212
C.1

TEST CASE GENERATION FOR PATH COVERAGE

by

Miran Salame

Submitted in partial fulfillment of the requirements

for the Degree of Master of Science

Thesis Advisor: **Dr. Nashaat Mansour**

Department of Computer Science

LEBANESE AMERICAN UNIVERSITY

June 2000

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDI ES

We hereby approve the thesis of

Miran Salame

candidate for the *Master of Science* degree.

(signed [redacted]

(chair)

M. Abboud [redacted]

Ramzi Haraty [redacted]

date 26- June- 2000

We also certify that written approval has been obtained for any proprietary material contained therein.

Abstract

Simulated Annealing (SA) and Genetic Algorithms (GA) have been proposed for generating integer-value test cases for path coverage. In this work, we suggest significant improvements to these algorithms and present empirical results that show their capabilities. The improvements cover algorithmic and implementation issues. They also add the capability of generating real-value subject programs for path coverage. We empirically compare the SA and GA algorithms with a hill-climbing algorithm, Korel's algorithm, for integer-value subject programs and compare SA and GA with each other on real-value subject programs. Our empirical work uses eight subject programs and a total of 49 paths. The results show that : (a) SA and GA are superior to Korel's algorithm in the number of covered paths, (b) SA tends to perform slightly better than GA in terms of the number of covered paths, and (c) GA is faster than SA; however, Korel, when it succeeds in finding the solution, is the fastest.

To my parents

Ali & Samira

My husband

Ayad

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Nashaat Mansour for his guidance throughout my M.S. studies. Thanks is also due to Dr. Abboud and Dr. Haraty for being on my committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my husband and my parents for their long support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing is an essential technique for assuring software quality, where it aims for giving confidence about the correctness of software. However, testing is an expensive phase of the software development life cycle in terms of labor and cost. Thus, it is important to develop automatable testing methods. In particular, automating the process of determining input data, which satisfy selected testing criteria, is important.

There are four levels of software testing: module, integration, system, and acceptance testing. This work is concerned with module testing, where different testing criteria lead to a variety of testing methods. These methods are usually classified as black-box and white-box (Beizer, 1990; Marciniak, et al., 1994). Black-box methods are based on the functions of the software, where test cases are derived from the module specifications (Hetzel, 1991). Examples of such techniques are equivalence partitioning, boundary value analysis, random testing, and functional analysis-based testing (Beizer, 1990; Duran and Ntafos, 1994; Hamlet and Taylor, 1990; Howden, 1986; Stocks and Carrington, 1993).

White-box methods are based on the internal structure of the software, where test cases are selected so that structural components are covered. Examples of such techniques are statement testing, branch testing, path testing, predicate testing, dataflow testing, structured testing, and domain testing (Jeng, 1994; Korel, 1992; McCabe, 1982; Ramamoorthy et al., 1976 ; Rapps and Weyuker, 1985; Tai, 1993). The most powerful form of structural testing is path coverage, that is testing all paths. A good survey of testing methods is found in (Zhu et al., 1997).

Automating test case generation remains a very challenging problem. Reported automated test case generators are usually path-oriented, goal-oriented, random, or assertion-oriented (Korel, 1990; Korel and Alyami, 1996; Ramamoorthyly et al., 1976). To find input test cases to execute a selected path, symbolic execution and execution-oriented methods have been proposed (Korel and Alyami, 1996). In this thesis, we are interested in path testing, which is a powerful structural testing strategy.

Path testing is a complicated problem. All of the previously presented techniques are difficult to use in practice for realistic programs and require a large amount of computation. For instance, the problem of test case generation is in general unsolvable in the sense that there does not exist an effecient algorithm that can be used to find assignments to input variables that will satisfy any given path predicate, or even to determine its satisfiability (Huang, 1975).

In this work, we have improved the previously developed algorithms for path testing, precisely, the Simulated Annealing Algorithm (SA) (Mansour and Joumaa, 1998) and the Genetic Algorithm (Joumaa, 1997). The previous work, only established the validity of the idea of using SA and GA algorithms for path coverage with integer-value test cases. However, their capabilities were not demonstrated. In addition to the significant improvements, we expanded their capacity to solve real-value input subject programs rather than being limited to the integer type solely. In addition to that, we implemented Korel's algorithm, which is a path coverage algorithm. We experimented with the three algorithms on the same subject programs and we compared their performance and capacity. This comparison has never been done before, and there is « no data generated previously that could be used to compare the effeciency and effectiveness of simulated approaches and genetic algorithms approches » (Pargas et al., 1999).

This thesis is organized as follows. Chapter 2 presents the problem formulation and the objective function used. Chapter 3 describes the simulated annealing algorithm and the improvements that were performed on this algorithm. Chapter 4 describes the improved genetic algorithm. Chapter 5 describes the technique that was used to enable GA and SA to handle real-value input data. Chapter 6 describes Korel's algorithm. Chapters 7 and 8 present the empirical results for integer and real data respectively. Chapter 9 gives our conclusions.

Chapter 2

Path Objective Function

Let a module be represented by a directed control-flow graph, where its nodes are either assignment nodes or decision nodes. Associated with each decision node is a branch predicate, which is a logical expression. The edges leaving such nodes are annotated with true/false values for the predicate. A path through the module/program corresponds to a particular flow of control. Therefore, a path test case is derived so that the path from the beginning to the end of the program will be traversed along either the true or the false branches. That is, the test cases should satisfy the path predicate, which is a concatenation of branch predicates.

A path predicate is usually obtained by dragging branch predicates backward from the exit graph node to the start node. Passing through an assignment statement, the predicate remains unchanged unless the variable on the left-hand side of the assignment operator occurs in the predicate. In the latter case, the variable is replaced by the expression of the right hand side of the assignment statement. The final path predicates constructed by using this technique are always expressed in terms of constants and input variables. Consequently, to cause a path to be traversed during execution, we should find the appropriate values to the input variables so that the obtained path predicate is satisfied.

The objective function we describe next assumes that each predicate in the concatenated path predicates is in equality format. We achieve this format by using the following rules:

- Remove all not connectives involved. If a predicate is of the form (not E1 R E2), then replace it with equivalent expression (E1 R' E2), where E1 and E2 are expressions, R is a conditional relation, and R' is the complement of R.
- Transform inequalities to equalities as follows:

$a \neq b$	$\Leftrightarrow$	$(\exists x)_{\neq 0}$	$(x = b - a)$
$a < b$	$\Leftrightarrow$	$(\exists x)_{>0}$	$(x = b - a)$
$a \leq b$	$\Leftrightarrow$	$(\exists x)_{\geq 0}$	$(x = b - a)$
$a > b$	$\Leftrightarrow$	$(\exists x)_{>0}$	$(x = a - b)$
$a \geq b$	$\Leftrightarrow$	$(\exists x)_{\geq 0}$	$(x = a - b)$

The process can be represented as a series of mappings. Initially the input variables
{x1, x2....., xn} are set at random and converted to binary format bi, and then manipulated by GA and SA. The bit strings are then converted back to integer numbers in order to evaluate the predicates of the required path.

As an example, consider a program *P* under test, *P* maps its input values to two values Xpk and X'pk corresponding to the evaluation of the functions on the two sides of each predicate *K*, which occur in the control flow of the selected path. Thus, if there are *m* predicates in the selected path, *P* will map the input {x1, x2....., xn} to m pairs of values {Xp1 and X'p1}... {Xpk and X'pk} ... {Xpm and X'pm}. The mapping will be different for each path corresponding to the input. The mapping will use the integer format of the input and not the binary one, since complex expressions may be computed to evaluate each side of the predicates (Jones, Sthamer et al., 1996).

After obtaining the values corresponding to both sides of a predicate *K* for example, these values are converted back to binary format and passed to the objective function. The objective function for this may be based on the hamming distance between the operands of each predicate. The hamming distance operator maps pairs of bit strings relating to the K^{th} predicate to a numeric value. It is a count based on the number of different bits in the bit patterns for the two operands. The hamming operator is normally weighted so that the least significant bit has a weighting of 1, the next 2 etc. This is necessary because if the most significant bits differ, then substantial changes to the input pattern are usually needed (Jones, Sthamer et al., 1996).

Thus, the objective function is obtained by the following formula.

$$\sum_{i=0}^{n} 2^i$$

Consider the following example where the K^{th} predicate is : $A = 4$, and where A is an input variable and not an expression. Assume that the GA or the SA algorithm generated in one of its paths the value 7 for A. Then, 7 is converted to binary format to obtain 111, 4 is also converted to binary format to obtain 100. The two strings differ in bit indices 0 and 1, consequently, the value of the objective function is 1+2 = 3

In the case of compound conditions, the value of the objective function for each predicate may be determined separately, and an overall value calculated by using multiplication for *or* predicates and addition for *and* predicates. Consider the following example, where the path predicate is : $a < b$ and $a + 1 < c$. Assume that the value of the objective function obtained for the first predicate ($a < b$) is 10, and that of the second predicate ($a+1<c$) is 20, then the overall objective function value would be 30=10+20 since the conjunction relating the two predicates is «*and*»

We add that the proposed algorithms that will be described in later sections can generate negative and positive numbers. Therfore, when the problem requires generation of only negative or positive values, then these constraints should be added to the path predicate already formed. This is essential, because these constraints are problem dependent and not general.

Moreover, the additional variables that will be added in order to convert inequalities to equalities are considered as input variables and are manipulated by the Simulated Annealing and Genetic Algorithms.

Chapter 3

Improved Simulated Annealing Algorithm

Simulated annealing is motivated by an analogy to statistical mechanics of annealing in solids (Rutenbar, 1989). To coerce some material into a low energy state, the material is annealed, it is heated to a temperature that permits many atomic rearrangements, then cooled carefully and slowly, allowing it to come to thermal equilibrium at each temperature, until the material freezes. A low energy state usually means a highly ordered state.

The simulated annealing algorithm simulates the natural phenomenon by a search process in the solution space optimizing some cost function. It starts with some initial solution at a high temperature and then reduces the temperature gradually to a freezing point. At each temperature, regions in the solution space are searched by the algorithm.

Simulated annealing proved to be successful in solving many optimization problems. In this section, we describe how simulated annealing can be adapted to solve not only optimization but also structural path testing problem. An outline of the SA algorithm is given in Figure 1.

```
Initial configuration;
Determine initial temperature T(0);
Determine freezing temperature $T_f$;
While (T(i) > $T_f$ ) do
        for i= 1 to CONST
                Perturb configuration;
                If (accept_perturbation) then
                        Accept perturbation();
                End if;
        End for;
Save best so far;
T(i+1) = $\alpha$ T(i);
End while;


Procedure accept_perturbation()
        If ($\Delta z$ <= 0) then
                Return (true);
        Else
                If (random(0,1) < $e^{-\Delta z/T(i)}$) then
                        Return (true);
                Else
                        Return (false);
                End if;
        End if;
```

Figure1. The SA Algorithm.

3.1. Solution Representation

An important decision in the simulated annealing algorithm is to present the sample test in a suitable form. In our work, we have found that the most efficient way to search the domain is to express all the input variables in a single, concatenated binary bit string. Therefore, if there are *N* input variables and the *i*th variable occupies n_i bits, the total size of the bit string is thus, $S = \sum n_i$ (i = 1, ..., N).

In our algorithms, n_i was constant for all types of variables and it was equal to 8. Since we were using binary representation, values for each variable were formed by using the signed integer representation.

Note that the input variables include also the auxiliary variables that are added to covert inequalities to equalities, and they were manipulated by the simulated annealing algorithm. To achieve the equalities the rules that were discussed in Chapter 2 should be followed.

3.2. Objective Function

A detailed discussion of the objective function and it's applicability on SA is provided in Chapter 2.

3.3. Perturbation Operation

First, the configuration is randomly initialized, starting and freezing temperatures are computed based on the fitness value of the initial population. Then, perturbation to the configuration takes place by randomly selecting a bit position from the configuration and changing its value to the opposite value; if the original value is 1 then it is modified to 0 and vice versa. It can be seen that the perturbation loop works for a constant number of iterations, CONST, which is dependent on the complexity of the problem. In our work, we use

CONST = 50 * number of bits in the configuration.

In each iteration the perturbation is either accepted or rejected (Rutenbar, 1989).

3.4. Accept/ Reject Criterion

The acceptance criterion is outlined in Figure 1 in the accept_configuration procedure. This procedure checks the change in the cost function due to a perturbation. If the change decreases the cost function, the configuration is accepted. However, if the perturbation causes the cost function to increase, it is accepted only with a probability $e^{-\Delta z/T(i)}$. Note that for smaller temperature values T(i), the probability of accepting uphill moves becomes smaller, while at very low temperatures uphill moves are no longer accepted (Rutenbar, 1989).

3.5. Cooling Schedule

The initial temperature T_0 is the temperature that yields a high initial acceptance probability of uphill moves. It is calculated by first accumulating the difference between the old objective function value and the new one when the newly obtained objective function value is higher than the one obtained on the previous value of the chromosome. By doing such an accumulation and counting the number of times when such an increase in the objective function has occurred, we will use these two values to compute the average objective function by dividing the accumulated value of the increase that occurred on the objective function over the frequency of such an increase. Afterwards, the negation of the average value is divided by the logarithmic value of the selected acceptance probability.

The freezing point is the temperature at which the acceptance probability (2^{-30}); thus, uphill moves becomes impossible; whilst, allowing only greedy moves. The cooling schedule used in this work is simply $T(i+1) = \alpha T(i)$, with $\alpha = 0.98$.

3.6. Convergence and Algorithm's Complexity

Convergence is reached when no improvement is obtained in the fitness function over 100 consecutive passes. However, as the annealing algorithm searches the solution space, the best so far solution found is always saved. This guarantees that the output of the algorithm is the best solution it finds regardless of the temperatures and solutions it terminates at. The SA algorithm's complexity is O(number of temperatures * (number of variables)^2).

3.7. SA Applied to the Right Angle Triangle

To clarify the ideas mentioned in the preceding sections, the conversion of inequalities into equalities and the computation of the objective function respectively, we will provide an example taken from test case T2; precisely, the **right angle** triangle.

SA works in a standard manner for all test cases; however, the difference resides in the test cases themselves. In this sense, the transformations of inequalities to equalities and the calculation of the objective function are problem dependent; i.e., vary from one test case to the other. In what follows we will show a tracing of the SA program execution flow, when applied on the right angle triangle. The program flow graph of this chosen test case is shown in Figure 4, path 1, 2, 6, 8, 9, 11, 12.

```
 While (trials < 2000) and (ans = 'Y')

     Initialize the configuration, oldpop, by randomly filling it with 1's and 0's

     Convert the randomly generated binary values of the variables into integer
     format and save it in all-val.

     Compute the initial starting temperature by using the following algorithm:

                 Loop (10 * lchrom ) times

                     Accumulate +ve $\Delta E$ and save it in sum;

                 Compute Average-$\Delta E$ = sum / #of +ve $\Delta E$

                 Compute initial temperature = -(Avg-$\Delta E$ / log 0.9)

             Compute the freezing temperature by using the following formula:

                 Tfreeze = - (INC_OBJ / EXPOF2_PROB * 0.301)

                         = - (0.01 / -30 * 0.0301) = 0.001

             Initialize best so far to be equal to oldpop

         While (no convergence) and (temp > tfreeze) and (pass < Maxpass = 1000)

                 While (inner_pass < config_length * 50)

                         Perturb and save new configuration in newpop;

                         Compute the new objective function that is calculated in
                         the following manner for this particular test case:
```

if we follow the flow graph in Figure 4

we need first to have $a \neq b$:

$a \neq b \Rightarrow a + \alpha = b$

Convert $a + \alpha$ into binary format

Convert b into binary format

$\text{Res1} = \sum 2^i$ where i is the index of the bits that differ

Then we need to have $b \neq c$:

$b \neq c \Rightarrow b + \beta = c$

convert $b + \beta$ to binary format

Convert c to binary format

$\text{Res2} = \sum 2^i$ where i is the index of the bits that differ

Finally we need to obtain $a^2 * b^2 = c^2$

Convert $a^2 * b^2$ to binary format

Convert c^2 to binary format

$\text{Res3} = \sum 2^i$ where i is the index of the bits that differ

```
Newpop.fitness = res1 + res2 + res3
If newpop.fitness – oldpop.fitness < 0.0 then
    Save the new population in oldpop;
    Save best so far fitness value;
    Save temperature at which best fitness value
```

```
            occurred;

            Save best so far configuration;

            Save best so far configuration's equivalence in
            integer  format;
else

            accept perturbation with a probability of $e^{-\Delta E/t}$

End; {for inner loop}

If no improvement in the value of the objective function

occurs

for 100 consecutive trials then convergence occurs and

SA

stops its exploratory attempts. The values that SA

ended

up with  are printed to the user. Accordingly, the user

 will have the choice to allow SA to be restarted with  a

new seed and  thus  with  newly generated random

variables.   This option is allowed to take place for 2000

  times

Else

Temperature is decreased by a factor of 0.98,

temp= temp *0.98
```

3.8. Improved SA

The improvements made to the previous SA algorithm/code were algorithmic and implementation. Their objective was to use a better energy function and to explore more points in the solution space. The algorithmic improvements included:

- Updating the cooling schedule in a way that allows the temperature to be decreased by a factor of 0.98 (instead of 0.95).
- At each temperature, perturbation takes place to explore the possibility of finding the solution. In order to let the SA algorithm perform even better and to explore more points in the solution space, we decreased the freezing temperature, at which SA is supposed to stop exploring more points, from 0.011 to 0.001.
- Implementing the concept of convergence, which is reached when no improvement in the value of the objective function is observed after 100 consecutive passes. In this way we avoid spending time in exploring points that are deemed to failure.
- Changing the number of perturbations allowed at each temperature to reach equilibrium. We made it a function of the configuration length (50 times the configuration length).
- Changing the initial temperature from a user-defined fixed value to one computed as explained in Section 3.5.
- Changing the objective function to a weighted sum (as shown in Chapter 2) instead of a linear sum.
- The new version of SA allows the algorithm to be rerun several times and with every new attempt, SA starts from a different seed specified by the amount to be added to the previous one. This way, SA is given the chance

to explore many different starting points, and accordingly will increase the possibilities of getting the appropriate solution.

After implementing these changes, we observed better results than the old version of SA in terms of achieving more accurate solutions. However, the execution time has increased as a normal consequence of such extensive exploration of the solution space.

Chapter 4

Improved Genetic Algorithm

Genetic algorithms are search algorithms based on the Darwinian principles of natural selection and natural genetics. These genetic algorithms were first developed by Holland (Holland, 1975). They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm. In every generation, a new set of artificial strings is created using bits and pieces of the fittest of the old. While randomized, genetic algorithms efficiently exploit historical information to speculate new search points with expected improved performance. In other words, genetic algorithm is an example of a search procedure that uses random choice as a tool to guide a highly exploitative search through a coding of a parameter space.

In this thesis, GA was modified to improve its efficiency and performance to achieve optimal results. In this regard, the new improved GA is now capable of solving real number based problems as compared to the previous version, which was solely limited to solving integer number based problems.

4.1. Genetic Algorithm

An outline of the genetic is presented in Figure 2.

```
Random generation of initial population, size **POP**;
    Repeat
        Evaluate fitness of individuals;
        Preserve the fittest so far;
        Rank individuals and allocate reproduction trials;
        for  i=1 to POP step 2
          Randomly select 2 parents from list of reproduction trials;
          Apply crossover and mutation;
        Endfor
    Until (convergence)
    Solution = fittest;
```

Fig 2. Genetic Algorithm

4.2. Genetic Algorithm Operators

The following operators mainly constitute the genetic algorithm:

1) Reproduction
2) Crossover
3) Mutation

4.2.1. Reproduction Scheme

Reproduction is a process in which individual strings are copied to their objective function values. Initially each variable is initialized randomly; therefore, generating a randomly chosen population. The subsequent generation is done by selecting the fitter individuals from the old generation. The reproduction scheme consists of ranking the individuals followed by random selection of mates. In ranking, the individuals are sorted by their fitness value and are assigned a number of copies according to a predefined scale of equidistant values for the population. The ranks assigned to the fittest and least fit individual are 1.2 and 0.8 respectively. Individuals which rank 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 possibilities for assignment of copies (Jones, Sthamer et al, 1996).

The GA re-inserts the best so far solution in the population size whenever there is no improvement in the solution. When a new population is obtained, the fitness value of the individual with the least fitness value is compared with the fitness of the best so far individual. If the fitness of the best so far individual is worst than that of the best individual in the new population, then the new individual replaces the best so far; otherwise, we replace the individual with the worst fitness function in the new population with the best so far individual.

4.2.2. Crossover

After reproduction, crossover proceeds in two steps. First, members of the newly reproduced strings in the mating pool are mated at random. Second, each pair of strings undergoes crossing over as follows: an integer position, *k*, along the string is selected uniformly at random, between 1 and the string length less one [1, l-1]. Two new strings are created by swapping all characters between positions k+1 and l inclusively. Accordingly, The mechanics of reproduction and crossover mainly consist of random number generation, string copies, and some partial string exchanges.

4.2.3. Mutation

Mutation is the occasional (mutation rate is 0.01) random alteration of the value of a string position. This simply means changing a 1 to a 0 and vice versa. In other words, mutation is a random walk through the string space.

4.3. Fitness Evaluation

We use a fitness function that is equal to the reciprocal of the objective function described in Chapter 2.

4.4. Convergence and Algorithm's Complexity

Convergence is reached when no improvement in the value of the fitness value is observed after 100 consecutive passes; (i.e., GA terminates its search in the solution space when no improvement is observed). In this way we avoid spending time in exploring points that are deemed to failure. The algorithm's complexity is

O(number of generations * pop * number of variables).

4.5. Improved GA

To better results and to improve the performance of GA, we made some modifications to the previous GA. The algorithmic improvements included:

- Using a corrected fitness function which is equal to the reciprocal of the objective function described in Chapter 2.
- Introducing convergence detection step to save the GA from exploring more points than what is useful in the solution space.
- The new version of GA allows the algorithm to be rerun several times and with every new attempt, GA starts from a different seed specified by the amount to be added to the previous one. This way, GA is given the chance to explore many different starting points, and accordingly will increase the possibilities of getting the appropriate solution.

After implementing such changes, we observed improvement in the solutions of GA, the results of which will be shown in Chapters 7 and 8.

Chapter 5

Enhanced SA and GA for Using Real-Value Input Data

To expand the capability of SA and GA in terms of dealing with data of type different from integer, we added new features on the improved version of SA and GA, enabling them to solve test cases, having real numbers as the variables data types. To enable SA and GA to generate variables of type real, we had to consider each real number as two separate parts, the ones before and after decimal point, then dealing with each part as a variable by itself. For this purpose we developed a new function that separates the real number into 2 different numbers. This is described in more detail in Section 5.1. After obtaining the parts before and after the decimal point, the objective function is then applied to these parts.

In case we have test cases that require arithmetic operations, we consider each variable as being composed of two parts; (i.e., two variables, one representing the part before the decimal point, and the other representing the part after the decimal point). Then, these parts are randomly generated, and a new function, responsible for combining these parts and returning a real variable value, is called. This function is described in detail in Section 5.2. After obtaining the variables as real numbers, we performed the arithmetic operations on the required variables. Then, the result of this operation was sent to the function that splits each variable into two different parts.

This was followed by applying the objective function to each part separately. A detailed example will be provided in Section 5.4 to clarify this idea.

5.1 Split Real Numbers

This function takes as an input a single variable of type real, and returns the parts before and after the decimal point as two different numbers. It works as follows:

1) Assign the real number X to an integer number. By doing this, we obtain the part before the decimal point, call it part *a*.
2) Subtract part *a* from X and save this value in part *b*.
3) Keep on multiplying part *b* by 10 until the ceiling of part *b* becomes equal to the number itself.
4) After exiting this loop, we will obtain the part after the decimal point.

Step 3 is needed because we can't predict how many digits are there after the decimal point, because it is randomly generated.

To clarify the idea behind this function, let's consider a simple example. Assume X=245.9867

After step one we will have part a = 245.

After step two, we will obtain part b = 245.9867 – 245 = 0.9867

```
Loop
  Part b = part b * 10
Until ceiling(part b) = part b.
```

When we exit this loop, we will get the part after the decimal part.

5.2 Return Real Numbers

This function is responsible for combining the two parts of a variable value, before and after decimal point. It then returns the variable value as a real number. This function works as follows:

It checks if the part after decimal is less than x , then it applies the formula:

real number = part before decimal + (part after decimal / x)

where x can be 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000, 1000000000

Thus, this function covers all the possible number of digits that could be generated randomly, and it finally returns one real number. For example, assume that a certain variable is composed of two parts that are 124 and 89,765 as the parts before and after decimal point respectively. When this function is called the part after the decimal point, which is 89,765, satisfies the fifth condition allowing this function to return the real number

124 + (89, 765 / 100,000) = 124.89765.

5.3 Solution Representation

In order to represent the solution we expressed all the input variables in a single, concatenated binary bit string, and we assumed that each input variable is equivalent to two input variables. Therefore, if there are N input variables (real numbers), then it is equivalent to $2N$ input variables. Accordingly, if we have N input variables it should be considered as $2N$ input variables, and the i^{th} variable is followed by *$i+1^{st}$* variable, which represent the parts before and after the decimal point respectively.

In other words, if we have *N* input variables of type real, then we consider having *2N* input variables and if we assume that the i^{th} variable occupies n_i bits, then the total size of the bit string is the summation of n_i (i = 1, ..., 2N). The n_i was assigned the value 8. In addition to the actual variables, we should consider the auxiliary variables that are needed to convert inequalities into equalities (Huang's technique). These auxiliary variables are computed in a way that considers the parts before and after the after the decimal point of the actual input variables as being different numbers. In this regard, if the equations in a certain test case require *n* auxiliary variables to convert inequalities into equalities, then *2n* auxiliary variables are needed to do such conversions. This is due to the fact that we're considering the two parts of the real numbers separately.

Finally, to represent the last variable values that either SA or GA end up with, we considered each two numbers and applied the function "Return Real Numbers" on these consecutive numbers. This process is repeated until all the variable values are considered and presented. However, this doesn't apply to the auxiliary variables that are still represented individually, (i.e., each number apart). To clarify how this newly developed version of SA/GA work when applied on real numbers, a detailed example is provided in the following section.

5.4 Example on New SA / New GA Applied to Real Numbers

We present an example about applying SA with real numbers. Assume we have a test case that requires $x + y < 2.5$. The execution of SA/GA when run on this test case is as follows:

Step1: Consider that *x* is composed of two parts, x1 and x2; (i.e., the parts before and after the decimal point respectively).

Step2: Consider that *y* is composed of two parts, y1 and y2; i.e., the parts before and after the decimal point respectively.

Step3: Transform $x + y < 2.5$ into the following 2 equations:

$$(x1 + y1) + \alpha = 2$$

$$(x2 + y2) + \beta = 5$$

We end up with having 4 input variables instead of 2 and 2 auxiliary variables instead of 1. In other word, we converted the function

$x + y < 2.5$ to $(x + y) + \delta = 2.5$ (2 input variables, 1 auxiliary variable)

Then $(x + y) + \delta = 2.5$ to $(x1 + y1) + \alpha = 2$

and $(x2 + y2) + \beta = 5$

x1, x2 are the parts before and after the decimal point respectively on the variable x.

y1, y2 are the parts before and after the decimal point respectively on the variable y.

Step4: Generate x1, x2, y1, y2, α, β in sequence by assigning random values to these variables and save it in the bit-string.

Step5: Combine x1 and x2 to get x by calling the function Return Real Numbers.

Step6: Combine y1 and y2 to get y by calling the function Return Real Numbers.

Step7: Add x and y, then save the result in z.

Step8: Split the real number z into two separate parts, z1 and z2, the parts before and after the decimal point respectively, by calling the function Split Real Number.

Step9: Add α to z1 in order to get $z1 + \alpha$.

Step10: Call the objective function to compute $z1 + \alpha = 2$ (i.e. $(x1 + y1) + \alpha = 2$).

Step11: Add β to z2 in order to get $z2 + \beta$.

Step12: Call the objective function to compute $z2 + \beta = 5$ (i.e. $(x2+y2) + \beta=5$).

Repeat steps 1 to 12 until x1, x2, y1, y2, α, β are generated in a manner that satisfies the above equations. Then, go to step 13.

Step13: To represent the solution consider (x1,x2) and call the function Return Real Number to obtain the final value of x. Now consider (y1,y2) to obtain y. In other word, repeat step 13 until all real actual input variables are presented.

Step14: Consider α then β by simply presenting them from within the holding array.

Chapter 6

Korel's Algorithm for Path Testing

Korel's technique focuses on pathwise test data generators that accept as input the computer program and a testing criterion and then automatically generate test data that meet the selected criterion. In Korel's method, test data are developed using actual values of input variables. When the program is executed on some input data, the program execution flow is monitored. If during program execution, an undesirable flow at some branch is observed then a real-valued function is associated with this branch. This function is positive when a branch predicate is false, and negative when the branch predicate is true (Korel, 1990).

The main objective is to minimize the branch function associated with the predicate that caused the target path not to be passed. To minimize this branch function, we start searching for a minimum with the first input variable while keeping all the other input variables constant until the solution is found; (i.e., the branch function becomes negative). Otherwise, we continue searching starting with the next input variables. The search proceeds in this manner until all input variables are explored in turn. After completing such a cycle, the procedure continuously cycles around the input variables until the solution is found or no progress is achieved; (i.e., decrement of the objective function can be made for any input variable). In the later case, the search process fails to find the solution. In the exploratory search, the selected input variable is increased or decreased by a small amount, while the remaining input variables are held constant. These are called the exploratory moves. For each variable change, the program is executed and the constraint is checked for possible violation by comparing successful sub-path P1 with the path that is actually being traversed. If P1 has been traversed, branch function $F_i(x)$ is evaluated for the new input. On the other hand, if P1 has not been traversed, the constraint violation is reported. In these exploratory moves, the value of the branch function is compared to the value of the branch function of the previous input.

In this way, it is thus possible to indicate a direction in which to proceed. If the branch function is improved (decreased) when x_j is decreased, the search should proceed in the direction of decreasing x_j. If both the decrement and increment of x_j do not cause the improvement of the branch function, the exploratory search fails to determine the direction for the search. In this case, the next input variable is selected for consideration. This process continues until the branch function becomes negative or no progress (decrement of the branch function) can be made for any input variable during the exploratory search. In the later case, the search procedure fails to find the solution to the test data generation problem (Korel, 1990). The method is outlined in Figure 4. The complexity of Korel's algorithm is

O(number of variables * max path length * number of trials per variables).

```
While (not passed) Do
Randomize;
Randomly select initial input (X0)
If P is traversed then
  X0 is the solution to the test data generation problem
Else
   Let T = <$t_{p1}$, $t_{p2}$, …, $t_{pz}$> be a program path traversed on X0
   Let P1 = < $n_{k1}$, $n_{k2}$, …, $n_{kp}$> be the longest subpath of P, referred to as a
   successful subpath of P on X0 such that for all j, $1 \le j \le i$ , $n_{kj} = t_{pj}$.
   The branch violation occurs on execution of branch ($n_{ki}$ , $n_{ki+1}$).

   Let $F_i(x)$ be a branch function of branch ($n_{ki}$ , $n_{ki+1}$). The first subgoal is to
   find a value of x which will preserve the traversal of P1 and cause $F_i(x)$
   to be negative or zero at $n_{ki} \rightarrow (n_{ki} , n_{ki+1})$ thus will be successfully
   executed. This process of solving subgoals is repeated until the solution
   to the main goal is found.

   Start searching for a minimum with the first input variable x.
   Keep all other input variables constant until the solution to the main
   goal is found
   i.e. branch function becomes negative.

   Continue the search with the next input variable $x_2$.
   The search proceeds in this manner until all input variables
   $x_1$, …, $x_n$ are explored in turn.

  After completing such a cycle, the procedure continuously cycles
  around the input variables until the solution is found or no
  progress (decrement of the branch function ) can be made for
  any input variable.
  In the later case, the search process fails to find the solution.

 The selected input variable $x_j$ is increased and decreased by a small amount
 Called exploratory moves.
 Check for this input two things:
        If subpath P1 is executed;
        If $F_i(x)$ is decreased;
    If these two conditions are satisfied then
        continue in the same direction of search
    Else if no search direction is found then
        take another variable and repeat the above process.
```

Fig 3. Korel's Algorithm.

Chapter 7

Empirical Results for Integer Data

In our experimental results we tested the four algorithms, SA, improved SA, Korel, GA, and improved GA on a number of subject programs. By doing this we were able to compare the obtained results and come up with a conclusion regarding the performance of each algorithm. The algorithms were developed using Turbo C under MS-DOS, and were run on Pentium with a 500 MHz memory. We tested these algorithms on a number of subject programs the description of which is found in Table 1.

Table 1

Programs	Cyclomatic Complexity	Description of Program
Triangle	5	This test case aims at classifying the type of a triangle. It reads the lengths of the three sides of a triangle, classifies the triangle as being scalene, isosceles, right, isoright, or equilateral. The flow graph of this program is presented in Figure 2.
Grading	6	The below program code segment computes a numerical score and then records the letter grade.
Role Dice	6	This subject program generates sum1 in a way that satisfies the game status.
Role Dice 2	10	This subject program generates sum1 and sum2 in a way that satisfies the game status.
Interview	14	This subject program generates subject, college, age and then decides whether to interview the candidate or not.
Answer	2	This subject program generates an answer then decides whether it satisfies the arithmetic operation or not.
Guess	3	It guesses a number then checks whether it's the magic number.
Compute	2	It generates two numbers then checks whether they satisfy a certain arithmetic operation.

Table 1. Subject Programs.

7.1. Subject Program 1

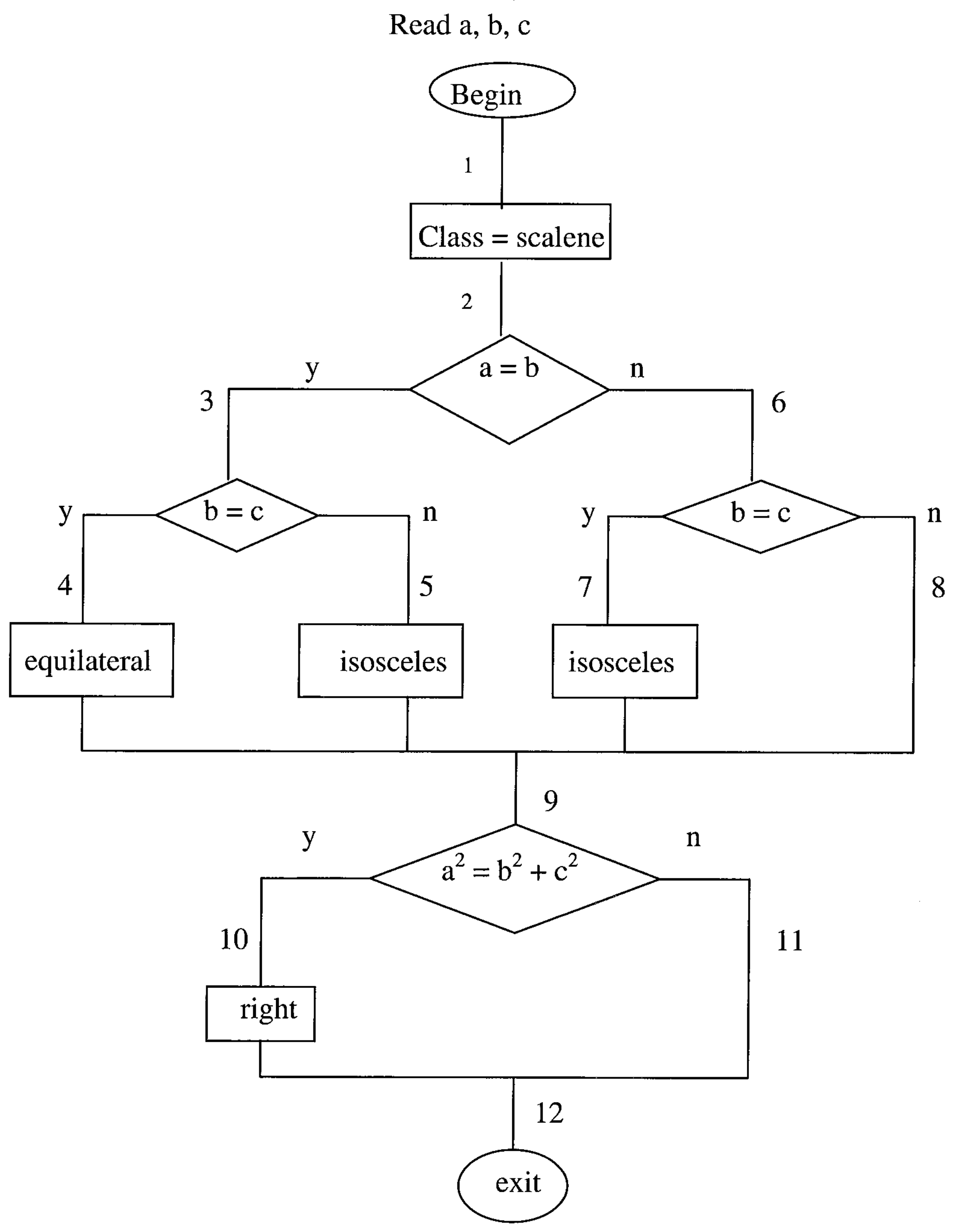

Figure 4 Flow Chart of Subject Program 1.

This subject program consists of the following paths:

Path1: 1, 2, 3, 5, 9, 11, 12 (isosceles with a = b)
Path2: 1, 2, 6, 7, 9, 11, 12 (isosceles with b = c)
Path3: 1, 2, 3, 4, 9, 11, 12 (equilateral)
Path4: 1, 2, 6, 8, 9, 10, 12 (right)
Path5: 1, 2, 6, 7, 9, 10, 12 (isosceles + right)
Path6: 1, 2, 6, 8, 9, 11, 12 (scalene)

Path1	**Covered**	**Values**	**Time**
SA	No	a=127 b=60 c=112	5.76sec
Improved SA	Yes	a=b=5 c=3	114sec
Korel	Yes	a=b=1 c=2	0sec
SA&Korel	Yes	a=b=2 c=1	0sec
GA	No	a=0, b=0, c=0	2.3sec
Improved GA	Yes	a=b=5, c=6	2.4sec

Table 2. Results of Path1 for Subject Program 1.

Path2	**Covered**	**Values**	**Time**
SA	Yes	a=41 b=c=30	5.6sec
Improved SA	Yes	a=4 b=c=7	114sec
Korel	Yes	a=3 b=c=1	0sec
SA&Korel	Yes	a=3 b=c=1	0sec
GA	Yes	a=4, b=4,c=4	2.3sec
Improved GA	Yes	a=15, b=c=13	2.41sec

Table 3. Results of Path2 for Subject Program 1.

Path3	**Covered**	**Values**	**Time**
SA	No	a=125 b=108 c=63	5.27sec
Improved SA	Yes	a=b=c=1	82 sec
Korel	Yes	a=b=c=1	0sec
SA&Korel	Yes	a=b=c=2	0sec
GA	No	a=5, b=4, c=4	2.3sec
Improved GA	Yes	a=b=c=2	2.4sec

Table 4. Results of Path3 for Subject Program 1.

Path4	**Covered**	**Values**	**Time**
SA	No	a=126 b=78 c=99	5.82sec
Improved SA	Yes	a=29 b=20 c=21	105sec
Korel	No	A=1516 b=11763 c=25890	1.15sec
SA&Korel	Yes	a=5 b=3 c=4	122sec
GA	No	a=1, b=0, c=1	2.4sec
Improved GA	Yes	a=5, b=3, c=4	2.4sec

Table 5. Results of Path4 for Subject Program1.

Path5	**Covered**	**Values**	**Time**
SA	No	a=95 b=90 c=31	5.21sec
Improved SA	No	a=6 b=c=4	80.5sec
Korel	No	a=3 b=c=2	1.098sec
SA&Korel	No	a=24 b=c=17	123sec
GA	No	a=1, b=1, c=0	2.4sec
Improved GA	No	a=116, b=80,c=84	2.4sec

Table 6. Results of Path5 for Subject Program1.

Path6	**Covered**	**Values**	**Time**
SA	Yes	a = 98 b= 47 c= 86	6.31sec
Improved SA	Yes	a = 3 b = 5 c= 4	128sec
Korel	Yes	a = 3 b = 2 c=1	0sec
SA&Korel	Yes	a = 3 b = 2 c=1	0sec
GA	No	a=b=2, c=5	2.3sec
Improved GA	Yes	a=4, b=5, c=8	2.5sec

Table 7. Results of Path6 for Subject Program1.

7.2. Subject Program 2

```
1.  #Right = 0;
2.  Do  j = 1 to #Questions;
3.  If key(j) = Answer(j) then #right = #right + 1;
5.  End
6.  Score = (#Right / #Questions) * 100
7.  If  Score >= 90 then Grade = A;
9.  Else If  Score  >=  80 then Grade = B;
11. Else If Score >= 70 then Grade = C;
13. Else Grade = F;
```

Figure 5 Program Segment Code for Subject Program 2.

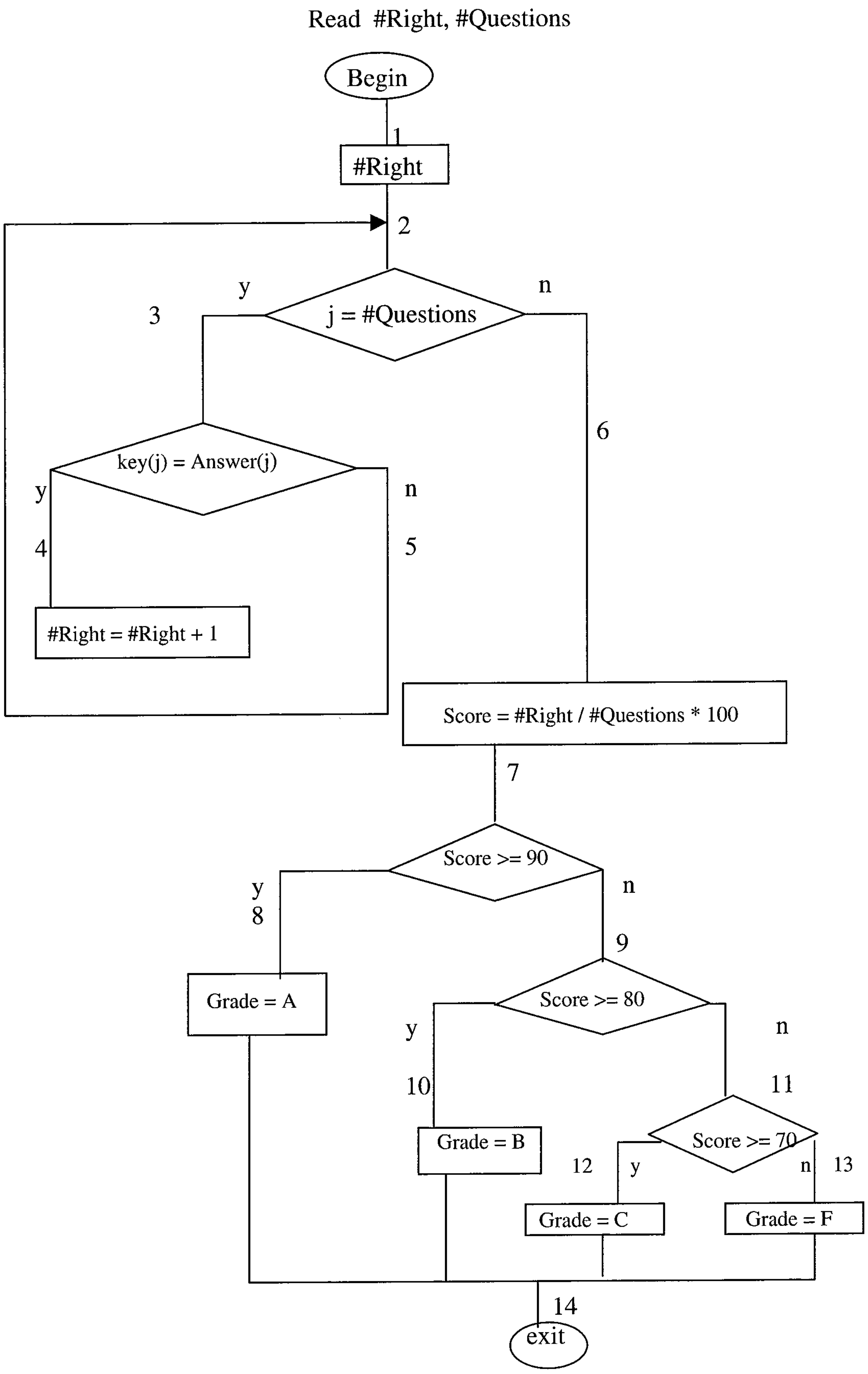

Figure 6 Flow Chart of Program Subject 2.

The following paths have been tested:

Path1: 1, 2, 6, 7, 8, 14	(Grade = A)
Path2: 1, 2, 6, 7, 9, 10, 14	(Grade = B)
Path3: 1, 2, 6, 7, 9, 11, 12, 14	(Grade = C)
Path4: 1, 2, 6, 7, 9, 11, 13, 14	(Grade = F)

Path1	**Covered**	**Values**	**Time**

SA	Yes	Questions = 102
		Right = 94
			1.37 sec

Improved SA	Yes	Questions = 120
		Right = 111
			43 sec

Korel	Yes	Questions = 1
		Right = 1
			0.05 sec

SA&Korel	Yes	Questions = 2
		Right = 2
			0 sec

GA	Yes	Questions = 112
		Right = 116
			2.3sec

Improved GA	Yes	Questions = 127
		Right = 118
			2.3sec

Table 8. Results of Path1 for Subject Program 2.

Path2	**Covered**	**Values**	**Time**

SA	No	Questions = 113
		Right = 103
			1.37sec

Improved SA	Yes	Questions = 120
		Right = 101
			40 sec

Korel	No	Questions = 3
		Right = 2
			2.14 sec

SA&Korel	Yes	Questions = 39
		Right = 32
			50 sec

GA	Yes	Questions = 94
		Right = 78
			2.3sec

Improved GA	Yes	Questions = 85
		Right = 72
			2.3sec

Table 9. Results of Path2 for Subject Program 2.

Path3	Covered	Values	Time

SA	Yes	Questions = 94
		Right = 75
			1.37sec

Improved SA	Yes	Questions = 91
		Right = 70
			40 sec

Korel	No	Questions = 3
		Right = 2
			18.24sec

SA&Korel	Yes	Questions = 114
		Right = 87
			76sec

GA	Yes	Questions = 113
		Right = 85
			2.4sec

Improved GA	Yes	Questions = 71
		Right = 55
			2.3sec

Table 10. Results of Path3 for Subject Program 2.

Path4	Covered	Values	Time

SA	Yes	Questions = 20
		Right = 7
			0.82sec

Improved SA	Yes	Questions = 73
		Right = 16
			4 sec

Korel	Yes	Questions =2
		Right = 1
			0 sec

SA&Korel	Yes	Questions = 3
		Right = 2
			0 sec

GA	Yes	Questions = 85
		Right = 4
			2.3sec

Improved GA	Yes	Questions = 104
		Right = 5
			1.9sec

Table 11. Results of Path4 for Subject Program 2.

7.3. Subject Program 3

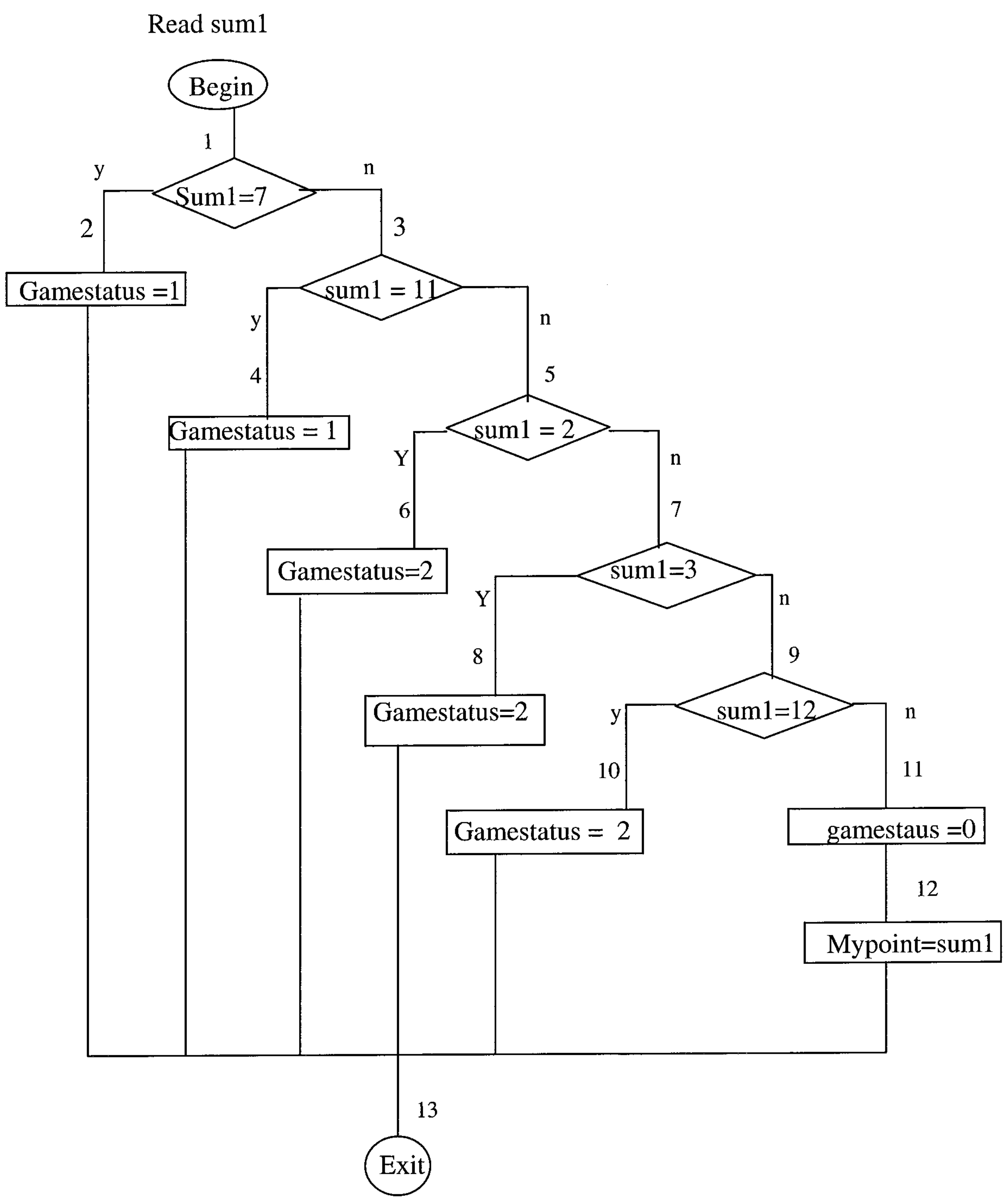

Figure 7 Flow Chart of Subject Program 3 (Roledice part I).

This subject program consists of the following paths:

Path1: 1, 2, 13
Path2: 1, 3, 4,13
Path3: 1, 3, 5, 6, 13
Path4: 1, 3, 5, 7, 8, 13
Path5: 1, 3, 5, 7, 9, 10, 13
Path6: 1, 3, 5, 7, 9, 11, 12, 13

Path1	**Covered**	**Values**	**Time**
SA	Yes	Sum1=7	1.9sec
Improved SA	Yes	Sum1=7	7.7sec
Korel	Yes	Sum1=7	0sec
SA&Korel	Yes	Sum1=7	0sec
GA	Yes	Sum1=7	1.9sec
Improved GA	Yes	Sum1=7	1.6sec

Table 12. Results of Path1 for Subject Program 3.

Path2	**Covered**	**Values**	**Time**
SA	Yes	Sum1=11	2.2sec
Improved SA	Yes	Sum1=11	0.7sec
Korel	No	Sum1=2	0.6sec
SA&Korel	Yes	Sum1=11	6sec
GA	Yes	Sum1=11	2.2sec
Improved GA	Yes	Sum1=11	1.9sec

Table 13. Results of Path2 for Subject Program 3.

Path3	**Covered**	**Values**	**Time**
SA	Yes	Sum1=2	1.2
Improved SA	Yes	Sum1=2	16.9sec
Korel	Yes	Sum1=2	0sec
SA&Korel	Yes	Sum1=2	0sec
GA	Yes	Sum1=2	2.5sec
Improved GA	Yes	Sum1=2	2.2sec

Table 14. Results of Path3 for Subject Program 3.

Path4	**Covered**	**Values**	**Time**
SA	Yes	Sum1=3	1.5sec
Improved SA	Yes	Sum1=3	85sec
Korel	Yes	Sum1=3	0sec
SA&Korel	Yes	Sum1=3	0sec
GA	Yes	Sum1=3	2.7sec
Improved GA	Yes	Sum1=3	2.5sec

Table 15. Results of Path4 for Subject Program 3.

Path5	**Covered**	**Values**	**Time**
SA	No	Sum1=28	1.8sec
Improved SA	Yes	Sum1=12	139sec
Korel	No	Sum1=4	0.6sec
SA&Korel	Yes	Sum1=12	138sec
GA	No	Sum1=57	3.07sec
Improved GA	No	Sum1=30	2.8sec

Table 16. Results of Path5 for Subject Program 3.

Path6	**Covered**	**Values**	**Time**
SA	Yes	Sum1=119	1.8sec
Improved SA	Yes	Sum1=127	164sec
Korel	Yes	Sum1=4	0.6sec
SA&Korel	Yes	Sum1=0	0sec
GA	Yes	Sum1=112	3.07sec
Improved GA	Yes	Sum1=125	2.8sec

Table 17. Results of Path6 for Subject Program 3.

7.4 Subject Program 4

Read sum1,sum2

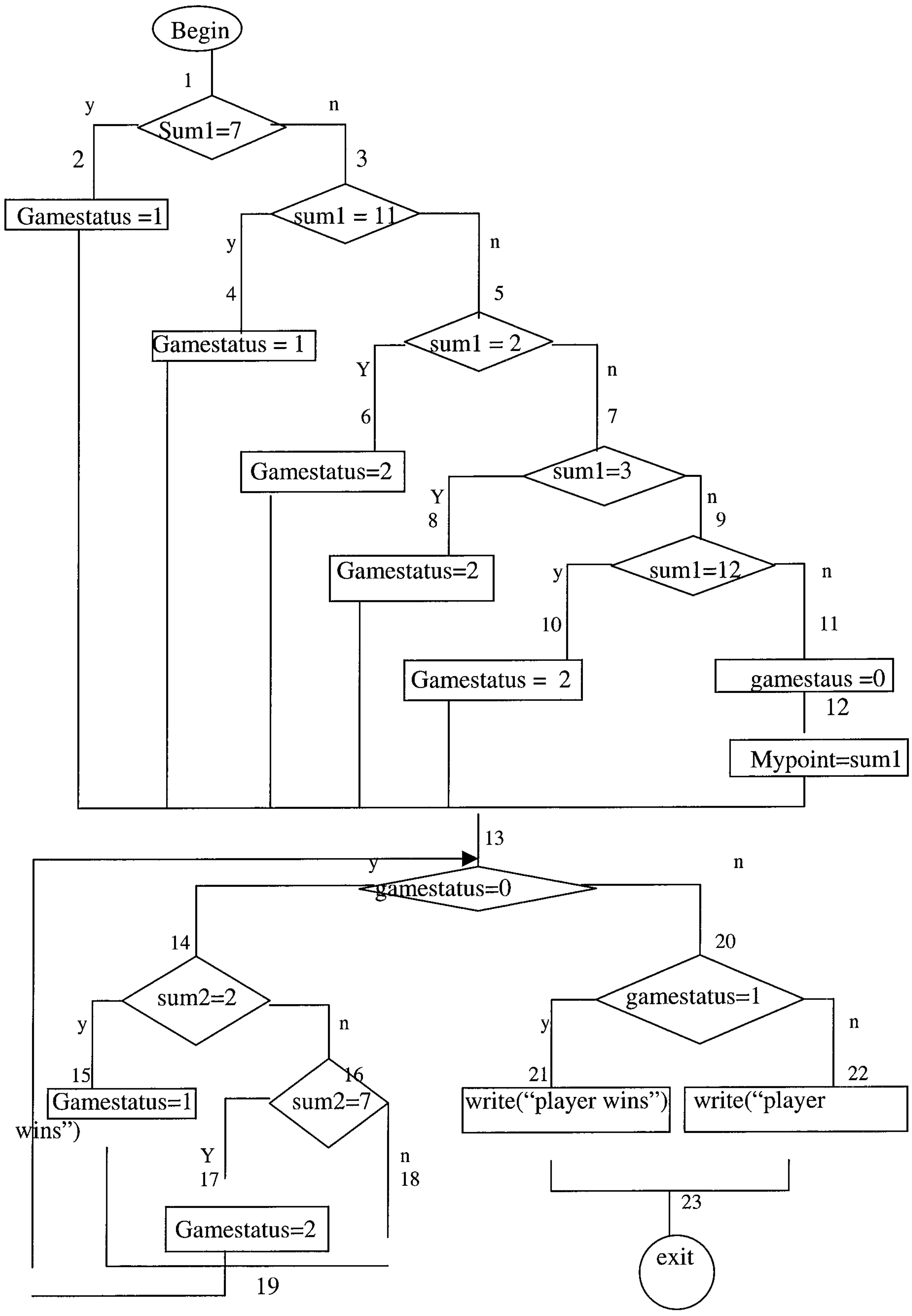

Figure 8 Flow Chart of Subject Program 4 (Roledice partII).

This subject program subject consists of the following paths:

Path1: 1, 2, 13, 20, 21, 23
Path2: 1, 3, 4, 13, 20, 21, 23
Path3: 1, 3, 5, 6, 13, 20, 22, 23
Path4: 1, 3, 5, 7, 8, 13, 20, 22, 23
Path5: 1, 3, 5, 7, 9, 10, 13, 20, 22, 23
Path6: 1,3,5,7,9,11,12,13,14,15,19,20,21,23
Path7: 1,3,5,7,9,11,12,13,14,16,17,19,20,22,23

Path1	**Covered**	**Values**	**Time**

SA	Yes	Sum1=7
		Sum2=10
			0.7sec

Improved SA	Yes	Sum1=7
		Sum2=25
			9.3sec

Korel	Yes	Sum1=7
		Sum2=1
			0sec

SA&Korel	Yes	Sum1=7
		Sum2=3
			0sec

GA	Yes	Sum1=7
		Sum2=93
			2sec

Improved GA	Yes	Sum1=7
		Sum2=33
			1.8sec

Table 18. Results of Path1 for Subject Program 4.

Path2	**Covered**	**Values**	**Time**

SA	Yes	Sum1=11
		Sum2=105
			1.09sec

Improved SA	Yes	Sum1=11
		Sum2=17
			23.5sec

Korel	No	Sum1=2
		Sum2=2
			0.9sec

SA&Korel	Yes	Sum1=11
		Sum2=66
			34sec

GA	Yes	Sum1=11
		Sum2=102
			2.4sec

Improved GA	Yes	Sum1=11
		Sum2=1
			2.1sec

Table 19. Results of Path2 for Subject Program 4.

Path3	**Covered**	**Values**	**Time**
SA	Yes	Sum1=2 Sum2=102	1.4sec
Improved SA	Yes	Sum1=2 Sum2=114	41sec
Korel	Yes	Sum1=2 Sum2=1	0sec
SA&Korel	Yes	Sum1=2 Sum2=3	0sec
GA	Yes	Sum1=2 Sum2=25	2.6sec
Improved GA	Yes	Sum1=2 Sum2=36	2.3sec

Table 20. Results of Path3 for Subject Program 4.

Path4	**Covered**	**Values**	**Time**
SA	Yes	Sum1=3 Sum2=87	1.7sec
Improved SA	Yes	Sum1=3 Sum2=119	119sec
Korel	Yes	Sum1=3 Sum2=1	0sec
SA&Korel	Yes	Sum1=3 Sum2=3	0sec
GA	No	Sum1=2 Sum2=24	2.9sec
Improved GA	Yes	Sum1=3 Sum2=70	2.6sec

Table 21. Results of Path4 for Subject Program 4.

Path5	**Covered**	**Values**	**Time**
SA	No	Sum1=77 Sum2=90	1.9sec
Improved SA	Yes	Sum1=12 Sum2=66	182sec
Korel	No	Sum1=4 Sum2=2	0.8sec
SA&Korel	Yes	Sum1=12 Sum2=54	131sec
GA	No	Sum1=13 Sum2=91	2.8sec
Improved GA	No	Sum1=13 Sum2=114	2.5sec

Table 22. Results of Path5 for Subject Program 4.

Path6	**Covered**	**Values**	**Time**
SA	Yes	Sum1=69 Sum2=2	2.1sec
Improved SA	Yes	Sum1=96 Sum2=2	162sec
Korel	Yes	Sum1=4 Sum2=2	0.05sec
SA&Korel	Yes	Sum1=4 Sum2=2	0sec
GA	Yes	Sum1=127 Sum2=2	3.2sec
Improved GA	Yes	Sum1=97 Sum2=2	3.13sec

Table 23. Results of Path6 for Subject Program 4.

Path7	**Covered**	**Values**	**Time**
SA	No	Sum1=97 Sum2=39	2.4sec
Improved SA	Yes	Sum1=127 Sum2=7	311sec
Korel	Yes	Sum1=0 Sum2=7	0.05sec
SA&Korel	Yes	Sum1=0 Sum2=7	0.05sec
GA	No	Sum1=107 Sum2=5	3.5sec
Improved GA	Yes	Sum1=109 Sum2=7	3.3sec

Table 24. Results of Path7 for Subject Program 4.

7.5. Subject Program 5

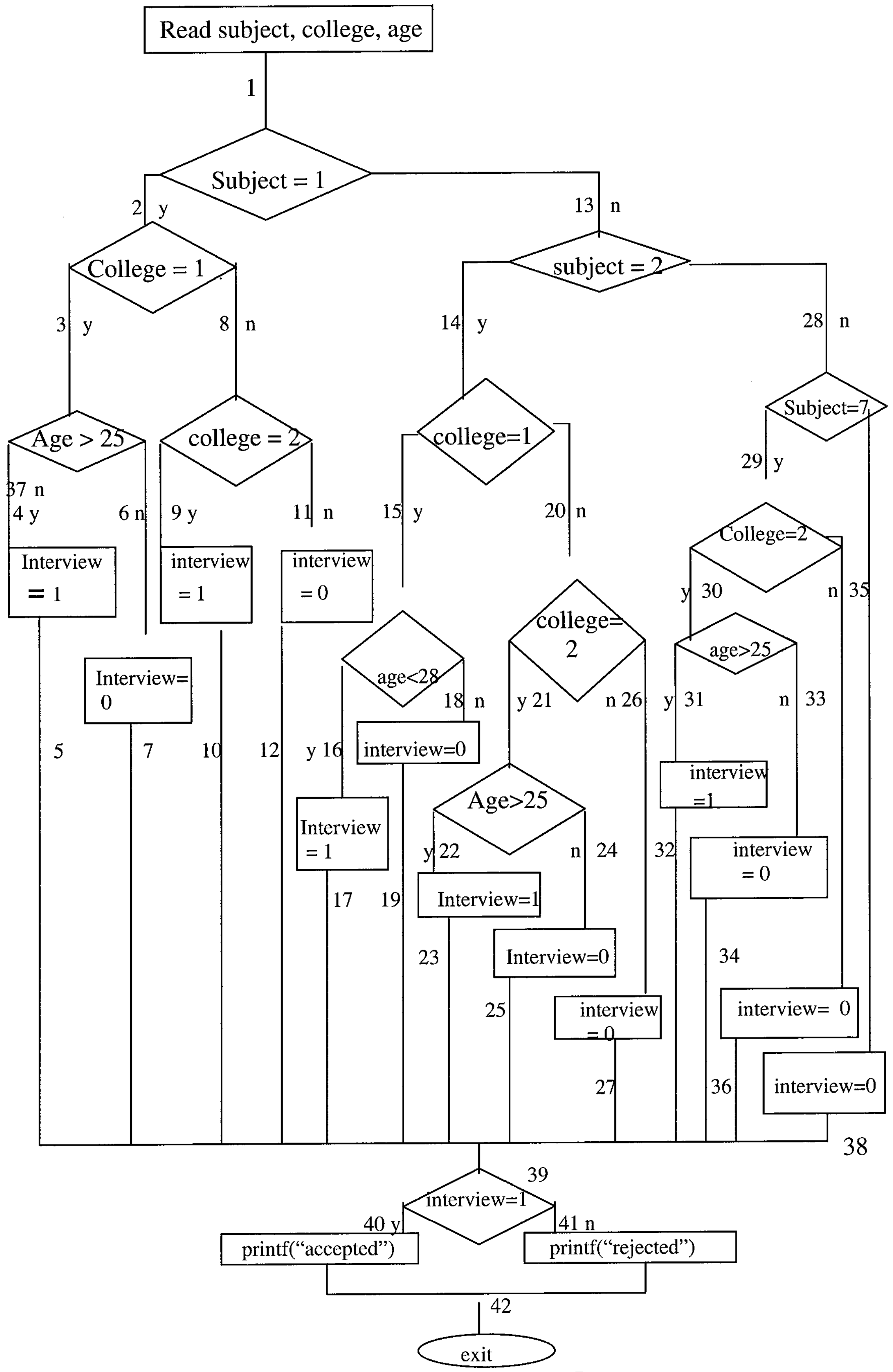

Figure 9 Flow Chart of Subject Program 5.

This subject program subject consists of the following paths:
Path1:1,2,3,4,5,39,40,42
Path2: 1,2,3,6,7,39,41,42
Path3: 1,2,8,9,10,39,40,42
Path4: 1,2,8,11,12,39,41,42
Path5: 1,13,14,15,16,17,39,40,42
Path6: 1,13,14,15,18,19,39,41,42
Path7: 1,13,14,20,21,22,23,39,40,42
Path8: 1,13,14,20,21,24,25,39,41,42
Path9: 1,13,14,20,26,27,39,41,42
Path10: 1,13,28,29,35, 36, 39, 41,42
Path11: 1, 13, 28, 29, 30, 33, 34, 39, 41, 42
Path12: 1,13, 28, 29, 30, 31, 32, 39, 40, 42
Path13: 1, 13, 28, 37, 38, 39, 41, 42

Path1	Covered	Values	Time
Improved SA	Yes	Subject=1 college=1 age125	68sec
Korel	Yes	Subject=1 college=1 age=26	0.27sec
Improved GA	Yes	Subject=1 college=1 age=38	2.25sec

Table 25. Results of Path1 for Subject Program 5.

Path2	Covered	Values	Time
Improved SA	Yes	subject=1 college=1 age16	44sec
Korel	No	Subject=2 college=0 age=2	2.03sec
Improved GA	Yes	Subject=1 college=1 age=15	2.2sec

Table 26. Results of Path2 for Subject Program 5.

Path3	Covered	Values	Time
Improved SA	Yes	subject=1 college=2	34sec
Korel	Yes	Subject=1 college=2 age=2	0sec
Improved GA	Yes	Subject=1 college=2	2.14sec

Table 27. Results of Path3 for Subject Program 5.

Path4	Covered	Values	Time
Improved SA	Yes	subject=1 college=64	67sec
Korel	Yes	Subject=1 college=3 age=2	0.05sec
Improved GA	Yes	Subject=1 college=91	2.25sec

Table 28. Results of Path4 for Subject Program 5.

Path5	Covered	Values	Time
Improved SA	Yes	subject=2 college=1 age=24	110sec
Korel	Yes	Subject=2 college=1 age=1	0.05sec
Improved GA	Yes	Subject=2 college=1 age=23	2.5sec

Table 29. Results of Path5 for Subject Program 5.

Path6	Covered	Values	Time
Improved SA	Yes	subject=2 college=1 age=62	105sec
Korel	Yes	Subject=2 college=1 age=28	0sec
Improved GA	Yes	Subject=2 college=1 age=119	2.5sec

Table 30. Results of Path6 for Subject Program 5.

Path7	Covered	Values	Time
Improved SA	Yes	subject=2 college=2 age=89	156sec
Korel	Yes	Subject=2 college=2 age=26	0sec
Improved GA	Yes	Subject=2 college=2 age=27	2.8sec

Table 31. Results of Path7 for Subject Program 5.

Path8	Covered	Values	Time
Improved SA	Yes	subject=2 college=2 age=24	151sec
Korel	Yes	Subject=2 college=2 age=0	0sec
Improved GA	Yes	Subject=2 college=2 age=8	2.8sec

Table 32. Results of Path8 for Subject Program 5.

Path9	Covered	Values	Time
Improved SA	Yes	subject=2 college=64	103sec
Korel	Yes	Subject=2 college=3 age=3	0.05sec
Improved GA	Yes	Subject=2 college=109	2.5sec

Table 33. Results of Path9 for Subject Program 5.

Path10	Covered	Values	Time
Improved SA	Yes	subject=7 college=14	106sec
Korel	Yes	Subject=7 college=1 age=2	0.2sec
Improved GA	Yes	Subject=7 college=24	2.5sec

Table 34. Results of Path10 for Subject Program 5.

Path11	Covered	Values	Time
Improved SA	Yes	subject=7 college=2 age=3	164sec
Korel	Yes	Subject=7 college=2 age=1	0.3sec
Improved GA	Yes	Subject=7 college=2 age=4	2.4sec

Table 35. Results of Path11 for Subject Program 5.

Path12	Covered	Values	Time
Improved SA	Yes	subject=7 college=2 age=62	157sec
Korel	No	Subject=3 college=0 age=2	2.14sec
Improved GA	Yes	Subject=7 college=2 age=84	2.8sec

Table 36. Results of Path12 for Subject Program 5.

Path13	Covered	Values	Time
Improved SA	Yes	subject=64	66sec
Korel	Yes	Subject=0 college=2 age=2	0.3sec
Improved GA	Yes	Subject=102	2.3sec

Table 37. Results of Path13 for Subject Program 5.

Chapter 8

Empirical Results for Real-Value Input Data

In this chapter we present the experimental results of the test cases applied to the new SA and new GA using real-value input data.

8.1. Subject Program 6

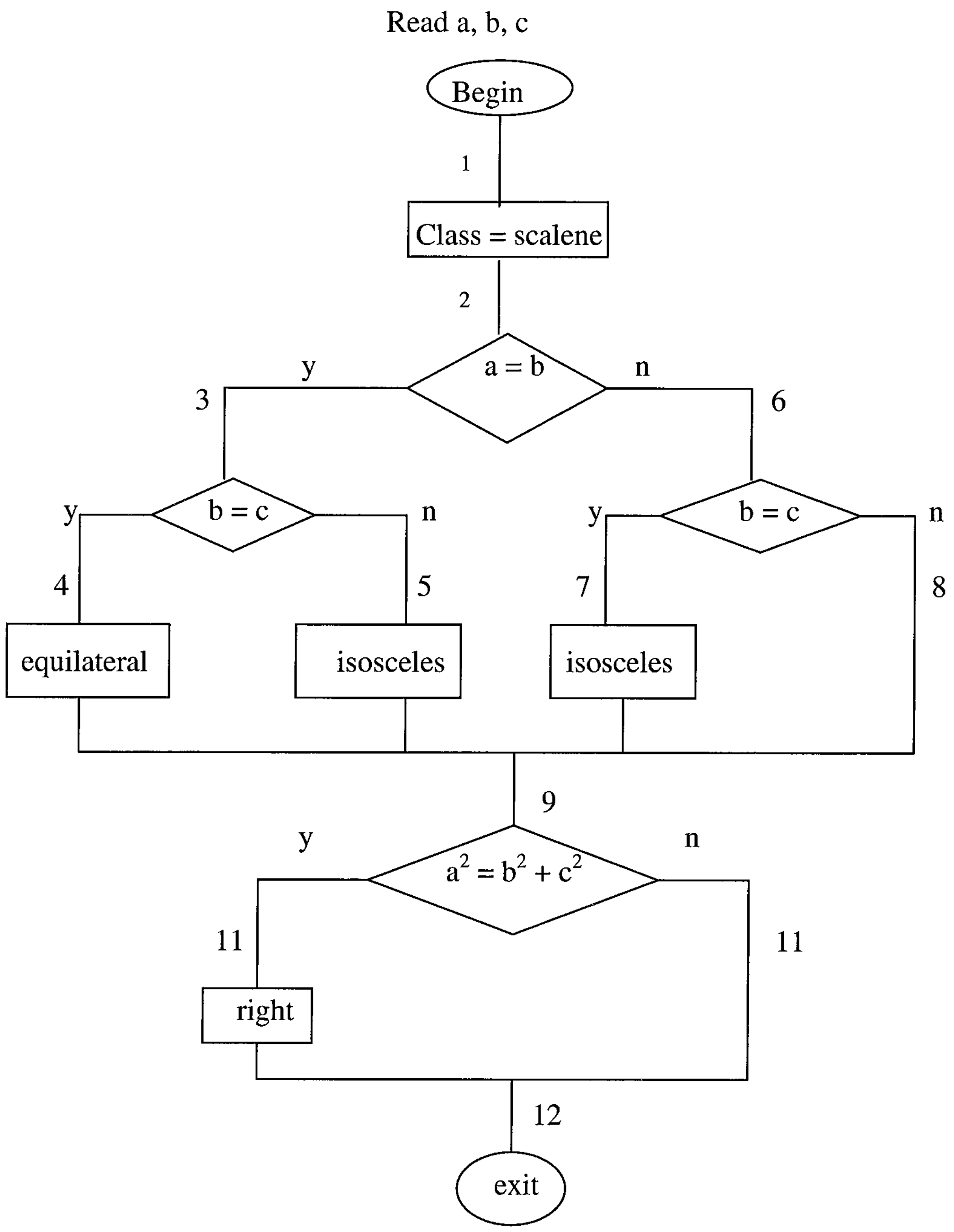

Figure 10. Flow Chart of Subject Program 6.

This subject program consists of the following paths:

Path1: 1, 2, 3, 5, 9, 11, 12 (isosceles with a = b)
Path2: 1, 2, 6, 7, 9, 11, 12 (isosceles with b = c)
Path3: 1, 2, 3, 4, 9, 11, 12 (equilateral)
Path4: 1, 2, 6, 8, 9, 10, 12 (right)
Path5: 1, 2, 6, 7, 9, 10, 12 (isosceles + right)
Path6: 1, 2, 6, 8, 9, 11, 12 (scalene)

Path1	**Covered**	**Values**	**Time**
SA	Yes	a=b=25.49 c=2.66	275sec
GA	Yes	a=b=8.74 c = 5.38	3.24sec

Table 38. Results of Path1 for Subject Program 6.

Path2	**Covered**	**Values**	**Time**
SA	Yes	a=7.47 b=c=8.98	275sec
GA	Yes	a=4.103, b=c=5.68	3.2sec

Table 39. Results of Path2 for Subject Program 6.

Path3	**Covered**	**Values**	**Time**
SA	Yes	a=b= c=90.107	113.84s
GA	Yes	a=b=c=123.109001	2.96sec

Table 40. Results of Path3 for Subject Program 6.

Path4	**Covered**	**Values**	**Time**
SA	Yes	a=126.199997 b=121.220001 c=35.102001	165sec
GA	No	a=7.92 b=1.47 c=4.101	3sec

Table 41. Results of Path4 for Subject Program 6.

Path5	**Covered**	**Values**	**Time**
SA	No	a=35.24 b=24.47 c=25.47	261sec
GA	No	a=121.919998, b=83.113998, c=89.112	3.4sec

Table 42. Results of Path5 for Subject Program 6.

Path6	**Covered**	**Values**	**Time**
SA	Yes	a = 48.48 b=49.123 c=2.110	298sec
GA	Yes	a=1.46 b=2.99 c=1.105	3.02sec

Table 43. Results of Path6 for Subject Program 6.

8.2. Subject Program 7

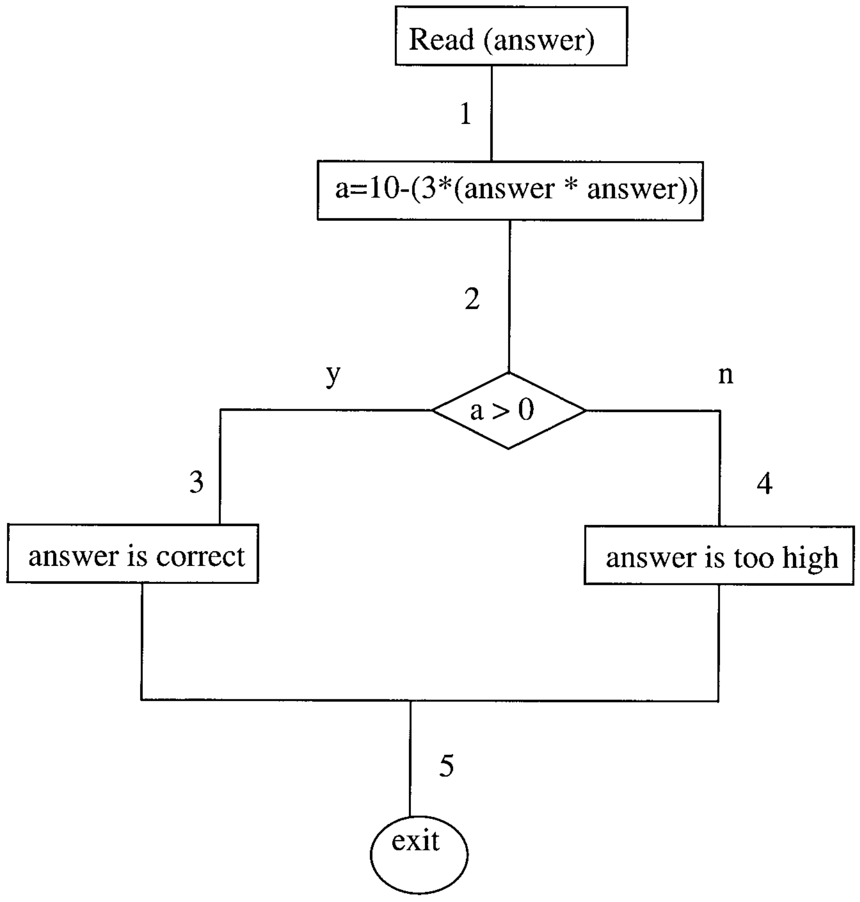

Figure 11 Flow Chart of Subject Program 7.

This subject program contains the following paths:

Path1 : 1, 2, 3, 5
Path2 : 1, 2, 4, 5

	Paths	**Solved**	**Results**	**Fitness**	**Time**
Improved SA	Path1	Yes	Answer= 0.116	0	36 sec
Improved SA	Path2	Yes	Answer = 5.15	0	19 sec
Improved GA	Path1	Yes	Answer= 0.108	0	2 sec
Improved GA	Path2	Yes	Answer = 3.32	0	2 sec

Table 44. Results of Path1, Path2 for Subject Program 7.

8.3. Subject Program 8

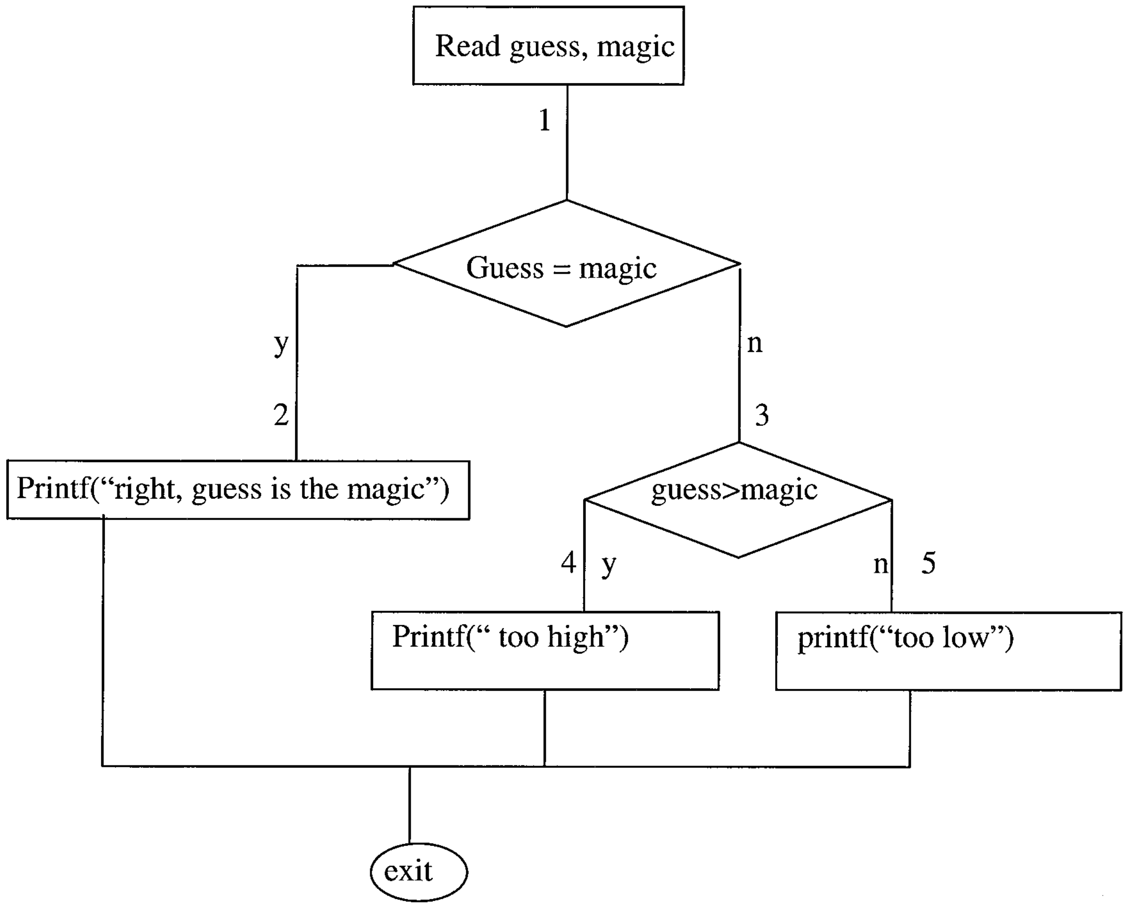

Figure 12 Flow Chart of Subject Program 8.

This subject program contains the following paths:

Path1: 1, 3, 5, 6
Path2: 1, 3, 4, 6
Path3: 1, 2, 6

	Paths	Solved	Results	Fitness	Time
Improved SA	Path1	Yes	Guess=3.5 value= 65.5	0	94 sec
Improved SA	Path2	Yes	Guess=48.75 Value=27.75	0	66 sec
Improved SA	Path3	Yes	Guess=1.102 Value=1.102	0	72 sec
Improved GA	Path1	Yes	Guess=8.25 Value=97.25	0	2.47
Improved GA	Path2	Yes	Guess=103 Value=93	0	2.36
Improved GA	Path3	Yes	Guess=97.75 Value=97.75	0	2.26sec

Table 45. Results of Path1, Path2, Path3 for Subject Program 8.

8.4. Subject Program 9

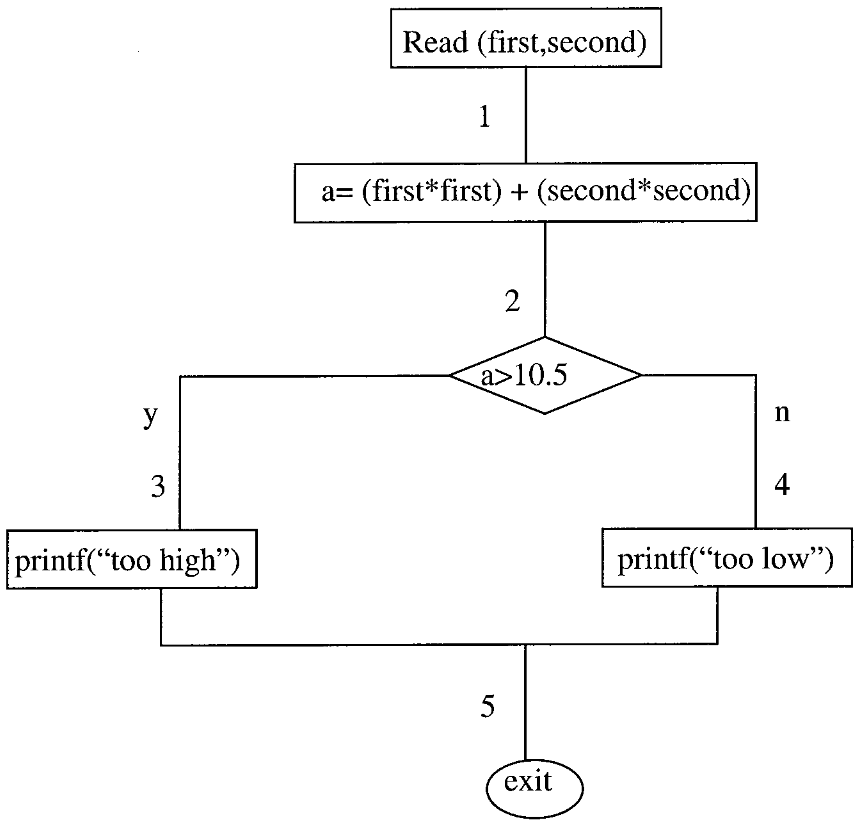

Fig. 13 Flow Chart of subject program 9

This subject program contains the following paths:

Path1 : 1, 2, 3, 5
Path2 : 1, 2, 4, 5

	Paths	**Solved**	**Results**	**Fitness**	**Time**
Improved SA	Path1	Yes	First = 1.24 Second = 7.12	19	124sec
Improved SA	Path2	Yes	First = 2.5 Second = 1.5	34	126 sec
Improved GA	Path1	Yes	First = 5.82 Second = 9.24	74	2.63
Improved GA	Path2	Yes	First = 0.8 Second = 3.1	40	2.63

Table 46. Results of Path1, Path2 for Subject Program 9.

Chapter 9

Discussion of Results

After doing a thorough testing for the improved SA, improved GA, and Korel's Algorithm, we came up with important observations enabling us to compare these algorithms efficiency and efficacy. After running these algorithms on 36 paths in 5 subject programs of integer-value input data, SA failed on 1 path, GA on 3 paths and Korel on 10 paths. On real-value input data, SA and GA were run on 13 paths in 4 subject programs. SA failed to cover only one path; whereas GA failed on 2 paths. Results are found in Chapters 7 and 8. It's important in this context to mention the observations deduced from the empirical results.

First we noticed that new SA implementation proved to give better results than the old SA. This is due to the fact that the old implementation when run on the test cases at hand, failed to solve many paths except for one which is the "isoright" path in the triangle test case (test case T2). It's important to mention in this context that the new SA was able to give either the right angle or the isosceles triangle as its solution, but it failed to find them both simultaneously. This implies that it was able to find a partial solution for this particular path.

The second observation resides in the fact that the new version of GA generated better results than the old one in terms of correctly solving more paths. For instance, new GA solved the right angle path that is considered one of the most complicated paths; whereas, old SA failed, in addition to many other paths the results of which are shown in Chapter 8.

The third observation is that, when new SA, new GA, and Korel were compared, we noticed that the first two algorithms gave better results than Korel. This is based on the fact that Korel failed to traverse many paths like the right angle (path 4, test case T1), and paths 2 and 3 in test case T2, in addition to many other paths the results of which are found in Chapter 8. Meanwhile, SA and GA were able to traverse these paths. Although Korel's algorithm failed to find many paths, it is still considered the fastest algorithm among all other algorithms, when it succeds in finding the soltuion. This is due to the fact that Korel, in case it succeeds in finding the solution, it'll take a small number of iterations and a negligible execution time. However, SA and GA require more trials to find the solution, but execution time is shorter than that of Korel when it fails. This is due to the fact that when Korel fails to find the solution, it will require a high number of trials requiring a long execution time. In other words, Korel is faster than GA and SA in case of success, but much slower in case of failure. However, these two are more efficient in finding the right solutions. According to the empirical results, the success rate of SA, in the integer-value input data, is 92.3%, thao of GA is 84.6% and that of Korel is 72.2%.

Since Korel, when it succed, is considered the fastest algorithm, we propose combining SA and Korel. The algorithm obtained after the combination allows Korel to run first, if it finds a solution, the execution terminates. Otherwise, SA starts running using the values that Korel ended up with, rather than starting from random variables. The aim of this proposition is saving execution time when Korel succeeds in solving a particular test case; meanwhile, resorting to SA to handle complex subject programs. Accordingly, the execution time will be less than that of SA by itself.

When comparing SA and GA, we observed that SA found more solutions than GA. For instance GA failed on paths on path 5 of test case T3 and path 5 of test case T4, whereas, SA succeeded in solving them, in addition to all the other paths of the all the test cases except for one path that is the isoright mentioned previously. Moreover, SA needs less memory space due to its simplified underlying data structure; whereas, GA needs more memory space to hold the population of individuals. Accordingly, when the number of variables increases, GA requires more space to hold the population. The fact that might increase its execution time and decrease the difference in execution time between SA and GA. According to the empirical results of the real-value input data, the success rate of SA is 92.3% and that of GA is 84.6%.

Finally, it's important to mention the difficulties and limitations of SA, GA and Korel. When preparing the input files for the test cases that are to be run on SA/GA, the user needs to specify, beforehand, the auxiliary variables that are needed to transform inequalities into equalities; meanwhile, Korel doesn't require such a specification. Furthermore, for each path within the same test case, a separate input file should be written in order for SA/GA to run using such an input file; Korel, on the other hand, requires a single general input file for the whole paths within the same test case. SA/GA do not recognize whether the specified path has been traversed successfully or not; whereas Korel does. Korel doesn't support compound conditions within the same predicate (like or, and); whereas, SA/GA support such predicates.

Korel can deal with integer values only, without supporting real numbers; whereas, SA/GA support both real and integer numbers.

Chapter 10

Conclusion

In this thesis we improved the Simulated Annealing Algorithm and the Genetic Algorithms, to obtain better results than the previous versions. Moreover, we implemented Korel's algorithm as a hill climbing algorithm and compared it's performance and path-coverage capability with that of GA and SA. We also expanded the capabilities of SA and GA enabling them to solve integers as well as real-value input data. Meanwhile, Korel's algorithm is limited to integer values only.

The empirical results showed that SA and GA were able to cover more paths than Korel, and that SA tends to perform better than GA. However, GA is faster than SA, and Korel, when it works, is the fastest. For this reason, we propose combining SA and Korel aiming at decreasing the execution time by exploiting Korel's ability to solve simple subject programs rapidly and that of SA to handle the complex ones.

References

Beizer, B. (1990) 'Software Testing Techniques', New York, York. Van Nostranf Reinhold, 145- 172.

Duran, J. and Ntafos, S. (1984) 'An Evaluation of Random Testing', Transaction on Software Engineering, 10 (4), 438-443.

Hamlet, G. and Taylor, R. (1990) 'Partition Testing does not Inspire Confidence', *IEEE Transactions on Software Engineering*, 1402 – 1411.

Hetzel, B. (1991) 'Software Testing : Some Troubling Issues and Opportunities', Presented to the British Computer Society Special Interest Group in Software Testing.

Holland, H. (1975) 'Adaptation in Natural and Artificial Systems', University of Michigan press, Ann arbor.

Howden, W. (1986) 'A functional Approach to Program Testing and Analysis', Transaction on Software Engineering, 12 (10), 997-1005.

Huang, J. (1975) 'An Approach to Program Testing', University of Houston..

Jeng, B. (1994) 'Integrating Data Flow and Domain Testing', Asia-pacific Software Engineering Conference, Tokyo, Japan, 123- 135.

Jones, B., Sthamer, H and Eyres, D. (1996) 'Automatic Structural Testing Using Genetic Algorithms', *Software Engineering journal*, 299-306.

Joumaa, R. (1997) 'Simulated Annealing and Improved Genetic Algorithms for Path Testing', Thesis, Lebanese American University, Beirut, Lebanon, 31-41.

Korel, B. (1990) 'Automated Software Test Generation', *IEEE Transactions on Software Engineering*, 870-879.

Korel, B. (1992) 'Dynamic Method for Software Test Data and Generation', *Journal of Software Testing, Verification and Reliability,* 203-213.

Korel, B. and Al-Yami, A. (1996) 'Assertion-oriented Automated Test Data Generation', *International Conference on Software Engineering,* 71-80.

MaCabe, J. (1982) 'Structured Testing', US Government Prating Office, Washington, D.C.

Mansour, N. and Joumaa, R. (1998) 'Simulated Annealing Algorithm for Path Testing', *Conference on Software Engineering,* Las Vagas, Nevada, U.S.A., 138-141.

Marciniak, J., Curts, B. et al.(1994) 'Encyclopedia of Software Engineering', New York, John wiley and Sons inc.,1330- 1358.

Pargas, R. P., Harrold, M. J. and Peck, R. R. (1999) 'Test-Data Generation Using Genetic Algorithms', *Software Testing Generation and Reliability,* 263-282.

Ramamoorthy, C. V., Ho, S. F. and Chen, W. T. (1976) 'On the automated generation of program test data', *IEEE Transactions on Software Engineering,* 2(4), 293-300.

Rapps, S. and Weyuker, E. (1985) 'Selecting Software Test Data Using Data Flow Information', *IEEE Transactions on Software Engineering*, 367-375.

Rutenbar, A. (1989) 'Simulated Annealing Algorithms : An overview', *IEEE Transactions on Software Engineering.*

Stocks, P. A. and Carrington, D.A. (1993) 'Test Template Framework: A Specification-based Testing Case Study', *International Conference on Software Engineering,* 405-414

Tai, K-C. (1993) 'Predicate-based Test Generation for Computer Programs', *International Conference on Software Engineering,* 267-276

Zhu, H., Hall, P.A. and May, J. H. (1997) 'Software Unit Test Coverage and Adequacy', *ACM Computing Surveys,* 29(4), 366-423.