INTERNATIONAL JOURNAL OF APPLIED MATHEMATICS AND INFORMATICS
DOI: 10.46300/91014.2021.15.13 Volume 15, 2021

Methods and Tools for Visualization of Graphs and Graph Algorithms*

V.N. Kasyanov
Laboratory for Program Construction and Optimization
Institute of Informatics Systems
Novosibirsk, Russia
kvn@iis.nsk.su

Received: March 26, 2019. Revised: May 20, 2021. Accepted: October 23, 2021. Published: November 16, 2021.

Abstract—**Graphs are the most common abstract structure encountered in computer science and are widely used for structural information visualization. In the paper, we consider practical and general graph formalism of so called hierarchical graphs and present the Higres and ALVIS systems aimed at supporting of structural information visualization on the base of hierarchical graph models.**

Keywords—hierarchical graph; graph algorithm; graph drawing; graph algorithm animation; information visualization; visualization system

I. Introduction

Graphs are the most common abstract structure encountered in computer science. Any system that consists of discrete states (or sites) and connections between them can be modeled by a graph. Graphs and graph algorithms are used almost everywhere in computer science (see, for example, [13, 14]).

Graph models can be used in practice only along with support tools that provide visualization, editing and processing of graphs. For this reason many graph visualization systems, graph editors and libraries of graph algorithms have been developed in recent years. Examples of these tools include VCG [19], daVinci [6], Graphlet [11], GLT&GET [18], yEd [21] and aiSee [1].

In some application areas the organization of information is too complex to be modeled by a classical graph. To represent a hierarchical kind of diagramming objects, some more powerful graph formalisms have been introduced, e.g. higraphs [8] and compound digraphs [20]. The higraphs are an extension of hypergraphs and can represent complex relations, using multilevel "blobs" that can enclose or intersect each other. The compound digraphs are an extension of directed graphs and allow both inclusion relations and adjacency relations between vertices, but they are less general then the higraph formalism. One of the recent non-classical graph formalisms is the clustered graphs [5]. A clustered graph consists of an undirected graph and its recursive partitioning into subgraphs. It is a relatively general graph formalism that can handle many applications with hierarchical information, and is amenable to graph drawing.

Hence, there is a need for tools capable of visualization of such structures. Although some general-purpose graph visualization systems provide recursive folding of subgraphs, this feature is used only to hide a part of information and cannot help us to visualize hierarchically structural information. Another weak point is that usual graph editors do not have a support for attributed graphs. Though the GML file format, used by Graphlet, can store an arbitrary number of labels associated with graph elements, it is impossible to edit and visualize these labels in the Graphlet graph editor. The standard situation for graph editors is to have one text label for each vertex and, optionally, for each edge.

Graph drawing is a useful way of representation of graph models, and visualization of graphs is used in many applications for the design and analysis of communication networks, related documents, as well as static and dynamic structures of programs [9]. However, systems of related objects frequently are dynamic. For example, relations between objects or properties of objects can be changed. If transformation processes can be formalized and presented in the algorithmic form then it is useful to create a graphical representation of transformations.

An algorithm animation visualizes the behavior of an algorithm by producing an abstraction of both the data and the operations of the algorithm [17]. Initially it maps the current state of the algorithm into an image, which then is animated based on the operations between two succeeding states in the algorithm execution. Animating an algorithm allows for better understanding of the inner workings of the algorithm, furthermore it makes apparent its deficiencies and advantages thus allowing for further optimization.

Methods and systems of graph algorithm animations allow us to study graph algorithms and in particular the processes in connected systems. Research in visualization of algorithms is mostly focused on the construction of examples of dynamic visualization. Visualization of a graph is a graphic representation of graph elements. Usually graph elements

* The work was partially supported by the Russian Foundation for Basic Research (grant 12-07-0091) and the Dynasty Foundation (grant NG13-076).

match some shapes, which makes it possible to build a graph image. For example, vertices are displayed in a form of circles and, edges in a form of arc lines, broken lines or smooth curves. Applications of graph algorithm visualization can be divided into two types according to the method they implement: interesting events and the data-driven method [3]. Methods of the first type are based on selection of events that occur during execution of an algorithm, for example, comparing the vertex attribute value or removing an edge. Methods of this type create some visual effects for each interesting event. Methods of the second type are based on data changing. During an operation, the memory status is changed, for example, the values of variables. Further these changes are visualized in some understandable way. In the simplest case such changes can be displayed in a form of a table of variable values. This approach is used in debuggers of integrated development environments.

The existing algorithm visualizers have several disadvantages. One of the major drawbacks is that if there is a need to build visualization of an algorithm arbitrarily close to the original algorithm, then it is necessary to build a new visualizer. As a rule, visualizers also do not show the correspondence between the algorithm instructions and the generated visual effects or do not allow reassignment of visual effects to the corresponding events.

In the paper, we consider a practical and general graph formalism called hierarchical graphs and graph models [12]. It is suited for visual processing and can be used in many areas where the strong structuring of information is needed [15, 16]. We present also the Higres and ALVIS systems that are aimed at supporting of information visualization on the base hierarchical graph modes. The Higres system is a visualization tool and an editor for attributed hierarchical graphs and a platform for execution and animation of graph algorithms [10]. The ALVIS system is intended for building of graph algorithm visualizations with the help of a flexible system of visual effects and using a visualized graph algorithm as an input parameter.

II. Hierarchical Graphs and Graph Models

A. Hierarchical graphs

Let G be a graph of some type, e.g. G can be an undirected graph, a digraph or a hypergraph. A graph C is called a *fragment* of G, denoted by $C \subseteq G$, if C includes only elements (vertices and edges) of G. A set of fragments F is called *a hierarchy of nested fragments* of the graph G, if $G \in F$ and $C_1 \subseteq C_2$, $C_2 \subseteq C_1$ or $C_1 \cap C_2 = \emptyset$ for any $C_1, C_2 \in F$.

A *hierarchical graph* $H = (G,T)$ consists of a graph G and a rooted tree T that represents an immediate inclusion relation between fragments of a hierarchy F of nested fragments of G. G is called the *underlying graph* of H. T is called the *inclusion tree* of H.

A hierarchical graph H is called a *connected* one, if each fragment of H is connected graph, and a *simple* one, if all fragments of H are induced subgraphs of G.

It should be noted that any clustered graph H can be considered as a simple hierarchical graph $H=(G, T)$, such that G is an undirected graph and the leaves of T are exactly the trivial subgraphs of G (See Fig. 1).

A *drawing* D of a hierarchical graph $H = (G,T)$ is a representation of H in the plane such that the following properties hold (See Fig. 1).

- Each vertex of G is represented either by a point or by a simple closed region. The region is defined by its *boundary* - a simple closed curve in the plane.
- Each fragment of G is drawn as a simple closed region which includes all vertices and subfragments of the fragment.
- Each edge of G is represented by a simple curve between the drawings of its endpoints.

D is a *structural* drawing of H if all edges of any fragment of H are located inside the region of the fragment.

Fig.1 gives an example of a nonstructural drawing of a hierarchical graph. A hierarchical graph is called a *planar* one if it has such a structural drawing that there are no crossing between distinct edges and the boundaries of distinct fragments.

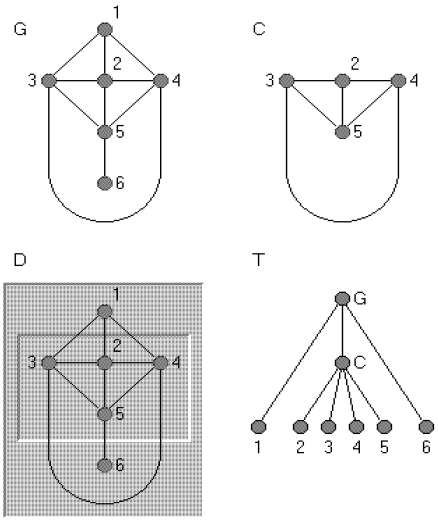

Fig. 1. A simple hierarchical graph $H = (G,T)$ and its drawing D

B. Hierarchical graph models

Let V be a set of objects called simple *labels* (e.g. V can include some numbers, strings, terms and graphs). Let W be a set of *label types* of graph elements and let a label set $V(w) = V_1 \times V_2 \times \ldots \times V_s$, where $s \geq 1$ and for any i, $1 \leq i \leq s$, $V_i \subseteq V$, be associated with each $w \in W$.

A labelled hierarchical graph is a triple (H,M,L), where H is a hierarchical graph, M is *a type function* which assigns to each element (vertex, edge and fragment) h of H its type $M(h) \in W$, and L is a *label function,* which assigns to each element h of H its label $L(h) \in V(M(h))$.

The semantics of a hierarchical graph model is provided by an equivalence relation which can be specified in different ways, e.g. it can be defined via *invariants* (i.e. properties being inherent in equivalent labelled graphs) or by means of so-called *equivalent* transformations that preserve the invariants.

III. System Higres

A. Graph models in Higres

A hierarchical graph supported by the Higres consists of vertices, fragments and edges which we call objects (See Fig. 2). Vertices and edges form an underlying graph. This graph can be directed or undirected. Multiple edges and loops are also allowed.

The semantics of a hierarchical graph is represented in Higres by means of object types and external modules (see below). Each object in the graph belongs to an object type with a defined set of labels. Each label has its data type, name and several other parameters. A set of values is associated with each object according to the set of labels defined for the object type to which this object belongs. These values, along with partitioning of objects to types, represent the semantics of the graph. New object types and labels can be created by the user.

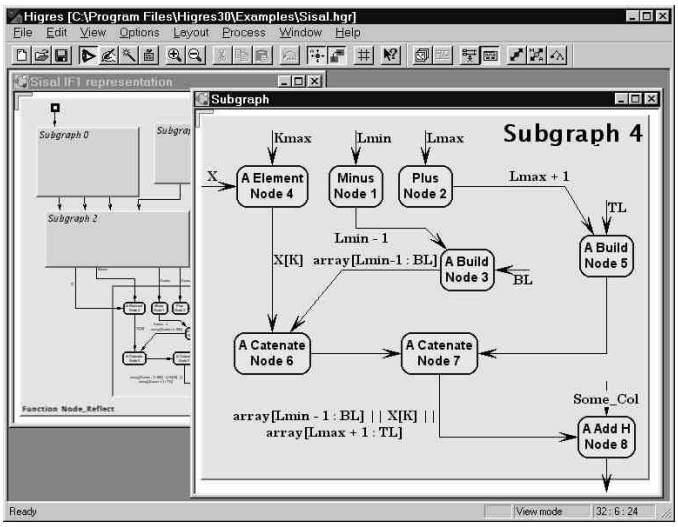

Fig. 2. A hierarchical graph in the Higres system

B. Visualization

In the Higres system each fragment is represented by a rectangle. All vertices of this fragment and all subfragments are located inside this rectangle. Fragments, as well as vertices, never overlap each other. Each fragment can be closed or open (See Fig. 2). When a fragment is open, its content is visible; when it is closed, it is drawn as an empty rectangle with only label text inside it. A separate window can be opened to observe each fragment. Only content of this fragment is shown in this window, though it is possible to see this content inside windows of parent fragments if the fragment is open.

Most part of visual attributes of an object is defined by its type. This means that semantically relative objects have similar visual representation. The Higres system uses a flexible technique to visualize object labels. The user specifies a text template for each object type. This template is used to create the label text of objects of the given type by inserting labels' values of an object.

Other visualization features include the following:

- various shapes and styles for vertices;
- polyline and smooth curved edges;
- various styles for edge lines and arrows;
- the possibility to scale graph image to an arbitrary size;
- edge text movable along the edge line;
- colour selection for all graph components;
- external vertex text movable around the vertex;
- font selection for labels text;
- two graphical output formats;
- a number of options to control the graph visualization.

Now Higres uses three graph drawing algorithms for automatic graph allocation. The first one is a force method, which is very close to original algorithm from [4]. The second one is our improvement of the first. The third one allocates rooted trees on layers.

C. The user interface

The comfortable and intuitive user interface was one of our main objectives in developing Higres. The system's main window contains a toolbar that provides a quick access to frequently used menu commands and object type selection for creation of new objects. The status bar displays menu and toolbar hints and other useful information on current edit operation.

The system uses two basic modes: view and edit. In the view mode it is possible only to open/close fragments and fragment windows, but the scrolling operations are extended with mouse scrolling. In the edit mode the left mouse button is used to select objects and the right mouse button displays the popup menu, in which the user can choose the operation he/she wants to perform. It is also possible to create new objects by selecting commands in this menu. The left mouse button can be also used to move vertices, fragments, labels texts and edge bends, and resize vertices and fragments. All edit operations are gathered in a single edit mode. To our opinion, it is more useful approach (especially for inexperienced users) than division into several modes. However, for adherents of the last case we provide two additional modes. Their usage is optional but in some cases they may be useful: the "creation" mode for object creation and "labels" mode for labels editing.

Other interface features include the following:

- almost unlimited number of undo levels;

- optimized screen update;
- automatic elimination of objects overlapping;
- automatic vertex size adjusting;
- grid with several parameters;
- a number of options that configure the user interface;
- online help available for each menu, dialog box and editor mode.

D. Algorithm animation

To run an algorithm in the Higres system, the user should select an external module in the dialog box. The system starts this module and opens the process window that is used to control the algorithm execution. Higres provides the run-time animation of algorithms. It also caches samples for the repeated and backward animation. A set of parameters is defined inside a module. These parameters can be changed by the user at any execution step. The module can ask user to input strings and numbers. It can also send any textual information to the protocol that is shown in the process window.

A wide range of semantic and graph drawing algorithms can be implemented as external modules. As examples now we have modules that simulate finite automata, Petry nets and imperative program schemes (See Fig. 3). The animation feature can be used for algorithm debugging, educational purposes and exploration of iteration processes such as force methods in graph drawing.

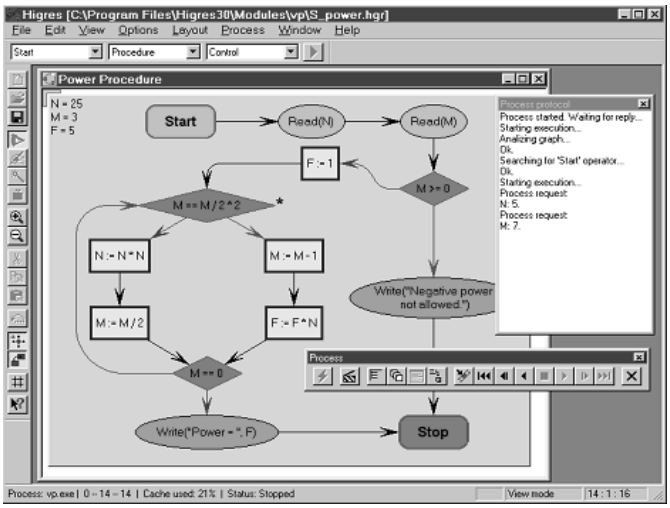

Fig. 3. Animation of simulation of imperative program scheme

A special C++ API that can be used to create external modules is provided. This API includes functions for graph modification and functions that provide interaction with the Higres system. It is unnecessary for programmer, who uses this API, to know the details of the internal representation of graphs and system/module communication interface. Hence, the creation of new modules in the Higres system is a rather simple work.

IV. System ALVIS

A. Interactive visualization model

A new algorithm visualization model based on the dynamic approach and hierarchical graph models has been created. The main point of the suggested model is that the given algorithm is formulated in some programming language that allows us to use instructions operating with graphs and to execute the program derived from the text of the algorithm after a set of transformations. More details about the model can be found in [7]. The result of the program execution is information which is to be used in creation of the underlying algorithm visualization. An example of such instruction can be adding an edge or a change in the attributes of vertices. The following example shows the breadth-first search algorithm for any graph. In the given case, Get and Set instructions are used for reading and changing the graph element's attribute values. These instructions have formats Get(Vertex, AttributeName) and Set(Vertex, AttributeName, AttributeValue), respectively. To construct a visualization of the breadth-first search algorithm, the state attribute is appointed to each graph vertex. The value of the state attribute reflects whether the vertex was visited during graph traversal.

```
VertexQueue.Enqueue(Graph.Vertices[0]);
while (VertexQueue.Count > 0)
{
Vertex v = VertexQueue.Dequeue();
Set(v, "state", "visited");
foreach(Edge e in v.InEdges)
{
Vertex t =e.PortFrom.Owner;
string c = Get(t, "state");
if(c != "visited")
{
Set(t, "state", "visited");
VertexQueue.Enqueue(t);
}
}
foreach(Edge e in v.OutEdges)
{
Vertex t = e.PortTo.Owner;
string c = Get(t, "state");
if(c != "visited")
{
Set(t, "state", "visited");
VertexQueue.Enqueue(t);
}
}
}
VertexQueue.Clear();
```

Each instruction of the algorithm generates one or more images of the current state of the graph model. The graph model is an annotated hierarchical marked graph. It is useful to highlight the current executing instruction in each image because it allows a user to keep attention on valuable events

at this moment. To solve the problem of highlighting the current executing instruction in the image, the following approach is used. Each text line of an algorithm can be interpreted as a function. Also, each text line has a numeric index in all text lines. So that order value is added to arguments of the function corresponding to the text line. This additional parameter is the number of the current executing algorithm instruction. After this transformation, the text of the breadth- first search algorithm from the above example looks like this:

```
VertexQueue.Enqueue(Graph.Vertices[0]);
while (WhileCondition(2,
VertexQueue.Count > 0))
{
Vertex v = VertexQueue.Dequeue(3);
Set(4, v.ID, “state”, “visited”);
foreach(Edge e in ForeachCollection(5, v.InEdges))
{
Vertex t = e.PortFrom.Owner;
string c = Get(7, t, ”state”);
if(If Condition(8, c != ”visited”))
{
Set(10, t, ”state”, ”visited”);
VertexQueue.Enqueue(11, t);
}
}
foreach(Edge e in ForeachCollection(13,
v.OutEdges))
{
Vertex t = e.PortTo.Owner;
string c = Get(16, t, ”state”);
if(If Condition(17, c != ”visited”))
{
Set(19, t, ”state”, ”visited”);
VertexQueue.Enqueue(20, t);
}
}
}
VertexQueue.Clear( );
```

The above example shows changes in the attributes of the graph elements, too. This is a typical situation for algorithms implementing only traversal of a graph - a method when all graph vertices are visited one by one. For example, the Prüfer sequence of a given tree is generated by iteratively removing vertices from the tree u ntil only two vertices remain. To perform this operation, the RemoveVertex( ) in struction should be used, which leads to generation of a visual effect of the corresponding vertex disappearing. Here is an example of the Prüfer encoding algorithm, how it can be formulated as a parameter of the graph algorithm visualization system:

```
Int i = 0;
List<Vertex> Leafs = new List<Vertex>( );
int n = Graph.Vertices.Count;
while(i++ <= n-2)
{
Leafs.Clear();
foreach(Vertex v in Graph.Vertices)
    if(v.OutEdges.Count == 0) Leafs.Add(v);
Vertex codeItem =
Leafs[0].InEdges[0].PortFrom.Owner;
Output.Add(codeItem);
RemoveVertex(Leafs[0]);
}
```

Each algorithm instruction generates some information during execution of the transformed text of the original algorithm. This information describes the number of the current instruction, the name of an attribute of a graph element, the previous value of the attribute, a new value of the attribute and the identifier of the graph element. This information allows us to get the full log of operations executed over graph elements. This operation log contains the detailed information on the state of the graph model during the algorithm running. Further the log of operations, the input graph and the original text of the algorithm can be used to generate the algorithm visualization. Each operation log entry corresponds to some graphical effect over visual representation of graph elements. The simplest example of the visual effect for the breadth-first search algorithm is to change the color of the graph vertex representation when a state attribute of the vertex has been changed and to change the color of the text of the corresponding instruction.

B. Algorithm visualization system

The ALVIS system for graph algorithm visualization on the base of the described model has been constructed. It implements visualization in two steps: first, the algorithm text is transformed into a p rogram ready for execution; after that the program is executed with the given graph as a parameter. The result is a log of items, each of which contains information about changes in the graph model state. Second, the visualizer receives the input graph, the original algorithm text, the log of execution and visual effects settings. As a result, the visualization system works out a sequence of images corresponding to the graph model of intermediate states of the algorithm.

The ALVIS system includes two main components: an algorithm execution module and a graph algorithm visualizer. It is assumed that data are passed between these and other components of the syste m in a text form. It means tha t components of the system can be implemented on different platforms and with different tools.

The purpose of the algorithm execution module is to generate the execution log. The algorithm running is separated from its visualization. This allows us to perform the algorithm once and after that the operation log can be used to visualize and refine the visualization many times. This can be useful when computationally-intensive algorithms are visualized. In such cases the second cycle of execut ion of the algorithm is complex.

To provide correct work of the algorithm execution module, it is necessary to meet a significant condition. Since any existing compiler or interpreter can be used to create this module, the algorithm must be formulated in the language supported by the selected compiler or interpreter. Actually this is not a restrictio n on the algorithm implementation language since many programming languages allow graph structures to be used in the p rogram source code. So, the given algorithm text can be considered as a ready program source code. Also this allows us to transmit the input graph in this compiled program and to generate the log of operations.

Another significant restriction relates to the algorithmic complexity. In this approach, it is reasonable to visualize only efficient algorithms, because it will take much time to build the operation log of execution of an efficient algorithm. We can use a small input graph for this case. This assumption allows us to construct visualization for a reasonable time.

The algorithm execution module takes the given algorithm text in an appropriate programming language, executes it and returns the log of operations generated during the algorithm run on a particular graph. The log of executed operations contains information about all changed attributes of graph elements and about graph elements added or removed during the execution. Further this information is used to generate the algorithm visualization.

The second main component of the visualization system is the visualizer itself. At its input, this component receives the algorithm text, the graph, the log of operations and additional graphical options. A log information item is added by special instructions created at the stage of preparation of the algorithm text. For example, these special instructions are the functions: Set( ), Get( ), IfCondition( ), WhileCondition( ) and ForeachCollection( ). Their first argument is the number of the corresponding text line. IfCondition( ) and WhileCondition( ) do not perform any changes in the graph model state but at least allow us to make a visual selection of the text line where it was inserted. ForeachCollection( ) is to be used to generate information which allows highlighting a set of vertices before they will be actually enumerated. To add these functions into appropriate places of the original text of the algorithm, it is sufficient to use a contextual replacement. The purpose of the preparation stage is to eliminate the need for declarative structures, which have no relation to the actual nature of the algorithm.

A log item may also contain information about the value of an attribute of a graph element. A graph element is a vertex, an edge or a port. If there is a vertex with its incident edge, then a port is a point where the edge enters the vertex. When rendering, it can be useful that the points are allocated for these additional objects. Ports simplify calculation of coordinates of grap hical primitives which represent the edge elements. Strictly mathematically, it is possible to simulate a port with a labeled vertex. So the class of graphs with ports is isomorphic to the class of all graphs.

An attribute of a vertex, an edge or a port can have a string name and a string value. The log of operations stores the previous value of the attribute for a particular graph element. This information is also useful for building the visualization, since it is possible to make a smooth visual effect from a previous value of an attribute to its new value.

It is not obvious how to bind information from a log item to the visual effect. In this case, a user needs to interfere in order to set an explicit binding between the set of attributes in the text of the algorithm and the desired visual effects. For example, if the operation of a log item is about changing the coordinates of the graph element reflected with the use of the attribute "position", then it is reasonable to bind the attribute with the visual effect, which leads to a shift of the graph element. Another user example is to bind all log items to the effect of a color mark of a current graph element under processing. It can be a current vertex visited in the algorithm of dept h-first search or in any other graph traversal. In this aspect the suggested approach is close to the interesting events approach, where an algorithm instruction is an interesting event.

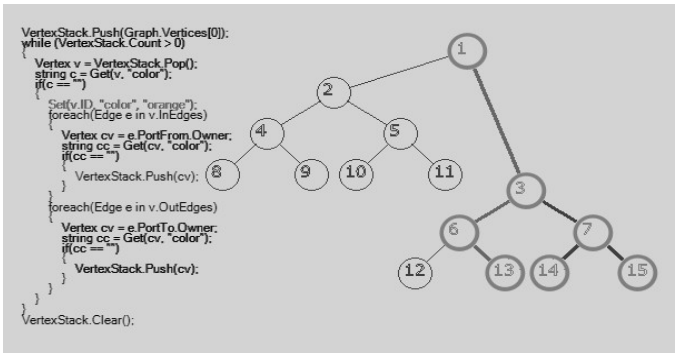

Fig. 4. Visualization of the depth-first search algorithm

Fig. 4 shows an example of visualization of the depth-first search algorithm on the graph, which is actually a binary tree graph. The figure is one of the screenshots taken during the process of visualization of the depth-first search algorithm. The left side of the figure displays the text of the algorithm formulated in terms of graphs. The attribute of a graph vertex state indicates the fact that the vertex has already been visited during the process of the graph traversal. A line of the algorithm text has one of the following states: dark thin, light thin and thick. The first state means that the instruction has been executed at least once. The second s tate means that the current image and the last shown visual effect is the result of this instruction. The last state means that the instruction has not been executed yet. The right part of the figure displays the graph model, which is a hierarchical graph with attributes. Only if this attribute is set, the corresponding attribute will be created during visualization. In this example, the visited vertices get the state attribute that changes the color of a vertex. Also, this attribute's value corresponds to the increase of line width showing the graph vertex circle. Vertices shown in a thin line have not been visited yet.

There are methods that improve understanding of a graph algorithm visualization based on visual effects. If there is a rendering context of a visual effect for a log item, then this context can be used to improve understanding of the algorithm. For example, a smooth visualization along the

edge connecting the previous and the current vertices can be used for visualization of the depth-first search algorithm. In this case, the context of visualization for the current vertex is the previous vertex. If the previous vertex is not incident to the current one, then the following method can be used. It is necessary to find the shortest path from the current vertex to the previous one and, after that, to apply a smooth visualization along the edges of this path. This method helps us to improve understanding of visualization of the algorithm because a user can track the path of the graph vertices traversal. Fig. 4 shows how visualization is used with a rendering context. In this example, vertex 13 has been visited after vertex 14 and vertex 14 is the rendering context for vertex 13. This means that, when vertex 13 is visited, all edges from the shortest path between these two vertices will be rendered ac- cording to the visual effect specified in the settings. So, the thickened light line is used for drawing edges which belong to the path from the current vertex to the root of the tree graph. A thick dark line is used for drawing edges which are incident to already visited vertices.

Displaying of additional data structures can also be used to improve understanding of visualization of a graph algorithm. For example, the depth-first search algorithm uses a stack and the breadth-first search algorithm visualization uses a queue. The content of a stack or a queue can be rep- resented as a graph. Since the visualization system allows us to use the hierarchical graphs, a stack graph or a queue graph can be included into a graph model for a particular visualization. So the working graph model consists of a graph with two vertices. The first vertex contains a stack graph and the second contains an input graph. Such graph model can be visualized with the created module of the system of graph algorithm visualization. The queue or stack size is changed during execution of the given algorithm and the corresponding vertices are added or removed from the stack graph. Hierarchical graphs are helpful for this purpose. If there is no stack or queue, then a tree of fragments only consists of one fragment, the input graph. For a stack the graph model consists of three fragments: a root and two children. The first child is the input graph and the second is a graph representation of the stack. So, if the given algorithm uses an input graph and N additional structures, then the tree of fragments contains N+2 elements. It is a root element and its N+1 children, one of which is the input graph and others are graph representations of additional data structures.

V. Conclusion

In the paper, a practical and general graph formalism of hierarchical graphs and graph models was considered. It is suited for visual processing and can be used in many areas where the visualization of structural information is needed.

The Higres system being a visualization tool and an editor for attributed hierarchical graphs and a platform for execution and animation of graph algorithms was presented. The ALVIS system which builds the algorithm visualization with the help of a flexible system of visual effects and using a visualized graph algorithm as an input parameter was described.

Acknowledgment

The author is thankful to all colleagues taking part in the projects described. The work was partially supported by the Russian Foundation for Basic Research (grant N 12-07-0091) and the Dynasty Foundation (grant NG13-076).

References

[1] aiSee http://www.absint.com/aisee/

[2] C. Demetrescu, I. Finocchi. "A general-purpose logic-based visualization framework", Proc. of the 7th Internat. Conf. in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99), Plzen, 1999, pp. 55–62.

[3] C. Demetrescu, I. Finocchi, J. T. Stasko. "Specifying Algorithm Visualizations: Interesting Events or State Mapping?", Lecture Notes in Computer Science, Vol. 2269, 2002, pp.16–30.

[4] P. Eades, "A heuristic for graph drawing", Congressus Numerantium, Vol. 42, 1984, 149-160.

[5] Q.W. Feng, R.F. Cohen, P. Eades, "Planarity for clustered graphs", Lecture Notes in Computer Science, Vol. 979, 1995, pp. 213-226.

[6] M. Fröhlich, M. Werner, "Demonstration of the interactive graph visualization system daVinci", Lecture Notes in Computer Science, Vol. 959, 1995, pp. 266-269.

[7] D.S. Gordeev."Graph algorithm visualization: interpretation of algorithm as a program", Informatics in Research and Education, Novosibirsk, IIS, 2012, pp.149-160. (in Russian).

[8] D. Harel, "On visual formalism, Comm". ACM, Vol. 31, No. 5, 1988, pp. 514-530.

[9] I. Herman, G. Melançon, M.S. Marshall, "Graph visualization and navigation in information visualization: a survey", IEEE Trans. on Visualization and Computer Graphics, Vol. 6, 2000, pp. 24-43.

[10] Higres http://pco.iis.nsk.su/higres

[11] M. Himsolt, "The Graphlet system (system demonstration)", Lecture Notes in Computer Science, Vol. 1190, 1997, pp. 233-240.

[12] V.N. Kasyanov. "Hierarchical graphs and graph models: problems of visual processing", Problems of informatics systems and programming, Novosibirsk, IIS, 1999, pp. 7-32. (in Russian).

[13] V.N. Kasyanov, V.A. Evstigneev, Graph Theory for Programmers. Algorithms for Processing Trees, Kluwer Academic Publishers, 2000. 432 p.

[14] V.N. Kasyanov, V.A. Evstigneev. Graphs in Programming: Processing, Visualization and Application, St. Petersburg, BHV-Petersburg, 2003, 1104 p. (In Russian).

[15] V.N. Kasyanov, E.V. Kasyanova, Visualization of Graphs and Graph Models, Siberian Scientific Publ., 2010, 123 p.(in Russian).

[16] V.N. Kasyanov, I.A. Lisitsyn. "Hierarchical graph models and visual processing", Proc.of Conference on Software: Theory and Practice. 16th IFIP World Computer Congress 2000, Beijing, PHEI, 2000, pp. 179-182.

[17] A. Kerren and J. Stasko. "Algorithm animation - introduction", Lecture Notes in Computer Science, Vol. 2269, 2002, pp. 1-15.

[18] B. Madden, P. Madden, S. Powers, M. Himsolt, "Portable graph layout and editing", Lecture Notes in Computer Science, Vol. 1027, 1996, pp. 385-395.

[19] G. Sander, "Graph layout through the VCG tool", Lecture Notes in Computer Science, Vol. 959, 1995, pp.194-205.

[20] K. Sugiyama, K. Misue, "Visualization of structured digraphs", IEEE Trans. on Systems, Man and Cybernetics, Vol. 21, No. 4, 1999, pp. 876-892.

[21] yEd http://www.yworks.com/en/

Creative Commons Attribution License 4.0 (Attribution 4.0 International, CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0
https://creativecommons.org/licenses/by/4.0/deed.en_US