# Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search

Eugenia Díaz, Javier Tuya, Raquel Blanco
*Department of Computer Science, University of Oviedo*
*Campus de Viesques s/n, Gijón, Asturias,33203 SPAIN*
*eugenia@lsi.uniovi.es, tuya@lsi.uniovi.es, rblanco@lsi.uniovi.es*

## Abstract

*The use of techniques for automating the generation of software test cases is very important as it can reduce the time and cost of this process. The latest methods for automatic generation of tests use metaheuristic search techniques, i.e. Genetic Algorithms and Simulated Annealing. There is a great deal of research into the use of Genetic Algorithms to obtain a specific coverage in software testing but there is none using the metaheuristic Tabu Search technique. In this paper, we explain how we have created an efficient testing technique that combines Tabu Search with Korel's chaining approach. Our technique automatically generates test data in order to obtain branch coverage in software testing.*

## 1. Introduction

Software testing is an expensive process, typically consuming at least 50% of the total costs involved in software development [1]. With techniques for automating the generation of software testing, we will be able to test the software more efficiently while reducing the time taken up by this task, thus reducing the cost and increasing the quality of the final product.

Among the different approaches used for the automation of this process, we may distinguish between random techniques [9] (test data are generated randomly to cover the input variables domains as much as possible), static techniques [2] (the program under test is not executed) and dynamic techniques [6] (the program under test is executed).

The most recent dynamic techniques for automatic generation of tests, for obtaining a specific system coverage, follow two approaches: one is to represent the system by linear inequalities that are solved by a certain technique [4] and, the another is to use metaheuristic search techniques, in which the testing problem is treated as a search or optimization problem whose goal is to find the appropriate tests to obtain program coverage that is as high as possible. One of these metaheuristics, genetic algorithms, is the most widely used [5] [7] [8] [10]. The simulated annealing technique has likewise been used by [11]. Another metaheuristic technique that can be applied to automatic test data generation is Tabu Search. There are a great variety of real-world problems that can be solved by Tabu Search, however, in software testing, there are no prior studies in which Tabu Search has been used. This is the reason why we explain, in this paper, how we have used Tabu Search to automatically generate test data to obtain branch coverage in software testing.

A brief description of the Tabu Search algorithm is given in Section 2. In the third section we explain how we applied Tabu Search to obtain tests for branch coverage. Finally, we present the conclusions reached together with further lines of research.

## 2. The Tabu Search technique

Tabu Search (TS) is a metaheuristic search technique based on the premise that, in order to qualify as intelligent, problem solving must incorporate adaptive memory and responsive exploration [3]. Thus, the algorithm of Tabu Search is based on that of the next k neighbors, while maintaining a tabu list (memory) of visited neighbors that are forbidden. A general Tabu Search algorithm appears in Figure 1.

The Tabu Search algorithm has a number of parameters that have to be chosen on the basis of the problem to be solved: the objective function (fitness function) which has to measure the cost of a solution, an appropriate candidate list strategy (to try to choose good neighbor candidates that goes beyond a local optimum without exploiting the examination of elements in the neighborhood) and, it is also necessary to define short-term memory and long-term memory and their respective strategies. Short-term memory stores the recent moves of the search as tabu. Long-term memory, on the other hand, stores the frequency with which a move occurs in order to penalize frequently visited moves that diversify the search.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

```
begin
  Initialise current solution
  Calculate the cost of current solution and store it as best cost
  Store current solution as new solution
  Add new solution to tabu list
  do
    Calculate neighbourhood candidates
    Calculate the cost of candidates
    Store the best candidate as new solution
    Add new solution to tabu list
    if (the cost of new solution<best cost) then
        Store new solution as best solution
        Store the cost of new solution as best cost
    endif
    Store new solution as current solution
  while NOT Stop Criteria
  end
```

**Figure 1.** Tabu Search algorithm

## 3. Using Tabu Search to obtain branch coverage

We have developed an algorithm, based on Tabu Search, to generate tests that enable us to automatically obtain the maximum percentage of branch coverage in any given software. Furthermore, as our objective is to obtain branch coverage, we need to know throughout the whole search which branches of the program have been covered by tests and which not. To do so, we use the program control flow graph (whose nodes represent statements and whose edges represent the flow of control between statements) to store relevant information about a node, like for example whether it has been reached or not and the best test that reached it.

Next, we explain how we have applied the TS algorithm to solve the problem of obtaining branch coverage in software testing.

### 3.1 Algorithm goal and solution

First of all, we need to define the goal to reach, namely obtaining maximum branch coverage. If there are unfeasible branches, this value will be unknown. For this reason, we established a stopping criterion for the algorithm: when all the branches have been reached (all nodes covered) or when a maximum number of iterations of the TS algorithm have been surpassed.

On the other hand, a current solution (test case) for us is formed by a set of given values ($v_1$, $v_2$, …,$v_n$) for the input variables (x, y, z,....,n) of the program under test. When the current solution is the best one reaching a node or for reaching a child node, it is stored as the best solution of the node together with its cost. The cost of a test is calculated using the fitness function shown in sub-section 3.5.

Initially, a random solution is generated as the current solution. Via this test, the program under test is executed to check along which branches it has gone and the cost that said test has incurred in each node that it has reached. As this is the first solution found, it will be stored in each reached node.

When the algorithm finishes, it shows each best test that has reached each leaf node on the control flow graph and each best test that has reached each non-leaf node that it was not possible to continue testing in its child nodes (possible unfeasible branches).

### 3.2 Tabu lists: the memory of the algorithm

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. In our approach, short-term memory stores the tests that have been the best for the goal node and long-term memory stores the worst tests during all the search process.

All the tests generated (and not only the best) could remain in memory, but that would slow down the search if the tabu list is large and it would not suppose a major improvement, since the test candidates that are generated are based on the best test in each node. For this reason, avoiding repeating the best test would be to avoid repeating many tests candidates. Also, as only the neighboring candidates of the best tests will be explored, these tests will be tabu for the rest of the search process, since their neighborhood has already been explored.

Figure 2 shows a simple example of how the algorithm works with tabu lists. The algorithm's goal is to achieve the global minimum x* (test data case that achieves the node goal) defined as $f(x^*)<f(y)\ \forall y \in D, y \neq x^*$, where D is the set of feasible values for x and f(x) is the fitness function, which is calculated as explained in sub-section 3.5. Suppose that the initial solution is x0 (see Figure 2). This is the best solution known, so it is added to the short-term memory tabu list. Next, to try to achieve x*, some neighboring candidates of the best solution up until that moment ($x_0$) are generated as explained in sub-section 3.3. Of all the candidates, the test data $x_i$ selected is that which satisfies $f(x_i)<f(x_0)$. For example, it obtains $x_1$ as the best candidate. $x_1$ is then inserted here in the short-term memory tabu list. Next, the algorithm will use $x_1$ to find a new $x_i$, for example x2, which will then be inserted in the short-term memory. This whole process will be repeated until the algorithm achieves the goal x* or until it arrives at a strong local minimum. A strong local minimum $x_n$ is that test data that it has not been possible to find in its explored neighborhood an $x_i$ that satisfies $f(x_i)<f(x_n)$.

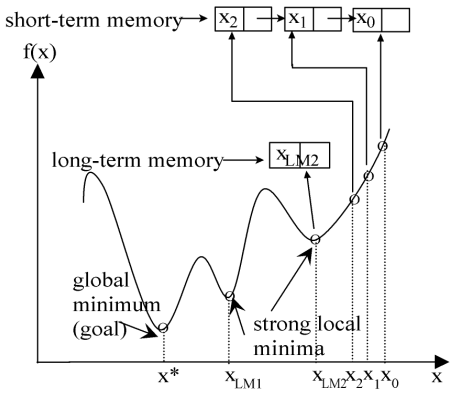


**Figure 2.** Tabu lists creation

The tabu algorithm should preclude getting stuck in the local minima. The long-term memory tabu list is used for this. Those $x_n$ that have been local minima during the search process are stored in this list. In the example, if the algorithm reaches the minimum local $x_{LM2}$ before achieving the goal, $x_{LM2}$ is stored in the long-term memory tabu list (Figure 2). Once a local minimum has been found, the tabu algorithm applies backtracking, as explained in subsection 3.4.

The long-term memory will have few test data cases. For this reason, it is maintained during the testing process without too much effect on the algorithm's performance. On the other hand, the short-term memory could store enough test cases and this could reduce the algorithm's performance. This, however, does not occur since this memory is often deleted (when the goal node changes). Thus, the use of memory in the testing tabu algorithm is not critical.

### 3.3 Neighbor selection process

Inside the loop (see Figure 1), the first step will be to calculate the neighboring candidates. The idea here is that if one test case in the control flow graph has covered the parent node but not the child node, a neighboring test can be found that reaches the child node using the test that covers the parent node and which is the best one found up until then (Figure 3). This idea uses the chaining approach of [6], 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 our algorithm, we have selected as the node to cover in each iteration the following node without cover in preorder and whose parent is already covered.

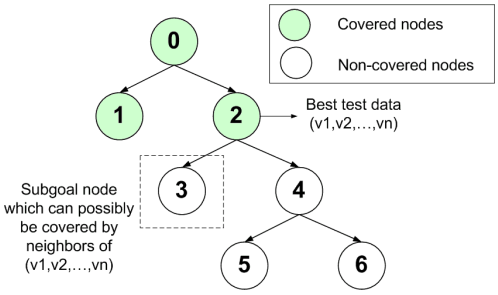


**Figure 3.** Subgoal node selection

Once we have established the subgoal node, 4*n neighbors are generated starting from the best known test for its parent, n being the number of input variables of the program under test. Our technique consists in generating two near neighbors and two neighbors further from the test that is the base. That is to say, if the base test is ($v_1$, $v_2$, …,$v_n$), we maintain the same values for all $v_k$ that satisfy $k \neq i$ and we generate four new values for $v_i$. Thus, we have 4 new test data neighbors:

$(v_1, v_2, ..., v_i+\lambda,\ldots,v_n)$
$(v_1, v_2, ..., v_i-\lambda,\ldots,v_n)$
$(v_1, v_2, \ldots, v_i+\mu, ...,v_n)$
$(v_1, v_2, ..., v_i -\mu,\ldots,v_n)$

where $\lambda$ is a low step length and $\mu$ is a high step length. In our implementation, the values for $\lambda$ and $\mu$ are dependent on the type and range of the input variables and although they are fixed at the beginning of the algorithm, they change during execution, taking into account the evaluation of the generated tests. Thus, it would be possible to carry out bigger jumps when there are no appropriate neighbors and a very fine adjustment of the search when the neighbors come closer to the desired evaluation.

Once the tests candidates are generated, we verify whether these are tabu tests, in which case they are rejected. If a candidate is not tabu, it is executed with the program under test and if it has been the best test known for some of the nodes that it has reached, it is stored as the best test for the node.

### 3.4 Treatment for unfeasible branches

Furthermore, in order to prevent the algorithm spending all its iterations in an attempt to cover unfeasible nodes and/or in trying to reach a child node from a bad test in the parent, we have defined two parameters whose values are dependent on the magnitude of the program to be tested: a maximum number of neighbors to try to reach a child node with the same test in the parent and a maximum number of neighbors to reach a child node from its parent node. Thus, when one of this maximum has been reached and it has not been possible to achieve the target child node, backtracking is carried out to try to search for

a new better test for the parent node (which is marked as not covered). The memory of the TS algorithm, the tabu lists, is of great importance here, since it avoids reconsidering as better tests those that have already been tested and were rejected for not being able to reach the goal node.

### 3.5 Fitness function

Before executing the TS algorithm, it is necessary to instrument the program being tested so as to introduce, at each node, the sentences able to assign the cost of the test that reaches it. The use of a good cost function is fundamental for the algorithm to work correctly. As we see it, for us, the best test for a node is the test that has more possibilities of allowing permutation between branches, i.e. the test that when executing one node has the most boundary values reaching that node. Then, the cost function of our algorithm is calculated as shown in Table 1.

**Table 1.** TS fitness function

| Element | TS fitness function |
|---|---|
| Boolean | 0 |
| x=y, x≠y | abs(x-y) |
| x<y, x≤y | y-x |
| x>y, x≥y | x-y |
| x∧y | Min(cost(x),cost(y)) |
| x∨y | if x is TRUE and y is TRUE then<br>Min(cost(x),cost(y))<br>else $\sum_{c_i FALSE} \cos t(c_i)$<br>end if |
| ¬x | Negation is moved inwards and propagated over x |

## 4. Conclusions and further work

Tabu search is a metaheuristic search technique based on the algorithm of the next k neighbors. Although there are a great variety of real-world problems that can be solved by Tabu Search, this is the first work of its application to software testing.

The structure of TS, which consists in searching inside the neighborhood of a solution and remembering the best solutions, makes TS a simple, intuitive technique to apply to the generation of branch coverage tests. Furthermore, the experimental results we have obtained with our tabu algorithm make TS an effective technique for obtaining very high branch coverage.

This is a new work that opens two clear investigation lines: the study of the behavior of the algorithm when their tabu parameters change and the application of TS to obtain other types of software coverage (path coverage, multiple condition coverage and loop coverage).


## Acknowledgements

This work was funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, project TIC2001-1143-C03-03


## References


[1] Beizer, B., Software Testing Techniques, 2nd. Ed. Van Nostrand Reinhold. 1990

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

[3] Glover, F. Tabu search part i, ii. ORSA Journal on Computing, 1(3). 1989.

[4] Gupta, N., Mathur, A. P., Soffa, M. L. Generating Test Data for Branch Coverage.15th IEEE International Conference on Automated Software Engineering (ASE'00), Grenoble, France, September 2000.

[5] Jones, B., Sthamer, H.and Eyers, D. Automatic structural testing using genetic algorithms. The Software Engineering Journal 11. 1996

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

[7] Lin, J., Yeh, P. Automatic test data generation for path testing using GAs. Information Sciences 131. 2001.

[8] Michael, C., McGraw, G., Schatz, M., Walton C. Genetic Algorithms for Dynamic Test Data Generation. 12h IEEE International Conference on Automated Software Engineering (ASE'97), Tahoe NV, November 1997.

[9] Ntafos, S. On Random and Partition Testing, Intl. Symp. on Software Testing and Analysis,. 1998

[10] Pargas, R.P., Harrold, M.J., Peck, R.R. Test data generation using genetic algorithms. The Journal of Software Testing, Verification and Reliability, 9. 1999

[11]. Tracey, N., Clark, J., Mander, K. Automated program flaw finding using simulated annealing. International Symposium on software testing and analysis. ACM/SIGSOFT. 1998