Journal of Positive School Psychology http://journalppw.com
2022, Vol. 6, No. 10, 3428-3431

Structural Methods For Program Testing

M.M.Aripov

Associate Professor of the Department of Informatics, Kokand State Pedagogical Institute

Abstract – Structural methods for program testing are described, such as branch testing, program verification, symbolic testing, and generating structural tests. An algorithm for the minimum coverage of the program graph based on the packing adjacency matrix and a specific example of the minimum coverage of the program graph are given.

Keywords - testing, program graph, minimum program graph coverage, packed adjacency matrix, DD-paths, vertex, branches, g-graph, h-graph, algorithm complexity.

INTRODUCTION

Testing - checking the operation of the program based on the results of its execution on specially selected sets of initial data - tests. The program can be tested either completely (full testing) or selectively (selective testing) at individual points in the source data space. With random testing, the reliability of a program cannot be fully guaranteed. If tests are offered by the programmer, then they can cover only those parts of the program with which the programmer is most familiar. Therefore, many hidden errors may remain undetected. Full testing on all possible input sets of the program or even testing of all paths in the program structure is unrealistic, since the number of tests will be unacceptably large. For example, if the number of inputs is ten and each input of the program can take on ten values, the number of elementary tests required to complete the test would be 1010.

Branch testing. A more stringent requirement is that the chosen paths must span all branches of the program structure, or all branches across the board (dynamic testing or branch testing). This approach ensures that all statements and all branches are tested once. Experience shows that a significant number of errors arise due to inaccuracies in the formulation of exit conditions from loops, so it is proposed to introduce an additional requirement that each loop be tested by two tests, one of which would lead to the execution of the loop with a return, and the other would go through the loop without return.

Program verification. Any testing using numerical sets of initial data allows you to check the program only in a limited number of points in the space of initial data, so more general methods are of greatest interest. This includes, first of all, the verification of programs - the proof of their correctness using mathematical methods for proving theorems. To do this, the program is presented as a sequence of more or less simple statements, the proof of which is not difficult. This process can be automated, but practical results in this direction are still insignificant. The fact is that the proof of even relatively simple statements is a procedure that requires high qualifications and is subject to automation only in some rare cases. Due to the great complexity of the proof, errors are possible here, which from a practical point of view, despite the apparent rigor, lead to the fact that the verification method cannot guarantee the complete reliability of the verified program.

Symbolic testing. In contrast to verification, program testing consists in checking the correctness of the numerical results of program operation with specially selected values of input variables - test sets. In some cases, testing can also be done symbolically - by executing procedures based on symbolic inputs (notations

of input variables that allow expressing program outputs also in symbolic form). Different symbolic inputs and outputs correspond to different program paths. If there are a limited number of such paths, then symbolic execution can be used to validate the program using symbolic input and output expressions. The advantage of symbolic testing over numerical testing is that if a numerical test allows you to check the operation of a program on individual numerical values of input sets, then symbolic testing operates on sets of initial data determined by constraints. Symbolic expressions of program paths can be obtained either by forward substitution or by back substitution. Direct substitution corresponds to the actions performed when implementing a certain path in the program structure. With direct substitution, symbolic execution is carried out for each executable statement with storage of intermediate symbolic expressions of variables. In the case of back substitution, restrictions on the input variables are built "from the bottom up" when passing the path on the program graph in the opposite direction. As a result, the same restrictions are obtained as in direct substitution. However, with back substitution, no memory is needed to remember the symbolic records of variables. But with direct substitution, there is the possibility of early detection of unfeasible paths with conflicting constraints on the initial data. In symbolic testing, cyclic sections of the program present a certain difficulty, since in this case the number of iterations is unknown. The problem can most simply be overcome by substituting some pre-estimated number of iterations. However, in this case, the resulting restrictions may not be accurate. The second difficulty is related to the presence of modules in the program. The latter is overcome by the symbolic execution of the modules encountered on the given path. The third difficulty is related to the symbolic execution of data arrays. The fact is that in some cases the value of the variable is set only during the execution of the program. This difficulty can be overcome by introducing additional (hypothetical) restrictions corresponding to various possible cases.

Generation of structural tests. The shortcomings mentioned above are devoid of structural testing of programs on specific numerical initial data [1-,3]. Test generation consists in choosing a set of paths that completely cover the program graph, and in determining the test data on which these paths are executed. A program graph (control graph) is a structural model of a program that shows the relationship between its elements. The vertices of the graph represent the branching and union operators, and the arcs represent the data processing and transmission operators. The graph is represented as a packed adjacency matrix (PAM). The packed adjacency matrix A = { a_{ij}} of a graph with v vertices is a (v x l) matrix (l is the maximum exit degree of the i-th vertex). The degree of entry $d_{inp}(v_i)$ and exit $d_{out}(v_i)$ of some vertex of the graph means, respectively, the number of incoming and outgoing arcs from the vertices. Each row i of the PAM is filled in random order with the numbers of vertices that are adjacent to vertex i. The representation of graphs in the form of PAM has the following advantages over other existing representations: for large graphs, the number of columns of PAM is much less than the number of columns of the corresponding adjacency matrix; it is relatively easy to model the process of moving along the graph to build paths; reduces graph processing time. The test criterion is the criterion of branches, where a program branch is understood as a certain sequence of statements that are executed strictly one after another. Thus, a branch is a linear section of a program. To construct the minimum coverage, the graph is divided into DD-paths using the CMS of the original graph. The set of vertices with output degree $d_{out}(v_i)>1$, input and output vertices are denoted as D-vertices. Then a DD-path is a simple path between two D-vertices, such that there are no D-vertices within its boundaries. Then the cycles and loops are determined and the arcs closing them are excluded.

The proposed algorithm for constructing a minimum cover (MPOC) of a graph consists of the following steps.

Stage 1. The vertex i is looked through and the adjacent vertex j is determined, the

number of which is the maximum among the numbers of adjacent vertices, where i Є { l , n - 1;} n is the number of graph vertices.

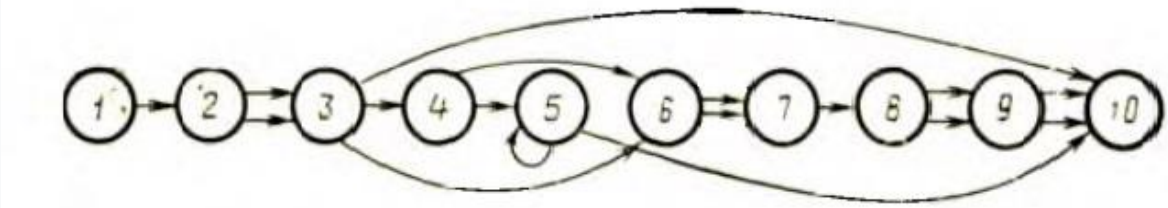

Fig. 1. An example of a program graph

Stage 2. The arc (v_i, v_j) is viewed. If d_{inp} (v_i) > 1 and d_{out}(v_j) >1 , then the arc g(v_i,v_j) is excluded. If d_{out} (v_i) > 1 and d_{inp} (v_j) = 1, then the arc h(v_i, v_j) is marked.

Step 3. Substitute i = j and repeat steps 1-2 until j is equal to the number of the final (output) vertex. The path is fixed as a sequence of values j.

Stage 4. If there are no arcs of type g in the constructed path, then the last arc of type h is excluded.

Stage 5. Stages 1–2 are repeated until the constructed path contains no arcs of type g and h

An example of constructing a minimal coverage of a program graph. Let the program graph shown in Fig. 1. Graph arcs mean a sequence of computational program operators, graph vertices — branching and union operators. After eliminating the closing cycles of arcs (they are tested separately), the graph in Fig. 1 is described by the following PAM:

1	2	0	0
2	3	3	0
3	4	6	10
4	5	6	0
5	10	0	0
6	7	7	0
7	8	0	0
8	9	9	0
9	10	10	0
10	0	0	0

The first stages of the MPOC algorithm give the following results:

Stage 1. Set i = 1, j = 2. {1, 2}

Stage 2. The arc (v_i, v_j) is not excluded and is not marked.

Stage 1. Set i = 2, j = 3. {1, 2, 3}

Stage 2. One of the arcs (v_2, v_3) is excluded

Stage 1. Set i = 3, j = 10. p1 = {1, 2, 3, 10}

Stage 2. The arc (v_3, v_{I0}) is eliminated.

Stage 1. Set i = 3, j = 6.

Stage 2. The arcs (v_6, v_7), (v_8, v_9), (v_9, v_{I0}) are excluded, the arc

h(v_3, v_6) noted.

The procedures of stages 1–2 are repeated until the path to the final vertex of the graph v_{10} corresponding to the receipt of the calculation result is determined. In this case, the first path p1 = { l , 2, 3, 10} is determined after three steps. The following steps, repeated until there are no arcs of type g and h in the constructed path, allow us to determine the following paths:

p_2 = {1, 2, 3, 6, 7, 8, 9, 10},

p_3 = {1. 2, 3, 4, 6, 7, 8, 9, 10},

p_4 = {1. 2, 3, 4, 6. 7, 8, 9, 10},

p_5 = {1, 2, 3, 4, 5, 10}.

To create one path in the worst case, n operations are required, and to build the minimum number of operations, m operations are required, where m is the minimum number of paths that cover all branches of the program graph. Therefore, the complexity of the developed algorithm is

O(|v| x |m|) => O(|v|)

The developed algorithm is more efficient than the algorithm proposed in [5], since in this algorithm the vertices are excluded after creating a certain path, i.e. additional time required.

Literature

1. Iyudu K.A., Aripov M.M. Automating the generation of paths for testing programs written in Fortran. Programming, 1986, No. 7.
2. Iyudu K.A., Aripov M.M. Testing a program based on the minimum coverage of its graph. Control systems and machines, 1985, No. 6.
3. Iyudu K.A., Aripov M.M. Automation of structural testing of programs. Republican conference. Reliability and quality of software. Abstracts of reports. Lvov, January 29-31, 1985
4. Kulikov S.S. Software testing. Minsk, 2020.
5. Simon C., Ntafos S., Louis Hakimi. On structured digrafs and program testing. IEEE Trans. On Computers, vol. C-30, № 1, January 1981.