Open access • Journal Article • DOI:10.1016/J.INFSOF.2008.11.001

Automated test data generation using a scatter search approach — Source link

Raquel Blanco, Javier Tuya, Belarmino Adenso-Díaz

Institutions: University of Oviedo

Published on: 01 Apr 2009 - Information & Software Technology (Butterworth-Heinemann)

Topics: Test case, Test data generation, Automatic test pattern generation, Code coverage and Local search (optimization)

Final preprint version published in Information and Software Technology, 51(4) 708-720, April 2009
© Elsevier, doi:10.1016/j.infsof.2008.11.001
Submitted: January 21th, 2008, Accepted: November 12th, 2008

Automated test data generation using a Scatter Search approach

Raquel Blanco[(1)*], Javier Tuya[(1)], Belarmino Adenso-Díaz[(2)]
[(1)]Department of Computer Science, [(2)]Department of Management Science
University of Oviedo
Campus de Viesques, Gijón, Asturias, 33204 Spain
{rblanco | tuya | adenso}@uniovi.es

Abstract

The techniques for the automatic generation of test cases try to efficiently find a small set of cases that allow a given adequacy criterion to be fulfilled, thus contributing to a reduction in the cost of software testing. In this paper we present and analyze two versions of an approach based on the Scatter Search metaheuristic technique for the automatic generation of software test cases using a branch coverage adequacy criterion. The first test case generator, called TCSS, uses a diversity property to extend the search of test cases to all branches of the program under test in order to generate test cases that cover these. The second, called TCSS-LS, is an extension of the previous test case generator which combines the diversity property with a local search method that allows the intensification of the search for test cases that cover the difficult branches. We present the results obtained by our generators and carry out a detailed comparison with many other generators, showing a good performance of our approach.

Keywords: Software testing, automatic test case generation, branch coverage, Scatter Search, Local Search, metaheuristic search techniques.

1 Introduction

Software testing is the process of executing a program in order to find faults, thus helping developers to improve the quality of the product when the discovered faults are solved and reducing the cost produced by these faults. A software test consists of a set of test cases, each of which is made up of the input of the program, called test data, and the output that must be obtained. As the target of software testing is to find faults, a test is successful if an error is found.

Testing is a very important, though expensive phase in software development and maintenance; it has been estimated that software testing entails between 30 percent [28] and 50 percent [4] of

* Corresponding author. Tel.: +34 985 182 689; fax: +34 985 181 986.
E-mail address: rblanco@uniovi.es (Raquel Blanco)

software development. A challenging part of this phase entails the generation of test cases. This generation is crucial to the success of the test because it is impossible to achieve a fully tested program given that the number of test cases needed for fully testing a software program is infinite [46], and a suitable design of test cases will be able to detect a great number of faults. For these reasons, the techniques for automatic generation of test cases try to efficiently find a small set of cases that allow an adequacy criterion[†] to be fulfilled, thus reducing the cost of software testing [22][51] and resulting in more efficient testing of software products .

The search for an optimal solution in the test case generation problem has a great computational cost and for this reason these techniques try to obtain near optimal solutions. As a consequence, they have attracted growing interest from many researchers in recent years. On the other hand, the nature of Software Engineering problems is ideal for the application of metaheuristic techniques, as is shown in the work of Harman & Jones [27], and besides they obtain good results in test case generations [38]. The search space of solutions in test case generation is very large and many metaheuristic techniques explore a region closer to a specific solution. The metaheuristic technique known as Scatter Search [24] has obtained good results in many combinatorial problems, including the set covering problem [50], routing problems [5][52] and the project scheduling problem [64]. These problems have a similar structure as the problem we deal here: they are modelled by means of a graph and their objective is to find a set of solutions that allow covering the requirements represented by this graph according to several constraints. So the use of Scatter Search to explore the wide search space of solutions of software testing seems to be a promising research area.

Scatter Search [24] is an evolutionary method that works on a population of solutions of the problem to be solved (the definition of a solution depends on the specific problem), which are stored in a set of solutions (Reference Set). The solutions in this set are combined looking for new solutions that hopefully are better than the original ones. The Reference Set stores the best solutions that have been generated so far. To determine whether a solution is good, not only its quality (cost) but also its diversity in the set of solutions is considered. The diversity of the set of solutions is a general concept which indicates the difference among their members in relation to certain attribute. A diverse set of solutions allows by the mating of its members a wider search space to be explored in order to find solutions. In our case, the solutions are the test data,

[†] "An adequacy criterion is considered to be a stopping rule that determines whether sufficient testing has been done […] and provides measurements of test quality" [65]

which must be diverse enough to cover the situations required to fulfill a decision coverage criterion.

The rest of the paper is organized as follows. The next section presents an overview of related studies on the automation of test case generation that use metaheuristic search techniques. Section 3 details our Scatter Search approaches for the automatic generation of test cases called TCSS and its extension, TCSS-LS. In Section 4 we present the results of our generators and carry out a comparison with the generators developed by other authors. Finally, Section 5 presents the conclusions of this paper.

2 Background

The test data adequacy criteria used in software testing can be divided into structural testing, fault-based testing and error-based testing [65]. Structural testing uses a coverage measure to specify the test requirements. Fault-based testing tries to detect faults in the program. Error-based testing checks the program at certain points that we know to be problematic. Depending on the source of information used for test case selection, the above criteria can be also divided into program-based, which generate test cases from the code of the program under test, and specification-based, which generate test cases from the program specification.

Several approaches have been used for the automation of test cases generation for program-based structural testing. Among these approaches, we may distinguish between random generation [48], static techniques [16] and dynamic techniques [31]. The random technique generates test cases randomly and is widely used to perform comparisons with other techniques. This technique has been refined by Adaptive Random Testing [11][12][13], which incorporates procedures that aim to obtain an even distribution of test cases. Static techniques generate test cases from several constraints based on the input variables of the program under test. Some recent studies use constraint solvers [7][34] and constraint logic programming [26][43]. The static techniques have several problems, such as the treatment of loops, the resolution of computed storage locations or their computational cost [8][41][44].

Dynamic techniques carry out a direct search of test cases by means of the execution of the program under test, which has been previously instrumented. As the value of all variables is known at runtime, many of the problems relating to the static techniques can be avoided [41]. Most dynamic techniques use metaheuristic search techniques (Genetic Algorithms, Genetic Programming, Simulated Annealing, Tabu Search, Scatter Search, etc.). The application of metaheuristic algorithms to solve problems in Software Engineering was proposed by the SEMINAL network (Software Engineering using Metaheuristic INnovative ALgorithms) and is widely explained in [14]. One of these applications is software testing, in which the testing

problem is treated as a search or optimization problem, as is shown in several surveys [38][41]. Besides, as has been pointed out previously, this problem is ideal for the application of metaheurístic techniques [27].

The most widely used metaheuristic technique in this yield is Genetic Algorithms [25]. This technique is based on the principles of genetics and Darwin's theory of evolution. The Genetic Algorithm operates on a population which is improved in each iteration of the search by means of the use of three natural processes: selection, crossover and mutation. This technique is used in many papers to achieve several coverage criteria: Jones *et al.* [29][30], Miller *et al.* [45], Pargas *et al.* [51] and Sthamer [54] use Genetic Algorithms to obtain branch coverage, Michael *et al.* [44] to achieve condition-decision coverage, Ahmed & Hermadi [1], Bueno & Jino [8], Lin & Yeh [36] and Watkins & Hufnagel [60] to reach path coverage, Wegener *et al.* [62] to obtain several coverage criteria, and Girgis [23] to obtain def-use coverage. Other papers that apply Genetic Algorithms to generate test cases are the works of Watkins *et al.* [61], which use two Genetic Algorithms to generate test suites that are then used to train a series of decision tree in order to create rules for classifying test cases, Alshraideh & Bottaci [3], which use this metaheuristic technique to cover string predicates, Del Grosso *et al.* [15], which generate test cases that detect buffer overflows and Ngo & Tan [47], which present an approach for infeasible path detection and integrates it with a test data generator that adopts the test data generation technique based on Genetic Algorithms proposed by Tonella [55].

Genetic Programming has been used in the work of Emer & Vergilio [21], which present an approach called GPBT (Genetic Programming Based Testing) to create alternatives of the program under test and describes how this approach can help the tester in selection and evaluation of test data sets, and by Vergilio & Pozo [58], which apply the Grammar-Guided Genetic Programming (GGGP) approach to the classification task in the context of data mining of relational databases and the selection of test cases using the mutation testing adequacy criterion in the context of software testing. Simulated annealing has been used by Tracey *et al.* [56] to generate test cases based on the system specifications and to test exceptions, and by Waeselynck *et al.* [59], which investigate measures of landscape to apply this technique to test generation. Tabu Search has been used to obtain branch coverage in the works of Díaz *et al.* [19][20] and to obtain branch, path and loop coverage by Díaz [17]. Bueno *et al.* [9] present a new algorithm called Simulated Repulsion to generate diverse test data, according the diversity measures proposed in their work, and evaluate the effect of diversity on data flow coverage and mutation testing. McMinn & Holcombe [42] describe a hybrid solution that combines the principles of evolutionary algorithms with an extended chaining approach to find test cases that

cover a target. Bühler & Wegener [10] describe the application of evolutionary algorithms to the automation of functional testing.

The aforementioned papers focus on the description of the proposed solution, the majority of comparisons being carried out using a random algorithm. Only a few works perform a comparison with other algorithms. The following papers present several approaches based on different metaheuristic techniques (Genetic Algorithms are used in all the studies) and carries out a comparison of these. Mansour & Salame [37] apply Genetic Algorithms and Simulated Annealing to generate test cases to achieve path coverage and also carries out a comparison with Korel's Algorithm [31]. Xiao *et al.* [63] obtain test cases for the condition-decision coverage using the metaheuristic techniques Genetic Algorithms, Simulated Annealing, Genetic Simulated Annealing and Simulated Annealing with Advanced Adaptive Neighbourhood. Li *et al.* [35] study the application of Genetic Algorithm and Hill Climbing to regression test case prioritization and investigates their effectiveness in comparison with three greedy algorithms. Alba & Chicano [2] generate test cases to reach condition coverage using two Genetic Algorithms and two approaches based on Evolutionary Strategies.

Another metaheuristic technique that can be applied to automatic test case generation is Scatter Search [24][33] (see section 3.1 below). This technique has been used to solve many problems, as is shown in [39]. However, the only papers that use the Scatter Search technique to automate the generation of test cases are the works of Blanco *et al.* [6] and Sagarna & Lozano [53], which use this technique to obtain branch coverage. The approaches adopted in these works differ in several aspects to the Scatter Search technique, such as the selection of the node used in each iteration of the search process, the contents of the sets of solutions used by the technique to generate the new solutions of the problem to solve, the updating process of these sets of solutions and their size. In this paper we describe two versions of a Scatter Search approach to obtain branch coverage called TCSS and TCSS-LS. TCSS is a refined version of the preliminary approach described in [5]. Both TCSS and TCSS-LS differ from the approach presented in [53] in the use of the Scatter Search technique, since they use Scatter Search from the beginning of the search of test cases, whereas the work described in [53] proposes to begin the search with an Estimation of Distribution Algorithm (EDA) and to use the Scatter Search to increase the branch coverage obtained by the EDA.

3 Scatter Search approach to software testing

In this section we explain our adaptation of the Scatter Search technique to the automatic generation of test cases to obtain branch coverage. Section 3.1 briefly explains the Scatter Search technique. In Section 3.2 we present the general aspects of our Scatter Search approach for automatic test case generation, called TCSS. Section 3.3 shows the search process of new

test cases (new solutions), while Sections 3.4, 3.5 and 3.6 explain the methods that appear in this search process. Finally, Section 3.8 describes the use of the local search method of the TCSS-LS version.

3.1 Scatter Search technique

As said in section 1, Scatter Search [24] is an evolutionary method that works on a set of solutions, called the Reference Set, which stores the best solutions that have been generated so far. The solutions in this set are combined in order to obtain new ones, trying to generate each time better solutions, according to quality and diversity criteria.

The basic scheme of the Scatter Search algorithm can be seen in Figure 1 [33]. The Scatter Search algorithm begins by using a diversity generation method to generate P diverse solutions, to which an improvement method is applied. Then the Reference Set is created with the best solutions from P and the most diverse in relation to the solutions already in the Reference Set. As new solutions are generated, the algorithm produces subsets of the Reference Set using a subset generation method, and applies a solution combination method in order to obtain new solutions, to which an improvement method is applied. Then a Reference Set update method evaluates the new solution to verify whether they can update the Reference Set, as they are better than some solutions stored in the set. If so, the best solutions are included in the Reference Set and the worst solutions are dropped. So, the final solution of the problem to solve is stored in the Reference Set.

------- FIGURE 1 ------

3.2 TCSS: A Scatter Search approach to test coverage

Our Scatter Search approach, called TCSS (Test Coverage Scatter Search) has given rise to two versions: a first version of TCSS which is based on a preliminary approach described in [6] and a second version that uses a local search method to intensify the search of test cases that can cover the most difficult branches, which is called TCSS-LS (Test Coverage Scatter Search – Local Search).

Both versions of TCSS work on the control flow graph associated with the program under test. The control flow graph is a directed graph G=(N, E, s, e), where N is a set of nodes, E is a set of directed edges $a_{ij} = (n_i,n_j) \in N \times N$, $s \in N$ is the initial node and $e \in N$ is the output node of the graph. Each node $n \in N$ represents a linear sequence of computations for the program under test and has its own Reference Set. Each arch a_{ij} represents the transfer of the execution control of the program from node n_i to node n_j when the associated arch decision is true. By means of this

control flow graph, it is possible to determine the branches covered by the test cases generated, since the program under test has been instrumented to know the followed path.

An example of control flow graph generation can be seen in Figure 2. Node 0 represents the initialization of the variable "value" and the control statement "if (x1>4)". The two exit edges indicate the true evaluation and the false evaluation of the node decision. Node 1 corresponds to the control statement "if (x2<5 && x3<10)" and also has two exit edges. Node 2 joins the two statements included in the "if" part of the control statement of node 1 and the node 3 joins the statements of the "else" part. Node 4 represents the end of block if-else. Finally, node 5 represents the statements of "else" part of node 0. By means of this control flow graph, it is possible to determine paths that start in the root node and finish in a leaf node. These paths represent one possibility of ending the program and each of them is formed by the nodes reached during the execution of the program. The three paths of the example program are (0-1-2-4), (0-1-3-4) and (0-5). This control flow graph has three types of nodes: root node which marks the beginning of the execution of the program (node 0), branch node which represents a branch of the program (nodes 1, 2, 3 and 5) and non-branch node which represents the start or end of a block (node 4).

------- FIGURE 2 ------

The goal of TCSS is to generate test cases that allow all the branches of the program to be covered. This general goal is divided in subgoals, each of which consists in finding test cases that reach a particular branch node of the control flow graph. For instance, the goals of the control flow graph in the Figure 2 are nodes 1, 2, 3 and 5 (the root node is always reached and the node 4 is always reached when a test case achieves the node 2 or the node 3). Each decision of the program generates two branch nodes which represent the true evaluation and the false evaluation of the decision. Therefore, if all the branch nodes of the control flow graph have at least one test case that reaches them, we can conclude that all the branches of the program have been covered and all the decisions have been evaluated in their two alternatives.

The nodes of the control flow graph store information during the process of test case generation to reach the subgoals. This information allows the covered branches to be known and is used to make progress in the search process. The root node and each branch node store this information in an own set of solutions, called Reference Set. Unlike the original Scatter Search algorithm, our approach has several Reference Sets. Each Reference Set is called S_k, where k is the number of the node, and is formed by B_k elements $T_k^c = < \overline{x}_k^c, p_k^c, fb_k^c, fc_k^c >$, $c \in \{1..B_k\}$, where:

- $\overline{x}_k^c$ is a solution, i.e., a test case that reaches node n_k. Each solution $\overline{x}_k^c$ consists of a set of given values for the input variables (x1, x2,..., xn) of the program under test that satisfy the decisions of the previous nodes to node n_k on the path that has been covered. For primitive data types the elements of the solution $\overline{x}_k^c$ =(x1, x2,…, xn) take the value of the variable, in the case of arrays each of their positions is an element of the solution, and when structures are used each of their components is an element of the solution.
- p_k^c is the path covered by the solution (test case), i.e., the sequence of the nodes of the control flow graph reached by the solution.
- fb_k^c is the distance to the sibling node, i.e., the node whose input decision is the negation of the node n_k decision (the distance between two nodes is defined in Section 3.5). This distance indicates how close the solution came to cover the sibling node.
- fc_k^c is the distance to the child node that has not been reached by the solution (the distance between two nodes is defined in Section 3.5). This distance indicates how close the solution came to cover the child node.

The control flow graph of Figure 2 with the information stored in the Reference Sets of the root node and the branch nodes can be seen in Figure 3. Note that the node 4 does not contain set S_k.

------- FIGURE 3 ------

The set of solutions of a node n_k (S_k) has a maximum size B_k. This size is different for each node n_k and depends on the complexity of the source code situated below node n_k, which is calculated by the cyclomatic complexity [40] of the control flow graph of the program, with node n_k as the root node. This value is multiplied by a fixed factor. Thus, a reasonable number of solutions to generate new solutions are available. The different sizes of the sets S_k are a result of the division of the problem into subgoals. The solutions stored in a node n_k are used to make progress in the search process by means of the combinations that generate new solutions; TCSS tries to reach the nodes located at lower levels than node n_k in the control flow graph with these combinations. Therefore the sets S_k of the nodes nearer to the root of the control flow graph are larger in size than the sets S_k of the nodes nearer to the leaves.

TCSS will try to make the sets as diverse as possible using a diversity function. Thus it tries to explore a wide search space in order to find solutions that can cover different branches of the program. The diversity of a solution of a set S_k is a measure related to the path covered by all solutions of the set.

3.3 Search process

The goal of TCSS is to obtain maximum branch coverage, i.e., to find solutions that allow coverage of all the nodes of the control flow graph. As these solutions are stored in the nodes, our goal is therefore for all the nodes to have at least one element in their set S_k. However, this goal cannot be reached when the program under test has unfeasible branches. Therefore, TCSS also stops its execution when a maximum number of test cases has been generated. Initially, the sets S_k are empty and they are filled by the generator by means of its iterations. Figure 4 shows the scheme of the TCSS search process.

------- FIGURE 4 ------

In each iteration, TCSS selects a node to generate new solutions. The root node is chosen at the first iteration and random solutions are generated in its set S_k. In subsequent iterations, the generator selects a node and generates new solutions by means of the combination of several solutions of its set S_k. The selected node fulfils the following conditions:

- It has not been evaluated, i.e., it has not been used to generate new solutions.
- Its parents have been already evaluated.
- Among all candidates, it has the greater number of elements in its set of solutions S_k.

If the selected node does not have at least two solutions to perform the combinations, a backtracking process, which is not considered in the original Scatter Search algorithm, is carried out. This backtracking process uses a Mutation method in the first version of TCSS and a Local Search method in the TCSS-LS version. Section 3.7 explains the backtracking process.

TCSS tries to select a node from the top of the control flow graph that has a larger number of solutions in its set S_k than the other nodes, because it can contain more diverse solutions due to having more paths below these. On the other hand, the different sizes of the sets S_k also help in the selection of the node to evaluate, as the top nodes have more solutions because they can store more elements.

The instrumented program under test is executed with each new solution and the sets S_k of the nodes reached in this execution are updated. Then the selected node is marked as evaluated and a new node is chosen to generate new solutions.

The final solution of the generator consists of the sets of values for the input variables that cover the branches of the control flow graph, which are in the sets S_k, the branch coverage reached and the time consumed in the search process.

3.4 Generation of new solutions

When TCSS selects a node n_k to generate new solutions, it chooses a constant number of elements $T_k^c = <\overline{x}_k^c, p_k^c, fb_k^c, fc_k^c>$ from the set S_k of this node n_k that has not been used to generate new solutions. The generator tries to select solutions ($\overline{x}_k^c$) that cover different paths (p_k^c) and have less distance to the sibling node (fb_k^c). Thus, TCSS attempts to generate diverse solutions that can cover the branch of the sibling node. As the solutions that cover a node can be stored in the set S_k of the child node reached, TCSS converts the problem of covering the other child node into a problem of covering the sibling node.

Then all possible pairs ($\overline{x}_k^j$, $\overline{x}_k^h$), j≠h, are formed to carry out the following combinations, element by element: $\overline{x}_k^{ji}+\Delta_i$, $\overline{x}_k^{ji}-\Delta_i$, $\overline{x}_k^{hi}+\Delta_i$, $\overline{x}_k^{hi}-\Delta_i$, where $\Delta_i=|\overline{x}_k^{ji}-\overline{x}_k^{hi}|/2$ and the index i covers all input variables. These combinations have been also used in [6][53] and are based on the linear combinations of Laguna & Marti [32], which uses the first three combinations and include a random element in the calculation of Δ_i. Each new solution is checked to determine whether its values overflow the ranges of the input variables. In that case, the value of the input variable that overflows the range is substituted by the limit of the range.

Using these combinations, TCSS tries to generate solutions further from the original ones and solutions situated between them. Furthermore, the solution selection criteria attempt to combine diverse solutions (they cover different paths) which are closer so as to cause a branch jump.

3.5 Calculation of distances

In order to calculate the distances fb_k^c and fc_k^c of a node n_k, TCSS instruments the decisions of the nodes of the program under test. When a solution reaches a node n_k, TCSS calculates fb_k^c using the input decision of the sibling node and calculates fc_k^c using the input decision of the child node that has not been reached by the solution (the sibling node of the child node reached by the solution). For instance, if the solution $\overline{x}_1^1$ in Figure 3 reaches nodes 1 and 2, the distance fb_1^1 is calculated with the input decision of node 4 (!(x1>4)) and the distance fc_1^1 is calculated with the input decision of node 3 (!(x2<5 && x3<10)). The definition of distances fb_k^c and fc_k^c is shown in Table 1. These distances are based on those in [31][57].

------- TABLE 1 ------

First, each condition of the decision used is evaluated according the values of the input variables that constitute the solution. Then, the value of the distances fb_k^c and fc_k^c is calculated. When the decision has AND operators, the evaluations of false conditions are added to calculate the distances, as these false conditions impede reaching the sibling node. When the decision has OR operators, the distance is the minimum value of the evaluation of all conditions, as all conditions

are false and the node (sibling or child) is reached when one of them becomes true. When negations appear in the decision, De Morgan's law is used.

3.6 Updating the sets of solutions

To update the sets S_k of the nodes reached in an execution of the program under test, TCSS checks the sizes of these sets and the state of the nodes. Thus, the following situations are considered:

- The node has been evaluated and its set S_k is not full; the solution is included in the set.
- The node has been evaluated and its set S_k is full; the solution is not included in the set.
- The node has not been evaluated and its set S_k is not full; the solution is included in the set.
- The node has not be evaluated and its set S_k is full; first the solution is included in the set (provisionally the set exceeds its maximum size) and then the diversity function is applied to determine the solution that must leave the set.

TCSS adds solutions to the sets of evaluated nodes since these new solutions can be used in the backtracking process.

The diversity function is applied over the subset $S_{p*}=\{T_{p*}^{1},\ldots,T_{p*}^{q}\} \subseteq S_k$, $T_{p*}^{i}=<\overline{x}_{p*}^{i};p_{p*};fb_{p*}^{i};fc_{p*}^{i}>$, which represents the solutions stored in node n_k that cover the path p_{p*} with more occurrences in the set S_k. The most similar solution to the rest of solutions that cover the same path (the less diverse) leaves the set S_k. The diversity value of a solution is calculated according the diversity function defined as:

$$div(<\overline{x}_{p*}^{m};\ p_{p*}>;S_{p*}) = \sum_{y=1..q}\left(\sum_{i=1..n}\left|\frac{\overline{x}_{p*}^{m_i}-\overline{x}_{p*}^{y_i}}{range_i}\right|\right)$$

where index y=1..q covers the solutions of S_{p*}, index i=1..n covers the input variables and $range_i$ is the range of values of the input variable i.

When there are two or more solutions with the same diversity value, TCSS checks the value of their distance fb. The solution with the higher value of distance fb, i.e., the further one that can cause a branch jump, leaves the set S_k.

TCSS applies the diversity function over the subset that covers the path with more occurrences because it tries to equilibrate the number of solutions that cover different paths, thus achieving more diversity.

3.7 Backtracking process

The most difficult situation in the search process occurs when node n_k selected to generate new solutions does not have at least two solutions in its set S_k to perform the combinations. This situation happens when no test case reaches node n_k (or only one test case reaches it) and, in addition, the other candidates do not have at least two solutions either. As TCSS cannot progress in the search process, it therefore applies a backtracking process. This process tries to increase the size of the set S_k of node n_k by means of the generation of new solutions using the parent node.

TCSS has two options to generate new solutions in the backtracking process:

- To perform combinations: this option is possible when the parent node has solutions that have not been used in previous combinations in its set S_k. In this case, the combinations are performed with these solutions. This is the reason why the sets S_k of the evaluated nodes accept solutions when they are not full, so the parent node has a set of solutions that can be used in the backtracking process.
- To perform mutation operations: when all solutions of the parent node have been used in previous combinations, TCSS select some solutions of the parent node to generate new solutions by means of a mutation operation. This mutation operation consists in changing the value of some input variable for other random value according to a mutation probability.

The instrumented program under test is executed with each new solution and the sets S_k of the nodes reached in this execution are updated as explained in Section 3.6. If any new solution reaches node n_k, TCSS carries out backtracking again. If the backtracking backs away to the root node, the algorithm carries out a regeneration process, as the current solutions do not allow the obtaining of total coverage. This regeneration process cleans the sets S_k of the node that belong to some path that ends in node n_k and randomly generates the solutions of the root node. The sets cleaned in the regeneration process store one solution to remember the covered nodes.

3.8 TCSS-LS: TCSS with a Local Search method

Comparison of a previous version of TCSS with another approach based on a technique that works with a Local Search procedure (Tabu Search) [18] shows that the Tabu Search approach is more efficient in the last iterations of the search of test cases, due to the fact that the Local Search is focused on finding solutions that reach the nodes that have not yet been covered , whereas the Scatter Search approach is more efficient in the first iterations of the search, due to the use of the diversity property, which allows many different branches to be reached with less test cases. The behaviour of both techniques suggests the incorporation of a Local Search

method in the TCSS generator to improve the search of test cases for some branches in which the diversification used by Scatter Search finds it difficult to cover. This incorporation has given rise to TCSS-LS.

TCSS-LS uses a Local Search method in the backtracking process, instead of the mutation operations used by TCSS, as this is the point in which TCSS begins to have difficulty to cover some specific branches. Thus, TCSS-LS implements an intensification process to try to cover the nodes that have not been reached in the diversification process. After those nodes have been covered, TCSS-LS uses diversification strategies to cover the rest of the nodes.

To generate solutions that cover a node n_k, the local search method selects a solution (called the original solution ($\overline{os}$)) from the set S_k of the parent node and carries out, at the most, MAX_ATTEMPT iterations to reach the node n_k. This solution has the lower value of the distance to the child node to reach (fc_i^c) in order to try to guide the search.

In each iteration, TCSS-LS generates 2n new solutions (where n is the number of input variables of the solution) from the best solution (called the current solution ($\overline{cs}$)) of the previous iteration:

- $\overline{ns1} = \overline{cs}^i + \delta_i$
- $\overline{ns2} = \overline{cs}^i - \delta_i$

where the index i covers all input variables and δ_i is defined as follows:

$$\delta_i = \begin{cases} \dfrac{fc_{cs}}{jump\_reducer} & \text{if } \begin{cases} \text{TCSS calculates } \overline{ns1} \text{ and } \delta_i \le max\_rightjump \\ \text{or TCSS calculates } \overline{ns2} \text{ and } \delta_i \le max\_leftjump \end{cases} \\ \dfrac{max\_rightjump}{jump\_reducer} \cdot Ln\left(\dfrac{fc_{cs} \cdot (e-1)}{fc_{os}} + 1\right) & \text{if TCSS calculates } \overline{ns1} \text{ and } \delta_i > max\_rightjump \\ \dfrac{max\_leftjump}{jump\_reducer} \cdot Ln\left(\dfrac{fc_{cs} \cdot (e-1)}{fc_{os}} + 1\right) & \text{if TCSS calculates } \overline{ns2} \text{ and } \delta_i > max\_leftjump \end{cases}$$

where *jump_reducer* is a parameter that is increased to reduce the jump and the maximum jumps are:

- max_rightjump = Upper Range – $\overline{cs}^i$
- max_leftjump = $\overline{cs}^i$ – Lower Range

Then the generator executes the program under test with each new solution and evaluates the solutions to determine whether any solution reaches the parent node and improves the value of the distance fc of $\overline{cs}$ in order to use the best one as current solution. If no solution improves $\overline{cs}$ and the local search has not carried out MAX_ATTEMPT iterations from the original

solution, TCSS-LS reduces the jump used in δ_i to generate nearer solutions to $\overline{cs}$ in the next iteration. If the local search has performed MAX_ATTEMPT iterations from the original solution and node n_k has not been reached, TCSS-LS selects another solution ($\overline{os}$) from the set S_k of the parent node to carry out the search. When all solutions of the parent node have been used in the local search method, TCSS-LS performs a backtracking process and uses the solutions of another ancestor to reach node n_k. The local search finishes when node n_k is covered or when the backtracking process in this local search backs away to the root node.

The sets S_k of the nodes reached during the local search are updated as described in Section 3.6, except for the node that has the solution $\overline{os}$ used in the search. Although the set S_k of this node is not updated during the execution of the new solutions, it is updated when the local search finishes or selects another $\overline{os}$ to continue the search. The $\overline{os}$ is replaced with the best new solution that improves it.

If node n_k has not been reached by means of the local search, TCSS-LS uses the new solutions stored in the sets S_k to carry out the combinations as the last attempt to cover node n_k. Then node n_k is labelled as evaluated.

4 Performance analysis

In this section we present the results obtained by our approach when compared with other methods using several benchmark programs and several input ranges to analyze its efficiency. A review of the literature was carried out to identify the benchmark programs. Although the literature presents many benchmark programs, the selection of a set of such programs to compare approaches is not easy because not all of them have been used with the same adequacy criterion, their source code is not always available, or the results presented are not complete (range and type of input variables, number of test cases generated, time consumed, percentage of coverage reached), all of which makes comparison difficult. Table 2 presents the set of selected benchmark programs for which the comparison can be carried out according to our background research. For each benchmark the table shows a brief description, its abbreviation, the number of branches, the nesting level and the cyclomatic complexity.

------- TABLE 2 ------

A comparison of TCSS-LS with TCSS is carried out in Section 4.1 to analyze the improvement that TCSS-LS represents. In Section 4.2 we compare the results of TCSS-LS with the results of other approaches published so far. As other authors did, we have also compared TCSS and

TCSS-LS with the most basic procedure, a random algorithm, and the results of the approaches based on Scatter Search always outperform the random algorithm.

In all cases for our experiments, the stopping condition used for the generators TCSS and TCSS-LS was that of reaching 100% branch coverage or reaching 200000 generated test cases. For each benchmark and each input range, we carried out 10 runs with the generators, taking average values. All runs were carried out on a Pentium processor 1.50GHz with a RAM memory of 512 MB.

4.1 TCSS vs TCSS-LS

The benchmark programs presented in Table 2 were used to compare the behaviour of both version of TCSS in terms of the number of test cases generated and the time consumed. Whenever possible, each benchmark was executed with both integer and float ranges.

Each benchmark was executed with three different signed ranges for the integer input variables: a low range (L), a medium range (M) and a high range (H) (8, 16 and 32 bits respectively), a signed range that uses 32 bits (H) for the float input variables and an unsigned range that uses 8 bits (L) for the char input variables. Benchmarks CD and TW were not executed with the low integer range due to the decision of an "if" statement in the source code that would avoid reaching 100% coverage with this range.

The results obtained by the two generators can be seen in Table 3. Each instance represents a benchmark that has been executed with a specific input range and a specific type of input variables (C for char variables, F for float variables and I for integer variables). For both generators, the percentage of coverage reached, the number of test cases that the generator creates to achieve this coverage and the time consumed (in seconds) are shown for each instance. Note that the test case generators use a large set of test cases to cover all branches, because during the search process they generate test cases that reach branches that had already been covered by other test cases; however not all of them form the set of test cases used in the test process of the program. To obtain the minimum set of test cases that are executed in the program under test to cover all branches during the test process we select a test case from each set S_k of a leaf node. For example, TCSS-LS uses 271 test cases to achieve the total coverage of the TM program (instance 27 of Table 3), but we select only 12 test cases from the sets S_k, one for each leaf node, to test the program, reaching the 100% branch coverage. To perform a correct comparison, the number of test cases generated and the time consumed must be compared when both generators obtain the same percentage of branch coverage. For this reason, when TCSS-LS obtains more coverage than TCSS, the table also shows in brackets the number

of test cases generated and the time consumed by TCSS-LS to reach the same coverage obtained by TCSS.

------- TABLE 3 ------

The results obtained by the two generators indicate that TCSS-LS always reaches the coverage achieved by TCSS. From all 40 instances, TCSS achieves 100% coverage in 23 instances and TCSS-LS improves the coverage, reaching 100% in 33 instances. For the remaining 7 instances in which both generators do not reach 100% coverage, TCSS-LS increases the coverage obtained by TCSS in 5 of these.

As regards the test cases generated and the time consumed to achieve the same coverage (25 instances), TCSS-LS requires fewer test cases in 19 instances, in which it consumes less time (instances 3, 6, 7, 15-19, 21, 22, 24-29, 34-36). Only in 5 instances it generates a larger number of test cases than TCSS (instances 9-11, 20, 31) and in 3 of these, it consumes more time (instances 9-11). In these instances, the local search does not cover some nodes that are then covered by the solutions generated by means of the new combinations performed after the backtracking. Those nodes are successors of a node with an equality condition which contains a variable that appears in the decision of the successor nodes. When the local search tries to achieve the successor nodes, it generates new solutions modifying only the value of a variable of the current solution used (the original solution is stored in the set S_k of parent node). As only one variable of the equality condition is modified, it becomes false. The combination of solutions modifies the value of both variables in the same proportion, so that equality remains true.

In the 15 instances in which TCSS-LS increases the coverage reached by TCSS, it generates fewer test cases to obtain more coverage in 8 instances (instances 1, 2, 4, 5, 12, 30, 32, 33); besides, in 7 of these it consumes less time (instances 1, 2, 4, 5, 30, 32, 33). In the remaining instances, TCSS-LS generates more test cases to increase coverage, but it requires fewer test cases and consumes less time to obtain the maximum coverage achieved by TCSS in 4 instances (instances 8, 13, 14, 39), it generates the same number of test cases and consumes the same time in 2 instances (instances 38, 40) and only in 1 instance it generates more test cases, though it consumes less time than TCSS (instance 37).

On the other hand, TCSS-LS obtains better results (more coverage, fewer test cases and less time) than TCSS in all benchmarks except LR, QFS and TMM. For benchmarks QFS and TMM, TCSS only obtains better result in the low range, while for benchmark LR TCSS it obtains better results in the three integer ranges.

The improvement obtained by TCSS-LS is shown in Figure 5. This figure shows the ratios (logarithmic 10 scale) for test cases (CR) and the ratios for time (TR) which relate TCSS to TCSS-LS. The rhombus points represent the CR of the instances in which TCSS and TCSS-LS achieve the same maximum coverage. The asterisk points represent the CR of the instances in which TCSS-LS reaches more coverage than TCSS. The line "time" represents the TR of each instance. The ratios are calculated by means of the quotient of the test cases (or time) of TCSS divided by the test cases (or time) of TCSS-LS. When the generators do not reach the same maximum coverage, the ratios are calculated with the test cases (or time) needed to obtain the coverage that both generators are capable of reaching, which always coincides with the coverage achieved by TCSS. The instances in Table 3 have been arranged according to input range (L, M, H and float range) and benchmark name in order to show in the figure the influence of range in the improvement provided by TCSS-LS.

------- FIGURE 5 ------

As shown in Figure 5, TCSS-LS generates fewer test cases and consumes less time for the majority of instances (34 of 40). Moreover, the differences between the test cases are large when TCSS-LS generates fewer test cases than TCSS (for example, instance 17) and are small in the contrary case (for example, instance 31). On the other hand, the ratios also increase with the increase in the range of the input variables; hence, the improvement given by TCSS-LS is greater for high ranges.

In order to check that both the number of test cases generated and the time consumed by TCSS-LS are significantly smaller than the test cases generated and the time consumed by TCSS and the number of times that TCSS-LS obtains better results than TCSS is also significant, we carried out a statistical analysis with α=0.05.

The first hypothesis to verify is whether TCSS-LS generates less test cases than TCSS. As the test that enables this hypothesis to be verified (the paired t-test) depends on the normality of the distributions, the Kolmogorov-Smirnov test is carried out to check the normality of the variable $D_{TestCases}$=$TestCases_{TCSS-LS}$ - $TestCases_{TCSS}$. A very small p-value (<0.001) is obtained, so we cannot assume that the variable follows a normal distribution.

Therefore, the Wilcoxon signed rank test for paired data is applied to verify the null hypothesis of median equality (H_0:m_D=0, H_1:m_D<0). The p-value obtained by the analysis is smaller than 0.001<α and therefore the hypothesis H_0:m_D=0 can be rejected. Besides, the average number of test cases generated by TCSS-LS is less than the average generated by TCSS. So we can assume that TCSS-LS generates fewer test cases than TCSS.

The second hypothesis to verify is whether TCSS-LS consumes less time than TCSS. Again, the variable $D_{Time}=Time_{TCSS-LS}$ - $Time_{TCSS}$ does not follow a normal distribution and the Wilcoxon signed rank test for paired data is applied. Once more, the hypothesis $H_0:m_D=0$ can be rejected, as the p-value$<0.001<\alpha$. Besides, the average time consumed by TCSS-LS is also less than the average time consumed by TCSS. So we can assume that TCSS-LS consumes less time than TCSS.

We use the McNemar test to verify the third hypothesis that TCSS-LS "wins" more times than TCSS. We consider a generator to be better than the other if it achieves a higher percentage of branch coverage or if it generates fewer test cases when both generators reach the same percentage of branch coverage. In Table 3 it can be seen that TCSS-LS obtains better results than TCSS in 34 instances, whereas it obtains worse results in 5 instances. Both generators obtain the same result in 1 instance. The p-value obtained is smaller than $0.0001<\alpha$ and therefore the difference between both generators is significant, TCSS-LS obtaining better results than TCSS for more instances.

4.2 TCSS-LS vs other generators

The most interesting comparison for a generator is the comparison performed with the generators of other studies. To carry out this comparison, it is necessary for the generators to obtain results for the same coverage criterion and use the same type of input variables and the same input range. In this section we compare the results obtained by TCSS-LS and the generators of other works by means of the use of the benchmark programs presented in Table 2.

On the other hand, as previously mentioned, in order to carry out a thorough comparison, it is also necessary to know the following results obtained by each generator after its execution: the number of test cases generated, the time consumed and the percentage of branch coverage reached. Although in some cases the range of input variables is not explicitly indicated, this input range can be estimated through the results of the random generator that the studies report (instances 3, 5, 6, 34, 41, 44 in Table 4, which is explained below). Unlike the previous section, a time comparison cannot be performed, as the time consumed by all generators is not published for all instances.

The results obtained by TCSS-LS in comparison with the results of other approaches are shown in Table 4. Once more, the percentage of coverage reached, the test cases generated to achieve this coverage and the time consumed (in seconds) by each generator are shown for each instance (an instance is again defined by a benchmark, a specific input range and a specific type of input variables). For the results of other approaches, the column "Generator" shows the type of generator used to execute the corresponding instance and the reference from which these results

have been obtained. Once more, a signed input range is used for float and integer variables and an unsigned input range is used for char variables.

------- TABLE 4 ------

TCSS-LS reaches 100% coverage in 39 instances and in 30 of these it generates fewer test cases than the other generators (instances 2-12, 16, 19-23, 25, 26, 28-30, 32-39). Of the other 9 instances, it generates fewer test cases than some generators in 2 of these (instances 1 and 27). In the remaining instances, TCSS-LS does not achieve 100% coverage (instances 15, 41-44) or it generates more test cases than the other generators (instances 13, 14, 17, 18, 24, 31, 40).

After analyzing the results, TCSS-LS generates fewer test cases to obtain a higher or equal percentage of coverage (regardless of the type of input variables and input range) than the other generators for the benchmarks BM, CB, CD, LR, QF and TMM. For the remaining benchmarks, the following situations can be observed:

- AF: TCSS-LS generates more test cases than the best generator when the low range is used (instance 1), but it generates fewer test cases than the other two generators.
- ND: TCSS-LS generates more test cases than TSGen.
- QFS: TCSS-LS generates fewer test cases when the input range is increased.
- RS: TCSS-LS generates more test cases than the best generator, but it generates fewer test cases than the other two generators.
- TM: TCSS-LS generates fewer test cases than the other generator for all ranges and types of input variables except for the low integer range, in which the number of test cases generated is similar.
- TS: TCSS-LS generates fewer test cases than the other generators for small ranges and generates more test cases for the high range used.
- TW: TCSS-LS does not obtain 100% coverage because it does not cover a node with an orthogonal condition, which is covered in two of ten runs (this condition is reached in all runs when the input range uses 11 bits). TCSS-LS cover the other branches of the benchmark with fewer test cases than some generators. However a correct comparison cannot be carried out as the number of test cases generated by the other generators for the same coverage is not known.

Figure 6 shows the ratios (logarithmic 10 scale) for the test cases (CR) which relates the best generator of the other works for each instance to TCSS-LS (in each instance the generator which obtains the best result is compared against TCSS-LS). The rhombus points represent the CR of the instances in which TCSS-LS reaches more or equal coverage than the best generator of the other works (when TCSS-LS achieves more coverage than the other generator, the ratio is

calculated with the test cases generated to obtain the same coverage of that generator). The asterisk points represent the CR of the instances in which TCSS-LS reaches less coverage than the best generator of the other approaches. These CR are not comparable, since they are calculated with test cases that allow different percentage of coverage to be reached.

------- FIGURE 6 ------

As is shown in the figure, TCSS-LS once again generates fewer test cases for the majority of instances (32 of 44) and the differences between the test cases are large when TCSS-LS generates fewer test cases than the best generator of the other works for each instance (for example, instance 16).

As in the previous subsection, a statistical analysis with α=0.05 was carried out. This analysis compares the results of TCSS-LS with the results obtained by the best generator of the other works for each instance (again, in each instance the generator which obtains the best result is compared against TCSS-LS). Unlike the comparison between TCSS-LS and TCSS, the parameter "time consumed" is not checked because it is not presented in the published results of most generators.

The first hypothesis to verify is whether TCSS-LS generates fewer test cases than the best generator of the other works for each instance. As in the previous section, the variable $D_{TestCases}$ does not follow a normal distribution and the Wilcoxon signed rank test for paired data is applied with the same hypothesis. The p-value obtained by the analysis is $0.004<\alpha$ and therefore the hypothesis $H_0:m_D=0$ can once again be rejected. Moreover, the average number of test cases generated by TCSS-LS is lower than the average generated by the best generators of the other works for each instance. Hence, we can assume that TCSS-LS generates fewer test cases than the other generators.

The second hypothesis to verify is whether TCSS-LS "wins" more times than the best generators of the other works for each instance. As in the previous section, the McNemar test was used to check this hypothesis. TCSS-LS obtains better results than the other generators in 30 instances and obtains worse results than some generators in 14 instances. The p-value obtained is $0.0229<\alpha$ and therefore TCSS-LS once more obtains better results for more instances than the other generators.

5 Conclusions

This paper presents two automatic generators of software test cases, called TCSS and TCSS-LS. Both generators are based on the metaheuristic technique Scatter Search, while TCSS-LS also

uses a Local Search procedure. TCSS and TCSS-LS use the control flow graph associated with the program under test and handles a set of solutions in each node of the graph, thus facilitating the division of the general goal in subgoal.

TCSS diversifies the search of test cases by means of the diversity function in order to cover a great number of branches, while TCSS-LS diversifies the search in the first iterations and then intensifies the search through a Local Search procedure to obtain test cases that cover the most difficult nodes.

In many research studies, it is usual to compare the generators with a random generator or with other tailored generators, but a comparison with the generators of other works is not presented due to the difficulty of carrying out a complete comparison with other generators, since each study uses different benchmarks and different ranges to evaluate the generators and all the data needed to perform a comparison (range of the input variables, number of test cases generated, percentage of branch coverage achieved, time consumed) is not always available. In spite of this difficulty, in this paper we have compared the results of TCSS-LS with the results reported by several works for the different benchmarks and input ranges they use.

The results of the experiments and the statistical studies show that TCSS-LS can be applied to the generation of test cases to obtain branch coverage and that it performs statistically better than the other test case generators included in the comparison. The results also show that the use of several techniques working together, such as Scatter Search and Local Search, improves the efficiency of test case generation.

Acknowledgements

This study was funded by the Department of Education and Science (Spain) and ERDF funds under the National Programme for Research, Development and Innovation, projects IN2TEST (TIN2004-06689-C03-02), Test4SOA (TIN2007-67843-C06-01), RePRIS (TIN2007-30391-E).

References

[1] M.A. Ahmed, I. Hermadi, GA-based multiple paths test data generator, Computers and Operations Research 35(10) (2008) 3107-3124.

[2] E. Alba, F. Chicano, Observations in using parallel and sequential evolutionary algorithms for automatic software testing, Computers and Operations Research 35(10) (2008) 3161-3183.

[3] M. Alshraideh, L. Bottaci, Search-based software test data generation for string data using program-specific search operators, Software Testing Verification and Reliability 16(3) (2006) 175-203.

[4] B. Beizer, Software testing techniques, second edition, Van Nostrand Reinhold, 1990.

[5] E. Benavent, A. Corberán, E. Piñana, I. Plana, J.M. Sanchis, New heuristic algorithms for the windy rural postman problem, Computers and Operational Research 32(12) (2005) 3111-3128.

[6] R. Blanco, J. Tuya, E. Díaz, B.A. Díaz, A Scatter Search approach for automated coverage in software testing, in: Proceedings of the First International Conference on Knowledge Engineering and Decision Support, 2004, pp. 387-393.

[7] B. Botella, A. Gotlieb, C. Michel, Symbolic execution of floating-point computations, Software Testing Verification and Reliability 16(2) (2006) 97-121.

[8] P.M.S. Bueno, M. Jino, Automatic test data generation for program paths using Genetic Algorithms, International Journal of Software Engineering and Knowledge Engineering 12(6) (2002) 691-709.

[9] P.M.S. Bueno, W.E. Wong, M. Jino, Improving random test sets using the diversity oriented test data generation, in: Proceedings of the Second International Workshop on Random Testing, 2007, pp. 10-17.

[10] O. Bühler, J. Wegener, Evolutionary functional testing, Computers and Operational Research 35(10) (2008) 3144-3160.

[11] K.P. Chan, T.Y. Chen, D. Towey, Restricted random testing: adaptive random testing by exclusion, International Journal of Software Engineering and Knowledge Engineering 16(4) (2006) 553-584.

[12] T.Y. Chen, F.-C. Kuo, R.G. Merkel, S.P. Ng, Mirror adaptive random testing, Information and Software Technology 46(15) (2004) 1001-1010.

[13] T.Y. Chen, R. Merkel, Quasi-Random testing, IEEE Transactions on Reliability 56(3) (2007) 562-568.

[14] J. Clarke, J.J. Dolado, M. Harman, R.M Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Proceedings – Software 150(3) (2003) 161-175.

[15] C. Del Grosso, G. Antoniol, E. Merlo, P. Galinier, Detecting buffer overflow via automatic test input data generation, Computers and Operational Research 35(10) (2008) 3125-3143.

[16] R.A. DeMillo, A.J. Offutt, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17(9) (1991) 900-910.

[17] E. Díaz, Generación automática de pruebas estructurales de software mediante Búsqueda Tabú , PhD Thesis Department of Computer Science, University of Oviedo (2004).

[18] E. Díaz, R. Blanco, J. Tuya, Applying Tabu and Scatter Search to automated software test case generation, in: Proceedings of the Sixth Metaheuristics International Conference, 2005, pp. 290-297.

[19] E. Díaz, J. Tuya, R. Blanco, Automated software testing using a metaheuristic technique based on Tabu Search, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003, pp. 310-313.

[20] E. Díaz, J. Tuya, R. Blanco, J.J. Dolado, A tabu search algorithm for Software Testing, Computers and Operational Research 35(10) (2008) 3052-3072.

[21] M.C. Emer, S.R. Vergilio, Selection and evaluation of Test Data Based on Genetic Programming, Software Quality Journal 11 (2003) 167-186.

[22] R. Ferguson, B. Korel, The Chaining Approach for Software Test Data Generation, ACM Transactions on Software Engineering and Methodology 5(1) (1996) 63-86.

[23] M.R. Girgis, Automatic test data generation for data flow testing using a genetic algorithm, Journal of Universal Computer Science 11(6) (2005) 898-915.

[24] F. Glover, M. Laguna, R. Martí, Fundamentals of Scatter Search and Path Relinking, Control and Cybernetics 39(3) (2000) 653-684.

[25] D. Goldberg, Genetic Algorithms in search, optimization, and machine learning, Addison-Wesley, Boston, MA, USA, 1989.

[26] A. Gotlieb, T. Denmat, B. Botella, Goal-oriented test data generation for pointer programs, Information and Software Technology 49(9-10) (2007) 1030-1044.

[27] M. Harman, B.F. Jones, Search-based software engineering, Information and Software Technology 43(14) (2001) 833-839.

[28] A. Hartman, Is ISSTA Research Relevant to Industry?, ACM SIGSOFT Software Engineering Notes 27(4) (2002) 205-206.

[29] B.F. Jones, D.E. Eyres, H.H. Sthamer, A strategy for using Genetic Algorithms to automate branch and fault-based testing, The Computer Journal 41(2) (1998) 98-107.

[30] B.F. Jones, H.H. Sthamer, D.E. Eyers, Automatic structural testing using genetic algorithms, The Software Engineering Journal 11(5) (1996) 299-306.

[31] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16(8) (1990) 890-879.

[32] M. Laguna, R. Marti, Experimental testing of advanced Scatter Search designs for global optimization of multimodal functions, Technical report TR11-2000, Departamento de Estadística e Investigación Operativa, University of Valencia, 2000.

[33] M. Laguna, R. Martí, Scatter Search: Methodology and Implementations in C, Kluwer Academic Publishers, Boston, MA, USA, 2002.

[34] J.J. Li, D. Weiss, H. Yee, Code-coverage guided prioritized test generation, Information and Software Technology 48(12) (2006) 1187-1198.

[35] Z. Li, M. Harman, R.M. Hierons, Search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33(4) (2007) 225-237.

[36] J. Lin, P. Yeh, Automatic test data generation for path testing using Gas, Information Sciences 131 (1-4) (2001) 47-64.

[37] N. Mansour, M. Salame, Data generation for path testing, Software Quality Journal 12 (2004) 121-136.

[38] T. Mantere, J.T. Alander, Evolutionary software engineering, a review, Applied Soft Computing 5(3) (2005) 315-331.

[39] R. Martí, Scatter Search—Wellsprings and Challenges, European Journal of Operational Research 169(2) (2006) 351–358.

[40] T.J. McCabe, A complexity measure, IEEE Transaction on Software Engineering 2(4) (1976) 308-320.

[41] P. McMinn, Search-based software test data generation: a survey, Software Testing Verification and Reliability 14(2) (2004) 105-156.

[42] P. McMinn, M. Holcombe, Evolutionary testing using an extended chaining approach, Evolutionary Computation 14(1) (2006) 41.64.

[43] C. Meudec, ATGen: automatic test data generation using constraint logic programming and symbolic execution, Software Testing Verification and Reliability 11(2) (2001) 81-96.

[44] C. Michael, G. McGraw, M. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering 27(12) (2001) 1085-1110.

[45] J. Miller, M. Reformat, H. Zhang, Automatic test data generation using genetic algorithm and program dependence graphs, Information and Software Technology 48 (2006) 586-605.

[46] G. Myers, The art of software testing, Ed. John Wiley & Sons, 1979.

[47] M.N. Ngo, H.B.K. Tan, Heuristics-based infeasible path detection for dynamic test data generation, Information and Software Technology 50 (2008) 641-655.

[48] S. Ntafos, On random and partition testing, in: Proceedings of International Symposium on Software Testing and Analysis, 1998, pp. 42-48.

[49] A.J. Offut, J. Pan, K. Tewary, T. Zhang, Experiments with Data Flow and Mutation Testing. Technical Report ISSE-TR-94-105, 1994.

[50] J.A. Pacheco, A. Álvarez, S. Casado, J.F. Alegre, Heuristic solutions for locating health resources, IEEE Intelligent Systems 23 (1) (2008) 57-63.

[51] R.P. Pargas, M.J. Harrold, R.R. Peck, Test data generation using genetic algorithms, Journal of Software Testing Verification and Reliability 9 (1999) 263-282.

[52] R.A. Russell, W.C. Chiang, Scatter search for the vehicle routing problem with time windows, European Journal of Operational Research 169(2) (2006) 606-622.

[53] R. Sagarna, J.A. Lozano, Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms, European Journal of Operational Research 169(2) (2006) 392-412.

[54] H.H. Sthamer, The automatic generation of software test data using genetic algorithms, PhD Thesis, University of Glamorgan, 1996.

[55] P. Tonella, Evolutionary testing of classes, in: Proceedings of ACM SIGSOFT International Symposium of Software Testing and Analysis, 2004, pp. 119-128.

[56] N. Tracey, J. Clark, K. Mander, Automated program flaw finding using simulated annealing, in: Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, 1998, pp. 73-81.

[57] N. Tracey, J. Clark, K. Mander, J. McDermid, Automated test-data generation for exception conditions, Software Practice and Experience, 30(1) (2000) 61-79.

[58] S.R. Vergilio, A. Pozo, A grammar-guided genetic programming framework configured for data mining and software testing, International Journal of Software Engineering and Knowledge Engineering 16(2) (2006) 245-267.

[59] H. Waeselynck, P. Thévenod-Fosse, O. Abdellatif-Kaddour, Simulated annealing applied to test generation: landscape characterization and stopping criteria, Empirical Software Engineering 12(1) (2007) 35-63.

[60] A. Watkins, E.M. Hufnagel, Evolutionary test data generation: a comparison of fitness functions, Software Practice and Experience 36 (2006) 95-116.

[61] A. Watkins, E.M. Hufnagel, D. Berndt, L. Johnson, Using genetic algorithms and decision tree induction to classify software failures, International Journal of Software Engineering and Knowledge Engineering 16(2) (2006) 269-291.

[62] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology 43(14) (2001) 841-854.

[63] M. Xiao, M. El-Attar, M. Reformat, J. Miller, Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques, Empirical Software Engineering 12(2) (2007) 183-239.

[64] D.S. Yamashita, V.A. Armentano, M. Laguna, Scatter search for Project scheduling with resource availability cost, European Journal of Operational Research, 169(2) (2006) 623-637.

[65] H. Zhu, P.A.V. Hal, J.H.R. May, Software Unit Test Coverage and Adequacy, ACM Computing Surveys 29(4) (1997) 366-427.

Table Captions

Table 1: TCSS distance function

Table 2: Benchmark programs

Table 3: Results obtained in the experimentation comparing TCSS and TCSS-LS: % coverage achieved, test cases generated and time consumed. Each instance is defined by a benchmark executed with a specific input range (L for 8 bits, M for 16 bits and H for 32 bits) and a specific type of input variable (C for char, F for float and I for integer). The values in brackets are the test cases generated and the time consumed by TCSS-LS to reach the same coverage obtained by TCSS.

Table 4: Results obtained in the experimentation comparing TCSS-LS and other generators: % coverage achieved, test cases generated and time consumed. Each instance is defined by a benchmark executed with a specific input range and a specific type of input variable (C for char, F for float and I for integer).

Figure Captions

Figure 1: Basic scheme of Scatter Search

Figure 2: Control flow graph example

Figure 3: TCSS control flow graph

Figure 4: TCSS scheme

Figure 5: Ratios between the test cases generated and time consumed for TCSS and TCSS-LS. The ratios are calculated by the quotient TCSS / TCSS-LS

Figure 6: Ratios between the test cases generated by the best generator of other works and TCSS-LS. The ratios are calculated by the quotient best_other_generator / TCSS-LS

Table 1: TCSS distance function

Condition	**eval(Condition, $\bar{x}$)**
x=y, x≤y, x≥y	|x-y|
x≠y, x<y, x>y	|x-y| + σ
Decision	**fb_k^c, fc_k^c**
C1 AND C2	$\sum \text{eval}(C_j, \bar{x})\ \forall C_j$ False
C1 OR C2	$\text{Min}(\text{eval}(C_j, \bar{x}))\ \forall C_j$
¬C	Negation is propagated using De Morgan's law

Table 1: TCSS distance function

Table2: Benchmark programs

Benchmark	Abbr.	Number of Branches	Nesting Level	Cyclomatic Complexity	Refe-rence
Atof: Converts an array of characters into a floating point number.	AF	30	2	17	[62]
BisectionMethod: Calculates the square root of a number using the bisection method	BM	8	3	5	[49]
ComplexBranch: An artificial program that contains several difficult branches.	CB	24	3	14	[62]
CalDay: Calculates the day of the week	CD	22	2	12	[2]
LineRectangle: Determines the position of a line with respect to a rectangle.	LR	36	12	19	[20]
NumberDays: Calculates the number of days between two dates.	ND	86	10	44	[20]
QuadraticFormula: Solves a quadratic equation.	QF	4	2	3	[49]
QuadraticFormulaSthamer: Determines the type of quadratic equations roots: real and unequal, real and equal or complex.	QFS	6	3	4	[54]
RemainderSthamer: Calculates the remainder of a division.	RS	18	5	10	[54]
TriangleMyers: The classical classify triangle which determines the following types of triangle: equilateral, isosceles, scalene or no-triangle.	TM	12	5	7	[46]
TriangleMichael: The classify triangle problem with a different implementation.	TMM	20	6	11	[44]
TriangleSthamer: A more complete classify triangle problem, as it also checks the right angles of the triangle.	TS	26	12	14	[54]
TriangleWegener: A classify triangle problem which determines the following types of triangle: equilateral, isosceles, orthogonal, obtuse angle or no-triangle.	TW	26	3	14	[62]

Table 2: Benchmark programs

Table 3: Results obtained by TCSS and TCSS-LS

Instance	Benchmark	Type	Range	Results for TCSS			Results for TCSS-LS		
				% Cov.	Test Cases	Time (sec)	% Cov.	Test Cases	Time (sec)
1	AF	C	L	97.19	101144	141.1	100.00	17134 (12167)	8.3 (4.7)
2	BM	F	H	88.00	52680	39.8	100.00	233 (194)	0.2 (0.2)
3	CB	I	L	100.00	6059	30.0	100.00	4740	17.1
4	CB	I	M	91.92	121223	608.1	100.00	5385 (3838)	15.2 (10.8)
5	CB	I	H	74.23	31211	41.2	100.00	26752 (472)	39.4 (0.8)
6	CD	I	M	100.00	38075	8.8	100.00	558	0.6
7	CD	I	H	100.00	39209	9.1	100.00	561	0.7
8	LR	F	H	97.50	5054	2.4	100.00	5939 (2601)	2.1 (1.2)
9	LR	I	L	100.00	1986	1.2	100.00	4213	1.4
10	LR	I	M	100.00	1581	1.0	100.00	3538	1.3
11	LR	I	H	100.00	2188	1.2	100.00	5818	1.7
12	ND	I	L	99.79	98174	193.2	100.00	76271 (76271)	269.1 (269.1)
13	ND	I	M	6.17	42410	17.9	100.00	115380 (2992)	361.7 (1.3)
14	ND	I	H	2.87	100912	43.3	99.04	148880 (1298)	164.5 (0.6)
15	QF	F	H	100.00	26234	3.8	100.00	51	0.0
16	QF	I	L	100.00	111	0.0	100.00	49	0.0
17	QF	I	M	100.00	26234	3.8	100.00	51	0.0
18	QF	I	H	100.00	26234	3.8	100.00	51	0.0
19	QFS	F	H	100.00	16024	2.5	100.00	2978	0.3
20	QFS	I	L	100.00	618	0.2	100.00	1938	0.2
21	QFS	I	M	100.00	25053	3.9	100.00	4409	0.4
22	QFS	I	H	100.00	23838	3.7	100.00	604	0.1
23	RS	I	L	100.00	101	0.1	100.00	101	0.1
24	RS	I	M	100.00	27676	4.8	100.00	482	0.3
25	RS	I	H	100.00	27679	4.8	100.00	484	0.3
26	TM	F	H	100.00	806	0.3	100.00	406	0.2
27	TM	I	L	100.00	368	0.2	100.00	261	0.1
28	TM	I	M	100.00	881	0.3	100.00	371	0.2
29	TM	I	H	100.00	951	0.2	100.00	607	0.2
30	TMM	F	H	96.00	9995	6.2	100.00	4519 (624)	2.4 (0.8)
31	TMM	I	L	100.00	1028	0.8	100.00	1568	0.8
32	TMM	I	M	99.50	26431	15.2	100.00	5148 (5148)	2.6 (2.6)
33	TMM	I	H	92.00	117416	73.6	100.00	29420 (16395)	9.6 (5.2)
34	TS	F	H	88.46	886	0.8	88.46	880	0.8
35	TS	I	L	100.00	20029	8.8	100.00	18162	4.5
36	TS	I	M	88.46	1164	0.9	88.46	1132	0.8
37	TS	I	H	88.46	1778	0.9	88.85	6079 (2176)	40.4 (0.9)
38	TW	F	H	11.54	4	0.0	96.15	3693 (4)	2.5 (0.0)
39	TW	I	M	92.31	3561	2.3	96.92	17487 (1531)	60.5 (1.5)
40	TW	I	H	11.54	4	0.0	96.15	2969 (4)	2.1 (0.0)

Table 3: Results obtained in the experimentation comparing TCSS and TCSS-LS: % coverage achieved, test cases generated and time consumed. Each instance is defined by a benchmark executed with a specific input range (L for 8 bits, M for 16 bits and H for 32 bits) and a specific type of input variable (C for char, F for float and I for integer). The values in brackets are the test cases generated and the time consumed by TCSS-LS to reach the same coverage obtained by TCSS.

Table 4: Results obtained by TCSS-LS and other generators

Instance	Benchmark	Type	Range	TCSS-LS			Other approaches			
				% Cov	Test Cases	Time (sec)	Generator	% Cov	Test Cases	Time (sec)
1	AF	C	7 bits	100	13509	9.81	EDA [53]	100	7685	
							EDA-SS [53]	91.33	570306	
							SS [53]	80	1504311	
2	AF	C	8 bits	100	17133	8.28	GA [62]	100	35263	
3	BM	F	8 bits	100	160	0.17	GA [51]	100	2900	
4	CB	I	10 bits	100	4650	12.15	EDA [53]	100	11930	
							EDA-SS [53]	100	24154	
							SS [53]	100	38984	
5	CB	I	16 bits	100	5384	15.1	GA [62]	100	28978	
6	CD	I	32 bits	100	561	0.68	dES [2]	97.88	2188	7.47
							panES [2]	97.73	2586	23.97
							dGa [2]	90.91	304	10.43
							panGA [2]	90.91	75	28.53
7	LR	F	100	100	8977	2.69	TSGen [20]	100	29191	58.86
8	LR	F	1000	100	6523	2.27	TSGen [20]	100	24606	43.91
9	LR	F	100000	100	9922	2.93	TSGen [20]	100	33303	60.69
10	LR	I	16 bits	100	3538	1.25	TSGen [18]	100	27312	
11	LR	I	24 bits	100	5000	1.53	TSGen [18]	100	65091	
12	LR	I	32 bits	100	5817	1.69	TSGen [18]	100	177967	
13	ND	I	8 bits	100	76271	269.1	TSGen [20]	100	25765	84.27
14	ND	I	16 bits	100	115379	361.6	TSGen [20]	100	28081	96.63
15	ND	I	32 bits	99.04	159914	164.5	TSGen [20]	100	65317	251.3
16	QF	I	32 bits	100	50	0.02	GA [45]	100	26700	
17	QFS	F	100	100	1855	0.19	TSGen [17]	100	746	0.39
18	QFS	F	500	100	2296	0.24	TSGen [17]	100	2063	1.14
19	QFS	F	1000	100	2681	0.28	TSGen [17]	100	5150	2.91
20	QFS	F	2000	100	2680	0.28	TSGen [17]	100	12664	8.76
21	QFS	F	10000	100	2675	0.28	TSGen [17]	88.33	15969	11.14
22	QFS	I	100	100	766	0.11	GA [30]	100	1200	
							GA [54]	100	1373	0.46
23	QFS	I	200	100	1833	0.20	GA [54]	100	1975	0.66
24	QFS	I	400	100	2961	0.30	GA [54]	100	2642	0.88
25	QFS	I	800	100	2073	0.22	GA [54]	100	3408	1.29
26	QFS	I	1000	100	2545	R	GA [54]	100	4247	1.47
27	RS	I	16 bits	100	482	0.28	SS [53]	100	141	
							EDA-SS [53]	100	2197	
							EDA [53]	100	2360	
28	TM	F	100	100	404	0.16	TSGen [20]	100	697	0.39
29	TM	F	1000	100	402	0.15	TSGen [20]	100	819	0.47
30	TM	F	100000	100	405	0.15	TSGen [20]	100	1435	0.86
31	TM	I	8 bits	100	260	0.13	TSGen [20]	100	217	0.11
32	TM	I	16 bits	100	370	0.15	TSGen [20]	100	738	0.44
33	TM	I	32 bits	100	607	0.15	TSGen [20]	100	19552	21.42
34	TMM	I	9 bits	100	705	0.84	GA[51]	100	13200	
35	TMM	I	10 bits	100	1004	0.93	EDA-SS [53]	100	3439	
							EDA [53]	100	3875	
							SS [53]	100	27007	
36	TMM	I	32 bits	100	29419	9.64	GA [45]	100	97300	
37	TS	I	100	100	11881	3.41	TSGen [17]	100	12601	30.2
							GA [54]	100	17789	8.4
							GA [30]	100	18800	
							GA-uniformcrossover [29]	100	33696	
							GA-doublecrossover [29]	100	40244	
							GA-singlecrossover [29]	100	43824	
38	TS	I	200	100	30578	7.23	GA [54]	100	48490	23.5

39	TS	I	400	100	56911	12.43	GA [54]	100	126943	60.4
40	TS	I	10 bits	100	78987	16.78	SS [53]	100	14188	
							EDA-SS [53]	100	14259	
							EDA [53]	100	22720	
41	TW	F	16 bits	96.15	2748	2.16	GA [62]	100	42086	
42	TW	F	98304	96.15	2170	1.78	EDA-SS [53]	100	4250	
							EDA [53]	100	6200	
							SS [53]	100	19357	
43	TW	I	15 bits	96.15	2698	1.89	SS [53]	100	1108	
							EDA-SS [53]	100	3272	
							EDA [53]	100	6150	
44	TW	I	16 bits	96.92	17487	60.5	GA [62]	100	16915	

Table 4: Results obtained in the experimentation comparing TCSS-LS and other generators: % coverage achieved, test cases generated and time consumed. Each instance is defined by a benchmark executed with a specific input range and a specific type of input variable (C for char, F for float and I for integer).

Figure 1: Basic scheme of Scatter Search

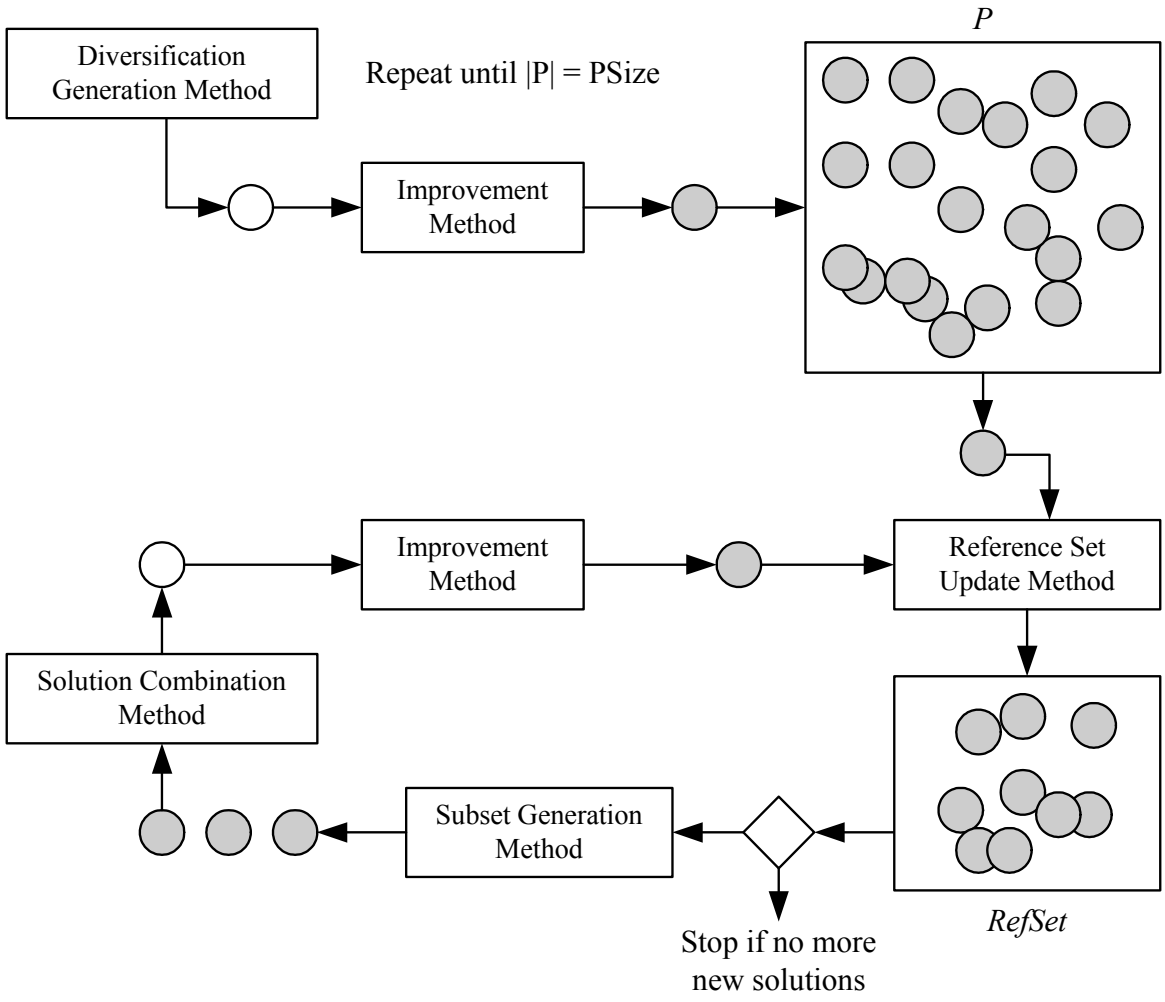

Figure 2: Control flow graph example

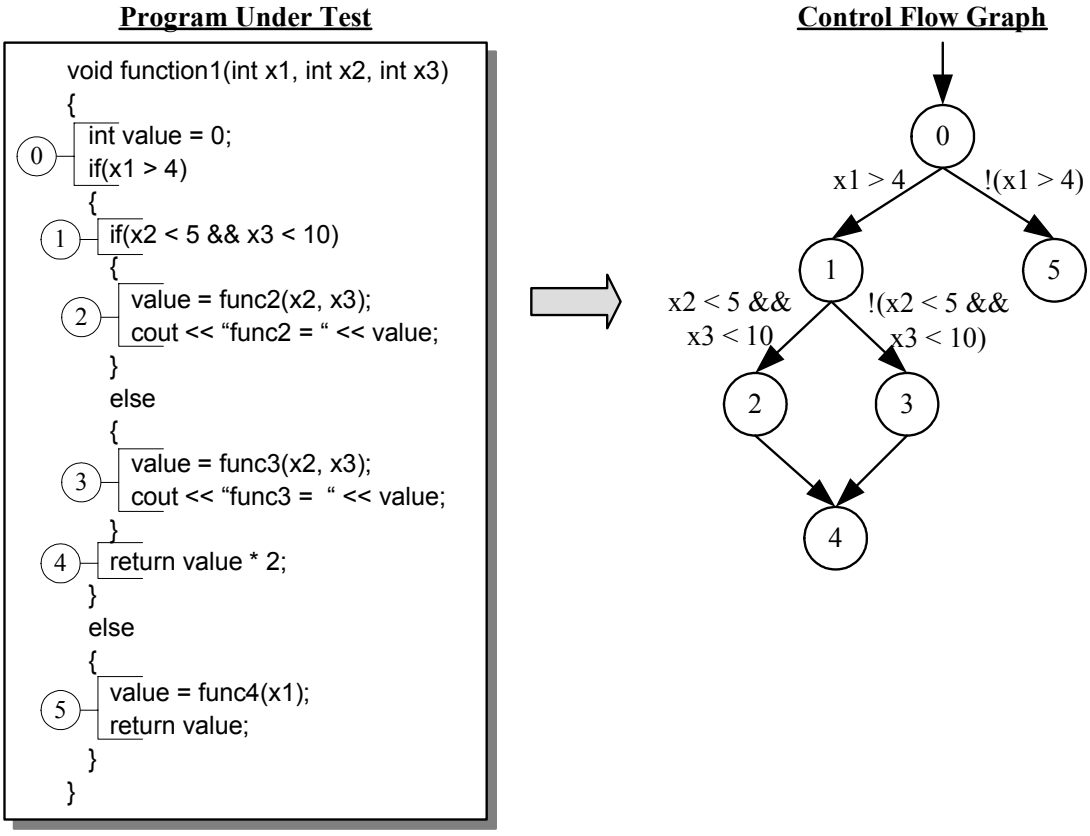

Figure 3: TCSS control flow graph

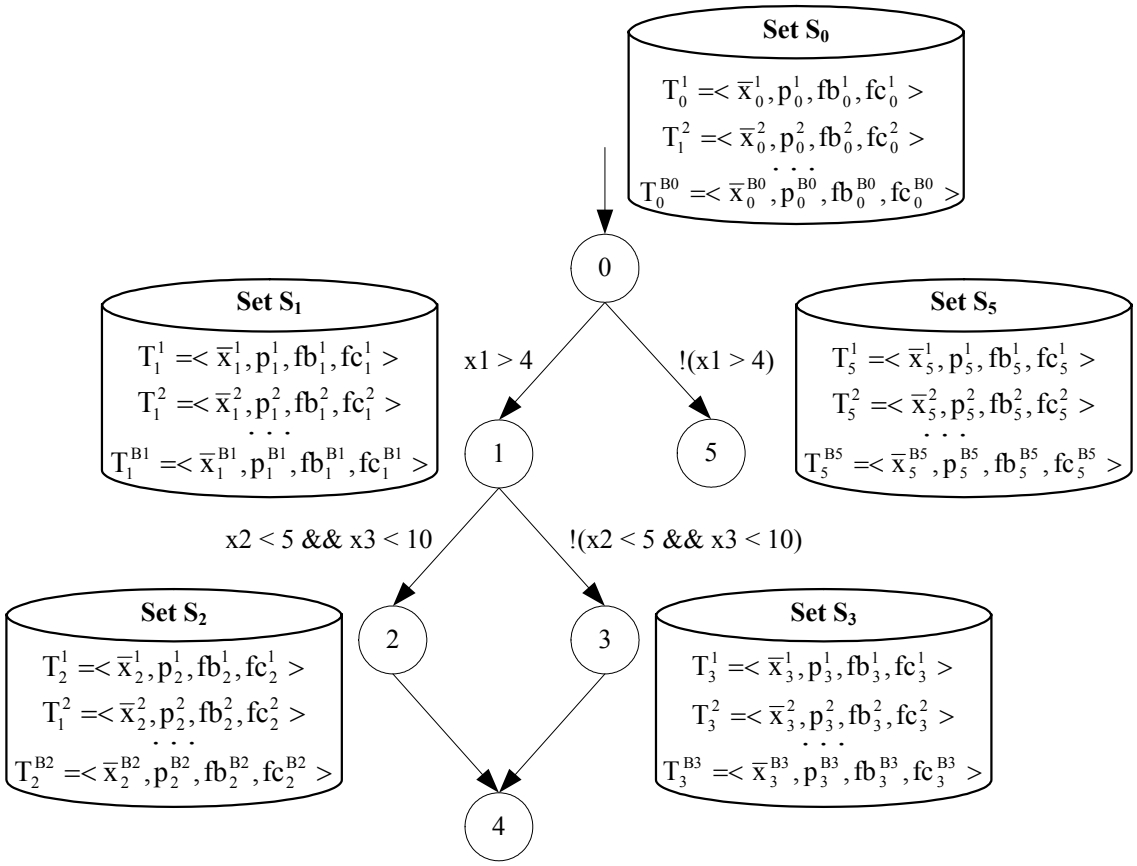

Figure 4: TCSS scheme

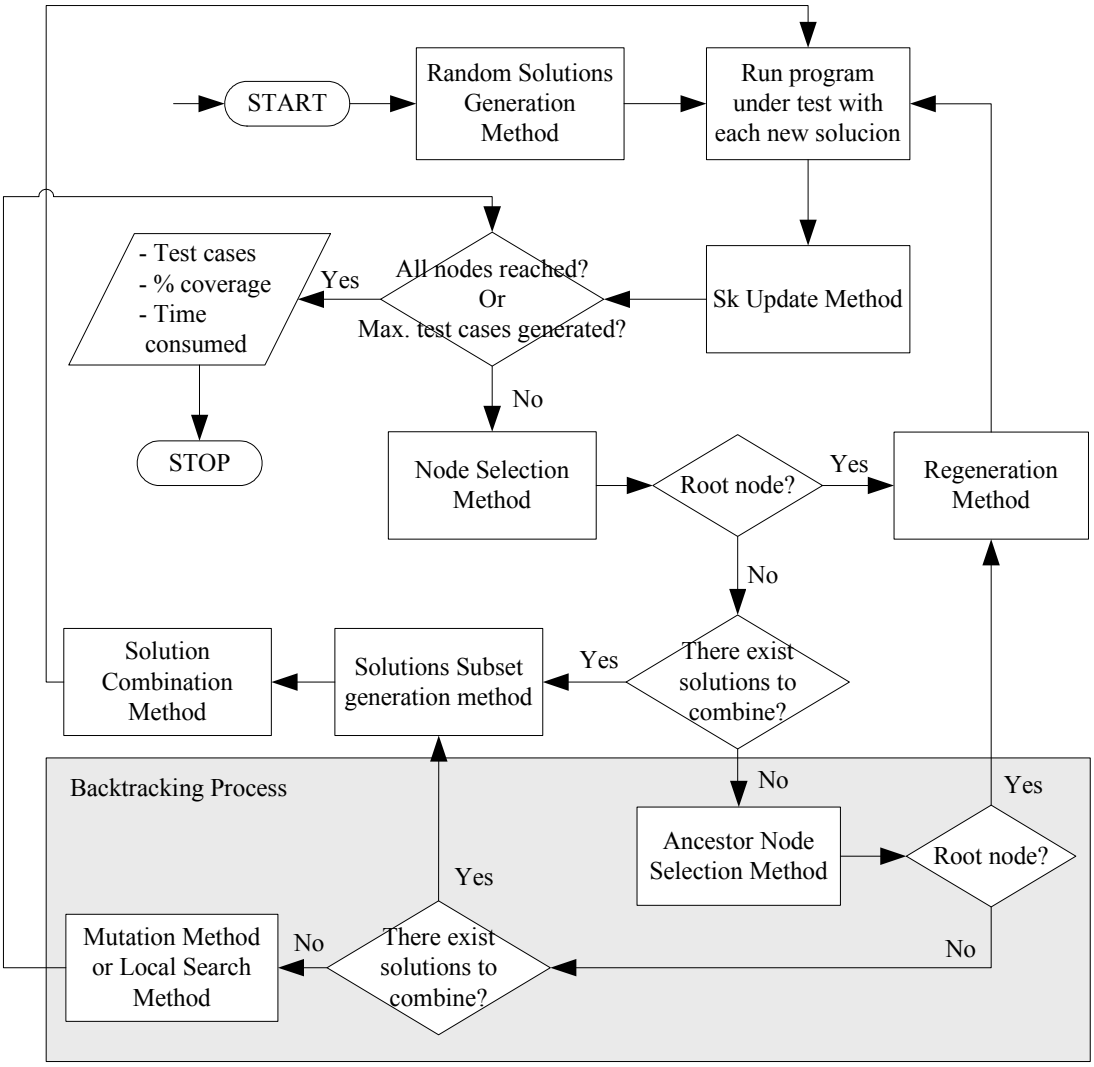

Figure 5: Ratios between TCSS and TCSS-LS

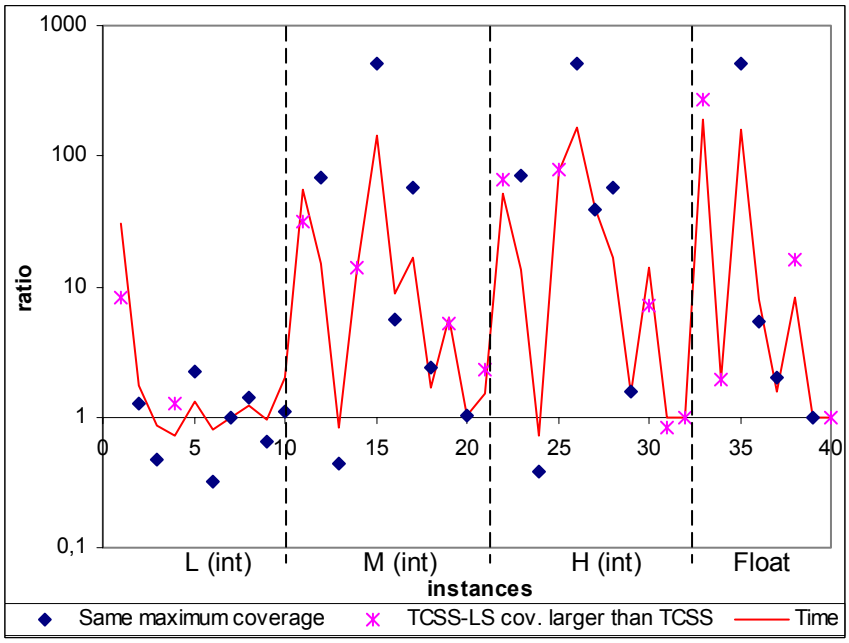

Figure 6: Ratios between TCSS-LS and other generators