PROGRAM VERIFICATION STRATEGY AND EDGE RANKING OF GRAPHS

Dariusz Dereniowski, Marek Kubale

Department of Algorithms and System Modeling
Faculty of Electronics, Telecommunications and Informatics
Gdańsk Univerity of Technology
Gabriela Narutowicza 11/12
80-952 Gdańsk Wrzeszcz, Poland
{deren,kubale}@eti.pg.gda.pl

Abstract

There are several criteria for testing program correctness. In this paper we deal with the problem of automatic software testing under the assumption that the set of tests (assertions) is given for selected blocks of code. We simplify the analysis by assuming that the program being tested contains exactly one bug, but this does not lead to loss of generality. We consider not only practical aspects of the above problem and a graph-theoretical model in general but some chromatic aspects of a graph searching model as well.

Keywords: edge ranking, graph searching, partial order, software testing

Introduction

Program verification is an important and complex technological process useful in debugging, testing, maintenance and understanding of computer programs. In this paper we deal with test verification strategy. By a test we mean verification of the correctness of a piece of computer program. This verification relays on the assertion whether a given piece of code behaves in accordance with its specification. We focus on a theoretical model of the problem assuming that the program at hand is tested by a collection of reliable tests.

The quality of testing process is affected by the quality of individual tests. There are several measures of the quality of testing. We can calculate this quality by computing the fraction of the number of blocks of code executed during the test to the number of all blocks (block coverage) or by computing the number of different decisions made during the execution (which is the number of different Boolean values obtained during computation of all "**if**" statements in the program) to the number of all possible decisions (decision coverage). We obtain another criterion by analyzing the paths of execution, where each path is a sequence of instructions between the line of code containing the definition of a variable to the line where the variable has been used [5]. Herein we skip relevant aspects of execution of individual tests. We merely assume that erroneous code is identified whenever there is a bug within it.

Given a computer program, our aim is to design an optimal test strategy for it. By a block we mean a set of consecutive statements that can be tested individually. This means that each block is accompanied by a test whose execution allows to claim whether there is a bug in it or not. Therefore we assume that the information on which the set of statements may be tested individually is part of the input to the problem. The optimization criterion is the number of tests executed in the worst case, which corresponds to the complexity of testing. In general, not all tests must be executed to identify a bug in the program. If, after executing a test, we know there is a faulty instruction in the corresponding piece of code, then the further testing is confined to this fragment of program. Otherwise, if there is no bug in it, the further testing is realized for the segments of code not including the fragment just verified.

Observe that the above process may not lead to finding the faulty instruction. Such a situation occurs when the tests have been constructed in such a way that we have an input-output pair (or assertion) for a block of code and in each test we compare a correct output (which serves as a template) with the output computed by this block. If the output of the tested block of code is incorrect then we may assume that one can select some variables (defined in the program) with incorrect values obtained for the particular input and at the particular point of the program execution. If we have such information then we may use efficient techniques based on static or dynamic program slicing for finding the buggy instruction [1].

It is possible to represent the code of program by means of a graph [2, 5, 6], and we will follow this direction in the paper. More precisely, given a tree T describing the structure of the program, we will show that an optimal search strategy corresponds to the edge ranking number of T, which can be found in linear time [8]. Theoretical considerations will be illustrated by a series of practical examples.

Basic notions

To make this paper self-contained we begin with the definitions of basic notions concerning graph theory.

Graph G is an ordered pair of two sets V and E. Set V is called the set of *vertices* and E is called the set of *edges*, where each edge is an unordered pair $\{x,y\}$ of vertices $x,y \in V$. A *path* in graph G joining vertices x and y is any set of pairwise different vertices $v_1,\ldots,v_k$ such that $v_1 = x$, $v_k = y$ and $\{v_i, v_{i+1}\}\in E$ for each $i = 1,\ldots,k - 1$. G is a *tree*, if there is exactly one path between any pair of vertices. Tree T is *rooted*, if it has a distinguished vertex $r\in V$ called the *root*. By the *level of vertex x* in a rooted tree T we mean the length of the path joining x and r. A graph G is said to be *connected*, if there is a path between any pair of vertices. Thus any tree is a connected graph. In fact, it is the smallest connected graph in the sense that any tree with one edge deleted is disconnected. If G is disconnected then by a *connected component* we mean any of its connected subgraphs maximal in the sense of the number of vertices.

Let G be a *simple* graph, i.e. one without loops or parallel edges. By an *edge ranking* of G we mean a function c: $E \rightarrow \{1,\ldots,k\}$ such that every path joining x and y with $c(x) = c(y)$ contains an edge z such that $c(z) > c(x)$. The elements of set $\{1,\ldots,k\}$ are called *colors*. The smallest number k for which such a function c exists is called the *edge ranking number* of G and denoted $\chi'_r(G)$.

A *connected component* of a graph G is its connected subgraph H such that no connected subgraph of G, other than H, contains H. Given $S \subseteq G$, an *induced* subgraph of G is the graph with the vertex set S and the edge set $\{\{u,v\}\in E(G)\colon u,v\in S\}$, and is denoted by $G[S]$. If T is a rooted tree and v is a vertex of T then the symbol $T[v]$ is used to denote the tree induced by v and all its descendants in T.

The edge ranking is a hard combinatorial problem [7,3]. However, there is known a linear time algorithm for finding an optimal edge ranking of a tree [8]. We will exploit this fact later on. In particular, we will use the following

Fact 1. *If c is an edge k-ranking of G then there is exactly one edge e with color k. If G is a tree then after deleting e from G the remaining graph has exactly two connected components which are trees.*

and a more general consequence of Fact 1, namely

Fact 2. *If c is an edge k-ranking of G then each connected component of the subgraph* $G[c^{-1}(\{1,\ldots,l\})]$, *where* $k \geq l$, *contains at most one edge with color l.*

Formal model

We begin with some preliminary assumptions. First of all, we assume that there is just one bug in the code of program. Although such a situation appears rarely, this does not violate the generality of our considerations. At charge of executing one additional test applied to the whole program, we can state whether the program contains errors or not. If so, the verification process will help us to identify a faulty block. If, on the other hand, the program contains several bugs, then our strategy allows us to find one of them. In that case we can eliminate that bug and then search for other bugs.

Our theoretical considerations are supported by a simple code shown in Fig. 1(*a*). The program is depicted in the form of pseudocode, where letters A, B,…,Q denote the consecutive blocks of it. Each procedure may contain any other statements except those which belong to inner blocks. Fig. 1(*b*) shows a structure of procedure B of Fig. 1(*a*) consisting of two inner blocks. In the following we will explain that the method of identifying the bug allows us to decide whether the bug is in instructions 1, 4 or 7, but the structure of assertions in Fig. 1 does not give more information, i.e. in which of the three mentioned blocks of instructions the bug is contained.

```
(a)                                  (b)
proc A:       proc D:                procedure B:
   call B;       call J;                instructions 1;
   call C;       call K;
   call D;       call L;                {Assertion 1: precondition}
                                        instructions 2;
proc B:       proc F:                   call E;
   call E;       call M;                instructions 3;
   call F;       call N;                {Assertion 1: postcondition}

proc C:       proc I:                   Instructions 4;
   call G;       call O;
   call H;                              {Assertion 2: precondition}
   call I;    proc K:                   instructions 5;
                 call P;                call F;
                 call Q;                instructions 6;
                                        {Assertion 2: postcondition}

                                        instructions 7;
```

Fig. 1 (*a*) An example of program structure; (*b*) a structure of procedure B from Fig. 1(*a*).

Assume that the bug is in the fragment of code denoted by ‘instructions 7’ in Fig. 1(*b*). Later on we will continue this example to illustrate the way of identifying this error.

Now we come back to a more formal treatment of the subject. Let V be a finite set of elements. Suppose there is a partial order relation $\mathrm{R} \subseteq V^2$. Following [2] we introduce

Definition 1. By a *search algorithm* A of element $v \in V$ in a partial order (V,R) we mean an ordered triple (x, A_1, A_2), where $x \in V$, $|V| \neq 1$. A_1 is a search algorithm of v in set $V_1 = \{y \in V: y\mathrm{R}x\}$ and it is invoked by A in the case of $v \in V_1$. A_2 is a search algorithm of v in set $V \backslash V_1$ and it is invoked by A when $v \notin V_1$. If $|V| = 1$ then the algorithm stops and the element that we are looking for has been found.

Suppose that v is in V. Then we define the number of steps s of A, namely $s(A) = 0$ if $|V| = 1$, and otherwise

$$s(A) = \max\{s(A_1), s(A_2)\} + 1.$$

Given two program blocks u and v, we say that u *is contained in* v, if u is a set of statements belonging to v or u belongs to a block which belongs to v. In a set of blocks of a program we define a partial order relation R so that $u\mathrm{R}v$ if and only if block u belongs to v. Similarly to [2] we assume that Hasse diagram of R is a rooted tree. Fig. 2 shows such a tree for the program of Fig. 1(*a*) together with an edge ranking of its edges, where we treat the Hasse diagram as a simple tree.

Now we are ready to describe the idea of test strategy. Suppose that a given block u contains an erroneous statement. We choose any block of code $u' \subseteq u$ and perform a test for u'. If there is a bug in the block, then we confine the further testing to u'. Otherwise, we focus our search strategy on the blocks of u other than u'.

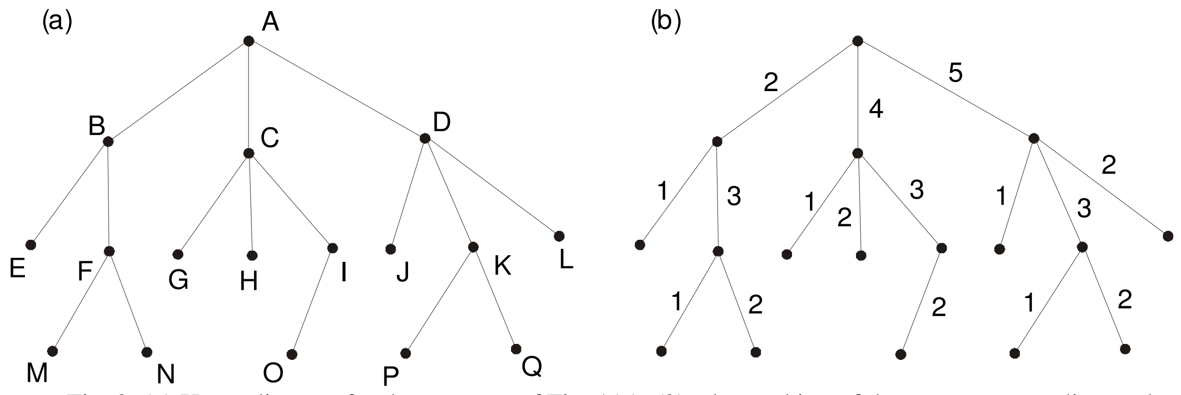

Fig. 2. (*a*) Hasse diagram for the program of Fig. 1(*a*); (*b*) edge ranking of the tree corresponding to the diagram

The above idea can be described more formally as follows: the process of testing is a search algorithm A, whose input is u, and for a fixed u' performs test of u'. If the test shows that there is a bug in A_1 then we perform algorithm A_1 with input u', otherwise we perform A_2 for $u\backslash u'$. The timing of the whole computer program is measured by the number of comparisons and is equal to $s(A)$.

Construction of a search algorithm

In the previous sections we have formulated a model for the program verification strategy in the case that a computer program is represented as a graph. Now we give a construction of a search algorithm assuming that a rooted tree T is given which describes the structure of the program. Also, we continue the example of Fig.1.

Suppose that c is an optimal edge ranking of the tree T. The procedure for finding an optimal search strategy is as follows

```
Procedure CreateSearchStrategy
    Input : tree $T$, edge ranking $c$;
    Output: search algorithm $A$
    begin
        if $|V(T)| = 1$ then
            $A$:= 'return $v$';
        else begin
            $\{u,v\}$:= edge such that $c(\{u,v\}) \geq c(e)$ for $e \in E(T)$; $u$ is the father of $v$ in $T$;
            $A_1$ := CreateSearchStrategy($T[v]$, $c|_{T[v]}$);
            $A_2$ := CreateSearchStrategy($T\text{-}T[v]$, $c|_{T-T[v]}$);
            $A := (v, A_1, A_2)$;
        end
        return $A$;
    end.
```

The processing time of the above procedure is linear with respect to the number of vertices of T and, consequently, to the number of blocks in the program being tested. This is so because an optimal edge ranking c of the edges of T can be found in linear time [8] and once we have got function c we are able to find the edge with the highest rank in constant time.

Theorem 1. *Procedure* **CreateSearchStrategy** *finds a search algorithm A which in time* $w(A) = \chi'_r(T)$ *finds a node in T corresponding to the faulty block in the program.*

Proof. Let x be the vertex corresponding to the block containing a bug. We prove by induction on $|V(T)|$ that if $x \in V(T)$ then x will be found by the search algorithm A. If $|V(T)| = 1$ then the only element $V(T)$ is x, which means that Theorem 1 is true, since A will not perform any test and $\chi'_r(T) = 0$. If $|V(T)| > 1$ then A:= $(v, A_1,$

A_2). According to the definition, A performs a test of a piece of code corresponding to the subgraph $T[v]$, where v is defined in the body of **CreateSearchStrategy**. If the test is possible then $x \in V(T[v])$ and by the induction hypothesis we know that x will be found in $\chi'_r(T[v])$ steps. Otherwise, we know that $x \in V(T-T[v])$ and the block x will be found in $\chi'_r(T-T[v])$ steps. Since $\chi'_r(T) = \max\{\chi'_r(T[v], \chi'_r(T-T[v])\}$ and in view of (1), the thesis of the theorem follows. □

Example

Herein we continue the example of construction of search algorithm for the code shown in Fig. 1. The corresponding tree T is shown in Fig. 2(*a*). We assume that an optimal coloring c of T is shown in Fig. 2(*b*). Fig. 3 depicts a search algorithm restricted to subtree $T[D]$. The right-hand successor of each node leads to the algorithm for searching the corresponding subtree, executed in the case of positive answer of the test, i.e. when there is a bug in the piece of code. The left-hand successor leads the search algorithm that is executed in the case of negative answer of the test. According to the definition of the search algorithm, the performance of it reduces to the length of a longest path from the root to a terminal vertex (leaf). The algorithms corresponding to the leaves have been denoted by $A_1,\dots, A_6$ in Fig. 3. For the remaining vertices the search algorithm is an ordered triple, where the recursive call depends on the output of the corresponding test.

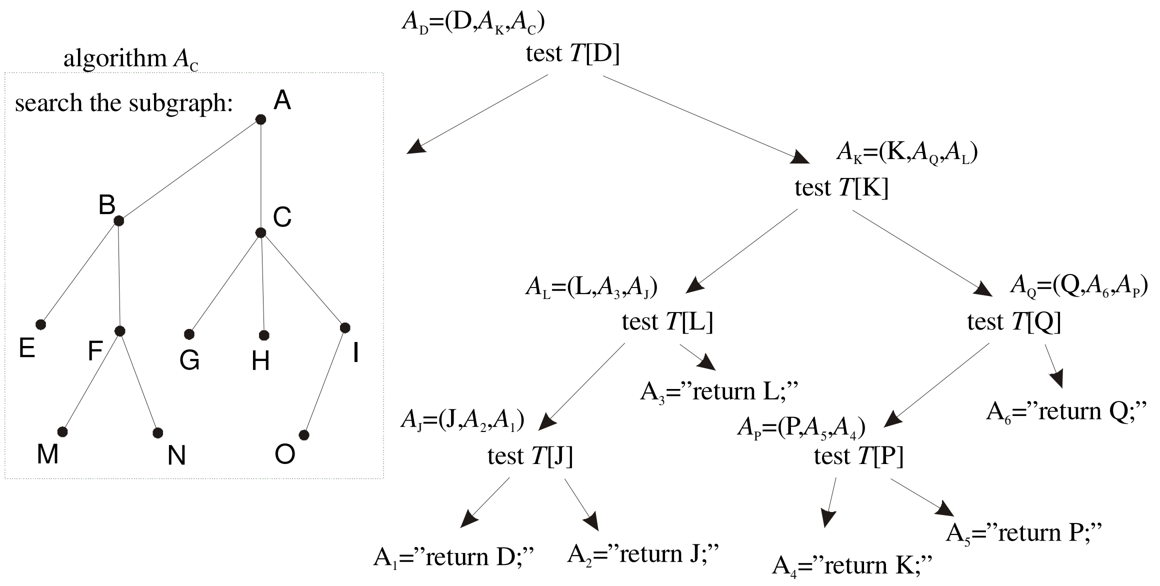

Fig. 3. The search algorithm A constructed on the basis of edge ranking of tree T shown in Fig. 2(*b*).

In the example of Fig. 1(*b*) there is one bug within the block called 'Statements 7'. We run algorithm A for graph T. Performing "test T[D]" allows to state that T[D] does not contain the bug. The domain of searching after the first test is shown in Fig. 4(*a*). Due to the coloring shown in Fig. 2(*b*) the following decision is made after the test of code corresponding to subtree T[C], which leads to the domain of searching confined as shown in Fig. 4(*b*), since the fragment of code corresponding to T[C] is free of bugs. The highest color in the subtree of Fig. 4(*b*) is put on {B,F}, which means that the succeeding round of the algorithm consists of performing the test for T[F]. The answer will be negative, which restricts our subgraph to that of the shape in Fig. 4(*c*). The next round is "test T[B]", which confines the domain to {B,E}. Finally, the algorithm A performs the last test, namely the one corresponding to T[E], which restricts the search domain to vertex B. This means that the bug is in the statements of B other than those belonging E and F or in the procedure which called B among the statements which are contained in the assertion containing B.

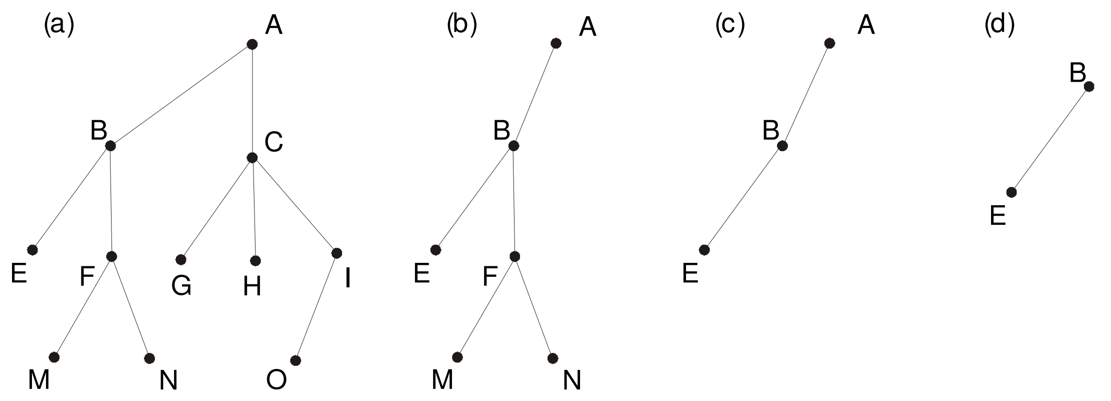

Fig. 4. The succeeding steps of the algorithm for example of Fig. 1

Summary

In this paper we assumed that the test execution times are unknown, which leads to the above theoretical model. In practice we may have some additional information about the running time of individual tests. In that case we can propose a generalization of the above approach, where a modification of the corresponding graph coloring problem plays a central role. In this modification the weights of the edges of the tree describing the structure of the program correspond to running times of the appropriate tests. This generalized version of our problem unfortunately appears to be strongly NP-hard for trees [3]. Since we do not know an efficient exact algorithm solving this problem, in practice we may only use estimations for the running times of the tests, like their complexities, and find an approximate solution. It is known an $O(\log n)$-approximation algorithm with running time $O(n\log n)$ for solving this problem [3]. In another generalization one may consider searching structures which are not acyclic and it turns out that the corresponding optimization problem is also NP-hard [4].

References

[1]H. Agrawal, J.R. Horgan, Dynamic program slicing. Proceedings of the ACM SIGPLAN'90 Conference on Programming, Language Design and Implementation, pp. 246-256, 1990

[2]Y. Ben-Asher, E. Farchi, I. Newman, Optimal search in trees. SIAM J. Comp. Vol. 6, pp. 2090-2102, 1999

[3]D. Dereniowski, Edge ranking of weighted trees. Discrete Appl. Math. Vol. 154, pp. 1198-1209, 2006

[4]D. Dereniowski, Edge ranking and searching in partial orders, Discrete Appl. Math. (to appear)

[5]J.R. Horgan, S. London, M.R. Lyu, Achieving software quality with testing coverage measures. Computer, Vol. 27, pp. 60-69, 1994

[6]M.J. Lipman, J. Abrahams, Minimum average cost testing for partially ordered components. IEEE Transactions on Information Theory, Vol. 41, pp. 287-291, 1995

[7]T.W. Lam, F.L. Yue, Edge ranking of graphs is hard. Discrete Appl. Math. Vol. 85, pp. 71-86, 1998

[8]T.W. Lam, F.L. Yue, Optimal edge ranking of trees in linear time. Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 436-445, 1998