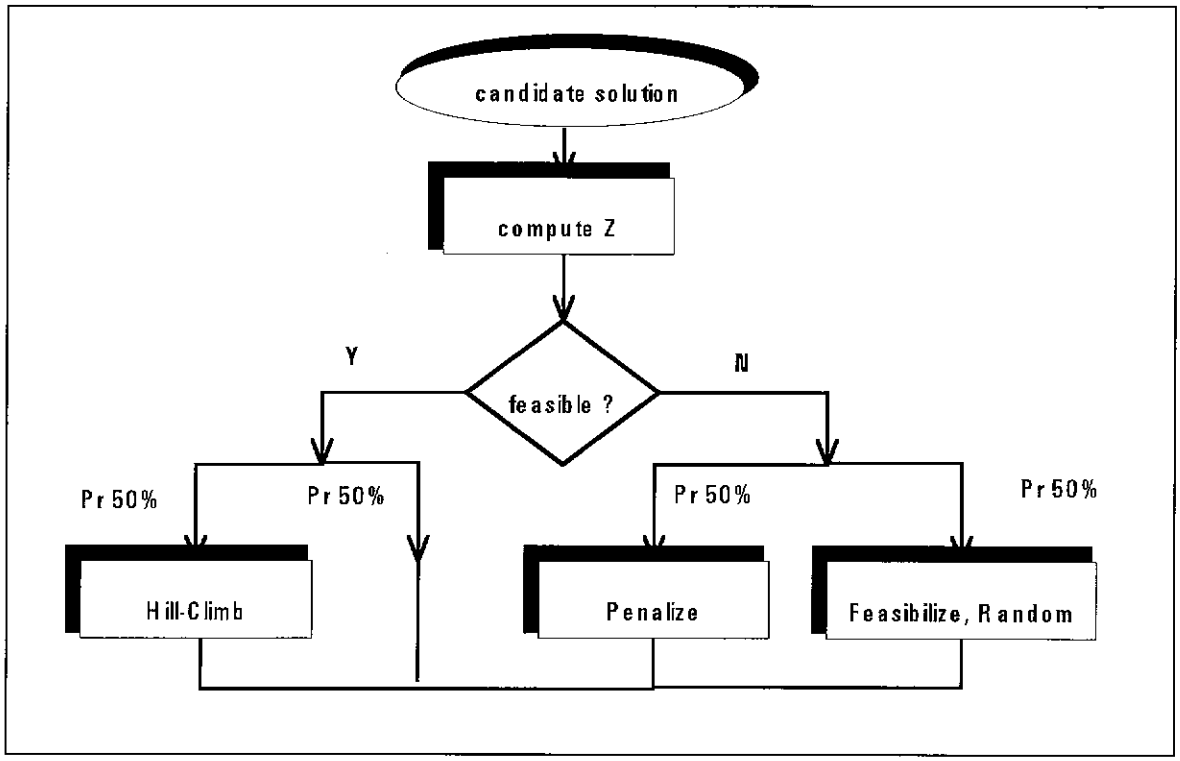


Figure 4. Feasibilization, Penalizing and Hill-climbing

At the detection of an infeasible string i , HGA computes a simple penalty based on the maximum shortage difference, $\sum_{j=1}^{n} a_{ij}X_j - b_i$. This difference, which represents the minimum additional resources needed by the string i to be completed (i.e satisfy all problem constraints), is added to **Z**. The infeasible string is then allowed to enter population intact. In this way the penalized infeasible string will have a lower probability of survival. However, 50% of the infeasible offspring are randomly selected for heuristic completion.

A random feasibilization heuristic is applied by randomly selecting a violated (infeasible) row, then randomly selecting a gene capable of reducing this violation. That is, it increases by one the number of test cases selected to the retest subset. The process is then repeated

until all constraints are satisfied. This helps to insure that the population includes some feasible candidates that could be exploited in subsequent generations, and helps maintain the locus of the search near the feasible region.

To refine solution quality, a simple problem-specific hill-climbing procedure that can decrease the individual's fitness values is incorporated. HGA randomly selects one-half of all feasible solutions formed in each generation and applies the following fast hill-climbing procedure that searches nearby solution space using a simple "drop one" methodology. One by one, the value of each gene is provisionally decremented. If the resulting solution is feasible , the provisional change is made permanent and then a new solution becomes the incumbent. If infeasible, the procedure restores the gene to its original value and selects another gene. The procedure terminates when no single gene in the current solution can be decremented to form a new, less costly and feasible solution. This hill-climbing procedure enables individuals to rapidly climb the peaks which speeds up the evolution process.

2.6. Termination Criterion

The termination criterion is satisfied when we converge to a solution. In this work, convergence is detected when the best-so-far string does not change its **Z** value for 20 generations.

Chapter 4

Experimental Results

In this chapter, we present a conventional branch and bound algorithm for solving the optimal restesting problem. Also, we experimentally compare the results of HGA with those of B&B and SA at both the module and program levels.

1. Branch and Bound Algorithm (B&B) for Optimal Retesting.

In chapter 1, we have presented the optimal retesting problem formulation. This formulation includes all the attributes that should exist in any problem to be solved by B&B strategy. These attributes concentrate on the branchability, boundability, and combinatorial nature of the problem [Thesen 1978]. However, the strategy must be merged with the structure of the problem at hand to form an acceptable algorithm.

In our B&B, we implicitly construct a tree describing all solutions to the problem, and conduct a guided search in this tree for the best feasible solution.

At each node of the constructed tree, we calculate a bound on the possible value of any solution that might happen to be further in the graph. If the bounds show that any such solution must necessarily be worse than the best solution we have found so far, then we do not need to go on exploring this part of the graph.

The calculation of these bounds is combined with depth-first search (DFS): (a) to prune certain branches of a tree or to close certain paths in a graph, and (b) to choose which of the open paths looks the most promising so that it can be explored next. Moreover, a priority list to hold these nodes is also needed.

The following are some design choices that have been chosen to implement our B&B :

a) A min-heap is used as a data structure for holding the nodes that have been generated and not yet explored [Brassard and Brately 1988].

b) The accumulated sum of all variable values generated so far is used as a bound function.

$$\sum_{i=1}^{i=level} X_i \quad ; X \in \{0, 1\}$$

c) After reaching a leaf node:

i) If the variable is feasible, delete all non-promising (variables with higher bound values) entries from the heap.

ii) If the heap is still not empty, the most promising variable any where is selected for back-tracking. If there exist more than one promising variable, select the one with the least tree height.

2. Experimental Results

In this section, we experimentally compare the results of HGA with those of B&B and SA at both the module and program levels. All three algorithms find optimal numbers of test cases, except when B&B fails to converge due to exponential explosion. Consequently, the comparison will be based on performance only. The experiments were done on an AViion 5000 UNIX machine. With each module/program are associated : the number of the modified segment, control-flow graph of **M** segments, a table of **N** test cases and their segment coverage information.

2.1. Module-Level Experimental Results

Tables 1 and 2 give the execution times of the three algorithms for small-size modules (m_1 - m_3) and large-size modules (m_4 - m_7), respectively. The results are also depicted in Figures 1 and 2. These results show that B&B would be the fastest algorithm for small-size modules and small number of test cases. But, as the module size and the number of

test cases grow, HGA clearly becomes the fastest. SA exhibits a similar behavior to that of HGA, but it remains slower.

	M(Segments)	N (Test cases)	Modified Seg.	Time(ms) B&B	SA	HGA
m_1	9	6	3	10	270	30
m_2	9	8	3	10	400	70
m_3	9	32	25	1450	2160	280

Table 1. The execution times of B&B, SA, and HGA for modules m_1 to m_3.

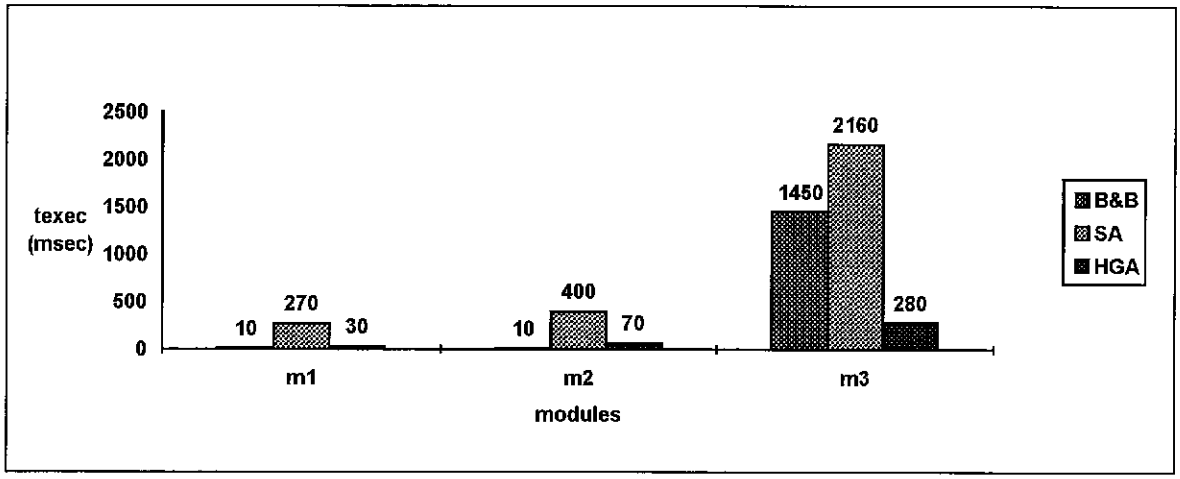


Figure1. The execution times of B&B, SA, and HGA for the modules presented in Table 1.

	M(Segments)	N (Test cases)	Modified Seg.	Time(sec) B&B	SA	HGA
m_4	40	24	30	5.849	2.010	0.170
m_5	31	64	25	12.277	4.749	0.470
m_6	34	128	19	97.937	10.038	0.580
m_7	60	144	11	388.232	6.729	1.290

Table2. The execution times of B&B, SA, and HGA for modules m_4 to m_7.

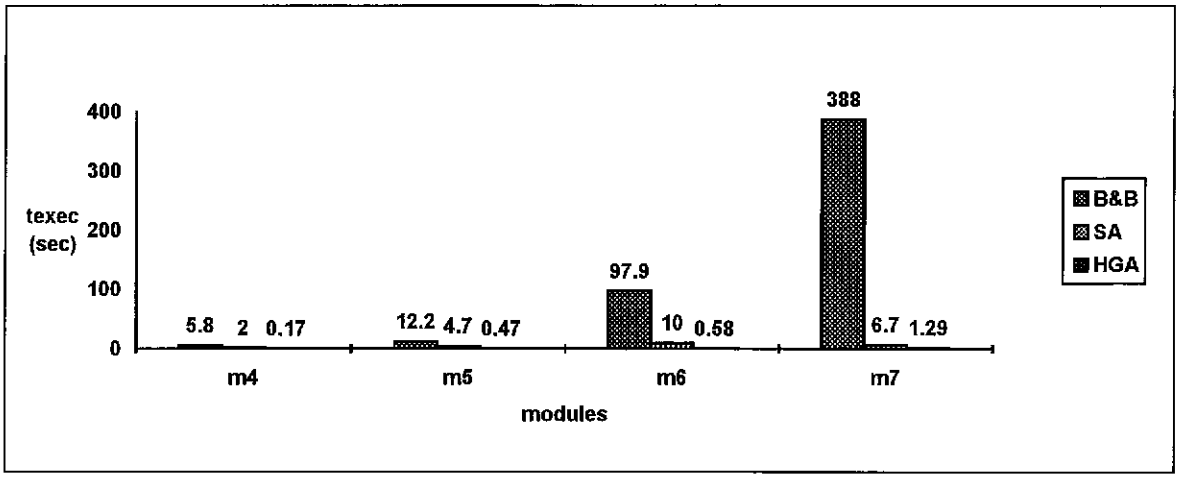


Figure 2. The execution times of B&B, SA, and HGA for modules presented in Table 2.

2.2. Program-Level Experimental Results

Tables 3-6 and Figures 3-10 give the execution times and the diferrence in execution times of SA and HGA for 15 different program sizes and numbers of test cases. The control-flow graphs representing these programs are randomly generated. The execution times for B&B are not included because they are either very large or infinitive. Also, the solutions (minimum number of retests) are not included since the two algorithms have converged to similar solutions. Clearly, HGA is faster than SA for all the 15 test cases. Moreover, as the program size and the number of test cases grow, the difference in execution times of HGA and SA increases.

			Time(seconds)	
	M(segments)	N(test cases)	SA	HGA
P_1	20	1000	73	6
P_2	20	2000	147	11
P_3	20	4000	309	24
P_4	20	6000	473	38

Table 3. Program level results for programs P_1-P_4.

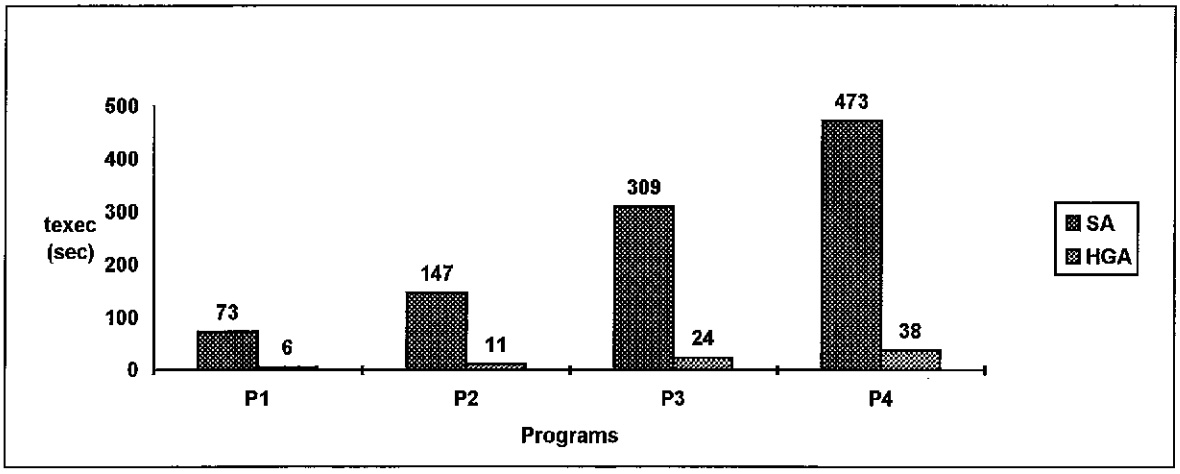


Figure 3. The execution times of SA and HGA for programs in Table 3 .

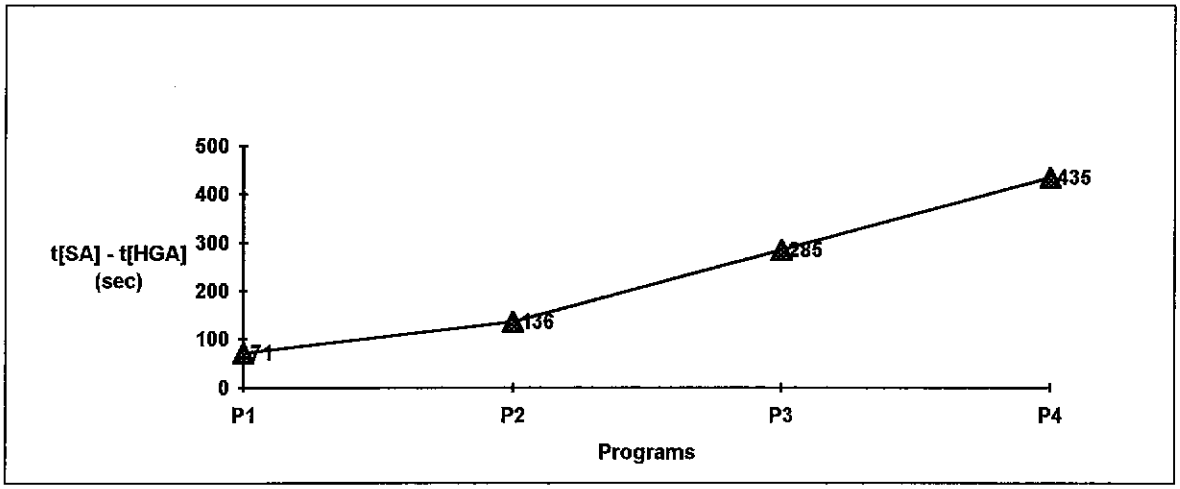


Figure 4. The difference in execution time between SA and HGA for programs in Table 3 .

	M(segments)	N(test cases)	Time(seconds) SA	HGA
P_5	80	1000	162	27
P_6	80	2000	335	37
P_7	80	4000	664	81
P_8	80	6000	997	132

Table 4. Program level results for programs P_5-P_8.

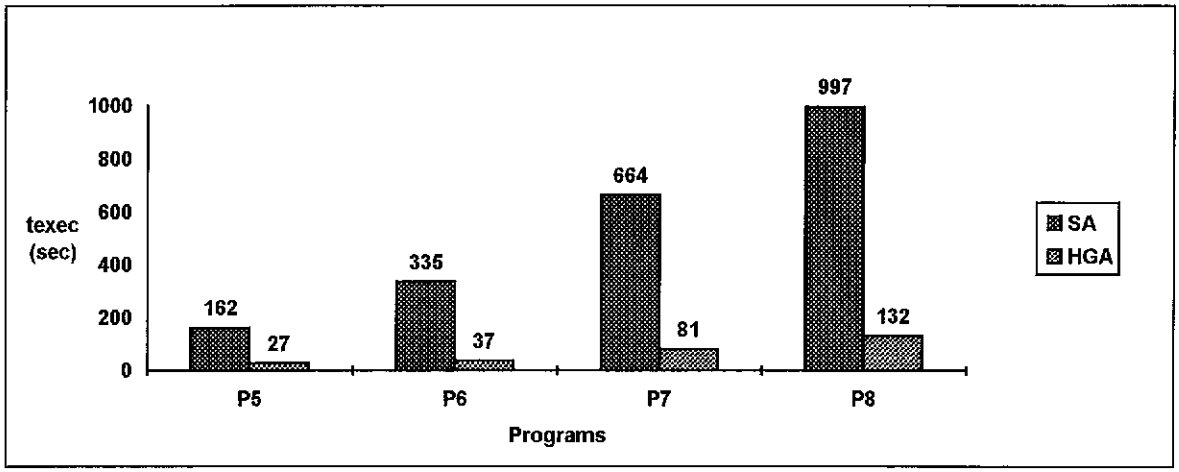


Figure 5. The execution times of SA and HGA for programs in Table 4.

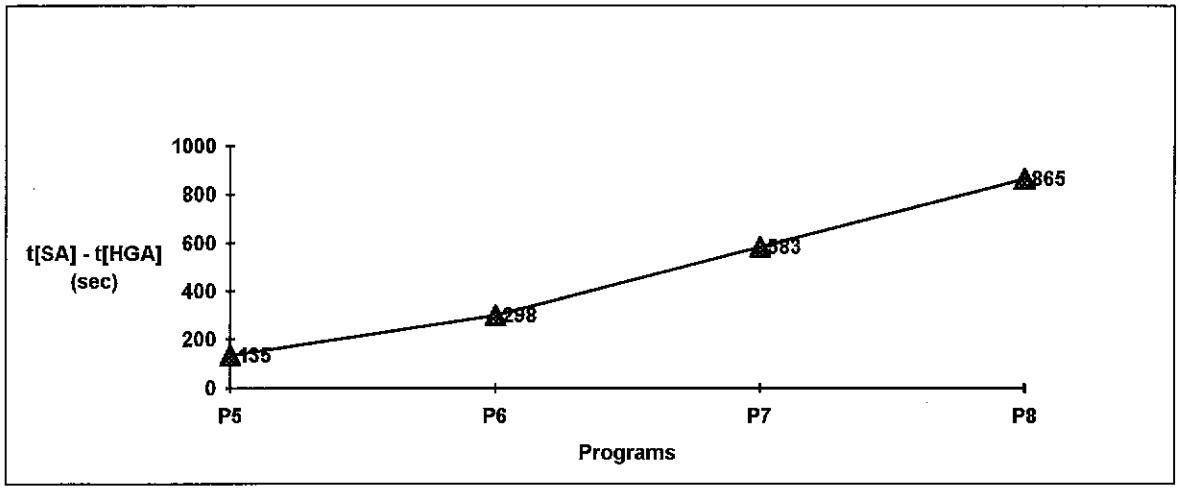


Figure 6. The difference in execution time between SA and HGA for programs in Table 4.

			Time(seconds)	
	M(segments)	N(test cases)	SA	HGA
P_9	200	1000	329	45
P_{10}	200	2000	659	59
P_{11}	200	4000	1325	211
P_{12}	200	6000	2002	581

Table 5. Program level results for programs P_9-P_{12}.

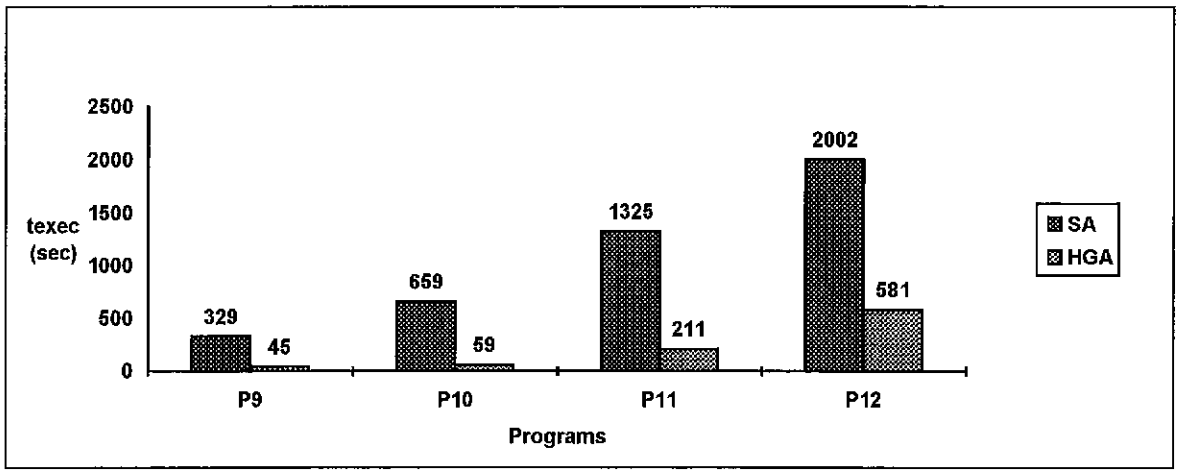


Figure 7. The execution times of SA and HGA for programs in Table 5.

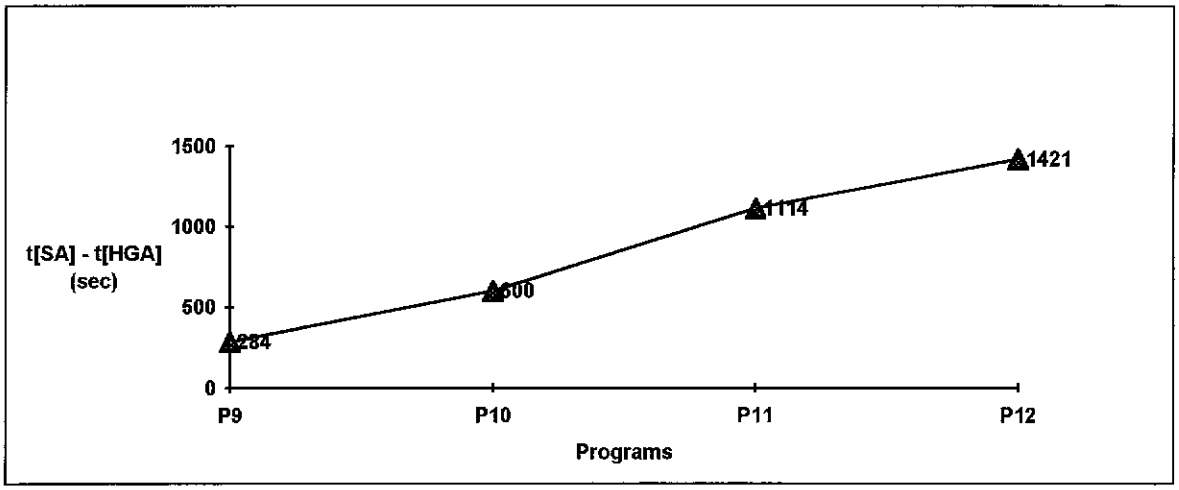


Figure 8. The difference in execution time between SA and HGA for programs in Table 5.

	M(segments)	N(test cases)	Time(seconds) SA	HGA
P_{13}	400	1000	600	83
P_{14}	400	2000	1213	336
P_{15}	400	4000	3560	1304

Table 6. Program level results for programs P_{13}-P_{15}.

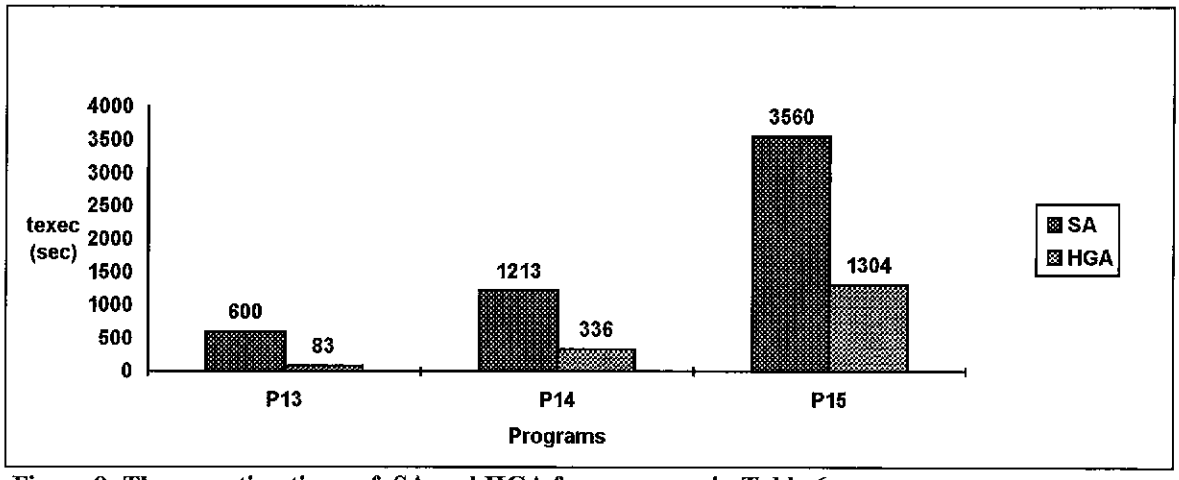


Figure 9. The execution times of SA and HGA for programs in Table 6.

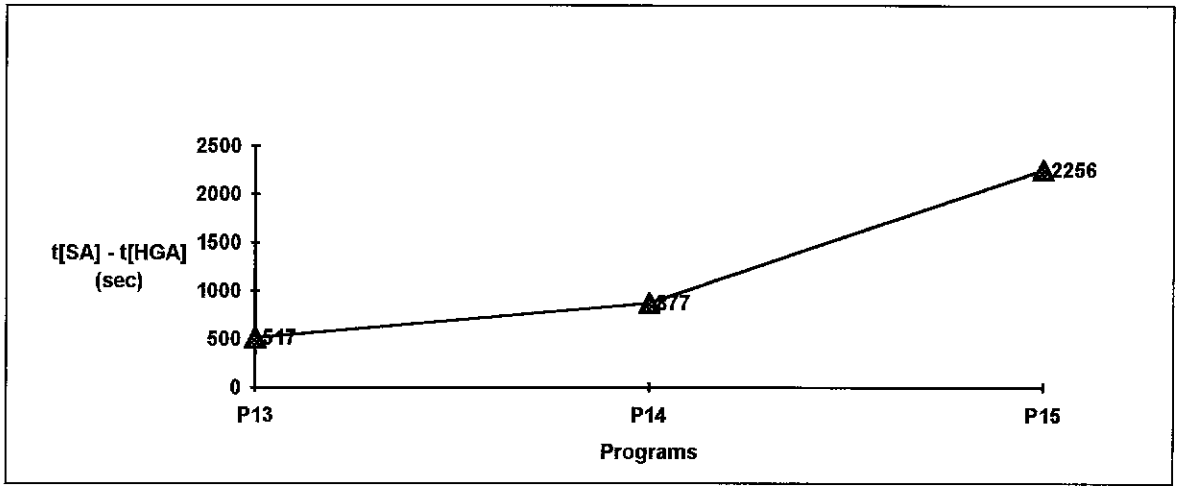


Figure 10. The difference in execution time between SA and HGA for programs in Table 6.

3. HGA Sensitivity to Population Size

The population size (POP) is one of the critical parameters in any GA. No general rule may determine exactly this problem dependent parameter, and the user has to set it before executing the GA. A small population leads to less diversity among individuals and possibly to fast convergence to a bad optimum. Nevertheless, a large population leads GA to be computationally expensive although they are more likely to converge to a good optimum.

Figure 11 shows the performance of HGA executed on P_8 for different POP sizes. A population of 8 individuals converges to the same solution as that of 40, but with less execution time. A POP size of 10 exhibits a better performance than that of 8 and 40.

The results of HGA on P_4 depicted in Figure 12 shows a fluctuating solution quality curve. A population of 8 individuals converges to the same solution as that of 40, but with less execution time.

Figures 13 through 15 show straight line-like solution quality curve which implies that HGA is not sensitive to the choice of POP size in these cases as far as the solution quality is concerned. However, the execution time curve shows an increasing trend which makes HGA computationally expensive for large POP sizes.

In short, Figures 11 through 15 show that a POP size of 8 individuals offers good solutions in acceptable time. For this reason with have used it for all program level experiments conducted in section 2.

				Time(seconds)
POP. Size	**M**(segments)	**N**(solutions)	**Solution**	**HGA**
8	80	6000	5	132
10	80	6000	4	694
40	80	6000	5	2354

Table 7. Performance summary of HGA using P_8 for different POP sizes.

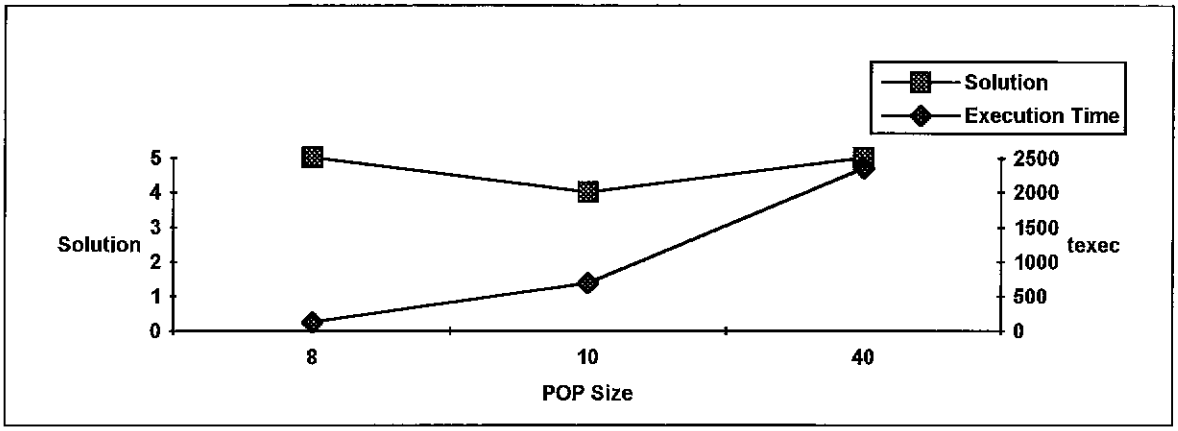


Figure 11. Performance of HGA for different Population Sizes using P_8.

				Time(seconds)
POP. Size	**M**(segments)	**N**(solutions)	**Solution**	**HGA**
8	400	2000	6	336
10	400	2000	7	235
20	400	2000	6	1809

Table 8. Performance summary of HGA using P_{14} for different POP sizes.

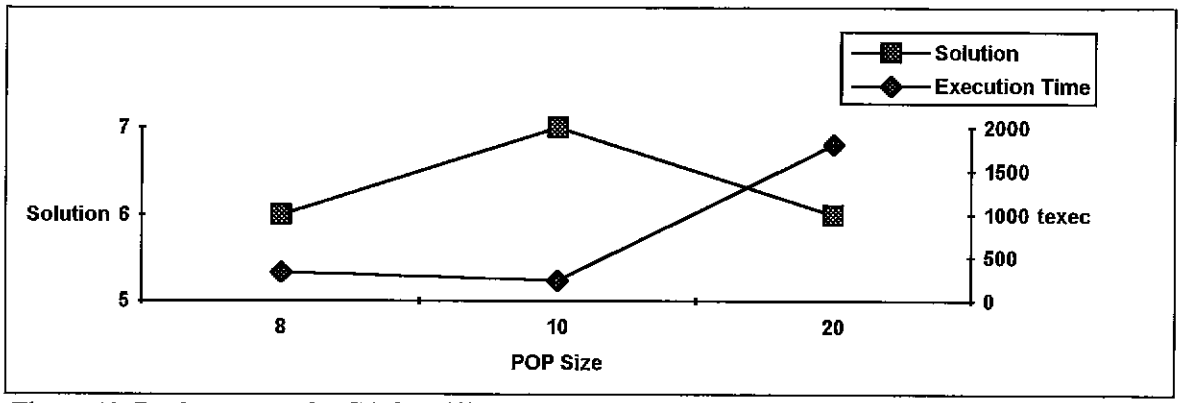


Figure 12. Performance of HGA for different Population Sizes using P_{14}.

POP. Size	M(segments)	N(test cases)	Solution	Time(seconds) HGA
8	80	2000	5	37
10	80	2000	5	42
20	80	2000	5	83
80	80	2000	4	1264

Table 9. Performance summary of HGA using P_5 for different POP sizes.

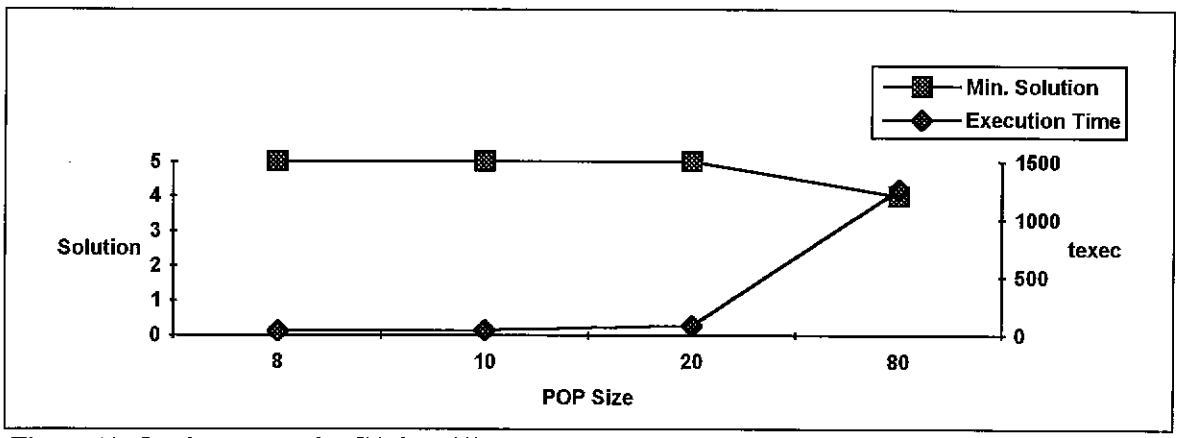


Figure 13. Performance of HGA for different Population Sizes using P_5.

POP. Size	M(segments)	N(solutions)	Solution	Time(seconds) HGA
8	20	4000	3	24
10	20	4000	3	46
20	20	4000	3	56
40	20	4000	3	135

Table 10. Performance summary of HGA using P_3 for different POP sizes.

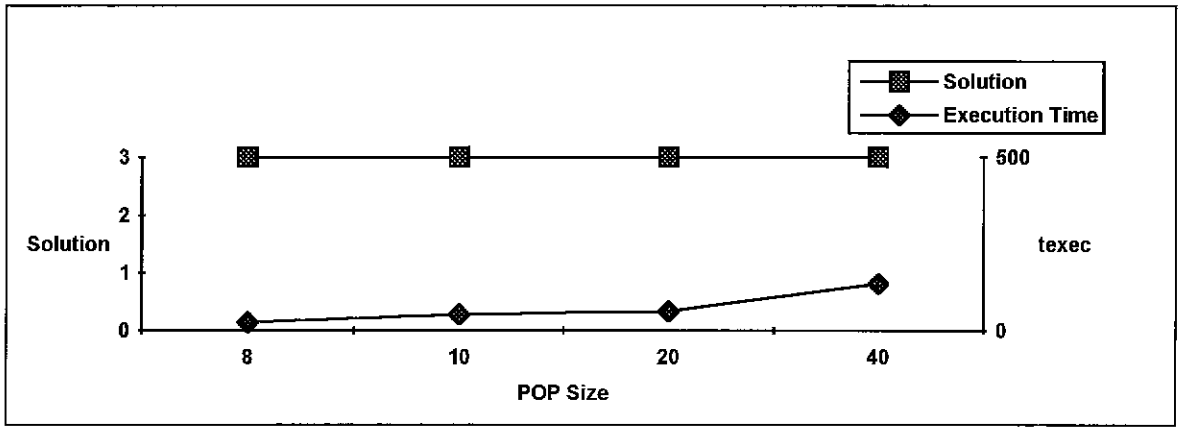


Figure 14. Performance of HGA for different Population Sizes using P_3.

POP. Size	M(segments)	N(solutions)	Solution	Time(seconds) HGA
8	20	6000	2	38
10	20	6000	2	44
20	20	6000	2	393

Table 11. Performance summary of HGA using P_4 for different POP sizes.

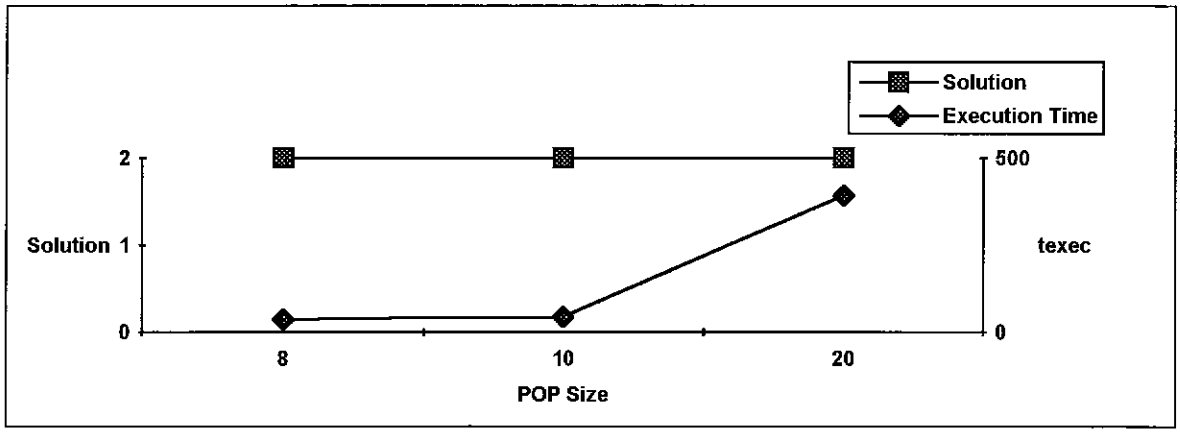


Figure 15. Performance of HGA for different Population Sizes using P_4.

4. Concluding Remarks

In this chapter, a comparative study was conducted between our HGA and B&B and SA algorithms. The experiments were done at both the module and program levels. While the three algorithms converged to the same solutions, it has been shown that HGA is faster than both B&B and SA.

Chapter 5

Approaches to Regression Testing

This chapter examines current approaches to regression testing and investigates their assumptions, limitations, and implementation levels. Moreover, software testing methodologies that are related to regression testing as dynamic testing and static testing, as well as overviews of related techniques such as black-box and white-box testing are described below.

In static analysis techniques, the structure of the code is analyzed but the code itself is not executed . It provides the tester with global information concerning the program structure. Hence, it is particularly well suited for finding logical errors [Hartmann and Robson 1988]. However, in dynamic testing strategies the test cases are executed and the results are evaluated to investigate the run-time behavior of the program. In general, dynamic testing techniques include function testing (black-box) and structural (white-box) testing.

Black-box testing (or data driven, or input-output) testing, is based on the concept that one may regard the software under test as a black box and does not need to be concerned

with the internal structure and behavior of the program. The only interest is in determining whether there are instances in which the program does not conform to its specifications. However, in function testing the internal structure and logic of the software under test is observed and analyzed.

1. A Survey of Existing Regression Testing Methodologies

Harrold and Soffa developed an incremental data-flow tester which can be used for regression testing purposes [Harrold and Soffa 1988]. The tool combibes data-flow testing with incremental data-flow analysis to aid in unit and integration regression testing. During the initial testing, the system stores the previous data-flow analysis results and test cases. After a module is modified, the system analyzes the effects of the changes on the test history and determines new and existing definition-use pairs for retesting. Existing test cases are reused whenever possible, and the system identifies those definition-use pairs that have not been exercised by any test case. However, this tool does not help with test generation and uses only a white-box testing method.

A similar approach based on semantic differencing has been recently proposed in [Binkley 1992]. In this technique, programs are represented by dependence graphs, not control-flow graphs, and program slicing [Gallagher and Lyle 1991, Weiser 1984] is used to partition the new program into preserved points and affected points. The test cases selected for retesting are those that test the behavior of the affected points. These test

cases can be run on a smaller program produced by semantic differencing, which captures the behavior of the affected points only.

In 1993, Harrold and Soffa presented a new methodology for controlling the size of test suite at the unit level [Harrold and Soffa 1993]. This technique selects a representative set of test cases from a test suite that provides the same coverage criteria as the entire test suite. This selection is performed by identifying then eliminating, the redundant and obsolete test cases in the test suite. The representative set could also be used to identify those test cases that should be rerun to test the program after it has been changed. The technique is independent of any testing methodology and it only requires an association between a testing requirement and the test cases that satisfy the requirement. Two basic approaches : a) incremental [Harrold and Soffa 1988] and b) retest-all [Leung and White 1991] could benefit from this technique which could be integrated into a testing methodology or be used as a stand-alone tool.

Rothermel and Harrold have implemented a selective regression testing algorithm [Rothermel and Harrold 1993]. The algorithm constructs control-dependency graphs for program versions, and uses these graphs to determine which tests from the existing test suite may exhibit changed behavior on the new version. The algorithm selects from the initial test suite every test that might expose errors in the modified program, and does this without prior knowledge of program modifications. Their technique is also useful for selecting obsolete test cases, as for identifying regions in the modified code where

coverage may need to be established through generation of new test cases. Unlike most selective revalidation algorithms, this algorithm is language construct independent, and it also covers new or deleted code edits. However, the algorithm does not distinguish between semantic and syntactic changes that could caught real behavior differences; thus a small swap statement between two unrelated assignments could be caught by the algorithm as a real change in behavior where it should not be. Moreover, the algorithm could be used for inter-procedural regression testing although further exploration of inter-procedural applications and support are still needed for adequacy criteria.

A regression testing strategy based on input partitioning and cause-effect graphing of the program specification has been described in [Yau and Kishimoto 1987]. This strategy derives the input partitions of the modified program using the program specification and code. Then, each new or changed input partition is executed at least once. Symbolic execution is also used to identify the input partitions that are not modified and to aid in test cases generation. This strategy can handle any kind of modification, including changes to the programs control structure. Moreover, it is based on an input partition approach, which is suitable for program validation because it generates test cases by reflecting the changes in both the program specification and code. In practice, Yau and Kishimoto's technique didn't work well when executing algorithmic processes (such as sorting routines). Furthermore, since it is based on cause-effect graphing to represent the program specification and symbolic-execution techniques for test-case execution, this methods can

become complex and may produce a large output. Similar to most regression testing strategies, this strategy concentrates on unit regression testing.

Leung and White [Leung and White 1989] have suggested classifying the initial test cases as reusable, retestable, obsolete, and adding new-structural and new-specification test cases. Then, corrective regression testing may involve test cases from the reusable, retestable, and new-structural classes, whereas progressive regression testing may involve test cases from all five classes. The guiding principle of this strategy was to view the regression testing problem as composed of two sub problems : the test selection problem and the test plan-update problem, and to structure the retesting process into two phases : the test classification phase and the test update phase.

In 1990, Leung and White have extended their work to cover programs with global variables [Leung and White 1990 a]. A basis set of testing problems for parameters was identified, and the testing problem of global variables was mapped onto a combination of these basis. It has been shown that global variables could be treated like parameters for testing purposes and could be tested accordingly. In 1990, the same authors [Leung and White 1990 b] have proposed building 'fire-walls" to confine integration testing to a small set of modules rather then allowing it to spread to many other modules. The construction of a firewall involves the modules that are modified and their direct ascendants and direct descendants. Then in 1991, Leung and White presented a test cost

model and identified the conditions under which the selective strategy is more economical than retest-all strategy.

Recently, Leung and White [Leung and White 1992] have brought together their previous work on regression testing and provided some additional new work on regression testing methods for function testers and for global variables. The new methodology involves regression testing of modules where dependencies due to both control-flow and data-flow are taken into account. The control-flow dependency is modeled as a call-graph, and the fire-wall defined to include all affected modules which must be retested. Global variables are considered as the remaining data-flow dependency to be modeled; an approach to testing and regression testing these variables was also presented.

Recently, IBM is developing an internal regression testing tool called Test Manager based upon the fire-wall concept for integration testing described by [White et al. 1993]. The objective of test manager is to aid in unit testing, integration testing, and function testing levels, and to produce a reduced regression test set to verify software changes indicated by the user. This tool assumes that procedures are hierarchical, i.e, each procedure belongs to a unique model, and each module belongs to a unique component. Sharing of procedures between modules and components is still needed. This requires the modification of test manager to include non-hierarchical situations. Also, test manager does not accommodates the research that has been accomplished on by Lee and White [Lee and White 1992] on regression testing global variables and data-flow dependencies. Adding this capability of

regression testing variables to test manager is still needed, mainly at integration testing and function testing levels. Moreover, the tool is internal to IBM platforms, and extending it to include other platforms may not be possible.

A post maintenance testing system was described by [Benedusi et al. 1988], and also deals with unit regression testing. It implements a path testing strategy by selecting tests to exercise a specific set of paths. The regression testing strategy uses the following two steps : *(i)* identify those paths that have been added, those deleted, and those modified; *(ii)* update and re-run the test cases which exercise the modified and new paths. Algebraic expressions are used to represent the program before and after the program modification. By comparing these expressions using elementary algebraic operations, it is possible to classify paths affected by the modification into new, deleted, modified and unmodified paths. By storing test cases and their associated program paths in a table, it is straightforward to identify those tests needed to be re-run based on the change code.

Chapter 6

Conclusions and Further Research

A genetic algorithm has been presented for finding the minimum number of test cases that have to be rerun for regression testing a modified program. It is based on an integer programming problem formulation and on a control-flow graph representation of the program.

The algorithm is hybridized by some design choices to overcome the problem of infeasibility encountered in classical GAs and to improve the efficiency of the genetic search. These choices include elitist ranking, feasibilization, penalization, and a hybridized hill-climbing procedure. However, its main feature is that it can be applied to various module and program sizes; in contrast with mathematical optimization methods, its complexity does not grow exponentially with the program size. The application of the algorithm shows that it finds an optimal number of retests faster than the other known methods.

Based on the work presented in this thesis, further research tasks may involve :

(a) Developing a new regression testing tool that could be applied to corrective regression testing at both the module and program levels. This tool would construct the different types of matrices (illustrated in chapter 2) needed by our HGA.

(b) Exploring perfective and adaptive regression testing.

Appendix A

Possible Enhancements

This Appendix examines how data dependency analysis could be used to further reduce the size of the data needed for constructing the 0-1 integer model presented in chapter 2. Also, it includes Hartmann and Robsons' extensions of Fischer algorithm. [Fischer 1977; Hartmann and Robson 1990]. These extensions are applied to corrective regression testing to revalidate functions whose control structure have been changed. Moreover, the utilization of the purposes of the set/use matrices, the interface between our HGA and the tools that generate information needed by the 0-1 integer model are illustrated through an example.

1. Data Structures and Data Dependency

On general, Fischer's algorithm relies on both the control-flow and the data-flow analysis of the target program, in order to accumulate information concerning the connectivity, reachability, and test coverage of segments and their usage of both local and global variables. While the data represented by the connectivity, reachability, and test case dependency (or cross reference) matrices was presented in chapter 2, this section describes the other two matrices : module level set/use and the global variable set/use.

The module level set/use matrix is used to reduce the number of selected test cases associated with the modified module [Yau and Collofello 1978]. It records the usage of both global and local variables, and parameters in the different program segments, by placing an 'S', 'U', or 'X' in the matrix to represent a variable being either set, used, set-and-used, respectively.

The global variable set/use matrix is used to monitor the ripple effects of modifications by reflecting the status of each global variable, argument, parameter, and local variable. Global variables are included in the matrix to facilitate the identification of modules in which the global variables are used. Arguments are included for invocation of those modules whose arguments have change during the modification [Hartmann and Robson 1990]. The utilization of the purposes of the set/use matrices is illustrated through an example in section 3 of this Appendix.

2. Limitations of Fischer Algorithm

The Fischer methodology that has been defined only for the FORTRAN language has the following major limitations. *(i)* It is incapable of dealing with modifications done to a program control structure. *(ii)* The method has been discussed and demonstrated for segments within a program routine where the modifications are constrained to changes within a particular segment and only very limited approaches are made in the application of his technique outside the scope of the example routine [Hartmann and Robson 1989].

3. Extensions to the Fischer's Method as Proposed by Hartmann and Robson

The extended Fischer algorithm can be applied with corrective regression testing to revalidate either simple iterations undertaken with program segments, or functions whose control structure have been changed.

3.1. Modifications done with in a program segment

For modifications done with in a program segment the following applies :

a) After the modifications have been undertaken on a segment, the retest strategy can be applied with respect to the altered function first, and then as part of the overall program structure, resulting in the determination of a subset of test cases.

b) The set/use matrix for the modified function may then be updated to make possible adjustments to the status of any existing or new global/local variables.

c) The subset of test cases identified early may than be executed.

3.2. Modifications done within a program routine

For Modifications done within a program routine the following applies:

a) After the modifications have been undertaken on a particular function such that its control flow is altered, the retest strategy can first be applied as part of the overall program structure, and a subset of test cases is determined.

b) All the matrices described earlier are reestablished for just this function. A new set/use matrix is build as the modified source code is parsed, and any existing or new parameters and global/local variables, introduced by the calling function, are updated in the respective set/use matrix.

c) The subset of test cases which was identified earlier is now executed. With respect to the newly structured function, this will determine which segments are being covered by the existing test data, and thus any segments that remain uncovered thereafter must be accounted for by the generation of new test cases.

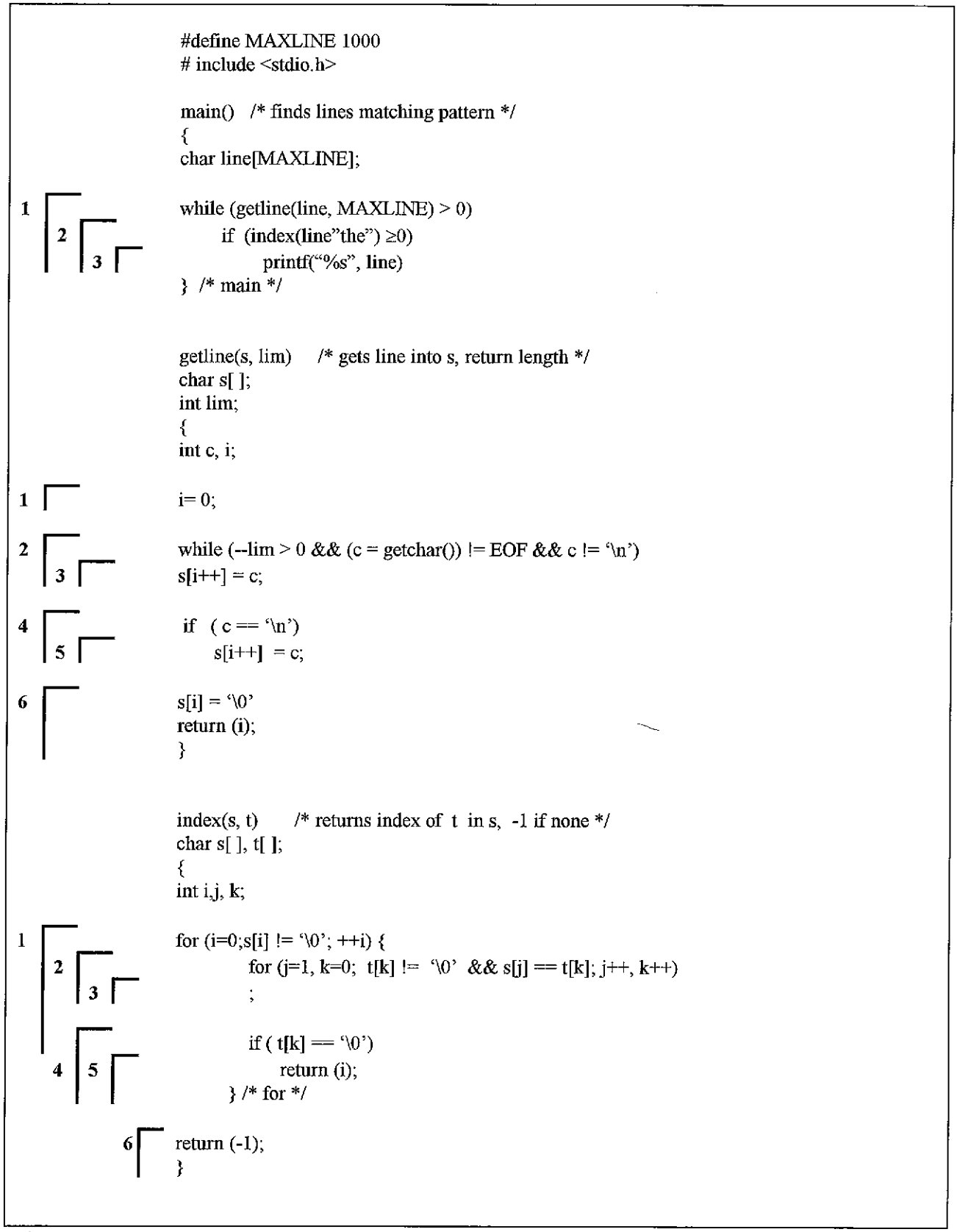
4. Example

The utilization of the purposes of the set/use matrices, the extension's to Fischer Algorithm, and the interface between our HGA and the tools that generate information needed by the 0-1 integer programming model can best be illustrated through this example.

Figure 1 shows a pattern matching program that searches a text for the string 'the', and prints any lines containing it on the standard output. The code in Figure 1 is annotated along the side of the source code with bold face values, which are the segment numbers of program functions. Figure 2 shows the corresponding control-flow graphs.

Let us assume that maintenance alteration has been made on segment 5 in the function *getline* of the program in Figure 1.

First, a tool from the prototype environment performs a control-flow analysis of the source code, segmenting the program and then establishing reachability between segments. Next, the tool parses the source code to establish segment interconnections, storing the results in the connectivity matrix. Then, it uses Warshall's algorithm [Warshal 1962] to generate the reachability matrix. Segments' test coverage in the test-case dependency matrix are created by a test-case generation tool. Another tool is used to complete the set/use matrices, which will contain parameter information and global and local variables. Figures 3 shows the generated connectivity, and reachability matrices for the functions *main* and *getline* and Figure 4 shows their corresponding module level set/use matrix. Whereas, Figure 8 contains the generated global variable set/use matrix for the global variable *line*.



```
#define MAXLINE 1000
# include <stdio.h>

main()   /* finds lines matching pattern */
{
char line[MAXLINE];

1  while (getline(line, MAXLINE) > 0)
2     if  (index(line"the") ≥0)
3        printf("%s", line)
} /* main */


getline(s, lim)     /* gets line into s, return length */
char s[ ];
int lim;
{
int c, i;

1  i= 0;

2  while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
3  s[i++] = c;

4  if  ( c == '\n')
5     s[i++]  = c;

6  s[i] = '\0'
   return (i);
   }


index(s, t)      /* returns index of  t  in s,  -1 if none */
char s[ ], t[ ];
{
int i,j, k;

1  for (i=0;s[i] != '\0'; ++i) {
2        for (j=1, k=0;  t[k] !=  '\0'  && s[j] == t[k]; j++, k++)
3        ;

4        if ( t[k] == '\0')
5           return (i);
      } /* for */

6  return (-1);
   }
```

Figure 1. An annotated listing of a pattern-matching C program.

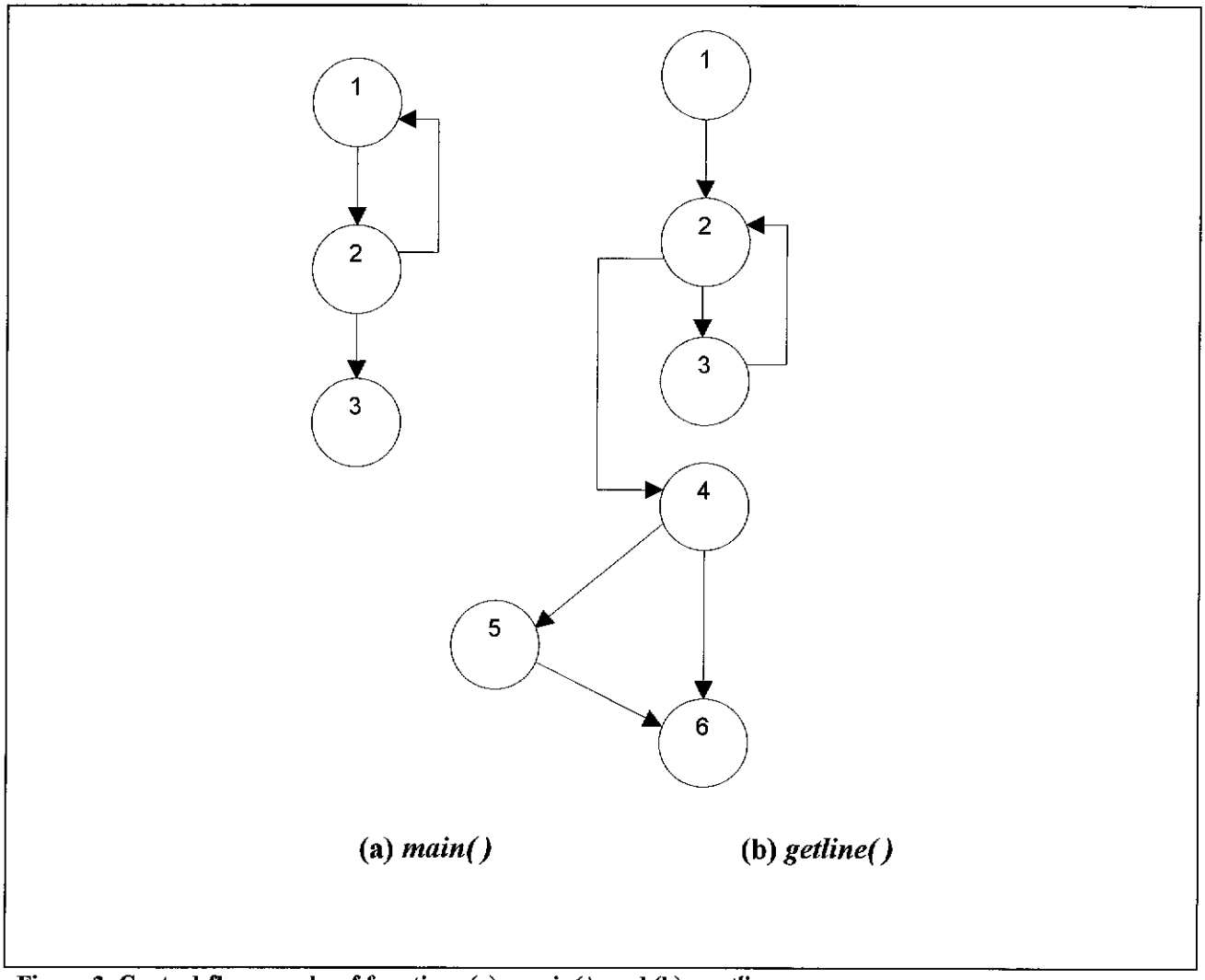


Figure 2. Control-flow graphs of functions (a) : *main( )*, and (b) : *getline.*

		To Seg 1	2	3
From Seg.	1	0	1	0
	2	1	0	1
	3	1	0	0

		To Seg 1	2	3
From Seg.	1	1	1	1
	2	1	1	1
	3	1	1	1

Line[ ]	Segs. S	U	U

Figure 3. Connectivity, reachability, and set/use matrices for the function *main()*..

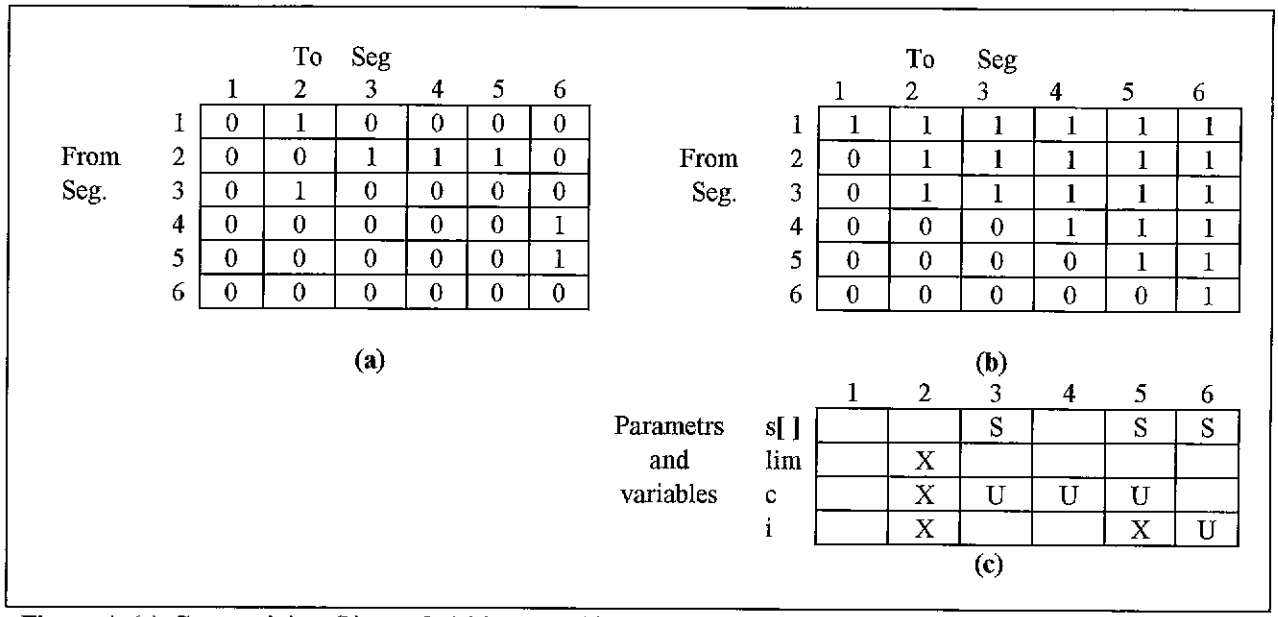


Figure 4. (a):Connectivity, (b):reachability, and (c):set/use matrices for the function *getline*..

Analysis of the module set/use matrix can now be used to reduce the number of selected test cases. This analysis is performed by the algorithms **A** and **B** presented by Fischer [Fischer 1981] to identify segments in which the data elements can-affect or be affected-by the modification. Algorithm **A** is used to determine all segments containing data elements which potentially affect data conditions used in the modified segment.

Algorithm **B** is used to determine all segments containing data elements which are potentially affected-by data conditions set by the modified segment.

By returning to modified module (function *getline*), the first step is to use the algorithm **A**. A logical AND is then performed between the result of the algorithm and column five (since segment 5 is modified) of the reachability matrix. This identifies the segments which reach-to and affect the modification of segment five. Figure 5 illustrates this analysis.

Column five of reachability matrix	1	1	1	1	1	0
Result of the Algorithm A	1	1	1	0	1	0
Logical AND	1	1	1	0	1	0

Figure 5. Data/logic dependencies reaching-to segment 5.

The second step is to identify segments reached-from the modified segment and data elements affected-by the modification. The analysis precedes in the same manner, except the logical AND is performed between the result of algorithm **B** and the fifth row of the reachability matrix. Figure 6, demonstrates the logical AND operation to determine the segments affected-by and reached-from the modified segment.

Row five of reachability matrix	0	0	0	0	1	1
Result of the Algorithm B	0	0	0	0	1	1
Logical AND	0	0	0	0	1	1

Figure 6. Data/logic dependencies reaching from segment 5.

Finally a logical OR between the result of step one and the result of step two is performed to identify the final $\mathbf{b}_i$s used in the 0-1 integer programming model. The 0-1 model then is solved by invoking our HGA and a set of test cases is selected. Figure 7 illustrates the logical OR operation.

Step one result (Figure 5)	1	1	1	0	1	0
Step two result (Figure 6)	0	0	0	0	1	1
Logical OR operation	1	1	1	0	0	1

Figure 7. Final b_i's used in the 0-1 integer programming model.

Now we illustrate the effects of modifications done on remote modules, and the use of the global variable set/use matrix to identify test cases that need to be rerun as a result of this modification. Since segment 5 in function *getline* is modified, this modification affects the parameter variable 'S' which has been used in this segment. The effects of this modification are reflected in the global variable set/use matrix (since 'S' maps to variable 'Line' in function *main*). By scanning the global variable set/use matrix, the modules which use the modified parameter variable can be identified. For example as shown in Figure 8, if the parameter variable 'S' is modified in module 5, then modules *main* and *index* are also identified for further analysis. Once the modules affected by the remote modification are identified, each individual module must be analyzed to determine the specific segments within the module which are affected-by the remote modification. This identification of segments is accomplished via the module set/use matrix because it maps variables (global, local, arguments) in a module to the segments of the module. Since segments affected-by the remote modification are identified, the procedure for selecting

test cases to be retested is the same as the procedure described in the example above. we merge the sets of test cases to produce the final test subset.

Global/Calling parameters variables		main	getline	index
	S/Line	X	S	U

Figure 8. Global variable set/use matrix for the global variable *line*.

5. Concluding Remarks

We have presented the extension's of Ficher's algorithm as proposed by Hartmann and Robson as the basis for corrective selective revalidation of functions whose control structure have been changed. Also, we have illustrated through an example the utilization purposes of the tools used to construct the different types of matrices needed by the 0-1 integer model.

References

Baker, J.E. (1985) Adaptive selective methods for genetic algorithms. *Int. Conf. on Genetic Algorithms*, 101-111.

Benedusi, P., Cimitile, A., and DeCarlini, U. (1988) Post-maintenance testing based on path change analysis. *IEEE Conference on Software Maintenane*, 352-368.

Bennet, K.H. (1991) The software maintenance of large software systems : management, methods and tools. *Reliability Engineering and Systems Safety*, 32, 135-154.

Binkley, D. (1992) Using semantic differencing to reduce the cost of regression testing. *IEEE Conference on Software Maintenane*, 41-50.

Brassard, G. and Brately, P. (1988) Algorithmics Theory and Practice. London : Printce-Hall International Inc.

Davis, R.E. and Kendrik D.A. (1971) A branch-and-bound algorithm for zero-one mixed integer programming problems. *Operations Research*, 19, 1036-44.

Easton, F. and Mansour, N. (1993) A distributed genetic algorithm for employee staffing and scheduling problems. Technical Report, School of Management, Syracuse University.

Fischer, K. (1977) A test selection method for the validation of software maintenance modifications. *IEEE Int. Conf. COMPSAC 77*, 421-426.

Fischer, K., Raji, F., and Chruscicki, A. (1981) A methodology for re-testing modified software. *National Telecommunications Conf.*, B6.3.1-B6.3.6.

Gallagher, K.B., and Lyle, J.R. (1991) Using program slicing in software maintenance. *IEEE Trans. Software Engineering*. 17(8), 752-761.

Goldberg D.E. (1989) Genetic Algorithms in Search, Optimaization and Machine Learning. Addison-Wesley.

Grefenstette J.J. (1986) Optimization of control parameters for genetic algorithms. *IEEE Trans. on Systems, Man, and Cybernetics*. 16(1), Jan-Feb, 122-128.

Harrold, M., and Soffa, M. (1988) An incremental approach to unit testing during maintenance. *IEEE Conference on Software Maintenane*. 362-367.

Harrold, M., Gupta, R., and Soffa, M. (1993) A methodology for controlling the size of a test suite. *ACM Trans. Software Engineering and Methodology*. July, 2(3), July, 270-285.

Hartmann, J., and Robson, D.J. (1989) Revalidation during the software maintenance phase. *IEEE Conference on Software Maintenane*. 70-79.

Hartmann, J., and Robson, D.J. (1990) Techniques for selective revalidation. *IEEE Conferance on Software Maintenance*. 31-36.

Holland J.H. (1975) Adaptation of Natural and Artificial Systems. Univ. of Michigan Press.

Horwitz, E., and Sahni, S. (1987) Fundamentals of Computer Algorithms. Computer Science Press.

IEEE (1983) Standard glossary of software engineering terminology. *IEEE Std.* 729-1983. IEEE Press.

Leung, H.K.N., and White, L. (1989) Insights into regression testing. *IEEE Conference on Software Maintenane*. 60-69.

Leung, H.K.N., and White, L. (1990 a) Insights into testing and regression testing global variables. *Software Maintenance : Research and Practice,* Vol. 2, 209-222.