University of Alberta

Empirical studies on test data generation using optimization techniques

by

Man Xiao 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2004

Your file Votre référence
ISBN: 0-612-95881-7
Our file Notre référence
ISBN: 0-612-95881-7

Acknowledgment

I have been supported by many people during my research, which led to this thesis. First of all, I would like to express my deepest sense of gratitude to my supervisors Dr. Marek Reformat and Dr. James Miller for their believing my potential, patient guidance, encouragement and excellent advice throughout this study. Dr. Marek Reformat has provided me unlimited amount of encouragement, expert guidance, stimulating suggestions and proper directions that have enabled constant improvement and refinement of this research. Dr. James Miller's invaluable expertise in Software Engineering along with his advice inspired me throughout this research. I would not have finished my Master's program without their help. I am deeply indebted to them for their generous financial and intellectual support.

I am thankful to Sarah McEvoy and M.Sc. Paul J. Iglinski for their fruitful collaboration in providing the programs under test in this research.

I would like to express my appreciation to Dr. Petr Musilek, Dr. Jozef Szymanski, Dr. James Miller and Dr. Marek Reformat for spending their invaluable time on reading and correcting this thesis, and for being the members of the examining committee.

I am also thankful to Kenneth Raiche for his encouragement and support throughout this research.

Finally, I take this opportunity to express my profound gratitude to my beloved parents, Xiuwen Ye and Zhanhua Xiao, for their unconditional love, infinite encouragement, and support throughout my life. They have provided me the strength and courage to pursue my dreams that finally lead to the completion of my research. Thank you.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Problem of software testing

Software has become a significant part of computer system since 1970s and it plays a more and more important role in our modern society. In a computerized embedded world, the faults in the software can cause huge losses.

In order to reveal the faults in software and ensure software performs as intended, software should be validated in its life cycle. Thus, validation of software is receiving increasing notice. There are many methods to assess software—for example code reviews, code inspection, formal specification. However, software testing is the most common, widely accepted and practiced method of validating software. Software testing is a process of exercising software in a controlled and systematic way in its intended environment. Test is one of the most important techniques used in industry to assess a software product and reduce the risk of failure [Gar99]. Testing can help find the error earlier in the lift cycle of the software development, hence reduce the cost of fixing error in the later stage and reduce the cost of whole development process.

Unfortunately, software testing is a tricky job and it is a very expensive process—the cost of testing exceeds the cost of design and coding, typically consuming at least 50% of the total costs of developing software [Bei90]. The generation of test data is one of the most difficult and important problems in the testing process. Test data generation is the process of identifying a set of test data that satisfies a selected testing criterion. It plays a critical role in software testing process. However, it is a labour-intensive and costly component in software testing. The effort involved in selecting test data in industry typically represents at least 40% of the total testing costs [Bei90]. While automation of the testing process—the maintenance and

execution of test case—is becoming popular, most of the generation of test data is still a manual activity.

1.2 Automation of test data generation

The automation of test data generation is a desirable way to drastically reduce the time, effort, labour and cost in software testing. Moreover, the automation of test data generation can increase the quality of software testing and help testers gain more confidence of the whole testing process. Hence, automation of test data generation is becoming a promising issue in software testing and attracting many researchers' interest. A number of approaches on automated test data generation have been presented in the literature. In this thesis, we focus on dynamic structural test data generation.

Typically, the approaches on dynamic test data generation are based on a paradigm. In this paradigm, the test data generation problem is reduced to a function minimization problem, which can use heuristic optimization techniques to solve. A number of optimization techniques have been applied to the dynamic test data generation: Standard Genetic algorithm, Differential Genetic algorithm, Hill-climbing, Simulated annealing.

1.3 Thesis contributions

The work presented in this thesis aims at investigating the performance of different optimization techniques in test data generation. A series of experiments are conducted. The ultimate goal is to identify the suitability of different optimization techniques to the generation of test cases. Four optimization techniques Genetic Algorithm, Simulated Annealing, Genetic Simulated Annealing and Simulated Annealing with Advanced Adaptive Neighborhood are implemented and integrated with test data generation system. The results of conducted experiments are thoroughly analyzed and compared. To our knowledge, there is no report that Genetic Simulated Annealing and Simulated Annealing with Advanced Adaptive Neighborhood have been used in test-data generation. Condition-decision coverage is used as the test adequacy criterion in our experiments. It is a more complicated and reliable test

adequacy criterion than test adequacy criterion used in most previous approaches. For example, statement coverage is used in [Kor96] and [PHP99], and branch coverage is used in [CCCL96] and [PHP99]. The experiments are performed on five C/C++ programs. Empirical results are provided in this thesis, as well as the detailed analysis of the performance of each optimization methods on each program. The results show that generally, different optimization techniques have different suitability and limitations. The results of the experiments have allowed for identification of optimization techniques that are the most promising, as well as the ones that should be avoided in the case of building systems for automatic generation of test cases.

The future research directions addressing the limitations of these optimization methods are provided. This allows us to work in the future research area, which can help the test data generation to be more successful and efficient.

1.4 Thesis outline

The remainder of this thesis is structured as follows.

Chapter 2 provides the introduction of software testing. Several key concepts are discussed in this chapter, which include Functional testing, Structural testing and, test adequacy criterion. This chapter also introduces several common test adequacy criteria and provides an overview of test data generation problem.

Chapter 3 provides a survey of previous approaches in automated test data generation.

Chapter 4 provides an overall description of four optimization algorithms used in this thesis, which are Genetic Algorithm, Simulated Annealing, Genetic Simulated Annealing and Simulated Annealing with Advanced Adaptive Neighborhood.

Chapter 5 presents the approach used in this thesis and introduces the overall methodology presented in this thesis. These include the introduction of test adequacy criterion, the strategy, the optimization algorithms, the programs under test and the experiment procedure used in this work.

Chapter 6 presented the experimental result on five different programs. This chapter gives a detailed analysis of each target program and the comparison of performance of five test generation systems on each program.

Chapter 7 concludes the thesis, discusses the result presented in this thesis and provides an outlook on future research directions.

Chapter 2 Software testing

In order to assess a software product, reduce the risk of failure and establish the confidence of the software performing as intended, software should be validated in its life cycle. Testing is the most common, widely accepted and practiced method of validating software.

This chapter introduces the background of software testing by clarifying several key concepts in software testing.

2.1 Functional testing and Structural testing

Basically, depending on the source of information used in the test plan, the approaches in the software testing can be divided into two categories: functional testing and structural testing.

Functional testing is also called black–box testing. Functional testing is specification based. To conduct a functional testing, software testers derive the test cases from the given specification of the target program; these test cases are used to test if the target software product can meet the expected functional requirements. So functional testing focuses on the target program's expected functional requirements. No implementation information of the source code but the specification of the program is needed for a functional testing, so the target program should be seen as a "black box". As Pressman summarized in [Pre00, pp448], functional testing focus on finding five different categories of errors: incorrect or missing functions, interface errors, data structure error or external data access errors, behavior or performance errors and initialization and termination errors.

Structural testing is also called white–box testing or glass-box testing. As the name implies, structural testing addresses the examination of the control structure of

the target program, i.e. testers need to treat the target program as a "white box". Structural testing is program based. Hence, the complete knowledge of construction of the target program is needed to apply a structural testing. Using structural testing, testers need to generate test cases to exercise the internal operations, such as loop, conditional decisions, internal data structure, etc. The objective of structural testing is to ensure that the internal operations of the target program perform as expected. Obviously, this is what functional testing cannot reach.

As discussed above, functional testing and structural testing examine different aspects of a software product, so the combination of these two testing methods is necessary to ensure a software product has the desired features and quality. Moreover, it is impossible to obtain a reliable software product without an appropriate structural testing since *structural testing is the cornerstone of all testing* [Bei96]. Hence, structural testing is a very important issue in software testing.

The works presented in this thesis focus on structural testing.

2.2 Test Adequacy Criterion

In order to measure the quality of the source code, the first step is to choose an appropriate test adequacy criterion. Test adequacy criterion is an important issue in software testing. Previous researches [Hor94][Chi94][DM94] show that good test adequacy criterion is essential at uncovering faults, which helps testers improve the quality of software testing. Test adequacy criterion is used to define whether a test is an adequate test or not, and thus determine if a software product under the test is acceptable. As mentioned before, this thesis focuses on structural testing; hence the following sections provide a description of several common test adequacy criteria used in previous structural testing approaches. Most of them are based on the code coverage of the target program. The basic coverage criteria are briefly described below.

2.2.1 Statement coverage

Statement coverage is the simplest coverage criterion. As the name suggests, it requires that every statement in the program must be executed at least once. However when it comes to the compound conditions and control flow structure, a test set that

fulfills 100% statement coverage may still not be enough since each statement only requires to be executed once. Faults in the source code that may cause potential failures may still not be found due to the limitation of this coverage criterion. This is a very simple criterion.

2.2.2 Branch coverage

Branch coverage is also called decision coverage, which requires the test to take the true and false outcomes of every decision in the program, even if there is no code associated with these outcomes. For example, if we need to carry out the test to satisfy the branch coverage of following code:

```
if tri=1 && i+j>k then
t=2
else
t=1
end
```

We should generate a set of test data which cause both the *true* and *false* outcomes of the condition **tri=1 && i+j>k**, that means the test suite should execute both **t=2** and **t=1**.

Branch coverage is based on the control flow structure, so the test set that fulfills the branch coverage increases the confidence of the control flow structure of the program compared to the test set that only fulfills the statement coverage. However when it comes to compound conditions, the weakness still exists. For example, consider the following code:

```
if x>1 or( x<0 && y>0 && z>y) then
...
end
```

In order to take the *true* branch of the decision, the only thing we need to do is to make x larger than 1; the fault in the rest of the condition can easily be overlooked.

2.2.3 Condition coverage

As the name suggests, condition coverage reports the *true* or *false* outcome of each Boolean sub-expression, separated by logical-and and logical-or if they occur. Condition coverage measures each condition in a decision independently. Consider the following decision:

if tri=1 && i+j>k then

t=2

else

t=1

end

the test cases that fulfill the condition coverage should satisfy the following conditions:

1. Cause **tri=1** to take on *true* and *false* at least one time

2. Cause **i+j>k** to takes on *true* and *false* at least one time.

2.2.4 Condition-decision coverage

Condition-decision coverage combines condition coverage and decision coverage. To obtain the condition-decision coverage of a target program, the tester should generate the test cases such that all conditions in all decision in the target program take on both *true* and *false* outcomes at least once, and exercise the *true* and *false* outcomes of every decision. For example, consider following situation:

if tri=1 && i+j>k then

t=2

else

t=1

end

the test cases should satisfy following conditions:

1. Cause **tri=1**to take on *true* and *false* at least one time

2. Cause **i+j>k** to takes on *true* and *false* at least one time

3. Cause **if tri=1 && i+j>k** to take on *true* and *false* at least one time and execute the corresponding code at least one time.

Note that the test cases such that each condition in the target program take on both *true* and *false* outcomes at least once, do not always cause each decision to take on both *true* and *false* outcomes at least once. For example, let a and b be two independent input, consider the following code fragment:

if a<0 and b>1

The possible test cases such that each condition takes on both *true* and *false* outcomes at least once are as follows:

a=-1, b=0;

a=0, b=0;

a=2, b=3;

a=3, b=-1.

All of four sets of test cases above cannot exercise the *true* branch of decision

if a<0 and b>1.

Obviously, condition-decision coverage is more complicate and reliable than the statement coverage and branch coverage. Generation of the test cases fulfilling this coverage can help developers gain more confidence in the quality of the source code. Hence, the test criterion used in this thesis is condition-decision coverage.

2.2.5 Multiple condition coverage

Multiple condition coverage is a more expensive test criterion than condition coverage. To achieve full multiple condition coverage, testers need to generate test data to exercise every possible combination of *true* and *false* outcomes of conditions in a decision. For example, consider the following code:

if tri=1 && i+j>k then

t=2

else

t=1

end

multiple condition coverage requires 4 test cases, which are described as the following:

tri=1 && i+j>k,

tri≠1 && i+j>k,

tri=1 && i+j≤k,

tri≠1 && i+j≤k.

2.3 Generation of test-data

Once the test adequacy criterion has been chosen, the next step is to generate an adequate test for the target program to satisfy the test adequacy criterion chosen, i.e., to find the appropriate test input to satisfy the given test adequacy criterion. This process is called test-data generation. With appropriate test cases generated, the test set will exercise the specific features of the target program that are required by the test adequacy criterion. Therefore test data generation is a significant issue in software testing, and there are numerous approaches addressing this issue. Traditionally, test data have been generated by testers manually, but sometimes it can be very difficult to find the test cases which satisfy the test criterion, so it is one of the most labour-intensive parts in software testing.

The cost of testing exceeds the cost of design and coding, typically consuming at least 50% of the total costs of developing software, and the effort involved in selecting test data in industry typically represents at least 40% of the total testing costs [Bei90].

Chapter 3 Automated test data generation

As discussed in the previous chapters, software testing is a high-cost process in the software development, and the most expensive problem of software testing is the generation of test data. Automation is an important step to reduce the cost in software testing. Compared to the traditional manual generation of software test data, the automation of test data generation will reduce the time and labor consuming in the software testing. This will lead to reduction of the cost of the whole software development. Moreover, the automation of test data generation may increase the quality of software testing and help testers gain more confidence of the whole testing process. Automated test data generation can help software developers produce a highly reliable software product at reasonable cost. All this means that automated test data generation is a promising issue in software testing attracting many researchers' interest. This chapter provides a survey of previous approaches in automated test data generation.

There are a number of approaches for test-data generation presented in the literature. Using the testing classification presented in the previous section, the automated test data generation can be divided into two categories, which are automated structural test data generation and automated functional test data generation. There are two categories of automated structural test data generation: static and dynamic.

3.1 Static test data generation

Instead of the actual value, the symbolic expression is used as input data in the static test data generation approach. In static test data generation, the target program is executed only symbolically. The static approach originated in 1970s, Clarke's work [Cla76] on symbolic execution is one of the earliest approaches. To generate a set of

constraints on test data, he designed a method to symbolically execute a given path of ANSI Fortran program. Then he used different programming techniques to obtain the test data, which executed the specified path under test. To improve the processing of arrays, Ramamoorthy et al presented a approach called CASEGEN in [RHC76]. This approach is built on Clarke's work but it does not symbolically execute an array, which is dependent on the input data until the constraint satisfaction stage. Coen-Porisini et al [CPD93] attempted to solve the memory problem in CASGEN by using an incremental approach. This approach binds each variable to a set of symbolic values and to constraints of these symbolic values, which are incrementally updated. This approach reduced the growth of computational space and time of algorithm, but the testing of higher level software units may still need expensive symbolic execution.

Static approaches are promising but all of them encounter common problems in the loop analysis, array analysis and pointer analysis. Most of these problems come from the limitation of the symbolic execution, and thus hinder the general acceptance of the static method.

3.2 Dynamic test data generation

Different from the static approach, the dynamic approach in test data generation actually executes the target program and attempts to solve the problems of the static approach. This requires a test data generation system to generate the actual values of input variables to actually execute the target program, and to collect the run-time information. By collecting the run-time information during the execution, the test-data generator evaluates the current test result and finds out how close it is to the desired result. In the subsequent executions, the generator modifies the test data gradually until it satisfies the given test objectives.

3.2.1 Earlier Dynamic test-data generation

Miller and Spooner [MS76] used the dynamic methods to automatically generate test data in the 1970s. In this approach, parts of the program to be tested are seen as numeric functions, which can be evaluated by executing the program, and whose value is minimal for those inputs that satisfy the adequacy criterion. Therefore, the

basic idea of this approach can be characterized as reducing the test data generation problem to a numeric function minimization problem, which measures the quality of a set of test data and represents the test requirements. Thus, it enables the application of heuristic optimization techniques in test data generation approaches. Their approach focuses on testing a particular path. In this approach, a path is selected in the target program to be tested, all the conditions in this path are represented as a set of constraints. A numerical function is generated so that the value of the function is positive when all of the constraints are satisfied (i.e. this selected path is taken). Now, the goal of the generation system is to find the test data so that this function has a positive value.

The result of this approach encouraged other researchers to work on dynamic test data generation. Building on Miller and Spooner' s work, Korel presented a dynamic approach of test data generation in [Kor90] [Kor96] [FK96]. In these approaches, the execution of the test unit is under the control of a monitor; a function is devised in a way that it will only be assigned a negative value when the desired branch is taken. A simple function minimization technique is used to find the input data. Additionally, dynamic data-flow information is used to improve the function minimization technique. At the beginning of the search, a goal (a function) is established according to the test requirements. If the code which is suppose to be tested cannot be reached, a subgoal should be established in such a way that the desired code will be reached. So the subsequent search works for attempting to satisfy this subgoal, and then another new subgoal can be set up for this subgoal. More subgoals may be created in the same way and this can be a recursive process. This approach is termed as chaining in [Kor96] [FK96]. The approaches in [Kor90] use a gradient descent algorithm for the function minimization, which modifies the values of variables slightly so that the function value always improves. Although the experiments in [Kor96] used an enhanced gradient descent algorithm, the search technique is still a local search technique, thus the search process can still be trapped in the local minimum[MMS01]. Korel suggested the use of global optimization techniques to solve this problem.

Gallagher et al introduced another software test data generator ADTEST in [GN97]. This approach specifies an entire path in advance, and then the goal is to

find an input that executes the desired path. Since it is known which branch must be taken for each condition on the path, all of these conditions can be combined in a single function whose minimization leads to an adequate test input. The ADTEST system begins by trying to satisfy the first condition on the path, adding the second condition only after the first condition has been satisfied. As more conditions are reached, they are incorporated in the function that the algorithm seeks to minimize.

3.2.2 Tracey's work

Tracey developed a general automatic test-data generation framework for the safety-critical software to improve the quality of safety-critical software testing, which is introduced in [Tra00]. As earlier approaches, this approach also sees the test-data generation as a constraint-solving problem, and uses a search-based approach to develop a flexible framework for test-data generation. Five search techniques are implemented into the framework: Random search, Hill-climbing search, Simulated annealing search, Genetic algorithm search, and Genetic algorithm with hill-climbing search.

There are some important distinctions that set this approach apart from earlier approaches. Firstly, this framework is targeted at generating negative test-data that means it can illustrate a failure of the system. Secondly, this framework is flexible so it can implement different search techniques and can be targeted at different testing criteria.

Tracey's work shows that, tuning parameter of each search-based technique has limited effect on performance. However, the setting of parameter will have a greater effect, when the search space of safety-critical software is simple and small, or when the problem gets larger and more complex.

The result of Tracey's work is encouraging. However, Tracey also points out that it may be because his experiments were limited to safety-critical systems; the software structure is not complex, so the system is easy to be decomposed into a number of subsystems and each subsystem can be decomposed into a number of software units. Furthermore, the data structure of the safety-critical system is only limited to the numeric or enumeration types; this may also help this approach achieve its success.

3.2.3 Michael's work

Michael et al [MMS01] presented a GADGET (the Genetic Algorithm Data Generation Tool), which used dynamic test data generation. Their research is built on previous work and attempts to apply the automatic test-data generation schemes to complex and large programs. Three optimization techniques standard GA, differential GA, gradient descent algorithms, as well as the random test data generation are implemented in GADGET. This approach aims to examine the relation between the program complexity and the difficulty of test data generation.

In GADGET, the target program was instrumented with additional code, which was used to report the objective function information to an execution controller in GADGET. Before starting the search process, a seed input is used to execute the program, and after the first execution, a coverage table is initialized for the purpose of tracking if a condition/decision is satisfied or not. After that, a series of test requirements are subjected to the search in turn according to the coverage table Whenever a test input satisfies a new test requirement, no matter whether it is the one the test data generator currently working on or not, the new test input is recorded for the future use and the coverage table is updated. Michael's approach is different from the previous approaches in the way that it doesn't concentrate on one specific path to the desired location. Instead, it only works on the condition that has been reached but has not been covered yet. Since the goal is to obtain the complete coverage of the target program, the search processes for every test requirements are not independent. This is because when the test data generator works on a certain requirement, many other requirements are often coincidentally satisfied.

The test adequacy criterion in GADGET is condition-decision coverage. GADGET was applied to programs with various sizes and complexity, including a real-world autopilot control program called b737 that has 2,046 source lines of code, which Michael et al believed was the largest program reported in test data generation literature. The target programs are written in C and C++, so compared to other previous work which only focus on simple programs using simplified programming languages, their experiments can be applied on more complex and more difficult problems. Different from the previous approaches, GADGET greatly simplifies the

dynamic test data generation by skipping complicated control flow analysis for a specific path. However, in GADGET, only those conditions that have been reached are subjected to the optimization procedure, while other conditions/decisions are given up.

The results in [MMS01] show that the Random test data generator was successful in simple programs, but it didn't perform so well when the complexity and the size of the target program increases. The standard genetic search algorithm performed best overall, while the differential genetic algorithm performed better in some programs.

3.3 Other test data generation methods

The dynamic domain reduction procedure (DDR) [OJP97] attempts to combine both the static methods and dynamic methods in the structural test-data generation. This approach is built on both the constraint-based [DO91] goal-oriented [Kor90] methods, and uses dynamic analysis and domain-based symbolic execution. The result presented is pretty encouraging (the good results of DDR have between 93% and 100% all-uses coverage), although it still has limitations in the array and loop analysis.

3.4 Conclusions

A number of automated approaches have been presented in the test data generation literature. Static approaches are based on the idea of using symbolic expressions to represent the test input, and all of these static approaches have some common limitations in arrays and pointer analysis, which hinder the general acceptance of static methods in test-data generation.

Since Miller and Spooner proposed a test data generation in [MS76], all of the approaches on dynamic test-data generation are based on this paradigm. The basic idea of this paradigm is to formulate the test objectives as a series of numerical functions. This means transforming the test data generation to a function maximization (minimization) problem. In previous research, a number of heuristic optimization techniques such as simulated annealing, gradient decent, and genetic

algorithms have been implemented in test-data generation system to perform the function minimization.

There are some successes in some particular problems. However, it is still difficult to apply them to general problems. Most of them only work on simple programs written in simplified programming languages and limit the test data to numeric types. Obviously, for general problems, the program under test can have very complex structure and is not limited to numeric types. Moreover, most of the dynamic test data generation approaches encounter the local minima or the plateaus during their search, so how to guide the search to escape from them is also another problem.

Clearly, all of the approaches discussed above have their own limitations, thus they are limited by lack of generality. Moreover, many techniques still require manual support such as manual path selection or manual refinement of the abstract tests into executable concrete tests, which means the level of automation is not sufficient. Simple search techniques and limitations on the software under test also restrict these dynamic approaches to test data generation from applying to large scale industrial software systems.

Chapter 4 Optimization search techniques

Many heuristic search techniques have been proven to be powerful and flexible for solving complex optimization problems, and they have also been applied to software testing problems. Among these optimization search techniques, there are two techniques that are used most commonly in test data generation: Genetic Algorithms and Simulated Annealing. This chapter provides a brief introduction of these two search techniques and the approaches originated from them.

4.1 Genetic Algorithms

Genetic Algorithms (GAs) are based on an abstract model of the natural genetic evolutionary process. They were developed by Holland *et* al. in the 1970s [Hol75]. Generally, the solutions of the optimization problem are represented as *genotypes* [Hol75] or *chromosomes* [Sch87]. Genetic algorithms start by creating a population of a fixed number of *chromosomes* randomly, and then each *chromosome* in the initial population is evaluated according to a fitness function. Genetic algorithms then select the parents in the population based on their values of the fitness function and produces new offspring through the crossover and mutation, just like evolution in biology. So, the basic procedure of genetic algorithms is as follows:

Step 1 Create initial population randomly (generate solutions of target problem randomly and represent each solution as a *chromosome*)

Step 2 Evaluate every individual in the population according to the fitness function that is problem dependent

Step 3 Select the parents based on the value of fitness function

Step 4 Combine the parents to produce the offspring (crossover)

Step 5 Mutate the offspring

Step 6 Evaluate the offspring according to the fitness function, if the stop criterion is satisfied then stop, otherwise set the offspring as the new population and go to Step 3

Figure 4-1 GA algorithm

Where the common stop criterion used is the global minima that is found or if genetic algorithms stop making progress.

Basically, genetic algorithms include three phases: evaluating, selecting, and applying genetic operators to *chromosomes*. Each phase corresponds to one phase in natural evolution. In genetic algorithms, the evaluation is analogous to the environmental determination of survability in biology, and the selection is analogous to natural selection, and the crossover corresponds to the sexual reproduction in nature. As in nature, if the solution of the given optimization problem is represented as a chromosome in a proper way, the good genes (encoding of genetic information) will be kept during the evolution and the offspring will have better and better fitness function values (survivability) compared to the ancestors.

Traditionally, the parameters of a given optimization problem are represented as bit strings. One method represents them as binary form [Hol75], which is based on an analogy of the chromosomes in nature. The gray code representation, which has an adjacency property, proven to be a better representation in some applications [CS88] [JSE95]. At the same time, the non-binary representations have also been investigated [Ant89][JM91][Mic96][FMJ98]. Thus, the encoding representation is problem dependent.

Evaluation is based on the fitness function that is problem dependent, and it is determined by the test adequacy criteria mentioned in chapter 2. The fitness function

is discussed in detail later. After being evaluated, each individual in the population is assigned a fitness value.

Once each individual in the population has been evaluated, selection can occur. Selection is also called reproduction. It is used to decide which chromosomes will be chosen to be parents and be contributed to successive generations. This is normally based on the evaluated fitness function value of individuals in the current population [Hol75]. Basically, in the selection phrase, there is a bias towards the well-fitted individuals, i.e. the well-fitted individuals have more chances to be selected. There are a number of selection methods [BH91]; the common selection methods include roulette wheel sampling selection, tournament selection, and ranking selection [Bar85].

Roulette wheel sampling selection selects individuals based on their fitness value; each individual's possibility of being chosen as parents is stochastic and proportional to its fitness value [Hol75][Gol89a]. The selection possibility of each individual $P_{sel}(i)$ is calculated as

$$P_{sel}(i) = \frac{f(i)}{\sum_{i=1}^{n} f(i)} \qquad (4.1)$$

Where n is the size of population, i=1...n, and $f(i)$ is the fitness value of each individual. Imagine a roulette wheel, where each individual in the population is corresponds to a slice in the wheel, and the size of the slice is proportional to the individual's $P_{sel}(i)$. Individuals are chosen to generate the new generation by spinning the roulette wheel.

As the name suggests, tournament selection uses n tournaments to choose n individuals. In each tournament, the individual with the best fitness in a group of k elements is chosen; the others are eliminated in the population. The most widely used tournament selection is the binary tournament, i.e. k=2.

Ranking Selection was first introduced by Baker [Bar85] to overcome the strong bias in the Roulette wheel sampling selection. In ranking selection, the population is ordered based on the fitness, and each individual in the population is assigned a rank r

according to its fitness. This method then selects parents in the population based on their ranks rather than their actual fitness. This selection method still has a bias towards fitter solutions but also allow all solutions in the population to have a chance to be selected.

After the parents are chosen, the new population can be produced by applying genetic operators to parents. Generally, genetic operators include crossover and mutation.

Crossover is a random exchange of genetic information between two parent strings to produce offspring strings. Single point crossover is a simple crossover; the name means that there is only one crossover point. Once the crossover point is generated randomly in a pair of parent strings, the substrings that are defined by that the crossover point will be recombined together and produce two children. Firgure 4-2 is an example of single point crossover.

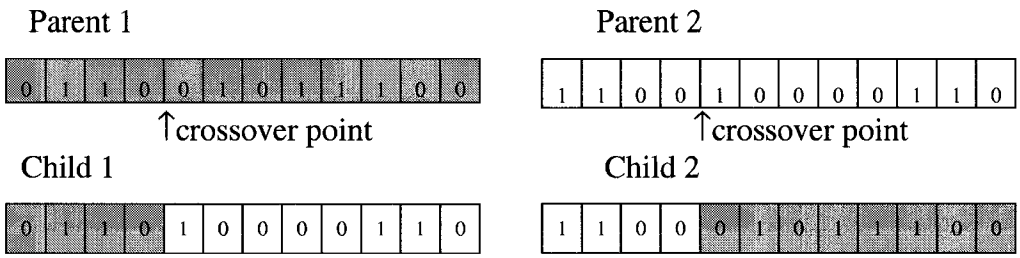

Figure 4-2 Single point crossover of GA

Figure 4-2 shows how the encoding of the genetic information of the parents is kept in the children.

Multipoint crossover requires more than one crossover points in a pair of selected parent strings. The working principle is shown in Figure 4-3.

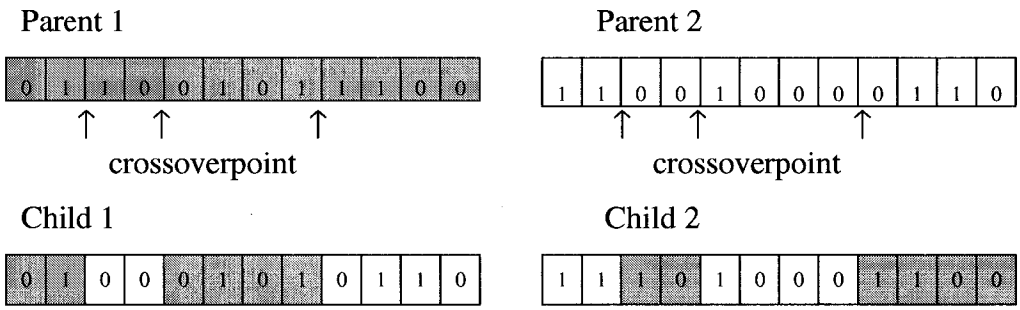

Figure 4-3: Multipoint crossover of GA

Mutation is used to keep the diversity in the population and prevent GA from being trapped in a local optimum. Typically, a simple mutation is implemented by flipping one bit that is randomly selected in the string infrequently, i.e. changing 1 to 0 and vice versa with a small probability.

Figure 4-4 shows an example of a simple mutation.

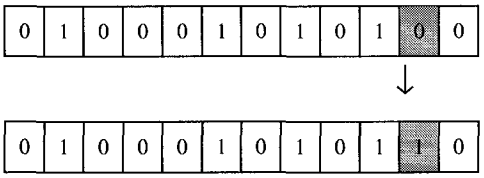

Figure 4-4: Mutation of GA

As Darrell Whitley pointed out in [Dar93], Genetic Algorithms are often described as a global search method that does not use gradient information. Thus it is a more general working method than other global optimization methods.

4.2 Simulated Annealing

Simulated Annealing originates from the analogy between the annealing process of solids and the problem of solving combinatorial optimization problems. In condensed matter physics, annealing is a process that cools a solid in a heat bath to reach a minimal energy state(ground state). At initial high temperatures, all molecules of the solid randomly arrange themselves in a liquid state, and as the temperature descends gradually, the crystal structure becomes more ordered and reaches a frozen state when the temperature drops to zero. If the temperature drops too quickly the cristal will not reach the thermal equilibrium at each temperature, hence, the defects will be frozen into the crystal structure and the crystal will not reach a minimal energy state but a meta-stable state (i.e. being trapped in a local minimum energy state). So a proper initial temperature and a proper cooling schedule are important to an annealing process. Metropolis designed a Monte Carlo method to simulate the annealing process at a fixed temperature. An initial state of a thermodynamic system was chosen at initial energy E and initial temperature T, the initial state is perturbed and the change in energy ΔE is computed. If the change in energy is negative the new perturbed state is accepted. If the change in energy is positive the new perturbed state is accepted with a probability given by the Boltzmann factor $exp\ -(\Delta E/T)$. This processes is then

repeated sufficient times to give good sampling statistics for the current temperature, and then the temperature is decremented and the entire process repeated until a frozen state is achieved at T=0.

In the 1980s, Kirkpatrick, Gelatt and Vecchi and Cerny [KGV83], suggested that a form of simulated annealing could be used to solve complex optimization problems. In their research, the current state of the thermodynamic system is analogous to the current solution to the combinatorial problem, the energy equation for the thermodynamic system is analogous to the objective function, and the ground state is analogous to the global minimum in the whole search space. At the beginning of the SA search, an initial temperature is set up and a solution of the target optimization problem is generated randomly in the search space. The new solution will be selected randomly from the neighborhood of the current solution that is analogous to the state perturbation of the Metropolis simulation. Solution transition will be accepted according to the Metropolis acceptance criterion. At each temperature, this process is continued until equilibrium is reached, and this process will repeat at each temperature while the temperature gradually drops down until the temperature reaches a very low value (obviously, at this temperature no solution transition will be accepted).

Osman et al [OK96] summarized the SA algorithm as below:

Step 1: Generate an initial random or heuristic solution S.

Set an initial temperature T, and other cooling schedule parameters.

Step 2: Choose randomly $S' \in N(S)$, and compute $\Delta = C(S') - C(S)$

Step 3: **If**:

(i) S' is better than S $(\Delta < 0)$,or

(ii) S' is worse than S but "accepted" by the randomization process at the present temperature T, i.e. $e^{\left(\frac{-\Delta}{T}\right)} > \theta$, (where $0 < \theta < 1$ is a random number).

Then replace S by S'.

Else Retain the current solution S.

Step4: Update the temperature T depending on a set of rules, including:

(i) The cooling schedule used.

(ii) Whether an improvement was obtained in Step 3 above.

(iii) Whether the neighborhood N(S) has been completely searched.

Step 5: If a "stopping test" is successful stop, else go to Step 2.

Figure 4-5: SA algorithm

Though simulated annealing is a general purpose search strategy, a number of decisions should be made during its implementation. In [Dow93], Dowsland classified the implementation decisions into two categories: generic decisions, which affect the search process itself and problem, and specific decisions, which depend on the problem domain and representation.

The generic decisions are those involving the parameters of the cooling schedule, which are the initial temperature, how the temperature is reduced and how many neighborhood solutions should be examined at each temperature. Numerous theoretical and practical cooling schedules have been presented in the literature. Though there is no fixed rule for setting the cooling schedule, the desirable guides for

the parameters of the cooling schedule are suggested. Firstly, the initial temperature should be "high enough" to allow reasonably free exchange of solutions around the search space. Secondly, the temperature should cool slowly enough to allow the search to escape from the local optimal solutions. Finally, the iterations at each temperature can be a constant number and also can change as the search progress, for example, when the temperature is low, the iterations increase to allow the search to fully examine the neighborhood space.

The specific decisions are concerned with how to represent the solution space, define the neighborhood structure and quantify the cost function.

A solution's neighborhood is defined as reachable solutions to it. In other optimization problems, it can be very complicated and flexible. Since depending on different specific features of the solution, we can define different neighborhood structures. This thesis addresses the software testing problem, so the neighborhood structure is relatively easy to define, but the neighborhood range still has effect on the efficiency of SA, which will be shown in our results and will be discussed later.

While in the original SA, the neighborhood range is fixed, Corona [Cor87] proposed an adaptive method of SA for continuous optimization problems. This method tends to adjust the neighborhood range to keep the acceptance rate of 0.5. This will be discussed in detail in the next section.

The cost function is used to evaluate the solution of the problem. An appropriate cost function is essential to the implementation of Simulated annealing. It should represent the problem, provide guidelines to desirable areas of the search space and therefore lead the search to the global optimum. The cost function selected will be calculated for every solution during the whole search process, so it should be calculated efficiently, which is also important when we consider the cost function.

4.3 Simulated Annealing with Advanced Adaptive Neighborhood (SA/AAN)

Different from the original Simulated Annealing, the neighborhood range of Simulated Annealing with Advanced Adaptive Neighborhood (SA/AAN) is not fixed. In [Cor87], the neighborhood range is adjusted to keep the acceptance rate of 0.5.

This method uses the following equations to control the neighborhood range g(p)in continuous optimization problems.

$$g(p)=\begin{cases} 1+c\dfrac{p-p1}{p2} & if\ p>p1 \\ (1+c\dfrac{p2-p}{p2})^{-1} & if\ p<p2 \\ 1 & otherwise \end{cases} \qquad (4.2)$$

$$p=n/N \qquad (4.3)$$

Where c is a scaling parameter, $p1$=0.6, $p2$=0.4. The acceptance rate p can be calculated from the number of acceptance n within the period N where the neighborhood range is constant.

The generation of the new candidate solution becomes very easy with the application of the Corona's method for continuous optimization problems solved by SA. Let xi be the current solution, r is a uniform random number with the interval [-1, 1] and m is the neighborhood range. The next candidate solution xi' can be generated by the following.

$$xi' = xi + rm \qquad (4.4)$$

According to Corona's method, the neighborhood range is adjusted to keep the acceptance rate of 0.5.

Mitsunori MIKI et al [MHO02] investigated the performance of Corona's method and found out that when the solution is at a local optimum and the neighborhood range is smaller than the distance of local optimum area, the magnification factor of Corona's method is not big enough to allow the solution to escape from the local optimum. Thus, the solution is trapped to the local optimum. So they proposed another new adaptive method, named SA/AAN, for controlling the neighborhood range in continuous optimization problems to obtain good solutions in shorter annealing steps. This method introduces a parameter to control

$$g(p)=\begin{cases} H0(p') & if\ p>p1 \\ 0.5 & if\ p<p2 \\ 1 & otherwise \end{cases} \qquad (4.5)$$

$H0 = H0 \times H1$ Initial value of $H0$=2.0

$$H1 = \begin{cases} 2.0 & if\ p' > p1 \\ 0.5 & if\ p' < p2 \\ 1.0 & otherwise \end{cases} \quad (4.6)$$

Where p is calculated by the same equation in Corona's method and p' is calculated by the following,

$$p' = l/L \quad (4.7)$$

L is the neighborhood range's parameter adjustment interval, which is set as 200 in their experiments, while l is the number of acceptance within the period L. The acceptance rate is set to 0.5 at the beginning, then it is decreased gradually in the annealing process.

The acceptance rate in SA/AAN can be maintained to a low value, which is different from Corona's method. This allows the neighborhood range to decrease gradually and sometimes increase on a large scale and this prevent the solution to be stuck in the local optima. In their experiments, this method is found to be very effective in continuous optimization problems.

4.4 Genetic Simulated Annealing (GSA)

As discussed before, SA and GA are powerful methods in optimization problems, while both of them have their own limitations. In [KKD95], Seichi Koakutsu et al discussed the characteristic of SA and GA. One of the essential features of SA is its stochastic hill climbing. SA introduces small random changes in the neighborhood so that it can search the solution space exhaustively but this also cause a weakness of this method, which is too computation-intensive. On the other hand, the crossover operation of GA and the population of GA allow it to search for the global optimum in the large search solution space roughly and quickly, but it has no explicit way to create the small moves in the solution space [KKD95]. In order to combine the good features of these two methods, Seichi Koakutsu et al [KKD95] designed a new method, named Genetic Simulated Annealing. Genetic simulated annealing combines the hill-climbing feature of SA and the crossover operation of GA. The process of GSA is presented in [KKD95] as follows.

```
GSA_algorithm(Np,Na,T0, α)
{    X={x1,...,xNp};
       xL*= the best solution among X;
       xG*= xL *; /initialize the global best-so-far/
    while (stop criterion is not met)
    {T=T0;  /initial temperature/
       /jump/
       select the worst solution xi from X;
         select two solutions, xj, xk from X such that f(xj)<>f(xk)
              xi=Crossover(xj, xk)
       /SA based local search/
       while not frozen or not meet the stopping criterion)
       {for (loop=1; loop<=Na; loop++)
              {x'=Mutate(xi);
              Δf=f(x')-f(xi);
              r=rand()
              if (Δf<0 or r<exp(-Δf/T))
              xi =x';
              if (f(xi)<f(xL*))
              xL*= xi;}
       T=T×α  /lower temperature/}
       if (f(xL*)<f(xG*))
       xG*= xL*;
       xi= xL*;
    f(xL*)=unlimited; }
       return xG*}
```

Figure 4-6 GSA algorithm

As Figure 4-6 shows, GSA starts with a population that the population size is Np. There are three main operations in GSA: SA-based local search, GA-based crossover operation and population update. SA-based local search creates the small change in

the local search space and preserves the local best-so-far solution. GA-based crossover operation creates big jumps in the search space when the search comes to large flat areas or the system is frozen. Note that the parents selection in [KKD95] is random selection, which is different from the GSA we used in our experiment. GSA updates the population by replacing the worst solution, which is conducted in two different ways. The worst solution in the population is replaced with the solution produced by the crossover. Furthermore, at the end of the local SA-based search, the worst solution is replaced with the local best-so-far solution in the local SA-based search.

Genetic Simulated Annealing (GSA) is applied to the Non-slicing floor-plan design problems, which is the one of the most difficult problems in layout, and is compared with SA. The result in [KKD95] showed that GSA improved the average chip area by 12.4% and the average wire length by 2.95% over SA with the same computing resource.

Chapter 5 Description of experimental studies

5.1 Description of strategy taken

The works presented in this thesis is based on the approach that transforms a problem of automatic generation of test cases into a function minimization problem. The goal of this section is to explain this approach. This approach has been adapted from a system called GADGET [MMS01].

5.1.1 Test adequacy criterion

As it was discussed in chapter 3, condition-decision coverage is a test adequacy criterion that combines condition coverage and decision coverage. To obtain the condition-decision coverage, the tester should generate such a test data that all conditions in a decision take on both *true* and *false* outcomes at least once, and exercise the *true* and *false* outcomes of every decision. Condition-decision coverage is more complicated but also more reliable than statement coverage and branch coverage. It is also less expensive than the multiple condition coverage. These are the most important reasons for selecting condition-decision coverage as the test criterion in our work.

5.1.2 Coverage table

To generate the test cases that exercise all conditional branches in the source code, we need to generate the test cases to reach those conditions first. The tester should find a way to reach the desired code location. For example, consider the following fragment of code:

```
if (tri= =0) {
        if ((i+j<=k)||(i+k<=j)||(j+k<=i))
                tri=4;
        else    tri=1;
        return tri;
        }
```

In order to execute the statement **if ((i+j<=k)||(i+k<=j)||(j+k<=i))**, the test case need to satisfy the first condition **if (tri= =0).** In the past, researchers used different

strategies to find a path leading the desired code location. This involves complicated control flow analysis for a specific path. Instead of concentrating on a specific path, another approach presented in [Chang96] is based on coverage table, which attempts to cover all the conditional branches in the whole program. This approach is also used in GADGET and our experiments. With this approach, a coverage table is generated. The purpose of this table is to keep track of all conditional branches already covered by existing test cases. Once a conditional branch is reached, that means that one branch of this condition has been taken, the function minimization is applied on that condition to find the test case which takes the other branch of this condition. Consider the following code fragment:

1: if (t= =1) printf ("triangle is scalene\n");
2: else if (t= =2) printf ("triangle is isosceles\n");
3: else if (t= =3) printf ("triangle is equilateral\n");
4: else if (t= =4) printf ("this is not a triangle\n");

Table 5-1 illustrates the coverage status of each condition or decision with test case that cover the *true* branch of line 3.

. Table 5-1: An example of coverage table

	True	False
Line 1	-	X
Line 2	-	X
Line 3	X	-
Line 4	-	-

Table 5-1 shows that the existing test case has already covered the *false* branch of line 1, line 2 and the *true* branch of line 3. The existing test cases already reach the first three conditions but haven't reached the fourth yet. According to the coverage table, the testers can apply optimization techniques to generate the test cases, which exercise the *true* branch of line1, the *true* branch of line 2, or the *false* branch of line 3. With this strategy, the testers are able to skip complicated control analysis to find the path to a specific code location.

5.1.3 Function minimization and objective function

Dynamic test data generation, as described in Chapter 3, is based on a concept of transformation of a problem of finding test cases into a problem of numerical maximization (or minimization). A function is built as the result of this transformation. This function is then maximized (or minimized) using different optimization techniques.

The key idea of the approach is the creation of a function that guides the search for a test set that satisfies the condition-decision coverage. Each branch of each condition in the target program is represented by a function that will be called an objective function. So the goal of the search process is to minimize the value of all objective functions. The objective function is used to evaluate how good the test case is. The value assigned to the function for a given test case indicates how close the test case satisfies the test criterion. Basically, the objective function is devised as follows.

- If the target condition cannot be reached, the objective function will be given a penalty which is a very large value.
- If the target condition can be reached, but the desired branch of the target condition cannot be exercised, the objective function will have a value that is between 0 and the large value, representing how good the current test case is.
- If the target branch of the target condition is exercised, the value of the objective function is 0, which is the optimum of the objective function.
- If there are more than one condition which are connected with AND or OR operators in the target decision, the + operator is used for AND operations, and the minimum operator is used for OR operations. Consider the following decision build using 3 conditions with AND operators:

  IF cond1 AND cond2 AND cond3

  In order to take the true branch of this decision, the overall objective function that combines all the conditions is:

  $\mathfrak{J}=\mathfrak{J}1+\mathfrak{J}2+\mathfrak{J}3$

  Where $\mathfrak{J}$ is the objective function value for the whole decision, and $\mathfrak{J}1,\mathfrak{J}2,\mathfrak{J}3$ are the objective function values for the single conditions in this decision.

On the other hand, consider the following decision which is built using OR operators:

IF cond1 OR cond2 OR cond3

In this case, to take the true branch of this decision, the objective function is:

$\Im$=min($\Im$1,$\Im$2,$\Im$3)

where $\Im$ is the objective function value for the whole decision, and $\Im$1,$\Im$2,$\Im$3 are the objective function values for each condition in this decision.

The form of the objective function depends on a condition existing in a branch. Some examples of conditions with their objective functions are presented in Table 5.2.

Table 5-2 Example of the objective function

Decision type	Example	Objective function
Equality	if (i= =j) (true)	if program can reach this condition $\Im$=abs(i-j)/SF else $\Im$=p
True/false	if (tri= =1) (true)	if program can reach this condition if tri= =1 $\Im$=0 else $\Im$=p*m else $\Im$=p
True/false	i+j>=k (true)	if program cannot reach the condition "if (tri= =1)&& (i+j>=k)" $\Im$=p else if i+j>=k $\Im$=0 else $\Im$=1-abs(i+j-k)/SF
True/false	if (tri==1)&& (i+j>=k) (true)	if program cannot reach this condition $\Im$=p else if tri≠1 $\Im$1=p*m $\Im$2=abs(i+j-k)/SF $\Im$=$\Im$1+$\Im$2 else if i+j>k $\Im$=0 else $\Im$=abs(i+j-k)/SF

Where p is a significant value representing the penalty of not reaching a condition; it is 2147483647 in our experiments. m is a constant value between 0 and 1. *SF* is a

constant value to make sure that *abs(i+j-k)* is always smaller than 2147483647, and it is 3 in the Triangle classification program.

EXAMPLE: Let's look at a fragment of program as an example illustrating the construction of objective function that will be minimized. Let the code be like the following:

if(i==j)

... ...

if (tri==1)&& (i+j>=k)

In order to seek test data to excise both the *true* and *false* outcomes of the condition **if(i==j)**, we need to seek test data to reach this condition. If the program's execution fails to reach this code, the objective function will be given a worst value, i.e. large value p. If we need to take the *true* branch of this condition, and the program can reach this condition, the objective function will be given a value **abs(i-j)/SF** to measure how close i and j are to each other. So the objective function of *true* outcome is shown below:

$$True\ \Im = \begin{cases} p & unreached \\ abs(i-j)/SF & otherwise \end{cases}$$

For example, p =2147483647 and SF=3. If we need to take the *false* outcome of the condition, if i is equal to j, the objective function will be given a poor value, otherwise the objective function value will be 0. So the objective function of *false* outcome is shown below:

$$False\ \Im = \begin{cases} 0 & i \neq j \quad reached \\ p' & i = j \quad reached \\ p & unreached \end{cases}$$

where p =2147483647, P'=0.7*P and SF=3. Note that in the above situation, the two branches of the condition are also the two branches of the decision.

In the following line of the code, there are two conditions in the decision **if(tri==1)&& (i+j>=k);** we need to evaluate two conditions separately and we also need to evaluate the whole decision. The objective functions are shown below:

For condition 1 **(tri==1),**

$$True\ \Im 1 = \begin{cases} 0 & tri = 1 \quad reached \\ p \times 0.45 & tri \neq 1 \quad reached \\ p & unreached \end{cases}$$

$$False\ \Im 1 = \begin{cases} 0 & tri \neq 1 \quad reached \\ p \times 0.45 & tri = 1 \quad reached \\ p & unreached \end{cases}$$

For condition 2 **(i+j>=k),**

$$True\ \Im 2 = \begin{cases} 0 & i + j \geq k \quad reached \\ abs(k - i - j)/SF & i + j < k \quad reached \\ p & unreached \end{cases}$$

$$False\ \Im 2 = \begin{cases} 0 & i + j < k \quad reached \\ 1 + abs(i + j - k)/SF & i + j \geq k \quad reached \\ p & unreached \end{cases}$$

For the decision **if(tri==1)&& (i+j>=k),**

True $\Im$=$\Im$1+$\Im$2

False $\Im$=Minimum ($\Im$1,$\Im$2)

Using the generated objective functions, the test data generator can use optimization techniques to search for the test cases that satisfy the given test adequacy criterion.

5.1.4 Generation of test cases - methodology

The first step in generation of test cases is derivation of test requirements. These test requirements are derived from condition-decision coverage of the target program. Recalled from Section 2.2.4, this leads to two test requirements for each condition and this means each condition should take both *true* and *false* branch at least once. This also leads to two test requirements for each decision if the decision is built with multiple conditions (in GADGET [MMS01], they believed that these two test requirements are satisfied by any test set that meets the test requirements for each condition). As discussed in the Section 2.2.4, this is not always the case. Each test requirements is represented as the corresponding objective function.

The second step deals with establishing a coverage table. The purpose of the table is to record the condition information of the target program, and keep track of

whether a test requirement is tested or not. Based on the test requirements derived from the test adequacy criterion, the target program is instrumented with additional code for each condition. This allows for reporting the value of parameters in the target program. These values will be used to calculate the value of the objective function during the search process.

Before the test data generation system starts the search process, a seed input is generated randomly in the particular input space based on the specification. The seed is used to execute the program under test for the first time. Typically, the first execution covers some percentage of source code, which means that some test requirements are satisfied. After the first execution, the seed input and the coverage percentage are recorded and the coverage table is initialized. The initialized coverage table provides the information of the reached test requirements.

According to the coverage table, the test data generator finds out which condition that can be reached and have not yet been covered completely (see the Section 5.1.2). The test data generation system then applies optimization techniques to optimize an objective function built on each reached test requirement in turn. For each reached but unsatisfied test requirement, the test data generation system attempts to use function minimization techniques to satisfy them.

During the search process, the initial population is seeded with the test cases which causes the current test requirement to be reachable. If the size of the population is larger than the number of current test cases, which makes the test requirement reachable, then the generation system will generate the additional test cases randomly in the input space. Whenever there is a new test requirement being satisfied or being reached in the search process the coverage table is updated, and the test cases, the number of target program is executed and the coverage percentage are also recorded for future use.

For each reached but unsatisfied test requirement, the search process is repeated until the stopping criterion is met. The stopping criterion is either the finding of successful test data that satisfies the current test requirement or the reaching of the maximum number of iterations by the test data generator. The test data generator then starts another search process for the next reachable test requirement.

After all of the reachable test requirements have been subjected to the search, GADGET stops. While in our work, if there are still unreachable test requirements in the coverage table, the test data generator applies function minimization on those unreachable test requirements. In this process, the seed input are generated randomly.

At the end of the working process for each target program, the coverage percentage and the corresponding target program execution times are kept for the later result analysis.

Basically, the working process used in this work is illustrated in Figure 5-1:

Step 1 Preparation process

- Deriving test requirements according to the condition-decision coverage.
- Prepare coverage table.

Step 2 Initialization process

- Random generation of test cases.
- Initial execution of the program under test with generated test cases;
- Monitoring the coverage status of all test requirements during the execution.
- Coverage table initialization.
- Store seed and relevant information.

Step 3 Reached test requirements search process (using optimization algorithms)

- Seeded with the test cases that reach the current test requirement; additional test cases are generated randomly if it is necessary.
- Execute the target program with every test set; record the coverage, the test cases and the relevant information; update the coverage table.
- Repeat the search process for current test requirement until the stooping criterion is met.

Step 4 Repeat step 3 until all of the reached test requirements have been subjected to the search

Step 5 Repeat step 3 for the unreachable test requirements; seeded with random test input.

Figure 5-1 Working process of the test data generation system

5.2 Experimental setup

This section introduces the methodology used to conduct the experiments presented in this thesis. This includes the description of the optimization algorithms used, the programs under test and the procedures used in this work.

5.2.1 Optimization algorithms

Four optimization algorithms are used in the experiments; they are Genetic Algorithm (GA), Simulated Annealing (SA), Genetic Simulated Annealing (GSA) and Simulated Annealing with Advanced Adaptive Neighborhood (SA/AAN). The overall description of these optimization algorithms has been presented in Chapter 4.

Random test data generator is also used for the purpose of comparison.

The following describes the basic implementation decisions of these search methods used in this thesis.

GA: Binary gray code Genetic Algorithm is used in our experiments. The selection schema in GA is Roulette wheel sampling selection, which selects individuals based on their fitness value. Single point crossover and the uniform mutation are applied.

SA: In the experiments, the neighborhood is generated by incrementing and decrementing the input data. For example, consider there are three input parameter x, y, z, and the current test case is (x_i, y_i, z_i). Every element in the neighborhood N is defined as $(x_i \pm r_1 m_1, y_i \pm r_2 m_2, z_i \pm r_3 m_3)$, where m_1, m_2, m_3 are three randomly generated values between 0 and 1, and r_1, r_2, r_3 are the neighborhood ranges for each input parameter respectively.

SA/AAN: The neighborhood is generated by incrementing and decrementing the input data.

GSA: The selection schema in GSA is binary tournament selection, which selects the individual with the better fitness in a tournament pair. Single point crossover and the un-uniform mutation are applied.

Random: Random test data generator generates the test data pseudo-randomly from the input space and use these test data to execute the programs under test.

5.2.2 Implementation of test data generation

All of the target programs in our experiments are written in C/C++, and the test data generator is written in Ruby. An I/O project is created as a channel for transferring information between the test data generator and the target program. The basic working process is as follows:

> The test cases are generated by the test data generator; the test case is written into a I/O object, which works as a channel between Ruby and C/C++. The test case is then transferred to the target program as input data through the I/O object and the target program is executed with this input. During the execution, the instrumented code in the target program output some message and write them into the I/O object. When program execution finishes, the test data generator will read all of the messages from the I/O object as a string and extract the useful information from this string. This information is used to calculate the objective function and update the coverage table. In some cases, the target program will also output some messages to the screen, such as "**Please enter the number**". Since skipping these messages does not change the structure of the target program, in our experiments, these messages output are disabled in order to simplify the I/O processing,

5.2.3 Tested programs

Five different c/c++ programs are tested in our experiments. Although they are not large programs, all of them involve some amount of nested conditional structures, and some of them have very complicated compound conditions, which increase the complexity of the programs.

The programs tested in this work are as follows:

1 Hex_dec conversion

Hex_dec conversion is a program, which decides if the input string is a legal hexadecimal number and converts the legal hexadecimal number to a decimal number. The input string has a limited number of characters; in our case, the number of characters varies in a range between 3 and 13. The possible characters include almost all of the characters that can be entered from a keyboard.

2 Timeshuttle [IM]

Timeshuttle is a program that requires the user to input a destination date of the time travel, which includes month, day and year. It returns a corresponding message to the user. In the experiments, the input data is limited to (0,64) for month, (0,128) for day and (0,16192) for year.

This program has 40 decisions and 17 functions.

3 Perfect number

The perfect number program is used to decide if a number is a perfect number and if it is a prime number. This program includes 16 decisions, 17 conditions.

4 Triangle classification

A triangle classification program, which includes some nested conditions and an enumerated data type variable, is used in our experiments as well as in Michael's approach [MMS01]. The triangle classification requires a user to input three integers as three sides of a triangle, and if then decides which type of triangle it is. The output to user has four results, " scalene", "isosceles", "equilateral" and "not a triangle". This program has 14 decisions.

5 Rescue [IM]

The rescue program requires the user to input a number, and decides if the input is a legal secret code. The legal secret code should be a 5-digit number and satisfies some other rules. If the input is legal secret code, the program decodes the legal secret code and returns the corresponding secret message to the user. This program has 16 decisions.

The source code of these five programs is shown in the Appendices.

5.2.4 Experiment procedure

In order to make comparisons of the different optimization algorithms' performances on the target programs, the comparison is based on the number of the runs of the program under test instead of the real computation time. This is because the execution of the target program is the most time-consuming part in the test data generation, especially for the real large programs. Moreover, compared to the real computation time, this provides a uniform comparison platform without the affect of operation systems and workstation performance.

For every program under test, ten complete test-generation runs with each test generator were performed. For the **Rescue** program and the **Triangle classification** program, we also conduct the experiments on two different input spaces, which allow us to investigate the effect of input space on the performance of different optimization methods. The result tables and coverage plots are provided for each program to illustrate the results.

5.2.5 Terms (vocabulary, glossary)

The following describes the several phrases we use in this thesis.

Target program—In this thesis, the programs under test are referred as the target program.

Test requirement—It is the test objective in the search process; in this work, a test requirement means taking the *true/false* branch of a particular condition/decision. Test data generation is a process to find the test cases to satisfy the test requirements in turn.

Input space—It is the input range of the random generated test cases. It is based on the specification of the program under test and it plays a considerable role in test data generation.

Candidate test cases—In this work, the test cases that can successfully reach a condition/decision are recorded. When the test data generator starts to work on this condition/decision, the test data generator is seeded with these test cases. In this thesis, these test cases are also referred as candidate test cases.

Chapter 6 Empirical Results

Five test data generation approaches are applied to a set of programs to obtain complete condition-decision coverage. A number of experiments on each program are performed and results are reported.

This chapter gives a detailed analysis of each target program and the comparison of performance of five test generation systems on each program.

6.1 Hex_dec conversion

6.1.1 Analysis of the source code

The code of Hex_dec conversion is provided in the Appendices. The decision branches are shown below.

```
... ...
            ... ...
            if(c>='0' &&c<='9'||c>='a'&&c<='f'||c>='A'&&c<='F') //1
                ... ...
            else
... ...
            if (i<=MAX) //2
            ... ...
            else printf("\nMaximum 7 digits of hex number");
            ... ...
            for (j=0;s[j]!='\n'; j++) //3
            if (s[j]>='0'&&s[j]<='9') //4
                ... ...
            if (s[j]>='a'&&s[j]<='f') //5
                ... ...
```

if (s[j]>='A'&&s[j]<='F') //6

... ...

... ...

There are 6 decisions in **Hex_dec** that need to be evaluated. 36 test requirements need to be satisfied to obtain complete condition-decision coverage. Thus, maximum of 36 objective functions are generated to allow the test generators to calculate the value of the objective function $\Im(x)$. For example, consider the following fragment of code:

if(c>='0' &&c<='9'||c>='a'&&c<='f'||c>='A'&&c<='F')

{...

}

The instrumented code is shown below (Italic character means the instrumented code):

printf("&");

printf("%c",c);

if(c>='0' &&c<='9'||c>='a'&&c<='f'||c>='A'&&c<='F')

{p*rintf("@");*

...}

During the execution, the instrumented code reports the value of **c** to the test data generation system. This allows for calculating the objective function and measures how close it is to the desired value.

There are 6 conditions and one decision need to be evaluated independently. According to the requirement of condition-decision coverage, 14 test requirements should be satisfied. Thus, 14 objective functions are generated as below.

To ensure the decision

If(c>='0' &&c<='9'||c>='a'&&c<='f'||c>='A'&&c<='F')

takes value "*true*", the following function is built:

$$True:\ \Im(x)=\begin{cases} 0 & '9'\geq c\geq'0','f'\geq c\geq'a'',''F'\geq c\geq'A' & reached \\ \min imum(\Im 1,\Im 2,\Im 3) & otherwise & reached \\ p & & unreached \end{cases}$$

where $\Im 1(x)=\begin{cases} c-'9' & c>'9' \\ '0'-c & c<'0' \end{cases}$

$\Im 2(x)=\begin{cases} c-'F' & c>'F' \\ 'A'-c & c<'A' \end{cases}$

$\Im 3(x)=\begin{cases} c-'f' & c>'f' \\ 'a'-c & c<'a' \end{cases}$

To ensure the decision

If(c>='0' &&c<='9'||c>='a'&&c<='f'||c>='A'&&c<='F')

takes value "*false*", the following function is built:

$$\Im(x)=\begin{cases} 0 & unsatisfied('9'\geq c\geq'0','f'\geq c\geq'a','F'\geq c\geq'A') & reached \\ \Im 1+\Im 2+\Im 3 & otherwise & reached \\ p & & unreached \end{cases}$$

where $\Im 1(x)=\begin{cases} c-'9' & c>'9' \\ '0'-c & c<'0' \end{cases}$

$\Im 2(x)=\begin{cases} c-'F' & c>'F' \\ 'A'-c & c<'A' \end{cases}$

$\Im 3(x)=\begin{cases} c-'f' & c>'f' \\ 'a'-c & c<'a' \end{cases}$

In the similar way, functions representing other 12 cases are built:

c>='0'

True: $\Im(x)=\begin{cases} 0 & c\geq 0 & reached \\ '0'-c & c\leq 0 & reached \\ p & & unreached \end{cases}$

False: $\Im(x)=\begin{cases} 0 & c<0 & reached \\ c-'0'+1 & c\geq 0 & reached \\ p & & unreached \end{cases}$

c<='9'

True: $\Im(x)=\begin{cases} 0 & c\leq'9' & reached \\ c-'9' & c>'9' & reached \\ p & & unreached \end{cases}$

$$\textit{False: } \Im(x)=\begin{cases} 0 & c>'9' \quad \textit{reached} \\ '9'-c+1 & c\le'9' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

c>='a'

$$\textit{True: } \Im(x)=\begin{cases} 0 & c\ge'a' \quad \textit{reached} \\ 'a'-c & c\le'a' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

$$\textit{False: } \Im(x)=\begin{cases} 0 & c<'a' \quad \textit{reached} \\ c-'a'+1 & c\ge'a' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

c<='f'

$$\textit{True: } \Im(x)=\begin{cases} 0 & c\le'f' \quad \textit{reached} \\ c-'f' & c>'f' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

$$\textit{False: } \Im(x)=\begin{cases} 0 & c>'f' \quad \textit{reached} \\ 'f'-c+1 & c\le'f' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

c>='A'

$$\textit{True: } \Im(x)=\begin{cases} 0 & c\ge'A' \quad \textit{reached} \\ 'A'-c & c\le'A' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

$$\textit{False: } \Im(x)=\begin{cases} 0 & c<'A' \quad \textit{reached} \\ c-'A'+1 & c\ge'A' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

c<='F'

$$\textit{True: } \Im(x)=\begin{cases} 0 & c\le'F' \quad \textit{reached} \\ c-'F' & c>'F' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

$$\textit{False: } \Im(x)=\begin{cases} 0 & c>'F' \quad \textit{reached} \\ 'F'-c+1 & c\le'F' \quad \textit{reached} \\ p & \textit{unreached} \end{cases}$$

Where p is a significant value. In the experiments, p=2147483647.

6.1.2 A Comparison of Five Test Data Generations Approaches

Ten complete test-generation runs with each test generator were performed for the Hex_dec program. The following tables show the test results of running GA, GSA, SA, SA/AAN and Random test generator to the Hex_dec program. The input data is limited to a string of characters, whose length is between 3 and 13. That means users can enter from 3 to 13 characters as the input data. The minimum length is 3 to ensure the complexity of the problem, otherwise, if there is no limitation of the minimum length, the problem will become very easy to solve. For example, to generate a single digit hexadecimal number is much easier than to generate a 3 digit one. The possible characters include most characters that can be input from the keyboard. For each test generator, the condition-decision coverage and the stopping criterion are showed in each result table, as well as the average coverage percentage of ten runs.

Table 6-1 Result table of Random Generator

No.	Covered Test requirements (Maximum: 36)	Percentage coverage (%)	Stopping criterion
1	35	97.22	Random test data generator stops after 10,000 target-program execution.
2	35	97.22	
3	35	97.22	
4	34	94.44	
5	35	97.22	
6	34	94.44	
7	34	94.44	
8	35	97.22	
9	33	91.67	
10	35	97.22	
Average coverage	**34.5**	**95.83**	

Table 6-2 Result table of Genetic Algorithm
(Population size: 50, Generation: 30)

No.	Covered Test requirements (Maximum: 36)	Percentage Coverage (%)	Stopping criterion
1	35	97.22	For each test requirement, GA stops when it finds the test case, which satisfies the test requirement, or after 30 generations if it still cannot satisfy the test requirement.
2	35	97.22	
3	35	97.22	
4	35	97.22	
5	35	97.22	
6	35	97.22	
7	35	97.22	
8	35	97.22	
9	35	97.22	
10	35	97.22	
Average coverage	**35**	**97.22**	

Table 6-3 Result table of Simulated Annealing
(Neighbor: 80)

No.	Covered Test requirements (Maximum: 36)	Percentage coverage (%)	Stopping criterion
1	35	97.22	For each requirement, SA stops when it finds the test case which satisfies the test requirement or stops after 25 temperature steps if it still cannot satisfy the test requirement.
2	35	97.22	
3	35	97.22	
4	35	97.22	
5	35	97.22	
6	35	97.22	
7	35	97.22	
8	35	97.22	
9	35	97.22	
10	35	97.22	
Average coverage	**35**	**97.22**	

Table 6-4 Result table of Genetic Simulated Annealing
(Population size: 50, Generation: 800, Neighbor: 10)

No.	Covered Test requirements (Maximum: 36)	Percentage coverage (%)	Stopping criterion
1	34	94.44	For each requirement, GSA stops when it finds the test case which satisfies the test requirement or stops after 800 generations if it still cannot satisfy the test requirement.
2	28	77.78	
3	35	97.22	
4	32	88.89	
5	32	88.89	
6	32	88.89	
7	16	44.44	
8	34	94.44	
9	34	94.44	
10	34	94.44	
Average coverage	**31.1**	**86.39**	

Table 6-5 Result table of Simulated Annealing with Advanced Adaptive Neighborhood
(Neighbor: 50)

No.	Covered Test requirements (Maximum: 36)	Percentage coverage (%)	Stopping criterion
1	35	97.22	For each requirement, SA/AAN stops when it finds the test case which satisfies the test requirement or stops after 40 temperature steps if it still cannot satisfy the test requirement.
2	35	97.22	
3	35	97.22	
4	35	97.22	
5	35	97.22	
6	35	97.22	
7	35	97.22	
8	35	97.22	
9	35	97.22	
10	35	97.22	
Average coverage	**35**	**97.22**	

The above result tables show the different performance of different test data generations. GA, SA and SA/AAN exhibit the best performances, which are close to 100% coverage; Random generator also almost achieves the same coverage as well. On the other hand, the average performances of GSA test generation are below 90%. The result table of GSA shows that there is one run that only achieves 44.44% coverage.

The most challenging test requirement in the Hex_dec program is to take the *false* branch of the second decision

if (i<=MAX)

which requires that the test case is a valid Hexadecimal number, and the length of the input string should be greater than 7. Note that in our experiments, the possible input characters include most characters that can be entered from the keyboard. This makes it tough to generate a valid 8 digit of Hex number. In our experiments, none of the test data generator can generate a test case to satisfy this condition.

6.1.3 Coverage plots for Five Test Data Generators

Figure 6-1 shows the coverage plots of five test data generators, which summarize graphically the results of the experiments. It represents obtained coverage as a function of number of executions of program under test. Comparing to other programs used in this work, the structure of Hex_dec program is much simpler and has much less lines of code, thus, all of the five test data generators perform well on it. The graph shows that GA, SA and SA/AAN have the best performance. Though GA, SA and SA/AAN achieve almost full coverage after they execute the Hex_dec program 4500 times, there is a slight difference between SA and other two test generators. The coverage plot in Figure 6-1 shows this difference in the experiments. To obtain 90% coverage, GA and SA/AAN only need to execute the program less than 500 times, while SA needs 1500 times. GA and SA/AAN also hit their peaks earlier than SA. By contrast, to obtain 90% coverage, Random generator needs to execute the target program 5500 times and needs another 2000 times of target-program execution to hit the peak. GSA performs poorly on the Hex_dec program compared to the other four test data generators; the performance of GSA is even worse than Random generator. The result of the experiments shows that the coverage of GSA is still below 90% after 9500 times of target-program execution. GSA hit its peak in about 5500 times of target-program execution, after that, it fails to improve.

Generally, GA and SA/AAN perform better than other three test data generators.

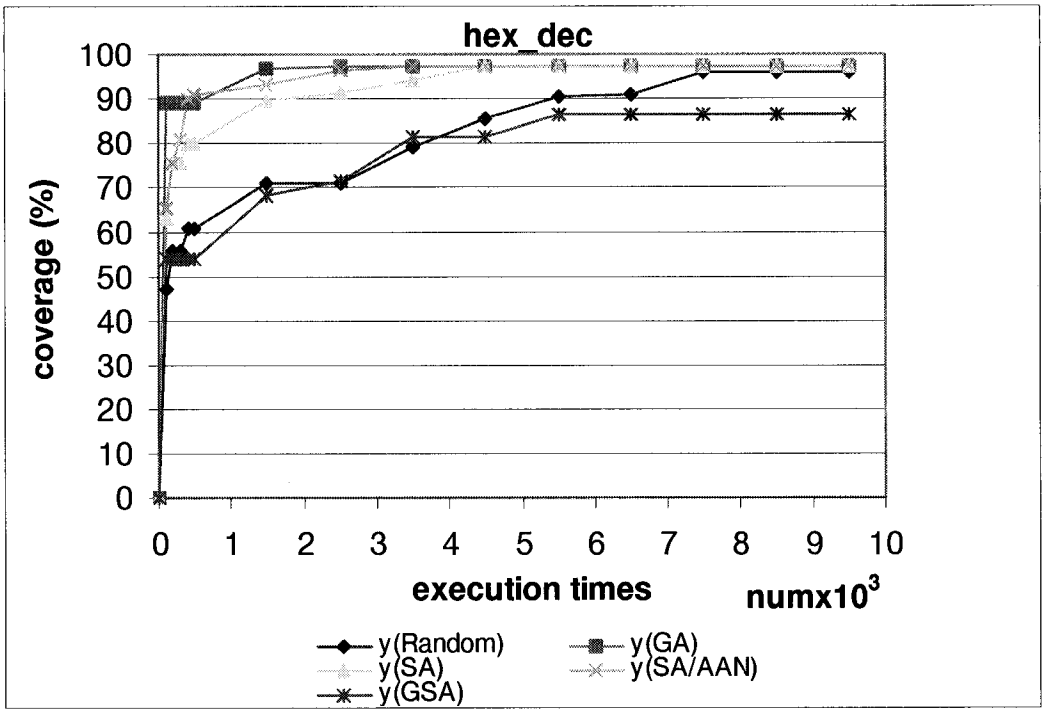

Figure 6-1 Coverage plots of five search methods on Hex_dec program

6.1.4 GA and SA/AAN: Two methods that have best performance

This section concentrates on discussing two test data generators that have the best performance on the Hex_dec program. For SA/AAN and GA test data generator, the maximum, the minimum and the average coverage of each method in 10 experiments are plotted against the number of the execution of the Hex_dec program, and shown in Figure 6-2.

Even if both the GA and SA/AAN achieve the same coverage in 3500 times of target-program execution, there is still a difference between these two generators. GA hits the peak in the very early stage of some experiments, which only take 100 times of target-program execution, though some experiments need 2500 times. In contrast, to achieve the highest coverage, SA/AAN needs to execute the target program 1500 times in its best run. While in its worst run, SA/AAN needs to execute the target program 3500 times to hit the peak.

Generally, though GA and SA/AAN can achieve the same highest coverage, SA/AAN needs more effort than GA to achieve the same coverage.

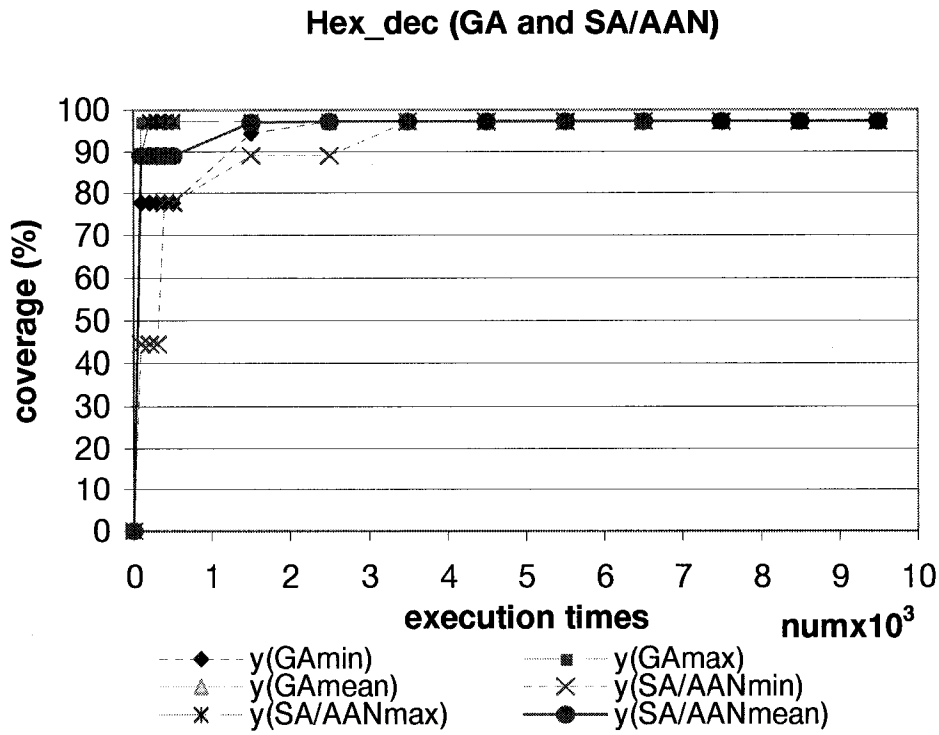

Figure 6-2 Comparison of GA and SA/AAN on Hex_dec program

6.2 Timeshuttle

6.2.1 Analysis of the source code

There are 40 decisions in the Timeshuttle program, which are identified as below.

```
int main( void )
{
        … …
        if (validInput) //1
        {
                … …
        }
        else
        {
                … …
        }
        … …
}
void plannedTrip(Month mToday, int dToday, int yToday, Month m, int d, int y)
{
        … …
}
```

```
void randomTrip(Month mToday, int dToday, int yToday, Month m, int d, int y)
{
        … …
}
bool isLeapYear(int year)
{
        … …
}
int gregorianDay(Month m, int d, int y)
{
        … …
        if (isValidDate(m, d, y)) //2
        {
                … …
                if (y == YEAR1) //3
                {
                        … …
                }
                else
                {
                        … …
                        for (int i = YEAR1+1; i < y; i++) //4
                        {
                                … …
                        }
                        … …
                }
                … …
        }
        … …
}
bool isValidDate(Month m, int d, int y)
{
        … …
        if (m < JAN || m > DEC) //5
                … …
        else if (d > daysInMonth(m, y) || d < 1) //6
                … …
        else if (y < YEAR1 || y > YEARMAX) //7
                … …
        else if (y == YEAR1) //8
        {
                if (m < OCT) //9
                        … …
                else if (m == OCT && d < 15) //10
                        … …
```

```
            else
                ... ...
        }
        else
            ... ...
        }
int getYearDay(Month m, int d, int y)
{
        ... ...
        for (Month mo=JAN; mo<m; mo = static_cast<Month>(mo + 1)) //11
        {
            ... ...
        }
        ... ...
}

Weekday getWeekday(int gDay)
{
        ... ...
}
string dayName(Weekday w)
{
        string name;
        switch(w)
        {
            case SUN: //12
                name = "Sunday"; break;
            case MON: //13
                name = "Monday"; break;
            case TUE: //14
                name = "Tuesday"; break;
            case WED: //15
                name = "Wednesday"; break;
            case THU: //16
                name = "Thursday"; break;
            case FRI: //17
                name = "Friday"; break;
            case SAT: //18
                name = "Saturday"; break;
        }
        ... ...
}

string monthName(Month m)
{
        ... ...
```

```
        switch(m)
        {
                case JAN: //19
                        name = "January"; break;
                case FEB: //20
                        name = "February"; break;
                case MAR: //21
                        name = "March"; break;
                case APR: //22
                        name = "April"; break;
                case MAY: //23
                        name = "May"; break;
                case JUN: //24
                        name = "June"; break;
                case JUL: //25
                        name = "July"; break;
                case AUG: //26
                        name = "August"; break;
                case SEP: //27
                        name = "September"; break;
                case OCT: //28
                        name = "October"; break;
                case NOV: //29
                        name = "November"; break;
                case DEC: //30
                        name = "December"; break;
        }
        return name;
}

void getTodaysDate(Month& m, int& d, int& y)
{
        ... ...
}

void gDay2MDY(int gDay, Month& m, int& d, int& y)
{
        ... ...
        if (gDay <= YEAR1DAYS) //31

        {
                ... ...
                if (gDay <= daysInMonth(OCT, y) - 14) //32

                        ... ...
                else
```

```
                {
                        … …
                }
        }
        else
        {
                … …
                while (gDay > daysInYear(y)) //33

                {
                        … …
                }
        }

        while (gDay > daysInMonth(m, y)) //34

        {
                … …
        }
        … …
}

int daysInMonth(Month m, int y)
{
        … …
        switch(m)
        {
                case JAN: case MAR: case MAY: case JUL: case AUG: case OCT:
case DEC: //35

                        … …
                case APR: case JUN: case SEP: case NOV: //36
                        … …
                case FEB: //37
                        … …
        }
        … …
}

int daysInYear(int y)
{
        … …
}

int randInt(int a, int b)
{
```

```
        ... ...
}

void outputMessage(int days, Month mToday, int dToday, int yToday,
                                        Weekday w, Month m, int d, int y)
{
        ... ...
        if (days < 0) //38
                ... ...
        else
                ... ...
        if (days < 0) //39
                ... ...
        else
                ... ...
}

bool getInputDate(Month&m, int& d, int& y)
{
  ... ...
        if (!valid) //40
                ... ...
}
```

Compared to other programs used in this thesis, this program has more functions and a more complicated relationship between the input parameters and the variables that appear in the conditions that will be evaluated. Most of the decisions do not include multiple conditions but unfortunately, there are some nested decisions. For example, consider the following fragment:

```
if (m < JAN || m > DEC)  //5
        ... ...
        else if (d > daysInMonth(m, y) || d < 1)  //6
                ... ...
        else if (y < YEAR1 || y > YEARMAX)  //7
                ... ...
        else if (y == YEAR1)   //8
        {
                if (m < OCT)   //9
                        ... ...
                else if (m == OCT && d < 15)  //10
                        ... ...
                else
```

In order to obtain a test case to satisfy the decision **if (m < OCT),** the test case also needs to satisfy the condition **if (m < JAN || m > DEC)** and **(y == YEAR1)** and take

the false branches of **(d > daysInMonth(m, y) || d < 1)** and **(y < YEAR1 || y > YEARMAX)**. Since the test data generators discussed in this thesis does not involve any static analysis of the source code, the test data generators only rely on the instrumented code to guide the search. The code is instrumented as below:

```
printf("a");
printf("%d",m);
printf("b");
if (m < OCT) //9
printf("c");
printf("%d",d);
                    ... ...
          else if (m == OCT && d < 15) //10
                    ... ...
          else
```

The instrumented code reports the value of **m** and **d** allowing for calculating the objective function.

So the generating of the objective function for the decision **if (m < OCT)** is still straightforward.

To take the "*true*" outcome of the decision **if (m < OCT)**, the objective function is generated as below:

$$\Im(x,y,z)=\begin{cases} 0 & m<oct & reached \\ m-oct+1 & otherwise & reached \\ p & & unreached \end{cases}$$

To take the "*false*" outcome of the decision **if (m < OCT)**, the objective function is generated as below:

$$\Im(x,y,z)=\begin{cases} 0 & m\geq oct & reached \\ oct-m & otherwise & reached \\ p & & unreached \end{cases}$$

where oct=10, and p is a significant value 2147483647.

There are totally 87 test requirements needed to be satisfied to obtain full condition-decision coverage in the Timeshuttle program.

6.2.2 A Comparison of Five Test Data Generation Approaches

Ten complete test-generation runs with each test generator were performed for the Timeshuttle program. The following tables 6-6, 6-7, 6-8, 6-9, 6-10 show the test

results of running GA, GSA, SA, SA/AAN and Random test generator on the Timeshuttle program. For each test generator, the condition-decision coverage, the stopping criterion and the number times the program is executed in every run are shown in their respective result tables, as well as the mean coverage percentage of ten runs.

Table 6-6 Result table of Random Generator

No.	Covered Test requirements (Maximum: 87)	Percentage coverage (%)	Stopping criterion
1	76	87.36	Random test data generator stops after 30,000 target-program execution.
2	76	87.36	
3	76	87.36	
4	76	87.36	
5	76	87.36	
6	76	87.36	
7	76	87.36	
8	76	87.36	
9	76	87.36	
10	76	87.36	
Average coverage	**76**	**87.36**	

Table 6-7 Result table of Genetic Algorithm
(Population size: 200, Generation: 20)

No.	Covered Test requirements (Maximum: 87)	Percentage coverage (%)	Stopping criterion
1	79	90.80	For each test requirement, GA stops when it finds the test case, which satisfies the test requirement, or after 20 generations if it still cannot satisfy the test requirement.
2	77	88.51	
3	79	90.80	
4	87	100	
5	79	90.80	
6	77	88.51	
7	79	90.80	
8	84	96.55	
9	84	96.55	
10	86	98.85	
Average coverage	**81.1**	**93.22**	

Table 6-8 Result table of Simulated Annealing
(Neighbor: 50)

No.	Covered Test requirements (Maximum: 87)	Percentage coverage (%)	Stopping criterion
1	81	93.10	For each requirement, SA stops when it finds the test case, which satisfies the test requirement, or stops after 80 temperature steps if it still cannot satisfy the test requirement.
2	81	93.10	
3	77	88.51	
4	77	88.51	
5	76	87.36	
6	81	93.10	
7	82	94.25	
8	84	96.55	
9	82	94.25	
10	76	87.36	
Average coverage	**79.8**	**91.72**	

Table 6-9 Result table of Genetic Simulated Annealing
(Population size: 150, temperature step:60, Generation: 200, Neighbor: 20)

No.	Covered Test requirements (Maximum: 87)	Percentage coverage (%)	Stopping criterion
1	81	93.10	For each requirement, GSA stops when it finds the test case, which satisfies the test requirement or stops after 200 generations if it still cannot satisfy the test requirement.
2	82	94.25	
3	77	88.51	
4	77	88.51	
5	75	86.21	
6	75	86.21	
7	82	94.25	
8	75	86.21	
9	75	86.21	
10	77	88.51	
Average coverage	**77.6**	**89.20**	

Table 6-10 Result table of Simulated Annealing with Advanced Adaptive Neighborhood
(Neighbor: 50)

No.	Covered Test requirements (Maximum: 87)	Percentage coverage (%)	Stopping criterion
1	76	87.36	For each requirement, SA/AAN stops when it finds the test case which satisfies the test requirement or stops after 80 temperature steps if it still cannot satisfy the test requirement.
2	72	82.76	
3	79	90.80	
4	77	88.51	
5	81	93.10	
6	79	90.80	
7	72	82.76	
8	79	90.80	
9	77	88.51	
10	79	90.80	
Average coverage	**77.1**	**88.62**	

The most challenging condition in this program is decision 10

else if (m == OCT && d < 15)

in the code above. In order to take the *true* branch of this condition, the test case should be (10, d, 1582), where d should be smaller than 15. In our experiments, only two runs of GA generate test cases that satisfy this condition; one of them also satisfies the other 86 test requirements successfully and obtains a complete coverage. The other one, fails to find the test case that exercises the *false* branch of **m == OCT,** so it only satisfies 86 test requirements.

6.2.3 Coverage plots for Five Test Data Generators

In order to make a further detailed comparison of different methods, the different test generator's performances on the Timeshuttle program are summarized graphically as a coverage plot. The number of test requirements satisfied is shown as the coverage percentage, which is plotted against the number of executions of the target program in the Figure 6-3.

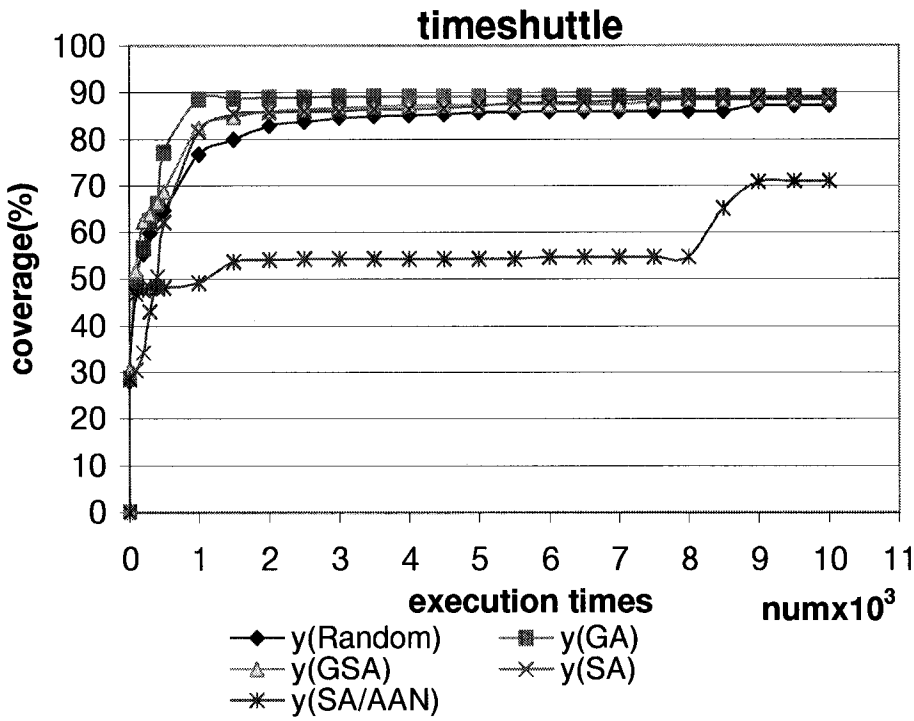

Figure 6-3 Coverage plots of five search methods on Timeshuttle program

As we see from the graph, GA still performs best on the Timeshuttle program since the very early stage. SA obtains almost the same highest coverage as GA, although it takes more effort to hit the peak. GSA also performs well on the Timeshuttle program in 10,000 times of target-program executions, covering about 88% of the code, which is slightly lower than GA and SA. But after that, as we can see from the result table 6-9, in 10 runs, some of runs are stuck in the local optimum and seldom have improvement, while 3 runs can keep on improving. Random generator hits its peak in about 4,500 times of target-program execution, but the search is fruitless afterwards. It is interesting that the plots show that SA/AAN performs so poorly on the Timeshuttle program, even worse than Random generator. The coverage is still below 60% after SA/AAN executes the program 8,000 times. To make a further analysis, several experiments are conducted. During these experiments, the neighborhood size and the temperature step are adjusted, and the new result is surprising. The following graph shows the comparison of performance of SA/AAN with different sets of parameters.

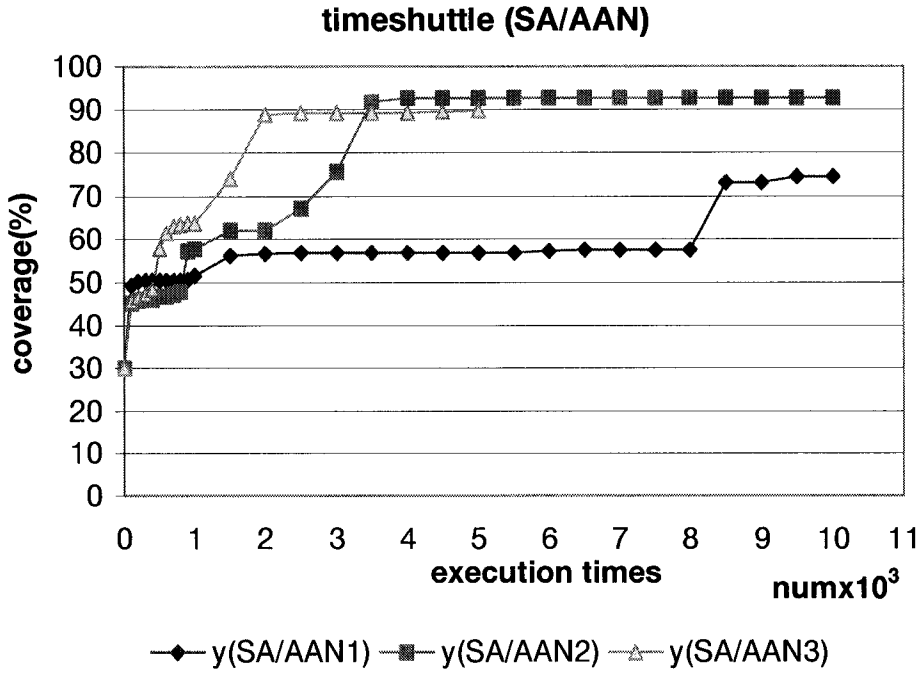

Figure 6-4 Coverage plots of SA/AAN with different parameters
SA/AAN1: neighborhood size: 50 temperature step: 80
SA/AAN2: neighborhood size: 20 temperature step: 20
SA/AAN3: neighborhood size: 10 temperature step: 20

For each set of parameters, the highest coverage is shown as the result table below.

Table 6-11 Result of SA/AAN with different parameters

Run	SA/AAN1 (%)	SA/AAN2 (%)	SA/AAN3 (%)
1	87.36	88.51	88.51
2	82.76	82.76	88.51
3	90.80	88.51	75.86
4	88.51	87.36	86.21
5	93.10	88.51	87.36
6	90.80	88.51	80.46
7	82.76	88.51	88.51
8	90.80	85.06	87.36
9	88.51	88.51	73.56
10	90.80	87.36	87.36
Average coverage	88.62	87.36	83.91

While with the larger neighborhood size and more temperature step, SA/AAN can obtain a higher coverage through its long search process. As we see from Figure 6-4,

with smaller neighborhood size and less temperature steps, SA/AAN can obtain some coverage very quickly. This seems hard to explain, but after we look through the working process of the SA/AAN generator, this can be explained easily. The working process of the test data generator is a working process that attempts to satisfy all of the test requirements. For each test requirements, SA/AAN will terminate either because it finds the successful test case or it has worked for enough temperature steps. Thus, when SA/AAN works on some test requirements that are hard to find the successful test case, SA/AAN with smaller neighborhood size and less temperature steps waste less effort on them. For example, consider the decision **if (m < OCT)** and **else if (m == OCT && d < 15),** as we discussed in the previous sections, they are most challenging test requirements in Timeshuttle program and they are intrinsically hard to satisfy. SA/AAN spends a lot of effort to attempt to cover these conditions before it gives up and starts to work for the next test requirement. Apparently, this can happen to any other test data generator discussed in this thesis, but it is more obvious in this situation, since SA/AAN can find the test case for some test requirements very quickly but for some other test requirements, it does not have the ability to handle them. Unfortunately, it seems an unavoidable problem for software testing program using this strategy.

6.2.4 Two methods that have best performance

As we see from the result tables and the coverage plots, GA and SA have the best performance on the Timeshuttle program. To make a further comparison of these two generators' performance, the maximum, the minimum and the average coverage of each method in 10 experiments are plotted against the execution times of the Timeshuttle program, shown as Figure 6-5.

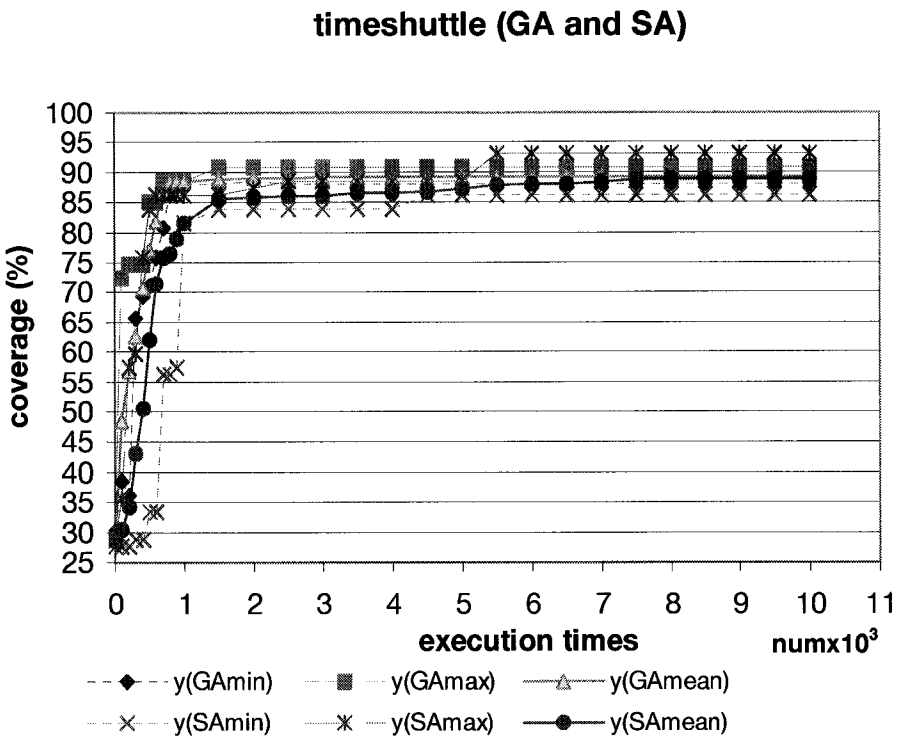

Figure 6-5 Comparison of GA and SA on Timeshuttle program

The graph shows that in 10,000 times of target-program execution, the performance of GA and SA are similar to each other. However, SA seems to have some runs where it performs very well and where it performs not so well. The performance of GA is more stable. Note that, this graph only shows the performance of these two generators in 10,000 target-program executions. We find out that both of these two generators achieve their highest coverage after that. For example, GA achieves 100% coverage after it executes the Timeshuttle program for 17,415 times.

6.3 Perfect number program

6.3.1 Analysis of the source code

To analyze the program, the decision points are shown below.

```
…… 
        if ((num % 7) == 0) //1
                …… 
        if ((num % 11) == 0) //2
                …… 
        if ((num % 13) == 0) //3
                …… 
```

```
        else

… …
        switch(p)
        {
        case 1: //4
                … …
        case 2: //5
                … …
        case 3: //6
                … …
        default: //7
        … …
        }

void myint::sum()
{
        … …

        if (check == 0) //8
                … …
        else
                … …
}


void myint::prime()
{

        for (i = 1; i < (0.5 * num); i++) //9
        {
                if (i != num && num != 1) //10
                {
                        if ((num % i) == 0) //11
                                … …
                }
                else
                        ;
        }

        if (p > 0) //12
                … …
        else
                … …
}
void myint::perfect()
```

```
{if (num % 2 == 0)  //13
        for (i =num/2; i >=2 ; i--)//14
        {
                if (num % i == 0) //15
                        …… ……
        }
        if (perfectsum == num) //16
                if num>100      //17
                if num>1000     //18
                …… ……
        else
                …… ……
}
int main()
{
        …… ……
}
```

There are 37 test requirements that should be satisfied to obtain complete condition-decision coverage of the perfect number program.

The objective functions are generated in the similar way as in Hex_dec and Timeshuttle program. For example, for the decision 18

if num>1000 //18

$$True: \Im(x)=\begin{cases} 0 & num>1000 \quad reached \\ 1000-num+1 & otherwise \quad reached \\ p & unreached \end{cases}$$

$$False: \Im(x)=\begin{cases} 0 & num\leq 1000 \quad reached \\ num-1000 & otherwise \quad reached \\ p & unreached \end{cases}$$

Where p is 2147483647. The instrumented code below is used to collect the information for calculating the objective function.

```
printf( "p" );
printf("%d",num);
printf("q" );
if num>1000     //18
```

6.3.2 A Comparison of Five Test-Data Generation Approaches

The tables from 6-12 to 6-16 show experimental results of 10 runs of each test data generators on perfect number program.

Table 6-12 Result table of Random Generator

No.	Covered test requirements (Maximum: 37)	Percentage coverage (%)	Stopping criterion
1	31	83.78	Random test data generator stops after 10,000 target-program execution.
2	31	83.78	
3	31	83.78	
4	31	83.78	
5	31	83.78	
6	31	83.78	
7	31	83.78	
8	31	83.78	
9	31	83.78	
10	31	83.78	
Average coverage	**31**	**83.78**	

Table 6-13 Result table of Genetic Algorithm
(Population size: 50 Generation: 30)

No.	Covered test requirements (Maximum: 37)	Percentage coverage (%)	Stopping criterion
1	37	100	For each test requirement, GA stops when it finds the test case, which satisfies the test requirement, or after 30 generations if it still cannot satisfy the test requirement.
2	37	100	
3	36	97.30	
4	37	100	
5	36	97.30	
6	35	94.60	
7	35	94.60	
8	36	97.30	
9	35	94.60	
10	36	97.30	
Average coverage	**36**	**97.30**	

Table 6-14 Result table of Simulated Annealing
(Neighbor: 50)

No.	Covered test requirements (Maximum: 37)	Percentage coverage (%)	Stopping criterion
1	36	97.30	For each requirement, SA stops when it finds the test case which satisfies the test requirement or stops after 30 temperature steps if it still cannot satisfy the test requirement.
2	36	97.30	
3	36	97.30	
4	36	97.30	
5	35	94.60	
6	36	97.30	
7	36	97.30	
8	36	97.30	
9	36	97.30	
10	36	97.30	
Average coverage	**35.9**	**97.03**	

Table 6-15 Result table of Genetic Simulated Annealing
(Population size: 50, temperature step: 20,Generation: 200 Neighbor:10)

No.	Covered test requirements (Maximum: 37)	Percentage coverage (%)	Stopping criterion
1	31	83.78	For each requirement, GSA stops when it finds the test case which satisfies the test requirement or stop after 200 generations if it still cannot satisfy the test requirement.
2	31	83.78	
3	31	83.78	
4	31	83.78	
5	35	94.60	
6	31	83.78	
7	31	83.78	
8	34	91.89	
9	32	86.49	
10	32	86.49	
Average coverage	**31.9**	**86.22**	

Table 6-16 Result table of Simulated Annealing with Advanced Adaptive Neighborhood
Neighbor: 50

No.	Covered test requirements (Maximum: 37)	Percentage coverage (%)	Stopping criterion
1	35	94.60	For each requirement, SA/AAN stops when it finds the test case which satisfies the test requirement or stops after 30 temperature steps if it still cannot satisfy the test requirement.
2	35	94.60	
3	35	94.60	
4	35	94.60	
5	35	94.60	
6	36	97.30	
7	35	94.60	
8	35	94.60	
9	35	94.60	
10	35	94.60	
Average coverage	**35.1**	**94.86**	

The results show that GA and SA have the best performance. GA achieves complete coverage in three runs.

The most challenging condition in Perfect number program is decision 18

if num>1000 //18

which requires that test case is a perfect number bigger than 1000. Although in ten runs experiments, the average coverage of SA and GA are the same, in three runs, GA obtains complete condition-decision coverage, while SA always fails to find the test case to cover this condition.

6.33 Coverage plots for Five Test Data Generators

The coverage plots for five test data generators are shown below.

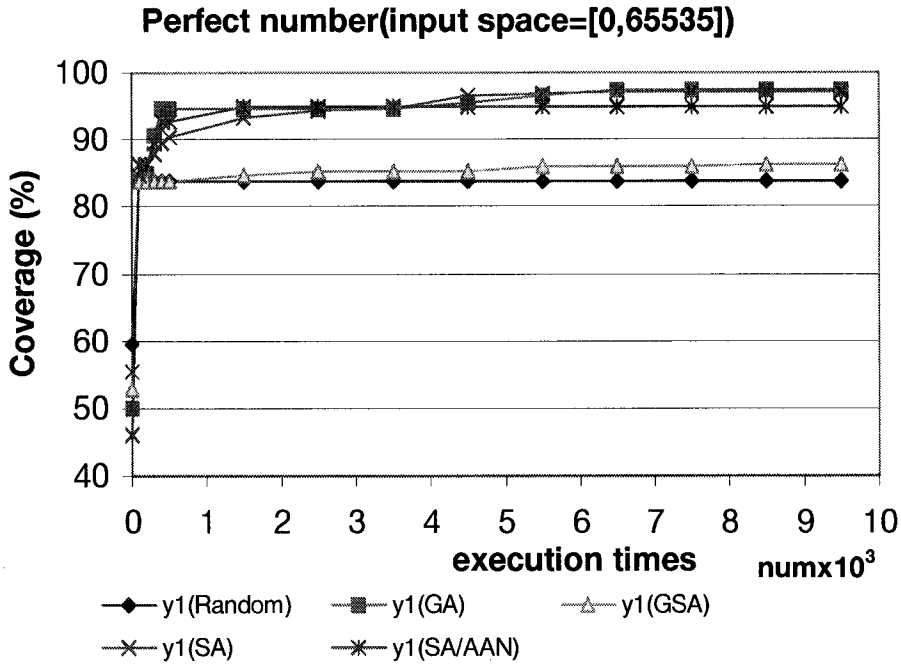

Figure 6-6 Coverage plots of five search methods on Perfect number program

The graph shows that each test data generator almost achieves their highest coverage in the very early stage. Random generator fails to improve after that, only covering about 83% of the code. The coverage of GSA is almost the same as Random test data generator in the early stage, however, it still improves slowly. In the end, GSA obtains higher coverage than Random generator. There is a big gap between the performance of these two generators and the other three. For GA, SA and SA/AAN, all of them achieve above 90% coverage in about 1,500 times of target-program execution, SA/AAN almost has no improvement after that, while GA and SA still improve slowly.

6.34 Two methods that have best performance

The graphs that show the maximum, the minimum and the average coverage of two test data generators in 10 experiments working on two different input spaces are provided to make a further comparison.

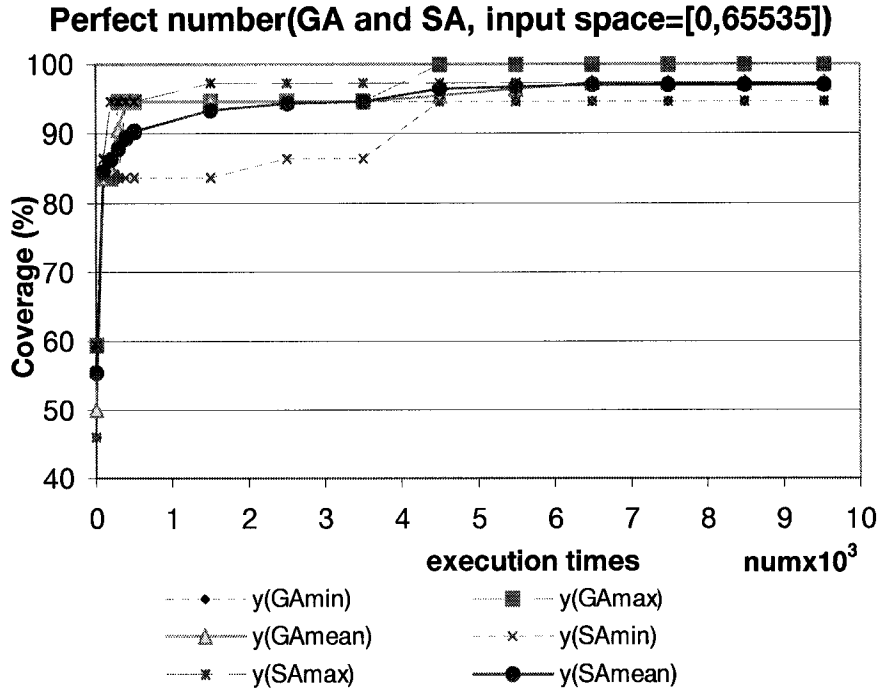

Figure 6-7 Comparison of GA and SA on Perfect number program with input space [0,65535]

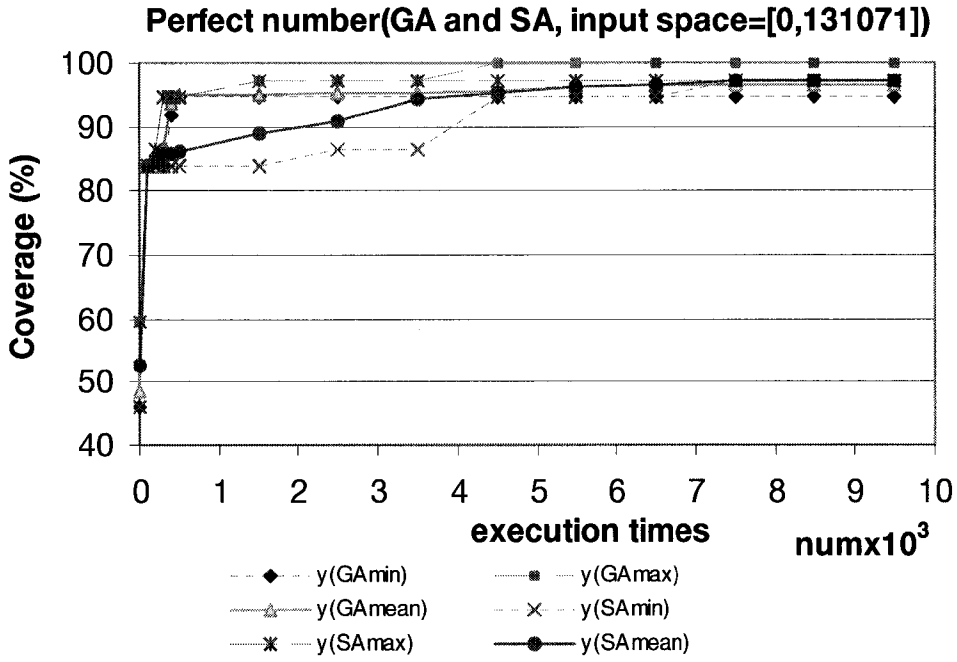

Figure 6-8 Comparison of GA and SA on Perfect number program with input space [0,131071]

The result tables for GA and SA working on the input space [0,131071] are shown as Table 6-17 and Table 6-18.

Table 6-17 Result table of Genetic Algorithm
(Population size: 50 Generation: 30)

No.	Covered test requirements (Maximum: 37)	Percentage coverage (%)	Stopping criterion
1	36	97.30	For each test requirement, GA stops when it finds the test case, which satisfies the test requirement, or after 30 generations if it still cannot satisfy the test requirement.
2	35	94.59	
3	37	100	
4	35	94.59	
5	36	97.30	
6	35	94.59	
7	36	97.30	
8	35	94.59	
9	35	94.59	
10	37	100	
Average coverage	**35.7**	**96.49**	

Table 6-18 Result table of Simulated Annealing
(Neighbor: 50)

No.	Covered test requirements (Maximum: 37)	Percentage coverage (%)	Stopping criterion
1	36	97.30	For each requirement, SA stops when it finds the test case which satisfies the test requirement or stops after 30 temperature steps if it still cannot satisfy the test requirement.
2	36	97.30	
3	36	97.30	
4	36	97.30	
5	36	97.30	
6	36	97.30	
7	36	97.30	
8	36	97.30	
9	36	97.30	
10	36	97.30	
Average coverage	**36**	**97.30**	

The graph shows that after 4,000 times of execution of the target program, we still can tell the maximum coverage curve of GA and the minimum coverage curve of SA clearly, while the other four curves are so close to each other that we can hardly tell the difference. Combining the result tables, we find out that for the input space

[0,65535], GA performs much better in two runs than in other runs. On the other hand, SA performs much worse in one single run than others. Although the average coverage of GA and SA are almost the same, GA has the ability to achieve complete coverage on both input spaces.

The Figure 6-7 and Figure 6-8 are so similar to each other, which suggests that for the perfect number program, the difference between these two input spaces hardly has influence on the result. Note that as shown in the result tables of GA and SA for the input space, GA still achieves complete coverage in two runs.

6.4 Triangle classification program

6.41 Analysis of the source code

The code for Triangle classification is shown in the appendices, and the decision points are shown below:

```
int triangle (int i, int j, int k){
    ... ...
    if ((i<=0) ||(j<=0) ||(k<=0))    //1
    ... ...
      if (i==j)                      //2
            ... ...
      if (i==k)                      //3
    ... ...
      if (j==k)                      //4
    ... ...
      if (tri==0)                    //5
        {
         if ((i+j<=k)||(i+k<=j)||(j+k<=i))  //6
                    ... ...
              else
      ... ...
            }
      if  (tri>3)                          //7
      ... ...
      else
            if ((tri==1) &&(i+j>k))        //8
            ... ...
            else if ((tri==2) &&(i+k>j))   //9
              ... ...
            else if ((tri==3) &&(j+k>i)) //10
            ... ...
             else
```

```
                ... ...
}
int main(){int a,b,c,t;
        ... ...
if (t==1)                                   //11
{
 ... ...
 }
else if (t==2)                              //12
  {
   ... ...
}
else if (t==3){                             //13
 ... ...
}
else if (t==4){                             //14
 ... ...}
... ...
}
```

To obtain complete condition-decision coverage, there are 51 test requirements to be satisfied.

Consider the following code fragment:

else if ((tri==2) &&(i+k>j)) //9

In order to collect the information to calculate the objective function, the additional code is instrumented in the following way:

```
printf("a");
printf("%d", tri);
printf("%d", i);
printf("b");
printf("%d", j);
printf("c");
printf("%d", k);
printf("d");
else if ((tri==2) &&(i+k>j)) //9
Printf("b");
```

To ensure the condition **tri==2** take value "*true*", the following function is built:

$$True\ \Im 1 = \begin{cases} p & unreached \\ p \times m & reached\ tri \neq 2 \\ 0 & reached\ tri == 2 \end{cases}$$

Execution of the instrumented code provides information about values of **tri, i, j** and **k** allowing for calculating the value of $\Im(x, y, z)$. Where p is a significant value and it

is the maximum range of input space 2147483647 in the triangle program, m is a value between 0 and 1, which is 0.45 in our experiments.

To ensure the condition **tri==2** take value "*false*", the following function is built:

tri==2

$$\Im 1=\begin{cases} 0 & tri \neq 2 \quad reached \\ p \times m & tri = 2 \quad reached \\ p & unreached \end{cases}$$

In the similar way, functions representing all of the other 4 cases are built:

(i+k>j)

$$True\ \Im 2=\begin{cases} 0 & i+k > j \quad reached \\ 1+abs(i+k-j)/3 & i+k \leq j \quad reached \\ p & unreached \end{cases}$$

$$False\ \Im 2=\begin{cases} 0 & i+k \leq j \quad reached \\ abs(i+k-j)/3 & i+k > j \quad reached \\ p & unreached \end{cases}$$

if(tri==2)&& (i+k>j)

$$True\ \Im=\begin{cases} p & unreached \\ \Im 1+\Im 2 & reached \end{cases}$$

$$False\ \Im=\begin{cases} p & unreached \\ \min imum(\Im 1, \Im 2) & reached \end{cases}$$

6.42 A Comparison of Five Test-Data Generation Approaches

The tables below show the result of each test data generator working on the triangle classification program.

Table 6-19 Result table of Random Generator
Input space 1 is [-65536,65535]; input space 2 is [-2147483648,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 51)	Covered Test requirements Input space2 (Maximum: 51)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	24	24	47.06	47.06	Random test data generator stops after 30,000 target-program execution.
2	24	24	47.06	47.06	
3	24	24	47.06	47.06	
4	24	24	47.06	47.06	
5	24	24	47.06	47.06	
6	24	24	47.06	47.06	
7	24	24	47.06	47.06	
8	24	24	47.06	47.06	
9	24	24	47.06	47.06	
10	34	24	66.67	47.06	
Average coverage	**25**	**24**	**49.02**	**47.06**	

Table 6-20 Result table of Genetic Algorithm
(Population size: 100 Generation: 50)
Input space 1 is [-65536,65535]; input space 2 is [-2147483648,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 51)	Covered Test requirements Input space2 (Maximum: 51)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	47	47	92.16	92.16	For each test requirement, GA stops when it finds the test case that satisfies the test requirement, or after 20 generations if it still cannot satisfy the test requirement.
2	49	49	96.08	96.08	
3	48	47	94.12	92.16	
4	49	49	96.08	96.08	
5	47	47	92.16	92.16	
6	47	47	92.16	92.16	
7	49	49	96.08	96.08	
8	47	47	92.16	92.16	
9	47	47	92.16	92.16	
10	49	47	96.08	92.16	
Average coverage	**47.9**	**47.6**	**93.92**	**93.33**	

Table 6-21 Result table of Simulated Annealing
(Neighborhood size: 100 Temperature step: 50)
Input space 1 is [-65536,65535]; input space 2 is [-2147483648,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 51)	Covered Test requirements Input space2 (Maximum: 51)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	33	24	64.71	47.06	For each requirement, SA stops when it finds the test case, which satisfies the test requirement or stops after 50 temperature steps if it still cannot satisfy the test requirement.
2	45	23	88.24	45.10	
3	35	24	68.63	47.06	
4	23	24	45.10	47.06	
5	36	24	70.59	47.06	
6	24	24	47.06	47.06	
7	42	24	82.35	47.06	
8	37	24	72.55	47.06	
9	40	24	78.43	47.06	
10	23	23	45.10	45.10	
Average coverage	**33.8**	**23.8**	**66.27**	**46.67**	

Table 6-22 Result table of Genetic Simulated Annealing
(Population size: 100, Generation: 30, Neighbor: 20, Temperature step: 10)
Input space 1 is [-65536,65535]; input space 2 is [-2147483648,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 51)	Covered Test requirements Input space2 (Maximum: 51)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	43	21	84.31	41.18	For each requirement, GSA stops when it finds the test case which satisfies the test requirement or stops after 30 generations if it still cannot satisfy the test requirement.
2	43	21	84.31	41.18	
3	39	21	76.47	41.18	
4	40	21	78.43	41.18	
5	39	21	76.47	41.18	
6	37	21	72.55	41.18	
7	44	21	86.27	41.18	
8	44	21	86.27	41.18	
9	42	21	82.35	41.18	
10	43	21	84.31	41.18	
Average coverage	**41.4**	**21**	**81.18**	**41.18**	

Table 6-23 Result table of Simulated Annealing with Advanced Adaptive Neighborhood
Neighbor: 100
Input space 1 is [-65536,65535]; input space 2 is [-2147483648,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 51)	Covered Test requirements Input space2 (Maximum: 51)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	44	44	86.27	86.27	For each requirement, SA/AAN stops when it finds the test case which satisfies the test requirement or stops after 30 generations if it still cannot satisfy the test requirement.
2	41	46	80.39	90.20	
3	44	44	86.27	86.27	
4	46	37	90.20	72.55	
5	46	44	90.20	86.27	
6	44	44	86.27	86.27	
7	44	46	86.27	90.20	
8	44	45	86.27	88.24	
9	44	44	86.27	86.27	
10	46	41	90.20	80.39	
Average coverage	**44.3**	**43.5**	**86.86**	**85.29**	

From the result tables above, we can see none of the test data generators achieve a complete coverage even on the small input space. GA, which performs best, still fails to cover the *true* branch of condition/decision **if (tri>3)** and **else if (t==3),** which require that 3 equal integers as input. It is interesting that in the best runs, SA/AAN only satisfies 46 test requirements. After investigating the uncovered conditions, we find out this is because SA/AAN finds the test cases that satisfy **i==j** but **i+j<k.** This causes that the *true* branch of the decision 8 **if tri==1&&i+j>k** cannot be executed. For example, the test case (43615,135876,43615) cannot exercise the *true* branch of decision 8. This also happen to decision 9

if tri==2&& i+k>j

and decision 10

else if tri==3&&j+k>i.

On the other hand, this situation does not happen to GA. Through analyzing the result, we find out that even if GA generate such a test case, it always has the ability to generate another test case to make **if tri==1&&i+j>k** to take the *true* outcome. For example, we find a test case (7271,7271,22811) in one run, and we also find another

test case (7271,7271,9956). This is because in our experiments, test data generators keep the test data that can reach the condition (see Chapter 5), and when the test data generator starts to work on satisfying this condition, the initial population is seeded with these candidate test cases. GA is population based so it has the ability to keep these good seeds or the good gene, and in this case, it is (7271,7271,z).

On the other hand, SA/AAN starts with the single candidate seed. So this candidate seed should be selected from the test cases, which can reach the condition **if tri==1&&i+j>k,** and it has little chance to keep the good seed for subsequent neighborhood searches.

For example, there are 3 candidate test cases, which are (1783,567,567), (4357,4357,10896), (1199,90234,90234). All of them can reach the condition **if tri==1&&i+j>k,** however only (4357,4357,10896) has the good gene which may cause the subsequent test cases to cover the condition **if tri==1&&i+j>k**. SA/AAN selects a seed randomly from these three test cases, so there is only a 33% chance that the valuable test case (4357,4357,10896) will be selected. Moreover, even if SA/AAN selects the good seed, in subsequent neighborhood searches, SA/AAN has little chance to keep this good gene due to its neighborhood structure design.

6.43 Coverage plots for Five Test Data Generators

The coverage plots comparing the performance of five test data generators on the triangle program are shown as Figure 6-9 and Figure 6-10. In both of the two input spaces, GA has the best performance overall.

For the small input space [-65536,65535], the Random generator hit the peak early, but after that, it has little improvement. On the other hand, SA and GSA improve slowly and eventually outperform Random test data generator. GA covers about 90% of the code in 4,000 target program executions. SA/AAN only covers about 82% of the code.

For the input space [-2147483648, 2147483647], GA covers about 93% of the code in about 9,500 target program executions. SA/AAN performs nearly as well as GA, covering about 85% code. On the other hand, the other three generators perform poorly on the triangle program. Random generator hits its peak in the early stage of search process, but the search is fruitless after that. GSA has the similar situation but

the performance is worse, only covering about 25% of the code. SA hits its peak later than Random generator but still has no improvement after it executes the target program 1,500 times. It is interesting that Random generator outperforms both SA and GSA on the large input space. The reason why SA fails is that in this huge search space, if the seed inputs are large negative integers, with the small neighborhood range, SA will needs to spend too many program-executions to get the positive test input, which is required by most of test requirements. On the other hand, with the large neighborhood range, that is similar to Random generator, it is hard to find the test input that can satisfy the test requirement like "**if (i==k)**". SA may find the test input eventually but it needs to execute the program for a very long time, which is too expensive. Since our experiments stop at about 30,000 times of target-program execution, we cannot compare the performance of these generators after that.

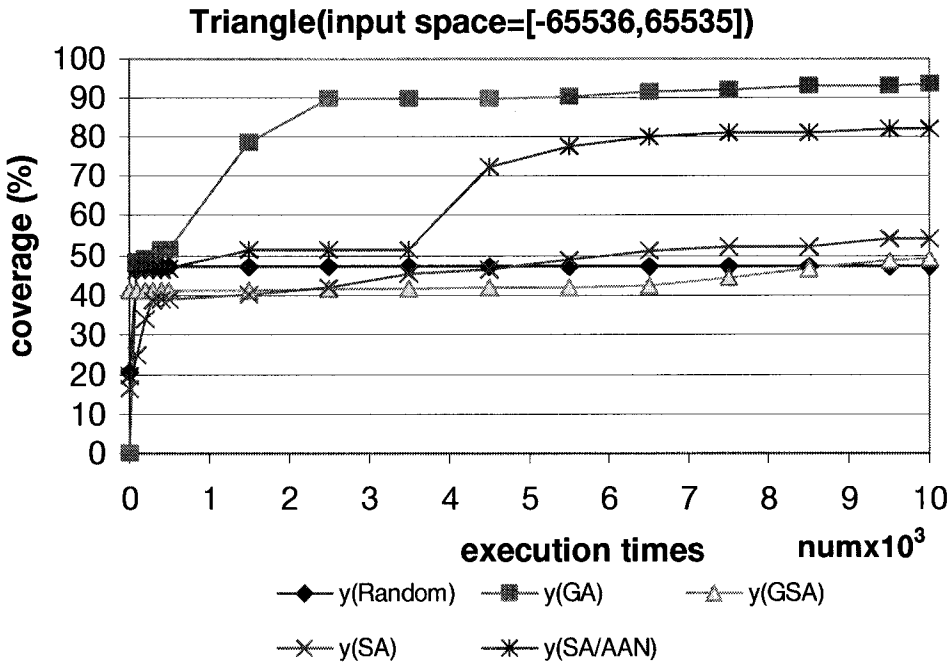

Figure 6-9 Coverage plots of five search methods on Triangle classification program with input space [-65536, 65535]

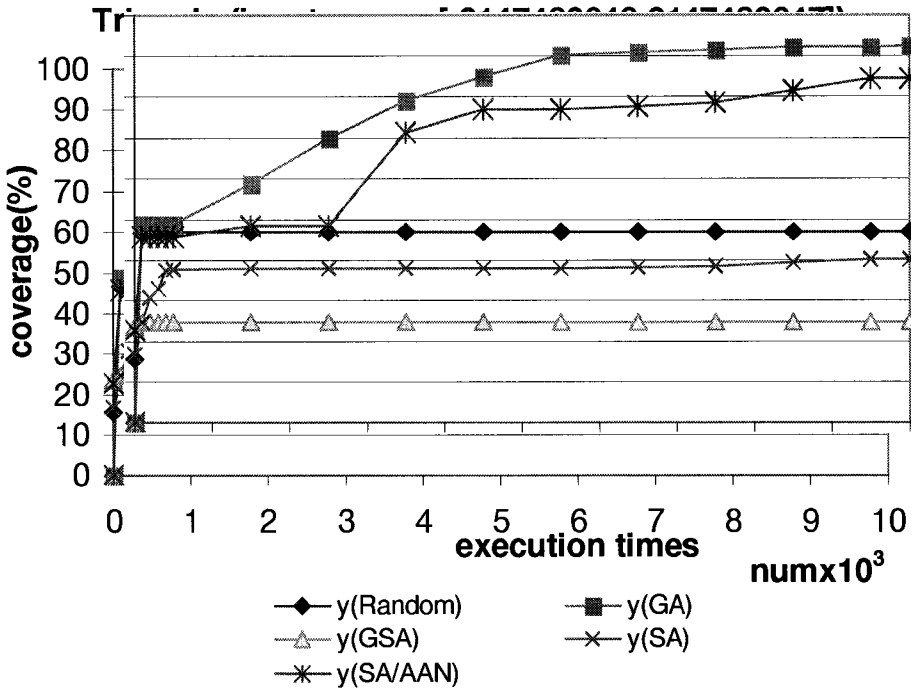

Figure 6-10 Coverage plots of five search methods on Triangle classification program with input space [-2147483648, 2147483647]

Comparing the performance of five test data generators on two input spaces, we can see GA, SA and SA/AAN perform better on the small input space than on the large input space. For both of the two input spaces, Random data generator has the same performance. SA/AAN does not perform better working on the small input space. This may be because that we use the same parameter set on both of the two input spaces.

6.44 Two methods that have best performance

For the triangle program, GA and SA/AAN perform much better than the other three test generators. To make a detailed comparison of two generators, for the two input spaces, the maximum, the minimum and the average coverage of each generator in 10 experiments are plotted against the number of the execution of the triangle program, shown in Figure 6-11 and Figure 6-12.

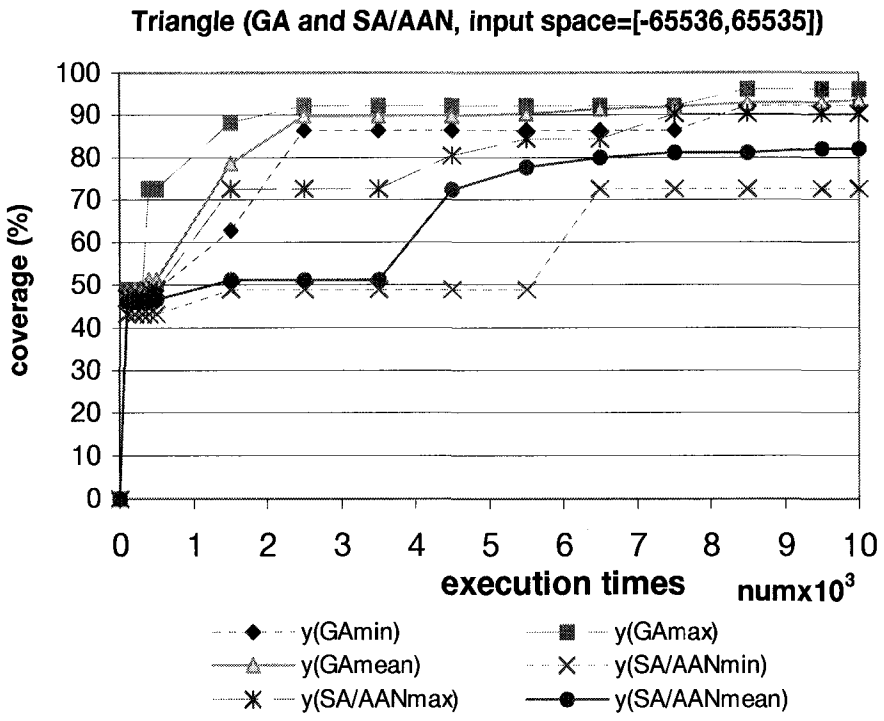

Figure 6-11 Comparison of GA and SA on Triangle classification program with input space [-65536,65535]

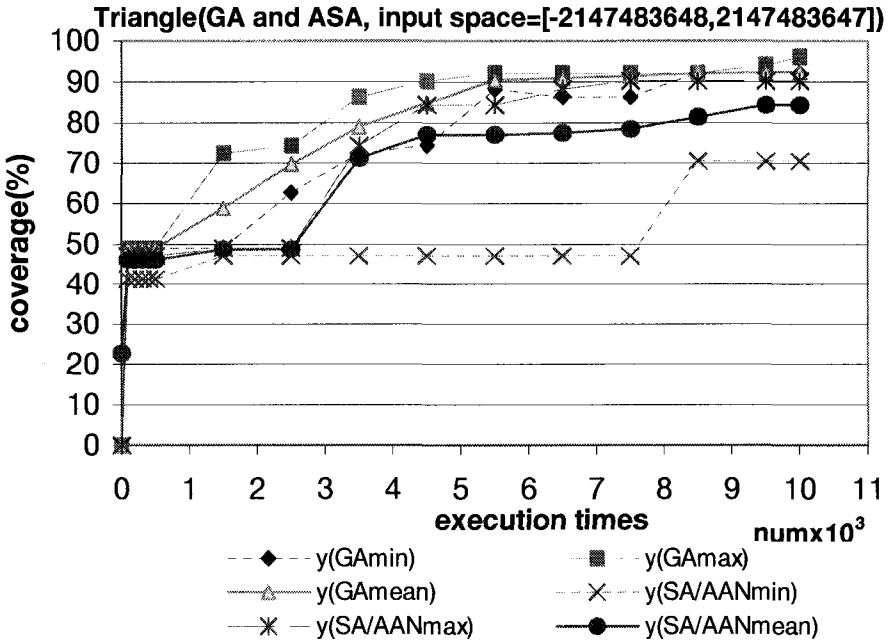

Figure 6-12 Comparison of GA and SA on Triangle classification program with input space [-2147483648,2147483647]

The graphs above show the difference of two test data generation systems on the triangle program. There is not much difference between the minimum coverage and the maximum coverage of GA, but the difference between the minimum coverage and the maximum coverage of SA/AAN is quite obvious.

Combining with the result table, for the input space [-2147483648,2147483547], we can see the coverage of one run of SA/AAN is only 72.55%. This suggests that for the triangle program, although the highest achieved coverage is the same for both SA/AAN and GA, the overall performance of SA/AAN is not so stable as GA.

6.5 Rescue program

6.5.1 Analysis of the source code

The source code of the Rescue program is shown in the Appendices. The decision points are shown below.

```
... ...
if(!(code > 9999 && code < 100000))  // 1
{
        ... ...
}
else
{... ...

... ...
        if (!(sum%2 == 0))  //2
        {
        ... ...
        }
        else
        {
                ... ...
                if(rescueDay < 1 || rescueDay > 7)  //3
                {
                        ... ...
                }
                else
                {
                        if(digit4 == digit5)  //4
                        {
                                ... ...
                        }
                        else if(digit4 > digit5)  //5
```

```
{
        … …
}
else
{
        … …
}

if((rendezvousPt != 2) && (rendezvousPt != 7) &&
(rendezvousPt != 8)) //6
{
        … …
}
else
{
        … …
        switch(rescueDay)
        {
                case 1: //7
                        … …

                        case 2: //8
                        … …
                case 3: //9
                        … …
                case 4: //10
                        … …
                case 5: //11
                        … …
                case 6: //12
                        … …
                case 7: //13
                        … …
                default:
                        … …
        } // end of switch


        switch(rendezvousPt)
        {
                case 2: //14
                        … …
                case 7: //15
                        … …
                case 8: //16
                        … …
```

```
                                        default:
                                                ... ...
                                        } // end of switch
... ...
                                }
                        }
                }
        }
        return 0;
}
```

There are 16 decision branches (which are identified in bold) in **Rescue**. According to the definition of condition-decision coverage, the test generators need to generate the test set which satisfy 46 test requirements to obtain a complete coverage. Recall from the function minimization technique discussed in the previous Section 5.1.3, each test requirement is reduced to a function minimization problem. The program is instrumented with additional code that reports the information needed to the test data generator to calculate the value of the objective function $\Im(x)$. For example, the first branch in Rescue is

if(!(code > 9999 && code < 100000))

There are two conditions in it, **code > 9999** and **code < 100000.** To obtain the complete condition-decision coverage, test data must make each condition take the *true* and *false* value, and exercise both the *true* and *false* branches of the decision. Thus, 6 test requirements need to be satisfied to obtain complete condition-decision coverage. These 6 test requirements and their corresponding objective functions are shown below.

To ensure the decision

if(!(code > 9999 && code < 100000))

takes value "*true*", the following function is built:

$$True:\ \Im(x)=\begin{cases} 0 & code\geq 100000 \quad reached \\ \min imum(code-9999,100000-code) & code\leq 9999 \quad reached \\ p & 9999<code<100000 \quad unreached \end{cases}$$

Where p is a significant value 2147483647.

The instrumented code is shown as italic characters below:

printf("a");

printf("%d",code);

printf("b");

if(!(code > 9999 && code < 100000))

printf("c");

Execution of the instrumented code provides information about the value of **code.** This information allows us to calculate the value of $\Im(x)$.

To ensure the decision

if(!(code > 9999 && code < 100000))

takes value "*false*", the following function is built:

$$False:\ \Im(x)=\begin{cases} 0 & 9999<code<100000 \\ 9999-code+1 & code\le 9999 \\ code-100000+1 & code\ge 100000 \\ p & unreached \end{cases}$$

In the similar way, functions representing all of the other 5 cases are built:

if(code > 9999)

$$True:\ \Im(x)=\begin{cases} 0 & code>9999 & reached \\ 9999-code+1 & code\le 9999 & reached \\ p & & unreached \end{cases}$$

$$False:\ \Im(x)=\begin{cases} 0 & code\le 9999 & reached \\ code-9999 & code>9999 & reached \\ p & & unreached \end{cases}$$

if (code < 100000)

$$True:\ \Im(x)=\begin{cases} 0 & code<100000 & reached \\ code-100000+1 & code\ge 100000 & reached \\ p & & unreached \end{cases}$$

$$False:\ \Im(x)=\begin{cases} 0 & code\ge 100000 & reached \\ 100000-code & code<100000 & reached \\ p & & unreached \end{cases}$$

For the fourth branch **if(digit4 == digit5),** the way to construct the objective function is slightly different from the first branch. Two test requirements must be satisfied; the objective functions are shown below.

$$True: \Im(x) = \begin{cases} 0 & digit4 = digit5 \\ abs(digit4 - digit5) + p' & digit4 \neq digit5 \\ p & unreached \end{cases}$$

$$False: \Im(x) = \begin{cases} p' & digit4 = digit5 \\ 0 & digit4 \neq digit5 \\ p & unreached \end{cases}$$

p and p' are two significant values, where $p'<<p$. In the experiments, p= 2147483647 and p' =0.7p.

In the experiment on the Rescue program, 46 objective functions are generated in the similar way discussed above.

6.5.2 A Comparison of Five Test Data Generation Approaches

Ten complete test-generation runs with each test generator were performed for the Rescue program. Experiments were conducted on two different input spaces. The following tables 6-24—6-28 show the test results of running GA, GSA, SA, SA/AAN and Random test generator on the rescue program in the input space of [0,2147483647]. For each test generator, the condition-decision coverage of each run, the average coverage in 10 runs and the stopping criterion are showed in each result table.

Table 6-24 Result table of Random Generator
Input space 1 is [0,524287]; input space 2 is [0,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 46)	Covered Test requirements Input space2 (Maximum: 46)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	43	3	93.48	6.52	Random test data generator stops after 40,000 target-program execution.
2	43	9	93.48	19.57	
3	43	3	93.48	6.52	
4	43	6	93.48	13.04	
5	43	16	93.48	34.78	
6	43	8	93.48	17.39	
7	43	9	93.48	19.57	
8	43	8	93.48	17.39	
9	43	3	93.48	6.52	
10	43	10	93.48	21.74	
Average coverage	**43**	**7.5**	**93.48**	**16.30**	

Table 6- 25 Result table of Genetic Algorithm
(Population size: 100, Generation: 30)
Input space 1 is [0,524287]; input space 2 is [0,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 46)	Covered Test requirements Input space2 (Maximum: 46)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	46	46	100	100	For each test requirement, GA stops when it finds the test case, which satisfies the test requirement, or after 30 generations if it still cannot satisfy the test requirement.
2	46	46	100	100	
3	45	46	97.83	100	
4	46	45	100	97.83	
5	46	45	100	97.83	
6	46	46	100	100	
7	46	45	100	97.83	
8	46	46	100	100	
9	46	45	100	97.83	
10	45	46	97.83	100	
Average coverage	**45.8**	**45.6**	**99.57**	**99.13**	

Table 6- 26 Result table of Simulated Annealing
(Neighbor: 200)
Input space 1 is [0,524287]; input space 2 is [0,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 46)	Covered Test requirements Input space2 (Maximum: 46)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	45	3	97.83	6.52	For each requirement, SA stops when it finds the test case, which satisfies the test requirement, or stops after 20 temperature steps if it still cannot satisfy the test requirement..
2	45	3	97.83	6.52	
3	45	3	97.83	6.52	
4	45	3	97.83	6.52	
5	45	3	97.83	6.52	
6	45	3	97.83	6.52	
7	45	3	97.83	6.52	
8	45	3	97.83	6.52	
9	45	3	97.83	6.52	
10	45	3	97.83	6.52	
Average coverage	**45**	**3**	**97.83**	**6.52**	

Table 6- 27 Result table of Genetic Simulated Annealing
(Population size: 50, temperature: 20, Generation: 800, Neighbor: 10
Input space 1 is [0, 524287]; input space 2 is [0,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 46)	Covered Test requirements Input space2 (Maximum: 46)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	45	40	97.83	86.96	For each requirement, GSA stops when it finds the test case which satisfies the test requirement or stops after 800 generations if it still cannot satisfy the test requirement
2	46	3	100	6.52	
3	46	10	100	21.74	
4	46	3	100	6.52	
5	46	3	100	6.52	
6	46	42	100	91.30	
7	46	5	100	10.87	
8	46	43	100	93.48	
9	46	32	100	69.57	
10	46	9	100	19.57	
Average coverage	**45.9**	**19.0**	**99.78**	**41.30**	

Table 6- 28 Result table of Simulated Annealing with Advanced Adaptive Neighborhood
(Neighbor: 100)
Input space 1 is [0,524287]; input space 2 is [0,2147483647]

No.	Covered Test requirements Input space1 (Maximum: 46)	Covered Test requirements Input space2 (Maximum: 46)	Percentage coverage (%) Input space1	Percentage coverage (%) Input space2	Stopping criterion
1	46	46	100	100	For each requirement, SA/AAN stops when it finds the test case which satisfies the test requirement or stops after 20 temperature steps if it still cannot satisfy the test requirement
2	46	45	100	97.83	
3	46	46	100	100	
4	46	46	100	100	
5	46	46	100	100	
6	46	46	100	100	
7	46	46	100	100	
8	46	46	100	100	
9	46	46	100	100	
10	46	46	100	100	
Average coverage	**46**	**45.9**	**100**	**99.78**	

The result tables above show the different performances of different test data generations. GA and SA/AAN exhibit the best performances, with the average coverage close to 100% coverage, while the average performances of SA, GSA and Random test generation are below 50%.

In the Rescue program, the most important decision is decision 1

if(!(code > 9999 && code < 100000)) // 1

If the test case cannot take the false branch of this decision, it will fail to reach the rest of the conditions and decisions. That is the reason why SA, Random test data generator and GSA have poor performance on Rescue program.

In the Rescue program, the most challenging conditions are decision 7 to decision 13, which are deeply nested inside four if-else statements. In order to cover these conditions, the test data should satisfy other four else statements first.

6.5.3 Coverage plots for Five Test Data Generators

To make a further detailed comparison of different methods, the different test generator's performances are plotted as coverage plots. The numbers of test

requirements satisfied are shown as the coverage percentage, which are plotted against the number of executions of the target program in the graphs below.

From the graphs, we can see GA and SA/AAN have the best performance.

Figure 6-13 summarizes the results of five test data generators working on the input space of [0, 524287], all of the test data generators perform very well on this small input space. After executing the Rescue program 2,500 times, GA, SA/AAN and GSA have already obtained almost 100% coverage. In the early stages of the search process, SA performs worst, but after that, it has a big jump and outperforms Random generator. Random generator performs worse than the other four test data generators, and its coverage is about 94%.

For the input space [0,2147483647], though both GA and SA/AAN achieve almost complete coverage after they execute the Rescue program 7000 times, there is a slight difference between two test generators. The coverage plot in Figure 6-14 shows this difference in the experiments. To obtain 90% coverage, GA only needs to execute the program 2500 times, while SA/AAN needs to execute the program 3500 times. Note after SA/AAN obtains 90% coverage, it improves very fast and hits its peak faster than GA. Generally, SA/AAN needs to execute the program less times to obtain complete coverage of the Rescue program than GA. There is a significant difference between the performance of these two test data generators and the other three. For the other three test data generators, their performances improve slowly throughout the experiments. After executing the Rescue program 9500 times, the coverage of SA, GSA and Random generator are still below 20%. GSA performs better than Random generator and SA.

For the Rescue program, the small input space means a small search space, which allows the test data generation to find the test set satisfying the test requirements more easily as compared to a large input space. Therefore, all of five test data generators perform better when they work on a small input space than they work on a large input space. Figure 6-13 shows that when these five test data generators work on the small input space, GA, SA/AAN and GSA achieve nearly 100% in about 2,500 target-program executions. SA also achieves about 97% coverage, while Random test generator achieves about 90% coverage. Comparing the Figure 6-13 and Figure 6-14,

we can see those test generations which do not perform well on a large input space have great improvement when they work on a small input space.

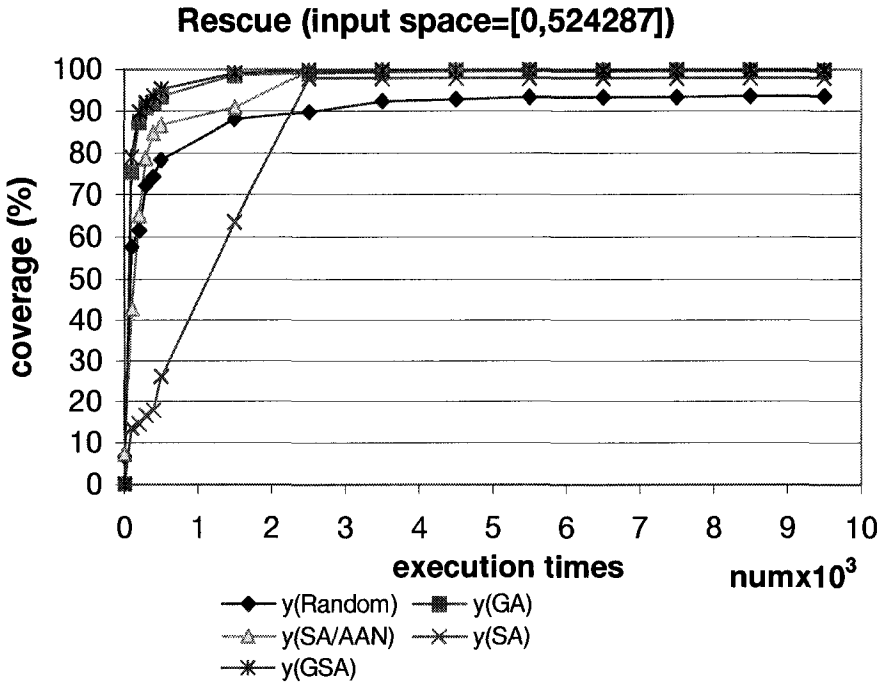

Figure 6-13 Coverage plots of five search methods on theRescue program with input space [0, 524287]

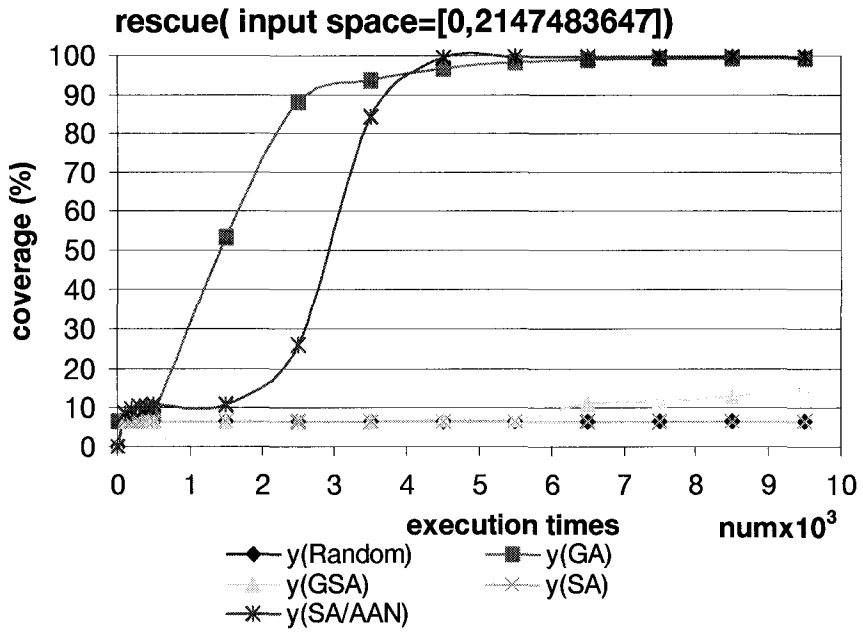

Figure 6-14 Coverage plots of five search methods on the Rescue program with input space [0, 2147483647]

6.5.4 GA and SA/AAN: Two methods that have best performance

This section concentrates on discussing two test data generators that have the best performance on the Rescue program. For SA/AAN and GA test data generators, the maximum, minimum and average coverage in 10 experiments are plotted against the number of the execution of the Rescue program.

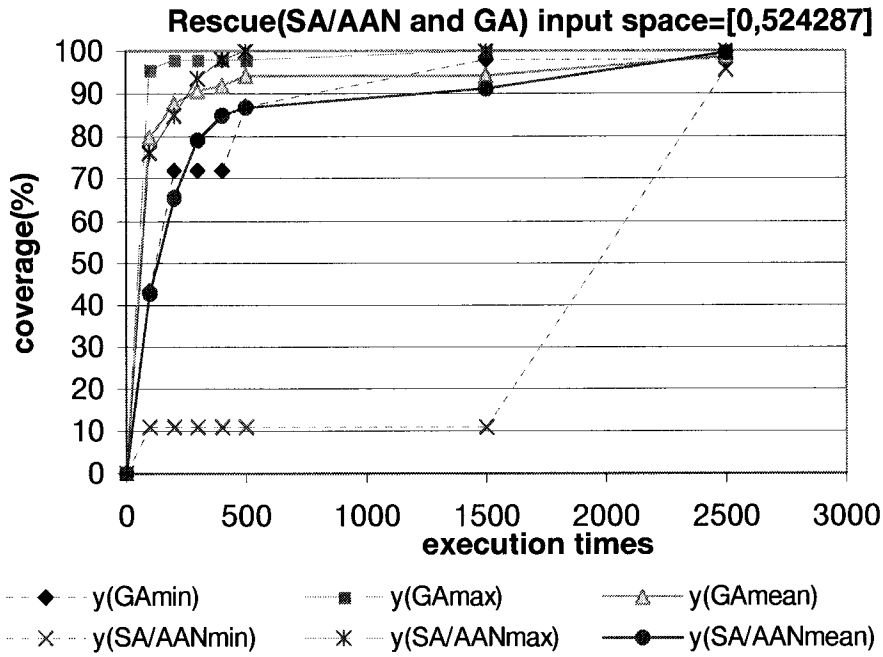

Figure 6-15 Comparison of GA and SA on Rescue program with input space [0,524287]

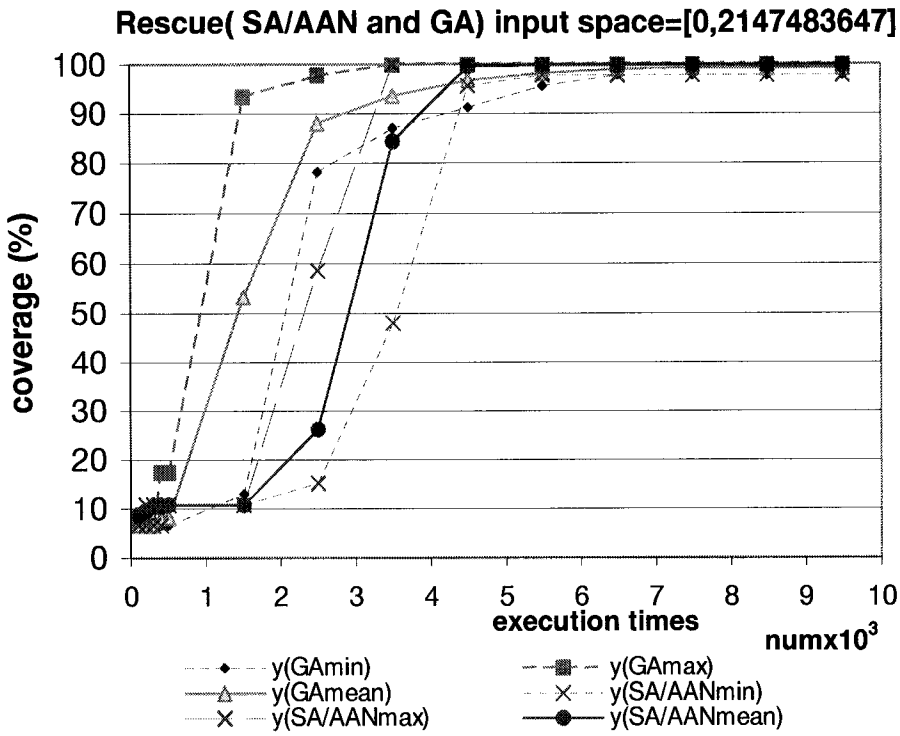

Figure 6-16 Comparison of GA and SA on Rescue program with input space [0,2147483647]

Figure 6-15 is a coverage plot comparing the performance of SA/AAN and GA on the Rescue program when the input space is [0, 524287]. When SA/AAN and GA work on this small input space, both of these two test data generators hit their peak in about 2,500 target-program executions. Actually, in 10 runs, SA/AAN obtains the complete coverage in 1,000 target program executions. While in one single run, SA/AAN performs poorly. Figure 6-15 shows the difference of these two test data generators. Although the maximum coverage curves of these two test data generators are close to each other, generally, GA still performs much better than SA/AAN in the first 500 executions.

Figure 6-16 shows the difference in performances of the two methods on the Rescue program when the input space is [0,2147483647]. In the first 3,000 target-program executions, GA performs much better than SA/AAN, even if the maximum coverage of SA/AAN is still lower than the minimum coverage of GA. After that, the curves of two methods start to close to each other. SA/AAN hits its peak in about 4,500 target-program executions, while GA still improves slowly. As shown in the

result table, SA/AAN achieves complete coverage in 9 runs while GA only achieves complete coverage in 5 runs.

Comparing the performance of these two teat data generators on the Rescue program, SA/AAN improves slowly in the very early stage of test generation, while GA improves its coverage stably and relative slowly throughout the whole search process.

The performance of SA/AAN on the Rescue program may be explained by the working theory of the SA/AAN and the feature of the Rescue program. Through analyzing the Rescue program, we can see that all of the test cases needed to satisfy all of the test requirements exist in a small range [10000, 99999]. In the early stage of search process, SA/AAN works on the first conditional branch **if(!(code > 9999 && code < 100000)),** and the objective function guides the search to the desired small range [10000, 99999]. After SA/AAN finds the test case in this small range, the false branch of this conditional branch is satisfied, and the next test requirement is reached, thus this test case is stored for future use. When SA/AAN works on the next test requirement, this test case is used as a seed. As the neighborhood is very large at the beginning, the test data generator searches around the whole neighborhood. If the new candidate solution is not in the range [10000, 99999], it will be given a penalty and will not be accepted. Thus the neighborhood range starts to decrease until the neighborhood range decrease as a small value, i.e. the new candidate solution is also in the small range [10000, 99999]. Then SA/AAN only searches the solution around the small neighborhood. Hence, SA/AAN works very efficiently and the coverage improves very fast. So once SA/AAN finds the range [10000, 99999], it is very likely to obtain high coverage.

6.6 conclusions

In this chapter, we have reported experimental results from five different C/C++ programs using dynamic test data generation. Four optimization algorithms are implemented in the test data generation system. In our knowledge, two of them, Genetic Simulated Annealing and Simulated Annealing with Advanced Adaptive Neighborhood have not been reported in the test data generation literature.

Generally, in the experiments for five target programs, Genetic Algorithm has the best overall performance. It achieves complete condition-decision coverage on the Timeshuttle, Rescue, and Perfect number program. For the Triangle and Hex_dec programs, GA achieves 93.33% and 97.22% respectively. In our experiments, the test requirements in these two programs that GA cannot satisfy cannot be satisfied by the other optimization techniques either. As discussed in the previous sections in this chapter, GA makes good use of the coverage table approach that is applied in the experiments. This approach keeps track of the test cases if they can reach some conditions but cannot cover those conditions. These test cases will become candidate test cases when the test data generator starts to work on these conditions. GA has the ability to keep the good gene inherited from ancestors and contribute it to successive generations. This means GA keeps the good features of the candidates. Such a mechanism helps it find the test cases quickly. This is shown and discussed in detail in the Section 6.42. However other optimization methods used in the experiments do not make good use of the candidates. In the Hex_dec program, GA fails to satisfy one test requirement, which is taking the false branch of decision 3

if (i<=MAX)

which requires that the test case is a valid 8 or more than 8 digit Hexadecimal number. In our experiments, none of the test data generator can generate a test case to satisfy this condition.

In the experiments, GA is tuned by adjusting four parameters: the mutation probability, the crossover probability, the population size and the number of generation. The results show that the 50 parameters do not have a big effect on performance. So GA is easy to implement and has few problem specific decision need to be made. It is a very efficient tool on test data generation.

On three programs under test (Rescue, Triangle and Hex_dec), Simulated Annealing with Advanced Adaptive Neighborhood has very good performance, while on other two programs it does not. For example, SA/AAN performs very well on the Rescue program, even better than GA on the input space [0,2147483647]. In this case, the

search space is huge [0,2147483647], while the desired test cases of rescue program only exist in a small range [10000,99999]. As discussed in the Section 6.5.2, the most important decision is decision 1

if(!(code > 9999 && code < 100000)) // 1

Once the test data generator fails to find the test case, which takes the false branch of this decision, it will fail to reach the other decisions. Although the search space is huge, SA/AAN finds such test case that exists in this small range [10000,99999] very quickly and thus find the test cases that satisfy other test requirements quickly. A possible explanation is that its neighborhood range is not fixed. SA/AAN can make the big jump in the search space and also can make the small change in the neighborhood. The big jump in the huge search space helps it find the desired test cases in the small range [10000,99999] efficiently. Through examination of the result, it has been found that SA/AAN is good at handling the condition like

if a>b

In this situation, it can find a solution quickly relying on the information that the instrumented code provides.

Compared to SA, the adaptive neighborhood helps SA/AAN search the search space roughly and quickly, while this also cause some limitations. In the experiments, on the Timeshuttle program, SA/AAN does not have good performance. The reason is that in some cases, SA/AAN cannot adjust the step size (neighborhood range) flexibly. For example, in the big plateaus, the neighborhood range may decrease too fast, which causes the neighborhood range to decrease as a value smaller than one, and thus the SA/AAN is stuck at one point. While in some other cases, the neighborhood range may increase too fast, so SA/AAN starts to search the space beyond the input space, which is fruitless.

Generally, the results in the experiments show that SA/AAN has the ability to find the solution to satisfy some test requirements very quickly. It works better than other optimization techniques in large input spaces, especially for the program with simple input and a large search space like the Rescue program.

The results of the experiments show that the performance of SA is consistent and predicable while it is very time-consuming. It performs remarkably poorly in the large search space problem, which is due to its limitations. Small changes in the neighborhood make it move slowly to the desirable search space, especially for the Rescue program. The performance of SA is worse than Random test data generator. This is because SA only searches the neighborhood of the seed test case, thus, if the seed test case is not in the small range [10000,99999], it needs a lot of effort to reach this desired search space. In the experiments, SA gives up before it finds this area, thus it fails to reach other test requirements. On the other hand, Random test data generator generates each test case randomly, so it has a better chance to reach this desired area [10000,99999]. However, in the small search space, the performance of SA is stable and it is unlikely stuck in the plateaus since the neighborhood range in SA is fixed and will not decrease to a value smaller than one, i.e. zero in our experiments. However for a small input space [0,524287], SA still fails to obtain complete coverage. The test requirements it fails to satisfy is either " **code<9999**" or " **code>100000**", which are the lower bound and the upper bound of this small range respectively. This means if the input test case of SA is close to upper bound 100000, SA fails to satisfy "**code<9999**", since it takes too much effort for SA to generate a test case smaller than 9999, and vice versa. This is shown in the results of the Timeshuttle and Perfect number program.

Generally, GSA does not perform well in our experiments, and its performance is only slightly better than Random test data generator. However we have not yet tuned GSA in our experiments; this may be the reason why GSA does not perform well. Possible improvement can be achieved in a process of tuning the algorithm. Many parameters can be adjusted in order to improve the performance of GSA.

Random test data generator has a good performance on the simple program with small input space, for example, Hex_dec program. But it has poor performance on those programs with complicated control structure or large input space. The result of Random test data generator resembles those reported in [MMS01].

Chapter 7 Conclusions and future works

Software testing is an essential part in the software development process. It is used to reveal faults in software and ensure that the software performs as intended. Unfortunately, it is an expensive process which typically consumes at least 50% of the total costs of developing software [Bei90]. The most expensive part in software testing is the construction of test data. It is tedious, difficult and labor consuming. This process typically represents at least 40% of the total testing costs [Bei90]. Hence, automation of test data generation is a desirable way to reduce the cost and improve the quality of software.

This thesis presents empirical results of dynamic test data generation based on a function minimization technique. Four optimization methods are used in the experiments, and the Random test data generator is used for the purpose of comparison. A set of experiments is conducted on five different C/C++ programs.

The results show that GA has the best overall performance. It seems that GA is most suitable to take advantage of the existence of a coverage table. Usage of test cases that have reached the conditions/decisions for building an initial population gives GA a better exploration of the search space. This is one of the reasons why it outperforms all of the other four methods on most of the target programs. GA obtains complete coverage on the Perfect number, Timeshuttle and Rescue program, and obtains 97% and 93% coverage on the Hex_dec and Triangle classification program respectively. Besides the fact that GA achieves high coverage, results of our experiments ALSO show that the parameters of GA do not have a big effect on performance, which makes it easy to implement.

In the case of ranking optimization techniques, the second spot is assigned to Simulated Annealing with Advanced Adaptive Neighborhood. In some cases, for programs with simple input and a large input space, SA/AAN performs very well. While in other cases, the performance of SA/AAN is not so good. The possible reason is that the adaptive neighborhood adjustment in this algorithm has limitation in handling big plateaus. For example, in a big plateau, the neighborhood range may decrease too fast, which will cause the neighborhood range to decrease to a value smaller than one, and thus SA/AAN is stuck at one point. On the other hand, in some other cases, the neighborhood range may increase too fast, which can be bigger than the input space, thus the search is fruitless.

In our experiments, the performance of SA is consistent. SA delivers very similar results for a given program across all 10 runs. It performs well on a small search input space, but when it comes to the large input space, it needs to spend much more effort, perform more executions of the program under test, to reach coverage levels similar to the ones obtained by GA.

GSA does not perform well in our experiments like we expected. A possible reason for this can be the large number of parameters that have to be set for this optimization technique. Thus, it appears that the experiments performed have not found parameter values that would lead to good results.

Some possible future research directions are presented below.

- More intelligent neighborhood adjustment can be implemented in SA/AAN. For example, the neighborhood range can be adjusted when it is not bigger than one or when it is bigger than the range of search space. This may prevent the search to stall on the local optimum or plateau. It would also be possible to investigate the effect of the cooling schedule on performance.
- As discussed before, the effect of tuning parameter for some of the optimization methods used in this thesis have not been investigated yet. It would be desirable to conduct the experiments with different parameter setting, especially for SA/AAN and GSA.

- In this thesis, the experiments presented are limited to small programs. It is essential to evaluate the performance of these optimization methods on the test data generation using real, large-scale software.
- Plateaus in the search space are common problems in dynamic test data generation. This is normally caused by the two-valued decision and the deeply nested conditions. The results in our experiments show that these problems hinder optimization methods form working efficiently. It would be desirable to introduce an improved strategy, for example, a static strategy, to deal with this problem.
- More complicated selection methods can be implemented in GSA to improve its performance.
- It would be possible to build a framework that implements multiple optimization methods to generate the test data for real, large-scale software. So for different parts of software under test, the framework assigns the problem of test data generation to different optimization techniques. For example, for the program with simple input and large input space, SA/AAN would be used to generate the test data.

Bibliography

[Bak85] J. E. Baker, Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates (Hillsdale), 1985.

[Bei90] B.Beizer. *Software System Techniques*. Thomson Computer Press, 2nd edition, 1990.

[BH91] Thomas Bäck, Frank Hoffmeister, Extended Selection Mechanisms in Genetic Algorithms, p. 92-99, Proc. of the Fourth Int. *Conf. on Genetic Algorithms (Conference paper)* 1991

[Bur67] W. Burkhardt. Generating programs from syntax. Computing, 2(1):83-94, 1967.

[Cla76] L. Clark. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215-222, September 1976.

[Coh90] J.Cohen. Constraint logic programming languages. *Communications of the ACM*, 33:52-68, July 1990

[Cor87] Corana, A., Marchesi, M., Martini, C. and Ridella, S.: Minimizing Multimodal Functions of Continuous Variables with the "Simulated Annealing" Algorithm, *ACM Trans. on Mathematical Software*, Vol. 13, No. 3, pp. 262-280, 1987.

[CPD93] A. Coen-Porisini and F. Depaoli. Array representation in symbolic execution. Computer Language, 18(3): 197-216,1993.

[CS88] R. A. Caruana and J. D. Schaffer. Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In *Proceedings of the Fifth International conference on Machine Learning*. Morgan Kaufmann, 1988

[Dan63] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[DO91] R. Demillo and A. Offutt. Constraint-based automatic test data generation. *IEEE transactions on software Engineering*, 17(9):900-910,1991.

[Dow93] Kathryn A, Dowsland. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 2 – Simulated Annealing, pages 20-69. McGraw Hill, 1993.

[FK96] R. Ferguson and B. Korel, The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63-86,1996.

[Gar99] Stewart Gardiner, editor. *Testing Safety-related Software: A Practical Handbook*, Springer, 1999.

[Glo86] F. Glover (1986) Future paths for integer programming and links to artificial intelligence, *Computers and Operations Research*, 1, 533-549.

[Glo89] F. Glover, Tabu search, Part I, ORSA Journal on Computing. 1(1989)190-206.

[Glo90] F. Glover, Tabu search, Part II, ORSA Journal on Computing. 2(1990)4-32.

[GN97] M. J. Gallagher and V. L. Narasimhan. Adtest: A Test Data Generation Suite for Ada Software Systems. IEEE Transactions. Software Engineering, Vol. 23, No. 8, pp. 473-484, Aug. 1997.

[Gol89] D. E Goldberg. *Genetic Algorithms in search, optimization, and machine learning*, Addison Wesley, New York, 1989.

[Hol75] Holland, J.H., *Adaptation in Natural and Artificial Systems*. University of Michigan press, 1975.

[IM] Paul J. Iglinski, Sarah McEvoy Solutions of the assignments of the course ENCMP100 (Computer Programming for Engineers)

[Inc87] D. Ince. The automatic generation of test data. *Computer Journal*, 30(1): 63-69, 1987.

[JSE95] B. Jones, H. Sthamer, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of 3rd conference on Software Quality Management,* volume 2, pages 435–444, 1995.

[JM91] C. Z. Janikow, Z. Michalewiz. An experimental comparison of binary and floating point representations in genetic algorithms. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms,* pages 31-36. Morgan Kaufmann, 1991.

[KGV83] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671-680, May 1983.

[KKD95] Seichi Koakutsu, Maggie Kang, Wanye Wei-Ming Dai, Genetic Simulated Annealing and Applications to Non-slicing Floorplan Design. (UCSC-CRL-95-52). 1995.

[Kor90] B. Korel. Automated software test data generation. *IEEE transactions on software Engineering*, 16(8):870-879, 1990.

[Kor96] B. Korel. Automated software test data generation for programs with procedures. In *International Symposium on Software Testing and Analysis*, pages 209-215. ACM/SIGSOFT, 1996.

[Mar94] Marc Roper. *Software testing*. McGraw-Hill Book Company European, 1994

[MHO02] Mitsunori MIKI, Tomoyuki HIROYASU, Keiko ONO. Simulated Annealing with Advanced Adaptive Neighborhood. In *Computational Intelligence and Applications (Proceedings of the 2nd International Workshop on Intelligent Systems Design and Applications :* ISDA-02), pp. 113-118, (2002).

[MMS01] Christoph C. Michael, Gary McGraw, Michael A. Schatz Generating Software Test Data by Evolution. *IEEE transactions on software Engineering*, 27(12):1085-1110, 2001.

[MS76] W. Miller and D. Spooner. Automated generation of floating–point test data. *IEEE transactions on software Engineering*, SE-2 (3): 223-226, September 1976.

[Off91] A. Jefferson Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391-409, 1991.

[OK96] Abraham H. Osman and James P. Kelly. Meta-heuristics: An Overview *Meta-Heuristics: Theory & Applications.* 1-22, 1996.

[OJP97] A. Jefferson Offutt, Zhenyi Jin, Jie Pan. The dynamic domain reduction procedure for test data generation. *Software Practice and Experience*, 29(2):167-193, January, 1997.

[PHP99] Roy P. Pargas, Mary Jean Harrold, Robert R Peck. Test-data Generation Using Genetic Algorithms. *Journal of Software Testing, Verifications, and Reliability,* vol. 9, pp.263-282, September 1999.

[Pre00] Roger S. Pressman, *Software engineering-A practitioner's approach.* McGraw-Hill publishing company. European adaptation 5th edition, 2000.

[RHC76] C. Ramamoorthy, F. Ho, and W. Chen. On the automated generation of program test data. IEEE transactions on Software Engineering, SE-2(4):293-300, 1976

[Sch87] Schaffer, J.D. (1987) Some effects of selection procedures on Hyperplane Sampling by Genetic Algorithms. In. *Genetic Algorithms and Simulated Annealing.* L. Davis, ed. Pitman.

[Tra00] Nigel James Tracey. A Search-Bases Automated Test-data Generation Framework For Safety-Critical Software, PhD Thesis, University of York, 2000.

Appendices

Hex_dec conversion program

```
# include <stdio.h>
# define MAX 7
 long int htoi (char[]);
int main ()
{
    char t[MAX];
    char c;
                int i=0;
                printf("\nInput a hex number:");
                while ((c=getchar())!='\n')
                {
                        if(c>='0' &&c<='9'||c>='a'&&c<='f'||c>='A'&&c<='F')
                        t[i++]=c;
                        else
                        {
                                printf("\nNot a valid hex number");
                                return 0;
                        }
                }
                if (i<=MAX)
                {
                        t[i]='\n';
                        printf("decimal number: %d\n",htoi(t));
                }
                else printf("\nMaximum 7 digits of hex number");
                return 0;
}

long int htoi(char s[])
{       int j;
        long int n;
        n=0;
        for (j=0;s[j]!='\n';j++)
                { if (s[j]>='0'&&s[j]<='9')
                        n=n*16+s[j]-'0';
                  if (s[j]>='a'&&s[j]<='f')
                        n=n*16+s[j]-'a'+10;
                  if (s[j]>='A'&&s[j]<='F')
                        n=n*16+s[j]-'A'+10;
                }
        return (n);
```

}

Timeshuttle program

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
using namespace std;       //introduces namespace std
enum Month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT,
NOV, DEC};
enum Weekday {SUN,MON,TUE,WED,THU,FRI,SAT};
void randomTrip(Month mToday, int dToday, int yToday, Month m, int d, int y);
void plannedTrip(Month mToday, int dToday, int yToday, Month m, int d, int y);
bool isLeapYear(int year);
int gregorianDay(Month m, int d, int y);
bool isValidDate(Month m, int d, int y);
int getYearDay(Month m, int d, int y);
Weekday getWeekday(int gDay);
string dayName(Weekday w);
string monthName(Month m);
void getTodaysDate(Month& m, int& d, int& y);
void gDay2MDY(int gDay, Month& m, int& d, int& y);
int daysInMonth(Month m, int y);
int daysInYear(int y);
int randInt(int a, int b);
void outputMessage(int days, Month mToday, int dToday, int yToday,
Weekday w, Month m, int d, int y);
bool getInputDate(Month&m, int& d, int& y);
const int YEAR1 = 1582;
const int YEARMAX = 4316;
const int YEARDAYS = 365;
const int YEAR1DAYS = daysInYear(YEAR1) - getYearDay(OCT, 14, YEAR1);
int main( void )
{
    srand(time(0));
    bool validInput;
    Month mDest;
    int dDest, yDest;
    Month mToday;
    int dToday, yToday;
    getTodaysDate(mToday, dToday, yToday);
    validInput = getInputDate(mDest, dDest, yDest);
    if (validInput)
    {
            plannedTrip(mToday, dToday, yToday, mDest, dDest, yDest);
```

```
    }
    else
    {
            randomTrip(mToday, dToday, yToday, mDest, dDest, yDest);
    }
    return 0;
}
void plannedTrip(Month mToday, int dToday, int yToday, Month m, int d, int y)
{
    Weekday w;
    int destDay;
    int daysTraveled;
    int today;
    today = gregorianDay(mToday, dToday, yToday);
    destDay = gregorianDay(m, d, y);
    daysTraveled = destDay - today;
    w = getWeekday(destDay);
 outputMessage(daysTraveled, mToday, dToday, yToday, w, m, d, y);
}
void randomTrip(Month mToday, int dToday, int yToday, Month m, int d, int y)
{
    Weekday w;
    int destDay;
    int daysTraveled;
    int today;
    today = gregorianDay(mToday, dToday, yToday);
    daysTraveled = randInt(-10000, 10000);
    destDay = today + daysTraveled;
    w = getWeekday(destDay);
    gDay2MDY(destDay, m, d, y);
    outputMessage(daysTraveled, mToday, dToday, yToday, w, m, d, y);
}
bool isLeapYear(int year)
{
    return (year % 4 ==0) && (year %100 != 0 || year % 400 == 0);
}
int gregorianDay(Month m, int d, int y)
{
    int gregDay = 0;
    int yearDay;
    if (isValidDate(m, d, y))
    {
            yearDay = getYearDay(m, d, y);
            if (y == YEAR1)
            {
                    gregDay = yearDay - (YEARDAYS - YEAR1DAYS);
```

```
            }
            else
            {
                gregDay = YEAR1DAYS;
                for (int i = YEAR1+1; i < y; i++)
                {
                    gregDay += daysInYear(i);
                }
                gregDay += yearDay;
            }
    }
    return gregDay;
}
bool isValidDate(Month m, int d, int y)
{
    bool isValid;
    if (m < JAN || m > DEC)

        isValid = false;
    else if (d > daysInMonth(m, y) || d < 1)
        isValid = false;
    else if (y < YEAR1 || y > YEARMAX)
        isValid = false;
    else if (y == YEAR1)
    {
        if (m < OCT)
            isValid = false;
        else if (m == OCT && d < 15)
            isValid = false;
        else
            isValid = true;
    }
    else
        isValid = true;
    return isValid;
}
int getYearDay(Month m, int d, int y)
{
    int yearDay = 0;
    for (Month mo=JAN; mo<m; mo = static_cast<Month>(mo + 1))
    {
        yearDay += daysInMonth(mo, y);
    }
    return yearDay + d;
}
```

```
Weekday getWeekday(int gDay)
{
    Weekday w = static_cast<Weekday>((gDay+4) % 7);
    return w;
}
string dayName(Weekday w)
{
    string name;
    switch(w)
    {
            case SUN:
                    name = "Sunday"; break;
            case MON:
                    name = "Monday"; break;
            case TUE:
                    name = "Tuesday"; break;
            case WED:
                    name = "Wednesday"; break;
            case THU:
                    name = "Thursday"; break;
            case FRI:
                    name = "Friday"; break;
            case SAT:
                    name = "Saturday"; break;
    }
    return name;
}

string monthName(Month m)
{
    string name;
    switch(m)
    {
            case JAN:
                    name = "January"; break;
            case FEB:
                    name = "February"; break;
            case MAR:
                    name = "March"; break;
            case APR:
                    name = "April"; break;
            case MAY:
                    name = "May"; break;
            case JUN:
                    name = "June"; break;
            case JUL:
```

```
                name = "July"; break;
        case AUG:
                name = "August"; break;
        case SEP:
                name = "September"; break;
        case OCT:
                name = "October"; break;
        case NOV:
                name = "November"; break;
        case DEC:
                name = "December"; break;
    }
    return name;
}


void getTodaysDate(Month& m, int& d, int& y)
{
    time_t currentTime;
    time( &currentTime);
    m = static_cast<Month>(localtime( &currentTime)->tm_mon + 1);
    d = localtime( &currentTime)->tm_mday;
    y = localtime( &currentTime)->tm_year + 1900;
}

void gDay2MDY(int gDay, Month& m, int& d, int& y)
{
    y = YEAR1;
    d = 0;
    m = JAN;
    if (gDay <= YEAR1DAYS)
    {
        m = OCT;
        if (gDay <= daysInMonth(OCT, y) - 14)
            d = 14;
        else
        {
            gDay -= daysInMonth(OCT, y) - 14;
            m = NOV;
        }
    }
    else
    {
        gDay -= YEAR1DAYS;
        y++;
        while (gDay > daysInYear(y))
```

```
            {
                gDay -= daysInYear(y);
                y++;
            }
    }

    while (gDay > daysInMonth(m, y))
    {
        gDay -= daysInMonth(m, y);
        m = static_cast<Month>(m + 1);
    }
    d += gDay;
}


int daysInMonth(Month m, int y)
{
    int days = 0;
    switch(m)
    {
        case JAN: case MAR: case MAY: case JUL: case AUG: case OCT:
case DEC:
            days = 31;
            break;
        case APR: case JUN: case SEP: case NOV:
            days = 30;
            break;
        case FEB:
            days = 28 + isLeapYear(y);
            break;
    }
    return days;
}

int daysInYear(int y)
{
    return YEARDAYS + isLeapYear(y);
}

int randInt(int a, int b)
{
    return (a + rand() % (b - a + 1));
}

void outputMessage(int days, Month mToday, int dToday, int yToday,
                                    Weekday w, Month m, int d, int y)
```

```
{
    cout << "Traveling ";
    if (days < 0)
            cout << -days;
    else
            cout << days;
    cout << " days into the ";
    if (days < 0)
            cout << "past";
    else
            cout << "future";
    cout << " from today" << endl;
    cout << monthName(mToday) << " " << dToday << ", " << yToday << endl;
    cout << "will bring you to" << endl;

    cout << dayName(w) << " " << monthName(m) << " " << d << ", " << y <<
endl;
}

bool getInputDate(Month&m, int& d, int& y)
{
    bool valid;
    int mo;
    cout << "Enter the destination date of your time travel (month day year):" <<
endl;
    cin >> mo >> d >> y;
    m = static_cast<Month>(mo);
    valid = isValidDate(m, d, y);
    if (!valid)
            cout << "Your date is not valid. You will be given a random trip into
time.\n";
    cout << endl;
    return valid;
}
```

Perfect number program

```
#include <iostream>
#include "perfectnum.h"
using namespace std;
perfectnum:: perfectnum ()
{
        num = 0;
        p = 0;
        i = 0;
```

```
        perfectsum = 1;
        temp = 0;
}
perfectnum:: perfectnum(int number)
{
        num = number;
        p = 0;
        i = 0;
        perfectsum = 1;
        check = 0;

}


void perfectnum::multiple()
{


        if ((num % 7) == 0)
                p = 1;
        if ((num % 11) == 0)
                p = 2;
        if ((num % 13) == 0)
                p = 3;
        else
                ;

        switch(p)
        {
        case 1:
                cout << "It is multiple of 7" << endl;
                break;
        case 2:
                cout << " It is multiple of 11" << endl;
                break;
        case 3:
                cout << " It is multiple of 13" << endl;
                break;
        default:
                cout << "It is not multiple of 7, 11, or 13" << endl;
        }

}

void perfectnum::sum()
{
```

```
        check = num + 1;
        check = num % 2;
        if (check == 0)
                cout << "The number is odd" << endl;
        else
                cout << "The number is even" << endl;
}

void perfectnum::prime()
{
        for (i = 1; i < (0.5 * num); i++)
        {
                if (i != num && num != 1)
                {
                        if ((num % i) == 0)
                                p++;
                }
                else
                ;
        }

        if (p > 0)
                cout << "It is not a prime number" << endl;
        else
                cout << " It is a prime number" << endl;
}
void perfectnum::perfect()
{
        if (num % 2 == 0)
        {
                for (i =num/2; i >=2 ; i--)
                {
                        if (num % i == 0)
                        {
                                perfectsum += i;
                        }
                }

                if (perfectsum == num)
                {       cout << "The number is a perfect number" << endl;
                        if (num>100)
                        cout<<"It is bigger than 100"<<endl;
                        if (num>1000)
                        cout<<"It is bigger than 1000"<<endl;
                }
                else
```

```
                cout << "The number is not a perfect number" << endl;
        }
else
;
}
int main()
{
        int number;
                cin >> number;
                perfectnum user(number);
                cout << number << endl << endl;
                user.multiple();
                cout << endl << endl;
                user.sum();
                cout << endl << endl;
                user.prime();
                cout << endl << endl;
                user.perfect();
                cout << endl;
        return 0;
}
```

Rescue program

```
**********************************************************
File name: rescue.cpp
Description:
This program contains four rules, which can be used to crack
the secret code to save the co-op student
**********************************************************/
#include <iostream>
#include <cstring>

using namespace std;  //introduces namespace std
int main( void )
{
        int base;                       // intermediate variable for extracting digits
        int code;                       // the secret code
        int digit1, digit2, digit3, digit4, digit5; // digits in the code
        int rescueDay;          // code for rescue day
        string day;                     // string with day of the week
        int rendezvousPt;       // code for rendezvous point
        string point;           // string for rendezvous point
        int sum = 0;            // sum of digits
```

```
        const string FALSE_MSG = "False message: "; // constant part of false
message
        // Get code
        cout << "Please enter a code to break: ";
        cin >> code;

        // Check if the input is a five-digit number
        if(!(code > 9999 && code < 100000))        {
                cout << FALSE_MSG << "Not a five-digit number." << endl;
        }
        else
        {
                // Obtain the individual digits first
                base = code;
      digit5 = base % 10;
      base /= 10;
      digit4 = base % 10;
      base /= 10;
      digit3 = base % 10;
      base /= 10;
      digit2 = base % 10;
      base /= 10;
      digit1 = base % 10;

      sum = digit1 + digit2 + digit3 + digit4 + digit5;

                // Check if sum is even
                if (!(sum%2 == 0))
                {
                        cout << FALSE_MSG << "Sum is odd." << endl;
                }
                else
                {
                        // Obtain the rescue day
                        rescueDay = (digit1 * digit2) - digit3;
                        // Check if rescue day is valid
                        if(rescueDay < 1 || rescueDay > 7)
                        {
                                cout << FALSE_MSG << "Invalid rescue day." <<
endl;
                        }
                        else
                        {
                                // Obtain the rendezvous point
                                if(digit4 == digit5)
                                {
```

```
                rendezvousPt = 0;  // Assign an invalid number
        }
        else if(digit4 > digit5)
        {
                rendezvousPt = digit5 + 1;
        }
        else
        {
                rendezvousPt = digit5 - 1;
        }
        // Check if rendezvous point is valid
        if((rendezvousPt != 2) && (rendezvousPt != 7) &&
(rendezvousPt != 8))
        {
                cout << FALSE_MSG << "Invalid rendezvous
point." << endl;
        }
        else
        {
                // Get the rescue day
                switch(rescueDay)
                {
                        case 1:
                                day = "Monday";
                                break;
                        case 2:
                                day = "Tuesday";
                                break;
                        case 3:
                                day = "Wednesday";
                                break;
                        case 4:
                                day = "Thursday";
                                break;
                        case 5:
                                day = "Friday";
                                break;
                        case 6:
                                day = "Saturday";
                                break;
                        case 7
                                day = "Sunday";
                                break;
                        default:
                                cout << "Error in rescue day!!! ";
                                break;
```

```
                                        } // end of switch
                                        // Get the rendezvous point
                                        switch(rendezvousPt)
                                        {
                                                case 2:
                                                        point = "fountain";
                                                        break;
                                                case 7:
                                                        point = "large tree";
                                                        break;
                                                case 8:
                                                        point = "church";
                                                        break;
                                                default:
                                                        cout << "Error in rendezvous
point!!! ";
                                                        break;
                                        } // end of switch

                                        // Print message
                                        cout << "Rescue on " << day << " at the " <<
point << "." << endl;
                                }
                        }
                }
        }
        return 0;
}
```