# A tabu search algorithm for structural Software Testing

**Eugenia Díaz[*] [(1)], Javier Tuya[(1)], Raquel Blanco[(1)], José Javier Dolado[(2)]**
(1) Department of Computer Science, University of Oviedo
Campus de Viesques, Gijón, Asturias, 33204 Spain
(2) Departmet of Computer Science and Languages
University of the Basque Country, Spain

**Abstract**

This paper presents a tabu search metaheuristic algorithm for the automatic generation of structural software tests. It is a novel work since tabu search is applied to the automation of the test generation task, whereas previous works have used other techniques such as genetic algorithms. The developed test generator has a cost function for intensifying the search and another for diversifying the search that is used when the intensification is not successful. It also combines the use of memory with a backtracking process to avoid getting stuck in local minima. Evaluation of the generator was performed using complex programs under test and large ranges for input variables. Results show that the developed generator is both effective and efficient.

*Keywords: Software testing; Structural testing; Automatic test generation; Tabu search*

## 1 Introduction

The application of Metaheuristic Algorithms to solve problems in Software Engineering was proposed by the SEMINAL project (Software Engineering using Metaheuristic INnovative Algorithms) and is detailed in [1]. One of these applications is the test generation process of Software Testing.

Software Testing consists of a set of activities conducted with the aim of finding errors in software. As the number of test cases[1] needed for fully testing a software program is a huge number [2], in practice, it is impossible to achieve a fully tested program. It has been estimated that software testing entails 50 percent of software development [3]. This cost can be significantly reduced with the automation of test generation. Among the different approaches used for the automation of this process, we may distinguish

[*] Corresponding author: madiaz@uniovi.es
Ph: +34 985 182497
FAX: +34 985 181986

[1] A test case is an ordered pair of test inputs and expected outputs (oracle).

*Final preprint version published in Computers and Operations Research.* 35(10) 3049-3392, 2008.

between specification-oriented approaches (or black-box testing), which generate the test cases from the program specification, and implementation-oriented approaches (or white-box testing), which generate the test cases from the code of the program under test.

Test cases have to be generated according to the test adequacy criterion [4], which '*is considered to be a stopping rule that determines whether sufficient testing has been done … and provides measurements of test quality*'. Some of these criteria are the structural criteria that specify testing requirements in terms of the coverage of a particular set of elements of the program under test or its specification. Previous work on automatic test generation for structural criteria can be divided into static methods (as for example [5]) that generate the tests[2] without executing the program under test, and dynamic methods (as for example [6]) that carry out a direct search of the tests through the execution of the program, which has to be previously instrumented. The most recent dynamic methods for automatic test generation use the metaheuristic search techniques called genetic algorithms and simulated annealing where the testing problem is treated as a search or optimization problem. One of these metaheuristics, genetic algorithms [7], is the most widely used technique.

The first work that suggested the use of genetic algorithms for the structural test generation was [8] in 1992. However, it is from 1995 on when genetic algorithms began to be used frequently for the automatic generation of tests. [9] used genetic algorithms for the search of tests that cover a path or a set of paths of the program under test (data-flow criteria), and a genetic generator was developed in [10] for the branch coverage criterion. Subsequently, the doctoral thesis [11] investigated how the variations in the genetic parameters (mutations, crossover, fitness function...) influence the results obtained by a genetic generator for achieving branch coverage and loop coverage. Some of these results are also published in [12] and [13]. On the other hand, TGen [14] was developed for the statement coverage and branch coverage criteria. TGen introduces the variation of using the control dependence graph [15] instead of the control flow graph (which will be detailed in Section 2) used by other genetic generators. [16] proposed a fitness function called SIMILARITY that was used by their system to search tests that cover a selected path of the program under test.

---

[2] We refer to test as a set of test inputs of a test case since, in structural testing, the goal is concentrated in the generation of test inputs (this way, the oracle problem is not considered).

The aforementioned works use binary encoding limited to a representation that allows only integer types, in contrast with [17, 18, 19, 20], which are capable of generating tests for programs with input variables of a real type. Wegener et al. [17] showed an evolutionary test environment that performs automatic test data generation for most structural test methods, presenting its results for the branch coverage criterion. Michael et al. [18] reported a system, called GADGET, for the condition/decision coverage criterion. GADGET has two different implementations: standard [7], which uses a binary representation, and differential [21], which allows a representation defined by the user to be used. In the work [19] genetic algorithms are used for the search of tests that reach a goal path. This generator accepts real data type but its representation is binary. As in the aforementioned paper, the system described in [20] also has the goal of path testing and uses binary encoding.

However, there are few studies based on simulated annealing for automatic test generation with structural criteria. On the one hand, there is the work presented in [22] for the branch coverage criterion and on the other the work presented in [20], which, like their genetic generator, is intended for path testing.

Another metaheuristic technique that can be applied to automatic test generation is tabu search [23]. There are a great variety of real-world problems that can be solved by tabu search, such as job shop scheduling [24], multiprocessor task scheduling [25], vehicle routing problems [26], graph coloring [27] and many other combinatorial optimization problems [28, 29]. However, although tabu search is cited in several studies in software testing, such as [18] or [22], no data have been reported with the exception of our initial study [30]. This is the reason why we detail in this paper how tabu search can be applied to the automatic generation of tests to obtain branch coverage and present the results obtained.

In relation to our initial work [30], this paper includes important novelties as for example the use of two cost functions (one is used to intensify the search and the other to diversify the search), the improvement of the process for the generation of neighbour candidates (which includes four cases for the automatic adjustment of the steps) and a more elaborated backtracking process that incorporates three stages of performance. Moreover, in this paper we present the results of the application of Tabu Search to test generation together with a summary of previous related work with other metaheuristic techniques and a discussion about their published results.

The rest of the paper is organized as follows. The next section describes the problem of the automatic generation of tests for the structural branch coverage criterion and the tool scheme that we have developed. Section 3 details our test generator based in Tabu Search. In Section 4 the results published in previous works are summarized and our results are presented and analysed. Finally, Section 5 includes the conclusions of this paper.

## 2 Problem overview

The combination of program-oriented approaches with structural criteria produces testing methods called Program-Based Structural Testing methods. For these methods, most adequacy criteria are based on the flow graph model of the program structure. A flow graph (or control flow graph) [31] is a directed graph *G=(N, E, s, e)*, where N is a set of nodes, E is a set of directed edges (arcs) $a_{ij}$ between nodes of the form $(n_i,n_j)$, being $n_i,n_j \hat{I} N$ $(E \acute{I} N \times N)$, $s$ $(s \hat{I} N)$ is the initial node and e $(e \hat{I} N)$ is the final node of the graph. Each node $n \in N$ represents a linear sequence of computations for the program under test. Each arc $a_{ij}$ represents the transfer of the execution control of the program from the node $n_i$ to the node $n_j$ when the associated arc condition (or decision[3]) is true.

The branch coverage criterion (also known as the decision coverage criterion) requires all control transfers (decisions) to be exercised during testing. The percentage of decisions executed is a measurement of test adequacy. Thus, for the branch coverage criterion, an automatic test generator is totally effective if it generates a set of test inputs T that covers all feasible decisions. Moreover, the automatic generation of T must be carried out consuming a reasonable time (efficiency), since the automatic test generation is carried out in order to reduce the manual time taken up by this task, thus reducing the final cost in the testing process.

Tabu search, on the other hand, is a metaheuristic search technique based on the use of historical information about a neighbourhood search aimed at helping the search to overcome local optima. The general algorithm of tabu search is based on that of the next k neighbours while maintaining memory that avoids repeating the search in the same area (tabu) of the solution space. The details about tabu search can be consulted in [23, 29, 32].

---

[3] A decision can be formed by a simple condition or composed of several conditions

The application of tabu search to automatic test generation requires the creation of an automatic generator in which the problem of test generation must be represented in terms of tabu search elements (solutions, cost function and many other elements). In this respect, we have developed a test generator based on tabu search, called TSGen (**T**abu **S**earch **Gen**erator), that uses the general scheme of tabu search to generate a set of coverage tests and which has several additional elements (as for example a backtracking process) for improving its results even further. TSGen will be detailed in the next section.

Like dynamic test generators, TSGen generates the tests from an instrumented version of the program under test. As manual program instrumentation is a heavily time-consuming task and may introduce errors with the consequent errors in test generation, we have created a tool that automatically carries out the entire process related to test generation and in which TSGen is integrated. The scheme of this tool appears in Figure 1.

Our tool uses the C/C++ source code of the program under test and automatically generates the tests needed to achieve branch coverage. To do so, it has three modules:

- A parser/control flow graph generator that creates the needed control flow graph (called CFG) from the source code under test.
- A parser/instrumenter that creates the instrumented source code from the CFG and the source code under test.
- TSGen, which generates tests and obtains the results from the instrumented source code and the CFG.

The CFG stores which branches have been covered together with a great deal of other pertinent information, such as for example the best solutions found. In our CFG, each program branch is represented by a node that stores the branch decision (among many other data). The program loops are treated as branches and represented as nodes that store the loop decisions. The nodes representing decisions are the CFG decision nodes. The CFG also has a root node (initial node) and non-decision nodes which represent pieces of unconditional code. For instance, Figure 2 depicts a simple example of a program and its corresponding CFG. It has seven nodes: the root node (node 0), two non-decision nodes (node 4 and 6) and four decision nodes (nodes 1, 2, 3 and 5) for which TSGen has to find the tests that cover them.

Furthermore, although branch coverage is treated in this paper, the modularity of our tool allows other coverage criteria to be used without carrying out too many changes. In this respect, we have also implemented condition/decision coverage by modifying the parser modules to divide the decisions in conditions and without changes in TSGen.

## 3 Tabu search for automatic test generation

In this section, we present how we have applied tabu search to software testing. The developed generator, TSGen, has the goal of covering all the branches of the program under test, i.e. to cover all the nodes of its control flow graph CFG (detailed in the previous section).

TSGen generates tests (partial solutions) and executes them as input for the program under test. A test $\bar{x}$ is formed by a vector (or tuple) of given values ($v_1$, $v_2$, …, $v_n$) for the input variables ($x_1$, $x_2$,..., $x_n$) of the program under test. The set of values for a variable $x_i$ is determined by its type (integer, float or character).

The general TSGen algorithm appears in Figure 3.

TSGen generates tests based on the test that is the Current Solution (CS). Initially, the Current Solution is a random test, but, inside the loop (see Figure 3), TSGen selects it according to which subgoal node has to be covered. The subgoal node selection process will be explained in subsection 3.3.

Using the Current Solution, TSGen generates a set of neighbouring test candidates (in a way that will be explained in subsection 3.4). When a test is generated, TSGen checks whether it is a tabu test. A test is tabu if it is stored in the TSGen memory. In short, TSGen has a memory formed by two tabu lists: the short-term tabu list (ST) and the long-term tabu list (LT), which will be detailed in subsection 3.5. If a generated test is not tabu, the instrumented program under test is executed to check which branches (nodes) it has covered and the cost (calculated as detailed in subsection 3.2) incurred by said test. However, if a generated test is tabu, it will be rejected. During the search process, the best solutions found are stored together with their costs in the CFG. Thus, when an executed test has a lower cost in a CFG node than the current cost stored in that node, that test is stored as the best solution for that node.

The program under test could have unfeasible or very difficult branches to be covered. For this reason, TSGen includes a backtracking process that will be explained in

subsection 3.6. In the backtracking process, TSGen will reject the Current Solution and store it in its LT memory. As a result of the backtracking process, TSGen will regenerate the search to the subgoal node or will mark the subgoal node as unfeasible and start the search to reach a new non-covered subgoal node.

### 3.1 Goal and final solution of TSGen

The goal that TSGen has to achieve is that of generating the tests that obtain the maximum branch coverage for the program under test. This value is calculated as:

$$\%Max = 100 - \frac{NumberOfUnfeasibleBranches}{NumberOfTotalBranches} * 100 \quad (1)$$

A branch is unfeasible if there is no test to cover it. Therefore, if there are no unfeasible branches, the maximum branch coverage will be 100%. However, the maximum branch coverage is unknown before testing the program. For this reason, we established the following as the stopping criterion for TSGen: when all the branches have been reached or when a maximum number of iterations (MAXIT) of TSGen has been surpassed.

As the Final Solution of the problem of branch coverage, when it finishes, TSGen shows the final branch coverage achieved, the time consumed, the branches (if these exist) that have not been covered and each best test that is needed to reach the final branch coverage.

### 3.2 Cost functions

A cost function (or fitness function) measures how good a solution is in relation to achieving the search goal. The use of a good cost function is fundamental for the test generator to work correctly and its definition depends on how the search problem is addressed.

Previous studies on the use of metaheuristic search techniques for automatic test generation employ only one cost function to measure the distance of a test to the goal. TSGen uses a new approach that consists in distinguishing two kinds of costs with different meanings:

- The cost for a test $\bar{x}$ when it does not reach a node $n_j$ but it does reach its parent node $n_i$. This cost is used to intensify the search and is calculated by means of the cost function called $f_{nj}^{ni}(\bar{x})$, as will be explained in subsection 3.2.1.

- The cost for a test $\bar{x}$ when it reaches a node $n_i$. This cost is used to diversify the search and is calculated by means of the cost function called $fp_{ni}(\bar{x})$ (parent cost), as will be explained in subsection 3.2.2.

For each executed test $\bar{x}$, TSGen calculates both costs ($f_{nj}^{ni}(\bar{x})$ and $fp_{ni}(\bar{x})$). The best tests found are stored (together with their cost) in the CFG. Specifically, a CFG node $n_i$ stores the Best Known Solution for trying to reach its child node $n_j$ ($BKS_{nj}^{ni}$) and the Best Solution that reaches $n_i$ ($BS_{ni}$). These best solutions are stored for all the CFG nodes with the exception of the root node, which does not store the BS since all tests reach it unconditionally, and the leaves nodes, which do not store the BKS since they have no child nodes. Figure 4 depicts an example of the best tests stored for a node $n_i$ that has two child nodes, $n_j$ and $n_k$, in which $Cond_{ni}$ is the decision that has to be true to reach $n_i$ and $Cond_{nj}^{ni}$ (or simply $Cond_{nj}$) and $Cond_{nk}^{ni}$ (or simply $Cond_{nk}$) are the decisions that have to be true, respectively, to reach $n_j$ and $n_k$ from $n_i$.

During the search process, TSGen will frequently use $BKS_{nsgoal}^{ni}$ as the Current Solution to try to cover the subgoal node $n_{sgoal}$. By means of this solution, the search will be intensified in a 'good' region. However, if this search is not successful ($n_{sgoal}$ is not covered in a number of iterations), TSGen will use $BS_{ni}$ as the Current Solution in order to diversify the search towards unexplored regions. Anyway, when TSGen finds a best test for any node, it will be stored in the CFG carrying out, this way, a 'parallel search' for all the branches.

### 3.2.1 Cost function $f_{nj}^{ni}(\bar{x})$

The cost function $f_{nj}^{ni}(\bar{x})$ is used to evaluate those tests which, although they reach the node $n_i$, make the decision $Cond_{nj}^{ni}$ false (and do not reach the node $n_j$). The definition of $f_{nj}^{ni}(\bar{x})$ is shown in Table 1 (the constant $\sigma$ is used to avoid obtaining a zero cost in the inequality decisions). This definition is based on those of [6, 33]. It assigns a lower cost to those tests that are nearer to making the branch decision $Cond_{nj}^{ni}$ (=$Cond_{nj}$) true, although, in contrast to these, our cost function will always calculate a value above zero.

The goal of TSGen will be to minimize $f_{nj}^{ni}(\bar{x})$. This minimization will involve an intensification of the search in a region of the input domain where there are good tests for trying to reach $n_j$ from $n_i$.

### 3.2.2 Cost function $fp_{ni}(\bar{x})$

The cost function $fp_{ni}(\bar{x})$ is used to evaluate the cost of a test $\bar{x}$ in each node $n_i$ reached by it (i.e. TSGen uses $fp_{ni}(\bar{x})$ when $\bar{x}$ makes $Cond_{ni}$ true).

During the search process, the evaluation using this cost function will determine the test that is stored in the CFG as $BS_{ni}$. This solution is only used as the Current Solution when the solution $BKS_{nsgoal}^{ni}$ has failed as the Current Solution. The most probable reason for this situation is that the intensification that $BKS_{nsgoal}^{ni}$ produced in one region $R_{ip}$ of the domain of $n_i$ ($D_i$), $D_i$ being a discontinuous domain, occurred in the erroneous region. Thus, if $D_i$ is defined as the union of some regions $D_i=R_{i1}\cup R_{i2}\cup...\cup R_{iw}$ and the domain of the subgoal node is $D_{nsgoal}\subset R_{im}$, it happens that the search is trying to reach $D_{nsgoal}$ from a test that is in $R_{ip}$, being $p\in[1,w]$ and $p\neq m$. When this occurs, it is in the interest of TSGen to escape from region $R_{ip}$ and to try and reach another neighbouring region that will diversify the search. The best tests will hence be those in region boundaries and $fp_{ni}(\bar{x})$ has accordingly been designed to assign them a lower cost.

Table 2 displays the definition of $fp_{ni}(\bar{x})$.

The $BS_{ni}$ solutions will be used during a backtracking process to try to diversify the search in other regions. When TSGen finishes, the Final Solution is made up of tests that are stored as the $BS_{ni}$ in the CFG. Since these tests are the ones found nearest to the domain boundaries, they will be more likely to find functional errors in the program under test than another set of coverage tests that are located more inside the domains.

## 3.3 Selection of the subgoal node

The subgoal node ($n_{sgoal}$) is the CFG node that TSGen has to cover during an iteration. The final goal of TSGen is to cover all the nodes of the CFG. During the search, however, this goal is divided in subgoals bearing in mind that TSGen has to calculate a set of neighbouring candidates based on the CS. The idea here is that if one test in the CFG has covered the parent node but not the child node, a neighbouring test can be found that reaches the child node using the test that covers its parent node and which is the best one found up until then. This idea is based on the chaining approach [34], which consists in the concept that the parts of a program can be treated as subgoals, where each subgoal is solved using function minimization search techniques.

In each iteration, TSGen selects as $n_{sgoal}$ the following CFG node without cover in preorder and whose parent has already been covered by a previously generated test.

Once $n_{sgoal}$ is selected, TSGen sets up the CS with the $BKS_{nsgoal}^{nparent}$ (or with the $BS_{nparent}$ if there is a backtracking process; see subsection 3.6). If the subgoal node has more than one parent node, the CS will be the $BKS_{nsgoal}^{nparent}$ with a lower cost (and not tabu). For example, let us suppose the CFG in Figure 5 and that there is not a backtracking process. The only nodes that have not yet been covered are 7 and 10. In this situation, TSGen selects node 7 as the subgoal node. As node 4 and node 6 are not decision nodes, node 7 has three parent decision nodes: 2, 3 and 5. Therefore, TSGen selects the $BKS_7^{nparent}$ with the lowest cost and not tabu as the CS. Assuming none of the $BKS_7^{nparent}$ is tabu, if $f_7^3(BKS_7^3)<f_7^2(BKS_7^2)<f_7^5(BKS_7^5)$, then the CS will be $BKS_7^3$.

### 3.4 Generation of neighbouring candidates

TSGen generates 4*n neighbouring candidates of the CS, n being the number of input variables of the program under test. In short, the technique consists in generating two near-neighbour tests and two neighbour tests further from the CS. That is to say, if the CS is ($v_1$, $v_2$, …,$v_n$), TSGen maintains the same values for all $v_k$ that satisfy $k \neq i$ and generates four new values for $v_i$:

$$(i)\ v_k' = v_k + s(l) \quad (ii)\ v_k'' = v_k - s(l) \quad (iii)\ v_k''' = v_k + s(m) \quad (iv)\ v_k^{iV} = v_k - s(m)$$

where $s(\lambda)$ is a short step length and $s(\mu)$ is a long step length ($\lambda$ and $\mu$ are TSGen parameters).

The values for $s(\lambda)$ and $s(\mu)$ are dependent on the type and range of the input variables and although they are fixed respectively to the $\lambda$ and $\mu$ values at the beginning of TSGen, they change during execution, taking into account the evaluation of the generated tests. In this way, it will be possible to carry out larger jumps when there are appropriate neighbours in the last iteration, and a very fine adjustment of the search when the neighbours do not improve the CS cost. In short, in each iteration, TSGen applies one of the following four cases for the automatic adjustment of the steps:

1. Initializing: used to start the search in a new zone.
2. Increasing: used to extend the search in distant zones of the CS.
3. Centring: used to centre the search in a zone near to the CS.
4. Intensifying: used to intensify the search in the nearest zone to the CS. In this case, for real variables, the step lengths can take values lower than 1.

Once the steps are applied, TSGen generates 4 new tests neighbours with the variations of each $v_k$: *(i)* $(v_1,v_2,..,v_k',..,v_n)$ *(ii)* $(v_1,v_2,..,v_k'',..,v_n)$ *(iii)* $(v_1,v_2,..,v_k''',..,v_n)$ *(iv)* $(v_1,v_2,..,v_k^{iV},..,v_n)$

For each neighbour candidate $C_v$ generated, TSGen verifies whether it is a tabu test, in which case it is rejected. If, on the other hand, $C_v$ is not tabu, it is executed with the program under test and if it was the best test (BS and/or BKS solutions) for some of the reached nodes, it is stored together with its cost in the CFG.

If some of the generated candidates cover $n_{sgoal}$, TSGen will change the subgoal node in the next iteration. In any case, the processes of subgoal node selection, CS establishment and neighbour candidates generation will be repeated until TSGen verifies the stopping criterion.

### 3.5 Tabu lists: the memory of TSGen

One of the main characteristics of tabu search is that it has short-term memory and long-term memory, along with their corresponding handling strategies. Tabu memory has to improve the effectiveness of the algorithm (percentage of branch coverage achieved) without too great a loss in efficiency. In TSGen, memory is used to store tabu tests, i.e. tests that, though generated, are not used as input for executing the program under test.

In each iteration, TSGen's goal is to achieve the global minimum $\bar{x}^*$, i.e. to generate a test $\bar{x}^*$ that achieves the subgoal node. Since the neighbouring candidates are generated on the basis of the CS, the CS is stored in the short-term memory tabu list so as to avoid repeating the same search for the same subgoal node. This memory is controlled by a tabu tenure (a TSGen parameter), whose value was determined by means of experimentation with different input ranges and different programs under test. The results obtained showed that a good value for the tabu tenure is in the interval [nb, 5*nb], nb being the number of bits of the range for the input variables. Using a tabu tenure value within this interval, TSGen reaches a good final solution without consuming a long time for it. In general, we have observed in our experiments that the more complex (in terms of software coverage) the program under test is, the better the performance of TSGen when using a large value within the obtained interval.

On the other hand, it could happen that the CS were a local minima, i.e. a test starting from which TSGen does not find a new CS in its explored neighbourhood during a specific number or iterations established by the MAXCS parameter (which will be

detailed in the next subsection). The tabu algorithm should preclude getting stuck in the local minima. The long-term memory tabu list is used for this. Those $\bar{x}_n$ that are local minima during the search process are stored in this list. Once a local minimum has been found, TSGen applies a backtracking process, as will be explained in the next subsection. An example of memory management is presented in our initial work [30].

The long-term memory will store few tests. For this reason, it is maintained throughout the search process without too much effect on TSGen's performance. The short-term memory, on the other hand, could store a large number of tests and this could reduce TSGen's performance. This, however, does not occur, since this memory is controlled by the tabu tenure and it is often deleted (when the subgoal node changes). Thus, the use of memory in TSGen is not critical.

### 3.6 Backtracking process

In order to prevent TSGen spending all its iterations in an attempt to cover unfeasible nodes and/or in trying to reach a child node ($n_{sgoal}$) from a bad CS, TSGen applies a backtracking process that is determined by two parameters (the values of which may be dependent on the magnitude of the program to be tested):

- MAXCS: Maximum number of iterations to try to reach a child node with the same CS. This avoids TSGen getting stuck when it tries to reach a node using a bad CS.
- MAXNS: Maximum number of iterations that a node can be the $n_{sgoal}$. This avoids TSGen getting stuck when it tries to reach an unfeasible node because when a node has been the subgoal node in MAXNS times, TSGen marks it as a possible unfeasible and it will never be the subgoal of the search.

The backtracking process has three different stages:

- Stage 1 is applied whenever MAXCS is reached and the MAXNS value for $n_{sgoal}$ has not been reached. In this stage, the current $n_{sgoal}$ is kept. TSGen tries to reach it using a new CS. This new CS will be $BS_{nparent}$ or another $BKS_{nsgoal}^{notherparent}$ in the case of $n_{sgoal}$ having more than one parent node (previously, the CS was $BKS_{nsgoal}^{nparent}$). For example, Figure 5 depicted the selection of $BKS_7^3$ as the CS. If the search using this CS does not reach node 7 in MAXCS iterations,

TSGen applies the first stage of backtracking, obtaining the situation shown in Figure 6. In this way, $BKS_7^3$ is added to the LT tabu list and the new CS will be $BKS_7^2$ (assuming that at that instant $f_7^2(BKS_7^2)<f_7^5(BKS_7^5)$).

- Stage 2 is applied whenever Stage 1 had not success and the MAXNS value for $n_{sgoal}$ has not been reached. In this stage, the $n_{sgoal}$ is changed but TSGen does not mark it as unfeasible. The backtracking process spreads up in the CFG and TSGen tries to regenerate the solutions. If this backtracking process reaches the root node, TSGen will generate a new random test from which to continue the search.
- Stage 3 is applied whenever TSGen reaches the MAXNS value for $n_{sgoal}$. In this stage, the $n_{sgoal}$ is changed and TSGen marks it as unfeasible.

TSGen's memory is of great importance during the backtracking process, since it avoids reconsidering as better tests those that have already been tested and were rejected.

# 4 Results

This section analyzes the published data for other metaheuristic generators from previous work (subsection 4.1) and the results obtained by our tabu generator (subsection 4.2).

## 4.1 Related work and results

Table 3 summarizes the existing data from previous works that use metaheuristic techniques to automatically generate white-box tests using a structural adequacy criterion. Each row is labeled with the reference of the work for which its data are shown. The first column displays the metaheuristic technique used. The following five columns show the test adequacy criterion that is used for test generation. The structural criterion is subdivided in the control-flow criteria (in which the statement, branch, condition/decision (C/D) and loop coverage are detailed) and data-flow criteria. The columns 'Type of input' show the data type that accepts the generator for the input variables. The column 'Explicit ranges for the input' represents whether the range used

for the input variables in the experiments is explicitly stated in the published work. The following three columns summarize which results have been shown for the developed generator according to the data: number of tests generated, percentage of coverage (success) achieved and time consumed in reaching the percentage of coverage. Although many of the previous metaheuristic generators use the same technique (i.e. Genetic Algorithms), in none of them are the results obtained compared with those obtained by other metaheuristic generators. However, in many of them there is a comparison of the obtained results with those obtained by a random generator (because its algorithm is public). For this reason, the last three columns of the table indicate which results of the metaheuristic generator are compared with those of a random generator.

Observing the adequacy criteria columns, it can be seen that the most widely used criteria have been the branch coverage criterion. The type of the input variables that a test generator accepts determines whether it could be used to test a certain program or not. In this respect, the metaheuristic generators developed before the year 2001 are only able to manage integer input variables.

In order to carry out a comparison of the results of a new generator and the results of previous generators, it is necessary to know what ranges were used for the benchmark input variables in the experiments carried out for the previous generators. However, as can be observed in the column ‘Explicit ranges for the input’ in Table 3, there are five works in which the range is not explicit and one work [17] in which the range is only explicit for some of the used benchmarks. In those works in which the range is explicit, there is a tendency to use very small ranges. Thus, for example for a triangle classifier benchmark (in the specific version used in each study), the range used is ±20 in [9], ±100 in [10], ±400 in [11], ±100 in [13], 12 bits in [16] and 8 bits for integers and 16 bits for real values in [20]. This use of small ranges does not allow the behaviour of the generators to be observed in the testing of benchmarks with larger ranges, which are the most common in real programs.

For those studies in which the range is explicit, it would be possible to carry out similar experiments for a new generator, but in order to establish a comparison of the results it is necessary to know, the percentage of coverage reached and the time consumed by the previous generators (together with a measure that allows the time to be extrapolated for any other machine). As can be seen in Table 3, the percentage of coverage achieved is a

result that is shown in all previous studies. The average of several runs is usually shown (with the exception of [18). On the other hand, however, the time consumed by the generator is not shown in most of the works. The majority of these do not show any data as regards time, or they just show a time interval in which all the experiments are included (for all the benchmarks), as for example in [17], which reports that the time for evolutionary tests varied for the different test objects between 160 and 254 seconds.

The results obtained by an automatic test generator should be compared with the results obtained by other existing generators carrying out experiments under the same situations in both cases. However, as was previously stated, to do so it is necessary to know a great deal of data, which in most cases is incomplete. This lack of data would not exist if the generators to compare with were of public use, as the same experiments carried out for a new generator could always be repeated with these. The generators in Table 3 are not public, but there is a generator that can always be used to carry out comparisons: the random generator. Comparison with a random generator, as [35] proposed, allows the results of a new generator to be indirectly compared with those of previous generators. The time consumed in generating a test depends on the generator and the machine used. Since the time consumed by random testing can be obtained on any machine, showing the comparison of the time consumed by a developed generator with that of a random generator will allow the time of the developed generator to be extrapolated for any other machine.

It can be observed in Table 3 that, with the exception of [16] and [20], all previous studies have carried out some kind of comparison with the results of a random generator. However, most of them only include a comparison related to the percentage of coverage reached and the number of tests generated. That is to say, these studies do not bear in mind that the time consumed in the generation of each random test is less than the time consumed in the generation of each non-random test and, for this reason, the comparison could be biased. The time comparison exists in only two previous studies [11, 13]. In both of these, the range used for the majority of the experiments is very small and besides, in [13], the comparison is carried out for only one benchmark. This means that there are scant data (and with small ranges) concerning the efficiency of these test generators in addition to the non-existence of data about the efficiency of the remaining previous generators.

In short, it is very difficult to compare the results of a developed generator with the results obtained by the generators of previous works. As mentioned above, there are several reasons: many of the works use only integer ranges for the input variables and therefore it is not possible to carry out a comparison for the real range; the range used for the input variables is not very clear; the results and comparisons published are not always complete; and often the time compared with a random generator is not reported, this data being necessary in order to compare the efficiency. The above reasons may be the motive why previous works do not compare their results with those of other previous metaheuristic generators.

## 4.2 TSGen results

The evaluation of an automatic test generator should be carried out using programs under test that present some difficulties for a test generator (for example: if the test adequacy criterion is the branch coverage criterion, the program should have some branches that are difficult to cover) in order for them to be considered benchmarks. As was shown in [18], the difficulty of covering a branch depends on how deeply conditional clauses are nested (nesting complexity) and the number of Boolean conditions in each decision (condition complexity). In addition to these factors, it is necessary to add the type of the operators that appear in the conditions and decisions, since it will be more difficult to cover a branch with the AND operator than a branch with the OR operator and it will be more difficult to make equality conditions true than inequality conditions. The question of which benchmarks have to be used for the evaluation of a structural automatic test generator is currently open to debate, this problem being one of the study themes of the SEMINAL project. Since a set of benchmarks defined as the standard set does not exist, the previous studies that have developed metaheuristic test generators have used one or several programs that include some of the previously detailed difficult factors. However, the code of these programs is not usually published (they are described only by their functionality and/or other characteristics, such as for example their cyclomatic complexity). The limitation thus arises that these programs cannot be used for the evaluation of a new test generator.

In this section, we present the results obtained by TSGen in automatic test generation using the branch coverage criterion for a typical benchmark (the triangle classifier program) and two more complex programs (the line rectangle classifier and the number

of days between two dates), which have been specifically created with many branches that are difficult to reach.

The obtained results were compared with those of a random generator due to the existing difficulties of establishing a time comparison with the results of other previous generators, as detailed in the previous subsection. However, contrary to what occurs in many previous works, our comparison includes the results of the time consumed by both generators (thus obtaining a reference of how efficient TSGen is). Moreover, in order to carry out a more complete study of the performance of TSGen, we compared the evolution of the results using different ranges for the input variables. Comparing this evolution with that of the random generator gives us an idea of how good our generator is in those situations in which a random generator begins to lose effectiveness (due to the range increase).

For each input range, we carried out 100 experiments with the random and tabu generators. The results presented below are the average number of tests they need to generate in order to obtain a certain percentage of branch coverage and the average time consumed. Specifically, for each benchmark under test, a table is shown with the final results and a figure (such as, for example, Figure 8) that contains two graphs with the evolution of the cumulative percentage of branch coverage (vertical axis) for both generators. In the left-hand graphs, the horizontal axis represents the number of tests generated in logarithmic 10 scale, whereas in the right-hand graphs, the horizontal axis represents the time in logarithmic 10 scale.

For TSGen, in all the experiments, the initial solution was chosen at random, the tabu tenure was fixed to the value 3*nb and the initial values used as parameters for the calculation of the steps were $\lambda$=1 and for $\mu$ a value that depends on the bits of the range used:

$$\mu \begin{cases} 10^{nb/8} & nb \leq 16 \text{ bits} \\ 10^{(nb/8)+2} & nb > 16 \text{ bits} \end{cases}$$ nb being the number of bits of the range used.

All experiments were carried out on a Pentium 4-3.4 Ghz with a RAM memory of 1 Gb. In all of these, the stop condition used was that of reaching 100% branch coverage or reaching 10,000,000 generated tests.

**a) The triangle classifier program**

The programs most widely used as benchmarks are usually programs with few branches that present one or several of the difficulties to be covered for an automatic test generator. Among these programs, the most common is a program that classifies triangles on the basis of their three sides.

Although the code of this program is not always available [18, 19], when it is shown it can be observed that different versions and implementations exist. For example, the classifier benchmark of [10, 11, 12, 13] classifies the triangle in no-triangle, isosceles, equilateral, scalene and right-angled, that of [17] differs from the previous one in that it only bears in mind one type of right-angled triangle (instead of the three possible) and the benchmarks of [16] and [18] do not classify the triangle as right-angled. Moreover, the benchmarks of [16] and [18] do the same, but their code is different. Also, in [20], there is a 'strange' version and implementation which does not distinguish the no-triangle type. Therefore, there is neither a unique functionality nor code for the "triangle classifier program".

The "triangle classifier program" that is used by [2] has been used as a benchmark in this section. The control flow graph of this benchmark is displayed in Figure 7. It has three input variables (A, B, C), which may be integer or real and that represent a no-triangle or a triangle that is isosceles, equilateral or scalene.

In total, this program has 12 branches, with two decisions of AND type and its two corresponding OR decisions (the 'else' branches). The maximum nesting level is 5 and four of the deepest branches (nodes 5, 6, 9 and 11) have a condition of equality. These equalities increase the complexity for automatically finding the suitable tests, which will be even greater when A, B and C are of the real type. The more difficult branch to cover is the branch that classifies the triangle as equilateral (node 6), since it has a nesting level value of 4 and is conditioned by two AND decisions and two equality decisions.

The results obtained for this benchmark using integer and real input variables are summarized in Table 4. In this table, it can be appreciated that TSGen obtains the best results in all ranges: it only needs to use a short time to obtain full branch coverage, in contrast with the results of the random generator, which in most cases was not able to achieve 100% coverage. For example, with an integer input range of 32 bits, TSGen

needs an average of 21.4 seconds to obtain full branch coverage. On the other hand, in 298 seconds, the random generator achieved only 58.3% branch coverage.

Figure 8 depicts the evolution of the number of tests generated and the time consumed for the different integer input ranges. In the left-hand graph, it can be observed that the evolution of the accumulated percentage of coverage for TSGen in relation to the number of tests generated is very similar for all ranges: the number of tests generated to reach a certain coverage does not exponentially increase with the range of the input variables. On the other hand, the evolution for the random generator is highly range dependent: it generates few tests to reach approximately 58% branch coverage (which corresponds to covering those easy branches that have no equality conditions). However, the number of tests that it needs to go beyond 58% is much greater when the range is increased from 8 to 16 and 32 bits. The right-hand graph depicts the efficiency for both generators. For a small range, such as the 8 bit range, TSGen is faster starting from around 92% coverage, but for larger ranges, TSGen is faster starting from a lower percentage of coverage, namely 75% for a 16 bit range and 58% for the 32 bit range. In short, both graphs shows that both the effectiveness and efficiency of TSGen are not very dependent of the range of input variables, reaching 100% branch coverage for all ranges without the need for an exponential increase either in the tests generated or in the time consumed.

For real ranges (Figure 9), TSGen is again highly independent of the input range and reaches 100% coverage in around 1 second, whereas the random generator does not go beyond 58% coverage in more than 5 minutes.

**b) The line rectangle classifier**

The “Line rectangle classifier” program determines the position of a line in relation to a rectangle. It has eight real input variables, four of these (xr1, xr2, yr1, yr2) represent the

coordinates of a rectangle and the other four input variables (xl1, xl2, yl1, yl2) represent the coordinates of a line. There are four different outputs: 1- The line is completely inside the rectangle; 2- The line is completely outside the rectangle; 3- The line is partially covered by the rectangle; and 4- Error: The input values do not define a line and/or a rectangle. Besides, if the output is not erroneous, the program indicates whether the line is horizontal, vertical or inclined and, the side(s) of the rectangle that is (are) cut when the line is partially covered. To reach one of the three possible correct outputs, the program needs to check whether the line is horizontal, vertical or inclined and on the basis of this determine whether it intersects the rectangle or not. As can be observed in Figure 10, these checks add very difficult branches to be reached: with a huge nesting level (node 40 has a nesting complexity of 12) and nested inside AND, OR and/or equality decisions. In total, the number of branches of this program is 36.

This program is different from that used in [17] named “Is_line_covered_by_rectangle” because, although its code (or control flow graph) is not shown, its reported features are 24 branches and a maximum nesting level of 4 (in contrast with the value of 12 in our case).

The final results obtained are summarized in Table 5. As for the “Triangle classifier”, TSGen obtains the best results in all ranges: it only needs to use around 1 minute to obtain full branch coverage, in contrast with the results of the random generator, which did not go beyond 58.33% branch coverage in around 20 minutes. In short, the random generator does not reach the branches that are labeled in the control flow graph as nodes 5, 6, 7, 8, 9, 10, 14, 15, 16, 17, 18, 19, 20, 21 and 40. These branches have a high nesting level and/or they are preceded by complex decisions.

The evolution of both generators is depicted in the graphs in Figure 11. These show that the evolution of TSGen is practically independent of the range used for the input variables: the number of tests generated and the time consumed have similar values for all ranges in the same accumulated percentage of coverage.

**c) The number of days between two dates**

The ‘Number of days’ program calculates the number of days that there are between two input dates of the current century. It has six integer input variables: three

correspond to the initial date (day0, mon0, year0) and the other three to the final date (day1, mon1, year1). It has 86 branches and many of these are very complex. Initially, the first decision that the input dates have to make true is that the months must have a value between 1 and 12 and the years value must have a value between 2000 and 2100. Next, for those dates that make the first branch true, the program determines the maximum number of days for the input months and checks whether those values are within the month ranges. Besides, when the two dates are correct, the program checks whether the initial date is prior to the final date and interchanges them otherwise. Once there are two correctly ordered dates, the program counts the number of days between the dates taking into account whether the year is a leap year or not. All these restrictions mean that this program includes a lot of branches with equality conditions. Moreover, some of them use the remainder operator (%), which adds discontinuity to the decisions domains and therefore a greater difficulty in finding the tests that cover those branches.

The nesting level is very high for the greater part of the branches and, in combination with the AND decisions, the equality conditions and the use of the remainder operator, make this program very appropriate, because of its difficulty, to evaluate the effectiveness and efficiency of an automatic test generator for the branch coverage criterion. The control flow graph of the program is shown in Figure 12.

The results obtained are summarized in Table 6. The results show how TSGen obtains full branch coverage for all the ranges, while the random generator reaches only 1.16% branch coverage when the range is greater than 8 bits.

The evolution of both generators can be observed in the graphs in Figure 13. For the 8 bit range, TSGen always needs to generate fewer tests than the random generator (left-hand graph) and it is faster from 12% branch coverage. When the range is increased (16 and 32 bits), TSGen does not need to carry out an exponential increase for either the number of tests generated or the time spent. On the other hand, when the range is increased, the random generator is not capable of making the first decision (node 1) true and only reaches 1.16% branch coverage.

# 5 Summary and Conclusions

This paper presents TSGen, an automatic generator of software tests for a given program. Although there are a great variety of real-world problems that have been solved by tabu search, no results of its application to software testing have been published by other authors yet. In this respect, this is the first work in which the metaheuristic technique tabu search has been used to solve the problem of automatic test generation.

The representation carried out was designed to be effective and efficient: the 'Current Solution' is selected taking into account the chaining approach; the memory stores only the most significant tests; the number of neighbouring tests is dependent on the number of input variables; the values for the steps are dependent on the type and range of the input variables and are automatically adjusted during execution (taking into account the evaluation of the generated tests); and the cost function $f_{nj}^{ni}(\bar{x})$ is capable of precisely measuring how far a test is in order to make a branch decision true.

Furthermore, we have introduced a set of improvements with regard to the tabu search general scheme that increases the effectiveness and efficiency of TSGen even more: the best tests are stored in the control flow graph, carrying out a 'parallel search' for the other branches; the subgoal node is selected bearing in mind the current search situation; there is a backtracking process able to detect bad current solutions and unfeasible nodes; and another cost function ($fp_{ni}(\bar{x})$) is defined that diversifies the search when the intensification carried out with $f_{nj}^{ni}(\bar{x})$ has not been successful.

Furthermore, the representation carried out avoids TSGen getting stuck in the local minima (as could occur using Simulated Annealing when the number of local minima is larger and the cooling is not very slow). Besides, it does not use binary encoding, with the consequent saving in the consumed time that encoding/decoding would imply, as occurs in the majority of test generators based on Genetic Algorithms.

TSGen was evaluated using benchmarks that have branches for which it is very difficult to find a test that reaches them because of the existence of several of the factors that influence the complexity of reaching a branch (high nesting complexity, high condition complexity, the use of equality and remainder operators in conditions and the use of AND operators in decisions). In spite of the existence of these difficulties, TSGen achieves 100% branch coverage for all the benchmarks and consumes a reasonable time

to do so. Moreover, as was shown in the results section, the performance of TSGen is highly independent with regard to the range of input variables: it reaches 100% coverage even for 32 bit ranges without the need for an exponential increment in the consumed time (or in the number of tests generated).

The lack of published data does not allow a complete comparison with the results of other previous automatic generators to be carried out: generally all the data are not available and, besides, the range used for the input variables is usually too small to be considered significant for the evaluation of a test generator. To avoid these problems in future comparisons with TSGen, in this paper the TSGen results have been shown using large ranges (up to 32 bits) and the time consumed has been documented for the different experiments comparing it with that of a random generator, which will allow an extrapolation of its efficiency to any other machine.

## Acknowledgements

This work was funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, projects TIN2004-06689-C03-02, TIN2004-06689-C03-01 and TIN2005-24792-E.

**References** (in order of appearance)

1. Clarke J, Dolado J J, Harman M, Hierons R M, Jones B, Lumkin M, Mitchell B, Mancoridis S, Rees K, Roper M, Shepperd M. Reformulating software engineering as a search problem. IEE Proceedings - Software 2003;150(3): 161-75.
2. Myers G J. The Art of Software Testing. Ed. John Wiley & Sons, 1979.
3. Beizer B. Software Testing Techniques, 2nd. Ed. Van Nostrand Reinhold, 1990.
4. Zhu H, Hall P A V, May J H R. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 1997;29(4): 366-427.
5. DeMillo R A, Offutt A J. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering, 1991;17(9): 900-10.
6. Korel B. Automated software test data generation. IEEE Transactions on Software Engineering, 1990;16(8): 870-79.

7. Goldberg D. Genetic Algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA,1989.

8. Xanthakis S, Ellis C, Skourlas C, Gall A L, Katsikas S, Karapoulios K. Application of Genetic Algorithms to Software Testing. 5th International Conference on Software Engineering, 1992. p. 625-36.

9. Watkins A L. The automatic generation of test data using genetic algorithms. 4th Software Quality Conference, 1995; vol. 2:300-09.

10. Jones B, Sthamer H, Yang X, Eyres D. The automatic generation of software test data sets using adaptive search techniques. 3rd International Conference on Software Quality Management, 1995; vol. 2:435-44.

11. Sthamer H. The Automatic Generation of Software Test Data Using Genetic Algorithms. PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, April 1996.

12. Jones B, Sthamer H, Eyres D. Automatic Structural Testing Using Genetic Algorithms. Software Engineering Journal, 1996;11(5): 299-306.

13. Jones B, Eyres D, Sthamer H. A strategy for using genetic algorithms to automate branch and fault-based testing. Computer Journal,1998;41(2):98-107.

14. Pargas R, Harrold M J, Peck R. Test-data generation using genetic algorithms. Journal of Software Testing, Verification and Reliability, 1999;9(4): 263-82.

15. Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 1987;9(3): 319-49.

16. Lin J-C, Yeh P-L. Automatic test data generation for path testing using GAs. Information Sciences, 2001; 131: 47-64.

17. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. Information and Software Technology, 2001;43:841-54.

18. Michael C, McGraw G, Schatz M. Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, 2001;27(12): 1085-1110.

19. Siqueira P M, Jino M. Automatic test data generation for program paths using genetic algorithms. International Journal of Software Engineering and Knowledge Engineering, 2002;12(6): 691-709.

20. Mansour N, Salame M. Data generation for path testing. Software Quality Journal, 2004;12: 121-36.

21. Storn R. On the usage of differential evolution for function optimization. North American Fuzzy Information Processing Society, 1996. p. 519-23.

22. Tracey N, Clark J, Mander K, McDermid J. An Automated Framework for Structural Test-Data Generation. 13th IEEE Conference on Automated Software Engineering, 1998, p. 285-88.

23. Glover F. Tabu search: part I. ORSA Journal on Computing, 1989;1(3): 190-206.

24. Dell'Amico M, Trubian M. Applying tabu search to the job-shop scheduling problem. Annals of Operations Research 1993;41:231–52.

25. Hubscher R, Glover F. Applying tabu search with influential diversification to multiprocessor scheduling. Computers&Operations Research, 1994;21(8):877–84.

26. Gendreau M, Laporte G, Musaraganyi C, Taillard W E. A tabu search heuristic for the heterogeneous fleet vehicle routing problem. Computers&Operations Research 1999;26: 1153–73.

27. Hertz A, Werra D. Using tabu search techniques for graph coloring. Computing 1987;39: 345–51.

28. Osman IH, Laporte G. Metaheuristics: a bibliography. Annals of Operations Research 1996;63: 513–623.

29. Glover F, Laguna M. Tabu search. Dordrecht: Kluwer Academic Publishers, 1997.

30. Díaz E, Tuya J, Blanco R. Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search. 18th IEEE International Conference on Automated Software Engineering, 2003. p.310-313.

31. Deo N. Graph theory with applications to Engineering and Computer Science. Ed. Prentice Hall, 1974.

32. Glover F. Tabu search: part II. ORSA Journal on Computing 1990;2: 4–32.

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

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

35. Ince D C. The automatic generation of test data. The Computer Journal, 1987;30(1): 63-69.

## CVs

**Eugenia Díaz** received the MS and PhD degrees in Computer Science from the University of Oviedo, Spain in 1998 and 2005, respectively. Currently, she is an assistant professor in the Computing

Department at the University of Oviedo. She has authored several papers about software testing with metaheuristic techniques. Her research interests are in software engineering, especially software testing and its automation.

**Javier Tuya** received both his MS and PhD degrees in the Polytechnic School of Engineering of Gijón, Spain in 1988 and 1995, respectively. He has worked as manager in many software development projects and held the position of CIO of the University of Oviedo, Spain. Currently he is associate professor in the Computing Department at the University of Oviedo. His current research interests are in the ?eld of software management, process improvement, veri?cation & validation and testing. He has published in different international conferences and journals, and is member of professional associations such as IEEE, IEEE Computer Society and ACM.

**Raquel Blanco** is an assistant professor of Department of Computer Science at the University of Oviedo, Spain. She received the MS degree in Computer Science from the University of Oviedo, in 2002. Currently, she is a Ph.D student that is working on the automatic test case generation. Her research interests are in the area of software engineering, specially in software testing.

**Jose Javier Dolado** received both his MS and PhD in computer science from the University of the Basque Country, Spain in 1985 and 1989, respectively. He is a professor in the Department of Computer Languages and Systems at the University of the Basque Country, Spain. He was awarded three prizes for his academic achievements. His current research interests are in software measurement, empirical software engineering, dynamics of the software development process, qualitative reasoning and complex systems. His works have appeared in several refereed journals and he has served on various program committee relating to international conferences on software quality and process improvement. He is a member of the ACM, ACM Sigsoft, IEEE, and IEEE Systems, Man and Cybernetics Society.

# Figure captions



# Figures

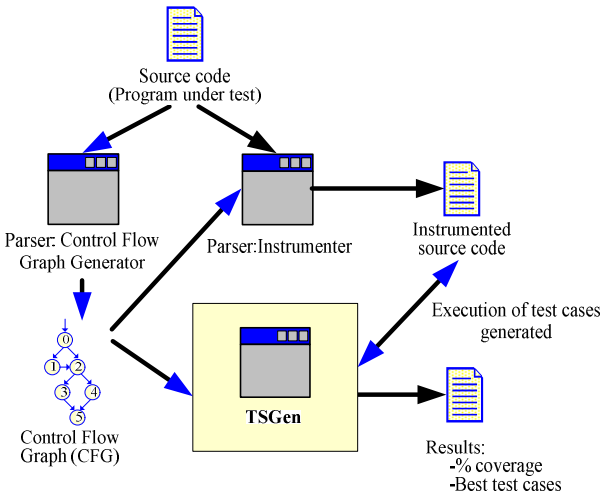
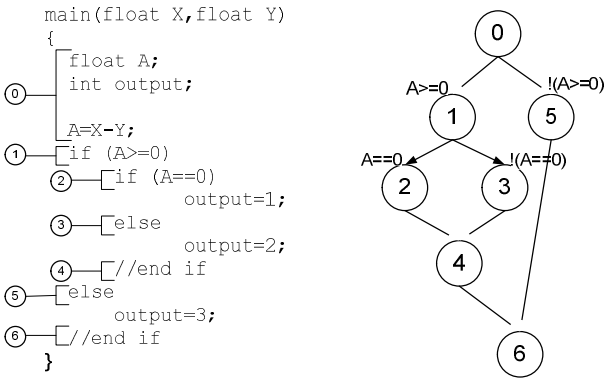



**Figure 1. Tool Scheme**



```
begin
Initialise Current Solution
Calculate the cost of Current Solution
Store Current Solution in CFG
Add Current Solution to tabu list ST
Select a subgoal node to be covered
do
    Calculate neighbourhood candidates
    Calculate the cost of candidates: each non tabu candidate is executed
    for each candidate do
        if (candidate cost in node n <CFG cost in node n) then Store candidate in CFG endif
    endfor
    if (subgoal node not covered) then Add Current Solution to tabu list ST
    else Delete tabu list ST
    endif
    Select a subgoal node to be covered
    Select Current Solution using the CFG
    if (Current Solution is depleted) then
        Add Current Solution to tabu list LT
        Apply a backtracking process: new Current Solution and maybe new subgoal node
    endif
while (NOT all nodes covered AND number of iterations<MAXIT)
end
```

**Figure 3 - TSGen general algorithm**

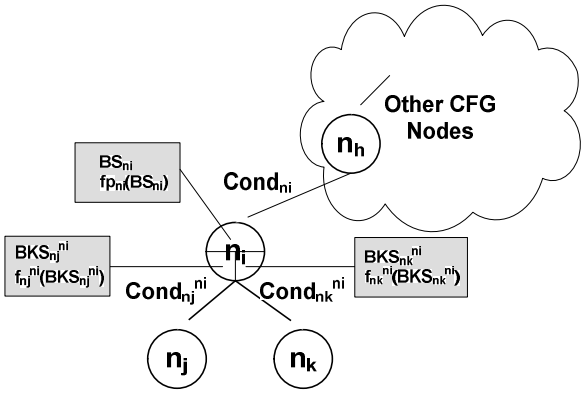



**Figure 4- Example of the best tests stored together with their cost for a CFG node $n_i$**

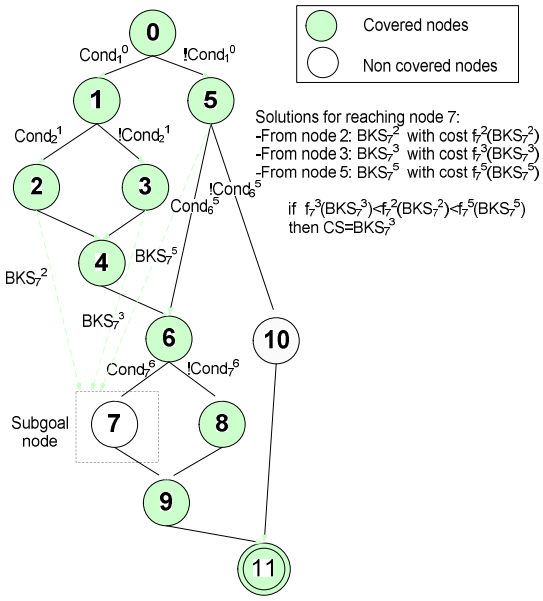
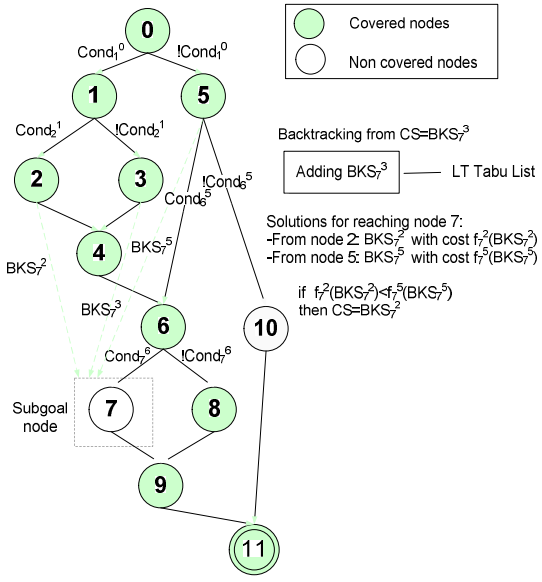
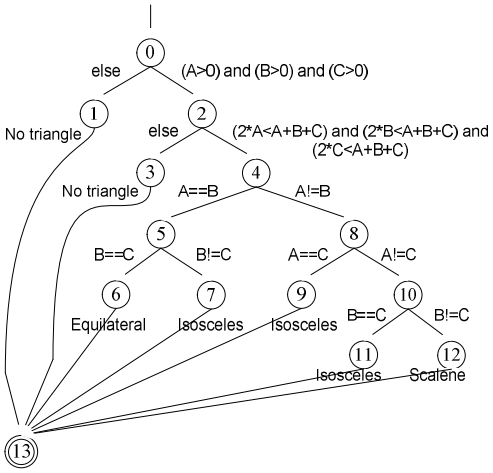



Figure 5. Example of subgoal node and CS selection





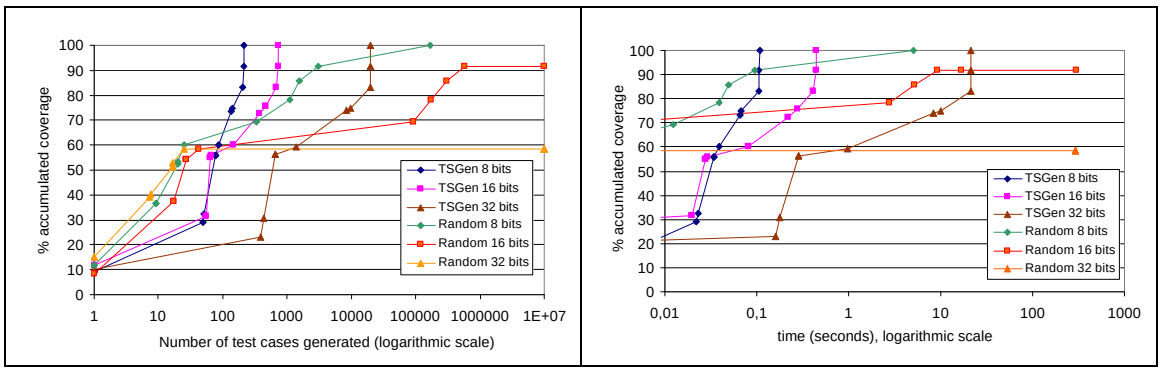




**Figure 8. Evolution of the number of tests generated and time for the "Triangle Classifier" program using integer input variables**

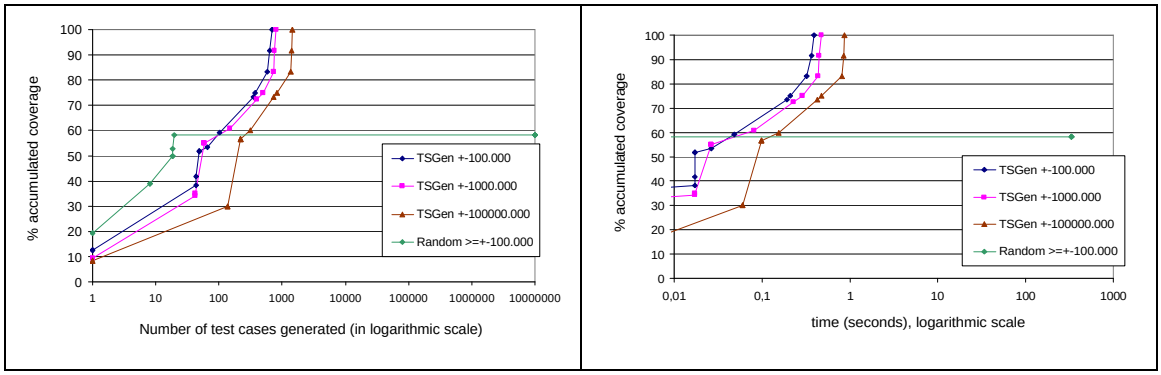



**Figure 9. Evolution of the number of tests generated and time for the "Triangle Classifier" program using real input variables**

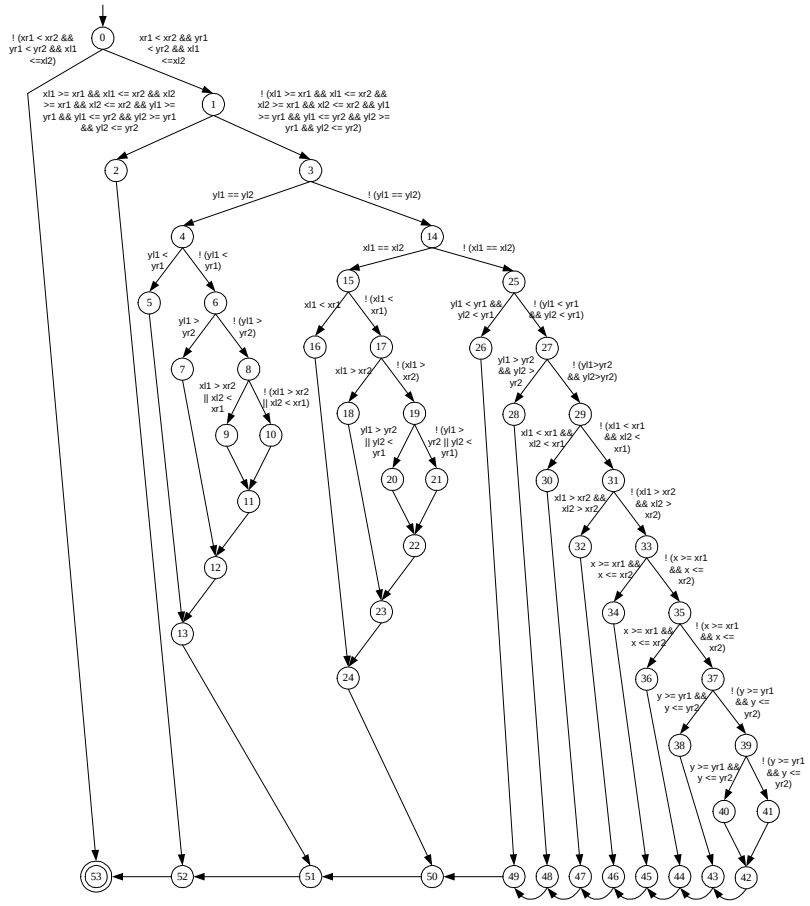



**Figure 10. "Line Rectangle Classifier" control flow graph**

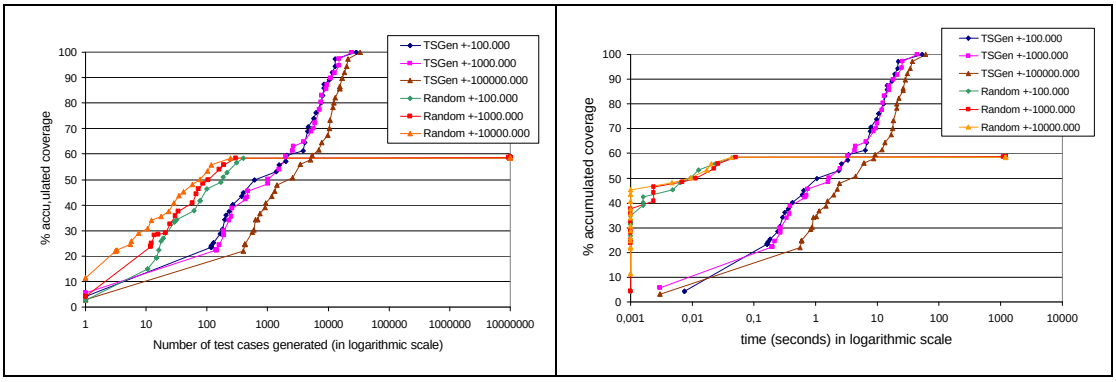



**Figure 11. Evolution of the number of tests generated and time for the "Line Rectangle Classifier" program**

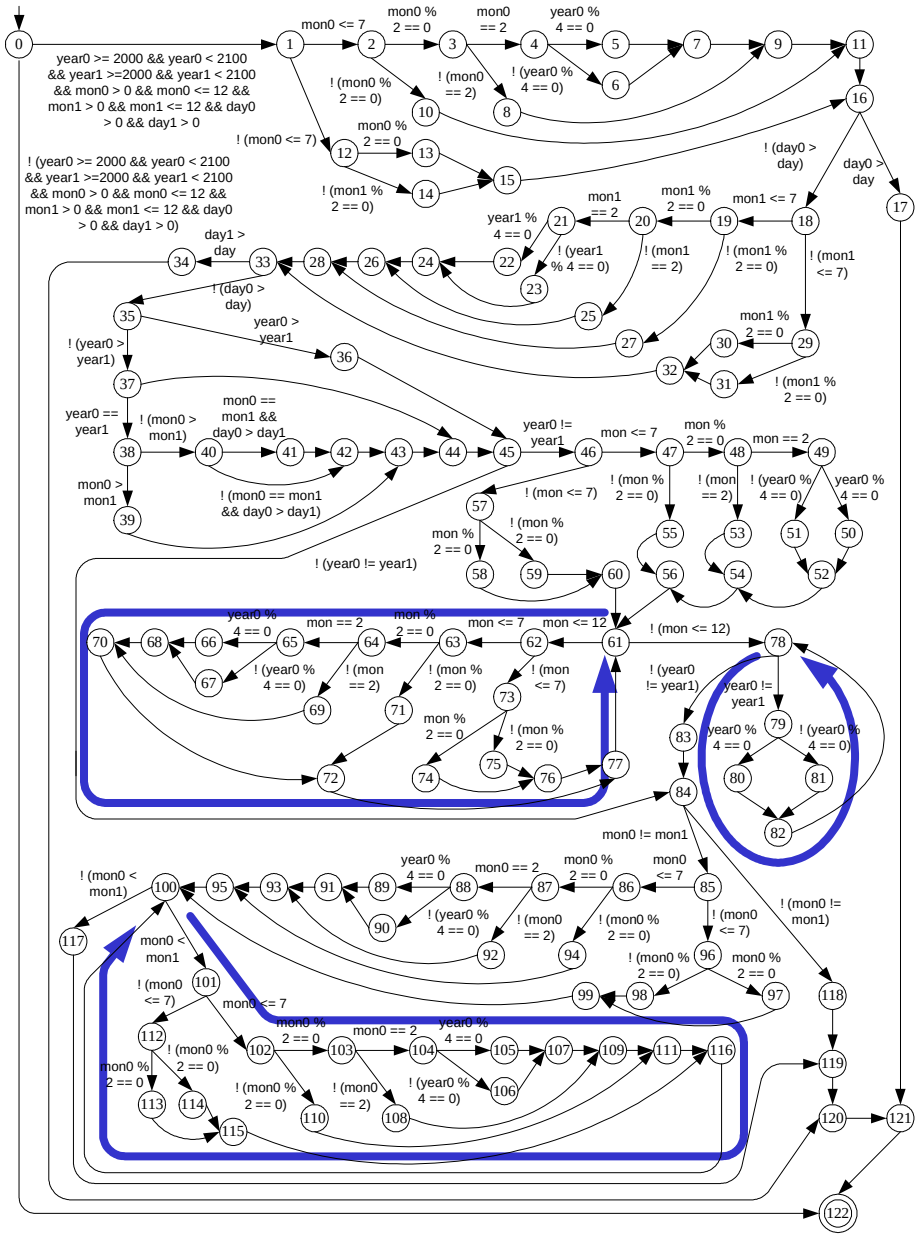



**Figure 12. The “Number of days” control flow graph**

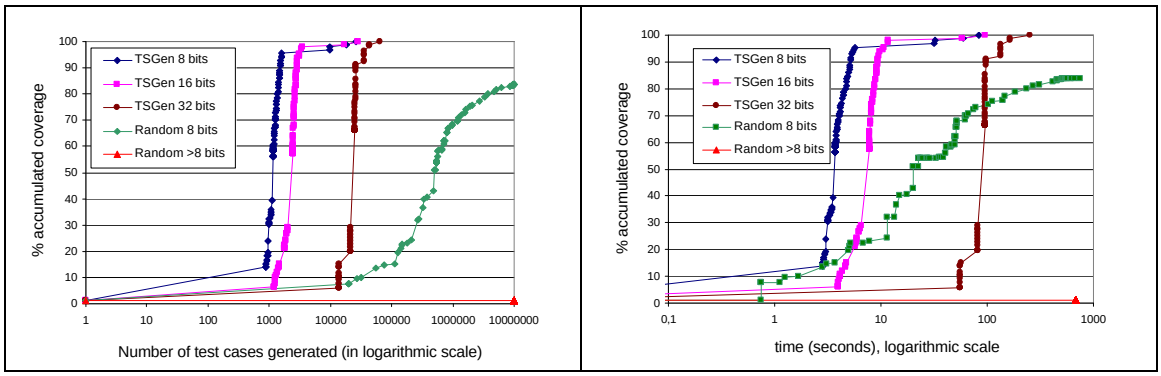



**Figure 13. Evolution of the number of tests generated and time for the ‘Number of days’ program**

# Tables

**Table 1. TSGen cost function $f_{nj}^{ni}(\bar{x})$**

| **Type operator** | **Cond$_{nj}^{ni}$** | **TSGen $f_{nj}^{ni}(\bar{x})$** |
|---|---|---|
| Relational with equality | x=y<br>x=y<br>x=y | \|x-y\| |
| Relational without equality | x?y<br>x<y<br>x>y | \|x-y\|+σ  being σ˜ 0 and σ>0 |
| AND | $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ | // the cost is the sum of the costs of all the false clauses<br>$\sum_{c_i FALSE} f_{nj}^{ni}(\bar{x})_{ci}$ |
| OR | $c_1 \vee c_2 \vee \ldots \vee c_n$ | // the cost is the minimum of the costs of all the false clauses<br>Minimum($f_{nj}^{ni}(\bar{x})_{ci}$) $\forall c_i$=FALSE |
| NOT | ¬c | Negation is propagated using the law of De Morgan<br>$\neg(c_1 \wedge c_2)$ is treated as $\neg c_1 \vee \neg c_2$: $f_{nj}^{ni}(\bar{x})$=Minimum($f_{nj}^{ni}(\bar{x})_{\neg ci}$) $\forall c_i$=FALSE<br>$\neg(c_1 \vee c_2)$ is treated as $\neg c_1 \wedge \neg c_2$: $f_{nj}^{ni}(\bar{x}) = \sum_{c_i FALSE} f_{nj}^{ni}(\bar{x})_{\neg c_i}$ |

**Table 2. TSGen cost function $fp_{ni}(\bar{x})$**

| **Type operator** | **Cond$_{ni}$** | **TSGen $fp_{ni}(\bar{x})$** |
|---|---|---|
| Relational with equality | x=y<br>x=y<br>x=y | \|x-y\| |
| Relational without equality | x?y<br>x<y<br>x>y | \|x-y\|-σ  being σ˜ 0 and σ>0 |
| AND | $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ | Minimum($fp(\bar{x})_{ci}$) $\forall c_i$=TRUE |
| OR | $c_1 \vee c_2 \vee \ldots \vee c_n$ | $\sum_{c_i TRUE} fp(\bar{x})_{c_i}$ |
| NOT | ¬c | Negation is propagated using the law of De Morgan |

**Table 3. Features of the results of previous works**

| | Technique | Adequacy criterion: Structural (coverage) | | | | | Type of input | | Explicit ranges for the input | Published Results | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Control-flow | | | | Data-flow | integer | real | | For the developed generator | | | Includes comparison with a random generator respect to | | |
| | | State-ment | Branch | C/D | Loop | | | | | Number of tests generated | % coverage (or success) | Time | Number of tests generated | % coverage (or success) | Time |
| [9] (1995) | GA | | | | | x | x | | A | x | x | | x | x | |
| [10] (1995) | GA | | x | | | | x | | A | x | x | | x | x | |
| [11] (1996) | GA | x | x | | x | | x | | A | x | x | x | x | x | x |
| [12] (1996) | GA | | x | | x | | x | | | x | x | N | x | x | |
| [22] (1998) | SA | | x | | | | x | | | | x | N | | x | |
| [13] (1998) | GA | | x | | | | x | | A | x | x | x** | x | x | x** |
| [14] (1999) | GA | x | x | | | | x | | | x | x | | x | x | |
| [16] (2001) | GA | | | | | x | x | x | A | x | x | | | | |
| [17] (2001) | GA | x | x | | | | x | x | S | x | x | N | x | x | |
| [18] (2001) | GA | | | x | | | x | x | | S* | x* | N | S* | x* | |
| [19] (2002) | GA | | | | | x | x | x | | x | x | N | x | x | |
| [20] (2004) | GA, SA | | | | | x | x | x | A | x | x | x** | | | |

* not mean results

** only for one of the used benchmarks

Technique: GA-Genetic Algorithms, SA-Simulated Annealing

Explicit ranges for the input: A- for All the benchmarks, S- for Some benchmarks

Time for the developed technique: N- Not detailed: only reported an interval time in which are included all experiments

**Table 4. Final results for the "Triangle Classifier" program**

| | TSGen | | | Random | | |
|---|---|---|---|---|---|---|
| | Tests | % cov. | Time (sec.) | Tests | % cov. | Time (sec.) |
| **Range 8 bits** | 217 | 100 | 0.111 | 170,315 | 100 | 5 |
| **Range 16 bits** | 738 | 100 | 0.442 | 10,000,000 | 91.67 | 298 |
| **Range 32 bits** | 19552 | 100 | 21.425 | 10,000,000 | 58.33 | 298 |
| | | | | | | |
| **Range ±100.000** | 697 | 100 | 0.395 | 10,000,000 | 58.33 | 330 |
| **Range ±1000.000** | 819 | 100 | 0.475 | 10,000,000 | 58.33 | 330 |
| **Range ±100000.000** | 1435 | 100 | 0.864 | 10,000,000 | 58.33 | 330 |

**Table 5. Final results for the "Line Rectangle Classifier" program**

| | TSGen | | | Random | | |
|---|---|---|---|---|---|---|
| | Tests | % cov. | Time (sec.) | Tests | % cov. | Time (sec.) |
| **Range ±100.000** | 29191 | 100 | 53,86 | 10,000,000 | 58.33 | 1210 |
| **Range ±1000.000** | 24606 | 100 | 43,91 | 10,000,000 | 58.33 | 1212 |
| **Range ±100000.000** | 33303 | 100 | 60,69 | 10,000,000 | 58.33 | 1210 |

**Table 6. Final results for the "Number of days" program**

| | TSGen | | | Random | | |
|---|---|---|---|---|---|---|
| | Tests | % cov. | Time (sec.) | Tests | % cov. | Time (sec.) |
| **Range 8 bits** | 25765 | 100 | 84.27 | 10,000,000 | 83.60 | 742 |
| **Range 16 bits** | 28081 | 100 | 96.63 | 10,000,000 | 1.16 | 686 |
| **Range 32 bits** | 65317 | 100 | 251.38 | 10,000,000 | 1.16 | 686 |