Evaluation of programs in Informatics Contests: case of implementation of graph algorithms

Jūratė Skūpienė
Institute of Mathematics and Informatics
Akademijos st., 4, LT-08663 Vilnius, Lithuania
+370-5-2109344
jurate@ktl.mii.lt

Antanas Žilinskas
Vytautas Magnus University
Vileikos str. 8, LT-44404 Kaunas, Lithuania
+370-6-8601234
antanasz@ktl.mii.lt

ABSTRACT
During various high school olympiads and competitions in informatics there are presented a lot of tasks with graphs. Contestants express their algorithms in programs which are graded using *black-box* method. The code analysis that would help to understand the algorithm is time consuming and therefore not applied in the grading process. To assist it graph visualization tool IOVIZ was created. It visualizes graphs implemented in Pascal source programs. The tool was tested with programs designed by the competitors during Lithuanian Olympiads in Informatics.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education.*

General Terms
Algorithms, Measurement, Performance, Human Factors.

Keywords
Informatics olympiads, programming contests, visualization, program animation, graph algorithms.

1. EVALUATION PROBLEMS IN INFORMATICS OLYMPIADS
There are many informatics (computing, programming – there is a variety in naming) competitions intended for high school students. The most prestigious competition is International Olympiad in Informatics (IOI). IOI is an annual international informatics competition for individual contestants from various invited countries, accompanied by social and cultural programs, initiated at the 24th General Conference of the UNESCO in 1987 [13]. There is a variety of other competitions that are organized in a quite similar way to IOI's. For more references [12], [9], [18], [1].

IOI competitions focuses on informatics problems of an algorithmic nature. The IOI contestants are required to express their algorithms in one of the allowed programming languages and they must engineer their programs to run flawlessly, because marking is based on automated execution [20]. The allowed programming languages are Pascal, C and C++.

Formal algorithm correctness verification methods that are used in the scientific community are not suitable in the informatics olympiads due to timing restrictions. In most of the competitions due to a vast amount of solutions grading is done using so called black-box method, i.e. it is done automatically, by running programs with various test inputs designed in advance, no sources are analyzed. For each correct output, produced within certain pre-defined limits, points are awarded. It is stated in [5]: „In practice, sometimes an incorrect solution scores far too many points, sometimes an asymptotically better solution scores less points than a worse one, and sometimes a correct solution with a minor mistake scores zero point." We refer for more references on grading problems to [17], [5], [22].

Therefore there are many cases where the authors of the tasks or the evaluators might be willing to analyze some programs designed during the contest. However, analysis of program source in order to understand the algorithm is not always easy due to a variety of cultural elements in their programs (e. g. the variables are named by the contestant's native language words) as well as different (and sometimes poor) programming style. A tool that could simplify understanding and analysis of competitors' programs, their tricks and faults would be valuable.

Tasks, which comprise graphs are very common in various olympiads and competitions in Informatics. In the final round of Lithuanian National Olympiad such tasks comprise over 20% of all tasks [2]. The percentage in the Baltic Olympiads is similar or even higher. In IOI'2005 three out of six tasks contained graphs either in their description or it was meant in model solutions [10]. It might be assumed that tasks with graphs occur in nearly every informatics olympiad. However neither the word graph nor other related terms (e.g. graph vertex, edge, etc.) usually are used directly in task formulations. Typically, they are described indirectly, using some kind of metaphors [21].

2. VISUALIZATION AIDED EVALUATION
Visualization can help to aid programs that contain implemented graphs by visualizing graphs and their behavior during program execution (e.g. which graph vertexes have been visited, etc).

An important practical task in creating algorithm visualizations is to specify how the visualization is connected or applied to the algorithm. There are two main approaches to algorithm visualization [15], [3]. One of them is based on *interesting event* paradigm. The important or interesting events in the program source have to be identified and calls to visualization procedures have to be inserted into the source. The second approach, called *state mapping,* creates visualization automatically depending upon the values of the program variables.

The choice of the approach is influenced by the conditions under which the contestants programs are analyzed. On the one hand the evaluators or the task designers have no prior knowledge of how

it is designed. Moreover, the program source can have a poor programming style. On the other hand they are expected to know the task formulation and background as well as some model solutions. There is also a wide spread tradition in the olympiads in informatics – the evaluators do not change competitors source code.

Among the major event driven approach drawbacks are the unavoidable source modification (or at least its augmentation) and the requirement to know the source code quite well in order to identify all the interesting points. This implies that only the state driven approach to visualization of competition programs is possible. State driven designs can create visualization without (much) code modification, but they cannot be easily customized. Abstractions are more difficult to represent, and state driven visualizations lack smooth transitions [19].

Conventional debuggers also have some features of state mapping approach, for they provide variable values as the values change during program execution [3]. This similarity of approaches determined that IOVIZ was designed as a debugger with visualization possibilities. The user willing to get graph animated has to interpret the meaning of variables, identify which of them represent graph data structures and choose the method of graph implementation from the list of available ones.

When designing IOVIZ, some efforts were made to create a tool, simple to use. A more complicated tool might be met with some resistance as it happens in other cases with algorithm visualization tools. Especially taking into account that informatics olympiads are not daily event and take place just two-three times a year. The reasons for unwillingness to accept new algorithm visualization tools were presented in [8].

Once the program is run with the debugger in order to understand how the program works and what kind of algorithm was encoded, it is enough to analyze the program execution with small data. Large data sets basically help to determine how effective the solutions are and they are highly important in automatic grading but not in a step by step program execution analysis. Therefore the IOVIZ is not intended to display large graphs.

There are created other environments for graph visualization. However they are meant for the tutors and for teaching purposes and they require some intervention into the source code. An example of such an environment can be EVEGA (An Educational Visualization Environment for Graph Algorithms) [16].

IOVIZ acts as a simple Pascal IDE. IOVIZ has only the very basic features and can't be considered as a replacement of Pascal IDE for application development. IOVIZ uses FPS package which has integrated FreePascal compiler and GDB debugger. More reference on FPS [6]. In some specific cases GDB support for Pascal is limited [7], and those limitations are inherited in IOVIZ.

Motivation for Pascal. As distinct from universities, *Pascal* is still popular in international high school olympiads competitions, even though its popularity is slowly decreasing. Pascal is the dominating programming language in Lithuanian National competitions. Survey of IOI'2004 reveals that 46.58% contestants indicated that they used Pascal debugger during the competition [11]. In BOI'2005 (Baltic Olympiad in Informatics) 49.1% of contestants used Pascal, while in Lithuanian National competition in 2005 89.3% of contestants used Pascal.

GDB also supports C/C++, the other two languages allowed in informatics olympiads. Therefore support for those languages can be added to IOVIZ as well.

3. ANALYSIS OF GRAPH IMPLEMENTATION IN PROGRAMS DESIGNED DURING CONTESTS

There are two most common computational representations of graphs: *adjacency lists* and *adjacency matrices*. These representations of graphs can be implemented in different ways, e.g. an adjacency list can be encoded using pointers, array of records, two-dimensional array. The competitors might think of many other (not necessary reasonable) implementations. The graphs in the tasks or their solutions can also be very different and have various features or attributes. Moreover, there are some tasks in the olympiads, where a good solution should be memory effective and a common implementation of graphs and other data structures would not of work because of predefined memory limitations. Example of such a task could be *The Troublesome Frog* used in IOI'2002 [14]. We investigated the real graph implementations designed by the competitors during informatics contests.

We analyzed three tasks presented in Lithuanian Olympiad in Informatics and their solutions designed during the contest. Lithuanian Olympiads in Informatics are organized under IOI model and therefore inherit its grading challenges and problems. Below the brief formulations of the tasks are presented and commented in graph terms.

*Task 1. Acquaintance*s (2001, Final round). N persons are expected to attend a party. It is known that if any two persons have a common acquaintance (or make one during the party) they will get acquainted during the party. However, one person didn't come to the party and as a result the people split into groups who had no common friends, i.e. it became impossible for all the party attendants to become acquainted. Write a program to find a person whose illness might cause such a situation. In other words friendships relations represent a connected undirected graph. Find an articulation vertex (whose removal disconnects the graph). It is known that the graph contains at least one articulation vertex.

Task 2. Virus (2003, Final round). Computer network makes an undirected (not necessarily connected) graph. Each computer either has anti-virus protection or not. The virus starts from computer A and passes in parallel all the edges leaving A. It travels through network and destroys every edge it passes through. If the virus reaches the computer with anti-virus protection, then both the virus and the anti-virus protection are destroyed. Write a program to find when (and if) the virus reaches computer B.

Task 3. Computer Network (2005, Final round). A set of computers and switches have to be connected into one connected network, i.e. to form a *tree*. Each computer has to be connected to exactly one switch, while to one switch there can be connected many devices (either computers or switches). Given the expenses of connecting every possible pair of devices, write a program to find the network connection with the lowest expenses.

In IOVIZ there were implemented three graph implementations: *Adjacency lists implemented as array of records*, *Adjacency lists*

implemented as two-dimensional array and *Adjacency matrix implemented as two-dimensional array*.

Let us make several notes on others not yet implemented in IOVIZ graph representation cases. Set based graph implementations use Pascal *set* data type. They are not very common, because a *set* in Pascal cannot contain more than 256 elements, and therefore this type is not suitable to implement graphs containing more vertices. Pointers in graph implementations were used only for memory saving reasons, e.g. instead of a two-dimensional array, there was a pointer to the two-dimensional array.

Graph representation as list of edges in some cases was used to represent a tree. In other cases the contestants might have been affected by input data format, where the graph was presented as list of edges. This is not implemented in IOVIZ, and should be considered as one of guidelines for IOVIZ improvement together with other above mentioned implementations.

Complicated or unusual (e.g. array of strings) representations of graphs make a very small percentage of total programs and they are not considered for implementation in IOVIZ.

4. GRAPHS' VISUALIZATION

Graphs in IOVIZ tool are visualized in such a way that for the evaluator it would be as easy as possible to use it. IOVIZ has main commands of a debugger, i.e. only the features that are important when the program is traced in order to understand how it works.

Data structures (i.e. variables), which are displayed as graphs can be considered as another type of watches. The user has to indicate the variable(s), which represents the graph(s) and how the graph is implemented, i.e. to choose from the available implementations. All the visualization settings can be changed or updated during debugging (tracing). In general it is difficult to predict which graph layout is most suitable for a particular task (data), so the user can drag vertices and edges and modify the layout.

Animations which change graph layouts automatically are confusing, because it is complicated for the user to follow all the changes, happening on the screen [4]. IOVIZ does not change graph layout automatically. However the graph might loose its good layout when many new vertexes during algorithm execution are added. The choice is left for the user, who decides when and if to rearrange the graph automatically or manually.

The analysis of programs, designed during contests, shows a tendency to avoid more complicated data structures to represent complicated graphs. Instead several separate data structures are created and the graph is assembled from the components.

For example in *Network* task it is required to find one graph which connects all the computers and switches into one network. However, in many competitors' programs two different graphs were created. One graph – to represent computer–switch connections, another – switch interconnections.

Table 1. Graph implementation in analyzed programs

		Task		
		Acquaintances	***Virus***	***Network***
Total No of programs		42	32	117
Adjacency lists	Array of records	6	12	-
	Two-dimensional array	12	-	-
	Two-dimensional array using pointers	-	-	1
	Two-dimensional bit array	1	-	-
	Array of records using pointer	-	-	-
Adjacency matrix	Two-dimensional array	10	20	47
	Two-dimensional array using pointers	5	-	2
List of edges	Two-dimensional array	-	-	27
	Several arrays	-	-	16
	Array of records	-	-	29
	Array of strings	-	-	6
	Stored in a text file	-	-	4
	Other complicated structure	-	-	9
Set based implementation	Array of sets	4	1	-
	Array of records, neighbouring vertices stored in sets	1	2	-
Bipartite graph	Array	-	-	21
	Array of records	-	-	1
Graph implemented in different ways		2	3	48
Graph not implemented		6	-	16

Another example comes from *Virus* task. If the graph was implemented as an array of records, then the information whether the computer (graph vertex) has anti-virus protection or not was stored in one of the record's fields, i.e. in the same data structure. If graph was implemented as a two-dimensional array, then additional vertex attributes (existence of anti-virus protection) were implemented as a separate one-dimensional array.

Table 2. Statistics of graph components implemented separately from the main graph

		Task		
		Acquaintances	*Virus*	*Network*
No of programs with graphs implemented		36	32	101
No of programs without extra components		5	7	8
No of programs with	1 component	17	7	29
	2 components	10	7	30
	3 components	4	6	18
	≥4 components	-	5	17
Vertex components	Set	8	3	1
	One dimensional array	7	11	4
	Dynamical list	1	-	-
	Array of enumerations	-	-	1
Vertex list components	One dimensional array	19	16	26
	Array of records	1	5	-
	Array of records using pointer	-	1	-
Graphs as components		2	3	81

The components encountered were classified into the following categories. *Graph components* are graphs themselves and can be added as components to other graphs and they can be treated as separate graphs as well. *Vertex components* are lists of all graph vertexes with assigned values to each vertex (see Fig 1 and 2). They can be added as components to other graphs, though they can't be visualized separately in IOVIZ. *Vertex list components* are lists of vertexes without assigned vales, i.e. being included into the list already means possession of certain attribute. They can also be added to graphs as components and can't be visualized separately (see. Fig 3). The color is utilized to portray this type of components. IOWIZ allows joining to the graph at most two such components.

IOVIZ allows visualizing several different graphs at the same time, and they can be assembled from various components. The same components can be added to different graphs at the same time. When assembling graphs there has to be one-to-one mapping in all the components (e.g. vertex no 1 should keep the same index in all the components). Among the analyzed programs we found no cases where this mapping would be violated. However in Network task when the resulting network was presented as two separate graphs and the same indexes were used in both cases, there was no possibility to join them. This is not implemented in IOVIZ.

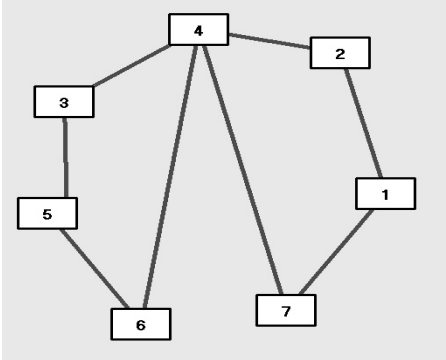

Figure 1. Task: *Acquaintances*. Graph image after reading input data.

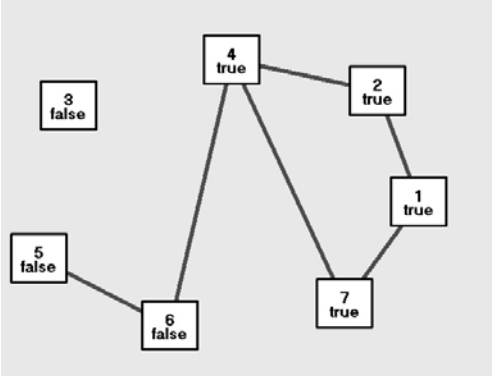

Figure 2. Task: *Acquaintances.* The same graph as in Fig 1; Edges, leaving from vertex 3 were removed and the connectivity of the remaining graph is checked by recursively traversing it; vertex component indicates whether the vertex was already reached.

There also were encountered cases where graph was implemented as two-dimensional array and where a column of the array contained vertex components. This can't be identified automatically and IOVIZ foresees a possibility to mark one column or a row as not displayable one or as a vertex component (see Figures 4 and 5).

5. CONCLUSIONS AND FUTURE WORK

Graph visualization tool IOVIZ was created to adapt the needs of evaluators and problem designers that have to analyze Pascal programs with graphs implemented, designed by contestants during informatics olympiads and other similar contests. IOVIZ satisfies the most essential requirement: state-mapped visualization, which does not require good understanding of program being analyzed and visualization of graphs implemented in competitors' programs.

The investigation of solutions (programs) to three graph problems shows the most common ways of graph implementation and a tendency to implement more complicated graphs by decomposing graphs into separate structures. IOVIZ allows assembling graph from various components.

There are a few ways to continue the research and to improve IOVIZ. Several other most common graph representations should be implemented in IOVIZ, more flexibility of graph assembling from the components can be given and support for C/C++, the other two programming languages allowed in high school informatics olympiads can be added. In this research there were analyzed only programs solving the tasks of Lithuanian Olympiad in Informatics. Analysis of graph implementations in programs designed by contestants during higher level olympiads, e.g. Baltic or International Olympiads in Informatics might show interesting results.

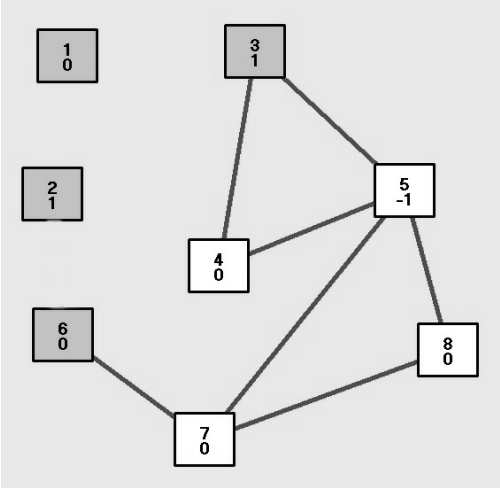

Figure 3. Task: *Virus.* Graph vertexes have two additional components: 1) *vertex component*: one dimensional array where each vertex is assigned either –1 (computer has anti virus protection) or 0 (no anti-virus protection) or the time when the computer was infected; 2) *vertex list component:* set of infected vertices (they are colored in grey). Computers 1 and 2 have been disconnected by the spreading virus.

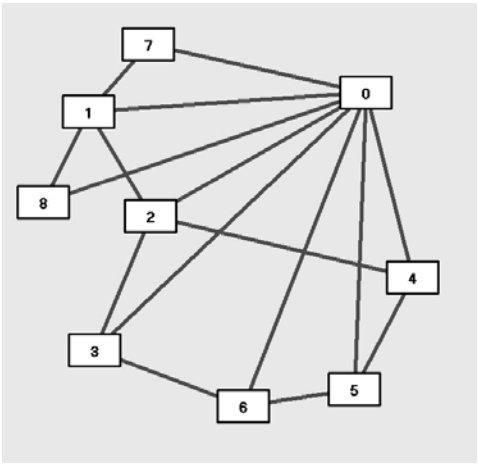

Figure 4. Task: *Acquaintances. View of graph after reading input data; Graph was implemented as two-dimensional array, however, the 'zero' column was introduced to store the degree of a vertex. Therefore the graph is misrepresented.*

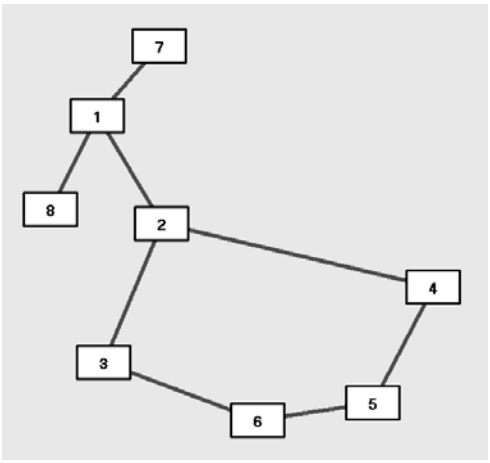

Figure 5. Task: *Acquaintances. 'Zero' column is marked as not displayable. Now the graph representation is correct.*

6. REFERENCES

[1] Boersen R., Phillips M., Programming contests: Two innovative models from New Zealand. *International Workshop, Perspectives on Computer Science Competitions for (High School) Students*, *http://www.bwinf.de/competition-workshop/*; January 25-28, 2006.

[2] Dagienė V., Skūpienė J., Analysis of solving methods and complexity of Algorithmic problems in Lithuanian Olympiads in Informatics. In *Lithuanian Mathematical Journal*, ISSN 0132-2818, 43(spec. ed.), 2003, 209-214. *(in Lithuanian).*

[3] Demetrescu C., Finocchi I., Stasko, J, Specifying Algorithm Visualizations: Interesting Events or State Mapping? *In Software Visualization*, Springer-Verlag Berlin Heidelberg, 16-30.

[4] Diehl S., Görg C., Kerren A. Animating Algorithms Live and Post Mortem. In *Software Visualization*, LNCS 2269, 2002, 46–57.

[5] Forišek M. On suitability of programming competition tasks for automated testing. In *International Workshop, Perspectives on Computer Science Competitions for (High School) Students*, http://www.bwinf.de/competition-workshop/; January 25-28, 2006.

[6] *Free Pascal System FPS* http://aldona.mii.lt/pms/fps/en/index.html.

[7] *GDB: The GNU Project Debugger*, *GDB Documentation*, http://www.gnu.org/software/gdb/documentation/;

[8] Hundhausen C., Douglass S., Stasko J., A Meta-Study of Algorithm Visualization Effectiveness. In *Journal of Visual Languages and Computing, Vol. 13, No. 3*, June 2002, 259-290.

[9] *International Workshop, Perspectives on Computer Science Competitions for (High School) Students*, http://www.bwinf.de/competition-workshop/ , January 25-28, 2006.

[10] *IOI'2005 Tasks and Solutions*, ISBN 83-917700-9-5, 2005.

[11] *IOI'2004 material* http://olympiads.win.tue.nl/ioi/ioi2004/surveys/contestants.html;

[12] *IOI, International Olympiad in Informatics*, http://www.IOInformatics.org/.

[13] *IOI Regulations*; http://olympiads.win.tue.nl/ioi/rules/index.html, 2002.

[14] *IOI 2002 Competition*, Yong-In, Korea, 2002, 14-19.

[15] Kerren A., Stasko. Algorithm Animation J.T. In *Software Visualization*, LNCS 2269, pp. 1–15, 2002.

[16] Khuri S., Holapfel K. EVEGA: An Educational Visualization Environment for Graph Algorithms. In *Proceedings of the 6th Annual Conference on Innovaton and Technology in Computer Science Education, ITiCSE 2001*. ACM Press, 2001.

[17] van Leeuwen W. T. *A Critical Analysis of the IOI Grading Process with an Application of Algorithm Taxonomies.* Master's Thesis, Technische Universiteit Eindhoven, Faculty of Mathematics and Computing Science, http://www.win.tue.nl/˜wstomv/misc/ioi-analysis/thesis-final.pdf, October 2005.

[18] Manzoor S., Analyzing Programming Contest Statistics, In *International Workshop, Perspectives on Computer Science Competitions for (High School) Students*, http://www.bwinf.de/competition-workshop/; January 25-28, 2006.

[19] Sumner N., Banu D., Dershem H. JSAVE: Simple and Automated Algorithm Visualization Using the Java Collection Framework. In *Proceedings of the tenth annual Consortium for Computing Sciences in Colleges*, October 2003.

[20] Verhoeff. T. The 43rd International Mathematical Olympiad: A Reflective Report on IMO 2002. In *Computing Science Report 02-11*, Fac. of Math . and Comp. Sc., Eindhoven Univ. of Technology, Netherlands, August 2002.

[21] Verhoeff T. *Concepts, Terminology, and Notations for IOI Competition Tasks.* Document presented at IOI 2004 in Athens, 12 Sep. 2004. http://olympiads.win.tue.nl/ioi/sc/documents/terminology.pdf

[22] Verhoeff T. The IOI Is (not) a Science Olympiad, *Competitions for (High School) Students*, http://www.bwinf.de/competition-workshop/; January 25-28, 2006.