Olympiads in Informatics, 2010, Vol. 4, 76–86

Testing of Programs with Random Generated Test Cases *

Krassimir MANEV
Department of Mathematics and Informatics, Sofia University
J. Bourchier blvd. 5, 1164 Sofia, Bulgaria
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences
G. Bontchev str. 8, 1113 Sofia, Bulgaria
e-mail: manev@fmi.uni-sofia.bg

Biserka YOVCHEVA, Milko YANKOV, Peter PETROV
Department of Mathematics and Informatics, Shumen University
Universitetska str. 115, 9712 Shumen, Bulgaria
e-mail: bissy_y@yahoo.com, m.yankov@f5bg.net, peshoto_bg@yahoo.com

Abstract. Testing of computer programs is an essential part of the evaluation process of a programming contest. It is a mix of functional and non functional testing and a specific case of the "black box" testing well known from the domain of Software engineering. The paper discusses one of the possible forms of creating test cases for a program that implement an unknown algorithm – the random generation of test cases – and the problems that could arise when random generated test cases are used for evaluation of programs submitted by participants in programming contests. Rooted trees were chosen for the start of the research because of their simplicity. More deep problems and more interesting results could be expected for general graphs and other combinatorial objects.

Key words: program testing, functional and non functional testing, "black box" testing, graphs, rooted trees, random generation, height, width and branching statistics of rooted trees

1. Introduction

In the late 70s' of the past century the intuitive *debugging* of computer programs was replaced by more systematic, mathematically reasoned and well planned, *software testing* (Mayers, 1979). With the development of the software industry and the growing of the number of proposed on the market software products testing of software became one of the essential stages of the software products' life cycle. Nowadays significant amount of the resources of the software developers are dedicated to testing of products under development. Something more, as a result of the efforts of many programmers and researchers *Software testing* became a scientific domain – sub-domain of *Software engineering* (Kaner *et al.*, 1999) – and corresponding discipline was included in the universities' curricula. One of the approaches of Software testing is the *random generation* of test cases.

*This work was partly supported by the Foundation "America for Bulgaria" and the Scientific Research Fund of Sofia University under the contract No. 247/2010.

Evaluation of the programs, submitted by the contestants during programming contests, is a specific kind of software testing. Random generation of test cases is a part of evaluation of competitive programs too. It was assumed for long years that the authors of the tasks have the best knowledge of the formulated by themselves problems and that they are able to prepare the best test cases. Some examples of Bulgarian national programming contests, and even from international contests, suggest that the author sometime underestimate the careful preparation of test cases.

During an IOI (about 10 years ago) a task on a graph $G(V, E)$ with a weight function c: $V \rightarrow N$ on the vertices was proposed. Some other function f: $V \rightarrow N$ has to be calculated and a vertex v to be found such that $f(v) = \max f(v_i)$, $v_i \in V$. Being in shortage of time one of Bulgarian contestants submitted as a solution a program that prints the vertex w, such that $c(w) = \max c(v_i)$, $v_i \in V$, and obtained enormously big amount of points. We could guess that some of the test cases were random generated.

Some years ago were popular optimization tasks (reduced versions of which are NP-complete), with "relative" evaluation. The programs that give a result nearest to the searched optimum received 100% of the points and the other programs received an amount of points proportional to the distance between their result and the best one. We proposed such task in one of Bulgarian national contests. After the grading we decided to change the random generated test cases with other random generated and the results of the contestants changed dramatically.

Examples show that, even inevitable, the random generation of test cases is full of risks. Random generation of test cases is one of the first approaches used for so called *automated test data generation* (Bird and Munoz, 1983; Duran and Ntafos, 1984) which is still in use in software industry. The advantages and negatives of the approach are widely discussed from the point of view of Software engineering. Here we will discuss some negatives of the approach when it is used for evaluation of solutions of the participants in programming contests.

The discussion is based on the example of *rooted trees*. Tasks on graph structures are among the most used for competitions in programming (Manev, 2008) and generating of test cases for such tasks is not trivial. Trees and rooted trees are simplest kind of graph structures and it is natural to start discussion with them. Because rooted trees have such interesting for random generating characteristics as height, width and branching, which are not intrinsic for not rooted trees, we prefer to start with the former for our considerations.

In Section 2 some notions from Software testing theory are given and the objectives of testing competitive programs are stressed. The necessary definitions from the Theory of graph structures are given in Section 3 as well as an example of a task on a rooted tree and some negatives following from not correct random generated test cases. All experiments in this and following sections were made with the standard function rand()GNU C/C++ programming environment. Section 4 is defining two experiments aimed to estimated the ability of rand()to generate random rooted trees. Some results of the experiments are summarized and commented in Section 5. Section 6 contains ideas for future research.

2. Testing of Programs – Some Notions and Objectives

In general, two major sub-domains are considered in the theory and the practice of software testing – *functional* and *non functional testing*. We will use the following notation and assumptions for functional testing of a program module.

...ach program module M is designed and implemented in a way to satisfy some *functional specification* F: $D \to R$, where D is the set of possible inputs and R – a set containing possible results (outputs). The set D is called *domain* of F.

Because M is implementing F, it could be considered as a function M: $D \to R$ too. We have to stress that the function M usually is a *partial* (not total), i.e., there are elements in D for which M is not defined on $\delta \in D$ – i.e., starting on an input δ, M crashes or enter an infinite loop. The set $D_M \subseteq D$ of all elements $\delta \in D$ at which $M(\delta)$ is defined is called a *domain* of M. We will suppose that R is large enough and when M stops it gives always a result ρ from R.

The ideal goal (the super goal) of the *functional testing* is to prove the *functional correctness* of M *toward* F, i.e., for given F and M to check whether $D_M = D$, and if it is true, to check whether $M(\delta) = F(\delta)$, for each $\delta \in D$. It is obvious that achieving the super goal of the functional testing in general case is very difficult and **rather impossible**. First, because it is impossible to find the domain D_M in general case – Stop-problem is undecidable for usual computational models. And second, because the size of D is usually very large.

In practice we take a reasonably small $D_T \subseteq D$ called *test set* or *test data*. Elements of D_T, called *test cases*, have to be chosen in such way that if $M(\tau) = F(\tau)$ for each $\tau \in D_T$ we could **consider** M (more or less) **functional correct** toward F. It is obvious that with such testing we are not able to check at all first condition of the functional testing – $D_M = D_F$ – because of the possibility that M crashes or enters an infinite loop for some $\tau \in D_T$. For the purpose additional efforts are necessary, as mentioned below.

The other form of testing, called *non functional*, is also interesting. Different qualities of the program module could be a goal of such testing when a commercial programming product is tested – usability, reliability (when the module provide some service), etc. One quality of specific interest for evaluation of programming contests is the CPU time elapsed by the program. For checking this quality, together with the test set D_T, a *time limit* constant L is chosen. The execution of the program on the test case is stopped if the elapsed time goes over L and the program is considered not correct for the corresponding test case.

Testing with time limit has a double purpose. First, it compensate the principle impossibility to obtain $D_M(\tau)$ in case when M crashes or enters in an infinite loop working on τ. But more important, especially for programming contests evaluation, is the possibility to estimate in such way the CPU time of the algorithm used (for definition of the notion *time complexity* see, for example, Cormen *et al.*, 2001). That is why the test set has to contain test cases of different input size, including very large test cases and the time limit to be chosen in such way that the possible algorithms of different time complexity to be identified. Random generation is the natural way for construction of the large tests.

We will finish discussion of the notions of the domain with another classification of the testing approaches. Two different cases could be defined depending of whether the source code of the module is used in the testing process or not. If the source code could be used for generation of the test set then the testing is called *transparent box* testing. If the source code could not be used – the testing is called *black box* testing.

During the evaluation of programming contest the codes of the tested programs are available. But by different reasons the usage of transparent box testing is not possible. The main of these reasons is that with the methods of transparent box testing different (by the number and the content of the test cases) test sets will be generated for the different programs (depending of the programs themselves), which is not acceptable for evaluation of programming contests. That is why in programming contests **the black box testing was always used**.

3. Testing with Random Generated Rooted Trees

Here we will give only the necessary for the rest of the paper definitions, concerning rooted trees. For the other used notions see, for example (Manev, 2008). By the classic definition, the graph $T(V, E)$ is a *tree* if it is connected and has no cycles. For the purposes of algorithmics the notion *rooted tree* is more helpful. One of the possible inductive definitions of rooted tree is:

DEFINITION. (i) The graph $T(\{r\}, \varnothing)$ is a rooted tree. r is a *root* and a *leaf* of T; (ii) Let $T(V, E)$ be a rooted tree with root r and leaves $L = \{v_1, v_2, \ldots, v_k\}$, $v \in V$ and $w \notin V$; (iii) Then T' ($V' = V \cup \{w\}$, $E' = E \cup \{(v, w)\}$) is also a rooted tree. r is a root of T' and leaves of T' are $(L - \{v\}) \cup \{w\}$.

By the *Definition* rooted trees are undirected graphs but an *implicit orientation* on the edges of the rooted tree exists. Following *Definition* we will say that v is a *parent* of w and that w is a *child* of v. Obviously each rooted tree is a tree and each tree could be rebuild as rooted when we choose one of the vertices for a root.

For each vertex v of the tree $T(V, E)$, rooted in the vertex r, we define the *height* $h(v)$ of v as the length of the path from r to v and the *height* $h(T)$ of T, $h(T) = \max\{h(v)| v \in V\}$. For each vertex v of the rooted tree $T(V, E)$ we define the *branching* $b(v)$ of v as the number of children of v and the *branching* $b(T)$ of T, $b(T) = \max\{b(v)|v \in V\}$. Let we define the ith *level* of the rooted tree $T(V, E)$, $i = 0, 1, \ldots, n-1$ as $L_i = \{v|v \in V, h(v) = i\}$, were $n = |V|$. Then the *width* $w(T)$ of T is defined as $w(T) = \max_{i=0,1,\ldots,n-1}\{|L_i|\}$.

One of the simplest ways to represent the tree $T(V, E)$ with $V = \{1, 2, \ldots, n\}$, rooted in the vertex r, is the *list of parents* – a vector $(p_1, p_2, \ldots, p_n)$, where p_i is the parent of $i \neq r$ and $p_r = 0$. The list of parents could be implemented in an array p[] of integers and is very convenient when it is necessary to build a rooted tree.

Suppose that for a programming contest the following task is prepared:

Task. A rooted tree T with n vertices, labeled fro 1 to n, is given with its list of parents, $3 \leqslant n \leqslant 1000$. Let the root of the tree be the vertex labeled with 1. Write a program to find a vertex of T with maximal height.

Following *Definition* we built a very simple generator of random rooted trees without any additional conditions on the structure of the generated objects. The essential part of the source is given below:

```
...
int n\textbf{,} p[MAXN], i;
p[1] = 0; p[2] = 1; // vertex 1 is the root
for(i = 3; i <= n; i++)
   p[i] = rand() % (i - 1) + 1;
...
```

For evaluation of contestants' program we decide to make 100 tests – of $10, 20, \ldots, 1000$ vertices, respectively and for each test to assign 1 point. What will happen if a contestant has submitted a program, which for each test case print n – the maximal label of a vertex? We generated 100000 random test sets with 100 test cases each. The average result of the contestant is 5.29 points which is too much for the invested efforts. But even worst is that among the 100000 random generated there is a test set for which the contestant will obtain 16 points.

The obvious defect of the used generator is that the vertex with label n is always a leaf of the tree and the chance to be a leaf with a maximal height is big enough. To eliminate such defect we could append at the end:

```
for(i = 2; i <= n; i++)
    p[i] = perm(p[i]);
```

where the function `perm()` calculate some random permutation of the numbers $2, 3, \ldots, n$. And then to eliminate the test cases in which the vertex n is a leaf with maximal height. In such way the trivial program, which print he value of n will not obtain points. For the experiments described below we are using this ameliorated version of our simple generator.

4. Experiments on the Randomness of Generated Rooted Trees

In the following our goal is to check the "randomness" of the obtained by our generator rooted trees. For the purpose let us first stress that the upper bound 1000 for the parameter n is too small. Quite reasonable rooted trees could have 100000 vertices because the input data for such rooted tree will be less than 700 KB and reading such input will not increase dramatically the time necessary for execution of the chosen algorithm. Something more, suppose that the author have in mind two different algorithms – a naïve one of time complexity $O(n \log n)$ and sophisticated one of time complexity $O(n)$. With a black box testing it will be difficult to identify which of the algorithms is used by the contestant if

the maximal number of the vertices is about 10000, for example. Especially in the case when the second algorithm has relatively big multiplicative constant. The identification of the used algorithm when the rooted tree is with 100000 vertices is trivial.

In order to obtain relatively *reliable test set*, i.e., a test set that is able to convince us in the functional correctness of the tested program we have to answer some important questions that arise on this stage:

- How many tests cases are necessary?
- Which will be the size of the rooted tree in each test case?
- Is it necessary to incorporate some specific structure in the generated rooted trees?

In Bulgarian national contest each task is evaluated with 100 points. That is why the practice is the number of test cases in the test set to be some divisor of 100 grater then or equal 10 – 10, 20, 25, 50 or 100 – and for each test case, which successfully passed the testing 10, 5, 4, 2 or 1 point to be assigned, respectively. It is obvious that such practice is not related at all with the specific task and the expected algorithms. Two major negatives trivially follow from such approach. First, it is quite possible that the chosen number of test cases is not enough even for an average confidence in the correctness of the tested algorithm and its implementation. And second, it is possible that all generated in such way test cases to be, in some sense, equivalent and to happen that the test set punish some kinds of mistakes and tolerate other kinds.

Having in mind the last reason some authors are limiting themselves to generating by hand only test cases with a small number of vertices in order to incorporate in them some structure corresponding to the particularities of the task. Such practice is also unacceptable because it could lead to receiving huge amount of points for trivial (as time complexity) and even wrong solutions. The only way to escape it is *the automatic generation of the "big" test cases*.

There are two ways of generating test case with a program. First of them is the random generation of many test cases with one program having as a single parameter the size of the generated object. The second is to create a specific program for each test case that is incorporating some structure in the generating object or to use many parameters in the generator, in order to escape equivalent test cases. The second possibility is obviously consuming much more of the author's time and is rarely used. Because of frequent using of random generation of test cases a question arises: *how reliable are the test cases generated randomly without incorporating any structure elements in the generated objects or tuning of various parameters*?

Below we are describing some experiments with small random generated rooted trees in order to observe some trends and to establish some directions for future research of the problems.

For generating of full set of rooted trees with n vertices, without excluding the isomorphic cases, a very simple program was built too. It generates all subsets of $n-1$ edges and eliminates each of the subsets that forms disconnected graph or graph with a cycle. The obtained trees are transformed to rooted trees choosing for a root the vertex 1.

Two kinds of experiments were organized in order to obtain some statistics about the randomly generated rooted trees. The main characteristics mentioned above – the height, the branching, and the width of the rooted trees – were observed.

The goal of the first experiment was to check the ability of the standard function rand()to build really random rooted trees. For the purpose, first, all $RT(n)$ rooted trees with at most 8 vertices were generated, without eliminating the isomorphic cases (we postpone eliminating of the isomorphic cases for the moment). For each generated rooted tree with n vertices the three characteristics were calculated and the corresponding distributions $(h_1, \ldots, h_{n-1})$, $(b_1, \ldots, b_{n-1})$ and $(w_1, \ldots, w_{n-1})$ were obtained, where h_i, b_i and w_i are the number of rooted trees of n vertices with height, branching and width equal to i, respectively.

Then the same number $RT(n)$ of random rooted trees with n vertices was generated and the corresponding distributions $(H_1, \ldots, H_{n-1})$, $(B_1, \ldots, B_{n-1})$ and $(W_1, \ldots, W_{n-1})$ were compared with the distributions $(h_1, \ldots, h_{n-1})$, $(b_1, \ldots, b_{n-1})$ and $(w_1, \ldots, w_{n-1})$. Our preliminary expectations were that the function rand(), dedicated to generate random numbers in a given linear interval, will not be able to generate really random rooted trees – objects that have nonlinear structure. The obtained results are given in the next section.

The goal of the second experiment was to check is it possible with a small number of random generated rooted trees to obtain statistics for the observed parameters which are close enough to the statistics of the full set.

For the second experiment only the rooted trees of 7 vertices was used. Random subsets of 20, 30, 40 and 50 such trees were generated. For each subset the relative distribution (in %) of the observed characteristics was calculated and compared with the corresponding, also relative, distribution for the full set. Our preliminary expectations were that small subsets of random generated rooted trees will not be able to "cover" statistically the full set. The obtained results of this experiment are also given in the next section.

5. Experimental Results and Comments

5.1. *Statistics of Random Generated Rooted Trees with up to 8 Vertices*

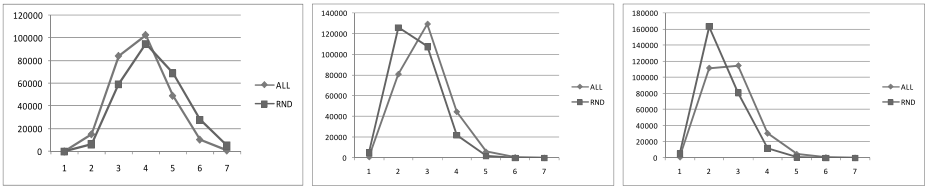
Because there is a single rooted tree (with a root in the vertex with label 1) when the number of the vertices n is 1 or 2, the smallest interesting case is $n = 3$. There are three different rooted trees in this case (two of them are really isomorphic but we are considering isomorphic trees with different labeling of the vertices as different). The set of random generated 3 rooted threes coincided with the set of all trees.

When $n \geqslant 4$, as it was expected, the distributions of the trees and random trees become different (see the results for $n = 8$ in Table 1 and on Fig. 1.

It is possible to classify the deviation of the random generated distribution from the real distribution (for all three observed characteristics) as "displacement of the peak". For

Table 1
Rooted trees with 8 vertices

k	Trees of height k		Trees of width k		Trees of branching k	
	ALL	RND	ALL	RND	ALL	RND
1	19	1	857	5040	857	5040
2	15166	6321	80976	126000	111394	163170
3	84067	59472	129403	107520	114421	80850
4	102563	94710	44581	21875	30459	12005
5	49016	68880	5897	1659	4583	1029
6	10456	27720	411	49	411	49
7	857	5040	19	1	19	1



a. Height b. Width c. Branching

Fig. 1. Rooted trees with 8 vertices.

the heights – the random generated values leftmost of the peak of the distribution curve are less then the real values and the random generated values rightmost of the peak are grate than corresponding real values. We will call such deviation *right displacement* of the peak. For width and branching the displacement is in the opposite direction – let us call it *left displacement*. Statistically, as big the height of the rooted tree is as small is its width (or branching), which explain the opposite directions of the displacements.

We have not an explication of the observed displacement but the trend is very clear. That is why we could suggest to incorporate minor correction in the generator when random tests are generated without any additional constrains on the structure of the rooted trees – to lift a bit the average height (or to decrease width/branching) of the generated objects.

5.2. *Statistics for Small Subsets of Random Generated Rooted Trees with 7 Vertices*

The results of the second experiment for the distribution by height are summarized in Table 2. In order to compare the obtained distributions by height they are transformed in percentages. As a criterion for *similarity* of two distributions was use the traditional χ^2 criterion – the sum of the squares of the differences between real percentage and those obtained by the random generation, divided to the real percentage (last element in each of the columns from 8 to 13).

Table 2

Random generated rooted trees with 7 vertices – height distributions

h	ALL	RND	R20	R30	R40	R50	%ALL	%RND	%R20	% R30	%R40	%R50
1	1	13	0	0	0	0	0.01%	0.08%	0.00%	0.00%	0.00%	0.00%
2	1056	1829	2	5	5	2	6.28%	10.88%	10.00%	16.67%	12.50%	4.00%
3	5550	6819	6	13	12	14	33.02%	40.57%	30.00%	43.33%	30.00%	28.00%
4	6240	6063	9	8	20	27	37.13%	36.07%	45.00%	26.67%	50.00%	54.00%
5	3240	1872	3	4	3	6	19.28%	11.14%	45.00%	13.33%	7.50%	12.00%
6	720	211	0	0	0	1	4.28%	1.26%	0.00%	0.00%	0.00%	2.00%
	16807	16807	20	30	40	50	0%	$\chi^2 = 12$	$\chi^2 = 43$	$\chi^2 = 29$	$\chi^2 = 22$	$\chi^2 = 13$

Table 3

Random generated rooted trees with 7 vertices – width distributions

W	ALL	RND	R20	R30	R40	R50	%ALL	%RND	%R20	% R30	%R40	%R50
1	720	211	0	0	0	1	4.28%	1.26%	0.00%	0.00%	0.00%	2.00%
2	9720	7539	9	14	23	33	57.83%	44.86%	45.00%	46.67%	57.50%	66.00%
3	5580	7233	10	13	12	12	33.20%	43.04%	50.00%	43.33%	30.00%	24.00%
4	750	1662	1	3	5	4	4.46%	9.89%	5.00%	10.00%	12.50%	8.00%
5	36	149	0	0	0	0	0.21%	0.89%	0.00%	0.00%	0.00%	0.00%
6	1	13	0	0	0	0	0.01%	0.08%	0.00%	0.00%	0.00%	0.00%
	16807	16807	20	30	40	50	0%	$\chi^2 = 18$	$\chi^2 = 16$	$\chi^2 = 17$	$\chi^2 = 19$	$\chi^2 = 8$

The experiment shows the expected amelioration of the similarity when the number of random generated objects is growing from 20 to 50 – values of 43, 29, 22 and 13. Applying the same χ^2 criterion on the generated in the previous experiment set of 16807 random trees of 7 vertices we could see that the significant increasing of number of generated trees do not lead to crucial amelioration of the distribution – the value $\chi^2 = 12$ in this case is practically the same as the value for 50 random generated trees.

As it is also expected the single tree with $h = 1$ had no real chance to be generated randomly. On the other side of the scale – a rooted tree with the maximal $h = 6$ was for the first time generated randomly only in the case of 50 generated trees. That means – if the task is such that the solution has to be tested on the marginal trees of height 1 and 6 then such tests have to be generated by hand.

From the other two experiments only the obtained distributions for the width of random generated subsets of 20, 30, 40 and 50 rooted trees is presented (see Table 3), because the results for distributions for branching, as it was mentioned above, is very similar.

Measuring the similarity with χ^2 criterion we could see that in the case of width distribution the sets with 20, 30 and 40 rooted trees approximate almost identically the real distribution with (similarity of 16, 17 and 19, respectively) and the set of 50 random generated trees approximate real distribution much better – similarity 8. But the similarity

of the width distribution of generated for the first experiment set of 16807 random rooted trees happens to be 18 – not better than in the case of 20 and 30 random generated trees. So, our conclusion that generating of too much test cases is not necessary is confirmed by the width distribution too.

6. Conclusions and Ideas for Further Research

Generating of random test cases for evaluation the programs of the participants in programming contests has no alternative. But with not controlled random test generation we could obtain test sets that do not permit a fair evaluation process – with to many similar test cases or with missing of some important test cases. When a black box testing is performed it seems important that the test set contains test cases, which cover the whole variety of values of the intrinsic characteristics of the generated objects. Our experiments, even very simple, confirm this conjecture.

Discussion in this paper has some particularities that have to be stressed. First, we are considering all (i.e., labeled) rooted trees *without excluding isomorphic cases*. Much more reliable results could be obtained if we consider only one of set of isomorphic trees – i.e., not labeled rooted trees. For the purpose corresponding software could be created for eliminating isomorphic cases.

Second, our experiments are made on trees with very small number of vertices. As it was mentioned above, the interesting for evaluation of competitive programs are trees with hundred thousand of vertices. It is obvious that *the same experiments are impossible for such large trees* because of impossibility to generate all of them in real time and to collect the necessary statistics. For the purpose some combinatorial and statistical results for the distribution of the observed characteristics have to be attracted.

And third, *each generation* in the described experiments *was performed once*. From statistical point of view it will be better if each of these generations was repeated as many times as possible in order to eliminate the statistical mistake. The reason to do the former was to simulate the process of generation of real test cases – the author of random test cases usually performs the generation once. And we would like to see what will happen in such case.

The presented experiments and corresponding discussions was made on the example of rooted trees. But the problems will be the similar with *generation of any object* that could be used as a test case for a task proposed for programming contests – general graph structures, sequences of numbers, strings, set of geometric objects, etc.

Eliminating the mentioned above limitation of the presented experiments as well as investigation of many other aspects of the problem could be subject of a future research. Better knowledge of the statistical distribution of important characteristic of included in competitive tasks discrete objects will permit us to generate better test sets even when a random generation is used. It is worth to make the necessary efforts in this direction in order to obtain more adequate evaluation of contestants' program.

References

Bird, D., Munoz, C. (1983). Automatic generation of random self-checking test cases. *IBM System Journal*, 22(2), 229–245.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. (2001). *Introduction to Algorithms*. MIT Press.

Duran, J., Ntafos, S. (1984). An evaluation of random testing. *IEEE Trans. on Software Engineering*, SE-10, 438–444.

Kaner, C., Falk, J., Nguyen, H.Q. (1999). *Testing Computer Software*, 2nd Ed. John Wiley and Sons, Inc.

Manev, K. (2008), Tasks in graphs. *Olympiads in Informatics*, 2, 90–104.

Myers, G.J. (1979). *The Art of Software Testing*. John Wiley and Sons.

K. Manev is a professor of discrete math and algorithms in Sofa University, Bulgaria, PhD in computer science. He was a member of NC for OI since 1982 and president of NC from 1998 to 2002. He was member of the organizing team of First and Second IOI, leader of Bulgarian team for IOI in 1998, 1999, 2000 and 2005. From 2000 to 2003 he was an elected member of IC of IOI. Since 2005 he represents the Host country of IOI'2009 in IC of IOI.

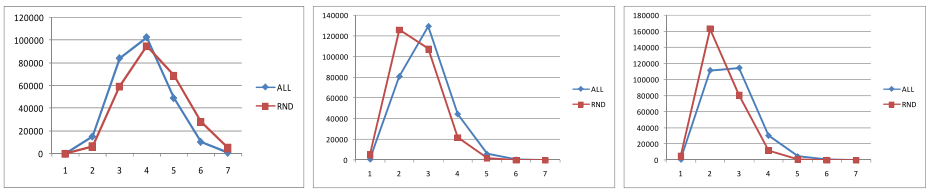
B. Yovcheva is assistant professor of informatics in Shumen University, PhD in education in iInformatics. She is creator of the private school A&B of Math and Informatics in Shumen and teacher of some of the most successfull Bulgarian participants in International and Balkan Olympiad in Informatics.

M. Yankov is an assistant professor of informatics in Shumen University. He is a coach of the students from Shumen University for competitions in programming.

P. Petrov is an young teacher in informatics in A&B School of Shumen. He is coach of the regional team of Informatics for 15.5 years of Shumen. He is an author/co-author of a book on algorithms and algorithmic problems. Since 2009 he is an assistant professor of informatics in Shumen University and coach of the students from Shumen University for competitions in programming.

Table 1
Rooted trees with 8 vertices

k	Trees of height k		Trees of width k		Trees of branching k	
	ALL	RND	ALL	RND	ALL	RND
1	19	1	857	5040	857	5040
2	15166	6321	80976	126000	111394	163170
3	84067	59472	129403	107520	114421	80850
4	102563	94710	44581	21875	30459	12005
5	49016	68880	5897	1659	4583	1029
6	10456	27720	411	49	411	49
7	857	5040	19	1	19	1



a. Height b. Width c. Branching

Fig. 1. Rooted trees with 8 vertices.

the heights – the random generated values leftmost of the peak of the distribution curve are less then the real values and the random generated values rightmost of the peak are grate than corresponding real values. We will call such deviation *right displacement* of the peak. For width and branching the displacement is in the opposite direction – let us call it *left displacement*. Statistically, as big the height of the rooted tree is as small is its width (or branching), which explain the opposite directions of the displacements.

We have not an explication of the observed displacement but the trend is very clear. That is why we could suggest to incorporate minor correction in the generator when random tests are generated without any additional constrains on the structure of the rooted trees – to lift a bit the average height (or to decrease width/branching) of the generated objects.

5.2. *Statistics for Small Subsets of Random Generated Rooted Trees with 7 Vertices*

The results of the second experiment for the distribution by height are summarized in Table 2. In order to compare the obtained distributions by height they are transformed in percentages. As a criterion for *similarity* of two distributions was use the traditional χ^2 criterion – the sum of the squares of the differences between real percentage and those obtained by the random generation, divided to the real percentage (last element in each of the columns from 8 to 13).